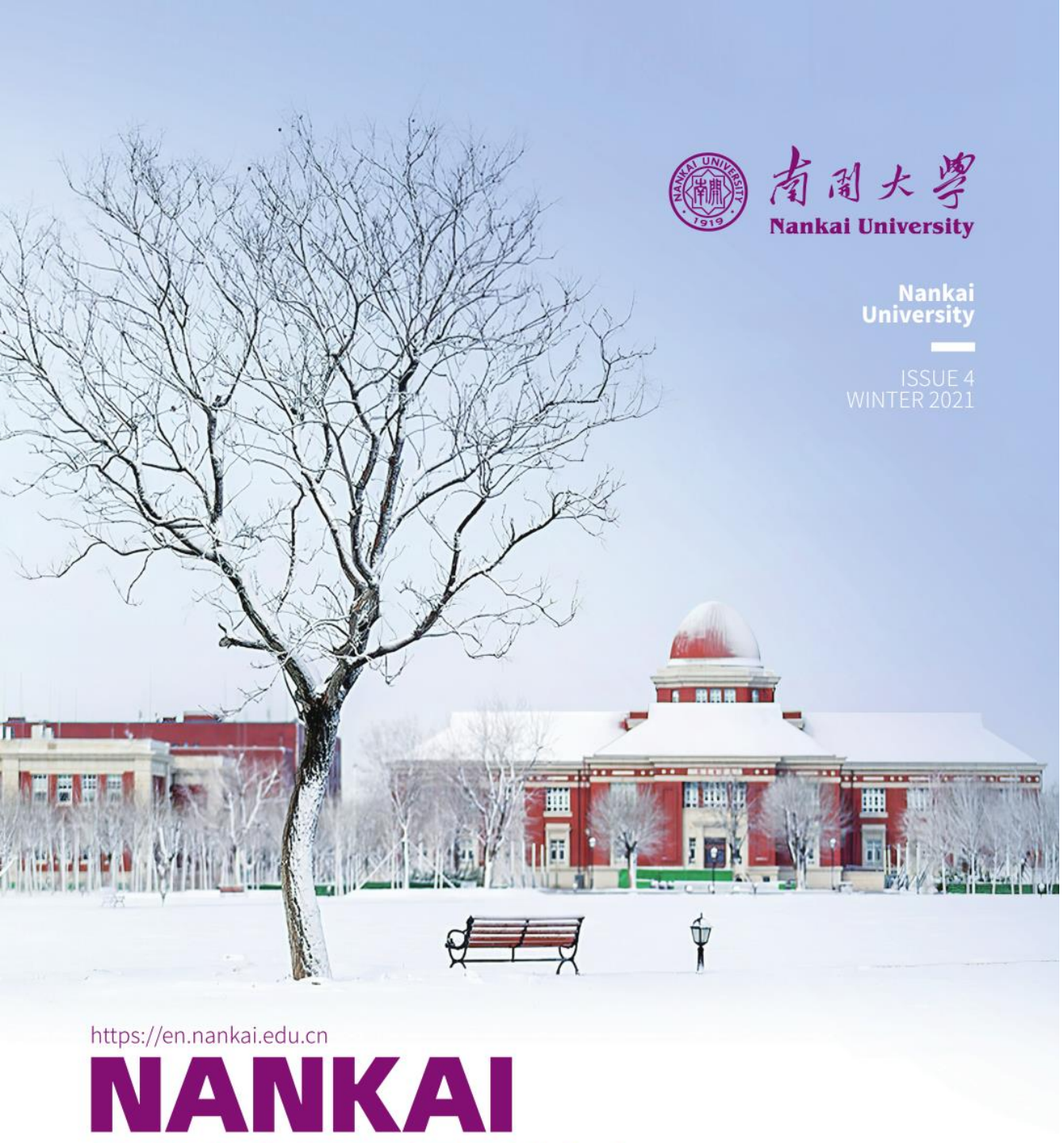

04-函数重载

龚成

cheng-gong@nankai.edu.cn

函数重载

函数重载定义

函数重载条件

函数重载优先级

函数重载类型

函数重载定义

- **对若干种不同的数据类型求和，虽然数据本身差别很大**（例如整数求和、向量求和、矩阵求和），具体的求和操作差别也很大，但**完成不同求和操作的函数却可以取相同的名字**（例如 sum、add 等）。
- 许多差别很大的打印函数可以都用同一个函数名 print，显示函数可以都用 display，从键盘或文件获取信息都称作 get，发送称为 send，接收称为 receive 等。

函数重载定义

- 函数重载是指在同一个作用域内，可以定义多个具有**相同名称**但**参数列表**（参数类型、参数个数或参数顺序）不同的函数。函数重载使得程序可以根据不同的参数列表选择合适的函数进行调用，从而提高代码的灵活性和可复用性。

```
int abs (int n) {return (n<0)? -n: n; }
float abs (float f ){if (f<0) f= -f;    return    f; }
double abs (double d) {if (d<0) return -d; return    d;}
```

- 函数名的重载并不是为了节省标识符（标识符的数量是足够的），而是为了方便程序员的使用。

函数重载条件

- 实现函数的重载必须满足下列条件之一：
  - 参数表中对应的参数类型不同。
  - 参数表中参数个数不同。
  - 参数表中不同类型参数的次序不同。
- 注意：返回类型不同是不能区分重载函数的。

```
void print (int);                    //整型
void print (point);                  //类 point 的对象
int sum (int，int);
int sum (int，int，int);
```

函数重载条件

- 在定义同名函数时应注意：
  - 返回类型不能区分函数

```
float add(int float);
int add(int float) ;                    //错误
