CS 1502H 程序设计思想与方法 (C++)
Chapter 13&15: 泛型机制-模版&异常处理

陈东尧
上海交通大学
2025年秋季

回顾：保护派生

继承类型中指明的访问特性

基类中的访问特性	**Public**	**Protected**	**Private**
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (隐藏)	Not accessible (隐藏)	Not accessible (隐藏)

```
class A
{
    public:
    int x;
    protected:
    int y;
    private:
    int z;
};
class B : public A
{
// x is public
// y is protected
// z is not accessible from B
};
class C : protected A
{
// x is protected
// y is protected
// z is not accessible from C
};
class D : private A
{
// x is private
// y is private
// z is not accessible from D
};
```

回顾：抽象类实例

- 对所有的汽车都要求有一个显示车辆信息的功能
- 如何保证每个类都有这个功能?
- 可以定义一个抽象类，包含一个显示车辆信息的纯虚函数，各类汽车类都从他派生

```
class  vehicle {
public:
    virtual void display() const = 0;
};
```

```
class  car : public vehicle {
public:
    ……
    void  display() const  { ……};
};
```

```
class  bus : public vehicle {
public:
    ……
    void  display() const  { ……};
};
```

回顾：抽象类的意义

保证进入继承层次的每个类都具有纯虚函数所要求的行为

•保证了围绕这个继承层次所建立起来的软件系统能正常运行，避免了这个继承层次的用户由于偶尔的失误（忘了为它所建立的派生类提供继承层次所要求的行为）而影响系统正常运行

作为基类，实现运行时的多态性

```
class shape{
  public:
     virtual void area()=0;
};
```

回顾：抽象类实例

- **下面程序用于计算各类形状的面积**

```
class shape{  public: virtual void area()=0;  };
class rectangle : public shape{  float w, h;
   public :  rectangle(float ww, float hh)  { w = ww;  h = hh; }
            void area( ) { cout << "\narea is: " << w*h; }  };
class circle : public shape{ float  r;
   public:  circle(float rr) { r = rr; }
            void area( ){ cout << "\narea is: " << 3.14*r*r;}  };
int main( )
{
   shape *ptr;  rectangle ob1(5,10);  circle ob2(10);
    ptr=&ob1; ptr->area();
    ptr=&ob2; ptr->area();
    return  0;
}
```

回顾：DoubleArray

```
class doubleArray
{
     int low;
     int high;
     double *storage;
public:
    doubleArray(int lh = 0, int rh = 0): low(lh), high(rh)
    {   storage = new double [high - low + 1]; }
    doubleArray(const doubleArray &arr);
    doubleArray &operator=(const Array & a);
    double & operator[](int index);
    ~doubleArray() {delete [] storage; }
};
```

第13章 泛型机制-模版

类模板 （Class Template）

- 类模板允许用户为类定义一种模式，使得类中的某些数据成员、某些成员函数的参数或返回值**能取任意数据类型**。
- 定义格式：

  template <class 标识符>

  class 类名{//…};

类模板 （Class Template）

- 类模板允许用户为类定义一种模式，使得类中的某些数据成员、某些成员函数的参数或返回值**能取任意数据类型**。

- 定义格式：

  template <class 标识符>

  class 类名{//…};

回顾：函数模版

```
template<模板形式参数表>
返回类型  函数名(形式参数表)
{
        //函数定义体
}
```

类模版实例

定义一个泛型的、可指定下标范围的、安全的数组

```
class doubleArray
{
     int low;
     int high;
     double *storage;
public:
    doubleArray(int lh = 0, int rh = 0): low(lh), high(rh)
    {   storage = new double [high - low + 1]; }
    doubleArray(const doubleArray &arr);
    doubleArray &operator=(const Array & a);
    double & operator[](int index);
    ~doubleArray() {delete [] storage; }
};
```

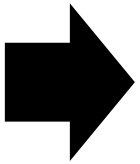
我们目前学到的

- 友元、组合继承 ➔ 赋予了**权限的灵活性**
- 运算符重载 ➔ 提升了**运算符使用的便捷性**
- 成员变量的类型能否灵活变化?

