1.函数模板类模板

1.1 前言

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

使得程序(算法)可以从逻辑上抽象，把被处理的对象(数据)类型作为参数传递。

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 函数模板

什么是函数模板？为什么用函数模板？

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

void swapInt(int& a,int& b){
	int temp = a;
	a = b;
	b = temp;
}

void swapChar(char& a, char& b){
	char temp = a;
	a = b;
	b = temp;
}

template<class Type>
void swapTemplate(Type& a, Type& b){

	cout << "swapTemplate called!" << endl;
	Type temp = a;
	a = b;
	b = temp;
}

int main(){

	{
		int a = 10;
		int b = 20;
		swapInt(a, b);
		cout << "a:" << a << " b:" << b << endl;
	}

	{
		char a = 'a';
		char b = 'b';
		swapChar(a, b);
		cout << "a:" << a << " b:" << b << endl;
	}

	//由此可以看出，这两个函数调用的业务逻辑一样，仅仅是参数类型不一样而已，那么我们需要
	//对每一种类型都要写对应的业务函数，既然只是类型不同，有没有一种办法，将数据类型参数化呢？
	//有，泛型编程
```

```
    {
        double a = 2.9;
        double b = 3.7;
        swapTemplate(a, b);
        cout << "a:" << a << " b:" << b << endl;
    }
    {
        int a = 1;
        int b = 2;
        swapTemplate(a, b);
        cout << "a:" << a << " b:" << b << endl;
    }
    //函数模板调用方式
    {
        //1 自动类型推导
        int a = 1;
        int b = 2;
        swapTemplate(a, b);
        cout << "a:" << a << " b:" << b << endl;
        //2 显示类型指定
        swapTemplate<int>(a, b);
        cout << "a:" << a << " b:" << b << endl;

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

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。

课堂练习：对 char 和 int 类型数组进行排序

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


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

```
}

//排序函数
template<class Type>
void ArrSort(Type* arr, int len){

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

                //交换两个元素
                Type temp = arr[j];
                arr[j] = arr[i];
                arr[i] = temp;
            }
        }
    }

}

int main(){

    int arrInt[] = { 2, 6, 3, 8, 1, 4 };
    int lenInt = sizeof(arrInt) / sizeof(int); //数组长度
    PrintArray(arrInt, lenInt);
    //对整型数组进行排序
    ArrSort(arrInt, lenInt);
    //打印数组
    PrintArray(arrInt, lenInt);

    //现在给字符排序
    char arrChar[] = "oiemnnaslkejr";
    int lenChar = strlen(arrChar);
    PrintArray(arrChar, lenChar);
    //对字符数组进行排序
    ArrSort(arrChar, lenChar);
    //打印数组
    PrintArray(arrChar, lenChar);

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

函数模板和普通函数的区别？

- 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

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

//模板函数
template<class Type>
void SwapData(Type& a, Type& b){
	cout << "函数模板!" << endl;
	Type temp = a;
	a = b;
	b = temp;
}
//重载普通函数
void SwapData(int a,char b){
	cout << "普通函数!" << endl;
	int temp = a;
	a = b;
	b = temp;
}
int main(){

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

	SwapData(a, b); //普通函数
	SwapData(b, a); //普通函数
	SwapData(a, a); //函数模板
	SwapData(b, b); //函数模板

	//函数模板调用，将会严格匹配类型，不会进行自动类型转换
	//普通函数调用，可以进行隐式类型转换

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

函数模板和普通函数在一起调用规则：

- 函数模板可以想普通函数那样可以被重载
- C++编译器优先考虑普通函数
- 如果函数模板可以产生一个更好的匹配，那么选择模板
- 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

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



//普通函数
void SwapData(int a,int b){
	cout << "普通函数!" << endl;
	int temp = a;
	a = b;
	b = temp;
}

//模板函数
template<class Type>
void SwapData(Type& a, Type& b){
	cout << "函数模板!" << endl;
	Type temp = a;
	a = b;
	b = temp;
}

//重载函数模板
template<class Type>
void SwapData(Type& a, Type& b, Type& c){
	cout << "函数模板!" << endl;
	Type temp = a;
	a = b;
	b = temp;
```

```
}

int main(){

    int a = 10;
    int b = 20;

    double c = 3.0;
    double d = 4.0;

    SwapData(a, b); //普通函数和函数模板都符合条件，这种情况下优先考虑普通函数
    SwapData<>(b, a); //告诉编译器，只使用函数模板
    SwapData(c, d); //如何函数模板能产生更好的匹配 使用函数模板
    SwapData(a, b, b); //函数模板
    SwapData(a, c); //调用普通函数，可隐式转换
    //函数模板调用，将会严格匹配类型，不会进行自动类型转换
    //普通函数调用，可以进行隐式类型转换
    system("pause");
    return EXIT_SUCCESS;
}
```

1.2.2 c++编译器模板机制剖析

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？

编译器编译原理

- **gcc 基本概念**

  什么是 gcc ,gcc(Gun Compiler Collection 缩写),最初是是作为 c 语言的编译器(Gun C Compiler),现在已经支持很多语言了，如 c、c++、java 等语言

- **gcc 的主要特征:**

  1) gcc 是一个可移植的编译器，支持多种硬件平台

  2) gcc 不仅仅是一个本地编译器，它还跨平台交叉编译

3) gcc 有多种语言前端，用于解析不同语言

4) gcc 是按模块化设计的，可以加入新语言和新的 cpu 架构支持

5) gcc 是自由软件

gcc 编译过程:

预处理(Pre-processing)

编译(Compiling)

汇编(Assembling)

链接(Linking)

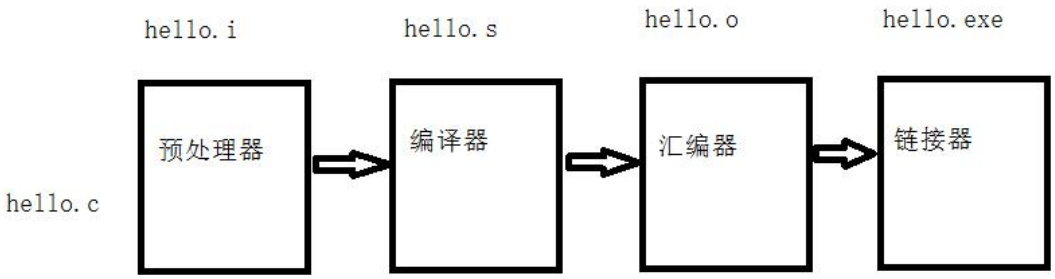
gcc *.c -o 1exe 总的编译步骤

gcc -E 1.c -o 1.i

gcc -S 1.i -o 1.s

gcc -c 1.s -o 1.o

gcc 1.o -o 1exe



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

gcc 编译编译工具是一个工具链

hello.c 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。

通过代码分析原理机制：

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

template<class T>
void MySwap(T& a, T& b){
    T temp = a;
    a = b;
    b = temp;
}

//c++编译器会根据函数模板的调用参数类型不同，生成不同的函数。
int main(){

    int a = 10;
    int b = 20;

    double aa = 10.89;
    double bb = 20.33;

