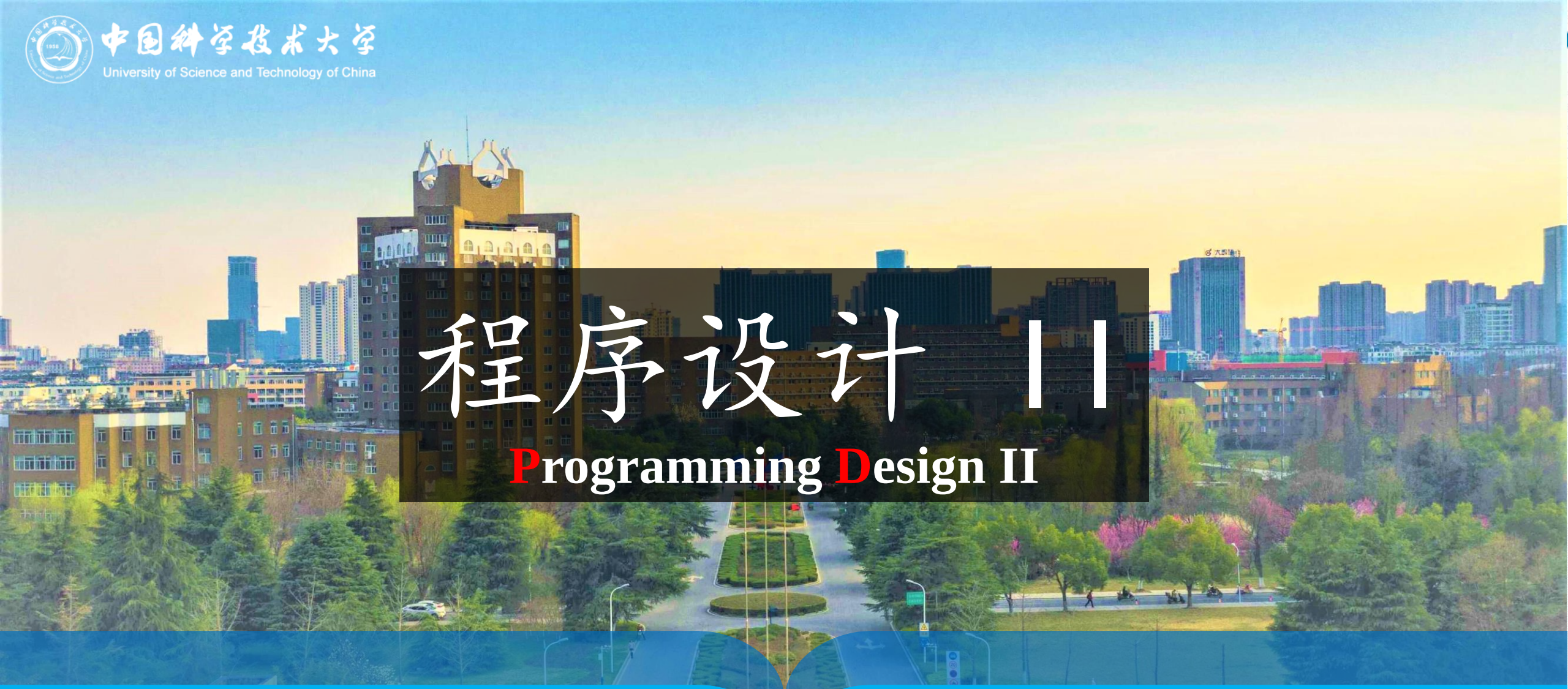

中国科学技术大学
University of Science and Technology of China
程序设计 II
Programming Design II

C++类的成员

主讲：吴锋

目录
CONTENTS

成员变量类型

- 成员变量可以是基本类型、构造类型，甚至是一个另一个类的对象、对象指针或引用。
- 类对象成员初始化一般使用成员初始化表。

```
class X {
private:
    int   basetypemem;              // 基本类型
    char str[20];                   // 构造类型
    Cstack    obj_stack;            // 另一个类的对象，必须之前定义
    CTime    *obj_time;             // 另一个类的对象指针
    CTime    &obj_time;             // 另一个类的对象引用
    …
};
```