类模版实例

定义一个泛型的、可指定下标范围的、安全的数组

```
class doubleArray
{
     int low;
     int high;
     double *storage;
public:
    doubleArray(int lh = 0, int rh = 0): low(lh), high(rh)
    {   storage = new double [high - low + 1]; }
    doubleArray(const doubleArray &arr);
    doubleArray &operator=(const Array & a);
    double & operator[](int index);
    ~doubleArray() {delete [] storage; }
};
```



```
template <class T>
class Array
{
     int low;
     int high;
     T *storage;
public:
    Array(int lh = 0, int rh = 0): low(lh), high(rh)
   {   storage = new T [high - low + 1]; }
    Array(const Array &arr);
    Array &operator=(const Array & a);
    T & operator[](int index);
    ~Array() {delete [] storage; }
};
```

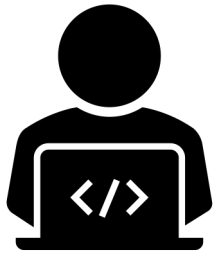
类模板的成员函数的定义

- 类模板的成员函数都是函数模板，模板参数与类模板相同

形式

```
template<模板形参>
返回类型  类模板名<形式参数>::成员函数名（函数的形参表)
{
    函数体
}
```

实例



代码演示

template_class_example.cpp

```cpp
template <typename T>
class Array {
    private:
    T *ptr;
    int size;
    public:
    Array(T arr[], int s);
    void print();
};

template <typename T>
Array<T>::Array(T arr[], int s) {
    ptr = new T[s];
    size = s;
    for(int i = 0; i < size; i++)
    ptr[i] = arr[i];
}

template <typename T>
void Array<T>::print() {
    for (int i = 0; i < size; i++)
    cout<<" "<<*(ptr + i);
    cout<<endl;
}

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    Array<int> a(arr, 5);
    a.print();
    return 0;
}
```

类模版成员函数举例

- Array类型的成员函数实现，构造函数

```
template <class T>
Array<T>::Array(const Array<T> &arr)
{
    low = arr.low;
    high = arr.high;
    storage = new T [high - low + 1];
    for (int i = 0; i < high -low + 1; ++i)
            storage[i] = arr.storage[i];
}
```

类模版成员函数举例

- Array类型的成员函数实现，运算符重载

```
template <class T>
Array<T> &Array<T>::operator=(const Array<T> & a)
{
   if (this == &a) return *this;
   delete [] storage;
   low = a.low;
   high = a.high;
   storage = new T [high - low + 1];
   for (int i=0; i <= high - low; ++i)
       storage[i] = a.storage[i];

   return *this;
}
```

```
template <class T>
T & Array<T>::operator[](int index)
{
    if (index < low || index > high) {
        cout << "下标越界";
       exit(-1);
     }
    return storage[index - low];
}
```

注： exit和return的区别

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
using namespace std;
class Test {
public:
        Test() { printf("Inside Test's Constructor\n"); }

        ~Test()
        {
                printf("Inside Test's Destructor");
                getchar();
        }
};
int main() {
        Test t1;

        // using exit(0) to exit from main
        exit(0);
}
```

调用exit（）会立即终止程序。调用return则会保证完整地运行变量的析构。

例如，运行exit（）仅回收静态变量，其他局部变量不回收

```
Inside Test's Constructor
```

类模板举例：顺序存储

```
template <class T>
class queue   //抽象类
{
   public:
      virtual int find(T x)=0; //确保不同方式实现的队列都有查询功能
};
```

类模板举例：顺序存储

```
//顺序存储
template <class T>
class seqQue: public queue<T>
{
    private:   T a[10]; //顺序存储
    public:
        seqQue(T b[], int n);
        int find(T x);
};
```

类模板的实例化

- 编译器从模板**生成一个特定**的类或函数的过程称为模板的实例化
- 类模板实例化后形成一个模板类
- 类模板的实例化格式如下：

  类模板名<模板的实际参数> 对象名;

- 如：

