异常处理

例子：

```
void f(){
   sock a;
   a=sock("10.100.21.61",5670);
   send(a, "Hello! ");
}


int f(){
   sock a;
   a=sock("10.100.21.61",5670);
   if(a<0){
      cout<<"不能获得网络连接"<<endl;
      return(-1);
   }
   send(a, "Hello! ");
}
int main(){if(f() == -1)  cout<<"失败! ";return 0; }
```

首先，我们来看第一个函数`void f()`。这个函数声明了一个`sock`类型的变量`a`，然后尝试使用IP地址`"10.100.21.61"`和端口号`5670`来创建一个网络连接，并将这个连接赋值给变量`a`。接着，它尝试通过这个连接发送字符串`"Hello! "`。但是，这个函数有一些问题：

1. 它没有检查`sock()`函数是否成功创建了连接。如果连接失败，后续的`send()`函数可能会出错。
2. 函数的返回类型是`void`，这意味着它不能返回任何值。如果我们需要知道函数是否执行成功，这种返回类型就不够用了。

接下来，我们看第二个函数`int f()`。这个函数与第一个函数类似，但有一些重要的改进：

1. 函数的返回类型改为了`int`，这意味着它可以返回一个整数值，通常用于表示函数执行的状态。
2. 在调用`sock()`函数之后，它检查变量`a`是否小于`0`。如果是，说明网络连接没有成功建立，于是它输出一条错误信息`"不能获得网络连接"`，并返回`-1`。
3. 如果网络连接成功建立，它会像第一个函数一样发送字符串`"Hello! "`，但这里并没有显示地返回一个值。为了代码的健壮性，最好在函数的最后添加一个`return 0;`来表示成功。

最后，我们来看`main()`函数。这个函数调用了`f()`函数，并检查其返回值是否为`-1`。如果是，它输出`"失败! "`。

但是，这段代码还有一些可以改进的地方：

1. `sock`类型、`sock()`函数、`send()`函数等都没有在代码中定义，我们假设它们是在其他地方定义的。
2. 在`int f()`函数的最后，最好显式地返回一个表示成功的值，如`return 0;`。

另外，你需要注意的是，这段代码可能是伪代码或者是一个简化的示例，因为它缺少了很多实际编程中需要的细节，比如错误处理、资源管理等。

需要异常处理的原因

1

```
class Student{
    int id;
    string *p;
public:
    Student(){ p = new string[5000];}//如果构造函数出错，返回途径不好处理
    ~Student(){delete [] p;}
};
void main(){
    Student a;
}
```

