C++复习

day6继承

单继承

继承性是面向对象程序设计中最重要的机制。这种机制**提供了无限重复利用程序资源的一种途径。**通过C++语言中的继承机制，**可以扩充和完善旧的程序设计以适应新的需求。**这样不仅可以节省程序开发的时间和资源，并且为未来程序增添了新的资源。

```
class Student
{   int num;
    char name[30];
    char sex;
public:
    void display( )                    // 对成员函数display的定义
    {cout<<"num: "<<num<<endl;
    cout<<"name: "<<name<<endl;
    cout<<"sex: "<<sex<<endl;}
};

class Studend1
{   int num;             // 此行原来已有
    char name[20];           // 此行原来已有
    char sex;            // 此行原来已有
    int age;
    char addr [20] ;
public:
    void display( ) ;                // 此行原来已有
    {cout<<"num: "<<num<<endl;       // 此行原来已有
     cout<<"name: "<<name<<endl;     // 此行原来已有
     cout<<"sex: "<<sex<<endl;           // 此行原来已有
     cout<<"age: "<<age<<endl;
     cout<<"address: "<<addr<<endl;}
};
```

利用原来定义的类Student作为基础，再加上新的内容即可，以减少重复的工作量。 C++提供的继承机制就是为了解决这个问题。

在C++中所谓“继承”就是在一个已存在的类的基础上建立一个新的类。已存在的类称为“基类(base class)”或“父类(father class)”。新建立的类称为“派生类(derived class)”或“子类(son class)”。

```
class Student1: public Student//声明基类是Student
{private:
        int age;    //新增加的数据成员
        string addr;  //新增加的数据成员
  public:
    void display_1( )  //新增加的成员函数
    {  cout<<"age: "<<age<<endl;
     cout<<"address: "<<addr<<endl;
        }
};
```

类A派生类B：类A为基类，类B为派生类。一个派生类可以从一个基类派生，也可以从多个基类派生。**从一个基类派生的继承称为单继承；从多个基类派生的继承称为多继承。**

通过继承机制，可以利用已有的数据类型来定义新的数据类型。所定义的新的数据类型不仅拥有新定义的成员，而且还同时拥有旧的成员。**我们称已存在的用来派生新类的类为基类，又称为父类。**由已存在的类派生出的新类称为派生类，又称为子类。

在建立派生类的过程中，基类不会做任何改变，**派生类则除了继承基类的所有可引用的成员变量和成员函数外，还可另外定义本身的成员变量和处理这些变量的函数**，由于派生类可继承基类的成员变量和成员函数，因此在基类中定义好的数据和函数等的程序代码可重复使用，这样可以提高程序的可靠性。

当从已有的类中派生出新的类时，可以对派生类做以下几种变化：

1、可以继承基类的成员数据或成员函数。

2、可以增加新的成员变量。

3、可以增加新的成员函数。

4、可以重新定义已有的成员函数。

5、可以改变现有的成员属性。

在C++中有二种继承：单一继承和多重继承。当一个派生类仅由一个基类派生时，称为单一继承；而当一个派生类由二个或更多个基类所派生时，称为多重继承。

```
#include <iostream>
using namespace std;
class A
{
public:
    int z; // 公有数据成员
    A(int a,int b,int c) {
        cout << "调用类A带参：构造函数.\n";
        x = a;
        y = b;
        z = c;
    }
    int getx() {
        return x;
    }
    int gety() {
        return y;
    }
    int getz() {
```

```
        return z;
    }
    void print() {
        cout << "x=" << x << ",y=" << y << ",z=" << z << endl << endl;
    }

protected:
    int y; // 保护数据成员
private:
    int x;  // 私有数据成员
};
class B : public A
{
public:
    B(int a,int b,int c,int d,int e):A(a,b,c){
        cout << "调用派生类B带参：构造函数.\n";
        m = d;
        n = e;
    }
    void print() {
        cout << "x=" << getx() << ",y=" << y << ",z=" << z << ",m="<<m<<",n="
<<n<<endl << endl;
    }
    int sum()
    {
        return (getx() + y + z + m + n);
    }
private:
    int m, n;
};
int main()
{
    B obj(1, 2, 3, 4, 5);  // 派生类创建对象obj，且初始化参数 1 2 3 4 5
    obj.print();
    cout << "\nsum=" << obj.sum() << endl;
    cout << "x=" << obj.getx() << endl;
    cout << "y=" << obj.gety() << endl;
    cout << "z=" << obj.getz() << endl;
    return 0;
}
```

但派生并不是简单的扩充，有可能改变基类的性质。

有三种派生方式：公有派生、保护派生、私有派生。

```
class B: public   A{...};
class B: protected   A{...};
class B: private   A{...};  //默认
class B:   A {...};  //默认为私有派生
```

默认的是私有派生。

从一个基类派生一个类的一般格式为：

```
    //派生类名     继承方式   基类名
class  ClassName:<Access>BaseClassName
{
       private:
        ......;       //私有成员说明
       public:
        ......;       //公有成员说明
       protected:
        ......;       //保护成员说明
  }
```

1、【公有派生】

```
class ClassName :  public BaseClassName
```

公有派生时，**基类中所有成员在派生类中保持各个成员的访问权限。**

基类：public：在派生类和类外可以使用

protected: 在派生类中使用

private: 不能在派生类中使用

2、【私有派生】

```
class ClassName:  private BaseClassName
```

私有派生时，基类中公有成员和保护成员在派生类中均变为私有的，在派生类中仍可直接使用这些成员，基类中的私有成员，在派生类中不可直接使用。

基类：public: (变为私有)在派生类中使用，类外不可使用

protected: (变为私有）在派生类中使用，类外不可使用

private: 不能在派生类中和类外使用

3、【保护派生】

```
class ClassName:  protected  BaseClassName
```

保护派生时，基类中公有成员和保护成员在派生类中均变为保护的和私有的，在派生类中仍可直接使用这些成员，基类中的私有成员，在派生类中不可直接使用。

基类：public: (变为保护)在派生类中使用，类外不可使用

protected: (变为私有）在派生类中使用，类外不可使用

private: 不能在派生类中和类外使用

protected 成员是一种具有血缘关系内外有别的成员。它对派生类的对象而言，是公开成员，可以访问，**对血缘外部而言，与私有成员一样被隐蔽。**

4、【抽象类与保护的成员函数】

当定义了一个类，这个类**只能用作基类**来派生出新的类，而不能用这种类来定义对象时，称这种类为抽象类。当对某些特殊的对象要进行很好地封装时，需要定义抽象类。

将**类的构造函数或析构函数**的访问权限定义为**保护**的时，这种类为**抽象类。**

当把类中的构造函数或析构函数说明为**私有**的时，所定义的类**通常是没有任何实用意义**的，一般情况下，不能用它来产生对象，也不能用它来产生派生类。

多继承

可以用多个基类来派生一个类。

格式为：

```
class  类名:<Access>类名1,..., <Access>类名n
{
      private:      ...... ;    //私有成员说明;
      public:       ...... ;    //公有成员说明;
      protected:    ...... ;    //保护的成员说明;
};

class  D: public  A, protected B, private C
{    ....//派生类中新增加成员
};
```

```
#include <iostream>
using namespace std;
class A
{
public:
    A(int a)
    {
        cout << "调用基类A：构造函数.\n";
        x = a;
    }
    ~A()
    {
        cout << "调用基类A：析构函数.\n";
    }
private:
    int x;
};

class B
{
public:
    B(int a)
    {
        cout << "调用基类B：构造函数.\n";
        y = a;
    }
    ~B()
    {
        cout << "调用基类B：析构函数.\n";
    }
```

```
private:
    int y;
};

class C :public A, public B
{
public:
    C(int a,int b):A(a),B(100)
    {
        cout << "调用派生类C：构造函数.\n";
        z = a;
    }
    ~C()
    {
        cout << "调用派生类C：析构函数.\n";
    }
private:
    int z;
};

int main()
{
    C obj(10, 50);
    return 0;
}
```

初始化基类成员

构造函数不能被继承,派生类的构造函数**必须调用基类的构造函数**来初始化基类成员基类**子对象。**

派生类构造函数的调用顺序如下：

基类的构造函数

子对象类的构造函数

派生类的构造函数

```
class B:public A{
         int  y;
         A  a1;
public:
         B(int a, int b):A(a),a1(3){y=b;}
         .......
};
```

当撤销派生类对象时，析构函数的调用正好相反。

```
#include <iostream>
using namespace std;
class A
{
public:
    A()
    {
        cout << "调用基类A：默认构造函数.\n";
```

```
    }
    A(int a)
    {
        cout << "调用基类A：构造函数.\n";
        x = a;
    }
    ~A()
    {
        cout << "调用基类A：析构函数.\n";
    }
private:
    int x;
};
class B
{
public:
    B(int a)
    {
        cout << "调用基类B：构造函数.\n";
        y = a;
    }
    ~B()
    {
        cout << "调用基类B：析构函数.\n";
    }

private:
    int y;
};
class C :public A, public B
{
public:
    C(int a, int b) :A(a), B(100)
    {
        cout << "调用派生类C：构造函数.\n";
        z = a;
    }
    ~C()
    {
        cout << "调用派生类C：析构函数.\n";
    }
private:
    int z;
    A obj1;
};
int main()
{
    C obj(10, 20);

    return 0;
}
```

虚继承

类B 是类A的派生类

类C 是类A的派生类

类D 是类B和类C的派生类

这样，类D中就有两份类A的拷贝

这种同一个公共的基类在派生类中产生多个拷贝，不仅多占用了存储空间，而且可能会造成多个拷贝中的数据不一致和模糊的引用。

在多重派生的过程中，若使公共基类在派生类中只有一个拷贝，则可将这种基类说明为虚基类。在派生类的定义中，只要在基类的类名前加上关键字virtual，就可以将基类说明为虚基类。

```
class B:public  virtual A{
public:
    int y;
    B(int a=0, int b=0 ):A(b) { y=a;}
};
```

由虚基类派生出的对象初始化时，**直接调用**虚基类的构造函数。因此，若将一个类定义为虚基类，**则一定有正确的构造函数可供所有派生类调用。**

再次强调，用虚基类进行多重派生时，若虚基类没有缺省的构造函数，则在每一个派生类的构造函数中**都必须有对虚基类构造函数的调用 （且首先调用）。**

```
#include <iostream>
using namespace std;
class A {
public:
    int x;
    A(int a = 0){
        cout << "调用类A：构造函数." << endl;
        x = a;
    }
};
class B : virtual public A {
public:
    int y;
    B(int a = 0, int b = 0) :A(a) {
        cout << "调用类B：构造函数." << endl;
        y = b;
    }
};
class C :virtual public A {
public:
    int z;
    C(int a = 0, int c = 0) :A(a) {
        cout << "调用类C：构造函数." << endl;
        z = c;
    }
};
class D :public B, public C
```

```
{
public:
    int dx;
    D(int a1, int b, int c, int d, int a2) :B(a1, b), C(a2, c){
        cout << "调用类D：构造函数." << endl;
        dx = d;
    }
};

int main()
{
    D obj(100, 200, 300, 400, 500);
    cout << "\nobj.x=" << obj.x << endl;
    obj.x = 20000;
    cout << obj.x << endl;
    cout << obj.y << endl;
    return 0;
}
```

