C++程序设计

主讲：王老师

尚德机构

学习是一种信仰

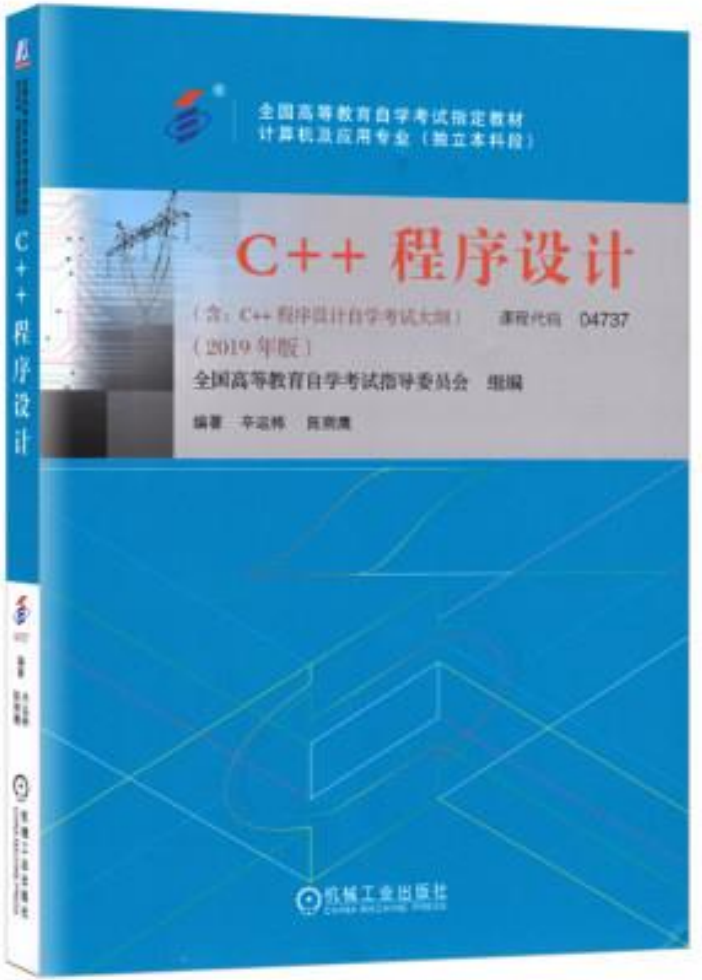
教材介绍

C++程序设计

（2019年版）

编著：辛运帷 陈朔鹰

机械工业出版社



考试题型

单选题　　1分×20题 = 20分

填空题　　1分×15题 = 15分

程序填空题　4分×5题 = 20分

程序分析题　6分×5题 = 30分

程序设计题　　2题 = 15分　(第一题5分，第二题10分)

第四章　运算符重载

本章主要内容

- 运算符重载的概念
- 重载赋值运算符
- 重载流插入运算符和流提取运算符
- *重载强制类型转换运算符
- 重载自增、自减运算符

4.1 运算符重载的概念

- C++中的表达式由运算符和操作数按照规则构成。例如，算术运算符包括加“+”、减“-”、乘“*”、除“/”和取模“%”。如果不做特殊处理，则这些算术运算符通常只能用于对基本数据类型的常量或变量进行运算，而不能用于对象之间的运算。
- 运算符重载，就是给已有的运算符赋予多重含义，使同一个运算符作用于不同类型的数据时产生不同的行为。运算符重载的目的是使得C++中的运算符也能够用来操作对象。
- 用于类运算的运算符通常都要重载。有两个运算符，系统提供了默认的重载版本：赋值运算符= 和 地址运算符&。
  - 对于=，系统默认重载为对象成员变量的复制。
  - 对于&，系统默认重载为返回任何类对象的地址。

4.1 运算符重载的概念

表4-1　可重载的运算符

双目算术运算符	**+(加)，-(减)，*(乘)，/(除)，%(取模)**
关系运算符	==(等于)，！=(不等于)，<(小于)，>(大于)，<=(小于等于)，>=(大于等于)
逻辑运算符	||(逻辑或)，&&(逻辑与)，!(逻辑非)
单目运算符	+(正)，-(负)，*(指针)，&(取地址)
自增自减运算符	++(自增)，--(自减)
位运算符	|(按位或)，&(按位与），~(按位取反)，^(按位异或)，<<(左移)，>>(右移)
赋值运算符	=(赋值)，+=(加法赋值)，-=(减法赋值)，*=(乘法赋值)，/=(除法赋值)，%=(取模赋值），&=(按位与赋值)，|=（按位或赋值)，^=(按位异或赋值），<<=(左移赋值），>>=(右移赋值)
空间申请与释放	new(创建对象），delete(释放对象），new[](创建数组），delete[](释放数组)
其他运算符	()(函数调用），->(成员访问），,(逗号)，[](下标)

