1.C++模板

1.1 模板概论

c++提供了函数模板(function template.)所谓函数模板，**实际上是建立一个通用函数，其函数类型和形参类型不具体制定，用一个虚拟的类型来代表。这个通用函数就成为函数模板。**凡是函数体相同的函数都可以用这个模板代替，不必定义多个函数，只需在模板中定义一次即可。在调用函数时系统会根据实参的类型来取代模板中的虚拟类型，从而实现不同函数的功能。

- c++提供两种模板机制:**函数模板**和**类模板**
- 类属 - 类型参数化，又称参数模板

总结:

- 模板把函数或类要处理的数据类型参数化，表现为参数的多态性，成为类属。
- 模板用于表达逻辑结构相同，但具体数据元素类型不同的数据对象的通用行为。

1.2 函数模板

1.2.1 什么是函数模板？

```
//交换 int 数据
void SwapInt(int& a,int& b){
    int temp = a;
    a = b;
    b = temp;
}

//交换 char 数据
void SwapChar(char& a,char& b){
    char temp = a;
```

```
    a = b;
    b = temp;
}
//问题：如果我要交换 double 类型数据，那么还需要些一个 double 类型数据交换的函数
//繁琐，写的函数越多，当交换逻辑发生变化的时候，所有的函数都需要修改，无形当中增加了代码的维护难度

//如果能把类型作为参数传递进来就好了，传递 int 就是 Int 类型交换，传递 char 就是 char 类型交换
//我们有一种技术，可以实现类型的参数化---函数模板


//class 和 typename 都是一样的，用哪个都可以
template<class T>
void MySwap(T& a,T& b){
    T temp = a;
    a = b;
    b = temp;
}

void test01(){

    int a = 10;
    int b = 20;
    cout << "a:" << a << " b:" << b << endl;
    //1. 这里有个需要注意点，函数模板可以自动推导参数的类型
    MySwap(a,b);
    cout << "a:" << a << " b:" << b << endl;

    char c1 = 'a';
    char c2 = 'b';
    cout << "c1:" << c1 << " c2:" << c2 << endl;
    //2. 函数模板可以自动类型推导，那么也可以显式指定类型
    MySwap<char>(c1, c2);
    cout << "c1:" << c1 << " c2:" << c2 << endl;
}
```

用模板是为了实现泛型，可以减轻编程的工作量，增强函数的重用性。

1.2.2 课堂练习

使用函数模板实现对 char 和 int 类型数组进行排序?

```
//模板打印函数
template<class T>
void PrintArray(T arr[],int len){
    for (int i = 0; i < len;i++){
        cout << arr[i] << " ";
    }
    cout << endl;
}

//模板排序函数
template<class T>
void MySort(T arr[],int len){

    for (int i = 0; i < len;i++){
        for (int j = len - 1; j > i;j--){
            if (arr[j] > arr[j - 1]){
                T temp = arr[j - 1];
                arr[j - 1] = arr[j];
                arr[j] = temp;
            }
        }
    }

}

void test(){

    //char 数组
    char tempChar[] = "aojtifysn";
    int charLen = strlen(tempChar);

    //int 数组
    int tempInt[] = {7,4,2,9,8,1};
    int intLen = sizeof(tempInt) / sizeof(int);

    //排序前 打印函数
    PrintArray(tempChar, charLen);
```

```
    PrintArray(tempInt, intLen);
    //排序
    MySort(tempChar, charLen);
    MySort(tempInt, intLen);
    //排序后打印
    PrintArray(tempChar, charLen);
    PrintArray(tempInt, intLen);
}
```

1.3 函数模板和普通函数区别

- 函数模板不允许自动类型转化
- 普通函数能够自动进行类型转化

```
//函数模板
template<class T>
T MyPlus(T a, T b){
    T ret = a + b;
    return ret;
}

//普通函数
int MyPlus(int a,char b){
    int ret = a + b;
    return ret;
}

void test02(){

    int a = 10;
    char b = 'a';

    //调用函数模板，严格匹配类型
    MyPlus(a, a);
    MyPlus(b, b);
    //调用普通函数
    MyPlus(a, b);
    //调用普通函数  普通函数可以隐式类型转换
    MyPlus(b, a);
```

```
    //结论：
    //函数模板不允许自动类型转换，必须严格匹配类型
    //普通函数可以进行自动类型转换
}
```

1.4 函数模板和普通函数在一起调用规则

- c++编译器优先考虑普通函数
- 可以通过空模板实参列表的语法限定编译器只能通过模板匹配
- 函数模板可以像普通函数那样可以被重载
- 如果函数模板可以产生一个更好的匹配，那么选择模板

```
//函数模板
template<class T>
T MyPlus(T a, T b){
    T ret = a + b;
    return ret;
}

//普通函数
int MyPlus(int a, int b){
    int ret = a + b;
    return ret;
}

void test03(){
    int a = 10;
    int b = 20;
    char c = 'a';
    char d = 'b';
    //如果函数模板和普通函数都能匹配，c++编译器优先考虑普通函数
    cout << MyPlus(a, b) << endl;
    //如果我必须要调用函数模板,那么怎么办?
    cout << MyPlus<>(a, b) << endl;
    //此时普通函数也可以匹配，因为普通函数可以自动类型转换
    //但是此时函数模板能够有更好的匹配
```

```
    //如果函数模板可以产生一个更好的匹配，那么选择模板
    cout << MyPlus(c,d);
}

//函数模板重载
template<class T>
T MyPlus(T a, T b, T c){
    T ret = a + b + c;
    return ret;
}

void test04(){

    int a = 10;
    int b = 20;
    int c = 30;
    cout << MyPlus(a, b, c) << endl;
    //如果函数模板和普通函数都能匹配，c++编译器优先考虑普通函数
}
```

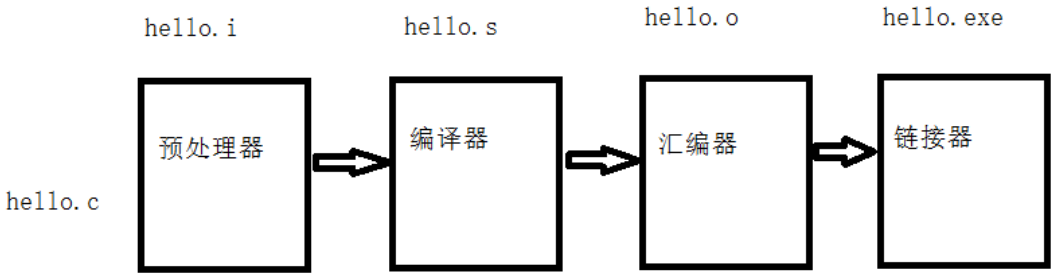
1.5 模板机制剖析

思考:为什么函数模板可以和普通函数放在一起?c++编译器是如何实现函数模板机制的?

