**Chapter 11 Working with Classes**

* **Operator Overloading**

The \* operator, when applied to an address, yields the value stored at that address. But applying \* to two numbers yields the product of the values.

For example, a common computing task is adding two arrays. Usually, it looks like the following for loop:

for (int i = 0; i < 20; i++)

evening[i] = sam[i] + janet[i]; // add element by element

But in C++, you can define a class that represents arrays and that overloads the + operator so that you can do this:

evening = sam + janet; // add two array objects

To overload an operator, you use a special function form called an **operator function**. An operator function has the following form, where *op* is the symbol for the operator being overloaded:

operatorop(argument-list)

For example, *operator+()* overloads the + operator and *operator\*()* overloads the \* operator. The op has to be a valid C++ operator; you can’t just make up a new symbol. For example, you can’t have an operator@() function because C++ has no @ operator. But the ***operator[]()*** function would overload the [] operator **because [] is the array indexing operator**.

Suppose, for example, that you have a *Salesperson* class for which you define an *operator+()* member function to overload the + operator so that it adds sales figures of one salesperson object to another. Then, if *district2*, *sid*, and *sara* are all objects of the Salesperson class, you can write this equation:

district2 = sid + sara;

The compiler, recognizing the operands as belonging to the *Salesperson* class, replaces the operator with the corresponding operator function:

district2 = **sid.operator+(sara);**

The function then uses the ***sid* *object implicitly*** (because it invoked the method) and the ***sara object explicitly*** (because it’s passed as an argument) to calculate the sum, which it then returns.

**下面是一个计算时间的程序：**

// **mytime0.h** -- Time class before operator overloading

#ifndef MYTIME0\_H\_

#define MYTIME0\_H\_

class Time

{

private:

int hours;

int minutes;

public:

Time();

Time(int h, int m = 0);

void AddMin(int m);

void AddHr(int h);

void Reset(int h = 0, int m = 0);

Time Sum(**const** Time & t) **const**;

void Show() **const**;

};

#endif

// **mytime0.cpp** -- implementing Time methods

#include <iostream>

#include "mytime0.h"

Time::Time()

{

hours = minutes = 0;

}

Time::Time(int h, int m )

{

hours = h;

minutes = m;

}

void Time::**AddMin**(int m)

{

minutes += m;

**hours += minutes / 60;**

**minutes %= 60;**

}

void Time::AddHr(int h)

{

hours += h;

}

void Time::Reset(int h, int m)

{

hours = h;

minutes = m;

}

**Time** Time::Sum(**const** Time & t) const

{

**Time sum**;

sum.minutes = *minutes + t.minutes*;

sum.hours = *hours + t.hours* + sum.minutes / 60;

sum.minutes %= 60;

***return sum***;

}

void Time::Show() const

{

std::cout << hours << " hours, " << minutes << " minutes";

}

Consider the code for the Sum() function. Note that the **argument is a reference** but that **the return type is not a reference**. The reason for *making the argument a reference is efficiency*.

**However, the return value cannot be a reference**. The reason is that the function creates a new Time object (sum) that represents the sum of the other two Time objects. ***Returning the object***, as this code does, **creates a copy of the object (sum) that the calling function can use**. ***If the return type were Time &***, however, the reference would be to the sum object. ***But the sum object is a local variable and is destroyed when the function terminates***, so the reference would be a reference to a non-existent object. ***Using a Time return type***, however, means the program constructs a copy of sum before destroying it, and the calling function gets the copy.

**[Caution]**

**Don’t return a reference to a local variable or another temporary object**. When the function terminates and the local variable or temporary object disappears, the reference becomes a reference to non-existent data.

**Reference一般用于调用在函数之前就已经存在的参数.**

// **usetime0.cpp** -- using the first draft of the Time class

#include <iostream>

#include "mytime0.h"

int main()

{

**using std::cout;**

**using std::endl;**

Time planning;

Time coding(2, 40);

Time fixing(5, 55);

Time total;

cout << "planning time = ";

planning.Show();

cout << endl;

cout << "coding time = ";

coding.Show();

cout << endl;

cout << "fixing time = ";

fixing.Show();

cout << endl;

***total = coding.Sum(fixing);*** // add coding and fixing

cout << "coding.Sum(fixing) = ";

total.Show();

cout << endl;

return 0;

}

**Adding an Addition Operator**

You just change the name of Sum() to the odder-looking name operator+(). That’s right: You just append the operator symbol (*+*, in this case) to the end of *operator* and use the result as a method name.

// **mytime1.h** -- Time class before operator overloading

#ifndef MYTIME1\_H\_

#define MYTIME1\_H\_

class Time

{

private:

int hours;

int minutes;

public:

Time();

Time(int h, int m = 0);

void AddMin(int m);

void AddHr(int h);

void Reset(int h = 0, int m = 0);

**Time operator+(const Time & t) const;**

void Show() const;

};

#endif

**// mytime1.cpp** -- implementing Time methods

#include <iostream>

#include "mytime1.h"

……

void Time::Reset(int h, int m)

{

hours = h;

minutes = m;

}

**Time** Time::***operator+***(const Time & t) const

{

Time sum;

sum.minutes = minutes + t.minutes;

sum.hours = hours + t.hours + sum.minutes / 60;

sum.minutes %= 60;

**return sum**;

}

void Time::Show() const

{

std::cout << hours << " hours, " << minutes << " minutes";

}

Like Sum(), **operator+() is invoked by a Time object**, takes a second Time object as an argument, and returns a Time object. Thus, you can invoke the operator+() method by using the same syntax that Sum() uses:

total = coding.operator+(fixing); // function notation

But naming the method operator+() also lets you use operator notation:

total = coding + fixing; // operator notation

The **object to the left of the operator** (coding, in this case***) is the invoking object***, and the **object to the right** (fixing, in this case) is the ***one passed as an argument***.

// **usetime1.cpp** -- using the second draft of the Time class

#include <iostream>

#include "mytime1.h"

int main()

{

using std::cout;

using std::endl;

Time planning;

Time coding(2, 40);

Time fixing(5, 55);

Time total;

**total = coding + fixing;**

// operator notation

cout << "coding + fixing = ";

total.Show();

cout << endl;

Time morefixing(3, 28);

cout << "more fixing time = ";

morefixing.Show();

cout << endl;

**total = morefixing.operator+(total);**

// function notation

cout << "morefixing.operator+(total) = ";

total.Show();

In short, the name of the operator+() function allows it to be invoked by using either function notation or operator notation. **The compiler uses the operand types to figure out what to do**:

int a, b, c;

Time A, B, C;

c = a + b; // use int addition

C = A + B; // use addition as defined for Time objects

**Can you add more than two objects**? For example, if t1, t2, t3, and t4 are all Time objects, can you do the following?

t4 = t1 + t2 + t3; // valid?

The way to answer this is to consider how the statement gets translated into function calls. Because addition is a left-to-right operator, the statement is first translated to this:

t4 = t1.operator+(t2 + t3); // valid?

