8 类型转换

8.1 类型转换的名称和语法

类型转换有 c 风格的,当然还有 c++风格的。c 风格的转换的格式很简单(TYPE) EXPRESSION,但是 c 风格的类型转换有不少的缺点,有的时候用 c 风格的转换是不合 适的,因为它可以在任意类型之间转换,比如你可以把一个指向 const 对象的指针转换 成指向非 const 对象的指针,把一个指向基类对象的指针转换成指向一个派生类对象的 指针,这两种转换之间的差别是巨大的,但是传统的 c 语言风格的类型转换没有区分这 些。还有一个缺点就是,c 风格的转换不容易查找,他由一个括号加上一个标识符组成, 而这样的东西在 c++程序里一大堆。所以 c++为了克服这些缺点,引进了 4 新的类型转换操作符。

C风格的强制类型转换(Type Cast)

```
TYPE b = (TYPE) a
```

C++提供了4种类型转换，分别处理不同的场合应用

static_cast	静态类型转换。
reinterpreter_cast	重新解释类型转换。
dynamic_cast	子类和父类之间的多态类型转换。
const_cast	去掉const属性转换。

8.2 转换方式

8.2.1 static_cast 静态类型转换

```
static_cast<目标类型> (标识符)
```

所谓的静态,即在编译期内即可决定其类型的转换,用的也是最多的一种。

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

int main(void)
{
    double dPi = 3.1415926;
    int num1 = (int)dPi;                    //c语言的 旧式类型转换
    int num2 = dPi;                         //隐式类型转换

    // 静态的类型转换:
    // 在编译的时 进行基本类型的转换 能替代c风格的类型转换 可以进行一部分检查
    int num3 = static_cast<int> (dPi); //c++的新式的类型转换运算符
    cout << "num1:" << num1 << " num2:" << num2 << " num3:" << num3 << endl;

    return 0;
}
```

8.2.2 dynamic_cast 子类和父类之间的多态类型转换

dynamic_cast<目标类型> (标识符)

用于多态中的父子类之间的强制转化。

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

class Animal
{
public:
    virtual void  cry() = 0;
};

class Dog : public Animal
{
public:
    virtual void  cry()
    {
        cout << "旺旺~ " << endl;
    }

    void doHome()
    {
        cout << "看家" << endl;
    }
};


class Cat : public Animal
```

```
{
public:
    virtual void  cry()
    {
        cout << "喵喵~ " << endl;
    }

    void doHome()
    {
        cout << "抓老鼠" << endl;
    }
};

int main(void)
{
    Animal *base = NULL;


    base = new Cat();
    base->cry();    //此时父类指针指向 猫

    //用于将父类指针转换成子类,
    Dog *pDog = dynamic_cast<Dog *>(base); //转换之后 讲父类指针转换成 子类狗指针
                                           // 但是由于父类指针此时指向的对象是猫,
                                           // 所以转换狗是失败的
    if (pDog != NULL)                      //如果转换失败则返回 NULL
    {
        pDog->cry();
        pDog->doHome();
    }

    Cat *pCat = dynamic_cast<Cat *>(base);//转换之后 讲父类指针转换成 子类猫指针
                                          //向下转换
    if (pCat != NULL)
    {
        pCat->cry();
        pCat->doHome();
    }

    return 0;
}
```

8.2.2 const_cast 去掉const属性转换

const_cast<目标类型>（标识符）//目标类类型只能是指针或引用。

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

struct A {
```

```
    int data;
};

int main(void)
{
    const A a = {200};

    A a1 = const_cast<A>(a);
    a1.data = 300;

    A &a2 = const_cast<A&>(a);
    a2.data = 300;
    cout<<a.data<<a2.data<<endl;

    A *a3 = const_cast<A*>(&a);
    a3->data = 400;
    cout<<a.data<<a3->data<<endl;

    const int x = 3;

    int &x1 = const_cast<int&>(x);
    x1 = 300;
    cout<<x<<x1<<endl;

    int *x2 = const_cast<int*>(&x);
    *x2 = 400;
    cout<<x<<*x2<<endl;

    return 0;
}
```

8.2.3 reinterpret_cast 重新解释类型转换

reinterpret_cast<目标类型> (标识符)

interpret 是解释的意思,reinterpret 即为重新解释,此标识符的意思即为数据的二进制形式重新解释,但是不改变其值。

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

class Animal
{
public:
    virtual void  cry() = 0;
};

class Dog : public Animal
{
```

```
public:
    virtual void  cry()
    {
        cout << "旺旺~ " << endl;
    }

    void doHome()
    {
        cout << "看家" << endl;
    }
};


class Cat : public Animal
{
public:
    virtual void  cry()
    {
        cout << "喵喵~ " << endl;
    }

    void doHome()
    {
        cout << "抓老鼠" << endl;
    }
};

class Book
{
public:
    void printP()
    {
        cout << "book" << endl;
    }
};

int main(void)
{
    Animal *base = NULL;

    //1 可以把子类指针赋给 父类指针 但是反过来是不可以的 需要 如下转换
    //Dog *pdog = base;
    Dog *pDog = static_cast<Dog *> (base);

    //2 把base转换成其他 非动物相关的 err
    //Book *book= static_cast<Book *> (base);

    //3  reinterpret_cast 可以强制类型转换
    Book *book = reinterpret_cast<Book *> (base);

    return 0;
}
```

建议1：

程序员要清除的知道：要转的变量，类型转换前是什么类型，类型转换后是什么类型。转换后有什么后果。

建议2：

一般情况下，不建议进行类型转换。

9 异常

1）异常是一种程序控制机制，与函数机制独立和互补

函数是一种以栈结构展开的上下函数衔接的程序控制系统,异常是另一种控制结构,它依附于栈结构,却可以同时设置多个异常类型作为网捕条件,从而以类型匹配在栈机制中跳跃回馈.

2）异常设计目的：

栈机制是一种高度节律性控制机制,面向对象编程却要求对象之间有方向、有目的的控制传动,从一开始，异常就是冲着改变程序控制结构，以适应面向对象程序更有效地工作这个主题，而不是仅为了进行错误处理。

