§14. 运算符重载

1. 重载的引入与分类

函数重载(视频补充)：同一作用域内的不同函数具有相同的名称

★ 适用于完成不同个数、不同类型数据的相同操作

★ 函数参数的个数、类型不能完全相同

重载函数调用时的匹配查找顺序：

(1) 寻找参数个数、类型完全一致的定义 (严格匹配)

(2) 通过系统定义的内部转换寻找匹配函数(系统转换)

(3) 通过用户定义的内部转换寻找匹配函数(自定义转换)

★ 若某一步匹配成功，则不再进行下一顺序的匹配

★ 若每一步中发现两个以上的匹配则出错

运算符重载：同一运算符可以针对不同的数据类型完成相同的操作

```
#include <iostream>
using namespace std;
int max(int x, int y)
{
    cout << sizeof(x) << ' ';   1
    return (x > y ? x : y);
}
double max(double x, double y)
{
    cout << sizeof(x) << ' ';   2
    return (x > y ? x : y);
}
int main()
{
    cout << max(10,   15)      << endl;      严格匹配1
    cout << max(10.2, 15.3)    << endl;      严格匹配2
    cout << max(10,   int(15.3)) << endl;    系统转换1
//  cout << max(5+4i, 15.3)    << endl;      需自定义转换
    return 0;
}
```

```
4 15
8 15.3
4 15
```

复数形式目前编译会错，如何定义复数以及定义复数向double的转换，具体见荣誉课相关内容

§14. 运算符重载

1. 重载的引入与分类

例：复数的相加(实部、虚部对应相加)

```
//方法1：传统C方法
#include <iostream>
using namespace std;
struct Complex {
   double real;
   double imag;
};
struct Complex complex_add(struct Complex a, struct Complex b)
{
   Complex c;
   c.real = a.real+b.real;
   c.imag = a.imag+b.imag;
   return c;
}
int main()
{
   Complex c1={3,4}, c2={4,5}, c3;
   c3=complex_add(c1, c2);
   cout << c3.real << "+" << c3.imag << "i" << endl;   //输出时未考虑imag
}                                                      //为负,此处忽略
```

§14. 运算符重载

1. 重载的引入与分类

例：复数的相加(实部、虚部对应相加)

```
//方法2：C++成员函数方法
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0) { real = r;   imag = i; }
    void display(){ cout << real <<"+"<< imag <<"i" << endl; }
    Complex complex_add(Complex &b);
};
Complex Complex::complex_add(Complex &b)
{  Complex c;
   c.real = this->real+b.real;
   c.imag = this->imag+b.imag;
   return c;
}
int main()
{  Complex c1(3,4), c2(4,5), c3;
   c3=c1.complex_add(c2);
   c3.display();
}
```

this指针

```
因页面篇幅，未考虑imag为负,本章后面都相同
void Complex::display()
{   cout << real;
    if (imag>=0)   //考虑负数的体外实现方式
        cout << '+';
    cout << imag << 'i' << endl;
}
```

c3 = c1.complex_add(c2)时
this指针指向c1，形参为引用，
不调用复制构造
返回时调用复制构造函数

§14. 运算符重载

1. 重载的引入与分类

例：复数的相加(实部、虚部对应相加)

```
//方法3：C++运算符重载方式
#include <iostream>
using namespace std;

class Complex {
   ...
};

int main()
{  Complex c1, c2, c3;
   ...
   c3 = c1+c2;
}
```

```
c3 = c1.complex_add(c2);
```

```
c3 = complex_add(c1, c2);
```

§14. 运算符重载

2. 运算符重载的方法

方法：定义一个重载运算符的函数，当执行运算符时，自动执行该函数，达到相应的目的

★ 转换为函数重载，符合函数重载的匹配规则

形式：

```
返回类型 operator 运算符(形参表)
{
    重载函数实现
}
```

★ 用 operator 运算符 来表示对应运算符的函数

operator + ⇔ +

operator * ⇔ *

★ 对象 运算符 另一个值(可以不是对象、可以无)被解释为 对象.operator运算符(另一个值)

c1 + c2 ⇔ c1.operator+(c2)

★ 运算符被重载后，原来用于其它数据类型上的功能仍然被保留(重载)，系统根据重载函数的规则匹配

+ 仍然能表示两个int/两个float的加

§14. 运算符重载

2. 运算符重载的方法

例：复数的相加(实部、虚部对应相加)

```
//方法3：C++运算符重载方式
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0) { real = r;   imag = i; }
    void display(){ cout << real <<"+"<< imag <<"i" << endl; }
    Complex operator+(Complex &b);
};
Complex Complex::operator+(Complex &b)
{  Complex c;
   c.real=real+b.real;
   c.imag=imag+b.imag;
   return c;
}
int main()
{  Complex c1(3,4),c2(4,5),c3;
   c3 = c1+c2;
   c3.display();
}
```

实现部分与方法2相同
两者除函数名外没什么不同
因此实现相同

c3 = c1.operator+(c2); 转为函数
c3 = c1.complex_add(c2);方法2

若希望以cout<<c3的形式输出，要重载cout

§14. 运算符重载

3. 运算符重载的规则

共9条

★ 对已有运算符进行重载，不能定义新运算符

★ 除5个运算符外都允许重载(唯一的三目 ?: 不允许)

★ 不能改变操作对象的个数

★ 不能改变优先级

★ 不能改变结合性

★ 不允许带默认参数

★ 重载运算符的两侧至少有一个是类对象

★ =和&系统缺省做了重载，=是对应内存拷贝,&取地址

★ 应当使重载运算符的功能与标准相同/相似(建议)

不能重载的运算符只有 5 个：

.	(成员访问运算符)
*	(成员指针访问运算符)
::	(域运算符)
sizeof	(长度运算符)
?:	(条件运算符)

前面例子：c3=c1+c2
+是重载，=是缺省重载
可根据需要自己写=的重载

§13. 动态内存申请
4. 含动态内存申请内存的类和对象
4.4. 含动态内存申请的对象的赋值与复制
4.4.1. 含动态内存申请的对象的赋值
4.4.2. 含动态内存申请的对象的复制
当对象数据成员包含动态申请的内存指针时，都可能错
解决方法：
赋值：后续模块 重载=运算符
复制：自定义复制构造函数替代系统缺省的复制构造

§13. 动态内存申请

遗留问题

用多编译器分别
运行下面两个例子

4.4. 含动态内存申请的对象的赋值与复制

4.4.1. 含动态内存申请的对象的赋值

★ 若对象数据成员是指针及动态分配的数据，则可能导致不可预料的后果

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
class test {
  private:
    int   a;
    int   b;
    char *c;
  public:
    test(const char *s="A") {
      a=0;   b=0;
      c = new char[20];
      strcpy(c, s);
      }
    void set(const char *s) {
      strcpy(c, s);
      }
    void display() {
      cout << c << endl;
      }
  };
int main()
{   test t1("hello"), t2;
    t1.display();                hello
    t2.display();                A
    t2=t1;
    t2.display();                hello
    t1.set("china");
    t1.display();                china
    t2.display();                china
}
```

有动态内存申请
执行结果错(与期望不同)

注意：
1、篇幅问题，假设申请成功
2、程序不完整，仅在构造函数中动态申请，未定义析构函数进行释放

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
class test {
  private:
    int   a;
    int   b;
    char *c;
  public:
    test(const char *s="A") {
      a=0;   b=0;
      c = new char[20];
      strcpy(c, s);
      }
    ~test() {
      delete []c;
      };
    void set(const char *s) {
      strcpy(c, s);
      }
    void display() {
      cout << c << endl;
      }
  };
int main()
{   test t1("hello"), t2;
    t1.display();                hello
    t2.display();                A
    t2=t1;
    t2.display();                hello
    t1.set("china");
    t1.display();                china
    t2.display();                china
}
```

