南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Advanced Programming

Lab 14, Exceptions, assertion

廖琪梅，王大兴，于仕琪，王薇

topics

- **Exception and Exception Handling**
  - try, throw, catch
  - throw(), noexcept, noexcept(true)
  - exception class, what()
  - catch-by-value vs catch-by-reference

- **Assertion**

Exception and Exception Handling

What is an exception?

An **exception** is a situation, which occurred by the runtime error. In other words, an exception is a runtime error. An exception may result in loss of data or an abnormal execution of program. Exception handling is a mechanism that allows you to take appropriate action to avoid runtime errors.

The default behavior for unexpected is to call **terminate**, and the default behavior for terminate is to call **abort()**, so the program is to halt. Local variables in active stack frames are not destroyed, because **abort()** shuts down program execution without performing such cleanup.

```
class Polygon
{
  protected:
    int width, height;
  public:
    Polygon (int a, int b) : width(a), height(b) {}
    virtual int area () = 0;
    void printarea()
      { cout << this->area() << '\n'; }

};

class Rectangle: public Polygon
{
  public:
    Rectangle(int a,int b) : Polygon(a,b) {}
    int area()
      { return width*height; }

};
```

```
void process(){
    Polygon *p1 =new Rectangle(INT_MAX,INT_MAX);
    p1->printarea();
    delete p1;
}
```

If an exception occurs, the following statements can not be executed. The memory can not be free.

```
void process(){
   Polygon *p =new Rectangle(INT_MAX,INT_MAX);

   try{
        p->printarea();
   }catch(...){
        delete p;
        throw;
   }

   delete p;
}
```

One solution is to use try-catch block to catch and handle the exception. In catch block, free the memory and throw the exception to the caller.

```
void process(){

    shared_ptr<Polygon>sp(newRectangle(INT_MAX,INT_MAX));
    sp->printarea();
}
```

Another solution is to use smart pointer, thus there is no need to free the memory.

Using an object to store a resource that needs to be automatically released(resources should be encapsulated inside objects) and relying on that object's destructor to release it.

Exception handling

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in programs by transferring control to special functions called ***handlers***. C++ provides **three keywords** to support exception handling.

- **try**: The **try** block contain statements which may generate exceptions. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.
- **throw**: When an exception occur in **try** block, it is thrown to the **catch** block using **throw** keyword.
- **catch**: The **catch** block defines the action to be taken when an exception occur. Exception handlers are declared with the keyword **catch**, which must be placed immediately after the **try** block.