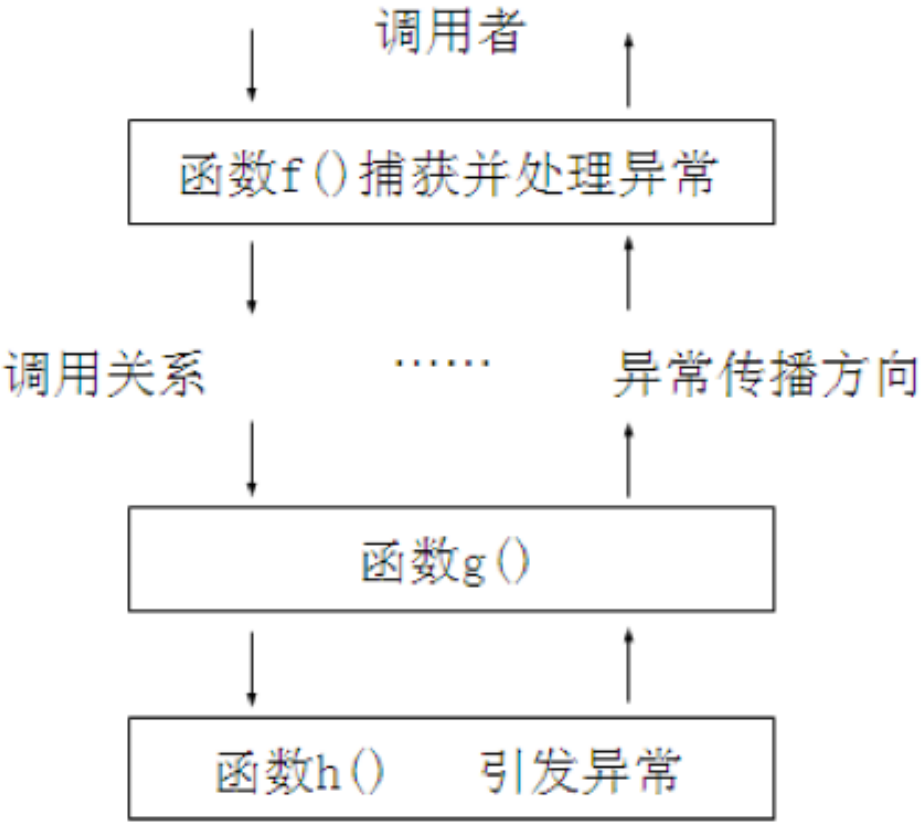
异常设计出来之后，却发现在错误处理方面获得了最大的好处。

9.1 异常处理的基本思想

9.1.1 传统的错误处理机制

通过函数返回值来处理错误。

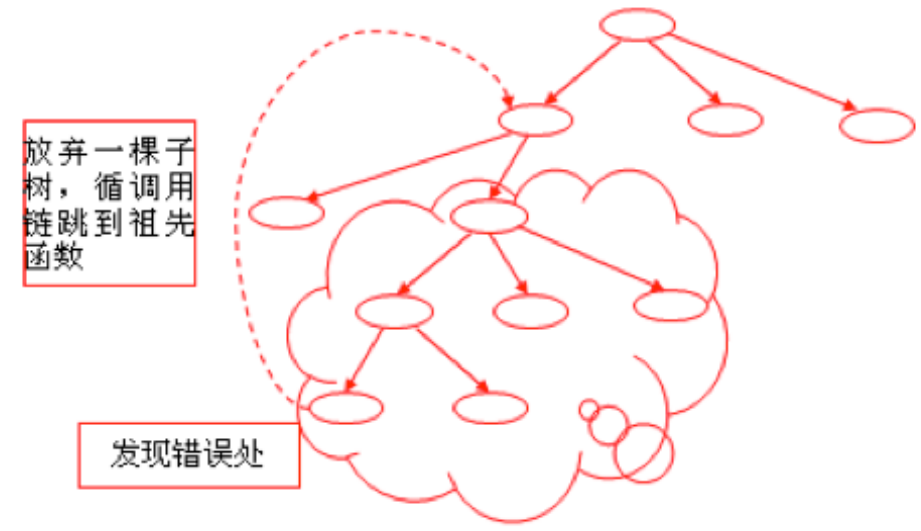
9.1.2 异常的错误处理机制



1）C++的异常处理机制使得异常的引发和异常的处理不必在同一个函数中，这样底层的函数可以着重解决具体问题，而不必过多的考虑异常的处理。上层调用者可以再适当的位置设计对不同类型异常的处理。

2）异常是专门针对抽象编程中的一系列错误处理的，C++中不能借助函数机制，因为栈结构的本质是先进后出，依次访问，无法进行跳跃，但错误处理的特征却是遇到错误信息就想要转到若干级之上进行重新尝试，如图

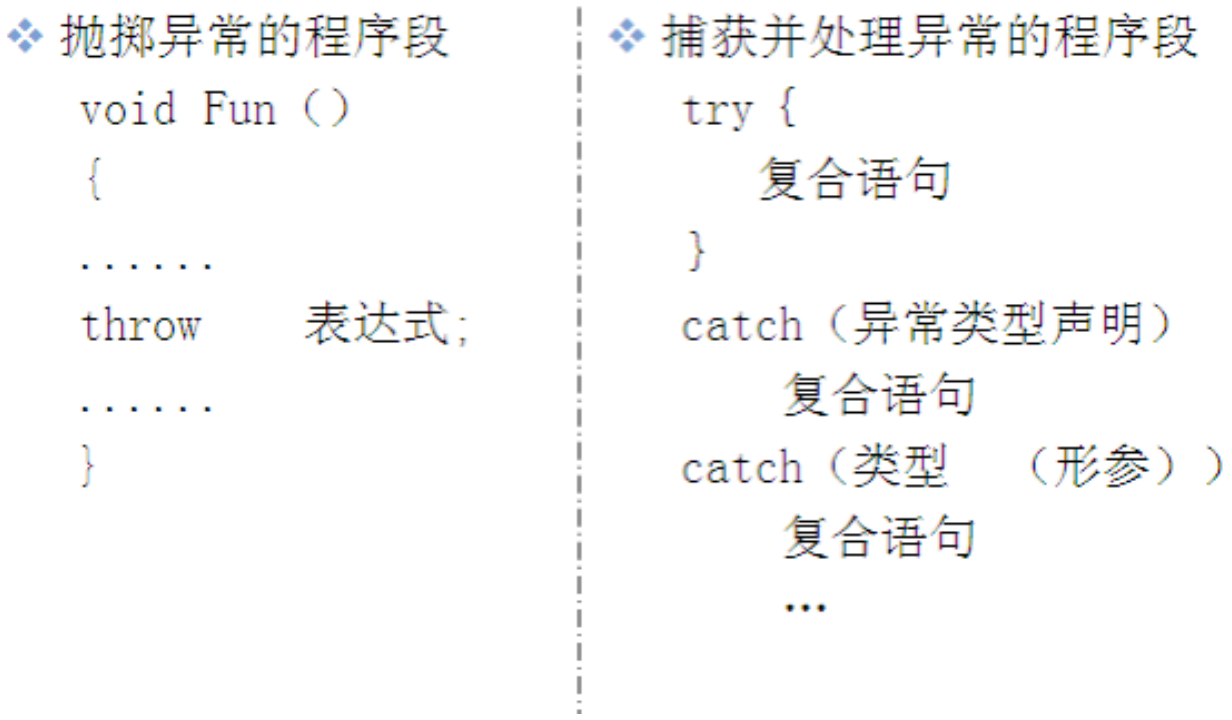
❖ 错误处理示意：



3）异常超脱于函数机制，决定了其对函数的跨越式回跳。
4）异常跨越函数

9.2 C++异常处理的实现

9.2.1 异常的基本语法



1）若有异常则通过throw操作创建一个异常对象并抛掷。
2）将可能抛出异常的程序段嵌在try块之中。控制通过正常的顺序执行到达try语句，然后执行try块内的保护段。
3）如果在保护段执行期间没有引起异常，那么跟在try块后的catch子句就不执行。程序从try块后跟随的最后一个catch子句后面的语句继续执行下去。
4）catch子句按其在try块后出现的顺序被检查。匹配的catch子句将捕获并处理异常（或继续抛掷异常）。
5）如果匹配的处理器未找到，则运行函数terminate将被自动调用，其缺省功能是调用abort终止程序。
6）处理不了的异常，可以在catch的最后一个分支，使用throw语法，向上扔。

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


int divide(int x, int y )
{
    if (y ==0)
    {
        throw x;
    }
    return x/y;
}

int main(void)
{
    try
    {
        cout << "8/2 = " << divide(8, 2) << endl;
        cout << "10/0 =" << divide(10, 0) << endl;
    }
    catch (int e)
    {
        cout << "e" << " is divided by zero!" << endl;
    }
    catch(...)
    {
        cout << "未知异常" << endl;
    }


    return 0;
}
```

9.2.2 栈解旋(unwinding)

异常被抛出后，从进入try块起，到异常被抛掷前，这期间在栈上的构造的所有对象，都会被自动析构。析构的顺序与构造的顺序相反。这一过程称为栈的解旋(unwinding)。

```
#include <iostream>

using namespace std;

class MyException {};

class Test
{
    public:
        Test(int a=0, int b=0)
        {
```

```
            this->a = a;
            this->b = b;
            cout << "Test 构造函数执行" << "a:" << a << " b: " << b << endl;
        }
        void printT()
        {
            cout << "a:" << a << " b: " << b << endl;
        }
        ~Test()
        {
            cout << "Test 析构函数执行" << "a:" << a << " b: " << b << endl;
        }
    private:
        int a;
        int b;
};

void myFunc() throw (MyException)
{
    Test t1;
    Test t2;

    cout << "定义了两个栈变量,异常抛出后测试栈变量的如何被析构" << endl;

    throw MyException();
}

int main(void)
{
    //异常被抛出后，从进入try块起，到异常被抛掷前，这期间在栈上的构造的所有对象>，
    //都会被自动析构。析构的顺序与构造的顺序相反。
    //这一过程称为栈的解旋(unwinding)
    try
    {
        myFunc();
    }
    catch(MyException &e)
    //catch(MyException ) //这里不能访问异常对象
    {
        cout << "接收到MyException类型异常" << endl;
    }
    catch(...)
    {
        cout << "未知类型异常" << endl;
    }

    return 0;
}
```

9.2.3 异常接口声明

1）为了加强程序的可读性，可以在函数声明中列出可能抛出的所有异常类型，例如：

```
void func() throw (A, B, C , D); //这个函数func（）能够且只能抛出类型A B C D及其子类型的异常。
```

2）如果在函数声明中没有包含异常接口声明，则次函数可以抛掷任何类型的异常，例如：

```
void func();
```

3）一个不抛掷任何类型异常的函数可以声明为：

```
void func()  throw();
```

4）如果一个函数抛出了它的异常接口声明所不允许抛出的异常，unexpected函数会被调用，该函数默认行为调用terminate函数中止程序。

9.2.4 异常类型和异常变量的生命周期

1）throw的异常是有类型的，可以使，数字、字符串、类对象。
2）throw的异常是有类型的，catch严格按照类型进行匹配。
3）注意 异常对象的内存模型 。

（一）传统的错误模型处理

```
//传统的错误处理机制
int my_strcpy(char *to, char *from)
{
    if (from == NULL)
    {
        return 1;
    }
    if (to == NULL)
    {
        return 2;
    }

    //copy是的 场景检查
    if (*from == 'a')
    {
        return 3; //copy时出错
    }
    while (*from != '\0')
    {
        *to = *from;
        to ++;
        from ++;
    }
```

