UNIT 4

- **Errors:**

Errors are the problems that occur in the program due to an illegal operation performed by the user or by the fault of a programmer, which halts the normal flow of the program. Errors are also termed bugs or faults. There are mainly two types of errors in programming.

1. Compile Time Errors

Compile Time Errors are those errors that are caught during compilation time. Some of the most common compile-time errors are syntax errors, library references, incorrect import of library functions and methods, uneven bracket pair(s), etc.

Example:

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

int main()
{
   cout << "Hello Scaler Topics!"

   return 0;
}
```

Output:

```
test.cpp: In function 'int main()':
test.cpp:6:35: error: expected ';' before 'return'
    cout << "Hello Scaler Topics!"
                                 ^
                                 ;
test.cpp:8:5:
    return 0;
```

In this program, the compiler detected the compilation error, i.e. **missing semi-colon.**

2. Run-Time Errors

Run-Time Errors are those errors that cannot be caught during compilation time. As we cannot check these errors during compile time, we name them Exceptions. Exceptions can cause some serious issues so we should handle them effectively.

Example:

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

int main()
{

   int a = 5;

   // Dividing the number a by zero, so the program will compile easily
   // but run time error will be generated.
   cout << a / 0;
   return 0;
}
```

Output:

```
test.cpp: In function 'int main()':
test.cpp:11:14: warning: division by zero [-Wdiv-by-zero]
   cout << a / 0;
         ~~^~~
```

- **Exception Handling in C++**

Exception handling in C++ was introduced to deal with abnormal run-time anomalies and abnormal conditions caused during run time.

As we know **exception(s)** or **error(s)** hinder the normal execution flow of a program so exception handing in C++ is one of the most important topics. We can formally define exception handling as - Exception Handling is the process of handling errors and exceptions such that the normal execution of the system is not halted.

One of the most common run time exceptions can be 0 division error. When we try to divide a number by 0, then the program will get executed successfully but during compile time, we will face an error causing an application crash.

For exception handling in C++, we use the try-catch-finally block.

Syntax:

```
try {
  // Block of code to try
  throw exception;
}
catch () {
  // Block of code to handle errors
}
```

❖ **Why Exception Handling?**

Exception handling in C++ checks the exception so that the normal execution of the system is not halted.

The main aim of Exception handling in C++ is to separate the error handling code from the normal code. We can try to handle exceptions without exception handing in C++. We can always use multiple if-else conditions to handle errors. As normal code also contains conditional statements like if-else so, these conditions can get mixed up with our error handling if-else conditions making the entire code less readable and less maintainable. So, we use try-catch blocks to easily manage exceptional handing in C++.
Using exception handling in C++, we can create a hierarchy of exception objects, group exceptions in namespaces or classes, and categorize exceptions according to their types.

Using exception handling in C++, we can throw any number of exceptions from a function but we can choose to handle some of the thrown exceptions. A program can throw both pre-defined as well as a custom exception(s) as per the need of the program.

Errors and Exceptions can be of two types in C++. The first one being Compile Time Errors. Compile Time Errors are those errors that are caught during compilation time. The other one is Run Time Error. Run-Time Errors are those errors that cannot be caught during compilation time. As we cannot check these errors during compile time, we name them Exceptions.

If all the thrown exceptions are not handled by the function itself, the function caller can handle the rest of the exceptions. If the caller does do not choose to catch them, then the exceptions are handled by the caller of the caller.