```
int a[30];
double b[20];
seqQue <int> q1(a，30); //将a中的元素排序并存入q1中
```

类模板的实例化

- 编译器从模板**生成一个特定**的类或函数的过程称为模板的实例化
- 类模板实例化后形成一个模板类
- 类模板的实例化格式如下：

  类模板名<模板的实际参数> 对象名;

- 如：

```
int a[30];
double b[20];
seqQue <int> q1(a，30);
```

我们可以用下列语句输出q1中查找23的结果：

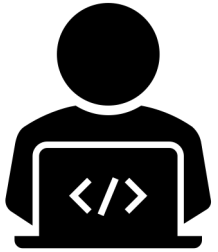
```
cout<<q1.find(23);
```

类模板应用举例

- 一个实现**任意类型数据**存取的类模板Store

例如我们的int、double、包括自己创建的struct Student，等等

```
struct Student
{
    int id;
    float score;
};
```

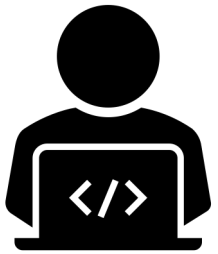


代码演示

store_example.cpp

任意类型数据存取的类模板Store

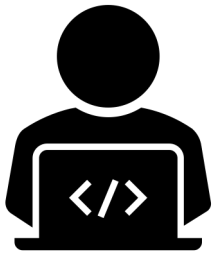
```
template <class T>
class Store {
    T item;
    bool haveValue;  //标记item是否存入内容
public:
    Store():haveValue(false){}
    T &getElem(){
        if(!haveValue) {
            cout<<"No value"<<endl;
            exit(-1);
        }
        return item;
    }
    void putElem(const T &x) {
        haveValue=true;
        item=x;
    }
};
```



代码演示

store_example.cpp

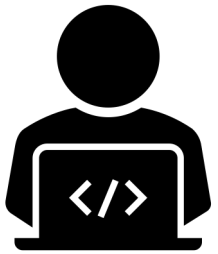
```
int main(){
        Store<int> s1,s2;
        s1.putElem(3);
        s2.putElem(-7);
        cout<<s1.getElem()<< “  “<<s2.getElem()<<endl;
        Student g={1000,90};
        Store<Student> s3;
        s3.putElem(g);
        cout<<s3.getElem().id<<endl;
        store<double>d;
        cout<<d.getElem()<<endl;
        return 0;
}
```



代码演示

store_example.cpp

```
int main(){
        Store<int> s1,s2;
        s1.putElem(3);
        s2.putElem(-7);
        cout<<s1.getElem()<< “  “<<s2.getElem()<<endl;
        Student g={1000,90};
        Store<Student> s3;
        s3.putElem(g);
        cout<<s3.getElem().id<<endl;   //访问s3元素的成员
        Store<double>d;
        cout<<d.getElem()<<endl; //d中没有元素，程序将会中止
        return 0;
}
```



代码演示

store_example.cpp

```
3 -7
1000
No value
```

模板的编译

- 当编译器看到模板的定义时，它不立即产生代码。只有看到模板**被使用**时，如定义类模板的对象时，编译器才产生特定类型的模板实例
- 类模板的成员函数本身是模板函数。类模板的成员函数只有在为程序所用时才进行实例化。如果该成员函数从未被使用，则不会实例化该成员函数

第15章 异常处理 （Error Handling）

本章框架

- 什么是异常
- 异常处理机制
- 异常规格说明

不具备异常处理能力的程序

```
#include <iostream>
#include <cmath>
using namesace std;

int main()
{
   double a, b, c, x1, x2, dlt;

    cout << "请输入方程的3个系数：" << endl;
   cin >> a >> b >> c;

   dlt = b * b - 4 * a * c;
   x1 = (-b + sqrt(dlt)) / 2 / a;
   x2 = (-b - sqrt(dl)) / 2 / a;

   cout << "x1=" << x1 << "   x2=" << x2 << endl;

   return 0;
}
```

什么是异常

- 程序执行过程中发生的一些不希望发生的事情

- 传统错误处理方法

什么是异常

- **程序执行过程中发生的一些不希望发生的事情**

- **传统错误处理方法**

  在发生错误的地方就地处理

  好处

  程序员阅读代码时能够直接看到错误处理情况。

  问题

  增加代码量，使应用程序本身的代码更加晦涩难懂，难以看出代码功能是否正确实现。

传统的异常处理