有动态内存申请
执行结果错(与期望不同)

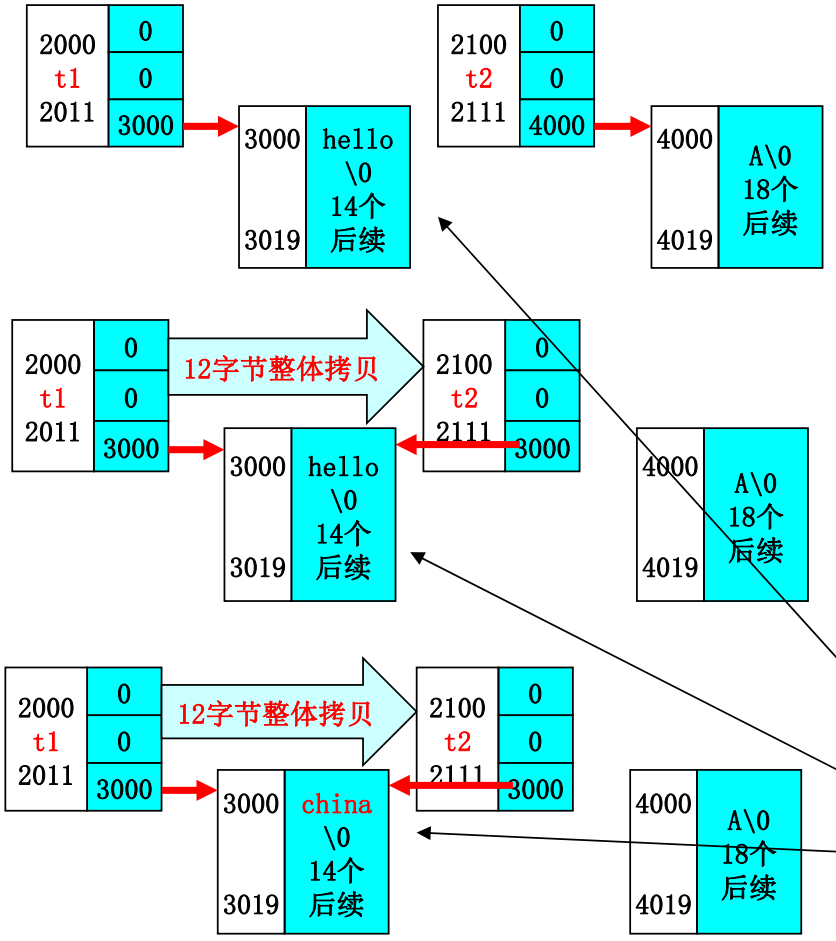
同左例，加入析构函数后，
不但执行结果错，而且
VS2019下会有错误弹窗，
GNU下虽然没有错误弹窗，
但仍然是错误的

4.4.含动态内存
4.4.1.含动态内
★ 若对象数据成

遗留问题

错误原因的图解及具体解释：
1、造成4000-4019这20个字节的内存丢失
2、t1/t2的c成员同时指向一块内存，通过t1的c修改改内存块，会导致t2的c值同时改变
3、若定义了析构函数，则main函数执行完成后系统会调用析构函数(按t2,t1的顺序)，t2调用析构函数释放3000-3019后，再调用t1的析构函数会导致重复释放3000-3019，错!!!

如何保证有动态内存时的赋值正确性?
后续模块 重载=运算符




```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;
class test {
  private:
    int   a;
    int   b;
    char *c;
  public:
    test(const char *s="A") {
      a=0;   b=0;
      c = new char[20];
      strcpy(c, s);
      }
    ~test() {
      delete []c;
      };
    void set(const char *s) {
      strcpy(c, s);
      }
    void display() {
      cout << c << endl;
      }
};
int main()
{   test t1("hello"), t2;
    t1.display();                hello
    t2.display();                A
    t2=t1;
    t2.display();                hello
    t1.set("china");
    t1.display();                china
    t2.display();                china
}
```

有动态内存申请
执行结果错(与期望不同)

同左例，加入析构函数后，不但执行结果错，而且VS2019下会有错误弹窗，GNU下虽然没有错误弹窗，但仍然是错误的

§14. 运算符重载

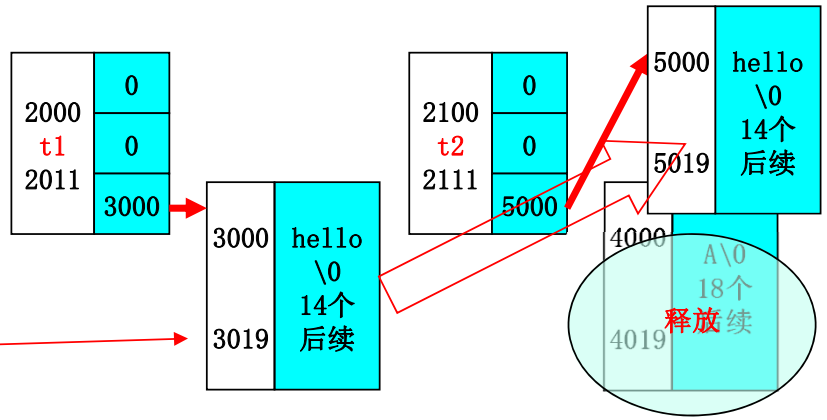
3. 运算符重载的规则

遗留问题的解决

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

class test {
  private:
    int   a;
    int   b;
    char *c;
  public:
    test(char *s="A") {
      a=0;   b=0;
      c = new char[20];
      strcpy(c, s);
      }
    ~test() {
      delete []c;
      };
    void set(char *s) {
      strcpy(c, s);
      }
    void display() {
      cout << c << endl;
      }
    test &operator=(const test &t);  //重载=的声明
  };
test &test::operator=(const test &t) //重载=体外实现
{     a = t.a;   b = t.b;
      delete []c;     //释放原空间
      c=new char[20]; //申请新空间
      strcpy(c, t.c);
      return *this;   //返回对象自身
}
```

```
int main()
{   test t1("hello"), t2;
    t1.display();              hello
    t2.display();              A
    t2=t1;
    t2.display();              hello
    t1.set("china");
    t1.display();              china
    t2.display();              hello
}
```



§14. 运算符重载

3. 运算符重载的规则

遗留问题的解决

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

class test {
  private:
    int   a;
    int   b;
    char *c;
  public:
    test(char *s="A") {
      a=0;   b=0;
      c = new char[20];
      strcpy(c, s);
      }
    ~test() {
      delete []c;
      };
    void set(char *s) {
      strcpy(c, s);
      }
    void display() {
      cout << c << endl;
      }
    test &operator=(const test &t);  //重载=的声明
  };
test &test::operator=(const test &t) //重载=体外实现
{     a = t.a;   b = t.b;
      delete []c;     //释放原空间
      c=new char[20]; //申请新空间
      strcpy(c, t.c);
      return *this;   //返回对象自身
}
```

```
int main()
{   test t1("hello"), t2;
    t1.display();              hello
    t2.display();              A
    t2=t1;
    t2.display();              hello
    t1.set("china");
    t1.display();              china
    t2.display();              hello
}
```

问：返回类型test &,不能是test,为什么?

