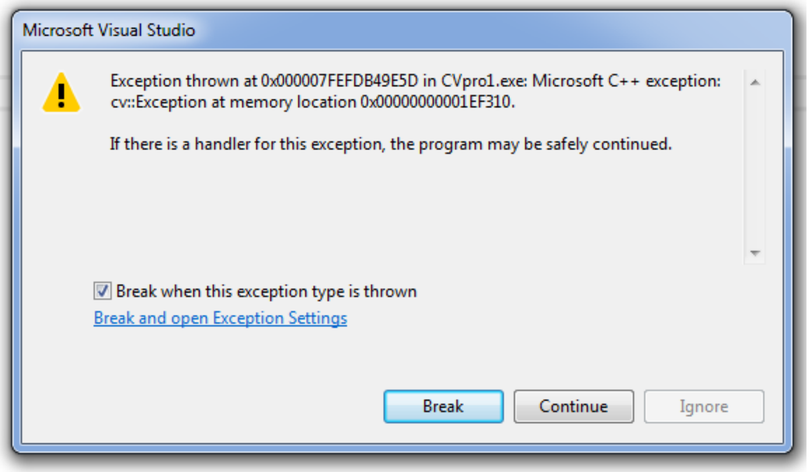
EXCEPTION HANDLING IN C++

What is exception?

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero. Exceptions provide a way to transfer control from one part of a program to another. For an instance, following is a type of exception that one might face while running any program.



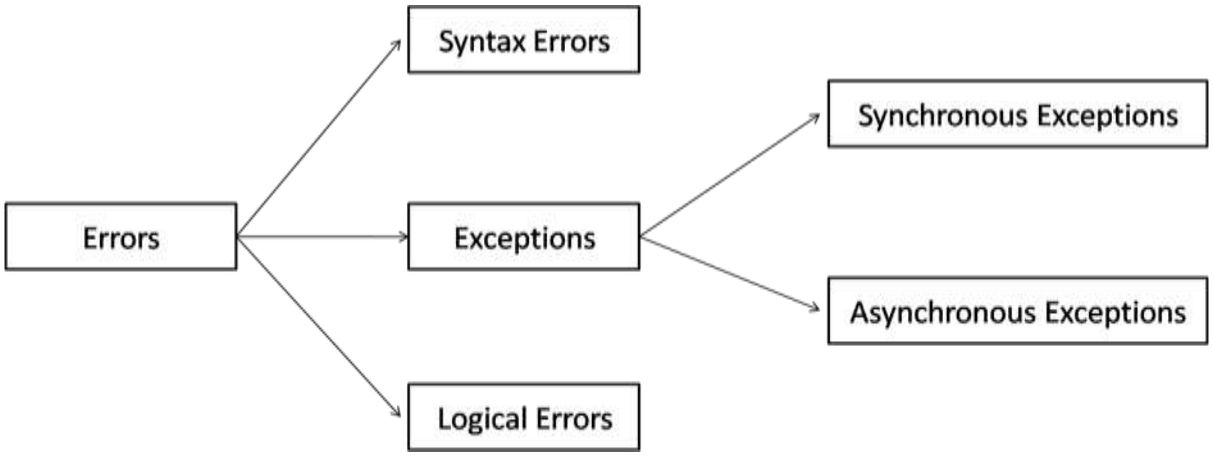
```
Exception thrown at 0x000007FEFDB49E5D in CVpro1.exe: Microsoft C++ exception:
cv::Exception at memory location 0x00000000001EF310.

If there is a handler for this exception, the program may be safely continued.
```

Microsoft Visual Studio
☑ Break when this exception type is thrown
Break and open Exception Settings
Break
Continue
Ignore

What are the types of exceptions?

There are generally two types of exceptions.