```

  - 采用引用参数不能区分函数

```
void print (double);
void print (double &);                  //错误
void print (const double &);            //错误
```

  - 有些派生基本类型的参数虽然可以区分同名函数，但在使用中必须注意可能出现二义性。
  - 包含可缺省参数时，可能造成二义性，应该尽量避免。

```
int abs(int);                           int sum (int a，int b，int c=0);
unsigned int abs(unsigned int);         int sum (int a，int b);
```

函数重载优先级

- 二义性的情形是指系统面对某一函数调用语句，有两个或两个以上同名函数与之匹配，应该避免；
- 遇到无准确匹配的函数定义时，C++系统并不马上按出错处理，它按下面的方式处理：
  - 通过数组名与指针变量、函数名与函数指针、某类型变量与const 常量之间的转换，去查看是否可实现匹配。
  - 把实参类型从低到高（按字长由短到长）进行基本类型及其派生类型的转换，再检查是否可匹配。
  - 查有无已定义的可变个数参数的函数，如有把它归为该函数。
  - 在进行上述尝试性的处理之后可能出现仍无匹配或匹配不唯一的情况，这时可能输出错信息或错误地运行。
- 这种情况下，由于数据类型之间的复杂转换关系可能造成找不到与之匹配的函数定义。

函数重载类型

- 在C++中函数重载通常可以被划分为两个类型：
  - 全局函数重载
  - 类成员函数重载
- 全局函数重载意味着在整个程序中，该函数被说明为重载的，具有相同函数名不同参数列表的多个函数，程序运行时会根据按照优先级调用对应函数。
- 类成员函数则仅在类的范围内是可用的，仅能通过对象进行调用。
- 两者的调用方式不同。

运算符重载

可重载运算符

运算参数

运算符重载定义

运算符重载调用

运算符重载类型

可重载运算符

- C++语言中的运算符实际上是函数的方便表示形式，例如，算术运算符“+”也可以表示为函数形式：

```
int add(int a, int b) {
   return a+b;
}
```