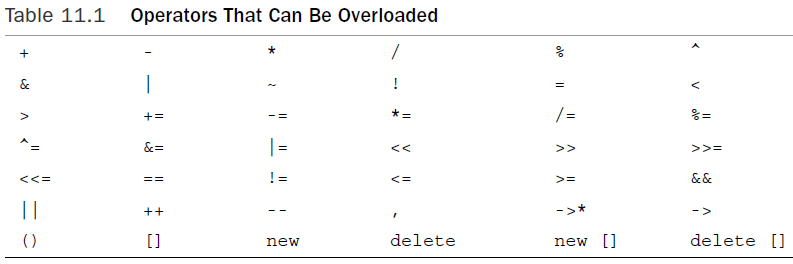
Then the function argument is itself translated to a function call, giving the following:

t4 = t1.operator+(t2.operator+(t3)); // valid? YES

**Is this valid? Yes**, it is. The function call ***t2.operator+(t3)*** returns a Time object that represents the sum of t2 and t3. This object then becomes the object of the ***t1.operator+()*** function call, and that call returns the sum of t1 and the Time object that represents the sum of t2 and t3. In short, the final return value is the sum of t1, t2, and t3, just as desired.

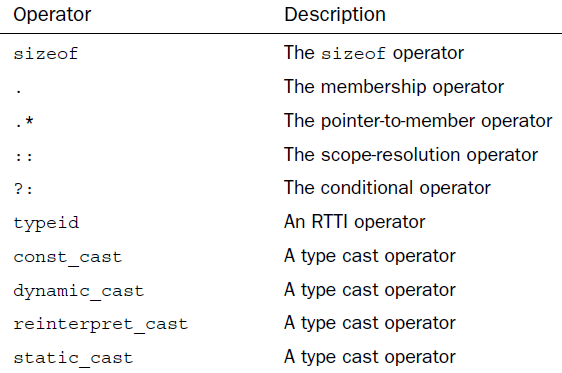
**Overloading Restrictions**

Most C++ operators (see Table 11.1) can be overloaded in the manner described in the preceding section.

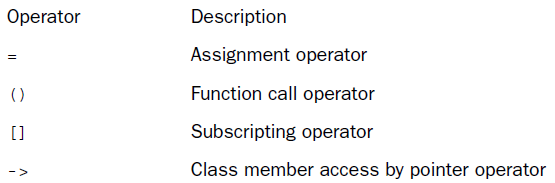
****

Overloaded operators (with some exceptions) don’t necessarily have to be member functions. However, at least one of the operands has to be a *user-defined type*.

1. **The overloaded operator must have at least one operand that is a user-defined type**. This prevents you from overloading operators for the standard types. Thus, you can’t redefine the minus operator (-) so that it yields the sum of two double values instead of their difference;
2. You can’t use an operator in a manner that violates the syntax rules for the original operator. For example, you can’t overload the modulus operator (%) so that it can be used with a single operand;
3. Similarly, **you can’t alter operator precedence**. So if you overload the addition operator to let you add two classes, the new operator has the same precedence as ordinary addition;
4. **You can’t create new operator symbols**. For example, you can’t define an operator\*\*() function to denote exponentiation;
5. You cannot overload the following operators:



1. Most of the operators in Table 11.1 can be overloaded by using either ***member*** or ***non-member*** functions. However, **you can use only member functions to overload the following operators**:



In addition to these formal restrictions, you should use **sensible restraint** in overloading operators. For example, you shouldn’t overload the \* operator so that it swaps the data members of two Time objects. Nothing in the notation would suggest what the operator did, so it would be better to define a class method with an explanatory name such as Swap().

**More Overloaded Operators**

// **mytime2.h** -- Time class after operator overloading

#ifndef MYTIME2\_H\_

#define MYTIME2\_H\_

class Time

{

private:

int hours;

int minutes;

public:

Time();

Time(int h, int m = 0);

void AddMin(int m);

void AddHr(int h);

void Reset(int h = 0, int m = 0);

**Time operator+(const Time & t) const;**

**Time operator-(const Time & t) const;**

**Time operator\*(double n) const;**

void Show() const;

};

#endif

// **mytime2.cpp** -- implementing Time methods

#include <iostream>

#include "mytime2.h"

Time Time::operator+(const Time & t) const

{

Time sum;

sum.minutes = minutes + t.minutes;

sum.hours = hours + t.hours + sum.minutes / 60;

sum.minutes %= 60;

return sum;

}

Time Time::operator-(const Time & t) const

{

Time diff;

int tot1, tot2;

tot1 = t.minutes + 60 \* t.hours;

tot2 = minutes + 60 \* hours;

diff.minutes = (tot2 - tot1) % 60;

diff.hours = (tot2 - tot1) / 60;

return diff;

}

Time Time::operator\*(double mult) const

{

Time result;

long totalminutes = hours \* mult \* 60 + minutes \* mult;

result.hours = totalminutes / 60;

result.minutes = totalminutes % 60;

return result;

}

* **Introducing Friends**

**C++ controls access to the private portions of a class object**. Usually, public class methods serve as the only access, but sometimes this restriction is too rigid to fit particular programming problems. In such cases, C++ provides another form of access: the ***friend***. Friends come in three varieties: **Friend functions**, **Friend classes** and **Friend member functions**.

By **making a function a friend to a class**, you allow the function the same access privileges that a member function of the class has.

In the previous Time class example, the ***overloaded multiplication operator is different*** from the other two overloaded operators in that it combines two different types. That is, the addition and subtraction operators each combine two Time values, but the multiplication operator combines a Time value with a double value. This restricts how the operator can be used. ***Remember, the left operand is the invoking object***. That is:

**A = B \* 2.75;**

translates to the following member function call:

A = B.operator\*(2.75);

But what about the following statement?

**A = 2.75 \* B;** // cannot correspond to a member function

Conceptually, 2.75 \* B should be the same as B \* 2.75, **but the first expression cannot correspond to a member function because 2.75 is not a type Time object**. Remember, the left operand is the invoking object, but 2.75 is not an object. *So the compiler cannot replace the expression with a member function call*.

**One way around** this difficulty is to tell everyone that you can only write B \* 2.75 but never write 2.75 \* B. This is a server-friendly, client-beware solution, and that’s not what OOP is about.

However, **there is another possibility—*using a nonmember function***. (Remember, most operators can be overloaded using either member or nonmember functions.) ***A nonmember function*** is ***not invoked by an object***; instead, any values it uses, including objects, are explicit arguments.

The function would have this prototype:

Time operator\*(double m, const Time & t);

**With the *non-member overloaded* operator function**, *the left operand of an operator expression corresponds to the first argument of the operator function, and the right operand corresponds to the second argument*. Meanwhile, the original ***member function*** handles operands ***in the opposite order***—that is, a Time value multiplied by a double value.

There is a special category of *nonmember* functions, called ***friends***, which can access private members of a class.

**Creating Friends**

The first step toward creating a friend function is to place a prototype ***in the class declaration*** and prefix the declaration with the keyword ***friend***:

**friend** Time operator\*(double m, const Time & t); // goes **in class declaration**

This prototype has two implications:

1. Although the operator\*() function is declared in the class declaration, *it is not a member function*. So it isn’t invoked by using the membership operator;
2. Although the operator\*() function is not a member function, *it has the same access rights as a member function;*

Time operator\*(double m, const Time & t) // friend not used in definition

{

Time result;

long totalminutes = t.hours \* mult \* 60 +t. minutes \* mult;

result.hours = totalminutes / 60;

result.minutes = totalminutes % 60;

return result;

}

Because it is not a member function, you don’t use the Time:: qualifier. Also you don’t use the friend keyword in the definition. With this declaration and definition, the statement

A = 2.75 \* B;

translates to the following and invokes the nonmember friend function just defined:

**A = operator\*(2.75, B);**

In short, a friend function to a class is a nonmember function that has the same access rights as a member function.