Example:

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

/*
Here, we want to throw exception (age as exception) if the age of the person is less than 18.
*/
int main()
{
   // checking if the age is more than 18 in try block.
   try
   {
      int age = 15;
      if (age >= 18)
      {
         cout << "Access granted.";
      }
      // Throwing custom exception if the age is less than 18.
       else
      {
         throw(age);
      }
   }
   // catching the thrown exception and displaying the desired output (access denied!)
   catch (int x)
   {
      cout << "Access denied!, Age is: " << x << endl;
   }
   return 0;
}
```

Output:

```
Access denied!, Age is: 15
```

- **Basic Keywords in Exception Handling**

As we know errors and exceptions can hinder the normal flow of program execution, so we use exception handling in C++. The exception handing in C++ is mainly performed using three keywords namely - try, catch, and throw.

Syntax:

```
try
{
   // code
   throw exception;
}
catch(exception e)
{
   // code for handling exception
}
```

1. C++ try:

The try block is used to keep the code that is expected to throw some exception. Whenever our code leads to any exception or error, the exception or error gets caught in the catch block. In simple terms, we can say that the try block is used to define the block of code that needs to be tested for errors while it is being executed.

Example: Suppose we are dealing with databases, we should put the code that is handling the database connection inside a try block as the database connection may raise some exceptions or errors.

2. C++ catch:

The catch block is used to catch and handle the error(s) thrown from the try block. If there are multiple exceptions thrown from the try block, then we can use multiple catch blocks after the try blocks for each exception. In this way, we can perform different actions for the various occurring exceptions. In simple terms, we can say that the catch block is used to define a block of code to be executed if an error occurs in the try block.

Example: Let us take the same above example that we are dealing with the database. Now, if during the connection, there is an exception raised inside the try block, then there should be a catch block present to catch or accept the exception and handle the exception. The catch block ensures that the normal flow of the code is not halted.

Note:

- The try and catch block comes is used in pairs.
- We can have multiple catch blocks for one try statement.

3. C++ throw:

The throw block is used to throw exceptions to the exception handler which further communicates the error. The type of exception thrown should be same in the catch block. The throw keyword accepts one parameter which is passed to the exception handler. We can throw both pre-defined as well as custom exception(s) as per the requirements.

Whenever we want to explicitly throw an exception, we use the throw keyword. The throw keyword is also used to generate the custom exception.

Let us take an example to understand the overall working and syntax of try, catch, and throw in exception handling in C++.

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

int main()
{
  int x = 99;
  cout << "Before the try block." << endl;
  try
  {
    cout << "Inside the try block." << endl;
    // Throwing an exception if the
    // value of x is smaller than 100.
    if (x < 100)
    {
      // Throwing the value of x as exception as x is now less than 100.
      throw x;
      cout << "After throw the throw block." << endl;
    }
}
  // Catching the value of x thrown by the throw keyword from the try block.
  catch (int x)
  {
    cout << "Exception caught in the catch block." << endl;
  }
  return 0;
}
```

Output:

```
Before the try block.
Inside the try block.
Exception caught in the catch block.
```

In the above example, the cout (print) statement after the throw line will never get executed.

If we do not know the type of throw used in the try block, we can always use the **"three dots" syntax (...)** inside the catch block, which will handle any type of exception. Let's see an example to understand the syntax and use case:

```
#include <iostream>
using namespace std;
/*
Here, we want to throw exception (a random number as exception) if the age of the person is less than 18.
*/
int main()
{
  try
  {
    int age = 25;
    if (age <= 18)
    {
      cout << "Access denied. Not for kids.";
    }
    else
    {
      // throwing any random value as exception as age is less than 18.
      throw 505;
    }
}
  // Catching the thrown exception and displaying access denied!
  catch (...)
  {
    cout << "Access denied! You need to be at least 18 years old." << endl;
  }
  return 0;
}
```

Output:

Access denied! You need to be at least 18 years old.

- **Templates in C++**

The templates are one of the most powerful and widely used methods added to C++, allowing us to write generic programs. Templates in C++ allow us to define generic functions and classes. Templates in C++ promote generic programming, meaning the programmer does not need to write the same function or method for different parameters.

- **Types of Templates in C++**

We can use templates in C++ to define generic functions and classes. We can represent templates in C++ in two different ways, namely - **function templates and class templates**.

1. **Function Templates**

Function templates are similar to normal functions. Normal functions work with only one data type, but a function template code can work on multiple data types. Hence, we can define function templates in C++ as a single generic function that can work with multiple data types.