4.1 运算符重载的概念

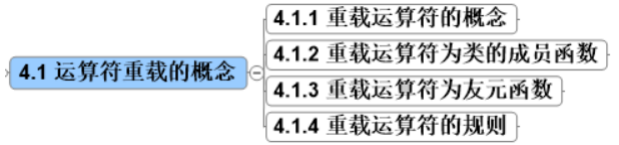


表4-2 不可重载的运算符和符号

成员访问运算符	.
成员指针访问运算符	.*, ->*
域运算符	::
长度运算符	sizeof
条件运算符	?:
预处理符号	#

4.1 运算符重载的概念

- 与其他函数一样，重载运算符有一个返回类型和一个参数列表。这样的函数称为**运算符函数**。
- 运算符可以被重载为全局函数，也可以被重载为类的成员函数。声明为全局函数时，通常应是**类的友元**。
- 故运算符函数是一种特殊的**友元函数或成员函数**。

课堂练习

逻辑运算符两侧运算对象的数据（　）。

A:是逻辑型数据

B:只能是整型数据

C:只能是整型或字符型数据

D:可以是任何类型的数据

课堂练习

逻辑运算符两侧运算对象的数据（　）。

A:是逻辑型数据

B:只能是整型数据

C:只能是整型或字符型数据

D:可以是任何类型的数据

答案：D

在逻辑运算中，0为假，非0为真，一般用数字1表示。

课堂练习

设x和y均为bool量，则x&&y为真的条件是（　）。

A:它们均为真

B:其中一个为真

C:它们均为假

D:其中一个为假

课堂练习

设x和y均为bool量，则x&&y为真的条件是（　）。

A:它们均为真

B:其中一个为真

C:它们均为假

D:其中一个为假

答案：A

课堂练习

若要对Data类中重载的加法运算符成员函数进行声明，下列选项中正确的是（　）。

A:Data +(Data);

B:Data operator+(Data);

C:Data +operator(Data);

D:operator+(Data,Data);

课堂练习

若要对Data类中重载的加法运算符成员函数进行声明，下列选项中正确的是（　）。

A:Data +(Data);

B:Data operator+(Data);

C:Data +operator(Data);

D:operator+(Data,Data);

答案：B

课堂练习

以下程序段中与语句k=a>b? 1:0;功能等价的是（　　）。

A: if(a>b) k=1; else k=0;

B: if(a>b) k=0;

C: if(a<b) k=0; else k=1;

D: if(a>b) k=1;

课堂练习

以下程序段中与语句k=a>b? 1:0;功能等价的是（　）。

A: if(a>b) k=1; else k=0;

B: if(a>b) k=0;

C: if(a<b) k=0; else k=1;

D: if(a>b) k=1;

答案：A

课堂练习

预处理命令在程序中开头的符号是（　）。

A:*

B:#

C:&

D:@

课堂练习

预处理命令在程序中开头的符号是（　）。

A:*

B:#

C:&

D:@

答案：B

课堂练习

下列运算符不能重载的是（　）。

A:!

B:sizeof

C:new

D:delete

课堂练习

下列运算符不能重载的是（　）。

A:!

B:sizeof

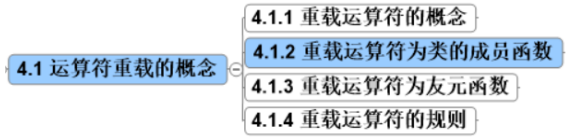
C:new

D:delete

答案：B

成员访问运算符	.
成员指针访问运算符	.*, ->*
域运算符	::
长度运算符	sizeof
条件运算符	?:
预处理符号	#

4.1.2 重载运算符为类的成员函数



见下页例题

```
#include<iostream>
using namespace std;
class myComplex  //复数类
{private:
      double real,imag;
public:
      myComplex();                          //构造函数
      myComplex(double r,double i);   //构造函数
      void outCom();                        //成员函数
      myComplex operator-(const myComplex &c);  //成员函数
      friend myComplex operator+(const myComplex &c1,const myComplex &c2);
};
myComplex::myComplex()
{     real=0;
      imag=0;}
myComplex::myComplex(double r,double i)
{     real=r;
      imag=i;}
void myComplex::outCom()
{     cout<<"("<<real<<","<<imag<<")"; }
myComplex myComplex::operator - (const myComplex &c)
{     return myComplex(this->real - c.real,this->imag - c.imag);}
myComplex operator+(const myComplex &c1,const myComplex &c2)
{     return myComplex(c1.real+c2.real,c1.imag+c2.imag);}
```

```
int main()
{
      myComplex c1(1,2),c2(3,4),res;
      c1.outCom();
      cout<<"operator+";
      c2.outCom();
      cout<<"=";
      res=c1+c2;
      res.outCom();
      cout<<endl;
      c1.outCom();
      cout<<"operator-";
      c2.outCom();
      cout<<"=";
      res=c1-c2;
      res.outCom();
      cout<<endl;
      return 0;
}
```

```
(1,2)operator+(3,4)=(4,6)
(1,2)operator-(3,4)=(-2,-2)
```