```
int main()
{
    double a, b, c, x1, x2, dlt;

    cout << "请输入3个参数：" << endl;
    cin >> a >> b >> c;

    if (a == 0)  cout << "不是一元二次方程" << endl;
    else {
      dlt = b * b - 4 * a * c;
      if (dlt >= 0) {
          x1 = (-b + sqrt(dlt)) / 2 / a;
          x2 = (-b - sqrt(dlt)) / 2 / a;
          cout << "x1=" << x1 << "   x2=" << x2 << endl;
       }
       else cout << "无根" << endl;
    }

    return 0;
}
```

解决问题的实现过程并不清晰

异常处理针对三种错误类型

- 语法错误
- 逻辑错误
- 运行错误

本章框架

- 什么是异常
- 异常处理机制
- 异常规格说明

面向对象中的异常处理

问题一

代码中穿插了异常处理代码，复杂化了解决问题的过程

把异常集中在一起处理

问题二

如何将检测到的错误告诉使用者

C++异常处理特性

将检测发现错误的代码与处理错误的代码分开

- 程序员的工作也可做相应分工
- 设计工具的程序员负责检测异常
- 使用工具的程序员则负责捕获与处理异常

将解决问题的代码与处理异常的代码分开来

- 将异常集中在一起处理，异常处理代码不再干扰主逻辑

异常处理机制

组成部分

- 异常抛出（throw）
- 异常捕获（try --- catch）
- 异常处理

异常抛出（throw）

作用

- 当程序发生异常情况，而在当前的环境中获取不到异常处理的足够信息，可以创建一包含出错信息的对象并将该对象抛出当前环境中。

格式

- throw 对象；

过程

- 类似于return的过程，传回调用它的函数，退出函数，回收所有局部对象。

异常抛出（throw）

作用

- 当程序发生异常情况，而在当前的环境中获取不到异常处理的足够信息，可以创建一包含出错信息的对象并将该对象抛出当前环境中。

格式

- throw 对象；

过程

- 类似于return的过程，传回调用它的函数，退出函数，回收所有局部对象。

例1

throw myerror（“something bad happened”）; // myerror是一个类，它以字符串变量为参数
throw 5；//int是一个内部类型，5是一个int类型的常数

抛出语句的格式

- 抛出异常的语句形式：

  throw <可选的操作数>;

- throw通常指定一个操作数。throw的操作数可以是任何类型，如果操作数是个对象，则称为异常对象。也可以抛出条件表达式而不是抛出对象，可以抛出不用于错误处理的对象。
- 抛出异常的过程类似于return的过程，生成和初始化throw操作数的一个临时副本，传回调用它的函数，退出函数，回收所有局部对象。

异常抛出实例 – 异常类定义

```
// Class DivideByZeroException to be used in exception
// handling for throwing an exception on a division by zero.
class DivideByZeroException {
public:
   DivideByZeroException() : message( "attempted to divide by zero" ) { }
   const char *what() const {return message;} //告诉用户出现了什么异常
private:
   const char *message; //记录出现的异常情况
};
```

异常抛出实例 – 异常抛出

```
double Div(int x, int y )
{
   if ( y == 0 )
        throw DivideByZeroException();

   return double( x ) / y;
}
```

异常抛出实例 – 异常抛出

```
double Div(int x, int y )
{
   if ( y == 0 )
        throw DivideByZeroException();

   return double( x ) / y;
}
```

这条语句用默认构造函数生成了一个DivedeByZeroException类的对象，并把这个对象返回给调用它的函数，div函数执行结束

异常捕获(try --- catch)

- 一个函数抛出异常，它必须假定该异常能被捕获和处理。异常捕获机制使得C++可以把问题集中在一处解决。
- 如果某段程序可能会抛出异常，则必须通知系统启动异常处理机制，即try语句块

启动异常捕获机制

- 异常处理代码的一般形式：

