一、面向对象程序设计
面向对象程序设计（Object-Oriented Programming，OOP）是一种新的程序设计范型。程序设计范型是指设计程序的规范、模型和风格，它是一类程序设计语言的基础。

面向过程程序设计范型是使用较广泛的面向过程性语言，其主要特征是：程序由过程定义和过程调用组成(简单地说,过程就是程序执行某项操作的一段代码,函数就是最常用的过程)。

面向对象程序的基本元素是对象，面向对象程序的主要结构特点是：第一，程序一般由类的定义和类的使用两部分组成；第二，程序中的一切操作都是通过向对象发送消息来实现的，对象接收到消息后，启动有关方法完成相应的操作。

对象：描述其属性的数据以及对这些数据施加的一组操作封装在一起构成的统一体。对象可认为是数据+操作。

类：类是具有相同的数据和相同的操作的一组对象的集合。

消息传递：对象之间的交互。

方法：对象实现的行为称为方法。

面向对象程序设计的基本特征：抽象、封装、继承、多态。

二、C++基础
~

2.1 C++的产生和特点
C++是美国贝尔实验室的 Bjarne Stroustrup 博士在 C 语言的基础上，弥补了 C 语言存在的一些缺陷，增加了面向对象的特征，于 1980 年开发出来的一种面向过程性与面向对象性相结合的程序设计语言。最初他把这种新的语言称为“含类的 C”，到 1983 年才取名为 C++。

相比 C 语言，C++的主要特点是增加了面向对象机制。

~

2.2 一个简单的 C++示例程序
详细创建步骤可参考博客【Visual Studio】 创建 C/C++项目

```
#include <iostream>     //编译预处理命令
using namespace std;      //使用命名空间

int add(int a, int b);           //函数原型说明
```

```
int main()  //主函数
{
    int x, y;
    cout << "Enter two numbers: " << endl;
    cin >> x;
    cin >> y;
    int sum = add(x, y);
    cout << "The sum is : " << sum << '\n';
    return 0;
}

int add(int a, int b) //定义 add()函数，函数值为整型
{
    return a + b;
}
```

2.3 C++在非面向对象方面对 C 语言的扩充

输入和输出

```
int i;
float f;
cin >> i;
cout << f;
------------
scanf("%d", &i);
printf("%f", f);
----------------
```

连续读入

```
cin >> a >> b >> c;
cin
```

在默认情况下，运算符“>>”将跳过空白符，然后读入后面与变量类型相对应的值。因此，给一组变量输入值时可用空格符、回车符、制表符将输入的数据间隔开。

当输入字符串（即类型为 string 的变量）时，提取运算符“>>”的作用是跳过空白字符，读入后面的非空白字符，直到遇到另一个空白字符为止，并在串尾放一个字符串结束标志‘\0’。

~

C++允许在代码块中的任何地方声明局部变量。

const 修饰符

在 C 语言中，习惯使用#define 来定义常量，例如#define PI 3.14，C++提供了一种更灵活、更安全的方式来定义常量，即使用 const 修饰符来定义常量。例如 const float PI = 3.14;

const 可以与指针一起使用，它们的组合情况复杂，可归纳为 3 种：指向常量的指针、常指针和指向常量的常指针。

指向常量的指针：一个指向常量的指针变量。

const char* pc = "abcd";
该方法不允许改变指针所指的变量，即
pc[3] = 'x'; 是错误的，
但是，由于 pc 是一个指向常量的普通指针变量，不是常指针，因此可以改变 pc 所指的地址，例如
pc = "ervfs";
该语句付给了指针另一个字符串的地址，改变了 pc 的值。
常指针：将指针变量所指的地址声明为常量

char* const pc = "abcd";
创建一个常指针，一个不能移动的固定指针，可更改内容，如
pc[3] = 'x';
但不能改变地址，如
pc = 'dsff'; 不合法
指向常量的常指针：这个指针所指的地址不能改变，它所指向的地址中的内容也不能改变。

const char* const pc = "abcd";
内容和地址均不能改变
说明：

如果用 const 定义整型常量，关键字可以省略。即 const in bufsize = 100 与 const bufsize = 100 等价；

常量一旦被建立，在程序的任何地方都不能再更改。

与#define 不同，const 定义的常量可以有自己的数据类型。

函数参数也可以用 const 说明，用于保证实参在该函数内不被改动。

void 型指针
void 通常表示无值，但将 void 作为指针的类型时，它却表示不确定的类型。这种 void 型指针是一种通用型指针，也就是说任何类型的指针值都可以赋给 void 类型的指针变量。

需要指出的是，这里说 void 型指针是通用指针，是指它可以接受任何类型的指针的赋值，但对已获值的 void 型指针，对它进行再处理，如输出或者传递指针值时，则必须再进行显式类型转换，否则会出错。

```
void* pc;
int i = 123;
char c = 'a';
pc = &i;
cout << pc << endl;              //输出指针地址 006FF730
```

```
    cout << *(int*)pc << endl;   //输出值 123
    pc = &c;
    cout << *(char*)pc << endl; //输出值 a
```

内联函数

在函数名前冠以关键字 inline，该函数就被声明为内联函数。每当程序中出现对该函数的调用时，C++编译器使用函数体中的代码插入到调用该函数的语句之处，同时使用实参代替形参，以便在程序运行时不再进行函数调用。引入内联函数主要是为了消除调用函数时的系统开销，以提高运行速度。

说明：

内联函数在第一次被调用之前必须进行完整的定义，否则编译器将无法知道应该插入什么代码

在内联函数体内一般不能含有复杂的控制语句，如 for 语句和 switch 语句等

使用内联函数是一种空间换时间的措施，若内联函数较长，较复杂且调用较为频繁时不建议使用

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

inline double circle(double r)   //内联函数
{
    double PI = 3.14;
    return PI * r * r;
}

int main()
{
    for (int i = 1; i <= 3; i++)
        cout << "r = " << i << " area = " << circle(i) << endl;
    return 0;
}
```

使用内联函数替代宏定义，能消除宏定义的不安全性

带有默认参数值的函数

当进行函数调用时，编译器按从左到右的顺序将实参与形参结合，若未指定足够的实参，则编译器按顺序用函数原型中的默认值来补足所缺少的实参。

```
void init(int x = 5, int y = 10);
init (100, 19);     // 100  ,    19
init(25);              // 25, 10
init();                 // 5,    10
```

在函数原型中，所有取默认值的参数都必须出现在不取默认值的参数的右边。

如　　int fun(int a, int b, int c = 111);
1
在函数调用时，若某个参数省略，则其后的参数皆应省略而采取默认值。不允许某个参数省略后，再给其后的参数指定参数值。

函数重载
在 C++中，用户可以重载函数。这意味着，在同一作用域内，只要函数参数的类型不同，或者参数的个数不同，或者二者兼而有之，两个或者两个以上的函数可以使用相同的函数名。

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

int add(int x, int y)
{
	return x + y;
}

double add(double x, double y)
{
	return x + y;
}

int add(int x, int y, int z)
{
	return x + y + z;
}

int main()
{
	int a = 3, b = 5, c = 7;
	double x = 10.334, y = 8.9003;
	cout << add(a, b) << endl;
	cout << add(x, y) << endl;
	cout << add(a, b, c) << endl;
	return 0;
}
```
说明：

调用重载函数时，函数返回值类型不在参数匹配检查之列。因此，若两个函数的参数个数和类型都相同，而只有返回值类型不同，则不允许重载。

```
int mul(int x, int y);
double mul(int x, int y);
```

函数的重载与带默认值的函数一起使用时，有可能引起二义性。

```
void Drawcircle(int r = 0, int x = 0, int y = 0);
void Drawcircle(int r);
Drawcircle(20);
```

在调用函数时, 如果给出的实参和形参类型不相符, C++的编译器会自动地做类型转换工作。如果转换成功，则程序继续执行，在这种情况下，有可能产生不可识别的错误。

```
void f_a(int x);
void f_a(long x);
f_a(20.83);
```

作用域标识符"::"
通常情况下，如果有两个同名变量，一个是全局的，另一个是局部的，那么局部变量在其作用域内具有较高的优先权，它将屏蔽全局变量。

如果希望在局部变量的作用域内使用同名的全局变量，可以在该变量前加上“::”，此时::value 代表全局变量 value，“::”称为作用域标识符。

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

int value;     //定义全局变量 value

int main()
{
    int value;   //定义局部变量 value
    value = 100;
    ::value = 1000;
    cout << "local value : " << value << endl;
    cout << "global value : " << ::value << endl;
    return 0;
}
```

强制类型转换
可用强制类型转换将不同类型的数据进行转换。例如，要把一个整型数（int）转换为双精度型数（double），可使用如下的格式：

```
int i = 10;
double x = (double)i;
```

或

```
int i = 10;
double x = double(i);
```

以上两种方法 C++都能接受，建议使用后一种方法。

new 和 delete 运算符
程序运行时，计算机的内存被分为 4 个区：程序代码区、全局数据区、堆和栈。其中，堆可由用户分配和释放。C 语言中使用函数 malloc()和 free()来进行动态内存管理。C++则提供了运算符 new 和 delete 来做同样的工作，而且后者比前者性能更优越，使用更灵活方便。

指针变量名 = new 类型
```
int *p;
p = new int;
```
delete 指针变量名
```
delete p;
```
下面对 new 和 delete 的使用再做一下几点说明:

用运算符 new 分配的空间，使用结束后应该用也只能用 delete 显式地释放，否则这部分空间将不能回收而变成死空间。

在使用运算符 new 动态分配内存时，如果没有足够的内存满足分配要求，new 将返回空指针（NULL）。

使用运算符 new 可以为数组动态分配内存空间，这时需要在类型后面加上数组大小。

指针变量名 = new 类型名[下标表达式];
```
int *p = new int[10];
```

释放动态分配的数组存储区时，可使用 delete 运算符。

```
delete []指针变量名;
delete p;
```

new 可在为简单变量分配空间的同时，进行初始化

指针变量名 = new 类型名(初值);
```
int *p;
p = new int(99);
.
delete p;
```
引用
引用（reference）是 C++对 C 的一个重要扩充。变量的引用就是变量的别名，因此引用又称别名。

类型 &引用名 = 已定义的变量名
1
引用与其所代表的变量共享同一内存单元，系统并不为引用另外分配存储空间。实际上，编

译系统使引用和其代表的变量具有相同的地址。

