C++程序设计精要教程

华中科技大学

第9章 异常与断言

◆9.1 异常处理

- 异常：一种意外破坏程序正常处理流程的事件、由硬件或者软件触发的事件。
- 异常处理可以将错误处理流程同正常业务处理流程分离，从而使程序的正常业务处理流程更加清晰顺畅。
- 异常发生后自动析构调用链中的所有对象，这也使程序降低了内存泄漏的风险。
- 由软件引发的异常用throw语句抛出，会在抛出点建立一个描述异常的对象，由catch捕获相应类型的异常。

第9章 异常与断言

◆9.1 异常处理

- 异常对象:描述异常事件的对象
- try：程序的正常流程部分，可引发异常，由异常捕获部分处理。
- thow：用于引发异常、继续传播异常
- catch：异常捕获部分。处理时，根据要捕获的异常对象类型处理相应异常事件，可以继续传播或引发新的异常。
- 一旦异常被某个catch处理过程捕获，则其它所有处理过程都会被忽略。
- C++不支持finally处理过程

异常处理的结构

```
……
try{
    f1( );
    f2( );
}
catch(…)
……
```

try语句块里的语句或函数调用都可能产生异常。异常应在某个try语句块中抛出

每个try语句块后必须至少有一个catch

try-catch语句块可以嵌套

任何一条语句都可以用try-catch语句替换(包括catch中的语句)

第9章 异常与断言

- 可以抛出任何类型的异常，抛出异常的throw语句一定要间接或直接在try语句块中
- 如果抛出的异常不在任何一个try语句块中，这种没有异常处理过程捕获的异常将引起程序终止执行
- 在try中抛出的异常如果没有合适的catch捕获，也将引起程序终止执行
- 一个try中可以抛出多种异常。

```
throw 1;
throw ‘c’;
throw new int(3);
throw new int[3]{1,2,3}
throw “abcdef”;
throw CExecption();
throw new CExecption();
```

异常处理的结构

异常(Exception)处理流程

```
try {
   // statements throw E1
   // statements throw E2
   …
 }
catch(E1){
  //handle the exception E1
}
catch(E2){
  //handle the exception E2
}
statements;
```

try语句块中的语句可能抛出各种异常。但只要抛出第一个异常，try语句块马上结束，转到catch子句。如果try语句块执行成功，则跳过catch，执行try{}catch{}的后续statements;

可定义多个catch子句截获可能抛出的各种异常。但最多只会执行其中一个异常处理过程。在相应异常被处理后，其他异常处理过程都将被忽略。如果异常处理成功，则执行后续statements。如果异常处理过程中又抛出新的异常，则后续statements不会被执行

异常处理的结构

异常的抛出都是由throw语句直接或间接抛出：

1：程序运行时的逻辑错误导致异常间接抛出，例如数组指针++导致越界时由运行库throw出异常

2：程序在满足某条件时，用throw语句直接抛出异常，如

```
if(满足某条件){
        throw “A exception occurred”;
```

抛出异常使用throw:

```
……
throw CExecption();
……
```

抛出的异常可以是任意类型：

```
throw 1;
throw ‘c’;
throw CExecption();
throw new int[4];
```

异常处理的结构

捕获异常：根据异常类型由catch子句捕获

```
……
catch（异常类型 变量）
{
    //异常处理
}
……
```

可以是指针等任何类型：

```
……
catch(int e){...}
catch(char ch){…;}
catch(CExecption e){…}
……
```

catch子句必须定义且只能定义一个参数，该参数不能是void，可以是值类型，指针、左值引用，不能是右值引用。

异常处理的结构

catch必须出现try语句块后面

```
try{ …
      if(rr==0) throw -1;
}
catch(int t){
      cout<<t<<endl;
}
```

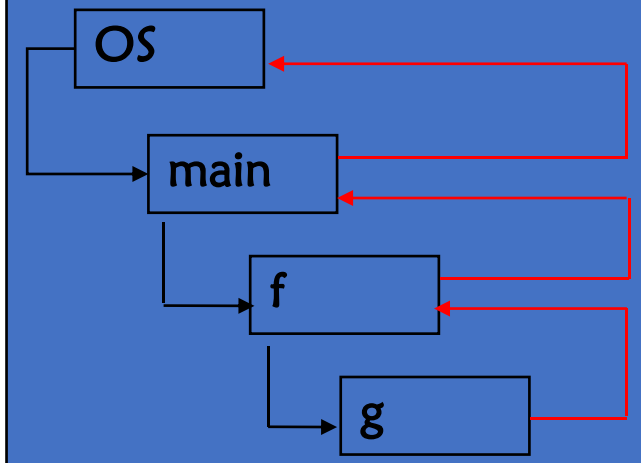
- 不能有单独的try语句块或者单独的catch语句块
- throw语句要求直接或者间接出现在try语句块中,异常必须被某个try的catch捕获。如果没有被任何catch捕获，则程序将被终止执行。