其他特性

当多继承的基类拥有相同的数据成员或者函数成员是，在子类调用时会发生冲突，需要加上基类的名作为限定符

```
class A{
public:
    int x;
    A(int a = 0){
        x = a;
    }
    void print(){
        cout << "调用类A的print()函数.\n";
        cout << "x=" << x << endl;
    }
};
class B{
public:
    int x;
    B(int a = 0){
        x = a;
    }
    void print(){
        cout << "调用类B的print()函数.\n";
        cout << "x=" << x << endl;
    }
};
class C{
public:
    int x;
    C(int a = 0){
        x = a;
    }
    void print(){
```

```
        cout << "调用类C的print()函数.\n";
        cout << "x=" << x << endl;
    }
};
class D:public A,public B,public C{
public:
    int d;
    D(int a = 0){
        d = a;
    }
};
int main()
{
    D d1;
    //d1.x = 1;//报错，发生冲突
    d1.A::x = 1;//指定基类名
    d1.B::x = 2;
    d1.C::x = 3;
    //d1.print();//报错，发生冲突
    d1.A::print();//指定想要调用的print函数的基类名
    d1.B::print();
    d1.C::print();
    return 0;
}
/*
调用类A的print()函数.
x=1
调用类B的print()函数.
x=2
调用类C的print()函数.
x=3
*/
```

当派生类中新增加的数据或函数与基类中原有的同名时，若不加限制，则优先调用派生类中的成员。

```
#include <iostream>
using namespace std;
class A
{
public:
    int x;
    void print(){
        cout << "调用类A的print()函数.\n";
        cout << "x=" << x << endl;
    }
};
class B
{
public:
    int y;
    void print(){
        cout << "调用类B的print()函数.\n";
        cout << "y=" << y << endl;
    }
};
```

```
class C:public A,public B
{
public:
    int y;  // 类B的和类C均有y的数据成员
};
int main(){
    C c1;
    c1.x = 100;
    c1.y = 200;  // 未指明的情况下，给派生类中的y赋值
    c1.B::y = 300; // 给基类B的y赋值
    c1.A::print();
    c1.B::print();
    cout << "y=" << c1.y << endl;  // 输出派生类中的y的值
    cout << "y=" << c1.B::y << endl; // 输出基类B中的y的值
    return 0;
}
```

命名空间

使用命名空间的目的是对标识符的名称进行本地化，以避免命名冲突。在C++中，变量、函数和类都是大量存在的。如果没有命名空间，这些变量、函数、类的名称将都存在于全局命名空间中，会导致很多冲突。

比如，如果我们在自己的程序中定义了一个函数**strcat**()，这将重写标准库中的**strcat**()函 数，这是因为这两个函数都是位于全局命名空间中的。命名冲突还会发生在一个程序中使用两个或者更多的第三方库的情况中。此时，很有可能，其中一个库中的名 称和另外一个库中的名称是相同的，这样就冲突了。这种情况会经常发生在类的名称上。比如，我们在自己的程序中定义了一个**string**类，而我们程序中使用的某个库中也可能定义了一个同名的类，此时名称就冲突了。

namespace关键字的出现就是针对这种问题的。由于这种机制对于声明于其中的名称都进行了本地化，就使得相同的名称可以在不同的上下文中使用，而不会引起名称的冲突。

namespace 关键字使得我们可以通过创建作用范围来对全局命名空间进行分隔。本质上来讲，一个命名空间就定义了一个范围。定义命名空间的基本形式如下：

```
namespace 名称{//声明}
```

在命名空间中定义的任何东西都局限于该命名空间内。

using 指令

使用 using namespace 指令，这样在使用命名空间时就可以不用在前面加上命名空间的名称。这个指令会告诉编译器，后续的代码将使用指定的命名空间中的名称。

```
//命名空间的嵌套
#include <iostream>
using namespace std;
namespace Onespace {  // 第一命名空间
    void print() {
        cout << "执行Onespace名字空间的print()函数." << endl;
    }
    namespace Twospace {  // 第二命名空间
        void print() {
            cout << "执行Twospace名字空间的print()函数." << endl;
```

```
        }
    }
}
//using namespace Onespace;//执行Onespace
using namespace Onespace::Twospace;//执行Twospace
int main(){
    print();
    return 0;
}
```

模板

模板是泛型编程的基础，泛型编程即以一种独立于任何特定类型的方式编写代码。模板是创建泛型类或函数的蓝图或公式。

1）函数模板

模板函数定义的一般形式如下所示：

```
template <typename type类型> 返回类型 函数名(参数列表)
{
  // 函数的主体
}
//例如
template <typename T>
inline T const& MaxFunc(T const& a, T const& b){
    //inline 内联函数
    //T const& 常量引用
    //MaxFunc 函数名
    //函数的主体
}
```

type 是函数所使用的数据类型的占位符名称。这个名称可以在函数定义中使用。

```
/*
* 求两个数字中，最大数？
*/
#include <iostream>
#include <string>
using namespace std;
template <typename T>
inline T const& MaxFunc(T const& a, T const& b){
    return a > b ? a : b;
}
int main(){
    int i = 90, j = 100;
    cout << "MaxFunc(i,j):" << MaxFunc(i, j) << endl;
    double f1 = 23.9, f2 = 45.88;
    cout << "MaxFunc(f1,f2):" << MaxFunc(f1, f2) << endl;
    string s1 = "Helloworld", s2 = "goodbye";
    cout << "MaxFunc(s1,s2):" << MaxFunc(s1, s2) << endl;
    return 0;
}
```

2）类模板

正如我们定义函数模板一样，我们也可以定义类模板。泛型类声明的一般形式如下所示：

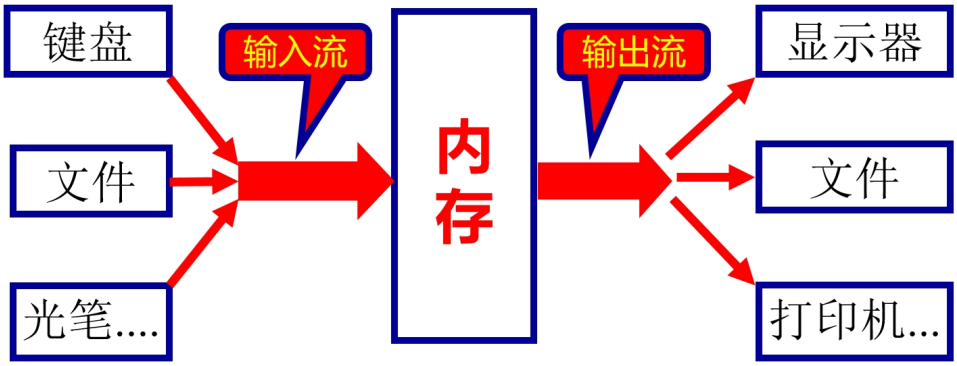
```
template <typename type类型> class class-name {
...
}

//例如
template<typename T> // 模板声明，其中T为类型参数
class CompareSS
{
    ...
}
```

type 是占位符类型名称，可以在类被实例化的时候进行指定。使用一个逗号分隔的列表来定义多个泛型数据类型。

```
#include <iostream>
#include <string>
using namespace std;
template<typename T> // 模板声明，其中T为类型参数
class CompareSS
{
public:
    CompareSS(T a, T b){
        x = a;
        y = b;
    }
    T maxfunc(){
        return (x > y ? x : y);
    }
private:
    int x, y;
};
int main(){
    // 用类模板定义对象，此时T被int参数化替代
    CompareSS<int> obj1(30, 90);
    cout << "\n最大值为:" << obj1.maxfunc() << endl;
    CompareSS<double> obj2(30.98, 90.56);
    cout << "\n最大值为:" << obj1.maxfunc() << endl;
    return 0;
}
```

day7 I/O流类库

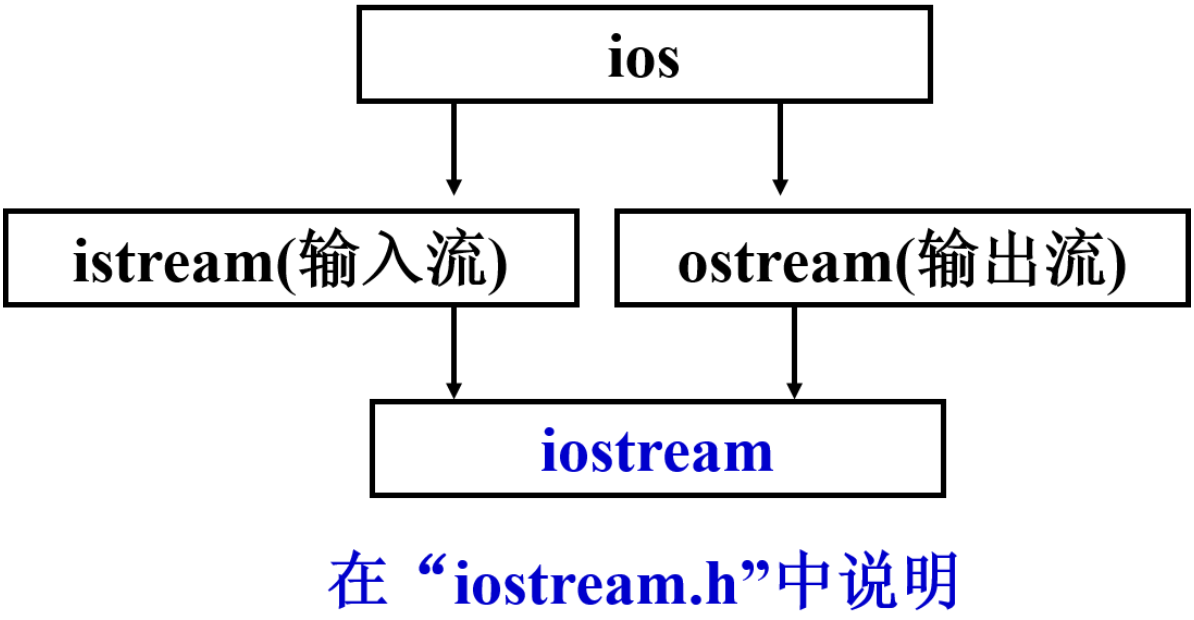
一、输入输出流（I/O Stream）



编译系统已经以运算符或函数的形式做好了对标准外设（键盘、屏幕、打印机、文件）的接口，使用时只需按照要求的格式调用即可。

cin>>x;　　cout<<x;　　cin.get(ch);

C++语言的I/O系统向用户提供一个统一的接口，**使得程序的设计尽量与所访问的具体设备无关**，在用户与设备之间提供了一个抽象的界面：**输入输出流。**



在“iostream.h”中说明

【重载输入(提取)和输出(插入)运算符】

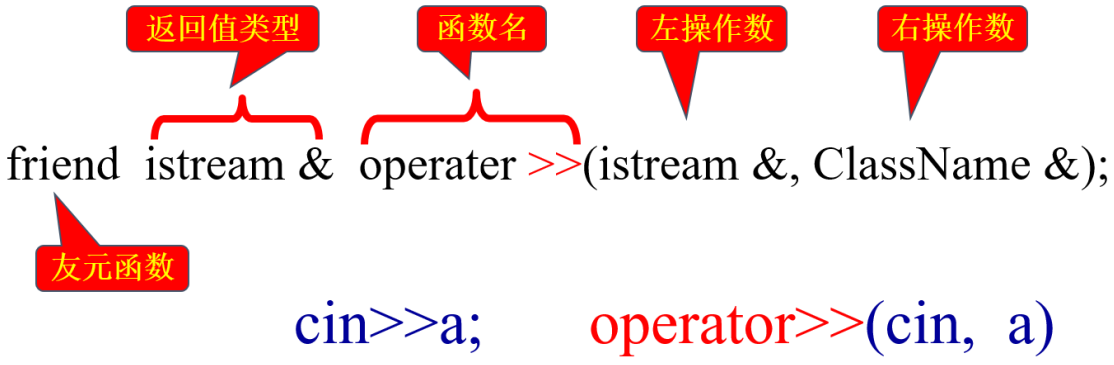
用标准流进行输入/输出时，系统自动地完成数据类型的转换。**对于输入流，要将输入的字符序列形式的数据变换成计算机内部形式的数据（二进制或ASCII）后，再赋给变量，变换后的格式由变量的类型确定。对于输出流，将要输出的数据变换成字符串形式后，送到输出流（文件）中。**

```
#include <iostream>
using namespace std;
class CinCount
{
public:
    CinCount(int a = 0, int b = 0)
    {
        c1 = a;
        c2 = b;
    }
    void showc1c2(void)
```

```
    {
        cout << "c1=" << c1 << '\t' << "c2=" << c2 << endl;
    }
    friend istream& operator>>(istream&, CinCount&);
private:
    int c1, c2;
};
istream& operator>>(istream& is, CinCount& cc)
{
    is >> cc.c1 >> cc.c2;
    return is;
}
int main()
{
    CinCount o1, o2;
    o1.showc1c2();
    o2.showc1c2();
    cin >> o1;
    cin >> o2;
    o1.showc1c2();
    o2.showc1c2();
    return 0;
}
```

