C++语言程序设计

《C++语言程序设计》

作者：郑莉 董渊 何江舟

清华大学出版社，2019年出版

第四章 类和对象

4.1 面向对象程序设计的基本特点

- 抽象：是指对具体问题进行概括，抽象出一类对象的公共属性并加以描述的过程。
  - 数据抽象
  - 行为抽象
- 封装：将抽象到的数据和行为相结合，形成一个有机的整体。
- 继承：在保持原有类特性的基础上，进行更详细的说明。
- 多态：一段程序能处理多种类型对象的能力。
  - 多态
  - 重载多态
  - 类型参数化多态
  - 包含多态

4.2 类和对象

4.2.1 类的定义

在面向对象程序设计过程中，程序模块是由类构成的。类是对逻辑上相关的函数与数据的封装，它是对问题的抽象描述。其语法形式如下：

```
class  类名
{
public:
    外部接口
protected:
    保护型成员
private:
    私有成员
};
```

4.2.1 类成员的访问控制

- 共有类型：定义了外部接口。
- 私有类型：只能被**类内成员函数**访问，类外部任何访问都是**非法的。**
- 保护类型：和私有成员性质类似，差别在于继承过程中对**新类**的影响。

> 数据成员应该声明为私有，在书写时公有成员写在最前面。

4.2.3 对象

声明对象

```
类名 对象名
```

> 对象所占的内存空间只用于存储**数据成员**，函数成不在每个对象中存放副本，只占据一份空间。

访问数据成员

```
对象名.数据成员
```

访问函数成员

```
对象名.函数成员
```

4.2.4 类的成员函数

1.成员函数的实现
函数的原型声明要写在类体中，原型说明了函数的参数表和返回值类型。而函数的具体实现是写在类定义之外的。与普通函数不同的是，实现成员函数时要指明类的名称，具体形式为：

```
返回值类型 类名：：函数成员名（参数表）
{
    函数体
}
```

2.成员函数调用中的目的对象
调用一个成员函数与调用普通函数的差异在于，需要使用“.”操作符指出调用所针对的对象，这一对象在本次调用中称为目的对象。

3.带默认形参值的成员函数
类成员函数的默认值，一定要写在**类定义中**，而不能写在类定义之外的函数实现中。

4.内联函数成员
内联函数的声明有两种方式：隐式声明和显式声明。

- 将函数体直接放在类体内，这种方法称之为隐式声明。
- 为了保证类定义的简洁，可以采用关键字**inline**显式声明的方式。

4.3 构造函数与析构函数

在定义对象的时候进行的数据成员设置，称为对象的初始化。C++程序中的初始化和清理工作，分别由两个特殊的成员函数来完成，他们就是构造函数和析构函数。

4.3.1 构造函数

构造函数的作用就是在对象被创建时利用特定的值构造对象，将对象初始化为一个特定的状态。

```
class 类名{
public:
    类名();
};
```

构造函数也是一个类的成员函数，除了具有一般成员函数的特征之外，还有以下特殊的性质：

- 构造函数的函数名与类名相同。
- 没有返回值
- 其通常被声明为公有函数。

只要类中有构造函数，编译器就会在建立新对象的地方自动插入对构造函数调用的代码，即构造函数在对象被创建的时候将被**自动调用**。构造函数可以是直接访问所有的数据成员，可以是内联函数，可以带有参数表，可以带默认值，可以重载。

无需提供参数的构造函数称为默认构造函数，如果没写构造函数，系统默认生成一个隐含的构造函数，参数列表和函数体均为空。 如果定义了构造函数，编译器不会生成隐含的默认构造函数。

4.3.2 复制构造函数

复制构造函数是一种特殊的构造函数，具有一般构造函数的所有特性，其形参是本类对象的引用，其作用是使用一个已经存在的对象（由复制构造函数的参数指定），去初始化同类的一个新对象。语法形式如下：

```
class 类名{
public:
    类名();
    类名(类名 &对象);
};
```

如果没有定义类的构造函数，系统会自动生成一个隐含的复制构造函数，这个隐含的复制构造函数的作用是把初始值对象的每一个数据成员的值都复制到新建立的对象。

复制构造函数在以下情况会被默认调用

```
//函数形参是类的对象
void fun1(Point p)
{
    cout << p.getX() << endl;
}
//函数返回值是类的对象
Point fun2()
{
    Point a(1, 2);//会消亡
    return a;
}
```

- 用类来初始化该类的另一个对象时

```
Point a(4, 5);

// 类的一个对象初始化另一个对象
Point b(a);
Point c = a;
cout << b.getX() << "\t" << c.getX() << endl;//4       4
```

- 函数的形参是类的对象

```
//函数形参是类的对象
fun1(b); //4
```

- 函数的返回值是类的对象

```
//函数返回值是类的对象
b = fun2();
cout << b.getX() << endl;//1
```

4.3.3 析构函数

析构函数用来完成对象被删除前的一些清理工作。析构函数是在对象生存期即将结束时的时刻被**自动调用**的。析构函数通常也是类的一个公有函数成员，它的名称是由类名前面加“~”构成，没有返回值，且不接收任何参数。其语法形式如下：

```
class 类名{
public:
    ~类名();
};
```

不进行显示说明，系统自动生成一个函数体为空的而隐含析构函数，如果人们希望对象被删除之前时刻西东完成某些事情，就可以将他们把他们写进析构函数中。

4.4 类的组合

4.4.1 组合

类的组合描述即使一个类内嵌其他类的对象做为成员。当创建类的时候，如果这个类具有内嵌对象成员，那么这个内嵌对象成员将首先被自动创建。创建对象时，既要对本类的基本数据类型进行初始化，也要对内嵌对象成员进行初始化。其定义形式：

```
类名::类名(形参表):内嵌对象1(形参表),内嵌对象2(形参表)...
{
    类的初始化
}
```

> 其中 内嵌对象1(形参表),内嵌对象2(形参表)... 被称为初始化列表，可以对内嵌对象初始化

构造函数的调用顺序如下：

调用内嵌对象的构造函数，调用顺序按照**内嵌对象在组合类中的定义中出现的次序。**内嵌对象在构造函数的初始化列表中出现的顺序与内嵌对象构造函数的调用顺序**无关。**执行本类构造函数的函数体。析构函数的执行顺序与构造函数刚好相反

4.6 结构体和联合体

4.6.1 结构体

结构体是一种特殊形态的类。结构体和类的唯一区别在于，结构体和类具有不同的默认访问控制属性：在类中，对于未指定访问控制属性的成员，其访问控制属性为**私有类型**；在结构体中，对于未指定访问控制属性的成员，其访问控制属性为**公有类型。**

4.6.2 联合体

联合体是一种特殊形态的类。它可以有自己的数据成员和函数成员，与结构体一样也是从C语言继承过来的。联合体的全部数据成员**共享同一组内存单元。**

第五章 数据的共享与保护

5.1 标识符的作用域与可见性

5.1.1 作用域

作用域是一个标识符在程序正文中的有效的区域，C++中标识符的作用域：**函数原型作用域、局部作用域（块作用域）、类作用域、文件作用域、命名空间作用域**

1. 函数原型的作用域
   在函数原型声明时形式参数的作用范围就是函数原型作用域。
2. 局部作用域（块作用域）
   函数体内声明的变量，其作用域从声明处开始，一直到声明所在块结束的大括号为止。
3. 类作用域
   类可以被看成一组由名成员的集合，类X的成员具有类作用域，对m的访问有如下3种方式：

- 如果在X的成员函数中没有声明同名的局部作用域标识符，那么在该函数内可以直接访问成员m
- 访问对象成员 X.m 或 X::m（访问类中静态成员）
- ptr->m（ptr为指向对象的一个指针）

4. 命名空间作用域
   为了避免重名冲突，使编译器能够区分来自不同库的同名实体，将不同的标识符集合在一个命名作用域（named space）内，是全局作用域的细分，本质上定义了实体所属的空间，一个命名空间确定了一个命名空间作用域。语法形式如下：

```
namespace namespace_name{
    //代码声明
}
```

使用某个命名空间中的函数、变量等实体，需要如下方法：

```
命名空间::实体名称
```

为了解决命名空间显得过于冗长，用using来指定命名空间：

```
using 命名空间名::标志符名;
using namespace 命名空间名;
```

前一种形式将指定的标识符暴露在当前的作用域内，使得在当前作用域中可以直接引用该标识符；后一种形式将指定命名空间内的所有标识符暴露在当前的作用域内。

具有命名空间作用域的变量也称为全局变量。

5.1.2可见性

从标识符引用的角度来看标识符的有效范围，即标识符是否可以被引用。表示从内层作用域向外层作用域“看”时能看见什么，如果标识符在某处可见，则就可以在该处引用此标识符。可见性的规则如下：

- 标识符要先声明，后使用
- 在同一作用域中，不能声明同名的标识符（重载函数除外）
- 在没有相互包含关系的不同的作用域中声明的同名标识符，互不影响
- 如果在两个或多个具有包含关系的作用域中声明了同名标识符，则外层标识符在内层不可见
- 如果某个标识符在外层中声明，且在内层中没有同一标识符的声明，则该标识符在内层中可见

5.2对象的生存期

对象从产生到结束的这段时间就是它的生存期，在对象的生存期内，对象将保持它的值（数据成员的值），直到被更新为止。

5.2.1静态生存期

静态生存期：如果对象的生存期与程序运行的生存期相同，我们就称它具有静态生存期，如果要在函数内部的局部作用域中声明具有静态生存期的对象，则要使用**static**关键字。

局部作用域中静态变量地特点是，它并不会随着每次函数调用而产生一个副本，也不会随着函数返回而失效。

> 定义时未指定初值的基本类型静态生存期变量，会被赋予0值初始化，而对于动态生存期变量，不指定初值意味着初值不确定。另，若定义数组时未初始化，初值也不确定；若定义数组时初值个数少于数组大小，其余自动为0。

5.2.2动态生存期

除了以上两种情况，其余的对象都具有动态生存期，称为局部生存期对象。局部生存期对象诞生于声明点，结束语声明所在的块执行完毕之时。

5.3 类的静态成员

5.3.1 静态数据成员

如果某个属性为整个类所共有，不属于任何一个对象，则采用**static**关键字来声明为静态成员。静态成员在每个类只有一个副本，由该类的所有对象共同维护和使用，从而实现了同一类的不同对象之间的数据共享。

静态数据成员具有静态生存期。由于静态数据成员不属于任何一个对象，因此可以通过类名对它进行访问，一般的用法是**“类名::标识符”。**在类的定义中只对静态数据成员进行引用性声明，然后在命名空间作用域中必须使用类名进行定义性声明。之所以类的静态数据成员需要在类定义之外再加以定义，是因为需要以这种方式专门为它们分配空间。

