



Advanced Programming

Lab 16

CONTENTS

- ❑ Learn to use friend classes
- ❑ Learn to use exception handling
- ❑ Know RTTI

2 Knowledge Points

2.1 Friend classes

2.2 Exception and exception handling

2.3 RTTI

2.1 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 is(are) not affected by access specifiers **public** or **private** or **protected**.

If one is not another and vice versa, so the ***is-a relationship*** of public inheritance doesn't apply. Nor 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.

A friend class can access **private** and **protected** members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class.

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?

```
#include <iostream>
#include "point.h"

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
    {
        std::cout << "The center is: ";
        center.show();
        std::cout << "The radius is: " << radius << std::endl;
    }
};
```

class containment(or
class composition)

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

Public member functions such as **getters** and **setters** of the **Point** class can be used in **move function**.

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

```
// point.h
#include <iostream>
#pragma once
class Circle;
class Point
{
    friend class Circle;

private:
    double x;
    double 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
    {
        std::cout << x << "," << y << std::endl;
    }
};
```

This declaration is unnecessary here which is called **forward declaration**.

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

```
// circle.h
#include <iostream>
#include "point.h"

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
    {
        std::cout << "The center is: ";
        center.show();
        std::cout << "The radius is: " << radius << std::endl;
    }
};
```

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

```
#include <iostream>
#include "point.h"
#include "circle.h"
using namespace std;
```

```
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 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.
- Friendship ***is not inherited*** – If a base class has a friend function, then the function doesn't become a friend of the derived class(es).

Friends should be used only for limited purpose. Too many functions or external classes are declared as friends of a class with protected or private data, it lessens the value of encapsulation of separate classes in object-oriented programming.

2.2 Exception and Exception Handling

1. 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, sourcecode filename, and line number are sent to the standard error, and then **abort()** function is called.

```
#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;
}
```

file name sourcecode filename
and line number expression

```
a.out: testAssert.cpp:12: int main(): Assertion 'x == 7' failed.
Aborted
```

abort() function is called and display message on the screen.

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.

2. 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.

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.

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

```
#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 quotient 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. If the divisor is zero, call **abort()** function to terminate the program.

```
#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 not be zero!" << endl;
        abort();
    }
    return (double) a / b;
}
```

Divisor can not be zero!
Aborted

You can use exit() function
without a message displayed.

Example of a program with the return a boolean value to judge the condition, add another argument to store the answer.

```
#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;
}
```

To judge whether the divisor
is zero by return value

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

The divisor can not be zero!

3. 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 contains 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 occurs 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 occurs. Exception handlers are declared with the keyword **catch**, which must be placed immediately after the **try** block.

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

```
try {  
    // protected code  
} catch (ExceptionType1 e1) {  
    // handle e1  
} catch (ExceptionType2 e2) {  
    // handle e2  
} catch (ExceptionType3 e3) {  
    // handle e3  
} catch (...) {  
    // default handle  
}
```



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?