在C++中允许用户重载运算符“<<”和“>>”，实现对象的输入和输出。重载这二个运算符时，**在对象所在的类中，将重载这二个运算符的函数说明该类的友元函数。**

重载（提取）运算符的一般格式为：

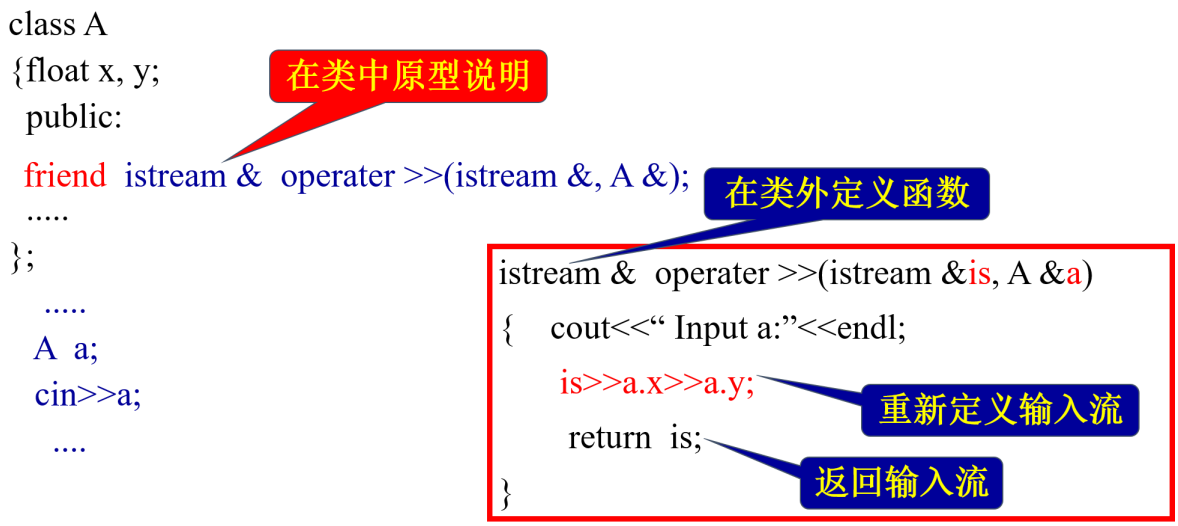


返回值类型：**类istream的引用，cin中可以连续使用运算符“>>”。**

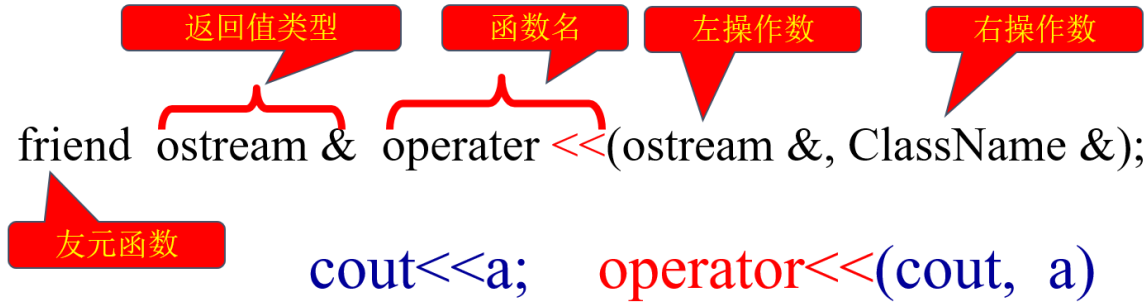
```
istream& operator>>(istream &is, CinCount &cc);

cin>>a;  operator>>(cin,a);

cin>>a>>b;
```

第一个参数：是“>>”的左操作数cin类型，类istream的引用。

第二个参数：是“>>”的右操作数，即预输入的对象的引用。



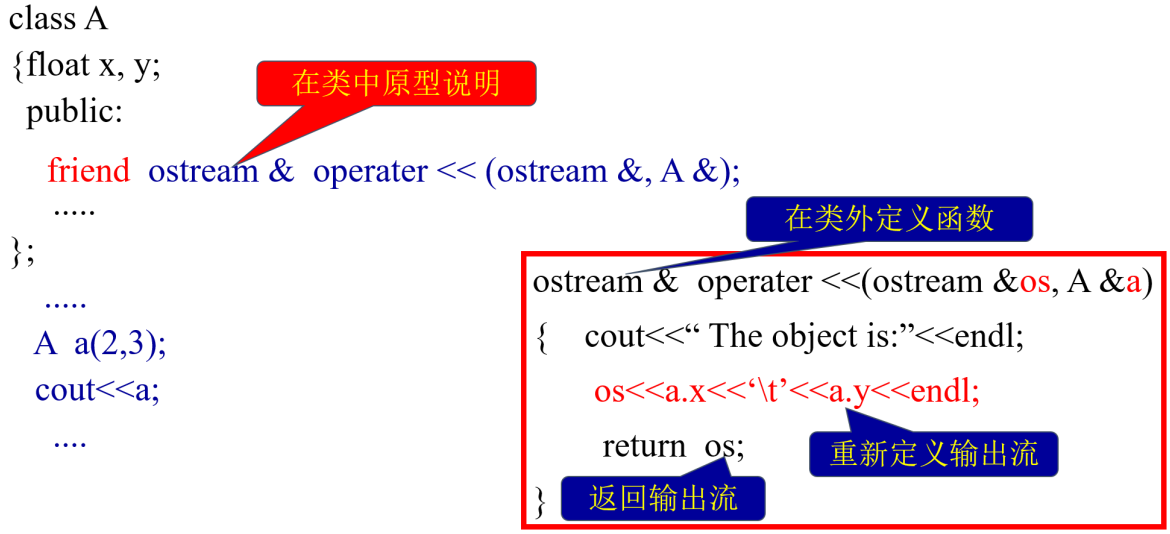
重载输出（插入）运算符的一般格式为：



与输入（提取）运算符比较：

```
friend  istream &  operater >>(istream &, ClassName &);
```

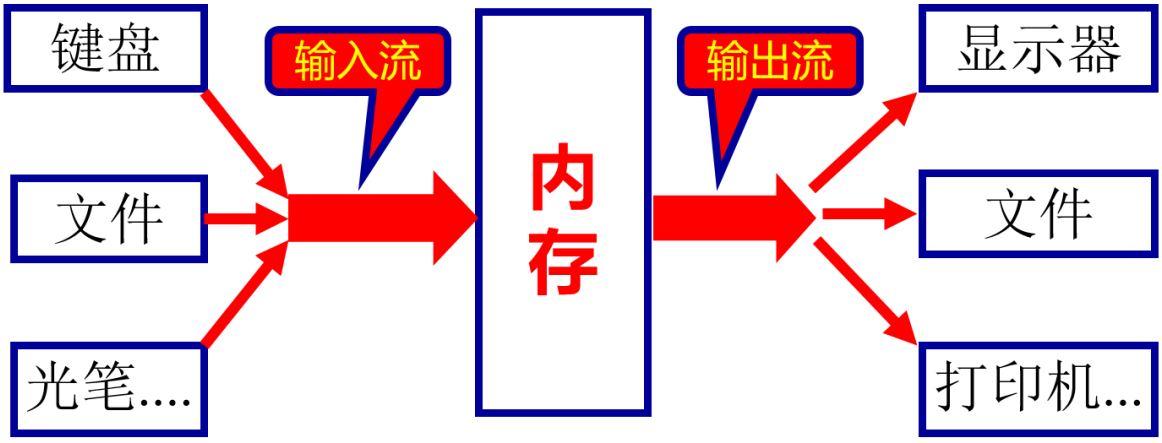
将输入流改为输出流。




```
#include <iostream>
using namespace std;
class InCount
{
public:
```

```
    InCount(int a = 0, int b = 0)
    {
        c1 = a;
        c2 = b;
    }
    void show(void)
    {
        cout << "c1=" << c1 << "\t" << "c2=" << c2 << endl;
    }
    /*
    friend 关键字表示这个函数是类的友元函数。友元函数可以访问类的私有成员和保护成员，即使它不是该类的成员函数。通过将一个函数声明为友元，类可以允许该函数直接访问其内部数据
    istream 是 C++ 标准库中的一个输入流类，用于处理输入操作，如从键盘读取数据。
    istream& 表示返回值是一个对 istream 对象的引用。这种返回值方式允许函数在连锁输入流操作中使用，例如 std::cin >> obj1 >> obj2;
    operator>> 是一个运算符重载函数，用于重载输入流运算符 >>。通过重载这个运算符，你可以定义如何将输入流的数据读取到 InCount 对象中。
    这个函数的形式与标准输入流操作符的形式相同，使得使用自定义类型的输入操作像使用内置类型一样自然。
    istream& is：这是输入流的引用，可以是标准输入流 std::cin 或者其他输入流对象。
    InCount& obj：这是对 InCount 对象的引用，表示将从输入流中读取的数据存储到这个对象中。
    函数返回一个 istream& 类型的引用，这样可以允许连续的输入操作。上面笔记提到过
    */
    friend istream& operator>>(istream&, InCount&);//输入流对应提取运算符，类内友元声明
    friend ostream& operator<<(ostream&, InCount&);//输出流对应插入运算符，类内友元声明
private:
    int c1, c2;
};
istream& operator>>(istream& is, InCount& cc)//类外定义，主要是为了访问私有成员
{
    is >> cc.c1 >> cc.c2;//这里的is就等同于cin,cc是类引用，cc.c1和cc.c2就是私有变量
    return is;
}
ostream& operator<<(ostream& os, InCount& cc)//类外定义
{
    //这里的os就等同于cout,cc是类引用，cc.c1和cc.c2就是私有变量
    os << "c1=" << cc.c1 << "\t" << "c2=" << cc.c2 << endl;
    return os;
}
int main()
{
    InCount obj1, obj2;
    cout << obj1 << obj2 << endl; // 调用输出函数
    cin >> obj1;
    cin >> obj2;
    cout << obj1 << obj2;
    return 0;
}
```

