|  |
| --- |
|  |
| C++ Primer 笔记 |
|  |

|  |
| --- |
| leilei  2013-8-2 |

目录

[第二章 开始学习c++ 1](#_Toc363205452)

[第三章 数据处理 1](#_Toc363205453)

[3.1简单变量 1](#_Toc363205454)

[3.2 const限定符 2](#_Toc363205455)

[3.3浮点数 2](#_Toc363205456)

[3.4 c++算术操作符 2](#_Toc363205457)

[第四章 复合类型 2](#_Toc363205458)

[4.1数组（array） 2](#_Toc363205459)

[4.2 字符串 3](#_Toc363205460)

[4.3 string类简介 3](#_Toc363205461)

[4.4 结构(struct)简介 4](#_Toc363205462)

[4.5 共用体（union） 5](#_Toc363205463)

[4.6枚举类型(enum) 5](#_Toc363205464)

[4.7指针和自由存储空间 5](#_Toc363205465)

[4.8指针、数组和指针算术 6](#_Toc363205466)

[第五章 循环和关系表达式 7](#_Toc363205467)

[5.3while循环 7](#_Toc363205468)

[5.5循环文本输入 8](#_Toc363205469)

[5.6嵌套循环和二维数组 8](#_Toc363205470)

[第六章 分支语句和逻辑操作符 8](#_Toc363205471)

[第七章 函数——c++的程序模块 9](#_Toc363205472)

[7.3函数与数组 9](#_Toc363205473)

[7.9函数指针 9](#_Toc363205474)

[第八章 函数探幽 9](#_Toc363205475)

[8.1内联函数 9](#_Toc363205476)

[8.2引用变量 10](#_Toc363205477)

[8.3默认参数 11](#_Toc363205478)

[8.4函数重载 12](#_Toc363205479)

[8.5函数模板 12](#_Toc363205480)

[8.6总结 13](#_Toc363205481)

[第九章 内存模型和名称空间 13](#_Toc363205482)

[9.1单独编译 13](#_Toc363205483)

[9.2存储持续性、作用域和链接性 13](#_Toc363205484)

[9.3布局new操作符 14](#_Toc363205485)

[9.4名称空间 15](#_Toc363205486)

[第十章 对象和类 16](#_Toc363205487)

[10.1过程性编程和面向对象编程 16](#_Toc363205488)

[10.2抽象和类 16](#_Toc363205489)

[10.3类的构造函数和析构函数 17](#_Toc363205490)

[10.4 this指针 18](#_Toc363205491)

[10.5对象数组 18](#_Toc363205492)

[10.7类作用域 18](#_Toc363205493)

[10.9总结 19](#_Toc363205494)

[第十一章 使用类 19](#_Toc363205495)

[11.1操作符重载 19](#_Toc363205496)

[11.6类的自动转换和强制类型转换(未看) 20](#_Toc363205497)

[第十二章 类的动态分配内存 20](#_Toc363205498)

[12.3总结 23](#_Toc363205499)

[第十三章 类继承 23](#_Toc363205500)

[13.1公有派生 23](#_Toc363205501)

[13.2有关派生类构造函数的要点 23](#_Toc363205502)

[13.3继承——is-a关系 23](#_Toc363205503)

[13.4多态公有继承 23](#_Toc363205504)

[13.5访问控制：protected 25](#_Toc363205505)

[13.6抽象基类 25](#_Toc363205506)

[13.7继承和动态内存分配 25](#_Toc363205507)

[13.8类设计回顾 26](#_Toc363205508)

[第十四章 C++中的代码重用 28](#_Toc363205509)

[14.1包含对象成员的类 28](#_Toc363205510)

[14.2私有继承 28](#_Toc363205511)

[14.3多重继承（MI） 30](#_Toc363205512)

[14.4 类模板 31](#_Toc363205513)

[14.5总结 31](#_Toc363205514)

[第十五章 友元、异常和其他 32](#_Toc363205515)

[15.1友元 32](#_Toc363205516)

[15.2嵌套类 32](#_Toc363205517)

[15.3异常 33](#_Toc363205518)

[15.4 RTTI 33](#_Toc363205519)

[第十六章 string类和标准模板库 34](#_Toc363205520)

[16.1 string类 34](#_Toc363205521)

[16.3 STL 35](#_Toc363205522)

[16.4通用编程技术 36](#_Toc363205523)

c++源代码风格1每行一条语句2每个函数的两个花括号各占一行3函数中的语句都对于花括号进行缩进4与函数名称相关的圆括号周围没有空白（空行将声明语句与程序的其他部分分开，或在变量前声明，c++的做法是尽可能在首次使用变量前声明）。

# 第二章 开始学习c++

变量名的约定：类型\_名称 或者是 类型(int)名称（akjl）

c语言输入输出是printf，scanf,他们都包含在头文件stdio.h中

操作符重载：通过重载，同一个操作符将有不同的含义。&操作符表示地址操作符又表示and按位操作符

c++程序应当为程序中使用的每个函数提供原型，而库函数在头文件中有声明，而在库文件中有他的定义。标准c库中提供了140多个预定义函数，而main（）函数不是关键字，由于它不是语法的组成

头文件stdlib.h中rand()函数不接受任何参数，并返回一个随机整数，其函数声明如下：int rand(void);

# 第三章 数据处理

## 3.1简单变量

内置c++类型有基本类型和复合类型

sizeof操作符能够打印类型或变量长度的单位字节数，如：sizeof（int）;sizeof a;//a 为变量

头文件climits包含了整型限制信息

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| schar\_max | signed char最大值 | char\_bit | char的位数 | ing\_min | int最小值 | uchar\_max | unsigned char最大值 |
| schar\_min | signed char最小值 | shrt\_max | short最大值 | long\_max | long最大值 | unit\_max | unsigned int最大值 |
| char\_max | char的最大值 | shrt\_min | short最小值 | long\_min | long最小值 | ulong\_max | unsigned long最大值 |
| char\_min | char的最小值 | int\_max | int最大值 |  |  | ushrt\_max | unsigned short最大值 |

c++的变量的初始化有两种，第一种c语言的：int a=15;第二种是c++的：int a(15);如果不初始化那么变量的值是不确定的

C++添加了两种类型：long long和unsigned long long 两种类型都至少是64位的

cout<<hex;//告诉编译器以16进制的方式输出

cout<<bianliang<<oct;//输出完以后切换到8进制的方式输出

cout<<bianliang<<dec//输出完以后切换回10进制

cout<<1982ul;c++打印常量时候，常量后面加l或l表示1982被存为long型，若是ul或ul表示存为unsigned long型，若是u或u表示存为unsigned int，若无后缀表示存为int。

char ch=’a’;cout.put(ch);//成员变量put()可以打印char型变量，我们也可以写成cout.put(‘n’)；

c++转义序列符 (括号里的为ascii符号)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 换行符(nl或lf) | \n | 退格(bs) | \b | 回车(cr) | \r | 反斜杠(\) | \\ | 单引号(‘) | \’ |
| 水平制表符(ht) | \t | 垂直制表符(vt) | \v | 振铃(bel) | \a | 问号(?) | \? | 双引号(“) | \” |

cout<<endl<<’\n’<<”\n”;//三种方式等效且都表示换行

通用字符名：通用字符名以\u或\u开头，\u后面在跟8个十六进制位，而\u后面跟16个十六进制位，这些位表示的是iso10646编码（iso10646是一种正在制定中的国际标准，位大量的字符提供字符编码，他与unicode合作，确保标准同步，unicode可以表示96000多种符号和49个手写符号，而ascii码就是unicode的一个子集）

cout<<”ni g\u00e2teau.\n”;打印如下ni gâteau //â的iso10646编码是00e2

特殊类型：wcha\_t 宽字符类型 bool类型，任何数字值或指针指都可以被隐式的转换为bool值。任何非零值都被转换为true，零被转换为false；反之 int aa=true;long bb=false;也成立。

## 3.2 const限定符

const long months=true；//const限定符用于定义常量

const限定符定义常量时①常量名一般全用大写，便于与变量区分；②要直接初始化，否则将无法初始化

## 3.3浮点数

浮点数的书写有两种方式：①常用的标注-942635；②-1.54e-4 5.6545e8 。浮点类型要添加头文件float.h，浮点型的类型有float(32位，其精度低，系统至少有6位有效数位或更低)，double(64位，精度至少在13位有效位)，long double(80,96,128位)。 浮点常量默认为double类型若希望常量位float，要用f或f后缀，对于long double类型。使用l或l(尽量用l，l像1) cout<<2.2l<<3.12f<<8.123;//8.123为double默认型

c++标准在默认输出时会删除小数点后末尾的零，如将23.450000显示成23.45。

cout.setf(ios\_base::fixed,ios\_base::flatfield);//可以覆盖默认设置，让打印末尾的0显示出来

## 3.4 c++算术操作符

求模操作符要求两个数必须为整数

潜在的数值转换问题①大浮点型→小浮点型：精度降低可能超出目标类型的取值范围，结果是不确定的。②浮点型→整型：小数丢失，原先的值可能超出目标类型的取值范围，结果不确定。③大整型→小整形：容易超出范围，通常只复制右边字节

强制类型转换：不会修改变量本身，而是创建一个新的指定类型的值。其格式有两种：

(long) bianliang;//c语言强制类型转换的方式

long (bianliang);//c++风格，其想法是让强制转换类型就想是函数调用

# 第四章 复合类型

复合类型：在c语言中叫派生类型，而c++中类使用了“派生”这个词,所以叫复合类型

## 4.1数组（array）

数组的三要素是：每个元素的类型，数组名，元素数, 如short mouths[12]

数组的下标总是从0开始到元素数减1结束；如果下标超界，编译器不会报错，但可能破坏数据或代码也可能异常终止。**所以必须确保有效下标。**

long kk[10]={10，12,6856,12}; //ok

long jj[10];//ok

jj[10]={1,2,3};//not allowed

long cc = jj;//not allowed

//数组初始化，有逗号隔开，大括号里元素可以少于元素总数，数组的这种初始化只能用与定义数组的时候。也不能把一个数组赋给另一个数组

cout<<sizeof kk;//输出kk这个数组的字节数，有40个字节

cout<<sizeof kk[1];//输出kk这个数组的第二个元素的位元组数他有4个字节

long kk[15]={0};//将没个元素都初始化为0的方法，为被定义的元素自动初始化为0。

long kk[]={0,523,54,3,4};//编译器计算出[]内的元素个数，这里为5个

long long\_kk=sizeof kk/sizeof(short);//我们不知道数组的个数是多少,这样就知道了

## 4.2 字符串

c++处理字符串的方式有两种1:来自c语言,常被称为c-风格字符串;另一种基于string类库

定义两个char型数组：

char dog[5]={‘a’,’b’,’c’,’d’,’c’};//定义 了一个char型数组，储存的不是一个字符串

char cat[5]={‘a’,’b’,’c’,’d’,’\0’};//储存的是一个字符串，到空字符结束。

//在确定存储的字符串所需的最短数组时，别忘了将结尾的空字符计算在内

另一种初始化字符串的方法：char kk[]=”nihao!!”;

//编译器会在字符串最后加上”\0”字符；字符串用双引号，字符单引号

拼接字符串常量：**下面的语句是等效的。**

cout<<”我们可以将字符串分开””写，但双引号紧靠着，中间无符号”;//不建议采用这种方法

cout<<”我们也可以将字符串直接写在一对双引号内”;

cout<<”我们也可以将字符串分开写，”

”并像这样放在两行里，第一行末尾没有分号”；//这中方法是解决一行内字符串太多，分行写

strlen()函数是标准函数库cstring的函数 用来确定字符串的长度。

char kk[15]=”nihao!”;//strlen只计算可见字符，不把空字符计操作数在内

cout<<strlen(kk)<<” ” <<sizeof(kk);//sizeof是确定数组的长度

kk[3]=’\0’; //当cout读到’\0’时变停止

cout<<kk<<” ”<<kk[5];//后面的字符忽略(未被清空)，’/0’有截取字符的功能

打印如下：6 15 nih !

读取一行字符串的输入:istream中的类(如cin)提供面向行的类成员函数：getline()和get().

getline()将丢去换行符，而get（）将换行符保留在输入序列中。

getline()有两个参数，第一个是数组名第二个是读取字符数-1（如若是20，最多度19个）

在读取指定数目的字符或遇到换行符时停止读取。不保存换行符，最后用空字符代替。

get()有几个变体。其中一个与getline相似他们接受参数相同，解释参数也相同。

该函数并不是用空字符代替换行符，而是插入，并将换行符保留在输入列中，这样get()将无法跨行，因为下个get()一开始就读到了换行符,并有把他保存在输入队伍中.

cin.get(kk,20);

cin.get();//get()另一个变体cin.get()可读取一个字符，即使是换行符,这样就可以保留在输入列中

cin.get(mm,20);

我们可以将两个类成员函数合并：cin.get(array,200).get();//第一个get()返回一个cin对象，该对象随后被用来调用第二个get()函数。同样：cin.get(array,200). cin.get(array,200);也是可以的等效于调用两次getline()。而cin.get().get()则不行，以为第一个get()不返回对象.

混合输入字符串和数字：

int a; char kk[100];

cin<<a;//如果就这样将跳过下一行，有一个回车；取消回车的方法：(cin<<a).get();//加括号

cin.getline(kk,100);//如果就这样，我们将无法输入字符串就跳到下一行，因为上一行有回车

c++程序长使用指针(不是数组)来处理字符串

## 4.3 string类简介

string类包含在头文件string里，且位于名称空间std中。string对象的方式与使用字符数组相同：可以使用c-风格字符串来初始化string对象；可以使用cin来将键盘输入存储到string对象中；可以使用cout来显示string对象；可以用数组的表示法访问其中的字符

理论上，char数组是存储一个字符串的存储单元，string类变量是一个表示字符串实体

string str1=”kjlsamdl”，str2,str3;//string对象我们可以看做是一个变量,可以连续定义

str2=str1;//我们可以将以个string对象赋给另一个string对象

str2+=str1;//我们可一将string对象相加，+=是将以个字符串附加到另一个string对象的末尾

str2+=“nikan”；//我们也可以将一个string对象的末尾加上一个字符串

string类是c++新增的string库来处理输入的字符串，c语言中的字符串可使用头文件cstring提供的函数来完成

char c1[20]=”jjjj”,c2[20];//数组也可以连续定义，**要注意不要超出数组的大小。**

strcpy(c2,c1);//strcpy()能将c1中的字符串赋值到c2中,**不能超出数组的大小，会破坏数据；发生错误。**

strcat(c2,”kkkk”) ;//strcat()函数能将字符串”kkkk”附加到c2字符串的末尾处  
cout<<c1<<’\t’<<c2<<’\t’<<strlen(c1);//strlen()也是cstring提供的函数，能读取数组中字符串的大小

打印结果：jjj jjjkkkk 4

string str=“123”；//如果没有初始化,str为0个字节，他能够随着输入字符串的大小自动调节

getline(cin,str)；//string对象的引入比istream晚很多，所以getline并不是string类方法，只能创建一个string类的友元函数。因此只能这样调用。

int kk=str.size();//size()是string对象的方法，他能够读出str内的字符串个数为3（不带’\0’空字符）

## 4.4 结构(struct)简介

结构可以存储多种类型的数据。

定义结构描述：1它描述并标记了能够存储在结构中的各种数据类型;2按描述创建变量

每个列表项都是一条声明语句，每一个声明语句都是一个结构成员

struct inflatable goose;//定义结构变量 这种风格是c语言中的

inflatanle vincent; //c++可以省略struct关键字

goose.price就相当于一个double类型的变量，可以像常规使用double变量那样来使用它们，访问类成员函数就是从访问结构成员这里衍生来的，结构也是c++oop（类）堡垒的基石

struct inflatable

{ char name[20];

float aa;

double bb;

