

重载运算与类型转换



C++ Primer第五版



第14章

重载的运算符是具有特殊名字的函数

```
cout<<item1 + item2;

print(cout, add(data1,data2));
```



一个运算符函数，它或是类的成员，或者至少含有一个类类型的参数：

```
//错误：不能为int重定义内置的运算符
int operator+(int, int);
```

运算符					
可以被重载的运算符					
+	-	*	/	%	^
&		~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	
+=	--	/=	%=	^=	&=
=	*=	<<=	>>=	[]	()
->	->*	new	new[]	delete	delete[]
不能被重载的运算符					
	::	.*	.	? :	

对于一个重载的运算符来说，其优先级和结合律与对应的内置运算符保持一致

直接调用一个重载的运算符函数

```
//一个非成员运算符函数的等价调用
data1 + data2;           //普通的表达式
operator+ (data1, data2); //等价的函数调用

data1 += data2;           //基于“调用”的表达式
data1.operator+=(data2); //对成员运算符函数的等价调用
```

- 通常情况下，不应该重载逗号、取地址符、逻辑与和逻辑或运算符
- 使用与内置类型一致的含义

选择作为成员或非成员

运算符定义为成员函数：左侧运算对象必须是运算符所属类的一个对象

```
//如果operator+是string类的成员：
```

```
string s = "world";
```

```
string t = s + "!"; //正确,s.operator+("!")。
```

```
string u = "hi" + s; //如果+是string的成员，则产生错误
```

```
//等价于"hi".operator+(s)
```

```
//const char*是一种内置类型，没有成员函数
```

```
//因为string将+定义成了普通的非成员函数，所以"hi"+s等价于operator+ (
```

```
"hi" s)
```

具有对称性的运算符可能转换任意一端的运算对象，通常应该是普通的非成员函数

重载输出运算符<<

```
ostream &operator<<(ostream &os, const Sales_data &item)
{
    os << item.isbn() <<" " <<item.units_sold <<" "
        <<item.revenue<<" " <<item.avg_price();
    return os;
}
```

输入输出运算符必须是非成员函数

```
Sales_data data;
data << cout; //如果operator<<是Sales_data的成员
```

重载输入运算符>>

```
istream &operator>> (istream &is, Sales_data &item)
{
    double price; //不需要初始化，先读入数据才会使用到
    is >> item.bookNo >> item.units_sold >> price;
    if(is) //检查输入是否成功
        item.revenue = item.units_sold * price;
    else
        item = Sales_data(); //输入失败：对象被赋予默认的状态
    return is;
}
```

输入运算符必须处理输入可能失败的情况，输出运算符不需要

通常情况下，把算术和关系运算符定义成非成员函数

```
//假设两个对象指向同一本书
Sales_data
operator+(const Sales_data &lhs, const Sales_data &rhs)
{
    Sales_data sum = lhs; //把lhs的数据成员拷贝给sum
    sum += rhs;           //把rhs加到sum中
    return sum;
}
```

如果类同时定义了算术运算符和相关的复合赋值运算符，则通常情况下应该使用复合赋值来实现算术运算符

```
bool operator==(const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.isbn() == rhs.isbn() &&
        lhs.units_sold == rhs.units_sold &&
        lhs.revenue == rhs.revenue;
}

bool operator!=(const Sales_data &lhs, const Sales_data &rhs)
{
    return !(lhs == rhs);
}
```


如果类包含==，则当且仅当<的定义和==产生的结果一致时才定义<运算符

不论型参的类型是什么，赋值运算符都必须定义为成员函数

```
vector<string> v;
v = {"a", "an", "the"};

class StrVec{
public:
    StrVec &operator=(std::initializer_list<std::string>);
    //... ..
}

StrVec &StrVec::operator=(std::initializer_list<std::string> il)
{
    //alloc_n_copy分配内存空间并从给定范围内拷贝元素
    auto data = alloc_n_copy(il.begin(), il.end());
    free(); //销毁对象中的元素并释放内存空间
    elements = data.first;
    first_free = cap = data.second;
    return *this;
}
```



复合赋值运算符通常情况下也应该定义为类的成员

```
//作为成员的二元运算符：左侧运算对象绑定到隐式的this指针
Sales& Sales_data::operator+=(const Sales_data &rhs)
{
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}
```

下标运算符必须是成员函数

```
class StrVec{
public:
    std::string& operator[](std::size_t n)
        {return elements[n];}
    const std::string& operator[](std::size_t n) const
        {return elements[n];}
    //...
private:
    std::string *elements; //指向数组首元素的指针
};
```