1. Synchronous Exceptions (occurs due to software error)
2. Asynchronous Exceptions (occurs due to hardware error)

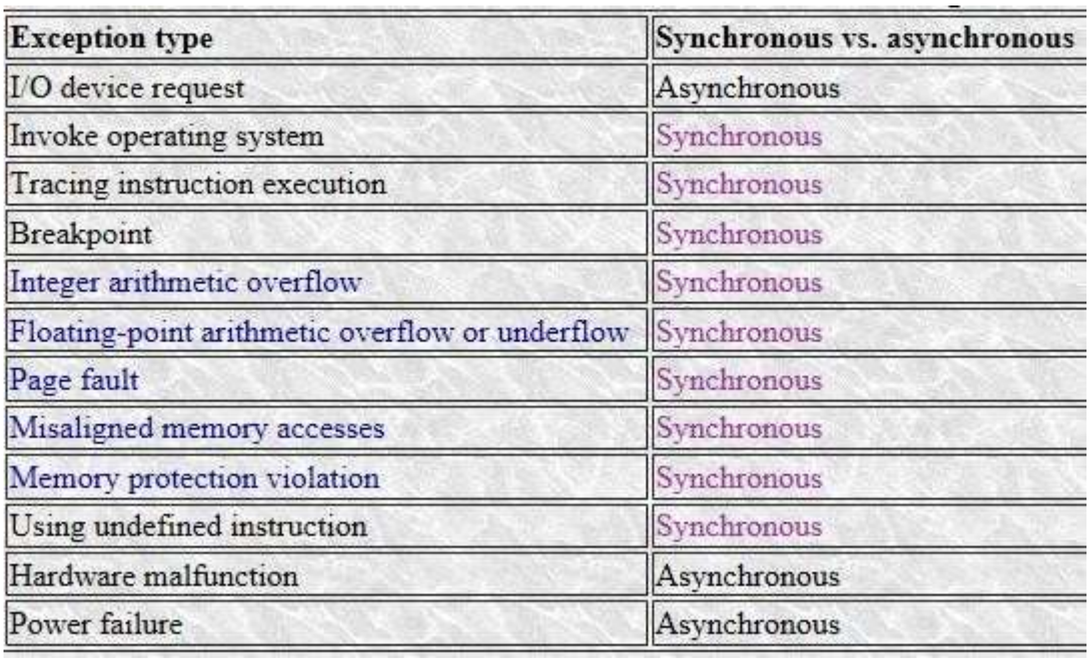


Exception handling is intended to support only synchronous exceptions, such as array range checks. The term synchronous exception means that exceptions can be originated only from throw expressions.

The C++ standard supports synchronous exception handling with a termination model. Termination means that once an exception is thrown, control never returns to the throw point.

Exception handling is not intended to directly handle asynchronous exceptions such as keyboard interrupts. However, you can make exception handling work in the presence of asynchronous events if you are careful. For instance, to make exception handling work with signals, you can write a signal handler that sets a global variable, and create another routine that polls the value of that variable at regular intervals and throws an exception when the value changes. You cannot throw an exception from a signal handler.

With the exception of hardware malfunctions, asynchronous events are caused by devices external to the processor and memory.



Exception type	Synchronous vs. asynchronous
I/O device request	Asynchronous
Invoke operating system	Synchronous
Tracing instruction execution	Synchronous
Breakpoint	Synchronous
Integer arithmetic overflow	Synchronous
Floating-point arithmetic overflow or underflow	Synchronous
Page fault	Synchronous
Misaligned memory accesses	Synchronous
Memory protection violation	Synchronous
Using undefined instruction	Synchronous
Hardware malfunction	Asynchronous
Power failure	Asynchronous

Explain exception handling mechanism.

C++ provides 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.

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) atry block. Then, when the code will lead to any error, that error/exception will get caught inside the catch block.