**The third solution is**, you can write this particular friend function as a non-friend by altering the definition so that it switches which value comes first in the multiplication.

Time operator\*(double m, const Time & t)

{

return t \* m; // **use t.operator\*(m)，其实调用了之前的operator\*的overload;**

}

The second version accessed *t.minutes* and *t.hours* explicitly, so it had to be a friend. *The third version* only ***uses the Time object t as a whole***, letting *a member function handle the private values*, **so this version doesn’t have to be a friend**. Nonetheless, there are reasons to make this version a friend, too. Most importantly, it ties the function in as part of the official class interface. Second, if you later find a need for the function to access private data directly, you only have to change the function definition and not the class prototype.

**[Tip]** If you want to overload an operator for a class and you want to use the operator with a **non-class term as the first operand**, you can use a friend function to reverse the operand order.

**A Common Kind of Friend: Overloading the << Operator**

One very useful feature of classes is that you can overload the **<<** operator so that you can use it with cout to **display an object’s contents**.

Suppose trip is a Time object. To display Time values, we’ve been using Show(). Wouldn’t it be nice, however, if you could do the following?

cout << trip; // make cout recognize Time class?

**The First Version of Overloading <<**

To teach the Time class to use cout, you can use a ***friend function***. Why? Because a statement like the following uses two objects, with the ***ostream class object (cout) first***:

cout << trip;

*If you use a Time* ***member function*** *to overload <<,* ***the Time object would come first***, as it did when you overloaded the \* operator with a member function. That means you would have to use the << operator this way:

trip << cout; // **if operator<<() were a Time member function**

This would be confusing. But by using a friend function, you can overload the operator this way:

**void** operator<<(**ostream & os**, **const Time & t**)

{

os << t.hours << " hours, " << t.minutes << " minutes";

}

This lets you use

cout << trip;

*<<左边的是overload定义里的第一个参数, 右边的是overload定义里的第二个参数*; cout是ostream的一个object;

The new Time class declaration makes the operator<<() function a friend function to the *Time* class. But this function, although not inimical to the ostream class, is not a friend to that class. The operator<<() function takes an ostream argument and a Time argument, so it might seem that this function has to be a friend to both classes. If you look at the code for the function, however, you’ll notice that the function accesses individual members of the Time object **but only uses the ostream object as a whole**. Because operator<<() accesses private Time object members directly, it has to be a friend to the Time class. But because it does not directly access private ostream object members, the function does not have to be a friend to the ostream class.

Note that the new operator<<() definition uses the ostream reference os as its first argument. Normally, os refers to the cout object, as it does in the expression cout << trip. But you could use the operator with other ostream objects, in which case os would refer to those objects.

The call cout << trip should use the cout object itself, not a copy, so the function passes the object as a reference instead of by value. Thus, the expression cout << trip causes *os* to be an alias for *cout*, and the expression cerr << trip causes os to be an alias for cerr. The Time object can be passed by value or by reference because either form makes the object values available to the function.

**The Second Version of Overloading <<**

The implementation just presented has a problem. Statements such as this work fine:

cout << trip;

But the implementation doesn’t allow you to combine the redefined << operator with the ones cout normally uses:

cout << "Trip time: " << trip << " (Tuesday)\n"; **// can't do**

原因: 前面overload后的<<左边必须是个ostream的object, 右边必须是个Time class的object; 而在上面这句里, 实际的操作流程是这样的：

(((cout << "Trip time: ") << trip) << " (Tuesday)\n");

第一次调用<<, 右边是character constant : "Trip time: ", 不是Time class的object, **这个没问题, C++自己知道这时不应该用你overload的函数, 而用cout自己针对<<的定义**; 但这二次针对trip调用完后函数没有返回, 因为overload时函数类型是void, **那么第三次针对Tuesday调用时<<左边是个void, 这不行**;

You can take the same approach with the friend function. You just revise the operator<<() function so that it returns a reference to an ostream object:

***ostream &*** operator<<(ostream & os, const Time & t)

{

os << t.hours << " hours, " << t.minutes << " minutes";

return os;

}

Note that the return type is ***ostream &***. Recall that this means that the function returns a reference to an ostream object. Because a program passes an object reference to the function to begin with, the net effect is that the function’s return value is just the object passed to it.

cout << "Trip time: " << trip << " (Tuesday)\n"; **// can do**

第一步, cout << "Trip time: ", invokes the particular **ostream definition of <<** that displays a string and returns the cout object reference;

第二步, cout << trip, the program uses the **Time declaration of <<** to display the trip values and to return the cout object reference again;

As a point of interest, this version of operator<<() also can be used for file output:

#include <fstream>

...

ofstream fout;

fout.open("savetime.txt");

Time trip(12, 40);

fout << trip;

The last statement becomes this:

operator<<(fout, trip);

And as Chapter 8 points out, the properties of class inheritance allow an ostream reference to refer to ostream objects and to ofstream objects.

**[Tip]** In general, to overload the << operator to display an object of class *c\_name*, **you use a friend function** with a definition in this form:

ostream & operator<<(ostream & os, const c\_name & obj)

{

os << ... ; // display object contents

**return os;**

}

修改后的函数文件如下:

// **mytime3.h** -- Time class with friends

#ifndef MYTIME3\_H\_

#define MYTIME3\_H\_

#include <iostream>

class Time

{

private:

int hours;

int minutes;

public:

Time();

Time(int h, int m = 0);

void AddMin(int m);

void AddHr(int h);

void Reset(int h = 0, int m = 0);

Time operator+(const Time & t) const;

Time operator-(const Time & t) const;

Time **operator\***(double n) const;

**friend** Time **operator\***(double m, const Time & t) { return t \* m; } **// inline definition**

**friend std::ostream & operator<<(std::ostream & os, const Time & t);**

};

#endif

You use the friend keyword only in the prototype found in the class declaration. You don’t use it in the function definition unless the definition is also the prototype.

// **mytime3.cpp** -- implementing Time methods

#include "mytime3.h"

……

**Time Time::**operator\*(double mult) const

{

Time result;

long totalminutes = hours \* mult \* 60 + minutes \* mult;

result.hours = totalminutes / 60;

result.minutes = totalminutes % 60;

return result;

}

**std::ostream** & operator<<(std::ostream & os, const Time & t) // friend definition doesn’t have prefix friend；

{

os << t.hours << " hours, " << t.minutes << " minutes";

return os;

}

……

* **Overloaded Operators: Member Versus Nonmember Functions**

For many operators, you have a choice between using member functions or nonmember functions to implement operator overloading. Typically, **the nonmember version is a friend function so that it can directly access the private data for a class**.