1.5.1 编译过程

hello.cpp 程序是高级 c 语言程序，这种程序易于被人读懂。为了在系统上运行 hello.c 程序，每一条 c 语句都必须转化为低级的机器指令。然后将这些机器指令打包成可执行目标文件格式，并以二进制形式存储于磁盘中。

预处理(Pre-processing) -> 编译(Compiling) ->汇编(Assembling) -> 链接(Linking)



hello.c经过预处理器，将宏展开，生成的文件hello.i
hello.i经过编译器，将文件编译成汇编语言，生成文件为hello.s
hello.s经过汇编器，将文件编译成目标文件hello.o（win下为hello.obj）
hello.o经过链接器，将文件编译成可执行文件

1.5.2 模板实现机制

函数模板机制结论：

- **编译器并不是把函数模板处理成能够处理任何类型的函数**
- **函数模板通过具体类型产生不同的函数**
- **编译器会对函数模板进行两次编译，在声明的地方对模板代码本身进行编译，在调用的地方对参数替换后的代码进行编译。**

1.6 模板的局限性

假设有如下模板函数：

```
template<class T>
void f(T a, T b)
{ ... }
```

如果代码实现时定义了赋值操作 a = b，但是 T 为数组，这种假设就不成立了

同样，如果里面的语句为判断语句 if(a>b),但 T 如果是结构体，该假设也不成立，另外如果是传入的数组，数组名为地址，因此它比较的是地址，而这也不是我们所希望的操作。

总之，编写的模板函数很可能无法处理某些类型，另一方面，有时候通用化是有意义的，但 C++语法不允许这样做。为了解决这种问题，可以提供模板的重载，为这些特定的类型提供具体化的模板。

```
class Person
{
public:
    Person(string name, int age)
    {
        this->mName = name;
        this->mAge = age;
    }
    string mName;
    int mAge;
};

//普通交换函数
template <class T>
void mySwap(T &a,T &b)
{
    T temp = a;
    a = b;
    b = temp;
}
//第三代具体化，显示具体化的原型和定意思以template<>开头，并通过名称来指出类型
//具体化优先于常规模板
template<>void  mySwap<Person>(Person &p1, Person &p2)
{
    string nameTemp;
    int ageTemp;

    nameTemp = p1.mName;
    p1.mName = p2.mName;
    p2.mName = nameTemp;

    ageTemp = p1.mAge;
    p1.mAge = p2.mAge;
    p2.mAge = ageTemp;

}
```

```
void test()
{
    Person P1("Tom", 10);
    Person P2("Jerry", 20);

    cout << "P1 Name = " << P1.mName << " P1 Age = " << P1.mAge << endl;
    cout << "P2 Name = " << P2.mName << " P2 Age = " << P2.mAge << endl;
    mySwap(P1, P2);
    cout << "P1 Name = " << P1.mName << " P1 Age = " << P1.mAge << endl;
    cout << "P2 Name = " << P2.mName << " P2 Age = " << P2.mAge << endl;
}
```

1.7 类模板

1.7.1 类模板基本概念

类模板和函数模板的定义和使用类似，我们已经进行了介绍。有时，有两个或多个类，其功能是相同的，仅仅是数据类型不同。

- 类模板用于实现类所需数据的类型参数化

```
template<class NameType, class AgeType>
class Person
{
public:
    Person(NameType name, AgeType age)
    {
        this->mName = name;
        this->mAge = age;
    }
    void showPerson()
    {
        cout << "name: " << this->mName << " age: " << this->mAge << endl;
    }
public:
    NameType mName;
    AgeType mAge;
};

void test01()
```

```
{
    //Person P1("德玛西亚",18); // 类模板不能进行类型自动推导
    Person<string, int>P1("德玛西亚", 18);
    P1.showPerson();
}
```

1.7.2 类模板做函数参数

```
//类模板
template<class NameType, class AgeType>
class Person{
public:
    Person(NameType name, AgeType age){
        this->mName = name;
        this->mAge = age;
    }
    void PrintPerson(){
        cout << "Name:" << this->mName << " Age:" << this->mAge << endl;
    }
public:
    NameType mName;
    AgeType mAge;
};

//类模板做函数参数
void DoBussiness(Person<string,int>& p){
    p.mAge += 20;
    p.mName += "_vip";
    p.PrintPerson();
}

int main(){

    Person<string, int> p("John", 30);
    DoBussiness(p);

    system("pause");
    return EXIT_SUCCESS;
}
```

1.7.3 类模板派生普通类

```
//类模板
template<class T>
class MyClass{
public:
    MyClass(T property){
        this->mProperty = property;
    }
public:
    T mProperty;
};

//子类实例化的时候需要具体化的父类，子类需要知道父类的具体类型是什么样的
//这样 c++编译器才能知道给子类分配多少内存

//普通派生类
class SubClass : public MyClass<int>{
public:
    SubClass(int b) : MyClass<int>(20){
        this->mB = b;
    }
public:
    int mB;
};
```

1.7.4 类模板派生类模板

```
//父类类模板
template<class T>
class Base
{
    T m;
};
template<class T >
class Child2 : public Base<double>  //继承类模板的时候，必须要确定基类的大小
{
public:
    T mParam;
};
```

```
void test02()
{
    Child2<int> d2;
}
```

1.7.5 类模板类内实现

```
template<class NameType, class AgeType>
class Person
{
public:
    Person(NameType name, AgeType age)
    {
        this->mName = name;
        this->mAge = age;
    }
    void showPerson()
    {
        cout << "name: " << this->mName << " age: " << this->mAge << endl;
    }
public:
    NameType mName;
    AgeType mAge;
};

void test01()
{
    //Person P1("德玛西亚",18); // 类模板不能进行类型自动推导
    Person<string, int>P1("德玛西亚", 18);
    P1.showPerson();
}
```

1.7.6 类模板类外实现

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
#include<string>
using namespace std;

template<class T1, class T2>
```

```
class Person{
public:
    Person(T1 name, T2 age);
    void showPerson();

public:
    T1 mName;
    T2 mAge;
};


