Exception Handling in C++

Difficulty Level : Medium • Last Updated : 22 Jun, 2022

One of the advantages of C++ over C is Exception Handling. Exceptions are runtime anomalies or abnormal conditions that a program encounters during its execution. There are two types of exceptions: a)Synchronous, b)Asynchronous (i.e., exceptions which are beyond the program's control, such as disc failure, keyboard interrupts etc.). C++ provides the following specialized keywords for this purpose:
try: Represents a block of code that can throw an exception.
catch: Represents a block of code that is executed when a particular exception is thrown.
throw: Used to throw an exception. Also used to list the exceptions that a function throws but doesn't handle itself.

Why Exception Handling?

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

1) *Separation of Error Handling code from Normal Code:* In traditional error handling codes, there are always if-else conditions to handle errors. 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.

2) *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).

3) *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.

C++ Exceptions:

When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an exception (error).

C++ try and catch:

Exception handling in C++ consists of three keywords: try, throw and catch:

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The throw keyword throws an exception when a problem is detected, which lets us create a custom error.

The catch statement allows you to define a block of code to be executed if an error occurs in the try block.

The try and catch keywords come in pairs:

We use the try block to test some code: If the value of a variable “age” is less than 18, we will throw an exception, and handle it in our catch block.

In the catch block, we catch the error if it occurs and do something about it. The catch statement takes a single parameter. So, if the value of age is 15 and that’s why we are throwing an exception of type int in the try block (age), we can pass “int myNum” as the parameter to the catch statement, where the variable “myNum” is used to output the value of age.

If no error occurs (e.g. if age is 20 instead of 15, meaning it will be greater than 18), the catch block is skipped.

Exception Handling in C++

1) The following is a simple example to show exception handling in C++. The output of the program explains the flow of execution of try/catch blocks.

CPP

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

int main()
{
   int x = -1;

   // Some code
   cout << "Before try \n";
   try {
      cout << "Inside try \n";
      if (x < 0)
      {
         throw x;
         cout << "After throw (Never executed) \n";
      }
   }
   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)
```

2) There is a special catch block called the 'catch all' block, written as catch(…), that can be used to catch all types of exceptions. For example, 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.

CPP

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

```
int main()
{
    try  {
       throw 10;
    }
    catch (char *excp)  {
        cout << "Caught " << excp;
    }
    catch (...)  {
        cout << "Default Exception\n";
    }
    return 0;
}
```

Output:

```
Default Exception
```

3) Implicit type conversion doesn't happen for primitive types. For example, in the following program, 'a' is not implicitly converted to int.

CPP

```
#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
```

4) If an exception is thrown and not caught anywhere, the program terminates abnormally. For example, in the following program, a char is thrown, but there is no catch block to catch the char.

CPP

```
#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'

This application has requested the Runtime to terminate it in an
unusual way. Please contact the application's support team for
more information.
```

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

5) A derived class exception should be caught before a base class exception. See this for more details.

6) Like Java, the C++ library has a standard exception class which is the base class for all standard exceptions. All objects thrown by the components of the standard library are derived from this class. Therefore, all standard exceptions can be caught by catching this type

7) Unlike Java, in C++, all exceptions are unchecked, i.e., the compiler doesn't check whether an exception is caught or not (See this for details). So, it is not necessary to specify all uncaught exceptions in a function declaration. Although it's a recommended practice to do so. For example, the following program compiles fine, but ideally the signature of fun() should list the unchecked exceptions.

CPP

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

// This function signature is fine by the compiler, but not recommended.
// Ideally, the function should specify all uncaught exceptions and function
// signature should be "void fun(int *ptr, int x) throw (int *, int)"
void fun(int *ptr, int x)
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}

int main()
{
    try {
       fun(NULL, 0);
    }
    catch(...) {
        cout << "Caught exception from fun()";
    }
    return 0;
}
```

Output:

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

A better way to write the above code:

CPP

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

// Here we specify the exceptions that this function
// throws.
void fun(int *ptr, int x) throw (int *, int) // Dynamic Exception specification
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}

int main()
{
    try {
       fun(NULL, 0);
    }
    catch(...) {
        cout << "Caught exception from fun()";
    }
    return 0;
}
```

Note : The use of Dynamic Exception Specification has been deprecated since C++11. One of the reasons for it may be that it can randomly abort your program. This can happen when you throw an exception of another type which is not mentioned in the dynamic exception specification. Your program will abort itself because in that scenario, it calls (indirectly) terminate(), which by default calls abort().

Output:

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

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

CPP

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

int main()
{
    try {
        try {
            throw 20;
        }
        catch (int n) {
            cout << "Handle Partially ";
            throw; // Re-throwing an exception
        }
    }
    catch (int n) {
        cout << "Handle remaining ";
    }
    return 0;
}
```

Output:

```
Handle Partially Handle remaining
```

A function can also re-throw a function using the same "throw; " syntax. A function can handle a part and ask the caller to handle the remaining.
9) When an exception is thrown, all objects created inside the enclosing try block are destroyed before the control is transferred to the catch block.

