

运算符重载

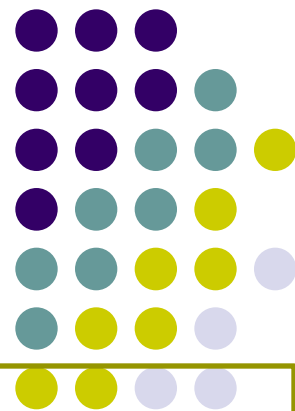
探讨：让类对象也能完成运算符的直观操作，如：

对象+对象

实际也是把类设计的更加强大

吴清锋
2021年春

这实际是一个问题的两个视角



提纲

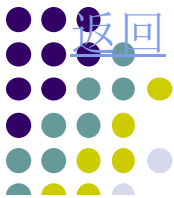
- 概述
- 引进运算符重载的必要性
- 不同类型的运算符的重载讨论

回顾：你所认识的运算符

- 有哪些运算符？
 - 1、算术运算符
 - 2、关系运算符
 - 3、逻辑运算符
 - 4、赋值运算符 ...
- 运算符三大要素：
 - 1、运算符优先级
 - 2、运算对象（操作数）类型和个数
 - 3、运算符执行过程
- 程序设计语言提供了一系列对数据进行操作的操作符。神奇的运算符，可以支持**不同类型数据**的操作。例：
运算符+： int+int 或是float +float 甚至是int+double、char+int
运算符/： 能对整数、单精度数和双精度数进行“/”运算，但是表现出来的结果可能不一样！
- 从效果上看，对于用户而言，同一个运算符，能够支持不同类型数据的操作，这实际就是重载！

回顾：一起走过的“重载”

- 回顾：引进函数重载的背景和意义
 - 经常有此类现象，有着不同的函数名字多个函数其功能是大致相同的。期望：给这些功能相同的函数起一个名字，只是它们各自的函数体不同，对应着不同的类型的数据；
 - C++中引进重载函数：允许**同一个函数名**对应着不同的实现，即各自有自己的函数体。
 - 函数重载的实质：函数重载就是对一个已有的函数赋予新的含义，使之实现新的功能。即一个函数名可以用来代表不同功能的函数。典型的例子——构造函数重载
- 运算符也可以重载，即对已有的运算符赋予多重含义，**同一个运算符**作用于不同类型的数据导致不同类型的行为；



引进运算符重载的必要性

- 期望：在定义某类时，希望**用户**在对类对象的操作也能使用传统的运算符
- **作为用户**，在标准库类型中，曾遇见**运算符重载**的问题，在讨论“和C-字符串常量的连接”中：
string s1=“hello”+“，”
//C-串不支持“+”
string s2=s1+“， ” +“world”
//string通过运算符重载，支持string的+功能；且前两部分运算后结果为string，正是它的存在能唤起它调用string类的“+”（而不是调用整型的+），因此能与“world”再做“+”
- 现在：我们**要作为一个开发人员**，考虑程序能支持运算符重载,赋予C++中的运算符新的含义，适应类编程，使类更像C++的基本数据类型，让对象可直接使用运算符



回顾：以前我们是如何做的？

- 当前是如何实现的？
一个实际的问题
- 总的来说：
用户期望更加“自然的”、“符合使用习惯”的运算符使用方式；
例子无法满足，但是不妨碍它成为一个参考。

基本思路

Complex obj3=obj1+obj2;



运行机理

函数是运算符重载
的基础

函数声明和定义

main函数中应用

成员函数

显示调用

隐式调用

一般全局函数

显示调用

隐式调用

函数类型 operator 运算符名称(形参列表)

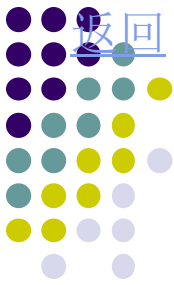
基础



以函数基础



运算符重载：通过函数来实现！



- 运算符函数的一般格式如下：

函数类型 **operator** **运算符名称** (形参列表)

{ 对运算符的重载处理 }

- operator**是关键字，专用于定义重载运算符函数；
- “operator 运算符”是函数名.若对于使用字母字符的运算符，如new等，在operator和运算符之间至少有一个空格，对于其他运算符，空格是可选的；
- 运算符表达式与函数执行之间对应关系
 - 触发机制：执行**含有重载的运算符**的表达式，系统就会调用“**operator 运算符**”函数；
 - 执行情况：**表达式操作数**是函数的参数，函数返回值是运算结果；

```
if (box1 < box2) { }
```

```
bool Box::operator<(const Box& aBox)
```

```
if (box1.operator<(box2)) { }
```


基本思路



运行机理

函数声明和定义

运算符重载

成员函数

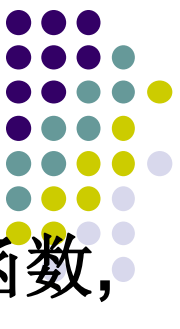
要比较：参数个数
要区分：特定应用情景
要考虑：函数定义和调用

一般全局函数

基础



函数基础



成员运算符函数的定义

- 在C++中,可以把运算符函数定义成某个类的成员函数,称为成员运算符函数。
- 成员运算符函数的原型
在类的内部声明格式如下:

```
class X {  
    //...  
    返回类型 operator运算符(形参表);  
    //...  
};
```

本质是个成员函数!

- 1 有访问属性的限定;
- 2 有类域的限定;
- 3 有**this**指针;
- 4 调用形式: 对象.成员函数;

在类外定义成员运算符函数的格式如下:

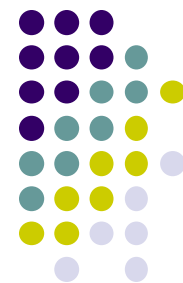
```
返回类型 X::operator运算符(形参表)  
{  
    函数体  
}
```



成员运算符函数的调用

- 通过函数的调用完成了运算符的操作
- 有两种形式，以“+”为例
 - 显示调用
对象.operator +()
 - 函数的调用形式，但不符合用户使用习惯
 - 隐式调用 【自动召唤神龙】
对象1+对象2
- 调用成员函数的对象是运算符的一个运算对象

例子:



```
class Complex {
```

```
    double real,imag;
```

```
public: //省略构造函数等
```

```
    Complex operator + (const Complex &x) const {
```

```
        Complex temp;
```

```
        temp.real=real+x.real;
```

```
        temp.imag=imag+x.imag;
```

```
        return temp;
```

```
    }
```

```
Complex a(10,20);
```

```
Complex b(a);
```

```
Complex sum;
```

- ◆ 途径一: `sum=a.operator +(b);` //通过函数调用来使用
- ◆ 途径二: `sum=a+b;` //通过+直接,由于操作数是类对象,找寻对应函数

首先把指定的运算表达式转化为对运算符函数的调用,运算目标转化为运算符函数的实参,然后根据实参的类型来确定需要调用的函数,这个过程是在编译过程中完成的.由于是类对象,则调用重载的函数.

友元运算符函数定义的语法形式



- 友元运算符函数的原型在类的内部声明格式如下:

```
class X {
```

```
    //...
```

```
friend 返回类型 operator运算符(形参表);
```

```
    //...
```

```
}
```

在类外定义友元运算符函数的格式如下:

```
返回类型 operator运算符(形参表)
```

```
{
```

```
    函数体
```

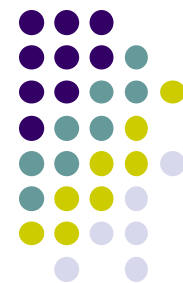
```
}
```

- 函数定义时, 根据运算符所需要的运算对象来考虑参数格式

本质是个一般全局函数!