5.3.2 静态函数成员

就是使用**static**关键字声明的函数成员，性质与静态数据成员一样。静态成员函数可以直接访问该类的**静态数据**和**函数成员**，而访问非静态成员必须通过对象名。因此通过静态函数成员访问非静态成员是相当麻烦的，一般情况下，它主要用来访问同一个类中的静态数据成员，维护对象之间共享的数据。

5.4 类的友元

友元关系提供了不同类或对象的成员函数之间、类的成员函数与一般函数之间进行数据共享的机制。通俗来说，友元关系就是一个类主动声明哪些其他类或函数是它的朋友，进而给它们提供对本类的访问特许。从一定程度上来讲，友元是对数据隐蔽和封装的破坏。

友元类的所有成员函数都自动成为友元函数。

5.4.1 友元函数

友元函数是在类中用关键字**friend**修饰的非成员函数。友元函数可以是一个普通的函数，也可以是其他类的成员函数。虽然它不是本类的成员函数，但是**在它的函数体中可以通过对象名访问类的私有和保护成员。**

5.4.2 友元类

若A类为B类的友元类，则A类的所有成员函数都是B类的友元函数，都可以访问B类的私有和保护成员。声明的语法形式为：

```
class B{
    ...
    friend class A;//声明A为B的友元类
    ...
};
```

关于友元，还有几点需要注意：

- 友元关系是不能传递的。
- 友元关系是单向的。
- 友元关系是不能继承的。

5.5 共享数据的保护

对于既需要共享又需要防止改变的数据应该声明为常量。常量的重要特性：必须初始化，而且不能被更新。

5.5.1 常对象

它的数据成员值在对象的整个生存期间不能被改变，常对象必须初始化，而且不能被更新。其语法形式如下：

```
const 类型说明符 对象名;
```

两个途径：

- 直接访问：通过对象名访问数据成员。常对象的数据成员被视为常量，通过常对象访问数据成员不允许被复制
- 间接访问：通过成员函数改变数据成员。不能通过常对象调用普通成员函数，常对象只能调用常成员函数

5.5.2 用const修饰的类成员

1. 常函数
   不会改变目的对象的数据成员的值；声明格式如下：

```
类型说明符 函数名(参数表) const
```

注意事项

- const是函数类型的一个组成部分，因此在函数的定义部分也要带**const**关键字
- 如果将一个对象说明为常对象，则通过该常对象只能调用它的**常成员函数**，而不能调用其他成员函数
- 无论是否通过常对象调用常成员函数，在常成员函数调用期间，目的对象都被视同为常对象，因此常成员函数不能更改目的对象的数据成员，也不能针对目的对象调用该类中没有用const修饰的成员函数（这就保证了在常成员函数中不会更改目的对象的数据成员的值）。
- const关键字可以用于对重载函数的区分

> 如果仅以const关键字为区分对成员函数重载，那么通过非const的对象调用该函数，两个重载的函数都可以与之匹配，这时编译器将选择最近的函数——不带const关键字的函数。
> 对于无须改变对象状态的成员函数，都应当使用**const**

2. 常数据成员
   类的成员数据可以使常量，使用**const**说明，如果在一个类中声明了常数据成员，那么任何函数都不能对该成员赋值，构造函数对常数据成员的初始化，只能通过构造函数的**初始化列表**。

5.5.3 常引用

引用的对象不能更新，且自动化为常对象，声明形式如下：

```
const 类型说明符 &引用名;
```

常引用做形参：为了追求高效率，且不双向传递。对于在函数中无需改变其值的参数，不宜使用普通引用方式传递，因为那会使得常对象无法被传入，采用传值方式或传递常引用的方式可避免这一问题，复制构造函数的参数一般也宜采用**常引用传递**。

第六章 数组、指针与字符串

6.1 数组

数组是具有一定顺序关系的若干对象的集合体，组成数组的对象称为该数组的元素。

6.1.1 数组的声明与使用

1. 数组的声明
   声明一个数组类型，应包括一下几个方面：

- 数组的名称
- 数组元素的类型
- 数组的结构(数组维数，每一维的大小等)

数组类型声明的一般形式：

```
数据类型 标识符[常量表达式1][常量表达式2]...;
```

数组中同类型数据的类型由**数据类型**给出，这个**数据类型**可以是整形、浮点型等基本类型，也可以是结构体、类等用户自定义类型。**标识符**指定数组名称。**[常量表达式1] [常量表达式2]...**称为数组的界，必须是在编译时就可求出的常量表达式，其值必须为正整数。有n个下标的数组称为n维数组。数组的下标的个数称为数组的维数。二维数组被当作一维数组的数组，更高维数组也如此。

> 数组元素个数是各个下标表达式的乘积

2. 数组的使用

使用数组时只能分别对数组的各个元素操作，数组元素通过下标区分，使用形式如下:

```
数组名[下标表达式1][下标表达式2]...
```

在使用过程中应注意一下两点：

- 数组元素下标可以是任何合法的算术表达式，但其结果必须是**整数**。
- 数组元素下标值不得超过所声明的上下界，否则运行时会出现数组越界错误。

6.1.2 数组的存储与初始化

1. 数组的存储
   数组元素在内存中是顺序、连续存储的。
2. 数组的初始化
   数组元素的初始化就是在声明数组时对部分或全部元素赋初值。
   若定义数组时没有指定任何一个元素的初值，对于静态生存期的数组，每个元素仍然会被赋值为0，但对于动态生存期的数组，每个元素的初值都是不确定的。

> 给出全部的元素，则第一维的下标个数可以不用显式说明，也可以按照第一位下标进行分组，使用花括号将每一组的数据括起来。

```
int a[2][3]={1,0,0,0,1,0};

//等价于
int a[][3]={1,0,0,0,1,0};

int a[2][3]={{1,0,0},{0,1,0}};

int a[][3]={{1,0,0},{0,1,0}};
```

6.1.3 数组作为函数参数

使用数组名传递参数时，传递的是**地址**，实参数组的个数不应少于形参数组的个数，如果在被调函数中对形参数组元素值进行改变，主调函数中实参数组的相应元素值也会改变。

> 数组作为参数时，一般不指定第一维的大小，即使指定，也会被忽略。

6.1.4 对象数组

对象数组的元素是类的**对象**，具有数据成员和函数成员。声明一个一维对象数组形式：

```
类名 数组名[常量表达式];
```

通过对象数组中元素对象访问公有成员，形式如下：

```
数组名[下标表达式].成员名
```

数组对象的初始化过程，实际上就是调用构造函数来对每一个元素对象进行初始化。如果在声明数组时给每一个数组元素指定初始值，在数组初始化过程中就会调用与形参类型相匹配的构造函数。

6.2 指针

指针是C++从C中继承过来的重要数据类型，提供了一种较为直接的地址操作手段。动态内存分配和管理离不开指针。

6.2.1 内存空间的访问方式

具有静态生存期的变量在程序开始运行之前就被分配了内存空间。具有动态生存期的变量是在程序运行时，遇到变量声明语句时被分配内存空间的。有时使用变量名访问内存空间不够方便，或者根本没有变量名可用，这时就需要直接用地址来访问内存单元。

6.2.2 指针变量的声明

指针是一种数据类型。指针变量用于存放内存单元地址。声明指针的语法形式：

```
数据类型 *标识符;
```

指针可以指向各种类型，包括基本类型、数组（数据元素）、函数、对象、指针。

6.2.3 与地址相关的运算“*”和“&”

“*”称为指针运算符，也称解析，表示获取指针所指的变量的值，是一元操作符。

“&”称为取地址运算符，用于得到一个对象的（存储单元）地址，是一元操作符。

“*”和“&”出现在声明语句中和执行语句中，其含义是不同的。作为一元运算符和二元运算符时含义也不同。

- （1）在声明语句中的一元运算符“*”和“&”

一元运算符“*”出现在声明语句中，在被声明的变量之前时，表示声明的是指针。

一元运算符“&”出现在声明语句中，在被声明的变量之前时，表示声明的是引用。

- （2）在非声明语句中的一元运算符“*”和“&”

一元运算符“*”出现在执行语句中，或在声明语句的初始化表达式中时，表示访问指针所指对象的内容。

一元运算符“&”出现在执行语句中，或在给变量赋值时出现在等号右边时，表示取对象的地址。

6.2.4 指针的赋值

定义指针后必须先赋值，然后才能引用。两种赋值方法：

（1）定义指针的同时进行初始化赋值：

```
存储类型 数据类型 *指针名=初始地址;     //定义指针的同时进行初始化赋值
```

（2）定义指针后，单独使用赋值语句：

```
指针名=地址;      //定义指针后，单独使用的赋值语句
```

如果使用对象的地址作为指针的初值，或在赋值语句中将对象地址赋给指针变量，该对象必须在赋值之前就声明过，而且这个对象的类型应该和指针类型一致。多个指针可指向同一个变量。

一个数组，可以用它的名称来直接表示它的起始地址。**数组名实际上就是一个不能被赋值的指针，即指针常量。**如：

```
int a[10];       //定义int型数组
int *ptr=a;      //定义并初始化int指针
```

关于指针的类型，还应注意：

（1）可以声明指向常量的指针，此时所指对象的值不变，指针本身可变（可指向另外的对象）：

```
int a;
const int *p1=&a;    //p1是常量指针
```

（2）可以声明指针类型的常量（指针常量），这时指针本身（的值，为地址）不能被改变：

```
int *const p2=&a;   //p2是指针常量
```

（3）一般情况下，指针的值只能赋给相同类型的指针。但void类型的指针，可以存储任何类型的对象地址，任何类型的指针都可以赋值给void类型的指针变量。经过使用类型显示转换，通过void类型的指针便可以访问任何类型的数据。void指针一般只在指针所指向的数据类型不确定时使用。

6.2.5 指针运算

指针与整数加减、指针自增自减都表示移动指针所指位置。

(p1+n1)可写作p1[n1]，都表示p1当前所指位置后方第n1个数的内容。类似也有(p1-n1)，可写作p1[-n1]。

慎用指针的算术运算：对指针进行算术运算时，一定要确保运算结果所指向的地址是程序中分配使用的地址。

两个相同类型的指针相等，表示指向的是同一地址。

不同类型的指针之间或指针与非0整数之间的关系运算是无意义的。

但指针变量可以和整数0作比较，0专用于表示**空指针**，即一个不指向任何有效地址的指针。给一个指针变量赋值为0，表示该指针是一个空指针，不指向任何地址。

```
int *p;     //声明一个int类型的指针p
p=0;        //将p设置为空指针，不指向任何地址
```

除0以外，赋给指针变量的值不能是其他整数，必须是地址常量（如数组名）或地址变量。空指针也可以用NULL来表示。如果不便于用一个有效地址给一个指针变量赋初值，那么应当用0作为它的初值，从而避免指向不确定地址的指针出现。

```
int *p=NULL;    //将int型的指针初始化为空指针
```

