chapter11 使用类

1）运算符重载

> [1] 格式：operator op (argument-list)

```
operator+ ()
operator* ()
operator[] ()

ps: op 必须是有意义的c++运算符，比如 @ 就无效
```

> [2] 本质：类对象相加的函数表示法

```
eg: Salesperson类重载了一个+运算符
d2 = sid + sara;
<=>
d2 = sid.operator+ (sara);
```

【1】实战展示说明：

（1）常规操作：【注意函数 sum( ) 】

```
// mytime0.h -- Time class before operator overloading
#ifndef MYTIME0_H_
#define MYTIME0_H_
class Time
{
private:
    int hours;
    int minutes;
public:
    Time();                             // 默认构造函数
    Time(int h, int m = 0);             // 构造函数
    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
```

```
// mytime0.cpp  -- implementing Time methods
#include <iostream>
#include "mytime0.h"
Time::Time()                    默认构造函数
{
    hours = minutes = 0;
}

Time::Time(int h, int m )       构造函数
{
    hours = h;
    minutes = m;
}

void Time::AddMin(int m)        min相加函数
```

```
{
    minutes += m;
    hours += minutes / 60;     poi!
    minutes %= 60;             poi!
}

void Time::AddHr(int h)        hour相加函数
{
    hours += h;
}

void Time::Reset(int h, int m) 重新设置函数
{
    hours = h;
    minutes = m;
}

const Time Time::Sum(const Time & t) const     求时间总量(hours,minutes)函数
{
    Time sum;
    sum.minutes = minutes + t.minutes;
    sum.hours = hours + t.hours + sum.minutes / 60;
    sum.minutes %= 60;
    return sum;
}

void Time::Show() const                        展示函数
{
    std::cout << hours << " hours, " << minutes << " minutes";
}
```

analysis：看一下Sum()函数的代码：
1）注意这里参数是引用：
目的是提高效率，如果按值传递Time对象也可以，但是传递引用的速度会更快，使用的内存会更少

2）返回值不能是引用：
因为函数**将创建一个新的Time类对象(sum)来表示另外两个Time类对象之和，返回对象(正如上述代码中)将创建对象的副本，而调用函数可以使用它。**然而，如果返回类型是Time&，则引用的是sum对象本体！但是sum对象本身是局部变量，在函数结束时就被删除，因此引用将指向一个不存在的对象。**使用返回类型Time意味着程序在删除sum之前构造它的拷贝，调用函数将得到该拷贝。**

3）const Time Time::Sum(const Time & t) const 说明：

```
  const Time   Time::Sum(const Time & t)      const
 ------------  --------------------------   ------
   返回值类型   函数头（参数）                不改变被引用对象
                要加作用域符说明属于类
```

```
// usetime0.cpp -- using the first draft of the Time class
// compile usetime0.cpp and mytime0.cpp together
#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;
}
```

（2）基于上述sum()函数说明何为“+重载“：

只需要将sum()改成operator+ (...)即可

```
// mytime1.h -- Time class before operator overloading
#ifndef MYTIME1_H_
#define MYTIME1_H_
class Time
{
private:
    int hours;
    int minutes;
public:
    ......
    Time operator+(const Time & t) const;
};
#endif
```

```
......
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;
}
```

```
// usetime1.cpp -- using the second draft of the Time class
// compile usetime1.cpp and mytime1.cpp together
#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 = ";
```