异常处理的结构

异常捕获的过程：栈展开

- 当一个try语句块有异常抛出时，首先检查与该try块相关的catch子句，如果找到了匹配的catch子句，则由该catch子句处理异常
- 如果该try块没找到匹配的catch子句，且该try块嵌套在外层try块里，则检查外层try块的catch子句，这个过程递归进行
- 如果到最外层try块还没找到，则退出当前函数，在调用当前函数的外层函数继续寻找，这个过程递归进行
- 如果到main函数还没找到匹配的catch子句，程序退出，将异常交给OS处理
- 上述过程称为栈展开过程，这里的栈指函数调用链形成的调用栈。

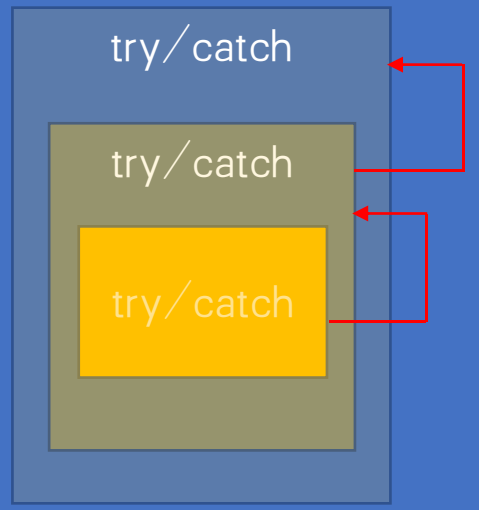
异常处理的结构



如果该异常没有被main处理，则异常被抛到OS去处理

如果该异常没有f被处理，则异常被抛到f的调用者main去处理

如果函数g里抛出的异常没有被处理，则异常被抛到g的调用者f去处理



在每个函数内部，则按照try/catch的嵌套关系逐层由内向外层查找匹配的catch子句

异常处理的结构

重新抛出异常和捕获所有异常

- throw语句通常后面跟一个表达式，表达式求值结果就是抛出的异常对象
- 但有时catch子句在捕获异常并作了一些处理后，需要由嵌套的外层try/catch或调用链更上一层的函数继续处理异常，这时可以用不带表达式的throw语句重新抛出异常，如：

```
catch(MyException &ex){
        //do something
        throw; //重新抛出异常
}
```

- 有时希望能捕获所有异常，或者不知道可能抛出的异常对象类型，这时可以用catch(…)来捕获所有异常。但要注意如果有其他catch子句，那么catch(…)应该在最后位置。

第9章 异常与断言

◆9.1 异常处理

- 异常处理过程必须定义且只能定义一个参数，该参数必须是省略参数或者是类型确定的参数，因此，异常处理过程不能定义void类型的参数。
- 如果是通过new产生的指针类型的异常，在catch处理过程捕获后，通常应使用合适的delete释放内存，否则可能造成内存泄漏。
- 如果继续传播指针类型的异常，则可以不使用delete。
- 抛出字符串常量如“abc”的异常需要用catch(const char *p)捕获，处理异常完毕可以不用delete p释放，因为字符串常量通常存储在数据段。

第9章 异常与断言

```
#include <iostream>
using namespace std;
class VECTOR
{
   int* data;                          //用于存储向量元素
   int  size;                          //向量的最大元素个数
public:
   VECTOR(int n);                      //构造最多存储n个元素的向量
   int& getData(int i);                //取下标所在位置的向量元素
   ~VECTOR( ) { if(data) { delete[]data; data=nullptr; } };
};
class INDEX {                          //定义异常的类型
   int index;                          //异常发生时的下标值
public:
   INDEX(int i) { index = i; }
   int getIndex( )const { return index; }
};
```

第9章 异常与断言

```
VECTOR::VECTOR(int n)
{
   if (!(data=new int[size = n]))        //分配内存失败则抛出异常
      throw(INDEX(0));                   //抛出一个异常对象
};
    int& VECTOR::getData(int i)
    {
       if (i < 0 || i >= size)                          //下标越界则抛出异常
          throw INDEX(i);
       return data[i];
    };
```

