# C++辅导笔记：基础知识

## 一、C++概述

(一) 发展历史

1980年，Bjarne Stroustrup博士开始着手创建一种模拟语言，能够具有面向对象的程序设计特色。在当时，面向对象编程还是一个比较新的理念，Stroustrup博士并不是从头开始设计新语言，而是在C语言的基础上进行创建。这就是C++语言。

1985年，C++开始在外面慢慢流行。经过多年的发展，C++已经有了多个版本。为次，ANSI和ISO的联合委员会于1989年着手为C++制定标准。1994年2月，该委员会出版了第一份非正式草案，1998年正式推出了C++的国际标准。

(二) C和C++

C++是C的超集，也可以说C是C++的子集，因为C先出现。按常理说，C++编译器能够编译任何C程序，但是C和C++还是有一些小差别。

例如C++增加了C不具有的关键字。这些关键字能作为函数和变量的标识符在C程序中使用，尽管C++包含了所有的C，但显然没有任何C++编译器能编译这样的C程序。

C程序员可以省略函数原型，而C++不可以，一个不带参数的C函数原型必须把void写出来。而C++可以使用空参数列表。

C++中new和delete是对内存分配的运算符，取代了C中的malloc和free。

标准C++中的字符串类取代了C标准C函数库头文件中的字符数组处理函数。

C++中用来做控制态输入输出的iostream类库替代了标准C中的stdio函数库。

C++中的try/catch/throw异常处理机制取代了标准C中的setjmp()和longjmp()函数。

## 二、关键字和变量

C++相对与C增加了一些关键字，如下：

typename bool dynamic\_cast mutable namespace

static\_cast using catch explicit new

virtual operator false private template

volatile const protected this wchar\_t

const\_cast public throw friend true

reinterpret\_cast try

bitor xor\_e and\_eq compl or\_eq

not\_eq bitand

在C++中还增加了bool型变量和wchar\_t型变量：

布尔型变量是有两种逻辑状态的变量，它包含两个值：真和假。如果在表达式中使用了布尔型变量，那么将根据变量值的真假而赋予整型值1或0。要把一个整型变量转换成布尔型变量，如果整型值为0，则其布尔型值为假；反之如果整型值为非0，则其布尔型值为真。布儿型变量在运行时通常用做标志，比如进行逻辑测试以改变程序流程。

#include iostream.h

int main()

{

bool flag;

flag=true;

if(flag) cout< return 0;

}

C++中还包括wchar\_t数据类型，wchar\_t也是字符类型，但是是那些宽度超过8位的数据类型。许多外文字符集所含的数目超过256个，char字符类型无法完全囊括。wchar\_t数据类型一般为16位。

标准C++的iostream类库中包括了可以支持宽字符的类和对象。用wout替代cout即可。

#include iostream.h

int main()

{

wchar\_t wc;

wc='b';

wout< wc='y';

wout< wc='e';

wout< return 0;

}

说明一下：某些编译器无法编译该程序(不支持该数据类型)。

## 三、强制类型转换

有时候，根据表达式的需要，某个数据需要被当成另外的数据类型来处理，这时，就需要强制编译器把变量或常数由声明时的类型转换成需要的类型。为此，就要使用强制类型转换说明，格式如下：

int\* iptr=(int\*) &table;

表达式的前缀(int\*)就是传统C风格的强制类型转换说明(typecast)，又可称为强制转换说明(cast)。强制转换说明告诉编译器把表达式转换成指定的类型。有些情况下强制转换是禁用的，例如不能把一个结构类型转换成其他任何类型。数字类型和数字类型、指针和指针之间可以相互转换。当然，数字类型和指针类型也可以相互转换，但通常认为这样做是不安全而且也是没必要的。强制类型转换可以避免编译器的警告。

long int el=123;

short i=(int) el;

float m=34.56;

int i=(int) m;

上面两个都是C风格的强制类型转换，C++还增加了一种转换方式，比较一下上面和下面这个书写方式的不同：

long int el=123;

short i=int (el);

float m=34.56;

int i=int (m);

使用强制类型转换的最大好处就是：禁止编译器对你故意去做的事发出警告。但是，利用强制类型转换说明使得编译器的类型检查机制失效，这不是明智的选择。通常，是不提倡进行强制类型转换的。除非不可避免，如要调用malloc()函数时要用的void型指针转换成指定类型指针。

## 四、标准输入输出流

在C语言中，输入输出是使用语句scanf()和printf()来实现的，而C++中是使用类来实现的。

#include iostream.h

main() //C++中main()函数默认为int型，而C语言中默认为void型。

{

int a;

cout< 

cin>>a; /\*输入一个数值\*/

cout< return 0;

}

cin,cout,endl对象，他们本身并不是C++语言的组成部分。虽然他们已经是ANSI标准C++中被定义，但是他们不是语言的内在组成部分。在C++中不提供内在的输入输出运算符，这与其他语言是不同的。输入和输出是通过C++类来实现的，cin和cout是这些类的实例，他们是在C++语言的外部实现。

在C++语言中，有了一种新的注释方法，就是‘//’，在该行//后的所有说明都被编译器认为是注释，这种注释不能换行。C++中仍然保留了传统C语言的注释风格/\*……\*/。

C++也可采用格式化输出的方法：

#include iostream.h

int main()

{

int a;

cout< 

cin>>a;

cout<

## 五、函数重载

在C++中，允许有相同的函数名，不过它们的参数类型不能完全相同，这样这些函数就可以相互区别开来。而这在C语言中是不允许的。

### 1.参数个数不同

#include iostream.h

void a(int,int);

void a(int);

int main()

{

a(5);

a(6,7);

return 0;

}

void a(int i)

{

cout< }

void a(int i,int j)

{

cout< }

### 2.参数格式不同

#include iostream.h

void a(int,int);

void a(int,float);

int main()

{

a(5,6);

a(6,7.0);

return 0;

}

void a(int i,int j)

{

cout< }

void a(int i,float j)

{

cout< }

## 六、变量作用域

C++语言中，允许变量定义语句在程序中的任何地方，只要在是使用它之前就可以；而C语言中，必须要在函数开头部分。而且C++允许重复定义变量，C语言也是做不到这一点的。看下面的程序：

#include iostream.h

int a;

int main()

{

cin>>a;

for(int i=1;i<=10;i++) //C语言中，不允许在这里定义变量

{

static int a=0; //C语言中，同一函数块，不允许有同名变量

a+=i;

cout<<::a<< < }

return 0;

}

## 七、new和delete运算符

在C++语言中，仍然支持malloc()和free()来分配和释放内存，同时增加了new和delete来管理内存。

### 1.为固定大小的数组分配内存

#include iostream.h

int main()

{

int \*birthday=new int[3];

birthday[0]=6;

birthday[1]=24;

birthday[2]=1940;

cout< < delete [] birthday; //注意这儿

return 0;

}

在删除数组时，delete运算符后要有一对方括号。

### 2.为动态数组分配内存

#include iostream.h

#include stdlib.h

int main()

{

int size;

cin>>size;

int \*array=new int[size];

for(int i=0;i array[i]=rand();

for(i=0;i cout<<'\n'< delete [] array;

return 0;

}

## 八、引用型变量

在C++中，引用是一个经常使用的概念。引用型变量是其他变量的一个别名，我们可以认为他们只是名字不相同，其他都是相同的。

### 1.引用是一个别名

C++中的引用是其他变量的别名。声明一个引用型变量，需要给他一个初始化值，在变量的生存周期内，该值不会改变。& 运算符定义了一个引用型变量：

int a;

int& b=a;

先声明一个名为a的变量，它还有一个别名b。我们可以认为是一个人，有一个真名，一个外号，以后不管是喊他a还是b，都是叫他这个人。同样，作为变量，以后对这两个标识符操作都会产生相同的效果。

#include iostream.h

int main()

{

int a=123;

int& b=a;

cout< a++;

cout< b++;

cout< return 0;

}

### 2.引用的初始化

和指针不同，引用变量的值不可改变。引用作为真实对象的别名，必须进行初始化，除非满足下列条件之一：

(1) 引用变量被声明为外部的，它可以在任何地方初始化

(2) 引用变量作为类的成员，在构造函数里对它进行初始化

(3) 引用变量作为函数声明的形参，在函数调用时，用调用者的实参来进行初始化

### 3.作为函数形参的引用

引用常常被用作函数的形参。以引用代替拷贝作为形参的优点：

引用避免了传递大型数据结构带来的额外开销

引用无须象指针那样需要使用\*和->等运算符

#include iostream.h

void func1(s p);

void func2(s& p);

struct s

{

int n;

char text[10];

};

int main()

{

static s str={123,China};

func1(str);

func2(str);

return 0;

}

void func1(s p)

{

cout< cout< }

void func2(s& p)