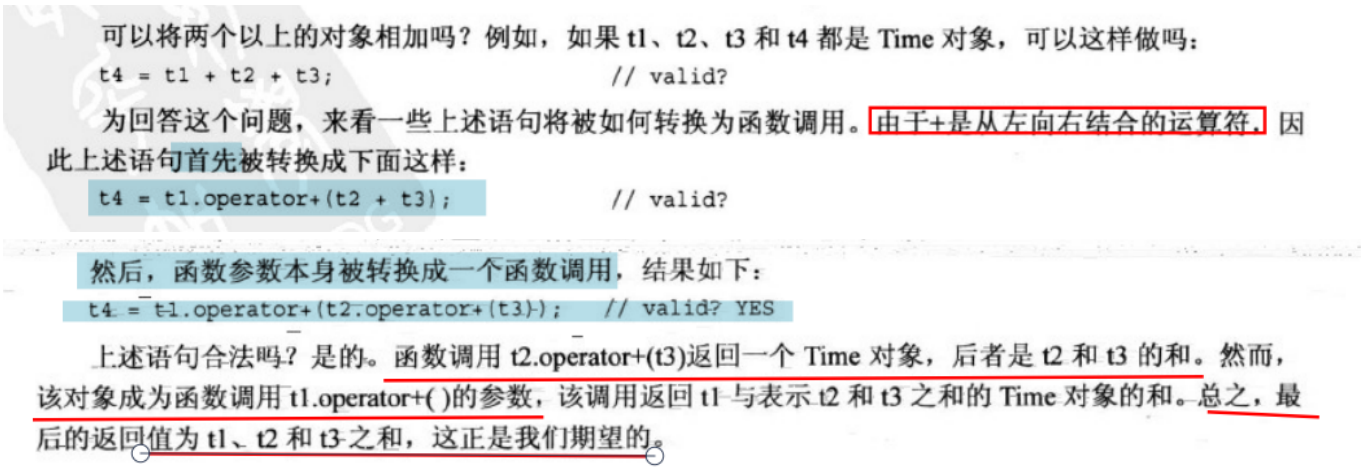
```
    coding.Show();
    cout << endl;

    cout << "fixing time = ";
    fixing.Show();
    cout << endl;

    total = coding + fixing;              重载+ 的直接使用
    // 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);   重载+ 的原理使用
    // function notation
    cout << "morefixing.operator+(total) = ";
    total.Show();
    cout << endl;
    // std::cin.get();
    return 0;
}
```

analysis：
和sum()一样，operator+()也是由Time类对象调用的：

```
total = coding.operator+ (fixing);
<=>
total = coding + fixing;
```

> ps: 在运算符表示法中，运算符左侧的对象是调用对象[隐式]，右侧的对象是作为参数被传递的对象[显式]，注意顺序！

（3）多对象相加的本质：



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;                    // valid?
```

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);           // valid?
```

然后，函数参数本身被转换成一个函数调用，结果如下：

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

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。

（4）类比上述sum()函数考察其他“重载“： - 与 *

```
// mytime2.h -- Time class after operator overloading
#ifndef MYTIME2_H_
#define MYTIME2_H_
class Time
{
private:
    int hours;
    int minutes;
public:
```

```
    ...
    Time operator-(const Time & t) const;
    Time operator*(double n) const;
    ...
};
#endif
```

```
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;
}
```

```
// usetime2.cpp -- using the third draft of the Time class
// compile usetime2.cpp and mytime2.cpp together
#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;   // use operator+()
    total.Show();
    cout << endl;

    diff = weeding - waxing;    // use operator-()
    cout << "weeding time - waxing time = ";
    diff.Show();
    cout << endl;

    adjusted = total * 1.5;      // use operator+()
    cout << "adjusted work time = ";
    adjusted.Show();
    cout << endl;
    // std::cin.get();
```

```
    return 0;
}
```

【2】重载的限制：

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;       // invalid for modulus operator
% shiva;   // invalid for overloaded operator
```

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。
- typeid：一个 RTTI 运算符。
- const_cast：强制类型转换运算符。
- dynamic_cast：强制类型转换运算符。
- 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。

2）友元

【1】简介：
c++控制对类对象私有部分的访问，通过让函数成为类的友元，可以赋予该函数与类成员函数相同的访问权限

【2】问题引入：

介绍如何成为友元前，先介绍为何需要友元。在为类重载二元运算符时（带两个参数的运算符）常常需要友元。将 Time 对象乘以实数就属于这种情况，下面来看看。

在前面的 Time 类示例中，重载的乘法运算符与其他两种重载运算符的差别在于，它使用了两种不同的类型。也就是说，加法和减法运算符都结合两个 Time 值，而乘法运算符将一个 Time 值与一个 double 值结合在一起。这限制了该运算符的使用方式。记住，左侧的操作数是调用对象。也就是说，下面的语句：