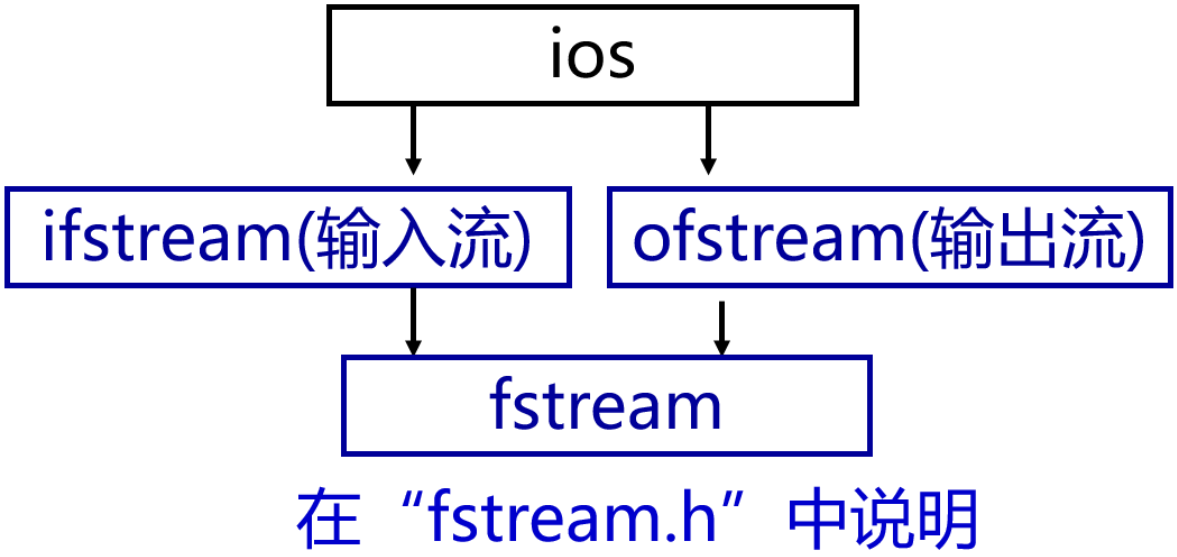
二、文件流



编译系统已经以**运算符或函数**的形式做好了对标准外设（**键盘、屏幕、打印机、文件**）的接口，使用时只需按照要求的格式调用即可。

```
cin>>x;   cout<<x;      cin.get(ch);
```

C++在头文件fstream.h中定义了C++的文件流类体系，当程序中使用文件时，**要包含头文件fstream.h。**



当使用文件时，在程序头有：**#include<fstream.h>。其中定义了各种文件操作运算符及函数。**

程序对文本文件的操作与对键盘、显示器的操作比较：

在涉及**文本文件**的操作时，将输入文件看成键盘，将输出文件看成显示器，格式不变。**只需在程序中增加打开与关闭文件的语句。**

C++标准库专门提供了3个类用于实现文件操作，它们统称为文件流类，这3个类分别为：

•ifstream：专用于从文件中读取数据；

•ofstream：专用于向文件中写入数据；

•fstream：既可用于从文件中读取数据，又可用于向文件中写入数据。

成员方法名	适用类对象	功 能
open()	fstream ifstream ofstream	打开指定文件，使其与文件流对象相关联。
is_open()		检查指定文件是否已打开。
close()		关闭文件，切断和文件流对象的关联。
swap()		交换 2 个文件流对象。
operator>>	fstream ifstream	重载 >> 运算符，用于从指定文件中读取数据。
gcount()		返回上次从文件流提取出的字符个数。该函数常和 get()、getline()、ignore()、peek()、read()、readsome()、putback() 和 unget() 联用。
get()		从文件流中读取一个字符，同时该字符会从输入流中消失。
getline(str,n,ch)		从文件流中接收 n-1 个字符给 str 变量，当遇到指定 ch 字符时会停止读取，默认情况下 ch 为 '\0'。
ignore(n,ch)		从文件流中逐个提取字符，但提取出的字符被忽略，不被使用，直至提取出 n 个字符，或者当前读取的字符为 ch。
peek()		返回文件流中的第一个字符，但并不是提取该字符。
putback(c)		将字符 c 置入文件流（缓冲区）。
operator<<	fstream ofstream	重载 << 运算符，用于向文件中写入指定数据。
put()		向指定文件流中写入单个字符。
write()		向指定文件中写入字符串。
tellp()		用于获取当前文件输出流指针的位置。
seekp()		设置输出文件输出流指针的位置。
flush()		刷新文件输出流缓冲区。
good()	fstream ofstream ifstream	操作成功，没有发生任何错误。
eof()		到达输入末尾或文件尾。

```
#include <iostream>
#include <fstream>//添加文件流头文件
using namespace std;
int main()
{
    const char* str = "http://www.163.com";
    // 我们要创建一个fstream类的对象
    fstream fs;
    // 将demo.txt文件和fs文件流建立关联
    fs.open("demo.txt", ios::out);
    fs.write(str, 100);
    fs.close();
    return 0;
}
```

打开文件可以通过以下两种方式进行：

调用流对象的 open 成员函数打开文件。

定义文件流对象时，通过构造函数打开文件。

【使用 open 函数打开文件】

先看第一种文件打开方式。以 ifstream 类为例，该类有一个 open 成员函数，其他两个文件流类也有同样的 open 成员函数：

void open(const char* szFileName, int mode)

第一个参数是指向文件名的指针，第二个参数是文件的打开模式标记。

模式标记	适用对象	作用
ios::in	ifstream fstream	打开文件用于读取数据。如果文件不存在，则打开出错。
ios::out	ofstream fstream	打开文件用于写入数据。如果文件不存在，则新建该文件；如果文件原来就存在，则打开时清除原来的内容。
ios::app	ofstream fstream	打开文件，用于在其尾部添加数据。如果文件不存在，则新建该文件。
ios::ate	ifstream	打开一个已有的文件，并将文件读指针指向文件末尾（读写指 的概念后面解释）。如果文件不存在，则打开出错。
ios:: trunc	ofstream	打开文件时会清空内部存储的所有数据，单独使用时与 ios::out 相同。
ios::binary	ifstream ofstream fstream	以二进制方式打开文件。若不指定此模式，则以文本模式打开。
ios::in | ios::out	fstream	打开已存在的文件，既可读取其内容，也可向其写入数据。文件刚打开时，原有内容保持不变。如果文件不存在，则打开出错。
ios::in | ios::out	ofstream	打开已存在的文件，可以向其写入数据。文件刚打开时，原有内容保持不变。如果文件不存在，则打开出错。
ios::in | ios::out | ios::trunc	fstream	打开文件，既可读取其内容，也可向其写入数据。如果文件本来就存在，则打开时清除原来的内容；如果文件不存在，则新建该文件。

ios::binary 可以和其他模式标记组合使用，例如：

ios::in | ios::binary表示用二进制模式，以读取的方式打开文件；

ios::out | ios::binary表示用二进制模式，以写入的方式打开文件。

在流对象上执行 open 成员函数，给出文件名和打开模式，就可以打开文件。判断文件打开是否成功，可以看"对象名"这个表达式的值是否为 true，如果为 true，则表示文件打开成功。

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    //ifstream inFile;
    //inFile.open(".\\demo.txt", ios::in);
    //if (inFile) // 条件成立，则说明文件打开成功
    //{
    //    cout << "\ndemo.txt文件打开成功." << endl;
    //    inFile.close();
    //}
    //else
    //{
    //    cout << "\ndemo.txt文件打开失败." << endl;
    //    return 1;
    //}
    //ofstream outFile;
    //outFile.open(".\\outdemo.txt", ios::out);
    //if (outFile) // 条件成立，则说明文件打开成功
    //{
    //    cout << "\noutdemo.txt文件打开成功." << endl;
```

```
    //    outFile.close();
    //}
    //else
    //{
    //    cout << "\noutdemo.txt文件打开失败." << endl;
    //}
    fstream ioFile;
    ioFile.open(".\\iodemo.txt", ios::in | ios::out | ios::trunc);

    if (ioFile)
    {
        cout << "\niodemo.txt文件打开成功." << endl;
        ioFile.close();
    }
    else
    {
        cout << "\niodemo.txt文件打开失败." << endl;
    }
    return 0;
}
```

调用open()方法打开文件，是文件流对象和文件之间建立关联的过程。那么，调用 close()方法关闭已打开的文件，就可以理解为是切断文件流对象和文件之间的关联。注意，close()方法的功能**仅是切断文件流与文件之间的关联**，该文件流并会被销毁，其后续还可用于关联其它的文件。

close()方法的用法很简单，其语法格式如下：

```
void close();
```

可以看到，该方法既不需要传递任何参数，也没有返回值。

1、【C++ ostream::write()方法写文件】

ofstream 和 fstream 的 write() 成员方法实际上继承自 ostream 类，其功能是将内存中 buffer 指向的 count 个字节的内容写入文件，基本格式如下：

```
ostream & write(char* buffer, int count);
```

其中，**buffer 用于指定要写入文件的二进制数据的起始位置；count 用于指定写入字节的个数。**也就是说，该方法可以被 ostream 类的 cout 对象调用，常用于向屏幕上输出字符串。同时，它还可以被 ofstream 或者 fstream 对象调用，用于将指定个数的二进制数据写入文件。

同时，该方法会返回一个作用于该函数的引用形式的对象。举个例子，obj.write() 方法的返回值就是对 obj 对象的引用。

```
#include <iostream>
#include <fstream>
using namespace std;
class student
{
public:
    int no;
```

```
    char name[10];
    int age;
};
int main()
{
    student stu;
    ofstream outFile("student.dat", ios::out | ios::binary);//outFile是类
名，"student.dat", ios::out | ios::binary是构造函数的传参
    if (!outFile)
    {
        cout << "\n打开文件student.txt失败." << endl;
        outFile.close();
    }
    else
    {
        cout << "\n打开文件student.txt成功." << endl;
    }
    while (cin >> stu.no >> stu.name >> stu.age)//循环写值
        outFile.write((char*)&stu, sizeof(stu));//循环以二进制格式写入文件
    outFile.close();
    return 0;
}
```

2、【C++ istream::read()方法读文件】

ifstream 和 fstream 的 read() 方法实际上继承自 istream 类，其功能正好和 write() 方法相反，即从文件中读取 count 个字节的数据。该方法的语法格式如下：

```
istream & read(char* buffer, int count);
```

其中，buffer 用于指定读取字节的起始位置，count 指定读取字节的个数。同样，该方法也会返回一个调用该方法的对象的引用。

```
#include <iostream>
#include <fstream>
using namespace std;
class student
{
public:
    int no;
    char name[10];
    int age;
};
int main()
{
    student stu;
    ifstream inFile("student.dat", ios::in | ios::binary); // 二进制读方式打开此文件
    if (!inFile)
    {
        cout << "\n打开失败." << endl;
        return 0;
    }
    else
        cout << "\nstudent.dat文件打开成功." << endl;
```

```
    while (inFile.read((char*)&stu, sizeof(stu)))// 二进制读方式读出此文件内容
    {
        cout << stu.no << "," << stu.name << "," << stu.age << endl;
    }
    inFile.close();
    return 0;
}
```

3、【C++ ostream::put()成员方法】

掌握了如何通过执行 cout.put() 方法向屏幕输出单个字符。fstream 和 ofstream 类继承自 ostream 类，因此 fstream 和 ofstream 类对象都可以调用 put() 方法。

当 fstream 和 ofstream 文件流对象调用 put() 方法时，该方法的功能就变成了向指定文件中写入单个字符。put() 方法的语法格式如下：

```
ostream& put (char c);
```

其中，c 用于指定要写入文件的字符。该方法会返回一个调用该方法的对象的引用形式。

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    char ch;
    ofstream outFile("outdemo.txt", ios::out | ios::binary);
    if (!outFile)
    {
        cout << "\noutdemo.txt文件打开失败." << endl;
        return 0;
    }
    else
        cout << "\noutdemo.txt文件打开成功." << endl;
    while (cin >> ch)
    {
        outFile.put(ch);
    }
    outFile.close();
    return 0;
}
```

4、【C++ istream::get()成员方法】

与put() 成员方法的功能相对的是 get() 方法，其定义在 istream 类中，借助 cin.get() 可以读取用户输入的字符。在此基础上，fstream 和 ifstream 类继承自 istream 类，因此 fstream 和 ifstream 类的对象也能调用 get() 方法。

当 fstream 和 ifstream 文件流对象调用 get() 方法时，其功能就变成了从指定文件中读取单个字符（还可以读取指定长度的字符串）。这里仅介绍最常用的 2 种：