```
    *to = '\0';

    return 0;
}

int main(void)
{
    int ret = 0;
    char buf1[] = "zbcdefg";
    char buf2[1024] = {0};

    ret = my_strcpy(buf2, buf1);
    if (ret != 0)
    {
        switch(ret)
        {
            case 1:
                printf("源buf出错!\n");
                break;
            case 2:
                printf("目的buf出错!\n");
                break;
            case 3:
                printf("copy过程出错!\n");
                break;
            default:
                printf("未知错误!\n");
                break;
        }
    }
    printf("buf2:%s \n", buf2);

    return 0;
}
```

(二) 抛出普通类型异常

```
//throw int类型异常
void my_strcpy1(char *to, char *from)
{
    if (from == NULL)
    {
        throw 1;
    }
    if (to == NULL)
    {
        throw 2;
    }

    //copy是的 场景检查
    if (*from == 'a')
    {
        throw 3; //copy时出错
    }
```

```
    while (*from != '\0')
    {
        *to = *from;
        to ++;
        from ++;
    }
    *to = '\0';
}
```

```
//throw char*类型异常
void my_strcpy2(char *to, char *from)
{
    if (from == NULL)
    {
        throw "源buf出错";
    }
    if (to == NULL)
    {
        throw "目的buf出错";
    }

    //copy是的 场景检查
    if (*from == 'a')
    {
        throw "copy过程出错"; //copy时出错
    }
    while (*from != '\0')
    {
        *to = *from;
        to ++;
        from ++;
    }
    *to = '\0';
}
```

(三) 抛出自定义类型异常

```
class BadSrcType {};
class BadDestType {};
class BadProcessType
{
    public:
        BadProcessType()
        {
            cout << "BadProcessType构造函数do \n";
        }


        BadProcessType(const BadProcessType &obj)
        {
```

```
            cout << "BadProcessType copy构造函数do \n";
        }

        ~BadProcessType()
        {
            cout << "BadProcessType析构函数do \n";
        }

};

//throw 类对象 类型异常
void my_strcpy3(char *to, char *from)
{
    if (from == NULL)
    {
        throw BadSrcType();
    }
    if (to == NULL)
    {
        throw BadDestType();
    }

    //copy是的 场景检查
    if (*from == 'a')
    {
        printf("开始 BadProcessType类型异常 \n");
        throw BadProcessType();
    }

    if (*from == 'b')
    {
        throw &(BadProcessType());
    }

    if (*from == 'c')
    {
        throw new BadProcessType;
    }
    while (*from != '\0')
    {
        *to = *from;
        to ++;
        from ++;
    }
    *to = '\0';
}

int main(void)
{
    int ret = 0;
    char buf1[] = "cbbcdefg";
    char buf2[1024] = {0};

    try
    {
```

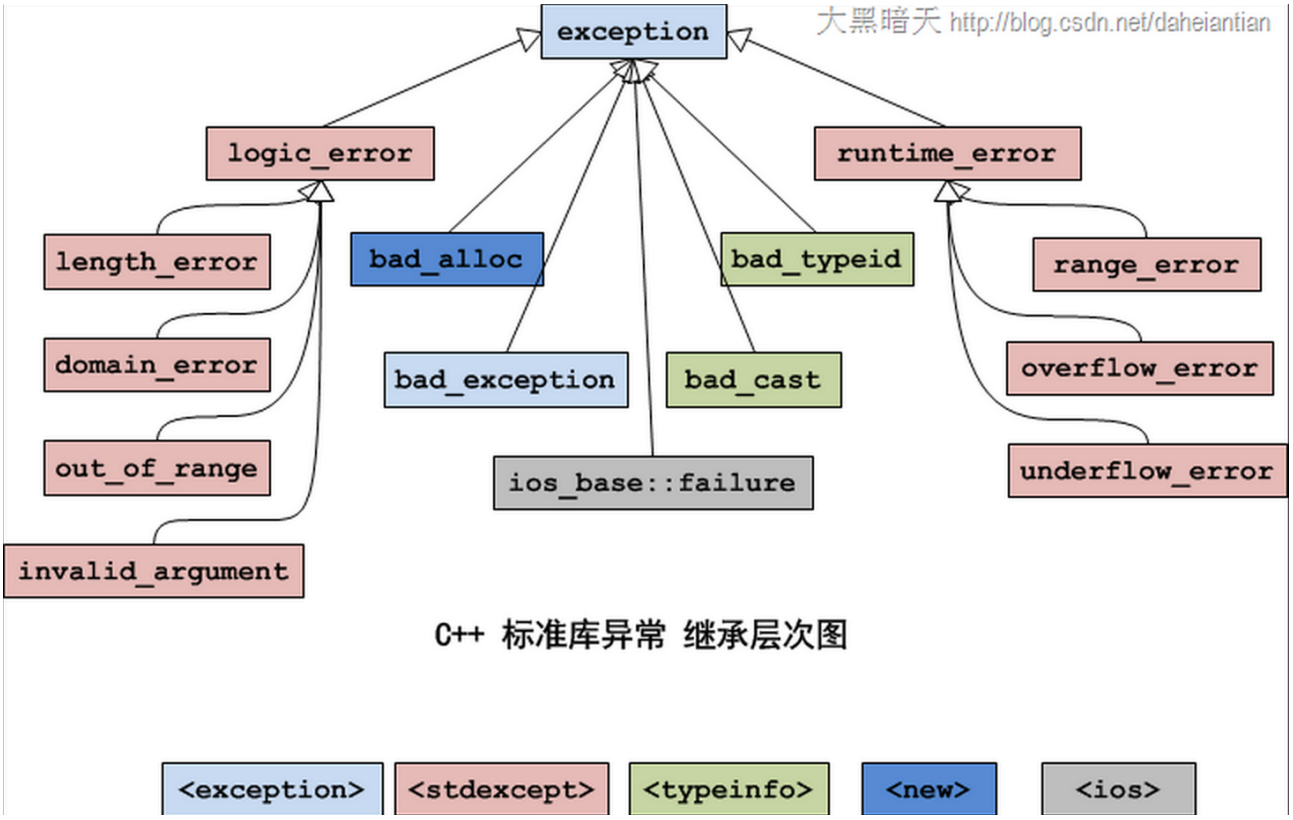
```
        //my_strcpy1(buf2, buf1);
        //my_strcpy2(buf2, buf1);
        my_strcpy3(buf2, buf1);
    }
    catch (int e) //e可以写 也可以不写
    {
        cout << e << " int类型异常" << endl;
    }
    catch(char *e)
    {
        cout << e << " char* 类型异常" << endl;
    }

    //---
    catch(BadSrcType e)
    {
        cout << " BadSrcType 类型异常" << endl;
    }
    catch(BadDestType e)
    {
        cout << " BadDestType 类型异常" << endl;
    }
    //结论1: 如果 接受异常的时候 使用一个异常变量,则copy构造异常变量.
    /*
       catch( BadProcessType e)
       {
       cout << " BadProcessType 类型异常" << endl;
       }
     */
    //结论2: 使用引用的话 会使用throw时候的那个对象
    //catch( BadProcessType &e)
    //{
    //  cout << " BadProcessType 类型异常" << endl;
    //}

    //结论3: 指针可以和引用/元素写在一块 但是引用/元素不能写在一块
    catch( BadProcessType *e)
    {
        cout << " BadProcessType 类型异常" << endl;
        delete e;
    }

    //结论4: 类对象时, 使用引用比较合适
    // -

    catch (...)
    {
        cout << "未知 类型异常" << endl;
    }

    return 0;
}
```

9.2.5 异常的层次结构

设计一个数组类 MyArray，重载[]操作，数组初始化时，对数组的个数进行有效检查

1） index<0 抛出异常eNegative
2） index = 0 抛出异常 eZero
3） index>1000抛出异常eTooBig
4） index<10 抛出异常eTooSmall
5） eSize类是以上类的父类，实现有参数构造、并定义virtual void printErr()输出错误。

9.3 标准程序库异常



每个类所在的头文件在图下方标识出来.

标准异常类的成员：

① 在上述继承体系中，每个类都有提供了构造函数、复制构造函数、和赋值操作符重载。

② logic_error类及其子类、runtime_error类及其子类，它们的构造函数是接受一个string类型的形式参数，用于异常信息的描述；

③ 所有的异常类都有一个what()方法，返回const char* 类型（C风格字符串）的值，描述异常信息。

异常名称	描述
exception	所有标准异常类的父类
bad_alloc	当operator new and operator new[]，请求分配内存失败时
bad_exception	这是个特殊的异常，如果函数的异常抛出列表里**声明了**bad_exception**异常**，当函数内部抛出了异常抛出列表中**没有的**异常，这是调用的unexpected函数中若抛出异常，不论什么类型，都会**被替换为**bad_exception**类型**
bad_typeid	使用typeid操作符，操作一个NULL指针，而该指针是带有虚函数的类，这时抛出bad_typeid异常
bad_cast	使用dynamic_cast转换**引用**失败的时候
ios_base::failure	io操作过程出现错误
logic_error	逻辑错误，可以在**运行前**检测的错误
runtime_error	运行时错误，仅在**运行时**才可以检测的错误

异常名称	描述
length_error	试图生成一个超出该类型最大长度的对象时，例如vector的resize操作
domain_error	参数的值域错误，主要用在数学函数中。例如使用一个负值调用只能操作非负数的函数
out_of_range	超出有效范围
invalid_argument	参数不合适。在标准库中，当利用string对象构造bitset时，而string中的字符不是’0’或’1’的时候，抛出该异常

异常名称	描述
range_error	计算结果超出了有意义的值域范围
overflow_error	算术计算上溢
underflow_error	算术计算下溢

案例1：

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

class Teacher
{
    public:
        Teacher(int age)  //构造函数，通过异常机制 处理错误
        {
            if (age > 100)
            {
                throw out_of_range("年龄太大");
            }
            this->age = age;
```

```
        }
    protected:
    private:
        int age;
};

int main()
{
    try
    {
        Teacher t1(102);
    }
    catch (out_of_range e)
    {

        cout << e.what() << endl;
    }

    return 0;
}
```

