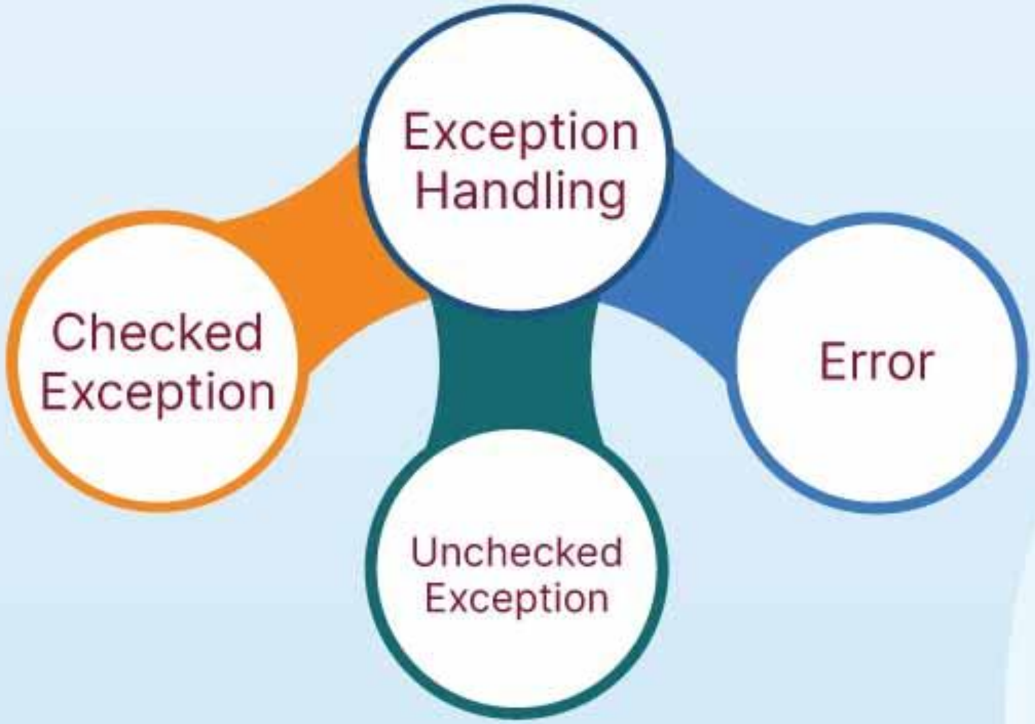
Exception Handling in C++



By – Mohammad Imran

Traditional Error Handling

Introduction

Traditional error handling refers to the methods used in programming to manage errors and exceptions without relying on structured exception handling mechanisms. This approach often involves checking for error conditions manually and responding appropriately within the flow of the program. Here are some key characteristics of traditional error handling:

Traditional Error Handling

Key Characteristics

Return Codes:

- Functions often return an error code or status value to indicate success or failure. For example, a function might return 0 for success and -1 for an error. The calling code must check this return value to determine if an error occurred.

```
int divide(int a, int b) {
    if (b == 0) {
        return -1; // Error code for division by zero
    }
    return a / b; // Success
}
```

By – Mohammad Imran

Traditional Error Handling

Key Characteristics

Conditional Check:

- Programmers typically implement conditional statements (like if statements) to check for error conditions. This can lead to a lot of nested or repetitive code, making it harder to read and maintain.

Error Handling Logic:

- Error handling logic is often interspersed throughout the main code, which can clutter the business logic and make the program harder to understand.

Traditional Error Handling

Key Characteristics

Logging and Messaging:

- Traditional error handling might involve logging errors to a file or displaying error messages to the user, but this is often done manually.

Lack of Separation:

- Error handling is not separated from the main logic, leading to tightly coupled code. This makes it difficult to maintain and can increase the risk of overlooking error handling in some cases.

Traditional Error Handling

Example

```
#include <iostream>
using namespace std;
int divide(int a, int b)
{
   if (b == 0)
   {
      return -1; // Error code for division by zero
   }
   return a / b; // Successful division
}
```

By – Mohammad Imran

Traditional Error Handling

Example

```
int main()
{
   int num1, num2;
   cin >> num1 >> num2;
   int result = divide(num1, num2);
   if (result == -1)
   {
      cout << "Error: Division by zero!" << endl;
   }
   else
   {
      cout << "Result: " << result << endl;
   }
   return 0;
}
```

By – Mohammad Imran

Traditional Error Handling

Limitations

Error Handling Complexity:

✓ As the number of functions increases, tracking and managing return codes can become cumbersome.