第9章 异常与断言

```
void main(void)
{
   VECTOR v(100);                    //定义向量最多存放100个元素
   try {
      v.getData(101)=30;             //调用int &operator[ ](int i)发生下标越界异常
   }
   catch (const INDEX & r){          //捕获INDEX及其子类类型的异常
      int  i = r.getIndex( );
      switch (i) {
         case 0:    cout << "Insufficient memory!\n"; break;
         default:   cout << "Bad index is " << i << "\n";
      }
   }//C++的异常处理没有finally部分，但是不同的编译器可能由不同的支持方法
   cout << "I will return\n";
}
```

第9章 异常与断言

◆9.1 异常处理

- 没有任何实参的throw用于传播已经捕获的异常。
- 任何throw后面的语句都会被忽略，直接进入异常捕获处理过程catch。
- 如果是通过new产生的指针类型的异常，并且该异常不再传播，则一定要使用delete释放，以免造成内存泄漏。

```
try{……
    throw 1;        //正确
    //或throw  //本try语句出现在某个catch中，或间接被某个catch包含
}
catch(int e){
    throw;          //正确
}
```

第9章 异常与断言

◆9.2 捕获顺序

- 先声明的异常处理过程将先得到执行机会，因此，可将需要先执行的异常处理过程放在前面。
- 异常的类型只要和catch参数的类型**相同或相容**即可匹配成功。即派生异常对象(及对象指针或引用)可以被带基类型对象、指针及引用参数的catch子句捕获。
  - 如果A的子类为B，B类异常也能被catch(A)、catch(const A)、 catch(volatile A)、catch(const volatile A) 等捕获，以及将上述A改为A&等的捕获
  - 如果A的子类为B，指向可写B类对象的指针异常也能被catch(A*)、 catch(const A*)、catch(volatile A*)、 catch(const volatile A*)等捕获。
- 如果产生了B类异常，且catch(const A &)在catch(const B &)之前，则catch(const A &)会捕获异常，从而使catch(const B &)没有捕获的机会。**因此，在排列catch时通常将catch父类参数的catch放在后面，否则子类(还包含指针或引用)参数的catch永远没有机会捕获异常。**
- 注意catch(const volatile void *)能捕获任意指针类型的异常，catch(…)能捕获任意类型的异常。

```
struct A{}; //没有定义任何构造函数，可用编译器提供的A()
struct B:A{}; //异常类需要描述异常的原因、发生现场、错误信息等
void  main(int argc, char* argv[])
{
   try   {   throw new B;  //注意这里抛出了异常对象的指针 }
   catch(A* x) //参数类型为指针
   {
      printf( “I will catch the exception”); //捕获异常
   }
   catch(B* x) //参数类型为指针
   {}     //无法捕获异常;
   catch(...) //表示可捕获任何异常
   {}     //无法捕获异常;
}
//正确的顺序应该为catch(B *x) {} catch(A *x) {} catch(…){}
//问题：假定throw (const B*) new B，哪一个catch将捕获异常?
```

第9章 异常与断言

定义VECTOR的INDEX和SHORTAGE异常处理过程

```
#include <stdio.h>
class VECTOR{
    int* data;
    int  size;
public:
    VECTOR(int n);
    int& getData(int i);                    //取下标所在位置的向量元素
    ~VECTOR( ) { delete[]data; };
};
class INDEX {
    int index;
public:
    INDEX(int i) { index = i; }
    int getIndex( )const { return index; }
};
```

第9章 异常与断言

```
struct SHORTAGE : INDEX {            // INDEX是父类，SHORTAGE为子类
   SHORTAGE(int i) : INDEX(i) { }
   using INDEX::getIndex;
};
VECTOR::VECTOR(int n)
{
   if (!(data = new int[size = n])) throw SHORTAGE(0);
};
int& VECTOR::getData(int i)
{
   if ((i < 0) || (i >= size)) throw INDEX(i);
   return data[i];
};
```

第9章 异常与断言

```
void main(void)
{
   VECTOR v(100);
   try { v.getData(101) = 30; }
   catch (const SHORTAGE&) {
      printf("SHORTAGE: Shortage of memory!\n");
   }
   catch (const INDEX & r) {
      printf("INDEX: Bad index is %d\n", r.getIndex( ));
   }
   catch (...) {
      printf("ANY: any error caught!\n");
   }
}
```