// inflatable是标签成为新的类型的名称；name、aa、bb是结构成员；最后的分号结束结构声明。当然我们可以不加结构名称inflatable而建立以个结构变量kk，用kk来调用其结构的成员，但因为没有结构名称，所以不能定义其他的结构变量，这样后续程序将不能使用该结构

}kk= //我们可以在定义结构时直接定义结构变量，我们也可以直接对其初始化。

{“leilei”, //初始化结构变量之间必须用逗号。

12.3, //要用逗号。c++不提倡使用外部变量，但提倡使用外部结构声明，还提倡外部声明符号常量

6.8}; //最后一个成员变量值后面不用加逗号，大括号的后面一定要加分号。

inflatable mm={“leilei”,12.3,6.8};//我们可以直接对结构变量进行初始化，成员间用逗号，末尾用分号

inflatable ll；ll=mm;//我们可以将结构赋给另一个同类型的结构

结构数组：

inflatable kk[2]= //结构数组是一个数组，而不是一个变量，这样数组中的每个值都是一个结构变量

{{“shuzu0”,0.5,13.546}, //每个结构数组变量初始化，用{}扩起来，每组变量用逗号, 可放一行也可分开

{“shuzu1”,513.1,56.133}}；//结构数组变量之间也用逗号进行分隔，也可以放在一行内

结构中的位字段

## 4.5 共用体（union）

是一种数据格式，它能够存储不同的数据类型，但只能同时存储其中的一种类型。结构能够同时存储各种类型，而共用体只能存储一种类型，因此共用体的长度为其最大成员的长度。公用体的用途之一是当数据项使用两种或更多种格式（但不会同时使用）时，可以节省空间

union mingcheng{long long\_aa;bouble bouble\_aa;}bianliang;//语法与结构一样，但每次只调用一个成员，可直接定义变量

cout<<bianliang.long\_aa;//输出成员变量的值

匿名共用体：没有名称，其成员将成为位于相同地址处的变量。显然每次只有一个成员是当前的成员

struct inflatable{char name[20];double hehe;union{long long\_aa;double double\_aa;};};

inflatable jiegou\_bianliang;cout<<jiegou\_bianliang.long\_aa;//匿名共用体成员被视为结构的成员，可直接调用

## 4.6枚举类型(enum)

提供了另一种创建符号常量的方式，这种方式可以代替const。也可以定义新的类型

enum kk{a,b,c,d,e};//kk被称为枚举，a.b.c等被称为枚举量，第一个枚举量为0，往后依次类推

kk band = d;//枚举变量只能等于枚举量，不能等于其他非kk枚举的枚举量

band = kk(12);//我们可以通过枚举来给枚举变量赋值,通过这种方法给枚举变量赋值的范围为：

上限：比这个枚举量的最大值大的最小的2的幂的值减一，如：上式中最大枚举量是4，比4大的2的幂为2^3=8,那上限是2^3-1=7；下限：如果不小于零则下限是0；如果小于零，则为小于最小枚举量的最大2的幂加一，如最小为-6，则最小范围为-(2^3)+1=-7

选择用多少空间来存储枚举由编译器决定，对于范围小的枚举使用一个字节或更小的空间；

enum kk={a,b=0,c=100,d,f};//枚举量可以相同(a和b的枚举量都为0)，当c=100，则d=101,f=102

## 4.7指针和自由存储空间

c++基本原理：面向对象编程与传统的过程性编程的区别在于，oop强调的是在运行阶段而不是编译阶段进行决策。指针用于存储值的地址，指针名表示的是地址。\*操作符被称为间接值或解除引用操作符，如m表示的是一个地址，\*m表示储存在该处的值。

int\* p\_kare=&kk;//被初始化的是指针本身，确定指针指向的变量，而不是确定它指向的变量的值。

操作符\*两边的空格是可选的。c程序员使用这种格式：int \*ptr;而c++使用这种格式：int\* ptr;在c++中，int\*是一种复合类型，是指向int的指针。

int\* p1,p2;//这样将创建一个指针p1和一个常规int变量p2。对于每一个指针变量名，都需要使用一个\*。

int\*a;\*a=23;//未初始化的指针的危险。应养成在定义指针时将指针初始化为一个确定的、适合的地址的习惯。

int\* pt=0xb8000000;//错误。c++不允许将一个整型赋给指针，指针类型与整型是不一样的。

int\* pt=(int\*)0xb8000000;//我们可以通过强制类型转换将整型转换为适当的地址类型。

为一个数据对象(可以是结构，也可以是基本类型)获得并指定分配内存的通用格式如下：typename pointer\_name =new typename;

int\* ps=new int;//定义一块新的内存区域

int\* pn=ps;//不要创建两个指针指向同一个内存块的指针，因为这将增加错误删除同一内存块两次的可能。

delete ps;//删除指针指向内存块而不是指针的值，释放ps的内存，但不会删除指针ps本身。可将其指向另一个新内存块

delete的使用①一定要配对使用，否则将发生内存泄露，被分配的内存将无法再使用了，如果泄露严重，程序将由于不断寻找更多内存而终止②不能释放已释放的内存块，否则结果将不确定，这意味着什么情况都可能发生③不能释放声明变量所获取的内存，不过，对空指针使用delete是安全的。

在编译时给数组分配内存被称为静态联编，意味着数组是在编译时加到程序中的，这种叫做静态数组；可以在程序运行时选择数组的长度被称为动态联编，意味着数组是在程序运行时创建的，这种叫做动态数组。

为数组分配内存的通用格式如下：type\_name = new type\_name [num\_elements];如：long\* p\_kk = new long [100];

释放一个new[]数组要用delete [] zhizhenming;

动态数组的使用

int\* a = new int [2];a[0]=0;a[1]=1;//初始化。可以将他看成int a[2]数组来处理,a存储的地址为数组第一个值的地址。

a+=1;//此时a[0]=1; a-=1;//此时a[0]=0

delete [] a;//将指针的值切换为数组的第一个值的地址，给delete[]提供正确的地址，便于释放。

## 4.8指针、数组和指针算术

指针和数组基本等价的原因在于指针算术和c++内部处理数组的方式

**规范一下：\*p不是指向类型的指针，而是完全等同于一个类型的变量；p才是指针。**

将指针变量加一，就是增加的值等于指向的类型占用的字节数。如：long\* a;a++;//a指针变量保存的地址加4

\*(pointname+j)=pointname[j]或者arrayname[j]=\*(arrayname+j)

//数组表示法与指针表示法之间区别区别：①可以修改指针值，而数组名是常量；②sizeof操作符对数组得到的是数组长度，而对指针得到的是指针的长度，即使指针指向的是一个数组。

指针和字符串：数组和指针的特殊关系可以扩展到c-风格字符串。

char flower[10]=”nihao”;//常规定义一个char型数组来存储字符串

cout<<flower<<”haizaima?\n”;//flower打印时并不是打印第一个字符，而是打印从该地址起往后的字符串，直到遇到‘\0’//后面”haizaima?\n”字符串打印，cout只知道第一个字符的地址，打印直到遇到’\0’结束

**在cout和多数c++表达式中，char数组名、指向char的指针以及应用引号的字符串常量都被解释为字符串第一个字符的地址。**

const char\* pp=”nihao”;// ”nihao”表示是其第一个字符的地址，所以pp所保存的地址为字符串第一个字符的地址。形式与char型数组一样，但char型数组的数组名可以重新定义数组的字符串，但是常量字符串const不能被重新定义。

cout<<flower<<endl<<(int\*)flower;//第一个打印的是字符数组中的字符串，第二个打印的是第一个字符的地址。

char型数组在初始化后重新定义时，初始化的字符串将被清空而重新定义

警告：在将字符串读入程序时，应使用已分配的内存地址，该地址可以是数组名也可以是new初始化过的指针。如果给cout提供一个指针，他将打印地址。但如果指针类型为char则cout将显示指向的字符串。如果要显示字符串的地址，则要用强制转换符转换成另一种指针类型，对其进行转换cout<<(int\*)p<<endl;  
strcpy()可以将一个字符串数组复制给另一个字符串数组：atrcpy(shuzu1,”zif”);//也以将字符串复制给数组

但当数组的空间不足时，会超出数组内存，超出部分将覆盖数组后面的其他内存；这样讲会影响程序运行，因此我们用strncpy这个函数来复制如：strncpy(shuzu,”nai znlls”,5);//它将字符串的前五个字符付给了数组，但它不能添加空字符‘\0’

所以要设置 shuzu[5]=’\0’;//我们要添加语句，确保字符数组的最后为空字符，要不然它存储的不是字符串了

使用new创建动态结构：在运行时创建数组优于在于编译时创建数组，对于结构也是如此。

struct inflatable{ char name[20];float volume ;double price;}；//创建结构模板 ，定义其中的结构成员。

inflatable \*ps = new inflotable;//定义一个指针指向在堆中创建的结构；ps表示指向结构的指针，\*ps是表示结构变量

cin.get((\*ps).name,20);//**由于创建的结构变量没有名称，而只有指向他的指针，所以可以用\*ps表示结构名称。**

cout<<ps->name<<endl;//**指针不是结构变量名称，不能用句点符，而要用->来访问结构成员**。

**记住：结构变量名来访问成员时用句点符(.)，当用指向结构变量名的指针来访问成员时要用->操作符**

# 第五章 循环和关系表达式

第五章的主要内容：

①for循环 ②表达式和语句 ③递增和递减操作符：++和-- ④组合赋值操作符 ⑤复合语句（语句块）⑥逗号操作符 ⑦关系操作符 ⑧while循环 ⑨typedef工具 ⑩get()字符输入方法 ⑾文件尾条件 ⑿嵌套循环和二维数组

**for循环：**其中的判断表达式都能转换成bool型，如判断表达式为i，若i=1为真，若i=0为假，而所有的表达式也都可以转化成bool类型的true或者false类型；c++表达式都是值与操作符的组合，**每个c++表达式都有值。**

maid=(cooks=4)+3;//在c++中可以这样做，但是不提倡这样做，这样只是为了让x=y=z=0；这样的语句存在

long a=100;cout<<a>1; //a>1这个表达式为真成数值1,他将显示一个数字1。要成为表达式就必须能确定其值。

cout.setf(ios.base::boolalpha); //将输出结果转化成bool类型，同时用true和false表示，而不是1和0

cout<<a<3; //设置以后就不显示数字0了，而是显示一个false

定义一个const值来表达数组中的元素个数是好的习惯。在声明数组和引用数组长度时，可以使用const值。

在string类的size()获得字符串中的字符数，如：string word ；cin>>word;int kk=word.size();

a++//表示先使用当前的，使用完后就加一。 ++a//表示先加一后，在使用其值。

组合赋值操作符：有+= -= \*= /= %=

复合语句（也叫语句块）：单独的一对大括号就是一个语句块，在一个语句块内定义的变量到语句块外面就释放了

逗号操作符：逗号操作符的优先级是最低的。如：i=20,j=i\*2;//两个语句是有先后顺序，翻过来是不行的

for(long i=0,j=10;i<j;i++,j--)//逗号操作符一般用于for循环语句中。这个循环可以将字符串中的字符反过来。

关系表达式：< > <= >= == !=;

//for第二个关系表达式如果为赋值表达式则其表达式的值是true

c-风格字符串的比较：我们要用到cstring类库中strcmp()可以比较两个字符串的大小，是否相等。如果两个字符串相同则该函数返回的值为零；如果第一个字符串按字母顺序排在第二个字符串之前，则返回一个负数；如果第一个字符串按字母顺序排在第二个字符之后，则返回一个正数。char aa[20];if(aa==”nihao”)//错误，这是比较两个字符串地址。正确的比较是：if（strcmp（aa，“nihao”）==0）

string类字符串的比较：string类的比较字符串方法是比较简单的。

string str1；str1 == “nihao”;//这里比较的是两个字符串，并非比较的是内存地址。

## 5.3while循环

while（1）；{…}//分号是一个结束语句，while是没有循环体，是一个死循环，编译时什么也不动

用whlie循环来计时：在ctime头文件中clock()可以解决计时问题。ctime提供了以个符号常量clocks\_per\_sec,该常量等于每秒钟包含的系统时间单位数。将系统时间除以这个值将得到秒数，将秒数除以这个变量可以得到以系统时间为单位的时间。clock\_t是clock()函数返回类型的别名， 我们可以将变量声明为clock\_t类型，他表示long、unsignde int或系统其他类型

类型别名：c风格的预处理命令#define 和 c++的typedef都可以来创建类型别名。

#define ch char\*;//在编译时所有的ch都被替换成char\*，这样ch kk;中的kk就是以个char型指针。这里用char\*代替ch,但是如果定义 ch a，b；则会被解析成 char \*a，b；但是typedef就不会出现这个问题，它能解决更复杂的情况。

typedef char\* ch;//我们也可以这样，有时我们只能这样，这是c++风格 这里我们用char\*来代替ch

do whlie循环：这种循环不如前两种，但他有他的特别用法，如让用户输入一 个数后在做判断，这个循环至少循环一次。

## 5.5循环文本输入

cin>>ch;// 使用原始的输入能将键盘输入的下一个字符输入到变量中，但他能将空格、换行符和制表符忽略。

cin.get(ch);//读取下一个字符,不管该字符是什么，并将其存储在ch中。其通过返回转换为false的bool值确定已达到eof

ch=cin.get();//成员函数调用cin.get()返回下一个输入字符,包括空格、换行符合制表符通过返回eof值来指出已达到eof

eof检测文件尾：eof是在文件iostream中定义的

## 5.6嵌套循环和二维数组

long kk[2][3]= //我们可以将其看成包含2个元素的数组，而每个元素有包含了3个long型整数的数组。

{{13,12,11}，//初始化时每一行内用逗号，每一行之间用逗号，同一维数组一样要将其在最后加分号。

{15,16,18}};//可以把他们写在一行内，但作为一种风格，分行写成表格的形式，清晰也便于查找，结尾用分号

char\* kk[2]={“nihao”,//定义并初始化指针数组。初始化和普通数组一样，只不过这里的值是字符串

”haizai ba !”}; //第一个字符串的地址，而调用kk[0]时就等于调用了第一个字符串

# 第六章 分支语句和逻辑操作符

if语句：条件操作符合错误防范，if(3==kk)//我们这样做如果==写成=，编译器就会报错。而if(kk=3)将恒成立

6.2逻辑表达式

or操作符||：i++ <6||i>6 //如果i=10，则在比较i>6时，i会变成11，运算顺序从左向右。

and操作符&&：if(1<kk<5)//这个判断永远为1，等价于if((1<kk)<5)//1<kk的值不是0就是1，它永远小于5。

not操作符！：操作符的优先级高于所有的关系操作符和算术操作符，对表达式求反，必须用括号括起来

逻辑and操作符的优先级高于逻辑or操作符。x!=0&&1/x>10;//如果x=0，第一个条件为假，后面的就不在判断了，避免了除零

C语言中逻辑操作符的另一种表示方式：就是用not、or、and来代替! || &&。在c语言中要加上头文件iso646.h，它们不是c语言的保留字，在c++中不能使用，它们都是C++的保留字。

