

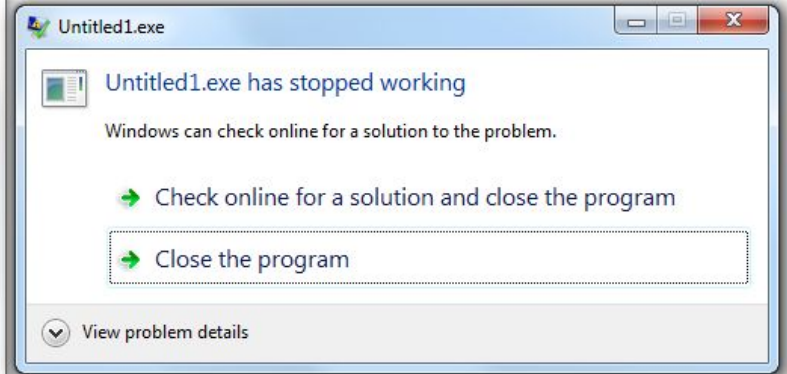
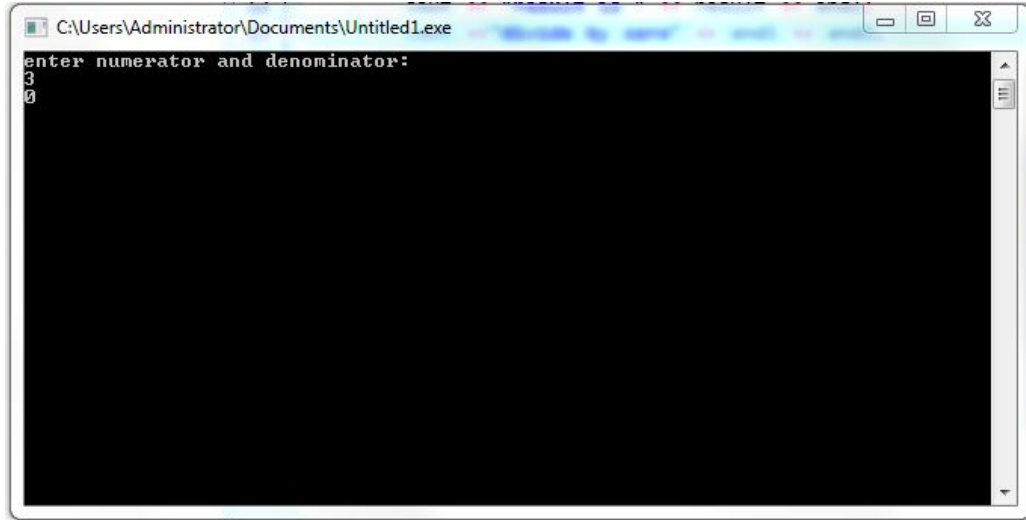
EXCEPTION HANDLING



EXCEPTION HANDLING

```
#include <iostream>
using namespace std;
double zeroDivision(int x, int y) {
    return (x / y);}
int main() {
    int numerator;
    int denominator;
    double result;
    cout << "enter numerator and denominator: " << endl;
    cin>>numerator>>denominator; // 3 and 0
    result = zeroDivision(numerator, denominator);
    cout << "result is " << result << endl;
    cout <<"divide by zero" << endl << endl;
    return 0;}
```

EXCEPTION HANDLING



EXCEPTIONS

Exception handling in C++ provides you with a way of handling unexpected circumstances like runtime errors. So whenever an unexpected circumstance occurs, the program control is transferred to special functions known as handlers

An exception is an abnormal condition that arises in a code sequence at run time

In other words, an exception is a run-time error

HANDLERS

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords:

- try
- catch
- throw

HANDLERS

Try - the try block identifies the code block for which certain exceptions will be activated. It should be followed by one/more catch blocks

Catch - a program uses an exception handler to catch an exception. It is added to the section of a program where you need to handle the problem. It's done using the catch keyword

Throw - when a program encounters a problem, it throws an exception. The throw keyword helps the program perform the throw