答：t2=t1理解为t2.operator=(t1),即this指针指向t2,而=的语义希望执行后t2被改变，因此返回test&

语义：若返回test，则返回时会调用复制构造函数，返回的就是临时对象而不是t2自身,因此返回不能是test

正确性：使用了缺省的复制构造函数导致运行出错

=> 续：具体错在哪里？如何分析内存?

另一个例子： complex operator+(complex &b);

c1+c2理解为c1.operator+(c2)，+的语义不能改变c1，因此+应该返回临时对象，所以返回值是 complex 而不是 complex&

结论：

函数重载的返回值应该由运算符的语义决定

问：operator=的返回类型 void 可以吗?

答：1、对本题而言，void正确

2、如果出现类似t3=(t2=t1) 就会错误

=> 第2章 赋值表达式的值等于左值

=> 不可以void

§14. 运算符重载

3. 运算符重载的规则

遗留问题的解决

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

class test {
  private:
    int   a;
    int   b;
    char *c;
  public:
    test(char *s="A") {
      a=0;   b=0;
      c = new char[20];
      strcpy(c, s);
      }
    ~test() {
      delete []c;
      };
    void set(char *s) {
      strcpy(c, s);
      }
    void display() {
      cout << c << endl;
      }
    test &operator=(const test &t);  //重载=的声明
  };
test &test::operator=(const test &t) //重载=体外实现
{     a = t.a;   b = t.b;
      delete []c;     //释放原空间
      c=new char[20]; //申请新空间
      strcpy(c, t.c);
      return *this;   //返回对象自身
}
```

```
int main()
{   test t1("hello"), t2;
    t1.display();              hello
    t2.display();              A
    t2=t1;
    t2.display();              hello
    t1.set("china");
    t1.display();              china
    t2.display();              hello
}
```

因为申请/释放都是20字节，
因此可以仍旧使用原来已申请的空间
(蓝色两句全部不要)

若要求test类按需申请，不浪费空间
则必须释放原空间再申请新空间

```
int main()
{
    test t1("hello"),t2;
    ...
    t2=t1;
}
```

```
//假设构造函数按需申请
test(char *s="A")
{   a=0;
    b=0;
    c=new char[strlen(s)+1];
    strcpy(c, s);
}
```

```
test &test::operator=(const test &t)
{     a = t.a;
      b = t.b;
      delete []c;                  //释放原2字节   必须有
      c=new char[strlen(t.c)+1]; //申请新6字节
      strcpy(c, t.c);
      return *this;   //返回对象自身
}
```

§14. 运算符重载

3. 运算符重载的规则

遗留问题的解决

```
#define _CRT_SECURE_NO_WARNINGS
#include <iostream>
#include <cstring>
using namespace std;

class test {
  private:
    int   a;
    int   b;
    char *c;
  public:
    test(char *s="A") {
      a=0;   b=0;
      c = new char[20];
      strcpy(c, s);
      }
    ~test() {
      delete []c;
      };
    void set(char *s) {
      strcpy(c, s);
      }
    void display() {
      cout << c << endl;
      }
    test &operator=(const test &t);  //重载=的声明
  };
test &test::operator=(const test &t) //重载=体外实现
{     a = t.a;   b = t.b;
      delete []c;     //释放原空间
      c=new char[20]; //申请新空间
      strcpy(c, t.c);
      return *this;   //返回对象自身
}
```

```
int main()
{   test t1("hello"), t2;
    t1.display();              hello
    t2.display();              A
    t2=t1;
    t2.display();              hello
    t1.set("china");
    t1.display();              china
    t2.display();              hello
}
```

本程序的=重载，在出现自赋值情况(即t2=t2/t1=t1等形式，虽然无意义，但语法并不禁止)时会怎样？

1、观察运行结果
2、分析错误原因
3、如何才能修改正确

§14. 运算符重载　　重要!!!

3. 运算符重载的规则

```
//复制构造函数和重载=的区别
class test {
  ...
  public:
    test(const test &t);              //复制构造函数的声明
    test &operator=(const test &t);   //重载=的声明
  };

test::test(const test &s)    //复制构造函数的体外实现
{   a=s.a;
    b=s.b;
    c=new char[20];     //申请新空间
    strcpy(c, s.c);
}

test &test::operator=(const test &t)    //重载=的体外实现
{     a = t.a;
      b = t.b;
      delete []c;      //释放原空间   (构造不需要)
      c=new char[20]; //申请新空间
      strcpy(c, t.c);
      return *this;    //返回对象自身（构造不需要）
}

int main()
{
    test t1("hello");
    test t2(t1); //复制构造函数
    t2 = t1;     //赋值运算
}
```

	复制构造函数	重载赋值运算符
系统缺省	有，对应内存拷贝	有，对应内存拷贝
必须定义的时机	含动态内存申请时	含动态内存申请时
调用时机	定义时用对象初始化 函数形参为对象 函数返回值为对象	执行语句中的=操作
调用时处理	新对象生成时调用，此时不可能调用其它形式的构造函数 从未动态内存申请， 因此不考虑释放	已有对象=操作时调用，在=前对象已生成，即已调用过某种形式的构造函数(包括复制构造函数) 已有动态内存申请， 因此要考虑释放

```
class test {
    ...
    test(char *s="A");
    test(const test &t);
};
//普通构造函数的体外实现
test::test(char *s)
{   a=0;
    b=0;
    c=new char[20];
    strcpy(c, s);
}
```

```
//复制构造函数的体外实现
test::test(const test &s)
{   a=s.a;
    b=s.b;
    c=new char[20];
    strcpy(c, s.c);
}
```

```
int main()
{
   test t1("hello"), t2=t1, t3;
    t3 = t2; //调用此句前，t1/t2/t3
             //已调用过不同的构造函数
}            //并动态申请过空间
```

§14. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

类成员函数：

c1 + c2 ⇔ c1.operator+(c2) 14.2的例

友元函数：

全局函数做友元函数 ：合适

其它类的成员函数做友元函数：不合适

其他类做友元 ：不合适

★ 因为要访问类的成员，所以需要定义为友元

★ 运算符的同一种重载实现，友元/成员只能选择一个

★ 用全局普通函数也能实现运算符重载，但因为只能访问public对象，或通过类的公有成员函数访问private部分，因此效率低，一般不用

§14. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

★ 运算符的同一种重载实现，友元/成员只能选择一个

例：复数的相加(实部、虚部对应相加)