成员变量类型

- 提前声明：一个类提前声明之后，这个类的对象指针、引用可以作为另一个类的成员变量。
- 但这个类的对象不能作另一个类的成员变量。根据编译器一遍扫描原则，不知道如何初始化。

```
class Stack;
class StackofStack {
private:
      int    topStack;
      Stack *stackstorage;     // 指针对象成员
      Stack &stackstorage2;    // 引用对象成员
};
```

成员函数定义

- 在类的定义体外定义：
  - 一般较复杂的函数都在体外定义，这样显得更简洁，也可以方便的隐藏具体实现细节。

```
class Stack {
public:
        int empty();
        void pop();
        int  top();
        void push(int);
};

int Stack::empty () {
     return (nb_elements==0);
}
```

成员函数定义

- 在类的定义体内定义：
  - 一般简单的函数才在类的定义体内定义。此时，编译器会自动优化成内联函数（即时未标注 inline）。

```
class Stack {
public:
        int empty() {  return (nb_elements==0); }
        void pop();
        int  top();
        void push(int);
};
```

成员函数的指针调用

- 除了直接调用成员函数外，还可以使用函数指针调用成员函数。

```
class Stack {
public:
        int empty();
        void pop();
        int  top();
        void push(int);
};

void main() {
          Stack s; s.top(); Stack *p = &s; p->top();
          int (Stack::*pftop)() = &Stack::top;
          (s.*pftop)(); (p->*pftop)();
}
```

隐含的 this 指针

- 一个类所有的对象的成员变量(除了静态变量)都有独立空间，但同一个类的所有对象成员共享同一组成员函数。
- 问题：当类的多个实例同时存在时，在一个类的成员函数内部引用一个数据成员，并没有指明一个类的具体实例。
- 解决办法：编译器给成员函数传递一个隐含的对象指针参数，该指针总是指向当前要引用的对象。这一隐含的指针称为“this指针”。

隐含的 this 指针

- 类中的非静态成员函数都隐式地声明 this 指针，在非静态成员函数中可以直接使用。
- 隐含的 this 是一个常量指针 (X * const this)，使用过程中不能修改隐含的 this 指针的值。

```
class  Ctest {
private:
    int n;
public:
    int getn() { return this->n; }          // 等价于return n;
    void setn() { this = new Ctest; }       // 非法操作
};
```

this 指针的用途之一

- 通过返回*this的引用，可以实现成员函数的链式调用。

```
Class X{
public:
        X &assign() { … return (*this); }
        X &setvalue() {… return (*this) ;}
        …
};

X objX;
objX.assign().setvalue().assign();    // 从左向右结合
```

this 指针的用途之二

- 判断是否是对象本身，防止自身赋值。

```
class String{
    private:
            char * str;
    public:
            String & assign(const String & s) {
                    if(this == &s) // 自身赋值
                        return (*this);
                    …
             }
};

String   s1("How are you?"), s2("Fine");
s1.assign(s2); s2.assign(s2);
```

◎ const 的作用

- **变量安全性**：如果想用运行期间产生的值初始化一个变量，并且知道在该变量的生命期内其值不变，则可用const限定该变量。
- **参数传递**：建议用const引用传递替代值传递，这样更加简单、高效。

```
void f(const int &i) { i++; } //错误，i不能改变
```

- **返回值**：按值返回自定义类型的const，可以阻止了返回值作为左值出现。

```
const X& X::f() { return (*this); }
f()=x1; f().modify(); modify(f()); //错误，对象不能改变
```

const 数据成员

```
class Fred {
  const int size;
public:
  Fred(int sz);
  void print();
};

Fred::Fred(int sz) : size(sz) {}
void Fred::print() { cout << size << endl; }

void main() {
  Fred a(1), b(2), c(3);
  a.print(), b.print(), c.print();
}
```