//类外实现
template<class T1, class T2>
Person<T1, T2>::Person(T1 name, T2 age){
    this->mName = name;
    this->mAge = age;
}


template<class T1, class T2>
void Person<T1, T2>::showPerson(){
    cout << "Name:" << this->mName << " Age:" << this->mAge << endl;
}

void test()
{
    Person<string, int> p("Obama", 20);
    p.showPerson();
}

int main(){

    test();

    system("pause");
    return EXIT_SUCCESS;
}
```

1.7.7 类模板头文件和源文件分离问题

Person.hpp

```
#pragma once

template<class T1,class T2>
class Person{
public:
    Person(T1 name,T2 age);
    void ShowPerson();
public:
    T1 mName;
    T2 mAge;
};

template<class T1, class T2>
Person<T1, T2>::Person(T1 name, T2 age){
    this->mName = name;
    this->mAge = age;
}

template<class T1, class T2>
void Person<T1, T2>::ShowPerson(){
    cout << "Name:" << this->mName << " Age:" << this->mAge << endl;
}
```

main.cpp

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
using namespace std;
#include<string>
#include"Person.hpp"

//模板二次编译
//编译器编译源码 逐个编译单元编译的

int main(){

    Person<string, int> p("Obama", 20);
    p.ShowPerson();


    system("pause");
    return EXIT_SUCCESS;
}
```

结论: 案例代码在 qt 编译器顺利通过编译并执行，但是在 Linux 和 vs 编辑器下如果只包含头文件，那么会报错链接错误，需要包含 cpp 文件，但是如果类模板中有友元类，那么编译失败!

解决方案: 类模板的声明和实现放到一个文件中，我们把这个文件命名为.hpp(这个是个约定的规则，并不是标准，必须这么写).

原因:

- **类模板需要二次编译，在出现模板的地方编译一次，在调用模板的地方再次编译。**
- **C++编译规则为独立编译。**

1.7.8 模板类碰到友元函数

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
using namespace std;
#include <string>

template<class T1, class T2> class Person;
//告诉编译器这个函数模板是存在
template<class T1, class T2> void PrintPerson2(Person<T1, T2>& p);

//友元函数在类内实现
template<class T1, class T2>
class Person{
    //1. 友元函数在类内实现
    friend void PrintPerson(Person<T1, T2>& p){
        cout << "Name:" << p.mName << " Age:" << p.mAge << endl;
    }

    //2.友元函数类外实现
    //告诉编译器这个函数模板是存在
    friend void PrintPerson2<>(Person<T1, T2>& p);

    //3. 类模板碰到友元函数模板
    template<class U1, class U2>
    friend void PrintPerson(Person<U1, U2>& p);
```

```
public:
    Person(T1 name, T2 age){
        this->mName = name;
        this->mAge = age;
    }
    void showPerson(){
        cout << "Name:" << this->mName << " Age:" << this->mAge << endl;
    }
private:
    T1 mName;
    T2 mAge;
};

void test01()
{
    Person <string, int>p("Jerry", 20);
    PrintPerson(p);
}


// 类模板碰到友元函数
//友元函数类外实现  加上<>空参数列表，告诉编译去匹配函数模板
template<class T1 , class T2>
void PrintPerson2(Person<T1, T2>& p)
{
    cout << "Name2:" << p.mName << " Age2:" << p.mAge << endl;
}

void  test02()
{
    Person <string, int>p("Jerry", 20);
    PrintPerson2(p);   //不写可以编译通过，写了之后，会找PrintPerson2的普通函数调用，
因为写了普通函数PrintPerson2的声明
}

int main(){

    //test01();
    test02();
    system("pause");
    return EXIT_SUCCESS;
}
```

1.8 类模板的应用

设计一个数组模板类(MyArray),完成对不同类型元素的管理

```
#pragma once
template<class T>
class MyArray
{
public:
	explicit MyArray(int capacity)
	{
		this->m_Capacity = capacity;
		this->m_Size = 0;
		// 如果T是对象，那么这个对象必须提供默认的构造函数
		pAddress = new T[this->m_Capacity];
	}

	//拷贝构造
	MyArray(const MyArray & arr)
	{
		this->m_Capacity = arr.m_Capacity;
		this->m_Size = arr.m_Size;
		this->pAddress = new T[this->m_Capacity];
		for (int i = 0; i < this->m_Size;i++)
		{
			this->pAddress[i] = arr.pAddress[i];
		}
	}

	//重载[] 操作符  arr[0]
	T& operator [](int index)
	{
		return this->pAddress[index];
	}
	//尾插法
	void Push_back(const T & val)
	{
		if (this->m_Capacity == this->m_Size)
		{
			return;
		}
		this->pAddress[this->m_Size] = val;
```

```
        this->m_Size++;
    }
    void Pop_back()
    {
        if (this->m_Size == 0)
        {
            return;
        }
        this->m_Size--;
    }
    int  getSize()
    {
        return this->m_Size;
    }
    //析构
    ~MyArray()
    {
        if (this->pAddress != NULL)
        {
            delete[] this->pAddress;
            this->pAddress = NULL;
            this->m_Capacity = 0;
            this->m_Size = 0;
        }
    }

private:
    T * pAddress;  //指向一个堆空间，这个空间存储真正的数据
    int m_Capacity; //容量
    int m_Size;   // 大小
};
```

测试代码：

```
class Person{
public:
    Person(){}
    Person(string name, int age){
        this->mName = name;
        this->mAge = age;
    }
public:
    string mName;
```

```
    int mAge;
};


void PrintMyArrayInt(MyArray<int>& arr){
    for (int i = 0; i < arr.getSize(); i++){
        cout << arr[i] << " ";
    }
    cout << endl;
}

void PrintMyPerson(MyArray<Person>& personArr)
{
    for (int i = 0; i < personArr.getSize(); i++){
        cout << "姓名：" << personArr[i].mName << " 年龄： " << personArr[i].mAge << endl;
    }

}

    MyArray<int> myArrayInt(10);
    for (int i = 0; i < 9; i++)
    {
        myArrayInt.Push_back(i);
    }
    myArrayInt.Push_back(100);
    PrintMyArrayInt(myArrayInt);

