# 11.2 计算时间：一个运算符重载示例

如果今天早上在Priggs的账户上花费了2小时35分钟，下午又花费了2小时40分钟，则总共花了多少时间呢？这个示例与加法概念很吻合，但要相加的单位（小时与分钟的混合）与内置类型不匹配，第7章通过定义一个travel\_time结构和将这种结构相加的sum()函数来处理类似的情况。现在将其推广，采用一个使用方法来处理加法的Time类。首先使用一个名为Sum()的常规方法，然后介绍如何将其转换为重载运算符。程序清单11.1列出了这个类的声明。

**程序清单11.2 mytime0.cpp**

// mytime0.h -- 使用运算符重载前的Time类

#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);

const Time Sum(const Time & t) const;

void Show() const;

};

#endif

Time类提供了用于调整和重新设置时间、显示时间、将两个时间相加的方法。程序清单11.2列出了方法定义。请注意，当总的分钟数超过59时，AddMin()和Sum()方法是如何使用整数除法和求模运算符来调整minutes和hours值的。另外，由于这里只使用了iostream的cout，且只使用了一次，因此使用std:cout比导入整个名称空间更经济。

**程序清单11.2 mytime0.cpp**

// mytime0.cpp -- 实现Time类方法

#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;

}

const 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";

}

来看一下Sum()函数的代码。注意参数是引用，但返回类型却不是引用。将参数声明为引用的目的是为了提高效率。如果按值传递Time对象，代码的功能将相同，但传递引用，速度将更快，使用的内存将更少。

然而，返回值不能是引用。因为函数将创建一个新的Time对象（sum），来表示另外两个Time对象的和。返回对象（如代码所做的那样）将创建对象的副本，而调用函数可以使用它。然而，如果返回类型为Time &，则引用的将是sum对象。但由于sum对象是局部变量，在函数结束时将被删除，因此引用将指向一个不存在的对象。使用返回类型Time意味着程序将在删除sum之前构造它的拷贝，调用函数将得到该拷贝。

警告：不要返回指向局部变量或临时对象的引用。函数执行完毕后，局部变量和临时对象将消失，引用将指向不存在的数据。

最后，程序清单11.3对Time类中计算时间总和的部分进行了测试。

**程序清单11.3 usetime0.cpp**

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

// 将usetime0.cpp和mytime0.cpp一起编译

#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);

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

total.Show();

cout << endl;

std::cin.get();

return 0;

}

下面是程序清单11.1、程序清单11.2和程序清单11.3组成的程序的输出：

planning time = 0 hours,0 minutes

coding time = 2 hours, 40 minutes

fixing time = 5 hours, 55 minutes

coding.Sum(fixing) = 8 hours, 35 minutes

## 11.2.1 添加加法运算符

将Time类转换为重载的加法运算符很容易，只要将Sum()的名称改为operator+()即可。这样做是对的，只要把运算符（这里为+）放到operator的后面，并将结果用作方法名即可。在这里，可以住标识符中使用字母、数字或下划线之外的其他字符。程序清单11.4和程序清单11.5反映了这些细微的修改。

**程序清单11.4 mytime1.h**

// mytime1.h -- 运算符重载前的Time类

#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

**程序清单11.5 mytime1.cpp**

// mytime1.cpp -- 实现Time的方法

#include <iostream>

#include "mytime1.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::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";

}

和Sum()一样，operator+()也是由Time对象调用的，它将第二个Time对象作为参数，并返回一个Time对象。因此，可以像调用Sum()那样来调用operator+()方法：

total = coding.operaror+(fixing); // 函数表示法

但将该方法命令为operator+()后，也可以使用运算符表示法：

total = coding + fixing; // 运算符表示法

这两种表示法都将调用operator+()方法。注意，在运算符表示法中，运算符左侧的对象（这里为coding）是调用对象，运算符右边的对象（这里为fixing）是作为参数被传递的对象。程序清单11.6说明了这一点。

**程序清单11.6 usetime1.cpp**

// usetime1.cpp -- 使用Time类的第2种形式

// 将usetime1.cpp和mytime1.cpp编译在一起

#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;

cout << "planning time = ";

planning.Show();

cout << endl;

cout << "coding time = ";

coding.Show();

cout << endl;

cout << "fixing time = ";

fixing.Show();

cout << endl;

total = coding + fixing; // 运算符表示法

cout << "coding + fixing = ";

total.Show();

cout << endl;

Time morefixing(3, 28);

cout << "more fixing time = ";

morefixing.Show();

cout << endl;

total = morefixing.operator+(total); // 函数表示法

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

total.Show();

cout << endl;

std::cin.get();

return 0;

}

下面是程序清单11.4～程序清单11.6组成的程序的输出。

planning time = 0 hours, 0 minutes

coding time = 2 hours, 40 minutes

fixing time = 5 hours,55 minutes

coding + fixing = 8hours, 35 minutes

more fixing time = 3 hours, 28 minutes

morefixing.operator+(total) = 12 hours,3 minutes

总之，operator+()函数的名称使得可以使用函数表示法或运算符表示法来调用它。编译器将根据操作数的类型来确定如何做：

int a, b, c,

Time A, B, C;

c = a + b; // 使用int相加

c = A + B; // 使用用于Time对象的相加

可以将两个以上的对象相加吗？例如，如果t1、t2、t3和t4都是Time对象，可以这样做吗：

t4 = t1 + t2 + t3; // 可以吗？

为回答这个问题，来看一些上述语句将被如何转换为函数调用。由于+是从左向右结合的运算符，因此上述语句首先被转换成下面这样：

t4 = t1.operator+(t2+t3);