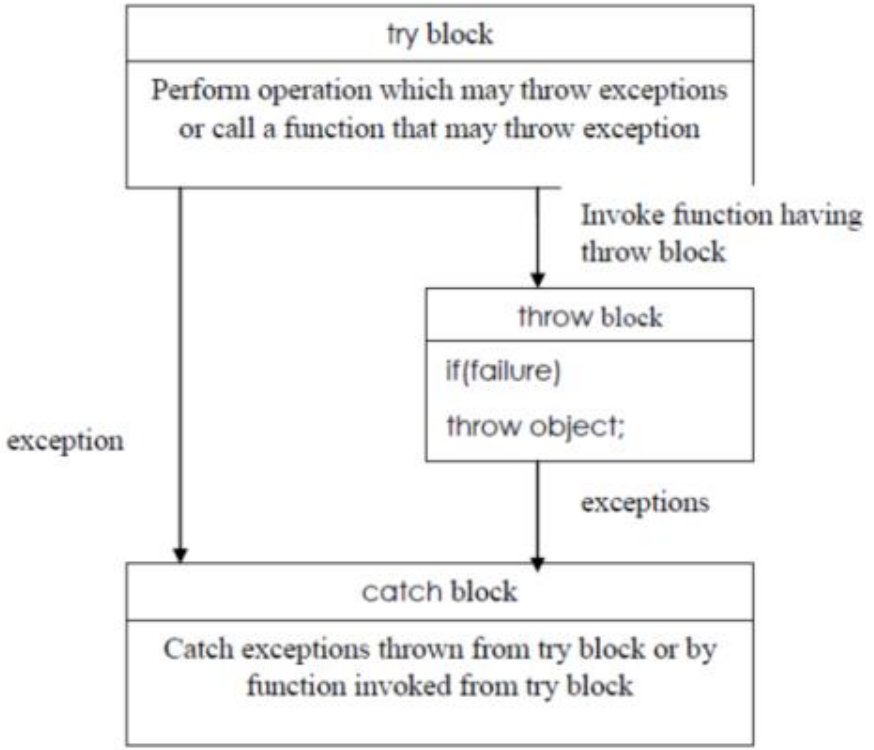


Figure: exception handling mechanism in C++

Example of a program with exception handling using try and catch, throw an exception in **try** block in main()

exceptions > exceptiondemo1.cpp > ...

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

int main()
{
    int a, b;
    double d;
    a = 5;
    b = 0;

    try{
        if(b == 0)
            throw "The divisor can not be zero!";
        d = (double)a/b;
        cout << "The quotient of " << a << "/" << b << " is:" << d << endl;
    }catch(const char* perror){
        cout << perror << endl;
    }catch(int code){
        cout << "Exception code:" << code << endl;
    }

    return 0;
}
```

match

```
The divisor can not be zero!
```

Example of a program with exception handling using try and catch, throw an exception in other function, handlers are in main()

exceptions > exceptiondemo2.cpp > ...

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

double Quotient(int a, int b);

int main()
{
    int a, b;
    double d;
    a = 5;
    b = 0;

    try{
        d = Quotient(a,b);
        cout << "The quotient of " << a << "/" << b << " is:" << d << endl;
    }catch(const char* perror){
        cout << perror << endl;
    }catch(int code){
        cout << "Exception code:" << code << endl;
    }

    return 0;
}

double Quotient(int a, int b)
{
    if(b == 0)
        throw 404;

    return (double)a/b;
}
```

match

```
Exception code:404
```

Note: In general, no conversions are applied when matching exceptions to catch clauses.

exceptions > exceptiondemo2.cpp > ...

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

double Quotient(int a, int b);

int main()
{
    int a, b;
    double d;
    a = 5;
    b = 0;

    try{
        d = Quotient(a,b);
        cout << "The quotient of " << a << "/" << b << " is:" << d << endl;
    }catch(const char* perror){
        cout << perror << endl;
    }catch(double code){
        cout << "Exception code:" << code << endl;
    }

    return 0;
}

double Quotient(int a, int b)
{
    if(b == 0)
        throw 404;

    return (double)a/b;
}
```

does not match

```
terminate called after throwing an instance of 'int'
Aborted
```

Define and using exception class

exceptions > exceptiondemo3.cpp > ...

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

// difine your exception class
class RangeError{
private:
    int iVal;
public:
    RangeError(int _iVal) {iVal = _iVal;}
    int getVal() {return iVal;}
};

char to_char(int n)
{
    if( n < numeric_limits<char>::min() || n > numeric_limits<char>::max())
        throw RangeError(n);

    return (char)n;
}

void gain(int n)
{
    try{
        char c = to_char(n);
        cout << n << " is character " << c << endl;
    }catch(RangeError &re){
        cerr << "Cannot convert " << re.getVal() << " to char\n" << endl;
        cerr << "Range is " << (int)numeric_limits<char>::min();
        cerr << " to " << (int)numeric_limits<char>::max() << endl;
    }
}

int main()
{
    gain(-130);

    return 0;
}
```

Define your exception class

Throw the exception and invoke the constructor

Catch and handle the exception

```
Cannot convert -130 to char

Range is -128 to 127
```

exceptions > exceptiondemo4.cpp > ...

