第八章　异常处理

本章导读

C++将输入(input)和输出(output)看着字节流。输入时，程序从输入流中抽取字节；输出时，程序将字节插入到输出流中。在C++中，cin代表标准输入，从键盘输入字符到程序；cout代表标准输出，从程序输出到屏幕。在C++中，除了从标准输入和输出外，还可以从文件实现输入和输出。

流(stream)是中介。流与键盘或文件等建立联系，程序与流交互，由流在与键盘或文件等交互。

学习目标：

1. 了解stream的概念；
2. 掌握标准输入输出；
3. 掌握文件的输出；
4. 掌握文本文件和二进制文件；
5. 掌握顺序文件和随机文件；

本章目录

异常处理是代码健壮性的重要组成，但在开发实践中常常忽略异常处理，似乎程序总是在无错误的理想状态下运行，比如：逻辑不会错误、内存总是足够、文件永远存在、硬盘长期不会损伤等。

忽视异常处理的原因很多，首先是意识问题，其次是检测错误常常是乏味且将导致代码量增多的工作。但是，忽视异常处理的后果将是非常严重。在C语言的异常处理较为笨拙且难以使用，而C++异常处理实现了异常触发和异常处理的分离，使得C++的异常处理变得简单易用，成为C++的重要特征之一。

第一节　快速入门

观察例程8-1会发现，当Y输入0时，函数Division()代码中将出现错误，并终止程序的运行，常规处理措施是将Division()函数修改如例程8-2所示。这种处理措施未必妥当。当程序规模较大时，可能已经做了很多工作，如果因为一个小小的错误就终止程序的运行，损失很大；但是对于Division()函数而言，仅仅知晓本函数的运行场景，未必知道发生这种错误时，该如何处理由此产生的各个可能。

例程8-1

```
第1行  #include<iostream>
第2行  using namespace std;
第3行
第4行  int Division(int a,int b){
第5行      return a/b;
第6行  }
第7行  int main(){
第8行      int X,Y;
第9行      cin>>X>>Y;
第10行     cout<<Division(X,Y);
```

```
第11行
第12行      return 0;
第13行  }
```

例程8-2

```
第1行  int Division(int a,int b){
第2行      if(b==0){
第3行          cout<<"除数不能为0!!!";
第4行          exit(1);
第5行      }else{
第6行          return a/b;
第7行      }
第8行  }
```

在C++中，当发生错误时，可以throw(抛出)一个异常(exception)，如例程8-3所示，对抛出的异常的处理如例程8-4所示。

例程8-3

```
第1行  int Division(int a,int b){
第2行      if(b==0)throw b;//throw的功能是抛出异常
第3行      return a/b;

第4行  }
```

例程8-4

```
第1行  #include<iostream>
第2行  using namespace std;
第3行
第4行  int Division(int a,int b){
第5行      if(b==0)throw b;//throw的功能是抛出异常
第6行      return a/b;
第7行  }
第8行  int main(){
第9行      int X,Y;
第10行     cin>>X>>Y;
第11行     try{
第12行         cout<<Division(X,Y);
第13行     }catch(int e){
第14行         cout<<"发生异常"<<e;
第15行     }
第16行
第17行     return 0;
第18行 }
```

比较例程8-2和例程8-4，会发现处理异常的代码很相似，但是对于例程8-2而言，异常处理在函数内部，仅仅考虑不满足本函数条件下的处理；而例程8-4，同样可以接收函数发出的异常，并且还可以处理函数外的各种情形。很明显，C++的这种处理方式更加合理。

从例程8-4可以看出，C++中的异常分为两个步骤即用throw语句触发并抛出异常以及用try……catch语句处理异常。在try语句块中发生的一个或多个异常，都将交由catch语句块处理，catch语句可以全部处理，也可以仅处理其中一部分。

仅仅抛出一个异常值，难免不知所云，如果抛出的异常含有更多信息，将更加有利。观察例程8-4第5行，会发现throw b抛出的是int型数值，此处也可以替换为class或者struct，如例程8-5所示，可以称此处的EX类为异常类，含有更加丰富的异常信息。