{

cout< cout< }

从表面上看，这两个函数没有明显区别，不过他们所花的时间却有很大差异，func2()函数所用的时间开销会比func2()函数少很多。它们还有一个差别，如果程序递归func1()，随着递归的深入，会因为栈的耗尽而崩溃，但func2()没有这样的担忧。

### 4.以引用方式调用

当函数把引用作为参数传递给另一个函数时，被调用函数将直接对参数在调用者中的拷贝进行操作，而不是产生一个局部的拷贝(传递变量本身是这样的)。这就称为以引用方式调用。把参数的值传递到被调用函数内部的拷贝中则称为以传值方式调用。

#include iostream.h

void display(const Date&,const char\*);

void swapper(Date&,Date&);

struct Date

{

int month,day,year;

};

int main()

{

static Date now={2,23,90};

static Date then={9,10,60};

display(now,Now: );

display(then,Then: );

swapper(now,then);

display(now,Now: );

display(then,Then: );

return 0;

}

void swapper(Date& dt1,Date& dt2)

{

Date save;

save=dt1;

dt1=dt2;

dt2=save;

}

void display(const Date& dt,const char \*s)

{

cout< cout< }

### 5.以引用作为返回值

#include iostream.h

struct Date

{

int month,day,year;

};

Date birthdays[]=

{

{12,12,60};

{10,25,85};

{5,20,73};

};

const Date& getdate(int n)

{

return birthdays[n-1];

}

int main()

{

int dt=1;

while(dt!=0)

{

cout< cin>>dt;

if(dt>0 && dt<4)

{

const Date& bd=getdate(dt);

cout< }

}

return 0;

}

# C++辅导笔记：类的设计、构造函数和析构函数

## 一、类的设计

### 1.类的声明

class 类名

{

private: //私有

...

public: //公有

...

};

### 2.类的成员

一般在C++类中，所有定义的变量和函数都是类的成员。如果是变量，我们就叫它数据成员如果是函数，我们就叫它成员函数。

### 3.类成员的可见性

private和public访问控制符决定了成员的可见性。由一个访问控制符设定的可访问状态将一直持续到下一个访问控制符出现，或者类声明的结束。私有成员仅能被同一个类中的成员函数访问，公有成员既可以被同一类中的成员函数访问，也可以被其他已经实例化的类中函数访问。当然，这也有例外的情况，这是以后要讨论的友元函数。

类中默认的数据类型是private，结构中的默认类型是public。一般情况下，变量都作为私有成员出现，函数都作为公有成员出现。

类中还有一种访问控制符protected，叫保护成员，以后再说明。

### 4.初始化

在声明一个类的对象时，可以用圆括号()包含一个初始化表。

看下面一个例子：

#include iostream.h

class Box

{

private:

int height,width,depth; //3个私有数据成员

public:

Box(int,int,int);

~Box();

int volume(); //成员函数

};

Box::Box(int ht,int wd,int dp)

{

height=ht;

width=wd;

depth=dp;

}

Box::~Box()

{

//nothing

}

int Box::volume()

{

return height\*width\*depth;

}

int main()

{

Box thisbox(3,4,5); //声明一个类对象并初始化

cout< return 0;

}

当一个类中没有private成员和protected成员时，也没有虚函数，并且不是从其他类中派生出来的，可以用{}来初始化。(以后再讲解)

### 5.内联函数

内联函数和普通函数的区别是：内联函数是在编译过程中展开的。通常内联函数必须简短。定义类的内联函数有两种方法：一种和C语言一样，在定义函数时使用关键字inline。如：

inline int Box::volume()

{

return height\*width\*depth;

}

还有一种方法就是直接在类声明的内部定义函数体，而不是仅仅给出一个函数原型。我们把上面的函数简化一下：

#include iostream.h

class Box

{

private:

int height,width,depth;

public:

Box(int ht,int wd,int dp)

{

height=ht;

width=wd;

depth=dp;

}

~Box();

int volume()

{

return height\*width\*depth;

}

};

int main()

{

Box thisbox(3,4,5); //声明一个类对象并初始化

cout< return 0;

}

这样，两个函数都默认为内联函数了。

## 二、构造函数

什么是构造函数？通俗的讲，在类中，函数名和类名相同的函数称为构造函数。上面的Box()函数就是构造函数。C++允许同名函数，也就允许在一个类中有多个构造函数。如果一个都没有，编译器将为该类产生一个默认的构造函数，这个构造函数可能会完成一些工作，也可能什么都不做。

绝对不能指定构造函数的类型，即使是void型都不可以。实际上构造函数默认为void型。

当一个类的对象进入作用域时，系统会为其数据成员分配足够的内存，但是系统不一定将其初始化。和内部数据类型对象一样，外部对象的数据成员总是初始化为0。局部对象不会被初始化。构造函数就是被用来进行初始化工作的。当自动类型的类对象离开其作用域时，所站用的内存将释放回系统。

看上面的例子，构造函数Box()函数接受三个整型擦黑素，并把他们赋值给立方体对象的数据成员。

如果构造函数没有参数，那么声明对象时也不需要括号。

### 1.使用默认参数的构造函数

当在声明类对象时，如果没有指定参数，则使用默认参数来初始化对象。

#include iostream.h

class Box

{

private:

int height,width,depth;

public:

Box(int ht=2,int wd=3,int dp=4)

{

height=ht;

width=wd;

depth=dp;

}

~Box();

int volume()

{

return height\*width\*depth;

}

};

int main()

{

Box thisbox(3,4,5); //初始化

Box defaulbox; //使用默认参数

cout< cout<

return 0;

}

### 2.默认构造函数

没有参数或者参数都是默认值的构造函数称为默认构造函数。如果你不提供构造函数，编译器会自动产生一个公共的默认构造函数，这个构造函数什么都不做。如果至少提供一个构造函数，则编译器就不会产生默认构造函数。

### 3.重载构造函数

一个类中可以有多个构造函数。这些构造函数必须具有不同的参数表。在一个类中需要接受不同初始化值时，就需要编写多个构造函数，但有时候只需要一个不带初始值的空的Box对象。

#include iostream.h

class Box

{

private:

int height,width,depth;

public:

Box() { //nothing }

Box(int ht=2,int wd=3,int dp=4)

{

height=ht;

width=wd;

depth=dp;

}

~Box();

int volume()

{

return height\*width\*depth;

}

};

int main()

{

Box thisbox(3,4,5); //初始化

Box otherbox;

otherbox=thisbox;

cout< return 0;

}

这两个构造函数一个没有初始化值，一个有。当没有初始化值时，程序使用默认值，即2，3，4。

但是这样的程序是不好的。它允许使用初始化过的和没有初始化过的Box对象，但它没有考虑当thisbox给otherbox赋值失败后，volume()该返回什么。较好的方法是，没有参数表的构造函数也把默认值赋值给对象。

class Box

{

int height,width,depth;

public:

Box()

{

height=0;width=0;depth=0;

}

Box(int ht,int wd,int dp)

{

height=ht;width=wd;depth=dp;

}

int volume()

{

return height\*width\*depth;

}

};

这还不是最好的方法，更好的方法是使用默认参数，根本不需要不带参数的构造函数。

class Box

{

int height,width,depth;

public:

Box(int ht=0,int wd=0,int dp=0)

{

height=ht;width=wd;depth=dp;

}

int volume()

{

return height\*width\*depth;

}

};

## 三、析构函数

当一个类的对象离开作用域时，析构函数将被调用(系统自动调用)。析构函数的名字和类名一样，不过要在前面加上 ~ 。对一个类来说，只能允许一个析构函数，析构函数不能有参数，并且也没有返回值。析构函数的作用是完成一个清理工作，如释放从堆中分配的内存。

我们也可以只给出析构函数的形式，而不给出起具体函数体，其效果是一样的，如上面的例子。但在有些情况下，析构函数又是必需的。如在类中从堆中分配了内存，则必须在析构函数中释放

# 二级C++辅导笔记：类的转换

C++的内部数据类型遵循隐式类型转换规则。假设某个表达市中使用了一个短整型变量，而编译器根据上下文认为这儿需要是的长整型，则编译器就会根据类型转换规则自动把它转换成长整型，这种隐式转换出现在赋值、参数传递、返回值、初始化和表达式中。我们也可以为类提供相应的转换规则。

对一个类建立隐式转换规则需要构造一个转换函数，该函数作为类的成员，可以把该类的对象和其他数据类型的对象进行相互转换。声明了转换函数，就告诉了编译器，当根据句法判定需要类型转换时，就调用函数。