Functional templates are more powerful than overloading a normal function as we need to write only one program, which can work on all data types.

Syntax of the template function:

```
template <class T> T function-name(T args)
{
  // body of function
}
```

In the syntax above:

- **T** is the type of argument or placeholder that can accept various data types.

- **class** is a keyword used to specify a generic type in a template declaration. As we have seen earlier, we can always write **typename** in the place of **class.**

Some of the pre-defined examples of function templates in C++ are sort(), max(), min(), etc. Let's take an example to understand the working and syntax of function templates in C++.

Example:

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

// Template function that will be adding two data.
template <typename T>
T add(T a, T b)
{
  return (a + b);
}

// Main function
int main()
{
  // Variables to store results of different data types.
  int ans1;
  double ans2;
// Calling template function with int parameters.
  ans1 = add<int>(2, 2);
  cout << "Sum of 2 + 2 is: " << ans1 << endl;

  // Calling template function with double parameters.
  ans2 = add<double>(2.5, 3.5);
  cout << "Sum of 2.5 + 3.5 is: " << ans2 << endl;

  return 0;
}
```

Output:

```
Sum of 2 + 2 is: 4
Sum of 2.5 + 3.5 is: 6
```

In the example above, we have defined a template function, namely add(). We can provide multiple data types as arguments for the function.

2. Class Templates

Just like the function templates in C++, we can also use class templates to create a single class that can work with the various data types. Just like function templates, class templates in C++ can make our code shorter and more manageable.

Syntax of the template function:

```
template <class T> class class-name
{
  // class body
}
```

In the syntax above:

- **T** is a placeholder template argument for the data type. T or type of argument will be specified when a class is instantiated.
- **class** is a keyword used to specify a generic type in a template declaration.

Some pre-defined examples of class templates in C++ are LinkedList, Stack, Queue, Array, etc. Let's take an example to understand the working and syntax of class templates in C++.

Example:

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

// Declaring a template class named Test.
template <class T>
class Test
{
private:
  // A variable (answer) of type T so that it can store results of various types.
  T answer;

public:
  // Constructor of Test class.
  Test(T n) : answer(n)
  {
cout << "Inside constructor" << endl;
```

```
    }

    T getNumber()
    {
      return answer;
    }
};

// Main function
int main()
{
    // Creating an object with an integer type.
    Test<int> numberInt(60);

    // Creating an object with double type.
    Test<double> numberDouble(17.27);

// Calling the class method getNumber with different data types:
    cout << "Integer Number is: " << numberInt.getNumber() << endl;
    cout << "Double Number = " << numberDouble.getNumber() << endl;

    return 0;
}
```

Output:

```
Inside constructor
Inside constructor
Integer Number is: 60
Double Number = 17.27
```

In the example above, we have defined a template class (Test) that returns the number of various data types. We have a return type T, meaning they can be of any type.

- **File Handling In C++**

Files are used to store data in a storage device permanently. File handling provides a mechanism to store the output of a program in a file and to perform various operations on it.

File Handling stands for the manipulation of files storing relevant data using a programming language. This enables us to store the data in permanent storage even after the program performs file handling for the same ends of its execution. C++ offers the library **fstream** for file handling. Here we will discuss the classes through which we can perform I/O operations on files. For taking input from the keyboard and printing something on the console, you might have been using the **cin**(character input stream) and **cout**(character output stream) of the **istream** and **ostream** classes. File streams are also somewhat similar to them. Only the console here is replaced by a file. A stream is an abstraction that represents a device on which operations of input and output are performed. A stream can be represented as a source or destination of characters of indefinite length depending on its usage.

In C++ we have a set of file handling methods. These include **ifstream, ofstream**, and **fstream.** These classes are derived from **fstrembase** and from the corresponding **iostream** class. These classes, designed to manage the disk files, are declared in fstream and therefore we must include fstream and therefore we must include this file in any program that uses files.