6.3字符函数库cctype

该函数库中的函数能够判断，输入的字符是大写，小写，数字，空格，符号等，比用逻辑操作符判断起来简单多了。

6.4 ?:操作符：它是c++唯一一个需要两个操作符的操作符，在程序员中广受欢迎。

6.5 switch语句：当选择不是一个范围时，超过3个以上选项，一般就用switch语句，都不符合就用default：i++;break;

case ‘c’://如果case后面没有break；那么将执行下一个case，直到碰到break。case标签都必须是一个单独值，非范围

case ‘c’:hanshu();break;//这样输入大写小写都行，这也是一种解决大小写一样的一种方法

6.6break和continue

break是用在switch语句或任何循环中，他能够跳出switch或循环后面的语句出执行

continue语句用于循环中，让程序跳过循环体当前一轮循环后面的代码，并开始下一轮循环；

goto语句：一般情况下不要用goto语句，容易造成错误，一般在多层嵌套循环内层，直接跳出，用goto比用多个break简单

# 第七章 函数——c++的程序模块

函数在声明中接受参数与在定义中接受参数可以用不一样的变量名。

如：void kk(long j);//声明 void kk(long k){}//定义

在函数参数的过程中，如果基本类型(非复合类型)不匹配，c++会自动强制类型转换，但尽量避免强制类型转换，造成数据破坏

## 7.3函数与数组

引用的属性和特别之处long kk=10;viod hans(long &kk);void hanshans(const long &kk)//声明变量、定义函数

为了保证被调用函数只使用调用函数的变量值，而不是是修改变量值，我们一般情况下是不用引用的，如果要用引用我们可以将被调用函数中的引用声明为const引用，这样我们只能使用该函数，但是不能修改它的值，这就像是const long\*const kk;

void hans(kk+3);//这是错误的，现在的c++ 是不允许将表达式传递给引用，而按值传递是允许的

void hanshans(kk+3);//这是正确的，在现在c++中如果参数定义为const引用时，我们可以给他传递非变量名的参数因为这样//它创建了临时变量，用引用指向临时变量时，但是函数不会修改器临时变量值，这样避免了想修改原函数变量值，但没修改的状况

//用const我们编程时不会去修改原函数的变量。**在c++中使用引用时尽量将引用声明为const这样传递的类型可以转换**

将引用于用于结构：引用非常适合用于结构和类，c++加入引用主要为了用于这些类型的，而不是基本的内置类型

## 7.9函数指针

**获取函数的地址:**只要使用函数名(后面不跟参数)即可

**声明函数指针:**假如有一个函数为:double pam(int) 则正确的指针类型声明为:double (\*pf) (int) 这与pam()的声明类似，只是把pam替换成了(\*pf)，由于pam是函数，则(\*pf)也是函数，而如果(\*pf)是函数，则pf是指向函数的指针

由于括号的优先级比\*高，则\*pf(int)表示，pf是一个返回指针的函数，而(\*pf)(int)，表示pf是一个指向函数的指针。

正确的声明了函数指针之后，可以将函数的地址赋给它:double pam(int); double (\*pf)(int); pf=pam

**使用指针来调用函数：**(\*pf)扮演了函数名的角色,所以使用(\*pf)的时候只需要将它看作是函数名即可:double pam(int); double (\*pf)(int); pf=pam;

double x = (\*pf)(5); 和double x = pf(5);都是正确的

# 第八章 函数探幽

## 8.1内联函数

内联函数的编译代码和其他程序代码“内联”起来了，也就是说编译器将使用相应的函数代码替换函数调用（**有多少次调用就有多少次拷贝**），内联函数的运行速度比常规函数快，但是需要占用更多的内存。如果代码的执行时间相对于调用时间短，则可以使用内联函数

函数声明前加上关键字inline，函数定义前加上关键字inline

内联函数和常规函数一样，也是按照值来传递的。

内联和宏

#define SQUARE(x) x\*x

a=SQUARE(5); b= SQUARE(5+2); c= SQUARE(a++);

则只有a能够被正确赋值。宏只是简单地替换，并不能有函数那么智能

## 8.2引用变量

引用是已定义的变量的别名.修改引用的值和修改变量的值是一样的效果

**创建引用变量：**int rats; int & rodents = rats; 其中&不是地址操作符，而是表示rodents是rats的引用变量，int &表示是指向int的引用，就像在声明中 char\* 表示的是指向char的指针。rats 和rodents的地址相同。必须在声明引用的时进行初始化。和const类型相似，必须在声明时初始化。

**将引用用作函数参数：**引用经常被用作函数的参数，使得函数中的变量名成为调用程序中的变量的别名，这种传递参数的方式叫做按引用传递。和按指针传递效果一样。

例子(交换两变量的值)：

void swapr(int &a,int &b)

{

int temp = a;

a=b;

b=temp;

}

void swapp(int \*a,int \*b)

{

int temp=\*a;

\*a=\*b;

\*b=temp;

}

void swapp1(int \*a,int \*b)

{

int \*temp=a;

a=b;

b=temp;

}

调用方法分别为 int a; int b; swapr(a,b); swap(&a,&b); int swapp1(&a,&b);

第三种方式不行，因为它只是交换了函数内部的指针指向的地址，并没有改变该地址的变量的值。当函数结束时，两个指针变量将会被销毁，则没有对两个变量的值产生任何变化。

**引用的属性和特别之处：**

按值传递可以接受多种类型的实参：cube(x+2);cube(x\*5)。但是按引用传递不可以接受多种类型的实参，但const引用可接受多种类型的实参。像cube(x+2);这种形式，因为表达式x+2并不是变量

如果参数为const引用时，C++允许生成临时变量，即按引用传递参数的函数调用的参数不是左值或者与相应的const引用参数的类型不匹配，则C++将创建类型正确的匿名变量，将函数调用的参数的值传递给该匿名变量，并让参数来引用该变量。因为如果接受引用的参数的函数的意图是修改作为参数传递的变量，则编译器将会让两个临时变量交换值，而原来的变量的值不变，const变量不允许改变变量的值，因此不会出错,可行。

如：refcube(const double x); int a; 可以接受：refcube(7); refcube(x+2); refcube(a);

尽可能的使用const：使用const可以避免无意中修改数据的编程错误；使用const可以接受const类型的数据，也可以接受非const类型的数据；使用const能使函数正确的生成临时变量，减少错误.

**将引用用于结构：**

struct sysop

{

char name[26];

char quote[64];

int used;

}

sysop copycat= use(looper);

const & sysop use(sysop & sysopref)

{

sysopref.used ++;

cout<<”name:”<<sysopref.name<<endl;

cout<<”quote:”<<sysopref.quote<<endl;

cout<<”used:”<<sysopref.used<<endl;

return sysopref;

}

int main()

{

sysop copycat;

sysop looper={“NAME”,”QUOTE”,0};

use(looper);

copycat=use(looper);

}

如果函数use()返回的是一个结构，则sysopref的内容将被复制到一个临时返回存储单元中，然后该临时返回存储单元的内容将被复制到copycat中。然而由于use返回的是一个指向looper的引用。这种情况下，looper的内容将被直接复制到copycat中，这是返回指向结构的引用而不是结构的优点：效率更高

返回引用时需要注意的问题：避免返回当函数终止的时候不再存在的内存单元引用

const sysop &clone(sysop &sysopref)

{

sysop newguy;

newguy = sysopref;

return newguy;

}

const sysop &clone(sysop &sysopref)

{

sysop \*newguy = new sysop;

\*newguy = sysopref;

return \*newguy;

}

左边的那种方式不行，因为在函数结束之后，临时变量newguy就会被销毁。

C++中使用new的方式可以，但是需要使用delete来释放它们，auto\_ptr模板可帮助自动完成释放工作。

**关于指针的一个小感悟：**

int \*pt;未初始化的int指针，系统会分配pt的存储空间（没保存int的地址），即并不会分配它指向的int的存储空间，存储空间不存在。

int \*pt=new int;系统会分配pt的存储空间，并且会分配它指向的int的存储空间，只是int的值是不确定的。int存储空间存在

int a=5; \*pt=a;这个的意思是，把a的值赋给\*pt，就是把a的值写到pt指向的int类型的存储空间去。前提是该存储空间应该存在。

pt=&a;这个的意思是，把a的地址赋给pt，就是把a的地址存储到pt的存储空间里面。如果原来分配了int变量的空间，也会被覆盖。

use()返回类型是const sysop &。表示不能使用它返回的引用去修改它指向的引用。不能使用：use(looper).used = 10;

**将引用用于类对象：**（使用string类作为例子）

主要说明了不能修改const类型的变量，不能返回已经被销毁的变量的引用

**对象、继承和引用：**

派生类继承了基类的方法，意味着派生对象可以使用基类对象的方法。

基类引用可以指向派生类对象，而如需进行强制类型转换，可以定义一个接受基类引用作为参数的函数，调用该函数时，可以将基类对象作为参数，也可以将派生类对象作为参数。

何时使用引用参数：

使用引用参数的主要的原因有两个：

* 程序员能够修改调用函数中的数据对象
* 通过传递引用而不是整个数据对象，可以提高程序的运行速度。

使用引用、指针、按住传递的一些指导原则：

1.对于使用传递的值而不对其作修改的函数：

* 如果数据对象很小，如内置数据类型或者小型数据类型，就使用传值调用。
* 如果数据对象是数组，则使用指针，因为这是唯一的选择，并将指针声明为指向const的指针
* 如果数据对象是较大的结构，则使用const指针或const引用，以提高效率，节省空间和时间
* 如果数据对象是类对象，则使用const引用。类设计的语义常常要求使用引用。传递类对象参数的标准方式是按引用传递。

2.对于修改调用函数中数据的函数:

* 如果数据对象时内置数据类型，就使用指针。
* 如果数据对象是数组，只能使用指针。
* 如果数据对象是结构，则使用引用或指针
* 如果数据对象是类对象。则使用引用。

## 8.3默认参数

通过函数原型来设定默认值，如：char \*left(const char\*str , int i = 1); i的默认值为1

对于带参数列表的函数，必须从右向左添加默认值，也就是说，要为某个参数设置默认值，则必须为它右边所有的参数提供默认值，因为实参按照从左到右的顺序依次被赋给相应的形参，不能跳过任何参数。

int one(int n,int m =4,int j=5); //valid

int two(int n,int m=5,int j); //invalid

int three(int n=1,int m =4,int j=5); //valid

beers=one(2); //the same as: one(2,4,5);

beers=one(1,2); //the same as: one(1,2,5);

beers=one(1,2,3); //the same as: one(1,2,3);

注意：只有原型制定了默认值，函数定义与没有默认参数时完全相同。

## 8.4函数重载

函数多态（函数重载）能够使多个同名的函数存在，术语多态指的是有多种形式，因此函数多态允许函数可以有多种形式。术语函数重载指的是可以有多个同名的函数，因此对名称进行了重载。

**函数重载的关键是函数的参数列表，也称为函数特征标。**（并不是返回值，不同返回值不能够进行函数重载）

但是一些看起来彼此不同的特征表不能共存，如：double cube(double x); 和 double cube(double & x);

因为：double x = 3; cuble(3); 与这两个原型都匹配，系统不知道调用哪个原型，因此它们不允许共存。

**何时使用函数重载：**仅当函数基本上执行相同任务，但是使用不同形式的数据时，才采用函数重载

## 8.5函数模板

函数模板是通用的函数描述，也就是说，他们使用通用类型来定义函数。

template<class Any>

void Swap(Any &a,Any &b)

{

Any temp=a;

a=b;

b=temp;

}

template<typename Any>

void Swap(Any &a,Any &b)

{

Any temp=a;

a=b;

b=temp;

}

以上两者等价。第一行指出要建立一个模板，并将类型命名为Any。

调用的方式为：int i=1;int j=2;Swap(i,j); double x=1.0; double y=2.0; Swap(x,y);

编译器会自动生成函数的int版本和double版本，不需要程序员参与。

**重载的模板：**可以像重载常规函数定义一样重载模板定义。

template<class Any>

void Swap(Any a[],Any b[],int n)

{

Any temp;

for(int i=0;i<n;i++)

{

temp=a[i]

a[i]=b[i];

b[i]=temp;

}

}

template<class Any>

void Swap(Any &a,Any &b)

{

Any temp=a;

a=b;

b=temp;

}

int i=1;int k=2;Swap(i,k);

int d1[]={1,3};int d2={5,4};int n=3;Swap(d1,d2,n);

**显示具体化：**即，使用具体化来覆盖常规模板

由于C++允许将一个结构赋给另一个结构，如果Any是一个结构，假设为struct sysop，也可使用第一种方式，则结构中的所有成员都会被交换，但是如果只想交换结构中的name，则需要提供一个具体化函数定义（称为显示具体化）来覆盖常规模板。

1,第三代具体化

* 对于给定的函数名，可以有非模板函数、显示具体化模板函数、模板函数以及它们的重载版本
* 显示具体化的原型和定义，应该以template<>打头，并通过名称来指出类型。
* “具体化”将覆盖“常规模板”，而“非模板函数（正常函数）”将覆盖“具体化”和“常规模板”

非模板函数（正常函数）： void swap(sysop &, sysop&)

模板函数： template<Any> void swap(Any &, Any &)

具体化的模板函数 template<> void swap(sysop &, sysop &)、template<> void swap<sysop>(sysop &, sysop &)

2,早期的具体化方案不再说明

**实例化和具体化：**

隐式实例化：直接调用swap(x,j);导致编译器生成一个使用int类型的实例，叫做隐式实例化

显示实例化：template void swap<int>(int,int&)声明所需要的类型int，并加上关键字template，编译器会使用swap()

模板生成int类型的函数定义。不需要对其进行定义

显示具体化：template<> void swap(int &, int &)或等价声明：template<> void swap<int>(int &, int &)。意思是：不要使用swap()模板来生成函数定义，而应使用独立的、专门的函数定义显式地为int类型生成函数定义，这些原型需要程序员编写函数的定义。

声明:template<> void swap<int>(int &, int &);

定义:template<> void swap<int>(int &, int &){….实现具体过程}

注意：试图在同一个编程单元中使用同一种类型的显示实例化和显示具体化，将会出错

## 8.6总结

引用变量是一种伪装指针，主要被用作处理类和结构对象的函数的参数。

函数参数的默认值（定义时必须从右向左提供默认值，使用函数时，变量赋值从左往右）

函数的特征标是其函数参数列表，决定了是否可以重载。

函数模板是自动完成重载函数的过程。

# 第九章 内存模型和名称空间

## 9.1单独编译

#ifndef COORDIN\_H\_ #define COORDIN\_H\_ …… #endif

## 9.2存储持续性、作用域和链接性

自动存储持续性：在函数定义中声明的变量的存储持续性为自动的。函数结束后自动销毁

静态存储持续性：在函数定义外定义的变量和使用关键字static定义的变量的存储持续性都是静态的。

动态存储持续性：使用new操作符分配的内存一直存在，知道使用delete操作符将其释放或者程序结束为止，这种内存的存储持续性为动态，有时又称为自由存储。

作用域和链接：

自动变量的作用域是局部的，静态变量的作用域是局部还是全局取决于它是如何被定义的。

**自动存储持续性：**

在默认情况下，在函数中声明的函数参数和变量的持续性为自动，作用域是局部的，没有链接性。

函数内的局部变量可以暂时隐藏函数外的同名变量。当程序离开函数时，函数外的变量又从新可见。

自动变量和堆栈：堆栈是后进先出（LIFO），程序使用两个指针来跟踪堆栈，一个指针指向栈底（堆栈开始的位置），一个指针指向堆顶（下一个可用内存单元）。

寄存器变量register：C++也支持使用register关键字来声明局部变量，他的持续性为自动，作用域是局部，没有链接性。关键字register表示变量不存储在堆栈里，而是存储在cpu的寄存器中，以方便快速访问。register int ireg = 9;如果变量存储在寄存器中则没有内存地址，因此不能对它进行取地址操作，因此&ireg是错误的。