第9章 异常与断言

◆9.3 函数的异常接口

- 通过异常接口声明的异常都是由该函数引发的、而其自身又不想捕获或处理的异常。
- 异常接口定义的异常出现在函数的参数表后面，用throw列出要引发的异常类型

```
void func(void) throw(A, B, C);
void func(void) const throw(A, B, C); (成员函数)
void anycept(void)；                    //可引发任何异常
void nonexcept (void) throw( ); //不引发任何异常
void no_except (void) throw(void);//不引发任何异常
void notanyexcept (void) noexcep;//不引发任何异常
```

异常接口声明

可以定义异常接口声明来说明函数可能会抛出什么类型异常或不会抛出异常，这个异常可交给函数的调用者处理。

异常接口通常用throw(类型表达式列表）形式定义 上述类型表达式可以是任何类型，但不得使用省略参数形式。函数可以抛出同接口类型表达式相同和相容的类型如子类型。

可用throw()表示函数不引发任何异常。

（C++11将上述异常接口声明定义为过时的）。

如果参数表后面不出现异常接口，表示该函数可能引发任何类型的异常。

C++11标准中，在函数参数列表后面加noexcept用于标识该函数不会抛出异常

异常接口声明必须在函数声明和函数定义时都出现。

```
struct A{
        string msg = "Exception A";
};
struct B{
        string msg = "Exception B";
};

void f1();                  //可能会抛出异常
void f2() throw(A);         //可能会抛出异常类型A，过时的，警告错误
void f3() throw(A,B);       //可能会抛出异常类型A，B，过时的，警告错误
void f4() throw();          //不会抛出异常，过时的，警告错误
void f5() noexcept ;        //不会抛出异常，C++11新标准
```

```
void f1(){
        throw A(); //抛出A类型异常
}
void f2() throw(A){
        throw B(); //抛出非A类型异常
}
void f3() throw(A,B){
        int i;
        cin >> i;
        if(i> 0)  throw A();
        else      throw B();
}
void f4() throw(){
        throw B(); //声明不抛出异常，但抛出A类型异常
}

void f5() noexcept{
        throw B(); //声明不抛出异常，抛出A类型异常
}
```

```
void g(){
   try {
//      f1();  //没有声明异常接口的（即可能会抛出任何异常），异常会被截获
//      f2();  //抛出了和异常接口声明类型不一致的异常，异常不会被截获
//      f3();  //异常会被截获
//      f4();  //声明不会抛出异常，但实际抛出了异常，异常不会被截获
        f5();  //声明不会抛出异常，但实际抛出了异常，异常不会被截获
   }
   catch(A &ex){
      cout << ex.msg;
   }
   catch(B &ex){
      cout << ex.msg;
   }
}
int main() {
   g();
}
```

凡是引发了和函数异常声明接口异常不一致的异常，称为不可预料的异常。不可预料的异常不会被捕获，即使catch子句的异常类型与实际出现的异常类型一致。

任何没有被捕获的异常最终由标准库函数terminate来处理。

```
D:\CLionProjects\ExceptionDemo\bin\ExceptionDemo.exe
terminate called after throwing an instance of 'TEST1::B'

Process finished with exit code 3
```

第9章 异常与断言

◆9.3 函数的异常接口

- 异常接口不是函数原型的一部分，不能通过异常接口来定义和区分重载函数，故其不影响函数内联、重载、缺省和省略参数
- 不引发任何异常的函数引发的异常、引发了未说明的异常称为不可意料的异常。
- 通过set_unexpected过程，可以将不可意料的异常处理过程设置为程序自定义的不可意料的异常处理过程，其设置方法和通过set_terminate过程设置终止处理函数类似，设置后也返回一个指原先的不可意料的异常处理过程的指针。不同的编译提供的设置函数可能不同。
- 函数模板可定义异常接口,类模板及模板类的函数成员定义异常接口(构造函数和析构函数都可以定义异常接口)

terminate函数

对于未经处理的异常或传播后未经处理的异常，C++的监控系统将调用void terminate()处理。缺省情况下terminate函数调用abort函数来终止程序。

可以调用set_terminate()函数，设置自定义的terminate处理函数（terminate handler）。

set_terminate的原型为

void (*set_terminate(void (*)()))();