- 1 无访问属性的限定;
- 2 通过友元与类关联;
- 3 无this指针;

例子:



```
Class Complex {  
    double real,imag;  
public: //省略构造函数等  
    friend complex operator + (const Complex &x, const Complex &y) ;  
};
```

```
Complex operator + (const Complex &x, const Complex &y) {  
    Complex temp;  
    temp.real=x.real+y.real;  
    temp.imag=x.imag+y.imag;  
    return temp;  
}
```

```
Complex a(10,20);  
Complex b(a);  
Complex sum;
```

- ◆ `sum=operator +(a , b);` //通过函数调用来使用
- ◆ `sum=a+b;` //通过+直接,由于操作数是类对象, 找寻对应函数

再一个例子：对<运算符进行重载

- 例子

- 代码分析

```
bool Box::operator<(const Box& aBox) const {  
    return volume()<aBox.volume();  
} //运算符重载函数定义在类外,且调用其他成员函数
```

```
bool operator<(const Box& aBox,const Box& bBox) {  
    return aBox.volume()<bBox.volume();  
} //运算符重载函数为一般函数
```

比较下，函数定义之间究竟有什么差别

```
if (box1.operator<(box2))
```

//翻译成函数调用

```
if (box1<box2) {  
    ....  
} //直接使用<运算符
```

```
If(operator<(box1,box2))
```

```
....  
} //翻译成函数调用
```



后面教学与学习思路

- 运算符种类

- 双目运算符

- - 单目运算符

分别从：成员函数和一般函数来分析

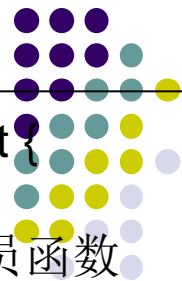
有归纳总结的本意！

- 若干特殊运算符

成员函数视角： 双目运算符重载



```
bool operator<(const Box& aBox) const {  
    return volume()<aBox.volume();  
} //重载函数定义在类中,且调用其他成员函数
```



- 对双目运算符而言,成员运算符函数的**形参表**中仅有一个参数,它作为运算符的右操作数,此时当前对象作为运算符的左操作数,它是通过**this**指针隐含地传递给函数的。
- 调用**——例如: **box1<box2**
一般而言,如果在类X中采用成员函数重载双目运算符@,成员运算符函数**operator@**所需的一个操作数由对象**aa**通过**this**指针隐含地传递,它的另一个操作数**bb**在参数表中显示,**aa**和**bb**是类X的两个对象,则以下两种函数调用方法是等价的:

aa @ bb;

// 隐式调用 **box1<box2**

aa.operator @(bb); // 显式调用 **box1.operator<box2**

友元视角： 双目运算符重载



- 当用友元函数重载双目运算符时,两个操作数都要传递给运算符函数。

```
bool operator<(const Box& aBox,const Box& bBox) {  
    return aBox.volume()<bBox.volume();  
} //运算符重载函数为一般函数
```

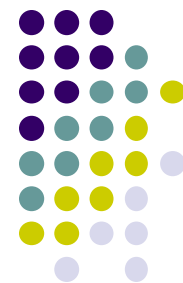
调用形式

一般而言,如果在类X中采用友元函数重载双目运算符@,而aa和bb是类X的两个对象,则以下两种函数调用方法是等价的:

aa @ bb; // 隐式调用

operator @(aa,bb); // 显式调用

成员函数视角： 单目运算符重载



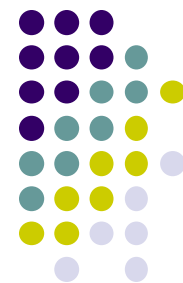
- 对单目运算符而言,成员运算符函数的**参数表**中没有参数,此时当前对象作为运算符的一个操作数
- 调用——例如: **-c1**
一般而言,采用成员函数重载单目运算符时,以下两种方法是等价的:

@aa; // 隐式调用

aa.operator @(); // 显式调用

成员运算符函数**operator @**所需的一个操作数由对象**aa**通过**this**指针隐含地传递。因此,在它的参数表中没有参数。

友元视角： 单目运算符重载



- 对单目运算符而言,友元运算符函数的**参数表**中有参数,操作数作为函数的参数
- 调用——例如: **-c1**
一般而言,采用友元重载单目运算符时,以下两种方法是等价的:

@aa; // 隐式调用

operator @(aa); // 显式调用

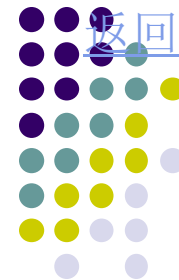
友元运算符函数**operator @**所需的一个操作数由对象**aa**作为函数的实参来传递。



成员运算符函数与友元运算符函数的比较

- 1、**参数：**对双目运算符而言,成员运算符函数带有一个参数,而友元运算符函数带有两个参数;对单目运算符而言,成员运算符函数不带参数,而友元运算符函数带一个参数。
- 2、**调用：**成员运算符函数和友元运算符函数可以用习惯方式调用,也可以用它们专用的方式调用。
- 3、**C++的大部分运算符**既可说明为成员运算符函数,又可说明为友元运算符函数。究竟选择哪一种运算符好一些,没有定论,这主要取决于实际情况和程序员的习惯。

成员函数与一般函数的抉择



- 之前的基本<运算符重载为比较同类型的两个数据元素;
- 问题复杂: 对Box1<25.0或是10<Box2型表达式的重载
 - 比较自定义类型对象（第一个操作数）和double类型的第二个操作数（重载为成员函数）

```
bool operator<(double aValue) const {  
    return volume()<aValue;  
} //Box对象传送给函数，作为隐式的this指针，double则传送为一个参数
```

- 再：比较double类型（第一个操作数）和自定义类型对象（第二个操作数）
 - 成员运算符函数总是把this指针提供为作操作数，而在这个表达式中，左操作数为double型，因此不能把运算符实现为成员函数，应该实现为全局运算符函数或友元函数。

```
bool operator<(const double aValue,const Box& aBox) {  
    return aValue< aBox.volume();  
}
```

运算符重载的运行机制

- 应保证运算符重载后的版本与其一般用法保持一致。如重载`+`运算符执行类对象中的相乘操作是不妥的。
- 那么，同一个运算符能支持不同类型的操作数，编译系统如何判别该执行哪一个功能？
根据表达式的上下文决定，即根据运算符两侧的数据类型。换句话说：若操作数是基本类型，就不会调用运算符重载函数
运算符重载的实质是通过函数来实现的，即：编写一个函数，重新定义某个运算符，使之在每次应用于类类型的对象时，系统自动调用该函数，以执行指定的操作；

重载运算符的规则

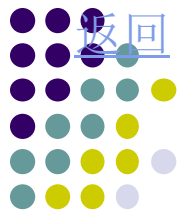
- C++ **不允许** 用户自己定义（发明）新的运算符，只能对已有的C++运算符进行重载；
- 重载时，不能修改运算符的优先级、结合性和操作数的个数；
- C++ **不能** 重载的运算符有：
 - . 成员访问运算符
 - .* 间接成员选择符
 - :: 域运算符
 - sizeof 长度运算符
 - ? : 条件运算符
- 重载运算符的函数不能有默认的参数；
- 重载的运算符必须和用户定义的自定义类型的对象一起使用，其参数至少应有一个是类对象；



若干值得注意的运算符

- 后面将讨论几种重要的运算符
除了一般规律外，需要注意这些运算符的特质
- 包括：
 - 赋值=运算符
 - << >>运算符
 - ++和--
 - () 和[]

重载：赋值运算符=



- 赋值运算符能把给定类型的对象复制到同一类型的另一个对象中。
- 如果没有提供重载的赋值运算符来复制类的对象,编译器会提供默认版本 `operator=()`;

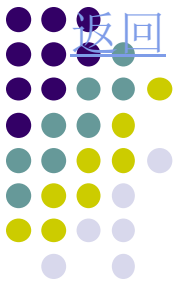
```
Cube& operator=(const Cube& aBox) {  
    side=aBox.side; //实现对各个成员的赋值操作  
    return *this; //this指针指向当前对象，return返回的就是当前对象  
} //调用形式就是box1=box2;
```

- `operator=()`的参数是一个常对象引用，避免对原对象的修改；
- `operator=()`的返回类型为对象的引用(引用,可避免不必要的复制操作，能作为左值)
- 函数体**return**语句：`return *this`
- 对于本体与实体不一致的情况，若没有自定义重载赋值运算符，会出现两个对象相互依赖的现象，即：两个对象共享样本对象中自定义的存储区。[图例](#)



如何写：赋值运算符形式参数的思考

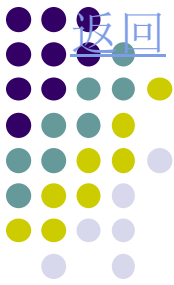
- **Cube& operator= (const Cube& aBox) { }**
- 形参中：
 - const意味着什么？
 - &意味着什么？
 - 如何实现引用的绑定？
 - 绑定时：实参不需要地址&；使用时，形参不要*
 - 引用没有占用地址



如何写：赋值运算符返回类型的思考1

Cube& operator= (**const Cube&** aBox) { }

- 第一个问题：为何返回值不是**void**?
基本功能可以实现load1=load2
- 但是，若有：load1=load2=load3时，由于
load1=load2=load3等价load1=(load2=load3)
假设成员函数operator=()，它等价于
load1.operator=(load2.opertaor=(load3));
可以看出，从**operator=()**返回的内容是另一个
operator=()调用的参数，此时返回值为**void**不合适