这时，a+b和add(a, b)的含义是一样的。因此，逻辑上运算符也可以重载。

可重载运算符

- C++语言规定，大多数运算符都可以重载，可重载的运算符如下：

（1）单目运算符：

–、~、!、++、--、new、delete

（2）双目运算符：

+、–、*、/、%	（算术运算）
&、|、^、<<、>>	（位运算）
&&、||	（逻辑运算）
==、!=、<、<=、>、>=	（关系运算）
=	（赋值运算）
+=、–=、*=、/=、%=	（赋值运算）
^=、&=、|=、>>=、<<=	（赋值运算）
,	（逗号运算）
<<、>>	（I / O运算）
( )、[ ]	（其他）

可重载运算符

- 所列可重载运算符几乎包含了 C++的全部运算符集，例外的是：
  - 限定符 . 和：：。
  - 条件运算符 ?：。
  - 取长度运算符 sizeof。

它们不可重载（不可赋予不同的操作）。

运算参数

- 算术运算符、逻辑运算符、位运算符和关系运算符中的<、>以及<=、>=，这些运算都与基本数据类型有关，通过运算符重载函数的定义，使它们也用于某些用户定义的数据类型。
- 指针运算符&和* 、下标运算符［］等运算所涉及的数据类型按 C++程序规定，并非只限于基本数值类型。因此，这些运算符可以自动地扩展到任何用户定义的数据类型，不需作重载定义就可“自动”地实现重载。

运算参数

- 单目运算符++和--实际上各有两种用法：前缀增（减）量和后缀增（减）量。其运算符重载函数的定义当然是不同的，对两种不同的运算无法从重载函数的原型上予以区分：函数名（operator ++）和参数表完全一样。为了区别前缀++和后缀++，C++语言规定，在后缀++的重载函数的原型参数表中增加一个 int 型的无名参数。

```
前缀++：  <类型>operator   ++()                  //作为类成员
          <类型>operator   ++(<类型>)            //作为类外函数
后缀++：  <类型>operator   ++   (int)            //作为类成员
          <类型>operator   ++(<类型>，int)       //作为类外函数
```

```
前缀++：  ++a 或 a.operator++()
              operator++(a)
后缀++：  a++ 或 a.operator++(0)
              operator++(a，0)
```

运算符重载定义

- 运算符的重载是一个特殊函数定义过程，这类函数总是以“operator<运算符>”作为函数名。
- 假设程序中定义了一个枚举类型的 Bool类型：

```
enum Bool {FALSE, TRUE};
```

- 用运算符+(双目)、*（双目)、&(单目)来表示或、与、非运算是十分方便的：

```
Bool operator + (Bool a , Bool b) {
 if ( (a == FALSE) && (b == FALSE) ) return FALSE;
 return TRUE;
}
Bool operator * (Bool a, Bool b) {
 if ( (a==TRUE) && (b==TRUE) ) return TRUE;
 return FALSE;
}
```

```
Bool operator & (Bool a) {
 if (a==FALSE) return TRUE;
 return FALSE;
}
```

运算符重载调用

- 运算符重载函数的调用可有两种方式：
  - 与原运算符相同的调用方式，如上例中的 b1+b2、b1* b2 等。
  - 一般函数调用方式，如 b1+b2，也可以写为 operator+（b1，b2）。被重载的运算符的调用方式，**优先级和运算顺序都与原运算符一致，其运算分量的个数也不可改变**。
- 运算符重载主要用于同类的形式定义的用户定义类型，例如，复数类型、集合类型、向量类型等，通过运算符重载把人们习惯的运算符引入到计算操作中，会收到很好的效果。

```
b3 = b1+b2;
b3 = b1*b2;
b3 = &b1;
b3 = (b1+b2) * FALSE;
b3 = operator+ (b1,b2);
```