```
A = B * 2.75;
```

将被转换为下面的成员函数调用：

```
A = B.operator*(2.75);
```

但下面的语句又如何呢？

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

从概念上说，2.75 * B 应与 B *2.75 相同，但第一个表达式不对应于成员函数，因为 2.75 不是 Time 类型的对象。记住，左侧的操作数应是调用对象，但 2.75 不是对象。因此，编译器不能使用成员函数调用来替换该表达式。

解决这个难题的一种方式是，告知每个人（包括程序员自己），只能按 B * 2.75 这种格式编写，不能写成 2.75 * B。这是一种对服务器友好-客户警惕的（server-friendly, client-beware）解决方案，与 OOP 无关。

然而，还有另一种解决方式——非成员函数（记住，大多数运算符都可以通过成员或非成员函数来重载）。非成员函数不是由对象调用的，它使用的所有值（包括对象）都是显式参数。这样，编译器能够将下面的表达式：

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

与下面的非成员函数调用匹配：

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

该函数的原型如下：

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

对于非成员重载运算符函数来说，运算符表达式左边的操作数对应于运算符函数的第一个参数，运算符表达式右边的操作数对应于运算符函数的第二个参数。而原来的成员函数则按相反的顺序处理操作数，也就是说，double 值乘以 Time 值。

使用非成员函数可以按所需的顺序获得操作数（先是 double，然后是 Time），但引发了一个新问题：非成员函数不能直接访问类的私有数据，至少常规非成员函数不能访问。然而，有一类特殊的非成员函数可以访问类的私有成员，它们被称为友元函数。

【3】创建友元：

（1）将其原型放在类声明中的，并在原型声明之前加上 **关键字friend**

```
class Time
{
private:
        ...
public:
        ......
        friend Time operator* (double m,const Time& t)
};

含义：
（1）虽然 operator* 函数是在类中声明的，但它不是成员函数，因此不能使用成员运算符来使用
（2）虽然 operator* 不是成员函数，但它与成员函数的访问权限相同
```

（2）编写函数定义

【类内声明+类外定义】

[1] 因为它不是成员函数，所以不要用 Time: : 限定符（限定符意思是 类自家人 属性，友元函数本质上还是外来人口）

[2] 定义中不用写friend

```
Time operator* (double m,const Time& t)
{
        Time result;
        long total_min = t.hours*m*60 + t.minutes*m;
        result.hours = total_min / 60;
        result.minutes = total_min % 60;
        return result;
}
```

【类内声明+类内定义】
[1] 此时：声明 (也叫作：原型) 与定义同步，类内“声明+定义”时要**friend**

```
class Time
{
private:
        ...
public:
        friend Time operator* (double m,const Time& t)
        {
                return t*m;
        }
        ......
};
```

【4】常用友元：重载<<运算符
（1）问题引入：

一个很有用的类特性是，可以对<<运算符进行重载，使之能与 cout 一起来显示对象的内容。与前面介绍的示例相比，这种重载要复杂些，因此我们分两步（而不是一步）来完成。

假设 trip 是一个 Time 对象。为显示 Time 的值，前面使用的是 Show( )。然而，如果可以像下面这样操作将更好：

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

之所以可以这样做，是因为<<是可被重载的 C++运算符之一。实际上，它已经被重载很多次了。最初，<<运算符是 C 和 C++的位运算符，将值中的位左移（参见附录 E）。ostream 类对该运算符进行了重载，将其转换为一个输出工具。前面讲过，cout 是一个 ostream 对象，它是智能的，能够识别所有的 C++基本类型。这是因为对于每种基本类型，ostream 类声明中都包含了相应的重载的 operator<<( )定义。也就是说，一个定义使用 int 参数，一个定义使用 double 参数，等等。因此，要使 cout 能够识别 Time 对象，一种方法是将一个新的函数运算符定义添加到 ostream 类声明中。但修改 iostream 文件是个危险的主意，这样做会在标准接口上浪费时间。相反，通过 Time 类声明来让 Time 类知道如何使用 cout。

（2）重载<<的版本一：

> **要让Time类知晓使用cout，必须使用友元函数：**

常规调用是：cout << trip ;
如果使用一个Time类成员函数来重载<<，Time对象会是第一个操作数，这意味着必须写成这种形式：trip << cout;
that's rediculous!
但是通过友元函数就可以写出合理形式：