如何写：赋值运算符返回类型的思考2

- 第二个问题：若是返回本类对象，会如何？

```
Complex Complex::operator+=(const Complex& aBox) {  
    real+=c.real; //实现对各成员的赋值操作  
    image+=c.image;  
    return *this;  
}
```

```
Complex Complex::operator+=(const Complex& aBox) {  
    Complex temp;  
    real+=c.real; //实现对各成员的赋值操作  
    image+=c.image;  
    temp= *this;  
    return temp;  
}
```

- 用返回的对象值初始化 **内存临时对象（调用拷贝构造函数）**，内存临时对象作为调用函数的结果。
- 由于返回对象值需要调用拷贝构造函数初始化内存临时对象，因此若对象有动态分配空间，则需定义拷贝构造函数

如何写：赋值运算符返回类型的思考3

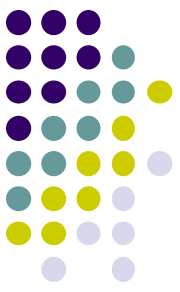
- 第三个问题：若是返回本类对象的引用，会如何？

```
Complex & Complex::operator+=(const Complex& aBox) {  
    real+=c.real; //实现对各成员的赋值操作  
    image+=c.image;  
    return *this; //返回值为对象引用不需要初始化内存临时对象，返回值是  
                //对象本身；效率高。  
}
```

- 返回的是对象的引用。系统要求执行流程返回到调用点时，返回值是存在的。而temp的auto特性，在返回前，是被撤销，所以有问题

```
Complex& Complex::operator+=(const Complex& aBox) {  
    Complex temp;  
    real+=c.real; //实现对各成员的赋值操作  
    image+=c.image;  
    temp= *this;  
    return temp;  
}
```

- 返回对象的引用，可以作为左值，能实现连续的书写



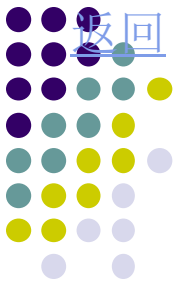
重载赋值运算符——如何实现

- **在定义上**，需要注意三个问题：
 - 函数的形参该如何写？
 - 函数的返回值问题？
 - 函数体该如何书写？
- 从上分析可以得到
 - 函数必须返回一个对象，该对象就是函数的左操作数；而且，如果为了避免不必要的复制函数调用操作，返回类型必须是该对象的引用



赋值运算符重载和拷贝构造函数

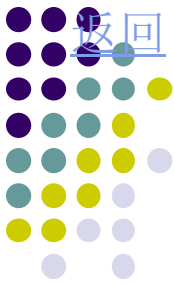
- 一般而言，需要自定义拷贝构造函数的类也需要自定义赋值运算符重载函数；
- 定义对象时给对象赋初值，此时调用的是**拷贝构造函数**；
- 程序语句部分的赋值语句，此时调用的是**赋值运算符重载函数**；



重载赋值运算符的其他问题

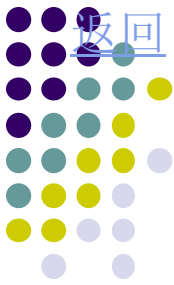
- 赋值操作符无论形参为何种类型，赋值操作符必须定义为**成员函数**；
- 反思：如果是一般函数（友元）会如何？

简单来说，可能会造成左边的值是个常量的问题。



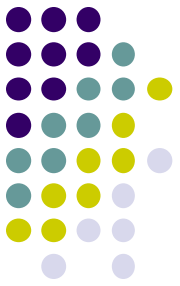
重载流输入输出运算符——概述

- C++的输入输出都是流来处理的。
- 编译系统在类库中提供：
 - 输入流类istream：管输入的流 → 有对象：cin
 - 输出流类ostream：管输出的流 → 有对象：cout
- 流插入运算符 “<<”和流提取运算符 “>>”是在C++类库中提供的；
- 可使用cout<<和cin>>对标准类型数据进行输入输出；



重载输入输出运算符的必要性

- 问题：
用户自己定义的类型对象，是**不能**直接用<<和>>来输出和输入的。
那现在是如何做到的？是写一个display或是myprin
- 如果想用它们输出和输入自己声明的类型的对象，必须对它们重载；
可以发现，运算符重载函数可以以display或是myprin为基础（函数体功能类似）



预备: cout的引用

程序中, 用户自己定义一个ostream的对象的引用s, 则s也可以使用运算符 “<<” .

```
#include <iostream>
int main( )
{ int num=10;
  ostream & scout=cout; //类对象cout的引用
  scout<<num<<endl;
  return 0;
}
```

<<和>>运算符重载的形式

- 声明形式:

`istream & operator >> (istream &, 自定义类&);`
`ostream & operator << (ostream &, 自定义类 &);`

- 重载运算符 “<<”和 “>>”的函数的第一个参数和函数类型都必须是`*stream &`类型，第二个参数是要进行输入操作的类;
- 只能将重载 “<<”和 “>>”的函数作为友元函数或普通函数，而不能将它们定义为成员函数;
- 调用形式:
 - 隐式调用: `cout<<obj;`
 - 显示调用: `operator<<(cout, Class_obj)`

再讨论：函数原型

- 函数原型：

```
istream & operator >> (istream &is, 自定义类&) {  
    is>>要输入的内容;  
    return is;  
}
```

```
Ostream & operator << (ostream &os, 自定义类 &) {  
    os<<要输出的内容; //建议只进行最小限度格式化  
    return os;  
}
```

再讨论：值返回与引用返回

- 返回为何是引用？
 - 当需要将运算结果作为左值时，实现“连续操作”，用引用返回
`cin<<a<<b;`
- 参数为何是引用？
 - 用户可通过定义一个ostream的对象的引用s，则引用s也可以使用运算符“<<”。

重载流插入和流提取运算符



● 重载流插入运算符

目的：希望“<<”运算符不仅能输出标准数据类型，而且能输出用户自己定义的对象。我们以复数对象输出为例。

```
#include <iostream.h>
class complex
{public:
    complex( ){real = 0; image = 0;}
    complex (double r, double i){real = r; image = i;}
    complex operator + (complex &c2) const;
    friend ostream & operator << (ostream &output, const complex &c);
private:
    double real, image;
};

complex complex :: operator +(complex &c2) const
{ return complex (real+c2.real, image+c2.image); }

ostream& operator << (ostream& output, const complex& c)
{output<< "(" <<c.real<< "+"
    <<c.image<< "i)" <<endl;
    return output; }

void main( )
{ complex c1(2,4), c2(3,5), c3;
  c3 = c2+c1; //需要解读下编译器究竟如何实现?
  cout<<c1; cout<<c2; cout << c3;
}
```

```
D:\wwwp\c++\Debug\c1...
(2+4i)
(3+5i)
(5+9i)
Press any key to continue
```




重载流插入和流提取运算符

- 重载流提取运算符

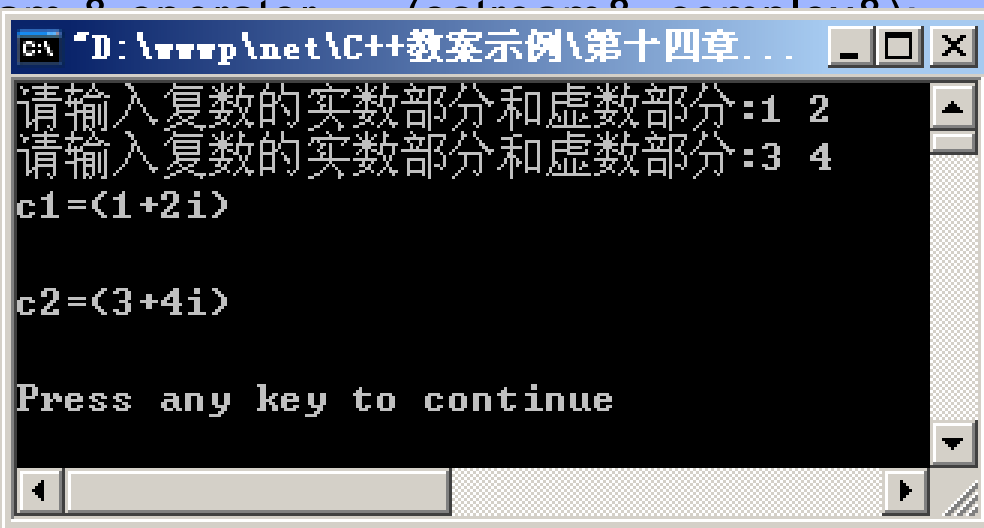
目的：希望“>>”运算符不仅能输入标准数据类型，而且能输入用户自己定义的类对象。我们仍以复数对象输入出为例。

```
#include <iostream.h>
class complex
{public:
    friend ostream& operator<< (ostream& out, const complex& c)
    friend istream& operator>> (istream& in, complex& c)
private:
    double real, image;
};

ostream& operator<< (ostream& out, const complex& c)
{out<< "("<<c.real<<"+i"<<c.image<<")";
return out;}

istream& operator>> (istream& in, complex& c)
{cout<< "请输入复数的实数部分和虚数部分:"<<endl;
in>>c.real>>c.image;
return in;}

void main( )
{ complex c1, c2;
  cin>>c1>>c2;
  cout<< "c1="<<c1<<endl;
  cout<< "c2="<<c2<<endl;
}
```