**静态持续变量：**

在默认情况下如果没有显式地初始化静态变量，编译器就将它设置为0，静态数组和结构将每个元素或成员的所有位都设置为0

int global = 1000;

double warming = 0.4;

static int one\_file = 50;

int main()

{ ….. return0; }

void update(double dt)

{

extern double warming;

warming++;

}

void local()

{ static double au=90；

double warming=8;

double a =::warming;

}

global的持续性为静态，作用域是整个文件，链接性为外部，即另外的文件也能访问它

one\_file的持续性为静态，作用域是整个文件，链接性为内部，即只有这个文件能访问它，其他文件不行。

1,静态持续性，外部链接性

extern duble warming; 表示使用extern对warming变量进行重新声明，该关键字的意思是，使用以前被外部定义的名称来使用该变量。C++提供了作用域解析操作符（::），当放在变量的前面时，表示使用该变量的全局版本。因此在local()中，::warming表示的是warming的全局版本，并不是局部变量。（其他文件如果需要使用warming都需要使用extern引用声明该变量）

2,静态持续性，内部链接性

static int one\_file = 50;表示one\_file只能在这个文件中使用，其他文件不能访问它。

3,静态存储持续性，无链接性

在函数内部定义的使用static修饰的变量，无链接性。

**说明符和限定符**

存储说明符：auto\register\static\extern\mutable（如果结构中某个成员被mutable修饰，则及时结构为const，该成员依然可以被修改，不受const限制）在同一个声明中不能使用多个说明符。

vc限定符：const（限定变量不能修改）\volatile（变量随时可能被改，每次取值都须访问变量，不能读缓存）

**const**：在默认情况下全局变量的链接性是外部的，但是const全局变量的链接性是内部的，在C++看来，全局const定义就像使用了static说明符一样。

const的内部链接性还意味着，每个文件都有自己的一组常量，而不是所有的文件共享一组常量，每个定义都是其所属文件私有的，这就是能够将常量定义放在头文件中（可以被任何文件引用）的原因

如果出于某种原因，程序员想要将const常量的链接性变成外部，可以使用extern来覆盖默认的内部链接性

extern const int aa= 100;这种情况下，必须在所有使用该常量的文件中用extern关键字来引用，并且初始化。

**函数和链接性**

C++不允许在一个函数内定义另一个函数，因此所有的函数的存储持续性都是静态的，在整个程序执行期间都存在。默认为链接性是外部的。但是有static修饰过，链接性即为内部的。必须同时在原型和函数定义中使用static关键字。

**语言链接性（language linkage）**

编译器执行名称矫正或名称修饰，为重载函数生成不同的符号名称。叫做语言链接性。

## 9.3布局new操作符

通常new负责在堆（heap）中找到一个能够满足要求的内存块。new操作符还有一个变型，称为布局new操作符，它让程序员能够指定要使用的位置。布局new操作符不能与delete配套使用，如果使用布局new操作符建立类对象，需要则必须显式调用析构函数，这将在12章中讲述。头文件为：#include<new>

#include<new>

struct chaff

{

char dross[20];

int slag;

};

char buffer1[50];

char buffer2[500];

int main()

{

chaff \*p1,\*p2;

int \*p3,\*p4;\*p5,\*p6

p1=new chaff;

p3=new int[20];

p2=new (buffer1) chaff;

p4=new (buffer2) int;

p5=new(buffer2) int;

p6=new(buffer(2)+sizeof(int));

……..

}

p2=new (buffer1) chaff; p4=new (buffer2) int;程序分别从buffer1、buffer2中分配空间给了两个指针。

p5会覆盖存储p4值的内存块，但是p6不会，会接着p4的地址重新分配内存.

## 9.4名称空间

声明区域：可以在其中进行声明的区域，其声明有效的区域。比如全局变量则声明区域为声明所在的文件，局部变量其声明区域为声明所在的代码块。

潜在作用域：声明点开始，到其声明区域的结尾。潜在作用域比声明区域小。变量必须定以后才能使用

作用域：变量对程序而言可见的区域是变量的作用域，变量并不是在其潜在作用域都可见的，可能会被嵌套声明区域中声明的同名变量所隐藏。

新的名称空间特性：通过定义一种新的声明区域来创建命名的名称空间，一个名称空间中的名称不会与另一个名称空间里的名称冲突，同时允许程序的其他部分使用该名称空间中声明的东西。

namespace Jack

{

double pail;

void fetch();

int pal;

struct Well{……};

}

namespace Jill

{

double bucket(double n){……};

void fetch();

int pal;

struct Hill{……};

}

名称空间是全局的，可以位于一个另名称空间中，但不能在代码块中。在默认情况下，名称空间中声明的名称的链接性是外部的。用户可定义名称空间，还有系统的全局名称空间。任何名称空间中的名称都不会与其他名称空间中的名称发生冲突。

通过作用解析操作符（::）来访问名称空间的名称：Jack::pal = 12;

**using声明和using编译指令**

我们不希望每次都使用作用解析操作符来访问名称空间里的名称，因此使用using声明和using编译指令来简化。

using声明由被限定的名称和它前面的关键字using组成：using Jill::fetch;

using编译指令使所有的名称可用：using namespace Jill;

**using声明和using编译指令的比较**

使用using编译指令导入一个名称空间中所有的名称，相当于大量使用作用域解析操作符。声明区域中依然可以定义同名的局部名称，局部名称将隐藏名称空间名。但是依然可以使用作用域解析操作符来调用。

使用using声明时，就像声明了相应的名称一样。不能在作用空间里定义同名变量。

嵌套名称空间namespace one{ namespace two{ int n;}}

使用方法：using namespace one::two;和using one::two::n;

未命名的名称空间namespace {int ice; int band;}

该名称空间中声明的名称的潜在作用域为：从声明点到声明区域末尾。由于没有名称，所以不能被其他程序调用。

C++标准不赞成在名称空间和全局作用域中使用关键字static….

不赞成：

static int counts; //全局静态变量

int others; //全局变量

int main(){…..}

推荐：

namespace {int counts;}

int others;

int main(){……}

**名称空间及其前途：**

指导原则：

使用在已命名的名称空间中的变量，而不是使用外部全局变量

使用在已命名的名称空间中的变量，而不是使用静态全局变量

导入名称时，首选使用作用解析操作符或using声明的方法

对于using声明，首选将其作用域设置为局部而不是全局

# 第十章 对象和类

重要的面向对象编程(opp)特性：抽象、封装和数据隐藏、多态、继承、代码的可重用性

## 10.1过程性编程和面向对象编程

面向对象编程，更注重从用户的角度考虑对象，描述对象所需的数据，描述用户与数据交互所需的操作，如何实现接口和数据存储，以及如何展示数据。

## 10.2抽象和类

在C++里面用户定义类型指的是实现抽象接口的类设计。

类型：决定了数据对象需要的内存数量；决定了如何解释内存中的位(如long和float内存中占用的位数相同，但是它们转换成数值的方法不同)；决定了使用数据对象执行的操作或方法（用户自定义或系统自带）。

**类：**是一种将抽象转化为用户数据类型的C++工具，它将数据表示和操控数据的方法组合成一个简洁的包。

#include <iostream>

#include <cstring>

class Stock //类的声明

{

private:

char company[30];

int shares;

double share\_val;

double total\_val;

void set\_tot(){total\_val=shares\*share\_val; }

public:

Stock(const char \*co,int mum=0,double price=0.0);

void acquire(const char \*co,int n,double pr);

void buy(int num,double price);

void sell(int num,double price);

void update(double price);

void show();

};

//下面是类的定义

Stock::Stock(const char \*co,int num,double price)

{

std::strncyp(company,co,29);

company[29]=’\0’;

if(n<0) shares=0;

else shares=n;

shre\_val=pr;

}

void Stock::acquire(const char \*co,int n,double pr)

{……}

void Stock::buy(int num,double price)

{……}

void Stock::sell(int num,double price)

{…..}

void Stock::update(double price)

{…..}

void Stock::show()

{…..}

**类规范有两个部分组成：**

**类声明：**以数据成员的方式描述数据部分，以成员函数（又称为方法）的形式描述公有接口。

**类方法定义：**定义如何实现类成员函数

**接口：**接口是一个共享框架，供两个系统（如两个类之间，或者计算机和打印机之间……）交互时使用。

类设计尽可能地将公有接口和实现细节分开。公有接口表示设计的抽象组件。将实现细节放在一起并将它们与抽象分开被称为**封装**。封装的另一个例子是：将类函数的定义和类声明放在不同的文件中。

**控制对成员的访问：**public、private。可以不必在类声明中使用关键字private，这是类对象的默认访问控制

类和结构：结构的默认访问类型是public，类的为private

实现类成员函数：

成员函数的定义和常规函数定义十分相似，也有返回类型和参数，它们还有两个特殊的特性：

定义成员函数时，使用作用域解析操作符（::）来表示函数所属的类 void Stock::update(double price)

类方法可以访问类的private组件

内联方法：其定义位于类声明中的函数都将自动成为内联函数，如果愿意也可以将类声明外定义的函数定义为内联函数，需要加上inline限定符。

内联函数的特殊规则要求在每个使用它们的文件中都对其进行定义。确保内联定义对多文件程序的所有文件都可用的、最简便的方法是：将内联定义放在定义类的头文件中。根据改写规则，在类声明中定义方法等同于用原型特换方法定义，然后在类声明的后面将定义改写为内联函数。因此定义位于类声明中的函数，和定义紧跟类声明后，并使用inline限定符的函数是等价的。

对象的方法使用

所创建的每个新对象都有自己的存储空间，用于存储其内部变量和类成员；但是同一个类的所有对象共享同一组类方法。例如，假设kate、jeo都是Stock的对象，则kate.shares\joe.shares占用的内存块不同。但是kate.show()\joe.show()都调用同一个方法。也就是说它们要执行同一个代码块，只是将这些代码用于不同的数据。

## 10.3类的构造函数和析构函数

struct thing

{ char \*pn;

int m; };

thing abc={“worry”,22}; //valid,结构的成员默认为public，可以直接赋值

stock hot={“shock”,200,50}; //invalid，类成员默认为private，不能直接访问

类构造函数，专门用于构造新对象，来初始化他们的数据成员。构造函数的名称与类名相同，构造函数的原型和函数头虽然没有返回值，但是没有被声明为void类型。实际上，构造函数没有声明类型。

不能将类成员名称用作构造函数的参数：

Stock::Stock(const char \*company,int shares,double share\_val) {…} //NO!

因为构造函数的参数表示的不是类成员，而是赋给类成员的值。因此不能相同，否则将出现shares=shares。。。

**使用构造函数：**

Stock food =Stock(“world cabbage”,250,1.25);

Stock food(“world cabbage”,250,1.25);

Stock \*pfood = new Stock(“world cabbage”,250,1.25);

可以使用对象来调用方法，food.show();但是不能使用对象来调用构造函数，因为在构造函数构造出对象之前，对象是不存在的。因此构造函数被用来创建对象，而不能通过对象来调用。

**默认构造函数：**

类声明为Stock food;时，如果类没有提供任何构造函数，则C++将自动提供默认构造函数Stock::Stock(){ }，但是当且仅当没有定义任何构造函数时，编译器才会提供默认构造函数。为类声明了构造函数后，程序员必须提供构造函数的定义（默认构造函数Stock::Stock(){}或者非默认构造函数Stock::Stock(const char \*co,int num,double price){}）。如果提供了非默认构造函数，但没有提默认构造函数，则声明为Stock food;将会出错。如果要创建对象，而不显式地初始化，则必须定义一个不接受任何参数的默认构造函数，定义默认构造函数可以有两种方式，一种是给已有的构造函数的所有参数提供默认值：Stock(const char \*co=”No name”,int mum=0,double price=0.0);，一个是，通过函数重载来定义另一个构造函数，一个没有任何参数的构造函数：Stock(); Stock::Stock(){….}

**析构函数**

用构造函数创建对象后,程序负责跟踪对象,直到其过期为止，对象过期时，程序自动调用析构函数，析构函数完成清理工作。和构造函数一样，析构函数的名称也很特殊，在类名前面加一个（~）符号。析构函数也没有返回类型和返回值。与构造函数不同的是，析构函数没有参数。~Stock(); Stock::~Stock(){ };

何时调用析构函数：这由便以其决定的。通常不应在代码中显式地调用析构函数。什么时候类对象过期，则程序会自动调用析构函数。程序必须有一个析构函数，如果程序员没有提供，则编译器将隐式地声明一个默认析构函数。

const成员函数

const Stock land=Stock(); land.show();

但是编译器会报错，因为show函数不能保证调用对象不被修改，但是land不能被修改。我们以前将函数参数声明为const引用或指向const的指针来解决这种问题，但是show()方法没有任何参数。想法，他所使用的对象都是由方法调用隐式地提供的。保证函数不会修改调用对象的另一种方法是：把show()声明称：void show() const;

这种方式声明和定义的类函数被称为const成员函数，在函数声明和定义中，在函数名后添加const。就像应尽可能将const引用和指针用作函数形参一样，只要类方法不修改调用对象，就应将其声明为const。

**构造函数和析构函数小结：**

如果构造函数只有一个参数,则将对象初始化为一个与参数的类型相同的值时,该构造函数将被调用.

