

11 使用类

- 运算符重载
- 友元函数
- 重载<<

■ 也许学习本章知识的最好方法

- 在开发的C++程序中使用其中的新特性。对这些新特性有了充分的认识后，就可以添加其他C++特性了。需要更全面了解C++的机制
- 不要觉得必须使用所有特性
- 不要在第一次学习时就试图使用所有特性

运算符重载

- 一种使对象操作更美观的技术
 - 目的：简化操作，符合人们日常使用操作符的习惯
 - 做法：重载操作符
 - 函数重载的一种：名称相同但特征标（参数列表）不同的函数
- 如，数组的加法做简化

```
1. for (int i = 0; i < 20; i++)  
2.     evening[i] = sam[i] + janet[i]; // add element by element  
3. // 简化成  
4. evening = sam + janet; // add two array objects
```

- 这种简单的加法表示法隐藏了内部机理，并强调了实质，是OOP的目标之一

- 运算符函数的格式
 - operator op(argument-list)
 - op有效的C++运算符: + []

- C++内部表达

```
1. district2 = sid + sara;           // 重载形式
2. district2 = sid.operator+(sara); // 编译器的内部表达!
```

计算时间：一个运算符重载示例

2 计算时间：一个运算符重载示例

- [P11.1 mytime0.h](#), [P11.2 mytime0.cpp](#)

- 时间加法的计算

- 常规的做法

```
1. Time Sum(const Time &t) const;
2. total = coding.Sum(fixing);
```

- [P11.3 usetime0.cpp](#)

```
1. class Time{
2.     private:
3.         int hours;
4.         int minutes;
5.     public:
6.         Time();
7.         Time(int h, int m = 0);
8.         void AddMin(int m);
9.         void AddHr(int h);
10.        void Reset(int h = 0, int m = 0);
11.        const Time Sum(const Time &t) const;
12.        void Show() const;
13.};
```

2.1 重载加法运算符

- [P11.4 mytime1.h](#), [P11.5 mytime1.cpp](#),
[P11.6 usetime1.cpp](#)

```
1. Time operator+(const Time & t) const;  
2. total = coding + fixed;
```

```
1. class Time{private:  
2.     int hours;  
3.     int minutes;  
4. public:  
5.     Time();  
6.     Time(int h, int m = 0);  
7.     Time operator+(const Time & t) const;  
8.     void Show() const;  
9. };  
  
10. Time Time::operator+(const Time & t) const  
11. {  
12.     Time sum;  
13.     sum.minutes = minutes + t.minutes;  
14.     sum.hours = hours + t.hours + sum.minutes / 60;  
15.     sum.minutes %= 60;  
16.     return sum;  
17. }
```

- 多数C++运算符(表11.1)可以重载
- 要求
 - 要求至少一个操作数是用户自定义
 - 不改变原有句法规则
 - 不能创建新的运算符
 - 不能重载的运算符....
 - 只能通过成员函数进行重载

Operator	Description
sizeof	The sizeof operator
.	The membership operator
.*	The pointer-to-member operator
::	The scope-resolution operator
?:	The conditional operator
typeid	An RTTI operator
const_cast	A type cast operator
dynamic_cast	A type cast operator
reinterpret_cast	A type cast operator
static_cast	A type cast operator

Operator	Description
=	Assignment operator
()	Function call operator
[]	Subscripting operator
->	Class member access by pointer operator

Table 11.1 Operators That Can Be Overloaded

+	-	*	/	%	^
&		~	!	=	<
>	+=	-=	*=	/=	%=
^=	&=	=	<<	>>	>>=
<<=	==	!=	<=	>=	&&
	++	--	,	->*	->
()	[]	new	delete	new []	delete []

2.3 其他重载运算符

■ -和*

1. Time **operator**-(const Time &t) const;
2. Time **operator***double n) const;

友元

■ 为什么需要友元？

- 公有的类方法是访问类成员的唯一途径，但是有时候这种限制太严格
- 某些特定情况，需要访问私有成员

■ 友元有3种

- 友元函数
- 友元类
- 友元成员函数

■ 【问题】：Time类示例中

- $A = B * 2.75;$

■ 将被转换为下面的成员函数调用：

- $A = B.operator*(2.75);$
- $A = 2.75 * B;$ //不能处理成类函数

■ 编译器不能使用成员函数调用替换该表达式

■ 告知开发者只能按 $B * 2.75$ 这种格式编写，不能写成 $2.75 * B$ 。

■ 采用非成员函数。本章采用这种方案。

- $A = operator*(2.75, B);$

■ 该函数的原型：

- $Time operator*(double m, const Time \&t);$
 - //该函数需要访问Time数据成员

■ 【新问题】

- 常规的非成员函数不能直接访问类的私有数据

■ 有一类特殊的非成员函数：友元函数

- 可以访问类的私有成员

3.1 创建友元

- 1. 将其原型放在类声明中，并在原型声明前加上关键字friend
 - friend Time operator*(double m, const Time &t); // goes in class declaration
- 2. 编写函数

```
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;
}
```