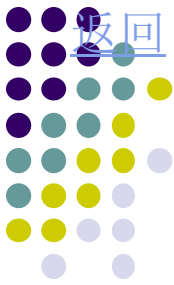


重载算术运算符

- “+”运算符，是一个二元运算符，且涉及到创建并返回新对象；
- 在重载算术运算符时，要考虑“+”对于对象的含义
 - 对于“盒子”类，“+”意味着什么？是各边长的简单累加，或是体积的累加，或是能容下多个的盒子？
- 例子

```
inline Box Box::operator+(const Box& aBox) const
{ return Box(length>aBox.length?length:aBox.length,
              breadth>aBox.breadth?breadth:aBox.breadth,
              height+aBox.height);
} // 定义为成员函数，且产生一个无名的对象，用值来初始化
```

```
inline Box operator+(const Box& aBox,const Box& bBox)
{
    // 函数体中，Box类对象的尺寸可通过公共的成员函数getdata等来访问
} // 定义为一般函数，且产生一个无名的对象，用值来初始化
```



+=运算符重载（1）

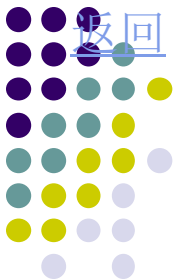
● 代码

```
Box& Box::operator+=(const Box& right)
{ length=length>right.length?length:right.length;
  breadth=breadth>right.breadth?breadth:right.breadth;
  height+=right.height;
  return *this; // 涉及赋值
}
```

```
Box b1,b2;
.....

b2+=b1;
```

- 函数功能:给左操作数***this**加上右操作数,修改了左操作数;
- 函数返回值:涉及赋值,需要返回一个引用;
- 函数主体中:使用在加号运算符中把**Box**对象加在一起的方法;



+=运算符重载（2）

- 可以使用一个运算符实现另外一个运算符。
- 例子：由+=实现+

```
Box Box::operator+(const Box& right) const
{
    return Box(*this)+=aBox;

} // Box(*this)调用拷贝构造函数
```

- 这一方法可应用于实现类的-=, *=等重载版本

重载单目运算符中的++和--

- ++和--只有一个操作数,是单目运算符;
- 对于++和--而言:
 - 问题: ++和--运算符有两种使用方式, 前置自增运算符和后置自增运算符, 作用是不一样的, 在重载时如何区分?
 - 解决: 在自增(自减)运算符重载函数中, 增加一个**int型虚拟形参**, 就是后置自增运算符函数

```
class Object
{ public:
    Object & operator++ (); //前置 ++a
    const Object operator++( int ) //后置 a++
    ... };
```

- 前缀形式的**返回类型**一般是当前对象递增后的引用;
- 后缀形式, 先在修改之前创建原对象的副本, 再将执行递增后的原对象返回。后缀运算符的返回值常声明为**const**

```
#include <iostream.h>    //成员函数
```

```
class Increase{
```

```
public:
```

```
    Increase(int x):value(x){}           //构造函数
```

```
    Increase & operator++();             //前增量
```

```
    Increase operator++(int);            //后增量
```

```
    void display(){
```

```
        cout <<"the value is " <<value <<endl; }
```

```
private:
```

```
    int value;
```

```
};
```

```
Increase & Increase::operator++() {
```

```
    value++;                             //先增量
```

```
    return *this;                        //再返回原对象
```

```
}
```

```
Increase Increase::operator++(int) {
```

```
    Increase temp(*this); //复制构造函数, 对象存放原有对象值
```

```
    value++;                //原有对象增量修改
```

```
    return temp;            //返回原有对象值
```

```
}
```





```
void main()  
{  
    Increase n(20);  
    n.display();  
    (n++) .display();  
    n.display();  
    ++n;  
    n.display();  
    ++(++n);  
    n.display();  
    (n++)++;  
    n.display();  
}
```

//显示临时对象值
//显示原有对象

//第二次增量操作对临时对象进行

此程序的运行结果为:

```
the value is 20  
the value is 20  
the value is 21  
the value is 22  
the value is 24  
the value is 25
```

```
#include <iostream.h>    //++也可以友元函数重载示例
```

```
class Increase{
```

```
public:
```

```
    Increase(int x):value(x){}
```

```
    friend Increase & operator++(Increase & );
```

//前增量，特别注意：由于要修改第一操作数，形参必须声明为引用

```
    friend      Increase      operator++(Increase      &,int) ;
```

```
        //后增量
```

```
    void display(){
```

```
        cout <<"the value is " <<value <<endl; }
```

```
private:      int value;
```

```
};
```

```
Increase & operator++(Increase & a) {
```

```
    a.value++;                                //前增量
```

```
    return a;                                //再返回原对象
```

```
}
```

```
Increase operator++(Increase& a, int) {
```

```
    Increase temp(a);    //通过拷贝构造函数保存原有对象值
```

```
    a.value++;            //原有对象增量修改
```

```
    return temp;          //返回原有对象值
```

```
}
```




```
void main() {  
    Increase n(20);  
    n.display();  
    (n++).display();  
    n.display();  
    ++n;  
    n.display();  
    ++(++n);  
    n.display();  
    (n++)++;  
    对象进行  
    n.display();  
}
```

此程序的运行结果为:

```
the value is 20  
the value is 20  
the value is 21  
the value is 22  
the value is 24  
the value is 25
```

//显示临时对象值
//显示原有对象

//第二次增量操作对临时



重载运算符()和[]



“()” “[]”不能用友元函数重载，只能采用成员函数重载。

1. 重载函数调用运算符()

对应的运算符重载函数为operator()(...).

设xobj为类X的一个对象，则表达式

xobj(arg1,arg2)

可被解释为：

xobj.operator()(arg1,arg2)

2. 重载下标运算符[]：掌握连续空间下，可判断下标是否超界

对应的运算符重载函数为operator[](...).

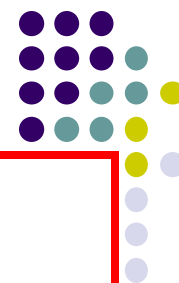
设xobj为类X的一个对象，则表达式

xobj[arg]

可被解释为：

xobj.operator[](arg)

- 例：将数组看成对象，要求数组的大小在定义时初始化，而且其大小在运算时改变，可以直接将对象当作数组使用。



扩充数组空间

```
#include<iostream.h>
```

```
class Array
```

```
{ int * p;
```

```
int size;
```

```
public:
```

```
Array(int num)
```

```
{ size=(num>6)?num:6;
```

```
p=new int[size];}
```

```
~Array( )
```

```
{ delete[] p;}
```

```
int & operator[ ](int idx) {
```

```
if(idx<size) return p[idx];
```

```
else{ expend(idx-size+1);
```

```
return p[idx];}
```

```
}//p[idx]非对象,[ ]是之前下标
```

```
void expend(int offset)
```

```
int * pi;
```

```
pi=new int [size+offset];
```

```
for(int num=0;num<size;num++)
```

```
pi[num]=p[num];
```

```
delete []p;
```

```
p=pi;
```

```
size=size+offset;
```

```
}//成员函数
```

```
void contract(int offset)
```

```
{ size=size-offest;
```

```
}
```