案例2：

```
#include <iostream>
#include <stdexcept>

using namespace std;

class Dog
{
    public:
        Dog()
        {
            parr = new int[1024*1024*100]; //4MB
        }
    private:
        int *parr;
};

int main()
{
    Dog *pDog;
    try{
        for(int i=1; i<1024; i++) //40GB!
        {
            pDog = new Dog();
            cout << i << ": new Dog 成功." << endl;
        }
    }
    catch(bad_alloc err)
    {
        cout << "new Dog 失败: " << err.what() << endl;
```

```
    }
    return 0;
}
```

10. 输入输出流

10.1 I/O流的概念和流类库的结构

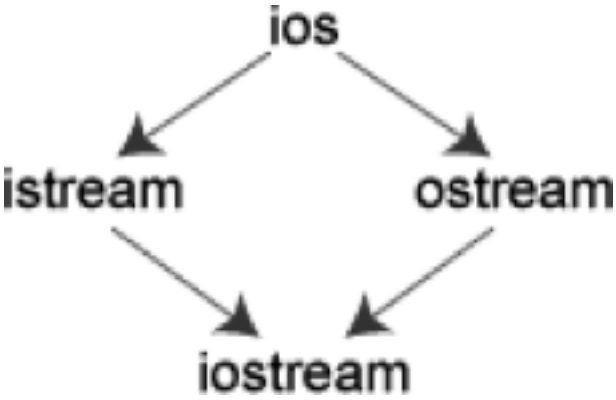
程序的输入指的是从输入文件将数据传送给程序，程序的输出指的是从程序将数据传送给输出文件。
C++输入输出包含以下三个方面的内容：
对系统指定的标准设备的输入和输出。即从键盘输入数据，输出到显示器屏幕。这种输入输出称为标准的输入输出，简称标准I/O。
以外存磁盘文件为对象进行输入和输出，即从磁盘文件输入数据，数据输出到磁盘文件。以外存文件为对象的输入输出称为文件的输入输出，简称文件I/O。
对内存中指定的空间进行输入和输出。通常指定一个字符数组作为存储空间(实际上可以利用该空间存储任何信息)。这种输入和输出称为字符串输入输出，简称串I/O。

C++的I/O对C的发展--类型安全和可扩展性

在C语言中，用printf和scanf进行输入输出，往往不能保证所输入输出的数据是可靠的安全的。在C++的输入输出中，编译系统对数据类型进行严格的检查，凡是类型不正确的数据都不可能通过编译。因此C++的I/O操作是类型安全(type safe)的。C++的I/O操作是可扩展的，不仅可以用来输入输出标准类型的数据，也可以用于用户自定义类型的数据。

C++通过I/O类库来实现丰富的I/O功能。这样使C++的输人输出明显地优于C 语言中的printf和scanf，但是也为之付出了代价，C++的I/O系统变得比较复杂，要掌握许多细节。

C++编译系统提供了用于输入输出的iostream类库。iostream这个单词是由3个部 分组成的，即i-o-stream，意为输入输出流。在iostream类库中包含许多用于输入输出的 类。常用的见表



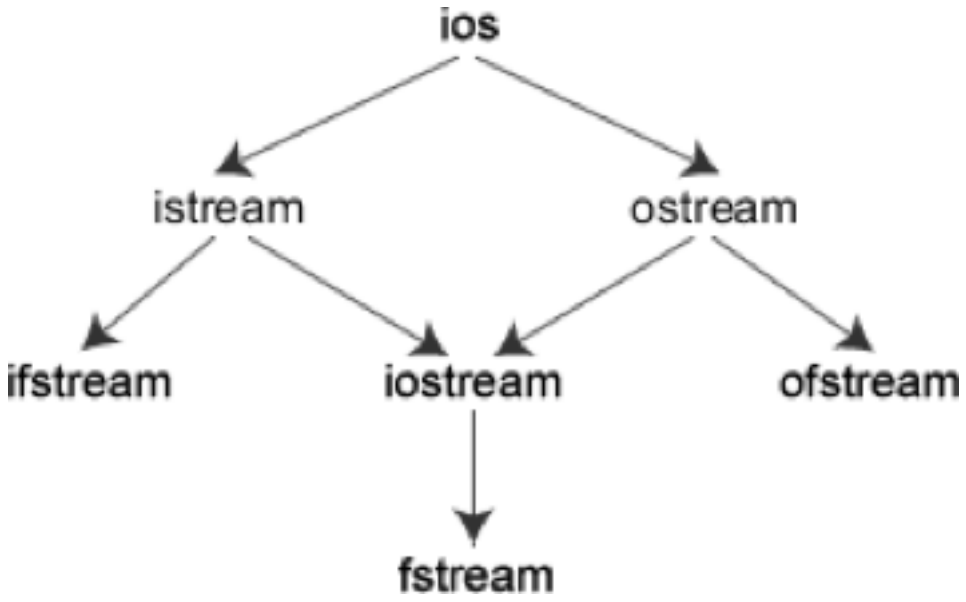
.I/O类库中的常用流类

类名	作用	在哪个头文件中声明
ios	抽象基类	iostream
istream ostream iostream	通用输入流和其他输入流的基类 通用输出流和其他输出流的基类 通用输入输出流和其他输入输出流的基类	iostream iostream iostream
ifstream ofstream fstream	输入文件流类 输出文件流类 输入输出文件流类	fstream fstream fstream
istrstream ostrstream strstream	输入字符串流类 输出字符串流类 输入输出字符串流类	strstream strstream strstream

ios是抽象基类，由它派生出istream类和ostream类，两个类名中第1个字母i和o分别代表输入(input)和输出(output)。 istream类支持输入操作，ostream类支持输出操作， iostream类支持输入输出操作。iostream类是从istream类和ostream类通过多重继承而派生的类。其继承层次见上图表示。

C++对文件的输入输出需要用ifstrcam和ofstream类，两个类名中第1个字母i和o分别代表输入和输出，第2个字母f代表文件 (file)。ifstream支持对文件的输入操

作， ofstream支持对文件的输出操作。类ifstream继承了类istream，类ofstream继承了类ostream，类fstream继承了 类iostream。见图



I/O类库中还有其他一些类，但是对于一般用户来说，以上这些已能满足需要了。

与iostream类库有关的头文件

iostream类库中不同的类的声明被放在不同的头文件中，用户在自己的程序中用#include命令包含了有关的头文件就相当于在本程序中声明了所需 要用到的类。可以换 一种说法：头文件是程序与类库的接口，iostream类库的接口分别由不同的头文件来实现。常用的有

- iostream 包含了对输入输出流进行操作所需的基本信息。
- fstream 用于用户管理的文件的I/O操作。
- strstream 用于字符串流I/O。
- stdiostream 用于混合使用C和C + +的I/O机制时，例如想将C程序转变为C++程序。
- iomanip 在使用格式化I/O时应包含此头文件。

在**iostream**头文件中定义的流对象

在 iostream 头文件中定义的类有 ios，istream，ostream，iostream，istream _withassign， ostream_withassign，iostream_withassign 等。

在**iostream**头文件中重载运算符
“<<”和“>>”本来在C++中是被定义为左位移运算符和右位移运算符的，由于在iostream头文件中对它们进行了重载， 使它们能用作标准类型数据的输入和输出运算符。所以，在用它们的程序中必须用#include命令把iostream包含到程序中。

```
#include <iostream>
```

1) >>a表示将数据放入a对象中。
2) <<a表示将a对象中存储的数据拿出。

10.2 标准I/O流

标准I/O对象:cin，cout，cerr，clog

cout流对象
cout是console output的缩写，意为在控制台（终端显示器）的输出。强调几点。
1) cout不是C++预定义的关键字，它是ostream流类的对象，在iostream中定义。 顾名思义，流是流动的数据，cout流是流向显示器的数据。cout流中的数据是用流插入运算符 “<<” 顺序加入的。如果有

```
cout<<"I "<<"study C++ "<<"very hard. << “wang bao ming ";
```