通过this是否指向常量进行匹配

```
//假设svec是一个StrVec对象
const StrVec cvec = svec; //把svec的元素拷贝到cvec中
//如果svec中含有元素，对第一个元素运行string的empty函数
if(svec.size() && svec[0].empty()){
    svec[0]="zero"; //正确
    cvec[0]="Zip"; //错误
}
```

应该定义前置和后置版本，通常被定义为类的成员

```
class StrBlobPtr{
public:
    //递增和递减运算符
    StrBlobPtr& operator++(); //前置运算符
    StrBlobPtr& operator--();
    //其他...
};

StrBlobPtr& StrBlobPtr::operator++(){//前置版本：返回递增/递减对象的引用
    //如果curr已经指向了容器的尾后位置，则无法递增它
    check(curr,"increment past end of StrBlobPtr");
    ++curr; //将curr在当前状态下向前移动一个元素
    return *this;
}

StrBlobPtr& StrBlobPtr::operator--(){
    //如果curr是0，则继续递减它将参数一个无效下标
    --curr; //将curr在当前状态下向后移动一个元素
    check(curr,"decrement past begin of StrBlobPtr");
    return *this;
}
```

后置版本为了区分，接受一额外的（不被使用）int类型的参数
编译器会为这个形参提供一个值为0的实参

```
class StrBlobPtr{
public:
    //递增和递减运算符
    StrBlobPtr operator++(int); //后置运算符
    StrBlobPtr operator--(int);
    //其他成员和之前的版本一致
};

//后置版本：递增/递减对象的值但是返回原值
StrBlobPtr StrBlobPtr::operator++(int){
    //此处无须检查有效性，调用前置递增运算符时才需要检查
    StrBlobPtr ret = *this; //记录当前的值
    ++*this; //向前移动一个元素，前置++需要检查递增的有效性
    return ret; //返回之前记录的状态
}

StrBlobPtr StrBlobPtr::operator--(int){
    //此处无须检查有效性，调用前置递减运算符时才需要检查
    StrBlobPtr ret = *this; //记录当前的值
    --*this; //向后移动一个元素，前置--需要检查递减的有效性
    return ret; //返回之前记录的状态
}
```

```
p++;
p.operator++(0);
++p;
p.operator++();
```


解引用运算符（*）和箭头运算符（->）

```
class StrBlobPtr{
public:
    std::string& operator*() const
    {
        auto p = check(curr, "deference past end");
        return (*p)[curr]; //( *p)是对象所指的vector
    }

    std::string* operator->() const
    {
        //实际工作委托给解引用运算符
        return & this->operator*();
    }
    //...
}
```

箭头运算符必须是类的成员，解引用运算符通常也是类的成员

这两个运算符的用法与指针或者vector迭代器的对应操作完全一致：

```
StrBlob a1 = {"hi", "bye", "now"};
StrBlobPtr p(a1); //p指向a1中的vector
*p = "okay"; //给a1的首元素赋值
cout<<p->size()<<endl; //打印4，这是a1首元素的大小
cout<<(*p).size()<<endl; //等价于p->size()
```

对箭头运算符返回值的限定

```
//根据类型的不同，point->mem分别等价于
(*point).mem; //point是一个内置的指针类型
point.operator->()->mem; //point是类的一个对象
```

重载的箭头运算符必须返回类的指针 or 自定义了箭头运算符的某个类的对象

像使用函数一样使用类的对象（函数对象）

```
struct absInt{
    int operator()(int val) const{
        return val<0?-val:val;
    }
};
```

函数调用运算符必须是成员函数。可以定义多个版本。

```
int i = -42;
absInt absObj;    //含有函数调用运算符的对象
int ui = absObj(i); //将i传递给absObj.operator()
```

函数对象类通常含有一些数据成员，用于定制调用运算符中的操作

```
class PrintString{
public:
    PrintString(ostream &o = cout, char c=' '): os(o),sep(c) {}
    void operator()(const string &s) const { os<<s<<sep; }
private:
    ostream &os;           //用于写入的目的流
    char sep;              //用于将不同输出隔开的字符
};

PrintString printer;       //使用默认值，打印到cout
printer(s);                //在cout中打印s，后面跟一个空格
PrintString errors(cerr, '\n');
errors(s);                 //在cerr中打印s，后面跟一个换行符

//函数对象常常作为泛型算法的实参
for_each(vs.begin(), vs.end(), PrintString(cerr, '\n'));
```

lamdba是函数对象

```
//根据单词长度进行排序
stable_sort(words.begin(), words.end(),
    [](const string &a, const string &b) { return a.size() < b.size(); });
//其行为类似于下面这个类的一个未命名对象
class ShorterString{
public:
    bool operator()(const &s1, const string &s2) const
    { return s1.size() < s2.size(); }
};

stable_sort(words.begin(), words.end(), ShorterString());
```

