# Programming Language

## Exception & File Handling

## *(Day 6)*

**By: Bhaskar Ghosh**
*bjghosh@gmail.com*

Updated: 6 August 2024

# C++ Programming Language

## Object Oriented Programming (Day 5 Part 1)

- Exception Handling in C++
- File Handling in C++
- Some interesting extras
  - Function with variable number of arguments
  - dynamic_cast – Casting Pointers and References
  - New features introduced in C++ 11
- Problem Solving using C++

# C++ Programming Language

## Runtime Error Conditions

- Errors that occur at program runtime can seriously interrupt the normal flow of a program, such as:
  - division by 0, or values that are too large or small for a type
  - no memory available for dynamic allocation
  - errors on file access, for example, file not found
  - attempt to access an invalid address in main memory
  - invalid user input
- Anomalies like these lead to incorrect results and may cause a computer to crash.
- One of the programmer's most important tasks is to predict and handle errors.

# C++ Programming Language

## Traditional Error Handling

- Traditional structured programming languages use normal syntax to handle errors:
  - errors in function calls are indicated by special return values
  - global error variables or flags are set when errors occur
- If a function uses its return value to indicate errors, the return value must be examined

```
if( func() > 0 )
      // Return value positive => o.k.
else
      // Treat errors
```

- Error variables and flags must also be checked after every corresponding action.

# C++ Programming Language

## Exception Handling

- Exception handling is based on keeping the normal functionality of the program separate from error handling.
- The basic idea is that errors occurring in one particular part of the program are reported to another part of the program.
- The calling environment performs central error handling.
- An application program no longer needs to continually check for errors, because in the case of an error, control is automatically transferred to the calling environment.
- An exception that occurs is recorded to the calling environment by means of a throw statement.

```
throw fault;   //throwing exception object
```

# C++ Programming Language

## Exception Handlers – try and catch blocks

- The part of a program that performs central error handling in the calling environment is referred to as an exception handler.
- An exception handler catches the exception object thrown to it and performs error handling.
- We need to specify two things when implementing exception handling:
  - the part of the program that can throw exceptions, and
  - the exception handlers that will process the various exception types.

```
try {
 //this block may throw exception
}
catch (EType1 obj)
{ //handle EType1 exception
}
catch (EType2 obj)
{ //handle EType2 exception
}
catch (...)
{   // General handler for
    // all other exceptions
}
```