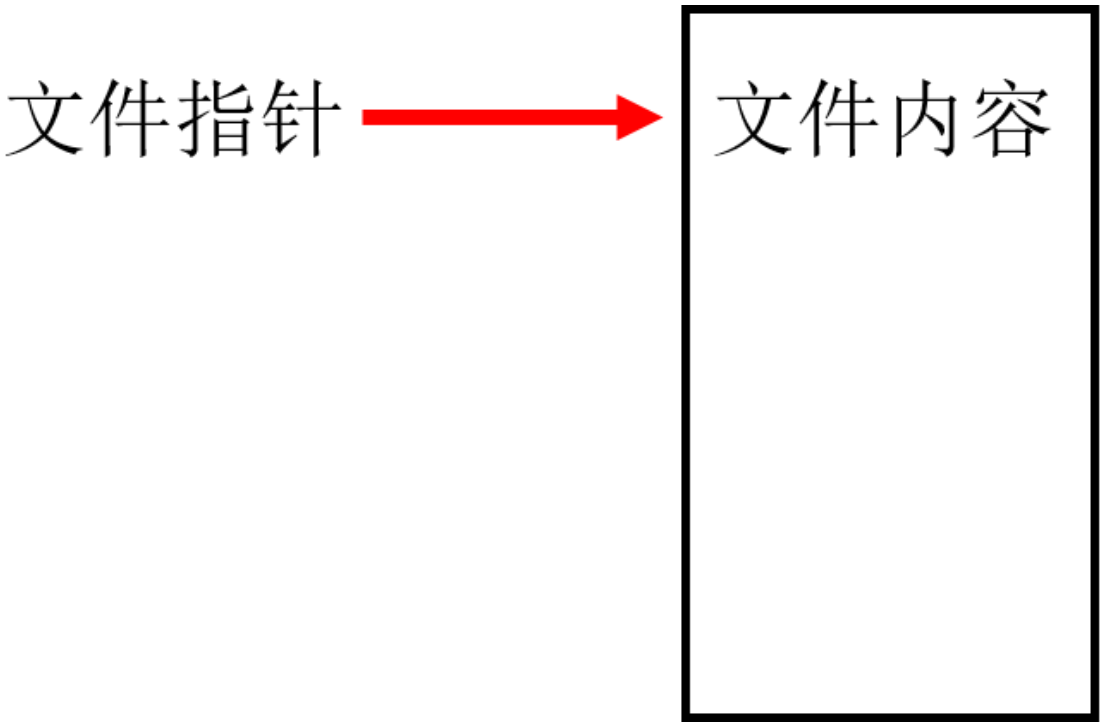
```
int get();

istream& get (char& c);
```

其中，第一种语法格式的返回值就是读取到的字符，只不过返回的是它的 ASCII 码，如果碰到输入的末尾，则返回值为 EOF。第二种语法格式需要传递一个字符变量，get() 方法会自行将读取到的字符赋值给这个变量。

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    char ch;
    ifstream inFile("outdemo.txt", ios::out | ios::binary);
    if (!inFile)
    {
        cout << "\noutdemo.txt文件打开失败." << endl;
        return 0;
    }
    else
        cout << "\noutdemo.txt文件打开成功." << endl;
    while ((ch=inFile.get())&& ch!=EOF)
    {
          cout << ch;
    }
    inFile.close();
    return 0;
}
```

5、【文件指针】

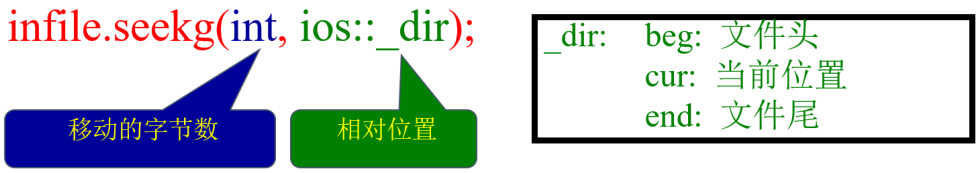
当一打开文件，文件指针位于文件头，并随着读写字节数的多少顺序移动。可以利用成员函数随机移动文件指针。



【随机读取二进制文件】

infile.seekg(int);//将文件指针移动到由参数指定的字节处

infile.seekg(100);//将文件指针移动到距离文件头100个字节处



infile.seekg(100, ios::beg);//移动到距文件头100个字节

infile.seekg(-100, ios::cur);//移动到距当前位置前100个字节

infile.seekg(-500, ios::end);//移动到距文件尾前500个字节

day8 STL（Standard Template Library）

STL是Standard Template Library的简称，中文名标准模板库，惠普实验室开发的一系列软件的统称。它是由Alexander Stepanov、Meng Lee和David R Musser在惠普实验室工作时所开发出来的。从根本上说，STL是一些“容器”的集合，这些“容器”有list,vector,set,map等，STL也是算法和其他一些组件的集合。这里的“容器”和算法的集合指的是世界上很多聪明人很多年的杰作。STL的目的是标准化组件，这样就不用重新开发，可以使用现成的组件。STL是C++的一部分，因此不用安装额外的库文件。

STL的版本很多，常见的有HP STL、PJ STL、 SGI STL等。

在C++标准中，STL被组织为下面的13个头文件：、、、、、、、、**<memory.h>**、、、、**和。**

一、STL(vector)

1、序列式容器vector：向量(vector) 连续存储的元素

vector 容器是 STL 中最常用的容器之一，它和 array 容器非常类似，都可以看做是对 C++ 普通数组的“升级版”。不同之处在于，array 实现的是静态数组（容量固定的数组），而 vector 实现的是一个动态数组，即可以进行元素的插入和删除，在此过程中，vector 会动态调整所占用的内存空间，整个过程无需人工干预。

vector 常被称为向量容器，因为该容器擅长在尾部插入或删除元素，在常量时间内就可以完成，时间复杂度为O(1)；而对于在容器头部或者中部插入或删除元素，则花费时间要长一些（移动元素需要耗费时间），时间复杂度为线性阶O(n)。

2、【创建vector容】

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    // 创建存储的double数据类型的一个vector容器-->v1
    vector<double> v1;
    // 创建容器v2且同时给它进行初始化元素和元素个数
    vector<int> v2{ 34,65,78,90,28,56,98,77,81,19 };
    // 创建vector容器，指定元素个数为50
```

```
    vector<int> v3(50);
    // 创建vector容器，它拥有10个字符为'A'
    vector<char> v4(10, 'A');
    // 将v4赋给v5
    vector<char> v5(v4);
    int a[] = { 10,20,30 };
    vector<int> v6(a, a + 2); // v6将会保存10，20，也就是保存a数组中的前两个元素
    vector<int> v7{ 1,2,3,4,5,6,7,8,9,10 };
    vector<int> v8(begin(v7), begin(v7) + 3); // v8将会保存为：1,2,3
    return 0;
}
```

3、【vector容器包含的成员函数】

【直接对应官方网站查询即可】
http://www.cplusplus.com/reference/vector/vector/

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    // 定义一个空的vector容器
    vector<char> vi;
    // 向容器添加S T L T E M P L A T E
    vi.push_back('S');
    vi.push_back('T');
    vi.push_back('L');
    vi.push_back('T');
    vi.push_back('E');
    vi.push_back('M');
    vi.push_back('P');
    vi.push_back('L');
    vi.push_back('A');
    vi.push_back('T');
    vi.push_back('E');
    // 输出容器vi元素的个数 size()
    cout << "元素个数为:" << vi.size() << endl;
    // 遍历容器
    cout << "\n输出vi容器数据元素:\n";
    for (auto i = vi.begin(); i < vi.end(); i++)
        cout << " " << *i << " ";
    cout << endl;
    //--------------------------------------------------------------------------
    // 插入数据元素到头部
    vi.insert(vi.begin(), 'V');
    // 遍历容器
    cout << "\n输出vi容器数据元素:\n";
    for (auto i = vi.begin(); i < vi.end(); i++)
        cout << " " << *i << " ";
    cout << endl;
    //--------------------------------------------------------------------------
    // 插入数据元素到尾部
    vi.insert(vi.end(), 'I');
    // 遍历容器
```

```
    cout << "\n输出vi容器数据元素:\n";
    for (auto i = vi.begin(); i < vi.end(); i++)
        cout << " " << *i << " ";
    cout << endl;
    cout << "\n输出vi容器首个元素为:" << vi.at(0) << endl << endl;
    return 0;
}
```

二、STL(deque)

1、deque 是 double-ended queue 的缩写，又称双端队列容器。

前面已接触过vector 容器，值得一提的是，deque 容器和 vecotr 容器有很多相似之处，比如：deque 容器也擅长在序列尾部添加或删除元素（时间复杂度为O(1)），而不擅长在序列中间添加或删除元素。deque 容器也可以根据需要修改自身的容量和大小。

和 vector 不同的是，deque 还擅长在序列头部添加或删除元素，所耗费的时间复杂度也为常数阶 O(1)。并且更重要的一点是，deque 容器中存储元素并不能保证所有元素都存储到连续的内存空间中。

当需要向序列两端频繁的添加或删除元素时，应首选 deque 容器。

2、【创建deque容器】

```
//#include <iostream>
//#include <deque>
//using namespace std;
//int main()
//{
//    // 创建deque容器，没有任何数据元素
//    deque<int> d1;
//    // 创建deque容器且有50个元素
//    deque<int> d2(50);
//    // 创建一个具有9个元素的deque容器，并且进行初始化值
//    deque<int> d3(9, 888);
//    // 容器之间可以赋值
//    deque<int> d4(10);
//    deque<int> d5(d4);
//    return 0;
//}
```

3、【deque容器可利用的成员函数】

【直接对应官方网站查询即可】

https://en.cppreference.com/w/cpp/container/deque

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

int main()
{
    // 定义空的容器
```

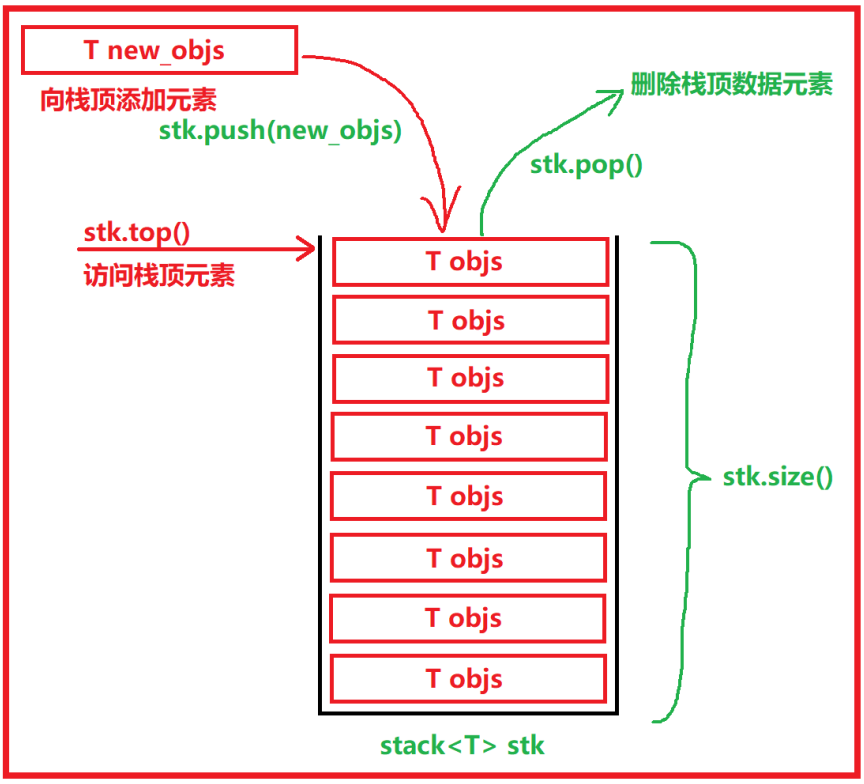
```
    deque<int> d1;
    // 向容器的尾部插入数字
    d1.push_back(10);
    d1.push_back(20);
    d1.push_back(30);
    d1.push_back(40);
    d1.push_back(50);
    d1.push_back(60);
    d1.push_back(70);
    d1.push_back(80);
    d1.push_back(90);
    d1.push_back(100);
    // 输出d1元素的个数
    cout << "输出d1容器元素的个数为:" << d1.size() << endl << endl;
    // 向d1容器头部添加数据元素值
    d1.push_back(888);
    // 输出容器所有元素值
    cout << "输出d1容器所有元素的值:" << endl;
    for (auto i = d1.begin(); i < d1.end(); i++)
        cout << " " << *i << " ";
    cout << endl << endl;
    // 删除容器头部的数据元素值
    d1.pop_front();
    // 输出容器所有元素值
    cout << "输出d1容器所有元素的值:" << endl;
    for (auto i = d1.begin(); i < d1.end(); i++)
        cout << " " << *i << " ";
    cout << endl << endl;
    return 0;
}
```