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

class EX{
public:
    EX(int a){
        if(a==0){
            exValue=a;
            exInfo="除数不能为0";
        }
    }
    friend ostream& operator<<(ostream&out,const EX e);
private:
    int exValue;
    string exInfo;
};
ostream& operator<<(ostream&out,const EX e){
    return out<<"错误信息:"<<e.exInfo<<" 错误值："<<e.exValue;
}

int Division(int a,int b){
    if(b==0)throw EX(b);//throw的功能是抛出异常
    return a/b;
}

int main(){
    int X,Y;
    cin>>X>>Y;
    try{
        cout<<Division(X,Y);
    }catch(EX e){
        cout<<e;
    }

    return 0;
}
```

第二节　标准异常类

C++标准库(Standard Library)含有很多类和函数，部分类或函数需要抛出异常，因此需要定义异常类。C++的标注异常类关系如图8-1所示。在C++标准库中，exception类是所有异常类的基类，等效模拟定义代码如例程8-6所示，定义于<exception>库文件中，类中包含虚函数what()，用于返回错误信息，返回值为“\0”结尾的字符数组。其他异常类都派生于exception基类。

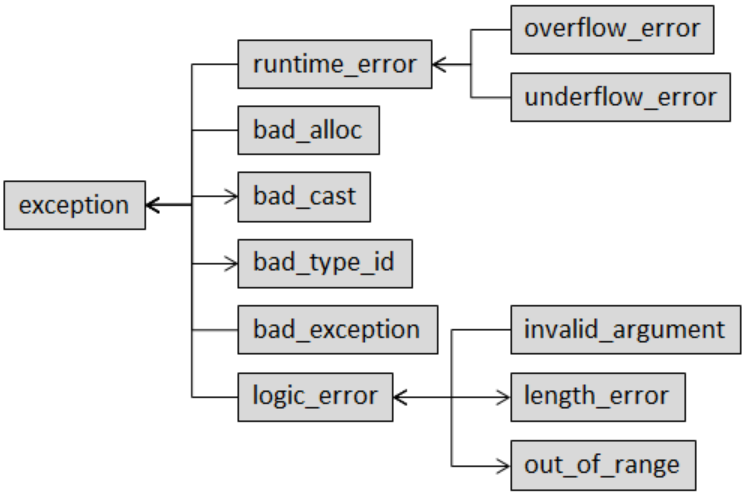

图8-1　C++标准异常类继承关系

例程8-6

```
第1行  class exception{
第2行  public:
第3行      exception();
第4行      exception(const char * const &);
第5行      exception(const exception&);
第6行      exception& operator=(const exception&);
第7行      virtual ~exception();
第8行      virtual const char * what() const;
第9行
第10行 private:
第11行     const char * Msg;
第12行 };
```

类run_time_error继承自exception基类，定义于库文件<stdexcept>中，用于描述运行时错误。派生类overflow_error用于描述向上溢出，即数值太大；派生类underflow_error用于描述向下溢出，即数值太小。

类logic_error继承自exception基类，定义与库文件<stdexcept>中。派生类invalid_argument用于描述向函数传递非法实参；length_error用于描述长度超范围；out_of_range用于描述取值超过允许范围。

类bad_alloc、bad_cast、bad_type_id以及bad_exception用于描述C++运算符抛出的异常。类bad_alloc在new运算符无法分配内存时所抛出的异常；类bad_cast在dynamic_cast运算符没有成功转换为引用类型时抛出的异常；类bad_type_id在执行typeid运算符其运算对象为空指针时抛出的异常；类bad_exception则用于描述从未预料的异常时所抛出的异常，当此时，C++将调用unexpected()函数。另，类bad_exception与异常规约等相关，请参看该小节。

例程8-7是bad_alloc异常类的使用，图8-2是new运算符所触发的bad_alloc异常的执行图。

例程8-7

```
第1行  #include<iostream>
第2行  using namespace std;
第3行
第4行  int main(){
第5行      try{
第6行          for(int i=0;i<100;++i){
第7行              new int[100000000];//1亿
第8行              cout<<"第"<<(i+1)<<"次"<<"创立数组"<<endl;
第9行          }
```

```
第10行      }catch(bad_alloc &e){
第11行          cout<<"发生异常，异常信息="<<e.what()<<endl;
第12行      }
第13行
第14行      return 0;
第15行  }
```

```
第1次创立数组
第2次创立数组
第3次创立数组
第4次创立数组
发生异常，异常信息=bad allocation
```

图8-2　new运算符所触发bad_alloc异常

第三节　自定义异常类

C++提供了不少异常类，涵盖异常的方方面面，一般情况下应使用标准异常类，虽然C++允许自创异常类，但应尽量避免使用自创的异常类。即便自创异常类，也应派生自基类exception及其其派生类，这样就能方便地使用exception类的共性，如what()虚拟函数等。例程8-8是定义创建三角形时发生的两种异常，一是三角形某条边的值小于等于0，二是三角形两边之和小于等于第三边。例程8-8用于处理边长小于等于0时的异常。

例程8-8