For example, consider the addition operator for the Time class. It has this prototype in the Time class declaration:

**// *member version***

Time operator+(*const Time & t*) const;

Instead, the class could use the following prototype:

**// *nonmember version***

**friend** Time operator+(*const Time & t1, const Time & t2*);

The addition operator requires two operands. **For the member function version**, one is passed implicitly via the ***this*** pointer and the second is passed explicitly as a function argument. **For the friend version**, both are passed as arguments.

A ***nonmember version, which is also friend function,*** of an overloaded operator function **requires as many formal parameters as the operator has operands**. A ***member version*** of the same operator **requires one fewer parameter** because one operand is passed implicitly as the invoking object.

Keep in mind that you must choose one or the other form when defining a given operator, ***but not both***. Because both forms match the same expression, defining both forms is an ambiguity error, leading to a compilation error. **这里说的not the both是针对同一个操作符在参数完全相同的情况下说的, 如果参数不同, 还是可以both的, 比如上一页的operator\*.**

* **More Overloading: A Vector Class**

这里的vector指的是几何上的向量.

Vectors are a natural choice for operator overloading. First, you can’t represent a vector with a single number, so it makes sense to create a class to represent vectors. Second, vectors have analogs to ordinary arithmetic operations such as addition and subtraction. This parallel suggests overloading the corresponding operators so you can use them with vectors.

To refresh your memory about namespaces, the listing places the **class declaration inside the VECTOR namespace.** Also the program uses ***enum*** to **create a couple constants** (RECT and POL) for identifying the two representations.

// **vect.h** -- Vector class with <<, mode state

#ifndef VECTOR\_H\_

#define VECTOR\_H\_

#include <iostream>

**namespace VECTOR**

{

class Vector

{

**public:**

**enum Mode {RECT, POL};** **// RECT for rectangular, POL for Polar modes**

**private:**

double x; // horizontal value

double y; // vertical value

double mag; // length of vector

double ang; // direction of vector in degrees

**Mode mode;** // RECT or POL

// private methods for setting values

void set\_mag();

void set\_ang();

void set\_x();

void set\_y();

**public:**

Vector();

Vector(double n1, double n2, **Mode form = RECT**); // Mode default is RECT

void reset(double n1, double n2, **Mode form = RECT**);

~Vector();

double xval() **const** {return x;} // report x value

double yval() **const** {return y;} // report y value

double magval() **const** {return mag;} // report magnitude

double angval() **const** {return ang;} // report angle

void polar\_mode(); // set mode to POL

void rect\_mode(); // set mode to RECT

***// operator overloading***

Vector operator+(const Vector & b) const;

Vector operator-(const Vector & b) const;

Vector operator-() const;

Vector operator\*(double n) const;

***// friends***

friend Vector operator\*(double n, const Vector & a);

friend std::ostream & operator<<(std::ostream & os, const Vector & v);

};

} // end namespace VECTOR

#endif

Notice that the four functions in Listing above that **report** component values are defined in the class declaration. ***This automatically makes them inline functions***. The fact that these functions are so short makes them excellent candidates for inlining.

// **vect.cpp** -- methods for the Vector class

**#include <cmath>**

#include "vect.h" // includes <iostream> in “vect.h”

**using std::sqrt;**

**using std::sin;**

**using std::cos;**

**using std::atan;**

**using std::atan2;**

**using std::cout;**

**namespace VECTOR**

{

**const double Rad\_to\_deg = 45.0 / atan(1.0);** // should be about 57.2957795130823

**// private methods**

// calculates magnitude from x and y

void Vector::set\_mag()

{

mag = sqrt(x \* x + y \* y);

}

void Vector::set\_ang()

{

if (x == 0.0 && y == 0.0)

ang = 0.0;

else

ang = atan2(y, x);

}

// set x from polar coordinate

void Vector::set\_x()

{

x = mag \* cos(ang);

}

// set y from polar coordinate

void Vector::set\_y()

{

y = mag \* sin(ang);

}

**// public methods**

Vector::Vector() ***// default constructor***

{

x = y = mag = ang = 0.0;

***mode = RECT*;**

}

// construct vector from rectangular coordinates if form is r

// (the default) or else from polar coordinates if form is p

**Vector::Vector**(double n1, double n2, **Mode form**)

{

**mode = form; // mode是private member**

if (form == RECT)

{

x = n1;

y = n2;

set\_mag();

set\_ang();

}

else if (form == POL)

{

mag = n1;

ang = n2 / Rad\_to\_deg;

set\_x();

set\_y();

}

else

{

cout << "Incorrect 3rd argument to Vector() -- ";

cout << "vector set to 0\n";

x = y = mag = ang = 0.0;

**mode = RECT;**

}

}

**void Vector:: reset(double n1, double n2, Mode form)**

{

mode = form;

if (form == RECT)

{

x = n1;

y = n2;

set\_mag();

set\_ang();

}

else if (form == POL)

{

mag = n1;

ang = n2 / Rad\_to\_deg;

set\_x();

set\_y();

}

else

{

cout << "Incorrect 3rd argument to Vector() -- ";

cout << "vector set to 0\n";

x = y = mag = ang = 0.0;

mode = RECT;

}

}

Vector::~Vector() // destructor

{

}

void Vector::polar\_mode() // set to polar mode

{

mode = POL; // mode itself is a private member

}

void Vector::rect\_mode() // set to rectangular mode

{

mode = RECT;

}

**// operator overloading**

// add two Vectors

**Vector Vector::operator+**(const Vector & b) const

{

return **Vector**(x + b.x, y + b.y);

}

// subtract Vector b from a

**Vector Vector::operator-**(const Vector & b) const

{

return **Vector**(x - b.x, y - b.y);

}

// reverse sign of Vector

**Vector Vector::operator-()** const // no parameters

{

return Vector(-x, -y);

}

// multiply vector by n

**Vector Vector::operator\*(double n)** const

{

return Vector(n \* x, n \* y);

}

**// friend methods**

// multiply n by Vector a

**Vector operator\*(double n, const Vector & a)**

{

return a \* n; **// convert double times Vector to Vector times double**

}

// display rectangular coordinates if mode is RECT,

// else display polar coordinates if mode is POL

**std::ostream & operator<<(std::ostream & os, const Vector & v)**

{

if (v.mode == **Vector::RECT**)

os << "(x,y) = (" << v.x << ", " << v.y << ")";

else if (v.mode == **Vector::POL**)

{

os << "(m,a) = (" << v.mag << ", "

<< v.ang \* Rad\_to\_deg << ")";

}

else

os << "Vector object mode is invalid";

**return os;**

}

} // end namespace VECTOR

Note how the constructor functions and the reset() function each set both the rectangular and the polar representations of the vector. Thus, either set of values is available immediately without further calculation.

The reset() method isn’t really needed. Suppose shove is a Vector object and that you have the following code:

shove.reset(100,300);

You can get the same result by using a constructor instead:

shove = Vector(100,300); // create and assign a temporary object

However, the set() method alters the contents of shove directly, whereas using the constructor adds the extra steps of creating a temporary object and assigning it to shove.

**Using a State Member**

A member like Mode is called state member.

Vector foolery(20.0, 30.0, **VECTOR::Vector::**POL); // set mag = 20, ang = 30

VECTOR是namespace, Vector是class; the identifier POL has class scope, so class definitions can just use the unqualified name. But the fully qualified name is ***VECTOR::Vector::POL*** because POL is defined in the Vector class, and Vector is defined in the VECTOR namespace.