有两种转换函数。一种是转换构造函数；另一种是成员转换函数。需要采用哪种转换函数取决于转换的方向。

## 一、转换构造函数

当一个构造函数仅有一个参数，且该参数是不同于该类的一个数据类型，这样的构造函数就叫转换构造函数。转换构造函数把别的数据类型的对象转换为该类的一个对象。和其他构造函数一样，如果声明类的对象的初始化表同转换构造函数的参数表相匹配，该函数就会被调用。当在需要使用该类的地方使用了别的数据类型，便宜器就会调用转换构造函数进行转换。

#include iostream.h

#include time.h

#include stdio.h

class Date

{

int mo, da, yr;

public:

Date(time\_t);

void display();

};

void Date::display()

{

char year[5];

if(yr<10)

sprintf(year,0%d,yr);

else

sprintf(year,%d,yr);

cout< }

Date::Date(time\_t now)

{

tm\* tim=localtime(&now);

da=tim->tm\_mday;

mo=tim->tm\_mon+1;

yr=tim->tm\_year;

if(yr>=100) yr-=100;

}

int main()

{

time\_t now=time(0);

Date dt(now);

dt.display();

return 0;

}

本程序先调用time()函数来获取当前时间，并把它赋给time\_t对象；然后程序通过调用Date类的转换构造函数来创建一个Date对象，该对象由time\_t对象转换而来。time\_t对象先传递给localtime()函数，然后返回一个指向tm结构(time.h文件中声明)的指针，然后构造函数把结构中的日月年的数值拷贝给Date对象的数据成员，这就完成了从time\_t对象到Date对象的转换。

## 二、成员转换函数

成员转换函数把该类的对象转换为其他数据类型的对象。在成员转换函数的声明中要用到关键字operator。这样声明一个成员转换函数：

operator aaa();

在这个例子中，aaa就是要转换成的数据类型的说明符。这里的类型说明符可以是任何合法的C++类型，包括其他的类。如下来定义成员转换函数；

Classname::operator aaa()

类名标识符是声明了该函数的类的类型说明符。上面定义的Date类并不能把该类的对象转换回time\_t型变量，但可以把它转换成一个长整型值，计算从2000年1月1日到现在的天数。

#include iostream.h

class Date

{

int mo,da,yr;

public:

Date(int m,int d,int y) {mo=m; da=d; yr=y;}

operator int(); //声明

};

Date::operator int() //定义

{

static int dys[]={31,28,31,30,31,30,31,31,30,31,30,31};

int days=yr-2000;

days\*=365;

days+=(yr-2000)/4;

for(int i=0;i days+=dys[i];

days+=da;

return days;

}

int main()

{

Date now(12,24,2003);

int since=now;

cout< return 0;

}

## 三、类的转换

上面两个例子都是C++类对象和内部数据对象之间的相互转换。也可以定义转换函数来实现两个类对象之间的相互转换。

#include iostream.h

class CustomDate

{

public:

int da, yr;

CustomDate(int d=0,int y=0) {da=d; yr=y;}

void display()

{

cout< }

};

class Date

{

int mo, da, yr;

public:

Date(int m=0,int d=0,int y=0) {mo=m; da=d; yr=y;}

Date(const CustomDate&); //转换构造函数

operator CustomDate(); //成员转换函数

void display()

{

cout< }

};

static int dys[] = {31,28,31,30,31,30,31,31,30,31,30,31};

Date::Date(const CustomDate& jd)

{

yr=jd.yr;

da=jd.da;

for(mo=0;mo<11;mo++)

if(da>dys[mo]) da-=dys[mo];

else break;

mo++;

}

Date::operator CustomDate()

{

CustomDate cd(0,yr);

for(int i=0;i cd.da+=da;

return cd;

}

int main()

{

Date dt(12,24,3);

CustomDate cd;

cd = dt; //调用成员转换函数

cd.display();

dt = cd; //调用转换构造函数

dt.display();

return 0;

}

这个例子中有两个类CustomDate和Date，CustomDate型日期包含年份和天数。

这个例子没有考虑闰年情况。但是在实际构造一个类时，应该考虑到所有问题的可能性。

在Date里中具有两种转换函数，这样，当需要从Date型变为CustomDate型十，可以调用成员转换函数；反之可以调用转换构造函数。

不能既在Date类中定义成员转换函数，又在CustomDate类里定义转换构造函数。那样编译器在进行转换时就不知道该调用哪一个函数，从而出错.

## 四、转换函数的调用

C++里调用转换函数有三种形式：第一种是隐式转换，例如编译器需要一个Date对象，而程序提供的是CustomDate对象，编译器会自动调用合适的转换函数。另外两种都是需要在程序代码中明确给出的显式转换。C++强制类型转换是一种，还有一种是显式调用转换构造函数和成员转换函数。下面的程序给出了三中转换形式：

#include iostream.h

class CustomDate

{

public:

int da, yr;

CustomDate(int d=0,int y=0) {da=d; yr=y;}

void display()

{

cout< }

};

class Date

{

int mo, da, yr;

public:

Date(int m,int d,int y)

{

mo=m; da=d; yr=y;

}

operator CustomDate();

};

Date::operator CustomDate()

{

static int dys[]={31,28,31,30,31,30,31,31,30,31,30,31};

CustomDate cd(0,yr);

for(int i=0;i cd.da+=da;

return cd;

}

int main()

{

Date dt(11,17,89);

CustomDate cd;

cd = dt;

cd.display();

cd = (CustomDate) dt;

cd.display();

cd = CustomDate(dt);

cd.display();

return 0;

}

## 五、转换发生的情形

上面的几个例子都是通过不能类型对象之间的相互赋值来调用转换函数，还有几种调用的可能：

参数传递

初始化

返回值

表达式语句

这些情况下，都有可能调用转换函数。

下面的程序不难理解，就不分析了。

#include iostream.h

class CustomDate

{

public:

int da, yr;

CustomDate() {}

CustomDate(int d,int y) { da=d; yr=y;}

void display()

{

cout< }

};

class Date

{

int mo, da, yr;

public:

Date(int m,int d,int y) { mo=m; da=d; yr=y; }

operator CustomDate();

};

Date::operator CustomDate()

{

static int dys[]={31,28,31,30,31,30,31,31,30,31,30,31};

CustomDate cd(0,yr);

for (int i=0;i cd.da+=da;

return cd;

}

class Tester

{

CustomDate cd;

public:

explicit Tester(CustomDate c) { cd=c; }

void display() { cd.display(); }

};

void dispdate(CustomDate cd)

{

cd.display();

}

CustomDate rtndate()

{

Date dt(9,11,1);

return dt;

}

int main()

{

Date dt(12,24,3);

CustomDate cd;

cd = dt;

cd.display();

dispdate(dt);

Tester ts(dt);

ts.display();

cd = rtndate();

cd.display();

return 0;

}

## 六、显式构造函数

注意上面Tester类的构造函数前面有一个explicit修饰符。如果不加上这个关键字，那么在需要把CustomDate对象转换成Tester对象时，编译器会把该函数当作转换构造函数来调用。但是有时候，并不想把这种只有一个参数的构造函数用于转换目的，而仅仅希望用它来显式地初始化对象，此时，就需要在构造函数前加explicit。如果在声明了Tester对象以后使用了下面的语句将导致一个错误：

ts=jd; //error

这个错误说明，虽然Tester类中有一个以Date型变量为参数的构造函数，编译器却不会把它看作是从Date到Tester的转换构造函数，因为它的声明中包含了explicit修饰符。

## 七、表达式内部的转换

在表达式内部，如果发现某个类型和需要的不一致，就会发生错误。数字类型的转换是很简单，这里就不举例了。下面的程序是把Date对象转换成长整型值。

#include iostream.h

class Date

{

int mo, da, yr;

public:

Date(int m,int d,int y)

{

mo=m; da=d; yr=y;

}

operator long();

};

Date::operator long()

{

static int dys[]={31,28,31,30,31,30,31,31,30,31,30,31};

long days=yr;

days\*=365;

days+=(yr-1900)/4; //从1900年1月1日开始计算

for(int i=0;i days+=da;

return days;

}

int main()

{

Date today(12,24,2003);

const long ott=123;

long sum=ott+today;

cout< return 0;

}

在表达式中，当需要转换的对象可以转换成某个数字类型，或者表达式调用了作用于某个类的重载运算符时，就会发生隐式转换。运算符重载以后再学习

# 二级C++辅导笔记：私有数据成员和友元

## 一、私有数据成员的使用

### 1.取值和赋值成员函数

面向对象的约定就是保证所有数据成员的私有性。一般我们都是通过公有成员函数来作为公共接口来读取私有数据成员的。某些时候，我们称这样的函数为取值和赋值函数。

取值函数的返回值和传递给赋值函数的参数不必一一匹配所有数据成员的类型。

#include iostream.h

class Date

{

int mo, da, yr;

public:

Date(int m,int d,int y) { mo=m; da=d; yr=y; }

int getyear() const { return yr; }

void setyear(int y) { yr = y; }

};

int main()