CPP

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

class Test {
public:
    Test() { cout << "Constructor of Test " << endl; }
    ~Test() { cout << "Destructor of Test " << endl; }
};

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

Output:

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

10) You may like to try Quiz on Exception Handling in C++.
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

263

Related Articles

1. Comparison of Exception Handling in C++ and Java
2. C++ | Exception Handling | Question 1
3. C++ | Exception Handling | Question 3
4. C++ | Exception Handling | Question 4

Stack Unwinding in C++

Difficulty Level : Medium • Last Updated : 25 Nov, 2021

Stack Unwinding is the process of removing function entries from function call stack at run time. The local objects are destroyed in reverse order in which they were constructed.

Stack Unwinding is generally related to Exception Handling. In C++, when an exception occurs, the function call stack is linearly searched for the exception handler, and all the entries before the function with exception handler are removed from the function call stack. So, exception handling involves Stack Unwinding if an exception is not handled in the same function (where it is thrown). Basically, Stack unwinding is a process of calling the destructors (whenever an exception is thrown) for all the automatic objects constructed at run time.

For example, the output of the following program is:

CPP

```cpp
// CPP Program to demonstrate Stack Unwinding
#include <iostream>
using namespace std;

// A sample function f1() that throws an int exception
void f1() throw(int)
{
    cout << "\n f1() Start ";
    throw 100;
    cout << "\n f1() End ";
}

// Another sample function f2() that calls f1()
void f2() throw(int)
{
    cout << "\n f2() Start ";
    f1();
    cout << "\n f2() End ";
}

// Another sample function f3() that calls f2() and handles
// exception thrown by f1()
void f3()
{
    cout << "\n f3() Start ";
    try {
        f2();
    }
    catch (int i) {
        cout << "\n Caught Exception: " << i;
    }
    cout << "\n f3() End";
}

// Driver Code
int main()
{
    f3();

    getchar();
    return 0;
}
```

Output

```
f3() Start
f2() Start
f1() Start
Caught Exception: 100
f3() End
```

Explanation:

AD

- When f1() throws exception, its entry is removed from the function call stack, because f1() doesn't contain exception handler for the thrown exception, then next entry in call stack is looked for exception handler.
- The next entry is f2(). Since f2() also doesn't have a handler, its entry is also removed from the function call stack.

- The next entry in the function call stack is f3(). Since f3() contains an exception handler, the catch block inside f3() is executed, and finally, the code after the catch block is executed.

Note that the following lines inside f1() and f2() are not executed at all.

```
cout<<"\n f1() End ";  // inside f1()

cout<<"\n f2() End ";  // inside f2()
```

If there were some local class objects inside f1() and f2(), destructors for those local objects would have been called in the Stack Unwinding process.

> ***Note:*** *Stack Unwinding also happens in Java when exception is not handled in same function.*

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

61

Related Articles

1. How to detect Stack Unwinding in a Destructor in C++?
2. stack empty() and stack size() in C++ STL
3. stack swap() in C++ STL
4. stack top() in C++ STL
5. Stack push() and pop() in C++ STL
6. stack emplace() in C++ STL
7. Implementing Stack Using Class Templates in C++
8. How to implement a Stack using list in C++ STL
9. Stack-buffer based STL allocator

Catching Base and Derived Classes as Exceptions in C++ and Java

Difficulty Level : Easy • Last Updated : 02 Mar, 2023

An Exception is an unwanted error or hurdle that a program throws while compiling. There are various methods to handle an exception which is termed exceptional handling.

Let's discuss what is Exception Handling and how we catch base and derived classes as an exception in C++:

- If both base and derived classes are caught as exceptions, then the catch block of the derived class must appear before the base class.
- If we put the base class first then the derived class catch block will never be reached. For example, the following **C++** code prints ***"Caught Base Exception"***.

C++

```
// C++ Program to demonstrate a
// Catching Base Exception
#include <iostream>
using namespace std;

class Base {
};
class Derived : public Base {
};
int main()
{
    Derived d;
    // Some other functionalities
    try {
        // Monitored code
        throw d;
    }
    catch (Base b) {
        cout << "Caught Base Exception";
    }
    catch (Derived d) {
        // This 'catch' block is NEVER executed
        cout << "Caught Derived Exception";
    }
    getchar();
    return 0;
}
```

Output

```
Caught Base Exception
```

*The **output** of the above **C++** code:*

```
prog.cpp: In function ‘int main()’:
prog.cpp:20:5: warning: exception of type ‘Derived’ will be caught
     catch (Derived d) {
     ^
prog.cpp:17:5: warning:    by earlier handler for ‘Base’
     catch (Base b) {
```

In the above C++ code, if we change the order of catch statements then both catch statements become reachable.