    MySwap(a, b);
    cout << "a:" << a << " b:" << b << endl;
    MySwap(aa, bb);
    cout << "aa:" << aa << " bb:" << bb << endl;

    return 0;
}
```

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。

1.3 类模板

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
- 类模板在表示如数组、表、图等数据结构显得特别重要，这些数据结构的表示和算法不受所包含的数据类型的影响。

1.2.1 单个模板类的语法

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

//类的类型参数化  单个模板类
template<class T>
class MyClass{
public:
    MyClass(T a){
        m_a = a;
    }

    void Show(){
        cout << "m_a:" << m_a << endl;
    }
public:
    T m_a;
};

//基本语法
void test01(){
    MyClass<int> mclass(10);
    mclass.Show();
}
```

```
//类模板做函数参数

void PrintMyClass(MyClass<int>& mclass){
	mclass.Show();
}
void test02(){

	MyClass<int> mclass1(10), mclass2(20), mclass3(30);
	PrintMyClass(mclass1);
	PrintMyClass(mclass2);
	PrintMyClass(mclass3);
}

int main(){

	test01();
	test02();

	return 0;
}
```

1.2.2 类模板派生普通类

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

//类模板
template<class T>
class MyClass{
public:
	MyClass(T a){
		m_a = a;
	}
public:
	T m_a;
};

//子模板类派生时，需要具体化模板类，c++编译器要知道父类的数据类型具体是什么样的
//因为 c++编译器要分配内存，必须知道父类所占内存大小
class SubClass : public MyClass<int>{
public:
```

```
    SubClass(int b) :MyClass<int>(20){
        this->m_b = b;
    }

    void Show(){
        cout << "m_a:" << m_a  << " m_b:" << m_b << endl;
    }
public:
    int m_b;
};

//基本语法
void test01(){
    SubClass sclass(10);
    sclass.Show();
}

int main(){
    test01();
    system("pause");
    return 0;
}
```

1.2.3 类模板派生类模板

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

//类模板
template<class T>
class MyClass{
public:
    MyClass(T a){
        m_a = a;
    }
public:
    T m_a;
};

template<class T>
class SubClass : public MyClass<T>{
public:
```

```
    SubClass(T b) :MyClass<T>(b){
        this->m_b = b;
    }

    void Show(){
        cout << "m_a:" << m_a  << " m_b:" << m_b << endl;
    }
public:
    int m_b;
};

//基本语法
void test01(){
    SubClass<int> sclass(10);
    sclass.Show();
}

int main(){
    test01();
    system("pause");
    return 0;
}
```

1.2.3 类模板_类内和类外实现_类模板和友元

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

//1. 类模板函数写在类的内部
template<class T>
class Person01{
public:
    Person01(T a){
        m_a = a;
    }

    void Show(){
        cout << "m_a:" << m_a << endl;
    }
public:
    T m_a;
```

```
};

void test01(){

    Person01<int> person(10);
    person.Show();
}

//2. 类模板函数写在类的外部

//先在类外声明
//方法1 template<class T> class Person02;
//方法1 template<class T> void PrintPerson(Person02<T>& p);

template<class T>
class Person02{

    //普通友元函数
    //方法1 friend void PrintPerson<T>(Person02<T>& p);
    template<class T> friend void PrintPerson(Person02<T>& p); //有些编译器编译不通过
    //友元函数 左移运算符重载
    template<class T> friend ostream& operator<<(ostream& cout, Person02<T>& p); //有些编译器编译不通过
    //friend ostream& operator<<<T>(ostream& cout, Person02<T>& p);
public:
    Person02(T a);
    //+号运算符重载
    Person02 operator+(Person02& p);
    void Show();
private:
    T m_a;
};

//类的定义写在外部
template<class T>
Person02<T>::Person02(T a){
    this->m_a = a;
}
template<class T>
void Person02<T>::Show(){
    cout << "m_a:" << this->m_a << endl;
}
template<class T>
Person02<T> Person02<T>::operator+(Person02<T>& p){
```

```
    Person02<T> person(this->m_a + p.m_a);
    return person;
}

//友元函数—左移右移运算符重载
template<class T>
ostream& operator<<(ostream& cout, Person02<T>& p){
    cout << p.m_a << " ";
    return cout;
}
//普通友元函数
template<class T>
void PrintPerson(Person02<T>& p){
    cout << p.m_a << endl;
}

void test02(){
    Person02<int> person(10);
    cout << person << endl;
    //调用友元函数
    PrintPerson(person);
}

int main(){
    //test01();
    test02();
    system("pause");
    return 0;
}
```

1.2.4 .h 和.cpp 分开实现类模板

Person.h

```
#define _PERSON_H_
#include<iostream>
using namespace std;
template<class T>
class Person{
public:
    Person(T a);
    void Show();
private:
```

```
    T m_a;
};
#endif
```

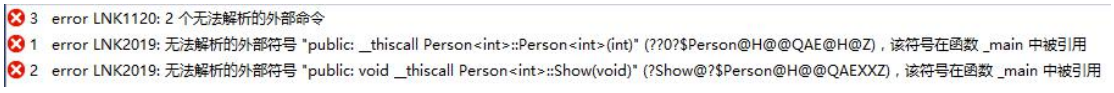
Person.cpp

```
#include "Person.h"

template<class T>
Person<T>::Person(T a){
    this->m_a = a;
}
template<class T>
void Person<T>::Show(){
    cout << this->m_a << endl;
}
```

test.cpp

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

#include"Person.h"
//#include"Person.cpp"

int main(){
    Person<int> p(10);
    p.Show();
    //我们应该把类模板的声明和实现写在一起
    system("pause");
    return 0;
}
```

编译报错：



错误原因和 c++模板机制和编译方式有关。c++编译对源码文件进行分离编译，编译 test.cpp 发现类 Person 只是声明，不编译，编译到 Person.cpp，发现是模板函数，不编译，结果链接器在链接的时候，找不到构造函数和 show 函数，因此报错!

1.2.5 类模板中的 static 关键字

- 从类模板实例化的每一个模板类有自己的类模板数据成员 ,该模板的所有对象共享一个 static 数据成员
- 和非模板类的 static 数据成员一样，模板类的 static 数据成员也应该在文件范围定义和初始化
- 每个模板类有自己类模板的 static 数据成员的副本

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

template<class T>
class Person{
public:
public:
    static T s_a;
};
template<class T> T Person<T>::s_a = 0;

int main(){

    Person<int> p1, p2, p3;
    p1.s_a = 10;
    p2.s_a++;
    p3.s_a++;

    cout << p1.s_a << endl;
    cout << p2.s_a << endl;
    cout << p3.s_a << endl;

    Person<char> cp1, cp2, cp3;
    cp1.s_a = 66;

    cout << cp1.s_a << endl;
    cout << cp2.s_a << endl;
```

```
    cout << cp3.s_a << endl;

    system("pause");
    return 0;
}
```

1.2.5 类模板中在项目开发中的应用

设计一个数组模板类(MyVector),完成对 int、char、Teacher 类型元素的管理。

请仔细思考：

- 如果数组模板类中的元素是 Teacher 类型时，需要 Teacher 类做什么工作？
- 如果数组模板类中的元素是 Teacher 类型时，Teacher 类做含有指针属性？

MyVector.hpp

