C++基本知识

1. 基本

- 编译: 将高级语言的程序翻译成机器语言
    - 解释执行: 边解释边执行
    - 编译执行: 先编译再执行
- 链接: 将多个目标文件链接成一个可执行文件
- 加载: 将可执行文件加载到内存中
- 调试
- 程序的基本结构
    - 注释
    - 编译预处理指令
    - 使用命名空间
    - 主程序
- 编译预处理:处理#的指令
- 库包含的格式
    - 库是预先做好的一些工具程序
    - 每个库要提供一个接口,告诉库的用户如何使用库提供的功能
    - 库包含就是把库的接口文件放入源文件,以便编译器检查程序中对库的调用是否正确

  #include <> 系统库

  #include "" 用户库
- 名字空间:
    - 在大型的程序中,每个源文件可能有相同的名字
    - 名字空间可以将名字封装在一个名字空间中,这样就可以避免名字冲突
    - 名字空间的使用:名字空间名::名字

- 使用名字空间的指令:using namespace 名字空间名;

2. 数据类型

2.1 自动类型推断

- auto:自动推断变量的类型并且需要赋初值

```
auto 变量名 = 初值;
```

它会忽略表达式的引用和const/volatile修饰符

- decltype:推断表达式的类型

```
decltype(表达式) 变量名;
```

2.2 重新命名类型名

- typedef:重新命名类型名

```
typedef 类型名 新类型名;
```

- using:重新命名类型名

```
using 新类型名 = 类型名;
```

2.3 占用的内存量

- sizeof:占用的内存量

```
sizeof(类型名);
sizeof(表达式);
```

2.4 转义字符

字符形式	含义
\n	换行
\t	水平制表
\b	退一格
\r	回车
\f	换页
\\	\
\’	’
\”	”
\ddd	1到3位八进制数代表的字符
\xhh	1到2位十六进制数代表的字符

2.5 常量

- const:常量

```
const 类型名 常量名 = 初值;
```

符号常量的初值可以是一个变量或表达式的结果。

合法:

```
int a;
cin >> a;
const int x = 2*a;
```

常量表达式的初值必须是编译时可以确定的。

合法:

```
constexpr int x = 2;
```

非法:

```
int a;
cin >> a;
constexpr int x = 2*a;
```

2.6 类型转换

- 强制类型转换

```
类型名 (表达式);
```

- static_cast:静态类型转换 static_cast<类型名>(表达式);：这是C++风格的静态类型转换。它用于在相关类型之间进行转换，例如整型和浮点型之间的转换。 例如：int x = static_cast(3.14); 将浮点数3.14转换为整数3。

```
static_cast<类型名>(表达式);
```

- dynamic_cast:动态类型转换 dynamic_cast<类型名>(表达式);：这是C++风格的动态类型转换。它用于在多态类型之间进行转换，通常用于基类指针或引用转换为派生类指针或引用。 例如：Derived* d = dynamic_cast<Derived*>(basePtr); 将基类指针basePtr转换为派生类指针d。

```
dynamic_cast<类型名>(表达式);
```

- reinterpret_cast:重新解释类型 reinterpret_cast<类型名>(表达式);：这是C++风格的重新解释类型转换。它用于将一个指针或引用转换为另一个指针或引用，不考虑类型的语义。 例如：int* p = reinterpret_cast<int*>(0x12345678); 将整数0x12345678转换为指针。

```
reinterpret_cast<类型名>(表达式);
```

- const_cast:去除常量属性 这是C++风格的去除常量属性的类型转换。它用于去除指针或引用的常量属性。 例如：const int* p = &x; int* q = const_cast<int*>(p); 将常量指针p转换为非常量指针q。

```
const_cast<类型名>(表达式);
```

2.7 枚举类型

- enum:枚举类型

```
enum 类型名 { 枚举值列表 };
enum 类型名 { 枚举值列表 } 变量名;
enum 类型名 { 枚举值列表 } 变量名 = 枚举值;
```

2.8 存储类别

存储类型 数据类型 变量名

- auto：在函数内或块内定义的变量缺省时是auto；进入块，分配空间，推出块，释放空间。

  auto int i;