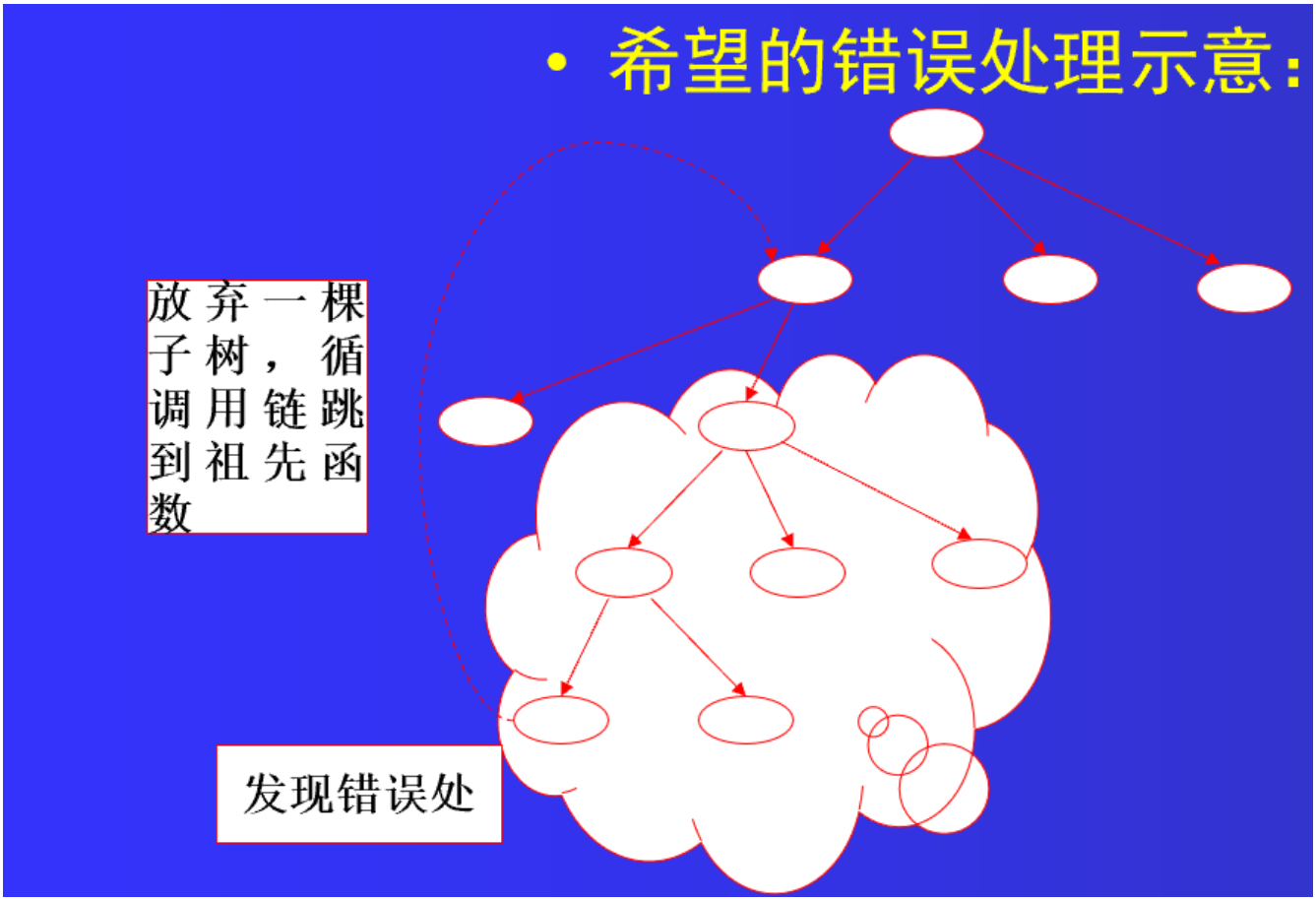
注释里的话“如果构造函数出错，返回途径不好处理”指的是，在`Student`类的构造函数中，如果`new string[5000]`这个动态内存分配操作失败了（例如，由于系统内存不足），那么就会抛出一个`std::bad_alloc`异常。然而，构造函数没有返回值（其返回类型是`void`），所以它不能通过返回一个错误代码来表示出现了错误。

2

传统的错误处理方式：

1 遇到错误，终止运行，**低级粗暴** 2 遇到错误，调用返回给上层函数错误信息，**忽略了模块体系 实现代价大** 3 遇到错误，改变全局错误变量的值，并函数返回，**破坏了程序结构** 4 遇到错误，调用事先设计好的下层错误处理函数，**可惜错误往往不是自己所能解决的了的**！