```
#include <iostream>
using namespace std;
int main()
{
	int i = 10;
	int &j = i;
	cout << "i = " << i << " j = " << j << endl;
	cout << "i 的地址为 " << &i << endl;
	cout << "j 的地址为 " << &j << endl;
	return 0;
}
```

上面代码输出 i 和 j 的值相同，地址也相同。

引用并不是一种独立的数据类型，它必须与某一种类型的变量相联系。在声明引用时，必须立即对它进行初始化，不能声明完成后再赋值。
为引用提供的初始值，可以是一个变量或者另一个引用。
指针是通过地址间接访问某个变量，而引用则是通过别名直接访问某个变量。
引用作为函数参数、使用引用返回函数值

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

void swap(int &a, int &b)
{
	int t = a;
	a = b;
	b = t;
}

int a[] = {1, 3, 5, 7, 9};

int& index(int i)
{
	return a[i];
}

int main()
{
	int a = 5, b = 10;
	//交换数字 a 和 b
	swap(a, b);
	cout << "a = " << a << " b = " << b << endl;
```

```
    cout << index(2) << endl;     //等价于输出元素 a[2]的值
    index(2) = 100;                //等价于将 a[2]的值赋为 100;
    cout << index(2) << endl;

    return 0;
}
```

对引用的进一步说明

不允许建立 void 类型的引用
不能建立引用的数组
不能建立引用的引用。不能建立指向引用的指针。引用本身不是一种数据类型，所以没有引用的引用，也没有引用的指针。
可以将引用的地址赋值给一个指针，此时指针指向的是原来的变量。
可以用 const 对引用加以限定，不允许改变该引用的值，但是它不阻止引用所代表的变量的值。

三、类和对象（一）

~

3.1 类的构成

类声明中的内容包括数据和函数，分别称为数据成员和成员函数。按访问权限划分，数据成员和成员函数又可分为共有、保护和私有 3 种。

```
class 类名{
    public:
        公有数据成员;
        公有成员函数;
    protected:
        保护数据成员;
        保护成员函数;
    private:
        私有数据成员;
        私有成员函数;
};
```

如成绩类

```
class Score{
    public:
    void setScore(int m, int f);
    void showScore();
    private:
    int mid_exam;
    int fin_exam;
};
```

对一个具体的类来讲，类声明格式中的 3 个部分并非一定要全有，但至少要有其中的一个部分。一般情况下，一个类的数据成员应该声明为私有成员，成员函数声明为共有成员。这样，内部的数据整个隐蔽在类中，在类的外部根本就无法看到，使数据得到有效的保护，也不会对该类以外的其余部分造成影响，程序之间的相互作用就被降低到最小。
类声明中的关键字 private、protected、public 可以任意顺序出现。
若私有部分处于类的第一部分时，关键字 private 可以省略。这样，如果一个类体中没有一个访问权限关键字，则其中的数据成员和成员函数都默认为私有的。
不能在类声明中给数据成员赋初值。
~

3.2 成员函数的定义
普通成员函数的定义

在类的声明中只给出成员函数的原型，而成员函数的定义写在类的外部。这种成员函数在类外定义的一般形式是：

返回值类型 类名::成员函数名(参数表){ 函数体}
1
例如，表示分数的类 Score 可声明如下：

```
class Score{
public:
    void setScore(int m, int f);
    void showScore();
private:
    int mid_exam;
    int fin_exam;
};

void Score::setScore(int m, int f)
{
    mid_exam = m;
    fin_exam = f;
}

void Score::showScore()
{
    cout << "期中成绩: " << mid_exam << endl;
    cout << "期末成绩：  " << fin_exam << endl;
}
```

内联成员函数的定义

隐式声明：将成员函数直接定义在类的内部

```
class Score{
```

```
public:
    void setScore(int m, int f)
    {
        mid_exam = m;
        fin_exam = f;
    }
    void showScore()
    {
        cout << "期中成绩: " << mid_exam << endl;
        cout << "期末成绩:  " << fin_exam << endl;
    }
private:
    int mid_exam;
    int fin_exam;
};
```

显式声明：在类声明中只给出成员函数的原型，而将成员函数的定义放在类的外部。

```
class Score{
public:
    inline void setScore(int m, int f);
    inline void showScore();
private:
    int mid_exam;
    int fin_exam;
};

inline void Score::setScore(int m, int f)
{
    mid_exam = m;
    fin_exam = f;
}

inline void Score::showScore()
{
    cout << "期中成绩: " << mid_exam << endl;
    cout << "期末成绩:  " << fin_exam << endl;
}
```

说明：在类中，使用 inline 定义内联函数时，必须将类的声明和内联成员函数的定义都放在同一个文件（或同一个头文件）中，否则编译时无法进行代码置换。

~

3.3 对象的定义和使用

通常把具有共同属性和行为的事物所构成的集合称为类。

类的对象可以看成该类类型的一个实例，定义一个对象和定义一个一般变量相似。

对象的定义

在声明类的同时，直接定义对象

```
class Score{
public:
    void setScore(int m, int f);
    void showScore();
private:
    int mid_exam;
    int fin_exam;
}op1, op2;
```

声明了类之后，在使用时再定义对象

```
  Score op1, op2;
```

对象中成员的访问

对象名.数据成员名对象名.成员函数名[(参数表)]op1.setScore(89, 99);
op1.showScore();

说明:

在类的内部所有成员之间都可以通过成员函数直接访问，但是类的外部不能访问对象的私有成员。

在定义对象时，若定义的是指向此对象的指针变量，则访问此对象的成员时，不能用“.”操作符，而应该使用“->“操作符。如

```
    Score op, *sc;
    sc = &op;
    sc->setScore(99, 100);
    op.showScore();
```

类的作用域和类成员的访问属性

私有成员只能被类中的成员函数访问，不能在类的外部，通过类的对象进行访问。

一般来说，公有成员是类的对外接口，而私有成员是类的内部数据和内部实现，不希望外界访问。将类的成员划分为不同的访问级别有两个好处：一是信息隐蔽，即实现封装，将类的内部数据与内部实现和外部接口分开，这样使该类的外部程序不需要了解类的详细实现；二是数据保护，即将类的重要信息保护起来，以免其他程序进行不恰当的修改。

对象赋值语句

```
    Score op1, op2;
    op1.setScore(99, 100);
    op2 = op1;
    op2.showScore();
~
```

3.4 构造函数与析构函数

构造函数

构造函数是一种特殊的成员函数，它主要用于为对象分配空间，进行初始化。构造函数的名字必须与类名相同，而不能由用户任意命名。它可以有任意类型的参数，但不能具有返回值。它不需要用户来调用，而是在建立对象时自动执行。

```
class Score{
public:
    Score(int m, int f);   //构造函数
    void setScore(int m, int f);
    void showScore();
private:
    int mid_exam;
    int fin_exam;
};

Score::Score(int m, int f)
{
    mid_exam = m;
    fin_exam = f;
}
```

在建立对象的同时，采用构造函数给数据成员赋值，通常由以下两种形式

```
类名 对象名[(实参表)]
Score op1(99, 100);
op1.showScore();
1
2
3
类名 *指针变量名 = new 类名[(实参表)]
Score *p;
p = new Score(99, 100);
p->showScore();
----------------------
Score *p = new Score(99, 100);
p->showScore();
```

说明：

构造函数的名字必须与类名相同，否则编译程序将把它当做一般的成员函数来处理。
构造函数没有返回值，在定义构造函数时，是不能说明它的类型的。
与普通的成员函数一样，构造函数的函数体可以写在类体内，也可写在类体外。
构造函数一般声明为共有成员，但它不需要也不能像其他成员函数那样被显式地调用，它是在定义对象的同时被自动调用，而且只执行一次。
构造函数可以不带参数。
成员初始化列表
在声明类时，对数据成员的初始化工作一般在构造函数中用赋值语句进行。此外还可以用成员初始化列表实现对数据成员的初始化。

```
类名::构造函数名([参数表])[:(成员初始化列表)]
{
    //构造函数体
}
class A{
private:
    int x;
    int& rx;
    const double pi;
public:
    A(int v) : x(v), rx(x), pi(3.14)       //成员初始化列表
    {    }
    void print()
    {
        cout << "x = " << x << " rx = " << rx << " pi = " << pi << endl;
    }
};
```

说明：类成员是按照它们在类里被声明的顺序进行初始化的，与它们在成员初始化列表中列出的顺序无关。

带默认参数的构造函数

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

class Score{
public:
    Score(int m = 0, int f = 0);      //带默认参数的构造函数
    void setScore(int m, int f);
    void showScore();
private:
    int mid_exam;
    int fin_exam;
};
```

```
Score::Score(int m, int f) : mid_exam(m), fin_exam(f)
{
	cout << "构造函数使用中..." << endl;
}

void Score::setScore(int m, int f)
{
	mid_exam = m;
	fin_exam = f;
}

void Score::showScore()
{
	cout << "期中成绩: " << mid_exam << endl;
	cout << "期末成绩:  " << fin_exam << endl;
}

int main()
{
	Score op1(99, 100);
	Score op2(88);
	Score op3;
	op1.showScore();
	op2.showScore();
	op3.showScore();

	return 0;
}
```

析构函数

析构函数也是一种特殊的成员函数。它执行与构造函数相反的操作，通常用于撤销对象时的一些清理任务，如释放分配给对象的内存空间等。析构函数有以下一些特点：

析构函数与构造函数名字相同，但它前面必须加一个波浪号（~）。
析构函数没有参数和返回值，也不能被重载，因此只有一个。
当撤销对象时，编译系统会自动调用析构函数。

```
class Score{
public:
	Score(int m = 0, int f = 0);
	~Score();		//析构函数
private:
	int mid_exam;
	int fin_exam;
```

```
};

Score::Score(int m, int f) : mid_exam(m), fin_exam(f)
{
    cout << "构造函数使用中..." << endl;
}

Score::~Score()
{
    cout << "析构函数使用中..." << endl;
}
```
说明：在以下情况中，当对象的生命周期结束时，析构函数会被自动调用：

如果定义了一个全局对象，则在程序流程离开其作用域时，调用该全局对象的析构函数。
如果一个对象定义在一个函数体内，则当这个函数被调用结束时，该对象应该被释放，析构函数被自动调用。
若一个对象是使用 new 运算符创建的，在使用 delete 运算符释放它时，delete 会自动调用析构函数。
如下示例：

```
#include <iostream>
#include <string>

using namespace std;

class Student{
private:
    char *name;
    char *stu_no;
    float score;
public:
    Student(char *name1, char *stu_no1, float score1);
    ~Student();
    void modify(float score1);
    void show();
};

Student::Student(char *name1, char *stu_no1, float score1)
{
    name = new char[strlen(name1) + 1];
    strcpy(name, name1);
    stu_no = new char[strlen(stu_no1) + 1];
    strcpy(stu_no, stu_no1);
    score = score1;
```

