chapter15 友元、异常和其他

1）友元

（1）友元类

类并非只能拥有友元函数，还可以将类作为友元，此时友元类的所有方法都可以访问原始类的私有和保护成员

例子：让Remote类成为Tv类的一个友元

```
// tv.h -- Tv and Remote classes
#ifndef TV_H_
#define TV_H_

class Tv
{
public:
    friend class Remote;   // Remote can access Tv private parts
    enum {Off, On};
    enum {MinVal,MaxVal = 20};
    enum {Antenna, Cable};
    enum {TV, DVD};

    Tv(int s = Off, int mc = 125) : state(s), volume(5),
        maxchannel(mc), channel(2), mode(Cable), input(TV) {}
    void onoff() {state = (state == On)? Off : On;}
    bool ison() const {return state == On;}
    bool volup();
    bool voldown();
    void chanup();
    void chandown();
    void set_mode() {mode = (mode == Antenna)? Cable : Antenna;}
    void set_input() {input = (input == TV)? DVD : TV;}
    void settings() const; // display all settings
private:
    int state;             // on or off
    int volume;            // assumed to be digitized
    int maxchannel;        // maximum number of channels
    int channel;           // current channel setting
    int mode;              // broadcast or cable
    int input;             // TV or DVD
};

class Remote
{
private:
    int mode;              // controls TV or DVD
public:
    Remote(int m = Tv::TV) : mode(m) {}
    bool volup(Tv & t) { return t.volup();}
    bool voldown(Tv & t) { return t.voldown();}
    void onoff(Tv & t) { t.onoff(); }
    void chanup(Tv & t) {t.chanup();}
    void chandown(Tv & t) {t.chandown();}
    void set_chan(Tv & t, int c) {t.channel = c;}
    void set_mode(Tv & t) {t.set_mode();}
    void set_input(Tv & t) {t.set_input();}
};
#endif
```

```cpp
// tv.cpp -- methods for the Tv class (Remote methods are inline)
#include <iostream>
#include "tv.h"

bool Tv::volup()
{
    if (volume < MaxVal)
    {
        volume++;
        return true;
    }
    else
        return false;
}

bool Tv::voldown()
{
    if (volume > MinVal)
    {
        volume--;
        return true;
    }
    else
        return false;
}

void Tv::chanup()
{
    if (channel < maxchannel)
        channel++;
    else
        channel = 1;
}

void Tv::chandown()
{
    if (channel > 1)
        channel--;
    else
        channel = maxchannel;
}

void Tv::settings() const
{
    using std::cout;
    using std::endl;
    cout << "TV is " << (state == Off? "Off" : "On") << endl;
    if (state == On)
    {
        cout << "Volume setting = " << volume << endl;
        cout << "Channel setting = " << channel << endl;
        cout << "Mode = "
            << (mode == Antenna? "antenna" : "cable") << endl;
        cout << "Input = "
            << (input == TV? "TV" : "DVD") << endl;
    }
}
```

友元声明可以位于公有、私有、保护部分，其所在位置无关紧要

（2）友元成员函数

可以选择仅让特定的类成员成为另一个类的友元，而不必让整个类成为友元，但是这样要注意各种声明和定义的顺序

例如，让 Remote() : : set_chan( ) 成为Tv类的友元的方法：在Tv类声明中将其声明为友元

```cpp
class Tv
{
        friend void Remote::set_chan(Tv& t,int c);
```

```
    ...
}
```

然而，要使编译器能够处理这条语句，它必须知道 Remote 的定义。否则，它无法知道 Remote 是一个类，而 set_chan 是这个类的方法。这意味着应将 Remote 的定义放到 Tv 的定义前面。Remote 的方法提到了 Tv 对象，而这意味着 Tv 定义应当位于 Remote 定义之前。避开这种循环依赖的方法是，使用前向声明（forward declaration）。为此，需要在 Remote 定义的前面插入下面的语句：

```
class Tv;              // forward declaration
```

这样，排列次序应如下：

```
class Tv;  // forward declaration
class Remote { ... };
class Tv { ... };
```