A call such as the following won’t compile because an *integer like 2 can’t implicitly be converted to an enum type*:

Vector rector(20.0, 30.0, **2**); // type mismatch - 2 not an enum type

Vector rector(20.0, 30.0, **VECTOR::Vector::Mode (2)**); // type cast

**Because operator<<() is a friend function and not part of the class scope, it has to use Vector::RECT instead of just RECT.** But it is in the VECTOR namespace, so it doesn’t need to use the fully qualified name of VECTOR::Vector::RECT.

**[Tip]** **If a method needs to compute a new class object, you should see if you can use a class constructor to do the work.** Not only does that save you trouble, it ensures that the new object is constructed in the proper fashion.

* **Automatic Conversions and Type Casts for Classes**

The following statements all generate numeric type conversions:

long count = 8; // int value 8 converted to type long

double time = 11; // int value 11 converted to type double

**int side = 3.33**; // **double value 3.33 converted to type int 3**

The C++ language does not automatically convert types that are not compatible. For example, the following statement fails because the left side is a pointer type, whereas the right side is a number:

int \* p = 10; // Fail, type clash

And even though a computer may represent an address internally with an integer, integers and pointers are conceptually quite different. For example, you wouldn’t square a pointer. However, when automatic conversions fail, you may use a type cast:

**int \* p = (int \*) 10; // Ok, p and (int \*) 10 both pointers**

This sets a pointer to the address 10 by type casting 10 to type pointer-to-int (that is, type int \*). Whether this assignment makes sense is another matter.

You may define a class sufficiently related to a basic type or to another class that it makes sense to convert from one form to another. In such a case, **you can tell C++ how to make such conversions automatically** or, perhaps, via a type cast.

// **stonewt.h** -- definition for the Stonewt class

#ifndef STONEWT\_H\_

#define STONEWT\_H\_

class Stonewt

{

private:

**enum {Lbs\_per\_stn = 14}; // Pounds per stone, Constant**

int stone; // whole stones

double pds\_left; // fractional pounds

double pounds; // entire weight in pounds

public:

Stonewt(double lbs); // constructor for double pounds

Stonewt(int stn, double lbs); // constructor for stone, lbs

Stonewt(); // default constructor

~Stonewt();

void show\_lbs() const; // show weight in pounds format

void show\_stn() const; // show weight in stone format

};

#endif

The *Stonewt* class has three constructors. They allow you to initialize a Stonewt object to a floating-point number of pounds or to a combination of stone and pounds.

Stonewt blossem(132.5); // weight = 132.5 pounds

Stonewt buttercup(10, 2); // weight = 10 stone, 2 pounds

Stonewt bubbles; // weight = default value

// **stonewt.cpp** -- Stonewt methods

#include <iostream>

using std::cout;

#include "stonewt.h"

// construct Stonewt object from double value

Stonewt::Stonewt(double lbs)

{

stone = **int (lbs)** / Lbs\_per\_stn;

pds\_left = int (lbs) % Lbs\_per\_stn + lbs - int(lbs);

pounds = lbs;

}

// construct Stonewt object from stone, double values

Stonewt::Stonewt(int stn, double lbs)

{

stone = stn;

pds\_left = lbs;

pounds = stn \* Lbs\_per\_stn +lbs;

}

Stonewt::Stonewt() // default constructor, wt = 0

{

stone = pounds = pds\_left = 0;

}

**Stonewt::~Stonewt() // destructor**

**{**

**}**

// show weight in stones

void Stonewt::show\_stn() const

{

cout << stone << " stone, " << pds\_left << " pounds\n";

}

// show weight in pounds

void Stonewt::show\_lbs() const

{

cout << pounds << " pounds\n";

}

Note that each constructor assigns values to all three private members. Thus, creating a *Stonewt* object automatically sets both representations of weight.

Because a Stonewt object represents a single weight, it makes sense to provide ways to convert an integer or a floating-point value to a Stonewt object. And you have already done so! ***In C++, any constructor that takes a single argument acts as a blueprint for converting a value of that argument type to the class type*.** Thus the following constructor serves as instructions for converting a type double value to a type Stonewt value:

Stonewt(double lbs); **// template for double-to-Stonewt conversion**

That is, you can write code like the following:

Stonewt myCat; // create a Stonewt object

**myCat = 19.6**; // automatically use **Stonewt(double)** to convert 19.6 to Stonewt

The program uses the *Stonewt*(double) constructor to construct a temporary Stonewt object, using 19.6 as the initialization value. Then member-wise assignment copies the contents of the temporary object into myCat. This process is termed an **implicit conversion** because it happens automatically, without the need of an explicit type cast.

***Only a constructor that can be used with just one argument works as a conversion function***. The following constructor has two arguments, so it cannot be used to convert types:

Stonewt(int stn, double lbs); // not a conversion function

However, it would act as a guide to int conversion if it provided a default value for the second parameter:

Stonewt(int stn, double lbs = 0); // int-to-Stonewt conversion

As programmers acquired more experience working with C++, however, they found that the automatic aspect isn’t always desirable because it can lead to unexpected conversions. So C++ added a new keyword, ***explicit***, to **turn off the automatic(implicit) aspect**. That is, you can declare the constructor this way:

**explicit** Stonewt(double lbs); **// no implicit conversions allowed**

Stonewt myCat; // create a Stonewt object

**myCat = 19.6;** // **Implicit conversion.** **Not valid** if Stonewt(double) is declared as explicit

**mycat = Stonewt(19.6);** **// ok, this is an explicit conversion**

mycat = (Stonewt) 19.6; // **ok**, old form for explicit typecast

**Can we do the reverse?** That is, **can you convert a Stonewt object to a double value, as in the following?**

Stonewt wolfe(285.7);

**double host = wolfe; // possible ?**

**The answer is that you can do this, but not by using constructors**. ***Constructors* only provide for converting *another type* *to the class type*.** To do the reverse, you have to use a special form of a C++ operator function called a **conversion function**.

If you define a Stonewt-to-double conversion function, you can use the following conversions:

Stonewt wolfe(285.7);

double host = double (wolfe); // syntax #1

double thinker = (double) wolfe; // syntax #2

Or you can let the compiler figure out what to do:

Stonewt wells(20, 3);

**double star = wells;** **// Implicit use of conversion function, let the compiler figure out what to do**

The compiler, noting that the right side is type Stonewt and the left side is type double, looks *to see if you’ve defined a conversion function that matches this description*. (If it can’t find such a definition, the compiler generates an error message to the effect that it can’t assign a Stonewt to a double.)