```
}

Student::~Student()
{
	delete []name;
	delete []stu_no;
}

void Student::modify(float score1)
{
	score = score1;
}

void Student::show()
{
	cout << "姓名: " << name << endl;
	cout << "学号: " << stu_no << endl;
	cout << "成绩:  " << score << endl;
}

int main()
{
	Student stu("雪女", "2020199012", 99);
	stu.modify(100);
	stu.show();

	return 0;
}
```

默认的构造函数和析构函数
如果没有给类定义构造函数，则编译系统自动生成一个默认的构造函数。

说明:

对没有定义构造函数的类，其公有数据成员可以用初始值列表进行初始化。

```
class A{
public:
	char name[10];
	int no;
};

A a = {"chen", 23};
cout << a.name << a.no << endl;
```

只要一个类定义了一个构造函数（不一定是无参构造函数），系统将不再给它提供默认的构造函数。

每个类必须有一个析构函数。若没有显示地为一个类定义析构函数，编译系统会自动生成一个默认的析构函数。

构造函数的重载

```
class Score{
public:
    Score(int m, int f);   //构造函数
    Score();
    void setScore(int m, int f);
    void showScore();
private:
    int mid_exam;
    int fin_exam;
};
```

注意：在一个类中，当无参数的构造函数和带默认参数的构造函数重载时，有可能产生二义性。

拷贝构造函数

拷贝构造函数是一种特殊的构造函数，其形参是本类对象的引用。拷贝构造函数的作用是在建立一个新对象时，使用一个已存在的对象去初始化这个新对象。

拷贝构造函数具有以下特点：

因为拷贝构造函数也是一种构造函数，所以其函数名与类名相同，并且该函数也没有返回值。
拷贝构造函数只有一个参数，并且是同类对象的引用。
每个类都必须有一个拷贝构造函数。可以自己定义拷贝构造函数，用于按照需要初始化新对象；如果没有定义类的拷贝构造函数，系统就会自动生成一个默认拷贝构造函数，用于复制出与数据成员值完全相同的新对象。
自定义拷贝构造函数

```
类名::类名(const  类名  &对象名)
{
    拷贝构造函数的函数体;
}
```

```
class Score{
public:
    Score(int m, int f);   //构造函数
    Score();
    Score(const Score &p);   //拷贝构造函数
    ~Score();                    //析构函数
```

```
    void setScore(int m, int f);
    void showScore();
private:
    int mid_exam;
    int fin_exam;
};

Score::Score(int m, int f)
{
    mid_exam = m;
    fin_exam = f;
}

Score::Score(const Score &p)
{
    mid_exam = p.mid_exam;
    fin_exam = p.fin_exam;
}
```

调用拷贝构造函数的一般形式为：

```
    类名  对象 2(对象 1);
    类名  对象 2 =  对象 1;
Score sc1(98, 87);
Score sc2(sc1);      //调用拷贝构造函数
Score sc3 = sc2;     //调用拷贝构造函数
```

调用拷贝构造函数的三种情况：

当用类的一个对象去初始化该类的另一个对象时；
当函数的形参是类的对象，调用函数进行形参和实参结合时；
当函数的返回值是对象，函数执行完成返回调用者时。
浅拷贝和深拷贝
浅拷贝，就是由默认的拷贝构造函数所实现的数据成员逐一赋值。通常默认的拷贝构造函数是能够胜任此工作的，但若类中含有指针类型的数据，则这种按数据成员逐一赋值的方法会产生错误。

```
class Student{
public:
    Student(char *name1, float score1);
    ~Student();
private:
    char *name;
    float score;
};
```

如下语句会产生错误
Student stu1("白", 89);
Student stu2 = stu1;
上述错误是因为 stu1 和 stu2 所指的内存空间相同，在析构函数释放 stu1 所指的内存后，再释放 stu2 所指的内存会发生错误，因为此内存空间已被释放。解决方法就是重定义拷贝构造函数，为其变量重新生成内存空间。

```
Student::Student(const Student& stu)
{
	name = new char[strlen(stu.name) + 1];
	if (name != 0) {
		strcpy(name, stu.name);
		score = stu.score;
	}
}
```

四、类和对象（二）
~

4.1 自引用指针 this
this 指针保存当前对象的地址，称为自引用指针。

```
void Sample::copy(Sample& xy)
{
	if (this == &xy) return;
	*this = xy;
}
```

~

4.2 对象数组与对象指针
对象数组

类名 数组名[下标表达式]
用只有一个参数的构造函数给对象数组赋值
Exam ob[4] = {89, 97, 79, 88};
用不带参数和带一个参数的构造函数给对象数组赋值
Exam ob[4] = {89, 90};
用带有多个参数的构造函数给对象数组赋值
Score rec[3] = {Score(33, 99), Score(87, 78), Score(99, 100)};
对象指针

每一个对象在初始化后都会在内存中占有一定的空间。因此，既可以通过对象名访问对象，也可以通过对象地址来访问对象。对象指针就是用于存放对象地址的变量。声明对象指针的一半语法形式为：类名 *对象指针名

```
Score score;
Score *p;
p = &score;
p->成员函数();
```
用对象指针访问对象数组

```
Score score[2];
score[0].setScore(90, 99);
score[1].setScore(67, 89);

Score *p;
p = score;     //将对象 score 的地址赋值给 p
p->showScore();
p++;       //对象指针变量加 1
p->showSccore();

Score *q;
q =&score[1];     //将第二个数组元素的地址赋值给对象指针变量 q
```

4.3 string 类

C++支持两种类型的字符串，第一种是 C 语言中介绍过的、包括一个结束符'\0'（即以 NULL 结束）的字符数组，标准库函数提供了一组对其进行操作的函数，可以完成许多常用的字符串操作。

C++标准库中声明了一种更方便的字符串类型，即字符串类 string，类 string 提供了对字符串进行处理所需要的操作。使用 string 类必须在程序的开始包括头文件 string，即要有以下语句：#include <string>

常用的 string 类运算符如下：

=、+、+=、==、!=、<、<=、>、>=、[]（访问下标对应字符）、>>（输入）、<<（输出）

1

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

int main()
{
	string str1 = "ABC";
	string str2("dfdf");
	string str3 = str1 + str2;
	cout<< "str1 = " << str1 << "    str2 = " << str2 << "    str3 = " << str3 << endl;
	str2 += str2;
	str3 += "aff";
```

```
    cout << "str2 = " << str2 << "    str3 = " << str3 << endl;
    cout << "str1[1] = " << str1[1] << "    str1 == str2 ? " << (str1 == str2) << endl;
    string str = "ABC";
    cout << "str == str1 ? " << (str == str1) << endl;
    return 0;
}
~
```

4.4 向函数传递对象

使用对象作为函数参数: 对象可以作为参数传递给函数, 其方法与传递其他类型的数据相同。在向函数传递对象时，是通过“传值调用”的方法传递给函数的。因此，函数中对对象的任何修改均不影响调用该函数的对象（实参本身）。

使用对象指针作为函数参数: 对象指针可以作为函数的参数，使用对象指针作为函数参数可以实现传值调用，即在函数调用时使实参对象和形参对象指针变量指向同一内存地址，在函数调用过程中，形参对象指针所指的对象值的改变也同样影响着实参对象的值。

使用对象引用作为函数参数: 在实际中，使用对象引用作为函数参数非常普遍，大部分程序员喜欢使用对象引用替代对象指针作为函数参数。因为使用对象引用作为函数参数不但具有用对象指针做函数参数的优点，而且用对象引用作函数参数将更简单、更直接。

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

class Point{
public:
    int x;
    int y;
    Point(int x1, int y1) : x(x1), y(y1)   //成员初始化列表
    { }
    int getDistance()
    {
        return x * x + y * y;
    }
};

void changePoint1(Point point)      //使用对象作为函数参数
{
    point.x += 1;
    point.y -= 1;
}

void changePoint2(Point *point)     //使用对象指针作为函数参数
{
    point->x += 1;
    point->y -= 1;
}
```

```
void changePoint3(Point &point)   //使用对象引用作为函数参数
{
	point.x += 1;
	point.y -= 1;
}


int main()
{
	Point point[3] = {Point(1, 1), Point(2, 2), Point(3, 3)};
	Point *p = point;
	changePoint1(*p);
	cout << "the distance is " << p[0].getDistance() << endl;
	p++;
	changePoint2(p);
	cout << "the distance is " << p->getDistance() << endl;
	changePoint3(point[2]);
	cout << "the distance is " << point[2].getDistance() << endl;

	return 0;
}
~
```

4.5 静态成员

静态数据成员

在一个类中，若将一个数据成员说明为 static，则这种成员被称为静态数据成员。与一般的数据成员不同，无论建立多少个类的对象，都只有一个静态数据成员的拷贝。从而实现了同一个类的不同对象之间的数据共享。

定义静态数据成员的格式如下：static 数据类型 数据成员名;

说明：

静态数据成员的定义与普通数据成员相似，但前面要加上 static 关键字。

静态数据成员的初始化与普通数据成员不同。静态数据成员初始化应在类外单独进行，而且应在定义对象之前进行。一般在 main()函数之前、类声明之后的特殊地带为它提供定义和初始化。

静态数据成员属于类（准确地说，是属于类中对象的集合），而不像普通数据成员那样属于某一对象，因此，可以使用“类名::”访问静态的数据成员。格式如下：类名::静态数据成员名。

静态数据成员与静态变量一样，是在编译时创建并初始化。它在该类的任何对象被建立之前就存在。因此，共有的静态数据成员可以在对象定义之前被访问。对象定以后，共有的静态数据成员也可以通过对象进行访问。其访问格式如下

对象名.静态数据成员名;
对象指针->静态数据成员名;
1
2
静态成员函数

在类定义中，前面有 static 说明的成员函数称为静态成员函数。静态成员函数属于整个类，是该类所有对象共享的成员函数，而不属于类中的某个对象。静态成员函数的作用不是为了对象之间的沟通，而是为了处理静态数据成员。定义静态成员函数的格式如下:

static 返回类型 静态成员函数名（参数表）;

与静态数据成员类似，调用公有静态成员函数的一般格式有如下几种:

类名::静态成员函数名(实参表);
对象.静态成员函数名(实参表);
对象指针->静态成员函数名(实参表);
1
2
3
一般而言，静态成员函数不访问类中的非静态成员。若确实需要，静态成员函数只能通过对象名（或对象指针、对象引用）访问该对象的非静态成员。

下面对静态成员函数的使用再做几点说明:

一般情况下，静态函数成员主要用来访问静态成员函数。当它与静态数据成员一起使用时，达到了对同一个类中对象之间共享数据的目的。
私有静态成员函数不能被类外部的函数和对象访问。
使用静态成员函数的一个原因是，可以用它在建立任何对象之前调用静态成员函数，以处理静态数据成员，这是普通成员函数不能实现的功能
编译系统将静态成员函数限定为内部连接，也就是说，与现行文件相连接的其他文件中的同名函数不会与该函数发生冲突，维护了该函数使用的安全性，这是使用静态成员函数的另一个原因。
静态成员函数是类的一部分，而不是对象的一部分。如果要在类外调用公有的静态成员函数，使用如下格式较好：类名::静态成员函数名()

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

class Score{
private:
```

```
    int mid_exam;
    int fin_exam;
    static int count;       //静态数据成员，用于统计学生人数
    static float sum;       //静态数据成员，用于统计期末累加成绩
    static float ave;       //静态数据成员，用于统计期末平均成绩
public:
    Score(int m, int f);
    ~Score();
    static void show_count_sum_ave();    //静态成员函数
};

Score::Score(int m, int f)
{
    mid_exam = m;
    fin_exam = f;
    ++count;
    sum += fin_exam;
    ave = sum / count;
}

Score::~Score()
{

}

/*** 静态成员初始化 ***/
int Score::count = 0;
float Score::sum = 0.0;
float Score::ave = 0.0;

void Score::show_count_sum_ave()
{
    cout << "学生人数: " << count << endl;
    cout << "期末累加成绩: " << sum << endl;
    cout << "期末平均成绩: " << ave << endl;
}

int main()
{
    Score sco[3] = {Score(90, 89), Score(78, 99), Score(89, 88)};
    sco[2].show_count_sum_ave();
    Score::show_count_sum_ave();

    return 0;
```

```
}
```