能否像下面这样排列呢？

```
class Remote;          // forward declaration
class Tv { ... };
class Remote { ... };
```

答案是不能。原因在于，在编译器在 Tv 类的声明中看到 Remote 的一个方法被声明为 Tv 类的友元之前，应该先看到 Remote 类的声明和 set_chan( )方法的声明。

还有一个麻烦。程序清单 15.1 的 Remote 声明包含了内联代码，例如：

```
void onoff(Tv & t) { t.onoff(); }
```

由于这将调用 Tv 的一个方法，所以编译器此时必须已经看到了 Tv 类的声明，这样才能知道 Tv 有哪些方法，但正如看到的，该声明位于 Remote 声明的后面。这种问题的解决方法是，使 Remote 声明中只包含方法声明，并将实际的定义放在 Tv 类之后。这样，排列顺序将如下：

```
class Tv;              // forward declaration
class Remote { ... };     // Tv-using methods as prototypes only
class Tv { ... };
// put Remote method definitions here
```

Remote 方法的原型与下面类似：

```
void onoff(Tv & t);
```

检查该原型时，所有的编译器都需要知道 Tv 是一个类，而前向声明提供了这样的信息。当编译器到达真正的方法定义时，它已经读取了 Tv 类的声明，并拥有了编译这些方法所需的信息。通过在方法定义中使用 inline 关键字，仍然可以使其成为内联方法。程序清单 15.4 列出了修订后的头文件。

（3）共同的友元

需要使用友元的另一种情况是，函数需要访问两个类的私有数据。从逻辑上看，这样的函数应是每个类的成员函数，但这是不可能的。它可以是一个类的成员，同时是另一个类的友元，但有时将函数作为两个类的友元更合理。例如，假定有一个 Probe 类和一个 Analyzer 类，前者表示某种可编程的测量设备，后者表示某种可编程的分析设备。这两个类都有内部时钟，且希望它们能够同步，则应该包含下述代码行：

```
class Analyzer;  // forward declaration
class Probe
{
    friend void sync(Analyzer & a, const Probe & p);  // sync a to p
    friend void sync(Probe & p, const Analyzer & a);  // sync p to a
    ...
};
class Analyzer
{
    friend void sync(Analyzer & a, const Probe & p);  // sync a to p
    friend void sync(Probe & p, const Analyzer & a);  // sync p to a
    ...
};
```

```
// define the friend functions
inline void sync(Analyzer & a, const Probe & p)
{
    ...
}
inline void sync(Probe & p, const Analyzer & a)
{
    ...
}
```

2）嵌套类

略

3）异常

【1】调用abort( )

（1）函数原型位于< cstdlib >中
（2）典型实现：向标准错误流发送一个消息“程序异常终止”(abnormal program termination)，然后终止程序。它还返回一个随实现而异的值，告诉操作系统处理失败

```
//error1.cpp -- using the abort() function
#include <iostream>
#include <cstdlib>
double hmean(double a, double b);
int main()
{
    double x, y, z;

    std::cout << "Enter two numbers: ";
    while (std::cin >> x >> y)
    {
        z = hmean(x,y);
        std::cout << "Harmonic mean of " << x << " and " << y
            << " is " << z << std::endl;
```

```
        std::cout << "Enter next set of numbers <q to quit>: ";
    }
    std::cout << "Bye!\n";
    return 0;
}

double hmean(double a, double b)
{
    if (a == -b)
    {
        std::cout << "untenable arguments to hmean()\n";
        std::abort();
    }
    return 2.0 * a * b / (a + b);
}

result:
Enter two numbers: 1 -1
untenable arguments to hmean()
```

ps：在 hmean( ) 程序中调用 abort( ) 函数将直接终止程序，而不是先返回到 main( )