按顺序将字符串"I ", "study C++ ", "very hard."插人到cout流中，cout就将它们送到显示器，在显示器上输出字符串"I study C++ very hard."。cout流是容纳数据的载体，它并不是一个运算符。人们关心的是cout流中的内容，也就是向显示器输出什么。
2)用 “ccmt<<” 输出基本类型的数据时，可以不必考虑数据是什么类型，系统会判断数据的类型，并根据其类型选择调用与之匹配的运算符重 载函数。这个过程都是自动的，用户不必干预。如果在C语言中用prinf函数输出不同类型的数据，必须分别指定相应的输出格式符，十分麻烦，而且容易出 错。C++的I/O机制对用户来说，显然是方便而安全的。
3) cout流在内存中对应开辟了一个缓冲区，用来存放流中的数据，当向cout流插 人一个endl时，不论缓冲区是否已满，都立即输出流中所有数据，然后插入一个换行符， 并刷新流（清空缓冲区）。注意如果插人一个换行符”\n “（如cout<<a<<"\n"），则只输出和换行，而不刷新cout 流(但并不是所有编译系统都体现出这一区别）。
4) 在iostream中只对"<<"和">>"运算符用于标准类型数据的输入输出进行了重载，但未对用户声明的类型数据的输入输出 进行重载。如果用户声明了新的

类型，并希望用” <<"和">>"运算符对其进行输入输出，按照重运算符重载来做。

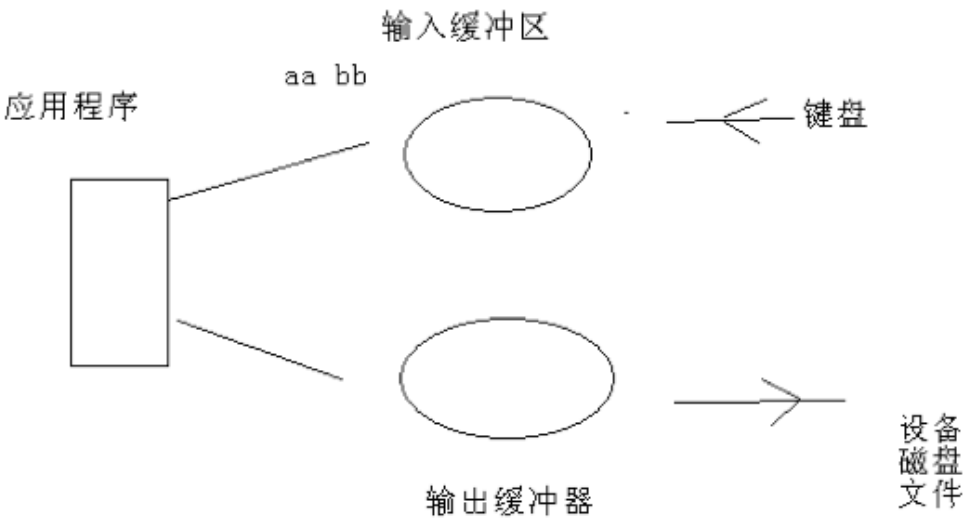
cerr流对象

cerr流对象是标准错误流，cerr流已被指定为与显示器关联。cerr的 作用是向标准错误设备(standard error device)输出有关出错信息。cerr与标准输出流cout的作用和用法差不多。但有一点不同：cout流通常是传送到显示器输出，但也可以被重定向 输出到磁盘文件，而cerr流中的信息只能在显示器输出。当调试程序时，往往不希望程序运行时的出错信息被送到其他文件，而要求在显示器上及时输出，这时 应该用cerr。cerr流中的信息是用户根据需要指定的。

clog流对象

clog流对象也是标准错误流，它是console log的缩写。它的作用和cerr相同，都是在终端显示器上显示出错信息。区别：cerr是不经过缓冲区，直接向显示器上输出有关信息，而clog中的信息存放在缓冲区中，缓冲区满后或遇endl时向显示器输出。

缓冲区概念：



1 读和写是站在应用程序的角度来说的

10.2.1 标准的输入流

标准输入流对象cin，重点掌握的函数

cin.get() //一次只能读取一个字符
cin.get(一个参数) //读一个字符
cin.get(三个参数) //可以读字符串
cin.getline()
cin.ignore()
cin.putback()

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

//2 输入字符串 你 好  遇见空格,停止接受输入
void main()
{
    char YourName[50];
    int myInt;
    long myLong;
    double myDouble;
    float myFloat;
    unsigned int myUnsigned;

    cout << "请输入一个Int: ";
    cin >> myInt;
    cout << "请输入一个Long: ";
    cin >> myLong;
    cout << "请输入一个Double: ";
    cin >> myDouble;

    cout << "请输入你的姓名: ";
    cin >> YourName;

    cout << "\n\n你输入的数是: " << endl;
    cout << "Int: \t" << myInt << endl;
    cout << "Long: \t" << myLong << endl;
    cout << "Double: \t" << myDouble << endl;
    cout << "姓名: \t" << YourName << endl;
    cout<< endl << endl;

    return 0;
}
```

```
#include <iostream>
```

```cpp
using namespace std;

//1 输入英文 ok
//2 ctr+z  会产生一个 EOF(-1)
int main()
{
    char ch;
    while( (ch= cin.get())!= EOF)
    {
        std::cout << "字符: " << ch << std::endl;
    }
    std::cout << "\n结束.\n";

    return 0;
}
```

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

//演示:读一个字符 链式编程
int main(void)
{
    char a, b, c;

    cin.get(a);
    cin.get(b);
    cin.get(c);

    cout << a << b << c<< endl;

    cout << "开始链式编程" << endl;

    cin.get(a).get(b).get(c);

    cout << a << b << c<< endl;

    return 0;
}
```

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

//演示cin.getline() 可以接受空格
int main(void)
{
    char buf1[256];
    char buf2[256];
```

```
    cout << "\n请输入你的字符串 不超过256" ;
    cin.getline(buf1, 256, '\n');
    cout << buf1 << endl;

    //
    cout << "注意: cin.getline() 和 cin >> buf2 的区别, 能不能带空格 " << endl;
    cin >> buf2 ; //流提取操作符 遇见空格 停止提取输入流
    cout << buf2 << endl;

    return 0;
}
```

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

//缓冲区实验
/*
   1 输入 "aa bb cc dd" 字符串入缓冲区
   2 通过 cin >> buf1; 提走了 aa
   3 不需要输入 可以再通过cin.getline() 把剩余的缓冲区数据提走
 */
int main()
{
    char buf1[256];
    char buf2[256];

    cout << "请输入带有空格的字符串,测试缓冲区" << endl;
    cin >> buf1;
    cout << "buf1:" << buf1 << endl;

    cout << "请输入数据..." << endl;

    //缓冲区没有数据,就等待; 缓冲区如果有数据直接从缓冲区中拿走数据
    cin.getline(buf2, 256);
    cout << "buf2:" << buf2 << endl;

    return 0;
}
```

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

// ignore
int main(void)
{
    int  intchar;
    char buf1[256];
    char buf2[256];

    cout << "请输入带有空格的字符串,测试缓冲区 aa  bbccddee " << endl;
```

```
    cin >> buf1;
    cout << "buf1:" << buf1 << endl;

    cout << "请输入数据..." << endl;
    cin.ignore(2);

    //缓冲区没有数据,就等待; 缓冲区如果有数据直接从缓冲区中拿走数据
    cin.getline(buf2, 256);
    cout << "buf2:" << buf2 << endl;

    return 0;
}
```

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

//案例:输入的整数和字符串分开处理
int main()
{
    cout << "Please, enter a number or a word: ";
    char c = cin.get();

    if ( (c >= '0') && (c <= '9') ) //输入的整数和字符串 分开处理
    {
        int n; //整数不可能 中间有空格 使用cin >>n
        cin.putback (c);
        cin >> n;
        cout << "You entered a number: " << n << '\n';
    }
    else
    {
        string str;
        cin.putback (c);
        getline (cin,str); // //字符串 中间可能有空格 使用 cin.getline();
        cout << "You entered a word: " << str << '\n';
    }

    return 0;
}
```

10.2.2 标准的输出流

标准输出流对象cout

- cout.flush()
- cout.put()
- cout.write()
- cout.width()
- cout.fill()
- cout.setf(标记)