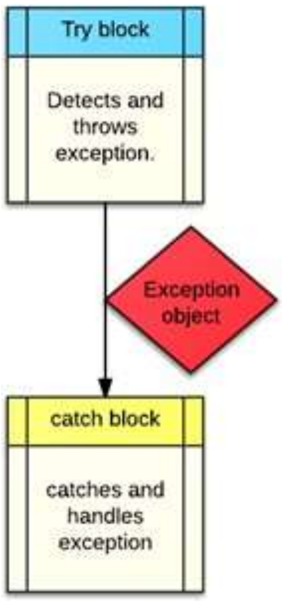
catch block

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

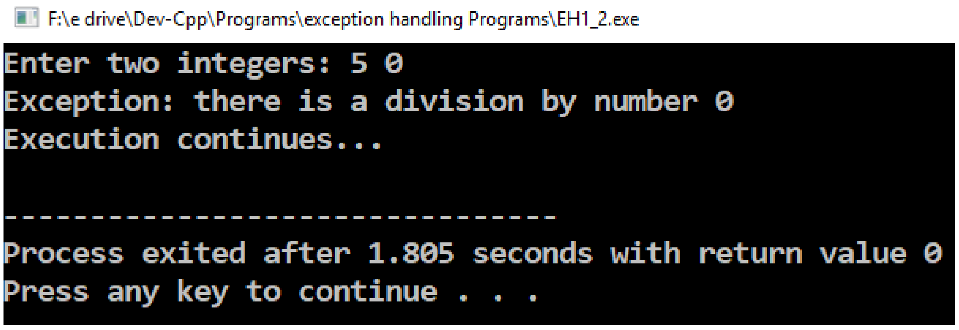
It is used to throw exceptions to exception handler i.e. it is used to communicate information about error. A throw expression accepts one parameter and that parameter is passed to handler.

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



1. Write a program to handle division by zero exception.

```
#include<iostream>
using namespace std;
int main()
{
	cout<<"Enter two integers: ";
	int num1,num2;
	cin>>num1>>num2;

	try
	{
		if(num2==0)
		throw num2;
		cout<<num1<<"/"<<num2<<" is: "<<(num1/num2)<<endl;
	}
	catch(int e)
	{
		cout<<"Exception: there is a division by number "<<e<<endl;
	}
	cout<<"Execution continues..."<<endl;
	return 0;
}
```

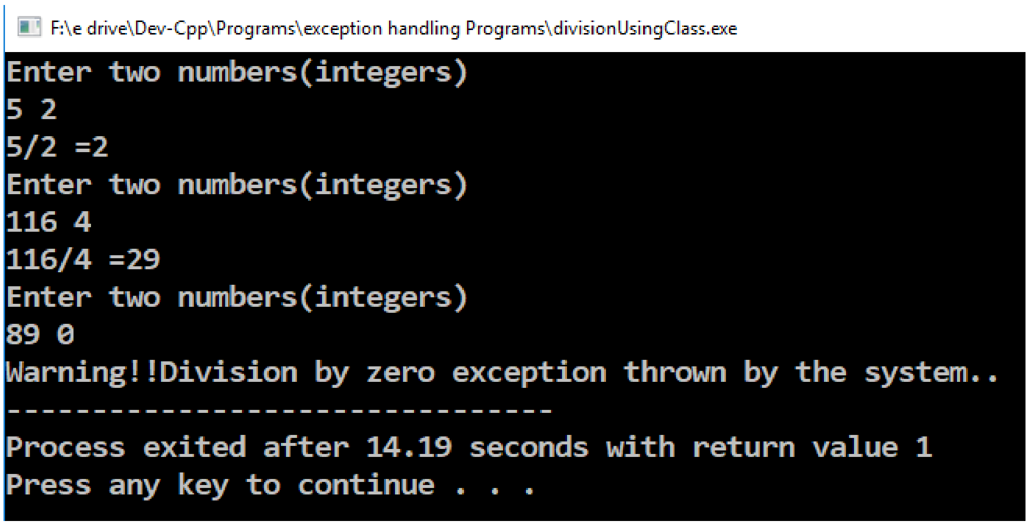


```
F:\e drive\Dev-Cpp\Programs\exception handling Programs\EH1_2.exe
Enter two integers: 5 0
Exception: there is a division by number 0
Execution continues...

--------------------------------
Process exited after 1.805 seconds with return value 0
Press any key to continue . . .
```