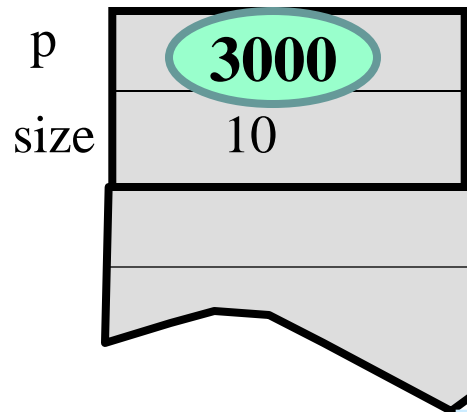
```
};
```

```

void main( )
{int num=0;
 Array a_Array(10);
 for(;num<10;num++)
  a_Array[num]=num;

 a_Array[10]=10;
 for(num=0;num<=10;num++)
  cout<<a_Array[num];
}

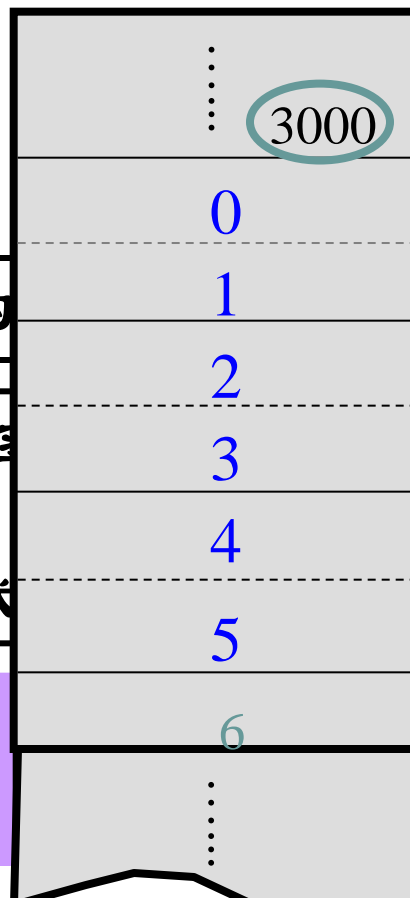
```



```

int & operator[ ](int idx) {
    if(idx<size) return p[idx];
    else{ expend(idx-size+1);
        return p[idx];}
} // p[idx]非对象,[]是之前下标

```



调用构造

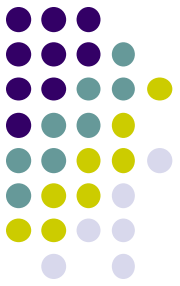
存

调用运算符重载函数
调用[]重载函数，
原有块不再需要其赋值
重新分配

输出：

0 1 2 3 4 5 6 7 8 9

000



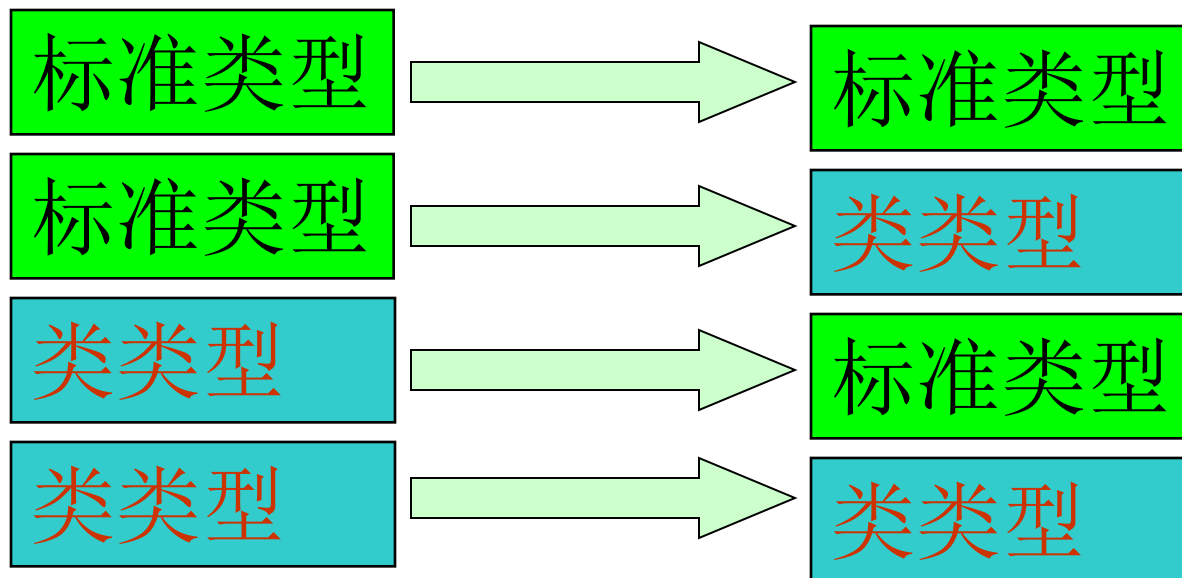
关于[]的思考:

- 与数组（字符数组）等关联；
- 用户可以自定义添加上新的功能，如：在进行下标访问时检查下标是否越界；
- 要求：针对上述功能，完善代码！

类型转换概述



- C++语言允许的类型转换有4种：

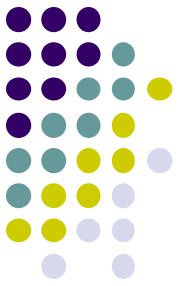


标准类型是除class, struct和union类型外的其他类型。



回顾：标准类型转换——隐式

- 对于标准类型，C++语言提供了两种类型转换：
 - 隐式类型转换
 - 显式类型转换
- 隐式转换发生在下述情况下：
 - 混合运算：级别低的向级别高的转换。
 - 将表达式的值赋给变量：表达式的值向变量类型的值转换。
 - 实参向形参传值：实参的值向形参的值进行转换。
 - 函数返回结果：返回的值向函数返回类型的值进行转换。



回顾：标准类型转换——显式

- 显式类型转换

- 1、强制法

- 例子： (类型名) 表达式 或者 (类型名)(表达式)

- 2、函数法

- 例子： 类型名(表达式)

- 它们都将表达式强制地转换为类型名所代表的类型的值。



已经探讨：标准类型转换为类类型

借助：特殊形式的构造函数

- 基础：用于类型转换的构造函数，即：
具有一个标准类型参数的构造函数说明了一种从参数类型到该类类型的转换。
- 类型是抽象的，因此类型的转换，实际是标准类型数据（常量或变量）向类对象的转换！
- 可以通过自定义的重载赋值号“=”的函数和构造函数实现转换：标准类型->类类型；它们都需要有标准类型的参数。

标准类型实参，转换为；
类类型形参

调用构造函数，生成
无名对象

标准类型转
换为类类型

```
void main( )  
{  
    ...  
}
```

```
class INTEGER {  
    int num; char str[10];  
public:  
    INTEGER(int);  
    INTEGER(const char *,int=0);  
    void mem_fun(INTEGER anint);  
};
```

```
void fun(INTEGER arg) {  
    INTEGER obj1=INTEGER(1);  
    INTEGER obj2="ChengDu";  
    int anint=10;  
    INTEGER obj3= INTEGER(anint);  
    obj1=20;  
    obj1.operator=(INTEGER(20));  
    obj2.mem_fun(3);  
    /*obj2.mem_fun(INTEGER(3));*/  
}
```

代码分析:



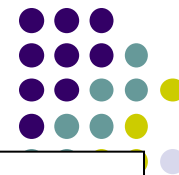
- `INTEGER obj1= INTEGER(1);`将1转换为类类型**INTEGER**
即: `INTEGER(1)`通过类型转换构造函数生成了一个临时无名空间
- `obj2="ChengDu";`, 编译尝试用构造函数**`X(const char*,int=0)`** 对赋值号右边的字符串进行类类型转换, 转换成功后, 赋给**INTEGER**的对象**obj2**。
- 语句**`obj2.mem_fun(3);`**中函数**`mem_fun`**由于需要一个**INTEGER**的对象作为参数, 故尝试用构造函数对实参进行转换, 转换成功后, 进行虚实参数匹配, 执行函数调用。
- 注意: 这样的转换是系统自动做的, 称为隐式类型转换。其中:
构造函数**`INTEGER(int)`**将整数类型转换为类类型**INTEGER**;
构造函数**`INTEGER(const char *,int=0)`**将字符串转换为类类型**INTEGER**。



类类型转换函数：类类型->基本类型

- 带一个参数的构造函数可以进行类型转换，但是它的转换功能很受限制。例如，想将一个类类型转换为基本类型就办不到。
- 类类型转换为基本类型：需要引入一种**特殊的成员函数：类型转换函数**，它在类对象之间提供一种类似显式类型转换的机制。

类类型转换函数



模型:

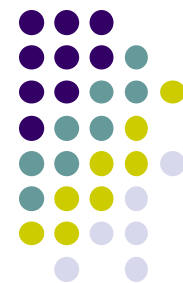
```
class_Name::operator type( )  
{  
    return(type 类型的实例);  
}
```