**To convert a class object to type *typeName*,** you use a conversion function in this form:

operator *typeName*();

Note the following points:

1. The conversion function must be a *class method*;
2. The conversion function must *not specify a return type*;
3. The conversion function must *have no arguments*;

// stonewt1.h -- revised definition for the Stonewt class

#ifndef STONEWT1\_H\_

#define STONEWT1\_H\_

class Stonewt

{

private:

enum {Lbs\_per\_stn = 14}; // pounds per stone

int stone; // whole stones

double pds\_left; // fractional pounds

double pounds; // entire weight in pounds

public:

Stonewt(double lbs); // construct from double pounds

Stonewt(int stn, double lbs); // construct from stone, lbs

Stonewt(); // default constructor

~Stonewt();

void show\_lbs() const; // show weight in pounds format

void show\_stn() const; // show weight in stone format

// conversion functions

***operator int() const;***

***operator double() const;***

};

#endif

// conversion functions

Stonewt::operator int() const

{

return int (pounds + 0.5);

}

Stonewt::operator double()const

{

return pounds;

}

**主程序中可以这么用：**

Stonewt poppins(9,2.8); // 9 stone, 2.8 pounds

**double p\_wt = poppins; // implicit conversion**

cout << "Convert to double => ";

cout << "Poppins: " << p\_wt << " pounds.\n";

cout << "Convert to int => ";

cout << "Poppins: " << **int (poppins)** << " pounds.\n"; **// explicit conversion**

上面程序最后一行用了cout << "Poppins: " << int (poppins) << " pounds.\n"; 如果不用这里的explicit type cast:

cout << "Poppins: " << **poppins** << " pounds.\n"; **// To int or double? Compiler error due to ambiguity**

Would the program use an implicit conversion, as in the following statement?

double p\_wt = poppins;

**The answer is no. In this example, nothing indicates whether the conversion should be to int or to double.** Facing this lack of information, the compiler would complain that you were using an ambiguous conversion. Nothing in the statement indicates what type to use. **Interestingly, if the class defined only the double conversion function, the compiler would accept the statement. That’s because with only one conversion possible, there is no ambiguity.**

You can have a similar situation with assignment. With the current class declarations, the compiler rejects the following statement as being ambiguous:

long gone = poppins; **// To int or double? Compiler error due to ambiguity**

**In C++, you can assign both int and double values to a long variable**, so the compiler legitimately can use either conversion function. But if you eliminate one of the two conversion functions, the compiler accepts the statement. For example, suppose you omit the double definition. Then the compiler will use the int conversion to convert poppins to a type int value. Then it converts the int value to type long when assigning it to gone.

前面的下面一行可以；

double p\_wt = poppins; **// OK, implicit conversion**

**这是因为p\_wt本身是double的, 所以compiler知道首选double conversion.**

When the class defines two or more conversions, you can still use an explicit type cast to indicate which conversion function to use. *You can use either of these type cast notations*:

long gone = (double) poppins; // use double conversion

long gone = int (poppins); // use int conversion

The moral is that often **it’s best to use explicit conversions and exclude the possibility of implicit conversions**.

int ar[20];

...

Stonewt temp(14, 4);

...

int Temp = 1;

...

cout << ar[temp] << "!\n"; // used temp instead of Temp

The Stonewt class defines an operator int(), so the Stonewt object temp is converted to the int 200 and be used as an array index. In C++98, the keyword explicit doesn’t work with conversion functions, but C++11 removes that limitation. So with C++11, you can declare a conversion operator as explicit:

class Stonewt

{

...

// conversion functions

**explicit** operator int() const;

**explicit** operator double() const;

};

*Another approach is to replace a conversion function with a non-conversion function that does the same task—but only if called explicitly*. That is, you can *replace*

Stonewt::operator int() { return int (pounds + 0.5); }

*with*

int Stonewt::Stone\_to\_Int() { return int (pounds + 0.5); }

It allows the following:

int plb = poppins.Stone\_to\_Int();

You should use implicit conversion functions with care. Often a function that can only be invoked explicitly is the best choice.

**Conversions and Friends**

As mentioned in the discussion of the Time class, you can use either a member function or a friend function to overload addition. (To simplify matters, assume that no conversion functions of the operator double() form are defined.) You can implement addition with the following **member function**:

Stonewt Stonewt::operator+(const Stonewt & st) const

{

double pds = pounds + st.pounds;

Stonewt sum(pds);

return sum; // data copy

}

Or you can implement addition as a **friend function** this way:

Stonewt operator+(const Stonewt & st1, const Stonewt & st2)

{

double pds = st1.pounds + st2.pounds;

Stonewt sum(pds);

return sum;

}

Remember, **you can provide the method definition or the friend definition *but not both***. Either form lets you do the following:

Stonewt jennySt(9, 12);

Stonewt bennySt(12, 8);

Stonewt total;

total **= jennySt** + **bennySt**; (1)

*Also given the Stonewt(double) constructor*, each form lets you do the following:

Stonewt jennySt(9, 12);

**double** kennyD = 176.0;

Stonewt total;

total = **jennySt** + **kennyD**; (2)

**But only the friend function lets you do this:**

Stonewt jennySt(9, 12);

**double** pennyD = 146.0;

Stonewt total;

total = **pennyD** + **jennySt**; (3) **// 被加的两个数顺序和(2)中不一样**

**[解释]**

1. 相当于

total = jennySt.operator+(bennySt); // member function

*or else*

total = operator+(jennySt, bennySt); // friend function

In either case, the actual argument types match the formal arguments. Also the member function is invoked, as required, by a Stonewt object.

1. 相当于

total = jennySt.operator+(kennyD); // member function

*or else*

total = operator+(jennySt, kennyD); // friend function

Again, the member function is invoked, as required, by a Stonewt object. This time, in each case, one *argument (kennyD) is type double*, which invokes the Stonewt(double) constructor to convert the argument to a Stonewt object.

但是, having an operator double() member function defined would create confusion at this point because that would create another option for interpretation. **Instead of converting kennyD to double and performing Stonewt addition, the compiler could convert jennySt to double and perform double addition**. Having too many conversion functions creates ambiguities.

(3)

total = operator+(pennyD, jennySt);// friend function

Here, both arguments are type double, which invokes the Stonewt(double) constructor to convert them to Stonewt objects.

However, ***the member function version*** wouldn’t be able to add jennySt to pennyD. Converting the addition syntax to a function call would look like this:

total = **pennyD**.operator+(jennySt); // not meaningful

**But this is meaningless** because only a class object can invoke a member function. ***C++ does not attempt to convert pennyD to a Stonewt object***. Conversion takes place for member function **arguments**, not for member function **invokers**.

**The lesson here is that defining addition as a friend makes it easier for a program to accommodate automatic type conversions**. **The reason is that both operands become function arguments, so function prototyping comes into play for both operands.**

**Choices in Implementing Addition**

Given that you want to add double quantities to Stonewt quantities, you have a couple choices.

**The first**, as you just saw, is to define the following as a friend function and have the Stonewt(double) constructor handle conversions of type double arguments to type Stonewt arguments:

operator+(const Stonewt &, const Stonewt &)

**The second choice** is to further overload the addition operator with functions that explicitly use one type double argument:

Stonewt operator+(double x); **// member function**

***friend*** Stonewt operator+(double x, Stonewt & s);

That way, the following statement exactly matches the *operator+(double x)* member function:

total = jennySt + kennyD; // Stonewt + double,  **member function**

And the following statement exactly matches the *operator+(double x, Stonewt &s)* friend function:

total = pennyD + jennySt; // double + Stonewt, ***friend***

书上这里提了两种办法的优劣, 不太理解.