{

Date dt(4,1,89);

cout< dt.setyear(97);

cout< return 0;

}

上面的例子很简单，不分析了。要养成这样的习惯，通过成员函数来访问和改变类中的数据。这样有利于软件的设计和维护。比如，改变Date类内部数据的形式，但仍然用修改过的getyear()和setyear()来提供访问接口，那么使用该类就不必修改他们的代码，仅需要重新编译程序即可。

### 2.常量成员函数

注意上面的程序中getyear()被声明为常量型，这样可以保证该成员函数不会修改调用他的对象。通过加上const修饰符，可以使访问对象数据的成员函数仅仅完成不会引起数据变动的那些操作。

如果程序声明某个Date对象为常量的话，那么该对象不得调用任何非常量型成员函数，不论这些函数是否真的试图修改对象的数据。只有把那些不会引起数据改变的函数都声明为常量型，才可以让常量对象来调用。

### 3.改进的成员转换函数

下面的程序改进了从Date对象到CustomDate对象的成员转换函数，用取值和赋值函数取代了使用公有数据成员的做法。(以前的程序代码在上一帖中)

#include iostream.h

class CustomDate

{

int da,yr;

public:

CustomDate() {}

CustomDate(int d,int y) { da=d; yr=y; }

void display() const {cout< int getday() const { return da; }

void setday(int d) { da=d; }

};

class Date

{

int mo,da,yr;

public:

Date(int m,int d,int y) { mo=m; da=d; yr=y; }

operator CustomDate() const;

};

Date::operator CustomDate() const

{

static int dys[] = {31,28,31,30,31,30,31,31,30,31,30,31};

CustomDate cd(0,yr);

int day=da;

for(int i=0;i cd.setday(day);

return cd;

}

int main()

{

Date dt(11,17,89);

CustomDate cd;

cd=dt;

cd.display();

return 0;

}

注意上面的程序中Date::operator CustomDate()声明为常量型，因为这个函数没有改变调用它对象的数据，尽管它修改了一个临时CustomDate对象并将其作为函数返回值。

## 二、友元

前面已经说过了，私有数据成员不能被类外的其他函数读取，但是有时候类会允许一些特殊的函数直接读写其私有数据成员。

关键字friend可以让特定的函数或者别的类的所有成员函数对私有数据成员进行读写。这既可以维护数据的私有性，有可以保证让特定的类或函数能够直接访问私有数据。

### 1.友元类

一个类可以声明另一个类为其友元，这个友元的所有成员函数都可以读写它的私有数据。

#include iostream.h

class Date;

class CustomDate

{

int da,yr;

public:

CustomDate(int d=0,int y=0) { da=d; yr=y; }

void display() const {cout< friend Date; //这儿

};

class Date

{

int mo,da,yr;

public:

Date(int m,int d,int y) { mo=m; da=d; yr=y; }

operator CustomDate();

};

Date::operator CustomDate()

{

static int dys[] = {31,28,31,30,31,30,31,31,30,31,30,31};

CustomDate cd(0, yr);

for (int i=0;i cd.da+=da;

return cd;

}

int main()

{

Date dt(11,17,89);

CustomDate cd(dt);

cd.display();

return 0;

}

在上面的程序中，有这样一句 friend Date; 该语句告诉编译器，Date类的所有成员函数有权访问CustomDate类的私有成员。因为Date类的转换函数需要知道CustomDate类的每个数据成员，所以真个Date类都被声明为CustomDate类的友元。

### 2.隐式构造函数

上面程序对CustomDate的构造函数的调用私有显示该类需要如下的一个转换构造函数:

CustomDate(Date& dt);

但是唯一的一个构造函数是：CustomDate(int d=0;int y=0)；

这就出现了问题，编译器要从Date对象构造一个CustomDate对象，但是CustomDate类中并没有定义这样的转换构造函数。不过Date类中定义了一个成员转换函数，它可以把Date对象转换成CustomDate对象。于是编译器开始搜索CustomDate类，看其是否有一个构造函数，能从一个已存在的CustomDate的对象创建新的CustomDate对象。这种构造函数叫拷贝构造函数。拷贝构造函数也只有一个参数，该参数是它所属的类的一个对象，由于CustomDate类中没有拷贝构造函数，于是编译器就会产生一个默认的拷贝构造函数，该函数简单地把已存在的对象的每个成员拷贝给新对象。现在我们已经知道，编译器可以把Date对象转换成CustomDate对象，也可以从已存在的CustomDate对象生成一个新的CustomDate对象。那么上面提出的问题，编译器就是这样做的：它首先调用转换函数，从Date对象创建一个隐藏的、临时的、匿名的CustomDate对象，然后用该临时对象作为参数调用默认拷贝构造函数，这就生成了一个新的CustomDate对象。

### 3.预引用

上面的例子中还有这样一句 class Date;

这个语句叫做预引用。它告诉编译器，类Date将在后面定义。编译器必须知道这个信号，因为CustomDate类中引用了Date类，而Date里也引用了CustomDate类，必须首先声明其中之一。

使用了预引用后，就可以声明未定义的类的友元、指针和引用。但是不可以使用那些需要知道预引用的类的定义细节的语句，如声明该类的一个实例或者任何对该类成员的引用。

### 4.显式友元预引用

也可以不使用预引用，这只要在声明友元的时候加上关键自class就行了。

#include iostream.h

class CustomDate

{

int da,yr;

public:

CustomDate(int d=0,int y=0) { da=d; yr=y; }

void display() const {cout< friend class Date; //这儿，去掉前面的预引用

};

class Date

{

... ...

};

Date::operator CustomDate()

{

... ...

}

int main()

{

... ...

}

### 5.友元函数

通常，除非真的需要，否则并不需要把整个类都设为另一个类的友元，只需挑出需要访问当前类私有数据成员的成员函数，将它们设置为该类的友元即可。这样的函数称为友元函数。

下面的程序限制了CustomDate类数据成员的访问，Date类中只有需要这些数据的成员函数才有权读写它们。

#include iostream.h

class CustomDate;

class Date

{

int mo,da,yr;

public:

Date(const CustomDate&);

void display() const {cout< };

class CustomDate

{

int da,yr;

public:

CustomDate(int d=0,int y=0) { da=d; yr=y; }

friend Date::Date(const CustomDate&);

};

Date::Date(const CustomDate& cd)

{

static int dys[] = {31,28,31,30,31,30,31,31,30,31,30,31};

yr=cd.yr;

da=cd.da;

for(mo=0;mo<11;mo++)

if(da>dys[mo]) da-=dys[mo];

else break;

mo++;

}

int main()

{

Date dt(CustomDate(123, 89));

dt.display();

return 0;

}

### 6.匿名对象

上面main()函数中Date对象调用CustomDate类的构造函数创建了一个匿名CustomDate对象，然后用该对象创建了一个Date对象。这种用法在C++中是经常出现的。

### 7.非类成员的友元函数

有时候友元函数未必是某个类的成员。这样的函数拥有类对象私有数据成员的读写权，但它并不是任何类的成员函数。这个特性在重载运算符时特别有用。

非类成员的友元函数通常被用来做为类之间的纽带。一个函数如果被两个类同时声明为友元，它就可以访问这两个类的私有成员。下面的程序说明了一个可以访问两个类私有数据成员的友元函数是如何将在两个类之间架起桥梁的。

#include iostream.h

class Time;

class Date

{

int mo,da,yr;

public:

Date(int m,int d,int y) { mo=m; da=d; yr=y;}

friend void display(const Date&, const Time&);

};

class Time

{

int hr,min,sec;

public:

Time(int h,int m,int s) { hr=h; min=m; sec=s;}

friend void display(const Date&, const Time&);

};

void display(const Date& dt, const Time& tm)

{

cout << dt.mo << '/' << dt.da << '/' << dt.yr;

cout << ' ';

cout << tm.hr << ':' << tm.min << ':' << tm.sec;

}

int main()

{

Date dt(2,16,97);

Time tm(10,55,0);

display(dt, tm);

return 0;

}

# 二级C++辅导笔记：析构函数和this指针

## 一、析构函数

前面的一些例子都没有说明析构函数，这是因为所用到的类在结束时不需要做特别的清理工作。下面的程序给出了一新的Date类，其中包括一个字符串指针，用来表示月份。

#include iostream.h

#include string.h

class Date

{

int mo,da,yr;

char \*month;

public:

Date(int m=0, int d=0, int y=0);

~Date();

void display() const;

};

Date::Date(int m,int d,int y)

{

static char \*mos[] =

{

January,February,March,April,May,June,

July,August,September,October,November,December

};

mo=m; da=d; yr=y;

if(m!=0)

{

month=new char[strlen(mos[m-1])+1];

strcpy(month, mos[m-1]);

}

else month = 0;

}

Date::~Date()

{

delete [] month;