  存储在栈空间中

- register：存储在寄存器：代替自动变量或形参，提高变量的访问速度。
- extern：在某函数中引用了一个在本函数后的全局变量时，需要在函数内用extern声明此全局变量。

  extern 类型 变量;

  当一个程序中有多个源文件组成时，用extern可引用另一文件中的全局变量。

2.9 static

整个程序的运行期间都存在，但访问被限定在程序的某一范围内。

- 静态的局部变量：

  第一次调用函数时被定义，程序运行时始终存在，但只能在被定义的函数内使用，函数执行结束后不消亡，再次调用函数时不重新定义，继续沿用原有特征，静态变量的值在函数调用之间保持不变。默认赋值为0.

- 静态的全局变量：

  只能被本源文件使用，不能被外来文件extern引用

3. I/O

- cin:输入

> 流提取运算符

用户读到空白字符会停止(空格，制表符和回车)

- cin.get()

从键盘接收一个字符，可以是空白字符

用法:

```
cin.get(字符变量名);
ch = cin.get();
```

- cout :输出

<< 流插入运算符

- cin.get(字符数组， 长度， 结束字符)：将结束字符留在输入流中
- cin.getline(字符数组， 长度， 结束字符)： 将结束字符从输入流中删除
- 输入异常：

用户的输入和程序的要求不匹配，忽略后面所有的输入语句

- cin.eof()
- cin.bad()
- cin.fail()
- cin.clear()

4. 函数

4.1 内联函数

- 内联函数：

  inline 直接插入到对应代码处

4.2 参数默认值

- 声明函数时指定

类型 函数名(形式参数类型 形式参数名 = 初值)

- 如果调用函数时没有指定实际参数时，编译器自动将默认值赋给形式参数：

  int SaveName (char *first, char *second= "", char *third = "", char *fouth="");

  无论缺省参数有几个，都必须放在参数序列的最后

4.3 函数模板

```
//定义形式

template<模板形式参数表>

返回类型  函数名(形式参数表) {
    //函数体
}

//example:

#include<iostream>

using namespace std;

template <class T>
T max(T a, T b) {
    return a>b?a:b;
}

int main() {
    cout << max(3,5) << endl;
    cout << max(3.4,5.6) << endl;
    cout << max('d','t') << endl;
    return 0;
}
```

- 显式实例化

某些模板参数在函数的形式参数表中没有出现，编译器无法推断模板实际参数的类型

```
template<class T1, class T2, class T3>
T1 calc(T2 x, T3 y) {
    return x+y;
}

calc<char,int,char>(5,'a');
```

```
//='f'
calc<int, int, char> (5,'a');
//=102
```

- 函数模板的特化：

```
//通用模板

template <typename T>
void print(T value) {
    //
}

template <>
void print<int>(int value) {
    //
}

template <>
void print<double> (double value) {
    //
}
```

4.4 数组作为函数的参数

- 数组退化为指针：

  当数组作为函数参数时，数组会退化为指向数组首元素的指针。这意味着在函数内部，arr 实际上是一个指针，而不是数组。

- 数组长度的传递：

  由于数组退化为指针，函数内部无法直接获取数组的长度。因此，需要额外传递数组的长度作为参数。

- 使用 std::array 或 std::vector：

  在现代C++中，推荐使用 std::array 或 std::vector 来代替原生数组，因为它们提供了更多的安全性和便利性。

所以在函数中对形式参数的数组进行修改可以修改实际参数。

解决方案：

1. const

```
void printArray(const int arr[], int size) {
    for (int i = 0; i < size; i++) {
        // 尝试修改数组元素，编译器会报错
        // arr[i] *= 2;
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}
```

2. 传递数组的副本：

   如果你希望在函数内部对数组进行修改，但不影响原始数组，可以在调用函数时传递数组的副本。这可以通过创建一个新的数组并将原始数组的元素复制到新数组中来实现。

- 二维数组作为函数参数

  二维数组的传递一定要指定第二个下标，并且时编译时的常量

5. 指针

5.1 reinterpret_cast

重新解释内存中的二进制数据

```
int i;
float *p = reinterpret_cast<float*>(&i);
```

5.2 指针与数组

一旦执行p=array，p就不再是数组名了，而是数组首元素的地址。并且对p的修改会影响到数组的值。

5.3 动态内存分配与回收

- 申请动态变量

  申请动态变量： p = new type;