```
try{
        可能抛出异常的代码
}
catch(类型1  参数1){处理该异常的代码        }
catch(类型2  参数2){处理该异常的代码        }
…       …
```

try语句块中包含了可能抛出异常的代码，一旦抛出了异常，则退出try语句块，进入try后面的异常捕获和处理

catch捕获异常

异常处理器放在catch块中，形式如下：

```
catch ( <捕获的异常类型> <可选参数> )
{
   <异常处理器代码>
}
```

注意

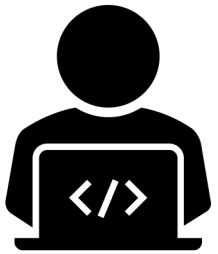
- 异常捕获是以类型为标志，与对象无关
- 对象是可选的
- 如果有对象，则可以在处理器中引用这个对象。如果不需要引用对象值，则可省略对象
- catch (…) 捕获任意类型的异常。

异常捕获原理

- 如果一个异常信号被抛出，异常处理器中第一个参数与异常抛出对象相匹配的函数将捕获该异常信号，然后进入相应的catch语句，执行异常处理程序。
- 如果找遍了所有catch处理器都不匹配，则函数执行结束，并将该异常抛给调用它的函数，由调用他的函数来处理该程序。

异常捕获实例：除零异常

```
double Div(int x, int y )
{    if ( y == 0 )
         throw DivideByZeroException();
     return double ( x ) / y; }
int main()
 {  int number1, number2;
    double result;
    cout << "Enter two integers (end-of-file to end): ";
    while ( cin >> number1 >> number2 ) {
       try { result = Div( number1, number2 );
              cout << "The quotient is: " << result << endl;
            }   catch ( DivideByZeroException ex ) {
                     cout << "Exception occurred: "   << ex.what() << '\n‘;   }
       cout << "\nEnter two integers (end-of-file to end): “;
     }
   cout << endl;
   return 0;
}
```



代码演示

try-catch-example.cpp

异常捕获实例：除零异常

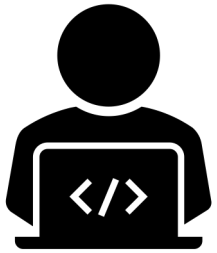
```
double Div(int x, int y )
{   if ( y == 0 )
        throw DivideByZeroException();
    return double ( x ) / y; }

int main()
 {  int number1, number2;
    double result;
    cout << "Enter two integers (end-of-file to end): ";
    while ( cin >> number1 >> number2 ) {
       try { result = Div( number1, number2 );
             cout << "The quotient is: " << result << endl;
           }  catch ( DivideByZeroException ex ) {
                 cout << "Exception occurred: "   << ex.what() << '\n';  }
       cout << "\nEnter two integers (end-of-file to end): ";
     }
   cout << endl;
   return 0;
}
```

(2) 抛出

(1) 捕获

(3) 处理



代码演示

try-catch-example.cpp

try语句块中包含了可能抛出异常的代码Div，一旦抛出了异常，则退出try语句块，进入try后面的异常捕获和处理

若没有抛出异常，跳过catch执行catch后的语句

输出结果

Enter tow integers (end-of-file to end); l00 7
The quotient is: 14.2857

Enter tow integers (end-of-file to end); 100 0
Exception occurred: attempted to divide by zero

Enter tow integers (end-of-file to end); 33 9
The quotient is: 3.66667

Enter tow integers {end-of-file to end):

输出结果

Enter tow integers (end-of-file to end): l00 7
The quotient is: 14.2857

Enter tow integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero

Enter tow integers (end-of-file to end): 33 9
The quotient is: 3.66667

Enter tow integers {end-of-file to end):

错误处理后依然在main中，没有退出程序。

异常抛出与检测实例二

抛出的异常不一定是对象，可以是一个结果为内置类型的表达式