# C++ Programming Language

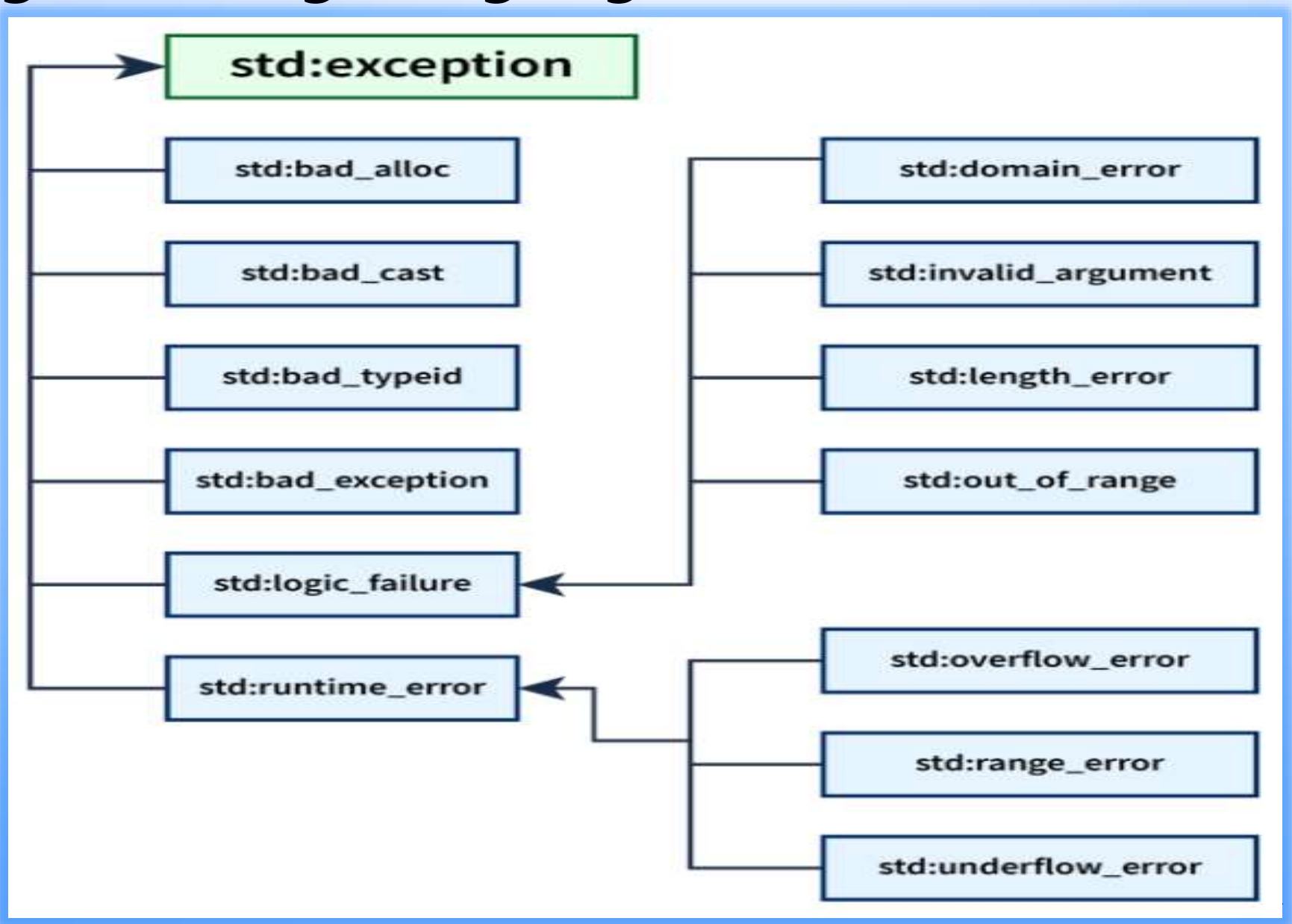
## Exception Handlers – try and catch blocks

- A try block contains the program code in which errors can occur and exceptions can be thrown.
- Each catch block defines an exception handler.
- We can throw any type of data using the throw statement, and then catch it in the catch block.
- We can also throw and then catch one of the built-in Exception Types defined inside the <exception> header file.
- Or, we can create out own User Defined Exception and use that type to handle the exception.

# C++ Programming Language

## C++ Standard Exception Types

**Built-in standard exceptions under the <exception> header file.**



# C++ Programming Language

## User-Defined Exceptions

- An User Defined Exception is the type of exception that is not defined in the standard libraries of C++.
- We can use the C++ **std::exception** class to define objects that can be thrown as exceptions.
- The **std::exception** class provides a ***virtual member function*** called **what()**.
- The **what()** function returns a null-terminated character sequence (string) of type const char*.
- It can be overridden in derived classes to provide a custom description or depiction of the exception.

# C++ Programming Language

## File Handling

- C++ provides a set of classes and methods in the **<fstream>** header, to manage and manipulate files.
- This enables operations on text files (***text mode***) and binary files (***binary mode***).
- The primary classes for file handling in C++ are:
  - **ofstream**: Represents the output file stream and is used to create and write to files.
  - **ifstream**: Represents the input file stream and is used to read from files.
  - **fstream**: Represents the file stream and can be used for both reading from and writing to files.