```
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0) {
        real = r;
        imag = i;
        }
    void display(){
        cout << real <<"+"<< imag <<"i" << endl;
        }
    friend Complex operator+(Complex &a, Complex &b);
};
Complex operator+(Complex &a, Complex &b) //全局函数
{  Complex c;
   c.real=a.real+b.real;
   c.imag=a.imag+b.imag;
   return c;
}
int main()
{  Complex c1(3,4),c2(4,5),c3;
   c3 = c1+c2;
   c3.display();
}
```

全局友元函数，没有this指针
通过 对象.成员 的形式调用
返回时调用复制构造函数

```
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0) {
        real = r;
        imag = i;
        }
    void display(){
        cout << real <<"+"<< imag <<"i" << endl;
        }
    Complex operator+(Complex &b);
};
Complex Complex::operator+(Complex &b) //成员函数
{   Complex c;
    c.real  = real  + b.real;
    c.image = image + b.image;
    return c;
}
int main()
{  Complex c1(3,4),c2(4,5),c3;
   c3 = c1+c2;
   c3.display();
}
```

成员函数，有this指针
直接 成员 的形式调用
返回时调用复制构造函数

§14. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

★ 用全局普通函数也能实现运算符重载，但因为只能访问public对象，或通过类的公有成员函数访问private部分，因此效率低，一般不用

```
#include <iostream>
using namespace std;

class Complex {
  public:
    double real;
    double imag;
    Complex(double r=0,double i=0) {
        real=r;
        imag=i;
        }
    void display(){
        cout << real <<"+" << imag <<"i" <<endl;
        }
};
Complex operator+(Complex &a, Complex &b) //全局非友元
{  Complex c;
   c.real=a.real+b.real;
   c.imag=a.imag+b.imag;
   return c;
}
int main()
{  Complex c1(3,4),c2(4,5),c3;
   c3 = c1+c2;
   c3.display();
}
```

通过直接访问公有成员方式实现
因为成员公有，可被任意外部访问
封装性受影响，无法保护内部数据

```
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0,double i=0) {
       real=r;
       imag=i;
       }
    void display(){
       cout << real <<"+" << imag <<"i" <<endl;
       }
    double getreal() { return real; }
    double getimag() { return imag; }
    void   setreal(double r) { real = r; }
    void   setimag(double i) { imag = i; }
};
Complex operator+(Complex &a, Complex &b) //全局非友元
{  Complex c;   double real, imag;
   real = a.getreal() + b.getreal();
   imag = a.getimag() + b.getimag();
   c.setreal(real);   c.setimag(imag);
   return c;
}
int main()
{  Complex c1(3,4),c2(4,5),c3;
   c3 = c1+c2;
   c3.display();
}
```

通过公有成员函数
访问私有成员，可
保护内部数据，但
效率受影响

§14. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

成员函数与友元函数的区别：

★ 对单目运算符

成员函数是空参数

友元函数是一个参数(必须是对象)

14.3规则7

★ 对双目运算符

成员函数是一个参数(可不是对象)

对象(this) 双目运算符 参数

友元函数是两个参数(一个必须是,一个可不是)

第一个参数 双目运算符 第二个参数

- 对两个都是对象的情况：没区别
- 对一个是对象的情况：

形式	C+d 成员实现	C+d 友元实现
c2 = c1 + 4	正确	正确
c2 = 4 + c1	错误	错误

若希望 c2 = 4+c1 正确，则需要重载实现 double + Complex
因为第一个参数不是类对象，该方式只能通过友元函数实现

§14. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

成员函数与友元函数的区别：

★ 对双目运算符

● 对一个是对象的情况：

```
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0)
       {   real = r;     imag = i; }
    void display(){
        cout << real <<"+" << imag <<"i" <<endl;
        }
    friend Complex operator+(Complex &a, double b);
  };
Complex operator+(Complex &a, double b) //全局函数
{  Complex c;
   c.real=a.real+b;//实部相加
   c.imag=a.imag;  //虚部不变
   return c;
}
int main()
{ Complex c1(3,4),c2;
  c2 = c1 + 4;  //正确
  c2 = 4 + c1;  //编译错
}
```

友元函数形式，
实现 Complex+double

```
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
 public:
    Complex(double r=0, double i=0)
       {   real = r;     imag = i; }
    void display(){
        cout << real <<"+" << imag <<"i" <<endl;
        }
    Complex operator+(double b);
  };
Complex Complex::operator+(double b) //成员函数
{  Complex c;
   c.real=real+b;//实部相加
   c.imag=imag;  //虚部不变
   return c;
}
int main()
{ Complex c1(3,4),c2;
  c2 = c1 + 4;  //正确
  c2 = 4 + c1;  //编译错
}
```

成员函数形式，
实现 Complex+double

§14. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

成员函数与友元函数的区别：

★ 对双目运算符

● 对一个是对象的情况：

两个友元重载

```
class Complex {
      ...
  public:
    friend Complex operator+(Complex &a, double b);    //友元函数
    friend Complex operator+(double   a, Complex &b); //友元函数
  };
Complex operator+(Complex &a, double b)
{  Complex c;
   c.real=a.real+b;  //实部相加
   c.imag=a.imag;    //虚部不变
   return c;
}
Complex operator+(double a, Complex &b)
{  Complex c;
   c.real=a+b.real;  //实部相加
   c.imag=b.imag;    //虚部不变
   return c;
}
```

c1 + 4

4 + c1

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;

  c2 = c1 + 4; //正确
  c2.display();
  c3 = 5 + c1; //正确
  c3.display();
} //double型常量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  double d1=4, d2=5;
  c2 = c1 + d1;//正确
  c2.display();
  c3 = d2 + c1;//正确
  c3.display();
} //double型变量
```

形式	C+d 成员实现	C+d 友元实现
c2 = c1 + 4	正确	正确
c2 = 4 + c1	错误	错误

若希望c2 = 4 + c1正确，则需要重载实现 double + Complex
因为第一个参数不是类对象，该方式只能通过友元函数实现

§14. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

成员函数与友元函数的区别：

★ 对双目运算符

● 对一个是对象的情况：

```
class Complex {
      ...
  public:
    Complex operator+(double b);
    friend Complex operator+(double a, Complex &b);
  };
Complex Complex::operator+(double b)     //成员函数
{  Complex c;
   c.real=real+b;  //实部相加
   c.imag=imag;    //虚部不变
   return c;
}
Complex operator+(double a, Complex &b)  //友元函数
{  Complex c;
   c.real=a+b.real;  //实部相加
   c.imag=b.imag;    //虚部不变
   return c;
}
```