默认情况下，lambda不能改变它捕获的变量，因此函数调用运算符是一个const成员函数。如果lamdba被声明为可变的，则调用运算符就不是const的了

表示lambda及相应捕获行为的类

```
//获得第一个指向满足条件元素的迭代器，该元素满足size() is>= sz
auto wc = find_if( words.begin(), words.end(),
    [sz](const string &a) { return a.size() >= sz; });

//该lambda表达式产生的类将形如：
class SizeComp{
    SizeComp(size_t n):sz(n) { } //该形参对应捕获的变量
    //该调用运算符的返回类型、形参和函数体都与lambda一致
    bool operator() (const string &s) const { return s.size() >= sz; }
private:
    size_t sz; //该数据成员对应通过值捕获的变量
};

auto wc = find_if(words.begin(), words.end(), SizeComp(sz));
```

标准库函数对象		
算术	关系	逻辑
plus<Type>	equal_to<Type>	logical_and<Type>
minus<Type>	not_equal_to<Type>	logical_or<Type>
multiplies<Type>	greater<Type>	logical_not<Type>
divides<Type>	greater_equal<Type>	
modulus<Type>	less<Type>	
negate<Type>	less_equal<Type>	

```
plus<int> intAdd; //可执行int加法的函数对象
negate<int> intNegate; //可执行int值取反的函数对象

int sum = intAdd(10,20); //使用intAdd::operator(int,int)求10和20的和
sum = intNegate(intAdd(10,20)); //使用intNegate::operator(int)

sum = intAdd(10, intNegate(10)); //sum=0;
```

在算法中使用标准库函数对象

```
//传入一个零时的函数对象用于执行两个string对象的>比较运算
//默认是使用<
sort(svec.begin(),svec.end(),greater<string>());

vector<string *> nameTable; //指针的vector
//错误：nameTable中的指针彼此之间没有关系，所以<将产生未定义的行为
sort(nameTable.begin(),nameTable.end(),[](string *a, string *b) { return a<b; });
//正确：标准库规定指针的less是定义良好的
sort(nameTable.begin(), nameTable.end(), less<string*>());
```

可调用对象与function，不同类型可能具有相同的调用形式

```
//普通函数
int add(int i, int j) {return i+j;}
//lambda,其产生一个未命名的函数对象类
auto mod = [](int i, int j) { return i%j; };
//函数对象类
struct divide{
    int operator() (int denominator, int divisor) {
        return denominator/divisor;
    }
};
//调用形式（call signature）都是int(int,int)
```

为了构建一个简单的桌面计算器，需要一个函数表

```
//构建从运算符到函数指针的映射关系
map<string, int (*)(int,int)> binops; //二元运算
//正确：add是一个指向正确类型函数的指针
binops.insert({"+",add}); //{"+",add}是一个pair

binops.insert({"%",mod}); //错误：mod不是一个函数指针
```

标准库function的操作

<code>function<T> f;</code>	<code>f</code> 是一个用来存储可调用对象的空 <code>function</code> ，这些可调用对象的调用形式应该与函数类型 <code>T</code> 相同（即 <code>T</code> 是 <code>retType(args)</code> ）
<code>function<T> f(nullptr);</code>	显式地构造一个空 <code>function</code>
<code>function<T> f(obj);</code>	在 <code>f</code> 中存储可调用对象 <code>obj</code> 的副本
<code>f</code>	将 <code>f</code> 作为条件：当 <code>f</code> 含有一个可调用对象时为真；否则为假
<code>f(args)</code>	调用 <code>f</code> 中的对象，参数是 <code>args</code>
定义为 <code>function<T></code> 的成员的类型	
<code>result_type</code>	该 <code>function</code> 类型的可调用对象返回的类型
<code>argument_type</code>	当 <code>T</code> 有一个或两个实参时定义的类型。如果 <code>T</code> 只有一个实参，则 <code>argument_type</code> 是该类型的同义词；如果 <code>T</code> 有两个实参，
<code>first_argument_type</code>	则 <code>first_argument_type</code> 和 <code>second_argument_type</code>
<code>second_argument_type</code>	分别代表两个实参的类型

```
function<int(int, int)> f1 = add; //函数指针
function<int(int, int)> f2 = divide(); //函数对象类的对象
function<int(int, int)> f3 = [](int i, int j){return i*j;};
cout<<f1(4,2)<<endl;
cout<<f2(4,2)<<endl;
cout<<f3(4,2)<<endl;

map<string, function<int(int, int)>> binops = { //重新定义map
    {"+",add}, //函数指针
    {"-",std::minus<int>()}, //标准库函数对象
    {"/",divide()}, //用户定义的函数对象
    {"*",[](int i,int j){return i*j;}}, //未命名的lambda
    {"%",mod} }; //命名了的lambda

binops["+"](10,5); //调用add(10,5)
binops["-"](10,5);
binops["/"](10,5);
binops["*"](10,5);
binops["%"](10,5);
```