运算符重载类型

- 运算符重载和函数重载类似，通常可以划分为
  - 全局函数重载
  - 类成员函数重载
- 值得注意的是，有些运算符重载被限定为仅能通过类成员的方式进行重载，而不能进行全局函数重载，这些运算符包括：
  - 赋值运算符=： 赋值运算符是用于将一个对象的值赋给另一个对象。
  - 下标运算符[]： 下标运算符用于访问数组、容器或自定义类型的元素。
  - 函数调用运算符()： 函数调用运算符用于将对象作为函数调用。

类成员方式重载运算符

重载方式

重载示例

重载方式

- 在自定义类中可以通过两种方式对运算符进行重载：
  - 按照**类成员**方式；
  - 按照**友元**方式。
- 运算符重载的定义是一个函数定义过程 ，其函数名处为operator〈运算符〉。
- 当以**类的公有成员函数方式**来重载运算符（也称为类运算符）时，具有如下特征：
  - 类成员函数内（定义处）**可处理与使用本类的私有成员**。
  - **总以当前调用者对象（*this）作为该成员函数的隐式第一运算分量**，若所定义的运算多于一个运算对象时，才将其余运算对象显式地列在该成员函数的参数表中。
- 当以**类的友元函数方式**来重载运算符(也称为友元运算符)时，具有如下特征：
  - 友元函数内（定义处）**可处理与使用本类的私有成员**。
  - **所有运算分量必须显式地列在本友元函数的参数表中（由于友元函数中没有 this 指 针）**，而且这些参数类型中至少要有一个是说明该友元的类或是对该类的引用。

重载方式

- 一般地说，**单目运算符重载常选用成员函数方式，而双目运算符重载常选用友元函数方式**。但不管选用哪种重载方式，对重载运算符的使用方法都是相同的。
- 注意，**被用户重定义（重载）的运算符，其优先级、运算顺序（结合性）以及运算分量个数都必须与系统中的原运算符相一致**，而且不可自创新的运算符。

```
class set {
        int elems [maxcard];      //集合各元素放于 elems 中
        int card;                 //集合中的实际元素个数 card
public:
        ...
        friend bool operator & (int, set);
        set operator + (set S2);
        ...
};
```

重载示例

- **类成员方式**重载示例
  - 自定义一个称为 point 的类，其对象表示平面上的一个点（x，y），并通过友员方式对该类重载二目运算符“+”和“^”，用来求出两个对象的和以及两个对象（平面点）的距离。各运算符的使用含义（运算结果）如下所示：

(1.2, -3.5) + (-1.5, 6) = (-0.3, 2.5);

(1.2, -3.5) ^ (-1.5, 6) = 9.87623。

```
class point {
        double x,y;
public:
        point (double x0=0, double y0=0) {x=x0; y=y0;}
        point operator + (point pt2);
        double operator ^ (point pt2);
        void display();
};
void point::display () {
        cout <<"( "<<x<<", "<<y<<" )"<<endl;
}
```

```
void main() {
        point s0, s1(1.2, -3.5), s2(–1.5, 6);
        cout<<"s0=";   s0.display();
        cout<<"s1=";   s1.display();
        cout<<"s2=";   s2.display();
        s0=s1+s2;              //两对象的“ +”运算，将调用“ operator +”函数
        cout<<"s0=s1+s2=";    s0.display();
        cout<<"s1^s2="<<(s1^s2)<<endl;       //将调用“ operator ^”函数
}
```

重载示例

- **友员方式重载**示例
  - 自定义一个称为 point 的类，其对象表示平面上的一个点（x，y），并通过友员方式对该类重载二目运算符“+”和“^”，用来求出两个对象的和以及两个对象（平面点）的距离。各运算符的使用含义（运算结果）如下所示：

(1.2, -3.5) + (-1.5, 6) = (-0.3, 2.5);