Readability:

✓ Code can become harder to read and maintain due to excessive conditional checks and error handling mixed with business logic.

Traditional Error Handling

Limitations

Inconsistent Handling:

✓ Different functions might handle errors differently, leading to inconsistency across the codebase.

Difficult Debugging:

✓ Identifying the source of an error may require more effort, as the error-handling logic is scattered throughout the code.

Error Introduction

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 as bugs or faults.

In C++, an error typically refers to a problem in the code that prevents it from compiling or executing correctly. Errors can be broadly categorized into three types: **syntax errors**, **runtime errors**, and **logical errors**.

Error Types

There are mainly two types of errors in programming by category.

- ✓ Compile Time Error
- ✓ Run Time Error

By – Mohammad Imran

Compile Time Error

Syntax Error

These occur when the code does not follow the rules of the C++ language, leading to compilation failures.

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.

Compile Time Error

Example

```
#include <iostream>
using namespace std;
int main()
{
   cout << "Hello, World!" << std::endl // Missing semicolon
   return 0;
}
```

The missing **semicolon(;)** at the end of the **cout** line will cause a syntax error, preventing the code from compiling.

By – Mohammad Imran

Run Time Error

These occur while the program is running, often due to invalid operations such as division by zero or accessing invalid memory.

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.

Exception

An ***exception*** is an event that occurs during the execution of a program that disrupts the normal flow of instructions during the execution of a program.

An exception is an unexpected problem that arises during the execution of a program our program terminates suddenly with some errors/issues. Exception occurs during the running of the program (runtime).

In other words, run time error is known as Exception.

Exception

Exceptions occur for numerous reasons, including invalid user input, code errors, device failure, the loss of a network connection, insufficient memory to run an application, a memory conflict with another program, a program attempting to divide by zero or a user attempting to open files that are unavailable.

Exception Types

There are two types of exceptions in C++

- ✓ **Synchronous:** Exceptions that happen when something goes wrong because of a mistake in the input data or when the program is not equipped to handle the current type of data it's working with, such as dividing a number by zero.
- ✓ **Asynchronous**: Exceptions that are beyond the program's control, such as disc failure, keyboard interrupts, etc.

Exception Types

Exceptions can come in the following two exception classes:

✓ **Checked exceptions.** Also called *compile-time exceptions*, the compiler checks these exceptions during the compilation process to confirm if the exception is being handled by the programmer. If not, then a compilation error displays on the system. Checked exceptions include SQLException and ClassNotFoundException.

Exception Types

- ✓ **Unchecked exceptions.** Also called *runtime exceptions*, these exceptions occur during program execution. These exceptions are not checked at compile time, so the programmer is responsible for handling these exceptions. Unchecked exceptions do not give compilation errors. Examples of unchecked exceptions include NullPointerException and IllegalArgumentException.

Exception

Example - 1

```
#include <iostream>
using namespace std;
int main()
{
   int a, b, x, y, res;
   cout << "Enter number : ";
   cin >> a;
   cout << "Enter two number : ";
   cin >> x >> y;
   b = x - y;
   res = a / b;
   cout << "Answer = " << res << endl;
   cout << "Completed" << endl;
   return 0;
}
```

OUTPUT

In some cases will run and some case not

By – Mohammad Imran

Exception Handling

Exception handling is the process of responding to unwanted or unexpected events when a computer program runs. Exception handling deals with these events to avoid the program or system crashing, and without this process, exceptions would disrupt the normal operation of a program.