  申请动态数组： p = new type[size];

  申请动态数组并初始化： p = new type[size]{value1, value2, ...};

申请动态变量并初始化： p = new type（value）；

new操作失败时返回空指针

- 释放动态变量的空间

  释放动态变量 delete p;

  释放动态数组 delete[] p;

  字符数组可以不加[]

- OS:

  heap: 堆，动态分配

  stack: 栈，自动分配；函数被调用时，空间被分配；函数执行结束后，空间被释放；

  静态分配：全局变量和静态变量，这在整个程序中都存在

5.4 字符串与指针

用法：

```
char *String;
String = "Hello";

char *String, ss[] = "abcde";
String = ss;

char *String;
String = new char[10];
cin >> String;
```

5.5 const 与指针

- const int *p;

  不能通过p修改指向的内容，但可以让p指向不同的变量

- int *const p = &x;

  不能让p指向不同的变量，但可以通过p修改指向的内容，而且必须给定初值；

- const int *const p = &x;

  既不能让p指向不同的变量，也不能通过p修改指向的内容，而且必须给定初值；

5.6 返回指针的函数

```
类型 *函数名 （形式参数表）；
```

6. 引用

给别的变量起别名，使用同一个内存单元。

- 引用的声明：

  类型 &引用名 = 变量名；

6.1 引用传递

指针参数：

```
void swap(int *m, int *n) {
    int temp = *m;
    *m = *n;
    *n = temp;
}
swap(&a, &b);
```

引用参数：

```
void swap(int &m, int &n) {
    int temp = m;
    m = n;
    n = temp;
}
swap(a, b);
```

实参必须是变量，不能是表达式或常量

6.2 const与引用

```
const 类型 &引用名 = 变量名;
```

控制变量的修改，不管被引用的对象是常量还是变量。

6.3 const_cast

- 修改一些指针/引用的const 权限
- 将const type * 转换为type *
- 将const type & 转换为type &
- 可以通过指针/引用修改指向的内容

7. 高级指针

main函数的形参：

```
int main(int argc, char *argv[])

int argc: 命令行参数的个数

char *argv[]: 指向命令行参数字符串的指针数组
```

在命令行输入**程序名**，输出结果为argc,argv

指向函数的指针：

```
返回类型 (*指针名)(形参表)

int isdigit(int c) {};

int (*p)(int);

p = isdigit;

auto p = isdigit;
```

8. 结构体变量

8.1 结构体

```
struct 结构体名 {
    类型 成员名;
}

结构体名 a,*b;
```

```
b = new 结构体名;

b = new 结构提名[10];
```

8.2 结构体访问

```
结构体名.成员名;
```

8.3 结构体指针的引用

```
(*指针).成员;
指针->成员;
```

9. 派生类

9.1 类定义格式

```
class 派生类名：派生方法 基类名 {
    //
};
```

派生方法：决定基类成员在派生类中的访问特性

派生类函数不能访问基类的私有成员

公有派生：基类的公有成员在派生类中仍然是公有成员 public;

私有派生：基类的公有成员在派生类中是私有成员 private;

派生实例

```
class base {
        int x;
  public:
        void setx(int k);
};

class derived1:public base {
        int y;
  public:
        void sety(int k);
};
```

Derived1数据成员

x

y

两个成员函数

setx

sety

访问特性	Derived1
不可访问	int x
private	int y
public	setx() sety()