不能直接将重载函数的名字存入function类型的对象中

```
int add(int i, int j) { return i + j; }  
Sales_data add(const Sales_data&, const Sales_data&);  
map<string, function<int(int, int)>> binops;  
binops.insert({"+", add}); //错误：哪个add？
```

//解决二义性问题的一个途径是存储函数指针

```
int (*fp)(int, int) = add;  
binops.insert({"+", fp});  
//也可以使用lambda来消除二义性  
binops.insert({"+", [](int a, int b){return add(a,b);}});
```

类型转换运算符是特殊成员函数，它负责将一个类类型的值转换成其他类型：
一般形式为：`operator type() const`;

//定义一个比较简单的类，令其表示0到255之间的一个整数

```
class SmallInt{
public:
    SmallInt(int i = 0): val(i)
    {
        if(i<0 || i>255)
            throw std::out_of_range("Bad SmallInt value");
    }
    operator int() const { return val; }
private:
    std::size_t val;
};
```

不能声明返回类型，
形参列表页必须为空

SmallInt si;

si = 4; //首先将4隐式地转换成SmallInt，然后调用SmallInt::operator=
si + 3; //首先将si隐式地转换成int，然后执行整数的加法

//内置类型转换将double实参转成int

SmallInt si = 3.14; //调用SmallInt(int)构造函数

//SmallInt的类型转换运算符将si转换成int

si + 3.14; //内置类型转换将所得的int继续转换成double

类型转换运算符是隐式执行的，不能传递实参：

```
class SmallInt;
operator int(SmallInt&); //错误：不是成员函数
```

```
class SmallInt{
public:
    int operator int() const;           //错误：指定了返回类型
    operator int(int = 0) const;        //错误：参数列表不为空
    operator int*() const { return 42; } //错误：42不是一个指针
};
```

类型转换运算符可能产生意外结果

//当istream含有向bool的类中转换时

```
int i = 42;
```

```
cin << i; //提升后的bool值（1或0）最终被左移42个位置
```

类型转换运算符可能产生意外结果

```
//当istream含有向bool的类中转换时
int i = 42;
cin << i; //提升后的bool值（1或0）最终被左移42个位置
```

为了防止上面的异常，C++ 11引入了 **显式的类型转换运算符**

```
class SmallInt{
public:
    //编译器不会自动指向这一类型转换
    explicit operator int() const { return val; }
    //其他成员与之前的版本一致
};

SmallInt si = 3; //正确：SmallInt的构造函数不是显式的
si + 3; //错误：此处需要隐式的类型转换，但类的运算符是显式的
static_cast<int>(si) + 3; //正确：显式地请求类型转换
```

也有例外，如果表达式被用作条件，编译器会将显式的类型转换自动应用于它（被隐式地执行）：

- if、while及do语句的条件部分
- for语句头的条件表达式
- !、||、&&的运算对象
- ?: 的条件表达式

向bool的类型转换通常在条件部分，因此operator bool一般定义为explicit的

```
while(std::cin >> value)
```

避免有二义性的类型转换

```
//最好不要再两个类之间构建相同的类型转换
struct B;
struct A {
    A() = default;
    A(const B&); //把一个B转换成A
    //其他数据成员
};

struct B{
    operator A() const; //也是把一个B转换成A
    //其他数据成员
};

A f(const A&);
B b;
A a = f(b); //二义性错误：含义是f(B::operator A())，还是f(A::A(const B&))？
A a1 = f(b.operator A()); //正确
A a2 = f(A(b)); //正确
```


重载运算与类型转换

 C++ Primer第五版

 第14章

重载的运算符是具有特殊名字的函数

```
cout<<item1 + item2;

print(cout, add(add1,data2));
```

一个运算符函数，它或是类的成员，或者至少含有一个类类型的参数：

```
//错误：不能为int重定义内置的运算符
int operator+(int, int);
```

运算符					
可以被重载的运算符					
+	-	*	/	%	^
&		~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	
+=	--	/=	%=	^=	&=
=	*=	<<=	>>=	[]	()
->	->*	new	new[]	delete	delete[]
不能被重载的运算符					
	::	.*	.	? :	