2. Write the above program using class.

```
#include<iostream>
using namespace std;
class Division
{
	protected:
		int a,b;
		public:
			Division()
			{
				cout<<"Enter two numbers(integers)"<<endl;
				cin>>a>>b;
```

```
                try
                {       if(b==0)
                        throw  b;
                        calc();

                }
                catch(int exc)
                {
                        cerr<<"Warning!!""Division by zero"" exception thrown
                        by the system..";
                        exit(1);
                }
        }
        void calc()
        {
                cout<<a<<"/"<<b<<" ="<<a/b<<endl;
        }
};
int main()
{
        while(1)
        {
                Division obj1;
        }
        return 0;
}
```



F:\e drive\Dev-Cpp\Programs\exception handling Programs\divisionUsingClass.exe

```
Enter two numbers(integers)
5 2
5/2 =2
Enter two numbers(integers)
116 4
116/4 =29
Enter two numbers(integers)
89 0
Warning!!Division by zero exception thrown by the system..
--------------------------------
Process exited after 14.19 seconds with return value 1
Press any key to continue . . .
```

Re- throwing an exception.

If a catch block cannot handle the particular exception it has caught, you can rethrow the exception. The rethrow expression i.e. throw; (throw without assignment_expression) causes the originally thrown object to be rethrown

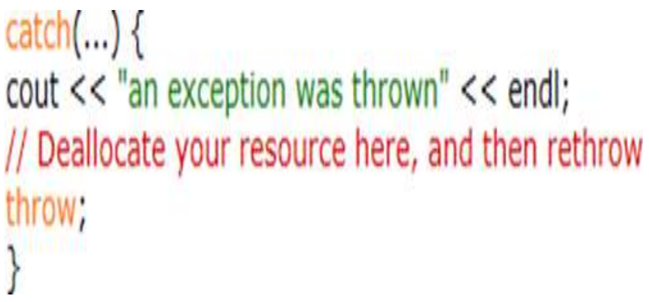
You usually want to rethrow an exception when you have some resource that needs to be released, such as a network connection or heap memory that needs to be deallocated.

• If an exception occurs, you don t necessarily care what error caused the exception you just want to close the connection you opened previously.

• After that, you’ll want to let some other context closer to the user (that is, higher up in the call chain) handle the exception.

• In this case the ellipsis specification is just what you want.
•You want to catch any exception, clean up your resource, and then rethrow the exception for handling elsewhere.

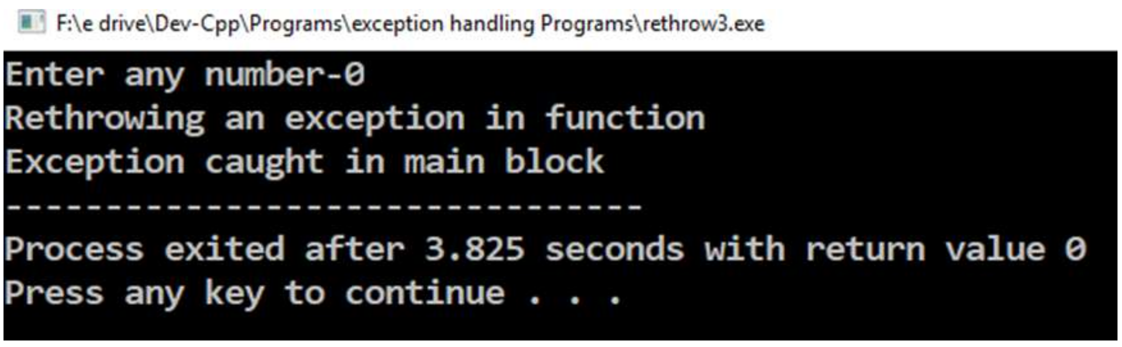
•You rethrow an exception by using throw with no argument inside a handler:



```
catch(...) {
cout << "an exception was thrown" << endl;
// Deallocate your resource here, and then rethrow
throw;
}
```