157

保护成员：一类特殊的私有成员，可以被派生类的成员函数访问

派生类对象的构造和析构

派生类对象的构造

基类的构造函数初始化基类数据成员

派生类构造函数初始化派生类新增加的数据成员

派生类构造函数自动调用基类的构造函数

派生类对象的析构

基类的析构函数析构基类数据成员

派生类析构函数析构派生类新增加的数据成员

派生类的析构函数自动调用基类的析构函数

派生类对象析构时，先执行派生类的析构函数，再执行基类的析构函数

派生类构造函数

派生类构造函数的格式

派生类构造函数名（参数表）： 基类构造函数名（参数表） { ……}

执行过程

先执行基类的构造函数，再构造派生类新增部分

注意事项

基类构造函数中的参数表通常来源于派生类构造函数的参数表，也可以用常量

若基类使用缺省或不带参数的构造函数，则在派生类定义构造函数时可略去基类构造函数调用

9.2 派生类作为基类

基类本身可以是一个派生类。

每个派生类继承他的直接积累的所有成员。

实例

```
class base {     int x;
  public:
    base(int xx) {x=xx; cout<<"constructing base\n";}
    ~base() {cout<<"destructint base\n";}
};
class derive1 : public base {      int y;
public:
    derive1(int xx, int yy): base(xx)    { y = yy;  cout << "constructing derive1\n";  }
    ~derive1() {  cout << "destructing derive1\n";   }
};
class derive2 : public derive1{     int z;
public:
     derive2(int xx, int yy, int zz):derive1(xx, yy)   { z = zz;  cout<<"constructing derive2\n";  }
    ~derive2() {cout<<"destructing derive2\n";}
 };
int  main()
{
    derive2 op(1, 2, 3);
    return 0;
}
```

```
constructing base
constructing derive1
constructing derive2
destructing derive2
destructing derive1
destructint base
```

10. 虚函数

基类中的一类成员函数，允许函数调用与函数体之间的联系在运行时才建立

通过基类指针或基类引用调用基类的虚函数时，会判断指针指向的对象。如指向的基类对象，则执行虚函数。如指向派生类对象，则检查派生类是否重定义了此函数。如重定义，则执行派生类的重定义函数，否则执行基类的虚函数

声明时函数头前加关键词virtual

11. 智能指针

3种智能指针的区别

多个指针指向同一对象时，3个指针有不同的行为

auto_ptr
在赋值时，将右边指针的控制权转移给左边指针，右值失去了控制权，右值析构时可能会出错

shared_ptr
允许多个指针指向同一个对象
析构时，最后一个析构的指针调用对象的析构函数

unique_ptr
明确指出不允许共享
当发生指针赋值时，编译出错

12. IO

与标准库

头文件	类型
iostream	istream从流中读取 ostream写到流中去 iostream对流进行读写，从istream和ostream派生
fstream	ifstream从文件中读取，由istream派生而来 ofstream写到文件中去，由ostream派生而来 fstream对流进行读写，由iostream派生而来
sstream	istringstream从string对象中读取，由istream派生而来 ostringstream写到string对象中去，由ostream派生而来 stringstream对string对象进行读写，由iostream派生而来

输出缓冲区的刷新

程序正常结束

作为main函数返回工作的一部分，将真正输出缓冲区的内容，清空所有的输出缓冲区；

缓冲区满

在写入下一个值之前，会刷新缓冲区；

强制刷新

用标准库的操纵符，如行结束符endl，显式地刷新缓冲区；
在每次输出操作执行结束后，用unitbuf操纵符设置流的内部状态，从而清空缓冲区；

关联输出流与输入流

在读输入流时，将刷新其关联的输出缓冲区。
在标准库中，将cout和cin关联在一起，因此每个输入操作都将刷新cout关联的缓冲区。

设置浮点数精度

浮点数的精度指实型数的有效位数

设置方法

流操纵符：setprecision（位数）
成员函数：precision （位数）

一旦设置了精度，将影响所有输出的浮点数的精度，直到下一个设置精度的操作为止。

设置小数点后的位数

将两个流操纵符fixed和setprecision结合使用。fixed表示以定点小数形式输出，此时setprecision的参数表示小数点后的位置

```
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    double x = 123.456789, y = 9876.54321;

    cout << fixed << setprecision(2) << x << '\t' << y << endl;
    cout << fixed << setprecision(3) << x << '\t' << y << endl;

    return 0;
}
```

```
123.46      9876.54
123.457     9876.543
```