4.6 友元

类的主要特点之一是数据隐藏和封装，即类的私有成员（或保护成员）只能在类定义的范围内使用，也就是说私有成员只能通过它的成员函数来访问。但是，有时为了访问类的私有成员而需要在程序中多次调用成员函数，这样会因为频繁调用带来较大的时间和空间开销，从而降低程序的运行效率。为此，C++提供了友元来对私有或保护成员进行访问。友元包括友元函数和友元类。

友元函数

友元函数既可以是不属于任何类的非成员函数，也可以是另一个类的成员函数。友元函数不是当前类的成员函数，但它可以访问该类的所有成员，包括私有成员、保护成员和公有成员。

在类中声明友元函数时，需要在其函数名前加上关键字 friend。此声明可以放在公有部分，也可以放在保护部分和私有部分。友元函数可以定义在类内部，也可以定义在类外部。

1、将非成员函数声明为友元函数

```
#include <iostream>
using namespace std;
class Score{
private:
    int mid_exam;
    int fin_exam;
public:
    Score(int m, int f);
    void showScore();
    friend int getScore(Score &ob);
};

Score::Score(int m, int f)
{
    mid_exam = m;
    fin_exam = f;
}

int getScore(Score &ob)
{
    return (int)(0.3 * ob.mid_exam + 0.7 * ob.fin_exam);
}

int main()
{
    Score score(98, 78);
```

```
    cout << "成绩为: " << getScore(score) << endl;

    return 0;
}
```

说明：

友元函数虽然可以访问类对象的私有成员，但他毕竟不是成员函数。因此，在类的外部定义友元函数时，不必像成员函数那样，在函数名前加上“类名::”。
因为友元函数不是类的成员，所以它不能直接访问对象的数据成员，也不能通过 this 指针访问对象的数据成员，它必须通过作为入口参数传递进来的对象名（或对象指针、对象引用）来访问该对象的数据成员。
友元函数提供了不同类的成员函数之间、类的成员函数与一般函数之间进行数据共享的机制。尤其当一个函数需要访问多个类时，友元函数非常有用，普通的成员函数只能访问其所属的类，但是多个类的友元函数能够访问相关的所有类的数据。
例子：一个函数同时定义为两个类的友元函数

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

class Score;      //对 Score 类的提前引用说明
class Student{
private:
    string name;
    int number;
public:
    Student(string na, int nu) {
        name = na;
        number = nu;
    }
    friend void show(Score &sc, Student &st);
};

class Score{
private:
    int mid_exam;
    int fin_exam;
public:
    Score(int m, int f) {
        mid_exam = m;
        fin_exam = f;
    }
    friend void show(Score &sc, Student &st);
```

```
};

void show(Score &sc, Student &st) {
	cout << "姓名：" << st.name << "　学号：" << st.number << endl;
	cout << "期中成绩：" << sc.mid_exam << "　期末成绩：" << sc.fin_exam << endl;
}

int main() {
	Score sc(89, 99);
	Student st("白", 12467);
	show(sc, st);

	return 0;
}
```

2、将成员函数声明为友元函数

一个类的成员函数可以作为另一个类的友元，它是友元函数中的一种，称为友元成员函数。友元成员函数不仅可以访问自己所在类对象中的私有成员和公有成员，还可以访问 friend 声明语句所在类对象中的所有成员，这样能使两个类相互合作、协调工作，完成某一任务。

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

class Score;	//对 Score 类的提前引用说明
class Student{
private:
	string name;
	int number;
public:
	Student(string na, int nu) {
		name = na;
		number = nu;
	}
	void show(Score &sc);
};

class Score{
private:
	int mid_exam;
	int fin_exam;
public:
	Score(int m, int f) {
		mid_exam = m;
```

```
            fin_exam = f;
        }
        friend void Student::show(Score &sc);
};

void Student::show(Score &sc) {
        cout << "姓名：" << name << "    学号：" << number << endl;
        cout << "期中成绩：" << sc.mid_exam << "    期末成绩：" << sc.fin_exam << endl;
}

int main() {
        Score sc(89, 99);
        Student st("白", 12467);
        st.show(sc);

        return 0;
}
```

说明：

一个类的成员函数作为另一个类的友元函数时，必须先定义这个类。并且在声明友元函数时，需要加上成员函数所在类的类名；

友元类

可以将一个类声明为另一个类的友元

```
class Y{
        ...
};
class X{
        friend Y;       //声明类 Y 为类 X 的友元类
};
```

当一个类被说明为另一个类的友元类时，它所有的成员函数都成为另一个类的友元函数，这就意味着作为友元类中的所有成员函数都可以访问另一个类中的所有成员。

友元关系不具有交换性和传递性。

~

4.7 类的组合

在一个类中内嵌另一个类的对象作为数据成员，称为类的组合。该内嵌对象称为对象成员，又称为子对象。

```
class Y{
        ...
};
```

```
class X{
    Y y;
    ...
};
~
```

4.8 共享数据的保护

常类型的引入就是为了既保护数据共享又防止数据被改动。常类型是指使用类型修饰符 const 说明的类型，常类型的变量或对象成员的值在程序运行期间是不可改变的。

常引用

如果在说明引用时用 const 修饰，则被说明的引用为常引用。常引用所引用的对象不能被更新。如果用常引用做形参，便不会产生对实参的不希望的更改。

const 类型& 引用名

```
int a = 5;
const int& b = a;
```

此时再对 b 赋值是非法的。

```
--------------------------
int add(const int& m, const int& n) {
    return m + n;
}
```

在此函数中对变量 m 和变量 n 更新时非法的

常对象

如果在说明对象时用 const 修饰，则被说明的对象为常对象。常对象中的数据成员为常量且必须要有初值。

类名 const 对象名[(参数表)];

```
const Date date(2021, 5, 31);
```

1

常对象成员

1、常数据成员

类的数据成员可以是常量或常引用，使用 const 说明的数据成员称为常数据成员。如果在一个类中说明了常数据成员，那么构造函数就只能通过成员初始化列表对该数据成员进行初始化，而任何其他函数都不能对该成员赋值。

```
class Date{
private:
```

```
    int year;
    int month;
    int day;
public:
    Date(int y, int m, int d) : year(y), month(m), day(d) {

    }
};
```

一旦某对象的常数据成员初始化后，该数据成员的值是不能改变的。

2、常成员函数

类型 函数名(参数表) const;

const 是函数类型的一个组成部分，因此在声明函数和定义函数时都要有关键字 const。在调用时不必加 const。

```
class Date{
private:
    int year;
    int month;
    int day;
public:
    Date(int y, int m, int d) : year(y), month(m), day(d){

    }
    void showDate();
    void showDate() const;
};

void Date::showDate() {
    //…
}

void Date::showDate() const {
    //…
}
```

关键字 const 可以被用于对重载函数进行区分。

说明：

常成员函数可以访问常数据成员，也可以访问普通数据成员。
常对象只能调用它的常成员对象，而不能调用普通成员函数。常成员函数是常对象唯一的对外接口。

常对象函数不能更新对象的数据成员，也不能调用该类的普通成员函数，这就保证了在常成员函数中绝不会更新数据成员的值。
五、继承与派生
继承可以在已有类的基础上创建新的类，新类可以从一个或多个已有类中继承成员函数和数据成员，而且可以重新定义或加进新的数据和函数，从而形成类的层次或等级。其中，已有类称为基类或父类，在它基础上建立的新类称为派生类或子类。

~

5.1 继承与派生的概念
类的继承是新的类从已有类那里得到已有的特性。从另一个角度来看这个问题，从已有类产生新类的过程就是类的派生。类的继承和派生机制较好地解决了代码重用的问题。

关于基类和派生类的关系，可以表述为：派生类是基类的具体化，而基类则是派生类的抽象。

使用继承的案例如下：

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

class Person{
private:
	string name;
	string id_number;
	int age;
public:
	Person(string name1, string id_number1, int age1) {
		name = name1;
		id_number = id_number1;
		age = age1;
	}
	~Person() {

	}
	void show() {
		cout << "姓名: " << name << "  身份证号: " << id_number << "  年龄: " << age
<< endl;
	}
};

class Student:public Person{
private:
	int credit;
```

```
public:
    Student(string name1, string id_number1, int age1, int credit1):Person(name1, id_number1, credit1) {
        credit = credit1;
    }
    ~Student() {