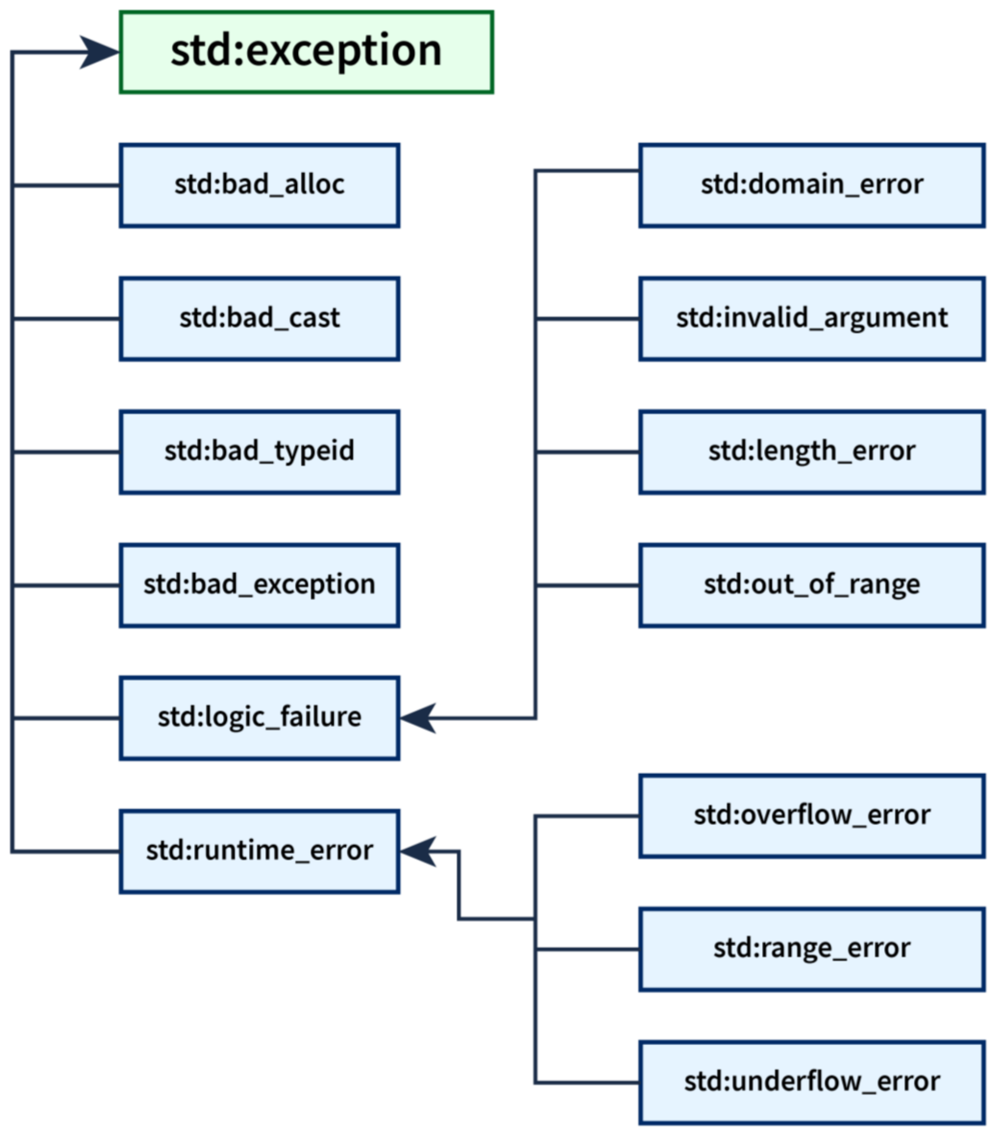
Exception handling differs from error handling in that the former involves conditions an application might catch versus serious problems an application might want to avoid. In contrast, error handling helps maintain the normal flow of software program execution.

By – Mohammad Imran

Exception Handling

Exception Classes

In C++, The C++ library has some built-in standard exceptions are defined under <exception> class that we can use inside our programs. The arrangement of parent-child class hierarchy is shown below:



By – Mohammad Imran

Exception Handling

Exception Classes

Class Name	Use Case
exception	std::exception is the parent class of all the standard C++ exceptions. So, it contains declarations and definitions of other exceptions.
bad_alloc	std::bad_alloc is the type of exception thrown by the allocation functions to report failure to allocate storage.
bad_cast	std::bad_cast is the type of exception thrown when a dynamic_cast to a reference type fails the run-time check.
bad_exception	std::bad_exception is used to handle unexpected exceptions in a C++ program.
bad_typeid	std::bad_typeid is the type of exception thrown when a typeid operator is applied to a dereferenced null pointer value of a polymorphic type.

By – Mohammad Imran

Exception Handling

Exception Classes

Class Name	Use Case
logic_error	std::logic_error exception is the type of exception that can be theoretically detected by reading the code.
domain_error	std::domain_error is the type of exception thrown when a mathematically invalid domain is used.
invalid_argument	std::invalid_argument is the type of exception thrown due to invalid arguments.
length_error	std::length_error is the type of exception thrown when a too big std::string is created.
out_of_range	std::out_of_range is the type of exception that report errors when we are trying to access elements which is outside of the defined range.