◎ const 对象

- const对象被初始化后，数据成员在其生命期内不被改变。

```
const X x(2);   // const 对象
```

- 对于公有数据，只要判断定义为const后，用户是否改变数据，如判断：x.i = 3; 为非法操作。
- 用户在调用成员函数时，对象需要知道哪些成员函数将会改变数据、哪些函数不会。
- 为此，需要强制声明和定义const成员函数，显式地告诉编译器这些函数对数据是安全的，可以被const 对象调用，其它函数则不可被其调用。

const 成员函数

- 在成员函数的声明和定义后面加上const使之成为const成员函数，保证不修改对象数据。

```
class X {
            int i;
public:
            X(int ii) : i(ii) {};
            int const_f() const;
            int f();
};

int X::const_f() const { return i; }
int X::f(){ … }
```

```
void main() {
  X x1(10);
  const X x2(20);
  x1.f();          // OK
  x1.const_f();    // OK
  x2.f();          // ERROR
  x2.const_f();    // OK
}
```

const 成员函数

- 声明为 mutable 的数据成员为 const 成员函数可以改变的数据成员，即允许const 对象改变的数据项（const 对象只能调用const 成员函数）。

```
class X {
    int i;
    int j;
    mutable int k;
public:
    int const_f() const;
};
int X::const_f() const {
      j = 1; // 错误，不能改变j的值
      k = 2; // OK， 可以改变k的值
    return i;
}
```

```
void main() {
  X x1(10);
  const X x2(20);
  x1.const_f();       // OK
  x2.const_f();       // OK
}
```

const 对象与成员函数

- 声明为 const 的对象是不能被赋值的。
- 声明为 const 的对象不能随便调用任意的成员函数，防止改变数据。

```
x2.f();  // 错误, f()非const 成员函数
```

- 声明为 const 的对象只能调用声明为 const 的成员函数，保证不改变数据。

```
x2.const_f() ; // OK
```

- const 的成员函数不能改变一般成员变量，否则编译器会报错，除非变量声明为mutable。

static 数据成员

- 拥有一块独立的存储区，每个类类型只有一个拷贝，而不管创建了多少个该类的对象，所有的对象都可以共享访问的数据成员。
- 静态数据成员遵从 public/private/protected 的访问控制规则，分为公有成员和私有成员。
- 静态成员通常用于存储类的公共数据。

```
class A{
public:
    static int i; // 可以通过 A::i 在类外使用
private:
    static int j; // 只能在类成员函数内或友元使用
};
```

static 变量的初始化

- 静态成员一般在类的定义(.h)之外，也就是类的成员函数实现的文件(.cpp)中初始化：

type classname::static_member = value;

```
// A.h
class A {
    static int i;     // 声明时前加 static，但初始化时不加
};

// A.cpp
int A::i=1;           // 静态数据成员由类的构造者创建和初始化。

// main.cpp
void main() {
     A obja;          //创建A的对象前，其静态成员应已初始化
}
```

static 常量的初始化

- 静态常量兼具静态成员和常量成员的特征：所有类的对象所共有，初始化后不可改变。
- 静态常量的有序类型 (如int, char, float, double) 可以在类的定义处初始化，其它类型 (如数组) 需在定义之外初始化。

```
// A.h
class A {
    static const int i = 10;
    static const double d = 0.5;
    static const char name[10];
};

// A.cpp
const char A::name[10] = "Alice";
```

static 数据成员（例）

```
class A{
public:
      static int i; // 公有静态成员可以外部访问 A::i
      void f();
private:
      static int j; // 私有静态成员只能内部或友元访问
};
void A::f() {
      // 类成员函数可以直接访问公有或私有静态成员
      i = 0; this->i = 1; A::i = 2;  j = 0; this->j = 1; A::j = 2;
}
void main() {
      A::i = 3; A a; a.i = 4; // 外部函数只能访问公有静态成员
}
```