如,构造函数原型为:Bozo(int age);则可以使用下面的任何一种形式来初始化对象：

Bozo one = Bozo(44); Bozo two(44); Bozo three =32;

接受一个参数的构造函数允许使用赋值句法来将对象初始化为一个值。

每个类只能有一个析构函数，而一个类可能有多个重载构造函数。如果构造函数式用了new，则必须提供使用delete的析构函数

## 10.4 this指针

this指针指向用来调用成员函数的对象，所有的类方法都将this指针设置为调用它的对象的地址。每个成员函数（包括构造函数和析构函数）都有一个this指针。this指针指向调用对象。如果方法要引用整个调用对象，则可以使用表达式\*this，因为this是对象的地址，而不是对象本身，即\*this（将解除引用操作符\*用于指针，将得到指针指向的值）。

## 10.5对象数组

声明对象数组的方法与声明标准数组的方法相同。可以分别用构造函数来初始化数组元素。

Stock mystuff[4];如果不进行初始化则系统会调用默认构造函数进行初始化。

Stock stock[10]={Stock(“Smart”,12,120),Stock(),Stock(“Momo”,13,10)};前三个按照规定的初始化，其他用默认构造函数

注意：初始化对象数组的方案是，首先使用默认构造函数创建数组元素，然后花括号中的构造函数将创建临时对象，然后将临时对象的内容复制到相应的元素中。因此，要创建类对象数组，则这个类必须有默认构造函数。

## 10.7类作用域

类定义的名称（如数据成员名和类成员函数名）的作用域都是整个类。因此作用域为整个类的名称，在类中是已知的，在类外是不可知的。因此，可以在不同的类中间使用同名的成员，不会冲突。外部也不能直接访问类成员，公有成员都不行。调用公有成员必须通过对象。

作用域为整个类的常量

类中不能声明const类型常量，因为const常量必须在声明时初始化，但声明类只是描述了对象的形式，并没有真正创建对象，因此没有用于存储的空间

在类中声明常量有两种方法：

第一种在类中声明一个枚举：

class Stock {

private:

enum{Len=30};

char company[len];

}

第二种使用关键字static（保存在栈中）

class Stock {

private:

static const int Len=30;

char company[len];

}

## 10.9总结

通常，将类声明分成两部分组成，这两部分通常保存在不同的文件中。类声明（包括由函数原型表示的方法）应放在头文件中。定义成员函数的源代码放在方法文件中。这样便于将接口描述与实现细节分开。

类是用户定义类型，对象是类的一个实例。

每个对象都存储自己的数据，而共享类方法。

如果方法需要显式地调用对象本身，则可以使用this指针，由于this指针表示的是对象的地址，\*this表示对象别名

# 第十一章 使用类

## 11.1操作符重载

一种形式的C++多态operator op(argument-list)

如：Time operator + (const Time &t) const;

Time Time::operator + (const Time &t) const

{

Time sum;

sum.minutes = minutes + t.minutes

sum.hours = hours + t.hours;

return sum;

}

为区分++操作符的前缀版本和后缀版本，C++将operator++最为前缀版本，将operator++( Time)作为后缀版本；其中的参数永远也不会被用到，所以不必指定其名称。

Time Time::operator ++{…} Time Time::operator ++(Time){…}

**重载限制**

1,重载的操作符不必是成员函数，但必须至少有一个操作数是用户定义类型，防止用户为标准类型重载操作符

2,使用操作符时不能违反操作符的原来的句法规则，不能改变操作符的优先级

3,不能定义新的操作符（C++原本不存在的操作符）

4,有些操作符不能被重载，有些操作符只能通过成员函数进行重载，大多数操作符都可以通过成员函数和非成员函数进行重载

**友元简介：**友元函数，友元类，友元成员函数

为类重载二元操作符时，如乘法操作符将一个Time和一个double类型结合在一起。

A=B\*2.45为：A=B.operate(2.45); 然而A=2.45\* B则不能转换。

有一种解决方式，非成员函数。原型为：Time operate\*(double m,const Time &t);则能使A=2.45\* B也能运算

但是常规非成员函数不能直接访问类的私有数据，但，有一类特殊的非成员函数可以访问私有成员，叫做友元函数

函数声明，放在类声明里面：friend Time operate\*(double m,const Time &t);

函数定义：由于他不是成员函数，所有不要使用Time::限定符，另外也不要在定义时使用关键字friend

friend Time operate\*(double m,const Time &t)

{….友元函数的访问权限与成员函数相同，都可以访问类的私有成员}

友元函数是否有悖于oop：只有类声明可以决定哪一个函数是友元，因此类声明仍然控制了那些函数可以访问私有数据。简而言之，类方法和友元只是表达类接口的两种不同机制。

如果要为类重载操作符，并将非类的一项作为其第一个操作数，则可以使用友元函数来反转操作数。

重载操作符，作为成员函数还是非成员函数。

对于很多操作符来说，可以选择使用成员函数或非成员函数来实现操作符重载。一般来说，非成员函数是友元函数，这样才能访问类的私有成员。

如：Time operate+ (const Time & t) const;和friend Time oprate+(const Time &t1, const Time &T2) const;

将转化为：T3=T1.operate(T2) T3=operate+(T1,T2)

在定义操作符的时候只能选择一种，因为这两种格式都是同一个表达式匹配，同时定义这两种格式将被视为二义性，导致编译错误。

## 11.6类的自动转换和强制类型转换(未看)

# 第十二章 类的动态分配内存

静态类成员特点:无论创建了多少对象,程序都只创建一个静态类变量副本。类的所有对象共享同一个静态成员static。

在类声明的时候不能对静态数据成员进行初始化，静态数据成员在类声明中声明，在包含类方法的文件中（类定义的时候）初始化。初始化时使用作用域操作符来指出静态成员所属的类，但是如果静态成员是整型或枚举型const，则可以在类声明中初始化。

在构造函数中使用new来分配内存是，必须在相应析构函数中使用delete来释放内存，如果使用new[]来分配内存，则应使用delete[]来释放内存。如：int a = new int(); delete a; 和 char\* a=new char[]; delete[] a;

如果函数用传值调用的方式调用类对象，则程序会生成一个临时变量，将类对象赋值给临时变量，并把临时变量传递给函数。

StringBad sailors=sports;等效于StringBad sailors=StringBad(sports);构造函数原型为：StringBad(const StringBad &);因为当构造函数只接受一个参数时，可以以赋值的方式初始化对象。

当你使用一个对象来初始化另一个对象时，编译器将自动生成上述构造函数（称为赋值构造函数，因为它创建对象的一个副本）。

**隐式成员函数**

C++自动提供了下面这些成员函数：默认构造函数、复制构造函数、赋值操作符、默认析构函数、地址操作符。

隐式地址操作符返回调用对象的地址。

**1.默认构造函数**：程序自动生成的构造函数，用户显式定义的没有参数的显式默认构造函数。用户定义的所有参数都有默认值的默认构造函数。以上三种都可以称为默认构造函数，并且程序只能有一个默认构造函数，如果后两种都在程序中定义，则会出现二义性，因为这样Class\_name a;将不知道调用哪个构造函数。

**2.复制构造函数：**复制构造函数用于将一个对象复制到新创建的对象中，它用于初始化过程中，而不是常规的赋值过程中。原型如下：Class\_name(const Class\_name &);它接受一个指向类对象的常量引用作为参数。

新建一个对象并将它初始化为同类现有对象时，复制构造函数将会被调用。以下四种声明都将调用复制构造函数

1.Class\_name a(b); 2.Class\_name a=b; 3. Class\_name a=Class\_name(b); 4.Class\_name \*pa=new Class\_name(b);

其中中间的两种可能会使用复制构造函数直接创建对象，也可能使用复制构造函数生成一个临时对象。然后将临时对象的内容赋给a，这取决于具体的实现，第4个声明使用b初始化一个匿名对象，并将新对象的地址赋给pa指针。

每当程序生成了对象副本时，编译器都将使用复制构造函数。具体来说，当函数按值传递对象或函数返回对象时，都将使用复制构造函数。按值传递意味着创建元是变量的一个副本，编译器生成临时对象是，也将使用复制构造函数。

功能、问题、解决方法：默认复制构造函数逐个复制非静态成员（成员复制也叫浅复制），复制的是成员的值，即如果类中包含了使用new初始化的指针成员，应当定义一个复制构造函数，以赋值指向的数据，而不是指针，这被称为深度复制，复制的另一种形式（成员复制或浅复制）只是复制指针值。浅复制仅浅浅的复制指针信息，而不会深入挖掘来复制指针引用的结构。

**3.复制操作符**：Class\_name & Class\_name::operator=(const Class\_name &);

C允许结构复制，而C++允许类对象复制，这个通过自动为类重载赋值操作符实现的。将已有的对象赋值给另一个对象时，将使用重载的赋值操作符：StringBad a(“I’m bad string.”); Stringbad b; b=a;

StringBad metoo = knot;这个是新创建的对象metoo，初始化为knot的值，因此使用复制构造函数，不过也可能会分两步来处理，使用复制构造函数创建一个临时变量，然后使用复制操作符将临时对象的值复制到新对象中。

与复制构造函数相似，复制操作符的隐式实现也对成员进行逐个复制，如果成员本身就是类对象，则程序将使用为这个类定义的复制操作符来复制该成员，但静态数据成员不受影响。

StringBad & StringBad::operator=(const StringBad &st)

{

if (this=&st) return \*this;

delete [] str;

len=st.len;

str = new char[len+1];

std::strcpy(str,st.str);

return \*this;

}

静态成员函数，不能通过对象来调用静态成员函数，甚至不能使用this指针。如果静态成员函数是在共有部分声明的，可以使用类名和作用域解析操作符来调用它。static int HowMany(){return num\_str;} int count=String::HowMany();

由于静态成员函数只与特定的对象相关联，因此只能使用静态数据成员，如静态方法HowMany只能访问静态成员num\_str，不能访问str len。

**在构造函数中使用new时应注意的事项**：

1.如果在构造函数中使用new来初始化对象的指针成员，则应在析构函数中使用delete。

2.new和delete必须相互兼容。new对应于delete[]

3.如果有多个构造函数，则必须以相同的方式使用new。要么都带中括号，要么都不带，因为只有一个析构函数，因此所有的构造函数都必须与它兼容。不过，可以在一个构造函数中使用new来初始化指针，而在另外一个构造函数中将指针初始化为空（NULL或0），这是因为delete（无论是带中括号还是不带中括号）都可以用于空指针。

应定义一个复制构造函数，通过深度复制将一个对象初始化为另一个对象。

具体来说，复制构造函数应分配足够的空间来存储复制的数据，并复制数据，而不仅仅是数据的地址，另外，还应更新所有受影响的静态类成员。

应当定义一个赋值操作符，通过深度复制将一个对象赋值给另一个对象。

当成员函数或独立的函数返回对象时，有几种返回方式可供选择，返回指向对象的引用，指向对象的const引用，指向对象的const对象。

如果方法或函数要返回局部对象，则应返回对象，而不是指向对象的引用。这种情况下将使用复制构造函数来生成返回的对象。如果方法或函数要返回一个没有公有复制构造函数的类（如ostream）的对象，它必须返回一个指向这种对象的引用。

返回const对象，则不能通过对象改变类的成员。

指针和对象：

使用常规表示法来声明指向对象的指针：String \*gl;

可以将指针初始化为指向已有的对象（不创建新对象）：String \*first=&saystring[0];

可以使用new来初始化指针，这将创建一个新的对象：String \*favorite =new String(saystring[choice]);

对类使用new将调用相应的类构造函数来初始化新创建的对象。

String \*a = new String; //默认构造函数

String \*b = new String(“bbb”); //String(const char \*);

String \*c = new String (saystring[0]); //String(const String &);

可以使用->操作符通过指针访问类方法：a->length();

可以对对象指针应用解除引用操作符（\*）来获得对象：String d = \*a;

**再谈new布局操作符**

程序员必须负责管理布局new操作符从中使用的缓冲区内存单元，防止两个变量的内存单元重叠。

如果使用布局new操作符来为对象分配内存，必须确保其析构函数被调用，布局new操作符不能与delete配合使用。例如，指针p3没有收到new操作符返回的地址，因此delete p3将导致运行阶段错误。因为new/delete系统知道已分配的字节块，但是对于new操作符对该内存块做了何种处理一无所知。解决方案是：显式地为使用布局new操作符的对象调用析构函数。显式调用析构函数时，必须指定要销毁的对象。对于使用布局new操作符创建的对象，应以与创建的顺序相反的顺序进行删除，原因在于，晚创建的对象可能依赖于早创建的对象，另外，仅当所有对象都被销毁后，才能释放用于存储这些对象的缓冲区。

#include<new>

class JustTesting

{

private:

string words;

int number;

public:

JustTesting (Const string &s = “JustTesting”,int n=0){ words =s; number =n;cout<<words<<” constructed;\n”;}

~ JustTesting(){cout<<words<<” destoryed;\n”;}

show()const{ cout<<words<<” , ”<<number<<endl;}

};

int main()

{

char\* buffer = new char[512];

JustTesting \*p1,\*p2;

p1=new (buffer) JustTesting;

p2=new JustTesting(“Heap1”,20);

p3= new (buffer+sizeof(JustTesting)) JustTesting(“BetterIdea”,6);

p4 = new JustTesting(“Heap2”,22);

…..

delete p2;delete p4;

p3->~ JustTesting();p1->~ JustTesting();

delete[] buffer;

}

**转换函数**（详细见11.6类的自动转换和强制转换,并未细看）：

要将单个值（type\_name）转换为类类型（c\_name），需要创建类构造函数：c\_name(type\_name value);

要将类类型（c\_name）转换为其他类型（type\_name），需要创建类构造函数：operator type\_name();虽然该函数没有声明返回类型，但是应返回所需类型的值。

在转换函数时要小心，可以在声明构造函数式使用关键字explicit，防止它被用于隐式转换。

在类中可以嵌套结构或者类声明。这样它们的作用域是内部的，只是在类中使用。

对于class中的const变量，不能再构造函数中给它赋值

Class Test

{

private:

const int size;

int & count;

…..

}

Test::Test(int a,int b):size(a),count(b)

{

size = a; //invalible

count = b; //invalible

}

由于调用构造函数时，对象将在大括号中的代码执行之前被创建。但是const 和引用对象必须在创建对象时被初始化，这样就会发生错误。因此，C++提供一种特殊的句法来完成上述工作，它叫做成员初始化列表。成员初始化列表由都好分隔的初始化列表组成完成（前面带冒号）。只有构造函数能够使用这种初始化列表句法。const类成员，被声明为引用的类成员必须使用这个句法，其他类成员也可以使用。

## 12.3总结

如果对象包含成员指针，同时他指向的内存是由new分配的。则释放用于保存对象的内存并不会自动释放对象成员指针指向的内存，因此在类构造函数中使用new类来分配内存时，应在类析构函数中使用delete来释放分配的内存。这样，当对象过期时，将自动释放其指针成员指向的内存。

# 第十三章 类继承

## **1**3.1公有派生

基类的公有成员将成为派生类的公有成员，基类的私有成员也将成为派生类的一部分，但是只能通过基类的公有和保护方法访问。

**派生类对象特征：**派生类对象存储了基类的数据成员（派生类继承了基类的实现）。派生类对象可以使用基类的方法（派生类继承了基类的接口）。派生类需要自己的构造函数。可以添加额外的数据成员和成员函数。

访问权限：派生类不能直接访问基类的私有成员，而必须通过基类方法进行访问。具体地说，派生类的构造函数必须使用基类的构造函数。创建派生类对象时，程序首先创建基类对象。可以理解为，将派生类对象初始化时，调用嵌套了基类构造函数的派生类构造函数。

RatedPlayer:: RatedPlayer(unsigned int r,const char\* fn,const char\* ln,bool ht):TableTennisPlayer(fn,ln,ht){ratubg =r;}

如果省略成员初始化列表RatedPlayer:: RatedPlayer(unsigned int r,const char\* fn,const char\* ln,bool ht){….}

又基类对象必须首先被创建，所以程序将使用默认的基类构造函数。

等效于：RatedPlayer:: RatedPlayer(unsigned int r,const char\* fn,const char\* ln,bool ht): TableTennisPlayer() {….}

除非要使用默认构造函数，否则应显示调用正确的基类构造函数。

## 13.2有关派生类构造函数的要点

基类对象首先被创建；派生类构造函数应通过成员初始化列表将基类信息传递给基类构造函数；派生类构造函数应初始化派生类新增数据成员；释放对象的顺序与创建对象的顺序相反，即首先执行派生类的析构函数，然后自动调用基类的析构函数。

派生类和基类的关系：派生类可以使用基类的非私有方法；基类指针可以在不进行显式类型转换的情况下指向派生类对象；基类引用可以在不进行显式转换时引用派生类对象，不过基类指针或引用只能调用基类方法。即派生类对象或地址可以赋值给基类引用或指针，但不可以将基类对象和地址赋给派生类引用和指针。此匹配是单向的。

## 13.3继承——is-a关系

**三种继承方式**：公有继承、保护继承、私有继承

公有继承：最常见的集成方式，它建立一种is-a关系，即派生类对象也是一个基类对象，可以对基类对象执行任何操作，也可以对派生类对象执行。这种关系称为is-a-kind-of，比如，苹果是一种水果。

## 13.4多态公有继承

如果希望同一个方法在基类和派生类中的行为是不同的，换句话说，方法的行为取决于调用该方法的对象，这种较复杂的行为称为多态——具有多种形态，就是指同一个方法的行为将随上下文而异。

两种重要的机制可以实现多态公有继承：**在派生类中重新定义基类的方法；使用虚方法**。

class Brass

{

private:

enum {MAX = 35};

char fullName[MAX];

long acctNum;

double balance;

public:

Brass(const char \*s = "Nullbody", long an = -1,

double bal = 0.0);

void Deposit(double amt);

virtual void Withdraw(double amt);

double Balance() const;

virtual void ViewAcct() const;

virtual ~Brass() {}

};

class BrassPlus : public Brass

{

private:

double maxLoan, rate ,owesBank;

public:

BrassPlus(const char \*s = "Nullbody", long an = -1,

double bal = 0.0, double ml = 500,

double r = 0.10);

BrassPlus(const Brass & ba, double ml = 500, double r = 0.1);

virtual void ViewAcct()const;

virtual void Withdraw(double amt);

void ResetMax(double m) { maxLoan = m; }

void ResetRate(double r) { rate = r; };

void ResetOwes() { owesBank = 0; }

};

Brass类在声明Withdraw()

Brass Piggy("Porcelot Pigg", 381299, 4000.00);

BrassPlus Hoggy("Horatio Hogg", 382288, 3000.00);

Brass &b1\_ref = Piggy;

Brass &b2\_ref = Hoggy;

如果ViewAcct()不是虚拟的，则:

b1\_ref. ViewAcct(); //use Brass:: ViewAcct()

b2\_ref. ViewAcct(); // use Brass:: ViewAcct()

如果ViewAcct()是虚拟的，则:

b1\_ref. ViewAcct(); //use Brass:: ViewAcct()

b2\_ref. ViewAcct(); // use BrassPlus:: ViewAcct()

虚函数的这种行为十分方便，因此，经常在基类中将派生类会重新定义的方法声明为虚方法，方法在基类中声明为虚拟的以后，它在派生类中将自动成为虚方法。

基类中声明了一个虚拟析构函数，这样做是为了确保释放派生对象时，按正确的顺序调用析构函数。

派生类的构造函数采用成员初始化列表的句法，将基类信息传递给基类构造函数。非构造函数不能使用成员初始化列表，但派生类可以调用共有的基类方法。示例代码如下：

BrassPlus::BrassPlus(const Brass & ba, double ml, double r)

: Brass(ba) // uses implicit copy constructor

{

maxLoan = ml;

owesBank = 0.0;

rate = r;

}

void BrassPlus::ViewAcct() const

{

Brass::ViewAcct(); // display base portion

cout << "Maximum loan: $" << maxLoan << endl;

cout << "Owed to bank: $" << owesBank << endl;

cout << "Loan Rate: " << 100 \* rate << "%\n";

}

**为何需要虚拟析构函数**：对于前一个示例，两个Brass的引用一个指向Brass对象，一个指向BrassPlus对象，如果析构函数不是虚拟的，则将只会调用Brass的析构函数，即使指针指向的是一个BrassPlus对象。如果析构函数是虚拟的，将调用相应对象类型的析构函数。使用虚拟析构函数能保证正确的析构函数序列被调用。

友元不能是虚函数，因为友元不是类成员，只有成员才能使虚函数。但是可以通过让友元函数使用虚拟成员函数来解决。

**虚函数注意事项：**

重新定义继承的方法，应确保与原来的函数类型完全相同，因为重新定义继承的方法并不是重载，也不是使用相同的函数特征标覆盖基类声明，而是隐藏同名的基类方法，不管参数特征标如何，但如果返回类型是基类引用或指针，则可以修改为指向派生类的引用或指针，这种特性被成为返回类型协变，因为允许返回类型随类类型的变化而变化。

如果基类声明被重载了，则应在派生类中重新定义所有的基类版本。不然另外的重载版本将被隐藏，派生类对象无法使用它们。

## 13.5访问控制：protected

关键字protected与private相似，在类外只能用公有类成员访问protected部分中的成员，protected与private的区别只有在基类派生的类中才会表现出来，派生类的成员可以直接访问基类的保护成员protected，但不能直接访问类的私有成员private。对外部世界来说，保护成员和私有成员相似，对派生类来说保护成员和公有成员相似。

## 13.6抽象基类

纯虚函数virtual void ViewAcct() const = 0; 纯虚函数声明的结尾处为 =0

当类声明中包含纯虚函数时，则不能创建该类的对象，只能用作基类。

**ABC函数**：真正的ABC必须包含一个纯虚函数。纯虚函数可以没有定义，但C++甚至允许纯虚函数有定义，但需要将这个类声明为抽象的（至少有一个纯虚函数）。可以将原型声明为虚拟的void Move(int nx,ny)=0;，仍可以在实现文件中提供方法的定义。派生类的函数可以调用它。

**ABC理念**：在设计ABC之前，首先应开发一个模型——指出变成问题所需要的类，以及它们之间的关系，基类不能被创建对象。 可以将ABC看作是一种必须实施的借口，ABC要求具体派生类覆盖其纯虚函数——迫使派生类遵循ABC所设置的接口规则。这种模型在基于组件的模式中很常见，在这种情况下，使得组件设计人员能够制定“接口约定”这样确保了从ABC派生的所有组件都至少支持ABC指定的功能。

## 13.7继承和动态内存分配

如果基类使用动态内存分配，并重新定义赋值和复制构造函数：

**1.派生类不使用new：**那么派生类可以使用默认析构函数，默认赋值函数，默认复制构造函数。因为，派生类都会自动调用基类的函数来完成派生类对象中的基类部分的操作，派生类对象成员的操作由默认函数来操作是合适的。

**2.派生类使用了new：**这种情况下，必须为派生类定义显式的析构函数、复制构造函数和赋值函数。派生类的析构函数、赋值操作符、复制构造函数都必须使用相应的基类方法来处理基类元素。这种要求是通过三种不同的方式来完成的；对于析构函数，这是自动完成的。对于构造函数，这是通过在初始化成员列表中调用基类的复制构造函数来完成的（如果不这样，将自动调用基类的默认构造函数），不需要在函数里对基类的对象赋值。对于赋值操作符，这是通过使用作用域解析操作符显式地调用基类的赋值操作符来完成的，必须给类的每个成员提供赋值操作符，不仅仅是新成员。

baseDMA & baseDMA::operator=(const baseDMA & hs)

{

If(this ==&hs) return \*this;

delete [] label;

label = new char[std::strlen(hs.label)+1];

std::strcpy(lable,hs.label);

rating = rs.rating;

return \*this;

}

hasDMA & hasDMA::operator=(const hasDMA & hs)

{

If(this ==&hs) return \*this;

baseDMA::operator=(hs);

style = new char[std::strlen(hs.style)+1];

std::strcpy(style,hs.style);

return \*this;

}

程序中，baseDMA::operator=(hs); //相当于使用基类的=操作符，把hs调用基类的=操作符执行：\*this=hs;将基类的成员按照基类的赋值操作符赋给了派生类对象

**使用动态内存分配和友元的继承范例**

hasDMA类（派生类）的友元 friend std::ostream & operator<<(std::ostream & os,const hasDMA & rs);

baseDMA类（基类）的友元 friend std::ostream & operator<<(std::ostream & os,const baseDMA & rs);

std::ostream & operator<<(std::ostream & os,const baseDMA & rs)

{

os<<”Label: ”<<rs.label<<endl;

os<<”Rating: ”<<rs.rating<<endl;

return os;

}

std::ostream & operator<<(std::ostream & os,const hasDMA & rs)

{

os<<(const baseDMA &)hs;

os<<”Style; ”<<hs.style<<endl;

return os;

}

os<<(const baseDMA &)hs因为友元不是成员函数，因此不能使用作用于解析操作符来指出要使用哪个函数，所以使用强制类型转换，以便匹配原型时能够选择正确的函数。

## 13.8类设计回顾

**编译器生成的成员函数：**

**1.默认构造函数**

默认构造函数要么没有参数，要么所有的参数都有默认值，如果没有定义任何构造函数，编译器将定义默认构造函数，让你能够创建对象。

自动生成的默认构造函数的另一项功能是，调用基类的默认构造函以及调用本身是对象成员所属类的默认构造函数。

如果派生类的构造函数的成员初始化列表没有显式地调用基类的构造函数，则编译器将调用基类的默认构造函数来构造派生类对象的基类部分，如果基类没有默认构造函数，将导致编译阶段错误。

如果定义了某种构造函数，编译器将不会定义默认构造函数，这种情况下，如果需要默认构造函数，必须自己提供。

提供构造函数的动机之一是确保对象总能被正确的初始化。另外，如果类包含指针成员，则必须初始化这些成员，因此，最好提供一个显式默认构造函数，将所有的类数据成员都初始化为合理的值。

**2.复制构造函数**

复制构造函数接受其所属类的对象作为参数，如：Class\_name(const Class\_name & );

使用复制构造函数的场合：将新的对象初始化为一个同一类对象；按值将对象传递给函数；函数按值返回对象；编译器生成临时对象。

如果程序没有使用（显式或隐式）复制构造函数，编译器将提供原型，但不提供函数定义；否则，程序将定义一个执行成员初始化的复制构造函数。也就是说，新对象的每个成员都被初始化为原始对象相应成员的值。

在某些情况下，默认复制构造函数的成员初始化是不合适的，如，使用new初始化的成员指针通常要求执行深复制，或者类可能包含需要修改的静态变量，上述情况，需要自己定义复制构造函数。

**3.赋值操作符**

默认的赋值操作符用于处理同类对象之间的赋值。不要将赋值和初始化混淆了。如果语句创建了新的对象，则使用初始化；如果语句修改已有对象的值，则是赋值。

如果需要显式定义复制构造函数，则基于相同的原因，也需要显式定义赋值操作符。

**其他的类方法**

**1.构造函数**

构造函数不同于其他类方法，因为它用于创建新的对象，而其他类方法只是被现有的对象调用。这是构造函数不被继承的原因之一。继承意味着派生类对象可以使用基类的方法，然而，构造函数在完成其工作之前，对象并不存在。

**2.析构函数**

析构函数不能被继承。释放对象时，程序将首先调用派生类的析构函数，然后调用基类的析构函数。如果基类有默认析构函数，编译器将为派生类生成默认析构函数。通常，对于基类，其析构函数应设置为虚拟的。

一定要定义显式析构函数来释放构造函数使用new分配的所有内存，并完成类对象所需的任何特殊的清理工作。对于基类，即时它不需要构造函数，也应提供一个虚拟析构函数

**3.转换**

使用一个参数就可以调用的构造函数定义了从参数类型到类类型的转换。例如，下述star类的构造函数原型：

Star(const char \*); //char \*转换为Star Star(const Spectral & , int members=1); // Spectral转化成Star

将可转换的类型传递给以类为参数的函数时，将调用转换构造函数。

Star north; north=”polaris”;第二条语句将调用Star::operator=(const Star \*)函数，使用Star::Star(const char\*)生成一个Star对象，该对象将被用作上述复制操作符函数的参数，这里假设没有定义将char\*赋给Star的赋值操作符。

在带一个参数的构造函数原型中使用explicit将禁止进行隐式转换，但仍允许显式转换：

Class Star

{…

public:

explicit Star(const char\*);

…};

Star north;

Star =”polaris”; //not allowed

Star =Star(“polaris”); //allowed

**4.按值传递对象与传递引用**

通常，编写使用对象作为参数的函数时，应按引用而不是按值传递对象。这样做的原因之一是为了提高效率。按值传递对象涉及到生成临时拷贝，即调用复制构造函数，然后调用析构函数。调用这些函数需要时间，复制大型对象比传递引用花费的时间要多得多。如果函数不修改对象，应将参数声明为const引用。

按引用传递对象的另外一个原因是，在继承使用虚函数时，被定义为接受基类引用参数的函数可以接受派生类

**5.返回对象和返回引用**

一些类方法返回对象，有些成员函数直接返回对象，而另一些则返回引用，有时必须返回对象，但如果可以步返回对象，则应返回引用，而不是返回对象。

返回对象涉及到生成返回对象的临时拷贝，这是调用函数的程序可以使用的拷贝。因此，返回对象的时间成本包括调用复制构造函数来声称拷贝所需的时间和调用析构函数删除拷贝所需的时间。返回引用可节省时间和内存，直接返回对象与按值传递对象相似：它们都生成临时拷贝。同样返回引用与按引用传递对象相似：调用和被调用的函数对同一个对象进行操作。

不过，并不总是可以返回引用，函数不能返回在函数中创建的临时对象的引用，因为当函数结束时，临时对象将消失，因此这种引用将是非法的，在这种情况下，应返回对象，以生成一个调用程序可以使用的拷贝。

通用规则是，如果函数返回在函数中创建的临时对象，则不要使用引用。如，加法后的返回值。

如果函数返回的是通过引用或指针传递给他的对象，则应按引用返回对象。

**6.使用const**

可以使用它来确保方法不修改参数：Star::Star(const char \*s){…}//函数不能改变s

可以使用const来确保方法不修改调用他的对象：void Star::show() const {…}这里const表示const Star \*this，而this表示调用的对象，表示不能使用show()函数修改对象。

通常，可以将返回引用的函数放在赋值语句的左侧，这实际上意味着可以将值赋给引用的对象，但可以使用const来确保引用或指针返回值不能用于修改对象中的数据：

const Stock & Stock::topval(const Stock & s) const

{

if(s.total\_val>total\_val) return s;

else return \*this;

}

该方法返回this或s的引用，因为都被声明为const，所以函数不能对它们进行修改，这意味着返回的引用也必须被声明为const

注意，如果函数将参数声明为指向const的引用或指针，则不能将该参数传递给另一个函数，除非后者也确保了参数不会被修改。

**公有继承的考虑因素**

3.赋值操作符

赋值操作符是不能继承的，原因很简单。派生类集成的方法的特征标与基类完全相同，但赋值操作符的特征标应该随着类的改变而改变的，这是因为他包含一个类型为其所属类的形参。

如果编译器发现程序将一个对象赋给同一个类的另一个对象，它将自动为这个类提供赋值操作符。不过如果对象属于派生类，编译器将使用基类赋值操作符来处理派生对象中基类部分的赋值。如果显式地为基类提供了赋值操作符，将使用该操作符，与此相似，如果成员是另一个类的对象，则对于该成员，将使用其所属类的赋值操作符。

# 第十四章 C++中的代码重用

**has-a关系**

has-a代表的是对象和它的成员的从属关系。如果你确定两件对象之间是has-a的关系，那么应该使用聚合；比如电脑是由显示器、CPU、硬盘等组成的，那么应该把显示器、CPU、硬盘这些类聚合成电脑类，而不是从电脑类继承。

## 14.1包含对象成员的类

//包含

class Student

{

private:

typedef std::valarray<double> ArrayDb;

std::string name; // contained object

ArrayDb scores; // contained object

….. }

//私有继承

class Student : private std::string, private std::valarray<double>

{

…..因为两个基类已经提供了所需的所有数据成员，所以私有继承提供了两个无名称的子对象成员。所以不需要私有数据了。

}

## 14.2私有继承

使用私有继承，基类的公有成员和保护成员都将成为派生类的私有成员，这意味着基类方法将不会成为派生对象的公有接口的一部分，但可以在派生类的成员函数中使用它们。

派生类不继承基类的接口，使用私有继承，类将继承实现。包含将对象作为一个命名的成员对象添加到类中，而私有继承将对象作为一个未被命名的继承对象添加到类中。我们将使用属于子对象来表示通过继承或包含添加对象。

因此私有继承提供的特性与包含相同：获得实现，但不获得接口。

class student : private std::string,private std::valarray<double>{….}

包含版本提供了两个被显式命名的对象成员，而私有继承提供了两个无名称的子对象成员。

**1.初始化基类组件**

隐式地继承组件而不是成员对象将影响代码的编写，因为再也不能使用name和scores来描述对象了。例如

包含将使用这样的构造函数：Student(const char \*str, const double \*pd, int n):name(str),scores(pd,n) { }