对于一个重载的运算符来说，其优先级和结合律与对应的内置运算符保持一致

直接调用一个重载的运算符函数

```
//一个非成员运算符函数的等价调用
data1 + data2;           //普通的表达式
operator + (data1, data2); //等价的函数调用

data1 += data2;           //基于“调用”的表达式
data1.operator+=(data2); //对成员运算符函数的等价调用
```

- 通常情况下，不应该重载逗号、取地址符、逻辑与和逻辑或运算符
- 使用与内置类型一致的含义

选择作为成员或非成员

运算符定义为成员函数：左侧运算对象必须是运算符所属类的一个对象

```
//如果operator+是string类的成员：  
string s = "world";  
string t = s + "!"; //正确,s.operator+("!") 。  
string u = "hi" + s; //如果+是string的成员，则产生错误  
//等价于"hi".operator+(s)  
//const char*是一种内置类型，没有成员函数  
//因为string将+定义成了普通的非成员函数，所以"hi"+s等价于operator+ (  
"hi",s)
```

具有对称性的运算符可能转换任意一端的运算对象，通常应该是普通的非成员函数

重载输出运算符<<

```
ostream &operator<<(ostream &os, const Sales_data &item)
{
    os << item.isbn() <<" " <<item.units_sold <<" "
        <<item.revenue<<" " <<item.avg_price();
    return os;
}
```

输入输出运算符必须是非成员函数

```
Sales_data data;
data << cout; //如果operator<<是Sales_data的成员
```

重载输入运算符>>

```
istream &operator>> (istream &is, Sales_data &item)
{
    double price; //不需要初始化，先读入数据才会使用到
    is >> item.bookNo >> item.units_sold >> price;
    if(is) //检查输入是否成功
        item.revenue = item.units_sold * price;
    else
        item = Sales_data(); //输入失败：对象被赋予默认的状态
    return is;
}
```

输入运算符必须处理输入可能失败的情况，输出运算符不需要

通常情况下，把算术和关系运算符定义成非成员函数

```
//假设两个对象指向同一本书
Sales_data
operator+(const Sales_data &lhs, const Sales_data &rhs)
{
    Sales_data sum = lhs; //把lhs的数据成员拷贝给sum
    sum += rhs; //把rhs加到sum中
    return sum;
}
```

如果类同时定义了算术运算符和相关的复合赋值运算符，则通常情况下应该使用复合赋值来实现算术运算符

```
bool operator==(const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.isbn() == rhs.isbn() &&
        lhs.units_sold == rhs.units_sold &&
        lhs.revenue == rhs.revenue;
}

bool operator!=(const Sales_data &lhs, const Sales_data &rhs)
{
    return !(lhs == rhs);
}
```

如果类包含==，则当且仅当<的定义和==产生的结果一致时才定义<运算符

不论形参的类型是什么，赋值运算符都必须定义为成员函数

```
vector<string> v;  
v = {"a", "an", "the"};  
  
class StrVec{  
public:  
    StrVec &operator=(std::initializer_list<std::string>);  
    //... ..  
}  
  
StrVec &StrVec::operator=(std::initializer_list<std::string> il)  
{  
    //alloc_n_copy分配内存空间并从给定范围内拷贝元素  
    auto data = alloc_n_copy(il.begin(), il.end());  
    free(); //销毁对象中的元素并释放内存空间  
    elements = data.first;  
    first_free = cap = data.second;  
    return *this;  
}
```

复合赋值运算符通常情况下也应该定义为类的成员

```
//作为成员的二元运算符：左侧运算对象绑定到隐式的this指针  
Sales& Sales_data::operator+=(const Sales_data &rhs)  
{  
    units_sold += rhs.units_sold;  
    revenue += rhs.revenue;  
    return *this;  
}
```

下标运算符必须是成员函数

```
class StrVec{
public:
    std::string& operator[](std::size_t n)
        {return elements[n];}
    const std::string& operator[](std::size_t n) const
        {return elements[n];}
    //...
private:
    std::string *elements; //指向数组首元素的指针
};
```

通过this是否指向常量进行匹配

```
//假设svec是一个StrVec对象
const StrVec cvec = svec; //把svec的元素拷贝到cvec中
//如果svec中含有元素，对第一个元素运行string的empty函数
if(svec.size() && svec[0].empty()){
    svec[0]="zero"; //正确
    cvec[0]="Zip"; //错误
}
```

应该定义前置和后置版本，通常被定义为类的成员

```
class StrBlobPtr{
public:
    //递增和递减运算符
    StrBlobPtr& operator++(); //前置运算符
    StrBlobPtr& operator--();
    //其他...
};

StrBlobPtr& StrBlobPtr::operator++(){//前置版本：返回递增/递减对象的引用
    //如果curr已经指向了容器的尾后位置，则无法递增它
    check(curr,"increment past end of StrBlobPtr");
    ++curr; //将curr在当前状态下向前移动一个元素
    return *this;
}

StrBlobPtr& StrBlobPtr::operator--(){
    //如果curr是0，则继续递减它将参数一个无效下标
    --curr; //将curr在当前状态下向后移动一个元素
    check(curr,"decrement past begin of StrBlobPtr");
    return *this;
}
```