操作符、控制符

flush
endl
oct
dec
hex
setbase
setw
setfill
setprecision

```
#include <iostream>
#include <iomanip>

using namespace std;

int main(void)
{
    cout << "hello world" << endl;
    cout.put('h').put('e').put('l').put('\n');

    cout.write("hello world", 4); //输出的长度

    char buf[] = "hello world";
    printf("\n");
    cout.write(buf, strlen(buf));

    printf("\n");
    cout.write(buf, strlen(buf) - 6);

    printf("\n");

    return 0;
```

```
}
```

```
#include <iostream>
#include <iomanip>

using namespace std;

//使用cout.setf()控制符

int main(void)
{
    //使用类成员函数
    cout << "<start>";
    cout.width(30);
    cout.fill('*');
    cout.setf(ios::showbase); //#include <iomanip>
    cout.setf(ios::internal); //设置
    cout << hex << 123 << "<End>\n";

    cout << endl;
    cout << endl;


    //使用 操作符、控制符

    cout << "<Start>"
        << setw(30)
        << setfill('*')
        << setiosflags(ios::showbase) //基数
        << setiosflags(ios::internal)
        << hex
        << 123
        << "<End>\n"
        << endl;

    return 0;
}
```

10.2.3 输出格式化

在输出数据时，为简便起见，往往不指定输出的格式，由系统根据数据的类型采取默认的格式，但有时希望数据按指定的格式输出，如要求以十六进制或八进制形式 输出一个 整数，对输出的小数只保留两位小数等。有两种方法可以达到此目的。

1）使用控制符的方法；

2）使用流对象的有关成员函数。分别叙述如下。

使用控制符的方法

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    int a;
    cout<<"input a:";
    cin>>a;
    cout<<"dec:"<<dec<<a<<endl; //以十进制形式输出整数
    cout<<"hex:"<<hex<<a<<endl; //以十六进制形式输出整数a
    cout<<"oct:"<<setbase(8)<<a<<endl; //以八进制形式输出整数a

    const char *pt="China"; //pt指向字符串"China"
    cout<<setw(10)<<pt<<endl; //指定域宽为,输出字符串
    cout<<setfill('*')<<setw(10)<<pt<<endl; //指定域宽,输出字符串,空白处以'*'填>充
    double pi=22.0/7.0; //计算pi值
    //按指数形式输出,8位小数
    cout<<setiosflags(ios::scientific)<<setprecision(8);
    cout<<"pi="<<pi<<endl; //输出pi值
    cout<<"pi="<<setprecision(4)<<pi<<endl; //改为位小数
    cout<<"pi="<<setiosflags(ios::fixed)<<pi<<endl; //改为小数形式输出

    return 0;
}
```

结果：

```
input a:16
dec:16
hex:10
oct:20
     China
*****China
pi=3.14285714e+00
pi=3.1429e+00
pi=0x1.9249249249249p+1
```

人们在输入输出时有一些特殊的要求，如在输出实数时规定字段宽度，只保留两位小数，数据向左或向右对齐等。C++提供了在输入输出流中使用的控制符(有的书中称为操纵符)

表 3.1 输入输出流的控制符

控制符	作 用
dec	设置数值的基数为10
hex	设置数值的基数为16
oct	设置数值的基数为8
setfill(c)	设置填充字符c，c可以是字符常量或字符变量
setprecision(n)	设置浮点数的精度为n位。在以一般十进制小数形式输出时，n代表有效数字。在以fixed(固定小数位数)形式和 scientific(指数)形式输出时，n为小数位数
setw(n)	设置字段宽度为n位
setiosflags(ios::fixed)	设置浮点数以固定的小数位数显示
setiosftags(ios::scientific)	设置浮点数以科学记数法(即指数形式)显示
setiosflags(ios::left)	输出数据左对齐
setiosflags(ios::right)	输出数据右对齐
setiosflags(ios::skipws)	忽略前导的空格
setiosflags(ios::uppercase)	数据以十六进制形式输出时字母以大写表示
setiosflags(ios::lowercase)	数据以十六进制形式输出时字母以小写表示
setiosflags(ios::showpos)	输出正数时给出“+”号

需要注意的是：如果使用了控制符，在程序单位的开头除了要加iostream头文件外，还要加iomanip头文件。

举例， 输出双精度数：

```
double a=123.456789012345; // 对a赋初值
```

1) cout<<a; 输出： 123.456
2) cout<<setprecision(9)<<a; 输出： 123.456789
3) cout<<setprecision(6); 恢复默认格式(精度为6)
4) cout<< setiosflags(ios::fixed); 输出： 123.456789
5) cout<<setiosflags(ios::fixed)<<setprecision(8)<<a; 输出：123.45678901
6) cout<<setiosflags(ios::scientific)<<a; 输出： 1.234568e+02
7) cout<<setiosflags(ios::scientific)<<setprecision(4)<<a; 输出：1.2346e02

下面是整数输出的例子：

```
int b=123456;  // 对b赋初值
```

1) cout<<b; 输出： 123456
2) cout<<hex<<b; 输出： 1e240
3) cout<<setiosflags(ios∷uppercase)<<b; 输出： 1E240
4) cout<<setw(10)<<b<<'，'<<b; 输出： 123456，123456
5) cout<<setfill('*')<<setw(10)<<b; 输出： **** 123456
6) cout<<setiosflags(ios∷showpos)<<b; 输出： +123456

如果在多个cout语句中使用相同的setw(n)，并使用setiosflags(ios::right)，可以实现各行数据右对齐，如果指定相同的精度，可以实现上下小数点对齐。

例如：各行小数点对齐。

```
int main( )
{
        double a=123.456,b=3.14159,c=-3214.67;

cout<<setiosflags(ios::fixed)<<setiosflags(ios::right)<<setprecision(2);
        cout<<setw(10)<<a<<endl;
        cout<<setw(10)<<b<<endl;
        cout<<setw(10)<<c<<endl;
        system("pause");
        return 0;
}
```

输出如下：

```
 123.46 (字段宽度为10，右对齐，取两位小数)
 3.14
 -3214.67
```

先统一设置定点形式输出、取两位小数、右对齐。这些设置对其后的输出均有效(除非重新设置)，而setw只对其后一个输出项有效，因此必须在输出a，b，c之前都要写setw(10)。

用流对象的成员函数控制输出格式

除了可以用控制符来控制输出格式外，还可以通过调用流对象cout中用于控制输出格式的成员函数来控制输出格式。用于控制输出格式的常用的成员函数如下：

表13.4 用于控输出格式的流成员函数

流成员函数	与之作用相同的控制符	作用
precision(n)	setprecision(n)	设置实数的精度为n位
width(n)	setw(n)	设置字段宽度为n位
fill(c)	setfill(c)	设置填充字符c
setf()	setiosflags()	设置输出格式状态，括号中应给出格式状态，内容与控制符setiosflags括号中的内容相同，如表13.5所示
unsetf()	resetioflags()	终止已设置的输出格式状态，在括号中应指定内容

流成员函数setf和控制符setiosflags括号中的参数表示格式状态，它是通过格式标志来指定的。格式标志在类ios中被定义为枚举值。因此在引用这些格式标志时要在前面加上类名ios和域运算符“::”。格式标志见表13.5。

表13.5 设置格式状态的格式标志

格式标志	作用
ios::left	输出数据在本域宽范围内向左对齐
ios::right	输出数据在本域宽范围内向右对齐
ios::internal	数值的符号位在域宽内左对齐，数值右对齐，中间由填充字符填充
ios::dec	设置整数的基数为10
ios::oct	设置整数的基数为8
ios::hex	设置整数的基数为16
ios::showbase	强制输出整数的基数(八进制数以0打头，十六进制数以0x打头)
ios::showpoint	强制输出浮点数的小点和尾数0
ios::uppercase	在以科学记数法格式E和以十六进制输出字母时以大写表示
ios::showpos	对正数显示“+”号
ios::scientific	浮点数以科学记数法格式输出
ios::fixed	浮点数以定点格式(小数形式)输出
ios::unitbuf	每次输出之后刷新所有的流
ios::stdio	每次输出之后清除stdout, stderr

```cpp
#include <iostream>
#include <iomanip>

using namespace std;

int main( )
{
    int a=21;
    cout.setf(ios::showbase);//显示基数符号(0x或)
    cout<<"dec:"<<a<<endl; //默认以十进制形式输出a
    cout.unsetf(ios::dec); //终止十进制的格式设置
    cout.setf(ios::hex); //设置以十六进制输出的状态
    cout<<"hex:"<<a<<endl; //以十六进制形式输出a
    cout.unsetf(ios::hex); //终止十六进制的格式设置
    cout.setf(ios::oct); //设置以八进制输出的状态
    cout<<"oct:"<<a<<endl; //以八进制形式输出a
    cout.unsetf(ios::oct);

    const char *pt="China"; //pt指向字符串"China"
    cout.width(10); //指定域宽为
    cout<<pt<<endl; //输出字符串
    cout.width(10); //指定域宽为
    cout.fill('*'); //指定空白处以'*'填充
    cout<<pt<<endl; //输出字符串

    double pi=22.0/7.0; //输出pi值
    cout.setf(ios::scientific); //指定用科学记数法输出
    cout<<"pi="; //输出"pi="
    cout.width(14); //指定域宽为
    cout<<pi<<endl; //输出pi值
    cout.unsetf(ios::scientific); //终止科学记数法状态
    cout.setf(ios::fixed); //指定用定点形式输出
    cout.width(12); //指定域宽为
    cout.setf(ios::showpos); //正数输出“+”号
    cout.setf(ios::internal); //数符出现在左侧
    cout.precision(6); //保留位小数
    cout<<pi<<endl; //输出pi,注意数符“+”的位置

    return 0;
}
```

结果是：

```
dec:21
hex:0x15
oct:025
    China
*****China
pi=**3.142857e+00
```

```
+***3.142857
```

对程序的几点说明：

1) 成员函数width(n)和控制符setw(n)只对其后的第一个输出项有效。如：
cout. width(6);