> NULL是一个在很多头文件中都有定义的宏，被定义为0。

6.2.6 用指针处理数组元素

把数组作为函数的形参，等价于把指向数组元素类型的指针作为形参。例如，下面三种写法出现在形参列表中是等价的：

```
void f(int p[]);
void f(int p[3]);
void f(int *p);
```

> a[i] *(pa+i) *(a+i) pa[i]都是等价的

6.2.7 指针数组

指针数组：如果一个数组的每个元素都是指针变量，这个数组就是指针数组。指针数组的每一个元素都必须是同一类型的指针。声明一维指针数组的语法形式如下：

```
数据类型 *数组名[下标表达式];   //声明一维指针数组，指针数组的每个元素都是指针变量
```

数组名是指针数组的名称，同时也是这个数组的首地址。由于指针数组的每个元素都是一个指针，必须先赋值后引用，因此声明指针数组必须赋初值。

6.2.8 用数组作为函数参数

如果以指针作为函数的形参，在调用时实参将值传递给形参，也就是使实参和形参指针变量指向同一个内存地址。这样在子函数运行过程中，通过形参指针对数据值的改变会影响实参指针所指向的数据值。

在C语言中，用指针作为函数的形参有三个作用：

1. 使形参和实参指向相同的内存空间，以达到参数双向传递的目的，即通过在被调函数中直接处理主调函数中的数据，来将函数的处理结果返回其调用者。C++中用引用实现这一功能（见第3章）。
2. 减少函数调用时数据传递的开销。C++中有时可用引用实现这一功能，有时还是需要用指针。
3. 通过指向函数的指针传递函数代码的首地址。

> 注意：如果函数体中不需要通过指针改变指针所指对象的内容，应在参数表中将其声明为指向常量的指针，这样使得常对象被取地址后也可作为该函数的参数。

在设计程序时，当某个函数中以指针或引用作为形参都可以达到同样的目的时，使用引用可以使程序的可读性更好些。

6.2.9 指针型函数

指针可以作为函数的返回值。当一个函数的返回值为指针类型时，这个函数就是指针型函数。最主要的目的是在函数结束时**把大量的数据从被调函数返回到主调函数中**。而通常非指针型函数调用结束后，只能返回一个变量或对象。指针型函数的定义如下：

```
数据类型 *函数名(参数表){      //定义指针型函数
    函数体                    //返回值为指针类型
}
```

6.2.10 指向函数的指针（函数指针）

函数指针是专门用来存放函数代码首地址的变量。调用函数的通常形式**“函数名(参数表)”**的实质就是**“函数代码首地址(参数表)”**。一旦函数指针指向了某个函数，它与函数名便具有同样的作用。可以像使用函数名一样使用指向函数的指针来调用函数。函数指针的声明如下：

```
数据类型 (*函数指针名)(形参表)  //声明函数指针
```

数据类型说明指针所指函数的返回值类型，第一个圆括号中的内容知名一个函数指针的名称；形参表则列出该指针所指函数的形参类型和个数。

由于对函数指针的定义在形式上比较复杂，如果在程序中出现多个这样的定义，多次重复这样的定义相当繁琐，可以使用**typedef**来进行重命名

```
typedef int(*DoubleIntFunction)(double);
DoubleIntFunction funcPtr;
```

函数指针在使用前要先赋值，使指针指向一个已经存在的函数代码的起始地址，语法形式如下：

```
函数指针名=函数名;
```

6.2.11 对象指针

1. 对象指针的一般概念
   对象指针是用于存放对象地址的变量。对象所占据的内存空间只是用于存放数据成员的，函数成员不在每一个对象中存储副本。对象指针声明形式如下：

```
类名 *对象指针名;
```

可以通过对象指针访问对象的成员，语法形式为：

```
对象指针名->成员名   //通过对象指针访问对象的成员
(*对象指针名).成员名      //上一行的等价形式
```

2. this指针
   this指针是一个隐含于每一个**类的非静态成员函数（包括构造函数和析构函数）**中的特殊指针。类的静态成员函数中没有this指针。

this指针用于指向正在被成员函数操作的对象。

this指针实际上是类成员函数的一个隐含参数。在调用类的成员函数时，目的对象的地址会自动作为this指针的值，传递给被调用的成员函数，这样被调函数就能够通过this指针来访问目的对象的数据成员。

this是一个指针常量，对于常成员函数，this同时又是一个指向常量的指针。

在成员函数中，可以使用**this**来标识正在调用该函数的对象。

当局部作用域中声明了与类成员同名的标识符时，对该标识符的直接引用代表的是局部作用域中所声明的标识符，这时，为了访问该类的成员，可以通过this指针（比如this->x）。

3. 指向类的非静态成员的指针
   可将类的成员（变量、函数、对象等）的地址存放到一个指针变量中，这样，可以通过这些指针直接访问对象的成员。
   指向类的成员的指针声明形式如下：

```
类型说明符 类名::*指针名; //声明指向数据成员的指针
类型说明符 (类名::*指针名)(参数表);  //声明指向函数成员的指针
```

对指向类的成员的指针赋值：

```
指针名=&类名::数据成员名;      //对数据成员指针赋值
指针名=&类名::函数成员名;      //对函数成员指针赋值
```

通过指向类的成员的指针（以及对象指针）访问成员：

```
对象名.*类数据成员指针名                          //访问类的数据成员
对象指针名->*类数据成员指针名                       //访问类的数据成员

(对象名.*类成员函数指针名)(参数表)                 //访问类的函数成员
(对象指针名->*类成员函数指针名)(参数表)             //访问类的函数成员
```

4. 指向类的静态成员的指针
   对类的静态成员的访问是不依赖于对象的。因此，可以用普通的指针来指向和访问静态成员。

6.3 动态内存分配

动态内存分配可以使程序在运行过程中**按照实际需要申请适量的内存**，使用结束后还可以**释放。**申请和释放过程一般称为**建立**和**删除**，使用**new**和**delete**操作符

运算符new的功能是动态分配内存，或者动态的创建堆对象，语法形式如下：

```
new 数据类型 (初始化参数列表)
```

如果内存申请成功，new运算便返回一个指向新分配内存首地址的类型的指针，可以通过这个指针访问堆对象；如果申请失败，会抛出异常

> 在建立一个类的对象时，如果该类存在用户定义的默认构造函数，则"new T"和 "new T()"这两种写法的效果都是相同的。

运算符delete的功能是删除由new建立的对象，释放指针所指向的内存空间，语法形式如下：

```
delete 指针名字
```

如果被删除的是对象，该对象的析构函数被调用，对于用new分配的内存，必须用delete加以释放，否则会导致动态分配的内存无法回收，使得程序占据的内存越来越大，这叫做**“内存泄露”**。

也可以创建数组类型的对象，语法形式为：

```
new 类型名 [数组长度]
```

用new建立的数组，用delete删除的时候，指针名前要加上[]，语法形式为：

```
delete[] 指针名字
```

动态生成一个整形数组，代码如下：

```
int *p=new int[10];
delete[] p;
```

用new也可以创建多维数组，形式如下：

```
new 类型名T[数组第一维][数组第二维]...
```

其中第一维可以使任何结果的正整数的表达式，而其他各维数组长度必须是结果为正整数的常量表达式。如果申请成功返回一个纸箱新分配内存首地址的指针，但不是**T类型**，而是一个指向T类型数组的指针，数组元素个数为除最左边一维外各维的下标表达式的乘积。

```
float *fp = new float[10][25][10];//error

float (*fp)[25][10]=new float[10][25][10];//Ok
```

6.4 用vector创建数组对象

无论是静态数组，还是用new创建的动态数组，都难以检测下标越界的额错误。C++标准库提供了被封装的动态数组——**vector**，这种被封装的数组可以具有各种类型。**vector**是一个类模板，不是类。用vector定义动态数组的形式：

```
vector<元素类型>数组对象名(数组长度);    //用vector定义动态数组
```

用vector定义的数组对象的所有元素都会被初始化。另外，初值也可以自行指定，但只能为所有元素指定相同的初值：

```
vector<元素类型>数组对象名(数组长度,元素初值);
```

访问vector数组对象的元素：

```
数组对象名[下标表达式]      //访问vector数组对象的元素
```

6.5 深复制与浅复制

隐含的复制构造函数完成的只是浅复制。默认的复制构造函数将两个对象的对应数据简单复制后，也就是说两个指针指向的是同一内存地址，表面上好像完成了复制，但是并没有形成真正的副本。

浅复制还有更大的弊病，在程序结束之前对象的析构函数会自动被调用，动态分配的内存空间会被释放。由于两个对象共用了同一块内存空间，因此两次调用析构函数，将该内存空间释放了两次，于是导致运行错误。

解决“浅复制”问题的方法是编写复制构造函数，实现“深复制”。

6.6 字符串

C语言中用字符型数组来存放字符串，C++中依然可以沿用这一方法。不仅如此，C++标准库还预定义了string类。

6.6.1 用字符数组存储和处理字符串

字符串常量是用一对双引号括起来的字符序列。在内存中按串中字符的排列次序顺序存放，每个字符占一个字节，并在末尾添加空字符（null character）‘\0’作为结尾标记。这实际上是一个隐含创建的类型为char的数组，一个字符串常量就表示这样一个数组的首地址。因此，可以把字符串常量赋给字符串指针，由于常量值是不能修改的，所以应将字符串常量赋给指向常量的指针。

如果创建一个char数组，每个元素存放字符串的一个字符，在末尾放置一个‘\0’，便构成了C++字符串。它的存储方式与字符串常量无异，但由于是程序员创建的数组，因此可以改变其内容，因而这就是字符串变量了。

对字符数组进行初始化赋值时，初值的形式可以是以逗号分隔的ASCII码或字符常量，也可以是整体的字符串常量（这时末尾的‘\0’是隐含的）。下面三种创建字符串变量的写法等价：

字符常量用单引号括起；字符串常量用双引号括起，用双引号括起的字符串隐式地包含结尾的空字符。字符常量（如‘S’）是字符串编码的简写表示，而“S”不是字符常量，它表示字符‘S’和‘\0’组成的字符串。

然而，用字符数组表示字符串后，执行很多字符串操作比较麻烦，需要借助**cstring**头文件中的字符串处理函数，比如**strcpy**函数（将一个字符串的内容复制到另一个字符串）、**strcat**函数（将两个字符串连接起来）、**strcmp**函数（按辞典顺序比较两个字符串的大小）。另外，当字符串长度很不确定时，需要用**new**来动态创建字符数组，最后还要用**delete**释放，这些都相当繁琐。