后置版本为了区分，接受一额外的（不被使用）int类型的参数
编译器会为此形参提供一个值为0的实参

```
class StrBlobPtr{
public:
    //递增和递减运算符
    StrBlobPtr operator++(int); //后置运算符
    StrBlobPtr operator--(int);
    //其他成员和之前的版本一致
};

//后置版本：递增/递减对象的值但是返回原值
StrBlobPtr StrBlobPtr::operator++(int){
    //此处无须检查有效性，调用前置递增运算符时才需要检查
    StrBlobPtr ret = *this; //记录当前的值
    ++*this; //向前移动一个元素，前置++需要检查递增的有效性
    return ret; //返回之前记录的状态
}

StrBlobPtr StrBlobPtr::operator--(int){
    //此处无须检查有效性，调用前置递减运算符时才需要检查
    StrBlobPtr ret = *this; //记录当前的值
    --*this; //向后移动一个元素，前置--需要检查递减的有效性
    return ret; //返回之前记录的状态
}
```

```
p++;
p.operator++(0);
++p;
p.operator++();
```


解引用运算符（*）和箭头运算符（->）

```
class StrBlobPtr{
public:
    std::string& operator*() const
    {
        auto p = check(curr, "deference past end");
        return (*p)[curr]; //( *p)是对象所指的vector
    }

    std::string* operator->() const
    {
        //实际工作委托给解引用运算符
        return & this->operator*();
    }
    //...
}
```

箭头运算符必须是类的成员，解引用运算符通常也是类的成员

这两个运算符的用法与指针或者vector迭代器的对应操作完全一致：

```
StrBlob a1 = {"hi", "bye", "now"};
StrBlobPtr p(a1); //p指向a1中的vector
*p = "okay"; //给a1的首元素赋值
cout<<p->size()<<endl; //打印4，这是a1首元素的大小
cout<<(*p).size()<<endl; //等价于p->size()
```

对箭头运算符返回值的限定

```
//根据类型的不同，point->mem分别等价于
(*point).mem; //point是一个内置的指针类型
point.operator()->mem; //point是类的一个对象
```

像使用函数一样使用类的对象（函数对象）

```
struct absInt{
    int operator()(int val) const{
        return val<0?-val:val;
    }
};

int i = -42;
absInt absObj; //含有函数调用运算符的对象
int ui = absObj(i); //将i传递给absObj.operator()
```

函数调用运算符必须是成员函数。可以定义多个版本。

函数对象类通常含有一些数据成员，用于定制调用运算符中的操作

```
class PrintString{
public:
    PrintString(ostream &o = cout, char c=' '): os(o),sep(c) {}
    void operator()(const string &s) const { os<<s<<sep; }
private:
    ostream &os; //用于写入的目的流
    char sep; //用于将不同输出隔开的字符
};

PrintString printer; //使用默认值，打印到cout
printer(s); //在cout中打印s，后面跟一个空格
PrintString errors(cerr, '\n');
errors(s); //在cerr中打印s，后面跟一个换行符

//函数对象常常作为泛型算法的实参
for_each(vs.begin(), vs.end(), PrintString(cerr, '\n'));
```

lamdba是函数对象

```
//根据单词长度进行排序
stable_sort(words.begin(), words.end(),
    [](const string &a, const string &b) { return a.size() < b.size(); });
//其行为类似于下面这个类的一个未命名对象
class ShorterString{
public:
    bool operator()(const &s1, const string &s2) const
    { return s1.size() < s2.size(); }
};

stable_sort(words.begin(), words.end(), ShorterString());
```

默认情况下，lambda不能改变它捕获的变量，因此函数调用运算符是一个const成员函数。如果lamdba被声明为可变的，则调用运算符就不是const的了

表示lambda及相应捕获行为的类

```
//获得第一个指向满足条件元素的迭代器，该元素满足size() is>= sz
auto wc = find_if( words.begin(), words.end(),
    [sz](const string &a) { return a.size() >= sz; });

//该lambda表达式产生的类将形如：
class SizeComp{
    SizeComp(size_t n):sz(n) { } //该形参对应捕获的变量
    //该调用运算符的返回类型、形参和函数体都与lambda一致
    bool operator() (const string &s) const { return s.size() >= sz; }
private:
    size_t sz; //该数据成员对应通过值捕获的变量
};

auto wc = find_if(words.begin(), words.end(), SizeComp(sz));
```