static 数据成员（例）

```
class A{
private:
      static A a;              // OK，静态成员的类型可以是自身
      A aa;                    // 错误，普通成员的类型不能是自身，因为会引起
                               // 构造函数的无限递归调用。
      A *aptr; A &aref;        // OK，只能是自身类型的指针或引用

      static int stdvar;
      int var;
public:
      void f1(int =stdvar);    // OK，静态成员可以作为默认参数
      void f2(int =var);       // 错误，普通成员则不可以，因为所有
        // 的类对象共用成员函数，对象生成前其成员未初始化
};
```

static 成员函数

- 为类的全体对象服务而不是类的特殊对象服务的成员函数，可以用类域名调用。
- 类的静态成员函数没有隐含的this指针，不能访问一般的数据成员和成员函数，只能访问静态数据成员和静态成员函数。

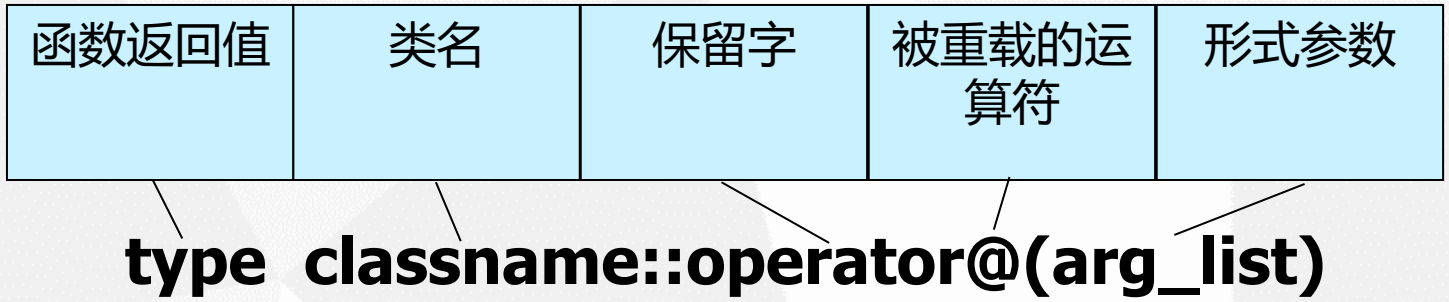
```
class X {
    int i; void g();
public:
    static void f(); // 声明时函数前加 static，实现时不加
};
void X::f() { i = 0; g(); } // 错误，不能调用一般成员或函数
void main() {
    X::f(); X x; x.f(); // 可以用类域名直接调用
}
```

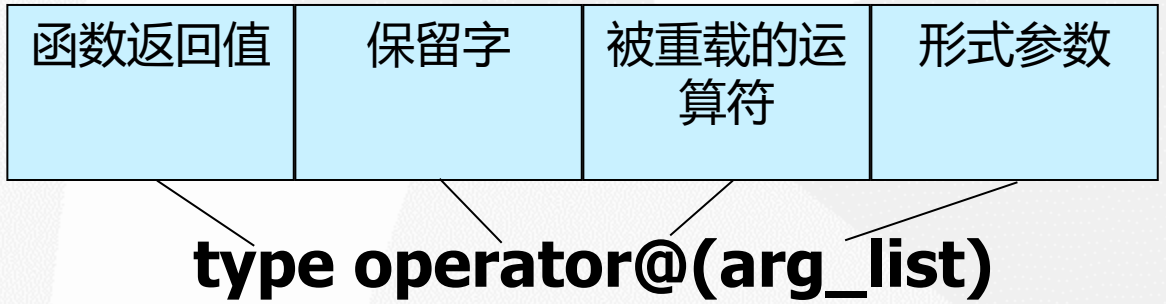
static 成员函数（例）

```
class Ctest{
private:
        static int count;
        static Ctest instance;
        Ctest() { ++count; }
public:
        ~Ctest(){ --count; }
        static int getCount(){ return count; }
        static Ctest& getInstance() { return instance; }
};

int Ctest::count = 0;
Ctest Ctest::instance;

Ctest &test = Ctest::getInstance();
int (*psf)() = &Ctest::getCount;        // 可以获取静态函数指针
cout << psf() << endl;                  // 效果等同于 Ctest::getCount()
```