    }
    void show() {
        Person::show();
        cout << "学分: " << credit << endl;
    }
};

int main() {
    Student stu("白", "110103**********23", 12, 123);
    stu.show();

    return 0;
}
```

从已有类派生出新类时，可以在派生类内完成以下几种功能：

可以增加新的数据成员和成员函数
可以对基类的成员进行重定义
可以改变基类成员在派生类中的访问属性
基类成员在派生类中的访问属性
派生类可以继承基类中除了构造函数与析构函数之外的成员，但是这些成员的访问属性在派生过程中是可以调整的。从基类继承来的成员在派生类中的访问属性也有所不同。

基类中的成员	继承方式	基类在派生类中的访问属性
private	public|protected|private	不可直接访问
public	public|protected|private	public|protected|private
protected	public|protected|private	protected|protected|private

派生类对基类成员的访问规则
基类的成员可以有 public、protected、private3 中访问属性，基类的成员函数可以访问基类中其他成员，但是在类外通过基类的对象，就只能访问该基类的公有成员。同样，派生类的成员也可以有 public、protected、private3 种访问属性，派生类的成员函数可以访问派生类中自己增加的成员，但是在派生类外通过派生类的对象，就只能访问该派生类的公有成员。

派生类对基类成员的访问形式主要有以下两种：

内部访问：由派生类中新增的成员函数对基类继承来的成员的访问。
对象访问：在派生类外部，通过派生类的对象对从基类继承来的成员的访问。

~

5.2 派生类的构造函数和析构函数

构造函数的主要作用是对数据进行初始化。在派生类中，如果对派生类新增的成员进行初始化，就需要加入派生类的构造函数。与此同时，对所有从基类继承下来的成员的初始化工作，还是由基类的构造函数完成，但是基类的构造函数和析构函数不能被继承，因此必须在派生类的构造函数中对基类的构造函数所需要的参数进行设置。同样，对撤销派生类对象的扫尾、清理工作也需要加入新的析构函数来完成。

调用顺序

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

class A{
public:
	A() {
		cout << "A 类对象构造中..." << endl;
	}
	~A() {
		cout << "析构 A 类对象..." << endl;
	}
};

class B : public A{
public:
	B() {
		cout << "B 类对象构造中..." << endl;
	}
	~B(){
		cout << "析构 B 类对象..." << endl;
	}
};

int main() {
	B b;
	return 0;
}
```

代码运行结果如下：

```
A 类对象构造中...
B 类对象构造中...
析构 B 类对象...
```

析构 A 类对象...
可见：构造函数的调用严格地按照先调用基类的构造函数，后调用派生类的构造函数的顺序执行。析构函数的调用顺序与构造函数的调用顺序正好相反，先调用派生类的析构函数，后调用基类的析构函数。

派生类构造函数和析构函数的构造规则

派生类构造函数的一般格式为：
派生类名(参数总表):基类名(参数表) {
    派生类新增数据成员的初始化语句
}

含有子对象的派生类的构造函数：
派生类名(参数总表):基类名(参数表 0),子对象名 1(参数表 1),...,子对象名 n(参数表 n)
{
    派生类新增成员的初始化语句
}
在定义派生类对象时，构造函数的调用顺序如下：

调用基类的构造函数，对基类数据成员初始化。

调用子对象的构造函数，对子对象的数据成员初始化。

调用派生类的构造函数体，对派生类的数据成员初始化。

说明：

当基类构造函数不带参数时，派生类不一定需要定义构造函数；然而当基类的构造函数哪怕只带有一个参数，它所有的派生类都必须定义构造函数，甚至所定义的派生类构造函数的函数体可能为空，它仅仅起参数的传递作用。
若基类使用默认构造函数或不带参数的构造函数，则在派生类中定义构造函数时可略去":基类构造函数名(参数表)"，此时若派生类也不需要构造函数，则可不定义构造函数。
如果派生类的基类也是一个派生类，每个派生类只需负责其直接基类数据成员的初始化，依次上溯。
~

5.3 调整基类成员在派生类中的访问属性的其他方法
派生类可以声明与基类成员同名的成员。在没有虚函数的情况下，如果在派生类中定义了与基类成员同名的成员，则称派生类成员覆盖了基类的同名成员，在派生类中使用这个名字意味着访问在派生类中声明的成员。为了在派生类中使用与基类同名的成员，必须在该成员名之前加上基类名和作用域标识符"::"，即

基类名::成员名

访问声明

访问声明的方法就是把基类的保护成员或共有成员直接写在私有派生类定义式中的同名段中，同时给成员名前冠以基类名和作用域标识符“::”。利用这种方法，该成员就成为派生类的保护成员或共有成员了。

```
class B:private A{
private:
    int y;
public:
    B(int x1, int y1) : A(x1) {
        y = y1;
    }
    A::show;                    //访问声明
};
```

访问声明在使用时应注意以下几点：

数据成员也可以使用访问声明。
访问声明中只含不带类型和参数的函数名或变量名。
访问声明不能改变成员在基类中的访问属性。
对于基类的重载函数名，访问声明将对基类中所有同名函数其起作用。
~

5.4 多继承
声明多继承派生类的一般形式如下：

```
class 派生类名:继承方式 1 基类名 1,...,继承方式 n 基类名 n {
    派生类新增的数据成员和成员函数
};
```

默认的继承方式是 private
多继承派生类的构造函数与析构函数：

与单继承派生类构造函数相同，多重继承派生类构造函数必须同时负责该派生类所有基类构造函数的调用。

多继承构造函数的调用顺序与单继承构造函数的调用顺序相同，也是遵循先调用基类的构造函数，再调用对象成员的构造函数，最后调用派生类构造函数的原则。析构函数的调用与之相反。

~

5.5 虚基类
虚基类的作用：如果一个类有多个直接基类，而这些直接基类又有一个共同的基类，则在最低层的派生类中会保留这个间接的共同基类数据成员的多份同名成员。在访问这些同名成员

时，必须在派生类对象名后增加直接基类名，使其唯一地标识一个成员，以免产生二义性。

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

class Base{
protected:
	int a;
public:
	Base(){
		a = 5;
		cout << "Base a = " << a << endl;
	}
};

class Base1: public Base{
public:
	Base1() {
		a = a + 10;
		cout << "Base1 a = " << a << endl;
	}
};

class Base2: public Base{
public:
	Base2() {
		a = a + 20;
		cout << "Base2 a = " << a << endl;
	}
};

class Derived: public Base1, public Base2{
public:
	Derived() {
		cout << "Base1::a = " << Base1::a << endl;
		cout << "Base2::a = " << Base2::a << endl;
	}
};

int main() {
	Derived obj;
	return 0;
}
```

代码执行结果如下

```
Base a = 5
Base1 a = 15
Base a = 5
Base2 a = 25
Base1::a = 15
Base2::a = 25
```

虚基类的声明：

不难理解，如果在上列中类 base 只存在一个拷贝(即只有一个数据成员 a)，那么对 a 的访问就不会产生二义性。在 C++中，可以通过将这个公共的基类声明为虚基类来解决这个问题。这就要求从类 base 派生新类时，使用关键字 virtual 将 base 声明为虚基类。

声明虚基类的语法形式如下：

```
class 派生类:virtual 继承方式 类名{
    .....
};
```

上述代码修改如下：

```
class Base1:virtual public Base{
public:
    Base1() {
        a = a + 10;
        cout << "Base1 a = " << a << endl;
    }
};

class Base2:virtual public Base{
public:
    Base2() {
        a = a + 20;
        cout << "Base2 a = " << a << endl;
    }
};
```

运行结果如下：

```
Base a = 5
Base1 a = 15
Base2 a = 35
Base1::a = 35
Base2::a = 35
1
```

虚基类的初始化：

虚基类的初始化与一般的多继承的初始化在语法上是一样的，但构造函数的调用顺序不同。在使用虚基类机制时应该注意以下几点：

如果在虚基类中定义有带形参的构造函数，并且没有定义默认形式的构造函数，则整个继承结构中，所有直接或间接的派生类都必须在构造函数的成员初始化列表中列出对虚基类构造函数的调用，以初始化在虚基类中定义的数据成员。
建立一个对象时，如果这个对象中含有从虚基类继承来的成员，则虚基类的成员是由最远派生类的构造函数通过调用虚基类的构造函数进行初始化的。该派生类的其他基类对虚基类构造函数的调用都被自动忽略。
若同一层次中同时包含虚基类和非虚基类，应先调用虚基类的构造函数，再调用非虚基类的构造函数，最后调用派生类构造函数。
对于多个虚基类，构造函数的执行顺序仍然是先左后右，自上而下。
若虚基类由非虚基类派生而来，则仍然先调用基类构造函数，再调用派生类的构造函数。
~

5.6 赋值兼容规则
在一定条件下，不同类型的数据之间可以进行类型转换，如可以将整型数据赋值给双精度型变量。在赋值之前，先把整型数据转换成双精度数据，然后再把它赋给双精度变量。这种不同数据类型之间的自动转换和赋值，称为赋值兼容。在基类和派生类对象之间也存有赋值兼容关系，基类和派生类对象之间的赋值兼容规则是指在需要基类对象的任何地方，都可以用子类的对象代替。

例如，下面声明的两个类：

```
class Base{
    .....
};
class Derived: public Base{
    .....
};
```

根据赋值兼容规则，在基类 Base 的对象可以使用的任何地方，都可以使用派生类 Derived 的对象来代替，但只能使用从基类继承来的成员。具体的表现在以下几个方面：

派生类对象可以赋值给基类对象，即用派生类对象中从基类继承来的数据成员，逐个赋值给基类对象的数据成员。

```
Base b;
Derived d;
b = d;
```

派生类对象可以初始化基类对象的引用。

```
Derived d;
```

Base &br = d;
派生类对象的地址可以赋值给指向基类对象的指针。

Derived d;
Base *bp = &d;
六、多态性与虚函数
多态性是面向对象程序设计的重要特征之一。多态性机制不仅增加了面向对象软件系统的灵活性，进一步减少了冗余信息，而且显著提高了软件的可重用性和可扩充性。多态性的应用可以使编程显得更简洁便利，它为程序的模块化设计又提供了一种手段。

~

6.1 多态性概述
所谓多态性就是不同对象收到相同的消息时，产生不同的动作。这样，就可以用同样的接口访问不同功能的函数，从而实现“一个接口，多种方法”。

从实现的角度来讲，多态可以划分为两类：编译时的多态和运行时的多态。在 C++中，多态的实现和连编这一概念有关。所谓连编就是把函数名与函数体的程序代码连接在一起的过程。静态连编就是在编译阶段完成的连编。编译时的多态是通过静态连编来实现的。静态连编时，系统用实参与形参进行匹配，对于同名的重载函数便根据参数上的差异进行区分，然后进行连编，从而实现了多态性。运行时的多态是用动态连编实现的。动态连编时运行阶段完成的，即当程序调用到某一函数名时，才去寻找和连接其程序代码，对面向对象程序设计而言，就是当对象接收到某一消息时，才去寻找和连接相应的方法。

一般而言，编译型语言（如 C，Pascal）采用静态连编，而解释型语言（如 LISP）采用动态连编。静态连编要求在程序编译时就知道调用函数的全部信息。因此，这种连编类型的函数调用速度快、效率高，但缺乏灵活性；而动态连编方式恰好相反，采用这种连编方式，一直要到程序运行时才能确定调用哪个函数，它降低了程序的运行效率，但增强了程序的灵活性。纯粹的面向对象程序语言由于其执行机制是消息传递，所以只能采用动态连编。C++实际上采用了静态连编和动态连编相结合的方式。

在 C++中，编译时多态性主要是通过函数重载和运算符重载实现的；运行时多态性主要是通过虚函数来实现的。

~

6.2 虚函数
虚函数的定义是在基类中进行的，它是在基类中需要定义为虚函数的成员函数的声明中冠以关键字 virtual，从而提供一种接口界面。定义虚函数的方法如下：

