C/C++ Program Design

LAB 15

CONTENTS

- Learn to use friend functions and friend classes

- Learn to use exception handling

2 Knowledge Points

2.1 Friend functions

2.2 Friend classes

2.3 Nested class

2.4 Exception and Exception handling

2.1 Friend Functions

A **friend function** of a class is a non-member function that has the right to access the **private** and **protected** class members.

To declare a non-member function as a friend of a class, place the function prototype in the class definition and precede it with the keyword **friend**.

friend **return_type functionName (parameter list);**

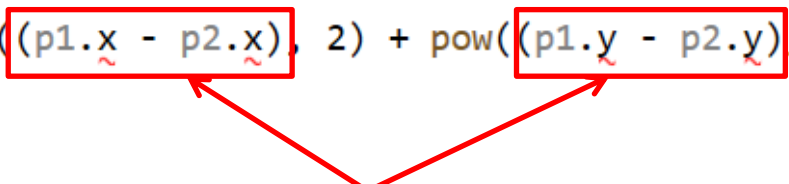```cpp
#include <iostream>
#include <cmath>
using namespace std;

class Point
{
private:
    double x, y;

public:
    Point(double xx = 0, double yy = 0)
    {
        x = xx;
        y = yy;
    }

    void show() const
    {
        cout << x << "," << y << endl;
    }
};

double distance(const Point &p1, const Point &p2)
{
    return sqrt(pow((p1.x - p2.x), 2) + pow((p1.y - p2.y), 2));
}
```

The private data members can not be accessed outside the class

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

class Point
{
private:
    double x, y;

public:
    Point(double xx = 0, double yy = 0)
    {
        x = xx;
        y = yy;
    }

    double getX() const { return x; }

    double getY() const { return y; }

    void show() const
    {
        cout << x << "," << y << endl;
    }
};
double distance(const Point &p1, const Point &p2)
{
    return sqrt(pow((p1.getX() - p2.getX()), 2) + pow((p1.getY() - p2.getY()), 2));
}

int main()
{
    Point p(1, 1);
    Point q(4, 5);
    cout << "The distance of the two points is:" << distance(p, q) << endl;

    return 0;
}
```

Can distance function be defined as the member function?

The distance of the two points is:5

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

class Point
{
private:
    double x, y;

public:
    Point(double xx = 0, double yy = 0)
    {
        x = xx;
        y = yy;
    }

    double getX() const { return x; }
    double getY() const { return y; }

    double distance(const Point &p1, const Point &p2)
    {
        return sqrt(pow((p1.x - p2.x), 2) + pow((p1.y - p2.y), 2));
    }

    void show() const
    {
        cout << x << "," << y << endl;
    }
};
int main()
{
    Point p(1, 1);
    Point q(4, 5);
    cout << "The distance of the two points is:" << distance(p, q) << endl;

    return 0;
}
```

The distance function is defined as the member function.

The member function can access the private members of the class

You cannot invoke the distance function like this, because it is a member function, which is invoked by **object using . operator**.

You can define the distance function as a **friend function**, which can access the private data of **Point** class.

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

class Point
{
private:
    double x, y;

public:
    Point(double xx = 0, double yy = 0)
    {
        x = xx;
        y = yy;
    }

    double getX() const { return x; }
    double getY() const { return y; }

    void show() const
    {
        cout << x << "," << y << endl;
    }

    friend double distance(const Point &p1, const Point &p2);
};

double distance(const Point &p1, const Point &p2)
{
    return sqrt(pow((p1.x - p2.x), 2) + pow((p1.y - p2.y), 2));
}

int main()
{
    Point p(1, 1);
    Point q(4, 5);
    cout << "The distance of the two points is:" << distance(p, q) << endl;

    return 0;
}
```

Declare the friend function inside the class, using **friend** keyword.

The friend function can access the private data of the class.

You can invoke the distance function directly, because it isn't a member function of the class.