For achieving file handling we need to follow the following steps:-

- Create a file
- Open a file
- Read from a file
- Write to a file
- Close a file

- **fstream**

The **fstream** is derived from the iostream class and the iostream is further derived from the istream and ostream classes, it provides the input as well as output streams to operate on file. If our stream object belongs to the fstream class, we can perform read and write operations on the file with the same stream object. This class is declared in the **fstream** header file.

- **ofstream:**

The **ofstream** is derived from the ostream class. It provides the output stream to operate on file. The output stream objects can be used to write the sequences of characters to a file. This class is declared in the **fstream** header file.

- **ifstream:**

The **ifstream** is derived from the istream class. It provides the input stream to operate on file. We can use that input stream to read from the file. This class is declared in the **fstream** header file.

- **File Operations in C++**

In C++, four different operations for file handling. They are:

- **open() –** This is used to create a file.
- **read() –** This is used to read the data from the file.
- **write() –** This is used to write new data to file.
- **close() –** This is used to close the file.

- **How to Open Files:**

Before performing any operation on a file, firstly open it. If you need to write to the file, open it using fstream or ofstream objects. If you only need to read from the file, open it using the **ifstream** object.

The three objects, that is, **fstream**, **ofstream**, and **ifstream**, have the open() function defined in them. The function takes this syntax:

open (file_name, mode);

- The **file_name** parameter denotes the name of the file to open.
- The **mode** parameter is optional. It can take any of the following values:

Mode – There are different modes to open a file and it explained in this below chart:

File Open Mode

Name	Description
ios::in	Open file to read
ios::out	Open file to write
ios::app	All the data you write, is put at the end of the file. It calls ios::out
ios::ate	All the data you write, is put at the end of the file. It does not call ios::out
ios::trunc	Deletes all previous content in the file. (empties the file)
ios::nocreate	If the file does not exists, opening it with the open() function gets impossible.
ios::noreplace	If the file exists, trying to open it with the open() function, returns an error.
ios::binary	Opens the file in binary mode.

Program for Opening File:

```
#include<iostream>
#include<fstream>
using namespace std;
int main(){
   fstream FileName;
   FileName.open("FileName", ios::out);
   if (!FileName){
      cout<<"Error while creating the file";
   }
   else{
      cout<<"File created successfully";
      FileName.close();
   }
   return 0;
}
```

Output: File created successfully

Explanation of above code

1. Here we have an iostream library, which is responsible for input/output stream.
2. We also have a fstream library, which is responsible for handling files.
3. Creating an object of the fstream class and named it as 'FileName'.
4. On the above-created object, apply the open() function to create a new file, and the mode is set to 'out' which will allow us to write into the file.
5. Using the 'if' statement to check for the file creation.
6. Prints the message to console if the file doesn't exist.
7. Prints the message to console if the file exists/created.
8. Using the close() function on the object to close the file.

- **Default Open Modes :**

- ✓ ifstream ios::in
- ✓ ofstream ios::out
- ✓ fstream ios::in | ios::out

We can combine the different modes using or symbol |.

- **Write To a File**

To create a file, use either the **ofstream** or **fstream** class, and specify the name of the file.

To write to the file, use the insertion operator (<<).

- **Syntax:**

FileName<<"Insert the text here";

Program for Writing to File:

```
#include<iostream>
#include<fstream>
using namespace std;
int main() {
   fstream FileName;
   FileName.open("FileName.txt", ios::out);
   if (!FileName) {
      cout<<" Error while creating the file ";
```

```
    }
    else {
        cout<<"File created and data got written to file";
        FileName<<"This is a blog posted on Great Learning";
        FileName.close();
    }
    return 0;
}
```

Output: File created and data got written to file

- **Read a File**

To read from a file, use either the **ifstream** or **fstream** class, and the name of the file.

Note that we also use a while loop together with the getline() function (which belongs to the ifstream class) to read the file line by line, and to print the content of the file:

- **Syntax:**

  FileName>>Variable;

- **Content of FileName.txt:**

Hello World.