【2】返回错误码

（1）ostream类的 get(void) 成员通常返回下一个输入字符的 ASCII码；但是到达文件尾时，将返回特殊值EOF

（2）多加一个参数：使用指针参数or引用指针来将值返回给调用程序，并使用函数的返回值来指出是成功还是失败

```
//error2.cpp -- returning an error code
#include <iostream>
#include <cfloat>  // (or float.h) for DBL_MAX
bool hmean(double a, double b, double * ans);
int main()
{
    double x, y, z;

    std::cout << "Enter two numbers: ";
    while (std::cin >> x >> y)
    {
        if (hmean(x,y,&z))
            std::cout << "Harmonic mean of " << x << " and " << y
                << " is " << z << std::endl;
        else
            std::cout << "One value should not be the negative "
                << "of the other - try again.\n";
        std::cout << "Enter next set of numbers <q to quit>: ";
    }
    std::cout << "Bye!\n";
    return 0;
}

bool hmean(double a, double b, double * ans)
{
    if (a == -b)
    {
        *ans = DBL_MAX;
        return false;
    }
    else
    {
        *ans = 2.0 * a * b / (a + b);
        return true;
    }
}
```

ps：第三参数可以是指针或引用。一般来说对内置的参数，倾向于采用指针，因为这样可以明显指出哪个参数用于提供答案

【3】异常机制

（1）组成：

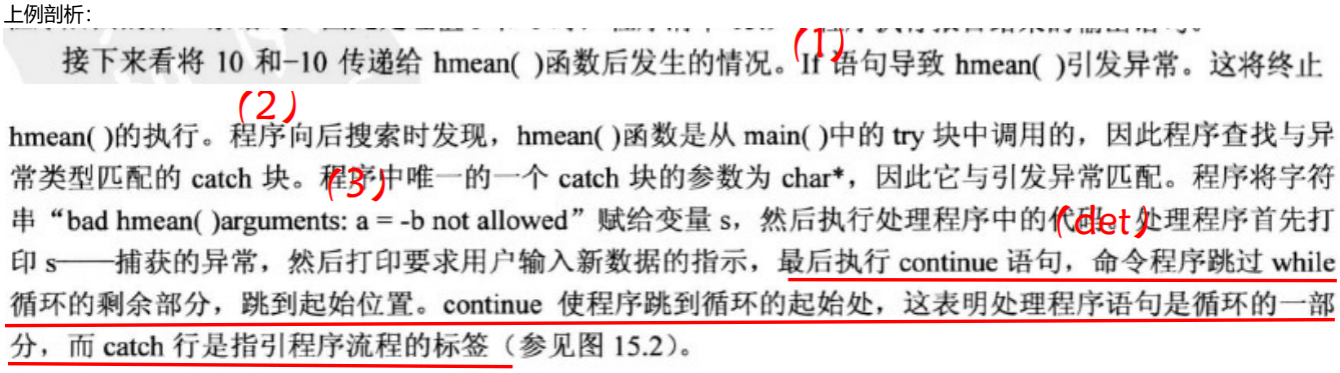
> [1]throw语句：实际上是跳转，命令程序转到另一句语句。**throw关键字**表示引发异常，随后的值(str或object)指出来异常的“人定特征”
> [2]catch模块：程序使用异常处理模块(exception handler)来捕获异常。**catch关键字**指出：当异常被引发时，程序应跳到这个位置执行
> [3]try模块：它内部存有*异常可能会被激活的代码块*，它后面跟着一个或多个catch块

（2）程序说明：

```
// error3.cpp -- using an exception
#include <iostream>
double hmean(double a, double b);

int main()
{
    double x, y, z;
    std::cout << "Enter two numbers: ";
    while (std::cin >> x >> y)
    {
        try {                   // start of try block
            z = hmean(x,y);
        }                       // end of try block

        catch (const char * s)                          ----------------------------这里 const char* 与 “人定特征”string吻合
        {// start of exception handler
            std::cout << s << std::endl;
            std::cout << "Enter a new pair of numbers: ";
            continue;
        }                       // end of handler
        std::cout << "Harmonic mean of " << x << " and " << y
            << " is " << z << std::endl;
        std::cout << "Enter next set of numbers <q to quit>: ";
    }
    std::cout << "Bye!\n";
    return 0;
}

double hmean(double a, double b)
{
    if (a == -b)
        throw "bad hmean() arguments: a = -b not allowed"; ----------------------------碰到异常时将“人定特征”设置为这个string语句
    return 2.0 * a * b / (a + b);
}
```