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

class MathException { };
class OverflowException : public MathException{ };
class UnderflowException : public MathException{ };
class ZeroDivideException : public MathException { };

double divide(int numerator, int denominator)
{
    if(denominator == 0)
        throw ZeroDivideException();

    double d = (double) numerator/denominator;
    return d;
}

int main()
{
    try{
        cout << divide(6,0) << endl;
    }catch(ZeroDivideException& zd){
        cerr << "Zero Divide Error" << endl;
    }catch(OverflowException& oe){
        cerr << "Overflow Error" << endl;
    }catch(UnderflowException& ue){
        cerr << "Underflow Error" << endl;
    }catch(MathException& me){
        cerr << "Math Error" << endl;
    }

    return 0;
}
```

Handling exceptions from an inheritance hierarchy

Note: A kind of conversion is applied when matching exceptions to **catch clauses**. That is inheritance-based conversions. A catch clause for base class exceptions is allowed to handle exceptions of derived class types, too.

```
Zero Divide Error
```

exceptions > exceptiondemo4.cpp > ...

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

class MathException { };
class OverflowException : public MathException{ };
class UnderflowException : public MathException{ };
class ZeroDivideException : public MathException { };

double divide(int numerator, int denominator)
{
    if(denominator == 0)
        throw ZeroDivideException();

    double d = (double) numerator/denominator;
    return d;
}

int main()
{
    try{
        cout << divide(6,0) << endl;
    }catch(MathException& me){
        cerr << "Math Error" << endl;
    }catch(ZeroDivideException& zd){
        cerr << "Zero Divide Error" << endl;
    }catch(OverflowException& oe){
        cerr << "Overflow Error" << endl;
    }catch(UnderflowException& ue){
        cerr << "Underflow Error" << endl;
    }

    return 0;
}
```

Note: catch clauses are always tried in the order of their appearance. Hence, it is possible for an exception of a derived class type to be handled by a catch clause for one of its base class types.

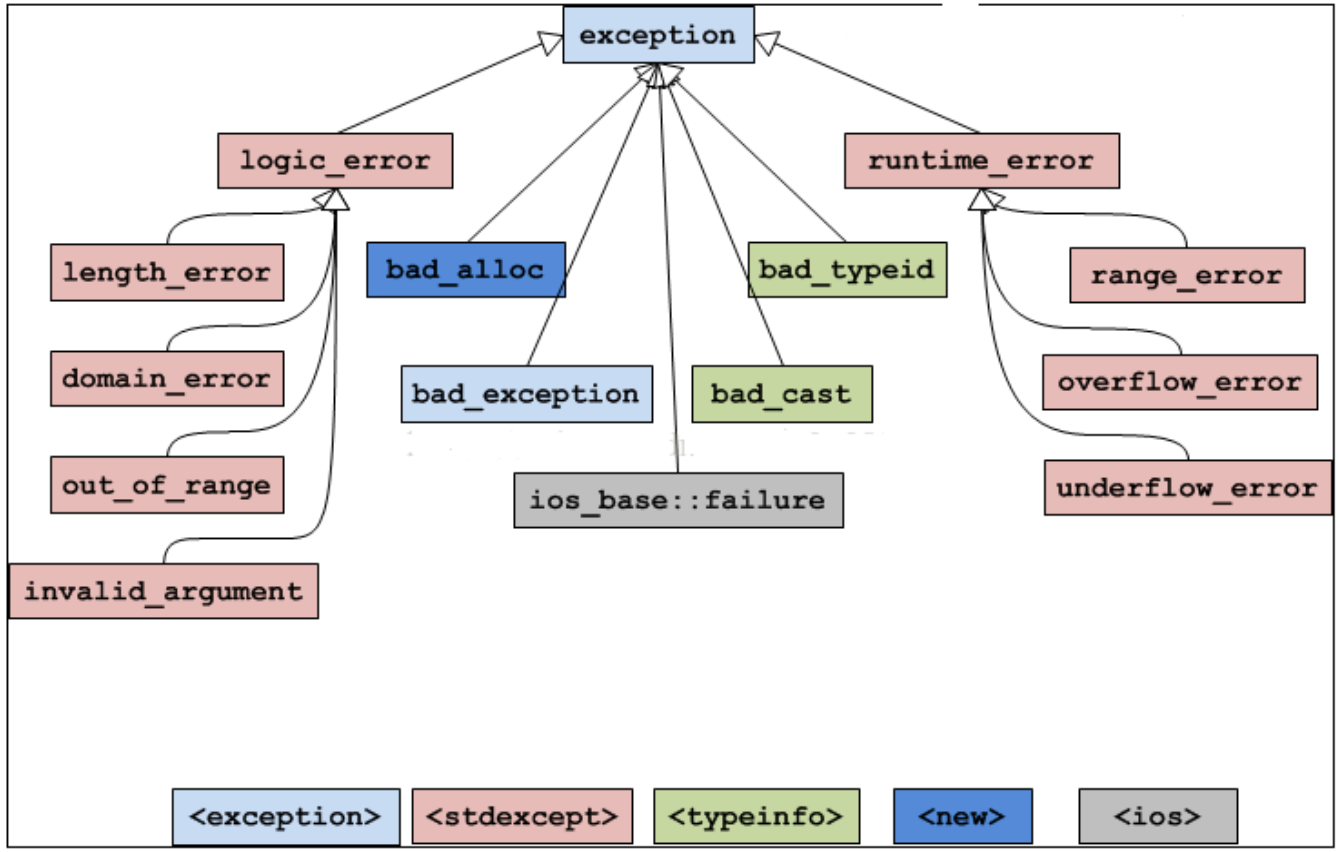
Compilers may warn you if a catch clause for a derived class comes after one for a base class.

```
exceptiondemo4.cpp: In function 'int main()':
exceptiondemo4.cpp:24:6: warning: exception of type 'ZeroDivideException' will be caught
   24 |     }catch(ZeroDivideException& zd){
      |      ^~~~~
exceptiondemo4.cpp:22:6: warning:    by earlier handler for 'MathException'
   22 |     }catch(MathException& me){
      |      ^~~~~
```

```
Math Error
```