```
第1行  #include<iostream>
第2行  #include<stdexcept>//logic_error定义在stdexcept中
第3行  using namespace std;
第4行
第5行  class negativeSideException:public logic_error{
第6行  public:
第7行      negativeSideException(double side):logic_error("Negative Side Exception!"){
第8行          this->side=side;
第9行      }
第10行 private:
第11行     double side;
第12行 };
第13行
第14行 class Triangle{
第15行 public:
第16行     Triangle(double side1,double side2,double side3){
第17行         checkSide(side1);checkSide(side2);checkSide(side3);
第18行         this->sideA=side1;
第19行         this->sideB=side2;
第20行         this->sideC=side3;
第21行     }
第22行     void printTriangle(){
第23行         cout<<"sideA="<<sideA<<"sideB="<<sideB<<"sideC="<<sideC<<endl;
第24行     }
第25行     inline double getPerimeter(){
第26行         return sideA+sideB+sideC;
第27行     }
```

```
第28行      inline double getArea(){
第29行          double s=getPerimeter()/2;
第30行          return sqrt(s*(s-sideA)*(s-sideB)*(s-sideC));
第31行      }
第32行  private:
第33行      double sideA,sideB,sideC;
第34行      void checkSide(double side){
第35行          if(side<=0)throw negativeSideException(side);
第36行      }
第37行  };
第38行
第39行  int main(){
第40行      try{
第41行          Triangle myTri(3,4,5);//将正数改成负数将触发异常
第42行          cout<<myTri.getArea()<<endl;
第43行      }catch(negativeSideException &e){
第44行          cout<<e.what()<<endl;
第45行      }
第46行
第47行      return 0;
第48行  }
```

第35行代码if(side<=0)throw negativeSideException(side)在side值小于等于0时抛出异常negativeSideException，而negativeSideException类继承自logic_error(定义于第5-12行)。由于negativeSideException继承自logic_error，logic_error继承自exception基类，因此，可以用what()函数返回异常信息。例程8-9是类logic_error的模拟等效实现。

例程8-9

```
第1行  class logic_error:public exception{
第2行  public:
第3行      logic_error(const string&Msg):exception(Msg.c_str()){}
第4行      logic_error(const char *Msg):exception(Msg){}
第5行  };
```

例程8-8没有处理两边之和不大于第三边时的异常，将在下一节“多重异常”中处理。

第四节　多重异常

try块在执行过程中，一般不会抛出异常，但也可能抛出一个或者多个异常，且抛出不同类型的异常。C++可以允许在try块后配置多个catch块，分别捕获不同类型的异常。例程8-10在例程8-8基础上，增加了“两边之和小于等于第三边”异常。在第60行和第62行分别catch异常类negativeSideException和异常类TriangleException。

例程8-10

```
第1行  #include<iostream>
第2行  #include<stdexcept>//logic_error定义在stdexcept中
第3行  using namespace std;
第4行
第5行  class negativeSideException:public logic_error{
第6行  public:
第7行      negativeSideException(double side):logic_error("Negative Side Exception!"){
```

```
第8行          this->side=side;
第9行      }
第10行 private:
第11行     double side;
第12行 };
第13行
第14行 class TriangleException:public logic_error{
第15行 public:
第16行     TriangleException(double side1,double side2,double side3)
第17行         :logic_error("The sum of some two sides is not greater than the third side!"){
第18行         this->sideA=side1;this->sideB=side2;this->sideC=side3;
第19行     }
第20行     void print(){
第21行         cout<<"sideA="<<sideA<<"sideB="<<sideB<<"sideC="<<sideC<<endl;
第22行     }
第23行 private:
第24行     double sideA,sideB,sideC;
第25行 };
第26行
第27行 class Triangle{
第28行 public:
第29行     Triangle(double side1,double side2,double side3){
第30行         checkSide(side1);checkSide(side2);checkSide(side3);
第31行         if(!isValid(side1,side2,side3))throw TriangleException(side1,side2,side3);
第32行         this->sideA=side1;
第33行         this->sideB=side2;
第34行         this->sideC=side3;
第35行     }
第36行     void printTriangle(){
第37行         cout<<"sideA="<<sideA<<"sideB="<<sideB<<"sideC="<<sideC<<endl;
第38行     }
第39行     inline double getPerimeter(){
第40行         return sideA+sideB+sideC;
第41行     }
第42行     inline double getArea(){
第43行         double s=getPerimeter()/2;
第44行         return sqrt(s*(s-sideA)*(s-sideB)*(s-sideC));
第45行     }
第46行 private:
第47行     double sideA,sideB,sideC;
第48行     void checkSide(double side){
第49行         if(side<=0)throw negativeSideException(side);
第50行     }
第51行     bool isValid(double side1,double side2,double side3){
```