三、STL(stack)

1、stack容器适配器

容器适配器是一个封装了序列容器的类模板，它在一般序列容器的基础上提供了一些不同的功能。之所以称作适配器类，是因为它可以通过适配容器现有的接口来提供不同的功能。

stack容器适配器中的数据是以 LIFO 的方式组织的，这和自助餐馆中堆叠的盘子、箱子中的一堆书类似。理论上的 stack 容器及其一些基本操作。只能访问 stack 顶部的元素；只有在移除 stack 顶部的元素后，才能访问下方的元素。



2、【创建stack容器】

```
#include <iostream>
#include <stack>
#include <list>
using namespace std;
int main()
{
    // 创建一个stack容器适配器
    list<int> LS{ 11,22,33,44,55,66,77,88,99 };
    stack<int, list<int>> mystack(LS);
    // 查询mystack存储元素的个数
    cout << "\nmystack栈容器数据元素个数为:" << mystack.size() << endl;
    // 输出栈容器数据元素的值:
    cout << "\n\n输出栈容器数据元素的值:" << endl;
    while (!mystack.empty())  // 没有元素返回真true,否则返回false
    {
        cout << mystack.top() << " ";
        // 将栈顶元素弹出去
        mystack.pop();
    }
    cout << endl;
    return 0;
}
```

3、【stack容器可利用的成员函数】

【直接对应官方网站查询即可】

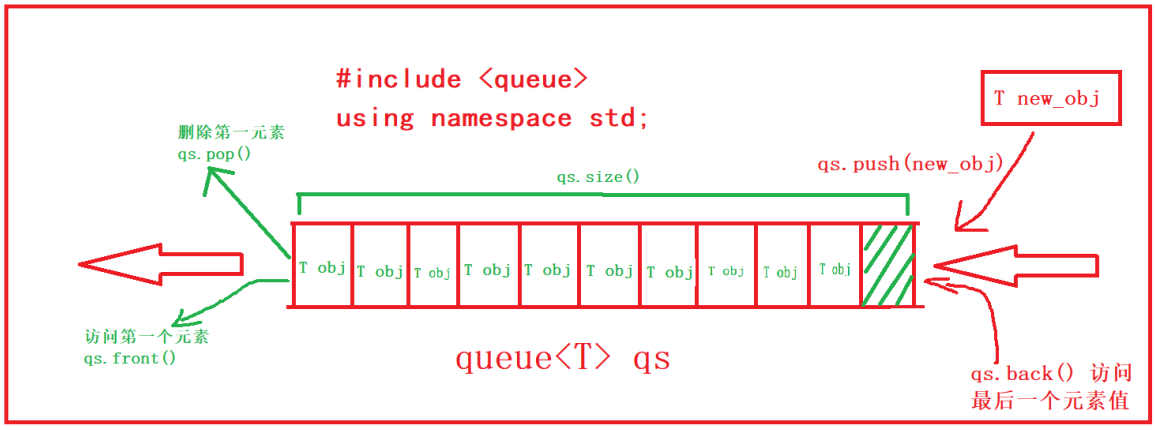
http://www.cplusplus.com/reference/stack/stack/

四、STL(queue)

1、【queue（double-ended queue，双端队列）】

队列也是一种逻辑数据结构，其具有先进先出的特性，只能在队的前端进行删除， 在队的后端进行插入。针对这种特性，可以实现一些较为复杂的逻辑。在实际应用中，部分程序也正需要这样一种顺序进出的数据处理方式。使用这样的逻辑处理方式，使得我们可以将更多精力放在如何处理顺序逻辑之外的事情，对于编程、开发来讲，提供了极大的方便。

只能访问 queue 容器适配器的第一个和最后一个元素。只能在容器的末尾添加新元素，只能从头部移除元素。许多程序都使用了 queue 容器。queue 容器可以用来表示商场结账队列或服务器上等待执行的数据库事务队列。对于任何需要用 FIFO 准则处理的序列来说，使用 queue 容器适配器都是好的选择。



2、【创建queue容器适配器】

```
#include <iostream>
#include <queue>
#include <list>
using namespace std;
int main()
{
    //// 创建一个使用list容器作为基础容器空的vq容器适配器
    //queue<int, list<int>> vq1;
    //deque<int> d1{ 1,2,3,4,5 };
    //queue<int> vq2(d1);
    //queue<int> vq3=vq2; // 等价 queue<int> vq3(vq2);
   // 构建一个queue容器适配器
    deque<int> d1{ 11, 22, 33, 44, 55 };
    queue<int> qd(d1); // 11, 22, 33, 44, 55
    // 求出qd存储元素的个数
    cout << "\n输出qd容器元素个数为:" << qd.size() << endl << endl;
    // 输出qd容器所有元素的值
    cout << "\n输出qd容器所有元素为:\n";
    while (!qd.empty())
    {
        cout << qd.front() <<"\t";
```

```
        // 访问过的元素出队列
        qd.pop();
    }
    cout << endl;
    // 求出qd存储元素的个数
    cout << "\n输出qd容器元素个数为:" << qd.size() << endl << endl;
    cout << endl;
    return 0;
}
```

3、【queue成员函数】

【直接对应官方网站查询即可】
http://www.cplusplus.com/reference/queue/queue/

五、STL(set)

STL 对这个序列可以进行查找、插入、删除序列中的任意一个元素，而完成这些操作的时间同这个序列中元素个数的对数成比例关系，并且当游标指向一个已删除的元素时，删除操作无效。而一个经过更正的和更加实际的定义应该是：一个集合(set)是一个容器，它其中所包含的元素的值是唯一的。这在收集一个数据的具体值的时候是有用的。集合中的元素按一定的顺序排列，并被作为集合中的实例。一个集合通过一个链表来组织，在插入操作和删除操作上比向量(vector)快，但查找或添加末尾的元素时会有些慢。具体实现采用了红黑树的平衡二叉树的数据结构。

1、【创建set容器】

```
#include <iostream>
#include <set>
#include <string>
using namespace std;
int main()
{
    // 创建一个set容器，为空的
    set<string> mys;
    cout << "\n\n测试容器mys长度:" << mys.size() << endl;
    cout << endl;
    mys.insert("www.baidu.com");
    mys.insert("www.163.com");
    mys.insert("www.126.com");
    mys.insert("www.0vice.com");
    cout << "\n输出mys容器所有元素:" << endl;
    for (auto it = mys.begin(); it != mys.end(); ++it)
    {
        cout << *it << endl;
    }
    cout << endl;
    return 0;
}
```

2、【set成员函数】

【直接对应官方网站查询即可】
http://www.cplusplus.com/reference/set/set/

六、STL(map)

映射和多重映射基于某一类型Key的键集的存在，提供对T类型的数据进行快速和高效的检索。对map而言，键只是指存储在容器中的某一成员。Map不支持副本键，multimap支持副本键。Map和multimap对象包涵了键和各个键有关的值，键和值的数据类型是不相同的，这与set不同。set中的key和value是Key类型的，而map中的key和value是一个pair结构中的两个分量。

```
// CLASS TEMPLATE map
template <
   class _Kty, // 指定键(key)的类型
   class _Ty, // 指定值(value)的类型
   class _Pr = less<_Kty>, // 指定排序规则
   class _Alloc = allocator<pair<const _Kty, _Ty>> // 指定分配器
对象的类型
>class map;
```

1、【创建map容器】

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main()
{
    // 创建一个空mm容器
    map<string, string, greater<string>> mm;
    // 直接向mm容器指定位置构造新键值对
    mm.emplace("C语言程序设计", "http://www.baidu.com");
    mm.emplace("C++语言程序设计", "http://www.google.com");
    cout << "\n输出mm容器存储键值对的个数为:" << mm.size() << endl << endl;
    if (!mm.empty())
    {
        for (auto iter = mm.begin(); iter != mm.end(); iter++)
        {
            cout << iter->first << "\t" << iter->second << endl;
        }
    }
    cout << endl;
    return 0;
}
```

2、【map成员函数】

【直接对应官方网站查询即可】
http://www.cplusplus.com/reference/map/map/

c++异常处理

程序中常见的错误有两大类：语法错误和运行错误。在编译时，编译系统能发现程序中的语法错误。

在设计程序时，应当事先分析程序运行时可能出现的各种意外的情况，制订出相应的处理措施，这就是程序的异常处理任务。

在运行没有异常处理的程序时，如果运行情况出现异常，由于程序本身不能处理，程序只能终止运行。如果在程序中设置了异常处理机制，则在运行情况出现异常时，由于程序本身已规定了处理的方法，于是程序的流程就转到异常处理代码段处理。

异常(exception)是运行时(run-time)的错误，通常是非正常条件下引起的，例如，下标(index)越界、new操作不能正常分配所需内存。C语言中，异常通常是通过被调用函数返回一个数值作为标记的。

C++中，函数可以识别标记为异常的条件，然后通告发生了异常。这种通告异常的机制称为**抛出异常(throwing an exception)**。

异常提供了一种转移程序控制权的方式。C++ 异常处理涉及到三个关键字：try、catch、throw。

throw：当问题出现时，程序会抛出一个异常。这是通过使用 throw 关键字来完成的。

catch：在您想要处理问题的地方，通过异常处理程序捕获异常。catch 关键字用于捕获异常。

try：try 块中的代码标识将被激活的特定异常。它后面通常跟着一个或多个 catch 块。

如果有一个块抛出一个异常，捕获异常的方法会使用 try 和 catch 关键字。try 块中放置可能抛出异常的代码，try 块中的代码被称为保护代码。

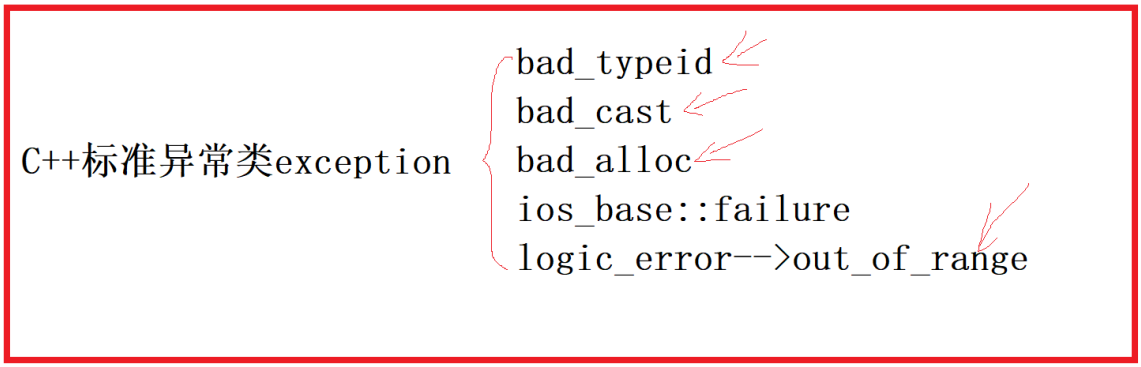
C++语言通过throw语句和try...catch语句实现对异常处理，throw语句的语法格式如下：

throw 表达式;

其实此语句抛出一个异常，异常是一个表达式，其值的类型可以是基本类型，也可以是类。

try..catch语句的语法格式如下：

try{语句组}
catch(异常类型)
{异常处理代码}
....
catch(异常类型)
{异常处理代码}



【C++异常处理基本语法】