Example:

```
#include<iostream> using namespace std;
void catch_Exception(int x)
{

try{




}


if(x==0) throw x; else
cout<<"Value stored ="<<x;
catch(int x)
{
cout<<"Rethrowing an exception in function\n"; throw;
}
}
int main()
{
int a;
cout<<"Enter any number-"; cin>>a;
try{
catch_Exception(a);
}
catch(int e)
{
cout<<"Exception caught in main block";
}

return 0;
}
```



F:\e drive\Dev-Cpp\Programs\exception handling Programs\rethrow3.exe

```
Enter any number-0
Rethrowing an exception in function
Exception caught in main block
--------------------------------
Process exited after 3.825 seconds with return value 0
Press any key to continue . . .
```

Exampe 2:

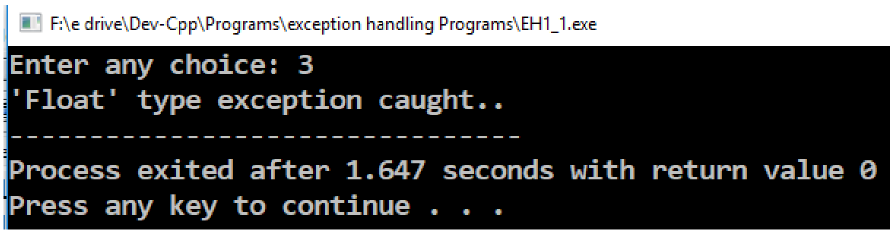
```
#include <iostream>
using namespace std;
// rethrowing exception using nested try ctach.
int main()
{
        try {
                try
                {       throw 20;

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

Write a program to handle different types of exceptions.

For this we can use either multiple catch or a generalised catch i.e. catch-all.
So far, we have only seen one catch block associated with a try block. But it is also possible to have multiple catch blocks associated with one try block.

Multiple catch blocks are used when we have to catch a specific type of exception out of many possible type of exceptions i.e. an exception of type char or int or short or long etc.
The catch block that matches with type of exception thrown is executed, while the rest of catch blocks are skipped.
Let's see the use of multiple catch blocks with an example.

```
#include<iostream>
using namespace std;
int main()
{
   int choice;
    try
   {
      cout<<"Enter any choice:
      "; cin>>choice;

      if(choice == 0)       cout<<"Hello!"<<endl;
      else if(choice == 1)  throw (100);    //throw integer value
      else if(choice == 2)  throw ('x');    //throw character value
      else if(choice == 3)  throw (1.23f); //throw float value else
              cout<<"Bye Bye !!!"<<endl;
   }
   catch(int e)
   {
          cout<<"'Integer' type exception caught..";
          }
          catch(float e)
   {
          cout<<"'Float' type exception caught..";
          }
          catch(char e)
   {
          cout<<"'Character' type exception caught..";
          }

   return 0;
}
```



```
F:\e drive\Dev-Cpp\Programs\exception handling Programs\EH1_1.exe
Enter any choice: 3
'Float' type exception caught..
--------------------------------
Process exited after 1.647 seconds with return value 0
Press any key to continue . . .
```

catch all

Functions can potentially throw exceptions of any data type, and if an exception is not caught, it will propagate to the top of your program and cause it to terminate. Since it’s possible to call functions without knowing how they are even implemented (and thus, what type of exceptions they may throw), how can we possibly prevent this from happening?

Fortunately, C++ provides us with a mechanism to catch all types of exceptions. This is known as a catch-all handler. A catch-all handler works just like a normal catch block, except that instead of using a specific type to catch, it uses the ellipses operator (…) as the type to catch.

```
#include <iostream>

int main()
{
	try
	{
		throw 5; // throw an int exception
	}
	catch (double x)
	{
		std::cout << "We caught an exception of type double: " << x << '\n';
	}
	catch (...) // catch-all handler
	{
		std::cout << "We caught an exception of an undetermined type\n";
	}
}
```

Because there is no specific exception handler for type int, the catch-all handler catches this exception. This example produces the following result:

```
We caught an exception of an undetermined type
```

The catch-all handler should be placed last in the catch block chain. This is to ensure that exceptions can be caught by exception handlers tailored to specific data types if those handlers exist.