By – Mohammad Imran

Exception Handling

Exception Classes

Class Name	Use Case
runtime_error	std::runtime_error exception is the type of exception that theoretically cannot be detected by reading the code.
overflow_error	std::overflow_error exception is the type of exception thrown if a mathematical overflow occurs.
underflow_error	std::underflow_error exception is the type of exception thrown if a mathematical underflow occurs.
range_error	std::range_error is the type of exception thrown when a value is out of the valid range for a specific operation or when an index exceeds the bounds of a container or array.

Exception Handling

Need

The following are the main advantages of exception handling over traditional error handling:

Separation of Error Handling Code from Normal Code*:* There are always if-else conditions to handle errors in traditional error handling codes. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try/catch blocks, the code for error handling becomes separate from the normal flow.

Need

Functions/Methods can handle only the exceptions they choose*:*

A function can throw many exceptions, but may choose to handle some of them. The other exceptions, which are thrown but not caught, can be handled by the caller. If the caller chooses not to catch them, then the exceptions are handled by the caller of the caller. In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it).

Exception Handling

Need

Grouping of Error Types*:* In C++, both basic types and objects can be thrown as exceptions. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, and categorize them according to their types.

Exception Handling

Keywords

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

- ✓ try
- ✓ catch
- ✓ throw

Exception Handling Keywords

try

The **try** keyword represents a block of code that may throw an exception placed inside the try block. It’s followed by one or more catch blocks. If an exception occurs, try block throws that exception.

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 exception while it is being executed.

Exception Handling Keywords

catch

The catch statement represents a block of code that is executed when a particular exception is thrown from the try block. The code to handle the exception is written inside the catch block.

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.

Exception Handling Keywords

throw

An exception in C++ can be thrown using the throw keyword. When a program encounters a throw statement, then it immediately terminates the current function and starts finding a matching catch block to handle the thrown exception.

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.

Exception Handling Keywords

Syntax

```
try
{
    // code that may raise an exception
    throw argument/message;
}
catch (exception)
{
    // code to handle exception
}
```

Exception Handling

Example - 2

```
#include <iostream>
using namespace std;
int main()
{
   int x = -1;
   cout << "Before try \n";
   try
   {
      cout << "Inside try \n";
      if (x < 0)
      {
         throw x;
         cout << "After throw (Never executed) \n";
      }
   }
```

By – Mohammad Imran

Exception Handling

Example - 2

```
    catch (int x)
    {
        cout << "Exception Caught \n";
    }
    cout << "After catch (Will be executed) \n";
    return 0;
}
```

OUTPUT

```
Before try
Inside try
Exception Caught
After catch (Will be executed)
```

By – Mohammad Imran

Exception Handling | Division by Zero | Example - 3

```
#include <iostream>
using namespace std;
class Exception1
{
  public:
   int division(int num, int a, int b)
   {
      int denominator;
      denominator = a-b;
      if (denominator == 0)
      {
         throw "Division by zero!, inf";
      }
      return int(num / denominator);
   }
};
```

By – Mohammad Imran

Exception Handling | Division by Zero | Example - 3

```
int main()
{
   Exception1 e1;
   int numerator, a, b, answer;
   cout << "Enter numerator : ";
   cin >> numerator;
   cout << "Enter two number : ";
   cin >> a >> b;
   try
   {
      answer = e1.division(numerator, a, b);
      cout << "Output: " << answer << endl;
   }
   catch (const char *msg)
   {
      cout << msg << endl;
   }
   return 0;
}
```

By – Mohammad Imran

Exception Handling

Multiple Catch

In C++, you can handle exceptions using multiple catch statements to deal with different types of exceptions separately. This allows you to provide specific handling for various error conditions that might arise during program execution.