```
/*  从键盘输入两个数，实现相除。  */
#include <iostream>
using namespace std;
int main()
{
    double x, y;
    cout << "请输入x,y的值:";
    cin >> x >> y;
    try {
        if (y == 0)
            throw - 1; // 抛出-1类型异常
        else if(x==0)
            throw - 1.0; // 抛出-1类型异常
        else
            cout << "x/y=" << x / y << endl << endl;
    }
    catch (int e)
    {
        cout << "catch(int) :" << e << endl;
    }
    catch (double d)
    {
        cout << "catch(double) :" << d << endl;
    }
    return 0;
```

```
}
```

c++语言的新特性

一、类型推导（auto/decltype）

auto 变量名 = 值;

decltype(表达式) 变量名[=值];

```
#include <iostream>
using namespace std;
int main()
{
    auto var1 = 250;//必须要初始化
    decltype(199.88) var2;
    cout << "var1占据字节个数:" << sizeof(var1) << endl;
    cout << "var2占据字节个数:" << sizeof(var2) << endl;
    return 0;
}
```

二、序列for循环

可以遍历数组、容器、string以及由begin/end函数定义的序列

```
#include <iostream>
using namespace std;
int main()
{
    for (const auto var : { 54,67,90,12,53,88,77,123,54,20 })
        cout << var << " ";
    cout << endl<<endl;
    for (const auto str : { "ABC","DEF","GHI","JKL" })
        cout << str << " ";
    cout << endl;
    return 0;
}
```

三、lamdba表达式

c++ 11 中的lambda表达式用于定义并创建匿名的函数对象。我们如何声明lambda表达式完整格式语法如下:

```
[capyure list](params list) mutable exception->return typr {function body}
```

```
#include <iostream>
#include <vector>
```

```cpp
#include <algorithm>
using namespace std;
bool comparefunc(int x, int y)
{
    return x < y;
}
int main()
{
    vector<int> myvr{ 93,78,133,21,89 };
    vector<int> lvec{8,3,5,2,1};
    // 平时常用的方法如下
    sort(myvr.begin(), myvr.end(), comparefunc);
    cout << "常用输出结果为：" << endl;
    for (int it : myvr)
        cout << it << ' ';
    cout << endl;
    // 下面我们使用lambda表达式
    sort(lvec.begin(), lvec.end(), [](int a, int b)->bool {return a < b; });
    cout << "lambda表达式输出结果为：" << endl;
    for (int il : lvec)
        cout << il << ' ';
    cout << endl;
    return 0;
}
```

四、构造函数：委托构造和继承构造

委托构造函数：它可以使用当前的类的其他构造函数来协助当前构造函数的初始化操作。

普通构造函数和委托构造函数区别：

- 他们两都是一个成员初始值列表与一函数体；
- 委托构造函数的成员初始值列表只有一个唯一的参数，就是构造函数。

当被委托构造函数当中函数体有代码，那么会先执行完函数体的代码，才会回来到委托构造函数。

继承构造函数：在c++语言当中的继承关系，只有虚函数可以被继承，二狗找函数不可以是虚函数，所以狗账号是不能被继承，但是我们可以通过某种手段，达到继承效果。

```cpp
#include <iostream>
#include <string>
using namespace std;
// 创建一个类
class TestC
{
public:
    // 普通构造函数
    //:_data(d), _str(s) 是初始化列表，它用于初始化类的成员变量。
    /*
    初始化列表的好处是：
    它在构造函数体执行之前就初始化了成员变量，通常更高效，尤其是对于复杂类型（比如对象）。
    对于常量成员和引用成员，必须使用初始化列表来初始化它们，因为它们在构造函数体内无法被赋值。
    */
    TestC(string s, int d) :_data(d), _str(s)
```

```
    {
        cout << "程序执行：普通构造函数的函数体" << endl;
    }
    // 委托构造函数
    TestC(int d) :TestC("测试TestC(int d)", d)
    {
        cout << "程序执行：委托构造函数TestC(int d) 的函数体" << endl;
    }
    void printdata()const
    {
        cout << "--------------------------------------" << endl;
        cout << "_str=" << _str << endl;
        cout << "_data=" << _data << endl;
        cout << "--------------------------------------" << endl << endl;
    }
private:
    int _data;
    string _str;
};

int main()
{
    TestC objc1("测试普通构造函数:TestC(string s, int d) ", 250);
    objc1.printdata();
    TestC objc2(890);
    objc2.printdata();
    // 委托构造函数字符串_str赋值操作，这个大家自己完成。
    return 0;
}
```

```
#include <iostream>
using namespace std;
struct A
{
    void func(double d) {
        cout << "基类A:" << d << endl;
    }
};
struct B :A
{
    using A::func;//c++ 11 标准当中利用using关键字的特点，是构造函数可以被继承
    void func(int i) {
        cout << "派生类B:" << i << endl;
    }
};
int main()
{
    A a;
    a.func(78);
    B b;
    b.func(87);
    return 0;
}
```

五、容器（array&forward_list)

C++ array (STL array)容器，array容器时c++11标准中新增的序列容器。

```
#include <iostream>
using namespace std;
//要使用array容器，就需要加上头文件
#include <array>
int main()
{
    array<int, 10> data{};
    //初始化data容器
    for (int i = 0; i < data.size(); i++) {
        data.at(i) = i;
    }
    //通过get()重载函数输出指定位置元素值
    cout << "输出结果为：" << get<6>(data) << endl << endl;
    return 0;
}
```

C++forward_list是C++11标准新增的一类容器，其底层实现和list容器类似，采用的也是链表结构，只是forward_list使用的是单向链表，而list使用的是双向链表。

```
#include <iostream>
#include <forward_list>
using namespace std;
int main()
{
    std::forward_list<int> values{ 10,20,30 };
    values.emplace_front(40);
    cout << "输出结果为1：" << endl;
    for (auto it = values.begin(); it != values.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
    values.emplace_after(values.before_begin(), 50);
    cout << "输出结果为2：" << endl;
    for (auto it = values.begin(); it != values.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
    values.reverse();
    cout << "输出结果为3：" << endl;
    for (auto it = values.begin(); it != values.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}
```

六、垃圾回收机制

一、标准c++没有垃圾回收机制原因：系统开销、耗内存、有替代方法、没有共同基类。

二、c/c++中经典垃圾回收算法：引用计数算法（有时间和空间上的开销）、标记-清除算法、节点拷贝算法。

七、正则表达式

正则表达式，又称规则表达式（英文：Regular Expression，在代码中常简写为regex、regexp、或RE），计算机科学的一个概念。正则表达式通常被用来检索、替换那些符合某个模式（规则）的文本。

许多程序设计语言都支持利用正则表达式进行字符串操作，例如，在Perl中就内奸了一个功能强大的正则表达式引擎。正则表达式这个概念最初是由Unix中的工具软件（例如sed和grep）普及开的。正则表达式通常缩写成“regex”，单数有regexp、regex，复数有regexps，regexes、regexen。

一、符号

正则表达式由一些普通字符和一些元字符（metacharacters）组成。普通字符包括大小写的字母和数字，而元字符则具有特殊的含义，我们下面会给与解释。

在最简单的情况下，一个正则表达式看上去就是一个普通的查找串，例如，正则表达式“testing”中没有包含任何元字符，它可以匹配“testing”和“testing123”等字符串，但是不能匹配“Testing”。

想要真正的用好正则表达式，正确的理解元字符是最重要的事情。下表列出了所有的元字符和对它们的一个简短的描述。

元字符	**描述(太多了，网上找一找)**
\	将下一个字符标记符、或一个向后引用、或一个八进制转义符。例如，”\\n“匹配\n。”\n“匹配换行符。序列”\\“匹配”\“而”\(“则匹配”(“。即相当于多种编程语言中都有的”转义字符“的概念。
^	匹配输入字行首。如果设置了RegExp对象的Multiline属性，^也匹配"\n"或”\r“之后的位置。
$	匹配输入行尾。如果设置了RegExp对象的Multiline属性，$也匹配"\n"或”\r“之前的位置。
*	匹配前面的子表达式任意次，例如，zo*能匹配”z“，也能匹配”zo“以及”zoo“。等价于{0，}。

二、速记理解技巧

正则表达式（Regex）是一种强大的文本处理工具，掌握一些基本的速记方法可以帮助您更有效地使用它。以下是一些有用的速记技巧和常见模式：

1. **基本符号**

- `.`：匹配任意单字符（除换行符）。
- `^`：匹配行的开头。
- `$`：匹配行的结尾。
- `*`：匹配零个或多个前面的字符。
- `+`：匹配一个或多个前面的字符。
- `?`：匹配零个或一个前面的字符。

- `{n}`：匹配前面的字符恰好出现 n 次。
- `{n,}`：匹配前面的字符至少出现 n 次。
- `{n,m}`：匹配前面的字符至少 n 次，至多 m 次。

2. **字符类**

- `[abc]`：匹配 a、b 或 c 中的任意一个字符。
- `[^abc]`：匹配除了 a、b、c 以外的任意字符。
- `[a-z]`：匹配 a 到 z 的任何字符。
- `[0-9]`：匹配任意数字。

3. **预定义字符类**

- `\d`：匹配任何数字（等价于 `[0-9]`）。
- `\D`：匹配任何非数字字符。
- `\w`：匹配任何字母数字字符（等价于 `[a-zA-Z0-9_]`）。
- `\W`：匹配任何非字母数字字符。
- `\s`：匹配任何空白字符（空格、制表符等）。
- `\S`：匹配任何非空白字符。

4. **分组和捕获**

- `()`：用于分组，并捕获匹配的内容。
- `(?:...)`：用于分组，但不捕获内容。
- `(?<name>...)`：命名捕获组。

5. **逻辑运算**

- `|`：表示“或”的关系，匹配左边或右边的表达式。例如 `a|b` 匹配 a 或 b。

6. **量词**

- `*?`、`+?`、`??`：非贪婪匹配（尽量少匹配）。

7. **边界**

- `\b`：单词边界。
- `\B`：非单词边界。

8. **常见模式**

- **匹配邮箱**：

```
[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}
```

- **匹配 URL**：

```
https?://[^\s/$.?#].[^\s]*
```

- **匹配日期（YYYY-MM-DD）**：

```
\d{4}-\d{2}-\d{2}
```

9. **速记技巧**

- **联想记忆**：将常用的正则表达式与生活中的例子联想在一起。

- **练习使用**：使用在线工具（如 regex101）进行练习和测试。
- **创建模板**：总结和记录一些常用的正则表达式模板，以便快速参考。

通过不断练习和使用这些技巧，您将能够更熟练地编写和理解正则表达式。希望这些信息对您有所帮助！

```
#include <iostream>
#include <string>
#include <regex>//正则表达式头文件
int main()
{
    std::regex string_reg("[1-9](\\d{5,11})");
    std::string strtest("1032");
    std::smatch matchresults;
    // 正则匹配
    if (std::regex_match(strtest, matchresults, string_reg))
    {
        std::cout << "Match:" << std::endl;
        // 输出表达式结果
        for (size_t i = 0; i < matchresults.size(); i++)
        {
            std::cout << matchresults[i] << std::endl;
        }
    }
    else
    {
        std::cout << "Not Match:" << std::endl;
    }
    return 0;
}
```

八、智能指针（shared_ptr&unique_ptr&weak_ptr）

头文件

unique_ptr智能指针：独占指针（内存占位己有，不支持赋值和拷贝）