    MyArray<Person> myArrayPerson(10);
    Person p1("德玛西亚", 30);
    Person p2("提莫", 20);
    Person p3("孙悟空",18);
    Person p4("赵信", 15);
    Person p5("赵云", 24);
    myArrayPerson.Push_back(p1);
    myArrayPerson.Push_back(p2);
    myArrayPerson.Push_back(p3);
    myArrayPerson.Push_back(p4);
    myArrayPerson.Push_back(p5);
```

2.C++类型转换

类型转换(cast)是将一种数据类型转换成另一种数据类型。例如，如果将一个整型值赋给一个浮点类型的变量，编译器会暗地里将其转换成浮点类型。

转换是非常有用的，但是它也会带来一些问题，比如在转换指针时，我们很可能将其转换成一个比它更大的类型，但这可能会破坏其他的数据。

应该小心类型转换，因为转换也就相当于对编译器说：忘记类型检查，把它看做其他的类型。

一般情况下，尽量少的去使用类型转换，除非用来解决非常特殊的问题。

> 无论什么原因，任何一个程序如果使用很多类型转换都值得怀疑.

标准 c++提供了一个显示的转换的语法，来替代旧的 C 风格的类型转换。

使用 C 风格的强制转换可以把想要的任何东西转换成我们需要的类型。那为什么还需要一个新的 C++类型的强制转换呢?

新类型的强制转换可以提供更好的控制强制转换过程，允许控制各种不同种类的强制转换。C++风格的强制转换其他的好处是，它们能更清晰的表明它们要干什么。程序员只要扫一眼这样的代码，就能立即知道一个强制转换的目的。

2.1 静态转换(static_cast)

- 用于类层次结构中基类（父类）和派生类（子类）之间指针或引用的转换。
  - 进行上行转换（把派生类的指针或引用转换成基类表示）是安全的；

- 进行下行转换 (把基类指针或引用转换成派生类表示) 时，由于没有动态类型检查，所以是不安全的。

- 用于基本数据类型之间的转换，如把 int 转换成 char，把 char 转换成 int。这种转换的安全性也要开发人员来保证。

```
class Animal{};
class Dog : public Animal{};
class Other{};

//基础数据类型转换
void test01(){
    char a = 'a';
    double b = static_cast<double>(a);
}

//继承关系指针互相转换
void test02(){
    //继承关系指针转换
    Animal* animal01 = NULL;
    Dog* dog01 = NULL;
    //子类指针转成父类指针,安全
    Animal* animal02 = static_cast<Animal*>(dog01);
    //父类指针转成子类指针，不安全
    Dog* dog02 = static_cast<Dog*>(animal01);
}

//继承关系引用相互转换
void test03(){

    Animal ani_ref;
    Dog dog_ref;
    //继承关系指针转换
    Animal& animal01 = ani_ref;
    Dog& dog01 = dog_ref;
    //子类指针转成父类指针,安全
    Animal& animal02 = static_cast<Animal&>(dog01);
    //父类指针转成子类指针，不安全
    Dog& dog02 = static_cast<Dog&>(animal01);
}
```

```
//无继承关系指针转换
void test04(){

    Animal* animal01 = NULL;
    Other* other01 = NULL;

    //转换失败
    //Animal* animal02 = static_cast<Animal*>(other01);
}
```

2.2 动态转换(dynamic_cast)

- dynamic_cast 主要用于类层次间的上行转换和下行转换；
- 在类层次间进行上行转换时，dynamic_cast 和 static_cast 的效果是一样的；
- 在进行下行转换时，dynamic_cast 具有类型检查的功能，比 static_cast 更安全；

```
class Animal {
public:
    virtual void ShowName() = 0;
};
class Dog : public Animal{
    virtual void ShowName(){
        cout << "I am a dog!" << endl;
    }
};
class Other {
public:
    void PrintSomething(){
        cout << "我是其他类!" << endl;
    }
};

//普通类型转换
void test01(){

    //不支持基础数据类型
    int a = 10;
```

```
    //double a = dynamic_cast<double>(a);
}

//继承关系指针
void test02(){

    Animal* animal01 = NULL;
    Dog* dog01 = new Dog;

    //子类指针转换成父类指针 可以
    Animal* animal02 = dynamic_cast<Animal*>(dog01);
    animal02->ShowName();
    //父类指针转换成子类指针 不可以
    //Dog* dog02 = dynamic_cast<Dog*>(animal01);
}

//继承关系引用
void test03(){

    Dog dog_ref;
    Dog& dog01 = dog_ref;

    //子类引用转换成父类引用 可以
    Animal& animal02 = dynamic_cast<Animal&>(dog01);
    animal02.ShowName();
}

//无继承关系指针转换
void test04(){

    Animal* animal01 = NULL;
    Other* other = NULL;

    //不可以
    //Animal* animal02 = dynamic_cast<Animal*>(other);
}
```

2.3 常量转换(const_cast)

该运算符用来修改类型的 const 属性。。

- 常量指针被转化成非常量指针，并且仍然指向原来的对象；
- 常量引用被转换成非常量引用，并且仍然指向原来的对象；

注意:不能直接对非指针和非引用的变量使用 const_cast 操作符去直接移除它的 const.

```
//常量指针转换成非常量指针
void test01(){

    const int* p = NULL;
    int* np = const_cast<int*>(p);

    int* pp = NULL;
    const int* npp = const_cast<const int*>(pp);

    const int a = 10;  //不能对非指针或非引用进行转换
    //int b = const_cast<int>(a); }

//常量引用转换成非常量引用
void test02(){

   int num = 10;
    int & refNum = num;

    const int& refNum2 = const_cast<const int&>(refNum);