Multiple Catch

Syntax

```
try {
    // Code that may throw exceptions
} catch (Type1& e) {
    // Handle exception of Type1
} catch (Type2& e) {
    // Handle exception of Type2
} catch (...) {
    // Handle any other exceptions
}
```

By – Mohammad Imran

Multiple Catch

Example - 4

```cpp
#include <iostream>
#include <stdexcept>
using namespace std;
void mightThrow(int flag)
{
   if (flag == 1)
   {
      throw runtime_error("Runtime error occurred!");
   }
   else if (flag == 2)
   {
      throw invalid_argument("Invalid argument provided!");
   }
   else
   {
      throw out_of_range("Out of range error!");
   }
}
```

Multiple Catch

Example - 4

```
int main()
{
   int flag;
   cout << "Enter a flag : ";
   cin >> flag;
   try
   {
      mightThrow(flag);
   }
   catch (const runtime_error& e)
   {
      cout << "Caught a runtime error: " << e.what() << endl;
   }
```

By – Mohammad Imran

Multiple Catch

Example - 4

```
    catch (const invalid_argument& e)
    {
        cout << "Caught an invalid argument error: " << e.what() <<
        endl;
    }
    catch (const out_of_range& e)
    {
        cout << "Caught an out of range error: " << e.what() << endl;
    }
    catch (...)
    {
        cout << "Caught an unknown error!" << endl;
    }
    return 0;
}
```

By – Mohammad Imran

Exception Handling

Property - 1 **Example - 5**

There is a special catch block called the ‘catch-all’ block, written as catch(…), that can be used to catch all types of exceptions.

In the following program, an int is thrown as an exception, but there is no catch block for int, so the catch(…) block will be executed.

Exception Handling

Property - 1 **Example - 5**

```
#include <iostream>
using namespace std;
class ThrowException
{
  public:
   void exception()
   {
     try
     {
        throw 10;
     }
     catch (char* excp)
     {
        cout << "Caught " << excp;
     }
```

By – Mohammad Imran

Exception Handling | Property - 1 | Example - 5

```
      catch (...)
      {
         cout << "Default Exception\n";
      }
   }
};
int main()
{
      ThrowException te;
      te.exception();
      return 0;
}
```

By – Mohammad Imran

Exception Handling Property - 2 Example - 6

Implicit type conversion doesn’t happen for primitive types.

In the following program, ‘a’ is not implicitly converted to int.

By – Mohammad Imran

Exception Handling

Property - 2 **Example - 6**

```
#include <iostream>
using namespace std;
int main()
{
   try
   {
      throw 'a';
   }
   catch (int x)
   {
      cout << "Caught " << x;
   }
   catch (...)
   {
      cout << "Default Exception\n";
   }
   return 0;
}
```

OUTPUT

```
Default Exception
```

By – Mohammad Imran

Exception Handling

Property - 3 **Example - 7**

If an exception is thrown and not caught anywhere, the program terminates abnormally.

In the following program, a char is thrown, but there is no catch block to catch the char.

We can change this abnormal termination behavior by writing our unexpected function.

Exception Handling

Property - 3 **Example - 7**

```
#include <iostream>
using namespace std;
int main()
{
   try
   {
      throw 'a';
   }
   catch (int x)
   {
      cout << "Caught ";
   }
   return 0;
}
```

OUTPUT

```
terminate called after
throwing an instance of
'char'
```

By – Mohammad Imran

Exception Handling

Property - 4 **Example - 8**

In C++, all exceptions are unchecked, i.e., the compiler doesn’t check whether an exception is caught or not. So, it is not necessary to specify all uncaught exceptions in a function declaration. However, exception-handling it’s a recommended practice to do so.

The following program compiles fine, but ideally, the signature of fun() should list the unchecked exceptions.

Exception Handling | Property - 4 | Example - 8

```
#include <iostream>
using namespace std;
void fun(int* ptr, int x)
{
   if (ptr == NULL)
      throw ptr;
   if (x == 0)
      throw x;
}
int main()
{
   try
   {
      fun(NULL, 0);
   }
   catch (...)
   {
      cout << "Caught exception from fun()";
   }
   return 0;
}
```

OUTPUT

```
Caught exception
from fun()
```

By – Mohammad Imran

Exception Handling | Property - 4 | Better Way of Example - 8

```
#include <iostream>
using namespace std;
void fun(int* ptr, int x) throw
(*int, int)
{
   if (ptr == NULL)
      throw ptr;
   if (x == 0)
      throw x;
}
int main()
{
   try
   {
      fun(NULL, 0);
   }
   catch (...)
   {
      cout << "Caught exception from
      fun()";
   }
   return 0;
}
```

OUTPUT

```
Caught exception
from fun()
```

By – Mohammad Imran

Exception Handling

Property - 5 **Example - 9**

In C++, try/catch blocks can be nested. Also, an exception can be re-thrown using “throw; “.