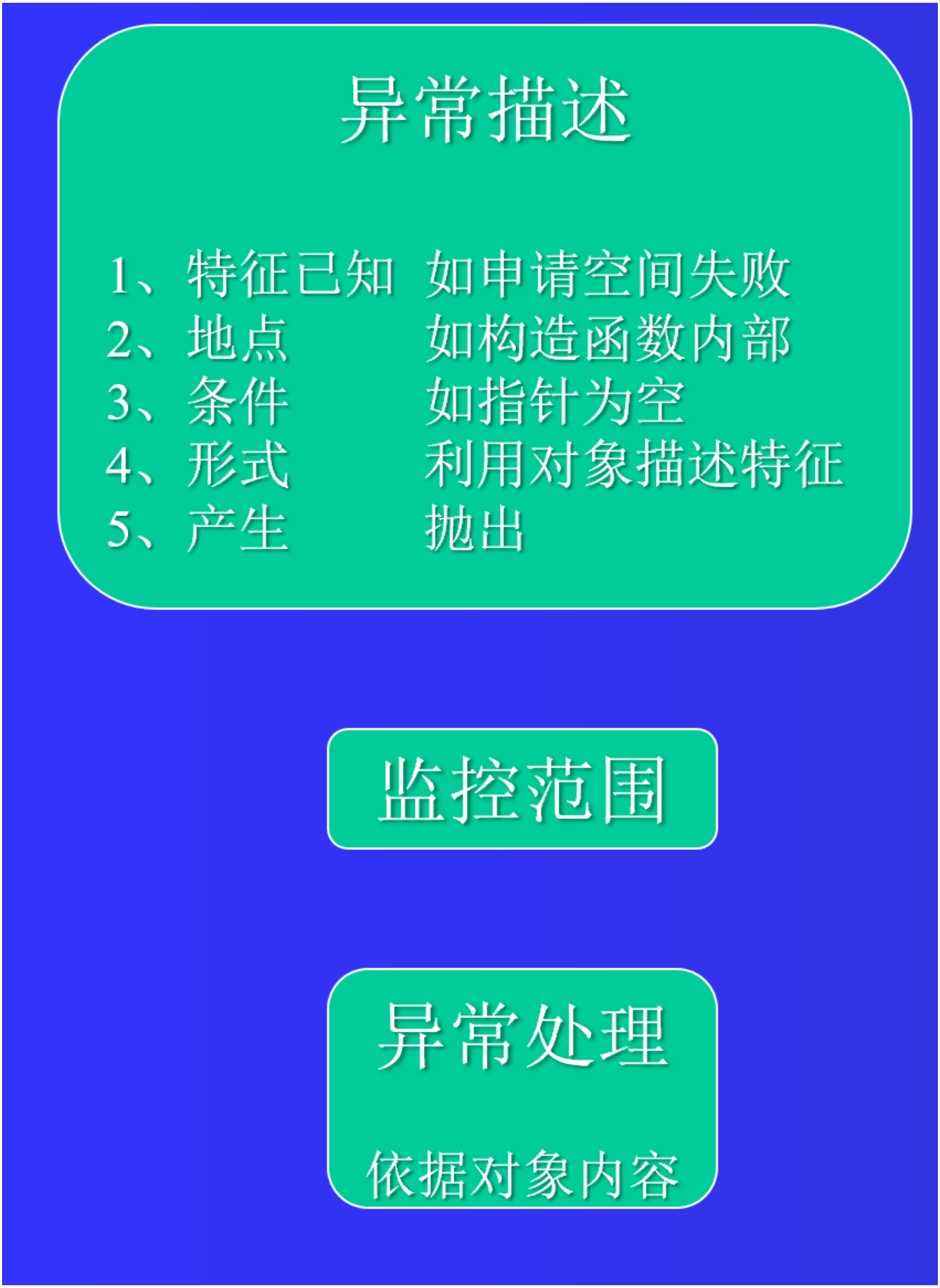
希望的错误处理方式：



函数机制，本质上是一种过程控制机制。

对面向对象程序来说，进行从发现错误到处理错误的设计，是一个超出过程控制能力的庞大控制体系。这也体现了面向对象程序的优势。

异常处理的几个要素、概念



首先，这段描述似乎是在讲解关于异常处理的概念和步骤。异常处理是一种编程技术，用于在程序运行时检测和处理错误情况。

以下是这些步骤的详细解释：

1. **特征已知**：这是指异常的特征或表现已经被明确识别。在这个例子中，提到了“如申请空间失败”，这表示当尝试分配内存空间但失败时，可能会产生一个异常。
2. **地点**：这是指异常发生的位置或上下文。在这个例子中，“如构造函数内部”表示异常可能在某个类的构造函数中发生。
3. **条件**：这是指导致异常发生的具体条件。在这个例子中，“如指针为空”表示当尝试访问或操作一个空指针时，可能会发生异常。
4. **形式**：这描述了如何描述或报告异常。在这个例子中，“利用对象描述特征”可能意味着通过创建一个异常对象来捕获和描述异常的详细信息。这样，当异常被抛出时，调用者可以访问这个对象以获取关于异常的更多信息。
5. **产生** 和 **抛出**：这两个词通常一起使用，表示异常的产生和抛出过程。当满足某个条件（如上面提到的空指针访问）时，程序会生成一个异常对象，并通过throw关键字将其抛出。这会导致程序控制流从当前位置中断，并跳转到相应的异常处理代码（如catch块）。