}

void Date::display() const

{

if(month!=0) cout< }

int main()

{

Date birthday(8,11,1979);

birthday.display();

return 0;

}

在Date对象的构造函数中，首先用new运算符为字符串month动态分配了内存，然后从内部数组中把月份的名字拷贝给字符串指针month。

析构函数在删除month指针时，可能会出现一些问题。当然从这个程序本身来看，没什么麻烦；但是从设计一个类的角度来看，当Date类用于赋值时，就会出现问题。假设上面的main()修改为“

int main()

{

Date birthday(8,11,1979);

Date today;

today=birthday;

birthday.display();

return 0;

}

这会生成一个名为today的空的Date型变量，并且把birthday值赋给它。如果不特别通知编译器，它会简单的认为类的赋值就是成员对成员的拷贝。在上面的程序中，变量birthday有一个字符型指针month，并且在构造函数里用new运算符初始化过了。当birthday离开其作用域时，析构函数会调用delete运算符来释放内存。但同时，当today离开它的作用域时，析构函数同样会对它进行释放操作，而today里的month指针是birthday里的month指针的一个拷贝。析构函数对同一指针进行了两次删除操作，这会带来不可预知的后果。

如果假设today是一个外部变量，而birthday是一个自变量。当birthday离开其作用域时，就已经把对象today里的month指针删除了。显然这也是不正确的。

再假设有两个初始化的Date变量，把其中一个的值赋值给另一个：

Date birthday(8,11,1979);

Date today(12,29,2003);

today=birthday;

问题就更复杂了，当这两个变量离开作用域时，birthday中的month的值已经通过赋值传递给了today。而today中构造函数用new运算符给month的值却因为赋值被覆盖了。这样，birthday中的month被删除了两次，而today中month却没有被删除掉。

## 二、重载赋值运算符

为了解决上面的问题，我们应该写一个特殊的赋值运算符函数来处理这类问题。当需要为同一个类的两个对象相互赋值时，就可以重载运算符函数。这个方法可以解决类的赋值和指针的释放。

下面的程序中，类中的赋值函数用new运算符从堆中分配了一个不同的指针，该指针获取赋值对象中相应的值,然后拷贝给接受赋值的对象。

在类中重载赋值运算符的格式如下：

void operator = (const Date&)

后面我们回加以改进。目前，重载的运算符函数的返回类型为void。它是类总的成员函数，在本程序红，是Date类的成员函数。它的函数名始终是operator =，参数也始终是同一个类的对象的引用。参数表示的是源对象，即赋值数据的提供者。重载函数的运算符作为目标对象的成员函数来使用。

#include iostream.h

#include string.h

class Date

{

int mo,da,yr;

char \*month;

public:

Date(int m=0, int d=0, int y=0);

~Date();

void operator=(const Date&);

void display() const;

};

Date::Date(int m, int d, int y)

{

static char \*mos[] =

{

January,February,March,April,May,June,

July,August,September,October,November,December

};

mo = m; da = d; yr = y;

if (m != 0)

{

month = new char[strlen(mos[m-1])+1];

strcpy(month, mos[m-1]);

}

else month = 0;

}

Date::~Date()

{

delete [] month;

}

void Date::display() const

{

if (month!=0) cout< char name[25];

cin >> name;

if (strncmp(name, end, 3) == 0) break;

ListEntry\* list = new ListEntry(name);

if (prev != 0) prev->AddEntry(\*list);

prev = list;

}

while (prev != 0)

{

prev->display();

ListEntry\* hold = prev;

prev = prev->PrevEntry();

delete hold;

}

return 0;

}

程序运行时，会提示输入一串姓名，当输入完毕后，键入end，然后程序会逆序显示刚才输入的所有姓名。

程序中ListEntry类含有一个字符串和一个指向前一个表项的指针。构造函数从对中获取内存分配给字符串，并把字符串的内容拷贝到内存，然后置链接指针为NULL。析构函数将释放字符串所占用的内存。

成员函数PrevEntry()返回指向链表前一个表项的指针。另一个成员函数显示当前的表项内容。

成员函数AddEntry()，它把this指针拷贝给参数的preventry指针，即把当前表项的地址赋值给下一个表项的链接指针，从而构造了一个链表。它并没有改变调用它的listEntry对象的内容，只是把该对象的地址赋给函数的参数所引用的那个ListEntry对象的preventry指针，尽管该函数不会修改对象的数据，但它并不是常量型。这是因为，它拷贝对象的地址this指针的内容给一个非长常量对象，而编译器回认为这个非常量对象就有可能通过拷贝得到的地址去修改当前对象的数据，因此AddEntry()函数在声明时不需要用const.

# 二级C++辅导笔记：类对象数组和静态成员

## 一、类对象数组

类的对象和C++其他数据类型一样，也可以为其建立数组，数组的表示方法和结构一样。

#include iostream.h

class Date

{

int mo,da,yr;

public:

Date(int m=0,int d=0, int y=0) { mo=m; da=d; yr=y;}

void display() const { cout< };

int main()

{

Date dates[2];

Date today(12,31,2003);

dates[0]=today;

dates[0].display();

dates[1].display();

return 0;

}

### 1.类对象数组和默认构造函数

在前面已经说过，不带参数或者所有参数都有默认值的构造函数叫做默认构造函数。如果类中没有构造函数，编译器会自动提供一个什么都不做的公共默认构造函数 。如果类当中至少有一个构造函数，编译器就不会提供默认构造函数。

如果类当中不含默认构造函数，则无法实例化其对象数组。因为实例花类对象数组的格式不允许用初始化值来匹配某个构造函数的参数表。

上面的程序中，main()函数声明了一个长度为2的Date对象数组，还有一个包含初始化值的单个Date对象。接着把这个初始化的Date对象赋值给数组中第一个对象，然后显示两个数组元素中包含的日期。从输出中可以看到，第一个日期是有效日期，而第二个显示的都是0。

当声明了某个类的对象数组时，编译器会为每个元素都调用默认构造函数。

下面的程序去掉了构造函数的默认参数值，并且增加了一个默认构造函数。

#include

class Date

{

int mo, da, yr;

public:

Date();

Date(int m,int d,int y) { mo=m; da=d; yr=y;}

void display() const { cout < };

Date::Date()

{

cout < mo=0; da=0; yr=0;

}

int main()

{

Date dates[2];

Date today(12,31,2003);

dates[0]=today;

dates[0].display();

dates[1].display();

return 0;

}

运行程序，输出为：

Date constructor running

Date constructor running

12/31/2003

0/0/0

从输出中可以看出，Date()这个默认构造函数被调用了两次。

### 2.类对象数组和析构函数

当类对象离开作用域时，编译器会为每个对象数组元素调用析构函数。

#include iostream.h

class Date

{

int mo,da,yr;

public:

Date(int m=0,int d=0,int y=0) { mo=m; da=d; yr=y;}

~Date() {cout< void display() const {cout< };

int main()

{

Date dates[2];

Date today(12,31,2003);

dates[0]=today;

dates[0].display();

dates[1].display();

return 0;

}

运行程序，输出为：

12/31/2003

0/0/0

Date destructor running

Date destructor running

Date destructor running

表明析构函数被调用了三次，也就是dates[0],dates[1],today这三个对象离开作用域时调用的。

## 二、静态成员

可以把类的成员声明为静态的。静态成员只能存在唯一的实例。所有的成员函数都可以访问这个静态成员。即使没有声明类的任何实例，静态成员也已经是存在的。不过类当中声明静态成员时并不能自动定义这个变量，必须在类定义之外来定义该成员。

### 1.静态数据成员

静态数据成员相当于一个全局变量，类的所有实例都可以使用它。成员函数能访问并且修改这个值。如果这个静态成员是公有的，那么类的作用域之内的所有代码(不论是在类的内部还是外部)都可以访问这个成员。下面的程序通过静态数据成员来记录链表首项和末项的地址。

#include iostream.h

#include string.h

class ListEntry

{

public:

static ListEntry\* firstentry;

private:

static ListEntry\* lastentry;

char\* listvalue;

ListEntry\* nextentry;

public:

ListEntry(char\*);

~ListEntry() { delete [] listvalue;}

ListEntry\* NextEntry() const { return nextentry; };

void display() const { cout< };

ListEntry\* ListEntry::firstentry;

ListEntry\* ListEntry::lastentry;

ListEntry::ListEntry(char\* s)

{

if(firstentry==0) firstentry=this;

if(lastentry!=0) lastentry->nextentry=this;

lastentry=this;

listvalue=new char[strlen(s)+1];

strcpy(listvalue,s);

nextentry=0;

}

int main()

{

while (1)

{

cout<<\nEnter a name ('end' when done): ;

char name[25];

cin>>name;

if(strncmp(name,end,3)==0) break;

new ListEntry(name);