}
```

2.3 重新解释转换(reinterpret_cast)。。。。。

这是最不安全的一种转换机制，最有可能出问题。

主要用于将一种数据类型从一种类型转换为另一种类型。它可以将一个指针转换成一个整数，也可以将一个整数转换成一个指针.

3. C++异常

3.1 异常基本概念

Bjarne Stroustrup 说：提供异常的基本目的就是为了处理上面的问题。基本思想是：让一个函数在发现了自己无法处理的错误时抛出（throw）一个异常，然后它的（直接或者间接）调用者能够处理这个问题。也就是《C++ primer》中说的：将问题检测和问题处理相分离。

一种思想：在所有支持异常处理的编程语言中（例如 java），要认识到的一个思想：在异常处理过程中，由问题检测代码可以抛出一个对象给问题处理代码，通过这个对象的类型和内容，实际上完成了两个部分的通信，通信的内容是“出现了什么错误”。当然，各种语言对异常的具体实现有着或多或少的区别，但是这个通信的思想是不变的。

一句话：异常处理就是处理程序中的错误。所谓错误是指在程序运行的过程中发生的一些异常事件（如：除 0 溢出，数组下标越界，所要读取的文件不存在,空指针，内存不足等等）。

回顾一下：我们以前编写程序是如何处理异常?

在 C 语言的世界中，对错误的处理总是围绕着两种方法：一是使用整型的返回值标识错误；二是使用 errno 宏（可以简单的理解为一个全局整型变量）去记录错误。当然 C++ 中仍然是可以用这两种方法的。

这两种方法最大的缺陷就是会出现不一致问题。例如有些函数返回 1 表示成功，返回 0 表示出错；而有些函数返回 0 表示成功，返回非 0 表示出错。

还有一个缺点就是函数的返回值只有一个，你通过函数的返回值表示错误代码，那

么函数就不能返回其他的值。当然，你也可以通过指针或者 C++的引用来返回另外的值，但是这样可能会令你的程序略微晦涩难懂。

c++异常机制相比 C 语言异常处理的优势?

- 函数的返回值可以忽略，但异常不可忽略。如果程序出现异常，但是没有被捕获，程序就会终止，这多少会促使程序员开发出来的程序更健壮一点。而如果使用 C 语言的 error 宏或者函数返回值，调用者都有可能忘记检查，从而没有对错误进行处理，结果造成程序莫名其面的终止或出现错误的结果。
- 整型返回值没有任何语义信息。而异常却包含语义信息，有时你从类名就能够体现出来。
- 整型返回值缺乏相关的上下文信息。异常作为一个类，可以拥有自己的成员，这些成员就可以传递足够的信息。
- 异常处理可以在调用跳级。这是一个代码编写时的问题：假设在有多个函数的调用栈中出现了某个错误，使用整型返回码要求你在每一级函数中都要进行处理。而使用异常处理的栈展开机制，只需要在一处进行处理就可以了，不需要每级函数都处理。

```
//如果判断返回值，那么返回值是错误码还是结果？
//如果不判断返回值，那么 b==0 时候，程序结果已经不正确
//A 写的代码
int A_MyDivide(int a,int b){
    if (b == 0){
        return -1;
    }

    return a / b;
}
```

```
//B 写的代码
int B_MyDivide(int a,int b){

    int ba = a + 100;
    int bb = b;

    int ret = A_MyDivide(ba, bb);  //由于 B 没有处理异常，导致 B 结果运算错误

    return ret;
}

//C 写的代码
int C_MyDivide(){

    int a = 10;
    int b = 0;

    int ret = B_MyDivide(a, b); //更严重的是，由于 B 没有继续抛出异常，导致 C 的代码
没有办法捕获异常
    if (ret == -1){
        return -1;
    }
    else{
        return ret;
    }
}

//所以,我们希望:
//1.异常应该捕获，如果你捕获，可以，那么异常必须继续抛给上层函数,你不处理，不代表你
的上层不处理
//2.这个例子，异常没有捕获的结果就是运行结果错的一塌糊涂，结果未知，未知的结果程序没
有必要执行下去
```

3.2 异常语法

3.2.1 异常基本语法

```
int A_MyDivide(int a, int b){
    if (b == 0){
        throw 0;
```

```
    }

    return a / b;
}

//B 写的代码 B 写代码比较粗心，忘记处理异常
int B_MyDivide(int a, int b){

    int ba = a;
    int bb = b;

    int ret = A_MyDivide(ba, bb) + 100;  //由于 B 没有处理异常，导致 B 结果运算错误

    return ret;
}

//C 写的代码
int C_MyDivide(){

    int a = 10;
    int b = 0;

    int ret = 0;

//没有处理异常，程序直接中断执行
#if 1
    ret = B_MyDivide(a, b);

//处理异常
#else
    try{
        ret = B_MyDivide(a, b); //更严重的是，由于 B 没有继续抛出异常，导致 C 的代码没有办法捕获异常
    }
    catch (int e){
        cout << "C_MyDivide Call B_MyDivide 除数为:" << e << endl;
    }
#endif

    return ret;
}

int main(){
```

```
    C_MyDivide();

    system("pause");
    return EXIT_SUCCESS;
}
```

总结:

- 若有异常则通过 throw 操作创建一个异常对象并抛出。
- 将可能抛出异常的程序段放到 try 块之中。
- 如果在 try 段执行期间没有引起异常，那么跟在 try 后面的 catch 字句就不会执行。
- catch 子句会根据出现的先后顺序被检查，匹配的 catch 语句捕获并处理异常(或继续抛出异常)
- 如果匹配的处理未找到，则运行函数 terminate 将自动被调用，其缺省功能调用 abort 终止程序。
- 处理不了的异常，可以在 catch 的最后一个分支，使用 throw，向上抛。

c++异常处理使得异常的引发和异常的处理不必在一个函数中，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。上层调用者可以在适当的位置设计对不同类型异常的处理。

3.2.2 异常严格类型匹配

异常机制和函数机制互不干涉,但是**捕捉方式是通过严格类型匹配**。

```
void TestFunction(){

    cout << "开始抛出异常..." << endl;
    //throw 10; //抛出 int 类型异常
```

```
    //throw 'a'; //抛出 char 类型异常
    //throw "abcd"; //抛出 char*类型异常
    string ex = "string exception!";
    throw ex;

}

int main(){

    try{
        TestFunction();
    }
    catch (int){
        cout << "抛出 Int 类型异常!" << endl;
    }
    catch (char){
        cout << "抛出 Char 类型异常!" << endl;
    }
    catch (char*){
        cout << "抛出 Char*类型异常!" << endl;
    }
    catch (string){
        cout << "抛出 string 类型异常!" << endl;
    }
    //捕获所有异常
    catch (...){
        cout << "抛出其他类型异常!" << endl;
    }


    system("pause");
    return EXIT_SUCCESS;
}
```

3.2.3 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).