C++对这些繁琐的操作进行了封装，形成了string类，可以更加方便地操作字符串。

6.6.2 string类

使用string类必须包含头文件string。string定义在命名空间std中。

严格地说，string并非一个独立的类，而是类模板basic_string的一个特定化实例。不过对使用者来说，其特点与一个类无异。

1. 构造函数的原型

```
string();//默认构造函数，建立一个长度为0的串
string(const string& rhs);//复制构造函数
string(const char *s);//用指针s所指的字符串常量初始化string类的对象
string(const string& rhs,unsigned int pos,unsigned int n);//将对象rhs中的串从位置pos开始n个字符，用来初始化string类的对象
string(const char *s,unsigned int n);//用指针s所指的字符串中的前n个字符初始化string类的对象
string(unsigned int n,char c);//将参数c中的字符重复n次，用来初始化string类的对象
```

2. string类的操作符
3. 常用成员函数功能简介

第七章：继承与派生

7.1 继承与派生

7.1.1 继承关系举例

保持已有特性而构造新类的过程称为**继承**；在已有类的基础上新增自己的特性而产生新类的过程称为**派生**；被继承的已有类称为**基类（或父类）**；派生出的新类称为**派生类（或子类）**。

- 继承的目的：实现代码重用
- 派生的目的：当新的问题出现，原有程序无法解决（或不能完全解决）时，需要对原有程序进行改造；

7.1.2 派生类的定义

在C++中，派生类的一般语法形式为：

```
class 派生类名:继承方式 基类1,继承方式 基类2，继承方式 基类3...
{
    派生类说明;
}
```

> 一个派生类，可以同时有多个基类，称为**多继承**，一个类只有一个直接基类，成为**单继承**。
>
> 在类组中，直接参与派生出某些类的基类称为**直接基类**，基类的基类或者更高层的基类称为**间接基类**。

7.1.3 派生类生成过程

- 吸收基类成员
  - 派生类就包含了所有基类中**除了构造和析构函数之外**的所有成员
- 改造基类成员
- 添加新的成员

7.2 访问控制

7.2.1 公有继承

- 当类的继承方式是共有继承时，基类的公有成员和保护成员的访问属性在派生类中不变，而类的私有成员则不可直接访问。
- 基类的共有成员和保护成员被继承到派生类中访问属性不变，仍作为派生类的共有成员和保护成员，派生类的其他成员可以直接访问。

- 在类外只能通过派生类的对象访问基类的公有成员，无论是派生类的成员还是基类的对象都无法直接访问基类的私有成员。

7.2.2 私有继承

- 当类的继承方式是私有继承时，基类的公有成员和保护成员都以为私有成员的身份出现在派生类中，而基类的私有成员在派生类中不可访问
- 派生类的共有成员和保护成员被继承后都作为私有成员，派生类的其他成员可以直接访问，但是类外通过派生类的对象无法直接访问
- 私有继承后，基类的成员无法在以后的派生类中发挥作用

7.2.3 保护继承

- 当类的继承方式是保护继承时，基类的公有成员和保护成员都以为私有成员的身份出现在派生类中，而类的私有成员不可直接访问
- 派生类的其他成员函数可以直接访问基类继承来的共有和保护成员，但在类外通过派生类的对象无法访问

继承方式	基类	派生类
共有继承	共有成员、保护成员	共有成员、保护成员
私有继承	共有成员、保护成员	私有成员
保护继承	共有成员、保护成员	保护成员

7.3 类兼容性规则

```
class B{}
class D:public B{}

B b1,*pb1;
D d1;
```

在需要基类的任何地方，都可以使用共有派生类的对象类替代；具体表现在：

1. 派生类的对象可以被赋值给基类对象；

```
b1=d1;
```

2. 派生类的对象可以初始化基类的引用；

```
B &rb=d1;
```

3. 指向基类的指针也可以指向派生类；

```
pb1=&d1c
```

在替代之后，派生类对象可以作为基类的对象使用，但只能使用从基类继承的成员。

7.4 派生类的构造和析构

7.4.1 构造函数

1. 基类无默认构造函数，基类也无。
2. 构造派生类对象时，要对基类的成员对象和新增的成员对对象进行初始化。
3. 基类的构造函数**没有被继承过来**，由于基类成员大部分无法访问，必须调用基类的构造函数，派生类的构造函数要有参数给基类的构造函数。
4. 如果基类初始化时需要调用基类带有参数表的构造函数，派生类必须声明构造函数。
   派生类构造函数的额一般语法形式为：

```
派生类名::派生类名(参数表):基类1(基类1的初始化参数表),...,基类n(基类n的初始化参数表),成员对象1(成员对象1的初始化参数表),...,成员对象m(成员对象m的初始化参数表)
{
    派生类构造函数的其他初始化操作;
}
```

派生类构造函数的执行一般次序如下

1. 调用基类构造函数，调用次序按照他们**继承时被声明的顺序**(左->右)。
2. 对派生类的新增成员进行初始化，调用顺序是他们**在类中声明的顺序。**
3. 执行派生类中的函数体内容。

7.4.2 复制构造函数

一个类如果没有编写复制构造函数，编译系统会自动的生成一个隐含的复制构造函数，并自动的调用基类的复制构造函数，然后对派生类新增对象一一执行复制。

7.4.3 析构函数

在类的派生过程中，基类的析构函数**也不能继承下来**，需要虚构的话，要在派生类中声明**新的析构函数**，这些析构函数的调用过程与构造函数的调用次序完全相反。

7.5 派生类成员的标识与访问

在派生类中，成员按照属性划分为以下4种：

- 不可访问的成员
- 私有成员
- 保护成员
- 共有成员

7.5.1 作用域分辨符

作用域分辨符 就是:: 它可以用来限定要访问的成员所在的类名称，一般形式为：

```
类名::成员名
类名::成员函数名
```

在不同的作用域声明的标识符，可见性原则【两个作用域互相包含】

- 外作用域声明了一个标识符，内作用域没有声明同名的标识符，则外作用域标识符仍然可见
- 外作用域声明了一个标识符，内作用域声明了同名的标识符，则外作用域标识符不可见

类的派生层次

- 派生类声明了一个和基类同名的新成员，派生的新成员隐藏了外层同名函数，直接使用成员名只能访问到派生类的新成员
- 派生类声明了一个和基类同名的新函数，即使函数的参数表不同，从基类继承的同名函数所有重载形式也会被隐藏

如果某个派生类的多个基类拥有同名的成员，同时，派生类有新增这样的同名函数，派生类成员将隐藏所有基类的同名成员。

如果派生类不添加新成员，则主函数中对象名.成员名会*报错**。

如果某个派生类的部分或全部直接基类都是从另一个共同的基类派生而来的，在这些直接基类中，从上一级继承来的成员拥有相同的名称，因此派生类中也就会产生同名的现象，对这种类型的同名函数也要使用作用域分辨符来唯一标识，而且必须用直接类来标识。

7.5.2 虚基类

可以将共同基类设为虚基类，这时从不同的路径继承过来的同名数据成员就在内存中只有一个副本，同一个函数名也只有一个映射，其语法形式如下：

```
class 派生类名:virtual 继承方式 基类名
```

7.5.3 虚基类及其派生类构造函数

如果虚基类声明有非默认的构造函数，并且没有声明默认形式的构造函数，在整个继承关系中，直接或间接继承虚基类的所有派生类，都必须通过构造函数的成员初始化列表对虚基类初始化

构建一个类的对象的一般顺序：

1. 如果该类有直接或者间接的虚基类，则先执行虚基类的构造函数。
2. 如果该类有其他基类，则按照它们在继承声明的列表中出现的次序，分别执行它们的构造函数，但构造过程中，不在执行它们基类的构造函数。
3. 按照在类定义中出现的顺序，对派生类中新增的成员对象进行初始化。对于类类型的成员对象，如果出现在构造函数初始化列表中，则以其中指定的参数执行构造函数。如果未出现，则执行默认构造函数；对于基本数据类型的成员对象，如果出现在构造函数的初始化列表中，则使用其中指定的值为其赋初值，否则什么也不做。
4. 执行构造函数的函数体。

第八章 多态性

多态是指同样的消息（对类的成员函数的调用）被**不同类型的对象**接收时导致**不同的行为**（不同的实现，也就是调用了不同的函数）。比如，使用同样的加号“+”，就可以实现整型数之间、浮点数之间、双精度浮点数之间的加法，以及这几种数据类型混合的加法运算。

多态就是将函数名称到函数入口地址（一个函数在内存中起始的地址就称为这个函数的入口地址）的运行期绑定机制。

多态的四种类型：**强制多态、重载多态、包含多态、类型参数化多态。**

> 又可分为：特殊多态性（只是表面的多态性）（包括强制多态、重载）和一般多态性（真正的多态性）（包括包含多态、类型参数化多态）。

- 强制多态：通过将一种类型的数据转换成另一种类型的数据来实现，也就是数据类型转换（隐式或显式）（比如前述的加法运算符在进行浮点数与整型数相加时，首先进行类型强制转换，把整型数变为浮点数再相加的情况）。
- 重载：指给同一个名字赋予不同的含义（普通函数和类的成员函数的重载、运算符重载）。
- 包含多态：C++中采用虚函数实现包含多态。
- 类型参数化多态：C++中采用模板实现类型参数化多态，包括函数模板和类模板。

多态从实现角度可分为两类：**编译时的多态、运行时的多态。**

- 编译时的多态：在编译过程中确定了同名操作的具体操作对象。
- 运行时的多态：在程序运行过程中才动态地确定操作所针对的具体对象（需要满足的三个条件见8.3.1节）。这种确定操作的具体对象的过程就是绑定。

绑定：指计算机程序自身彼此关联的过程，也就是把一个标识符名和另一个存储地址联系在一起的过程。用面向对象的术语讲，就是把一条消息（对类的成员的调用）和一个对象的方法相结合的过程。

按照绑定进行的阶段的不同，可分为两种绑定方法：**静态绑定、动态绑定。**

- 静态绑定（早期绑定，前绑定、编译期绑定）：绑定工作在编译连接阶段完成的情况。比如*强制、重载、参数多态**中操作对象的确定。
- 动态绑定（晚期绑定，后绑定、运行期绑定）：绑定工作在程序运行阶段完成的情况。比如包含多态中操作对象的确定。

8.2 运算符重载

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时导致**不同的行为。**运算符重载的实质是**函数重载。**在实现过程中，首先把指定的运算表达式转化为对运算符函数的调用，将运算对象转化为运算符函数的实参，然后根据实参的类型来确定需要调用的函数，这个过程是在编译过程中完成的。