THROWING EXCEPTIONS

Exceptions can be thrown anywhere within a code block using throw statement. The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown

```
double division(int a, int b) {  
    if( b == 0 ) {  
        throw "Division by zero condition!"; //throw b;  
    }  
    return (a/b);  
}
```

TRY

A block of code which may cause an exception is typically placed inside the try block. It's followed by one or more catch blocks. If an exception occurs, it is thrown from the try block.

```
try {  
    // protected code  
}
```


CATCHING EXPRESSIONS

The catch block following the try block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try {  
    // protected code  
} catch( ExceptionName exception1 ) {  
    // code to handle ExceptionName exception  
}
```

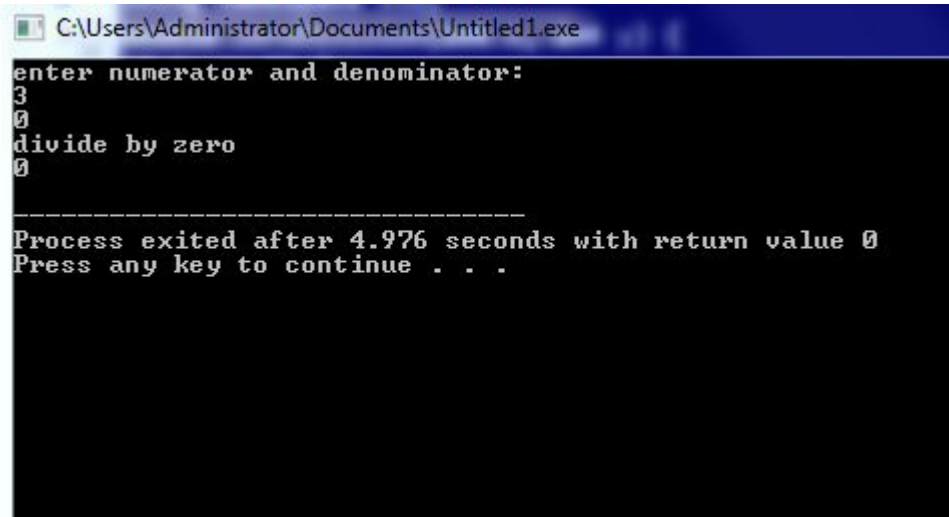
CATCHING EXPRESSIONS

The `ExceptionName` is the name of the exception to be caught.

The `exception1` are your defined names for referring to the exceptions.

EXAMPLE

```
#include <iostream>
using namespace std;
double zeroDivision(int x, int y) {
    if (y == 0) {
        throw y;}
    return (x / y);}
int main() {
    int numerator;
    int denominator;
    double result;
    cout << "enter numerator and denominator: " << endl;
    cin>>numerator>>denominator;
    try {
        result = zeroDivision(numerator, denominator);
        cout << "result is " << result << endl;}
    catch (int ex) {
        cout <<"divide by zero" << endl << ex << endl;}
    return 0;}
```



```
C:\Users\Administrator\Documents\Untitled1.exe
enter numerator and denominator:
3
0
divide by zero

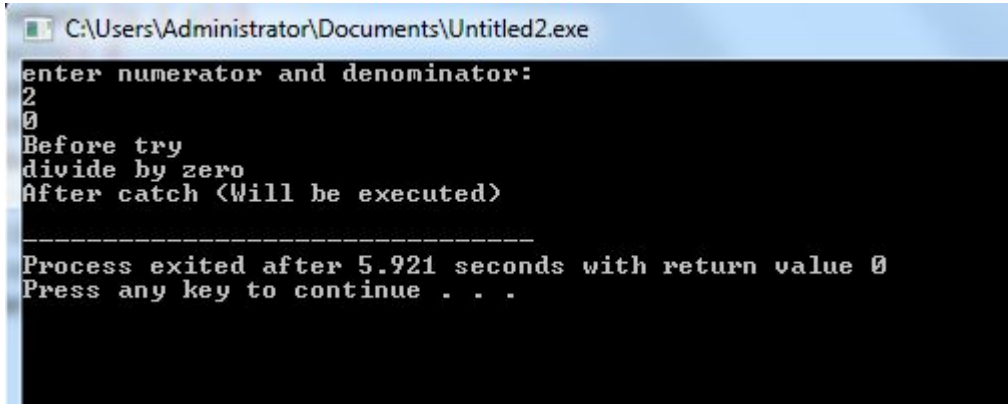
-----
Process exited after 4.976 seconds with return value 0
Press any key to continue . . .
```

EXAMPLE