Program for Reading from File:

```
#include<iostream>
#include <fstream>
using namespace std;
int main() {
    fstream FileName;
    FileName.open("FileName.txt", ios::in);
    if (!FileName) {
        cout<<"File doesn't exist.";
    }
    else {
        char x;
        while (1) {
```

```
            FileName>>x;
            if(FileName.eof())
               break;
            cout<<x;
         }
      }
      FileName.close();
      return 0;
   }
```

Output: Hello World.

- **Closing a file**

Closing a file is a good practice, and it is must to close the file. Whenever the C++ program comes to an end, it clears the allocated memory, and it closes the file. We can perform the task with the help of close() function.

- **Syntax:**

```
FileName.close();
```

Program to Close a File:

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
   fstream FileName;
   FileName.open("FileName.txt", ios::in);
   if (!FileName) {
      cout<<"File doesn’t exist";
   }
   else {
      cout<<"File opened successfully";
      }
   }
   FileName.close();
   return 0;
}
```

- **Append data in a File**

Both data and files must be entered by the user at run-time. Adding new data to a file without erasing the previous one is referred to as appending data.

Example:

// C++ program to open output file 'a.txt' and append data to it.

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

int main()
{
   ofstream fout;  // Create Object of Ofstream
   ifstream fin;
   fin.open("a.txt");
   fout.open ("a.txt",ios::app); // Append mode
   if(fin.is_open())
      fout<< "\n Writing to a file opened from program.\n"; // Writing data to file
   cout<<"\n Data has been appended to file";
   fin.close();
   fout.close(); // Closing the file
   return 0;
}
```

Output: Data has been appended to file

- **Copying a File to Another**

Here, how to develop a C++ program to copy the contents of one file into another file. Given a text file, extract contents from it and copy the contents into another new file. After this, display the contents of the new file.

- **Approach:**
  1. Open the first file which contains data. For example, a file named **"file1.txt"** contains three strings on three separate lines "Programming Language", "By C++ OOPs" and "Happy Coding!".

2. Open the second file to copy the data from the first file.
3. Extract the contents of the first file line by line and write the same content to the second file named “file2.txt” via while loop.
4. Extract the contents of the second file and display it via the while loop.

// C++ program to demonstrate copying the contents of one file into another file

```
#include < bits/stdc++.h>
using namespace std;

int main()
{
   // filestream variables
   fstream f1;
   fstream f2;

   string ch;

   // opening first file to read the content
   f1.open("file1.txt", ios::in);

   // opening second file to write the content copied from the first file
   f2.open("file2.txt", ios::out);

   while (!f1.eof()) {

      // extracting the content of first file line by line
      getline(f1, ch);

      // writing content to second file line by line
      f2 << ch << endl;
   }

   // closing the files
   f1.close();
   f2.close();

   // opening second file to read the content
   f2.open("file2.txt", ios::in);
   while (!f2.eof()) {
```

```
        // extracting the content of second file line by line
        getline(f2, ch);

    // displaying content
        cout << ch << endl;
    }

    // closing file
    f2.close();
    return 0;
}
```

Output:

```
Programming Language
By C++ OOPs
Happy Coding!
```

- **End of File in C++:**

C++ provides a special function, **eof( ),** that returns nonzero (meaning TRUE) when there are no more data to be read from an input file stream, and zero (meaning FALSE) otherwise.

Example:

```
#include <iostream.h>
#include <fstream.h>
#include <assert.h>

int main(void)
{
    int data;          // file contains an undermined number of integer values
    ifstream fin;      // declare stream variable name

    fin.open("myfile",ios::in);    // open file
    assert (!fin.fail( ));
    fin >> data;        // get first number from the file (priming the input statement)
                        // You must attempt to read info prior to an eof( ) test.
    while (!fin.eof( ))      //if not at end of file, continue reading numbers
    {
```

```
        cout<<data<<endl;    //print numbers to screen
        fin >> data;         //get next number from file
    }
    fin.close( );       //close file
    assert(!fin.fail( ));
    return 0;
}
```