监控范围：这可能指的是异常处理的监控范围或作用域。在异常处理中，我们通常使用try-catch块来监控可能引发异常的代码区域。如果在try块中的代码引发了异常，并且该异常与catch块中指定的类型匹配，则控制流将跳转到该catch块的开头，并执行其中的代码来处理异常。

异常处理 和 **依据对象内容**：这描述了如何处理异常。在catch块中，我们可以访问并检查异常对象的内容，以确定如何处理异常。这可以包括记录错误、恢复程序状态、通知用户或执行其他适当的操作。

总的来说，这段描述为我们提供了一个关于异常处理的高层次概述，包括异常的识别、发生位置、触发条件、描述方式、产生和抛出过程，以及如何处理异常。

异常处理示例

1

```
#define MAX  10
int f(int * a,int j){
   if(j>=0&&j<MAX)
        return(a[j]);
     else
        throw  A();//设A已经定义，A为异常特征报告
}
void main(){
    int j;
    int a[MAX]={0};
    try{
       cout<<f(a,15)<<endl;
    }
    catch(A & s){
      cout<<s<<endl;//设A已经重载<<
    }

//某种疾病的处理：已知病毒；检测病毒；处理病毒
```

首先，我们先看一下这段代码的基本结构和内容：

1. **宏定义**：

```
#define MAX  10
```

这里定义了一个宏MAX，值为10。这通常用于限制数组的大小或其他需要固定常数值的地方。

2. **函数f**：

```
int f(int * a,int j){
   if(j>=0&&j<MAX)
        return(a[j]);
    else
        throw  A(); // 这里假设A是一个异常类，且A的构造函数已经定义
}
```

函数f接收一个整数指针a和一个整数j作为参数。如果j的值在0到MAX-1（即0到9）之间，函数返回a[j]的值；否则，函数抛出一个类型为A的异常。

3. **主函数main**：

```
void main(){ // 注意：在C++中，main函数应该返回int类型，即int main()
    int j;
    int a[MAX]={0}; // 初始化一个大小为MAX的整数数组，所有元素为0
    try{
       cout<<f(a,15)<<endl; // 调用f函数，传入数组a和索引15
    }
    catch(A & s){ // 捕获类型为A的异常
      cout<<s<<endl; // 输出异常对象的信息，这里假设A类已经重载了<<操作符
    }
}
```

主函数main首先定义了一个整数j（虽然在这个例子中它没有被使用），然后定义并初始化了一个大小为MAX的整数数组a。接着，它尝试调用函数f并输出返回的结果。但是，因为f函数在接收到j=15时会抛出一个异常（因为j的值不在0到9之间），所以程序会跳转到catch块来处理这个异常。

现在，我们来看一下“某种疾病的处理：已知病毒；检测病毒；处理病毒”这个比喻：

- **已知病毒**：在这个例子中，已知的“病毒”就是索引值可能超出数组界限的潜在风险。
- **检测病毒**：函数f通过检查j的值是否在0到MAX-1之间来“检测”这个“病毒”。
- **处理病毒**：如果检测到“病毒”（即j的值不在有效范围内），函数f会抛出一个异常来“处理”它。这个异常被主函数中的try-catch块捕获，并输出相关的异常信息。

通过这种方式，程序能够在遇到潜在问题时及时地发现并处理它们，从而避免了可能的运行时错误或未定义行为。

2

```
class Student{
    int id;
    string *p;
public:
```

```
    Student(){
        p = new string[5000];
        if(!p)
           throw string("构造错误");}//检查是否成功分配了内存。如果p是nullptr（即内存分配失败），则抛出一个类型为string的异常，异常信息为“构造错误”。
    ~Student(){delete [] p;}
};
void main(){
    try{
        Student a;
    }
    catch(string &s){
        cout<<s.c_str()<<endl;
    }
}
```