```
cout <<20 <<3.14<<endl;
```

输出结果为 203.14

在输出第一个输出项20时，域宽为6，因此在20前面有4个空格，在输出3.14时，width (6)已不起作用，此时按系统默认的域宽输出（按数据实际长度输出）。如果要求在输出数据时都按指定的同一域宽n输出，不能只调用一次width(n)，而必须在输出每一项前都调用一次width(n>，上面的程序中就是这样做的。

2) 在表13.5中的输出格式状态分为5组，每一组中同时只能选用一种（例如dec、hex和oct中只能选一，它们是互相排斥的）。在用成员函数setf和 控制符setiosflags设置输出格式状态后，如果想改设置为同组的另一状态，应当调用成员函数unsetf（对应于成员函数self）或 resetiosflags（对应于控制符setiosflags），先终止原来设置的状态。然后再设置其他状态，大家可以从本程序中看到这点。程序在开 始虽然没有用成员函数self和控制符setiosflags设置用dec输出格式状态，但系统默认指定为dec，因此要改变为hex或oct，也应当先用unsetf 函数终止原来设置。如果删去程序中的第7行和第10行，虽然在第8行和第11行中用成员函数setf设置了hex和oct格式，由于未终止dec格式，因 此hex和oct的设置均不起作用，系统依然以十进制形式输出。

同理，程序倒数第8行的unsetf 函数的调用也是不可缺少的。

3) 用setf 函数设置格式状态时，可以包含两个或多个格式标志，由于这些格式标志在ios类中被定义为枚举值，每一个格式标志以一个二进位代表，因此可以用位或运算符“|”组合多个格式标志。如倒数第5、第6行可以用下面一行代替：

```
cout.setf(ios::internal | ios::showpos); //包含两个状态标志，用"|"组合
```

可以看到：对输出格式的控制，既可以用控制符(如例13.2)，也可以用cout流的有关成员函数(如例13.3)，二者的作用是相同的。控制符是在头文件

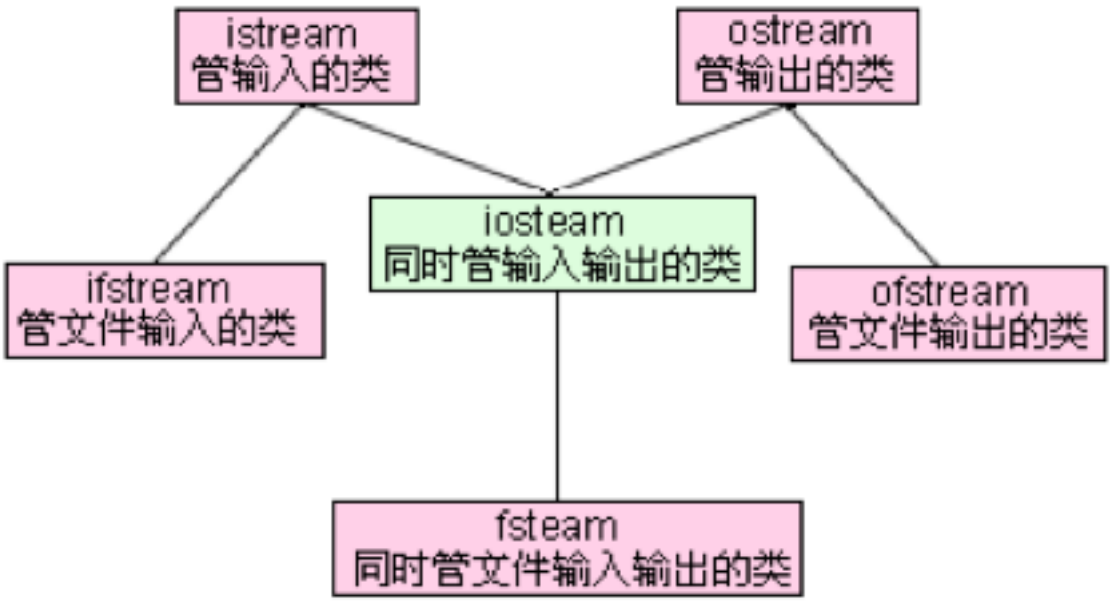
iomanip中定义的，因此用控制符时，必须包含iomanip头文件。cout流的成员函数是在头文件iostream 中定义的，因此只需包含头文件iostream，不必包含iomanip。许多程序人员感到使用控制符方便简单，可以在一个cout输出语句中连续使用多种控制符。

10.3 文件IO

10.3.1 文件流类和文件流对象

输入输出是以系统指定的标准设备（输入设备为键盘，输出设备为显示器）为对象的。在实际应用中，常以磁盘文件作为对象。即从磁盘文件读取数据，将数据输出到磁盘文件。

和文件有关系的输入输出类主要在fstream.h这个头文件中被定义，在这个头文件中主要被定义了三个类，由这三个类控制对文件的各种输入输出操 作，他们分别是ifstream、ofstream、fstream，其中fstream类是由iostream类派生而来，他们之间的继承关系见下图所 示。



由于文件设备并不像显示器屏幕与键盘那样是标准默认设备，所以它在fstream.h头文件中是没有像cout那样预先定义的全局对象，所以我们必须自己定义一个该类的对象。

ifstream类，它是从istream类派生的，用来支持从磁盘文件的输入。
ofstream类，它是从ostream类派生的，用来支持向磁盘文件的输出。
fstream类，它是从iostream类派生的，用来支持对磁盘文件的输入输出。

10.3.2 文件的打开与关闭

打开文件

所谓打开(open)文件是一种形象的说法，如同打开房门就可以进入房间活动一样。 打开文件是指在文件读写之前做必要的准备工作，包括：
1）为文件流对象和指定的磁盘文件建立关联，以便使文件流流向指定的磁盘文件。
2）指定文件的工作方式，如，该文件是作为输入文件还是输出文件，是ASCII文件还是二进制文件等。
以上工作可以通过两种不同的方法实现。

1) 调用文件流的成员函数open。如

```
ofstream outfile;  //定义ofstream类(输出文件流类)对象outfile
outfile.open("f1.dat",ios::out);  //使文件流与f1.dat文件建立关联
```

第2行是调用输出文件流的成员函数open打开磁盘文件f1.dat，并指定它为输出文件， 文件流对象outfile将向磁盘文件f1.dat输出数据。ios::out是I/O模式的一种，表示以输出方式打开一个文件。或者简单地说，此时f1.dat是一个输出文件，接收从内存输出的数据。

调用成员函数open的一般形式为：
文件流对象.open(磁盘文件名, 输入输出方式);
磁盘文件名可以包括路径，如"c:\new\\f1.dat"，如缺省路径，则默认为当前目录下的文件。

2) 在定义文件流对象时指定参数

在声明文件流类时定义了带参数的构造函数，其中包含了打开磁盘文件的功能。因此，可以在定义文件流对象时指定参数，调用文件流类的构造函数来实现打开文件的功能。如

ostream outfile("f1.dat",ios::out); 一般多用此形式，比较方便。作用与open函数相同。

输入输出方式是在ios类中定义的，它们是枚举常量，有多种选择，见表13.6。

表13.6 文件输入输出方式设置值

方式	作用
ios::in	以输入方式打开文件
ios::out	以输出方式打开文件（这是默认方式），如果已有此名字的文件，则将其原有内容全部清除
ios::app	以输出方式打开文件，写入的数据添加在文件末尾
ios::ate	打开一个已有的文件，文件指针指向文件末尾
ios: :trunc	打开一个文件，如果文件已存在，则删除其中全部数据，如文件不存在，则建立新文件。如已指定了 ios::out 方式，而未指定ios: :app，ios::ate，ios: :in，则同时默认此方式
ios:: binary	以二进制方式打开一个文件，如不指定此方式则默认为ASCII方式
ios::nocreate	打开一个已有的文件，如文件不存在，则打开失败。nocrcate的意思是不建立新文件
ios:: noreplace	如果文件不存在则建立新文件，如果文件已存在则操作失败，replace 的意思是不更新原有文件
ios::in | ios::out	以输入和输出方式打开文件，文件可读可写
ios:: out | ios::binary	以二进制方式打开一个输出文件
ios::in | ios::binar	以二进制方式打开一个输入文件

几点说明：

1) 新版本的I/O类库中不提供ios::nocreate和ios::noreplace。

2) 每一个打开的文件都有一个文件指针，该指针的初始位置由I/O方式指定，每次读写都从文件指针的当前位置开始。每读入一个字节，指针就后移一个字节。当文 件指针移到最后，就会遇到文件结束EOF（文件结束符也占一个字节，其值为-1)，此时流对象的成员函数eof的值为非0值(一般设为1)，表示文件结束 了。

3) 可以用“位或”运算符“|”对输入输出方式进行组合，如表13.6中最后3行所示那样。还可以举出下面一些例子：

ios::in | ios:: noreplace //打开一个输入文件，若文件不存在则返回打开失败的信息

ios::app | ios::nocreate //打开一个输出文件，在文件尾接着写数据，若文

件不存在，则返回打开失败的信息
ios::out | ios::noreplace //打开一个新文件作为输出文件，如果文件已存在则返回打开失败的信息
ios::in | ios::out | ios::binary //打开一个二进制文件，可读可写

但不能组合互相排斥的方式，如 ios::nocreate | ios::noreplace。

4) 如果打开操作失败，open函数的返回值为0(假)，如果是用调用构造函数的方式打开文件的，则流对象的值为0。可以据此测试打开是否成功。如