友元/成员重载

c1 + 4

4 + c1

形式	C+d 成员实现	C+d 友元实现
c2 = c1 + 4	正确	正确
c2 = 4 + c1	错误	错误

若希望c2 = 4 + c1正确，则需要重载实现 double + Complex
因为第一个参数不是类对象，该方式只能通过友元函数实现

无法做到两个成员函数重载，
因为 4 + c1 无法表示为成员函数形式
原因：第1个参数(左值)不是类

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;

  c2 = c1 + 4; //正确
  c2.display();
  c3 = 5 + c1; //正确
  c3.display();
} //double型常量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  double d1=4, d2=5;
  c2 = c1 + d1;//正确
  c2.display();
  c3 = d2 + c1;//正确
  c3.display();
} //double型变量
```

关于+交换律的说明：
1、两个对象+，即定义两个类的+重载后，无论 c1+c2
还是 c2+c1，都调用同一重载函数，
具体实现不同（ c1.operator+(c2) / c2.operator+(c1) ）
但结果相同，可以理解为交换律存在
2、对象+其它类型，例如 Complex+double，交换律不存在，
交换律的表面现象是通过多个函数的重载来实现的
3、同理适用于其它存在交换律的运算符

§14. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

成员函数与友元函数的区别：

★ 对双目运算符

● 对一个是对象的情况：

将double换成double &，观察区别

两个友元重载

```
class Complex {
      ...
  public:
    friend Complex operator+(Complex &a, double b);   //友元函数
    friend Complex operator+(double   a, Complex &b); //友元函数
  };
Complex operator+(Complex &a, double b)
{  Complex c;
   c.real=a.real+b;  //实部相加
   c.imag=a.imag;    //虚部不变
   return c;
}
Complex operator+(double a, Complex &b)
{  Complex c;
   c.real=a+b.real;  //实部相加
   c.imag=b.imag;    //虚部不变
   return c;
}
```

c1 + 4

4 + c1

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;

  c2 = c1 + 4; //正确
  c2.display();
  c3 = 5 + c1; //正确
  c3.display();
} //double型常量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  double d1=4, d2=5;
  c2 = c1 + d1;//正确
  c2.display();
  c3 = d2 + c1;//正确
  c3.display();
} //double型变量
```

两个友元重载

```
class Complex {
      ...
  public:
    friend Complex operator+(Complex &a, double &b);  //友元函数
    friend Complex operator+(double  &a, Complex &b); //友元函数
  };
Complex operator+(Complex &a, double &b)
{  Complex c;
   c.real=a.real+b;  //实部相加
   c.imag=a.imag;    //虚部不变
   return c;
}
Complex operator+(double &a, Complex &b)
{  Complex c;
   c.real=a+b.real;  //实部相加
   c.imag=b.imag;    //虚部不变
   return c;
}
```

c1 + 4

4 + c1

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;

  c2 = c1 + 4; //错
  c2.display();
  c3 = 5 + c1; //错
  c3.display();
} //double型常量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  double d1=4, d2=5;
  c2 = c1 + d1;//正确
  c2.display();
  c3 = d2 + c1;//正确
  c3.display();
} //double型变量
```

§14. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

成员函数与友元函数的区别：

★ 对双目运算符

● 对一个是对象的情况：

将double换成double &，观察区别

```
class Complex {
      ...
  public:
    friend Complex opera
    friend Complex opera
  };
Complex operator+(Comple
{  Complex c;
   c.real=a.real+b;  //
   c.imag=a.imag;    //
   return c;
}
Complex operator+(double
{  Complex c;
   c.real=a+b.real;  //
   c.imag=b.imag;    //
   return c;
}
```

情况：

```
Complex operator+(Complex &a, double  b)
      c2 = c1 + double型常量    正确
      c2 = c1 + double型变量    正确

Complex operator+(Complex &a, double &b)
      c2 = c1 + double型常量    错误
      c2 = c1 + double型变量    正确
```

原因：引用是变量的别名，因此若函数形参为引用
则实参只能是变量，不能是常量/表达式

两个友元重载

```
ble &b);  //友元函数
plex &b); //友元函数
```

+ 4

c1

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;

  c2 = c1 + 4; //正确
  c2.display();
  c3 = 5 + c1; //正确
  c3.display();
} //double型常量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  double d1=4, d2=5;
  c2 = c1 + d1;//正确
  c2.display();
  c3 = d2 + c1;//正确
  c3.display();
} //double型变量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;

  c2 = c1 + 4; //错
  c2.display();
  c3 = 5 + c1; //错
  c3.display();
} //double型常量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  double d1=4, d2=5;
  c2 = c1 + d1;//正确
  c2.display();
  c3 = d2 + c1;//正确
  c3.display();
} //double型变量
```

§6. 指针与引用

6.3. 引用(C++新增)

6.3.1. 引用的基本概念

6.3.2. 引用作函数参数

★ 当形参是引用时，不需要声明时初始化，调用时，形参不分配空间，而当作实参的别名，因此对形参的访问就是对实参的访问

★ 实参虽然是变量名，但传递给形参的实际上是实参的地址，同时形参不单独分配空间，只是虚实结合

★ 引用允许传递

★ 当引用做函数形参时，实参不允许是常量/表达式，否则编译错误

(形参为const引用时实参可为常量/表达式)

```
#include <iostream>
using namespace std;

void fun(int x)
{
    cout << x << endl;
}
int main()
{   int i=10;
    fun(i);   //正确
    fun(15);  //正确
    return 0;
}
```

```
#include <iostream>
using namespace std;

void fun(int &x)
{
    cout << x << endl;
}
int main()
{   int i=10;
    fun(i);   //正确
    fun(15);  //编译错
    return 0;
}
```

```
#include <iostream>
using namespace std;

void fun(const int &x)
{
    cout << x << endl;
}
int main()
{   int i=10;
    fun(i);   //正确
    fun(15);  //正确
    return 0;
}
```

§14. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

成员函数与友元函数的区别：

★ 对双目运算符

● 对一个是对象的情况：

将double &换成const double &，即可编译正确

两个友元重载

```
class Complex {
      ...
  public:
    friend Complex operator+(Complex &a, double b);    //友元函数
    friend Complex operator+(double   a, Complex &b); //友元函数
  };
Complex operator+(Complex &a, double b)
{  Complex c;
   c.real=a.real+b;  //实部相加
   c.imag=a.imag;    //虚部不变
   return c;
}
Complex operator+(double a, Complex &b)
{  Complex c;
   c.real=a+b.real;  //实部相加
   c.imag=b.imag;    //虚部不变
   return c;
}
```

c1 + 4

4 + c1

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;

  c2 = c1 + 4; //正确
  c2.display();
  c3 = 5 + c1; //正确
  c3.display();
} //double型常量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  double d1=4, d2=5;
  c2 = c1 + d1;//正确
  c2.display();
  c3 = d2 + c1;//正确
  c3.display();
} //double型变量
```

两个友元重载

```
class Complex {
      ...
  public:
   friend Complex operator+(Complex &a, const double &b); //友元
   friend Complex operator+(const double  &a, Complex &b);//友元
  };
Complex operator+(Complex &a, const double &b)
{  Complex c;
   c.real=a.real+b;  //实部相加
   c.imag=a.imag;    //虚部不变
   return c;
}
Complex operator+(const double &a, Complex &b)
{  Complex c;
   c.real=a+b.real;  //实部相加
   c.imag=b.imag;    //虚部不变
   return c;
}
```

c1 + 4

4 + c1

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;

  c2 = c1 + 4; //正确
  c2.display();
  c3 = 5 + c1; //正确
  c3.display();
} //double型常量
```

```
int main()
{ Complex c1(3,4);
  Complex c2, c3;
  double d1=4, d2=5;
  c2 = c1 + d1;//正确
  c2.display();
  c3 = d2 + c1;//正确
  c3.display();
} //double型变量
```