运算符重载

- 运算符重载是一个按特殊格式（operator@）声明的函数，从而改变了内部运算符@的含义。





运算符重载 (例: <<)

```
int i = 10;
i << 2;     // 这个<<是左移运算符;

cout << "hello world";     // 这个"<<"是什么?

// 原型定义( From ostream.h):
inline  ostream& ostream::operator<<(const unsigned char * _s)  {
  return operator<<((const char *) _s);
}

// 解释: cout<<"hello world"; 等价于调用运算符函数
// cout.operator<<("hello world");
```

运算符重载 (例: +)

- 复数相“加”(特殊成员函数/重载“+”实现)

```
class Complex{
private:
    double rpart, ipart;
public:
    Complex():rpart(0.0),ipart(0.0) {}
    Complex(double rp, double ip):rpart(rp),ipart(ip)  {}
    Complex operator+(const Complex & c) {
        Complex t; t.rpart = c.rpart+rpart; t.ipart = c.ipart+ipart;
        return t;
    }
    // 类似的可以重载-, *, / 操作，实现复数的减法、乘法和除法
};
void main() {
    Complex C1(2.0,3.0), C2(5.0,4.0), C3;
    C3 = C2 + C1; // C3 = C2 + C1 ⇔ C3 = C2.operator+(C1)
}
```

运算符重载 (例: +)

- 复数相“加”(重载“+”实现/外部函数)

```
class Complex{
  friend Complex operator+(const Complex &c1, const Complex &c2);
private:
      double rpart, ipart;
public:
      Complex():rpart(0.0),ipart(0.0) {}
      Complex(double rp,double ip):rpart(rp),ipart(ip)  {}
};
Complex operator+(const Complex & c1,const Complex &c2){
      Complex t; t.rpart = c1.rpart+c2.rpart; t.ipart = c1.ipart+c2.ipart;
      return t;
}
// 类似的可以重载-, *, / 操作，实现复数的减法、乘法和除法
void main() {
    Complex C1(2.0,3.0), C2(5.0,4.0), C3;
    C3 = C2 + C1; // C3 = C2 + C1 ⇔ C3 = operator+(C2, C1)
}
```

运算符重载说明

- 运算符重载只是一种“语法上的方便”(语法糖)；实质上是另一种函数调用的方式。

$$C1 + C2 \Leftrightarrow C1.operator+(C2)$$

- 参数列表中的参数个数取决于：
  - 运算符是一元的还是二元的；
  - 被定义为全局函数还是成员函数；

$$C1 + C2 \Leftrightarrow C1.operator+(C2)$$

$$C1 + C2 \Leftrightarrow operator+(C1, C2)$$

  - 对单目后缀运算符，需传递一个int常量(哑元常量值)

$$C1{+}{+} \Leftrightarrow C1.operator{+}{+}(1)$$

$${+}{+}C1 \Leftrightarrow C1.operator{+}{+}( )$$

运算符重载 (例: ++)

- 重载++运算符（对应的可以重载--运算符）

```
class Integer {
      int i ;
public:
      Integer(int ii):i(ii) {}
      Integer operator++() {              // 前缀++
            i++; return *this;
      }
      Integer operator++(int) {           // 后缀++ ,添加一个哑元
            Integer  temp = *this;        // 调用默认的拷贝构造函数
            i++; return temp;
      }
};
void main() {
     Integer  a(1), b(2);
     b = a++;  b = ++a;
}
```