设置域宽

域宽是指数据所占的字符个数

设置方法

成员函数：width（宽度）
流操纵符：setw（宽度）

注意

可用于输入，也可用于输出
适合于下一次输入或输出，之后的操作的宽度将被设置为默认值
一旦设置了域宽，下一个输出必须占满域宽
如果输出值的宽度比域宽小，则左边插入填充字符填充。默认的填充字符是空格
当没有设置输出宽度时，C++按实际长度输出

设置域宽

设置域宽也可用于输入
当输入是字符串时，读入域宽指定的字符个数
如有定义

```
char a[9] , b[9] ;
```

执行语句

```
cin >> setw(5) >> a >> setw(5) >> b;
```

用户在键盘上的响应为

```
abcdefghijklm
```

则字符串a的值为“abcd”，字符串b的值为“efgh”。

设置域宽

如整型变量a=123，b=456，则输出

```
cout << a << b;
```

将输出 123456

- 如语句

```
cout << setw(5) << x <<  setw(5) << y << endl;
```

的输出为

```
 123  456
```

每个数值占5个位置，前面用空格填充。

其他流操纵符

流操纵符	描述
skipws	跳过输入流中的空白字符，使用流操纵符**noskipws**复位该选项
left	输出左对齐，必要时在右边填充字符
right	输出右对齐，必要时在左边填充字符
showbase	指名在数字的前面输出基数，以**0**开头表示八进制，**0x**或**0X**表示十六进制。使用流操纵符**noshowbase**复位该选择
uppercase	指明当显示十六进制数时使用大写字母，并且在科学计数法输出时使用大写字母**E**。可以用流操纵符**nouppercase**复位
showpos	在正数前显示加号（+），可以用流操纵符**noshowpos**复位
scientic	以科学计数法输出浮点数
fixed	以定点小数形式输出浮点数
setfill	设置填充字符，它有一个字符型的参数

13. 文件IO

基于文件的 I/O

流式文件

文件的顺序访问

文件的随机访问

访问有记录概念的文件

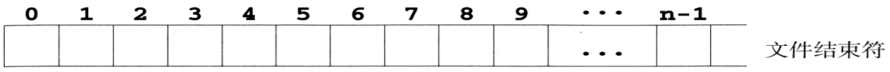
流式文件

C++语言把每一个文件都看成一个有序的字节流（把文件看成n个字节）

每一个文件以文件结束符(end-of-file marker)结束

当打开一个文件时，该文件就和某个流关联起来

与这些对象相关联的流提供程序与特定文件或设备之间的通信通道



ASCII文件和二进制文件

程序如何解释字节序列

ASCII文件

也被称为文本文件

将文件中的每个字节解释成一个字符的ASCII值

二进制文件

将文件中的每个字节仅看成是一个二进制比特串

由程序解释比特串的含义

如果二进制文件有4个字节，值为0000 0000 0000 0000 1111 1111 1111 1111

想解释成一个整型数，可以将这4个字节读入一个整型变量

想解释成float类型，可以将这4个字节读入一个float类型的变量

ASCII文件可以直接显示在显示器上，而直接显示二进制文件通常是没有意义的

定义一个流对象

- **C++有三个文件流类型**
- ifstream：输入文件流
- ofstream：输出文件流
- fstream：输入输出文件流
- 如：ifstream infile;

打开文件

两种打开方式

成员函数open

构造函数

函数参数

打开的文件名：一个字符串

文件打开模式

检查文件打开是否成功

打开失败，文件流对象值为0

打开模式	含义
in	打开文件，做读操作
out	打开文件，做写操作
app	在每次写操作前，找到文件尾
ate	打开文件后，立即将文件定位在文件尾
trunc	打开文件时，清空文件
binary	以二进制模式进行输入输出操作

默认打开模式

ifstream对象：以in模式打开

ofstream对象：以out模式打开

fstream对象：以in和out方式打开

打开文件示例

打开输入文件

用open函数：ifstream infile;
infile.open（“file1”）；或 infile.open（“file1”, ifstream::in);
用构造函数：ifstream infile（“file1”）；或 ifstream infile（“file1” , ifstream::in);