```
int Div(int x, int y)
{ if (y==0) throw y;
  return x/y;
}
```

异常抛出与检测实例二

```
int Div(int x, int y)
{
    if (y==0) throw y;
    return x/y;
}

int main()
{
    try   {
        cout << Div(6,3) << endl;
        cout << Div(10,0) << endl;
        cout << Div(5,2) << endl;
    }
    catch (int) {  cout << "divide by zero" << endl;  }
    cout << "It's Over" << endl;
    return 0;
}
```

```
2
divide by zero
It's Over
```

异常抛出与检测实例二

```
int Div(int x, int y)
{
    if (y==0) throw y;
    return x/y;
}

int main()
{
    try  {
        cout << Div(6,3) << endl;
        cout << Div(10,0) << endl;
        cout << Div(5,2) << endl;
    }
    catch (int) {  cout << "divide by zero" << endl;  }
    cout << "It's Over" << endl;
    return 0;
}
```

```
2
divide by zero
It's Over
```

优点：

- 轻量级。不需自己定义抛出机制

缺点：

- 可读性、复用性较低

解一元二次方程，带异常处理

```
#include <iostream>
#include <cmath>

class noRoot {};
class divByZero {};

double Sqrt(double x)
{   if (x < 0) throw noRoot();
    return sqrt(x);
}

double div(double x, double y)
{   if (y == 0) throw divByZero();
    return x / y;
}
```

```
int main()
{   double a, b, c, x1, x2, dlt;
    cout << "请输入3个参数： " << endl;
    cin >> a >> b >> c;
    try {
       dlt = Sqrt(b * b - 4 * a * c);
       x1 = div(-b + dlt,  2 * a);
       x2 = div(-b - dlt,  2 * a);
       cout << "x1=" << x1 << "    x2=" << x2 << endl;
     }   catch (noRoot) { cout << "无根" << endl; }
     catch (divByZero) {cout << "不是一元二次方程" << endl; }
    return 0;
}
```

本章框架

- 什么是异常
- 异常处理机制
- 异常规格说明

异常规格说明

- **传统函数声明**
- void f();

函数可以抛出任何异常。

- **带异常规格的函数声明**
- void f() throw();　　函数不会有异常抛出。
- Void f() throw(toobig, toosmall, divzero)；函数会抛出toobig, toosmall, divzero三种异常

异常规格说明示例

```
class up{};
class down{};
void f(int i) throw(up, down);

int main()
{
    for (int i=1;i<=3;++i)
        try { f(i); }    catch (up) {cout << "up catched" << endl; }
                         catch (down) {cout << "down catched" << endl;}
   return 0;
}

void f(int i) throw(up, down)
{
     switch(i)   {
         case 1: throw up();
         case 2: throw down();
     }
}
```

异常规格说明示例

```
class up{};
class down{};
void f(int i) throw(up, down);

int main()
{
    for (int i=1;i<=3;++i)
        try { f(i); }    catch (up) {cout << "up catched" << endl; }
                         catch (down) {cout << "down catched" << endl;}
   return 0;
}

void f(int i) throw(up, down)
{
     switch(i)   {
         case 1: throw up();
         case 2: throw down();
     }
}
```

(1) 捕获

(2) 抛出

(3) 处理

异常规格说明示例

```
class up{};
class down{};
void f(int i) throw(up, down);

int main()
{
    for (int i=1;i<=3;++i)
        try { f(i); }    catch (up) {cout << "up catched" << endl; }
                         catch (down) {cout << "down catched" << endl;}
   return 0;
}