例子:

```
operator int( )  
{ return num;  
}
```

- 类型转换函数的功能是将class_Name类型的对象转换为类型为type的实例。type可以是一个预定义类型，也可以是一个用户定义的类型。
- 类型转换函数没有参数，没有返回类型，但这个函数体内必须有一条返回语句，返回一个type类型的实例。
- 类型转换函数不能被重载，因为它没有参数。
- 类型转换函数只能定义为一个类的成员函数，而不能定义为类的友元函数。

例：类类型转换为整型。



```
• #include "iostream.h"
• class INTEGER
• {
•     int num;
• public:
•     INTEGER(int anint=0)
•         { num=anint; }
•     operator int( )
•     { return num; }
• };
```

```
void main( )
{ INTEGER obj(12);
  int anint=int(obj);
  cout<<anint<<endl;
  anint=(int)obj; //显示
  cout<<anint<<endl;
  anint=obj; //隐式
  //anint=obj.operator int( );
```

输出：

12

12

12



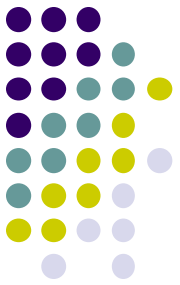
Ex4_25.cpp

`int anint=int(obj);`和`anint=(int)obj;`形式上看是用类型的强制类型转换规则.由于obj是一个类对象，它使用`int(obj)`或`(int)obj`的表达形式，实际执行`obj.operator int()`语句`anint=obj;`也用该类型转换函数进行。

```
int anint=obj;  
INTEGER obj=anint;
```

用类型转换函数进行转换

用构造函数进行转换



- 问题:如果一个类既有用于转换的构造函数, 又拥有类型转换函数,如:

- INTEGER(int);
- operator int();

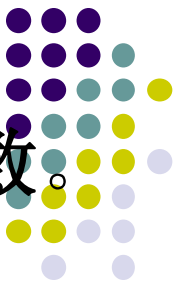
- 语句 **obj=obj+anint;** 该如何解释?

- ❖ 可以将表达式 **obj+anint** 解释为
 - **obj.operator int()+anint**
- ❖ 也可以这样解释为
 - **obj+INTEGER(anint)**

将obj转换得到一个整型数,与anint相加后得到一个整型结果,再用INTEGER(anint)将结果转换为X的类型,赋给obj。

将anint转换为一个INTEGER类型,与obj相加后再将结果赋给obj。

存在二义性, 编译不知道用那种方式进行解释。用户在这种情况下必须显示地使用类型转换函数。



- 用户在这种情况下必须显示地使用类型转换函数。
- 例如：

```
INTEGER obj1=anint;
```

```
obj=obj+obj1;           //两个对象加
```

或

```
obj=(int)obj+anint;      //两个整数加
```

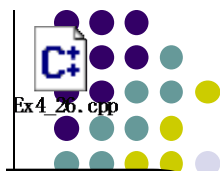
- 用户定义的类型转换函数只有在它们无二义性时才能隐式地使用。



总结

运算符	特殊要求	原因分析
赋值运算符=	使用成员函数	规避左值为常数可能
函数调用运算符（）	使用成员函数	
下标运算符[]	使用成员函数	
<< >>	友元函数或是一般函数	第一个参数不是类对象，是需要输入输出流的对象

例 4-26 类类型转换为类类型。定义一个类 `integer`，它可以处理32位的整数；同时定义另一个类 `real`，它可以处理32位的实数。



- `#include<iostream.h>`
- `class real;`
- `class integer`
- `{ long lval;`
- `public:`
- `integer()`
- `{lval=0;}`
- `integer(long d) {lval=d;}`
- `operator real();`
- `long & operator()()`
- `{ return lval; }`
- `friend integer operator+(integer num1,integer num2);`
- `integer operator-(integer num1);`
- `friend ostream & operator<<(ostream &s,integer num);};`

无参构造函数

有参构造函数

类型转换函数

() 重载函数

运算符"+"被重载为一个友元，而运算符 "-"被重载为一个成员函数，以说明它们对类型转换机制的影响。

+ 重载函数

输出重载函数



- class real
- { float rval;
- public:
- real(){rval=0;};
- real(float d){rval=d;};
- operator integer();
- float & operator()(){return rval;};
- friend real operator+(real num1,real num2);
- real operator-(real num1);
- friend ostream & operator<<(ostream &s,real num);
- };

- integer::operator real()
- { real num;
- num()=(float)lval;
- return num;
- }

在这个类型转换函数中，说明了一个具有real类型的自动对象num,num()返回num对象的内部数据成员rval的存储单元，语句num()= (float)lval;将integer类对象的内部数据lval通过标准类型转换为一个浮点类型，并将转换结果赋给num的内部数据rval,return语句将这个num对象返回，从而完成了一个类类型的转换。



- integer operator+(integer num1, integer num2)
- {
- integer num;
- num.lval=num1.lval+num2.lval;
- return num;
- }
- integer integer::operator-(integer num1)
- { integer num;
- num.lval=lval-num1.lval;
- return num; }
- ostream&operator<<(ostream &s, integer num)
- { return(s<<num.lval);
- }

- `real::operator integer()`
- `{ integer n;`
- `n()=(long)rval;`
- `return n; }`

类型转换函数将real类型转换为integer类型。这个类型转换函数，将具有real类型的对象转换为一个具有integer类型的对象。

```
real operator+(real num1,real num2)
{ real num;
  num.rval=num1.rval+num2.rval;
  return num;}
real real::operator-(real num1)
{ real num;
  num.rval=rval-num1.rval;
return num; }
ostream&operator<<(ostream &s,real num)
{ return(s<<num.rval);
}
```

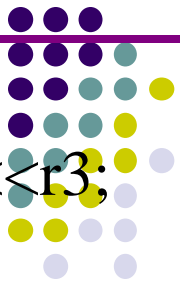
- `void main()`
- `{ integer i1;`
- `i1=50;`
- `cout<<"\n i1="<<i1;`
- `real r1=56.6; cout<<"\n`
`r1="<<r1;`
- `integer i2;`
- `i2=r1;`
- `//i2.operator=(r1.operator`
`integer())`
- `cout<<"\n i2=r1,i2="<<i2;`
- `real r2;`
- `r2=i1;`
- `cout<<"\n r2=r1,r2="<<r2;`
- `real r3;`

```

r3=real(i1)+r1;
cout<<"\n r3=real(i1)+r1,r3="<<r3;
r3=i1-r1;
cout<<"\n r3=i1-r1,r3="<<r3;
integer i3;

i3=i1+integer(r3);
cout<<"\n i3=i1+integer(r3),i3="<<i3;
i3=i1-r3;
cout<<"\n i3=i1-r3,i3="<<i3;
integer i4=24.5;
cout<<"\n i4=24.5,i4="<<i4;
real r4=40;
cout<<"\n r4=40,r4="<<r4;
cout<<"\n";}

```





- 该程序输出：
- $i1=50$
- $r1=56.6$
- $i2=r1, i2=56$
- $r2=r1, r2=50$
- $r3=\text{real}(i1)+r1, r3=106.6$
- $r3=i1-r1, r3=-6$
- $i3=i1+\text{integer}(r3), i3=44$
- $i3=i1-r3, i3=56$
- $i4=24.5, i4=24$
- $r4=40, r4=40$



- 这个测试程序，主要说明类型转换运算如何进行。

- 语句

- `r3=i1+r1;`

- 和语句 `i3=i1+r3;`

- 使"+"运算产生了二义性，编译器将不知道是使用

- `integer operator+(integer num1,integer num2)`

- 还是使用

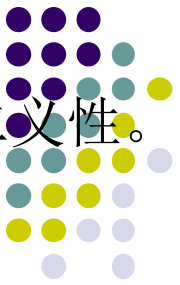
- `real operator+(real num1,real num2)`

- 或者说，编译器不知道在这两个语句中，应该使用转换运算符

- `integer::operator real()`

- 还是使用

- `real::operator integer()`



- 因此产生二义性，必须使用显式的类型转换方式来消除二义性。
- 例如：
 - `r3=real(i1)+r1;`
 - `i3=i1+(integer)r3;`
 - 语句
 - `i3=i1-r3;`
 - 因为"-"运算符被定义为一个成员函数，`i1-r3`只能被解释为
 - `i1.operator-(r3)`
 - 2所以不具有二义性。
 - 例 4-47 将上述例子改为友元类的情况。