§14. 运算符重载

4. 运算符重载函数做为类成员函数和友元函数

成员函数与友元函数的区别：

★ 建议对单目运算符采用成员函数方式，双目运算符采用友元函数方式

★ C++规定，某些运算符必须是成员函数形式(赋值=，下标[],函数()），某些运算符必须是友元函数形式(流插入<<，流提取>>)

§14. 运算符重载

5. 重载双目运算符

通过作业理解

§14. 运算符重载

6. 重载单目运算符

例：假设复数的-，规则为实部虚部全部换符号

```
#include <iostream>
using namespace std;

class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0)
    {   real = r; imag = i;
    }
    void display()
    {   cout << real <<"+" << imag <<"i" <<endl;
    }
    Complex operator-();
};
```

成员函数

为什么返回值不是complex & ？
答：不能返回自动变量的引用

```
Complex Complex::operator-( )     //成员函数
{   Complex c1;
    c1.real = -real;  //实部取反
    c1.imag = -imag;  //虚部取反
    return c1;
}
```

为什么不能如下方式实现 ？

```
Complex& Complex::operator-()
{  real = -real;  //实部取反
   imag = -imag;  //虚部取反
   return  *this;
}
```

答：因为-的语义不改变对象自身，所以要返回临时对象

同理：

```
int k = 10, m;
2*(-k); //k仍是10
m=-k;   //k仍是10
```

若仅有c2.display()，则看不出差异，测试用例很重要!!!

```
int main()
{   Complex c1(3,4),c2;
    c2 = -c1;
    c1.display();
    c2.display(); //输出-3+-4i形式
    return 0;
}
```

```
#include <iostream>
using namespace std;

class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0)
    {   real = r; imag = i;
    }
    void display()
    {   cout << real <<"+" << imag <<"i" <<endl;
    }
    friend Complex operator-(Complex &a);
};
```

友元函数

```
Complex operator-(Complex &a)  //友元函数
{   Complex c1;
    c1.real = -a.real;  //实部取反
    c1.imag = -a.imag;  //虚部取反
    return c1;
}
```

为什么不能如下方式实现 ？

```
Complex& operator-(Complex &a)
{  a.real = -a.real;//实部取反
   a.imag = -a.imag;//虚部取反
   return  a;
}
```

答：因为-的语义不改变对象自身，所以要返回临时对象

同理：

```
int k = 10, m;
2*(-k); //k仍是10
m=-k;   //k仍是10
```

若仅有c2.display()，则看不出差异，测试用例很重要!!!

```
int main()
{   Complex c1(3,4),c2;
    c2 = -c1;
    c1.display();
    c2.display(); //输出-3+-4i形式
    return 0;
}
```

§14. 运算符重载

6. 重载单目运算符

★ ++/--的前后缀区别

前缀：正常方式

后缀：多一个int型参数，不访问，仅进行区别

例：假设复数的++是实部++，虚部不动

为什么前缀++/--返回引用，后缀++/--返回对象？

答：根据语义，前缀是自身先++/--，再自身参与运算
因此返回引用，即对象自身，且不需调用复制构造
后缀是保存旧值，自身++/--，再旧值参与运算
因此返回对象，返回时调用复制构造产生临时对象

成员函数

```
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0) { real = r; imag = i; }
    void display(){ cout << real <<"+" << imag <<"i" <<endl;}
    Complex& operator++();     //前缀
    Complex  operator++(int);  //后缀
};
Complex& Complex::operator++() //前缀
{  real++;             //前后缀无所谓
   return *this;       //返回自身
}
Complex Complex::operator++(int) //后缀
{  Complex c1(*this); //复制构造函数，用对象自身初始化c1
   real++;             //前后缀无所谓，注意不是c1.real
   return c1;          //返回++前的值，符合后缀语义
}
int main()
{  Complex c1(3,4), c2;
   c2 = c1++;
   c1.display();
   c2.display();
   c2 = ++c1;
   c1.display();
   c2.display();
}
```

友元函数

```
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0) { real = r; imag = i; }
    void display(){ cout << real <<"+" << imag <<"i" <<endl;}
    friend Complex& operator++(Complex &a);      //前缀
    friend Complex  operator++(Complex &a, int); //后缀
  };
Complex& operator++(Complex &a) //前缀
{  a.real++;           //前后缀无所谓
   return a;           //全局函数没有this指针
}
Complex operator++(Complex &a, int) //后缀
{  Complex c1(a);      //复制构造函数，用对象自身初始化c1
   a.real++;           //前后缀无所谓，注意不是c1.real
   return c1;          //返回++前的值，符合后缀语义
}
int main()
{  Complex c1(3,4), c2;
   c2 = c1++;
   c1.display();
   c2.display();
   c2 = ++c1;
   c1.display();
   c2.display();
}
```

§14. 运算符重载

6. 重载单目运算符

通过作业理解

§14. 运算符重载

7. 重载流插入运算符和流提取运算符

7.1. 形式

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

★ >>本身是istream类(系统定义类)的成员函数，因此希望对istream类的 >> 运算符重载，使其能输入自定义类的内容

★ <<本身是ostream类(系统定义类)的成员函数，因此希望对ostream类的 << 运算符重载，使其能输出自定义类的内容

★ 不能用自定义类的成员函数方式来实现

假设：operator<< (Complex &c1, ostream &cout)

则意味着使用形式为： c1 << cout

结论：流插入和流提取不能通过重载Complex类来实现，而应该是重载istream/ostream类使其能接受自定义类

§14. 运算符重载

7. 重载流插入运算符和流提取运算符

7.2. 重载流提取运算符<<

例：假设复数的输出形式为 实部+虚部i

只能友元函数

```
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0) {
        real = r;
        imag = i;
        }
    friend ostream& operator << (ostream &out, Complex &a);
 };

ostream& operator<<(ostream &out, Complex &a)
{   out << a.real;
    if (a.imag>=0)
        out << '+';  //虚部小于0不需要+  (3-4i)
    out << a.imag << 'i';
    return out;
}

int main()
{   Complex c1(3,4);
    cout << c1 << endl;
}
```

§14. 运算符重载

7. 重载流插入运算符和流提取运算符

7.3. 重载流插入运算符>>

例：假设复数的输入形式为 实部 虚部

只能友元函数

```
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(double r=0, double i=0) {
        real = r;
        imag = i;
        }
    void display(){
        cout << real <<"+" << imag <<"i" <<endl;
        }
    friend istream& operator >> (istream &in, Complex &a);
 };
istream& operator>>(istream &in, Complex &a)
{   in >>  a.real >> a.imag;
    return in;
}
int main()
{   Complex c1;
    cin >> c1;
    c1.display();
}
```