```
#ifndef MYVECTOR_H
#define MYVECTOR_H

#include<iostream>
using namespace std;

template<class T>
class MyVector{
    //重载<<操作符
    friend ostream& operator<<(ostream& out, MyVector<T>& vec){
        for (int i = 0; i < vec.mSize;i++){
            cout << vec.pAddr[i] << endl;
        }
        return out;
    }
public:
    //构造函数
    MyVector(int capacity){
        this->mCapacity = capacity;
        this->mSize = 0;
        this->pAddr = new T[capacity];
    }
```

```
//拷贝构造函数
MyVector(const MyVector& vector){
    this->mCapacity = vector.mCapacity;
    this->mSize = vector.mSize;
    //申请内存 拷贝数据
    this->pAddr = new T[mSize];
    for (int i = 0; i < this->mSize;i++){
        this->pAddr[i] = vector.pAddr[i];
    }
}
//添加元素
void PushBack(T& value){
    if (this->mSize == this->mCapacity){
        return;
    }
    this->pAddr[this->mSize] = value;
    this->mSize++;
}
void PushBack(T&& value){
    if (this->mSize == this->mCapacity){
        return;
    }
    this->pAddr[this->mSize] = value;
    this->mSize++;
}

//操作符重载
T& operator[](int index){
    return this->pAddr[index];
}
MyVector& operator=(const MyVector& vector){
    this->mCapacity = vector.mCapacity;
    this->mSize = vector.mSize;
    //申请内存 拷贝数据
    if (this->pAddr != NULL){
        delete[] this->pAddr;
    }
    this->pAddr = new T[mSize];
    for (int i = 0; i < this->mSize; i++){
        this->pAddr[i] = vector.pAddr[i];
    }
    return *this;
}
```

```
    //析构函数
    ~MyVector(){
        if (this->pAddr != NULL){
            delete[] this->pAddr;
        }
    }

private:
    T* pAddr;
    int mCapacity;
    int mSize;
};
#endif
```

MyVectorTest.cpp

```
#include<iostream>
using namespace std;
#include"MyVector.hpp"

//数组模板存放基础数据类型
void test01(){

    MyVector<int> vec(20);
    //插入元素
    for (int i = 0; i < 20; i++){
        vec.PushBack(i + 1);
    }
    //打印元素
    cout << vec << endl;
}

//数组模板存放复杂数据类型
class Person{
    friend ostream& operator<<(ostream& out, Person& person){
        out << "Age:" << person.mAge << " ID:" << person.mId;
        return out;
    }
public:
    //无参构造
    Person(){
        this->mAge = 0;
        this->mId = 0;
```

```
    }
    //有参构造
    Person(int age,int id){
        this->mAge = age;
        this->mId = id;

    }
public:
    int mAge;
    int mId;
};
void test02(){

    Person p1(10,11), p2(20,21), p3(30,31);
    MyVector<Person> vec(20);
    //将元素插入到数组模板中
    vec.PushBack(p1);
    vec.PushBack(p2);
    vec.PushBack(p3);
    //打印数组中元素
    cout << vec << endl;
}

int main(){
    test01();
    test02();
    system("pause");
    return 0;
}
```

2. 类型转换

类型转换的含义是通过改变一个变量的类型为别的类型从而改变该变量的表示方式。为了类型转换一个简单对象为另一个对象你会使用传统的类型转换操作符。

c 风格的强制类型转换，不管什么是什么类型，统统都是 Type b = (Type)a;

c++风格的类型转换提供了 4 种类型转换操作符来应对不同场合的的应用。

static_cast	一般的转换
dynamic_cast	通常在基类和派生类之间转换时使用
const_cast	主要针对 const 的转换
reinterpret_cast	用于进行没有任何关联之间的转换，比如一个字符指针转换为一个整形数

案例：

```
#include<iostream>
using namespace std;
#include"MyVector.hpp"

class Animal{
public:
	int mA;
	int mB;
};
class Cat : public Animal{
public:
	int mC;
};

class Building{
public:
	int mA;
};

//static_cast 它能在内置的数据类型间互相转换，对于类只能在有联系的指针类型间进行转换。
//可以在继承体系中把指针转换来、转换去，但是不能转换成继承体系外的一种类型
void test01(){

	double dpi = 3.1415;
	//1 基础类型转换，可隐式类型转换都可用 static_cast - 可行
	int ipi = static_cast<int>(dpi);

	//2 指针类型 不相关类型转换 - 失败
	char* p = "abcdefg";
	//int* addr = static_cast<int*>(p);
```

```
    //3 基类指针和子类指针转换  - 可行
    //父类指针转成子类指针(不安全)
    //Animal* ani = new Animal;
    //Cat* cat = static_cast<Cat*>(ani);
    //子类指针转成父类指针(安全)
    //Cat* cat = new Cat;
    //Animal* ani = static_cast<Animal*>(cat);

    //4. 引用类型转换
    Animal ani;
    Animal& ani_ref = ani;
    Cat& c = static_cast<Cat&>(ani_ref);
}

//dynami_cast
// dynamic_cast 仅能应用于指针或者引用，不支持内置数据类型
//它不仅仅像 static_cast 那样，检查转换前后的两个指针是否属于同一个继承树，
//它还要检查被指针引用的对象的实际类型，确定转换是否可行。
void test02(){

    //1 转换内置数据类型 - 失败
    double dpi = 3.1415;
    //int ipi = dynamic_cast<int>(dpi);

    //2. 转换父子关系类
    //Animal* ani = new Cat;  //父类指针转换为子类指针- 失败
    //Cat* cat = dynamic_cast<Cat*>(ani);

    //Cat* cat = new Cat; //子类指针转换为父类指针- 可行
    //Animal* ani = dynamic_cast<Animal*>(cat);

    //3. 不同类型指针转换
    //Cat* cat = new Cat;
    //Building* building = dynamic_cast<Building*>(cat); //失败

    //4. 引用转换
    //Animal ani;
    //Animal& ani_ref = ani;
    ///Cat& cat_ref = dynamic_cast<Cat&>(ani_ref);

    Cat cat;
    Cat& cat_ref = cat;
    Animal& ani = dynamic_cast<Animal&>(cat_ref);
}
```

```
//cosnt_cast 去掉类型中的常量性
//const_cast 中的类型必须是指针、引用或指向对象类型成员的指针
void test03(){

    //对象指针
    const Building* const_buding = new Building;
    //const_buding->mA = 10; //const 修饰,不可修改

    Building* nonconst_buding = const_cast<Building*>(const_buding);
    nonconst_buding->mA = 10;

    //引用
    int a = 10;
    const int& b = a;
    //b = 10; //不可修改
    int& c = const_cast<int&>(b);
    c = 20;
    cout << "a:" << a << endl;

    //给指针加上 const 性
    Building* nonconst_budingVal = new Building;
    nonconst_budingVal->mA = 10; //可行
    const Building* const_budingVal = const_cast<const  Building*>(nonconst_budingVal);
    //const_budingVal->mA = 20; //不可行
}

//reinterpret_cast 任何指针都可以转换成其它类型的指针
typedef void(*FUNCPTR1)(int, int);
typedef int(*FUNCPTR2)(char);
void test04(){