然后，函数参数本身被转换成一个函数调用，结果如下：

t4 = t1.operaror+(t2.operator+（t3)); // 可以

上述语句合法吗？是的。函数调用t2.operator+(t3)返回一个Time-对象，后者是t2和t3的和。然而，该对象成为函数调用t1.operator+()的参数，该调用返回t1与表示t2和t3之和的Time对象的和。总之，最后的返回值为t1、t2和t3之和，这正是我们期望的。

## 11.2.2 重载限制

多数C++运算符（参见表11.1）都可以用这样的方式重载：重载的运算符（有些例外情况）不必是成员函数，但必须至少有一个操作数是用户定义的类型。下面详细介绍C++对用户定义的运算符重载的限制。

1．重载后的运算符必须至少有一个操作数是用户定义的类型，这将防止用户为标准类型重载运算符。因此，不能将减法运算符（-）重载为计算两个double值的和，而不是它们的差。虽然这种限制将对创造性有所影响，但可以确保程序正常运行。

2．使用运算符时不能违反运算符原来的句法规则。例如，不能将求模运算符（%）重载成使用一个操作数：

int x;

Time shiva;

% x; // 无效的求模运算

% shiva; // 无效的重载运算符

同样，不能修改运算符的优先级。因此，如果将加号运算符重载成将两个类相加，则新的运算符与原来的加号具有相同的优先级。

3．不能创建新运算符。例如，不能定义operator\*\*()函数来表示求幂。

4．不能重载下面的运算符。

* sizeof：sizeof运算符。
* ．：成员运算符。
* \*：成员指针运算符。
* ::：作用域解析运算符。
* ?:：条件运算符。
* rypeid：一个RTTI运算符。
* const\_cast：强制类型转换运算符。
* dynamic\_cast：强制类型转换运锋符。
* reinterpret\_cast：强制类型转换运算符。
* static\_cast：强制类型转换运算符。

然而，表11.1中所有的运算符都可以被重载。

5．表11.1中的大多数运算符都可以通过成员或非成员函数进行重载，但下面的运算符只能通过成员函数进行重载。

* =：赋值运算符。
* ()：函数调用运算符。
* []：下标运算符。
* ->：通过指针访问类成员的运算符。

注意：本章不介绍这里列出的所有运算符，但附录E对本书正文中没有介绍的运算符进行了总结。

**表11.1 可重载的运算符**

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

除了这些正式限制之外，还应在重载运算符时遵循一些明智的限制。例如，不要将\*运算符重载成交换两个Time对象的数据成员。表示法中没有任何内容可以表明运算符完成的工作，因此最好定义一个其名称具有说明性的类方法，如Swap()。

## 11.2.3 其他重载运算符

还有一些其他的操作对Time类来说是有意义的。例如，可能要将两个时间相减或将时间乘以一个因子，这需要重载减法和乘法运算符。这和重载加法运算符采用的技术相同，即创建operator-()和operator\*()方法。也就是说，将下面的原型添加到类声明中：

Time operator-(const Time & t) const;

Time operator\*(double n) const;

程序清单11.7是新的头文件。

**程序清单11.7 mytime2.h**

// mytime2.h -- 使用函数重载后的Time类

#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

然后将新增方法的定义添加到实现文件中，如程序清单11.8所示。

**程序清单11.8 mytime2.cpp**

// mytime2.cpp -- Time类方法的实现

#include <iostream>

#include "mytime2.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::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;

}

void Time::Show() const

{

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

}

完成上述修改后，就可以使用程序清单11.9中的代码来测试新定义了。

**程序清单11.9 usetime2.cpp**

// usetime2.cpp -- 使用Time类的第3种形式

// 将usetime2.cpp和mytime2.cpp一起编译

#include <iostream>

#include "mytime2.h"

int main()

{

using std::cout;

using std::endl;

Time weeding(4, 35);

Time waxing(2, 47);

Time total;

Time diff;

Time adjusted;

cout << "weeding time = ";

weeding.Show();

cout << endl;

cout << "waxing time = ";

waxing.Show();

cout << endl;

cout << "total work time = ";

total = weeding + waxing; // 使用operator+()

total.Show();

cout << endl;

diff = weeding - waxing; // 使用operator-()

cout << "weeding time - waxing time = ";

diff.Show();

cout << endl;

adjusted = total \* 1.5; // 使用operator\*()

cout << "adjusted work time = ";

adjusted.Show();

cout << endl;

// std::cin.get();

return 0;

}

下面是程序清单11.7～程序清单11.9组成的程序得到的输出：

weeding time = 4 hours, 35 minutes

waxing time = 2 hours, 47 minutes

total work time = 7 hours, 22 minutes

weeding time - waxing time = 1 hours, 48 minutes

adjusted work time = 11 hours, 3 minutes