总结1

1.Throw是情报密探；Try是监控某块的值班。

2.Catch处理异常，可有一个以上，只捕获try标记的。

3.Catch只能容纳一个形参，依靠类型匹配捕获。

4.可以跨层次。

情报密探（Throw）

- **情报密探（Throw）**：在程序中，当某个操作无法继续执行或出现了某种错误时，就会“抛出”一个异常。这个“抛出”异常的行为，就像情报密探发现了重要情报一样，需要立即报告给上级。
- 情报密探会携带一份详细的报告（异常信息），这个报告包含了发生错误的位置、错误的原因以及错误的详细描述等信息。

值班监控（Try）

- **值班监控（Try）**：try 块是程序的监控区域，类似于一个保安监控室，它负责监控代码块内的操作是否正常运行。当 try 块内的代码执行时，值班监控会密切关注是否有异常情况发生。
- 如果没有异常发生，try 块内的代码会正常执行完毕。但是，如果情报密探（throw）抛出了异常，值班监控就会立即捕获到这个异常，并寻找能够处理这个异常的“部门”（catch 块）。

异常处理部门（Catch）

- **异常处理部门（Catch）**：catch 块就是异常处理部门，它负责接收并处理 try 块中抛出的异常。每个 catch 块都可以指定它能够处理的异常类型。
- 当 try 块中抛出了异常时，值班监控（try 块）会寻找能够处理这个异常类型的 catch 块。如果找到了匹配的 catch 块，就会将异常信息传递给这个 catch 块进行处理。
- catch 块只能容纳一个形参，这个形参就是异常对象本身。通过异常对象的类型，catch 块可以判断是哪个类型的异常，并采取相应的处理措施。
- 一个 try 块后面可以跟随多个 catch 块，每个 catch 块处理不同类型的异常。当异常被某个 catch 块捕获后，后续的 catch 块将不再执行。

跨层次处理

- **跨层次处理**：异常处理机制是支持跨层次的。也就是说，如果一个 `try` 块没有匹配的 `catch` 块来处理异常，那么异常会被传递给上一层的调用者，由上一层的 `try-catch` 结构来处理。这个过程会一直持续到找到能够处理该异常的 `catch` 块为止，或者直到程序的最顶层都没有找到匹配的 `catch` 块，此时程序会终止执行。
- **示例1：**

在这个例子中：

- `functionC` 直接抛出一个 `std::runtime_error` 异常。
- `functionB` 调用 `functionC` 但没有自己的 `try-catch` 块，所以 `functionC` 抛出的异常会传递到 `functionB` 的调用者。
- `functionA` 调用 `functionB`，并且有一个 `try-catch` 块来捕获 `std::runtime_error` 类型的异常。当 `functionC` 的异常被传递到 `functionA` 时，它会被这个 `catch` 块捕获并处理。
- `main` 函数调用 `functionA`，并且也有一个 `try-catch` 块，但它用于捕获所有继承自 `std::exception` 的异常。然而，在这个例子中，由于 `functionA` 已经捕获了异常，所以 `main` 函数中的 `catch` 块不会被执行。

- **示例2：**

```
#include <iostream>
using namespace std;
void f0() { //..;
    throw string("请处理\n");
    //…;
}
void f1(){ //..;
    f0();
    //…;
}
void f2() { //..;
    f1();
    //…;
}
void f3() { //..;
    f2();
    // …;
}
void f4() { //..;
    f3();
    // …;
}
void f5() { //..;
    f4();
    //…;
}
int main() {
    try {// ..;
        f5();
        //…;
    }
    catch (string& s) {
        cout << s.c_str() << endl;
    }
```

```
}
```

result:

请处理

3

```
#include <iostream>
using namespace std;
double Div(double, double);
void main()
{
    try {//如果发生异常则只中断try里面块
        cout << "7.3/2.0=" << Div(7.3, 2.0) << endl;
        cout << "7.3/0.0=" << Div(7.3, 0.0) << endl;
        cout << "7.3/1.0=" << Div(7.3, 1.0) << endl;
        cout << "全部执行完毕. \n";
    }//try和catch必须相邻；顺序不能颠倒
    catch (double) {//参数可以省略但是类型不能省略；建议不要使用简单类型
        cout << "except of deviding zero.\n";
    }
    cout << "That is ok. \n";
}
double Div(double a, double b) {
    if (b == 0.0)
        throw b;//类型匹配必须严格 类型相同；Throw匹配不上任何catch时abort//Throw和catch可以跨函数放置
    return a / b;
}
```

result: 7.3/2.0=3.65 except of deviding zero. That is ok.

Throw匹配不上任何catch时abort： 如果抛出的异常没有被任何`catch`子句捕获（即没有找到匹配的异常类型），那么程序的行为通常是调用`std::terminate`函数，该函数默认行为是调用`std::abort`来终止程序。不过，你可以通过`std::set_terminate`函数来定制`std::terminate`的行为。

为了避免程序因为未捕获的异常而意外终止，你应该确保你的代码中对于所有可能抛出的异常类型都有相应的`catch`子句来处理，或者使用更通用的异常类型（如`std::exception`或其基类）作为最后的捕获手段。

注意：在实际的编程实践中，尽量避免使用过于宽泛的`catch(...)`子句（即捕获所有类型异常的子句），因为它可能会掩盖潜在的错误，并使调试变得困难。通常最好只捕获你能够处理并知道如何恢复的异常类型.

4

```
double Div(double, double);
void f(){
```

```
    try{cout <<"7.3/0.0=" <<Div(7.3, 0.0) <<endl;}
    catch(double){cout<<"double"<<endl;throw 'a';}
}
void g(){
     try {f();}
     catch(char){cout<<"gg"<<endl; throw;}
}
void main(){
   try{  g();}
   catch(int){cout <<"零做除数错误\n";}
   catch(...) { cout<<"缺省异常"<<endl;}//通用异常
   cout<<"处理完毕"<<endl;
}
double Div(double a, double b){
    if(b==0.0)
       throw b;
    return a/b;
}
```

result:

double gg 缺省异常 处理完毕

异常处理对类型匹配要求非常严格

异常捕捉的类型匹配之苛刻程度可以和模板的类型匹配相当,它不允许相容类型的隐式转换,比如,抛掷char类型用int型就捕捉不到. 例如下列代码不会输出"int exception.", 当然也不会输出"That's ok.", Abort（）进程被调用，因为异常没有被捕获。

```
int main(){
  try{
    throw 'H';
  }
  catch(int){
    cout<<"int exception.\n"
  }
  cout<<"That's ok.\n";
}
```

栈展开

```
#include <iostream>
using namespace std;
class A {
public:
    A() {}
    ~A() { cout << "erase A object." << endl; }
private:
    int k;
```

```
};

void f() {
    A s;
    throw 5;//异常被处理后各函数对象被释放；其它语句不执行
    cout << "f() doing" << endl;
}
void main() {

    try { f(); }
    catch (int) { cout << "f() error" << endl; }
}
```

result:

erase A object. f() error

在C++中，当异常被抛出时，程序的控制流会立即离开当前函数（在这个例子中是`f()`），并且开始寻找能够处理这个异常的`catch`块。在这个过程中，会进行所谓的栈展开（stack unwinding），它确保所有在抛出点之前已经构造且尚未销毁的局部对象（在这个例子中是`A`类型的对象`s`）的析构函数都会被调用。

现在，我们来分析你给出的代码：

1. `main()` 函数中调用了`f()`。
2. `f()` 函数中创建了一个`A`类型的局部对象`s`。
3. 在`s`对象创建之后，`f()`函数中抛出了一个整数异常（`throw 5;`）。
4. 因为`f()`函数中没有处理这个整数异常的`catch`块，所以控制流会离开`f()`函数，并查找外部是否有处理这个异常的`catch`块。
5. 在`main()`函数中，有一个`try-catch`块，它捕获了整数类型的异常。
6. 在离开`f()`函数之前，由于栈展开，`A`类型对象`s`的析构函数会被调用，输出 "erase A object."。
7. 控制流转移到`main()`函数中的`catch`块，输出 "f() error"。