返回一个指向原来的terminate处理函数的指针。

void (*set_terminate(void (*)()))()声明了一个函数set_terminate，其参数为函数指针，该指针指向原型为void ()的函数。set_terminate也返回一个函数指针，该指针指向原型为void ()的函数。

```
void (*old_terminate)();
void new_terminate(){
    cout << "Custom terminate" << endl;
    abort();
}
void g(){
    try {
            f5();       //声明不会抛出异常，但实际抛出了异常，异常不会被截获
    }
    catch(A &ex){ cout << ex.msg;}
    catch(B &ex){ cout << ex.msg;}
}
int main() {
    old_terminate = set_terminate(new_terminate);
    g();
    //如果程序运行到这里，说明没有异常
    //恢复终止处理函数set_terminate (old_terminate);
    set_terminate (old_terminate);
}
```

```
D:\CLionProjects\ExceptionDemo\bin\ExceptionDemo.exe
Custom terminate

Process finished with exit code 3
```

terminate函数

对于和旧的函数异常接口声明不一致的异常，实际上先由void unexpected()函数处理，再由void terminate()来处理。

同样地，可以调用set_unexpected函数来设置自己的unexpected函数。

```
void (*old_terminate)();
void (*old_unexpected_handler)();
void new_terminate(){ cout << "Custom terminate" << endl; abort(); }
void new_unexpected_handler(){
   cout << "Custom unexpected handler" << endl;
//   abort();
}
void g(){
   try { f2(); }      //抛出了和异常接口声明类型不一致的异常，异常不会被截获
   catch(A &ex){ cout << ex.msg;}
   catch(B &ex){cout << ex.msg;
}
int main() {
   old_unexpected_handler = set_unexpected(new_unexpected_handler);
   old_terminate = set_terminate(new_terminate);
   g();
   //如果程序运行到这里，说明没有异常
   set_unexpected(old_unexpected_handler);
   set_terminate (old_terminate);
}
```

可以看到，用旧式异常接口声明的函数如果抛出了与声明不一致的异常，不会被捕获，而是先由先由void unexpected()函数处理，再由void terminate()来处理

```
D:\CLionProjects\ExceptionDemo\bin\ExceptionDemo.exe
Custom unexpected handler
Custom terminate
```

```
void (*old_terminate)();
void (*old_unexpected_handler)();
void new_terminate(){ cout << "Custom terminate" << endl; abort(); }
void new_unexpected_handler(){
   cout << "Custom unexpected handler" << endl;
//   abort();
}
void g(){
   try { f5(); }      //抛出了和异常接口声明类型不一致的异常，异常不会被截获
   catch(A &ex){ cout << ex.msg;}
   catch(B &ex){cout << ex.msg;
}
int main() {
   old_unexpected_handler = set_unexpected(new_unexpected_handler);
   old_terminate = set_terminate(new_terminate);

   g();

   //如果程序运行到这里，说明没有异常
   set_unexpected(old_unexpected_handler);
   set_terminate (old_terminate);
}
```

对于C++11定义的noexcept函数，如果抛出了异常，只会由terminate处理。

```
D:\CLionProjects\ExceptionDemo\bin\ExceptionDemo.exe
Custom terminate

Process finished with exit code 3
```

第9章 异常与断言

对数组a的若干相邻元素进行累加

```
#include <iostream>
using namespace std;
//以下函数sum()可不处理它发出的const char *类型的异常
int sum(int a[ ], int t, int s, int c) throw (const char *)  //参数t指明数组大小
{  //以下语句若发出const char *类型的异常，此后的语句不执行
   if (s < 0 || s >= t || s + c<0 || s + c>t)
      throw "subscription overflow";
   int r = 0,x=0;
   for (x = 0; x < c; x++)
      r += a[s+x];
   return r;
}
```

第9章 异常与断言

```
void main( )
{
   int m[6]={1,2,3,4,5,6};
   int r=0;
   try{
      r=sum(m, 6, 3, 4);//发出异常后try中所有语句都不执行，直接到其catch
      r=sum(m, 6, 1, 3);//不发出异常
   }
   //以下const去掉则不能捕获const char *类型的异常，只读指针实参不能传递给可写指针形参e
   catch(char *p){ cout<<p; } //不能捕获throw "subscription overflow";
   catch(const char *e){ //还能捕获char *类型的异常，可写指针实参可以传递给只读指针形参e
      cout<<e;
   }//由于throw时未分配内存，故在catch中无须使用delete  e
}
```