打开输出文件

用open函数：ofstream outfile;
outfile.open（“file2”）； 或 outfile.open（“file2”, ofstream::out);
用构造函数：ofstream outfile（“file2”）；或 ofstream outfile（“file2” , ofstream::out);

打开输入输出文件

用open函数：fstream outfile;
outfile.open（“file2”）； 或 outfile.open（“file2”, fstream::in | fstream::out);
用构造函数：fstream outfile（“file2”）;
或 fstream outfile（“file2” , fstream::in | fstream::out);

文件关闭

用成员函数close

对于输出文件，close会将缓冲区中的内容写入文件

main函数执行结束时，会关闭所有打开的文件

良好的程序设计习惯：文件访问结束时，关闭文件

ASCII文件的顺序访问

• 与控制台读写完全一样

- 读文件：>>、get、getline、read
- 写文件：<<”、put、write
- 支持格式化读写

• 判断文件结束

- >>读：可以通过判断输入流对象值是否为0
- get读：判断读入字符是否是EOF
- 其他方式读：通过成员函数eof
- eof函数不需要参数，返回一个整型值。当读操作遇到文件结束时，该函数返回1，否则返回0。

294

ASCII文件访问实例

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ofstream out("file");
     ifstream in;
     int i;

     if (!out) {   cerr << "create file error\n";     return 1;  }
     for (i = 1; i <= 10; ++i)      out << i << ' ';
     out.close();

     in.open("file");
     if (!in) {  cerr << "open file error\n";  return 1;  }
     while  ( in >> i )      cout << i << ' ';
     in.close();
     return 0;
}
```

执行结果

执行该程序后，文件file中的内容为

1 2 3 4 5 6 7 8 9 10

该程序的输出结果是

1 2 3 4 5 6 7 8 9 10

296

读ASCII文件

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
    ifstream fin("test");
    char s[80];
    int i;
    float x;

    if (!fin) {cout << "cannot open input file\n"; return 1;}
    fin >> i >> x >> s;
    cout << i << " " << x << s;
    fin.close();

    return 0;
}
```

10 123.456"This

fin > > i > > x; fin.getline(s, 80, '\n');

298

文件定位指针的操作

- **获取文件定位指针的当前位置**
  - tellg：返回读文件指针值
  - tellp ： 返回写文件指针值
- **设置文件定位指针的位置**
  - seekg ： 设置读文件指针值
  - seekp ： 设置写文件指针值

305

文件定位指针的操作

• 获取文件定位指针的当前位置

- tellg：返回读文件指针值
- tellp ： 返回写文件指针值

• 设置文件定位指针的位置

- seekg ： 设置读文件指针值
- seekp ： 设置写文件指针值

305

ASCII文件的随机读写实例

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    fstream in("file");
    int i;

   if (!in) {cerr << "open file error\n"; return 1;}
   in.seekp(10);
   in << 20;
   in.seekg(0);
   while (in >> i)
       cout << i <<    '\n ';
   in.close();

   return 0;
}
```

执行前文件内容：1 2 3 4 5 6 7 8 9 10

执行后文件内容：1 2 3 4 5 207 8 9 10

显示器上显示
1
2
3
4
5
207
8
9
10

307

14. 异常处理

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

329

异常处理机制

• 组成部分

- 异常抛出
- 异常捕获
- 异常处理

330

异常抛出

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，发送给更大的环境中。

格式

throw 对象;

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。

331

例1

throw myerror(“something bad happened”);
throw 5;

异常抛出实例 – 异常类定义

```
class DivideByZeroException {
 public:
    DivideByZeroException()
          : message( "attempted to divide by zero" ) { }
    const char *what() const { return message; }
 private:
    const char *message;
};
```

332

异常抛出实例 – 异常抛出

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

   return static_cast< double > ( x ) / y;
}
```

333

异常捕获

• 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。

• 格式

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

334

catch捕获异常

格式

catch (<捕获的异常类型> <可选对象>)
{
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

335

catch (...)

捕获任意类型的异常

带异常处理的解一元二次方程

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

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;
}
```

338

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

340