```
#include <iostream>
#include <exception>
using namespace std;
double zeroDivision(int x, int y) {
    if (y == 0) {
        throw y;
    }
    cout << "After throw (Never executed) \n";
    return (x / y);
}
int main() {
    int numerator;
    int denominator;
    double result;
    cout << "enter numerator and denominator: " << endl;
    cin >> numerator >> denominator;
    cout << "Before try \n";
```

EXAMPLE

```
try {  
    result = zeroDivision(numerator, denominator);  
    cout << "result is " << result << endl; }  
catch (int e) {  
    cout << "divide by zero" << endl; }  
cout << "After catch (Will be executed) \n";  
return 0;}
```



```
C:\Users\Administrator\Documents\Untitled2.exe  
enter numerator and denominator:  
2  
0  
Before try  
divide by zero  
After catch (Will be executed)  
-----  
Process exited after 5.921 seconds with return value 0  
Press any key to continue . . .
```

EXAMPLE - WITHOUT CATCH

```
#include <iostream>
#include <exception>
using namespace std;
double zeroDivision(int x, int y) {
    if (y == 0) {
        throw y;}
    return (x / y);}
int main() {
    int numerator;
    int denominator;
    double result;
    cout << "enter numerator and denominator: " << endl;
    cin>>numerator>>denominator;
    try {
        result = zeroDivision(numerator, denominator);
        cout << "result is " << result << endl;    }
    return 0;}
```

In function 'int main()':

[Error] expected 'catch' before numeric constant

MULTIPLE CATCH CLAUSES

In some cases, more than one exception could be raised by a single piece of code

To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception

After one catch statement executes, the others are bypassed

MULTIPLE CATCH CLAUSES

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations

```
try { // protected code }  
catch( ExceptionName e1 )  
{ // catch block }  
  
catch( ExceptionName e2 ) { // catch block }  
  
catch( ExceptionName eN ) { // catch block }
```


MULTIPLE CATCH CLAUSES

```
#include <iostream>
#include <exception>
using namespace std;
double zeroDivision(int x, int y) {
    if (y == 0) {
        throw y;}
    return (x / y);}
int main() {
    int numerator;
    int denominator;
    double result;
    cout << "enter numerator and denominator: " << endl;
    cin>>numerator>>denominator;

    try {
        result = zeroDivision(numerator, denominator);
        cout << "result is " << result << endl;    }
```

MULTIPLE CATCH CLAUSES

```
catch (int e) {  
    cout <<"divide by zero" << endl << e << endl; }  
    catch (int e) {  
        cout <<"another exception" << endl << e << endl; }  
        catch (int e) {  
            cout <<"another exception" << endl << e << endl; }  
    return 0;}
```

CATCH ALL BLOCK

If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows -

```
try { // protected code }  
    catch(...)  
{ // code to handle any exception }
```

CATCH ALL BLOCK

```
#include <iostream>
using namespace std;
double zeroDivision(int x, int y) {
    if (y == 0) {
        throw y;}
    return (x / y);}
int main() {
    int numerator; int denominator; double result;
    cout << "enter numerator and denominator: " << endl;
    cin>>numerator>>denominator;
    try {
        result = zeroDivision(numerator, denominator);
        cout << "result is " << result << endl;  }
    catch(...){
        cout <<"divide by zero" << endl << endl;  }
    return 0;}
```

CATCH ALL BLOCK