4.1.3 重载运算符为友元函数


```
class myComplex                                    //复数类
{
private:
        double real, imag;
public:
        ...
        myComplex operator-(const myComplex &c);
        myComplex operator-(double r);
};
...
myComplex myComplex::operator-(const myComplex &c)
{       return myComplex(this->real - c.real, this->imag - c.imag); //返回一个临时对象
}
```

```
#include<iostream.h>
class myComplex  //复数类
{
private:
    double real,imag;
public:
    myComplex();                              //构造函数
    myComplex(double r,double i);     //构造函数
    void outCom();                           //成员函数
    friend myComplex operator+(const myComplex &c1,const myComplex &c2);
    friend myComplex operator-(const myComplex &c1,const myComplex &c2);//
    friend myComplex operator-(const myComplex &c1,double r);//
    friend myComplex operator-(double r,const myComplex &c1);//
};
myComplex::myComplex()
{
    real=0;
    imag=0;
}
myComplex::myComplex(double r,double i)
{
    real=r;
    imag=i;
}
```

```
void myComplex::outCom()
{
    cout<<"("<<real<<","<<imag<<")";
}

myComplex operator+(const myComplex &c1,const myComplex &c2)  //c1+c2
{
    return myComplex(c1.real+c2.real,c1.imag+c2.imag);       //返回一个临时对象
}

myComplex operator-(const myComplex &c1,const myComplex &c2)  //c1-c2 新增
{
    return myComplex(c1.real-c2.real,c1.imag-c2.imag);       //返回一个临时对象
}

myComplex operator-(const myComplex &c1,double r)  //c1-r 新增
{
    return myComplex(c1.real-r,c1.imag);       //返回一个临时对象
}

myComplex operator-(double r,const myComplex &c1)  //r-c1 新增
{
    return myComplex(r-c1.real,-c1.imag);       //返回一个临时对象
}
```

```
int main()
{
   myComplex c1(1,2),c2(3,4),res;
   c1.outCom();
   cout<<"operator+";
   c2.outCom();
   cout<<"=";
   res=c1+c2;
   res.outCom();
   cout<<endl;
   res=c1-5;
   res.outCom();
   res=5-c1;
   res.outCom();
  return 0;
 }
```

```
(1,2)operator+(3,4)=(4,6)
(-4,2)(4,-2)
```

课堂练习

友元运算符@obj被C++编译器解释为（　　）。

A:operator@(obj)

B:operator@(obj,0)

C:obj.operator@()

D:obj.operator@(0)

课堂练习

友元运算符@obj被C++编译器解释为（　　）。

A:operator@(obj)

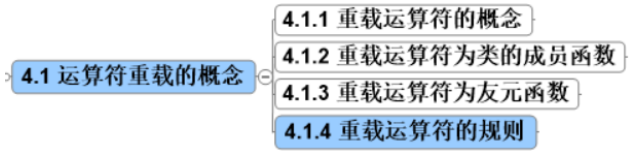
B:operator@(obj,0)

C:obj.operator@()

D:obj.operator@(0)