```
void operator<< (ostream& os , const Time& t)
{
        os << t.hours << "asjdhksadgh" << t.minutes << "sahk";
        //注意：返回值是void，因此没有return!
}
```

如此可以写出: cout << trip;

ps：

【1】这是Time类的友元函数，但不一定是ostream类的友元（原则上不允许修改ostream类！）

新的 Time 类声明使 operatro<<( )函数成为 Time 类的一个友元函数。但该函数不是 ostream 类的友元(尽管对 ostream 类并无害处)。operator<<( )函数接受一个 ostream 参数和一个 Time 参数，因此表面看来它必须同时是这两个类的友元。然而，看看函数代码就会发现，尽管该函数访问了 Time 对象的各个成员，但从始至终都将 ostream 对象作为一个整体使用。因为 operator<<( )直接访问 Time 对象的私有成员，所以它必须是 Time 类的友元。但由于它并不直接访问 ostream 对象的私有成员，所以并不一定必须是 ostream 类的友元。这很好，因为这就意味着不必修订 ostream 的定义。

注意，新的 operator<<( )定义使用 ostream 引用 os 作为它的第一个参数。通常情况下，os 引用 cout 对象，如表达式 cout << trip 所示。但也可以将这个运算符用于其他 ostream 对象，在这种情况下，os 将引用相应的对象。

【2】推荐写法：传递引用！

调用 cout << trip 应使用 cout 对象本身，而不是它的拷贝，因此该函数按引用（而不是按值）来传递该对象。这样，表达式 cout << trip 将导致 os 成为 cout 的一个别名；而表达式 cerr << trip 将导致 os 成为 cerr 的一个别名。Time 对象可以按值或按引用来传递，因为这两种形式都使函数能够使用对象的值。按引用传递使用的内存和时间都比按值传递少。

(3) 重载<<的版本二：

前面介绍的实现存在一个问题。像下面这样的语句可以正常工作：

```
cout << trip;
```

但这种实现不允许像通常那样将重新定义的<<运算符与 cout 一起使用：

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

要理解这样做不可行的原因以及必须如何做才能使其可行，首先需要了解关于 cout 操作的一点知识。请看下面的语句：

```
int x = 5;
int y = 8;
cout << x << y;
```

C++从左至右读取输出语句，意味着它等同于：

```
(cout << x) << y;
```

正如 iosream 中定义的那样，<<运算符要求左边是一个 ostream 对象。显然，因为 cout 是 ostream 对象，所以表达式 cout << x 满足这种要求。然而，因为表达式 cout << x 位于<< y 的左侧，所以输出语句也要求该表达式是一个 ostream 类型的对象。因此，ostream 类将 operator<<( )函数实现为返回一个指向 ostream 对象的引用。具体地说，它返回一个指向调用对象（这里是 cout）的引用。因此，表达式(cout << x)本身就是 ostream 对象 cout，从而可以位于<<运算符的左侧。

可以对友元函数采用相同的方法。只要修改 operator<<( )函数，让它返回 ostream 对象的引用即可：

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

注意，返回类型是 ostream &。这意味着该函数返回 ostream 对象的引用。因为函数开始执行时，程序传递了一个对象引用给它，这样做的最终结果是，函数的返回值就是传递给它的对象。也就是说，下面的语句：