注意，在`f()`函数中，`throw 5;`之后的`cout << "f() doing" << endl;`不会被执行，因为一旦异常被抛出，控制流就立即离开了`f()`函数。

继承机制对异常处理的影响

```
class A{
    char net[20];
public: A(char * net){strcpy(this->net,net);}
};
class B:public A{
    long card;
public: B(long card,char * net):A(net){this->card=card;}
};
class C:public B{
    int port;
public: C(int port,long card,char * net):B(card,net){this->port=port;}
};
void net(){
   throw B(1,"202.198.22.26");
}
```

```
void main(){
   try{net();}
   catch(C){cout<<"端口错误"<<endl;}
   catch(B){cout<<"网卡错误"<<endl;}
   catch(A){cout<<"网络错误"<<endl;}  //放到最前面如何?
}
```

首先，让我们梳理一下代码中的主要部分：

1. 类定义：
   - A 类有一个字符数组 net，它用于存储一个网络地址。
   - B 类继承自 A 类，并添加了一个 long 类型的成员 card，用于存储网卡信息。
   - C 类继承自 B 类，并添加了一个 int 类型的成员 port，用于存储端口信息。
2. 构造函数：
   - 每个类都有一个构造函数，它接受必要的参数来初始化对象的成员。
   - A 类的构造函数接受一个字符指针，并使用 strcpy 函数将其复制到 net 数组中。
   - B 类的构造函数接受一个 long 类型的 card 和一个字符指针 net，并调用 A 类的构造函数来初始化 net。
   - C 类的构造函数接受一个 int 类型的 port、一个 long 类型的 card 和一个字符指针 net，并调用 B 类的构造函数来初始化 card 和 net。
3. 函数 net()：
   - 此函数抛出一个 B 类型的异常，并传递一个网卡值和一个网络地址作为参数。
4. main() 函数：
   - 使用 try-catch 块来捕获和处理异常。
   - try 块中调用 net() 函数，该函数抛出一个 B 类型的异常。
   - catch 块按照从具体到一般的顺序排列，以尝试捕获并处理不同类型的异常。

现在，关于你的问题：“放到最前面如何？”你指的是将捕获 A 类型异常的 catch 块放到最前面。让我们分析这样做的影响：

> Catch(基类类型)能够捕获throw 派生类对象 Catch(基类指针类型)能够捕获throw 派生类指针 反之不可以；所以catch(基类)总放在catch(派生类)后面

如果真的执行注释，那么输出将从网卡错误->网络错误。

多态机制对异常处理的影响

```
class A{
    char net[20];
public:
    virtual  void x(){cout<<"A";}
};
class B:public A{
    long card;
public:
   void x(){cout<<"B";}
};
class C:public B{
   int port;
```

```
public:
   void x(){cout<<"C";}
};
void net(){
   throw B();}
void main(){
   try{net();}
   catch(A & s){s.x();} //多态调用
}
```

result:

B

异常申述（异常声明）

```
class A{
   char net[20];
public:
    virtual  void x(){cout<<"A";}
};
class B:public A{
    long card;
public:
    void x(){cout<<"B";}
};
class C:public B{
     int port;
public:
void x(){cout<<"C";}
};
void net(){
   throw B();}
-----------------------------------------------------
void net() throw(A,B,C);  //异常申述（异常声明）
void main(){
   try{net();}
   catch(A & s){s.x();}
}
```

在C++（不是C，因为C语言没有异常处理机制）中，`throw(A,B,C);` 是C++98及之前版本中的异常规格（exception specification）的一种用法，它用于声明一个函数可能抛出的异常类型。这被称为动态异常规格（dynamic exception specification）。