}

ListEntry\* next = ListEntry::firstentry;

while (next != 0)

{

next->display();

ListEntry\* hold = next;

next=next->NextEntry();

delete hold;

}

return 0;

}

程序首先显示提示信息，输入一串姓名，以end作为结束标志。然后按照输入顺序来显示姓名。构造函数将表项加入链表，用new运算符来声明一个表项，但并没有把new运算符返回的地址赋值给某个指针，这是因为构造函数会把该表项的地址赋值给前一个表项的nextentry指针。

这个程序和前面将的逆序输出的程序都不是最佳方法，最好的方法是使用类模板，这在后面再介绍。

main()函数取得ListEntry::firstentry的值，开始遍历链表，因此必需把ListEntry::firstentry设置成公有数据成员，这不符合面向对象程序的约定，因为这里数据成员是公有的。

### 2.静态成员函数

成员函数也可以是静态的。如果一个静态成员函数不需要访问类的任何实例的成员，可以使用类名或者对象名来调用它。静态成员通常用在只需要访问静态数据成员的情况下。

静态成员函数没有this指针，因为它不能访问非静态成员，所以它们不能把this指针指向任何东西。

下面的程序中，ListEntry类中加入了一个静态成员函数FirstEntry()，它从数据成员firstentry获得链表第一项的地址，在这儿，firstentry已经声明为私有数据成员了。

#include iostream.h

#include string.h

class ListEntry

{

static ListEntry\* firstentry;

static ListEntry\* lastentry;

char\* listvalue;

ListEntry\* nextentry;

public:

ListEntry(char\*);

~ListEntry() { delete [] listvalue;}

static ListEntry\* FirstEntry() { return firstentry; }

ListEntry\* NextEntry() const { return nextentry; };

void display() const { cout< };

ListEntry\* ListEntry::firstentry;

ListEntry\* ListEntry::lastentry;

ListEntry::ListEntry(char\* s)

{

if(firstentry==0) firstentry=this;

if(lastentry!=0) lastentry->nextentry=this;

lastentry=this;

listvalue=new char[strlen(s)+1];

strcpy(listvalue, s);

nextentry = 0;

}

int main()

{

while (1)

{

cout<<\nEnter a name ('end' when done):;

char name[25];

cin >> name;

if(strncmp(name,end,3)==0) break;

new ListEntry(name);

}

ListEntry\* next = ListEntry::FirstEntry();

while (next != 0)

{

next->display();

ListEntry\* hold = next;

next = next->NextEntry();

delete hold;

}

return 0;

}

函数ListEntry::FirstEntry()是静态的，返回静态数据成员firstentry的值。

### 3.公有静态成员

如果一个静态成员象上面程序一样是公有的，那么在整个程序中都可以访问它。可以在任何地方调用公有景泰成员函数，而且不需要有类的实例存在。但公有静态成员函数不完全是全局的，它不仅仅存在于定义类的作用域内。在这个作用域里面，只要在函数名前加上类名和域解析运算符::就可以调用该函数。

# 二级C++辅导笔记：类和堆

## 一、构造函数和析构函数

前面的例子已经运用了new和delete来为类对象分配和释放内存。当使用new为类对象分配内存时，编译器首先用new运算符分配内存，然后调用类的构造函数；类似的，当使用delete来释放内存时，编译器会首先调用泪的析构函数，然后再调用delete运算符。

#include iostream.h

class Date

{

int mo,da,yr;

public:

Date() { cout< ~Date() { cout< }

int main()

{

Date\* dt = new Date;

cout< delete dt;

return 0;

}

程序定义了一个有构造函数和析构函数的Date类，这两个函数在执行时会显示一条信息。当new运算符初始化指针dt时，执行了构造函数，当delete运算符释放内存时，又执行了析构函数。

程序输出如下：

Date constructor

Process the date

Date destructor

## 二、堆和类数组

前面提到，类对象数组的每个元素都要调用构造函数和析构函数。下面的例子给出了一个错误的释放类数组所占用的内存的例子。

#include iostream.h

class Date

{

int mo, da, yr;

public:

Date() { cout< ~Date() { cout< }

int main()

{

Date\* dt = new Date[5];

cout< delete dt; //这儿

return 0;

}

指针dt指向一个有五个元素的数组。按照数组的定义，编译器会让new运算符调用Date类的构造函数五次。但是delete被调用时，并没有明确告诉编译器指针指向的Date对象有几个，所以编译时，只会调用析构函数一次。下面是程序输出；

Date constructor

Date constructor

Date constructor

Date constructor

Date constructor

Process the date

Date destructor

为了解决这个问题，C++允许告诉delete运算符，正在删除的那个指针时指向数组的，程序修改如下：

#include iostream.h

class Date

{

int mo, da, yr;

public:

Date() { cout< ~Date() { cout< }

int main()

{

Date\* dt = new Date[5];

cout< delete [] dt; //这儿

return 0;

}

最终输出为：

Date constructor

Date constructor

Date constructor

Date constructor

Date constructor

Process the date

Date destructor

Date destructor

Date destructor

Date destructor

Date destructor

## 三、重载new和delete运算符

前面已经介绍了如何用new和delete运算符函数来动态第管理内存，在那些例子中使用的都是全局的new和delete运算符。我们可以重载全局的new和delete运算符，但这不是好的想法，除非在进行低级的系统上或者嵌入式的编程。

但是，在某个类的内部重载new和delete运算符时可以的。这允许一个类有它自己的new和delete运算符。当一个类需要和内存打交道时，采用这种方法来处理其中的细节，可以获得很搞的效率，同时避免了使用全局new和delete运算符带来的额外开销。因为全局堆操作时调用操作系统函数来分配和释放内存，这样效率很低。

如果确定某个类在任何时候，其实例都不会超过一个确定的值，那么就可以一次性为类的所有实例分配足够的内存，然后用该类的new和delete运算符来管理这些内存。下面的程序说明了如何对new和delete进行重载。

#include iostream.h

#include string.h

#include stddef.h

#include new.h

const int maxnames = 5;

class Names

{

char name[25];

static char Names::pool[];

static bool Names::inuse[maxnames];

public:

Names(char\* s) { strncpy(name,s,sizeof(name)); }

void\* operator new(size\_t) throw(bad\_alloc);

void operator delete(void\*) throw();

void display() const { cout< };

char Names::pool[maxnames \* sizeof(Names)];

bool Names::inuse[maxnames];

void\* Names::operator new(size\_t) throw(bad\_alloc)

{

for(int p=0; p {

if(!inuse[p])

{

inuse[p] = true;

return pool+p\*sizeof(Names);

}

}

throw bad\_alloc();

}

void Names::operator delete(void\* p) throw()

{

if(p!=0)

inuse[((char\*)p - pool)/sizeof(Names)] = false;

}

int main()

{

Names\* nm[maxnames];

int i;

for(i=0; i {

cout< char name[25];

cin >> name;

nm[i] = new Names(name);

}

for(i=0; i {

nm[i]->display();

delete nm[i];

}

return 0;

}

上面的程序提示输入5个姓名，然后显示它们。程序中定义了名为Names的类，它的构造函数初始化对象的name值。这个类定义了自己的new和delete运算符。这是因为程序能保证不会一次使用超过maxnames个姓名，所以可以通过重载默认的new和delete运算符来提高运行速度。

Names类中的内存池是一个字符数组，可以同时容纳程序需要的所有姓名。与之相关的布尔型数组inuse为每个姓名记录了一个true和false值，指出内存中的对应的项是否正在使用。

重载的new运算符在内存池中寻找一个没有被使用的项，然后返回它的地址。重载的delete运算符则标记那些没有被使用的项。

在类定义中重载的new和delete运算符函数始终是静态的，并且没有和对象相关的this指针。这是因为编译器会在调用构造函数之前调用new函数，在调用析构函数后调用delete函数。

new函数是在类的构造函数之前被调用的。因为这时内存中还不存在类的对象而且构造函数也没有提供任何初始化值，所以它不可以访问类的任何成员。同理，delete运算符是在析构函数之后被调用的，所以它也不可以访问类的成员。

## 四、异常监测和异常处理

### 1.检测异常

上面的例子还缺少必要的保护机制。比如，重载的delete运算符函数并没有检查它的参数，确认其是否落在内存池内部。如果你绝对相信自己编的程序中不会传递错误的指针值给delete运算符，那么可以省掉合法性检查以提高效率，特别是在优先考虑效率的程序中。否则应该使用预编译的条件语句。在软件的测试版本中加入这些检测，在正式的发行版本中去掉这些检查。

### 2.重载new和delete中的异常处理

上面的两个重载运算符函数都是用了异常处理。异常处理是C++的新内容之一，目前还没有讲到。在这里不必关心它是如何工作的。上面程序中，当试图分配超过内存池容量的Names缓冲区，重载的new运算符函数就会抛出异常，终止程序。