答案：A

4.1.4 重载运算符的规则



1)重载后运算符的含义应该符合原有的用法习惯。例如，重载“+”运算符，完成的功能就应该类似于做加法，在重载的“+”运算符中做减法是不合适的。

2)运算符重载**不能改变**运算符原有的语义，包括运算符的优先级和结合性。

3)运算符重载**不能改变**运算符操作数的个数及语法结构。

4)不能创建新的运算符，即重载运算符不能超出C++语言允许重载的运算符范围。

5)**重载运算符“( )”“[ ]”“->”或者赋值运算符“=”时，只能将它们重载为成员函数**，不能重载为全局函数。

6)运算符重载不能改变该运算符用于基本数据类型对象的含义。

课堂练习

下列关于运算符重载的表述中，正确的是（　）。

A:C++已有的任何运算符都可以重载

B:运算符函数的返回类型不能声明为基本数据类型

C:在类型转换符函数的定义中不需要声明返回类型

D:可以通过运算符重载来创建C++中原来没有的运算符

课堂练习

下列关于运算符重载的表述中，正确的是（　）。

A:C++已有的任何运算符都可以重载

B:运算符函数的返回类型不能声明为基本数据类型

C:在类型转换符函数的定义中不需要声明返回类型

D:可以通过运算符重载来创建C++中原来没有的运算符

答案：C

课堂练习

下列关于运算符重载的叙述，正确的是（　　）。

A:通过运算符重载，可以定义新的运算符

B:有的运算符只能作为成员函数重载

C:若重载运算符+，则相应的运算符函数名是+

D:重载一个二元运算符时，必须声明两个形参

课堂练习

下列关于运算符重载的叙述，正确的是（　　）。

A:通过运算符重载，可以定义新的运算符

B:有的运算符只能作为成员函数重载

C:若重载运算符+，则相应的运算符函数名是+

D:重载一个二元运算符时，必须声明两个形参

答案：B

课堂练习

如果表达式++a中的“++”是作为成员函数重载的运算符，若采用运算符函数调用格式，则可表示为（　）

A:a.operator++(1)

B:operator++(a)

C:operator++(a,1)

D:a.operator++( )

课堂练习

如果表达式++a中的“++”是作为成员函数重载的运算符，若采用运算符函数调用格式，则可表示为（　）

A:a.operator++(1)

B:operator++(a)

C:operator++(a,1)

D:a.operator++( )

答案：D

课堂练习

设有类A的对象Aobject，若用成员函数重载前置自增运算符，则++Aobject被编译器解释为（　）

A:Aobject.operator++( )

B:operator++(Aobject)

C:++(Aobject)

D:Aobject::operator++( )

课堂练习

设有类A的对象Aobject，若用成员函数重载前置自增运算符，则++Aobject被编译器解释为（　）

A:Aobject.operator++( )

B:operator++(Aobject)

C:++(Aobject)

D:Aobject::operator++( )

答案：A

课堂练习

以下关于运算符重载的描述中，错误的是（　）。

A:运算符重载其实就是函数重载

B:成员运算符比友元运算符少一个参数

C:需要使用关键字operator

D:成员运算符比友元运算符多一个参数

课堂练习

以下关于运算符重载的描述中，错误的是（　）。

A:运算符重载其实就是函数重载

B:成员运算符比友元运算符少一个参数

C:需要使用关键字operator

D:成员运算符比友元运算符多一个参数

答案：D

4.2 重载赋值运算符

C++中的赋值运算符“=”要求左右两个操作数的类型是匹配的，或至少是赋值兼容的。有时希望“=”两边的操作数的类型即使不赋值兼容也能够成立，这就需要对“=”进行重载。C++规定，“=”只能重载为成员函数。

若有类CL的两个对象s1和s2，则它们之间的赋值语句通常是下面这样的形式：

```
s1=s2;
```

当类CL中定义了成员函数，重载了赋值运算符后，上述赋值语句将解释为函数调用的形式：

```
s1.operator=(s2);
```

4.2.1 重载赋值运算符

```
myComplex operator+(const myComplex & c1, const myComplex & c2)
{
	return myComplex(c1.real + c2.real, c1.imag + c2.imag);
}
myComplex operator+(const myComplex& c1, double r)
{
	return myComplex(c1.real + r, c1.imag);
}
myComplex operator+(double r, const myComplex& c1)
{
	return myComplex(r + c1.real, c1.imag);
}
```

4.2.1 重载赋值运算符

```
myComplex operator-(const myComplex& c1, const myComplex& c2)
{
    return myComplex(c1.real - c2.real, c1.imag - c2.imag);
}
myComplex operator-(const myComplex& c1, double r)
{
    return myComplex(c1.real - r, c1.imag);
}
myComplex operator-(double r, const myComplex& c1)
{
    return myComplex(r - c1.real, -c1.imag);
}
```

4.2.1 重载赋值运算符

```
myComplex& myComplex::operator=(const myComplex& c1)
{
    this->real = c1.real;
    this->imag = c1.imag;
    return *this;
}
myComplex& myComplex::operator=(double r)
{
    this->real = r;
    this->imag = 0;
    return *this;
}
```