```
#include <iostream>
using namespace std;
double zeroDivision(int x, int y) {
    if (y == 0) {
        throw y;}
    return (x / y);}
int main() {
    int numerator; int denominator; double result;
    cout << "enter numerator and denominator: " << endl;
    cin>>numerator>>denominator;
    try {
        result = zeroDivision(numerator, denominator);
        cout << "result is " << result << endl;    }
    catch(...){
        cout <<"divide by zero" << endl << endl;    }
    return 0;}
```

NO PROPER CATCH

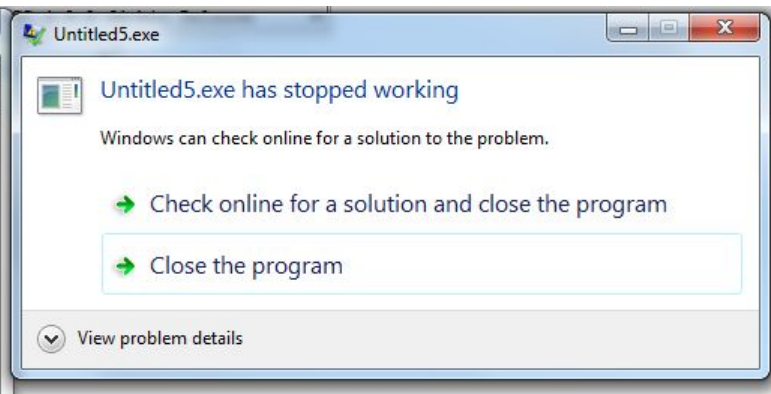
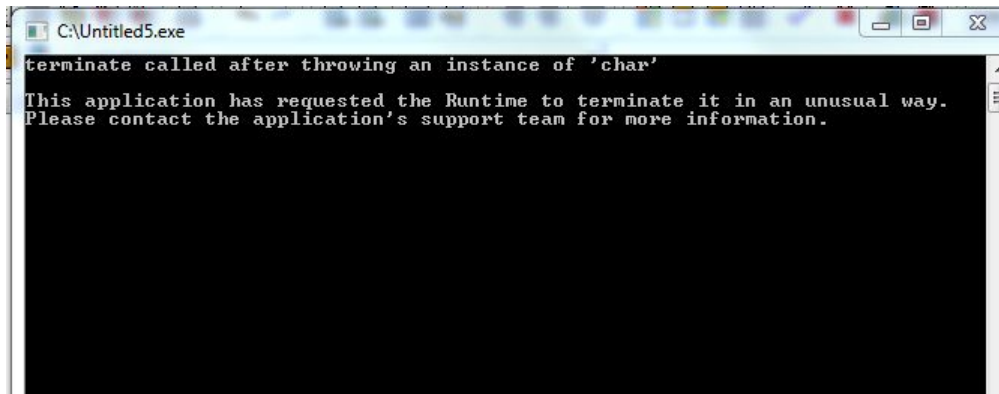
If an exception is thrown and not caught anywhere, the program terminates abnormally.

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

NO PROPER CATCH

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

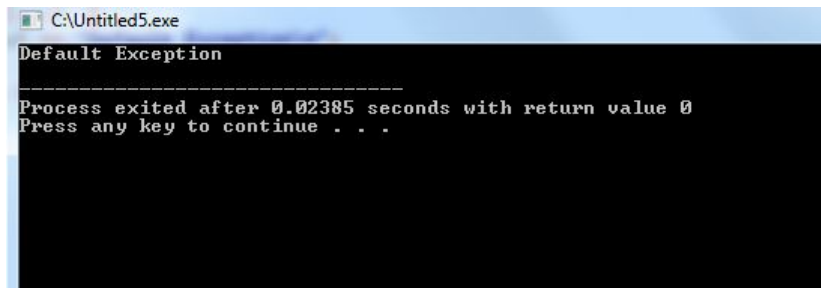
```
try {  
    throw 'a';  
} catch (int x) {  
    cout << "Caught ";  
}  
return 0;}
```



CORRECT VERSION

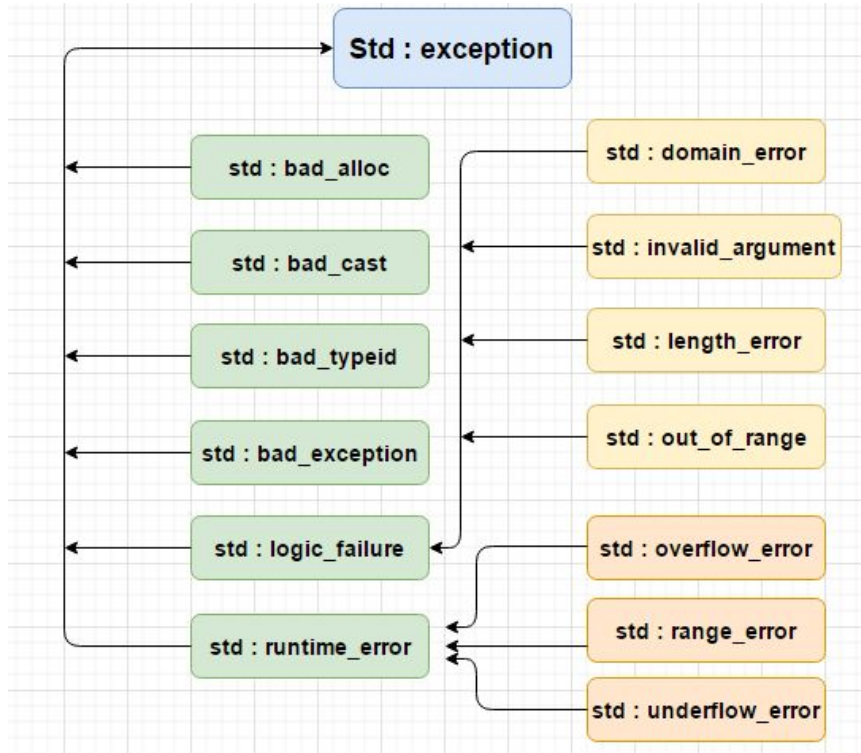
```
#include <iostream>
using namespace std;