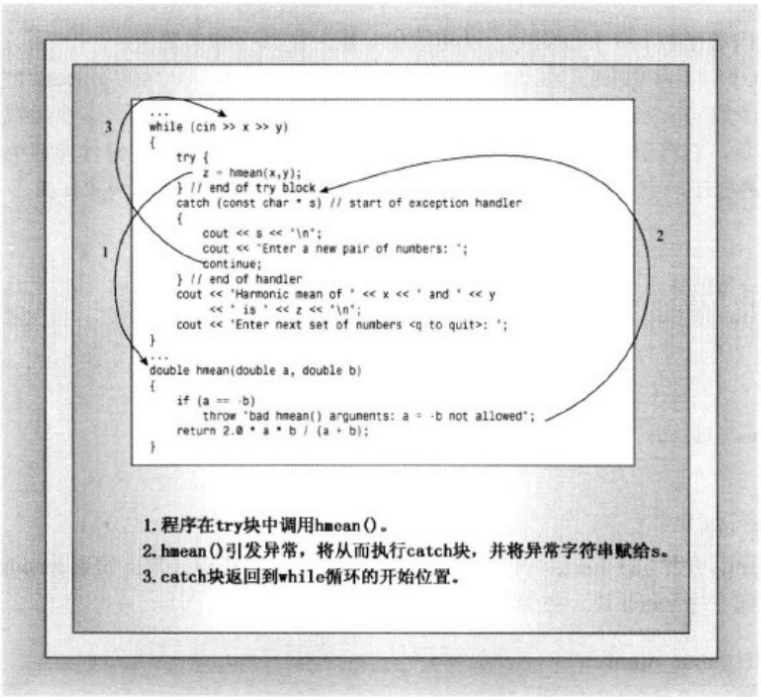
程序执行过程分析：
【0】程序按计划进行ing
【1】如果其中某条语句导致异常被引发，则会执行它的throw语句，这类似于执行返回语句，因为它也将终止函数的执行。但throw不是将控制权返回给调用程序，而是导致程序向函数调用的序列后退，直到找到包含try的函数
【2】找到包含它的try模块后，程序将寻找*与引发的异常类型匹配的异常处理程序*（这取决于上面throw的“人定特征”）
【3】现在已找到匹配的catch模块（当异常与该处理程序匹配时，程序将执行括号中的代码）
ps：执行完try块中的语句后，如果没有引发任何异常，则程序跳过try后面的catch块，直接执行处理程序后的第一条语句

上例剖析：



接下来看将 10 和−10 传递给 hmean( )函数后发生的情况。If 语句导致 hmean( )引发异常。这将终止 hmean( )的执行。程序向后搜索时发现，hmean( )函数是从 main( )中的 try 块中调用的，因此程序查找与异常类型匹配的 catch 块。程序中唯一的一个 catch 块的参数为 char*，因此它与引发异常匹配。程序将字符串“bad hmean( )arguments: a = -b not allowed”赋给变量 s，然后执行处理程序中的代码。处理程序首先打印 s——捕获的异常，然后打印要求用户输入新数据的指示，最后执行 continue 语句，命令程序跳过 while 循环的剩余部分，跳到起始位置。continue 使程序跳到循环的起始处，这表明处理程序语句是循环的一部分，而 catch 行是指引程序流程的标签（参见图 15.2）。

图示解析原理：



（3）将类对象作为异常类型

引发异常的函数将传递一个对象，如此的优点是：可以使用不同的类型来区分不同的函数在不同的情况下引发的异常

```
class bad_hmean
{
private:
    double v1;
    double v2;
public:
    bad_hmean(int a = 0, int b = 0) : v1(a), v2(b){}
    void mesg();
};

inline void bad_hmean::mesg()
{
```

```
        std::cout << "hmean(" << v1 << ", " << v2 <<"): "
                  << "invalid arguments: a = -b\n";
    }
```