赋值运算符必须重载为成员函数。

```
int main()
{   myComplex c1(1,2),c2(3,4),res;
    c1.outCom("\t\t\tC1"); c2.outCom("\t\t\tC2");
    res = c1 + c2;
    res.outCom("执行res=c1+c2→\tres");
    res = c1.addCom(c2);
    res.outCom("执行res=c1.addCom(c2)→\tres");
    res = c1 + 5;
    res.outCom("执行res=c1+5→\tres");
    res = 5 + c1;
    res.outCom("执行res=5+c1→\tres");
    res = c1;
    c1.outCom("\t\t\tC1");
    res.outCom("执行res=c1→\t\tres");
    c1.changeReal(-3);
    c1.outCom("执行c1.changeReal(-3)→\tc1");
    res.outCom("\t\t\tres");
    res = c1;
    res.outCom("执行res=c1→\t\tres");
    res = 7;
    res.outCom("执行res=7→\t\tres");
    res = 7 + 8;
    res.outCom("执行res=(7+8)→\tres");
    res = c1 = c2;
    c1.outCom("\t\t\tc1");
    c2.outCom("\t\t\tc2");
    res.outCom("执行res=c1=c2→\tres");
    return 0;}
```

4.2.2 浅拷贝和深拷贝

同类对象之间可以通过赋值运算符“=”互相赋值。如果没有经过重载，“=”的作用就是将赋值号右侧对象的值一一赋值给左侧的对象。这相当于值的拷贝，称为“**浅拷贝**”。

重载赋值运算符后，赋值语句的功能是将一个对象中指针成员变量指向的内容复制到另一个对象中指针成员变量指向的地方，这样的拷贝叫“**深拷贝**”。

```
#include<iostream>
using namespace std;
class pointer
{
public:
    int a;
    int *p;
    pointer( )
    {
        a=100;
        p=new int(10);
    }
    pointer(const pointer &tempp)
    {
        if(this !=&tempp)
        {
            a=tempp.a;
            p=tempp.p;
        }
    }
};
```

```
int main()
{
        pointer p1;
        pointer p2(p1);
        pointer p3=p1;
        cout<<"\n初始化后,各对象的值及内存地址"<<endl;
        cout<<"对象名\t对象地址   a的值   p中的值    p指向的值    p的地址"<<endl;
       cout<<"p1:\t"<<&p1<<",  "<<p1.a<<",    "<<p1.p<<",    "<<*p1.p<<",   "<<&p1.p<<endl;
       cout<<"p2:\t"<<&p2<<",  "<<p2.a<<",    "<<p2.p<<",    "<<*p2.p<<",   "<<&p2.p<<endl;
       cout<<"p3:\t"<<&p3<<",  "<<p3.a<<",    "<<p3.p<<",    "<<*p3.p<<",   "<<&p3.p<<endl;
       *p1.p=20;
       p2.a=300;
       cout<<"\n修改后，各对象的值及内存地址"<<endl;
       cout<<"对象名\t对象地址  a的值   p中的值   p指向的值   p的地址"<<endl;
       cout<<"p1:\t"<<&p1<<",  "<<p1.a<<",    "<<p1.p<<",    "<<*p1.p<<",   "<<&p1.p<<endl;
       cout<<"p2:\t"<<&p2<<",  "<<p2.a<<",    "<<p2.p<<",    "<<*p2.p<<",   "<<&p2.p<<endl;
       cout<<"p3:\t"<<&p3<<",  "<<p3.a<<",    "<<p3.p<<",    "<<*p3.p<<",   "<<&p3.p<<endl;
        return 0;
}
```

```
初始化后,各对象的值及内存地址
对象名   对象地址    a的值    p中的值     p指向的值     p的地址
p1:      0019FF28,   100,     007D1630,       10,     0019FF2C
p2:      0019FF20,   100,     007D1630,       10,     0019FF24
p3:      0019FF18,   100,     007D1630,       10,     0019FF1C

修改后，各对象的值及内存地址
对象名   对象地址   a的值    p中的值    p指向的值    p的地址
p1:      0019FF28,   100,     007D1630,       20,     0019FF2C
p2:      0019FF20,   300,     007D1630,       20,     0019FF24
p3:      0019FF18,   100,     007D1630,       20,     0019FF1C
```

4.3 重载流插入运算符和流提取运算符

在C++中，左移运算符“<<”可以和cout一起用于输出，故常被称为“流插入运算符”。右移运算符“>>”和cin一起用于输入，一般被称为流提取运算符。它们都是C++类库中提供的。在类库提供的头文件中已经对“<<”和“>>”进行了重载，使之分别作为流插入运算符和流提取运算符，能用来输出和输入C++基本数据类型的数据。cout是ostream类的对象，cin是istream类的对象，它们都是在头文件iostream中声明的。因此，凡是用“cout<<”和“cin>>”对基本数据类型数据进行输出/输入的，都要用#include指令把头文件iostream包含到本程序文件中。