8.2.1 运算符重载的规则

运算符重载的规则：

（1）C++中的运算符除了少数几个之外，全部可以重载，而且只能重载C++中已有的运算符，不能定义新运算符；

（2）重载后运算符优先级、结合性不变；

（3）运算符重载是针对新类型数据的实际需要，对原有运算符进行适当的改造。一般来说，重载的功能应当与原有功能相似，不能改变原运算符的操作对象个数，同时，至少要有一个操作对象是自定义类型。运算符重载，操作对象个数、结合性、优先级、语法结构均不变。

不能重载的运算符：

1. 类属关系（成员访问）运算符 **“.“**
2. 成员指针访问运算符 **“.*“**
3. 作用域分辨符 **“::“**
4. 三目运算符**“? :“**
5. 长度运算符**“sizeof“**。

运算符重载的两种形式：重载为类的**非静态成员函数**，重载为**非成员函数。**

运算符重载的一般语法形式：

```
返回类型 operator 运算符(形参表){      //运算符重载
    函数体
}
```

其中，返回类型即运算结果类型，operator为定义运算符重载函数的**关键字**，运算符是可重载的运算符名称，形参表中给出重载运算符所需要的参数和类型。

> 当以**非成员函数**形式重载运算符时，有时需要访问运算符参数所涉及的类的私有成员，这时可以把该函数声明为类的**友元函数。**

> 当运算符重载为类的**非静态成员函数**时，函数的参数个数比原来的操作数要**少一个**（后置“++”，“--”除外）。因为运算符重载为类的非静态成员函数时，第一个操作数会被作为函数调用的目的对象，因而无须出现在参数表中，函数体中可以直接访问第一个操作数的成员。
>
> 当运算符重载为非成员函数时，函数的参数个数与原操作数**个数相同**。因为运算符重载为非成员函数时，运算符的所有操作数必须显式地通过参数传递。

8.2.2 运算符重载为成员函数

运算符重载为类的非静态成员函数后，总是通过该类的某个**对象**来访问重载的运算符。

- 如果是双目运算符，左操作数是**对象本身**的数据，由this指针给出，右操作数则需要通过运算符重载函数的参数来传递。
- 如果是单目运算符，操作数由**对象的this指针**给出，就**不再需要任何参数**。

对于双目运算符B，如果要重载为类的成员函数，用来实现表达式oprd1 B oprd2，其中oprd1为A类的对象，则应当把运算符B重载为A类的成员函数，该函数只有一个形参，该形参的类型是oprd2所属的类型。重载之后，表达式oprd1 B oprd2就相当于函数调用oprd1.operator B(oprd2)。

对于前置单目运算符U，如果要重载为类的成员函数，用来实现表达式U oprd，其中oprd为A类的对象，则应当把运算符U重载为A类的成员函数，该函数没有形参。重载之后，表达式U oprd相当于函数调用oprd.oprerator U()。

> 对于后置运算符“++”和“--”，如果要重载为类的成员函数，用来实现表达式oprd++或oprd--，其中oprd为A类的对象，则应当把运算符++、--重载为A类的成员函数，这时函数要带有一个整型（int）形参。重载之后，表达式oprd++和oprd--就相当于函数调用oprd.operator++(0)和oprd.operator--(0)。这里的int类型参数在运算中不起任何作用，只是用于区别后置++、--和前置++、--（对于函数参数表中并未使用的参数，C++允许不给出参数名）。

8.2.3 运算符重载为非成员函数

运算符重载为非成员函数，运算符所需要的操作数都需要通过函数的形参表来表达（**参数个数不变**），在形参表中形参从左到右的顺序就是运算符操作数的顺序。

如果需要访问运算符参数对象的成员，可以将该函数声明为类的友元函数。

> 不要机械地将重载运算符的非成员函数声明为类的友元函数，应仅在需要访问类的私有成员或保护成员时再这样做：（1）如果不将其声明为友元函数，该函数仅依赖于类的接口，只要类的接口不变化，该函数的实现就无需变化；（2）如果将其声明为友元函数，该函数会依赖于类的实现，即使类的接口不变化，只要类的私有数据成员的设置发生了变换，该函数的实现就需要变化。

对于双目运算符B，如果要实现oprd1 B oprd2，其中oprd1和oprd2中只要有一个为自定义类型，就可以将运算符B重载为非成员函数，函数的形参为oprd1和oprd2。重载之后，表达式oprd1 B oprd2就相当于函数调用operator B(oprd1,oprd2)。

对于前置单目运算符U，如果要实现表达式U oprd，其中oprd为自定义类型，就可以将运算符U重载为非成员函数，函数的形参为oprd。重载之后，表达式U oprd相当于函数调用oprerator U(oprd)。

> 对于后置运算符“++”和“--”，如果要实现表达式oprd++或oprd--，其中oprd为自定义类型，就可以将运算符++、--重载为非成员函数。这时函数要有两个形参，一个是oprd，另一个是整型（int）形参。重载之后，表达式oprd++和oprd--就相当于函数调用operator++(oprd,0)和operator--(oprd,0)。这里的**int**参数个数不变类型参数在运算中不起任何作用，只是用于区别后置++、--和前置++、--（对于函数参数表中并未使用的参数，C++允许不给出参数名）

> “<<”操作符必须重载为**友元函数**

运算符重载形式的选择：运算符的两种重载形式各有优势。成员函数的重载方式更加方便，但有时出于以下原因，需要使用非成员函数的重载方式：

（1）要重载的运算符的第一个操作数是**不可更改的类型**。比如上例中“<<”运算符的第一个操作数的类型是ostream，是标准库类型，无法向其中添加成员函数。

（2）以非成员函数的形式重载，支持更灵活的类型转换。

8.3 虚函数

虚函数必须是**非静态的成员函数**。虚函数是**动态绑定**（绑定工作在程序运行阶段完成的情况）的基础。虚函数经过派生之后，在类族中就可以实现运行过程中的多态。

根据类型兼容规则，可以使用派生类的对象代替基类对象。如果用基类指针指向派生类对象，就可以通过这个指针来访问该对象，问题是访问到的只是从基类继承来的同名成员，无法访问到派生类的成员。解决这一问题的办法是：如果需要通过基类的指针指向派生类的对象，并访问某个与基类同名的成员，那么，可以在基类中将这个同名成员声明为虚函数。这样，通过基类类型的指针，就可以使属于不同派生类的不同对象产生不同的行为，从而实现运行过程的多态。

8.3.1 一般虚函数成员

虚函数必须是类的成员函数。只有非静态成员函数才可以是虚成员函数，换言之，只有对象成员函数才可以是虚成员函数。虚成员函数可以继承。一般虚函数的声明：

```
virtual 函数类型 函数名(形参表);  //类定义中虚函数的原型声明
```

虚函数的声明只能出现在类定义中的函数**原型声明**中，而不能出现在成员函数实现的时候。虚函数**一般不声明为内联函数**。因为对虚函数的调用需要动态绑定，而对内联函数的处理是静态的。

C++使用**vtable（虚成员函数表）**来实现虚成员函数的运行期绑定。虚成员函数表的用途是支持运行时查询，使得系统可以将某一函数名绑定到虚成员函数表中的特定入口地址……使用动态绑定会影响程序的效率，因为虚成员函数表需要额外的存储空间，而且对虚成员函数表进行查询也需要额外的时间。因此程序员应选择性地设定哪些函数是虚成员函数。

运行过程中的多态需要满足三个条件：

1. 类之间满足赋值兼容规则（类型兼容规则）；
2. 要声明虚函数；
3. 要由成员函数来调用或者是通过指针、引用来访问虚函数。如果是使用对象名来访问虚函数，则绑定在编译过程中就进行（为静态绑定），而无须再运行过程中进行。

类型兼容规则（见第7章7.3节）：在需要基类对象的任何地方，都可以使用公有派生类的对象来替代。类型兼容规则中的替代包括以下情况：

1. 派生类的对象可以隐含地转换为基类的对象；
2. 派生类的对象可以初始化基类的引用；
3. 派生类的指针可以隐含转换为基类的指针。

在替代之后，派生类对象就可以作为基类对象使用，但只能使用从基类继承来的成员。通过基类对象名、指针只能访问从基类继承的成员。

判断派生类的某个成员函数是否为虚函数：

1. 该函数是否与基类的虚函数有相同的名称；
2. 该函数是否与基类的虚函数有相同的参数个数及相同的对应参数类型；
3. 该函数是否与基类的虚函数有相同的返回值或者满足赋值兼容规则的指针、引用型的返回值。

若派生类的成员函数满足上述条件，就会被自动确认为虚函数。这时，派生类的虚函数便覆盖了基类的虚函数，不仅如此，派生类中的虚函数还会隐藏基类中同名函数的所有其他重载形式。

派生类覆盖基类的成员函数时，派生类函数的virtual关键字可以省略。但很多人习惯于在派生类函数中也使用virtual关键字，因为这样可以清楚地提示“这是一个虚函数”。

用指向派生类对象的指针仍然可以调用基类中被派生类覆盖的成员函数，方法是使用“::”进行限定。比如若在上例中将fun函数中的ptr->display()改为ptr->Base1::display()，那么，无论ptr所指向的对象的动态类型是什么，最终被调用的将总会是Base1类中的display()函数。

在派生类的函数中，有时需要先调用基类被覆盖的函数，再执行派生类特有的操作，这时，就可以用“基类名::函数名(...)”来调用基类中被覆盖的函数。

基类构造函数调用虚函数时，不会调用派生类的虚函数。因为当基类被构造时，对象还不是一个派生类的对象。析构时类似。

只有虚函数是动态绑定的，如果派生类需要修改基类的行为，就应该在基类中将相应的函数声明为虚函数。而基类中声明的非虚函数，通常代表那些不希望被派生类改变的功能，也是不能实现多态的。

一般不要重写继承而来的非虚函数，因为那会导致通过基类指针和派生类额指针或对象调用同名函数时，产生不同的结果，从而引起混乱。

在重写继承而来的虚函数时，如果函数有默认形参值，不要重新定义不同的值。原因是，通过一个指向派生类对象的基类指针，可以访问到派生类的虚函数，但默认形参值却只能来自基类的定义。

只有通过基类的指针或引用调用虚函数时，才会发生动态绑定。比如，如果将上例中fun函数的参数类型设定为Base1而非Base1*，那么三次fun函数的调用中，被执行的函数都会是Base1::display()。这是因为，基类的指针可以指向派生类的对象，基类的引用可以作为派生类对象的别名，但基类的对象却不能表示派生类的对象

8.3.2 虚析构函数

在C++中，不能声明虚构造函数，但是可以声明虚析构函数。

虚析构函数的声明：

```
virtual ~类名();       //声明虚析构函数
```

析构函数可继承、可派生。如果一个类的析构函数是虚函数，那么由它派生而来的所有子类的析构函数也是虚函数。析构函数设置为虚函数之后，在使用指针引用时可以动态绑定，实现运行时的多态，保证使用基类类型的指针就能够调用适当的析构函数针对不同的对象进行清理工作。