void f(int i) throw(up, down)
{
    switch(i)   {
        case 1: throw up();
        case 2: throw down();
    }
}
```

(1) 捕获

(2) 抛出

(3) 处理

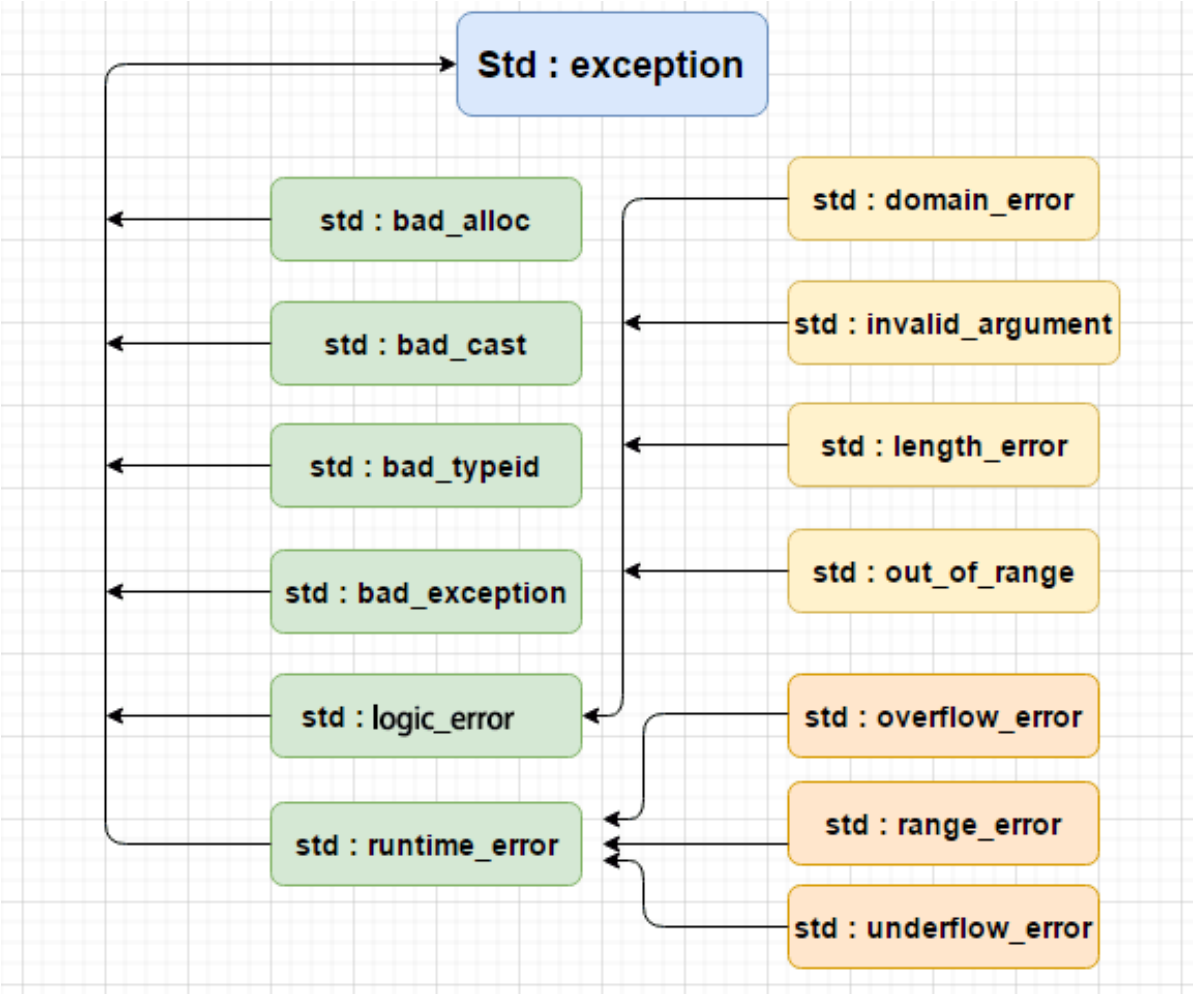
up catched
down catched

C++11的改进

- **摒弃了异常规格声明，理由是**
- 可以用注释解决
- 无法完整地说明可能抛出的异常

- **新功能**
- 关键字noexcept: void f() noexcept
- 有助于编译器的优化
- 运算符noexcept：判断表达式是否会抛异常。如：noexcept(f)结果是false

C++中的标准异常



https://docs.microsoft.com/en-us/cpp/cpp/errors-and-exception-handling-modern-cpp?view=

小结

- 为了提高程序的可靠性，程序需要对各种可能的异常进行处理
- 某些错误需要异地处理
- C++的异常机制由try、throw和catch构成