标准库函数对象		
算术	关系	逻辑
plus<Type>	equal_to<Type>	logical_and<Type>
minus<Type>	not_equal_to<Type>	logical_or<Type>
multiplies<Type>	greater<Type>	logical_not<Type>
divides<Type>	greater_equal<Type>	
modulus<Type>	less<Type>	
negate<Type>	less_equal<Type>	

```
plus<int> intAdd; //可执行int加法的函数对象
negate<int> intNegate; //可执行int值取反的函数对象

int sum = intAdd(10,20); //使用intAdd::operator(int,int)求10和20的和
sum - intNegate(intAdd(10,20)); //使用intNegate::operator(int)

sum = intAdd(10, intNegate(10)); //sum=0;
```

在算法中使用标准库函数对象

```
//传入一个零时的函数对象用于执行两个string对象的>比较运算
//默认是使用<
sort(svec.begin(),svec.end(),greater<string>());

vector<string *> nameTable; //指针的vector
//错误：nameTable中的指针彼此之间没有关系，所以<将产生未定义的行为
sort(nameTable.begin(),nameTable.end(),[](string *a, string *b) { return a<b; });
//正确：标准库规定指针的less是定义良好的
sort(nameTable.begin(), nameTable.end(), less<string*>());
```

可调用对象与function，不同类型可能具有相同的调用形式

```
//普通函数
int add(int i, int j) {return i+j;}
//lambda,其产生一个未命名的函数对象类
auto mod = [](int i, int j) { return i%j; };
//函数对象类
struct divide{
    int operator() (int denominator, int divisor) {
        return denominator/divisor;
    }
};
//调用形式（all signature）都是int(int,int)
```

为了构建一个简单的桌面计算机，需要顶一个函数表

```
//构建从运算符到函数指针的映射关系
map<string, int (*)(int,int)> binops; //二元运算
//正确：add是一个指向正确类型函数的指针
binops.insert({"+",add}); //{"+",add}是一个pair

binops.insert({"%",mod}); //错误：mod不是一个函数指针
```

标准库function的操作

<code>function<T> f;</code>	<code>f</code> 是一个用来存储可调用对象的空 <code>function</code> ，这些可调用对象的调用形式应该与函数类型 <code>T</code> 相同（即 <code>T</code> 是 <code>retType(args)</code> ）
<code>function<T> f(nullptr);</code>	显式地构造一个空 <code>function</code>
<code>function<T> f(obj);</code>	在 <code>f</code> 中存储可调用对象 <code>obj</code> 的副本
<code>f</code>	将 <code>f</code> 作为条件：当 <code>f</code> 含有一个可调用对象时为真；否则为假
<code>f(args)</code>	调用 <code>f</code> 中的对象，参数是 <code>args</code>
定义为 <code>function<T></code> 的成员的类型	
<code>result_type</code>	该 <code>function</code> 类型的可调用对象返回的类型
<code>argument_type</code>	当 <code>T</code> 有一个或两个实参时定义的类型。如果 <code>T</code> 只有一个实参，则 <code>argument_type</code> 是该类型的同义词；如果 <code>T</code> 有两个实参，
<code>first_argument_type</code>	则 <code>first_argument_type</code> 和 <code>second_argument_type</code>
<code>second_argument_type</code>	分别代表两个实参的类型

```
function<int(int, int)> f1 = add; //函数指针
function<int(int, int)> f2 = divide(); //函数对象类的对象
function<int(int, int)> f3 = [](int i, int j){return i*j;};
cout<<f1(4,2)<<endl;
cout<<f2(4,2)<<endl;
cout<<f3(4,2)<<endl;

map<string, function<int(int, int)>> binops = { //重新定义map
    {"+",add}, //函数指针
    {"-",std::minus<int>()}, //标准库函数对象
    {"/",divide()}, //用户定义的函数对象
    {"*",[](int i,int j){return i*j;}}, //为命名的lambda
    {"%",mod} }; //命名了的lambda

binops["+"](10,5); //调用add(10,5)
binops["-"](10,5);
binops["/"](10,5);
binops["*"](10,5);
binops["%"](10,5);
```


不能直接将重载函数的名字存入function类型的对象中

```
int add(int i, int j) { return i + j; }  
Sales_data add(const Sales_data&, const Sales_data&);  
map<string, function<int(int, int)>> binops;  
binops.insert({"+", add}); //错误：哪个add？
```

//解决二义性问题的一个途径是存储函数指针

```
int (*fp)(int, int) = add;  
binops.insert({"+", fp});  
//也可以使用lambda来消除二义性  
binops.insert({"+", [](int a, int b){return add(a,b);}});
```