8.4 纯虚函数与抽象类

抽象类是一种特殊的类，处于类层次的上层，一个抽象类自身无法实例化，也就是说无法定义一个抽象类的对象，只能通过继承机制，生成抽象类的非抽象派生类，然后再实例化。

抽象类是带有纯虚成员函数的类。

8.4.1 纯虚函数

对于在基类中无法实现的函数，可以只在基类中声明函数原型用于规定整个类族的统一接口形式，而在派生类中再给出函数的具体实现。C++中提供纯虚（成员）函数来实现这一功能。

纯虚（成员）函数是一个在基类中声明的虚函数，它在该基类中没有定义具体的操作内容，需要各派生类根据实际需要给出各自的定义。

纯虚函数的声明格式：

```
virtual 函数类型 函数名(参数表)=0;     //必须在基类中才能声明纯虚函数
```

纯虚函数的声明与虚函数成员的原型声明，在书写格式上的不同只在于在后面加了“=0”。

声明为纯虚函数之后，基类中就可以不再给出函数的实现部分。

纯虚函数的函数体由派生类给出。

其实，基类中仍然允许对纯虚函数给出实现，但即使给出实现，也必须由派生类覆盖，否则无法实例化。在基类中对纯虚函数定义的函数体的调用，必须通过“基类名::函数名(参数表)”的形式。

如果将析构函数声明为纯虚函数，必须给出它的实现，因为派生类的析构函数的函数体执行完成后，需要调用基类的纯虚函数。

注意：纯虚函数不同于函数体为空的虚函数：

纯虚函数	虚函数
根本没有函数体	空的虚函数的函数体为空
所在的类是抽象类，不能实例化	所在的类可以实例化
可以派生出新类，然后在新类中给出虚函数新的实现，并且该实现可以具有多态特征	可以派生出新类，然后在新类中给出虚函数新的实现，并且该实现可以具有多态特征

8.4.2 抽象类

抽象类：带有纯虚成员函数的类。

抽象类只能作为基类使用，不能声明对象。故抽象类又被称为抽象基类。

一个纯虚成员函数就可以使一个类成为抽象基类，一个抽象基类可以有其他不是纯虚成员函数或甚至不是虚函数的成员函数，还可以有数据成员。抽象基类的成员可以是private、protected或private。

抽象类的主要作用是通过它为一个类族建立一个公共接口，使它们能够更有效地发挥多态特性。而接口的完整实现，即纯虚函数的函数体，要由派生类自己定义。

所谓公共接口是一个成员函数的集合，任何支持该接口的类必须定义该集合中的所有函数，这些类应该以恰当的方式来定义这些成员函数。

抽象类派生出新类之后，如果派生类给出所有纯虚函数的函数实现，这个派生类就可以定义自己的对象，因而该派生类不再是抽象类；反之，如果派生类没有给出全部纯虚函数的实现，这时的派生类仍然是一个抽象类。

抽象类不能实例化，即不能定义一个抽象类的对象，但是可以定义一个抽象类的指针和引用。通过该指针或引用，可以指向并访问派生类的对象，进而访问派生类的成员，这种访问是具有多态特征的。

第九章 群体类和群体类数据的组织

群体数据：自定义类型的数据由多个基本类型或自定义类型的元素组成

群体类：对于群体数据，仅有系统预定义的操作是不够的，在很多情况下，还需要设计与某些具体问题相关的特殊操作，并按照面向对象的方法将数据与操作封装起来

9.1 函数模板和类模板

9.1.1 函数模板

函数只有参数的类型不同，功能完全一样。类似这样的情况，如果能写一段通用代码适用于多种不同数据类型，便会使代码的可重用性大大提高。使用函数模板就是为了这一目的。函数模板的定义形式为：

```
template <class 类型参数1, class类型参数2, ...>
返回值类型  模板名(形参表)
{
    函数体
}
```

代码中的class也可换成typename，template<参数表>class标识符，指明可以接受一个类模板名作为参数

9.1.2 类模板

使用类模板使用户可以为类定义一种模式，使得类中的某些数据成员，某些成员函数的参数，返回值或局部变量能取任意类型。

类是对一组对象的公共性质的抽象，而类模板则是对不同类的公共性质的抽象，因此类模板是属于更高层次的抽象。由于类模板需要一种或多种类型参数，所以类模板也常常称为参数化类。语法形式如下：

```
template <模板参数表>
class 类名
{
    类成员声明
}
```

如果需要在类外定义其成员函数，则要采用以下方法：

```
template <模板参数表>
类型名 类名<模板参数标志符列表>::函数名(参数表)
```

9.2 线性群体

9.2.1 线性群体的概念

群体是指由多个数据元素组成的集合体。群体可以分为两个大类：**线性群体和非线性群体。**

线性群体中的元素按位置排列有序，可以区分为第一个元素、第二个元素等。非线性群体不用位置顺序来标识元素。

线性群体中的元素次序与其逻辑位置关系是对应的。线性群体中，又可按照访问元素的不同方法分为：直接访问、顺序访问、索引访问

9.2.2 直接访问群体-数组类

9.2.3 顺序访问群体-数组类

9.2.4 栈类

栈状态有：**一般状态、栈满、栈空。**当栈中没有元素时称栈空，当栈中个数达到上限时称为栈满，栈中元素没有达到栈满状态时，即处于一般状态。

9.2.4 队列类

9.3 群体数据的组织

9.3.1 插入排序

9.3.2 选择排序

9.3.3 冒泡排序

9.3.4 冒泡排序

9.3.5 二分查找

第十章 泛型程序设计与C++标准模板库

10.1 泛型程序设计及stl的结构

10.1.1 泛型程序设计的基本概念

编写不依赖于具体数据类型的程序，将算法从特定的数据结构中抽象出来成为通用的，C++的**模板**为泛型程序设计奠定了关键的基础。

概念（concept）：泛型程序设计中的一个术语，它的内涵是这些功能，它的外延是具备这些功能的所有数据类型。

> 例如，“可以比大小、具有公有的复制构造函数并可以用‘=’赋值的所有数据类型”就是一个概念，可以将这个概念记作Sortable。

模型（model）：具备一个概念所需要功能的数据类型成为这一概念的一个模型。

> 例如，int数据类型就是Sortable概念的一个模型。

概念之间有包含和被包含的关系：对于两个不同的概念A和B，如果概念A所需求的所有功能也是概念B所需求的功能（即概念B的模型一定是概念A的模型），那么就说概念B是概念A的**子概念**（有些书上又把它称为**精炼（refinement）**）。

10.1.2 STL简介

STL即标准模板库最初是由HP公司的Alexander Stepanov和Meng Lee开发的一个用于支持C++泛函编程的模板库，1994年被纳入C++标准，成为C++标准库的一部分。由于C++标准库有多种不同的实现，因此STL也有不同的版本，但它们为用户提供的接口都遵循相同的标准。

STL提供了一些常用的数据结构和算法，例如vector就是STL提供的一个容器（以后将它称为向量容器），链表在STL中也有对应容器，排序、顺序查找、折半查找等算法在STL中都有现成的函数模板。

STL的更大意义在于，它定义了一套概念体系，为泛型程序设计提供了逻辑基础。STL中的各个类模板、函数模板的参数都是用这个体系中的概念来规定的。使用STL的一个模板时所提供的类型参数既可以是C++标准库中已有的类型，也可以是自定义的类型——只要这些类型是所要求概念的模型，因此，STL是一个开放的体系。

STL所涉及的四种基本组件：**容器、迭代器、函数对象、算法。**

1. 容器

容器（container）是容纳、包含一组元素的对象。容器库类中包括七种基本容器：**向量（vector）、双端队列（deque）、列表（list）、集合（set）、多重集合（multiset）、映射（map）、多重映射（multimap）**。

这七种容器可以分为两种基本类型：**顺序容器（sequence container）**和**关联容器（associative container）**。

顺序容器将一组具有相同类型的元素以严格的线性形式组合起来，**向量、双端队列和列表容器**就属于这一种。

关联容器具有根据一组索引来快速提取元素的能力，**集合和映射容器**就属于这一种。

> 使用不同的容器，需要包含不同的头文件。

2. 迭代器

迭代器（iterator）提供顺序访问容器中每个元素的方法。

指针本身就是一种迭代器，迭代器是泛化的指针。使用独立于STL容器的迭代器，需要包含头文件**<iterator>**。

其中，s.begin()，s.end()， ostream_iterator<int>(cout, " ")都是迭代器。s.begin()指向的是向量容器s的第一个元素， s.end()指向的是向量容器s的末尾（最后一个元素的下一个位置）。ostream_iterator<int>(cout, " ")是一个输出迭代器，其中，ostream_iterator是一个输出迭代器的类模板，上例中通过执行它的构造函数来建立一个输出迭代器对象。ostream_iterator的实例并不指向STL容器的元素，而是指向一个输出流。输出迭代器关联到的输出流cout和分隔符" "都是通过构造函数提供的。

3. 函数对象

函数对象（function object）是泛化的函数，是一个行为类似函数的对象，可以像调用函数一样调用函数对象。任何普通的函数和任何重载了“()”运算符的类的对象都可以作为函数对象使用。使用STL的函数对象，需要包含头文件**<functional>**。

上例中，negate<int>()就是一个函数对象。negate是一个类模板，它重载了“()”运算符，接收一个参数，该运算符返回的就是该参数的相反数。negate的模板参数int表示的是negate的“()”运算符接收和返回参数的类型。

4. 算法

STL包括七十多个算法，包括查找算法、排序算法、消除算法、计数算法、比较算法、变换算法、置换算法、容器管理等。这些算法的一个最重要的特性就是它们的统一性，并且可以广泛用于不同的对象和内置的数据类型。使用STL的算法，需要包含头文件**<algorithm>**。

10.2 迭代器

STL的算法利用迭代器对存储在容器中的元素序列进行遍历（迭代器是算法和容器的“中间人”）。迭代器提供了访问容器中每个元素的方法。

指针本身就是一种迭代器，迭代器是泛化的指针。虽然指针是一种迭代器，但迭代器却并不仅仅是指针。指针可以指向内存中的一个地址，通过这个地址可以访问相应的内存单元；而迭代器更为抽象，它可以指向容器中的一个位置，不必关心这个位置对应的真正物理地址，只需要通过迭代器访问这个位置的元素。

指针是算法和数据结构的“中间人”：遍历链表需要使用指针，对数组元素进行排序时也需要通过指针访问数组元素（数组名本身就是一个指针），指针便充当了算法和数据结构的“中间人”。