```
cout << trip;
```

将被转换为下面的调用：

```
operator<<(cout, trip);
```

而该调用返回 cout 对象。因此，下面的语句可以正常工作：

格式：

```
ostream& operator<< (ostream& os , const Time& t)
{
        os << t.hours << "asjdhksadgh" << t.minutes << "sahk";
        return os; //注意：返回值是ostream&，因此要return（cout对象）！
}
```

（4）实战分析：

```
// 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)       // 作为内联函数
        { return t * m; }   // inline definition
```

```
    friend std::ostream & operator<<(std::ostream & os, const Time & t);
};
#endif
```

```
// mytime3.cpp  -- implementing Time methods
#include "mytime3.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      // 友元函数作为内联函数
{
    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)
{
    os << t.hours << " hours, " << t.minutes << " minutes";
    return os;
}
```

```
//usetime3.cpp -- using the fourth draft of the Time class
// compile usetime3.cpp and mytime3.cpp together
#include <iostream>
#include "mytime3.h"
int main()
{
    using std::cout;
    using std::endl;
    Time aida(3, 35);
    Time tosca(2, 48);
    Time temp;

    cout << "Aida and Tosca:\n";
    cout << aida<<"; " << tosca << endl;

    temp = aida + tosca;     // operator+()
    cout << "Aida + Tosca: " << temp << endl;

    temp = aida* 1.17;  // member operator*()
    cout << "Aida * 1.17: " << temp << endl;

    cout << "10.0 * Tosca: " << 10.0 * tosca << endl;
    // std::cin.get();
    return 0;
}
```

3）类的自动转换和强制类型转换

（1）c++自身特性：

> [1] 如果这两种类型兼容，则c++将自动将这个值转换成 接收变量 的类型

```
long count = 8;    // 8 convert to type long
double time = 11;  // 11 convert to type double
int side = 3.33;   // 3.33 convert to type int 3
```

> [2] c++语言不能自动转换不兼容的类型

```
int* p = 10;  // invalid!

int* p = (int*) 10; // valid! 将10强制转换成int指针类型( int* )，将指针设置为10，至于是否有意义这里不关心!
```

（2）类中的转换：

【1】接受**一个参数**的构造函数将类型与 该参数相同的值 转换成类提供了蓝图

```
Stonewt(double lbs); // double-object convert to Stonewt-object
-----------------------------------------------------------------
Stonewt myCat; // create a Stonewt object
myCat = 19.6;  // use Stonewt(double) to convert 19.6 to Stonewt
```

使用构造函数 `Stonewt(double)` 来创建一个临时的Stonewt对象，并将 `19.6` 作为初始化值。随后，采用逐成员赋值的方式将该临时对象的内容复制到 `myCat` 中。这一过程叫做隐式转换（自动进行）

【2】**只有**接受一个参数的构造函数才能作为转换函数

```
Stonewt(int s,double lbs); // not a conversion func!
```

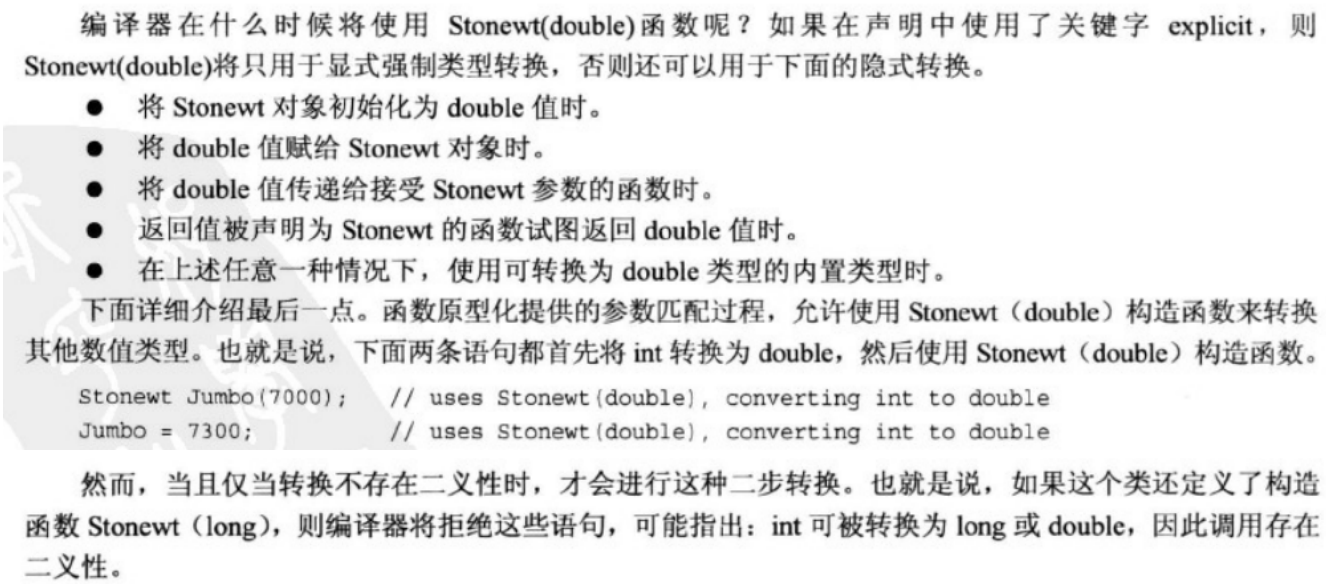
【3】c++增添了 关键字**explicit** 用于关闭 “构造函数用作自动类型转换函数”这一特性