自定义类型转换

- 问题：如何实现 Complex+double 或 double+Complex?
- 方法一：
  - 内部函数： Complex Complex:: operator+(double);
  - 外部函数： Complex operator+(double,Complex)
- 方法二：
  - 转换构造函数：
  - T-> X 转换,如： double -> Complex
  - 则定义函数：

    Complex Complex::Complex(const double)

转换构造函数

- 定义 double -> Complex的转换,实现: double + Complex, Complex + double, Complex + Complex:

```
class Complex{
      friend Complex operator+(const Complex &, const Complex &);
public:
      Complex(const double &d):rpart(d), ipart(0.0) {}
       …
};
Complex operator+(const Complex & c1,const Complex &c2){
       Complex t; t.rpart = c1.rpart+c2.rpart; t.ipart = c1.ipart+c2.ipart;
       return t;
}                                    // 例：3.0+Complex ， 将3.0转换成临时的复数对象。
```

重载转换运算符

- 定义转换运算符函数 X::operator T(){ …; }; 实现类 X -> T (通常是内部类型)的转换。
- 如： Complex -> double (或其它类型)

```
class Complex{
public:
      operator double();
      …
};
double Complex::operator double(){
      return rpart;
}
void main() {
     Complex c1(5.5 , 1.1); double     temp=1.0;
     temp = temp + c1;    // ⇔ temp = temp+double(c1);
}
```

重载二义性问题

- 若同时定义了转换运算符和转换构造函数，会引起二义性，导致编译器提示错误。

```
class X{
    friend X operator + (const X &, const X &);
private:
    int i;
public:
    X(int ii):i(ii) {}                  // 转换构造函数
    operator int(){ return i; }         // 转换运算符
};
X operator+(const X &, const X ) { return X(x1.i + x2.i); }
void main( ) {
    X x1(10); int x2 = 20;
    int i = x1 + x2;                    // 二义性：x1->int 还是 x2->X ？
    i = 12 + x1;                        // 二义性：x1->int 还是 12->X ？
}
```

重载二义性问题

- 用关键字 explicit （只能用于构造函数），来阻止构造函数自动转换（必须显式调用）。

```
class X{
   friend X operator + (const X &, const X &);
private:
   int i;
public:
    explicit X(int ii):i(ii) {}        // 转换构造函数
    operator int(){ return i; }        // 转换运算符
};
X operator+(const X &x1, const X &x2) { return X(x1.i+x2.i); }
void main( ) {
    X x1(10); int x2 = 20;
    int i = x1 + x2;                   // 消除二义性：x1->int
    i = X(12) + x1;                    // 消除二义性：12->X
}
```

运算符重载 (例: [])

- 重载[]实现关联数组访问，更加直观、方便

```
class Complex{
private:
    double rpart, ipart;
public:
    Complex(double rp=0.0,double ip=0.0):rpart(rp),ipart(ip)  {}
    double& operator[](const char *part) { // 参数可以是int型
          switch(part) {
          case "rpart": return rpart;
          case "ipart": return ipart;
          }
    }  // 返回类型前加 const，实现"只读"，防止内部变量被修改
};
void main() {
    Complex C1(2.0,3.0); C1["rpart"] = 1.0; C1["ipart"] = 4.0;
    cout << C1["rpart"] << "+" << C1["ipart"] << "i" << endl;
}
```

运算符重载 (例: ->)

- 重载->操作符，实现指针类
  - 有额外开销，不像内置指针一样有效率，但有可能提供一些额外的功能，让指针变得更智能。

```
class ComplexPtr{
private:
    Complex *ptr;
public:
    Complex(Complex &c): ptr(&c) {}
    Complex& operator*() { return *ptr; }
    Complex* operator->() { return ptr; }
};
void main() {
    Complex C(2.0,3.0);
    Complex *ptr = &C;
    ComplexPtr Cptr(C);  // 与 ptr 一样可以调用Complex成员函数
}
```

运算符重载 (内存管理)