The distance of the two points is:5

In the previous chapter we also used other friend functions, such as operator overloading functions.

```cpp
#include <iostream>
#ifndef __MYSTRING__
#define __MYSTRING__

class String
{
private:
    char* m_data;

public:

    String(const char* cstr = 0);

    String(const String& str);

    String& operator=(const String& str);

    ~String();

    char* get_c_str() const { return m_data; }

    friend std::ostream& operator<<(std::ostream& os, const String& str);

};

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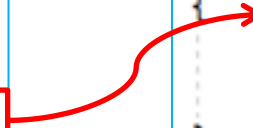
```cpp
#include <iostream>
using namespace std;

ostream& operator<<(ostream& os, const String& str)
{
    os << str.get_c_str();
    return os;
}
```

2.2 Friend Classes

Entire classes or member functions of other classes may be declared to be friends of another class.

To declare all member functions of class **ClassTwo** as friend of **ClassOne**, place a declaration of the form
friend class **ClassTwo;**
in the definition of class **ClassOne**. That means all member functions of class ClassTwo have the right to access the private and protected class members of ClassOne.

The **friend** declaration(s) can appear anywhere in a class and are not affected by access specifiers public or private or protected.

Let's consider an example: we have a **Circle** class in which it has a subobject (center point) of **Point** class-- class containment(composition). Can we access the center's private member in the Point class?

```cpp
class Circle
{
private:
    Point center;
    double radius;

public:
    Circle():center(0,0),radius(1.0) { }
    Circle(Point &p, double r):center(p),radius(r) { }

    Circle& move(Point& p)
    {
        center.x = p.getX();
        center.y = p.getY();

        return *this;
    }

    void show() const
    {
        center.show();
        cout << "radius:" << radius << endl;
    }
};
```

class containment(or class composition)

In move function we want to set the center to the new point p.
But you cannot access the center's private members x and y.

You can access the public members of the center.

This time you can declare the **Circle** class as a **friend class** of the **Point** class.

This declaration is necessary which is called **forward declaration**.

```
class Circle;
```

```
class Point
{
    friend Circle;

private:
    double x;
    double y;

public:
    Point(double xx = 0, double yy = 0)
    {
        x = xx;
        y = yy;
    }

    Point(Point& p)
    {
        x = p.x;
        y = p.y;
    }

    double getX() { return x; }
    double getY() { return y; }

    void show() const
    {
        cout << x << "," << y << endl;
    }
};
```

Declare the **Circle** class as a friend of the **Point** class. That means in Circle class, its member functions can access the private members of the Point class.

```
class Circle
{
private:
    Point center;
    double radius;

public:
    Circle():center(0,0),radius(1.0) { }
    Circle(Point &p, double r):center(p),radius(r) { }

    Circle& move(Point& p)
    {
        center.x = p.x;
        center.y = p.y;

        return *this;
    }

    void show() const
    {
        center.show();
        cout << "radius:" << radius << endl;
    }
};
```

Member function in the Circle class can access the private member of the Point class.

```cpp
int main()
{
    Point p1(1,1),p2(4,5);
    Circle c1;
    Circle c2(p1, 12);

    cout << "Before move:" << endl;
    c1.show();
    c2.show();

    cout << "After move:" << endl;
    c1.move(p1);
    c2.move(p2);
    c1.show();
    c2.show();

    return 0;
}
```
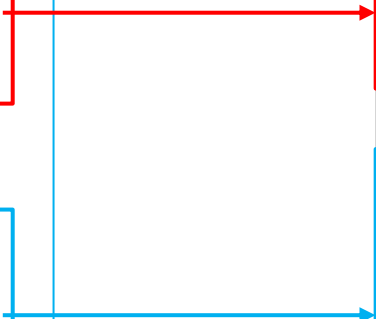
```
Before move:
The center is: 0,0
The radius is:1
The center is: 1,1
The radius is:12
After move:
The center is: 1,1
The radius is:1
The center is: 4,5
The radius is:12
```

Notes:

- Friendship *is granted, not taken*---for class B to be a friend of class A, class A must explicitly declare that class B is its friend.
- Friendship *is not symmetric*– if class A is a friend of class B, you cannot infer that class B is a friend of class A.
- Friendship *is not transitive* ---if class A is a friend of class B and class B is a friend of class C, you cannot infer that class A is a friend of class C.

When to use friend class?
   If one class(or object) is not another class(or object) and vice versa, so the *is-a relationship* of public inheritance doesn't apply. Nor it is either a component of the other, so the *has-a relationship* of containment or of private or protected inheritance doesn't apply. This suggests making the one class **a friend** to the other class.

   Although friends do grant outside access to a class's private portion, they don't really violate the spirit of object-oriented programming. Instead, they provide more flexibility to the public interface.

2.3 Nested Class

The class declared within another is called a nested class, and it helps avoid name clutter by giving the new type class scope.

The usual reasons for nesting a class are to assist in the implementation of another class and to avoid name conflicts.