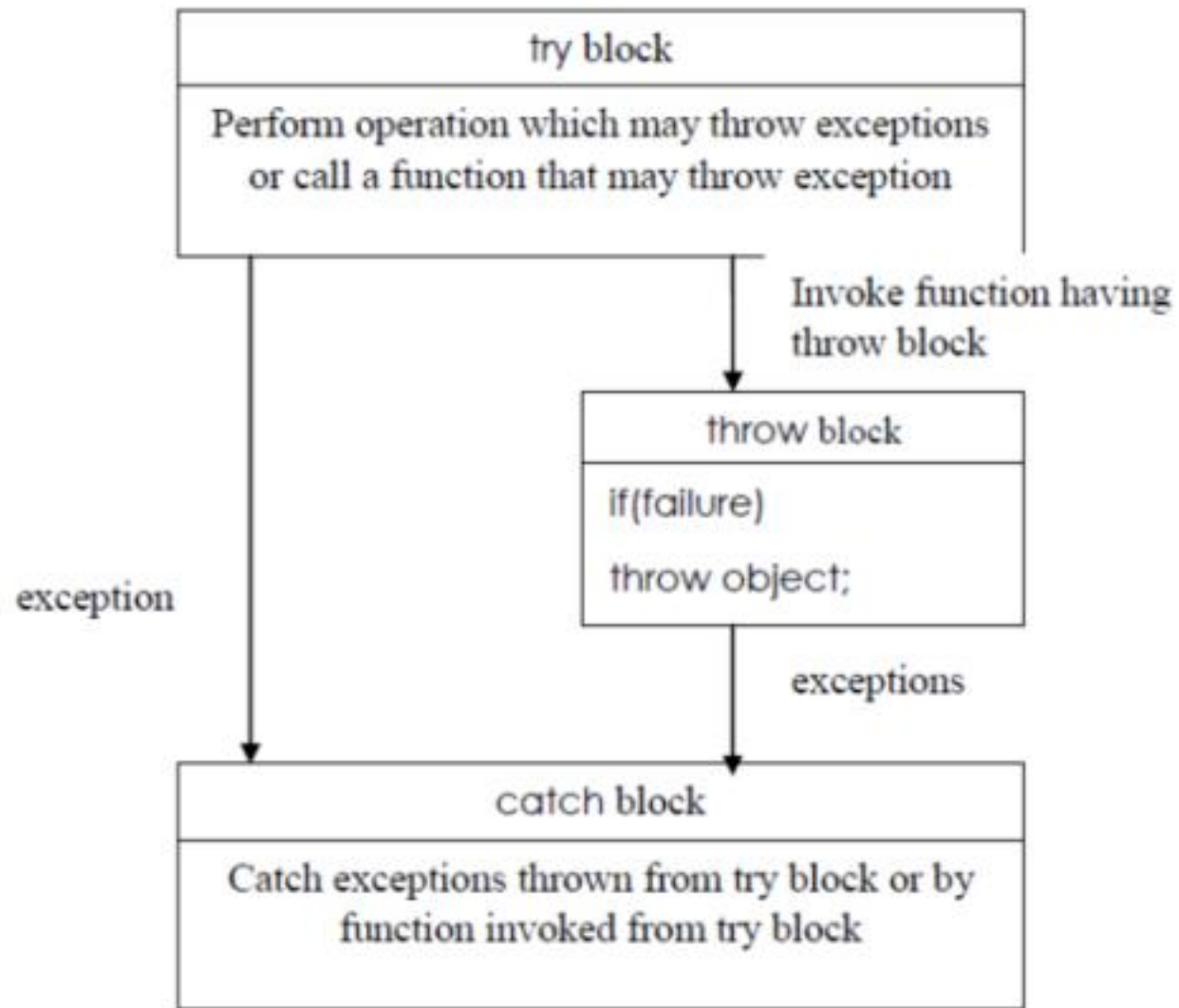


Figure: exception handling mechanism in C++

Steps taken during exception handling:

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)

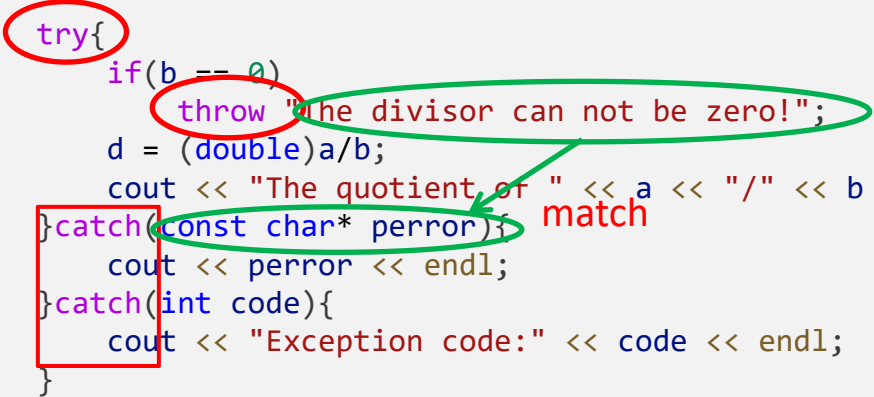
Example of a program with exception handling using **try** and **catch**, throw an exception in **try** block in main()

```
//exceptionDemo1.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;
}
```



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()

```
//exceptionDemo2.cpp
#include <iostream>
using namespace std;

double Quotient(int a, int b);

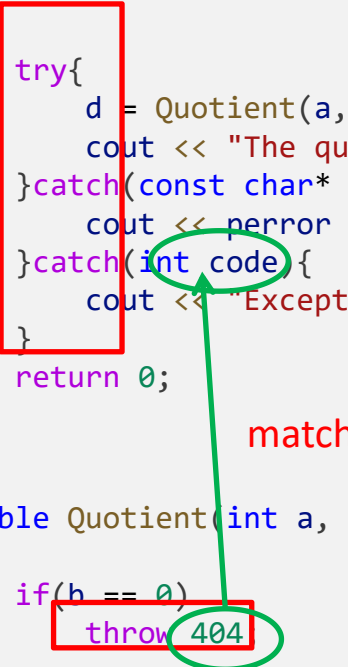
int main()
{
    int a, b;
    double d;
    a = 5;
    b = 0;

    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;
}
```