可以将一个 bad_hmean 对象初始化为传递给函数 hmean( )的值，而方法 mesg( )可用于报告问题（包括传递给函数 hmena( )的值）。函数 hmean( )可以使用下面这样的代码：

```
if (a == -b)
    throw bad_hmean(a,b);
```

上述代码调用构造函数 bad_hmean( )，以初始化对象，使其存储参数值。

程序清单 15.10 和 15.11 添加了另一个异常类 bad_gmean 以及另一个名为 gmean( )的函数，该函数引发 bad_gmean 异常。函数 gmean( )计算两个数的几何平均值，即乘积的平方根。这个函数要求两个参数都不为负，如果参数为负，它将引发异常。程序清单 15.10 是一个头文件，其中包含异常类的定义；而程序清单 15.11 是一个示例程序，它使用了该头文件。注意，try 块的后面跟着两个 catch 块：

```
try {                    // start of try block
    ...
}// end of try block
catch (bad_hmean & bg)     // start of catch block
{
    ...
}
catch (bad_gmean & hg)
{
    ...
} // end of catch block
```

如果函数 hmean( )引发 bad_hmean 异常，第一个 catch 块将捕获该异常；如果 gmean( )引发 bad_gmean 异常，异常将逃过第一个 catch 块，被第二个 catch 块捕获。

可以类比switch的并列调用！

举例说明：

```
// exc_mean.h  -- exception classes for hmean(), gmean()
#include <iostream>
class bad_hmean
{
private:
    double v1;
    double v2;
public:
    bad_hmean(double a = 0, double b = 0) : v1(a), v2(b){}
    void mesg();
};

inline void bad_hmean::mesg()
{
    std::cout << "hmean(" << v1 << ", " << v2 <<"): "
              << "invalid arguments: a = -b\n";
}

class bad_gmean
{
public:
    double v1;
    double v2;
    bad_gmean(double a = 0, double b = 0) : v1(a), v2(b){}
    const char * mesg();
};

inline const char * bad_gmean::mesg()
{
    return "gmean() arguments should be >= 0\n";
}
```

```cpp
//error4.cpp 
#include <iostream>
#include <cmath> // or math.h, unix users may need -lm flag
#include "exc_mean.h"
// function prototypes
double hmean(double a, double b);
double gmean(double a, double b);
int main()
{
    using std::cout;
    using std::cin;
    using std::endl;

    double x, y, z;
    cout << "Enter two numbers: ";

    while (cin >> x >> y)
    {
        try {                  // start of try block
            z = hmean(x,y);
            cout << "Harmonic mean of " << x << " and " << y
                << " is " << z << endl;
            cout << "Geometric mean of " << x << " and " << y
                << " is " << gmean(x,y) << endl;
            cout << "Enter next set of numbers <q to quit>: ";
        }// end of try block

        catch (bad_hmean & bg)    // start of catch block
        {
            bg.mesg();
            cout << "Try again.\n";
            continue;
        }

        catch (bad_gmean & hg)
        {
            cout << hg.mesg();
            cout << "Values used: " << hg.v1 << ", "
                 << hg.v2 << endl;
            cout << "Sorry, you don't get to play any more.\n";
            break;
        } // end of catch block
    }
    cout << "Bye!\n";
    // cin.get();
    // cin.get();
    return 0;
}

double hmean(double a, double b)
{
    if (a == -b)
        throw bad_hmean(a,b);
    return 2.0 * a * b / (a + b);
}

double gmean(double a, double b)
{
    if (a < 0 || b < 0)
        throw bad_gmean(a,b);
    return std::sqrt(a * b);
}
```