```cpp
#include<iostream>
using namespace std;
class Outer
{
public:
    class Inner
    {
    public:
        void Fun();
    };
public:
    Inner obj_;
    void Fun()
    {
        cout << "Outer::Fun...." << endl;
        obj_.Fun();
    }
};
void Outer::Inner::Fun()
{
    cout << "Inner::Fun..." << endl;
}
```

Declare a nested class **Inner** inside the **Outer** class

Define an object of the nested class **Inner**

Invoke the function of **Inner** object

Define the function of the nested class **Inner**, using a class qualifier in the outside

If a nested class is declared in a **public section** of a second class, it is available to the second class, to classes derived from the second class, and, because it's public, to the outside world. However, because the nested class has class scope, it has to be used with a **class qualifier** in the outside world.

```cpp
int main()
{
    Outer o;
    o.Fun();
    Outer::Inner i;
    i.Fun();

    return 0;
}
```

Define an object of the Outer class

Define an object of the Inner class using a class qualifier

```
Outer::Fun....
Inner::Fun...
Inner::Fun..
```

```cpp
// template class
template <class Item>
class QueueTP
{
private:
    enum { Q_SIZE = 10 };
    // define a nested class Node
    class Node
    {
    public:
        Item item;
        Node* next;
        Node(const Item& i) :item(i), next(0) {}
    };
    Node* front;     // pointer to front of Queue
    Node* rear;      // pointer to rear of Queue
    int items;       // current number of items in Queue
    const int qsize;    // maximum number of items in Queue
    QueueTP(const QueueTP& q) : qsize(0) {}
    QueueTP& operator=(const QueueTP& q) { return *this; }
public:
    QueueTP(int qs = Q_SIZE);
    ~QueueTP();
```

If a nested class is declared in a **_private section_** of a second class, it is known only to that second class. This applies, for example, to the **Node** class nested in the **Queue** declaration . Hence, **Queue** members can use **Node** objects and pointers to **Node** objects, but other parts of a program don't even know that the **Node** class exists. If you were to derive a class from **Queue**, **Node** would be invisible to that derived class, too, because a derived class can't directly access the private parts of a base class.

If the nested class is declared in a **_protected section_** of a second class, it is visible to that class but invisible to the outside world. However, in this case, a derived class would know about the nested class and could directly create objects of that type.

2.4 Exception and Exception Handling

1. What is 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.

Let's consider a simple example:

a is divided by b, if b equals to zero, what will happen?

Example of a program without exception handling

```cpp
#include <iostream>

using namespace std;

double Quotient(int a, int b);

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

    d = Quotient(a, b);
        cout << "The quotient of " << a << "/" << b << " is:" << d << endl;


    return 0;
}
double Quotient(int a, int b)
{
    return (double)a / b;
}
```

The qeotient of 5/0 is:inf

When divisor is zero, compiler generates a special floating-point value that represents infinity; *cout* displays this value as *Inf, inf or INF*.

Example of a program with if statement to judge whether the divisor is zero

```cpp
#include <iostream>

using namespace std;

double Quotient(int a, int b);