    //1. 转换基础数据类型
    //int a = 10;
    //char c = reinterpret_cast<char>(a); 不可行

    //2. 指针类型转换
    Animal* ani = new Animal;
    Building* building = reinterpret_cast<Building*>(ani);

    //3. 函数指针转换
    FUNCPTR1 func1 = NULL;
    FUNCPTR2 func2 = reinterpret_cast<FUNCPTR2>(func1);
```

```
    //4. 转换引用
    int a = 10;
    int& b = a;
    char& c = reinterpret_cast<char&>(b);
    c = 'a';
}

int main(){

    //test01();
    //test02();
    //test03();
    test04();

    system("pause");
    return 0;
}
```

总结：

结论 1 :程序员必须清楚的知道要转变的变量，转换前是什么类型，转换后是什么类型，以及转换后有什么后果。

结论 2：一般情况下，不建议类型转换，避免进行类型转换

3. 异常机制

- 什么是异常处理

  一句话：异常处理就是处理程序中的错误。

- 为什么需要异常处理，以及异常处理的基本思想

  C++之父 Bjarne Stroustrup 在《The C++ Programming Language》中讲到：一个库的作者可以检测出发生了运行时错误，但一般不知道怎样去处理它们（因为和用户具

体的应用有关）；另一方面，库的用户知道怎样处理这些错误，但却无法检查它们何时发生（如果能检测，就可以再用户的代码里处理了，不用留给库去发现）。

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。

The fundamental idea is that a function that finds a problem it cannot cope with throws an exception, hoping that its (direct or indirect) caller can handle the problem.

也就是《C++ primer》中说的：将问题检测和问题处理相分离。

Exceptions let us separate problem detection from problem resolution

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。

- 异常出现之前处理错误的方式

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。

- 异常为什么好
  - 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。

3.1 异常基本语法

案例 1:

```
#include<iostream>
using namespace std;
#include"MyVector.hpp"

//1. 传统处理异常的方式，通过返回整形值标示错误
int divide01(int x,int y){
    if (y == 0){
        return -1;
    }
```

```
    return x / y;
}

int CallDivide01(int x,int y){

    int ret = divide01(x, y);
    if (ret == -1){
        return -100; //我这里用-100 标示除数为 0 的异常
    }
    return ret;
}

void test01(){

    //处理异常
    int ret = divide01(10,0);
    if (ret == -1){
        return;
    }
    cout << "ret:" << ret << endl;

    //是正确的，但是我们错误的把结果当成异常来处理了
    ret = divide01(10, -10);
    if (ret == -1){
        return;
    }
    cout << "ret:" << ret << endl;

    //每一层函数都要处理异常,而且返回值不一样，有的函数用 0 标示正确，-1 标示错误，有的则相反
等
    CallDivide01(10,20);
}

//c++异常机制基本语法
int divide02(int x, int y){
    if (y == 0){
        throw y;
    }
    return x / y;
}
int CallDivide02(int x, int y){
    //divide02(x,y); //隐式再抛异常
    try{
        divide02(x,y);
```

```
    }
    catch (...){
        //异常不处理，继续上抛异常
        cout << "CallDivide02 不处理异常，再抛异常!" << endl;
        throw;
    }
}
void test02(){

    //1.发生异常之后，是跨函数
    try{
        divide02(10,0);
    }
    catch (int e){
        cout << "除数是:" << e << endl;
    }
    catch ( ...){
        cout << "抛出未知异常!" << endl;
    }

    //2. 抛出异常后，可以不处理，再往上抛异常
    try{
        CallDivide02(10, 0);
    }
    catch (int e){
        cout << "除数是:" << e << endl;
    }
    catch (...){
        cout << "抛出未知异常!" << endl;
    }
}

int main(){
    test01();
    test02();
    system("pause");
    return 0;
}
```

- 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。

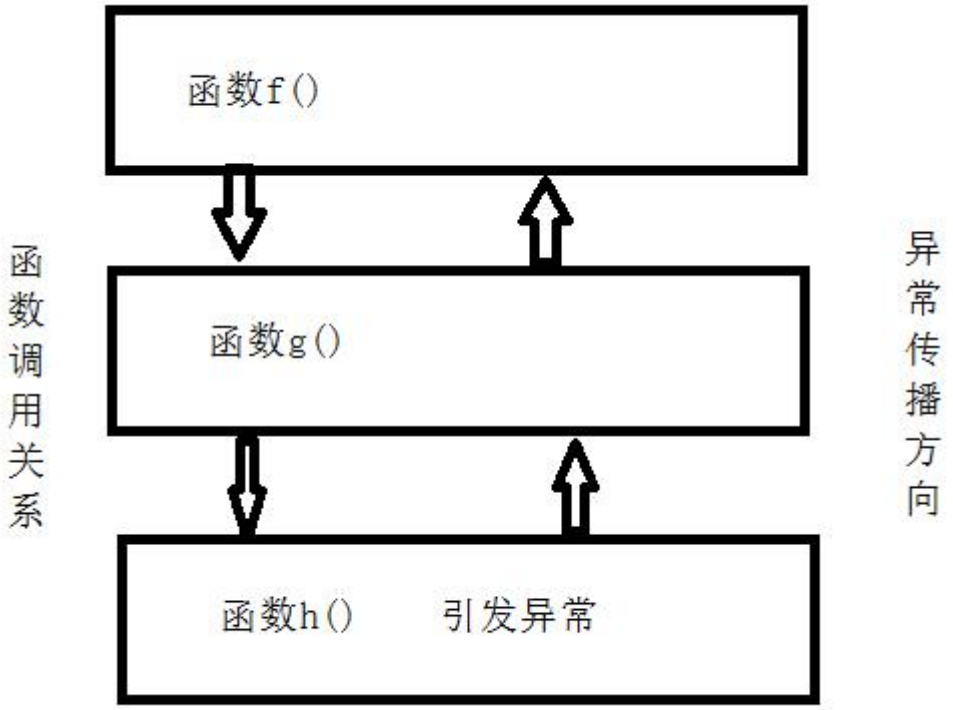
异常机制和函数机制互不干涉,但是**捕捉方式是通过严格类型匹配**。捕捉相当于函数返回类型的匹配，而不是函数参数的匹配，所以捕捉不用考虑一个抛掷中的多种数据类型匹配问题。

案例 2:

```
#include<iostream>
using namespace std;
#include"MyVector.hpp"

//1. 传统处理异常的方式，通过返回整形值标示错误
int divide01(int x, int y){
    if (y == 0){
        throw 'a';
    }
    return x / y;
}

void test01(){
    try{
        divide01(10,0);
    }
#if 0
    catch (int c){  //捕获不到异常
        cout << "异常!" << endl;
    }
#endif
    catch (char c){  //捕获到异常
```

```
        cout << "异常!" << endl;
    }
}

int main(){
    test01();
    system("pause");
    return 0;
}
```

异常处理的基本思想：



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。

异常是针对抽象编程中的一系列错误处理，c++不能借助函数机制，因为栈结构本质先进后出，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。

3.2 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;
#include"MyVector.hpp"

//异常类
class MyException{
public:
	MyException(){
		cout << "构造函数!" << endl;
	}
	~MyException(){
		cout << "析构函数!" << endl;
	}
};

int divide(){
	MyException ex1, ex2;
	cout << "要发生异常了...." << endl;
	throw 1;
}

int main(){

	try{
		divide();
	}
	catch(int e){
		cout << "捕获异常!" << endl;
	}

	system("pause");
	return 0;
}
```