(1.2, -3.5) ^ (-1.5, 6) = 9.87623。

```
class point {
    double x,y;
public:
    point (double x0=0, double y0=0) {x=x0; y=y0;}
    friend point operator + (point pt1, point pt2);
    friend double operator ^ (point pt1, point pt2);
    void display();
};
```

```
point operator + (point pt1, point pt2) {
    //二目运算符“ +”，求出两个对象的和
    point temp;
    temp.x=pt1.x+pt2.x;
    temp.y=pt1.y+pt2.y;
    return temp;
}
```

```
double operator ^ (point pt1, point pt2) {
    double d1, d2, d;
    d1=pt1.x-pt2.x;
    d2=pt1.y-pt2.y;
    d=sqrt(d1*d1+d2*d2);
    return (d);
}
```

重载示例

- 实现时通过“借用”一批老运算符（如，*、+、 - 、&、>=、> 等）来表示集合set 类对象的交、并、差、元素属于、包含、真包含等运算，各算符的具体运算含义必须首先在 set类中通过运算符重载函数由用户进行说明或自定义。

```
class set {
        int elems [maxcard];
        int card;
public:
        friend bool operator == (set, set);        // == : 判断两个集合是否相同
        friend bool operator != (set, set);        // != : 判断两个集合是否不相同
        friend set operator + (set, int);          // + : 将某元素加入到某集合中
        friend set operator + (set, set);          // + : 求两个集合的并集合
        friend set operator - (set, int);          // - : 将某元素从某集合中删去
        friend set operator * (set, set);          // * : 求两个集合的交集合
        friend bool operator < (set, set);         // < : 集合 1 是否真包含于集合 2 之中
        friend bool operator <= (set, set);        // <= : 集合 1 是否包含于集合 2 之中
};
```

其他重载

类型转换运算符重载

类构造函数重载

类成员函数重载

模板函数重载

类型转换运算符重载

- 类型转换符指的不同类型之间的转换函数，一般而言函数名就是类型名，比如：

```
int b = int(a);
```

- C++内置了基本类型的转换函数，包括int、float、bool、char等，但是注意这种转换不一定是安全的，可能会出现数据丢失，不能通过全局函数重载。

类型转换运算符重载

- 在自定义类型中，比如class中，可以分别通过构造函数（类成员函数）和类型转换运算符符重载（类成员函数）来实现任意类型与当前类型的互相转换：

```
class Feet {
private:
  double value;
public:
      // 转换构造函数（double 到 Feet）
      Feet(double val) : value(val) {}
      // 类型转换运算符重载（Feet 到 double）
      operator double() const {
         return value;
      }
};
```

类构造函数重载

- 类的构造函数都是重载函数

```
class TypeA
{
        int a;
        TypeA(int a):a(a){}
        TypeA(const TypeA& a):a(a.a){}
}
```

类成员函数重载

- 类的成员函数重载不同于全局函数重载

```
class TypeA
{
        int a;
        TypeA(int a):a(a){}
        TypeA(const TypeA& a):a(a.a){}
        TypeA& operator=(const TypeA& a){
                this->a = a.a;
                return *this;
        }
        void overloading(int b){
           count << "overloading(int b)" << endl;
        }
        void overloading(float b){
           count << "overloading(float b)" << endl;
        }
}
```

模板函数重载

- 模板函数重载包括全局模板函数与类成员模板函数的重载。

```
template <class T>
        add(T a, T b){
        return a+b;
}
int add(int a, float b){
    return a+b;
}
```

思考

- 什么是函数重载？使用它有什么好处？实现函数的重载必须满足什么条件？
- 什么是运算符的重载？各种不同的运算符是怎样进行重载的？运算符的重载与函数定义过程有什么联系？

Matrix的索引访问

```
class Matrix {
private:
    int rows;
    int cols;
    T* data;
    // 构造函数
public:
    Matrix(int m, int n) : rows(m), cols(n) {
        data = new T[rows * cols];
    }
    // 析构函数
    ~Matrix() {
        delete[] data;
    }
    // 重载()获取元素
    T& operator()(int i, int j) {
        return data[i * cols + j];
    }
}
```