必须重载为类的友元

使用友元函数重载运算符<<和>>

```
#include <iostream.h>
class test
{
private:
      int i;
      float f;
      char ch;
 public:
      test(int a=0,float b=0,char c='\0') {i=a;f=b;ch=c;}
      friend ostream &operator<<(ostream &,test);
      friend istream &operator>>(istream &,test &);
};

ostream &operator<<(ostream & stream,test obj)
{
      stream<<obj.i<<",";  //stream是cout的别名
      stream<<obj.f<<",";
      stream<<obj.ch<<endl;
      return stream;
}
```

```
istream &operator>>(istream & t_stream,test&obj)
{
      t_stream>>obj.i;    //t_stream是cin的别名
      t_stream>>obj.f;
      t_stream>>obj.ch;
      return t_stream;
}
void main( )
{
      test A(45,8.5,'W');
      operator<<(cout,A);
      test B,C;
      cout<<"Input as i f ch:";
      operator>>(cin,B);  operator>>(cin,C);
      operator<<(cout,B);  operator<<(cout,C);
}
```

```
45,8.5,W
Input as i f ch:5 5.8 A 2 3.4 a
5,5.8,A
2,3.4,a
```

4.3 重载流插入运算符和流提取运算符

```
#include <iostream>
#include <string>
#include <cstdlib>
using namespace std;
class myComplex                                          //复数类
{
private:
    double real, imag;
public:
    myComplex() : real(0), imag(0){}
    myComplex( double r, double i ) : real(r), imag(i){}
    friend ostream & operator<<(ostream & os, const myComplex & c);    //友元，插入
    friend istream & operator>>(istream & is, myComplex & c);          //友元，提取
};
ostream & operator <<(ostream & os, const myComplex & c)
{   if ( c.imag >= 0 )
        os<<c.real<<"+"<<c.imag<<"i";          //以 a+bi 的形式输出
    else
        os<<c.real<<"-"<<(-c.imag)<<"i";
    return os;
}
```

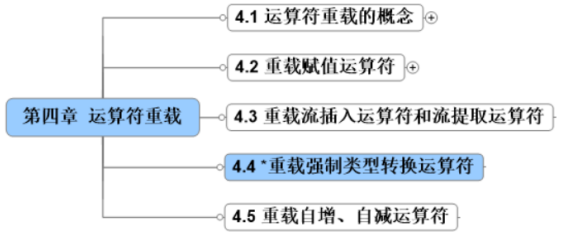
```
istream & operator >>(istream & is, myComplex & c)
{   string s;
    is>>s;      //将 a+bi 作为字符串读入，a+bi 中间不能有空格
    int pos = s.find("+", 0);                          //查找虚部
    if ( pos==-1 ) pos = s.find("-", 1);               //虚部为负数时
    string sReal = s.substr(0, pos);                   //分离出代表实部的字符串
    c.real = atof(sReal.c_str());                      //atof()能将参数内容转换成浮点数
    sReal = s.substr(pos, s.length()- pos - 1);        //分离出代表虚部的字符串
    c.imag = atof(sReal.c_str());
    return is;
}
```

```
int main()
{   myComplex c, c1;
    int n;
    cout<<"请输入两个复数([-]a±bi)和一个整数，以空格分隔"<<endl;
    cin>>c>>c1>>n;
    cout<<c<<", "<<n<<", "<<c1;
    return 0;
}
```

程序执行时的效果如下所示：

```
请输入两个复数([-]a±bi)和一个整数，以空格分隔
3.4-3i -5+6i 36↙
3.4-3i, 36, -5+6i
```

4.4 *重载强制类型转换运算符



在C++中，类型的名字（包括类的名字）本身也是一种运算符，即强制类型转换运算符。强制类型转换运算符是单目运算符，也可以被重载，但只能重载为成员函数，不能重载为全局函数。经过适当重载后，“(类型名) 对象”这个对对象进行强制类型转换的表达式就等价于“对象.operator 类型名( )”，即变成对运算符函数的调用。

4.4 *重载强制类型转换运算符

【程序 4-7】 重载强制类型转换运算符 double

```
#include <iostream>
using namespace std;
class myComplex
{
    double real, imag;
public:
    myComplex(double r = 0, double i = 0) : real(r), imag(i){};
    operator double()                    //重载强制类型转换运算符 double
    {   return real;
    }
};
int main()
{   myComplex c(1.2, -3.4);
    cout<<(double)c<<endl;               //输出 1.2
    double n = 12 + c;                   //等价于 double n = 12 + c.operator double()
    cout<<n;
    return 0;
}
```