Exception code:404

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

```
//exceptionDemo2.cpp
#include <iostream>
using namespace std;

double Quotient(int a, int b);

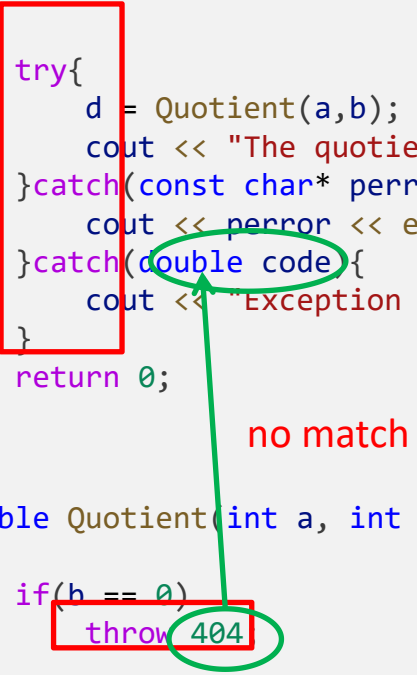
int main()
{
    int a, b;
    double d;
    a = 5;
    b = 0;

    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;
}
```



no match

terminate called after throwing an instance of 'int'
Aborted

Define and using exception class

```
// exceptionDemo3.cpp
#include <iostream>
#include <limits>
using namespace std;
```

```
// define your exception class
```

```
class RangeError
{
private:
    int iVal;

public:
    RangeError(int _iVal) { iVal = _iVal; }
    int getVal() { return iVal; }
};
```

Define your exception class

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

    return (char)n;
}
```

Throw the exception and invoke the constructor

```
void gain(int n)
{
    try
    {
        char c = to_char(n);
        cout << n << " is character " << c << endl;
    }
```

Catch and handle the exception

```
    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;
}
```

Cannot convert -130 to char

Range is -128 to 127

Handling exceptions from a inheritance hierarchy

```
//exceptionDemo4.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;
}
```

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