AD

Following is the modified program and it prints "Caught Derived Exception"

C++

```
// C++ Program to demonstrate a catching of
// Derived exception and printing it successfully
#include <iostream>
using namespace std;

class Base {};
class Derived : public Base {};
int main()
```

```
{
    Derived d;
    // Some other functionalities
    try {
        // Monitored code
        throw d;
    }
    catch (Derived d) {
        cout << "Caught Derived Exception";
    }
    catch (Base b) {
        cout << "Caught Base Exception";
    }
    getchar(); // To read the next character
    return 0;
}
```

Output

```
Caught Derived Exception
```

Output:

```
Caught Derived Exception
```

In java, catching a base class exception before derived is not allowed by the compiler itself. In C++, the compiler might give a warning about it but compiles the code.

For example, *the following Java code fails in compilation with the error message* ***"exception Derived has already been caught"***

Java

```
// Java Program to demonstrate
// the error filename Main.java
class Base extends Exception {
}
class Derived extends Base {
}
public class Main {
    public static void main(String args[])
    {
        try {
            throw new Derived();
        }
        catch (Base b) {
        }
        catch (Derived d) {
        }
    }
}
```

Error:

```
prog.java:11: error: exception Derived has already been caught
    catch(Derived d) {}
```

In both C++ and Java, you can catch both base and derived classes as exceptions. This is useful when you want to catch multiple exceptions that may have a common base class.

In C++, you can catch base and derived classes as exceptions using the catch block. When you catch a base class, it will also catch any derived classes of that base class. Here's an example:

C++

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

class BaseException : public exception {
public:
    virtual const char* what() const throw() {
        return "Base exception";
    }
};

class DerivedException : public BaseException {
public:
    virtual const char* what() const throw() {
        return "Derived exception";
    }
};

int main() {
    try {
        // code that might throw exceptions
        throw DerivedException();
    } catch (BaseException& e) {
        cout << "Caught exception: " << e.what() << endl;
    }
    return 0;
}
```

Output

```
Caught exception: Derived exception
```

In this example, a BaseException class is defined and a DerivedException class is derived from it. In the main() function, a DerivedException object is thrown. The catch block catches any BaseException object or derived object, and prints a message to the console indicating which exception was caught.

In Java, you can catch base and derived classes as exceptions using the catch block with multiple catch clauses. When you catch a base class, it will also catch any derived classes of that base class.

Here's an example:

Java

```java
class BaseException extends Exception {
    public BaseException() {
        super("Base exception");
    }
}

class DerivedException extends BaseException {
    public DerivedException() {
        super("Derived exception");
    }
}

public class ExceptionExample {
    public static void main(String[] args) {
        try {
            // code that might throw exceptions
            throw new DerivedException();
        } catch (DerivedException e) {
            System.out.println("Caught derived exception: " + e.getMessage());
        } catch (BaseException e) {
            System.out.println("Caught base exception: " + e.getMessage());
        }
    }
}
```

OUTPUT:

```
Caught derived exception: Derived exception
```

62

Related Articles

Catch block and type conversion in C++

Difficulty Level : Medium • Last Updated : 12 Nov, 2021

Predict the output of following C++ program.

C++

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

int main()
{
    try
    {
        throw 'x';
    }
    catch(int x)
    {
        cout << " Caught int " << x;
    }
    catch(...)
    {
        cout << "Default catch block";
    }
}
```

Output:

```
Default catch block
```

In the above program, a character 'x' is thrown and there is a catch block to catch an int. One might think that the int catch block could be matched by considering ASCII value of 'x'. But such conversions are not performed for catch blocks. Consider the following program as another example where conversion constructor is not called for thrown object.

C++

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

class MyExcept1 {};

class MyExcept2
{
public:

    // Conversion constructor
    MyExcept2 (const MyExcept1 &e )
    {
        cout << "Conversion constructor called";
    }
};

int main()
{
    try
    {
        MyExcept1 myexp1;
        throw myexp1;
    }
    catch(MyExcept2 e2)
    {
        cout << "Caught MyExcept2 " << endl;
    }
    catch(...)
    {
        cout << " Default catch block " << endl;
    }
    return 0;
}
```

Output:

AD