迭代器是算法和容器的“中间人”：在STL中，容器是封装起来的类模板，其内部结构无从知晓，而只能通过容器接口来使用容器。但是STL中的算法是通用的函数模板，并不专门针对某一个容器类型。算法要适用于多种容器，而每一种容器中存放的元素又可以是任何类型，这时普通指针就无法充当“中间人”了，必须要使用更为抽象的指针——迭代器，来作为算法和容器的“中间人”。就像声明指针时要说明其指向的元素一样，STL的每一个容器类模板中，都定义了一组对应的迭代器类。使用迭代器，算法函数可以访问容器中指定位置的元素，而无须关心元素的具体类型。

10.2.1 输入流迭代器和输出流迭代器

输入流迭代器、输出流迭代器都是**类模板。输入流迭代器**用于从一个输入流中连续地输入某种类型的数据；**输出流迭代器**用于向一个输出流中连续地输出某种类型的数据。**cin**是输入流的一个实例，**cout**是输出流的一个实例（

1. 输入流迭代器：一个类模板，用于从一个输入流中连续地输入某种类型的数据。比如：

```
template<class T>istream_iterator<T>; //istream_iterator省略了后面几个有默认值的模板参数
```

上例中，T是使用该迭代器从输入流中输入数据的类型。类型T需要满足两个条件：

- 有默认构造函数；
- 对该类型的数据可以使用“>>”从输入流输入。

一个输入流迭代器的实例需要由下面的构造函数来构造：

```
```c++
istream_iterator(istream& in);  //构造函数，用于构造输入流迭代器的实例
```
```

在该构造函数中，需要提供用来输入数据的输入流（例如cin）来作为参数。

输入流结束的判断：istream_iterator类模板有一个默认构造函数，用该默认构造函数构造出的迭代器指向的就是输入流的结束位置，将一个输入流于这个迭代器进行比较就可以判断输入流是否结束。

2. 输出流迭代器：是一个类模板，用于向一个输出流中连续地输出某种类型的数据。比如：

```
template<class T>ostream_iterator<T>; //ostream_iterator省略了后面几个有默认值的模板参数
```

上例中，T是向输出流中输出数据的类型。类型T只需满足一个条件

- 对该类型的数据可以使用“<<”向输出流输出。

一个输出流迭代器的实例可以用下面两个构造函数来构造：

```
```c++
ostream_iterator(ostream& out); //构造函数，用于构造输出流迭代器的实例
ostream_iterator(ostream& out, const char* delimiter);
```
```

其中，构造函数的参数out表示将数据输出到的输出流。参数delimiter是可选的，表示两个输出数据之间的分隔符。

引入输入流迭代器和输出流迭代器的意义：虽然输入流迭代器和输出流迭代器并不能比输入流和输出流提供更强大的功能，但由于它们采用迭代器的接口，在这两种迭代器的帮助下，输入流和输出流可以直接参与STL的算法。

输入流迭代器和输出流迭代器可以被看作适配器，它们将输入流和输出流的接口变更为迭代器的接口。

适配器（adapter）：指用于为已有对象提供新的接口的对象，适配器本身一般并不提供新的功能，只为了改变对象的接口而存在。

10.2.2 迭代器的分类

STL中迭代器根据功能可分为五类：输入迭代器、输出迭代器、前向迭代器、双向迭代器、随机访问迭代器。

五类迭代器又分别对应于五个概念。五个概念之间的关系：

（1）前向迭代器是输入迭代器的子概念，也是输出迭代器的子概念（**即前向迭代器肯定是输入迭代器，也肯定是输出迭代器**）；

（2）双向迭代器是前向迭代器的子概念；

（3）随机访问迭代器是双向迭代器的子概念。

下面阐述符合这些概念的迭代器所具备的功能,先约定：

符号	含义
P	一种迭代器数据类型
p1，p2	P类型迭代器的对象
T	P类型迭代器指向元素的数据类型
t	T类型的一个对象
m	当T是类或结构体时，T中任意一个可访问到的成员
n	一个整数

下面是**除了都有公有的复制构造函数和赋值运算符之外，所有迭代器都具备的功能**：

表达式	功能	返回值
++p1	使迭代器指向下一个元素	该表达式的返回值为p1自身的引用
p1++	使迭代器指向下一个元素	该表达式的返回类型是“不确定”的

1. 输入迭代器

输入迭代器**可以从序列中读取数据**，但**不一定能够向序列中写入数据。**输入迭代器**支持对序列进行不可重复的单向遍历。输入流迭代器是一种典型的输入迭代器。**

> **输入流迭代器只适用于作为那些只需要遍历序列一次的算法的输入。**

2. 输出迭代器

输出迭代器**可以向序列中写入数据**，但**不一定能够向序列中读取数据。**输出迭代器也**支持对序列进行不可重复的单向遍历。输出流迭代器是一种典型的输出迭代器。**

3. 前向迭代器

前向迭代器这一概念是输入迭代器和输出迭代器这两个概念的子概念。**前向迭代器既支持数据读取，也支持数据写入。**前向迭代器**支持对序列进行可重复的单向遍历。**

4. 双向迭代器

双向迭代器这一概念是前向迭代器这一概念的子概念。在前向迭代器所支持功能（**数据读取、数据写入、对序列进行可重复的单向遍历**）的基础上，**双向迭代器又支持迭代器向反方向移动。**

5. 随机访问迭代器

随机访问迭代器这一概念是双向迭代器这一概念的子概念。在双向迭代器所支持功能（**数据读取、数据写入、对序列进行可重复的单向遍历、迭代器向反方向移动**）的基础上，，**随机访问迭代器又支持直接将迭代器向前或向后移动n个元素。**因此，随机访问迭代器的**功能几乎和指针一样。**

10.2.3 迭代器的区间

STL算法的形参中常常包括**一对输入迭代器**，用它们**所构成的区间来表示输入数据的序列。**

10.2.4 迭代器的辅助函数

STL为迭代器提供了两个辅助函数模板——**advance**和**distance。**

1. 迭代器的辅助函数模板**advance**

```
// advance函数模板的原型是：
template<class InputIterator, class Distance>   //advance函数模板的原型
void advance(InputIterator& iter, Distance n);
```

> advance函数模板用来使迭代器iter前进n个元素。

对于双向迭代器和随机访问迭代器（这两种迭代器均支持反向移动），n可以取负值，表示让iter后退n个元素。对于一个随机访问迭代器iter，执行advance(iter, n)就相当于执行了iter+=n。

2. 迭代器的辅助函数模板**distance**

```
//advance函数模板的原型是：
template<class InputIterator>   //distance函数模板的原型
    unsigned distance(InputIterator first, InputIterator last);
```

> distance函数模板用来计算first经过多少次"++"运算后可以到达last，[first, last）必须是一个有效的区间。若first和last皆为随机访问迭代器，在[first, last)必须是一个有效的区间的前提下，distance(first, last)的值等于last-first。

10.3 容器

10.3.1 容器的基本功能和分类

设S是一种容器类型，s1和s2是S类型的实例。

- 容器的通用功能

```
S s1          用默认构造函数构造空容器
s1 op s2      支持关系运算符：==、!=、<、<=、>、>=
s1.begin()、s1.end()：获得容器首、尾迭代器
s1.clear()：将容器清空
s1.empty()：判断容器是否为空
s1.size()：得到容器元素个数
s1.swap(s2)：将s1和s2两容器内容交换
```

- 相关数据类型（S表示容器类型）

```
S::iterator：指向容器元素的迭代器类型
S::const_iterator：常迭代器类型
```

- STL为每个可逆容器都提供了逆向迭代器，逆向迭代器可以通过下面的成员函数得到：

```
s1.rbegin() ：指向容器尾的逆向迭代器
s1.rend()：指向容器首的逆向迭代器
```

- 逆向迭代器的类型名的表示方式如下：

```
S::reverse_iterator：逆向迭代器类型
S::constreverseiterator：逆向常迭代器类型
```

- 随机访问容器支持对容器的元素进行随机访问

```
s[n]：获得容器s的第n个元素
```

10.3.2 顺序容器

1. 顺序容器的基本功能

(1) 构造函数

```
S s(n,t);//构造一个由n个t元素构成的容器实例s
S s(n);//构造一个由n个元素构成的容器实例s，每个都是T()
S s(q1,q2);//使用[q1,q2)区间内的数据作为s的元素构造s
```

(2) 赋值函数

```
s.assing(n,t); //赋值后的容器由n个t元素构成
s.assing(n);  //赋值后的容器由n个元素，每个元素都是T()
s.assing(q1,q2);赋值后的容器为[q1,q2)区间内的数据
```

(3) 插入

```
s.insert(p1,t);  //在容器s中p1所指向的位置插入一个新的元素t，该函数会返回一个迭代器指向新插入的元素
s.insert(p1,n,t);  //在容器s中p1所指向的位置插入n个新的元素t，没有返回值
s.insert(p1,q1,q2);  //将[q1,q2)区间内的元素顺序插入到s容器中p1位置处。
```

(4) 删除

```
s.erase(p1);  //删除s1容器中p1所指向的元素，返回被删除的下一个元素的迭代器
s.erase(p1,p2);  //删除s1容器中[p1,p2)区间内元素，返回最后一个被删除元素的下一个元素的迭代器
s.erase();
```

(5) 改变容器大小

```
s.resize(n);  //将容器的大小变为n，如果原有的元素个数大于n，则容器末尾多余的元素会被删除；如果原油的元素个数小于n，则在容器的末尾会用T()填充
```

(6) 首尾元素直接访问

```
s.front();
s.back();
```

(7) 尾部插入删除元素

```
s.push_back(t);  //向容器尾部插入元素t
s.pop_back();  //取出最后面的元素
```

(8) 头部插入删除元素

```
s.push_front(t);    //向容器头部插入元素t
s.pop_front();  //取出前面第一个的元素
```

10.3.3 关联容器

10.4 函数对象

10.5 算法

第十一章 流类库和输入输出

11.1 I/O流的概念及流类库的结构

流是信息流动的一种抽象，它负责在数据的生产者和数据的消费者之间建立联系，并管理数据的流动

11.2 输出流

输出流是向文件中打印信息，最重要的三个输出流：ostream，ofstream，ostringstream

预先定义好的输出流对象：
1、cout：标准输出
2、cerr：标准错误输出，**没有缓冲**，发送给它的内容立即被输出
3、clog：类似于cerr，但是**有缓冲**，缓冲区满时被输出使用插入运算符和操纵符

11.2.1 构造输出流对象

ofstream支持磁盘文件输出，在构造函数中指定一个文件名，当构造这个文件时，文件是自动打开的:

```
//可以调用默认构造函数后使用open成员函数打开文件。
ofstram myFile("filename");

//声明一个静态文件输出流对象
ofstream myFile;
myFile.open("filename") ;//打开文件，使流对象与文件建立联系。

// 同样可以在构造或者open打开文件时指定模式。
ostream myFile("filename",ios_base::out | ios_base::binary);
```