```
virtual 返回类型 函数名(形参表) {
    函数体
}
```
在基类中的某个成员函数被声明为虚函数后，此虚函数就可以在一个或多个派生类中被重新

定义。虚函数在派生类中重新定义时，其函数原型，包括返回类型、函数名、参数个数、参数类型的顺序，都必须与基类中的原型完全相同。

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

class Family{
private:
	string flower;
public:
	Family(string name = "鲜花"): flower(name) { }
	string getName() {
		return flower;
	}
	virtual void like() {
		cout << "家人喜欢不同的花: " << endl;
	}
};

class Mother: public Family{
public:
	Mother(string name = "月季"): Family(name) { }
	void like() {
		cout << "妈妈喜欢" << getName() << endl;
	}
};

class Daughter: public Family{
public:
	Daughter(string name = "百合"): Family(name) { }
	void like() {
		cout << "女儿喜欢" << getName() << endl;
	}
};

int main() {
	Family *p;
	Family f;
	Mother mom;
	Daughter dau;
	p = &f;
	p->like();
	p = &mom;
```

```
    p->like();
    p = &dau;
    p->like();

    return 0;
}
```
程序运行结果如下:

```
家人喜欢不同的花:
妈妈喜欢月季
女儿喜欢百合
1
2
3
```
C++规定，如果在派生类中，没有用 virtual 显式地给出虚函数声明，这时系统就会遵循以下的规则来判断一个成员函数是不是虚函数：该函数与基类的虚函数是否有相同的名称、参数个数以及对应的参数类型、返回类型或者满足赋值兼容的指针、引用型的返回类型。

下面对虚函数的定义做几点说明:

由于虚函数使用的基础是赋值兼容规则，而赋值兼容规则成立的前提条件是派生类从其基类公有派生。因此，通过定义虚函数来使用多态性机制时，派生类必须从它的基类公有派生。
必须首先在基类中定义虚函数;
在派生类对基类中声明的虚函数进行重新定义时，关键字 virtual 可以写也可以不写。
虽然使用对象名和点运算符的方式也可以调用虚函数，如 mom.like()可以调用虚函数 Mother::like()。但是，这种调用是在编译时进行的静态连编，它没有充分利用虚函数的特性，只有通过基类指针访问虚函数时才能获得运行时的多态性
一个虚函数无论被公有继承多少次，它仍然保持其虚函数的特性。
虚函数必须是其所在类的成员函数，而不能是友元函数，也不能是静态成员函数，因为虚函数调用要靠特定的对象来决定该激活哪个函数。
内联函数不能是虚函数，因为内联函数是不能在运行中动态确定其位置的。即使虚函数在类的内部定义，编译时仍将其看做非内联的。
构造函数不能是虚函数，但是析构函数可以是虚函数，而且通常说明为虚函数。
~

在一个派生类中重新定义基类的虚函数是函数重载的另一种形式。

多继承可以视为多个单继承的组合，因此，多继承情况下的虚函数调用与单继承下的虚函数调用由相似之处。

~

6.3 虚析构函数
如果在主函数中用 new 运算符建立一个派生类的无名对象和定义一个基类的对象指针，并

将无名对象的地址赋值给这个对象指针，当用 delete 运算符撤销无名对象时，系统只执行基类的析构函数，而不执行派生类的析构函数。

```
Base *p;
p = new Derived;
delete p;
-----------------
```

输出：调用基类 Base 的析构函数
原因是当撤销指针 p 所指的派生类的无名对象，而调用析构函数时，采用了静态连编方式，只调用了基类 Base 的析构函数。

如果希望程序执行动态连编方式，在用 delete 运算符撤销派生类的无名对象时，先调用派生类的析构函数，再调用基类的析构函数，可以将基类的析构函数声明为虚析构函数。一般格式为

```
virtual ~类名(){
    .....
}
```

虽然派生类的析构函数与基类的析构函数名字不相同，但是如果将基类的析构函数定义为虚函数，由该类所派生的所有派生类的析构函数也都自动成为虚函数。示例如下，

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

class Base{
public:
    virtual ~Base() {
        cout << "调用基类 Base 的析构函数..." << endl;
    }
};

class Derived: public Base{
public:
    ~Derived() {
        cout << "调用派生类 Derived 的析构函数..." << endl;
    }
};

int main() {
    Base *p;
    p = new Derived;
    delete p;
    return 0;
```

}
输出如下：

调用派生类 Derived 的析构函数...
调用基类 Base 的析构函数...
1
2
~

6.4 纯虚函数
纯虚函数是在声明虚函数时被“初始化为 0 的函数”，声明纯虚函数的一般形式如下：

virtual 函数类型 函数名(参数表) = 0;

声明为纯虚函数后，基类中就不再给出程序的实现部分。纯虚函数的作用是在基类中为其派生类保留一个函数的名字，以便派生类根据需要重新定义。

~

6.5 抽象类
如果一个类至少有一个纯虚函数，那么就称该类为抽象类，对于抽象类的使用有以下几点规定：

由于抽象类中至少包含一个没有定义功能的纯虚函数。因此，抽象类只能作为其他类的基类来使用，不能建立抽象类对象。
不允许从具体类派生出抽象类。所谓具体类，就是不包含纯虚函数的普通类。
抽象类不能用作函数的参数类型、函数的返回类型或是显式转换的类型。
可以声明指向抽象类的指针或引用，此指针可以指向它的派生类，进而实现多态性。
如果派生类中没有定义纯虚函数的实现，而派生类中只是继承基类的纯虚函数，则这个派生类仍然是一个抽象类。如果派生类中给出了基类纯虚函数的实现，则该派生类就不再是抽象类了，它是一个可以建立对象的具体类了。
~

6.6 示例：利用多态计算面积
应用 C++的多态性，计算三角形、矩形和圆的面积。

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

/*** 定义一个公共基类 ***/
class Figure{
protected:
    double x, y;
public:
```

```
    Figure(double a, double b): x(a), y(b) {    }
    virtual void getArea()          //虚函数
    {
        cout << "No area computation defind for this class.\n";
    }
};

class Triangle: public Figure{
public:
    Triangle(double a, double b): Figure(a, b){    }
    //虚函数重定义，用于求三角形的面积
    void getArea(){
        cout << "Triangle with height " << x << " and base " << y;
        cout << " has an area of " << x * y * 0.5 << endl;
    }
};

class Square: public Figure{
public:
    Square(double a, double b): Figure(a, b){    }
    //虚函数重定义，用于求矩形的面积
    void getArea(){
        cout << "Square with dimension " << x << " and " << y;
        cout << " has an area of " << x * y << endl;
    }
};

class Circle: public Figure{
public:
    Circle(double a): Figure(a, a){    }
    //虚函数重定义，用于求圆的面积
    void getArea(){
        cout << "Circle with radius " << x ;
        cout << " has an area of " << x * x * 3.14 << endl;
    }
};

int main(){
    Figure *p;
    Triangle t(10.0, 6.0);
    Square s(10.0, 6.0);
    Circle c(10.0);

    p = &t;
```

```
    p->getArea();
    p = &s;
    p->getArea();
    p = &c;
    p->getArea();

    return 0;
}
```

程序输出如下：

```
Triangle with height 10 and base 6 has an area of 30
Square with dimension 10 and 6 has an area of 60
Circle with radius 10 has an area of 314
```

七、运算符重载

运算符重载是面向对象程序设计的重要特征。

~

7.1 运算符重载概述

运算符重载是对已有的运算符赋予多重含义，使同一个运算符作用于不同类型的数据导致不同的行为。

下面的案例实现+号运算符重载：

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

class Complex{
private:
    double real, imag;
public:
    Complex(double r = 0.0, double i = 0.0): real(r), imag(i) { }
    friend Complex operator+(Complex& a, Complex& b) {
        Complex temp;
        temp.real = a.real + b.real;
        temp.imag = a.imag + b.imag;
        return temp;
    }
    void display() {
        cout << real;
        if (imag > 0) cout << "+";
        if (imag != 0) cout << imag << "i" << endl;
    }
};
```

```
int main()
{
	Complex a(2.3, 4.6), b(3.6, 2.8), c;
	a.display();
	b.display();
	c = a + b;
	c.display();
	c = operator+(a, b);
	c.display();

	return 0;
}
```
程序输出结果如下：

```
2.3+4.6i
3.6+2.8i
5.9+7.4i
5.9+7.4i
~
```

这一章偷个懒😁

八、函数模板与类模板

利用模板机制可以显著减少冗余信息，能大幅度地节约程序代码，进一步提高面向对象程序的可重用性和可维护性。模板是实现代码重用机制的一种工具，它可以实现类型参数化，即把类型定义为参数，从而实现代码的重用，使得一段程序可以用于处理多种不同类型的对象，大幅度地提高程序设计的效率。

~

8.1 模板的概念

在程序设计中往往存在这样的现象：两个或多个函数的函数体完全相同，差别仅在与它们的参数类型不同。

例如：

```
int Max(int x, int y) {
	return x >= y ? x : y;
}

double Max(double x, double y) {
	return x >= y ? x : y;
}
```
能否为上述这些函数只写出一套代码呢？解决这个问题的一种方式是使用宏定义

```
#define Max(x, y)((x >= y) ? x : y)
```
1
宏定义带来的另一个问题是，可能在不该替换的地方进行了替换，而造成错误。事实上，由于宏定义会造成不少麻烦，所以在 C++中不主张使用宏定义。解决以上问题的另一个方法就是使用模板。

~

8.2 函数模板

所谓函数模板，实际上是建立一个通用函数，其函数返回类型和形参类型不具体指定，用一个虚拟的类型来代表，这个通用函数就称为函数模板。在调用函数时，系统会根据实参的类型（模板实参）来取代模板中的虚拟类型，从而实现不同函数的功能。

函数的声明格式如下