- `#include<iostream.h>`

- `class real;`

- `class integer`

- `{`

- `long lval;`

- `friend real;`

- `public:`

- `integer()`

- `{lval=0;}`

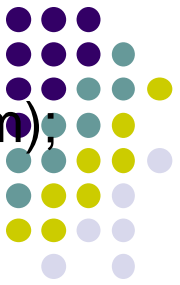
- `integer(long d)`

- `{lval=d;}`

- `operator real();`

- `friend integer operator+(integer num1,integer num2);`

- `integer operator-(integer num1);`



- friend ostream&operator<<(ostream &s,integer num);
- };
- class real
- {
- float rval;
- friend integer;
- public:
- real(){rval=0;};
- real(float d){rval=d;};
- operator integer();
- friend real operator+(real num1,real num2);
- real operator-(real num1);
- friend ostream&operator<<(ostream &s,real num);



```
};  
integer::operator real( )  
{  
    real num;  
    num.rval=(float)lval;  
    return num;  
}  
integer operator+(integer num1,integer num2)  
{  
    integer num;  
    num.lval=num1.lval+num2.lval;  
    return num;  
}  
ostream&operator<<(ostream &s,integer num);
```



```
{  
    return(s<<num.lval);  
}  
  
integer n;  
n.lval=(long)rval;  
return n;  
}  
  
real operator+(real num1,real num2)  
{  
    real num;  
    num.rval=num1.rval+num2.rval;  
    return num;  
}  
  
real real::operator-(real num1)
```



```
{  
    real num;  
    num,rval=rval-num1.rval;  
    return num;  
}  
  
ostream&operator<<(ostream &s,real num);  
{  
    return(s<<num.rval);  
}  
  
void main( )  
{  
    integer i1;  
    i1=50;  
    cout<<"\n i1="<<i1;
```



- `real r1=56.6;`
- `cout<<"\n r1="<<r1;`
- `integer i2;`
- `i2=r1;`
- `cout<<"\n i2=r1,i2="<<i2;`
- `real r3;`
- `r3=real(i1)+r1;`
- `cout<<"\n r3=real(i1)+r1,r3="<<r3;`
- `r3=i1-r1;`
- `cout<<"\n r3=i1-r1,r3"<<r3;`
- `integer i3;`
- `i3=i1+integer(r3);`
- `cout<<"\n i3=i1+integer(r3),i3="<<r3;`
- `i3=i1-r3;`



- `cout<<\n i3=i1-r3,i3=" "<<i3;`

- `integer i4=24.5;`

- `cout<<"\n i4=24.5,i4" "<<i4;`

- `real r4=40;`

- `cout<<"\n r4=40,r4=" "<<r4;`

- `cout<<"\n";`

- `}`

- 该程序的输出仍然为:

- `i1=50`

- `r1=56.6`

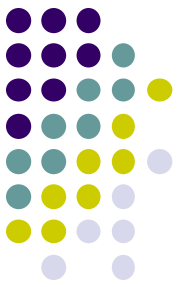
- `i2=r1,i2=56`

- `r2=r1,r2=50`

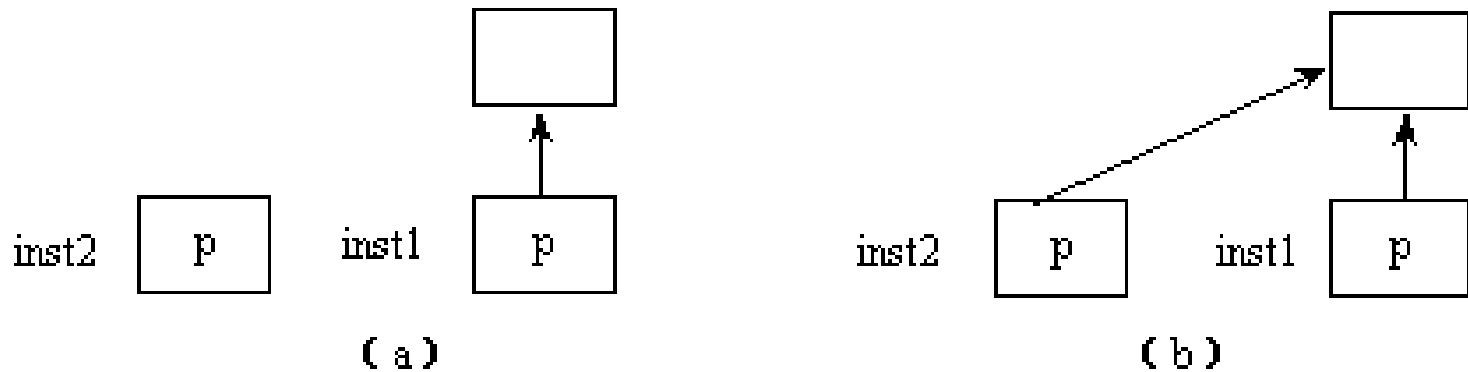
- `r3=real(i1)+r1,r3=106.6`



- $r3=i1-r1, r3=-6$
- $i3=i1+\text{integer}(r3), i3=44$
- $i3=i1-r3, i3=56$
- $i4=24.5, i4=24$
- $r4=40, r4=40$



自定义赋值运算符重载的必要性



对象内存分配

[返回](#)

考虑对复数的"加法"操作。

- `#include<iostream>`
- `class Complex {`
- `double re, im;`
- `public:`
- `Complex(double r,double i):re(r),im(i)`
- `{ }`
- `Complex() { re=0; im=0;}`
- `Complex add_Complex(Complex);`
- `void print()`
- `{cout<<re<<"+"<<im<<endl;}`
- `};`

- `Complex Complex::`
- `add_Complex(Complex`
- `cobj)`
- `{Complex temp;`
- `temp.re=re+cobj.re;`
- `temp.im=im+cobj.im;`
- `return temp;}`
- `void main()`
- `{Complex`
- `obj1(1,2),obj2(3,4);`
- `Complex obj3=`
- `obj1.add_Complex(obj2);`
- `obj3.print();}`

返回

- 对于2个复数相加，定义了**成员函数**`add_Complex(Complex)`实现加操作，使用**`obj1.add_Complex(obj2);`** 函数的调用形式,不太直观.
- 思考： 上述是通过成员函数来实现。如何通过友元来实现？
- 期待： `obj3=obj1+obj2;` //能实现+,必须通过运算符重载实现

```

• #include<iostream.h>
• class Complex
• { double re,im;
• public:
• Complex(double r,double
i)
• :re(r ),im(i)
• { }
• Complex( )
• { re=0;, im=0;}
• Complex
operator+(Complex);};
• Complex Complex ::
operator+(Complex cobj)
• { Complex temp;
• temp.re=re+cobj.re;
•
temp.im=im+cobj.im;
• return temp; }

```

```

• void main( )
• { Complex obj1(1,2),obj2(3,4);
• Complex obj3=obj1+obj2;
• // Complex obj3=obj1.operatr+(obj2)
• }

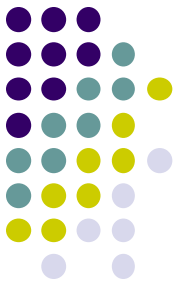
```

构造函数的重载，有
参数用表达式表的
方式初始化。

隐式
调用
方式

显示
调用
方式

“+”运算符重载
两个操作数，一个
是活此函数的对象，一个
是由函数参数提供



总结

运算符	特殊要求	原因分析
赋值运算符=	使用成员函数	规避左值为常数可能
函数调用运算符（）	使用成员函数	
下标运算符[]	使用成员函数	
<< >>	友元函数或是一般函数	第一个参数不是类对象，是需要输入输出流的对象

学习路线



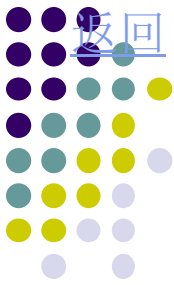
问题需求背景:

- 设计出类: **class Complex**
- 用户需求: 能使用运算符实现关于类对象的直接运算 **p=p1+p2;**

C++的机制支撑:

- 运算符重载机制
- 运算符的重载本质是通过函数来实现
operator + () 函数性质

程序员如何实现:



函数的返回值

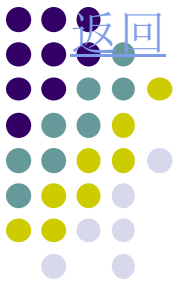
- 函数返回值语句
 return (返回值);
- 返回的值有哪些类型？
 - 基本类型
 - 构造类型：比如结构体类型、指针类型
 思考：数组如何返回？
- 返回值给谁？（谁来接收返回值？）
- 比较：
 - 一般函数返回值时，要生成一个值的副本。
 - 用引用返回值时，不生成值的副本。
- 关键的问题，什么时候使用返回值为引用？

函数返回时的情景分析

```
#include <iostream.h>
float temp;
float fn1(float r){
    temp = r*r*3.14;
    return temp;
}
```

```
float& fn2(float r){
    temp = r*r*3.14;
    return temp;
}
```