然而，你给出的 `void net() throw(A,B,C);` 这行代码的意思是说，函数 `net()` 可能会抛出类型为 `A`、`B` 或 `C` 的异常。如果 `net()` 函数抛出了这三种类型之外的异常，那么程序会调用 `std::terminate()` 终止执行。

但是，这种异常规格在C++11中已被弃用，并在C++17中完全移除。现代C++代码通常不使用这种异常规格，而是依赖于异常处理机制（`try-catch` 块）来捕获和处理异常。

如果你的意图是限制函数可能抛出的异常类型，并希望在编译时进行检查，那么你应该使用 `noexcept` 关键字，它是C++11引入的，用于声明函数不会抛出任何异常。例如：

```
void net() noexcept {
    // 函数体，这里不应该有任何throw语句
}
```

如果函数可能会抛出异常，但你不想限制异常的类型，那么你可以简单地不写任何异常规格，让函数能够抛出任何类型的异常。

另外，需要注意的是，即使你声明了异常规格，但如果在函数内部抛出了未声明的异常类型，编译器通常不会报错（至少在C++98/03中是这样），而是在运行时调用 `std::unexpected()`（如果定义了该函数的话），或者默认行为是调用 `std::terminate()` 来终止程序。因此，异常规格主要是一种编译时文档，用于告知其他开发者该函数可能抛出的异常类型，而不是一种强制性的运行时检查机制。

异常处理对for循环的影响

```
class A{};
void f3(){
  if(...) throw A; // 退出for循环
}
void f2(){
  f3();
};
void f1(){
  f2();
}
int main(){
  try{
    for(...; ...; ...)
      f1();
  }
catch(A){
    //输出循环结果
  }
}
```

异常处理（异常抛出和捕获）在C++中并不会直接影响for循环的结构或行为。但是，当在for循环的某个迭代中抛出异常时，如果异常没有被立即捕获，那么它会沿着调用栈向上传播，直到找到一个可以处理它的catch块。

在你的示例代码中，如果在f3函数中抛出了A类型的异常，并且这个异常没有在f3、f2或f1中被捕获，那么它最终会传播到main函数中的try块，并在那里的catch(A)块中被捕获。

一旦异常被捕获，控制流会立即跳转到catch块的开始处，并且for循环的当前迭代会立即终止。但是，请注意，for循环本身并不会因为异常而“退出”或“中断”。异常处理只是改变了控制流的正常流程，使得在发生异常时，程序可以跳转到一个安全的位置继续执行。

如果你的意图是在发生异常时立即停止整个for循环，并处理异常，那么你的代码已经实现了这一点。但是，如果你希望在捕获异常后继续执行for循环的后续迭代（这通常不是一个好的做法，因为它可能会导致意外的副作用），那么你需要将try-catch块移动到for循环的循环体内部，并相应地处理异常。

另外，你提到在catch块中“输出循环结果”。但是，请注意，由于异常的发生，循环可能在完全执行完所有迭代之前就终止了。因此，你可能无法获得完整的循环结果。如果你需要在循环结束后输出某些结果，你可能需要在循环之前或之后计算这些结果，并在适当的时候输出它们。

下面是一个修改后的示例代码，演示了如何在for循环的循环体内部使用try-catch块来处理异常：

```
#include <iostream>

class A {};

void f3() {
  // ... 其他代码 ...
  if (/* 某个条件 */) throw A(); // 抛出A类型的异常
  // ... 其他代码 ...
}

void f2() {
  f3();
}

void f1() {
  f2();
}

int main() {
  for (/* 初始化 */; /* 条件 */; /* 迭代 */) {
    try {
      f1();
    } catch (const A&) {
      // 输出关于异常的信息，但不会输出循环结果（因为可能不完整）
      std::cerr << "Caught exception of type A inside the loop." << std::endl;
      // 你可以选择继续循环（但通常不推荐这样做）
      // 或者使用break来退出循环
      break; // 退出循环
    }
    // 注意：这里的代码在异常被捕获后将不会执行（除非使用了break而不是throw）
  }

  // 在这里输出完整的循环结果（如果有的话）
  // 但由于可能发生了异常，这个结果可能是不完整的
}
```