五、重载new[]和delete[]

对于上面的程序，假如有下面的语句:

Names \*nms=new Names[10]

...

delete [] nms;

那么，这些语句会调用全局new和delete运算符，而不是重载过的new和delete。为了重载能为对象数组分配内存的new和delete运算符，必须像下面的程序一样，对new[]和delete[]也进行重载。

#include iostream.h

#include string.h

#include stddef.h

#include new.h

const int maxnames = 5;

class Names

{

char name[25];

static char Names::pool[];

static bool Names::inuse[maxnames];

public:

Names(char\* s) { strncpy(name,s,sizeof(name)); }

void\* operator new(size\_t) throw(bad\_alloc);

void operator delete(void\*) throw();

void display() const { cout< };

char Names::pool[maxnames \* sizeof(Names)];

bool Names::inuse[maxnames];

void\* Names::operator new[](size\_t size) throw(bad\_alloc)

{

int elements=size/sizeof(Names);

int p=-1;

int i=0;

while((i {

if(!inuse[i]) p=i;

++i;

}

// Not enough room.

if ((p==-1) || ((maxnames-p) for(int x=0; x return pool+p\*sizeof(Names);

}

void Names::operator delete[](void\* b) throw()

{

if(b!=0)

{

int p=((char\*)b- pool)/sizeof(Names);

int elements=inuse[p];

for (int i=0; i }

}

int main()

{

Names\* np = new Names[maxnames];

int i;

for(i=0; i {

cout< char name[25];

cin >> name;

\*(np + i) = name;

}

for(i=0; idisplay();

delete [] np;

return 0;

}

重载new[]和delete[]要比重载new和delete考虑更多的问题。这是因为new[]运算符时为数组分配内存，所以它必须记住数组的大小，重载的delete[]运算符才能正确地把缓冲区释放回内存池。上面的程序采用的方法比较简单，吧原来存放缓冲区使用标志的布尔型数组换成一个整型数组，该数组的每个元素记录new[]运算符分配的缓冲区个数，而不再是一个简单的true。当delete[]运算符函数需要把缓冲区释放回内存池时，它就会用该数组来确认释放的缓冲区个数

# 二级C++辅导笔记：类的其他几点问题

## 一、拷贝构造函数

拷贝构造函数在下列情况下被调用：用已经存在的对象去初始化同一个类的另一个对象；在函数的参数中，以传值方式传递类对象的拷贝；类对象的值被用做函数的返回值。拷贝构造函数和前面说到的转换构造函数有些相似。转换构造函数是把一个类的对象转化为另一个类的对象；拷贝构造函数是用一个已经存在的对象的值实例化该类的一个新对象。

不同对象间的初始化和赋值的区别：赋值操作是在两个已经存在的对象间进行的；而初始化是要创建一个新的对象，并且其初值来源于另一个已存在的对象。编译器会区别这两种情况，赋值的时候调用重载的赋值运算符，初始化的时候调用拷贝构造函数。

如果类中没有拷贝构造函数，则编译器会提供一个默认的。这个默认的拷贝构造函数只是简单地复制类中的每个成员。

#include iostream.h

#include string.h

class Date

{

int mo, da, yr;

char\* month;

public:

Date(int m = 0, int d = 0, int y = 0);

Date(const Date&);

~Date();

void display() const;

};

Date::Date(int m, int d, int y)

{

static char\* mos[] =

{

January, February, March, April, May, June,

July, August, September, October, November, December

};

mo = m; da = d; yr = y;

if (m != 0)

{

month = new char[strlen(mos[m-1])+1];

strcpy(month, mos[m-1]);

}

else

month = 0;

}

Date::Date(const Date& dt)

{

mo = dt.mo;

da = dt.da;

yr = dt.yr;

if (dt.month != 0)

{

month = new char [strlen(dt.month)+1];

strcpy(month, dt.month);

}

else

month = 0;

}

Date::~Date()

{

delete [] month;

}

void Date::display() const

{

if (month != 0)

cout << month <<' '<< da << , << yr << std::endl;

}

int main()

{

Date birthday(6,24,1940);

birthday.display();

Date newday = birthday;

newday.display();

Date lastday(birthday);

lastday.display();

return 0;

}

本例中，用到了两次拷贝构造函数。一个是使用普通的C++初始化变量的语句：

Date newday = birthday;

另一个是使用构造函数的调用约定，即把初始化值作为函数的参数：

Date lastday(birthday);

## 二、类的引用

在函数参数和返回值中，如果一定要使用传值方式，那么使用类对象的引用，是一个提高效率的方法。

类的数据成员也可以是一个引用，但必须注意：第一，一个引用必须初始化。通常一个类对象并不会像结构那样用大括号来初始化，而是调用构造函数。因此在构造函数里必须初始化类当中的引用成员。第二，引用是一个别名。尽管类里面的引用在使用方式上看起来和类的一般数据成员没有什么区别，但是作用在其上的操作，实际上是对用来初始化它的那么对象进行的。

#include iostream.h

class Date

{

int da, mo, yr;

public:

Date(int d,int m,int y)

{ da = d; mo = m; yr = y; }

void Display() const

{ cout << da << '/' << mo << '/' << yr; }

};

class Time

{

int hr, min, sec;

public:

Time(int h, int m, int s)

{ hr = h; min = m; sec = s; }

void Display() const

{ cout << hr << ':' << min << ':' << sec; }

};

class DateTime

{

const Date& dt;

const Time& tm;

public:

DateTime(const Date& d, const Time& t) : dt(d), tm(t)

{

//empty

}

void Display() const

{

dt.Display();

cout << ' ';

tm.Display();

}

};

int main()

{

Date today(7,4,2004);

Time now(15,20,0);

DateTime dtm(today, now);

dtm.Display();

return 0;

}

我们来看看这个程序中DateTime的构造函数的格式：冒号操作符引出了一个参数初始化表。必须使用这种格式来初始化引用数据成员，而不可以在函数体内来进行初始化工作。如果构造函数像上例一样不是内联的，那么最好不要在类声明中构造函数的原型上使用冒号和初始化值表，而是像下面这样，把参数初始化表放在定义构造函数的地方：

Class DateTime

{

const Date& dt;

const Time& tm;

public:

DateTime(const Date& d,const Time& t);

}

DateTime::DateTime(const Date& d,const Time& t):dt(d),tm(t)

{

//empty

}

可以使用构造函数的参数初始化表来初始化任何数据成员。特别是常量数据成员，和引用一样，只能在参数初始化表里进行初始化，这是因为不可以在构造函数内部为常量数据成员赋值。

当一个类含有引用数据成员时，一旦引用被实例化和初始化以后，就无法修改它的值，所以该类不可能彻底地重载赋值运算符函数。

## 三、构造函数的参数初始化表

如果类对象的某些数据成员没有载构造函数内部被初始化，那么必须使用构造函数的参数初始化表对他们进行初始化。否则，编译器不止到该如何初始化这些还等着在构造函数内部赋值的成员。我们习惯用参数初始化表来初始化所有数据成员。

class Date

{

int mo,da,yr;

public:

Date(int m=0,int d=0,int y=0);

};

class Employee

{

int empno;

Date datehired;

public:

Employee(int en,Date& dh);

};

可以用下面两种方法编写Employee类的构造函数：

Employee::Employee(int en,Date& dt)

{

empno=en;

datehired=dh;

}

或者；

Employee::Employee(int en,Date& dt):empno(en),datehired(dh)

{

//empty

}

虽然这两种方法效果是一样的，但是根据Date对象默认构造函数的复杂性的不同，这两种形式的效率差别是很大的。

## 四、对const修饰符的简单说明

如果一个对象被声明为常量，那么该对象就不可以调用类当中任何非常量型的成员函数(除了被编译器隐式调用的构造函数和析构函数)。看下面的代码；

#include iostream.h

class Date

{

int month,day,year;

public:

Date(int m,d,y):month(m),day(d),year(y) {}

void display()

{

cout< }

}

int main()

{

const Date dt(4,7,2004);

dt.display(); //error

return 0;

}

这个程序尽管编译时没有问题，但运行时却出错了。这是因为常量对象不能调用非常量函数。编译器只看函数的声明，而不在乎函数的具体实现。实际上函数的实现可以在程序中的任何地方，也可以是在另一个源代码文件中，这就超过了编译器的当前可见范围。

//date.h

class Date

{

int month,day,year;

public:

Date(int m,d,y);

void display();

};

//date.cpp

#include iostream.h

#include date.h

Date::Date(int m,d,y):month(m),day(d),year(y) {}

void Date::display()

{

cout< }

//program.cpp

#include iostream.h

#include date.cpp

int main()

{

const Date dt(4,7,2004);

dt.display();

return 0;