C++ Standard Exceptions

C++ provides a list of standard exceptions defined in which we can use in our programs.



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

noexcept, noexcept(true), and **throw()** are equivalent.

Syntax

`noexcept`	(1)	
`noexcept(`*expression*`)`	(2)	
`throw()`	(3)	(deprecated in C++17) (removed in C++20)

Exception specification used in function declaration, with no argument indicates that the function is not allowed to throw any exceptions.

```
class exception{
    public:
        exception () throw();  //constructor
        exception (const exception&) throw();  //copy constructor
        exception& operator= (const exception&) throw();  //assignment operator
        virtual ~exception() throw();  //destructor
        virtual const char* what() const throw();  //virtual function
}
```

what() is a public method provided by **exception class** which returns a string and it has been overridden by all the child exception classes.

Define your own exception class derived from exception class and override **what()** method

exceptions > exceptiondemo5.cpp > ...

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

class MyException : public exception
{
public:
    const char* what() const throw()
    {
        return "C++ Exception.";
    }
};

int main()
{
    try{
        throw MyException();
    }catch(MyException& me){
        cout << "MyException is caught." << endl;
        cout << me.what() << endl;
    }catch(exception& e){
        cout << "Base class exception is cautch." << endl;
        cout << e.what() << endl;
    }
    return 0;

}
```

```
MyException is caught.
C++ Exception.
```

Note: **use catch-by-reference for exception objects**

catch-by-value:Derived class exception objects caught as base class exceptions have their derivedness "sliced off." Such "sliced" objects are base class objects: they lack derived class data members, and when virtual functions are called on them, they resolve to virtual functions of the base class. So use **catch-by-reference** for exception objects and invoke the virtual function of the derived class.

exceptions > exceptiondemo6.cpp > ...

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

class MyException : public exception
{
public:
    virtual const char* what() const noexcept
    {
        return "C++ Exception.";
    }
};

int main()
{
    try{
        throw MyException();
    }catch(exception e){
        cout << "Base class exception is cautch." << endl;
        cerr << e.what() << endl;
    }
    return 0;

}
```

It will not throw any exception

Catch the exception by value

```
Base class exception is cautch.
std::exception
```

Invoke what() of the exception class rather than the MyException class.

exceptions > exceptiondemo6.cpp > ...