```
Default catch block
```

As a side note, the derived type objects are converted to base type when a derived object is thrown and there is a catch block to catch base type. See this GFact for more details.
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

23

Related Articles

1. Java Multiple Catch Block
2. Difference between Type Casting and Type Conversion
3. Conversion of Struct data type to Hex String and vice versa
4. Type Conversion in C++
5. Implicit Type Conversion in C with Examples
6. Type Conversion in C
7. Flow control in try catch finally in Java
8. Ask(), Wait() and Answer() Block in Scratch Programming
9. Where is an object stored if it is created inside a block in C++?
10. C++ Program for Block swap algorithm for array rotation

Previous Next

Article Contributed By :

GeeksforGeeks

Exception Handling and Object Destruction in C++

Difficulty Level : Easy • Last Updated : 20 Jan, 2023

An exception is termed as an unwanted error that arises during the runtime of the program. The practice of separating the anomaly-causing program/code from the rest of the program/code is known as Exception Handling.

An object is termed as an instance of the class which has the same name as that of the class. A destructor is a member function of a class that has the same name as that of the class but is preceded by a **'~'** (tilde) sign, also it is automatically called after the scope of the code runs out. The practice of pulverizing or demolition of the existing object memory is termed *object destruction*.
In other words, the class of the program never holds any kind of memory or storage, it is the object which holds the memory or storage and to deallocate/destroy the memory of created object we use destructors.

For Example:

CPP

```cpp
// C++ Program to show the sequence of calling
// Constructors and destructors
#include <iostream>
using namespace std;

// Initialization of class
class Test {
public:
    // Constructor of class
    Test()
    {
        cout << "Constructing an object of class Test "
             << endl;
    }

    // Destructor of class
    ~Test()
    {
        cout << "Destructing the object of class Test "
             << endl;
    }
};

int main()
{
    try {
        // Calling the constructor
        Test t1;
        throw 10;

    } // Destructor is being called here
      // Before the 'catch' statement
    catch (int i) {
        cout << "Caught " << i << endl;
    }
}
```

Output:

AD

```
Constructing an object of class Test
Destructing the object of class Test
Caught 10
```

When an exception is thrown, destructors of the objects (whose scope ends with the try block) are automatically called before the catch block gets executed. That is why the above program prints "***Destructing an object of Test***" before "***Caught 10***".

What Happens When an Exception is Thrown From a Constructor?

Example:

CPP

```cpp
// C++ Program to show what really happens
// when an exception is thrown from
// a constructor
#include <iostream>
using namespace std;

class Test1 {
public:
    // Constructor of the class
    Test1()
    {
        cout << "Constructing an Object of class Test1"
             << endl;
    }
    // Destructor of the class
    ~Test1()
    {
        cout << "Destructing an Object the class Test1"
             << endl;
    }
};

class Test2 {
public:
    // Following constructor throws
    // an integer exception
    Test2() // Constructor of the class
    {
        cout << "Constructing an Object of class Test2"
             << endl;
        throw 20;
    }
    // Destructor of the class
    ~Test2()
    {
        cout << "Destructing the Object of class Test2"
             << endl;
    }
};

int main()
{
    try {
        // Constructed and destructed
        Test1 t1;

        // Partially constructed
        Test2 t2;

        // t3 is not constructed as
        // this statement never gets executed
        Test1 t3; // t3 is not called as t2 is
                  // throwing/returning 'int' argument which
                  // is not accepted
                  //  is the class test1'
    }
    catch (int i) {
        cout << "Caught " << i << endl;
    }
}
```

Output:

```
Constructing an Object of class Test1
Constructing an Object of class Test2
Destructing an Object the class Test1
Caught 20
```

Destructors are only called for the completely constructed objects. When the constructor of an object throws an exception, the destructor for that object is not called.

Predict the output of the following program:

CPP

```
// C++ program to show how many times
// Constructors and destructors are called
#include <iostream>
using namespace std;

class Test {
    static int count; // Used static to initialise the scope
                      // Of 'count' till lifetime
    int id;

public:
  // Constructor
    Test()
    {
        count++;
        id = count;
        cout << "Constructing object number " << id << endl;
        if (id == 4)
            throw 4;
    }
  // Destructor
    ~Test()
    {
        cout << "Destructing object number " << id << endl;
    }
};

int Test::count = 0;
```

```
// Source code
int main()
{
    try {
        Test array[5];
    }
    catch (int i) {
        cout << "Caught " << i << endl;
    }
}
```

Output :

```
Constructing object number 1
Constructing object number 2
Constructing object number 3
Constructing object number 4
Destructing object number 3
Destructing object number 2
Destructing object number 1
Caught 4
```

40

Related Articles

1. Virtual destruction using shared_ptr in C++
2. Comparison of Exception Handling in C++ and Java
3. Top C++ Exception Handling Interview Questions and Answers
4. Exception Handling in C++
5. C++ | Exception Handling | Question 1
6. C++ | Exception Handling | Question 3
7. C++ | Exception Handling | Question 4
8. C++ | Exception Handling | Question 5
9. C++ | Exception Handling | Question 6