```
explicit Stonewt(double hbx);
```

这将关闭上述的隐式转换，但是仍然允许显式转换，即：显式强制转换

```
Stonewt myCat;
myCat = 19.6;  // invalid!
myCat = Stonewt(19.6);  // OK! an explicit conversion [new form]
myCat = (Stonewt) 19.6; // OK! an explicit conversion [old form]
```

【4】注意”二义性“：



编译器在什么时候将使用 Stonewt(double)函数呢？如果在声明中使用了关键字 explicit，则 Stonewt(double)将只用于显式强制类型转换，否则还可以用于下面的隐式转换。

- 将 Stonewt 对象初始化为 double 值时。
- 将 double 值赋给 Stonewt 对象时。
- 将 double 值传递给接受 Stonewt 参数的函数时。
- 返回值被声明为 Stonewt 的函数试图返回 double 值时。
- 在上述任意一种情况下，使用可转换为 double 类型的内置类型时。

下面详细介绍最后一点。函数原型化提供的参数匹配过程，允许使用 Stonewt（double）构造函数来转换其他数值类型。也就是说，下面两条语句都首先将 int 转换为 double，然后使用 Stonewt（double）构造函数。

```
Stonewt Jumbo(7000);   // uses Stonewt(double), converting int to double
Jumbo = 7300;          // uses Stonewt(double), converting int to double
```

然而，当且仅当转换不存在二义性时，才会进行这种二步转换。也就是说，如果这个类还定义了构造函数 Stonewt（long），则编译器将拒绝这些语句，可能指出：int 可被转换为 long 或 double，因此调用存在二义性。

【5】当构造函数只接受一个参数时，可以使用下面的格式来初始化类对象：

```
Stonewt hbx = 290;
```

这与之前说明的两类构造函数含义相同！

```
法1)   Stonewt hbx(290);
法2)   Stonewt hbx = Stonewt(290);
```

（3）转换函数：

【1】问题引入：

程序清单 11.18 将数字转换为 Stonewt 对象。可以做相反的转换吗？也就是说，是否可以将 Stonewt 对象转换为 double 值，就像如下所示的那样？

```
Stonewt wolfe(285.7);
double host = wolfe;  // ?? possible ??
```

可以这样做，但不是使用构造函数。构造函数只用于从某种类型到类类型的转换。要进行相反的转换，必须使用特殊的 C++运算符函数——转换函数。

转换函数是用户定义的强制类型转换，可以像使用强制类型转换那样使用它们。例如，如果定义了从 Stonewt 到 double 的转换函数，就可以使用下面的转换：

```
Stonewt wolfe(285.7);
double host = double (wolfe);      // syntax #1
double thinker = (double) wolfe;   // syntax #2
```

也可以让编译器来决定如何做：