```
第52行          return (side1+side2>side3)&&(side1+side3>side2)&&(side2+side3>side1);
第53行      }
第54行  };
第55行
第56行  int main(){
第57行      try{
第58行          Triangle myTri(1,2,3);//负数将触发异常,两边之和不大于第三边也将触发异常
第59行          cout<<myTri.getArea()<<endl;
第60行      }catch(negativeSideException &e){
第61行          cout<<e.what()<<endl;
第62行      }catch(TriangleException &e){
第63行          cout<<e.what()<<endl;
第64行          e.print();
第65行      }
第66行
第67行      return 0;
第68行  }
```

多个不同的异常类可以派生自同一个基类，如例程8-10所示，异常类TriangleException和negativeSideException都派生自类logic_error。当用catch()捕获多个不同类型的异常时，有时异常的次序对异常的捕获有影响。当将基类异常放在前时，由于派生的异常匹配该异常，因此将被触发，而不会触发其后的派生类异常。

第五节　异常规约

异常规约(exception specification)又称异常抛出列表，是在函数申明中列出函数可能抛出的异常类型。如未定义异常规约，则函数可以抛出任何异常。虽然省略异常规约会显得方便，但是很不好的习惯，非函数函数开发人员不知道函数将抛出哪些异常，就难以在try-catch中编写出相应的处理程序，也就难以编写出更加健壮的程序。异常规约的格式如下。其中exceptionList是异常列表，可以是一个或者多个；parameterList是参数列表；functionName是函数名称；rtnType是返回值类型。异常规约在定义函数原型时申明。例程8-11是异常规约的一个示例。

```
rtnType functionName(parameterList)throw(exceptionList)
```

例程8-11

```
第1行  void checkSide(double side)throw(negativeSideException);
第2行  Triangle(double side1,double side2,double side3)throw(TriangleException);
```

当一个函数有可能抛出多个异常时，可以罗列多个异常(异常之间用逗号分开)。当将throw()放于函数原型之后时，表明函数不能抛出任何异常。当将throw(...)置于函数原型之后时，表明函数能抛出各种异常。例程8-12异常规约的应用示例。当将14行代码注释后，则testFunction()抛出char型异常与异常规约不一致，catch()将不能捕获该异常，也不能处理该异常。而当14行代码catch(...)正常发挥作用时，则能捕获所有其他异常，即不在其他catch()中捕获的异常。类似switch-case中的case和default。具体罗列的catch()相当于case，catch(...)相当于default。

例程8-12

```
第1行  #include<iostream>
第2行  #include<exception>
第3行
第4行  //异常规约，抛出异常为int类型
第5行  void testFunction()throw(int){
第6行      throw 'x';//抛出异常为char类型
第7行  }
```

```
第8行
第9行  int main (void) {
第10行      try{
第11行          testFunction();
第12行      }
第13行      catch(int){std::cerr<<"caught int\n";}//捕获int型异常
第14行      catch(...){std::cerr<<"caught some other exception type\n";}
第15行      //捕获其他类型的异常，注意省略号的使用
第16行
第17行      return 0;
第18行  }
```

第六节　异常传播

当异常发生时，将被try-catch块所捕获，然后进行相应的处理。如果异常发生但没有被try-catch模块所捕获，则将终止程序的执行。如果try块内没有异常发生，则catch语句块将被跳过。当try块内的某条语句发生异常时，则该语句之后的语句将被跳过，同时C++开始搜寻对应的catch语句。搜寻时，C++沿着函数调用链逆向寻找，并从前向后检查每个catch块，如果匹配成功，则将异常对象赋予catch块参数，执行对应catch块内的代码，执行完毕后执行try后所有catch后的代码；如果匹配不成功，则继续向调用该函数的函数执行上述搜寻。如果在函数调用链条中都没有找到匹配的catch，则输出错误信息，终止程序运行。例程8-13有关异常传播程序，图8-3是其执行效果。

例程8-13