程序 4-7 的输出结果如下：

```
1.2
13.2
```

4.5 重载自增、自减运算符

自增运算符“++”和自减运算符“--”都可以被重载，但是它们有前置和后置之分。以“++”为例，对于整数k，“++k”和“k++”的语义是不一样的。当“++”用于对象时，也应该如此。例如，obj是一个类CDemo的对象，那么“++obj”和“obj++”的含义应该是不一样的。按照自增运算符及自减运算符的本来定义，“++obj”的返回值应该是obj被修改后的值，而“obj++”的返回值应该是obj被修改前的值。

4.5第五节 重载自增、自减运算符

```
#include<iostream>
using namespace std;
class CDemo
{
private:
        int n;
public:
        CDemo(int i=0):n(i){}
        CDemo & operator++();     //用于前置形式
        CDemo operator++(int);    //用于后置形式
        operator int(){return n;}
        friend CDemo & operator--(CDemo &);
        friend CDemo operator--(CDemo &,int);
};
CDemo & CDemo::operator++()   //前置++
{       n++;
        return *this;
}
CDemo CDemo::operator++(int k) //后置++，多一个参数
{
        CDemo tmp(*this);         //记录修改前的对象
        n++;
        return tmp;                    //返回修改前的对象
}
```

```
CDemo & operator--(CDemo & d) //前置--
{       d.n--;
        return d;
}

CDemo operator--(CDemo & d,int)//后置--
{       CDemo tmp(d);
        d.n--;
        return tmp;
}
int main()
{
        CDemo d(10);
        cout<<(d++)<<","; //等价于d.operator++（0）；输出10
        cout<<d<<“,”;        //输出11
        cout <<(++d)<<","; //等价于d.operator++（）；输出12
        cout<<d<<",";         //输出12
        cout<<(d--)<<“,”;    //等价于operator--（d,0）；输出12
        cout<<d<<“,”;         //输出11
        cout<<(--d)<<“,”;    //等价于operator--（d）；输出10
        cout<<d<<endl;      //输出10
        return 0;
}
```

10,11,12,12,12,11,10,10

```
#include<iostream>
using namespace std;
class CDemo
{private:
    int n;
public:
    CDemo(int i=0):n(i){}
    CDemo & operator++();     //用于前置形式
    CDemo  operator++(int);   //用于后置形式
    operator int()
    {return n;}
    CDemo & operator--();
    CDemo operator--(int);
};
CDemo & CDemo::operator++()   //前置++
{   n++;
    return *this;
}
CDemo CDemo::operator++(int k) //后置++，多一个参数
{
    CDemo tmp(*this);         //记录修改前的对象
    n++;
    return tmp;               //返回修改前的对象
}
CDemo & CDemo::operator--()    //前置--
{   n--;
    return *this;
}
CDemo CDemo::operator--(int k)//后置--
{   CDemo tmp(*this);
    n--;
    return tmp;
}
int main()
{
    CDemo d(10);
    cout<<(d++)<<",";  //等价于“d.operator++（0）；”输出10
    cout<<d<<",";      //输出11
    cout <<(++d)<<","; //等价于“d.operator++（）；”输出12
    cout<<d<<",";      //输出12
    cout<<(d--)<<",";  //等价于“d. operator--（0）；”输出12
    cout<<d<<",";      //输出11
    cout<<(--d)<<",";  //等价于d. operator--（）；输出10
    cout<<d<<endl;     //输出10
    return 0;
}
```

10,11,12,12,12,11,10,10

真题演练

函数int & min(int &,int &)返回参数中较小者，设有两整型变量int a=10;int b=15；在执行语句min(a,b)--;之后，a，b值分别为（ ）。

A:9，14

B:9，15

C:10，14

D:10，15

真题演练

函数int & min(int &,int &)返回参数中较小者，设有两整型变量int a=10;int b=15；在执行语句min(a,b)--;之后，a，b值分别为（ ）。

A:9，14

B:9，15

C:10，14

D:10，15

答案：B

真题演练

如果表达式++a中的“++”是作为成员函数重载的运算符，若采用运算符函数调用格式，则可表示为（　）。

A:a.operator++(1)

B:operator++(a)

C:operator++(a,1)

D:a.operator++( )

真题演练

如果表达式++a中的“++”是作为成员函数重载的运算符，若采用运算符函数调用格式，则可表示为（　）。

A:a.operator++(1)

B:operator++(a)

C:operator++(a,1)

D:a.operator++( )

答案：D

真题演练

设有类A的对象Aobject，若用成员函数重载前置自增运算符，则++Aobject被编译器解释为（　）。

A:Aobject.operator++( )

B:operator++(Aobject)

C:++(Aobject)