int main()
{
    try {
        throw 'a';
    }
    catch (int x) {
        cout << "Caught " << x;
    }
    catch (...) {
        cout << "Default Exception\n";
    }
    return 0;
}
```



```
C:\Untitled5.exe
Default Exception
-----
Process exited after 0.02385 seconds with return value 0
Press any key to continue . . .
```


C++ STANDARD EXCEPTIONS



C++ STANDARD EXCEPTIONS

C++ provides a list of standard exceptions defined in `<exception>` which we can use in our programs

Exception	Description
<code>std::exception</code>	An exception and parent class of all the standard C++ exceptions.
<code>std::bad_alloc</code>	This can be thrown by <code>new</code> .
<code>std::bad_cast</code>	This can be thrown by <code>dynamic_cast</code> .
<code>std::bad_exception</code>	This is useful device to handle unexpected exceptions in a C++ program
<code>std::bad_typeid</code>	This can be thrown by <code>typeid</code> .
<code>std::logic_error</code>	An exception that theoretically can be detected by reading the code.
<code>std::domain_error</code>	This is an exception thrown when a mathematically invalid domain is used
<code>std::invalid_argument</code>	This is thrown due to invalid arguments.
<code>std::length_error</code>	This is thrown when a too big <code>std::string</code> is created
<code>std::out_of_range</code>	This can be thrown by the <code>at</code> method from for example a <code>std::vector</code> and <code>std::bitset<>::operator[]()</code> .
<code>std::runtime_error</code>	An exception that theoretically can not be detected by reading the code.
<code>std::overflow_error</code>	This is thrown if a mathematical overflow occurs.
<code>std::range_error</code>	This is occurred when you try to store a value which is out of range.
<code>std::underflow_error</code>	This is thrown if a mathematical underflow occurs.

STD::BAD_ALLOC

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

int main () {
    try {
        int* myarray= new int[1000000000000000000];
    }

    catch (std::bad_alloc& e) {
        cout << "Standard exception: " << e.what() << endl;
    }
    return 0;
}
```

FILE EXCEPTIONS

Types of File Exceptions

File exceptions can be broadly categorized into several key types:

Exception Type	Description	Common Scenarios
<code>std::ios_base::failure</code>	Base exception for I/O stream errors	File not found, permission denied
<code>std::ifstream::failure</code>	Input file stream specific exceptions	Read errors, invalid file state
<code>std::ofstream::failure</code>	Output file stream specific exceptions	Write permission issues, disk full

STREAM VALIDATION

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

void checkStreamState(const string& filename) {
    ifstream file(filename);
    // Check various stream state flags
    if (file.fail()) {
        cerr << "File opening failed" << endl;
    }
    if (file.bad()) {
        cerr << "Unrecoverable stream error" << endl;
    }
    if (file.eof()) {
        cerr << "End of file reached" << endl;
    }
}
```

FILE VALIDATION

```
#include <fstream>
#include <iostream>
#include <stdexcept>
using namespace std;
class FileErrorHandler {
public:
    static bool validateFileOperation(const string& filename) {
        try {
            ifstream file(filename);
            // Multiple error detection mechanisms
            if (!file.is_open()) {
                throw runtime_error("Cannot open file");}
        }
```

FILE VALIDATION

```
        // Additional file validation
        file.exceptions(ifstream::failbit | ifstream::badbit);

        // Perform file content validation
        string line;
        if (!getline(file, line)) {
            throw runtime_error("Empty or unreadable file");
        }
        return true;
    }
    catch (const exception& e) {
        cerr << "File Error: " << e.what() << endl;
        return false;
    }
};
```

FILE VALIDATION

```
int main() {  
    string testFile = "/path/to/test/file.txt";  
    bool isValid =  
FileErrorHandler::validateFileOperation(testFile);  
  
    cout << "File Validation Result: "  
          << (isValid ? "Success" : "Failure")  
          <<endl;  
  
    return 0;  
}
```


CUSTOM EXCEPTIONS

CUSTOM EXCEPTION

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

class NegativeNumberException : public exception {
public:
    //const char* what() const throw()
    const char* what() const noexcept override
    {
        return "Custom Error: Negative number not allowed!";
    }
};
```

CUSTOM EXCEPTION

```
int squareRoot(int number) {  
    if (number < 0) {  
        throw NegativeNumberException();  
    }  
    // Simulating square root logic  
    return number * number; // just to keep it simple here  
}
```

CUSTOM EXCEPTION

```
int main() {  
    try {  
        int num;  
        cout << "Enter a positive number: ";  
        cin >> num;  
  
        int result = squareRoot(num);  
        cout << "Square of the number is: " << result << endl;  
    }  
    catch (const exception& e) {  
        cerr << "Exception caught: " << e.what() << endl;  
    }  
  
    return 0;  
}
```

CUSTOM EXCEPTION

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

// Define a new exception class that inherits from std::exception

class MyException : public exception {
private:
    string message;
public:
    MyException(const char* msg)
        : message(msg)
    {
    }

    const char* what() const noexcept override
    {
        return message.c_str();
    }
};
```

CUSTOM EXCEPTION

```
int main()
{
    try {
        // Throw our custom exception
        throw MyException("This is a custom exception");
    }
    catch (MyException& e) {
        // Catch and handle our custom exception
        cout << "Caught an exception: " << e.what() << endl;
    }

    return 0;
}
```

HOME TASK

You are tasked with developing a shopping cart system for an e-commerce website. The cart needs to hold up to 10 unique items, which can belong to three different product categories: Book, Electronics, and Clothing.

To manage the cart, you will create a class called UniqueCart, which will handle adding, removing, and checking the existence of items.

You need to implement an add method that allows items to be added to the cart, ensuring that duplicate items cannot be added. If an attempt is made to add an item that already exists, you should throw a "Duplicate Item Exception".

HOME TASK

The remove method should allow items to be removed from the cart, but if the item does not exist, it should throw an "Item Not Found Exception".

Additionally, the cart must be able to hold no more than 10 items at a time, and if a user tries to exceed this limit, an "Out of Bound Exception" should be thrown.

You will also need to implement a contains method to check whether a specific item exists in the cart, returning a boolean result. The cart should be capable of managing a collection of these different product types in a way that handles the exceptions and maintains the constraints specified.