The following program shows try/catch blocks nesting.

By – Mohammad Imran

Exception Handling | Property - 5 | Example - 9

```
#include <iostream>
using namespace std;
int main()
{
    try
    {
        try
        {
            throw 20;
        }
        catch (int n)
        {
            cout << "Handle Partially" << endl;
            throw; // Re-throwing an exception
        }
    }
    catch (int n)
    {
        cout << "Handle remaining ";
    }
    return 0;
}
```

OUTPUT

```
Handle Partially
Handle remaining
```

By – Mohammad Imran

Exception Handling

Property - 6 **Example - 10**

When an exception is thrown, all objects created inside the enclosing try block are destroyed before the control is transferred to the catch block.

The following program demonstrates the above property.

Exception Handling

Property - 6 **Example - 10**

```
#include <iostream>
using namespace std;
class Test
{
   public:
    Test()
    {
       cout << "Constructor of Test " << endl;
    }
    ~Test()
    {
       cout << "Destructor of Test " << endl;
    }
};
```

By – Mohammad Imran

Exception Handling

Property - 6 **Example - 10**

```
int main()
{
   try
   {
      Test t1;
      throw 10;
   }
   catch (int i)
   {
      cout << "Caught " << i << endl;
   }
   return 0;
}
```

OUTPUT

```
Constructor of Test
Destructor of Test
Caught 10
```

By – Mohammad Imran

Exception Handling

Property - 6 **Example - 10**

```
int main()
{
   try
   {
      Test t1;
      throw 10;
   }
   catch (int i)
   {
      cout << "Caught " << i << endl;
   }
   return 0;
}
```

OUTPUT

```
Constructor of Test
Destructor of Test
Caught 10
```

By – Mohammad Imran

Array Out of Range

An **Array Out of Range Exception** is an error that occurs when a program attempts to access an element in an array using an index that is outside the valid range of indices for that array. In programming, arrays are indexed collections of elements, and each index corresponds to a specific position in the array.

Array Out of Range

Example

```
#include <iostream>
using namespace std;
void arrayRange(int index)
{
   int x[] = {10, 20, 30, 40, 50};
   cout << x[index] << endl;
   cout << "Execution Completed";
}
int main()
{
   int ind;
   cin >> ind;
   arrayRange(ind);
   return 0;
}
```

OUTPUT

```
3
40
Execution Completed

6
1
Execution Completed
```

By – Mohammad Imran

Array Out of Range

Example - 11

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

int getElement(int arr[], size_t size, size_t index)
{
   if (index >= size)
   {
      throw out_of_range("Index is out of range!");
   }
   return arr[index];
}
```

By – Mohammad Imran

Array Out of Range

Example - 11

```
int main()
{
   const size_t ARR_SIZE = 5;
   int myArray[ARR_SIZE] = {10, 20, 30, 40, 50};
   size_t index;
   cout << "Enter the index (0-4): ";
   cin >> index;
   try
   {
      int value = getElement(myArray, ARR_SIZE, index);
      cout << "Value at index " << index << " is " << value <<
      endl;
   }
```

By – Mohammad Imran

Array Out of Range

Example - 11

```
    catch (const out_of_range& e)
    {
        cout << "Caught an exception: " << e.what() << endl;
    }
    catch (const exception& e)
    {
        cout << "Caught a generic exception: " << e.what() << endl;
    }
    return 0;
}
```

By – Mohammad Imran

Memory Allocation

A Memory Allocation Exception refers to an error that occurs when a program attempts to allocate memory dynamically but fails, typically due to insufficient available memory. In C++, this is commonly associated with the new operator, which throws a std::bad_alloc exception when it cannot allocate the requested memory.

Exception Handling

Key Points

✓ **Dynamic Memory Allocation:**

In C++, memory can be allocated at runtime using operators like new and malloc. This memory is not automatically managed, which means the programmer must ensure it's properly allocated and deallocated.

✓ **Causes:**

- Requesting more memory than the system can provide.
- Fragmentation of available memory leading to allocation failures.
- Exhaustion of the process's memory limits.

Exception Handling

Key Points

- ✓ **Exception Handling:**
  - When new fails to allocate memory, it throws a std::bad_alloc exception.
  - Programs can use try and catch blocks to handle these exceptions gracefully, allowing for error messages or fallback logic instead of crashing.
- ✓ **Impact:**
  - Unhandled memory allocation exceptions can lead to program crashes and undefined behavior.
  - Proper handling is essential for robust and resilient software.

By – Mohammad Imran

Memory Allocation Handling

Example - 12