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

    d = Quotient(a, b);
        cout << "The quotient of " << a << "/" << b << " is:" << d << endl;

    return 0;
}
double Quotient(int a, int b)
{
    if (b == 0)
    {
        cout << "Divisor can nto be zero!";
        abort();
    }

    return (double)a / b;
}
```
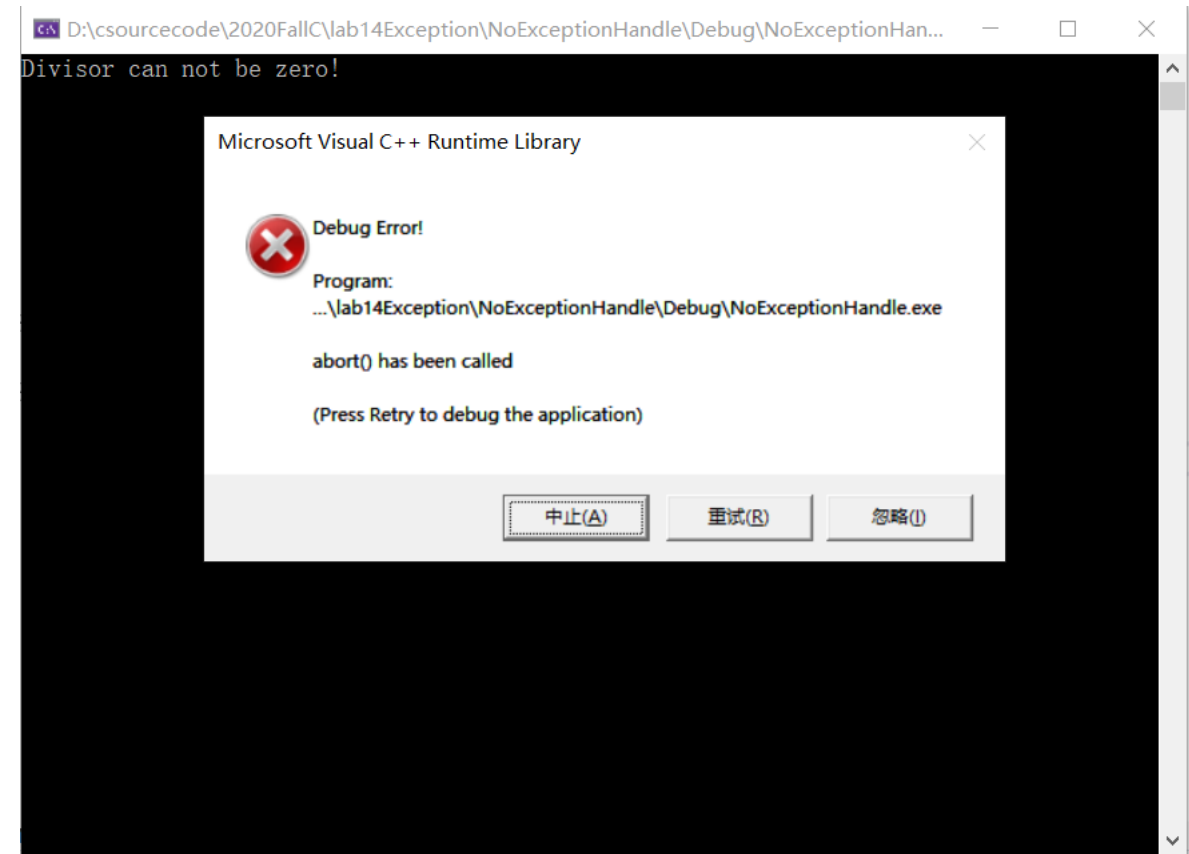
You can use exit() function.

D:\csourcecode\2020FallC\lab14Exception\NoExceptionHandle\Debug\NoExceptionHan... — □ ✕

Divisor can not be zero!

Microsoft Visual C++ Runtime Library ✕

Debug Error!

Program:
...\lab14Exception\NoExceptionHandle\Debug\NoExceptionHandle.exe

abort() has been called

(Press Retry to debug the application)

中止(A)   重试(R)   忽略(I)

Example of a program with the return value to judge the condition