```
template <typename 类型参数>
返回类型 函数名(模板形参表)
{
    函数体
}
```
也可以定义为如下形式
```
template <class 类型参数>
返回类型 函数名(模板形参表)
{
    函数体
}
```
实际上，template 是一个声明模板的关键字，它表示声明一个模板。类型参数（通常用 C++标识符表示，如 T、type 等）实际上是一个虚拟的类型名，使用前并未指定它是哪一种具体的类型，但使用函数模板时，必须将类型实例化。类型参数前需加关键字 typename 或 class，typename 和 class 的作用相同，都是表示一个虚拟的类型名（即类型参数）。

例 1：一个与指针有关的函数模板

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

template <typename T>
T Max(T *array, int size = 0) {
    T max = array[0];
    for (int i = 1  ; i < size; i++) {
        if (array[i] > max) max = array[i];
```

```
    }
    return max;
}

int main() {
    int array_int[] = {783, 78, 234, 34, 90, 1};
    double array_double[] = {99.02, 21.9, 23.90, 12.89, 1.09, 34.9};
    int imax = Max(array_int, 6);
    double dmax = Max(array_double, 6);
    cout << "整型数组的最大值是：" << imax << endl;
    cout << "双精度型数组的最大值是：" << dmax << endl;
    return 0;
}
```

21

例 2：函数模板的重载

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

template <class Type>
Type Max(Type x, Type y) {
    return x > y ? x : y;
}

template <class Type>
Type Max(Type x, Type y, Type z) {
    Type t = x > y ? x : y;
    t = t > z ? t : z;
    return t;
}

int main() {
    cout << "33,66 中最大值为 " << Max(33, 66) << endl;
    cout << "33,66,44 中最大值为 " << Max(33, 66, 44) << endl;
    return 0;
}
```

注意：

在函数模板中允许使用多个类型参数。但是，应当注意 template 定义部分的每个类型参数前必须有关键字 typename 或 class。

在 template 语句与函数模板定义语句之间不允许插入别的语句。

同一般函数一样，函数模板也可以重载。

函数模板与同名的非模板函数可以重载。在这种情况下，调用的顺序是：首先寻找一个参数完全匹配的非模板函数，如果找到了就调用它；若没有找到，则寻找函数模板，将其实例化，

产生一个匹配的模板参数，若找到了，就调用它。
~

8.3 类模板
所谓类模板，实际上就是建立一个通用类，其数据成员、成员函数的返回类型和形参类型不具体指定，用一个虚拟的类型来代表。使用类模板定义对象时，系统会根据实参的类型来取代类模板中虚拟类型，从而实现不同类的功能。

```
template <typename T>
class Three{
private:
    T x, y, z;
public:
    Three(T a, T b, T c) {
        x = a; y = b; z = c;
    }
    T sum() {
        return x + y + z;
    }
}
```

上面的例子中，成员函数（其中含有类型参数）是定义在类体内的。但是，类模板中的成员函数也可以在类模板体外定义。此时，若成员函数中有类型参数存在，则 C++有一些特殊的规定：

需要在成员函数定义之前进行模板声明；
在成员函数名前要加上“类名<类型参数>::”；
在类模板体外定义的成员函数的一般形式如下：

```
template <typename  类型参数>
函数类型  类名<类型参数>::成员函数名(形参表)
{
    .....
}
```

例如，上例中成员函数 sum()在类外定义时，应该写成

```
template<typename T>
T Three<T>::sum() {
    return x + y + z;
}
```

例子：栈类模板的使用

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

```
const int size = 10;
template <class T>
class Stack{
private:
	T stack[size];
	int top;
public:
	void init() {
		top = 0;
	}
	void push(T t);
	T pop();
};

template <class T>
void Stack<T>::push(T t) {
	if (top == size) {
		cout << "Stack is full!" << endl;
		return;
	}
	stack[top++] = t;
}

template <class T>
T Stack<T>::pop() {
	if (top == 0) {
		cout << "Stack is empty!" <<endl;
		return 0;
	}
	return stack[--top];
}

int main() {
	Stack<string> st;
	st.init();
	st.push("aaa");
	st.push("bbb");
	cout << st.pop() << endl;
	cout << st.pop() << endl;

	return 0;
}
```

九、C++的输入和输出

~

9.1 C++为何建立自己的输入/输出系统
C++除了完全支持 C 语言的输入输出系统外，还定义了一套面向对象的输入/输出系统。C++的输入输出系统比 C 语言更安全、可靠。

c++的输入/输出系统明显地优于 C 语言的输入/输出系统。首先，它是类型安全的、可以防止格式控制符与输入输出数据的类型不一致的错误。另外，C++可以通过重载运算符“>>”和"<<"，使之能用于用户自定义类型的输入和输出，并且向预定义类型一样有效方便。C++的输入/输出的书写形式也很简单、清晰，这使程序代码具有更好的可读性。

~

9.2 C++的流库及其基本结构
“流”指的是数据从一个源流到一个目的的抽象，它负责在数据的生产者（源）和数据的消费者（目的）之间建立联系，并管理数据的流动。凡是数据从一个地方传输到另一个地方的操作都是流的操作，从流中提取数据称为输入操作（通常又称提取操作），向流中添加数据称为输出操作（通常又称插入操作）。

C++的输入/输出是以字节流的形式实现的。在输入操作中，字节流从输入设备（如键盘、磁盘、网络连接等）流向内存；在输出操作中，字节流从内存流向输出设备（如显示器、打印机、网络连接等）。字节流可以是 ASCII 码、二进制形式的数据、图形/图像、音频/视频等信息。文件和字符串也可以看成有序的字节流，分别称为文件流和字符串流。

~

用于输入/输出的头文件
C++编译系统提供了用于输入/输出的 I/O 类流库。I/O 流类库提供了数百种输入/输出功能，I/O 流类库中各种类的声明被放在相应的头文件中，用户在程序中用#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。常用的头文件有：

iostream 包含了对输入/输出流进行操作所需的基本信息。使用 cin、cout 等流对象进行针对标准设备的 I/O 操作时，须包含此头文件。
fstream 用于用户管理文件的 I/O 操作。使用文件流对象进行针对磁盘文件的操作，须包含此头文件。
strstream 用于字符串流的 I/O 操作。使用字符串流对象进行针对内存字符串空间的 I/O 操作，须包含此头文件。
iomanip 用于输入/输出的格式控制。在使用 setw、fixed 等大多数操作符进行格式控制时，须包含此头文件。
用于输入/输出的流类
I/O 流类库中包含了许多用于输入/输出操作的类。其中，类 istream 支持流输入操作，类 ostream 支持流输出操作，类 iostream 同时支持流输入和输出操作。

下表列出了 iostream 流类库中常用的流类，以及指出了这些流类在哪个头文件中声明。

类名　　类名　　说明　　头文件
抽象流基类　ios　流基类　iostream
输入流类　　istream　通用输入流类和其他输入流的基类　iostream
输入流类　　ifstream　输入文件流类　　fstream
输入流类　　istrstream　　输入字符串流类　strstream
输出流类　　ostream　通用输出流类和其他输出流的基类　iostream
输出流类　　ofstream 输出文件流类　　fstream
输出流类　　ostrstream　　输出字符串流类　strstream
输入/输出流类　　iostream 通用输入输出流类和其他输入/输出流的基类　iostream
输入/输出流类　　fstream　输入/输出文件流类　　fstream
输入/输出流类　　strstream输入/输出字符串流类　strstream
~

9.3 预定义的流对象
用流定义的对象称为流对象。与输入设备（如键盘）相关联的流对象称为输入流对象；与输出设备（如屏幕）相联系的流对象称为输出流对象。

C++中包含几个预定义的流对象，它们是标准输入流对象 cin、标准输出流对象 cout、非缓冲型的标准出错流对象 cerr 和缓冲型的标准出错流对象 clog。

~

9.4 输入/输出流的成员函数
使用 istream 和类 ostream 流对象的一些成员函数，实现字符的输出和输入。

1、put()函数
　　cout.put(单字符/字符形变量/ASCII 码);
2、get()函数
　　get()函数在读入数据时可包括空白符，而提取运算符“>>”在默认情况下拒绝接收空白字符。
　　cin.get(字符型变量)
3、getline()函数
　　cin.getline(字符数组，字符个数 n，终止标志字符)
　　cin.getline(字符指针，字符个数 n，终止标志字符)
4、ignore()函数
　　cin.ignore(n，终止字符)
　　ignore()函数的功能是跳过输入流中 n 个字符（默认个数为 1），或在遇到指定的终止字符(默认终止字符是 EOF)时提前结束。
~

9.5 预定义类型输入/输出的格式控制
在很多情况下，需要对预定义类型（如 int、float、double 型等）的数据的输入/输出格式进行控制。在 C++中，仍然可以使用 C 中的 printf()和 scanf()函数进行格式化。除此之外，C++

还提供了两种进行格式控制的方法：一种是使用 ios 类中有关格式控制的流成员函数进行格式控制；另一种是使用称为操作符的特殊类型的函数进行格式控制。

1、用流成员函数进行输入/输出格式控制

设置状态标志的流成员函数 setf()
清除状态标志的流成员函数 unsetf()
设置域宽的流成员函数 width()
设置实数的精度流成员函数 precision()
填充字符的流成员函数 fill()
2、使用预定义的操作符进行输入/输出格式控制

3、使用用户自定义的操作符进行输入/输出格式控制

若为输出流定义操作符函数，则定义形式如下：

```
ostream &操作符名(ostream &stream)
{
    自定义代码
    return stream;
}
```

若为输入流定义操作符函数，则定义形式如下：

```
istream &操作符名(istream &stream)
{
    自定义代码
    return stream;
}
```

例如，

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

ostream &output(ostream &stream)
{
    stream.setf(ios::left);
    stream << setw(10) << hex << setfill('-');
    return stream;
}

int main() {
    cout << 123 << endl;
    cout << output << 123 << endl;
    return 0;
```

}
输出结果如下:

123
7b--------
~

9.6 文件的输入/输出
所谓文件，一般指存放在外部介质上的数据的集合。

文件流是以外存文件为输入/输出对象的数据流。输出文件流是从内存流向外存文件的数据，输入文件流是从外存流向内存的数据。

根据文件中数据的组织形式，文件可分为两类：文本文件和二进制文件。

在 C++中进行文件操作的一般步骤如下:

为要进行操作的文件定义一个流对象。
建立（或打开）文件。如果文件不存在，则建立该文件。如果磁盘上已存在该文件，则打开它。
进行读写操作。在建立（或打开）的文件基础上执行所要求的输入/输出操作。
关闭文件。当完成输入/输出操作时，应把已打开的文件关闭。
~