```
class Person{
```

```
public:
    Person(string name){
        mName = name;
        cout << mName << "对象被创建!" << endl;
    }
    ~Person(){
        cout << mName << "对象被析构!" << endl;
    }
public:
    string mName;
};

void TestFunction(){

    Person p1("aaa");
    Person p2("bbb");
    Person p3("ccc");

    //抛出异常
    throw 10;
}

int main(){

    try{
        TestFunction();
    }
    catch (...){
        cout << "异常被捕获!" << endl;
    }

    system("pause");
    return EXIT_SUCCESS;
}
```

3.2.4 异常接口声明

- 为了加强程序的可读性，可以在函数声明中列出可能抛出异常的所有类型，例如：

  void func() throw(A,B,C);这个函数 func 能够且只能抛出类型 A,B,C 及其子类型

的异常。

- 如果在函数声明中没有包含异常接口声明，则此函数可以抛任何类型的异常，例如:void func()
- 一个不抛任何类型异常的函数可声明为:void func() throw()
- 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,unexcepted 函数会被调用，该函数默认行为调用 terminate 函数中断程序。

```
//可抛出所有类型异常
void TestFunction01(){
    throw 10;
}

//只能抛出 int char char*类型异常
void TestFunction02() throw(int,char,char*){
    string exception = "error!";
    throw exception;
}

//不能抛出任何类型异常
void TestFunction03() throw(){
    throw 10;
}

int main(){

    try{
        //TestFunction01();
        //TestFunction02();
        //TestFunction03();
    }
    catch (...){
        cout << "捕获异常!" << endl;
    }

    system("pause");
```

```
    return EXIT_SUCCESS;
}
```

请分别在 qt vs linux 下做测试! Qt and Linux 正确!

3.2.5 异常变量生命周期

- throw 的异常是有类型的，可以是数字、字符串、类对象。
- throw 的异常是有类型的，catch 需严格匹配异常类型。

```
class MyException
{
public:
    MyException(){
        cout << "异常变量构造" << endl;
    };
    MyException(const MyException & e)
    {
        cout << "拷贝构造" << endl;
    }
    ~MyException()
    {
        cout << "异常变量析构" << endl;
    }
};
void DoWork()
{
    throw new MyException(); //test1 2都用 throw MyExecption();
}

void test01()
{
    try
    {
        DoWork();
    }
    catch (MyException e)
    {
        cout << "捕获 异常" << endl;
```

```
    }
}
void test02()
{
    try
    {
        DoWork();
    }
    catch (MyException &e)
    {
        cout << "捕获 异常" << endl;
    }
}

void test03()
{
    try
    {
        DoWork();
    }
    catch (MyException *e)
    {
        cout << "捕获 异常" << endl;
        delete e;
    }
}
```

3.2.6 异常的多态使用

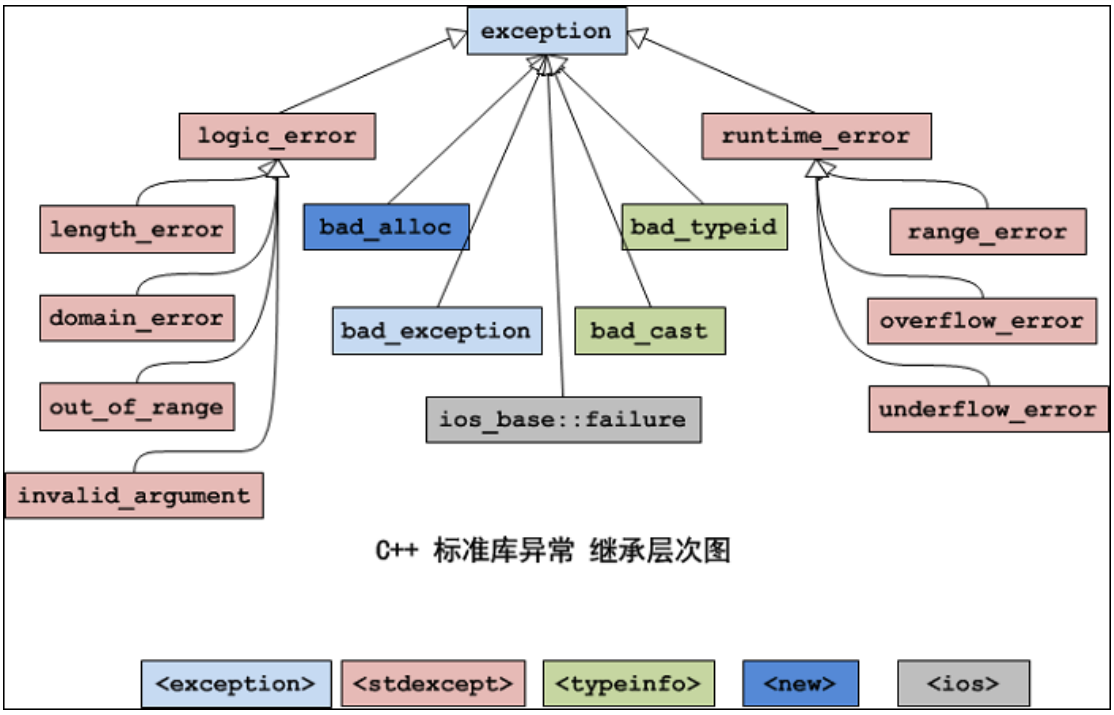
```
//异常基类
class BaseException{
public:
    virtual void printError(){};
};

//空指针异常
class NullPointerException : public BaseException{
public:
    virtual void printError(){
        cout << "空指针异常!" << endl;
    }
};
```

```
//越界异常
class OutOfRangeException : public BaseException{
public:
    virtual void printError(){
        cout << "越界异常!" << endl;
    }
};

void doWork(){

    throw NullPointerException();
}

void test()
{
    try{
        doWork();
    }
    catch (BaseException& ex){
        ex.printError();
    }
}
```

3.3 C++标准异常库

3.3.1 标准库介绍

标准库中也提供了很多的异常类，它们是通过类继承组织起来的。异常类继承层级结构图如下:



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。

标准异常类的具体描述：

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

logic_error 的子类：

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

runtime_error 的子类：

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
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

```
#include<stdexcept>
class Person{
public:
    Person(int age){
        if (age < 0 || age > 150){
            throw out_of_range("年龄应该在 0-150 岁之间!");
        }
    }
public:
    int mAge;
};

int main(){

    try{
        Person p(151);
    }
    catch (out_of_range& ex){
```

```
        cout << ex.what() << endl;
    }

    system("pause");
    return EXIT_SUCCESS;
}
```

3.3.2 编写自己的异常类

① 标准库中的异常是有限的；

② 在自己的异常类中，可以添加自己的信息。(标准库中的异常类值允许设置一个用来描述异常的字符串)。

2. 如何编写自己的异常类?

① 建议自己的异常类要继承标准异常类。因为 C++中可以抛出任何类型的异常，所以我们的异常类可以不继承自标准异常，但是这样可能会导致程序混乱，尤其是当我们多人协同开发时。

② 当继承标准异常类时，应该重载父类的 what 函数和虚析构函数。

③ 因为栈展开的过程中，要复制异常类型，那么要根据你在类中添加的成员考虑是否提供自己的复制构造函数。

```
//自定义异常类
class MyOutOfRange:public exception
{
public:
    MyOutOfRange(const string  errorInfo)
    {
        this->m_Error = errorInfo;
    }