§14. 运算符重载

8. 不同类型数据间的转换

8.1. 标准类型数据间的转换

隐式转换：　int + double;　//转换优先级、整型提升

　　int = double;　//以左值为准进行转换

第02模块的内容

显式转换：

　　C方式：　(int)89.5

　　C++方式：　int(89.5)　/ static_cast<int>(89.5)

★ 问题的引入：对于自定义类型，能否进行类型转换

```
class Time {
    ...; //假设包含hour/minute/second成员
};

int main()
{
   Time t1(14,15,23), t2;
   int sec;
   t2 = 3662; //期望t2为 1:1:2 (目前无法做到)
   sec = t1;  //期望sec为 51323 (目前无法做到)
}
```

```
class Complex {
    ...;  //假设实现0参和2参构造，无1参构造
};

int main()
{
   Complex c1(2.5, 3.5), c2;
   double d;
   c2 = 5.3; //期望c2为 5.3+0i (目前无法做到)
   d  = c1;  //期望d 为 2.5    (目前无法做到)
}
```

§14. 运算符重载

8. 不同类型数据间的转换

8.2. 用转换构造函数进行类型转换

形式：

```
类名(一个形参)
{
   函数实现
}
```

★ 只能带一个参数，非该类的数据类型，将该参数转换为对应的对象

★ 系统无缺省定义，根据需要自行定义并使用

§14. 运算符重载

8. 不同类型数据间的转换

8.2. 用转换构造函数进行类型转换

例：将double转换为Complex的规则，成为对应实部

```
#include <iostream>
using namespace std;

class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(){ real=0; imag=0;}
    Complex(double r, double i){ real=r; imag=i;}
    Complex operator+(const Complex &b)  //const引用
      { return  Complex(real+b.real, imag+b.imag);}
    Complex(double r) { real = r; imag = 0; } //转换构造
    void display()
      { cout << real <<"+"<< imag <<"i" << endl; }
};

int main()
{
  Complex c1(3,5), c2;
  c2=c1+2.5;              5.5+5i
  c2.display();
  c2=c1+Complex(2.5);     5.5+5i
  c2.display();

  return 0;
}
```

隐式调用转换构造函数，建立无名Complex对象，再与c1相加

显式调用转换构造函数，建立无名Complex对象，再与c1相加

重载函数调用时的匹配查找顺序：

（1）寻找参数个数、类型完全一致的定义（严格匹配）

（2）通过系统定义的内部转换寻找匹配函数

（3）通过用户定义的内部转换寻找匹配函数

★ 若每一步中发现两个以上的匹配则出错

c1+2.5

(1)是否有Complex+double? 无

(2)是否有系统内部定义的转换? 无

(3)是否有用户定义的内部转换? 有 d => C

c1+Complex(2.5)

(1)是否有Complex+Compelx? 有

§14. 运算符重载

8. 不同类型数据间的转换

8.2. 用转换构造函数进行类型转换

例：将double转换为Complex的规则，成为对应实部

```
#include <iostream>
using namespace std;

class Complex {
  private:
    double real;
    double imag;
  public:
    Complex(){ real=0; imag=0;}
    Complex(double r, double i){ real=r; imag=i;}
    Complex operator+(const Complex &b)   //const引用
      { return  Complex(real+b.real, imag+b.imag);}
    Complex(double r) { real = r; imag = 0; } //转换构造
    void display()
      { cout << real <<"+"<< imag <<"i" << endl; }
};

int main()
{
  Complex c1(3,5), c2;
  c2=c1+2.5;              5.5+5i
  c2.display();
  c2=c1+Complex(2.5);   5.5+5i
  c2.display();

  return 0;
}
```

问：为什么不能合并为
Complex(double r=0, double i=0)
  { real = r; imag = i; }
答：若缺省为一个参数时，与转换构造存在二义性

问：
1、为什么不用 Complex b
答：每次调用复制构造，效率低

2、为什么不用 Complex &b
答：实参不能是常量/表达式

隐式调用转换构造函数，建立无名Complex对象，再与c1相加

显式调用转换构造函数，建立无名Complex对象，再与c1相加

§14. 运算符重载

8. 不同类型数据间的转换

8.2. 用转换构造函数进行类型转换

例：整数n(0-86399)表示当天的秒，转换为Time对象

```
#include <iostream>
using namespace std;
class Time {
  private:
    int hour, minute, sec;
  public:
    Time() {hour=0; minute=0; sec=0;}
    Time(int second);
    Time(int h, int m, int s) { hour = h; minute = m; sec = s; }
    Time operator-(Time &t);
    void display(){ cout << hour << ":" << minute << ":" << sec <<endl; }
};
Time Time::operator-(Time &t)
{  int s;   Time tt;
   s = (hour*3600+minute*60+sec) - (t.hour*3600+t.minute*60+t.sec);
   tt.hour=s/3600;
   tt.minute=s%3600/60;
   tt.sec=s%60;
   return tt;
}
Time::Time(int second)
{  hour = second/3600; minute = second%3600/60; sec = second % 60;
}
int main()
{ Time t1, t2(3,0,0);
  t1 = Time(65234)-t2;
  t1.display();
}
```

```
65234 = 18:7:34
65234-3600*3 = 15:7:34
```

```
15:7:34
```

§14. 运算符重载

8. 不同类型数据间的转换

8.2. 用转换构造函数进行类型转换

★ 单一个参数的构造函数也可以不做为转换构造函数只有完成转换功能才称为转换构造函数

例：Time类

```
class Time {
   private:
      int hour;
      int minute;
      int sec;
   public:
      Time(int second);
   };

Time::Time(int second)     //转换构造
{
    hour   = second / 3600;
    minute = second % 3600 / 60;
    sec    = second % 60;
}
```

例：Time类

```
class Time {
   private:
      int hour;
      int minute;
      int sec;
   public:
      Time(int second);
   };

Time::Time(int second)  //带一个参数
{                       //的普通构造
    hour   = 0;
    minute = 0;
    sec    = second % 60;
}
```

两者无法重载，因为函数名、参数个数、参数类型完全相同

§14. 运算符重载

8. 不同类型数据间的转换

8.2. 用转换构造函数进行类型转换

★ 两种使用方式

类名(要转换类型的数据)　显式调用

对象名 = 要转换的数据　隐式调用

假设已实现无参/两参构造函数及两个复数+及转换构造

```
Complex c1(3,5), c2;
c2 = 3.5;                    //正确(隐式)
c2 = c1 + Complex(3.5);      //正确(显式)
c2 = c1 + 3.5;               //正确(当复数+的参数是const引用时)
```

§14. 运算符重载

8. 不同类型数据间的转换

8.2. 用转换构造函数进行类型转换

★ 也可以将一个类的对象转换为另一个类的对象

类名(要转换类型的对象)

对象名 = 要转换的另一个对象

例：假设两个类表示身高，单位分别是厘米和米

```
class B;
class A {
  private:
    int height;
  public:
    A(int h) { height = h; }
    friend B;
};
class B {
  private:
    double height;
  public:
    B() { height = 0; }
    B(A ha) { height = ha.height/100.0; }
    void display() { cout << height << endl; }
};
```

```
int main()
{
  A ha1(178), ha2(183);
  B hb1, hb2;

  hb1 = B(ha1);
  hb1.display();

  hb2 = ha2;
  hb2.display();

  return 0;
}
```

```
1.78
```

```
1.83
```