3.3 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锁不允许抛出的异常,unexcepted 函数会被调用，该函数默认行为调用 terminate 函数中断程序。

```
#include<iostream>
using namespace std;
#include"MyVector.hpp"

//表明这个函数能够且只能抛出这三种类型的异常
void TestException01() throw(int, float, char){
    //如果在类的内部，你抛出非指定三种异常类型，那么会导致程序调用 terminate 函数，中断执行
    throw "exception";
}

//如果你这个函数禁止抛任何异常
void TestException02() throw(){
    throw "exception"; //如果强制抛异常，那么也会导致程序调用 terminate 函数，中断执行
}

//这种普通函数，在函数名后不加 throw，那么可以抛出任何类型异常
void  TestException03(){
    throw "exception";
}

int main(){

    //TestException01();
```

```
    //TestException02();
    try{
        TestException03();
    }
    catch (char* e){
        cout << "异常捕获:" << e << endl;
    }


    system("pause");
    return 0;
}
```

3.4 异常类型和异常变量的生命周期

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

注意：异常对象的内存模型

```
#include<iostream>
using namespace std;
#include"MyVector.hpp"

//普通处理异常方式 通过 return err
int copy01(char* to,char* from){
    if (to == NULL){
        return -1;
    }
    if (from == NULL){
        return -2;
    }
    while (*from != '\0'){
        *to = *from;
        to++;
        from++;
    }
    return 0;
}
//普通处理异常
```

```
void test01(){

	char* src = "abcdefg";
	char buf[1024] = { 0 };
	int ret = copy01(buf,src);
	if (ret < 0){
		switch (ret)
		{
		case -1:
			cout << "目标空间为空!" << endl;
			break;
		case -2:
			cout << "源空间为空!" << endl;
			break;
		default:
			cout << "其他异常问题!" << endl;
			break;
		}
		return;
	}
	cout << "buf:" << buf << endl;
}

//抛出 int 类型异常
void copy02(char* to, char* from){
	if (to == NULL){
		throw  -1;
	}
	if (from == NULL){
		throw - 2;
	}
	while (*from != '\0'){
		*to = *from;
		to++;
		from++;
	}
}

void test02(){

	char* src = "abcdefg";
	char buf[1024] = { 0 };
	try{
		copy02(buf, NULL);
```

```
    }
    catch (int e){

        switch (e){
        case -1:
            cout << "目标空间为空!" << endl;
            break;
        case -2:
            cout << "源空间为空!" << endl;
            break;
        default:
            cout << "其他异常问题!" << endl;
            break;
        }
    }
    cout << "buf:" << buf << endl;
}

//抛出 char*类型异常
void copy03(char* to, char* from){
    if (to == NULL){
        throw "目标空间为空!";
    }
    if (from == NULL){
        throw "源空间为空!";
    }
    while (*from != '\0'){
        *to = *from;
        to++;
        from++;
    }
}

void test03(){

    char* src = "abcdefg";
    char buf[1024] = { 0 };
    try{
        copy03(buf, NULL);
    }
    catch (char* e){
        cout << e << endl;;
    }
    cout << "buf:" << buf << endl;
```

```
}

//抛出对象类型异常
//定义异常类

//目标空间空异常类
class TargetSpaceNullException{
public:
    TargetSpaceNullException(){
        cout << "TargetSpaceNullException 构造函数!" << endl;
    }
    TargetSpaceNullException(const TargetSpaceNullException& ex){
        cout << "TargetSpaceNullException 拷贝构造!" << endl;
    }
    ~TargetSpaceNullException(){
        cout << "TargetSpaceNullException 析构函数!" << endl;
    }
};
//源空间空异常类
class SourceSpaceNullException{
public:
    SourceSpaceNullException(){
        cout << "SourceSpaceNullException 构造函数!" << endl;
    }
    SourceSpaceNullException(const SourceSpaceNullException& ex){
        cout << "SourceSpaceNullException 拷贝构造!" << endl;
    }
    ~SourceSpaceNullException(){
        cout << "SourceSpaceNullException 析构函数!" << endl;
    }
};

void copy04(char* to, char* from){
    if (to == NULL){
        throw TargetSpaceNullException(); //产生匿名对象
    }
    if (from == NULL){
        //throw &(SourceSpaceNullException());
        throw new SourceSpaceNullException();
    }
    while (*from != '\0'){
        *to = *from;
        to++;
        from++;
```

```
    }
}

void test04(){

    char* src = "abcdefg";
    char buf[1024] = { 0 };
    try{
        copy04(buf, NULL);
    }
    catch (TargetSpaceNullException& e){ //是将匿名对象进行拷贝了吗?
        //结论 1:如果接受异常时，用异常变量去接，那么会拷贝匿名异常对象
        //结论 2:如果接受异常时，用引用去接，那么不需要拷贝
        cout << "目标空间为空!" << endl;
    }
    catch (SourceSpaceNullException* e){
        //结论 3 如果接受指针，那么要考虑指针是否是野指针的问题
        //这里应该手动管理异常类的生命周期
        cout << "源空间为空!" << endl;
        delete e; //析构异常对象空间
    }
    cout << "buf:" << buf << endl;
}
//继承在异常中的使用
//抽象异常基类
class AbstractBaseException{
public:
    virtual void printExceptionError() = 0;
};
//目标空间空异常类
class MyTargetSpaceNullException : public AbstractBaseException{
public:
    MyTargetSpaceNullException(){
        cout << "MyTargetSpaceNullException 构造函数!" << endl;
    }
    MyTargetSpaceNullException(const MyTargetSpaceNullException& ex){
        cout << "TargetSpaceNullException 拷贝构造!" << endl;
    }
    ~MyTargetSpaceNullException(){
        cout << "MyTargetSpaceNullException 析构函数!" << endl;
    }
    virtual void printExceptionError(){
        cout << "目标空间为空!" << endl;
    }
```