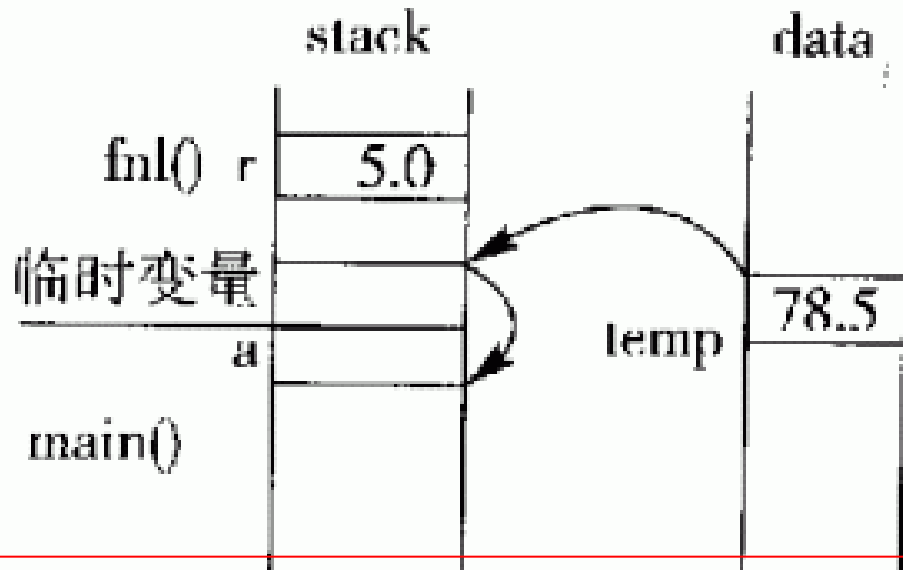
```
void main() {
    float a=fn1(5.0); //1
    float& b=fn1(5.0); //2:warning
    float c=fn2(5.0); //3
    float& d=fn2(5.0); //4
    cout<<a<<endl;
    cout<<b<<endl;
    cout<<c<<endl;
    cout<<d<<endl;
}
```



函数返回时的情景分析

```
#include <iostream.h>
float temp;
float fn1(float r){
    temp = r*r*3.14;
    return temp;
}
```

调用：float a=fn1(5.0);



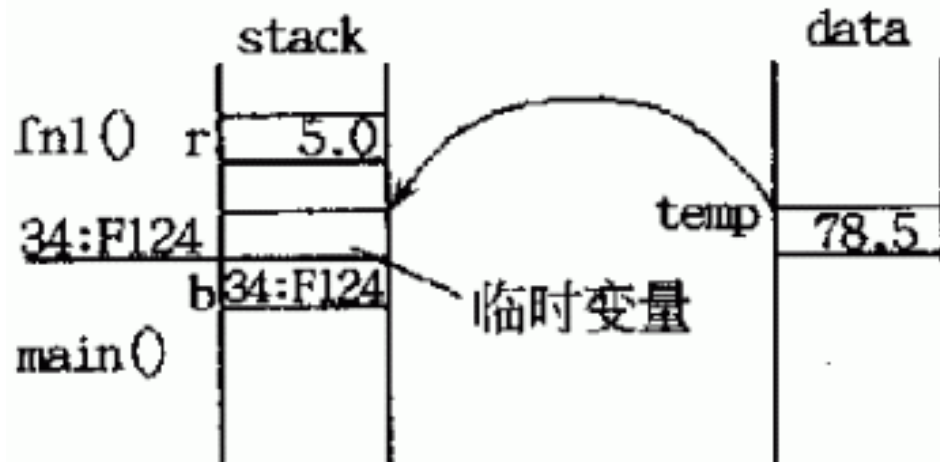
分析：

- 1其中变量**temp**是全局数据，驻留在全局数据区**data**。函数**main()**、函数**fn1()**或函数**fn2()**驻留在栈区**stack**。
- 2一般的函数返回值方式。返回全局变量**temp**值时，**C++**创建临时变量并将**temp**的值**78.5**复制给该临时变量。返回到主函数后，赋值语句**a=fn1(5.0)**把临时变量的值**78.5**复制给**a**。

函数返回时的情景分析

```
#include <iostream.h>
float temp;
float fn1(float r){
    temp = r*r*3.14;
    return temp;
}
```

调用: float& b=fn1(5.0);

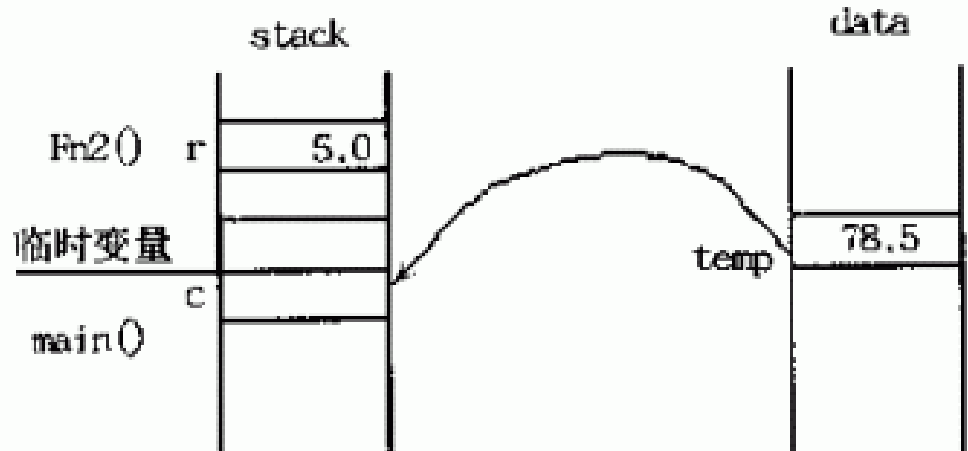


分析：函数`fn1()`是以值方式返回的，返回时，复制`temp`的值给临时变量。返回到主函数后，引用`b`以该临时变量来初始化，使得`b`成为该临时变量的别名。由于临时变量的作用域短暂，所以`b`面临无效的危险。根据C++标准，临时变量或对象的生命期在一个完整的语句表达式结束后便宣告结束，也即在“`float& b=fn1(5.0);`”之后，临时变量不再存在。所以引用`b`以后的值是个无法确定的值。

函数返回时的情景分析

```
#include <iostream.h>
float temp;
float& fn2(float r){
    temp = r*r*3.14;
    return temp;
}
```

调用： float c=fn2(5.0);

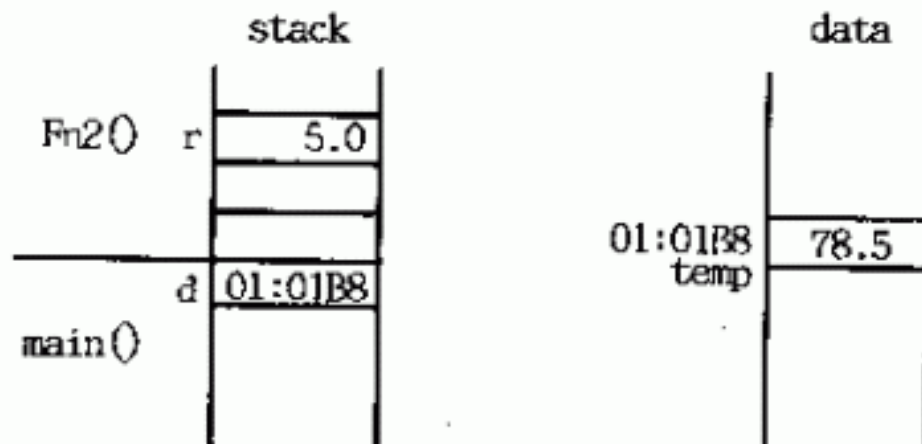


分析：函数`fn2()`的返回值不产生副本，所以，直接将变量`temp`返回给主函数。主函数的赋值语句中的左值，直接从变量`temp`中得到复制，这样避免了临时变量的产生。当变量`temp`是一个用户自定义的类型时，这种方式直接带来了程序执行效率和空间利用的利益。

函数的返回值的场景分析

```
#include <iostream.h>
float temp;
float& fn2(float r){
    temp = r*r*3.14;
    return temp;
}
```

调用: float& d=fn2(5.0);



分析：函数fn2()返回一个引用，因此不产生任何返回值的副本。在主函数中，一个引用声明d用该返回值来初始化，使得d成为temp的别名。由于temp是全局变量，所以在d的有效期内temp始终保持有效。