# C++ Programming Language

## File Operations

1. **Open a File**
   - Before performing any operation on a file, it must be opened.
   - In C++, we use the open() method.
2. **Reading/Writing from/to a File**
   - We can read from a file opened in input mode.
   - We can write to a file opened in output mode.
3. **Close a File**
   - After operations are done on a file, it must be closed.
   - In C++, we use the close() method.

```
ofstream outfile;
outfile.open("example.txt");

ifstream infile;
infile.open("example.txt");
```

```
outfile << "Writing to file";

string str; char ch;
ch = infile.get();
getline(infile, str);
```

```
outfile.close();

infile.close();
```

# C++ Programming Language

## File Opening Modes

| Mode | Syntax | Description |
|---|---|---|
| Read | ios::in | Opens file for reading purpose. |
| Write | ios::out | Opens a file for writing purposes. |
| Binary | ios::binary | All operations will be performed in binary mode. |
| Truncate before Open | ios::trunc | If the file already exists, all content will be removed immediately. |
| Append | ios::app | All provided data will be appended in the associated file. |
| At End | ios::ate | It opens the file and moves the read/write control at the End of the File. The basic difference between the ios::app and this one is that the former will always start writing from the end, but we can seek any particular position with this one. |

# C++ Programming Language

## Opening a File

1. Using constructor of file stream classes

```
ifstream iFile("sample.txt");
```

```
ofstream oFile("sample.txt");
```

2. Using the open method of the file stream object

```
ifstream iFile;
iFile.open("myFile.txt");
```

```
ofstream oFile;
oFile.open("myFile.txt");
```

3. Check whether the file has been successfully opened using is_open()

```
if(ifStream.is_open()){
    // Input File Opened successfully.
}
```

# C++ Programming Language

## Checking the File for errors

1. **By Checking the File Object**

```
ofstream oFile("sample.txt");
if (!my_file) {
      cout << "Error opening the file." << endl;
}
```

2. **Using is_open() function**

```
if(ofStream.is_open()){
    // Output File Opened successfully.
}
```

3. **Using fail() function**

```
if(ifStream.fail()){
  cout << "Error opening the file." << endl;
}
```

# C++ Programming Language

## Checking File State Flags

- The state flags of the file tell about the current state of the file.
- **eof()**
  - returns true if the end of the file is reached while reading the file.
- **fail()**
  - returns true when the read/write operation fails or a format error occurs.
- **bad()**
  - returns true if reading from or writing to a file fails.
- **good()**
  - checks the state of the current stream and returns true if the stream is good for working and hasn't raised any error.
  - good() returns false if any of the above state flags return true; otherwise, it returns true.

# C++ Programming Language

## Reading and Writing Binary Files

1. Open the file in binary write mode

```
ofstream oFile("data.bin", std::ios::binary);
```

2. Use the write function to serialize the object

```
Person p; // the Person object to write
oFile.write(reinterpret_cast<const char*>(&obj), sizeof(MyObject));
```

3. Open the file in binary read mode

```
ifstream iFile("data.bin", std::ios::binary);
```

4. Use the read function to deserialize the object

```
iFile.read(reinterpret_cast<const char*>(&obj), sizeof(MyObject));
```

# C++ Programming Language

## Reading and Writing Person Object from/to a Binary File

- Create a structure Person to declare variables.
- Open binary file to write.
- Check if any error occurs in file opening.
- Initialize the variables with data.
- If file open successfully, write the binary data using write function, and close the file.
- Open the binary file to read.
- Check if any error occurs in file opening.
- If file open successfully, read the binary data file using read function, and close the file.
- Check if any error occurs.

  Print the data.

# Problem Solving using C++

## logic Building and Debugging
## Design & Implementation

# C++ Programming Language

## Design & Implementation using C++

1. **Design a Banking System**
   - Think about the various classes that will be needed, and how the classes will be related.
   - Draw a class diagram to depict these relations.
2. **Implement the Banking System – Version 1**
   - Write the skeleton code for the classes first, using inheritance concepts, constructor, destructor, normal functions, friend function (if required) or virtual function (if required). In this step, write only function prototypes (declarations).
   - Now write code for all the functions.
3. **Enhance the Banking System implementation – Version 2**
   - Enhance the code by adding features like writing and reading to or from a binary file, and displaying the results.
   - Add exception handling.