【4】栈解退

(1) 普通函数调用的栈过程（函数栈）：

首先来看一看 C++通常是如何处理函数调用和返回的。C++通常通过将信息放在栈（参见第 9 章）中来处理函数调用。具体地说，程序将调用函数的指令的地址（返回地址）放到栈中。当被调用的函数执行完毕后，程序将使用该地址来确定从哪里开始继续执行。另外，函数调用将函数参数放到栈中。在栈中，这些函数参数被视为自动变量。如果被调用的函数创建了新的自动变量，则这些变量也将被添加到栈中。如果被调用的函数调用了另一个函数，则后者的信息将被添加到栈中，依此类推。当函数结束时，程序流程将跳到该函数被调用时存储的地址处，同时栈顶的元素被释放。因此，函数通常都返回到调用它的函数，依此类推，同时每个函数都在结束时释放其自动变量。如果自动变量是类对象，则类的析构函数（如果有的话）将被调用。

(2) throw-catch处理机制原理：

现在假设函数由于出现异常（而不是由于返回）而终止，则程序也将释放栈中的内存，但不会在释放栈的第一个返回地址后停止，而是继续释放栈，直到找到一个位于 try 块（参见图 15.3）中的返回地址。随后，控制权将转到块尾的异常处理程序，而不是函数调用后面的第一条语句。这个过程被称为栈解退。引发机制的一个非常重要的特性是，和函数返回一样，对于栈中的自动类对象，类的析构函数将被调用。然而，函数返回仅仅处理该函数放在栈中的对象，而 throw 语句则处理 try 块和 throw 之间整个函数调用序列放在栈中的对象。如果没有栈解退这种特性，则引发异常后，对于中间函数调用放在栈中的自动类对象，其析构函数将不会被调用。

图解：(非常直观，强烈推荐！)

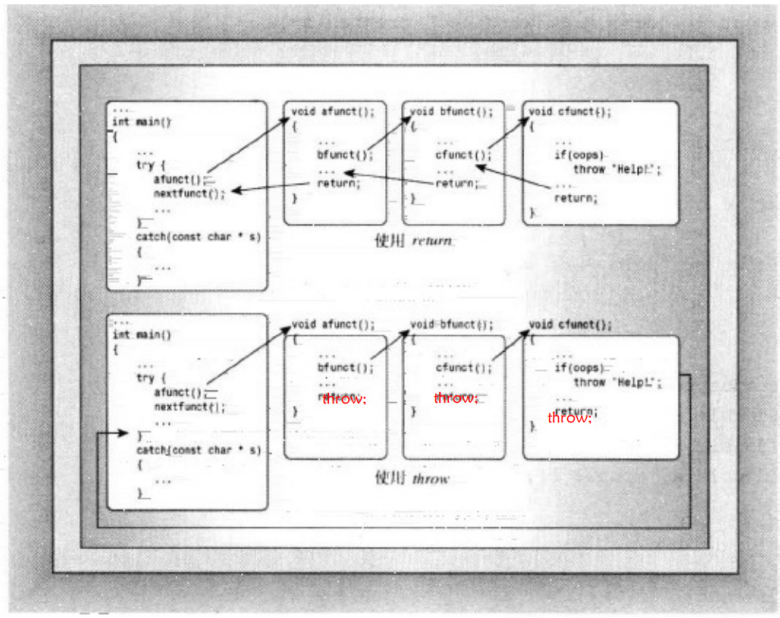


图 15.3 throw 与 return

【6】其他异常特性

> throw-catch机制类似于函数参数与函数返回的机制，但还是有区别：

（1）函数 func( ) 中的返回语句将控制权返回给调用 func( ) 的函数，但是throw语句（最终）将控制权向上返回到第一个这样的函数：包含能够捕获相应异常的try-catch组合

（2）引发异常时编译器总会创建一个临时拷贝，即使异常规范和catch模块中指定的是引用

```
class problem{......};
...
void super() throw(problem)
{
        ...
        if(oh_baby)
        {
                problem oops;
                throw oops;    // 这两句可以直接用：throw problem(); 来代替！
```

```
        ...
}
...

try {
        super();
}

catch(problem& p)
{
        ...
}
```