D:Aobject::operator++( )

真题演练

设有类A的对象Aobject，若用成员函数重载前置自增运算符，则++Aobject被编译器解释为（　）。

A:Aobject.operator++()

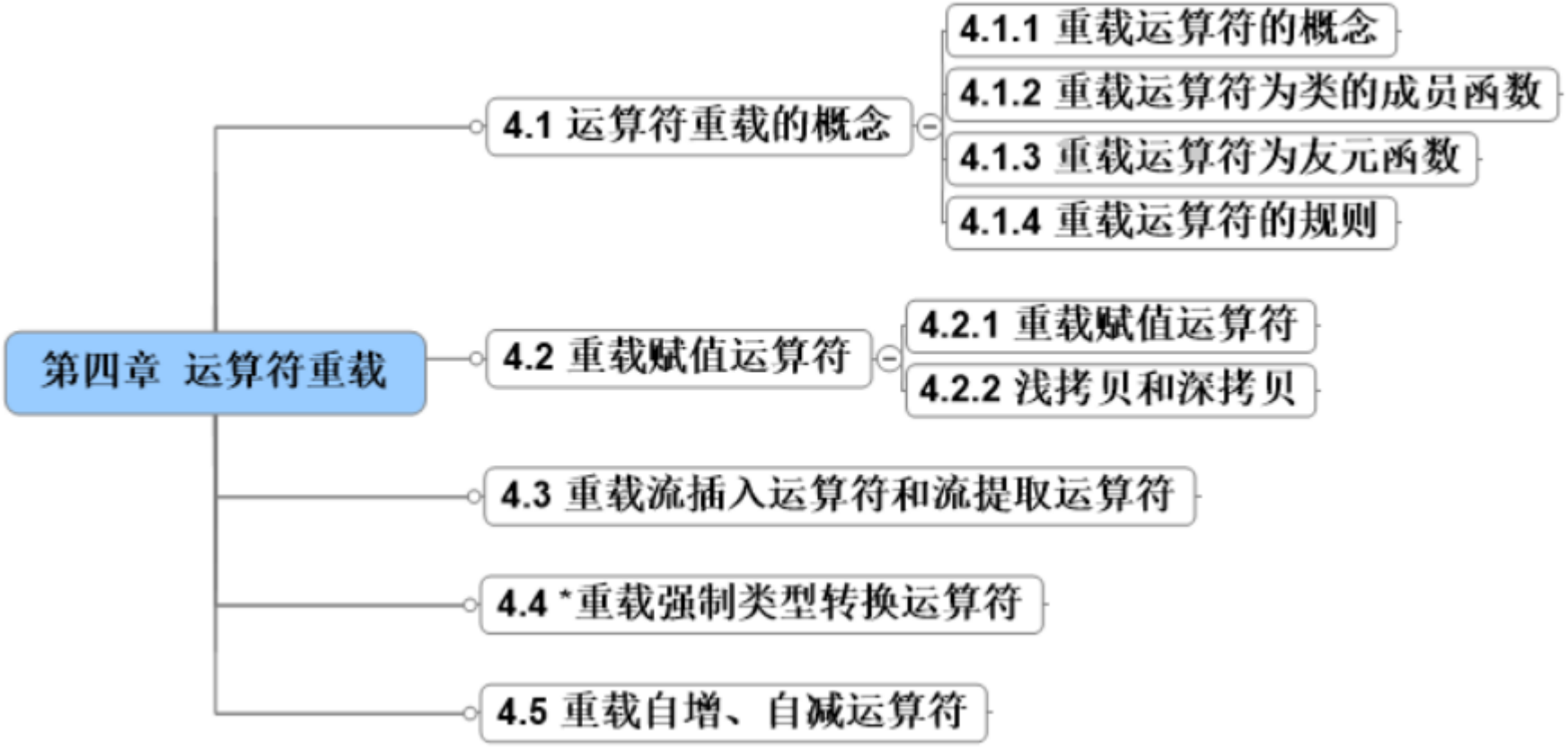
B:operator++(Aobject)

C:++(Aobject)

D:Aobject::operator++()

答案：A

本章总结


第四章 运算符重载
4.1 运算符重载的概念
4.1.1 重载运算符的概念
4.1.2 重载运算符为类的成员函数
4.1.3 重载运算符为友元函数
4.1.4 重载运算符的规则
4.2 重载赋值运算符
4.2.1 重载赋值运算符
4.2.2 浅拷贝和深拷贝
4.3 重载流插入运算符和流提取运算符
4.4 *重载强制类型转换运算符
4.5 重载自增、自减运算符

祝大家顺利通过考试！