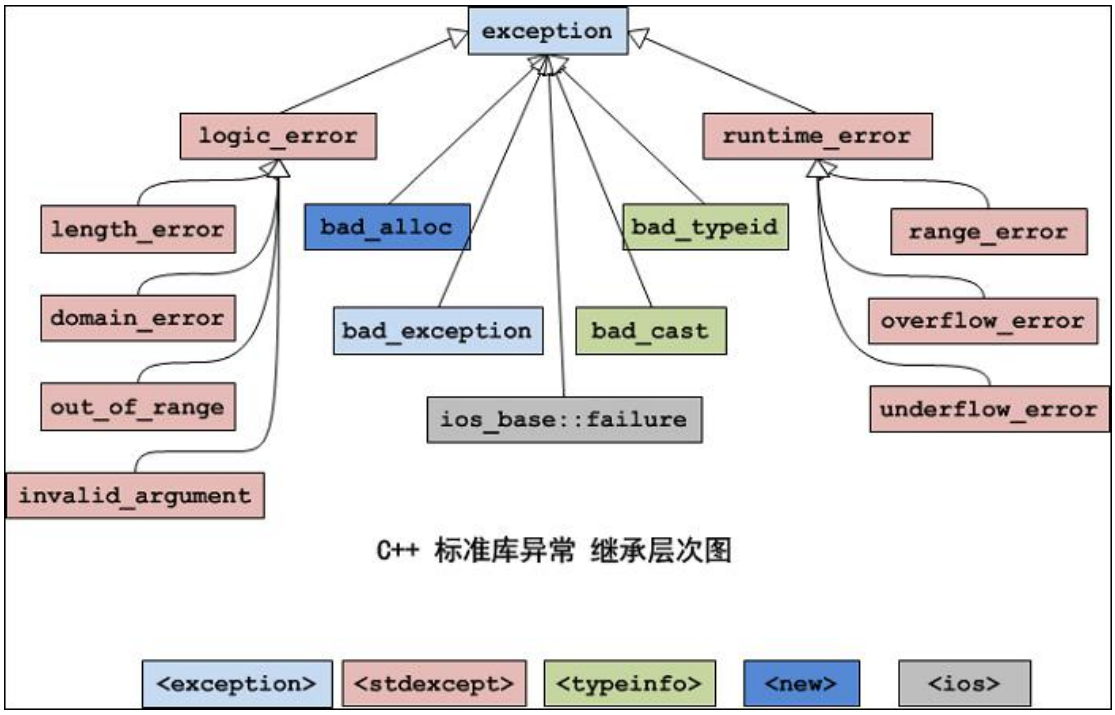
```
};
//源空间空异常类
class MySourceSpaceNullException : public AbstractBaseException{
public:
	MySourceSpaceNullException(){
		cout << "MySourceSpaceNullException 构造函数!" << endl;
	}
	MySourceSpaceNullException(const MySourceSpaceNullException& ex){
		cout << "MySourceSpaceNullException 拷贝构造!" << endl;
	}
	~MySourceSpaceNullException(){
		cout << "MySourceSpaceNullException 析构函数!" << endl;
	}
	virtual void printExceptionError(){
		cout << "源空间为空!" << endl;
	}
};

void copy05(char* to, char* from){
	if (to == NULL){
		throw MyTargetSpaceNullException();
	}
	if (from == NULL){
		throw MySourceSpaceNullException();
	}
	while (*from != '\0'){
		*to = *from;
		to++;
		from++;
	}
}

void test05(){

	char* src = "abcdefg";
	char buf[1024] = { 0 };
	try{
		copy05(NULL, src);
	}
	catch (AbstractBaseException& e){
		e.printExceptionError();
	}
	cout << "buf:" << buf << endl;
}
```

```
int main(){
    //test01();
    //test02();
    //test03();
    //test04();
    test05();
    system("pause");
    return 0;
}
```

3.5 c++标准库异常

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。标准异常被组织成八个。

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

编写自己的异常类

1. 为什么要编写自己的异常类？

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。

案例代码：

```
#define _CRT_SECURE_NO_WARNINGS

#include<iostream>
using namespace std;
#include"MyVector.hpp"



//标准库异常
```

```
class Person{
public:
    Person(int age,string name){
        if (age < 0 || age > 100){
            throw out_of_range("年龄应该在 0-100 之间的数字!");
        }
        mName = name;
        mAge = age;
    }
public:
    string mName;
    int mAge;
};

void test01(){

    try{
        Person p(101, "aaa");
    }
    catch (out_of_range& e){
        cout << e.what() << endl;
    }
}

//自定义异常类
class MyOutRangeException : public exception{
public:
    MyOutRangeException(const char* error){
        int len = strlen(error) + 1;
        this->pError = new char[len];
        strcpy(this->pError,error);
    }
    virtual const char*  what() const{
        return this->pError;
    }
    virtual ~MyOutRangeException(){
        cout << "析构函数!" << endl;
        if (pError != NULL){
            delete[] this->pError;
        }
    }
public:
    char* pError;
};
```

```
class Person02{
public:
	Person02(int age, string name){
		if (age < 0 || age > 100){
			throw MyOutRangeException("年龄应该在 0-100 之间的数字!");
		}
		mName = name;
		mAge = age;
	}
public:
	string mName;
	int mAge;
};

void test02(){

	try{
		Person02 p(101,"aaa");
	}
	catch (exception& e){
		cout << e.what() << endl;
	}
}

int main(){

	test01();
	test02();
	system("pause");
	return 0;
}
```

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。

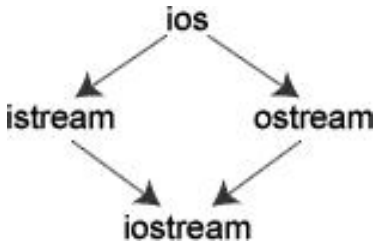
对内存中指定的空间进行输入和输出。通常指定一个字符数组作为存储空间(实际上可以利用该空间存储任何信息)。这种输入和输出称为字符串输入输出，简称串 I/O。

C++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

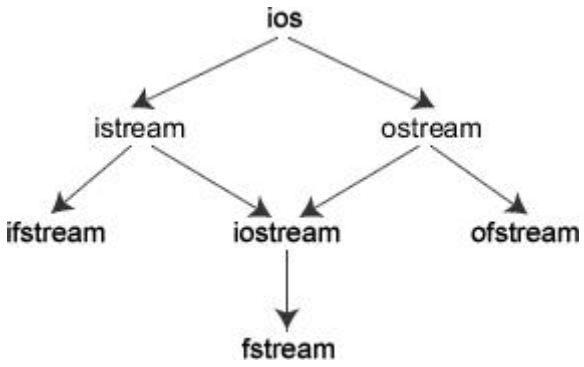


.I/O类库中的常用流类

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

cont 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。

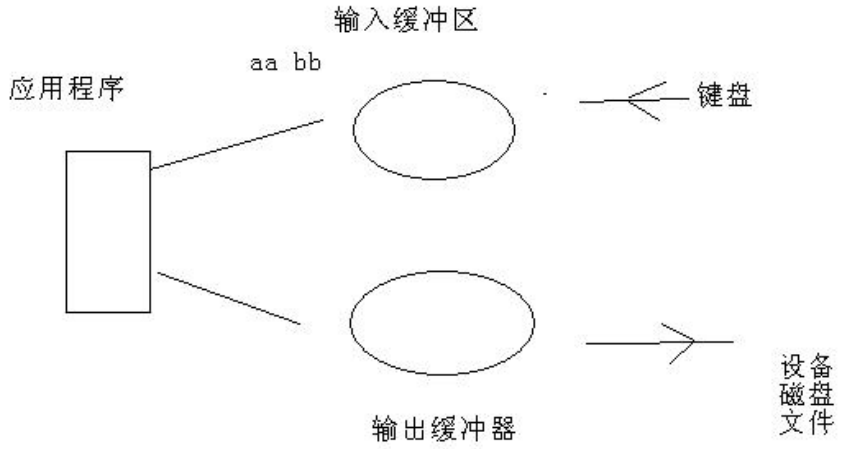
cerr 流对象

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。

缓冲区的概念:



1 读和写是站在应用程序的角度来说的

4.2.1 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()

```
#define _CRT_SECURE_NO_WARNINGS

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

//cin.get()
void test01(){

	char ch;
	while ((ch = cin.get()) != EOF){
		cout << ch << endl;
	}
}

//链式编程
void test02(){

#if 0
	char c1, c2, c3;
	cin.get(c1);
	cin.get(c2);
	cin.get(c3);
	cout << c1 << endl;
	cout << c2 << endl;
	cout << c3 << endl;
	cin.get();

	//开始链式编程
	cin.get(c1).get(c2).get(c3);
	cout << c1 << endl;
	cout << c2 << endl;
	cout << c3 << endl;
#endif

	//读取字符串
	char buf[1024] = { 0 };
	cin >> buf; //碰到空格不继续输出了
	cout << buf << endl;

}
```

```
//cin.getline
void test03(){

	char buf1[256] = { 0 };
	char buf2[256] = { 0 };

	cin.getline(buf1, 256); //读一行数据
	cout << "buf1 : " << buf1 << endl;
}

//cin.ignore() 忽略当前的字符 从当前位置开始忽略几个字符
void test04(){

	char buf1[256] = { 0 };
	char buf2[256] = { 0 };
	cin >> buf1;  //现在缓冲区有数据了
	cout << "buf1:" << buf1 << endl;
	cin.ignore(2); //不传参数，默认忽略一个字符
	char ch = cin.peek(); //peek 只是偷窥缓冲区中的数据，并不取走数据
	cout << ch << endl;
	cin.getline(buf2,256);
	cout << "buf2:" << buf2 << endl;
}

//cin.putback() 将数据放回缓冲区
void test05(){

	cout << "请输入一个字符串或者数字:" << endl;
	char ch = cin.get();//从缓冲区拿到第一个字符
	if (ch >= '0' && ch <= '9'){
		int number;
		cin.putback(ch);
		cin >> number;

		cout << "数字:" << number << endl;
	}
	else{
		char buf[1024] = {0};
		cin.putback(ch);
		cin.getline(buf,1024);
		cout << "字符串:" << buf << endl;

	}
```

```
}

int main(){

    //test01();
    //test02();
    //test03();
    //test04();
    test05();
    system("pause");
    return 0;
}
```

4.2.1 标准输出流

cout.flush() //刷新缓冲区

cout.put() //向缓冲区写字符

cout.write() //二进制流的输出

cout.width() //输出格式控制

cout.fill()

cout.setf(标记)

案例：

```
#define _CRT_SECURE_NO_WARNINGS

#include<iostream>
using namespace std;

//cout.put
void test01(){

    cout << "hello world!" << endl;
    cout.put('h').put('e').put('l') << endl;
```

```
}

//cout.write
void test02(){
	char* str = "hello world!";
	cout.write(str, strlen(str)) << endl;
	cout.write(str, strlen(str) - 2) << endl;
}

int main(){

	//test01();
	test02();

	system("pause");
	return 0;
}
```

C++格式化输出，C++输出格式控制

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

```
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
	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
    system("pause");
    return 0;
}
```

运行结果如下：

```
input a:34↙(输入 a 的值)
dec:34              (十进制形式)
hex:22              (十六进制形式)
oct:42              (八进制形式)
        China           (域宽为)
*****China          (域宽为,空白处以'*'填充)
pi=3.14285714e+00       (指数形式输出,8 位小数)
pi=3.1429e+00           (指数形式输出,4 位小数)
pi=3.143            (小数形式输出,精度仍为)
```

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

例：用流控制成员函数输出数据。

```
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);
    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
    system("pause");
    return 0;
}
```

运行情况如下：

```
dec:21(十进制形式)
hex:0x15          (十六进制形式,以 x 开头)
oct:025           (八进制形式,以开头)
     China        (域宽为)
*****China        (域宽为,空白处以'*'填充)
pi=**3.142857e+00     (指数形式输出,域宽,默认位小数)
+***3.142857        (小数形式输出,精度为,最左侧输出数符“+”)
```

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 I ios::showpos); //包含两个状态标志，用"|"组合

3) 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。

```
#define _CRT_SECURE_NO_WARNINGS


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

//使用类的成员函数
void test01(){

    int number = 20;
    cout.setf(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)
    cout << dec << number << endl; //10 进制输出
    cout << oct << number << endl; //8 进制输出
    cout << hex << number << endl; //16 进制输出

    cout.width(10);//设置输出宽度
    cout.fill('*');// 宽度不够的地方，用*代替
    //cout.setf(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
    cout << number << endl;
```

```
}

//使用控制符
void test02(){
    int number = 20;
    cout << hex
        << setiosflags(ios::showbase)
        << setiosflags(ios::internal)
        << setfill('^')
        << setw(10)
        << number
        << endl;
}

int main(){

    //test01();
    test02();
    system("pause");
    return 0;
}
```

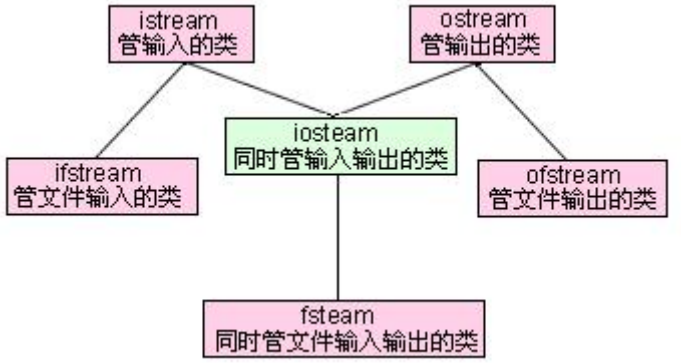
4.3 文件的读写

- 文件输入流 ifstream
- 文件输出流 ofstream
- 文件输入输出流 fstream
- 文件的打开方式
- 文件流的状态
- 文件流的定位：文件指针（输入指针、输出指针）
- 文本文件和二进制文件

4.3.1 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。

4.3.2 C++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 l 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 l 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";
```

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。

4.3.3 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。

案例 1：写文件，然后读文件

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

int main92()
{
    char fileName[80];
    char buffer[255];

    cout << "请输入一个文件名: ";
```

```
    cin >> fileName;

    ofstream fout(fileName, ios::app);
    fout << "1111111111111111111\n";
    fout << "2222222222222222\n";
    //cin.ignore(1,'\n');
    cin.getline(buffer,255); //从键盘输入
    fout << buffer << "\n";
    fout.close();

    ifstream fin(fileName);
    cout << "Here's the the content of the file: \n";
    char ch;
    while(fin.get(ch))
        cout << ch;

    cout << "\n***End of file contents.***\n";
    fin.close();
    system("pause");
    return 0;
}
```

案例 2（自学扩展思路）

ofstream 类的默认构造函数原形为：

```
ofstream::ofstream(constchar *filename, int mode = ios::out,
 int penprot = filebuf::openprot);
```

- filename：　　要打开的文件名
- mode：　　要打开文件的方式
- prot：　　打开文件的属性

其中 mode 和 openprot 这两个参数的可选项表见下表：

mode 属性表	
ios::app	以追加的方式打开文件
ios::ate	文件打开后定位到文件尾，ios:app 就包含有此属性
ios::binary	以二进制方式打开文件，缺省的方式是文本方式。两种方式的区别见前文
ios::in	文件以输入方式打开
ios::out	文件以输出方式打开

ios::trunc	如果文件存在，把文件长度设为 0

可以用“|”把以上属性连接起来，如 ios::out|ios::binary。

openprot 属性表	
属性	含义
0	普通文件，打开访问
1	只读文件
2	隐含文件
4	系统文件

可以用“或”或者“+”把以上属性连接起来 ，如 3 或 1|2 就是以只读和隐含属性打开文件。

```
#include <fstream>
usingnamespace std;

int main()
{
        ofstream myfile("c:\\1.txt",ios::out|ios::trunc,0);
        myfile<<"传智播客"<<endl<<"网址："<<"www.itcast.cn";
        myfile.close()
        system("pause");
}
```