```

//exceptionDemo4.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.

```

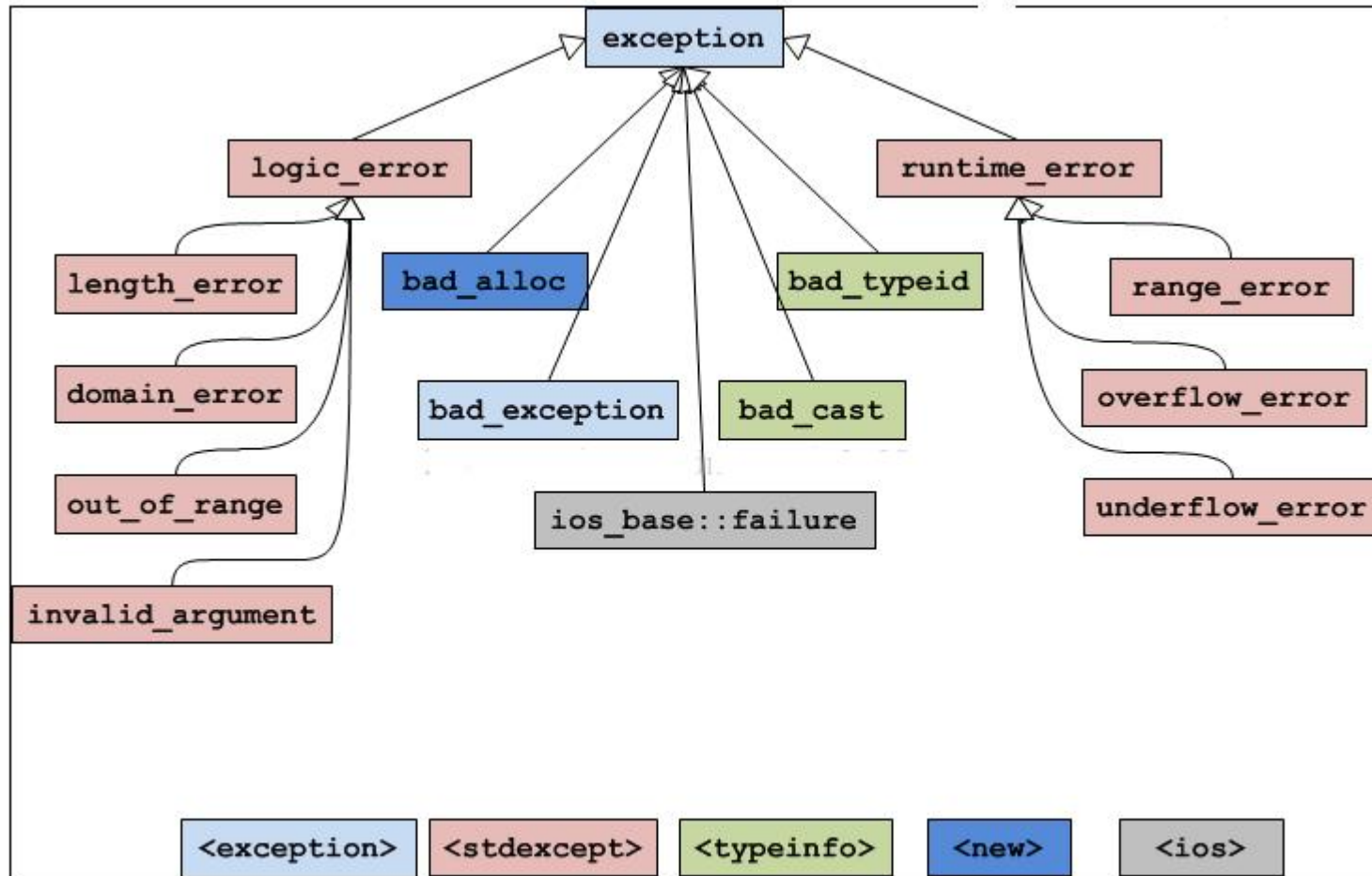
exceptiondemo4.cpp: In function 'int main()':
exceptiondemo4.cpp:25:6: warning: exception of type 'ZeroDivideException' will be caught
25 |     }catch(ZeroDivideException& zd){
   |         ^~~~~~
exceptiondemo4.cpp:23:6: warning:     by earlier handler for 'MathException'
23 |     }catch(MathException& 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 occurred when you try to store a value which is out of range.
std::underflow_error	This is thrown if a mathematical underflow occurs.


```
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
}
```

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

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

```
//exceptionDemo5.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 caught." << 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.

```
// exceptionDemo6.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 caught." << endl;
        cerr << e.what() << endl;
    }
    return 0;
}
```

Catch the exception
by value

Base class exception is caught.
std::exception

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

```
// exceptionDemo6.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 caught." << endl;
        cerr << e.what() << endl;
    }
    return 0;
}
```

Catch the exception
by reference

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

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

2.3 RTTI(Run-Time Type Identification)

RTTI stands for Runtime Type Identification. It is a mechanism to find the type of an object dynamically(in runtime) from an available pointer or reference to the base type.

RTTI is provided through two operators:

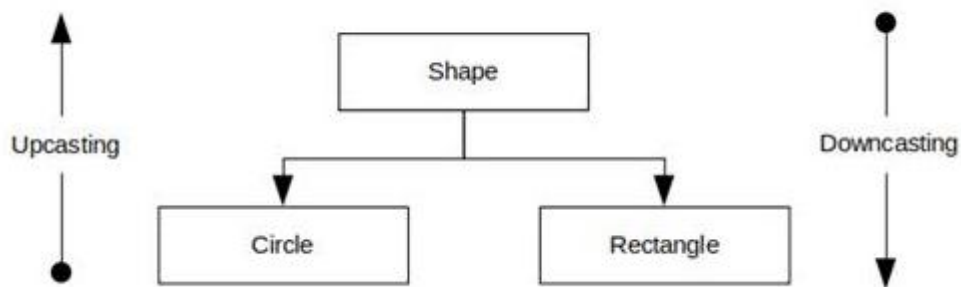
- The **typeid** operator, which returns the type of a given expression
- The **dynamic_cast** operator, which safely converts a pointer or reference to a base type into a pointer or reference to a derived type

A dynamic_cast has the following form:

dynamic_cast < type* > (expression)

dynamic_cast< type& > (expression)

In all cases, the type of **expression** must be either a class type that is publicly derived from the target type, a public base class of the target type, or the same as the target type. If **expression** has one of these types, then the cast will succeed. Otherwise, the cast fails. If a **dynamic_cast** to a pointer type fails, the result is 0. If a **dynamic_cast** to a reference type fails, the operator throws an exception of type **bad_cast**.