11.2.2 使用插入运算符和操纵符

插入："<<"运算符是为所有的标准c++数据类型预设计，用于创送字节到一个输出对象

1. 输出宽度：

   setw()操纵符或调用width()成员函数为每个项指定**输出宽度**

```
// 等价
for (int i = 0; i < 4; ++i) {
    cout << setw(6)<<name[i]<<setw(10)<<values[i] << endl;

    cout.width(6);
    cout << name[i] << setw(10)<<values[i] << endl;
```

2. 对齐方式

   输出流默认为**右对齐文本**，**setiosflags()**操纵符来设置左对齐（ios_base::left是ios_base的静态常量，因此引用时必须包括ios_base::前缀），resetiosflags操纵符关闭左对齐

```
for (int i = 0; i < 4; ++i)
{
    cout << setiosflags(ios_base::left)
        << setw(6)<<name[i]
        << setiosflags(ios_base::right)
        <<setw(10)<< values[i] << endl;
}
```

3. 精度

   浮点数输出精度默认值是**6**（小数位数加上整数位数总共6位）（会4舍5入）,**setprecision()**设置浮点数小数位数（包括小数点），ios_base::fixed（强制输出6位小数，例：100.0——100.000000。前提是浮点数），ios_base::scientific（科学计数法）

4. 进制

   - dec 十进制
   - hex 十六进制（默认大写）
   - oct 八进制

12.2.3文件输出成员函数

open函数：把流与一个特定的磁盘文件关联起来。需要指定打开模式。
put函数：把一个字符写到输出流中。
write函数：把内存中的一块内容写到一个文件输出流中
seekp和tellp函数：操作文件流的内部指针
close函数：关闭与一个文件输出流关联的磁盘文件
错误处理函数：在写到一个流时进行错误处理

11.2.4二进制输出文件

最初设计流的目的是用于文本，因此默认的输出模式是文本方式

11.2.5 字符串输出流

11.3 输入流

输入流是从文件读取信息

11.3.1 构造输入流对象

```
// 使用默认构造函数建立对象，然后调用open成员函数打开文件。
ifstream myFile;
myFile.open("filename");

//在调用构造函数建立文件流对象时指定文件名和模式，在构造过程中打开该文件：
ifstream myFile("filename");
```

11.3.2 使用提取运算符

提取运算符(">>")对于所有标准C++数据类型都是预先设计好的，他是从一个输入流对象获取字节最容易的方法。

11.3.3 输入流操作符

11.3.4 输入流相关函数

open函数：把流与特定的磁盘文件想关联。

close函数：关闭一个文件输入流关联的磁盘文件。

get函数：与>>函数很像，其在读入时包括**空白符**。

getline函数：从输入流中读取多少字符，并允许指定的终止符，读完后删除终止符。

read函数：从文件中读取字节到指定区域。

seekg函数：用来设置文件输入流读取位置指针。

tellg函数： 返回当前文件读指针的位置。

11.4 输入输出流

第十二章 异常处理

12.1 异常处理的基本思想

程序运行中的有些错误是可以预料但不可避免的，例如内存空间不足、硬盘上的文件被移动、打印机未连接好等由系统运行环境造成的错误。这是要力争做到允许用户排除环境错误，继续程序的运行；至少要给出提示信息。这就是异常处理程序的任务。

在一个大型软件中，发现错误的函数往往不具备处理错误的能力，这时，它就引发一个异常，希望它的调用者能够捕获这个异常并处理这个错误。若调用者无法处理这个错误，那么还可以继续传递给上级调用者去处理，这种传递会一直持续到异常被处理为止。如果程序始终没有处理这个异常，最终它会被传递到C++运行系统那里，运行系统捕获异常后通常只是简单地终止这个程序。

C++的异常处理机制使得异常的引发和处理不必在同一函数中。这样，底层函数可以着重解决具体问题，而不必过多地考虑对异常的处理。上级调用者可以在适当的位置设计对不同类型异常的处理。

12.2 C++异常处理的实现

C++语言提供对处理异常情况的内部支持。**try，throw和catch语句就是C++语言中用于实现异常处理的机制。**

12.2.1 异常处理的语法

throw 表达式语法：

```
throw 表达式
```

try块语法：

```
try
   复合语句
catch(异常声明)
    复合语句
catch(异常声明)
    复合语句
    ...
```

throw语句：如果某段程序中发现了自己无法处理的异常，就可以使用throw表达式来抛掷这个异常，将它抛掷给**调用者**。throw的操作数表示异常类型在语法上与return语句的操作数相似。如果程序中有多种要抛掷的异常，应该用不同的操作数类型来互相区别。

try语句：try子句后的复合语句是**代码的保护段**。如果预料某段程序代码（或对某个函数的调用）有可能发生异常，就将它放在try子句之后。如果这段代码（或被调函数）运行时真的遇到异常情况，其中的throw表达式就会抛掷这个异常。

catch语句：catch子句后的复合语句是**异常处理程序**，捕获由throw表达式抛掷的异常。异常声明部分指明了子句处理的异常的类型和异常参数名称，它与函数的形参是类似的，可以是某个类型的值，也可以是引用。其中的类型可以是任何有效的数据类型，包括C++的类。当异常被抛掷后，catch子句便依次被检查，若某个catch子句的异常声明的类型与被抛掷的异常类型一致，则执行该段异常处理程序。如果异常类型声明是一个省略号（...），catch子句便处理所有类型的异常，这段处理程序必须是try块的最后一段处理程序。

> 提示：异常声明的形式与函数形参的声明类似，catch子句的异常声明中也允许只指明类型，而不给出异常参数名称，只是在这种情况下在复合语句中就无法访问该异常对象了。

- 异常处理的执行过程：

  （1）程序通过正常的顺序执行到达try语句，然后执行try块内的保护段；

  （2）如果在保护段执行期间没有引起异常，那么跟在try块后的catch子句就不执行。程序从异常被抛掷的try块后跟随的最后一个catch子句之后的语句继续执行下去；

  （3）程序执行到一个throw表达式时，一个异常对象会被创建。若异常的抛出点本身在一个try子句内，则该try语句后的catch子句会按顺序检查异常类型是否与声明的类型匹配；若异常抛出点本身不在任何try子句内，或抛出的异常与各个catch子句所声明的类型皆不匹配，则结束当前函数的执行，回到当前函数的调用点，把调用点作为异常的抛出点，然后重复这一过程。

  （4）如果始终未找到与被抛掷异常匹配的catch子句，最终main函数会结束执行，则运行库函数terminate将被自动调用，而函数terminate的默认功能是终止程序。

  （5）如果找到了一个匹配的catch子句，则catch子句后的复合语句会被执行。复合语句执行完毕后，当前的try块（包括try子句和一系列catch子句）即执行完毕，即只要找到一个匹配的异常类型，后面的异常处理都将被忽略。

  细节：当以下条件之一成立时，抛出的异常与一个catch子句中声明的异常类型匹配：

  （1）catch子句中声明的异常类型就是抛出异常对象的类型或其引用；

  （2）catch子句中声明的异常类型是抛出异常对象的类型的公共基类或其引用；

  （3）抛出的异常类型和catch子句中声明的异常类型皆为指针类型，并且前者到后者可隐含转换。

12.2.2 异常接口声明

为了加强程序的可读性，使函数的用户能够方便地直到所使用的函数会抛掷哪些异常，可以在函数的声明中列出这个函数可能抛掷的所有异常类型。例如：

```c++
void fun() throw(A,B,C,D);  //函数fun()能且只能抛掷类型A,B,C,D及其子类型的异常
```

如果在函数的声明中没有包括异常接口声明，则此函数可以抛掷**任何类型的异常。例如：

```c++
void fun(); //可以抛掷任何类型的异常
```

一个不抛掷任何类型异常的函数可以进行如下形式的声明：

```c++
void fun() throw(); //不抛掷任何类型异常的函数的声明
```

> 细节：如果一个函数抛出了它的异常接口声明所不允许抛出的异常，那么unexpected函数会被调用，该函数的默认行为是调用terminate函数中止程序，用户也可以定义自己的unexpected函数，替换默认的函数。

12.3 异常处理的构造与析构

在程序中，找到一个匹配的catch异常处理后，如果catch子句的异常声明是一个值参数，则其初始化方式是复制被抛掷的**异常对象**；如果catch子句的异常声明是一个引用，则其初始化方式是**使该引用指向异常对象。**

C++异常处理的真正功能，不仅在于它能够处理各种不同类型的异常，还在于它具有为异常抛掷前构造的所有局部对象自动调用析构函数的能力。

栈的解旋：异常被抛出后，从进入try块（与截获异常的catch子句相对应的那个try块）起，到异常被抛掷前，这期间在栈上构造（且尚未析构）的所有对象都会被自动析构，析构的顺序与构造的顺序相反。这一过程称为栈的解旋。

一个在catch子句中声明异常参数（catch子句的参数）的例子：

```c++
catch (MyException & e) {...}
```

其中，也可以不声明异常参数（e）。在很多情况下只要通知处理程序有某个特定类型的异常已经产生就足够了。但是在需要访问异常对象时就要声明异常参数（e），否则将无法访问catch处理程序子句中的那个对象。例如：

```c++
catch (MyException) {
    //在这里将无法访问异常对象
}
```

用一个不带操作数的throw表达式可以将当前正在被处理的异常再次抛掷，这样一个表达式只能出现在一个catch子句中或在catch子句内部调用的函数中。再次抛掷的异常对象是源异常对象（不是副本）。例如：

```
try{
    throw MyException("some exception");
}catch(...){    //处理所有异常
    //...
    throw;      //将异常传给某个其他处理器
}
```

12.4 标准程序库异常处理

C++标准提供了一组标准异常类，这些类以基类Exception开始，标准程序库抛出的所有异常，都派生于该基类。基类Exception提供一个成员函数**what()**，用于返回错误信息（返回类型为const char*），在基类Exception中，what()函数的声明如下：

```
virtual const char* what() const throw();
```

标准异常类的继承关系：Exception类直接派生出bad_alloc，bad_cast，bad_typeid，bad_exception，ios_base::failure，runtime_error，logic_error类。其中，runtime_error类又直接派生出underflow_error，overflow_error，range_error类；logic_error类又直接派生出out_of_range，length_error，invalid_argument，domain_error类。runtime_error表示那些难以被预先检测到地异常，而logic_error表示那些可以在程序中被预先检测到的异常，也就是说如果小心地编写程序，logic_error类的异常能够避免。runtime_error和logic_error两个类及其派生类，都有一个接收const string &型参数的构造函数。在构造异常对象时需要将具体的错误信息传递给该函数，如果调用该对象的what()函数，就可以得到构造时提供的错误信息。