```cpp
#include <iostream>

using namespace std;

bool Quotient(int a, int b, int &c);

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

    if (Quotient(a, b, c))
        cout << "The quotient of " << a << "/" << b << " is:" << c << endl;
    else
        cout << "The divisor can not be zero!" << endl;

    return 0;
}
bool Quotient(int a, int b, int &c)
{
    if (b == 0)
        return false;
    else
    {
        c = a / b;
        return true;
    }
}
```

To judge whether the divisor is zero by return value

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

2. Exception handling

C++ provides **three keywords** to support exception handling

- **try**: The try block contain statements which may generate exceptions.

- **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.

The syntax for using **try/catch** as follows:

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

}
```

Catches any type exception

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.

How does exception handling work

- When a problem is detected during the computation, an exception is raised by using **keyword throw.**

- The raised exceptions are handled by **the catch block**. This exception handler is indicated by the **keyword catch**. **The catch constructor must be used immediately after the try block.**

- **try** block is responsible for testing the existence of exceptions.

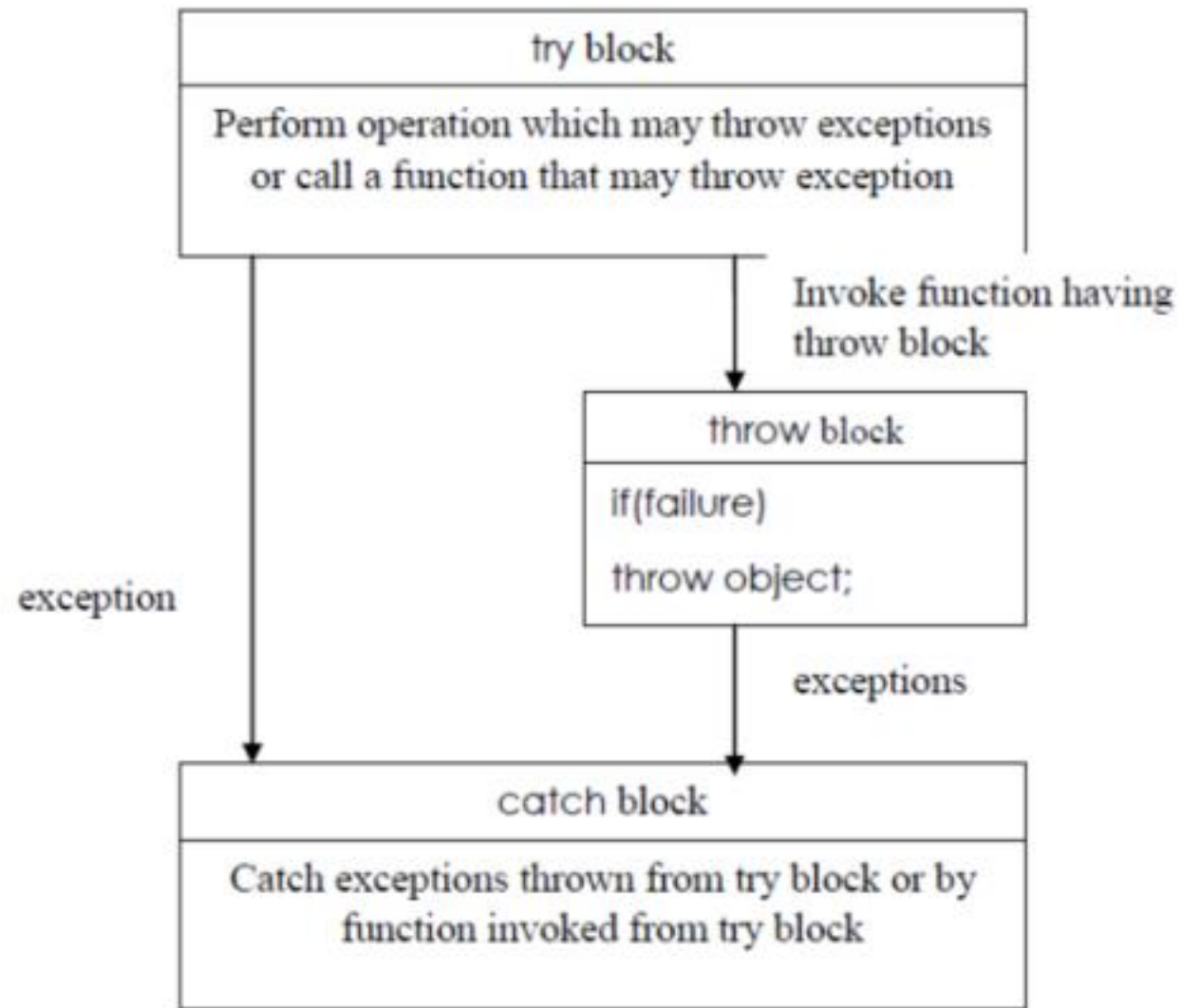The following figure explains more about this:

Steps taken during exception handling:

Figure: exception handling mechanism in C++

1. **Hit the exception**(detect the problem causing exception)

2. **Throw the exception**(inform that an error has occurred)

3. **Catch the exception**(receive the appropriate actions)

4. **Handle the exception**(take appropriate actions)

Example of a program with exception handling using **try** and **catch**

Throw exception in try block

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


int main()
{
    int a, b;
    double d;
    a = 5;
    b = 10;

    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;
    }
    return 0;
}
```

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

Example of a program with exception handling using **try** and **catch**
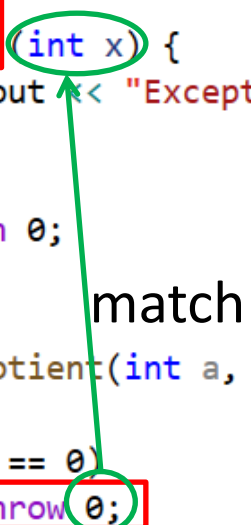
```cpp
#include <iostream>
using namespace std;

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 (int x) {
        cout << "Exception : the divisor can not be zero!" << endl;
    }

    return 0;
}

double Quotient(int a, int b)
{
    if (b == 0)
        throw 0;
    return (double)a / b;
}
```

match

```
Exception: the divisor can not be zero!
```

Example of a program with exception handling using **try** and **catch**

```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 (int code) {
        cout << "Exception : " << code << endl;
    }
    catch (const char* perror) {
        cout << perror << endl;
    }
    return 0;
}

double Quotient(int a, int b)
{
    if (b == 0)
        throw 404;
    return (double)a / b;
}
```

Exception : 404

Example of a program with exception handling using **try** and **catch**

```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 qeotient of " << a << "/" << b << " is:" << d << endl;
    }
    catch (int code) {
        cout << "Exception : " << code << endl;
    }
    catch (const char* perror) {
        cout << perror << endl;
    }
    return 0;
}

double Quotient(int a, int b)
{
    if (b == 0)
        throw "The divisor can not be zero!";
    return (double)a / b;
}
```

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

Define and using exceptions

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

//define your exception class
class RangeError {
public:
    int iVal;
    RangeError(int _iVal) { iVal = _iVal; }
};

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

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

int main()
{
    g(-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
```

Handling exceptions from a inheritance hierarchy

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

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

double divide(int numerator, int denominator)
{
    if(denominator == 0)
        throw ZeroDivideException();
    double d = (double) numerator / denominator;
    return d;
}
int main()
{
    try{
        cout << divide( numerator: 6, denominator: 0) << endl;
    } catch (ZeroDivideException) {
        cerr << "Zero Divide Error" << endl;
    } catch (Matherr) {
        cerr << "Math Error" << endl;
    }