```
#include <iostream>
#include <new> // For std::bad_alloc
using namespace std;
int main()
{
   size_t arraySize;
   cout << "Enter the number of elements : ";
   cin >> arraySize;
   try
   {
      int* myArray = new int[arraySize];
      for (size_t i = 0; i < arraySize; ++i)
      {
         myArray[i] = i + 1;
```

By – Mohammad Imran

Memory Allocation Handling

Example - 12

```
            cout << "Element " << i << ": " << myArray[i] << endl;
        }
        delete[] myArray;
    }
    catch (const bad_alloc& e)
    {
        cout << "Memory allocation failed: " << e.what() << endl;
    }
    catch (const exception& e)
    {
        cout << "An error occurred: " << e.what() << endl;
    }
    return 0;
}
```

By – Mohammad Imran

Coding Questions

By – Mohammad Imran

Question - 1

Write a C++ program to demonstrate multiple throw in a program and a single catch for all type of throw exception.

Click here to see code

By – Mohammad Imran

Question - 2

Write a C++ program to create user defined exception.

Click here to see code

By – Mohammad Imran

Questions -

- ✓ **Basic Exception Handling:**
  - Write a program that divides two numbers and handles division by zero exceptions.

- ✓ **Custom Exception Class:**
  - Create a custom exception class for handling invalid age inputs in a program that requires users to enter their age.

Click here to see code

By – Mohammad Imran

Questions -

✓ **Array Access:**

- Implement a function that retrieves an element from an array and throws an exception if the index is out of range. Test this function with user input.

✓ **Memory Allocation:**

- Write a program that attempts to allocate memory for a large array and catches the std::bad_alloc exception if the allocation fails.

Click here to see code

By – Mohammad Imran

Questions -

- ✓ **Function Pointer Exception:**
  - Write a program that uses function pointers. Handle exceptions if a function pointer is null when called.

- ✓ **Nested Exception Handling:**
  - Create a scenario where you have nested try and catch blocks, and demonstrate how exceptions propagate up the stack.

Click here to see code

By – Mohammad Imran

Questions -

✓ **Multiple Catch Blocks:**

- Write a function that throws different types of exceptions (e.g., std::runtime_error, std::invalid_argument) and handle them using multiple catch blocks.

✓ **Exception Specifications:**

- Implement a function that is declared with exception specifications (though deprecated in C++11) and discuss the implications.

Click here to see code

By – Mohammad Imran

Questions -

✓ **Using std::exception:**

- Create a custom exception class that inherits from std::exception and overrides the what() method to provide meaningful error messages.

✓ **Resource Management:**

- Write a class that manages a resource (e.g., file handle or dynamic memory). Ensure that the resource is properly released in the event of an exception.

[Click here to see code]

By – Mohammad Imran

Questions -

✓ **Handling Standard Exceptions:**

- Write a program that demonstrates how to catch standard exceptions (std::out_of_range, std::invalid_argument, etc.) and display their messages.

✓ **Re-throwing Exceptions:**

- Implement a function that catches an exception and re-throws it to be handled at a higher level.

Click here to see code

By – Mohammad Imran

Questions -

- ✓ **Assertion and Exception Handling:**
  - Use assertions in a program and show how they can complement exception handling in validating program state.

- ✓ **Custom Exception for a Stack Class:**
  - Implement a stack class with methods for push and pop, throwing exceptions for stack overflow and underflow conditions.

Click here to see code

By – Mohammad Imran

QUIZ

By – Mohammad Imran

Quiz - 1

```
#include <iostream>
using namespace std;
class MException
{
   int x;
  public:
   void accept()
   {
      cout << "Enter value : ";
      cin >> x;
      try
      {
         if(x == 0)
            throw (x);
         if(x == 100)
            throw ("Not Possible");
         if(x == 1000)
            throw (4.5);
         else
         {
            cout << "This is right
            value";
         }
      }
      catch(...)
      {
         cout << "Leave this value";
      }
   }
};
```

By – Mohammad Imran