文件使用完后可以使用 close 成员函数关闭文件。

ios::app 为追加模式，在使用追加模式的时候同时进行文件状态的判断是一个比较好的习惯。

```
#include <iostream>
#include <fstream>
usingnamespace std;
int main()
{
        ofstream myfile("c:\\1.txt",ios::app,0);
        if(!myfile)//或者写成 myfile.fail()
        {
                cout<<"文件打开失败，目标文件状态可能为只读！";
                system("pause");
                exit(1);
        }
        myfile<<"传智播客"<<endl<<"网址："<<" www.itcast.cn "<<endl;
        myfile.close();
}
```

在定义 ifstream 和 ofstream 类对象的时候，我们也可以不指定文件。以后可以通过成员函数 open()显式的把一个文件连接到一个类对象上。

例如：

```
#include <iostream>
#include <fstream>
usingnamespace std;
int main()
{
	ofstream myfile;
	myfile.open("c:\\1.txt",ios::out|ios::app,0);
	if(!myfile)//或者写成 myfile.fail()
	{
		cout<<"文件创建失败,磁盘不可写或者文件为只读!";
		system("pause");
		exit(1);
	}
	myfile<<"传智播客"<<endl<<"网址： "<<"www.itcast.cn"<<endl;
	myfile.close();
}
```

下面我们来看一下是如何利用 ifstream 类对象，将文件中的数据读取出来，然后再输出到标准设备中的例子。

```
#include <iostream>
#include <fstream>
#include <string>
usingnamespace std;
int main()
{
	ifstream myfile;
	myfile.open("c:\\1.txt",ios::in,0);
	if(!myfile)
	{
		cout<<"文件读错误";
		system("pause");
		exit(1);
	}
	char ch;
	string content;
	while(myfile.get(ch))
	{
		content+=ch;
		cout.put(ch);//cout<<ch;这么写也是可以的
```

```
        }
        myfile.close();
        cout<<content;
        system("pause");
}
```

上例中，我们利用成员函数 get()，逐一的读取文件中的有效字符，再利用 put()成员函数，将文件中的数据通过循环逐一输出到标准设备(屏幕) 上， get()成员函数会在文件读到默尾的时候返回假值，所以我们可以利用它的这个特性作为 while 循环的终止条件，我们同时也在上例中引入了 C++风格的 字符串类型 string，在循环读取的时候逐一保存到 content 中，要使用 string 类型，必须包含 string.h 的头文件。

我们在简单介绍过 ofstream 类和 ifstream 类后，我们再来看一下 fstream 类，fstream 类是由 iostream 派生而来，fstream 类对象可以同对文件进行读写操作。

```
#include <iostream>
#include <fstream>
usingnamespace std;
int main()
{
        fstream myfile;
        myfile.open("c:\\1.txt",ios::out|ios::app,0);
        if(!myfile)
        {
                cout<<"文件写错误,文件属性可能为只读!"<<endl;
                system("pause");
                exit(1);
        }
        myfile<<"传智播客"<<endl<<"网址："<<"www.itcast.cn"<<endl;
        myfile.close();

        myfile.open("c:\\1.txt",ios::in,0);
        if(!myfile)
        {
                cout<<"文件读错误,文件可能丢失!"<<endl;
                system("pause");
                exit(1);
        }
        char ch;
        while(myfile.get(ch))
        {
                cout.put(ch);
        }
        myfile.close();
```

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

由于 fstream 类可以对文件同时进行读写操作，所以对它的对象进行初始话的时候一定要显式的指定 mode 和 openprot 参数。

4.3.4 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。

案例 1

```
//二进制
int main()
{
    char fileName[255] = "c:/teacher.dat";
    ofstream fout(fileName,ios::binary);
    if(!fout)
    {
```

```
        cout << "Unable to open " << fileName << " for writing.\n";
        return(1);
    }

    Teacher t1(31, "31");
    Teacher t2(32, "32");
    fout.write((char *)&t1,sizeof Teacher);
    fout.write((char *)&t2,sizeof Teacher);
    fout.close();

    cout << "保存对象到二进制文件里成功!" << endl;

    ifstream fin(fileName,ios::binary);
    if(!fin)
    {
        cout << "Unable to open " << fileName << " for reading.\n";
        return (1);
    }
    Teacher tmp(100,"100");

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

    return 0;
}
```

```
#define _CRT_SECURE_NO_WARNINGS

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

//文件读操作
void test01(){

    char* fileName = "C:\\Users\\孟宝亮\\Desktop\\test.txt";

    //创建读文件的流对象
    //ifstream ism(fileName);
    ifstream ism;
```

```
    ism.open(fileName);
    if(!ism){
        cout << fileName << "打开失败!" << endl;
    }
#if 0
    char ch;
    //逐个字符读数据
    while (ism.get(ch)){
        cout << ch;
    }
#endif
    char buf[1024] = { 0 };
    while (ism.getline(buf,1024)){
        cout << buf << endl;
    }


    //关闭文件
    ism.close();

}

//文件写操作
void test02(){
    char* fileName = "C:\\Users\\孟宝亮\\Desktop\\test_output.txt";
    ofstream osm(fileName, ios::app);//追加方式写文件 如果文件不存在，则创建
    if (!osm){
        cout << "文件打开失败!" << endl;
    }
    osm << "hello world!";
    //关闭文件流
    osm.close();
}

//二进制方式读写文件
class Person{
public:
    Person(){
        memset(this->name, 0, 64);
        this->age = 0;
    }
    Person(char* name,int age){
        memset(this->name, 0, 64);
        strcpy(this->name, name);
```

```
        this->age = age;
    }
public:
    char name[64];
    int age;
};

void test03(){

    char* fileName = "C:\\Users\\孟宝亮\\Desktop\\test_binary.txt";
    ofstream osm(fileName, ios::binary);
    //创建对象
    Person p1("aaa", 20), p2("bbbb", 30);
    //对象写入文件 对象序列化
    osm.write((char*)&p1, sizeof(Person));
    osm.write((char*)&p2, sizeof(Person));
    osm.close();

    //创建输入流
    ifstream ism(fileName, ios::in);
    Person p3, p4;
    ism.read((char*)&p3, sizeof(Person));
    ism.read((char*)&p4, sizeof(Person));
    //输出内容
    cout << "p3.name:" << p3.name << " p3.age:" << p3.age << endl;
    cout << "p4.name:" << p4.name << " p4.age:" << p4.age << endl;
    //关闭文件输入流
    ism.close();
}
int main(){

    //test01();
    //test02();
    test03();

    system("pause");
    return 0;
}
```

4.4 作业练习

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。

4.5 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;
}
```

第三题

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

第四题

```
#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

```
#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;
}
```