**[Summary]**

之前的Stock类比较总价的函数是这么写的:

const **Stock &** Stock::topval(const Stock & s) const

{

if (s.total\_val > total\_val) // 2nd total\_val here is actually **this->total\_val;**

return s; // argument object

else

return **\*this**; // invoking object

}

然后具体的函数调用如下:

top = stock1.topval(stock2);

top = stock2.topval(stock1);

以加操作符overloading为例:

If *district2*, *sid*, and *sara* are all objects of the Salesperson class, you can write this equation:

district2 = sid + sara;

The compiler, recognizing the operands as belonging to the *Salesperson* class, replaces the operator with the corresponding operator function:

**district2 = sid.operator+(sara);**

The function then uses the ***sid* *object implicitly*** (because it invoked the method) and the ***sara object explicitly*** (because it’s passed as an argument) to calculate the sum, which it then returns.

而本章的时间类相加是这么做的:

**Time** Time::Sum(**const** Time & t) const

{

**Time sum**;

sum.minutes = *minutes + t.minutes*;

sum.hours = *hours + t.hours* + sum.minutes / 60;

sum.minutes %= 60;

***return sum***;

}

Note that the **argument is a reference** but that **the return type is not a reference**. The reason for *making the argument a reference is efficiency*. **The return value here CANNOT be a reference**. The reason is that the function creates a new Time object (sum) that represents the sum of the other two Time objects. ***Returning the object***, as this code does, **creates a copy of the object (sum) that the calling function can use**. ***If the return type were Time &***, however, the reference would be to the sum object. ***But the sum object is a local variable and is destroyed when the function terminates***, so the reference would be a reference to a non-existent object. ***Using a Time return type***, however, means the program constructs a copy of sum before destroying it, and the calling function gets the copy.

Time coding(2, 40);

Time fixing(5, 55);

Time total;

……

***total = coding.Sum(fixing);*** // add coding and fixing

**Don’t return a reference to a local variable or another temporary object**. **Reference一般用于调用在函数之前就已经存在的参数.**

如果利用operator overloading, 上面这个函数可以写成:

Time operator+(const Time & t) const; // Class里的member declaration

**Time** Time::***operator+***(const Time & t) const

{

Time sum;

sum.minutes = minutes + t.minutes;

sum.hours = hours + t.hours + sum.minutes / 60;

sum.minutes %= 60;

**return sum**;

}

Like Sum(), **operator+() is invoked by a Time object**, takes a second Time object as an argument, and returns a Time object. Thus, you can invoke the operator+() method by using the same syntax that Sum() uses:

**total = coding.operator+(fixing);** // function notation

But naming the method operator+() also lets you use operator notation:

**total = coding + fixing;** // operator notation

The **object to the left of the operator** (coding, in this case***) is the invoking object***, and the **object to the right** (fixing, in this case) is the ***one passed as an argument***.

In short, the name of the operator+() function allows it to be invoked by using either function notation or operator notation. **The compiler uses the operand types to figure out what to do.**

**Can you add more than two objects**? For example, if t1, t2, t3, and t4 are all Time objects, can you do the following?

t4 = t1 + t2 + t3; // valid?

The way to answer this is to consider how the statement gets translated into function calls. Because addition is a left-to-right operator, the statement is first translated to this:

t4 = t1.operator+(t2 + t3); // valid?

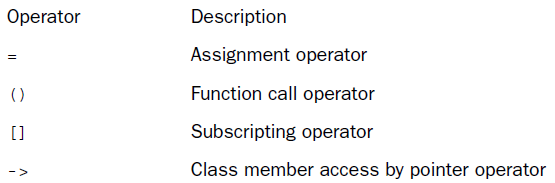
Then the function argument is itself translated to a function call, giving the following:

t4 = t1.operator+(t2.operator+(t3)); // valid? YES

**Is this valid? Yes**, it is. The function call ***t2.operator+(t3)*** returns a Time object that represents the sum of t2 and t3. This object then becomes the object of the ***t1.operator+()*** function call, and that call returns the sum of t1 and the Time object that represents the sum of t2 and t3. In short, the final return value is the sum of t1, t2, and t3, just as desired.

Overloaded operators (with some exceptions) don’t necessarily have to be member functions.

1. The overloaded operator must have at least one operand that is a user-defined type. 这个operand可以是函数参数, 也可以是隐藏的invoking object;
2. You can’t use an operator in a manner that violates the syntax rules for the original operator. For example, you can’t overload the modulus operator (%) so that it can be used with a single operand;
3. Similarly, **you can’t alter operator precedence**;
4. **You can’t create new operator symbols**. For example, you can’t define an operator\*\*() function to denote exponentiation;
5. Most of the operators in Table 11.1 can be overloaded by using either ***member*** or ***non-member*** functions. However, **you can use only member functions to overload the following operators**:



还有, overloading operator要适度. For example, you shouldn’t overload the \* operator so that it swaps the data members of two Time objects. Nothing in the notation would suggest what the operator did, so it would be better to define a class method with an explanatory name such as Swap().

// Three function declaration

**Time operator+(const Time & t) const;**

**Time operator-(const Time & t) const;**

**Time operator\*(double n) const;**

* **Introducing Friends**

Besides **C++’s control access to the private portions of a class object,** C++ also provides another form of access: the **friend**. Friends come in three varieties: **Friend functions**, **Friend classes** and **Friend member functions**.

By **making a function a friend to a class**, you allow the function the same access privileges that a member function of the class has. In the previous Time class example, the ***overloaded multiplication operator is different*** from the other two overloaded operators in that **it combines two different types(这个可能是需要对某个重载的操作符再定义一个friend的原因)**. This restricts how the operator can be used. Assume both A and B are Time class objects, and **remember, the left operand is the invoking object**.

**A = B \* 2.75;**

translates to the following member function call:

A = B.operator\*(2.75);

**But what about the following statement**?

**A = 2.75 \* B;** **// cannot correspond to a member function**

Conceptually, 2.75 \* B should be the same as B \* 2.75, **but the first expression, 2.75 \* B, cannot correspond to a member function because 2.75 is not a type Time object**. Remember, the left operand is the invoking object, but 2.75 is not an object. ***So the compiler cannot replace the expression with a member function call***.

**三种解决办法**, 一种是就这样, 只用B \* 2.75形式, 但这不符合OOP的要求; **第二种**就是***using a nonmember function.***

Remember, most operators can be overloaded using either member or nonmember functions. ***A nonmember function*** is ***not invoked by an object***; instead, **any values it uses, including objects, are explicit arguments**.

The function would have this prototype:

Time operator\*(double m, const Time & t);

注意这里参数的顺序, class member里的operator overloading默认要求是class object在左, 乘数参数在右, 所以这里的friend function就把顺序反了过来.