- 重载new/delete，实现自定义内存管理

```
class Complex {
public:
    void *operator new(size_t size) { …; }
    void operator delete(void* ptr, size_t size) { …; }
    void *operator new[](size_t size) { …; }
    void operator delete[](void* ptr, size_t size) { …; }
};

void main() {
    Complex *ptr1 = new Complex; delete ptr1;                  // 重载函数
    Complex *ptr2 = new Complex[10]; delete [] ptr2;           // 重载函数
    Complex *ptr3 = ::new Complex; ::delete ptr3;              // 默认操作符
    Complex *ptr4 = ::new Complex[10]; ::delete [] ptr4;       // 默认操作符
}
```

运算符重载 (例: =)

- 重载赋值运算符，实现利用=来对类对象赋值

```
class Complex{
private:
    double rpart, ipart;
public:
    Complex(double rp=0.0,double ip=0.0):rpart(rp),ipart(ip)  {}
    Complex(const Complex &c):rpart(c.rpart),ipart(c.rpart) {}
    Complex& operator=(const Complex & c) {
        if (this != &c) { rpart = c.rpart; ipart = c.ipart; }
        return *this; // 返回*this的引用目的是实现链式赋值
     } // c 也可以是其它类型，实现其它类型到复数的赋值（如果有意义）
};
void main() {
    Complex C1(2.0,3.0), C2;
    Complex C3 = C1;      // 调用拷贝构造函数
    C3 = C2 = C1;         // 调用赋值运算C3.operator=(C2.operator=(C1))
}
```

设计一个安全的类

- 一个安全的类通常须有几个特殊的成员函数：
  1. 构造函数：正确初始化对象；
  2. 析构函数：正确回收对象的空间；
  3. 拷贝构造函数：正确实现用对象去初始化对象；
  4. 重载赋值运算符：正确实现对象间的赋值。
- 基于安全性的考量，在一个自定义的类中通常需要显式的定义上述的成员函数。

设计一个安全的类

- 一个安全的复数类 Complex

```
class Complex{
private:
    double rpart, ipart;
public:
    // 1.（默认）构造函数
    Complex(double rp=0.0, double ip=0.0):rpart(rp),ipart(ip)  {}
    ~Complex() {} // 2. 析构函数
    // 3. 拷贝构造函数
    Complex(const Complex &c):rpart(c.rpart),ipart(c.rpart) {}
    // 4. 重载赋值运算符
    Complex& operator=(const Complex & c) {
        if (this != &c) { rpart = c.rpart; ipart = c.ipart; }
        return *this;
    }
}; // 除了析构函数，都可以定义为私有函数，防止被误用
```

运算符重载的限制

- 运算符重载不能改变内部算符的优先级、结合顺序及运算符的目数；

  C1+C2*C3 ⇔ C1+(C2*C3) ⇔ C1+C2.operator*(C3)
  ⇔ C1.operator+(C2.operator*(C3))

- . , ::, ?:, sizeof 五种运算符不能重载。
- 重载 =, [ ], ( ), -> 都必须是非静态的成员函数，目的是保证它们的第一个操作数是一个自定义对象，从而阻止改变这些运算符作用在基本数据类型上的含义的企图。
  - 如果 a[ ]; 调用的是重载的[ ]，则a一定是自定义类型的对象，而不是内置的基本数据类型。

小结：类的成员

- this 指针是一个隐含的常量指针，可以方便对类对象自身的成员变量和函数进行操作。
- const 能将对象、函数参数、返回值和成员函数定义为常量，还可以进行值替代，为程序设计提供了又一种非常好的类型检查形式及安全性。
- static 数据成员为类的对象之间提供了一种数据 共享的方式。静态成员函数是为类的全体对象服务而不是类的特殊对象服务。
- 运算符重载是实现程序简洁、直观的一种方法，几乎所有运算符都可以重载，但要搞清楚重载有什么益处，否则容易混淆。