第9章 异常与断言

◆9.3 函数的异常接口

- noexcept可以表示throw()或throw(void)。
- noexcept一般用在移动构造函数，析构函数、移动赋值运算符函数等函数后面。
- 如果移动构造函数和移动赋值运算符还要申请资源，则难免发生异常，此时不应将noexcept放在这些函数的参数后面。
- 保留字noexcept和throw()可以出现在任何函数的后面，包括constexpr函数和Lambda表达式的参数表后面。但 throw(除void外的类型参数)不应出现在constexpr函数的参数表后面，并且constexpr函数也不能抛出异常，否则不能优化生成常量表达式。

第9章 异常与断言

◆9.4 异常类型

- C++提供了一个标准的异常类型exception，以作为标准类库引发的异常类型的基类，exception等异常由标准名字空间std提供。
- exception的函数成员不再引发任何异常。
- 函数成员what()返回一个只读字符串，该字符串的值没有被构造函数初始化，因此必须在派生类中重新定义函数成员what()。
- 异常类exception提供了处理异常的标准框架，应用程序自定义的异常对象应当自exception继承。
- 在catch有父子关系的多个异常对象时，应注意catch顺序。

第9章 异常与断言

◆9.4 异常对象的析构

- 如果是通过new产生的指针类型的异常，在catch处理过程捕获后，通常应使用合适的delete释放内存，否则可能造成内存泄漏。
- 如果继续传播指针类型的异常，则可以不使用delete。
- 从最内层被调函数抛出异常到外层调用函数的catch处理过程捕获异常，由此形成的函数调用链所有局部对象都会被自动析构，因此使用异常处理机制能在一定程度上防止内存泄漏。但是，调用链中的指针通过new创建的异常对象不会自动释放。

第9章 异常与断言

局部对象的析构过程

```
#include <exception>
#include <iostream>
using namespace std;
class EPISTLE : exception {              //定义异常对象的类型
public:
    EPISTLE(const char* s) :exception(s) { cout<<"Construct: " << s; }
    ~EPISTLE( )noexcept { cout << "Destruct: " << exception::what( ); };
    const char* what( )const throw( ) { return exception::what( ); };
};
void h( ) {
    EPISTLE h("I am in h( )\n");
    throw new EPISTLE("I have throw an exception\n");
}
```

第9章 异常与断言

```
void g( ) {   EPISTLE g("I am in g( )\n");  h( );  }
void f( ) {              EPISTLE f("I am in f( )\n");    g( );  }
void main(void) {
   try {    f( );   }
   catch (const EPISTLE * m) {
      cout << m->what( );
      delete m;
   }
}
```

main()->f()->g()->h()->h抛出异常(指针)->局部对象h、g、f依次析构->main捕获异常并delete

第9章 异常与断言

◆9.6 断言

- 函数assert(int)在assert.h中定义。
- 断言（assert）是一个带有整型参数的用于调试程序的函数，如果实参的值为真则程序继续执行。
- 否则，将输出断言表达式、断言所在代码文件名称以及断言所在程序的行号，然后调用abort()终止程序的执行。
- 断言输出的代码文件名称包含路径（编译时值），运行时程序拷到其它目录也还是按原有路径输出代码文件名称。assert()在运行时检查断言。
- 保留字static_assert定义的断言在编译时检查，为真时不终止编译运行。

第9章 异常与断言

断言的用法

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
class  SET {
   int* elem, used, card;
public:
   SET(int card);
   virtual int has(int)const;
   virtual SET& push (int);                    //插入一个元素
   virtual ~SET( ) noexcept { if (elem) { delete elem;  elem = 0; } };
};
SET::SET(int c) {
   card = c;
   elem = new int[c];
   assert(elem);                                        //当elem非空时继续执行
   used = 0;
}
```

第9章 异常与断言

```
int SET::has(int v)const {
    for (int k = 0; k < used; k++)        if (elem[k] == v) return 1;
    return 0;
}
SET& SET::push(int v) {
    assert(!has(v));                      //当集合中无元素v时继续执行
    assert(used < card);                  //当集合还能增加元素时继续执行
    elem[used++] = v;
    return *this;
}
void  main(void)
{
    static_assert(sizeof(int)==4);        //VS2019采用x86编译模式时为真，不终止编译运行
    SET  s(2);                            //定义集合只能存放两个元素
    s.push(1).push(2);;                   //存放第1，2个元素
    s.push(3);                            //因不能存放元素3，断言为假，程序被终止
}
```