```
Stonewt wells(20, 3);
double star = wells;   // implicit use of conversion function
```

编译器发现，右侧是 Stonewt 类型，而左侧是 double 类型，因此它将查看程序员是否定义了与此匹配的转换函数。（如果没有找到这样的定义，编译器将生成错误消息，指出无法将 Stonewt 赋给 double。）

【2】创建形式：

```
operator typeName();
```

注意：
1）转换函数必须是类方法，需要通过类对象调用
2）转换函数不需要指定返回类型，因为typeName()写的就是待转换类型
3）转换函数不能有参数

eg:

```
operator double();
operator int();
......
```

程序分析：

```
// 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           (反转函数！！！)
    operator int() const;         此函数目的：实现四舍五入功能
    operator double() const;      此函数目的：直接返回该浮点数值
```

```
};
#endif
```

```
// stonewt1.cpp -- Stonewt class methods + conversion functions
#include <iostream>
using std::cout;
#include "stonewt1.h"
// construct Stonewt object from double value
Stonewt::Stonewt(double lbs)
{
    stone = int (lbs) / Lbs_per_stn;    // integer division
    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";
}

// conversion functions
Stonewt::operator int() const      ------实现四舍五入功能
{
    return int (pounds + 0.5);
}

Stonewt::operator double()const    ------直接返回该浮点数值
{
    return pounds;
}
```

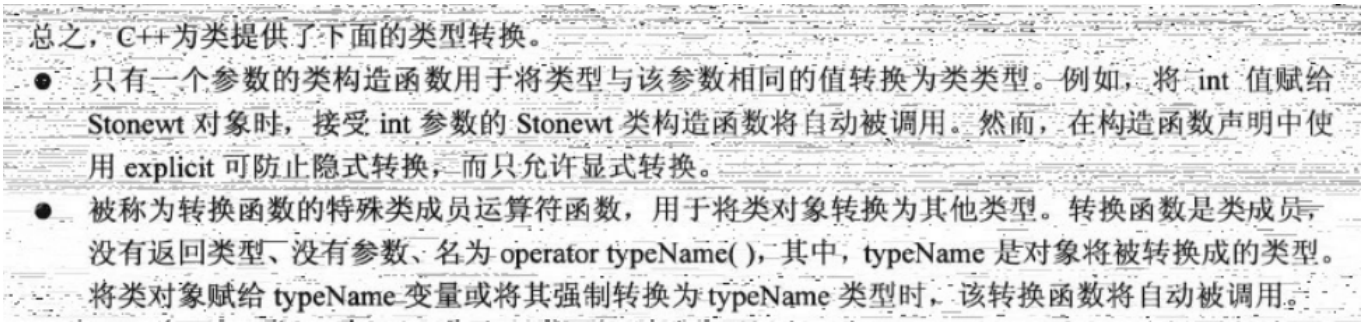
回头看看函数：

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

```
(1) Stonewt:: 说明是类方法，因此使用作用域解析符
(2) const 说明不改变被调用对象！
```

【3】当类定义了两种及以上的转换时，仍可以用显式强制类型转换来指出要使用哪个转换函数！

```
long gone = (double) poppins;// 将poppins转换为一个double值；然后赋值操作：将该double值转换成long类型
long gone = int(poppins);    //                    int                        int        long
```



总之，C++为类提供了下面的类型转换。

- 只有一个参数的类构造函数用于将类型与该参数相同的值转换为类类型。例如，将 int 值赋给 Stonewt 对象时，接受 int 参数的 Stonewt 类构造函数将自动被调用。然而，在构造函数声明中使用 explicit 可防止隐式转换，而只允许显式转换。
- 被称为转换函数的特殊类成员运算符函数，用于将类对象转换为其他类型。转换函数是类成员，没有返回类型、没有参数、名为 operator typeName( )，其中，typeName 是对象将被转换成的类型。将类对象赋给 typeName 变量或将其强制转换为 typeName 类型时，该转换函数将自动被调用。