**注意, 这里针对\*这个操作符的问题已经提出了两个解决办法：**

1. 就这样, 用的时候注意顺序;
2. 定义一个非class member的**普通函数**, 就像上面那个, 用参数显式传递的办法计算, 然后获取一个返回参数. 但因为这个函数不是member, 所以函数里不能直接用private member, 需要调用class提供的member function去获得class的private member, 不是太方便;

比如上面这个, 调用应该是 A = operate\*(2.75, B), AB都是class Time的object; 这个函数名也不一定是operate\*(), 也可以叫MultiplyClass()之类的任何名字就可以, 和普通的函数一样取名就可以; 但名字和member里的overloading起的一样在调用起来更好看;

**不对！书中不是这个意思！**

**书中意思是说, 你已经在class中overloading了乘法操作符, 然后A = B \* 2.75这种形式就可以调用了了, B是invoking object, A接受这个操作符的返回值; 对于A = 2.75 \* B这种形式的调用, 你需要定义一个和class里的overloading一样的函数名, 然后所有涉及到的参数全部用显式传递的办法, 没有invoking object的问题.**

The function would have this prototype:

Time **operator\***(double m, const Time & t);

**这样你调用A = 2.75 \* B这种形式时, compiler在发现member里的overloading不匹配时会知道去找另一个叫operator\*的函数, 就找到了这个Time operator\*(double m, const Time & t), 注意这里的调用和member overloading的形式一样, 但实际调用的函数确是Time operator\*(double m, const Time & t)这个. 这就是书中说的:**

Remember, most operators can be overloaded using either member or nonmember functions. **A nonmember function** is **not invoked by an object**; instead, **any values it uses, including objects, are explicit arguments**.

**Thus, the compiler could match the expression**

A = 2.75 \* B; // cannot correspond to a member function

**to the following nonmember function call:**

A = operator\*(2.75, B);

**The function would have this prototype: Time operator\*(double m, const Time & t).**

**这里又引出了第三种办法: friend.**

There is a special category of ***nonmember*** functions, called ***friends***, which can access private members of a class. **注意, friend不是member function,但却可以直接调用private member**, 这样就解决了方法二中的调用private member不方便的问题.

**Creating Friends**

The first step toward creating a friend function is to place a prototype **in the class declaration** and prefix the declaration with the keyword ***friend***:

**friend** Time **operator\***(double m, const Time & t); // goes **in class declaration**

This prototype has two implications:

1. Although the operator\*() function is declared in the class declaration, **it is not a member function**. ***So it isn’t invoked by using the membership operator;***
2. Although the operator\*() function is not a member function, it has the same access rights as a member function;

Time **operator\***(double m, const Time & t) // friend prefix not used in definition

{

Time result;

long totalminutes = t.hours \* mult \* 60 +t. minutes \* mult;

result.hours = totalminutes / 60;

result.minutes = totalminutes % 60;

return result;

}

Because it is not a member function, you don’t use the Time:: qualifier. Also you don’t use the friend keyword in the definition. With this declaration and definition, the statement

A = 2.75 \* B;

translates to the following and invokes the nonmember friend function just defined:

**A = operator\*(2.75, B);**

In short, a friend function to a class is a nonmember function that has the same access rights as a member function.

**方法三和方法二的区别在于方法三的声明在class member里, 并且前面有friend字样; 他们俩的definition都一样, 无非是方法二需要通过调用public member去获得private member, 而方法三可以直接调用. 注意方法二和三的参数顺序.**

**还有第四种办法:**

You can **write** **this particular friend function** as a **non-friend** by altering the definition so that it switches which value comes first in the multiplication.

Time operator\*(double m, const Time & t)

{

return t \* m; // **use t.operator\*(m)，其实调用了之前的operator\*的overload;**

}

第三种办法 accessed *t.minutes* and *t.hours* explicitly, so it had to be a friend. 第四种办法 only ***uses the Time object t as a whole***, letting *a member function handle the private values*, **so this version doesn’t have to be a friend**.

注意, 这个函数不需要是friend. 但是也可以把它声明成friend, 一是可以统一class的interface, 第二是如果将来需要用到class内部的private member, 就可以直接用了.

* **A Common Kind of Friend: Overloading the << Operator**

当要打印一个class内部的member时, 需要调用之前提到的show()函数, 这个函数里面会cout各个成员变量. 重载<<操作符的目的是可以用下面的形式打印出所有的成员变量:

cout << trip; // make cout recognize Time class object trip

1. **重载<<operator的第一种办法**

**第一种办法是用friend.** 为什么？

<<的用法通常如下:

cout << trip;

这种形式涉及到了两个object, 其中cout对应的ostream class object是第一个.

**如果用member function overloading**, 就像前面提到的对\*的重载一样, **那么invoking object会是第一个implicit参数**, 这时调用会变成:

trip << cout; // **if operator<<() were a Time member function**

**如果用friend function**, you can overload the operator this way:

**void** operator<<(**ostream & os**, **const Time & t**)

{

os << t.hours << " hours, " << t.minutes << " minutes";

}

This lets you use

cout **<<** trip;

<<左边的是overload定义里的第一个参数, 右边的是overload定义里的第二个参数; cout是ostream的一个object; **cout << trip其实是被编译成operator<< (cout, trip);**

Because operator<<() accesses private Time object members directly, it has to be a friend to the Time class. But because it does not directly access private ostream object members but only uses the ostream object as a whole, the function does not have to be a friend to the ostream class.

The call cout << trip should use the cout object itself, not a copy, so the function passes the object as a reference instead of by value. Thus, the expression cout << trip causes *os* to be an alias for *cout*, and the expression cerr << trip causes os to be an alias for cerr. The Time object can be passed by value or by reference because either form makes the object values available to the function.

1. **重载<<operator的第二种办法**

上面的方法有个问题: That implementation doesn’t allow you to combine the redefined << operator with the ones cout normally uses:

cout << "Trip time: " << trip << " (Tuesday)\n"; // can't do

**原因:** 前面overload后的 << 左边必须是个ostream的object, 右边必须是个Time class的object; 而在上面这句里, 实际的操作流程是这样的:

(((cout << "Trip time: ") << trip) << " (Tuesday)\n");

第一次调用<<, 右边是character constant : "Trip time: ", 不是Time class的object, **这个没问题, C++自己知道这时不应该用你overload的函数, 而用cout自己针对<<的定义**; 但这二次针对trip调用完后函数没有返回, 因为overload时函数类型是void, **那么第三次针对Tuesday调用时<<左边是个void, 这不行**;

**解决办法:**

You can take the same approach with the friend function. You just revise the operator<<() function so that it returns a reference to an ostream object:

***ostream &*** operator<<(**ostream & os**, const Time & t)

{

os << t.hours << " hours, " << t.minutes << " minutes";

**return os;**

}

这样解决了上面第二个<<没有返回值的问题;

第一步, cout << "Trip time: ", invokes the particular **ostream definition of <<** that displays a string and returns the cout object reference;

第二步, cout << trip, the program uses the **Time declaration of <<** to display the trip values and to return the cout object reference again;

这一章的总结的第11页给出了\*和<<这两个操作符重载的声明和定义:

public:

Time operator\*(double n) const;

friend Time **operator\***(double m, const Time & t) { return t \* m; } **// inline definition**

**friend std::ostream & operator<<(std::ostream & os, const Time & t);**

定义是:

**Time Time::operator\***(double mult) const

{

Time result;

long totalminutes = hours \* mult \* 60 + minutes \* mult;

result.hours = totalminutes / 60;

result.minutes = totalminutes % 60;

return result;

}

**std::ostream & operator<<**(std::ostream & os, const Time & t) // friend definition doesn’t have prefix friend；

{

os << t.hours << " hours, " << t.minutes << " minutes";

return os;

}