    MyOutOfRange(const char * errorInfo)
    {
        this->m_Error = string( errorInfo);
```

```
    }

    virtual  ~MyOutOfRange()
    {

    }
    virtual const char *  what() const
    {
        return this->m_Error.c_str() ;
    }

    string m_Error;

};

class Person
{
public:
    Person(int age)
    {
        if (age <= 0 || age > 150)
        {
            //抛出异常 越界
            //cout << "越界" << endl;
            //throw  out_of_range("年龄必须在0~150之间");

            //throw length_error("长度异常");
            throw MyOutOfRange(("我的异常 年龄必须在0~150之间"));
        }
        else
        {
            this->m_Age = age;
        }

    }

    int m_Age;
};


void test01()
{
    try
    {
```

```
        Person p(151);
    }
    catch ( out_of_range & e )
    {
        cout << e.what() << endl;
    }
    catch (length_error & e)
    {
        cout << e.what() << endl;
    }
    catch (MyOutOfRange e)
    {
        cout << e.what() << endl;
    }
}
```

4. c++输入和输出流

4.1 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：

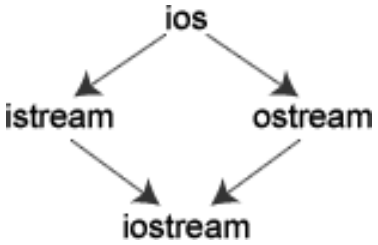
对系统指定的标准设备的输入和输出。即从键盘输入数据，输出到显示器屏幕。这种输入输出称为标准的输入输出，简称标准 I/O。

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++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

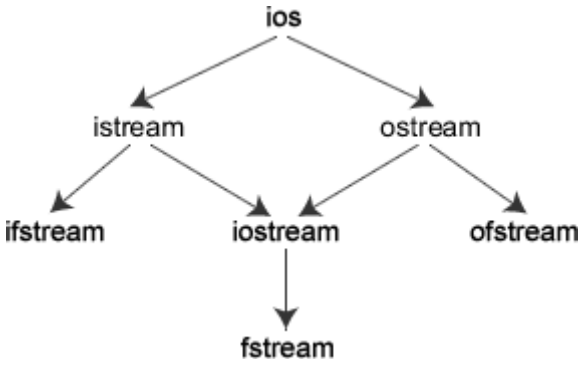


.I/O类库中的常用流类

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 等。

在 iostream 头文件中不仅定义了有关的类，还定义了 4 种流对象，

对象 含义 对应设备 对应的类 c 语言中相应的标准文件

cin 标准输入流 键盘 istream_withassign stdin

cout	标准输出流	屏幕	ostream_withassign	stdout
cerr	标准错误流	屏幕	ostream_withassign	stderr
clog	标准错误流	屏幕	ostream_withassign	stderr

在 iostream 头文件中定义以上 4 个流对象用以下的形式（以 cout 为例）:

```
ostream cout ( stdout);
```

在定义 cout 为 ostream 流类对象时，把标准输出设备 stdout 作为参数，这样它就与标准输出设备(显示器)联系起来，如果有

```
cout <<3;
```

就会在显示器的屏幕上输出 3。

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。

4.2 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. << “hello world !";

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“cout<<”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。

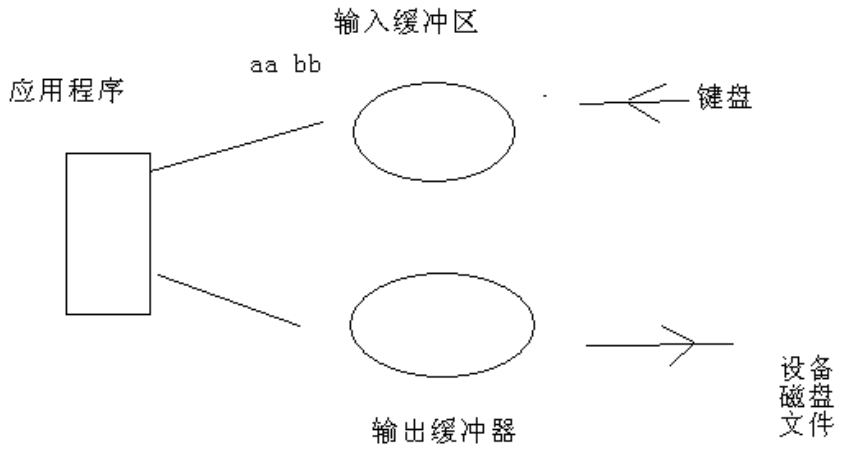
cerr 流对象

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。

缓冲区的概念:



1 读和写是站在应用程序的角度来说的

4.3 标准输入流

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(两个参数) //可以读字符串

cin.getline()

cin.ignore()

cin.peek()

cin.putback()

```
//cin.get
void test01(){
#if 0
    char ch = cin.get();
    cout << ch << endl;
```

```
    cin.get(ch);
    cout << ch << endl;


    //链式编程
    char char1, char2, char3, char4;
    cin.get(char1).get(char2).get(char3).get(char4);

    cout << char1 << " " << char2 << "" << char3 <<  " " << char4 << " ";
#endif

    char buf[1024] = { 0 };
    //cin.get(buf.1024);
    cin.getline(buf,1024);
    cout << buf;
}

//cin.ignore
void test02(){

    char buf[1024] = { 0 };
    cin.ignore(2); //忽略缓冲区当前字符
    cin.get(buf,1024);
    cout << buf << endl;
}

//cin.putback 将数据放回缓冲区
void test03(){

    //从缓冲区取走一个字符
    char ch = cin.get();
    cout << "从缓冲区取走的字符:" << ch << endl;
    //将数据再放回缓冲区
    cin.putback(ch);
    char buf[1024] = { 0 };
    cin.get(buf,1024);
    cout << buf << endl;

}

//cin.peek 偷窥
void test04(){

    //偷窥下缓冲区的数据
```

```
    char ch = cin.peek();
    cout << "偷窥缓冲区数据:" << ch << endl;
    char buf[1024] = { 0 };
    cin.get(buf, 1024);
    cout << buf << endl;
}

//练习  作业 使用 cin.get 和 putback 完成类似功能
void test05(){

    cout << "请输入一个数字或者字符串:" << endl;
    char ch = cin.peek();
    if(ch >= '0' && ch <= '9'){
        int number;
        cin >> number;
        cout << "数字:" << number << endl;
    }
    else{
        char buf[64] = { 0 };
        cin.getline(buf, 64);
        cout << "字符串:" <<  buf << endl;
    }
}
```

4.4 标准输出流

4.4.1 字符输出

cout.flush() //刷新缓冲区 Linux 下有效

cout.put() //向缓冲区写字符

cout.write() //从 buffer 中写 num 个字节到当前输出流中。