    return 0;
}
```

Output:

Zero Divide Error

exception class and what() method

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

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

int main()
{
    try {
        throw MyException();
    }
    catch (MyException& e) {
        cout << "MyException caught" << endl;
        cout << e.what() << endl;
    }
    catch (exception &e){
        // other errors
    }

    return 0;
}
```
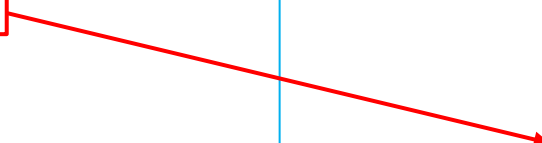
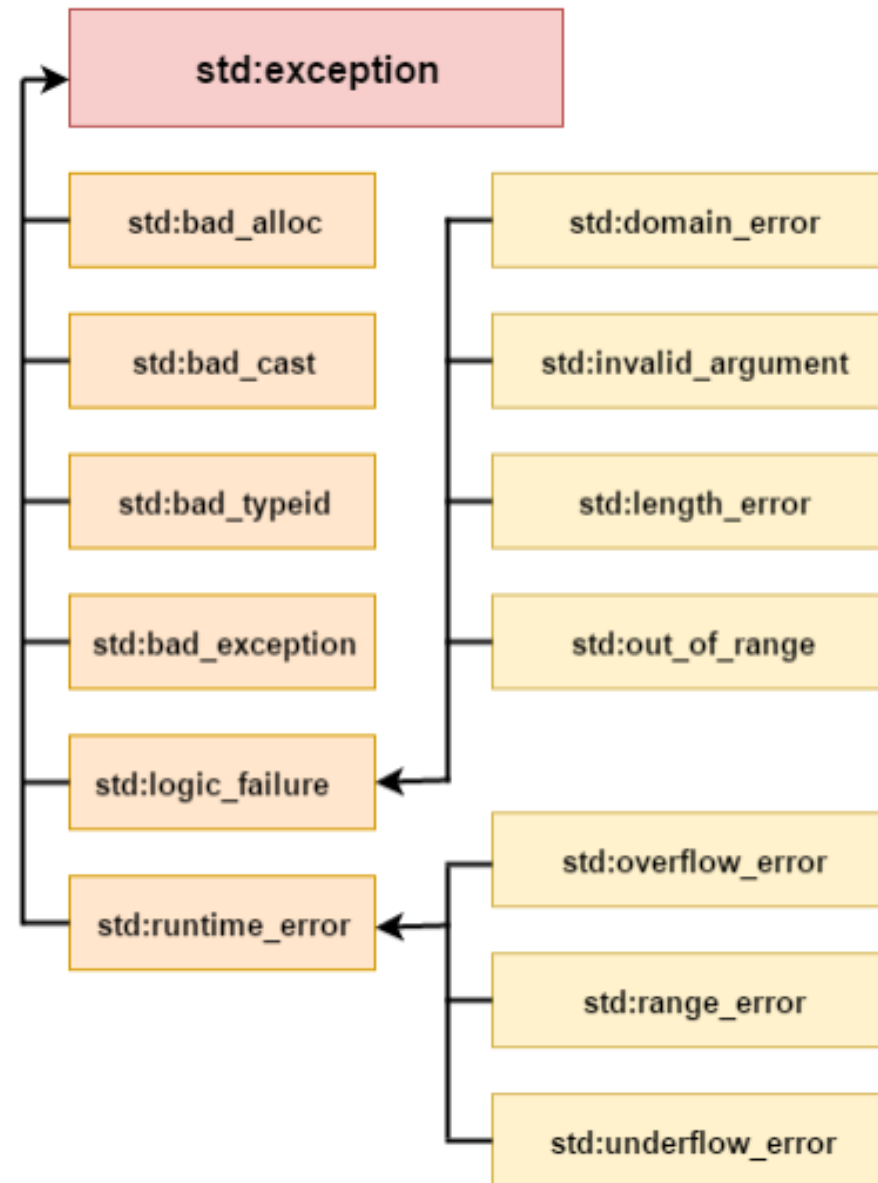You can define your own exceptions by inheritance of **exception class**.

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

```
MyException caught
C++ Exception
```

C++ Standard Exceptions

C++ provides a list of standard exceptions defined in which we can use in our programs. These are arranged in a parent-child class hierarchy shown right.

Here is the small description of each exception mentioned in the above hierarchy:

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.