私有继承的构造函数使用类名来标识：

Student(const char \*str, const double \*pd, int n):std::string(str),std::valarray<double>(pd,n) { }

**2.访问基类的方法**

使用私有继承时，只能在派生类的方法中，通过作用于解析操作符来调用基类的方法。

//包含

double Student::Average() const

{

if (scores.size() > 0)

return score.sum()/scores.size();

else

return 0;

}

//私有继承

double Student::Average() const

{

if (ArrayDb::size() > 0)

return ArrayDb::sum()/ArrayDb::size();

else

return 0;

}

总之，使用包含时将使用对象名来调用方法，而使用私有继承时，将使用类名和作用于解析操作符来调用方法。

**3.访问基类对象**

如果需要使用基类对象本身，那么派生类的代码如何访问内部的基类的未命名对象呢？使用强制类型转换。

由于Student类是从string类派生而来的，因此可以通过强制类型转换，将Student对象转换为string对象；结果为继承而来的string对象。如下返回一个引用，该引用指向用于调用该方法的student对象中的继承而来的string对象。

const string & Student::Name() const { return (const string &) \*this; }

**4.访问基类的友元函数**

用类名显式地限定函数名不适合于友元函数，这是因为友元不属于类，不过可以通过显示地转换为基类来调用正确的函数。

ostream & operator<<(ostream & os, const Student & stu)

{

os << "Scores for " << (const string &) stu << ":\n";

return os;

}

**6.使用包含还是私有继承？**

大多数C++程序员倾向于使用包含，易于理解，不容易出错。

私有继承提供的特性比包含多，继承能使用基类的保护成员，包含不能使用类数据成员的保护成员

派生类可以重新定义虚函数，但包含类不能。使用私有继承，从新定义的函数将只能在类中使用，不是公有的。

通常，应使用包含来建立has-a关系；如果新类需要访问原有类的保护成员，或需要重新定义虚函数，则应使用私有继承。

**7.保护继承**

class Student : private std::string, private std::valarray<double> {….}

保护继承是私有继承的变体，使用保护继承时，基类的公有成员和保护成员都将成为派生类的保护成员。和私有继承一样，基类的接口在派生类中也是可用的，但在继承层次结构之外是不可用的。使用私有继承时，第三代类不能使用基类的接口，因为基类的共有方法在派生类中变成了私有方法。使用保护继承，第三代类可以在类中使用基类的接口，因为基类的公有方法在派生类中变成了保护方法，因此第三代类可以使用它们。

|  |  |  |  |
| --- | --- | --- | --- |
| *特征* | *公有继承* | *保护继承* | *私有继承* |
| *公有成员变成* | 派生类的公有成员 | 派生类的保护成员 | 派生类的私有成员 |
| *保护成员变成* | 派生类的保护成员 | 派生类的保护成员 | 派生类的私有成员 |
| *私有成员变成* | 只能通过基类接口访问 | 只能通过基类接口访问 | 只能通过基类接口访问 |
| *能否隐式向上转换* | 能 | 能（只能在类内） | 不能 |

**8.使用using重新定义访问权限**

使用保护派生或私有派生，基类的公有成员将成为保护成员或私有成员。假设要让基类的方法在派生类外面可用，方法之一是定义一个使用该基类方法的派生类方法；外一种方法是，将函数调用包装在另一个函数调用中，即使用一个using声明（就像名称空间一样）来指出派生类可以使用特定的基类成员，即使采用的是私有派生。

//包含和私有继承、保护继承都可以使用

double Student::sum() const

{

return valarray<double>::sum();

或 return score.sum();

}

这样便能调用Student.sum();

//不适合包含，只适合继承。using声明只是用成员名，没有圆括号、函数特征标和返回类型。

class Student : private std::string, private std::valarray<double>

{…..

public:

using std::valarray<double>::min;

using std::valarray<double>::max;

……}

## 14.3多重继承（MI）

MI可能会给程序员带来许多新问题：从两个不同的基类继承同名方法；从两个或更多相关基类那里继承同一个类的多个实例。

如果Worker派生出Singer和Waiter，且Singer、Waiter共同派生出了SingingWaiter。Singer和Waiter都继承了Worker，这样SingingWaiter ed; Worker \*pw = &ed;这样，ed中包含两个Worker对象，有两个地址可以选择，所以必须使用类型转换来指定对象：Worker \*pw = (Waiter \*)&ed; 或 Worker \*pw = (Singer \*)&ed;

**1.虚基类vitual**

虚基类实得从多个类（它们的基类相同）派生出的对象只继承一个基类对象，例如，通过在类声明中使用关键字**virtual**，可以是Worker被用作Singer和Waiter的虚基类（virtual和public的次序无关紧要）

class Waiter : virtual public Worker{….} class Singer: public virtual Worker{….}

class SingingWaiter : public Singer, public Waiter{….}

现在SingingWaiter对象只包含Worker对象的一个拷贝。从本质上来说，继承的Singer和Waiter对象共享一个Worker对象，而不是各自引入自己的Worker对象拷贝。因为SingingWaiter现在只包含了一个Worker子对象，所以可以使用多态。

**2.新的构造函数规则**

使用虚基类时，需要对类构造函数采用一种新的方法。对于非虚基类，唯一可以出现在初始化列表中的构造函数是即使基类构造函数，但这些构造函数可能需要将信息传递给其基类

SingingWaiter(const Worker &wk,int p=0,int v=Singer::other):Waiter(wk,p),Singer(wk,v) { }

存在的问题是，自动传递信息时，将通过2条不同的途径（Waiter、Singer）将wk传递给Worker对象。为了避免这种冲突，C++在基类是虚拟的时，禁止信息通过中间类自动传递给基类。不过编译器必须在构造派生对象之前构造基类对象组件；在上述情况下，编译器将使用Worker的默认构造函数。

如果不希望默认构造函数来构造虚基类对象，则需要显式地调用所需的基类构造函数。因此，构造函数应该为：

SingingWaiter(const Worker &wk,int p=0,int v=Singer::other):Worker(wk),Waiter(wk,p),Singer(wk,v) { }

上述代码将显式地调用构造函数Worker(const Worker &)请注意，对于虚基类，这种用法是合法的，必须这样做；对于非虚基类，则是非法的。

如果类有间接虚基类，则除非只需使用该虚基类的默认构造函数，否则必须显式地调用该虚基类的某个构造函数。

**关于基类们的同名方法**

比如没有在singingWaiter中重新定义show方法，又，每个直接祖先里都有一个show方法，多重继承可能导致函数调用的二义性。可以使用作用域解析操作符来确定调用它：singingWaiter sw; sw.Singer::show()；更好的方法是在singingWaiter中重新定义show方法；

在重新定义的时候，任何一个基类组件都不能被忽略void singingWaiter::show(){ Singer::show(); Waiter::show(); }

这时候递增方式将不适合，可以使用模块化方式，即提供一个只显示Worker组件的方法和一个只显示Waiter组件或Singer组件（而不是Waiter和Singer组件）的方法，然后在singingWaiter::show()方法中将组件组合起来

void Worker::Data() const

{

cout << "Name: " << fullname << endl;

cout << "Employee ID: " << id << endl;

}

void Waiter::Data() const

{

cout << "Panache rating: " << panache << endl;

}

void Singer::Data() const

{

cout << "Vocal range: " << pv[voice] << endl;

}

void SingingWaiter::Data() const

{

Singer::Data();

Waiter::Data();

}

void SingingWaiter::Show() const

{

cout << "Category: singing Waiter\n";

Worker::Data();

Data();

}

下面介绍其他一些有关MI的问题

1.混合使用虚基类和非虚基类

当类通过多条虚拟途径和非虚拟途径继承某个特定的基类时，该类将包含一个表示所有的虚拟途径的基类子对象和分别表示各条非虚拟途径的多个基类子对象。

2.虚基类和支配

派生类中的名称优先于直接或间接基类中的同名名称。

如果某个名称优先于其他所有名称，则使用它时，即使不使用限定符（作用域解析操作符），也不会导致二义性。

MI小结

首先复习一下不使用虚基类的MI，这种形式的MI不会引入新的规则，不过如果一个类从两个不同的类那里继承了两个同名的成员，则需要在派生类中使用类限定符来区分它们。否则，编译器将指出二义性

如果一个类通过多种途径继承了一个非虚基类，则该类从每种途径分配继承非虚基类的一个实例。在某些情况下，这可能正是希望的，但是通常，多个基类实例都是问题。

当派生类使用关键字virtual来指示派生时，基类就成为虚基类。主要变化为：从虚基类的一个或多个实例派生而来的类将只继承了一个基类对象。为了实现这种特性，必须满足其他要求：

有间接虚基类的派生类包含直接调用间接基类构造函数的构造函数，这对于间接非虚基类来说是非法的；

通过优先选择解决名称二义性。

正如看到的。MI会增加编程的复杂程度，不过，这种复杂性主要是由于派生类通过多条途径继承同一个基类引起的，避免这种情况后，唯一需要注意的是，在必要时对继承的名称进行限定。

## 14.4 类模板

template <class Type>

class Stack

{

private:

enum {MAX = 10};

Type items[MAX];

int top;

public:

Stack();

bool isempty();

bool isfull();

bool push(const Type & item);

bool pop(Type & item);

};

template <class Type>

Stack<Type>::Stack()

{

top = 0;

}

template <class Type>

bool Stack<Type>::isempty()

{

return top == 0;

}

这里使用class并不意味着Type是一个类，而只是表Type是一个通用的类型说明符，在使用模版时，将使用实际的类型替换他，可以使用不太容易混淆的关键字typename代替class

如果编译器实现了新的export关键字，则可以将模版方法定义放在独立的文件中，条件是每个模版声明都以export开头：export template <class Type> class Stack {….}，将方法定义放在源代码文件中，包含头文件，保证声明可用。

使用模版类：

仅在程序包含模版并不能生成模版类，必须请求实例化。为此，不要声明一个类型为模版类的对象，实例化时使用所需的具体类型替换通用类型名。

Stack<int> si; Stack<string> ss;

## 14.5总结

公有继承能够建立is-a关系，这样派生类可以重用基类的代码。私有继承和保护继承也是的能够重用基类的代码，但是建立的是has-a的关系。使用私有继承时，基类的公有成员和保护成员将成为派生类的私有成员；使用保护继承时，基类的公有成员和保护成员将成为派生类的保护成员。无论是私有继承还是保护继承，基类的公有接口都将成为派生类的内部接口。这有时候被称为集成实现，但并不集成接口，因为派生类对象不能显式地使用基类的接口，因此不能将派生对象看作是一种基类的对象。由于这个原因，私有继承和保护继承在不进行显示类型转换的情况下，基类指针或引用将不能指向派生类对象。

还可以通过开发包含对象成员的类来重用类代码，这种方法被成为包含、层次化或组合，它建立的也是has-a关系。与私有继承和保护继承相比，包含更容易实现和使用，所以通常优先采用这种方式。不过，私有继承和保护继承与包含有一些不同的功能。例如，继承允许派生类访问基类的保护成员；还允许派生类重定义从基类那里继承的虚函数。因为包含不是继承，所以通过包含来重用类代码时，不能使用这些功能。另一方面，如果需要使用某个类的几个对象，则使用包含更合适。

多重继承（MI）使得能够在类设计中重用多个类的代码。私有MI或保护MI简历has-a关系，而公有MI建立is-a的关系。MI会带来一些问题，即多次定义同一个名称，继承多个基类对象，可以使用类限定符来解决名称二义性问题，使用虚基类来避免继承多个基类对象的问题，但使用虚基类后，就需要为编写构造函数初始化列表以及解决二义性问题引入新的规则。

类模版实得能够创建通用的类设计，其中类型（通常是成员类型）有类型参数表示。可以用typename代替class类定义（实例化）在声明类对象并指定特定类型是生成。

class Ic<short> sic;//隐式实例化，类名为Ic<short>，Ic<short>成为模版具体化

template class Ic<int>;//显示实例化，编译器将声明称一个int具体化——Ic<int>虽然尚未请求这个类的对象

template<> class Ic<char\*>//显示具体化—覆盖模版定义的具体类声明，从新声明。类型参数为char\*将使用专用定义。

类模版可以被部分具体化：template<class T> Pals<T, T, 10>{…}当n的值为10时，建立了一个具体化。

模版类可用作其他类、结构和模版的成员。

所有这些机制的目的都是为了让程序员能够重用经过测试的代码，而不用手工复制它们。这样可以简化编程工作，提供程序的可靠性。

# 第十五章 友元、异常和其他

## 15.1友元

类并非只能拥有友元函数，也可以将类作为友元，在这种情况下，友元类的所有方法都可以访问原始类的私有成员和保护成员。另外，也可以做更严格的限制，只将特定的成员函数指定为另一个类的友元。哪些函数、成员函数或类作为友元，只能是由类定义。而不能在外部加强友情。

比如电视机TV类，和遥控器Remote类，既不是is-a的关系，也不是has-a的关系，事实上，遥控器可以改变电视机状态，则表明应将Remote类作为TV类的一个友元。

class Tv

{

public:

friend class Remote; //友元类

….

class Tv

{

public:

//友元函数

friend void Remote::set\_chan(Tv & t,int c); ….