}

解决出错的问题有两个方法：第一是声明display()函数为常量型的

//in date.h

void display() const

//int date.cpp

void Date::display() const

{

cout< }

另一个解决方式就是省略掉Date对象声明里的const修饰符。

Date dt(4,7,2004);

还有另一个容易出错的地方：

void abc(const Date& dt)

{

dt.display(); //error 提示display没有const修饰符

}

函数abc()声明了一个Date对象的常量引用，这说明该函数不会修改传递进来的参数的值。如果Date::display()函数不是常量型的，那么在函数abc()里就不能调用它，因为编译器会认为Date::display()函数有可能会修改常量的值。

不论类对象是否是常量型的，它必须修改某个数据成员的值时，ANSI委员会设立了mutable关键字。

## 五、可变的数据成员

假设需要统计某个对象出现的次数，不管它是否是常量。那么类当中就应该有一个用来计数的整型数据成员。只要用mutable修饰符来声明该数据成员，一个常量型的成员函数就可以修改它的值。

#include iostream.h

class AValue

{

int val;

mutable int rptct;

public:

AValue(int v) : val(v), rptct(0) { }

~AValue()

{

cout< }

void report() const;

};

void AValue::report() const

{

rptct++;

cout << val << endl;

}

int main()

{

const AValue aval(123);

aval.report();

aval.report();

aval.report();

return 0;

}

# 二级C++辅导笔记：重载运算符

重载是由P.J.Plauger发现的。

## 一、重载运算符的时机

1、需要在定义的对象间相互赋值时，重载赋值运算符

2、需要在数字类型增加算术属性时，重载算术运算符

3、需要为定义的对象进行逻辑比较时，重载关系运算符

4、对于container，重载下标运算符[]

5、需要从I/O流中读写对象时，重载<<和>>运算符。

6、重载成员指针运算符 -> 以实现smart指针

7、在少数情况下重载new,delete运算符

8、不重载其他运算符

实际上任何用重载运算符完成的工作都可以使用成员函数来实现。

重载的运算符可以和原来的运算符不一定有必然联系，例如我重载'+'运算符，可以不做加法运算，而是把字符串连接起来。当然你要是用'+'运算符来做减法运算，也是可以的，不过这不是明智之举。

## 二、重载运算符的规则

1、重载的运算符不能违反语言的语法规则

2、如果一个运算符可以放在两个操作数之间，就可以重载它来满足类操作的需要，哪怕这种用法原本为编译器不能接受。

3、不能创造C++语言中没有的运算符

4、下列运算符不能重载

. 类成员运算符

.\* 成员指针运算符

:: 域解析运算符

?: 条件表达式运算符

5、重载时不能改变运算符的优先级

## 三、运算符重载

运算符重载是通过对运算符函数的重载来实现的。对于每一个运算符@，在C++中都对应一个运算符函数operator@，其中@为C++各种运算符。

运算符函数的一般原型为：

type operator@ (arglist)

其中type为运算结果的类型，arglist为操作数列表。

# 二级C++辅导笔记：重载双目运算符

在（五）我们已经介绍了重载赋值运算符，这里就不重新说明了。

## 一、作为类成员函数的重载

为了能进行类对象和一个整型值的加法运算，需要写一个类的成员函数来重载双目加法(+)运算符。该函数在类中的声明如下：

Date operator + (int) const;

函数的声明指出，返回值是一个Date类对象，函数名是运算符+，只有一个整型参数，而且函数是常量型的。当编译器发现某个函数以加上前缀operator的真实运算符作为函数名，就会把该函数当作重载运算符函数来处理。如果在表达式中，该运算符的左边是一个类对象，右边是一个参数类型的一个对象，那么重载运算符函数就会被调用。调用形式如下：

Date dt(6,9,2005);

dt=dt+100;

也可以显式的调用重载运算符函数：

dt.operator + (100);

下面代码重载了双目加法运算符来计算一个整数和一个Date类对象之和，并且返回Date类对象。

#include iostream.h

class Date

{

int mo,da,yr;

static int dys[];

public:

Date(int m=0,int d=0,int y=0)

{ mo=m; da=d; yr=y;}

void display() const

{ cout< Date operator + (int) const;

};

int Date::dys[]={31,28,31,30,31,30,31,31,30,31,30,31};

Date Date::operator+(int) const

{

Date dt=\*this;

n+=dt.da;

while(n>=dys[dt.mo-1])

{

n-=dys[dt.mo-1];

if(++dt.da==13)

{

dt.mo=1;

dt,yr++;

}

}

dt.da=n;

return dt;

}

int main()

{

Date olddate(1,1,2005);

Date newdate;

newdate=olddate+100;

newdate.display();

return 0;

}

## 二、非类成员的运算符重载

在重载运算符的原则中说到，要保持运算符的可交换性。而上面的程序只允许Date类对象在运算符的左边而整型值在右边，不支持下面的语句：

Date newdate=100+olddate;

所以，仅仅靠一个类的成员重载运算符是无法实现上面功能的。对重载双目运算符的类成员函数来说，总是认定调用函数的对象位于运算符左边。不过，我们可以再写一个非类成员的重载运算符函数，可以规定Date类的对象在运算符右边，而别的类型在运算符左边。例如，我们可以这样在类的外部定义一个函数：

Date operator + (int n,Date& dt)

下面代码在原先的基础上增加了一个非类成员函数来实现双目加法运算符的重载。

#include iostream.h

class Date

{

int mo,da,yr;

static int dys[];

public:

Date(int m=0,int d=0,int y=0)

{ mo=m; da=d; yr=y;}

void display() const

{ cout< Date operator + (int) const;

};

int Date::dys[]={31,28,31,30,31,30,31,31,30,31,30,31};

Date Date::operator+(int) const

{

Date dt=\*this;

n+=dt.da;

while(n>=dys[dt.mo-1])

{

n-=dys[dt.mo-1];

if(++dt.da==13)

{

dt.mo=1;

dt,yr++;

}

}

dt.da=n;

return dt;

}

Date operator + (int n,Date& dt)

{

return dt+n;

}

int main()

{

Date olddate(1,1,2005);

Date newdate;

newdate=olddate+100;

newdate.display();

return 0;

}

上面的例子中非类成员重载运算符函数调用了类中的重载+运算符来实现加法运算。如果类当中没有提供这样的函数，那么非类成员的重载运算符函数将被迫访问类的私有数据来实现加法运算。这样的话，需要把这个函数声明为类的友元，如下：

class Date

{

friend Date operator + (int n,Date&);

};

上例中重载运算符函数声明了全部两个参数，这是因为它不是类的成员，因此它不能作为类的成员函数被调用，就缺少了一个隐含的参数。

第一个重载加法运算符函数也可以用类的友元函数来实现。作为一种约定，这通常把所有为类重载的运算符都设定为该类的友元。

例子中只给出了重载加法的代码，我们同样可以来重载减法，乘除法等等。

## 三、重载关系运算符

如果想要对两个日期进行比较，比如出现下面这样的代码：

if(olddate 可以向上面用类似的方法重载关系运算符

#include iostream.h

class Date

{

int mo,da,yr;

public:

Date(int m=0,int d=0,int y=0)

{ mo=m; da=d; yr=y;}

void display() const

{ cout< int operator == (Date& dt) const;

int operator < (Date& dt) const;

};

int Date::operator== (Date& dt) const

{

return (this->mo==dt.mo && this->da==dt.da && this->yr==dt.yr);

}

int Date::operator < (Date& dt) const

{

if(this->yr == dt.yr)

{

if(this->mo == dt.mo) return this->da < dt.da;

return this->mo < dt.mo;

}

return this->yr < dt.yr;

}

int main()

{

Date date1(2,14,2005);

Date date2(6,9,2005);

Date date3(2,14,2005);

if(date1 {

date1.display();

cout< date2.display();

}

cout< if(date1==date3)

{

date1.display();

cout< date3.display();

}

return 0;

}

可以类似的重载其他关系运算符，如!=

int operator != (Date& dt) { return !(\*this==dt);}

## 四、其他赋值运算符

#include iostream.h

class Date

{

int mo,da,yr;

static int dys[];

public:

Date(int m=0,int d=0,int y=0)

{ mo=m; da=d; yr=y;}

void display() const

{ cout< Date operator + (int) const;

Date operator +=(int)

{ \*this=\*this+n; return \*this;}

};

int Date::dys[]={31,28,31,30,31,30,31,31,30,31,30,31};

Date Date::operator+(int) const

{

Date dt=\*this;

n+=dt.da;

while(n>=dys[dt.mo-1])

{

n-=dys[dt.mo-1];

if(++dt.da==13)

{

dt.mo=1;

dt,yr++;

}

}

dt.da=n;

return dt;

}

int main()

{

Date olddate(1,1,2005);

olddate+=100;

olddate.display();

return 0;

}