- **Upcasting** is the process where we treat a pointer or a reference of a derived class object as a base class pointer. It is automatically accomplished by assigning a derived class pointer or a reference to its base class pointer.

```
Shape *shape_ptr = nullptr;
Rectangle rec(10, 20);
shape_ptr = &rec;    //upcasting, need not explicitly cast
```

- **Down casting** is converting a base class pointer or reference to a derived class. It requires explicit cast.

```
Shape *shape_ptr = nullptr;
Rectangle rec(10, 20);
shape_ptr = &rec;
Rectangle *rec_ptr = nullptr;
rec_ptr = dynamic_cast<Rectangle *> (shape_ptr);
```

Using C-style cast or C++ **static_cast**, neither conversion is safety

```
Rectangle rec(10, 20);
Shape &shape_sr = rec;
try{
    Rectangle &rectangle_rr = dynamic_cast<Rectangle &> (shape_sr);
catch(std::bad_cast &bc){
    cerr << bc.what() << endl;
}
```

When failing to cast a reference, **dynamic_cast** throws **std::bad_cast** exception defined in the **typeinfo** header.

Note: When using **dynamic_cast** operator, there must be at least one virtual function in the parent class, otherwise it fails to compile.

The RTTI operators are useful when we have a derived operation that we want to perform through a pointer or reference to a base-class object and it is not possible to make that operation a virtual function.

```
#include <iostream>

#ifndef _CAST_H_
#define _CAST_H_

class Base
{
public:
    Base(){}
    virtual ~Base(){}

    void show()
    {std::cout << "Base funtion" << std::endl;}
};

class Inherit :public Base
{
public:
    Inherit(){}
    ~Inherit(){}

    void show()
    { std::cout << "Inherit funtion" << std::endl;}
};

#endif
```

```
#include <iostream>
#include "casttype.h"

int main()
{
    Base* pBase = new Inherit();
    pBase->show();
    delete pBase;
    return 0;
}
```

Invoke the show() of base class, though pBase points to the derived object.

```
#include <iostream>
#include "casttype.h"

int main()
{
    Base* pBase = new Inherit();
    if(Inherit* pInherit = dynamic_cast<Inherit*>(pBase))
    {
        pInherit->show();
    }
    delete pBase;
    return 0;
}
```

Invoke the show() of derived class, because pBase is converted to the derived pointer.

Note: We should use virtual functions if we can. When the operation is virtual, the compiler automatically selects the right function according to the dynamic type of the object.

typeid operator

typeid operator can tell you what type is the object.

typeid(expression)

The operand can be any expression or type name.

The *typeid* operator returns a reference to a **type_info** object, where type_info is a class defined in the typeid header file. The type_info class overloads the == and != operators so that you can use these operators to compare types.

If the type is polymorphic, the *typeid* operator provides the information about the most derived type that applies to the dynamic type; otherwise, it provides static type information.


```
Derived *dp = new Derived;
```

```
Base *bp = dp;
```

```
// compare the type of two objects at run time
```

```
if (typeid(*bp) == typeid(*dp))
```

```
{ ...
```

```
}
```

the operands of the typeid are objects, so use *dp not dp

```
// test whether the run-time type is a specific type
```

```
if (typeid(*bp) == typeid(Derived))
```

```
{ ...
```

```
}
```

type_info class includes a ***name()*** member that returns a string that is typically the name of the class.

```
//typeid.cpp

#include <iostream>
#include "casttype.h"

int main()
{
    Base* pBase = new Inherit();
    Inherit* pInherit = dynamic_cast<Inherit*>(pBase);

    std::cout << "The type of pBase pointed to is " << typeid(*pBase).name() << std::endl;
    std::cout << "The type of pBase is " << typeid(pBase).name() << std::endl;

    std::cout << "The type of pInherit pointed to is " << typeid(*pInherit).name() << std::endl;
    std::cout << "The type of pInherit is " << typeid(pInherit).name() << std::endl;

    delete pBase;
    return 0;
}
```

The type of pBase pointed to is 7Inherit
The type of pBase is P4Base
The type of pInherit pointed to is 7Inherit
The type of pInherit is P7Inherit

3 Exercise

Write a function **calculateAverage()** which takes four **int** arguments which are marks for four courses in the semester and returns their average as a float.

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.
```