```
第1行  #include<iostream>
第2行  #include<exception>
第3行  using namespace std;
第4行
第5行  void testFunction3()throw(char){
第6行      cout<<"In testFunciton3()!!!"<<endl;
第7行      throw '3';
第8行      cout<<"Out testFunciton3()!!!"<<endl;
第9行  }
第10行 void testFunction2()throw(int){
第11行     cout<<"In testFunciton2()!!"<<endl;
第12行     try{
第13行         testFunction3();
第14行     }catch(char e){
第15行         cout<<"Output Exception. "<<e<<" In testFunciton2()!!"<<endl;
第16行         //throw;//重新抛出异常，注意删除异常后的效果
第17行     }
第18行     throw 2000;
第19行     cout<<"Out testFunciton2()!!"<<endl;
第20行 }
第21行 void testFunction1()throw(double){
第22行     cout<<"In testFunciton1()!"<<endl;
第23行     try{
第24行         testFunction2();
第25行     }catch(int &e){
```

```
第26行            cout<<"Output Exception. "<<e<<" In testFunciton1()!"<<endl;
第27行        }
第28行        throw 1000.99;

第29行        cout<<"Out testFunciton1()!"<<endl;
第30行    }
第31行
第32行    int main(void){
第33行        try{
第34行            testFunction1();
第35行        }catch(char e){
第36行            cout<<"Output Exception. "<<e<<" In main()"<<endl;
第37行        }catch(int e){
第38行            cout<<"Output Exception. "<<e<<" In main()"<<endl;
第39行        }catch(double e){
第40行            cout<<"Output Exception. "<<e<<" In main()"<<endl;
第41行        }
第42行        cout<<"Out main()!"<<endl;
第43行
第44行        return 0;
第45行    }
```

```
In testFunciton1()!
In testFunciton2()!!
In testFunciton3()!!!
Output Exception.  3  In testFunciton2()!!
Output Exception.  2000  In testFunciton1()!
Output Exception.  1000.99  In main()
Out main()!
```

```
In testFunciton1()!
In testFunciton2()!!
In testFunciton3()!!!
Output Exception.  3  In testFunciton2()!!
Output Exception.  3  In main()
Out main()!
```

图8-3　异常传播

从图中可以看出，当testFunction3()抛出异常throw '3'时，向其调用函数testFunction2()寻找catch，发现能匹配；当该catch语句块执行语句块后，testFunction2()抛出异常throw 2000，因此从此开始寻找其调用函数testFunction1()函数的catch块，发现有其匹配的catch，执行之；执行完毕后，抛出异常throw 1000.99，搜寻其调用函数main()，找到其匹配catch并执行，然后执行catch后的第42行语句。

当将第16行的throw语句前的注释符号删除后，则将重新抛出当前异常，此处则相当于在testFunction2()中重新执行throw '3'，然而其调用函数testFunction1()找不到匹配的catch语句块，因此继续搜寻testFunction1()的调用函数main()，找到匹配catch，执行该catch内的语句后执行所有catch后的语句，即第42行语句。

第七节　小结

1. 使用异常处理机制，将使C++程序更加健壮。异常处理可以将错误处理代码和异常程序分离，因此程序更加易读易维护。
2. 当函数抛出异常时，C++将把异常抛出并传递给函数的调用者，并查找匹配的catch语句块，如果没有查找成功，则将沿着函数调用关系继续查找，直到找到为止。如果最终未能找到，则将终止程序执行，并输出错误信息。
3. throw语句能抛出任何类型的异常值，包括基本数据类型和类或结构类型。此值被传递至catch块，catch语句可以利用该值进行异常处理。
4. 当异常被抛出后，正常的程序流被中断，异常触发点后的语句将不被执行，并启动查找匹配catch过程，当第一次被匹配后，执行该catch内的语句，执行完毕后，跳转到所在try-catch后语句继续执行。

5. C++预定义了很多标准异常类，包括exception基类及其派生类runtime_error和logic_error。使用过程中，应尽量使用标准异常类。
6. 当标准异常类不能描述其异常时，可以自定义异常类，并最好继承标准异常基类或派生类。
7. 当定义函数可能抛出异常时，应在函数原型中申明异常类型，以便根据异常类别处理相应异常。
8. 异常处理机制将会消耗更多的运行时间和系统资源。简单逻辑错误，可用分支语句处理，不宜使用异常机制。异常处理机制常用于分支语句不能处理的情况。