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

class MyException : public exception
{
public:
    virtual const char* what() const noexcept override
    {
        return "C++ Exception.";
    }
};

int main()
{
    try{
        throw MyException();
    }catch(exception& e){
        cout << "Base class exception is cautch." << endl;
        cerr << e.what() << endl;
    }
    return 0;

}
```

It is overriding a virtual method of the base class.

Catch the exception by reference

```
Base class exception is cautch.
C++ Exception.
```

Invoke what() of the MyException class not the exception class.

Throw exceptions by value, catch them by reference.

Assertions in C/C++

Assertions are statements used to test assumptions made by programmers. It is designed as a macro in C/C++. Following is the syntax for assertion:

void assert(int expression);

If the expression evaluates to 0 (false), then the expression, source code filename, and line number are sent to the standard error, and then **abort()** function is called.

noexceptions > testassert.cpp > ...

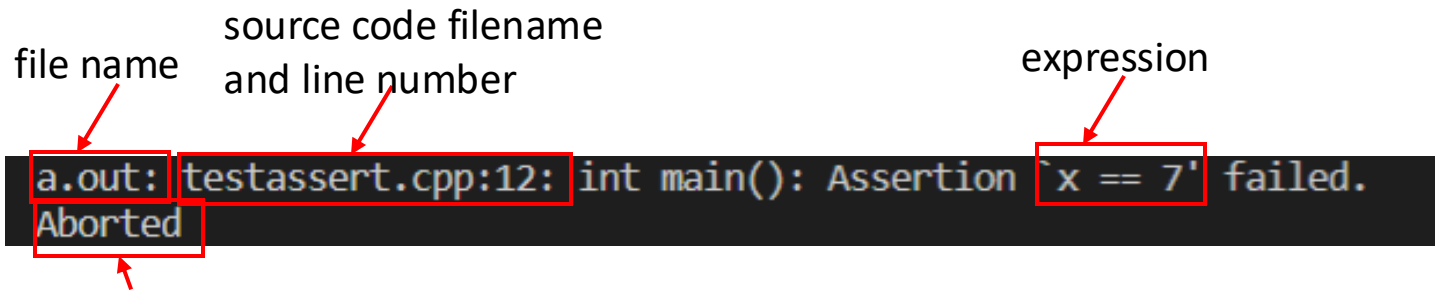
```cpp
#include <assert.h>
#include <iostream>
using namespace std;

int main()
{
    int x = 7;
    //x is accidentally changed to 9  */
    x = 9;

    // Programmer assumes x to be 7 in rest of the code
    assert(x == 7);

    // Rest of the code
    cout << "The original value of x is 7" << endl;

    return 0;
}
```



Assertions are mainly used to check logically impossible situations. For example, they can be used to check the state of a code which is expected before it starts running, or the state after it finishes running.

- Verify the validity of the passed argument at the beginning of the function.

```
int resetBufferSize(int nNewSize)
{
      assert(nNewSize >= 0);
      assert(nNewSize <= MAX_BUFFER_SIZE);
   ...
}
```

```
// is not recommended
assert(nOffset>=0 && nOffset+nSize<=m_nInfomationSize);

// is recommended, each assert test only on condition
assert(nOffset >= 0);
assert(nOffset+nSize <= m_nInfomationSize);
```

- Each assert tests only one condition, because when multiple conditions are tested at the same time, it is not intuitive to determine which condition failed if the assertion failed.

- Ignores assertions. We can completely remove assertions at compile time using the preprocessor NDEBUG. Put **#define NDEBUG** at the beginning of the code, before inclusion of <assert.h>. Therefore, this macro is designed to capture programming errors, not user or run-time errors, since it is generally disabled after a program exits its debugging phase.

Exercise:

1. Write a function **calculateAverage()** which takes four int arguments, which are marks of four courses, and returns their average as a float number.

The **calculateAverage()** function should take only valid range for marks which is between 0-100. If the marks are out of range throw an **OutOfRangeException** – define this exception as a class.

Invoke the **calculateAverage()** function in main function and get the following inputs and outputs:

```
Please enter marks for 4 courses:70 80 90 67
The average of the four courses is 76.75
Would you want to enter another marks for 4 courses(y/n)?y
Please enter marks for 4 courses:120 56 89 99
The parameter 1 is 120 which out of range(0-100).
Would you want to enter another marks for 4 courses(y/n)?y
Please enter marks for 4 courses:90 -87 67 92
The parameter 2 is -87 which out of range(0-100).
Would you want to enter another marks for 4 courses(y/n)?n
Bye, see you next time.
```

2. Define a generic class “GoldenRectangle” for a golden rectangle in the “golden_rectangle. h” , When testing based on the following main methods, the test results are shown in the bottom figure.

GoldenRectangle class:

- ✓ two private properties: length and width, and a generic data type.
- ✓ a public method, which returns a Boolean value.
  - ✓ The method calculates by dividing the long side by the short side. If the quotient is 1.618, it returns true; otherwise, it returns false.
  - ✓ In this method, If the shorter side is 0 (i.e. the divisor is 0), an assertion is triggered.

```cpp
#include "golden_rectangle.h"
int main() {
    GoldenRectangle<int>rect0(1618, 10);
    std::cout <<"1618x10 is golder Rectangle: "<<std::boolalpha <<rect0.isGolden() <<std::endl;

    GoldenRectangle<int>rect1(34, 21);
    std::cout <<"34x21 is golder Rectangle: "<<std::boolalpha <<rect1.isGolden() <<std::endl;

    GoldenRectangle<double>rect2(1.0, 1.618);
    std::cout <<"1.0x1.618 is golder Rectangle: "<<rect2.isGolden() <<std::endl;

    GoldenRectangle<float>rect3(10.0f, 6.0f);
    std::cout <<"10.0x6.0 is golder Rectangle: "<<rect3.isGolden() <<std::endl;

    GoldenRectangle<int>rect4(10, 0);

    std::cout <<"10x0 is golder Rectangle: "<<rect4.isGolden() <<std::endl;
    return 0;
}
```

```
1618x10 is golder Rectangle: false
34x21 is golder Rectangle: false
1.0x1.618 is golder Rectangle: true
10.0x6.0 is golder Rectangle: false
a.out: golden_rectangle.h:19: bool GoldenRectangle<T>::isGolden() const [with T = int]: Assertion `shorter != 0' failed.
Aborted (core dumped)
```

3. 1) define a generic class “**GoldenRectangle**” (similar with which in the excercise2) for a golden rectangle and 2)a class “**DivisionByZeroException**”(inherit from “exception” class) in the “golden_rectangle2. h” , 3) complete the main method, When testing based on the following main methods, the test results are shown in the bottom figure.

DivisionByZeroException class:

- ✓ public inherit from exception class.
- ✓ override “what” function, return a string as “Error: divisor MUST NOT be zero”

GoldenRectangle class:

- ✓ two private properties: length and width, and a generic data type.
- ✓ a public method, which returns a Boolean value.
  - ✓ The method calculates by dividing the long side by the short side. If the quotient is 1.618, it returns true; otherwise, it returns false.
  - ✓ In this method, If the shorter side is 0 (i.e. the divisor is 0), throw an DivisionByZeroException object.

```
#include "golden_rectangle2.h"
int main() {
    try {
        GoldenRectangle<double>rect0(1.0, 1.618);
        std::cout <<"1.0x1.618 is golder Rectangle: "<<rect0.isGolden() <<std::endl;

        GoldenRectangle<int>rect1(34, 21);
        std::cout <<"34x21 is golder Rectangle: " <<rect1.isGolden() <<std::endl;

        GoldenRectangle<int>rect2(10.0, 0.0);
        std::cout <<"10x0 is golder Rectangle: "<<rect2.isGolden() <<std::endl;
    }
    catch(const /*complete code here*/) {
        std::cerr <<e.what() <<std::endl;
    }
    catch(const /*complete code here*/) {
        std::cerr <<"other exception: "<<e.what() <<std::endl;
    }
    return 0;
}
```

```
1.0x1.618 is golder Rectangle: 1
34x21 is golder Rectangle: 0
10x0 is golder Rectangle: Error: divisor MUST NOT be zero
```