类型转换运算符是特殊成员函数，它负责将一个类类型的值转换成其他类型：一般形式为：`operator type() const`;

//定义一个比较简单的类，令其表示0到255之间的一个整数

```
class SmallInt{
public:
    SmallInt(int i = 0): val(i)
    {
        if(i<0 || i>255)
            throw std::out_of_range("Bad SmallInt value");
    }
    operator int() const { return val; }
private:
    std::size_t val;
};
```

不能声明返回类型，
形参列表页必须为空

SmallInt si;

si = 4; //首先将4隐式地转换成SmallInt，然后调用SmallInt::operator=
si + 3; //首先将si隐式地转换成int，然后执行整数的加法

//内置类型转换将double实参转成int

SmallInt si = 3.14; //调用SmallInt(int)构造函数

//SmallInt的类型转换运算符将si转换成int

si + 3.14; //内置类型转换将所得的int继续转换成double

类型转换运算符是隐式执行的，不能传递实参：

```
class SmallInt;
```

```
operator int(SmallInt&); //错误：不是成员函数
```

```
class SmallInt{
```

```
public:
```

```
    int operator int() const;           //错误：指定了返回类型
```

```
    operator int(int = 0) const;       //错误：参数列表不为空
```

```
    operator int*() const { return 42; } //错误：42不是一个指针
```

```
};
```

类型转换运算符可能产生意外结果

```
//当istream含有向bool的类中转换时
int i = 42;
cin << i; //提升后的bool值（1或0）最终被左移42个位置
```

为了防止上面的异常，C++ 11引入了 **显式的类型转换运算符**

```
class SmallInt{
public:
    //编译器不会自动指向这一类型转换
    explicit operator int() const { return val; }
    //其他成员与之前的版本一致
};

SmallInt si = 3; //正确：SmallInt的构造函数不是显式的
si + 3; //错误：此处需要隐式的类型转换，但类的运算符是显式的
static_cast<int>(si) + 3; //正确：显式地请求类型转换
```

也有例外，如果表达式被用作条件，编译器会将显式的类型转换自动应用于它（被隐式地执行）：

- if、while及do语句的条件部分
- for语句头的条件表达式
- ！、||、&&的运算对象
- ?: 的条件表达式

向bool的类型转换通常在条件部分，因此operator bool一般定义为explicit的

```
while(std::cin >> value)
```

避免有二义性的类型转换

```
//最好不要再两个类之间构建相同的类型转换
struct B;
struct A {
    A() = default;
    A(const B&); //把一个B转换成A
    //其他数据成员
};

struct B{
    operator A() const; //也是把一个B转换成A
    //其他数据成员
};

A f(const A&);
B b;
A a = f(b); //二义性错误：含义是f(B::operator A())，还是f(A::A(const B&))？
A a1 = f(b.operator A()); //正确
A a2 = f(A(b)); //正确
```


二义性与转换目标为内置类型的多重类型转换

```
struct A {
    A(int = 0); //最好不要创建两个转换源都是算术类型的类型转换
    A(double);
    operator int() const; //最好不要创建两个转换对象都是算术类型的类型转换
    operator double() const;
    //其他成员
};

void f2(long double);
A a;
f2(a); //二义性错误：含义是f(A::operator int())还是f(A::operator double())?
long lg;
A a2(lg); //二义性错误：含义是A::A(int)还是A::A(double)?

short s = 42;
//把short提升成int优于把short转换成double
A a3(s); //使用A::A(int)
```

重载函数与转换构造函数

```
struct C{
    C(int);
    //其他成员
};

struct D{
    D(int);
    //其他成员
};

void manip(const C&);
void manip(const D&);
manip(10); //二义性错误
manip(C(10)); //正确：调用manip(const C&)
```

需要使用构造函数或者强制类型转换，通常意味着程序的设计存在不足

重载函数与用户定义的类型转换

```
struct E{
    E(double);
    //其他成员
};

void manip2(const C&);
void manip2(const E&);
//二义性错误：两个不同的用户定义的类型转换都能用于此处
manip2(10); //C(10)还是E(double(10))
```

转换级别只有在请求同一个类型转换时才有用

函数匹配与重载运算符

```
class SmallInt {  
friend SmallInt operator+(const SmallInt&,const SmallInt&);  
public:  
    SmallInt(int = 0);           //转换源为int的类型转换  
    operator int() const { return val; } //转换目标为int的类型转换  
private:  
    std::size_t val;  
};  
  
SmallInt s1, s2;  
SmallInt s3 = s1 + s2;           //使用重载的operator+  
int i = s3 + 0;                  //二义性错误
```