```
if(outfile.open("f1.bat", ios::app) ==0)
    cout <<"open error";
```

或

```
if( !outfile.open("f1.bat", ios::app) )
    cout <<"open error";
```

关闭文件

在对已打开的磁盘文件的读写操作完成后，应关闭该文件。关闭文件用成员函数close。如

```
outfile.close( ); //将输出文件流所关联的磁盘文件关闭
```

所谓关闭，实际上是解除该磁盘文件与文件流的关联，原来设置的工作方式也失效，这样，就不能再通过文件流对该文件进行输入或输出。此时可以将文件流与其他磁盘文件建立关联，通过文件流对新的文件进行输入或输出。如

```
outfile.open("f2.dat",ios::app|ios::nocreate);
```

此时文件流outfile与f2.dat建立关联，并指定了f2.dat的工作方式。

10.3.3 C++对ASCII文件的读写操作

如果文件的每一个字节中均以ASCII代码形式存放数据,即一个字节存放一个字符,这个文件就是ASCII文件(或称字符文件)。程序可以从ASCII文件中读入若干个字符,也可以向它输出一些字符。

1)　用流插入运算符“<<”和流提取运算符“>>”输入输出标准类型的数据。“<<”和“ >>”都已在iostream中被重载为能用于ostream和istream类对象的标准类型的输入输出。由于ifstream和 ofstream分别是ostream和istream类的派生类；因此它们从ostream和istream类继承了公用的重载函数，所以在对磁盘文件的操作中，可以通过文件流对象和流插入运算符“<<”及 流提取运算符“>>”实现对磁盘文件的读写，如同用cin、cout和<<、>>对标准设备进行读写一样。

2)　用文件流的put、get、geiline等成员函数进行字符的输入输出，：用C++流成员函数put输出单个字符、C++ get()函数读入一个字符和C++ getline()函数读入一行字符。

```
#include <iostream>
#include <fstream>

using namespace std;


int main(void)
{
    char* fname = "c:/aaaa.txt";

    ofstream fout(fname, ios::app); //建一个 输出流对象 和文件关联;
    if (!fout)
    {
        cout << "打开文件失败" << endl;
        return ;
    }
    fout << "hello....111" << endl;
    fout << "hello....222" << endl;
    fout << "hello....333" << endl;
    fout.close();

    //读文件
    ifstream fin(fname); //建立一个输入流对象 和文件关联
    char ch;

    while (fin.get(ch))
    {
    cout <<ch ;
    }
    fin.close();

    return 0;
}
```

10.3.4 C++对二进制文件的读写操作

```cpp
#include <iostream>
#include <fstream>
using namespace std;

class Teacher
{
    public:
        Teacher()
        {
            age = 33;
            strcpy(name, "");
        }
        Teacher(int _age,const char *_name)
        {
            age = _age;
            strcpy(name, _name);
        }
        void printT()
        {
            cout << "age:" << age << "name:" << name <<endl;
        }
    protected:
    private:
        int  age;
        char name[32];
};

int main()
{
    const char* fname = "./11a.dat";
    ofstream fout(fname, ios::binary); //建一个 输出流对象 和文件关联;
    if (!fout)
    {
        cout << "打开文件失败" << endl;
        return -1;
    }
    Teacher t1(31, "t31");
    Teacher t2(32, "t32");
    fout.write((char *)&t1, sizeof(Teacher));
    fout.write((char *)&t2, sizeof(Teacher));
    fout.close();


    //
    ifstream fin(fname); //建立一个输入流对象 和文件关联
    Teacher tmp;

    fin.read( (char*)&tmp,sizeof(Teacher) );
    tmp.printT();

    fin.read( (char*)&tmp,sizeof(Teacher) );
    tmp.printT();

    fin.close();
```

```
    return 0;
}
```

10.3.5 作业及参考答案

1 编程实现以下数据输入/输出：
(1)以左对齐方式输出整数,域宽为12。
(2)以八进制、十进制、十六进制输入/输出整数。
(3)实现浮点数的指数格式和定点格式的输入/输出,并指定精度。
(4)把字符串读入字符型数组变量中,从键盘输入,要求输入串的空格也全部读入,以回车符结束。
(5)将以上要求用流成员函数和操作符各做一遍。
2编写一程序，将两个文件合并成一个文件。
3编写一程序，统计一篇英文文章中单词的个数与行数。
4编写一程序，将C++源程序每行前加上行号与一个空格。
5编写一程序，输出 ASCII码值从20到127的ASCII码字符表，格式为每行10个。

参考答案
第一题：

```
//ios类成员函数实现
#include<iostream>
#include<iomanip>
using namespace std;

int main()
{
    long a=234;
    double b=2345.67890;
    char c[100];

    cout.fill('*');
    cout.flags(ios_base::left);
    cout.width(12);
    cout<<a<<endl;
    cout.fill('*');
    cout.flags(ios::right);
    cout.width(12);
    cout<<a<<endl;
    cout.flags(ios::hex);
    cout<<234<<'\t';
    cout.flags(ios::dec);
    cout<<234<<'\t';
```

```
    cout.flags(ios::oct);
    cout<<234<<endl;
    cout.flags(ios::scientific);
    cout<<b<<'\t';
    cout.flags(ios::fixed);
    cout<<b<<endl;
    cin.get(c,99);
    cout<<c<<endl;

    return 0;
}

//操作符实现
#include<iostream>
#include<iomanip>
using namespace std;

int main()
{
    long a=234;
    double b=2345.67890;
    char c[100];

    cout<<setfill('*');
    cout<<left<<setw(12)<<a<<endl;
    cout<<right<<setw(12)<<a<<endl;
    cout<<hex<<a<<'\t'<<dec<<a<<'\t'<<oct<<a<<endl;
    cout<<scientific<<b<<'\t'<<fixed<<b<<endl;

    return 0;
}
```

第二题：

```
#include<iostream>
#include<fstream>

using namespace std;

int main()
{
    int i=1;
    char c[1000];

    ifstream ifile1("D:\\1.cpp");
    ifstream ifile2("D:\\2.cpp");
    ofstream ofile("D:\\3.cpp");

    while(!ifile1.eof()){
        ifile1.getline(c,999);
        ofile<<c<<endl;
    }
```

```
    while(!ifile2.eof()){
        ifile2.getline(c,999);
        ofile<<c<<endl;
    }

    ifile1.close();
    ifile2.close();
    ofile.close();

    return 0;
}
```

第三題：

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

bool isalph(char);

int main()
{
    ifstream ifile("C:\\daily.doc");

    char text[1000];
    bool inword=false;
    int rows=0,words=0;
    int i;

    while(!ifile.eof()) {
        ifile.getline(text,999);
        rows++;
        i=0;

        while(text[i]!=0){
            if(!isalph(text[i]))
                inword=false;
            else if(isalph(text[i]) && inword==false){
                words++;
                inword=true;
            }
            i++;
        }
    }

    cout<<"rows= " << rows <<endl;
    cout<<"words= "<< words <<endl;
    ifile.close();

    return 0;
}
```

```cpp
bool isalph(char c)
{
    return ((c>='A' && c<='Z') || (c>='a' && c<='z'));
}
```

第四题：

```cpp
#include<iostream>
#include<fstream>
using namespace std;

int main()
{
    int i=1;
    char c[1000];

    ifstream ifile("D:\\1.cpp");
    ofstream ofile("D:\\2.cpp");

    while(!ifile.eof()){
        ofile<<i++<<": ";
        ifile.getline(c,999);
        ofile<<c<<endl;
    }
    ifile.close();
    ofile.close();

    return 0;
}
```

第五题：

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

int main()
{
    int i,l;
    for(i=32;i<127;i++){
        cout<<char(i)<<" ";
        l++;
        if(l%10==0)cout<<endl;
    }
    cout<<endl;

    return 0;
}
```