（1）上述的p将指向oops的副本而不是oops本身！这很好，因为函数super( ) 执行完毕后oops将不复存在！

（2）思考：既然throw语句生成的是oops的副本，为什么代码中要使用引用呢？
通常，将引用作为返回值的原因是：避免创建副本以提高效率
但是在这里：
引用有一个重要的特征：基类引用可以执行派生类的对象。假设有一组通过继承关联起来的异常类型，则在异常类型中只需列出一个基类引用，它将可以与任何派生类对象匹配！

```
class bad_1 {...};
class bad_2 : public bad_1 {...};
class bad_3 : public bad_2 {...};
...

void duper()
{
        ...
        if(oh_no) throw bad_1();
        if(rats) throw bad_2();
        if(drat) throw bad_3();
        ...
}
...

try {
        duper();
}

catch(bad_3& be) {......}
catch(bad_2& ce) {......}
catch(bad_1& de) {......}
```

---

ps1：这里的使用要注意排列顺序！！！
【1】如果将bad_1&的catch模块放在最前面，它将捕获bad_1，bad_2 和 bad_3；
【2】通过按相反的顺序排列，bad_3异常将被bad_3&处理程序所捕获（各司其职！）

> tips：如果有一个异常类继承层次结构，应这样排列catch模块：
> 将 *捕获层次结构最下面的异常类* 的catch语句放在最前面，将*捕获基类异常*的catch语句放在最后面

ps2:：对于catch模块，使用省略号来表示异常类型，从而捕获任何异常！！！
格式：

```
catch(...) { //statements }
```

例子：

```
try {
        duper();
}
```

```
catch(bad_3& be) { //statements }
catch(bad_2& ce) { //statements }
catch(bad_1& de) { //statements }
catch(...) { //statements }       // catch whatever is left!
```

【7】exception类

c++定义了很多基于exception的异常类型
（1）stdexcept异常类：

> 类别一：logic_error（典型的逻辑错误）
> [1] domain_error：定义域错误
> [2] invalid_argument：函数传递了一个意料之外的值
> [3] length_error：没有足够的空间来执行所需的操作
> [4] out_of_bounds：指示索引错误

> 类别二：runtime_error（运行时错误）
> [1] range_error：计算结果没有上溢或下溢，但是不在函数允许的范围内
> [2] overflow_error：上溢错误（超出maximum）
> [3] underflow_error：下溢错误（小于minimum）

（2）bad_alloc异常 和 new：
当无法分配请求的内存量时，new返回一个空指针
ps：如果程序在系统上运行时没有出现内存分配的问题，可以尝试提高请求分配的内存量

（3）空指针 和 new：
c++标准提供了一种在失败时返回空指针的new，形式如下：

```
int* pi = new (std::nothrow) int;
int* pa = new (std::nothrow) int[666];
```

4）RTTI

RTTI：运行阶段类型识别 (RunTime Type Identification)

（1）用途：

在回答这个问题之前，先考虑为何要知道类型。可能希望调用类方法的正确版本，在这种情况下，只要该函数是类层次结构中所有成员都拥有的虚函数，则并不真正需要知道对象的类型。但派生对象可能包含不是继承而来的方法，在这种情况下，只有某些类型的对象可以使用该方法。也可能是出于调试目的，想跟踪生成的对象的类型。对于后两种情况，RTTI 提供解决方案。

（2）工作原理：
【1】dynamic_cast运算符：

> 如果可能的话，dynamic_cast运算符将使用*一个指向基类的指针*来生成*一个指向派生类的指针*

语法：

```
dynamic_cast<Type*> (pt);

如果指向的对象(*pt)的类型为 Type 或者是 从Type直接/间接派生而来的类型 ，则将指针pt转换为Type类型的指针----------【指针回溯】
否则：结果是0，即空指针
```

【2】typeid运算符：

> 返回一个指出对象的类型的值

【3】type_info结构：

> 存储了有关特定类型的信息

RTTI：只适用于包含虚函数的类

5）类型转换运算符

严格限制允许的类型转换，并添加4个类型转换符，使得转换过程更规范
【1】dynamic_cast：“指针回溯”
【2】const_cast：“改变值为 const 或 volatile（不稳定的）的类型”
【3】static_cast：
【4】reinterpret_cast：

基本用不到，用到时来这里查即可（书P526~P528）