```
#include <iostream>
#include <memory>
using namespace std;
int main()
{
    std::unique_ptr<int> p1(new int(24));
    cout << "*p1=" <<* p1 << endl << endl;
    std::unique_ptr<int> p2 = std::move(p1);
    cout << "*p2=" << *p2 << endl << endl;
    p2.reset(); // 显式释放内存
    p1.reset();
    std::unique_ptr<int>p3(new int(250));
    p3.reset(new int(666)); // 绑定动态对象
    cout << "*p3=" << *p3 << endl << endl;
```

```
    p3 = nullptr; // 显式销毁指向对象，同时智能指针变为空，p3.reset()
    std::unique_ptr<int> p4(new int(999));
    int* p = p4.release(); // 只是释放控制权，不会释放内存
    cout << "*p=" << *p << endl << endl;
    cout << "*p4=" <<*p4 << endl;//*p4没有数据
    delete p; // 释放堆区里的资源数据
    return 0;
}
```

shared_ptr智能指针(共享拥有同一堆分配对象的内存，支持复制与赋值操作)。

```
#include <iostream>
#include <memory>
using namespace std;
int main() {
    shared_ptr<int> p1(new int(456));
    shared_ptr<int> p2 = p1;
    cout << "p2 =" << p2.use_count() << endl;//引用次数
    cout << "*p1 =" << *p1 << endl;
    cout << "*p2 =" << *p2 << endl;
    cout << "p2 =" << p2.use_count() << endl;
    p1.reset();
    cout << "p2 =" << p2.use_count() << endl;
    return 0;
}
```

Make_shared函数：最安全的分配和使用动态内存的方法。

weak_ptr智能指针：配合shared_ptr而引入的一种智能指针协助shared_ptr工作，它的构造和析构不会引起引用计数的增加或减少。weak_ptr并不拥有资源的所有权，所以不能直接使用资源，可以从一个weak_ptr构造一个shared_ptr以取得共享资源的所有权。

```
#include <iostream>
#include <memory>
using namespace std;
int main() {
    shared_ptr<int> p1(new int(100));
    shared_ptr<int> p2 = p1;
    weak_ptr<int> wp = p1;
    cout << "count = " << wp.use_count() << endl;
    cout << "*p1 = " << *p1 << endl;
    cout << "*p2 = " << *p2 << endl;
    return 0;
}
```

九、关键字：nullptr&constexpr

nullptr关键字，它的出现为代替NULL。

```
#include <iostream>
using namespace std;
void Func(char* pc) {
    printf("\n调用函数1\n");
}
void Func(int num) {
    printf("\n调用函数2\n");
}
int main()
{
    Func(0);
    Func(nullptr);
    return 0;
}
```

constexpr关键字， 用于表示在编译时就能确定的常量值。

```
#include <iostream>
using namespace std;
int main()
{
    int x = 400;
    int y = 2000;
    cout << "x=" << x << endl;
    //constexpr int* p = &x;//错误的
    int* p = &x;
    *p = 129;
    cout << "x=" << x << endl;
    // p = &y; 错误
    return 0;
}
```

十、共享内存

服务端程序：1、创建共享内存区域部分 2、内存映射到当前应用程序进程 3、写入数据信息

客户端程序：1、打开共享内存区域部分 2、内存映射到当前应用程序进程 3、读出数据信息

```
#include <iostream>
#include <Windows.h>
using namespace std;
#define BUF_SIZE 1024
int main()
{
    // 共享数据信息
    char szBuffer[] = "C/C++企业级项目实战课程";
```

```
    // 创建共享文件句柄
    HANDLE hmFile = CreateFileMapping(INVALID_HANDLE_VALUE, NULL,
        PAGE_READWRITE, 0, BUF_SIZE, L"LSEDU");
    // 映射
    LPVOID lpBase = MapViewOfFile(hmFile, FILE_MAP_ALL_ACCESS, 0, 0, BUF_SIZE);
    // 将数据拷贝到共享内存
    strcpy((char*)lpBase, szBuffer);
    cout << "\n服务器程序端:" << (char*)lpBase << endl << endl;
    // 线程挂起等待其他线程读取数据
    Sleep(20000);
    // 删除文件映射
    UnmapViewOfFile(lpBase);
    // 关闭内存映射文件对象句柄
    CloseHandle(hmFile);
    return 0;
}
```

```
#include <iostream>
#include <Windows.h>
using namespace std;
#define BUF_SIZE 1024
int main()
{
    HANDLE hmfile = OpenFileMapping(FILE_MAP_ALL_ACCESS, NULL, L"LSEDU");

    if (hmfile) {
        LPVOID lpBase = MapViewOfFile(hmfile, FILE_MAP_ALL_ACCESS, 0, 0, 0);
        char szBuffer[BUF_SIZE] = { 0 };
        strcpy(szBuffer, (char*)lpBase);
        cout << "\n客户端程序端: " << szBuffer;
        cout<<endl;
        UnmapViewOfFile(lpBase);
        CloseHandle(hmfile);
    }
    else {
        printf("\n打开共享内存失败，请检查操作\n\n");
    }
    return 0;
}
```

十一、C++11 std_unordered_set

std::undered_set 的数据存储结构也是采用哈希表的方式结构操作，除次之外，在插入时不会自动排序。

```
#include <iostream>
#include <string>
#include <set>
#include <unordered_set>
int main()
```

```
{
    std::unordered_set<int> un_set;
    un_set.insert(23);
    un_set.insert(33);
    un_set.insert(12);
    un_set.insert(78);
    un_set.insert(99);
    std::cout << "\nunordered_set:" << std::endl;
    for (auto it : un_set)
    {
        std::cout << it << std::endl;
    }
    std::cout << std::endl;
    std::set<int> st;
    st.insert(23);
    st.insert(33);
    st.insert(12);
    st.insert(78);
    st.insert(99);
    std::cout << "\nset:" << std::endl;
    for (auto it : st)
    {
        std::cout << it << std::endl;
    }
    std::cout << std::endl;
    return 0;
}
```

十二、关联容器：unordered_map

unordered_map为一个关联容器，内部采用hash表结构，具备快速检索的功能。特性：关联性、无序性、map、唯一性。

```
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;
typedef unordered_map<string, string> strmap;
strmap merge(strmap str1, strmap str2)
{
    strmap temp(str1);
    temp.insert(str2.begin(), str2.end());
    return temp;
}
int main()
{
    strmap s1;
    strmap s2({ {"apple","red"},{"lemon","yellow"} }); // 使用数组初始化
    strmap s3({ {"orange","orange"},{"strawberry","red"} }); // 使用数组初始化
    strmap s4(s2); // 复制初始化
    strmap s5(merge(s3, s4)); // 移动初始化操作
    strmap s6(s5.begin(), s5.end()); // 范围初始化操作
    cout << "\n输出s6容器:\n";
```

```
    for (auto& x : s6)
        cout << " " << x.first << ":" << x.second;
    cout << endl;
    return 0;
}
```

十三、function函数对象

c++函数对象：函数对象指定义operator()的对象，语法形式如下：

class functionObjectType{

public:

Void operator()(){

  操作语句；

}

};

优势：比普通函数要灵活(可以拥有状态)，执行速度比函数指针要快。

我们以函数对象作为排序原则操作，具体如下：

```
#include <iostream>
#include <set>
using namespace std;
class TestA
{
public:
    TestA(string lname, string fname):_fname(fname),_lname(lname)
    {
        cout << "执行TestA类的构造函数\n";
    }
    string firstname()const
    {
        return _fname;
    }
    string lastname()const
    {
        return _lname;
    }
private:
    string _fname = nullptr;
    string _lname = nullptr;
};
class TestB
{
public:
    bool operator() (const TestA& t1, const TestA& t2)const
    {
        return t1.lastname() < t2.lastname() || t1.lastname() == t2.lastname()
&&
            t1.firstname() < t2.firstname();
```

```
    }
};
int main()
{
    set<TestA, TestB> sett;
    TestA t1("liu", "san");
    TestA t2("xiao", "ming");
    TestA t3("zhang", "san");
    TestA t4("wang", "xiao");
    sett.insert(t1);
    sett.insert(t2);
    sett.insert(t3);
    sett.insert(t4);
    for (auto i : sett)
    {
        cout << i.lastname() << "," << i.firstname() << endl;
    }
    cout << endl;
    return 0;
}
```

十四、atomic_flag应用

std::atomic_flag为原子布尔类型。它不同于所有std::atomic的特例化,他保证是免锁。
std::atomic,std::atomic_flag不提供加载或存储操作。

```
#include <iostream>
#include <atomic>
#include <vector>
#include <thread>
// ATOMIC_FLAG_INIT-->定义能以语句用于初始化操作，清除状态的初始化器
std::atomic_flag lock = ATOMIC_FLAG_INIT; //
void FuncAt(int args)
{
    for (int i = 0; i < 10; i++)
    {
        while (lock.test_and_set(std::memory_order_acquire));// 获得锁
        std::cout << "Output Threads:" << i << std::endl;
        lock.clear(std::memory_order_release); // 释放锁
    }
}
int main()
{
    std::vector<std::thread> vct;
    for (int i = 0; i < 2; i++)
    {
        vct.emplace_back(FuncAt, i);
    }
    for (auto& t : vct)
    {
        t.join();
    }
    return 0;
```

```
}
```

十五、条件变量：condition_variable

c++标准当中，使用条件变量condition_variable实现多线程间的同步操作；当条件不满足时，相关线程被一直阻塞，直到某种条件出现，这些线程才会被唤醒。

```
#include <iostream>
#include <thread>
#include <condition_variable>
#include <mutex>
std::mutex mx;
std::condition_variable scv;
bool ready = false;
void PrintID(int id)
{
    std::unique_lock <std::mutex> lock(mx);
    while (!ready)
    {
        scv.wait(lock); // 当前线程被阻塞，当全局标志位变为true之后，才唤醒
    }
    std::cout << "Threads : " << id << std::endl;
}
void RunFunc()
{
    std::unique_lock <std::mutex> lock(mx);
    ready = true; // 设置全局标志位为true
    scv.notify_all(); // 唤醒所有线程
}
int main()
{
    std::thread thrs[5];
    for (int i = 0; i < 5; i++)
        thrs[i] = std::thread(PrintID, i);
    std::cout << "5 threads ready to race......\n";
    RunFunc();
    for (auto& t : thrs)
        t.join();
    return 0;
}
```

十六、异常处理：exception

c++语言标准库当中有一些类代表异常，这些类都是从exception类派生出来的，比如：bad_typeid/bad_cast/bad_alloc/ios_base::failure等，他们都是exception类的派生类。

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

```
{
    virtual void Func() {
    }
};

class TestB :public TestA
{
public:
    void disp() {
        cout << "TestB OK." << endl;
    }
};
void dispObject(TestA& t)
{
    try
    {
        TestB& tb = dynamic_cast<TestB&>(t);
        // 在此转换若不安全，会抛出bad_cast异常
        tb.disp();
    }
    catch (bad_cast& e)
    {
        cerr << e.what() << endl;
    }
}
int main()
{
    TestA obja;
    dispObject(obja);
    return 0;
}
```

十七、std_thread多线程

```
#include <iostream>
#include <thread>
using namespace std;
void ThreadFunc1()
{
    cout << "ThreadFunc1()--A\n" << endl;
    this_thread::sleep_for(std::chrono::seconds(2));
    cout << "ThreadFunc1()--B" << endl;
}
void ThreadFunc2(int args,string sp)
{
    cout << "ThreadFunc2()--A" << endl;
    this_thread::sleep_for(std::chrono::seconds(7));
    cout << "ThreadFunc2()--B" << endl;
}
int main()
{
```

```
    thread ths1(ThreadFunc1);
    thread ths2(ThreadFunc2,10,"LS");
    ths1.join();
    cout << "join" << endl;
    ths2.detach();
    cout << "detach" << endl;
    return 0;
}
```