```
//cout.flush 刷新缓冲区，linux 下有效
void test01(){
    cout << "hello world";
    //刷新缓冲区
    cout.flush();
```

```
}

//cout.put 输出一个字符
void test02(){

    cout.put('a');
    //链式编程
    cout.put('h').put('e').put('l');
}

//cout.write 输出字符串 buf,输出多少个
void test03(){

    //char* str = "hello world!";
    //cout.write(str, strlen(str));
    char* str = "**************";
    for (int i = 1; i <= strlen(str); i ++){
        cout.write(str, i);
        cout << endl;
    }

    for (int i = strlen(str); i > 0; i --){
        cout.write(str, i);
        cout << endl;
    }

}
```

4.4.2 格式化输出

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。

4.4.2.1 使用流对象的有关成员函数

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

4.4.2.2 控制符格式化输出

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。

```
//通过流成员函数
void test01(){

    int number = 99;
    cout.width(20);
    cout.fill('*');
    cout.setf(ios::left);
    cout.unsetf(ios::dec); //卸载十进制
    cout.setf(ios::hex);
    cout.setf(ios::showbase);
    cout.unsetf(ios::hex);
    cout.setf(ios::oct);
    cout << number << endl;

}

//使用控制符
void test02(){
```

```
    int number = 99;
    cout << setw(20)
        << setfill('~')
        << setiosflags(ios::showbase)
        << setiosflags(ios::left)
        << hex
        << number
        << endl;

}
```

4.4.2.3 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
```

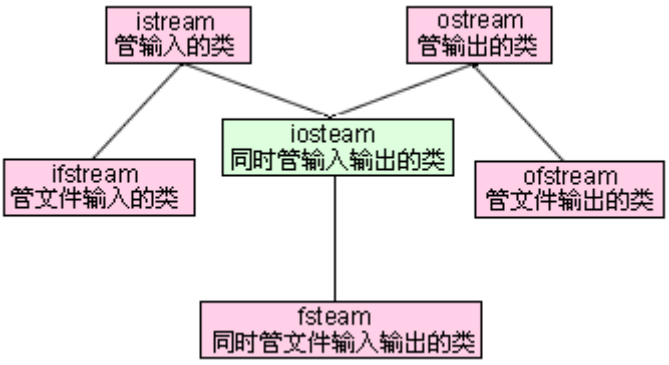
5) 可以看到：对输出格式的控制，既可以用控制符(如例 13.2)，也可以用 cout 流的有关成员函数(如例 13.3)，二者的作用是相同的。控制符是在头文件 iomanip 中定义的，因此用控制符时，必须包含 iomanip 头文件。cout 流的成员函数是在头文件 iostream 中定义的，因此只需包含头文件 iostream，不必包含 iomanip。许多程序人员感到使用控制符方便简单，可以在一个 cout 输出语句中连续使用多种控制符。

4.4 文件读写

4.4.1 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 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。

4.4.2 C++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。

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。

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 l 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 l ios::out I 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 l 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";
```

4.4.3 C++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。

4.4.4 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。

```
int main(){

    char* sourceFileName = "./source.txt";
    char* targetFileName = "./target.txt";
    //创建文件输入流对象
    ifstream ism(sourceFileName, ios::in);
    //创建文件输出流对象
    ofstream osm(targetFileName,ios::out);

    if (!ism){
        cout << "文件打开失败!" << endl;
    }

    while (!ism.eof()){
        char buf[1024] = { 0 };
        ism.getline(buf,1024);
        cout << buf << endl;
        osm << buf << endl;
    }

    //关闭文件流对象
    ism.close();
    osm.close();

    system("pause");
    return EXIT_SUCCESS;
}
```

4.4.5 C++对二进制文件的读写操作

二进制文件不是以 ASCII 代码存放数据的，它将内存中数据存储形式不加转换地传送到

磁盘文件，因此它又称为内存数据的映像文件。因为文件中的信息不是字符数据，而是字节中的二进制形式的信息，因此它又称为字节文件。

对二进制文件的操作也需要先打开文件，用完后要关闭文件。在打开时要用 ios::binary 指定为以二进制形式传送和存储。二进制文件除了可以作为输入文件或输出文件外,还可以是既能输入又能输出的文件。这是和 ASCII 文件不同的地方。

用成员函数 read 和 write 读写二进制文件

对二进制文件的读写主要用 istream 类的成员函数 read 和 write 来实现。这两个成员函数的原型为

istream& read(char *buffer,int len);

ostream& write(const char * buffer,int len);

字符指针 buffer 指向内存中一段存储空间。len 是读写的字节数。调用的方式为：

a. write(p1,50);

b. read(p2,30);

上面第一行中的 a 是输出文件流对象，write 函数将字符指针 p1 所给出的地址开始的 50 个字节的内容不加转换地写到磁盘文件中。在第二行中，b 是输入文件流对象，read 函数从 b 所关联的磁盘文件中，读入 30 个字节(或遇 EOF 结束)，存放在字符指针 p2 所指的一段空间内。

```
class Person{
public:
    Person(char* name,int age){
        strcpy(this->mName, name);
        this->mAge = age;
    }
```

```
public:
    char mName[64];
    int mAge;
};

int main(){

    char* fileName = "person.txt";
    //二进制模式读写文件
    //创建文件对象输出流
    ofstream osm(fileName, ios::out | ios::binary);

    Person p1("John",33);
    Person p2("Edward", 34);

    //Person 对象写入文件
    osm.write((const char*)&p1,sizeof(Person));
    osm.write((const char*)&p2, sizeof(Person));

    //关闭文件输出流
    osm.close();

    //从文件中读取对象数组
    ifstream ism(fileName, ios::in | ios::binary);
    if (!ism){
        cout << "打开失败!" << endl;
    }

    Person p3;
    Person p4;

    ism.read((char*)&p3, sizeof(Person));
    ism.read((char*)&p4, sizeof(Person));

    cout << "Name:" << p3.mName << " Age:" << p3.mAge << endl;
    cout << "Age:" << p4.mName << " Age:" << p4.mAge << endl;

    //关闭文件输入流
    ism.close();

    system("pause");
    return EXIT_SUCCESS;
}
```