**友元成员函数**

**共同的友元：**函数需要访问两个类的私有数据，它可以是一个类的成员，同时是另一个类的友元，不过又是将函数作为两个类的友元更合理。

## 15.2嵌套类

在C++中，可以将类声明放在另一个类中，在另一个类中声明的类被称为嵌套类。它通过新的类型类作用域来避免名称混乱。包含类的成员函数可以创建和使用被嵌套类的对象；而仅当声明位于公共部分才能在包含类的外面使用嵌套类，而且必须使用作用域解析操作符。

对类进行嵌套与包含并不同，包含意味着将类对象作为另一个类的成员，而对类进行嵌套不创建类成员，而是定义了一种类型，该类型仅在包含嵌套类声明的类中有效

**嵌套类和访问权限**

有两种访问权限适合嵌套类。首先，嵌套类的生命位置决定了嵌套类的作用域，即他决定了程序的哪些部分可以创建这种类对象。其次，和其他类一样，嵌套类的公有部分、保护部分和私有部分控制了外部对类成员的访问，在哪些地方可以使用嵌套类以及如何使用嵌套类，取决于作用域和访问控制。

1.作用域：如果嵌套类在另一个类的私有部分声明。只有那个包含类知道它。如果在保护部分声明，包含类和它所派生的类知道它。如果在共有部分声明，外部也知道它，可以使用类限定符来访问。

2.访问控制

类可见后，其决定作用的将是访问控制，包含类对嵌套类访问权的控制规则与对常规类相同。就是包含类对象只能显式地访问嵌套类对象的公有成员。

## 15.3异常

调用abort()

返回错误码

**异常机制：**

C++异常是对程序运行过程中发生的异常情况的一中相应。异常提供了将控制权从程序的一个部分传递到另一个部分的途径。对异常处理有三个组成部分：引发异常；捕获有处理程序的异常；使用try块。

throw语句实际上是跳转，即命令程序跳到另一条语句，throw关键字表示引发异常，紧随其后的值（例如字符串或对象）指出了异常的特征。

程序使用异常处理程序来捕获异常，处理程序以关键字catch开头。随后是位于括号中的类型声明，它指出了异常处理程序要响应的一场类型，然后是用一个花括号括起来的代码块，指出要采取的措施。catch关键字和异常类型用作标签，指出当异常被引发时，程序应跳到这个位置执行，异常处理程序也被称为catch块。

try块表示其中特定的异常可能被激活的代码块，它后面跟一个或者几个catch块，try块是由关键字指示的，关键字try的后面是一个由花括号括起的代码块，表明需要注意这些代码引发的异常。

try { // start of try block

z = hmean(x,y);

} // end of try block

catch (const char \* s) { // start of exception handler

std::cout << s << std::endl;

std::cout << "Enter a new pair of numbers: ";

continue;

}

double hmean(double a, double b)

{

if (a == -b)

throw "bad hmean() arguments: a = -b not allowed";

return 2.0 \* a \* b / (a + b);

}

将对象用作异常类型

exception类：C++可以把它用作其他异常类的基类，代码可以引发exception异常，也可以将exception类用做基类

1.stdexcept异常类

2.bad\_alloc异常和new

异常、类和继承

## 15.4 RTTI

RTTI是运行阶段类型识别的简称，旨在为程序在运行阶段确定对象的类型提供一种标准方式。

RTTI工作原理

C++有3个支持RTTI的元素：

1.dynamic\_cast操作符将使用一个指向基类的指针来生成一个指向派生类的指针；否则，该操作符返回0—空操作符。

class Grand{//has virtual methods..}

class Superb : public Grand{…}

class Magnificent : public Superb{…}

dynamic\_cast句法：Superb \*ps = dynamic\_cast<Superb \*>(pg)

ps是否能被安全地转换为Superb \*，如果可以，操作符将返回对象地址，否则返回一个空指针。

一般如果指向的对象（\*pt）的类型为Type或者是从Type直接或简介派生而来的类型，

则表达式：dynamic\_cast<Type\*>(pt)将指针pt转换为Type类型的指针；否则，结果为0，即空指针。

typeid操作符返回一个指出对象的类型的值。

type\_info结构存储了有关特定类型的信息。

只能将RTTI用于包含虚函数的类层次结构，原因在于只有于对这种类层次结构，才应该将派生对象的地址赋给基类指针。

# 第十六章 string类和标准模板库

## 16.1 string类

**1.构造字符串（string的构造函数）**

|  |  |  |
| --- | --- | --- |
|  | **名称** | **说明** |
| 公共方法.NET Compact Framework 提供支持XNA Framework 提供支持 | [String(Char\*)](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.chs/fxref_mscorlib/html/e79a6ceb-4a34-8525-8839-9d9054db8abc.htm) | 将 [String](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.chs/fxref_mscorlib/html/3e108182-236f-5ccb-b5ee-e91a6d09cea0.htm) 类的新实例初始化为由指向 Unicode 字符数组的指定指针指示的值。 |
| 公共方法.NET Compact Framework 提供支持XNA Framework 提供支持 | [String(array<Char>[]()[])](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.chs/fxref_mscorlib/html/5ac3c857-6fec-5f2f-2a67-b0ecc46b45ac.htm) | 将 [String](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.chs/fxref_mscorlib/html/3e108182-236f-5ccb-b5ee-e91a6d09cea0.htm) 类的新实例初始化为由 Unicode 字符数组指示的值。 |
| 公共方法.NET Compact Framework 提供支持XNA Framework 提供支持 | [String(SByte\*)](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.chs/fxref_mscorlib/html/73d76e47-5f24-12cc-eb18-4bf51f561661.htm) | 将 [String](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.chs/fxref_mscorlib/html/3e108182-236f-5ccb-b5ee-e91a6d09cea0.htm) 类的新实例初始化为由指向 8 位有符号整数数组的指针指示的值。 |
| 公共方法.NET Compact Framework 提供支持XNA Framework 提供支持 | [String(Char, Int32)](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.chs/fxref_mscorlib/html/7499a13a-8acb-848c-6387-6cb791eb0122.htm) | 将 [String](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.chs/fxref_mscorlib/html/3e108182-236f-5ccb-b5ee-e91a6d09cea0.htm) 类的新实例初始化为由重复指定次数的指定 Unicode 字符指示的值。 |
| 公共方法.NET Compact Framework 提供支持XNA Framework 提供支持 | [String(Char\*, Int32, Int32)](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.chs/fxref_mscorlib/html/eee1ce67-42c8-d86a-8325-f3cccb997593.htm) | 初始化为由指向 Unicode 字符数组的指定指针、该数组内的起始字符位置和一个长度指示的值。 |
| 公共方法.NET Compact Framework 提供支持XNA Framework 提供支持 | [String(array<Char>[]()[], Int32, Int32)](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.chs/fxref_mscorlib/html/8faec6a3-2e2b-2f91-c001-11ebe2c15293.htm) | 初始化为由 Unicode 字符数组、该数组内的起始字符位置和一个长度指示的值。 |
| 公共方法 | [String(SByte\*, Int32, Int32)](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.chs/fxref_mscorlib/html/690404a3-6e8c-e615-71e9-75fe77bc5bc1.htm) | 初始化为由指向 8 位有符号整数数组的指定指针、该数组内的起始字符位置和一个长度指示的值。 |
| 公共方法 | [String(SByte\*, Int32, Int32, Encoding)](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.chs/fxref_mscorlib/html/12cd41f0-4a60-fd5b-3bc2-bbec90fb0686.htm) | 初始化为由指向 8 位有符号整数数组的指定指针、该数组内的起始字符位置、长度以及 [Encoding](ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.chs/fxref_mscorlib/html/c590603b-8901-253c-78bf-171950a57438.htm) 对象指示的值。 |

**2.string类输入**

对于C风格字符串有三种方式

char info[100];

cin>>info;

cin.getline(info,100); //固定大小

cin.get(info,100);

对于string对象有两种方式。

string str;

cin>>str;

getline(cin,str); //可自动调节str大小

两个版本的getline()都有一个可选参数，制定使用那个自负来确定输入边界cin.getline(info,100,’;’); getline(cin,str,’;’);

string是个类，这是操作符重载，如：getline(cin,str); operator>>(cin,str);

**3.使用字符串**

length()和size()成员函数都返回字符串中的字符数。

string::nops是字符串可存储的最大字符数，通常是无符号int或无符号long的最大取值

|  |  |
| --- | --- |
| **方法原型** | **描述** |
| size\_type find (const string &str, size\_type pos=0) const | 从字符串的pos位置开始查找子字符串str，如找到则返回子字符串首次出现时其首字符的索引，否则返回string::nops |
| size\_type find (const char\* s, size\_type pos=0) const | 从字符串的pos位置开始查找子字符串s，如找到则返回子字符串首次出现时其首字符的索引，否则返回string::nops |
| size\_type find (const char\* s, size\_type pos=0,size\_type n) | 从字符串的pos位置开始查找子字符串s的前n个字符组成的子字符串，如找到则返回子字符串首次出现时其首字符的索引，否则返回string::nops |
| size\_type find (char ch, size\_type pos=0) const | 从字符串的pos位置开始查找字符ch，如找到则返回字符ch首次出现时其首字符的索引，否则返回string::nops |

string库还提供了相关方法，rfine()查找子字符串或字符最后出现的位置、find\_first\_of()在字符串中查找参数中任何一个字符首次出现的位置、find\_last\_of()在字符串中查找参数中任何一个字符最后依次出现的位置、find\_first\_not\_of()在字符串中查找第一个不被包含在参数中的字符、find\_last\_not\_of()字符串中查找最后一个不被包含在参数中的字符

**string还提供了哪些功能**

附录F

**16.2 auto\_ptr类**

auto\_ptr是一个模板类，用于管理动态内存分配的用法。（头文件：memory）

**使用auto\_ptr**

auto\_ptr模板定义了类似指针的对象，可以将new获得（直接或间接）的地址赋给这种对象。当auto\_ptr过期时，其析构函数将使用delete来释放内存。因此如果将new返回的地址赋给auto\_ptr对象时，无需记住稍后释放内存。

template <class X> class auto\_ptr{

public:

explicit auto\_ptr (X\* p = 0)throw(); // throw()表示构造函数不引发异常

….} //explicit表示构造函数是显式的，不支持隐式类型转换

将使用new的函数转换为使用auto\_ptr的函数：1.包含头文件memory；2.将指向string的指针替换成指向string的auto\_ptr对象；3.删除delete语句。

auto\_ptr <double> pd (new double); auto\_ptr <string> pd (new string); auto\_ptr <string> pd (new string(str));

new double 是new返回的指针，指向新分配的内存块。它是auto\_ptr<double>构造函数的参数，即它是对应于原型中形参p的实参。

**有关auto\_ptr的注意事项**

只能对new分配的内存使用auto\_ptr对象，而不要对由new[]分配的，或通过声明变量分配的内存使用它。

auto\_ptr <string> pd (new string(“It’s my bag.”));

auto\_ptr <string> lily;

lily = pd;

两个指针将指向同一个string对象，其中一个是另一个的拷贝。上述赋值语句不可接受，因为当ps和lily都过期时，程序将试图删除同一个对象两次。

要避免这种问题可以有多种方法：

1.定义复制操作符，使之执行深复制。

2.建立所有权概念，对于特定的对象，只有一个智能指针可以拥有它。智能指针的构造函数只能删除该智能指针拥有的对象。并使复制操作转让所有权，这就是用于auto\_ptr的策略。（但是如果所有权被转让，则前一个对象将不再引用该字符串。）

3.创建智能更高的指针，跟踪引用特定对象的智能指针数，这被成为引用计数，仅当最后一个指针过期时，delete才被调用。

## 16.3 STL

STL提供了一组表示容器、迭代器、函数对象和算法的模板。STL容器是同质的，即存储的值的类型相同；算法是完成特定任务的处方；迭代器是用来遍历容器的对象，与遍历数组的指针相似，是广义指针；函数对象是类似于函数的对象，可以是类对象或函数指针（包括函数名，因为函数名被用作指针）。STL不是面向对象的编程，而是一种不同的编程模式——通用编程技术。

vector模板类

**分配器**

与string类相似，各种STL容器模板都接受一个可选的模板参数，该参数制定使用哪个分配器对象来管理内存。例如，vector模板的开头与下面的类似

template <class T, class Allocator = allocateor<T> >

class vector {…..

如果省略该模版参数的值，则容器模板将默认使用allocator<T>类，这个类以标准方式使用new和delete

## 16.4通用编程技术

通用编程技术旨在编写独立于数据类型的代码。在C++中完成通用程序的工具是模板。

**迭代器：**是一个广义指针。事实上，它可以使指针，也可是是一个可以对其执行类似指针操作——如解除引用（\*p）和递增（p++）——的对象。每个容器类都定义了一个合适的迭代器，该迭代器的类型是一个名为interator的typedef，其作用域为整个类。

vector<double>::iterator pd; //定义一个迭代器

vector<double> scores; //定义一个double类型的容器

pd = scores.begin(); //让pd指向scores的第一个元素

\*pd = 22.3 //将pd指向的第一个元素的值设为22.3

++pd; //让pd指向下一个元素

**为何需要迭代器：**

模板使得算法独立于存储的数据类型，而迭代器是算法独立于使用的容器类型。模板还是和一个特定的数据结构（数组、链表…）关联在一起。

迭代器类型：输入迭代器、输出迭代器、正向迭代器、双向迭代器、随机访问迭代器。

**输入迭代器**

输入是从程序的角度来说的，即来自容器的信息被视为输入，就像来自键盘的信息对程序来说是输入一样，输入迭代器可被程序用作读取容器中的信息。只使用++来遍历容器信息。只读。

输出迭代器

将信息从程序传递给容器的迭代器叫做输出迭代器。用++来遍历容器信息。只写。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 迭代器功能 | 输入 | 输出 | 正向 | 双向 | 随机访问 |
| 解除引用读取 | 有 | 无 | 有 | 有 | 有 |
| 解除引用写入 | 无 | 有 | 有 | 有 | 有 |
| 固定和可重复排序 | 无 | 无 | 有 | 有 | 有 |
| ++i 、 i++ | 有 | 有 | 有 | 有 | 有 |
| --i 、 i-- | 无 | 无 | 无 | 有 | 有 |
| i[n] | 无 | 无 | 无 | 无 | 有 |
| i+n | 无 | 无 | 无 | 无 | 有 |
| i-n | 无 | 无 | 无 | 无 | 有 |
| i+=n | 无 | 无 | 无 | 无 | 有 |
| i-=n | 无 | 无 | 无 | 无 | 有 |

**容器概念**

容器是存储其他对象的对象。被存储的对象必须是同一种类型的，它们可以是OOP意义上的对象，也可以使内置类型值。被存储在容器中的数据为容器所有，这意味着如果容器过期了，里面的数据也会过期。（不过如果是数据是指针的话，则它指向的数据并不一定过期。）

不是任何类型的对象都能存储在容器中的，类型必须是可复制构造的和可赋值的。

**序列**

deque、list、queue、priority\_queue、stack、vector都是序列，队列能够在队尾添加元素，在队首删除元素。deque表示的双端队列允许在两端添加和删除元素。

序列要求其元素按严格的线性顺序排列，不会在两次迭代之间发生变化。

表 序列的要求

|  |  |  |
| --- | --- | --- |
| 表达式 | 返回值类型 | 说明 |
| X a(n, t) |  | 声明一个名为a的、由n个t组成的序列 |
| X(n, t) |  | 创建一个由n个t组成的匿名序列 |
| X a(i,j) |  | 声明一个名为a的序列，并将其初始化为区间[i, j)的内容 |
| X(i, j) |  | 创建一个匿名序列，并将其初始化为区间[i, j)的内容 |
| a.insert(p, t) | 迭代器 | 将t插入p前面 |
| a.insert(p, n, t) | void | 在p前面插入n个t |
| a.insert(p, i, j) | void | 在p前面插入区间[i, j)的内容 |
| a.erase(p) | 迭代器 | 擦除删除p指向的元素 |
| a.erase(p, q) | 迭代器 | 删除区间[p, q]区间的元素 |
| a.clear() | void | 把a清空，等价于a.clear(a.begin(), a.end()) |

表 序列的可选要求

|  |  |  |  |
| --- | --- | --- | --- |
| 表达式 | 返回值类型 | 含义 | 容器 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**1.vector**

vector是数组的一种类表示，它提供了自动内存管理功能，可以动态地改变vector对象的长度，并随着元素的添加和删除而增大和缩小，他提供了对元素的随机访问。在尾部添加和删除元素的时间是固定的，但是在头部或中间插入和删除元素的复杂度为现行时间。

vector是可翻转容器，增加了两个方法rbegin()和rend()，返回的迭代器都是类级类型reverse\_iterator。对这样的迭代器进行弟子增，将导致它反向遍历可反转容器。

**2.deque**

双端队列double-ended queue，在STL中类似于vector容器，支持随机访问，主要区别是，从deque对象的开始位置插入和删除元素的时间是固定的，不像vector中那样是线性时间。因为实现更复杂，所以它比vector实现更慢。

**3.list**

双向链表。除了第一个和最后一个元素外，每个元素都与前后的元素相链接，这意味着可以双向遍历链表。list和vector之间关键的区别在于，list在链表中任意位置进行插入和删除的时间都是固定的。因此，vector强调的是通过随机访问进行快速访问，而list强调的是元素的快速插入和删除。

list也是可反转容器。但是list不支持数组表示法和随机访问。与矢量迭代器不同，从容器中插入和删除元素后曼联表迭代器指向的元素将不变，因为它不会移动元素的位置，知识会改变链接信息。指向某个元素的迭代器在容器插入或删除其他元素后，仍指向它原来指向的元素，但它的链接元素可能和原来不同。