- 有了上述声明和定义后，下面的语句：
 - A = 2.75 * B;
- 将转换为如下语句，从而调用刚才定义的非成员友元函数：
 - A = operator*(2.75, B);

3.2 常用的友元：重载<<运算符

- [P11.10 mytime3.h](#), [P11.11 mytime3.cpp](#), [P11.12 usetime3.cpp](#)
- 1. <<的第一种重载版本
- 2. <<的第二种重载版本
- 为了支持连续的<<操作，必须让<<返回ostream&，以支持连续cout
 - `cout << a << b;`

```
cout << trip;  
void operator<<(ostream &os, const Time &t)  
{  
    os << hours << " hours, " << t.minutes  
    << "minutes";  
}
```

```
cout << "Trip time: " << trip << " (Tuesday)\n";  
ostream &operator<<(ostream &os, const Time &t)  
{  
    os << t.hours << " hours, " << t.minutes <<  
    " minutes";  
    return os;  
}
```

重载运算符：作为成员函数还是非成员函数

4 重载运算符：作为成员函数还是非成员函数

- operator+ 存在两种方式
 - 成员函数
 - Time operator+(const Time & t) const;
 - 非成员函数
 - friend Time operator+(const Time & t1, const Time & t2);

- 两者都可以，只能选其中一个
 - 同时定义这两种格式，将被视为二义性错误，导致编译错误

再谈重载：一个矢量类

5 再谈重载：一个矢量类

- 表示
 - 分量 x, y （直角坐标系）
 - 长度和方向（极坐标）
- 运算
- 示例5
- [P11.13 vect.h](#), [P11.14 vect.cpp](#), [P11.15 randwalk.cpp](#)

- mode, 控制不同的表示方式
 - 控制使用构造函数、reset()方法和重载的operator<<() 函数使用哪种形式

- +
 - 返回局部变量
 - 返回构造函数

- `Vector Vector::operator+(const Vector &b) const`
- {
- `return Vector(x + b.x, y + b.y); // return the constructed Vector`
- }

- 有一些信息
 - 可以作为成员存在，提前计算好，需要时直接返回成员
 - 不作为成员，返回时临时计算
- Vector对象中存储了矢量的直角坐标和极坐标，但公有接口并不依赖于这一事实
 - 将接口与实现分离是OOP 的目标之一，这样允许对实现进行调整，而无需修改使用这个类的代码
 - 两种坐标系可以都保留一份数据，也可以只保留一份，需要另一份数据实时进行计算
 - 涉及到时间（计算）和空间（存储）的权衡/取舍

5.4 使用vector类来模拟随机漫步

- 醉鬼走路问题
 - 随机走动
 - 距离原始位置不超过给定长度
- 随机生成角度，长度是输入的

类的自动转换和强制类型转换

6 类的自动转换和强制类型转换

- [P11.16 stonewt.h](#), [P11.17 stonewt.cpp](#), [P11.18 stone.cpp](#)
- 用构造函数来转换，隐式转换
 - `Stonewt Jumbo(7000);` // uses `Stonewt(double)`, converting int to double
 - `Jumbo = 7300;` // uses `Stonewt(double)`, converting int to double
- 用于转换的构造函数只能有一个参数
- `explicit`关闭隐式转换
 - `explicit Stonewt(double lbs);` // no implicit conversions allowed
- 但仍然允许显式转换，即显式强制类型转换：
 - `Stonewt myCat;` // create a `Stonewt` object
 - `myCat = 19.6;` // not valid if `Stonewt(double)` is declared as `explicit`
 - `mycat = Stonewt(19.6);` // ok, an explicit conversion
 - `mycat = (Stonewt)19.6;` // ok, old form for explicit typecast

- 把对象转成别的类型
 - `operator int();`
 - 谨慎地使用隐式转换函数
-
- [P11.19 stonewt1.h](#), [P11.20 stonewt1.cpp](#), [P11.21 stone1.cpp](#)

6.2 转换函数和友元函数

- 重载运算有两种方式

- 一般来说，访问私有类成员的唯一方法是使用类方法。C++使用友元函数来避开这种限制
- C++扩展了对运算符的重载，允许自定义特殊的运算符函数，这种函数描述了特定的运算符与类之间的关系。运算符函数可以是类成员函数，也可以是友元函数
- C++允许指定在类和基本类型之间进行转换的方式