Exception Handling in C++

Errors can be broadly categorized into two types. We will discuss them one by one.

1. Compile Time Errors
2. Run Time Errors

Compile Time Errors – Errors caught during compiled time is called Compile time errors. Compile time errors include library reference, syntax error or incorrect class import.

Run Time Errors - They are also known as exceptions. An exception caught during run time creates serious issues.

Errors hinder normal execution of program. Exception handling is the process of handling errors and exceptions in such a way that they do not hinder normal execution of the system. For example, User divides a number by zero, this will compile successfully but an exception or run time error will occur due to which our applications will be crashed. In order to avoid this we'll introduce exception handling technics in our code.

In C++, Error handling is done using three keywords:

- try
- catch
- throw

Syntax:

```
try
{
    //code
    throw parameter;
}
catch(exceptionname ex)
{
    //code to handle exception
}
```

`try` block

The code which can throw any exception is kept inside(or enclosed in) a `try` block. Then, when the code will lead to any error, that error/exception will get caught inside the `catch` block.

`catch` block

`catch` block is intended to catch the error and handle the exception condition. We can have multiple catch blocks to handle different types of exception and perform different actions when the exceptions occur. For example, we can display descriptive messages to explain why any particular excpetion occured.

`throw` statement

It is used to throw exceptions to exception handler i.e. it is used to communicate information about error. A `throw` expression accepts one parameter and that parameter is passed to handler.

`throw` statement is used when we explicitly want an exception to occur, then we can use `throw` statement to throw or generate that exception.

Understanding Need of Exception Handling

Let's take a simple example to understand the usage of try, catch and throw.

Below program compiles successfully but the program fails at runtime, leading to an exception.

```
#include <iostream>#include<conio.h>
using namespace std;
int main()
{
    int a=10,b=0,c;
    c=a/b;
    return 0;
}
```

The above program will not run, and will show **runtime error** on screen, because we are trying to divide a number with **0**, which is not possible.

How to handle this situation? We can handle such situations using exception handling and can inform the user that you cannot divide a number by zero, by displaying a message.

Using `try`, `catch` and `throw` Statement

Now we will update the above program and include exception handling in it.

```
#include <iostream>
#include<conio.h>
using namespace std;
int main()
{
    int a=10, b=0, c;
    // try block activates exception handling
    try
    {
        if(b == 0)
        {
            // throw custom exception
            throw "Division by zero not possible";
            c = a/b;
        }
    }
    catch(char* ex) // catches exception
    {
        cout<<ex;
    }
    return 0;
}
```

OUTPUT:

```
Division by zero not possible
```

In the code above, we are checking the divisor, if it is zero, we are throwing an exception message, then the `catch` block catches that exception and prints the message.

Doing so, the user will never know that our program failed at runtime, he/she will only see the message "Division by zero not possible".

This is **gracefully handling** the exception condition which is why exception handling is used.

Using Multiple `catch` blocks

Below program contains multiple `catch` blocks to handle different types of exception in different way.

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{
    int x[3] = {-1,2};
    for(int i=0; i<2; i++)
    {
        int ex = x[i];
        try
        {
            if (ex > 0)
                // throwing numeric value as exception
                throw ex;
            else
                // throwing a character as exception
                throw 'ex';
        }
        catch (int ex)  // to catch numeric exceptions
        {
            cout << "Integer exception\n";
        }
        catch (char ex) // to catch character/string exceptions
        {
            cout << "Character exception\n";
        }
    }
}
```

OUTPUT:

```
Integer exception

Character exception
```

The above program is self-explanatory, if the value of integer in the array `x` is less than 0, we are throwing a numeric value as exception and if the value is greater than 0, then we are throwing a character value as exception. And we have two different `catch` blocks to catch those exceptions.

Generalized `catch` block in C++

Below program contains a generalized `catch` block to catch any uncaught errors/exceptions. `catch(...)` block takes care of all type of exceptions.

```
#include <iostream>
#include<conio.h>
using namespace std;

int main()
{
    int x[3] = {-1,2};
    for(int i=0; i<2; i++)
    {
        int ex=x[i];
        try
        {
            if (ex > 0)
                throw ex;
            else
                throw 'ex';
        }
        // generalised catch block
        catch (...)
        {
            cout << "Special exception\n";
        }
    }
return 0;
}
```

OUTPUT:

```
Special exception
Special exception
```

In the case above, both the exceptions are being catched by a single `catch` block. We can even have separate `catch` blocks to handle integer and character exception along with th generalised `catch` block.

Standard Exceptions in C++

There are some standard exceptions in C++ under <exception> which we can use in our programs.

They are arranged in a parent-child class hierarchy which is depicted below:

- **std::exception** - Parent class of all the standard C++ exceptions.
- **logic_error** - Exception happens in the internal logical of a program.
  - **domain_error** - Exception due to use of invalid domain.
  - **invalid argument** - Exception due to invalid argument.
  - **out_of_range** - Exception due to out of range i.e. size requirement exceeds allocation.
  - **length_error** - Exception due to length error.
- **runtime_error** - Exception happens during runtime.
  - **range_error** - Exception due to range errors in internal computations.
  - **overflow_error** - Exception due to arithmetic overflow errors.
  - **underflow_error** - Exception due to arithmetic underflow errors
- **bad_alloc** - Exception happens when memory allocation with new() fails.
- **bad_cast** - Exception happens when dynamic cast fails.
- **bad_exception** - Exception is specially designed to be listed in the dynamic-exception-specifier.
- **bad_typeid** - Exception thrown by typeid.

← Prev (file-streams-in-cpp.php)

Next → (memory-management-in-cpp.php)

STL in C++
(/cpp/stl/)

C++ Tests
(/tests/?subject=cpp)

C++ Programs
(/cpp/programs)

What is Studytonight?

About Us (/about)

Authors (/authors)

Collaborate (/collaborate)

Testimonials (/testimonials)

Privacy Policy (/privacy)

Terms (/terms)

Contact Us (/contact)

Suggest (/suggest)

Tutorials

Android (/android)

Core Java (/java)

C++ (/cpp)

Data Structures (/data-structures)

Python (/python)

Network Programming (/network-programming-in-python)

DBMS & SQL (/dbms)

Servlet (/servlet)

More... (/library)

Tests

Core Java (/tests)

Android (/tests/?subject=android)

C++ (/tests?/?subject=cpp)

DBMS (/tests?/?subject=dbms)

C Language (/tests?/?subject=c)

More... (/tests)

Learn to Code