9.7 文件的打开与关闭
为了执行文件的输入/输出，C++提供了 3 个文件流类。

类名　　说明　　功能
istream　输入文件流类　　用于文件的输入
ofstream 输出文件流类　　用于文件的输出
fstream　输入/输出文件流类　　用于文件的输入/输出
这 3 个文件流类都定义在头文件 fstream 中。

要执行文件的输入/输出，须完成以下几件工作:

在程序中包含头文件 fstream。
建立流对象
使用成员函数 open()打开文件。
进行读写操作。
使用 close()函数将打开的文件关闭。
~

9.8 文本文件的读/写
例子：把字符串“I am a student.”写入磁盘文件 text.txt 中。

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

int main() {
	ofstream fout("../test.txt", ios::out);
	if (!fout) {
		cout << "Cannot open output file." << endl;
		exit(1);
	}
	fout << "I am a student.";
	fout.close();

	return 0;
}
```
例子：把磁盘文件 test1.dat 中的内容读出并显示在屏幕上。

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

int main() {
	ifstream fin("../test.txt", ios::in);
	if (!fin) {
		cout << "Cannot open output file." << endl;
		exit(1);
	}
	char str[80];
	fin.getline(str , 80);
	cout << str <<endl;
	fin.close();

	return 0;
}
~
```

9.9 二进制文件的读写

用 get()函数和 put()函数读/写二进制文件

例子：将 a~z 的 26 个英文字母写入文件，而后从该文件中读出并显示出来。

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

```
int cput() {
	ofstream outf("test.txt", ios::binary);
	if (!outf) {
		cout << "Cannot open output file.\n";
		exit(1);
	}
	char ch = 'a';
	for (int i = 0; i < 26; i++) {
		outf.put(ch);
		ch++;
	}
	outf.close();
	return 0;
}

int cget() {
	fstream inf("test.txt", ios::binary);
	if (!inf) {
		cout << "Cannot open input file.\n";
		exit(1);
	}
	char ch;
	while (inf.get(ch)) {
		cout << ch;
	}
	inf.close();
	return 0;
}

int main() {
	cput();
	cget();    //此处文件打不开，不知为什么。。。

	return 0;
}
```

用 read()函数和 write()函数读写二进制文件

有时需要读写一组数据（如一个结构变量的值），为此 C++提供了两个函数 read()和 write()，用来读写一个数据块，这两个函数最常用的调用格式如下：

```
inf.read(char *buf, int len);
outf.write(const char *buf, int len);
```

例子：将两门课程的课程名和成绩以二进制形式存放在磁盘文件中。

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

struct list{
	char course[15];
	int score;
};

int main() {
	list ob[2] = {"Computer", 90, "History", 99};
	ofstream out("test.txt", ios::binary);
	if (!out) {
		cout << "Cannot open output file.\n";
		abort();	//退出程序，作用与 exit 相同。
	}
	for (int i = 0; i < 2; i++) {
		out.write((char*) &ob[i], sizeof(ob[i]));
	}
	out.close();

	return 0;
}
```

例子：将上述例子以二进制形式存放在磁盘文件中的数据读入内存。

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

struct list{
	char course[15];
	int score;
};

int main() {
	list ob[2];
	ifstream in("test.txt", ios::binary);
	if (!in) {
		cout << "Cannot open input file.\n";
		abort();
	}
	for (int i = 0; i < 2; i++) {
		in.read((char *) &ob[i], sizeof(ob[i]));
		cout << ob[i].course << " " << ob[i].score << endl;
```

```
    }
    in.close();

    return 0;
}
```

检测文件结束

在文件结束的地方有一个标志位，即为 EOF。采用文件流方式读取文件时，使用成员函数 eof()可以检测到这个结束符。如果该函数的返回值非零，表示到达文件尾。返回值为零表示未达到文件尾。该函数的原型是：

```
int eof();
```

函数 eof()的用法示例如下：

```
ifstream ifs;
.
if (!ifs.eof())     //尚未到达文件尾
    ...
```

还有一个检测方法就是检查该流对象是否为零，为零表示文件结束。

```
ifstream ifs;
.
if (!ifs)
    ...
```

如下例子：

```
while (cin.get(ch))
    cut.put(ch);
```

这是一个很通用的方法，就是检测文件流对象的某些成员函数的返回值是否为 0，为 0 表示该流（亦即对应的文件）到达了末尾。

从键盘上输入字符时，其结束符是 Ctrl+Z，也就是说，按下【Ctrl+Z】组合键，eof()函数返回的值为真。

十、异常处理和命名空间

10.1 异常处理

程序中常见的错位分为两大类：编译时错误和运行时错误。编译时的错误主要是语法错误，如关键字拼写错误、语句末尾缺分号、括号不匹配等。运行时出现的错误统称为异常，对异常的处理称为异常处理。

C++处理异常的办法：如果在执行一个函数的过程中出现异常，可以不在本函数中立即处理，而是发出一个信息，传给它的上一级（即调用函数）来解决，如果上一级函数也不能处理，就再传给其上一级，由其上一级处理。如此逐级上传，如果到最高一级还无法处理，运行系统一般会自动调用系统函数 terminate()，由它调用 abort 终止程序。

例子：输入三角形的三条边长，求三角形的面积。当输入边的长度小于 0 时，或者当三条边都大于 0 时但不能构成三角形时，分别抛出异常，结束程序运行。

```
#include <iostream>
#include <cmath>
```

```
using namespace std;

double triangle(double a, double b, double c) {
	double s = (a + b + c) / 2;
	if (a + b <= c || a + c <= b || b + c <= a) {
		throw 1.0;		//语句 throw 抛出 double 异常
	}
	return sqrt(s * (s - a) * (s - b) * (s - c));
}

int main() {
	double a, b, c;
	try {
		cout << "请输入三角形的三个边长（a, b, c）: " << endl;
		cin >> a >> b >> c;
		if (a < 0 || b < 0 || c < 0) {
			throw 1;	//语句 throw 抛出 int 异常
		}
		while (a > 0 && b > 0 && c > 0) {
			cout << "a = " << a << " b = " << b << " c = " << c << endl;
			cout << "三角形的面积  = " << triangle(a, b, c) << endl;
			cin >> a >> b >> c;
			if (a <= 0 || b <= 0 || c <= 0) {
				throw 1;
			}
		}
	} catch (double) {
		cout << "这三条边不能构成三角形..." << endl;
	} catch (int) {
		cout << "边长小于或等于 0..." << endl;
	}
	return 0;
}
~
```

10.2 命名空间和头文件命名规则

命名空间：一个由程序设计者命名的内存区域。程序设计者可以根据需要指定一些有名字的命名空间，将各命名空间中声明的标识符与该命名空间标识符建立关联，保证不同命名空间的同名标识符不发生冲突。

1.带扩展名的头文件的使用

在 C 语言程序中头文件包括扩展名.h，使用规则如下面例子

```
#include <stdio.h>
```

2.不带扩展名的头文件的使用

C++标准要求系统提供的头文件不包括扩展名.h，如 string，string.h 等。

```
#include <cstring>
```

十一、STL 标准模板库

标准模板库（Standard Template Library）中包含了很多实用的组件，利用这些组件，程序员编程方便而高效。

11.1 Vector

vector 容器与数组类似，包含一组地址连续的存储单元。对 vector 容器可以进行很多操作，包括查询、插入、删除等常见操作。

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

int main() {
	vector<int> nums;
	nums.insert(nums.begin(), 99);
	nums.insert(nums.begin(), 34);
	nums.insert(nums.end(), 1000);
	nums.push_back(669);

	cout << "\n 当前 nums 中元素为: " << endl;
	for (int i = 0; i < nums.size(); i++)
		cout << nums[i] << " " ;

	cout << nums.at(2);
	nums.erase(nums.begin());
	nums.pop_back();

	cout << "\n 当前 nums 中元素为: " << endl;
	for (int i = 0; i < nums.size(); i++)
		cout << nums[i] << " " ;

	return 0;
}
~
```

11.2 list 容器

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

int main() {
	list<int> number;
```

```
    list<int>::iterator niter;
    number.push_back(123);
    number.push_back(234);
    number.push_back(345);

    cout << "链表内容：" << endl;
    for (niter = number.begin(); niter != number.end(); ++niter)
        cout << *niter << endl;
    number.reverse();
    cout << "逆转后的链表内容：" << endl;
    for (niter = number.begin(); niter != number.end(); ++niter)
        cout << *niter << endl;
    number.reverse();

    return 0;
}
~
```

11.3 stack

例子：利用栈进行进制转换

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

int main() {
    stack<int> st;
    int num = 100;
    cout << "100 的八进制表示为：";
    while (num) {
        st.push(num % 8);
        num /= 8;
    }
    int t;
    while (!st.empty()) {
        t = st.top();
        cout << t;
        st.pop();
    }
    cout << endl;

    return 0;
}
~
```

11.4 queue

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

int main() {
	queue<int> qu;
	for (int i = 0; i < 10; i++)
		qu.push(i * 3 + i);
	while (!qu.empty()) {
		cout << qu.front() << " ";
		qu.pop();
	}
	cout << endl;

	return 0;
}
~
```

11.5 优先队列 priority_queue

```
#include <iostream>
#include <queue>
#include <functional>
#include <cstdlib>
#include <ctime>
using namespace std;

int main() {
	priority_queue<int> pq;
	srand((unsigned)time(0));
	for (int i = 0; i < 6; i++) {
		int t = rand();
		cout << t << endl;
		pq.push(t);
	}
	cout << "优先队列的值： " << endl;
	for (int i = 0; i < 6; i++) {
		cout << pq.top() << endl;
		pq.pop();
	}

	return 0;
}
```

~

11.6 双端队列 deque
push_back();
push_front();
insert();
pop_back();
pop_front();
erase();
begin();
end();
rbegin();
rend();
size();
maxsize();
~

11.7 set
```
#include <iostream>
#include <set>
#include <string>
using namespace std;

int main() {
	set<string> s;
	s.insert("aaa");
	s.insert("bbb");
	s.insert("ccc");
	if (s.count("aaa") != 0) {
		cout << "存在元素 aaa" << endl;
	}
	set<string>::iterator iter;
	for (iter = s.begin(); iter != s.end(); ++iter)
		cout << *iter << endl;

	return 0;
}
```
~

11.8 map
```
#include <iostream>
#include <map>
#include <string>
using namespace std;
```

```
int main() {
	map<string, int> m;
	m["aaa"] = 111;
	m["bbb"] = 222;
	m["ccc"] = 333;
	if (m.count("aaa")) {
		cout << "键 aaa 对应的值为" << m.at("aaa") << endl;
		cout << "键 aaa 对应的值为" << m["aaa"] << endl;
	}

	return 0;
}
```

完结撒花~~~🌸🌸🌸🌸🌸