§14. 运算符重载

8. 不同类型数据间的转换

8.3. 用类型转换函数进行类型转换

含义：将一个类对象转换为另一个类型的数据

形式：

```
operator 类型名()
{
    函数体实现;
}
```

★ 返回类型不是缺省的int，由类型名决定，无参数

★ 只能是成员函数形式

★ 在需要类型转换的地方被系统自动地隐式调用

```
例：将Time类转换为秒
#include <iostream>
using namespace std;

class Time {
  private:
    int  hour;
    int  minute;
    int  sec;
  public:
    Time(int h=0, int m=0, int s=0) {
        hour = h;
        minute = m;
        sec = s;
        }
    operator int();   //类型转换函数
};

Time::operator int()  //类型转换函数的体外实现
{
    return hour*3600 + minute*60 + sec;
}

int main()
{
    Time t1(14,15,23);
    int k;
    k = 10 + t1;
    cout << k << endl;
}
```

14*3600 + 15*60 + 23 = 51323

51333

10 + t1，因为没有定义 int + Time 的重载，因此t1被转换为int(隐式调用类型转换函数)，再int相加

§14. 运算符重载

8. 不同类型数据间的转换

8.3. 用类型转换函数进行类型转换

含义：将一个类对象转换为另一个类型的数据

形式：

```
operator 类型名()
{
    函数体实现;
}
```

★ 返回类型不是缺省的int，由类型名决定，无参数

★ 只能是成员函数形式

★ 在需要类型转换的地方被系统自动地隐式调用

重载函数的匹配顺序：
(1)是否有double+Complex?
　　无
(2)是否有系统内部转换?
　　无
(3)是否有用户定义转换?
　　有　C => d

2.5 + c1，因为没有定义double+复数的重载，因此c1被转换为double
(隐式调用类型转换函数)，再double相加

例：将complex对象转换为double的规则：实部赋值

```
#include <iostream>
using namespace std;

class Complex {
  private:
    double real;
    double imag;
  public:
    Complex() {
        real = 0;
        imag = 0;
        }
    Complex(double r, double i) {
        real = r;
        imag = i;
        }
    operator double() {
        return real;
        }
};

int main()
{
   Complex c1(3,4);
   double d1;
   d1 = 2.5 + c1;          5.5
   cout << d1 << endl;
}
```

§14. 运算符重载

8. 不同类型数据间的转换

8.3. 用类型转换函数进行类型转换

含义：将一个类对象转换为另一个类型的数据

形式：

```
operator 类型名()
{
    函数体实现;
}
```

★ 返回类型不是缺省的int，由类型名决定，无参数

★ 只能是成员函数形式

★ 在需要类型转换的地方被系统自动地隐式调用

例：将complex对象转换为double的规则：实部赋值

```
#include <iostream>
using namespace std;

class Complex {
  private:
    double real;
    double imag;
  public:
    Complex() {
        real = 0;
        imag = 0;
        }
    Complex(double r, double i) {
        real = r;
        imag = i;
        }
//  operator double() {
//      return real;
//      }
};

int main()
{
   Complex c1(3,4);
   double d1;
   d1 = 2.5 + c1;
   cout << d1 << endl;
}
```

编译错!!!

1、无 double+复数 的重载

2、无 复数转double 的类型转换函数，也无法理解为 double+

3、无 double转复数 的转换构造函数及 复数+复数 的重载，也无法理解为复数+

§14. 运算符重载

8. 不同类型数据间的转换

8.3. 用类型转换函数进行类型转换

含义：将一个类对象转换为另一个类型的数据

形式：

```
operator 类型名()
{
    函数体实现;
}
```

★ 返回类型不是缺省的int，由类型名决定，无参数

★ 只能是成员函数形式

★ 在需要类型转换的地方被系统自动地隐式调用

例：将complex对象转换为double的规则：实部赋值

```
#include <iostream>
using namespace std;
class Complex {
  private:
    double real;
    double imag;
  public:
    Complex() {  real = 0;   imag = 0;  }
    Complex(double r, double i){
      real = r;
      imag = i;
      }
    operator double() {  return real;  }
    Complex operator+(const Complex &b) {
      return Complex(real+b.real, imag+b.imag);
      }
};

int main()
{
   Complex c1(3,4),c2(5,-10);
   double d1;
   d1 = 2.5 + c1;
   cout << d1 << endl;     5.5
   d1 = c1 + c2;
   cout << d1 << endl;     8
}
```

2.5 + c1，
因为没有定义double+复数的重载，
因此c1被转换为double
(隐式调用类型转换函数)，
再double相加，得5.5

c1 + c2，
复数的+重载得到8-6i，
赋值给d1时被转换为double
(隐式调用类型转换函数)，
d1的值为8

§14. 运算符重载

8. 不同类型数据间的转换

8.3. 用类型转换函数进行类型转换

含义：将一个类对象转换为另一个类型的数据

形式：

```
operator 类型名()
{
    函数体实现;
}
```

★ 返回类型不是缺省的int，由类型名决定，无参数

★ 只能是成员函数形式

★ 在需要类型转换的地方被系统自动地隐式调用

假设已实现类型转换及两个复数的+

```
d1 = 2.3 + c1; //c1     被隐式转换
d1 = c1  + c2; //c1+c2  被隐式转换
```

★ 定义转换函数后，注意与转换构造函数的区别

§14. 运算符重载

8. 不同类型数据间的转换

★ 拓展讨论：转换构造函数及类型转换函数的使用

```
#include <iostream>
using namespace std;

class Complex {
  private:
    double real;
    double imag;
  public:
    Complex() { real = 0; imag = 0; }
    Complex(double r, double i) { real = r; imag = i; }
    Complex(double r) { real = r; imag = 0; }   //转换构造函数
    operator double() { return real; }          //类型转换函数
    void display(){ cout << real <<"+"<< imag <<"i" << endl; }
    //注意：下面两个用的时候只能选择一个，另一个要注释掉!!!!!!!!!!
    Complex operator+(const Complex &c2);
    friend Complex operator+(const Complex &c1, const Complex &c2);
};
```

```
Complex Complex::operator+(const Complex &c2)
{   return Complex(real+c2.real, imag+c2.imag);
}
Complex operator+(const Complex &c1, const Complex &c2)
{   return Complex(c1.real+c2.real, c1.imag+c2.imag);
}
```

```
//注意：这两个用的时候
//只能选择一个，另一个
//要注释掉!!!!!!!!!!
```

```
int main()
{  Complex c1(3,4), c3;
   c3 = c1 + Complex(2.5);
   c3.display();
   c3 = c1 + 2.5;
   c3.display();
   c3 = 2.5 + c1;
   c3.display();
}
```

通过作业而完成

	+的实现	c3=c1+Complex(2.5)	c3=c1+2.5	c3=2.5+c1
无转换构造 无类型转换	友元			
	成员			
无转换构造 有类型转换	友元			
	成员			
有转换构造 无类型转换	友元			
	成员			
有转换构造 有类型转换	友元			
	成员			