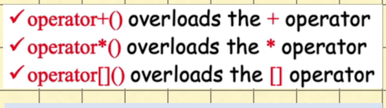
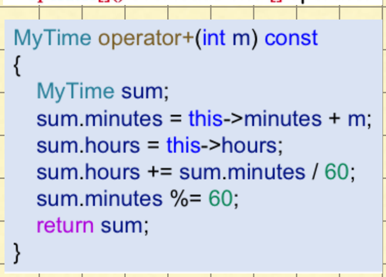
Operator Overloading: 重载操作符 (C++有, C中无)

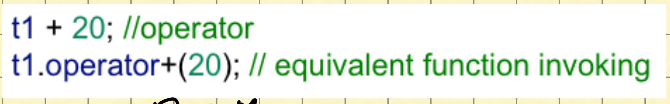


✓operator+() overloads the + operator
✓operator*() overloads the * operator
✓operator[]() overloads the [] operator



```cpp
MyTime operator+(int m) const
{
    MyTime sum;
    sum.minutes = this->minutes + m;
    sum.hours = this->hours;
    sum.hours += sum.minutes / 60;
    sum.minutes %= 60;
    return sum;
}
```

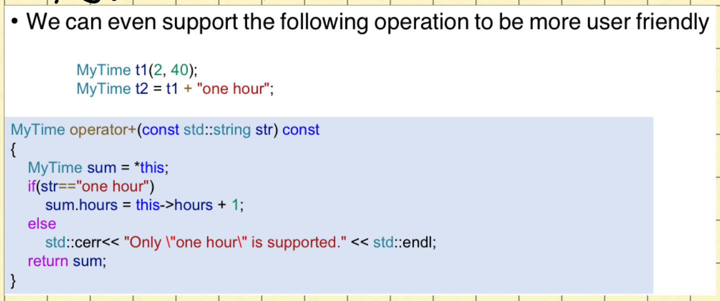
不仅传入了参数列表中的参数，同时暗中也传入了this.



```cpp
t1 + 20; //operator
t1.operator+(20); // equivalent function invoking
```

这两种调用等价.

有更多的重载

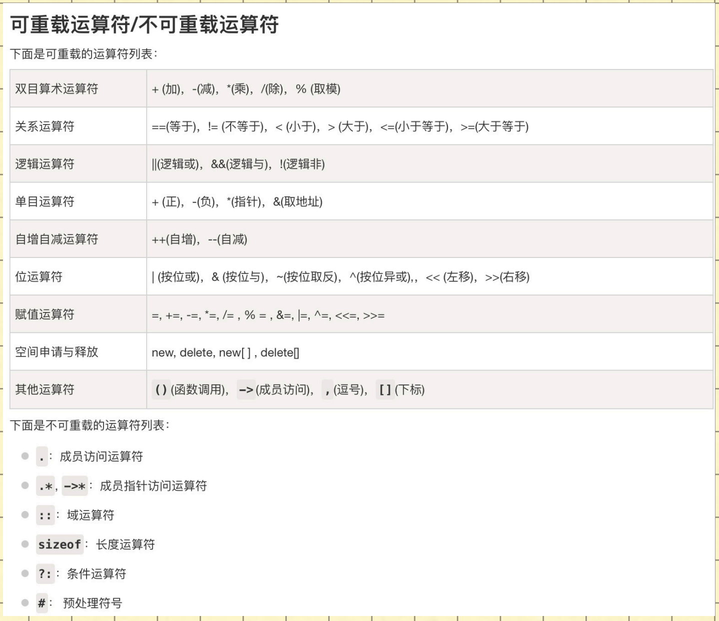


• We can even support the following operation to be more user friendly

```cpp
MyTime t1(2, 40);
MyTime t2 = t1 + "one hour";
```

```cpp
MyTime operator+(const std::string str) const
{
    MyTime sum = *this;
    if(str=="one hour")
        sum.hours = this->hours + 1;
    else
        std::cerr<< "Only \"one hour\" is supported." << std::endl;
    return sum;
}
```

重载中,要注意: ① 至少有一个操作数的类型是自己定义的.
② 不能违反原始操作符的规则
③ 不能创建新的操作符
④ = 、( ) 、[ ] 、-> 只能由成员函数进行重载
⑤ 不能重载 sizeof . .* :: ?: typeid const_cast dynamic_cast reinterpret_cast , static_cast



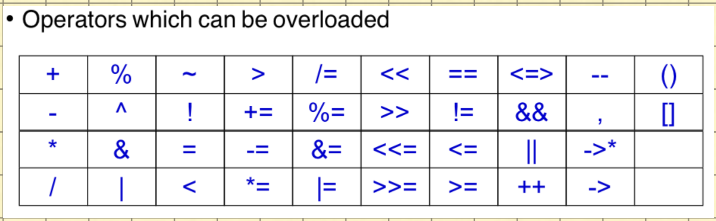
可重载运算符/不可重载运算符

下面是可重载的运算符列表：

双目算术运算符	+ (加)，-(减)，*(乘)，/(除)，% (取模)
关系运算符	==(等于)，!= (不等于)，< (小于)，> (大于)，<=(小于等于)，>=(大于等于)
逻辑运算符	||(逻辑或)，&&(逻辑与)，!(逻辑非)
单目运算符	+ (正)，-(负)，*(指针)，&(取地址)
自增自减运算符	++(自增)，--(自减)
位运算符	| (按位或)，& (按位与)，~(按位取反)，^(按位异或),，<< (左移)，>>(右移)
赋值运算符	=, +=, -=, *=, /= , % = , &=, |=, ^=, <<=, >>=
空间申请与释放	new, delete, new[ ] , delete[]
其他运算符	() (函数调用)，-> (成员访问)，, (逗号)，[] (下标)

下面是不可重载的运算符列表：

- `.`：成员访问运算符
- `.*`, `->*`：成员指针访问运算符
- `::`：域运算符
- `sizeof`：长度运算符
- `?:`：条件运算符
- `#`：预处理符号



• Operators which can be overloaded

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

只有成员/友元函数才能进行操作符重载.

[]只能在成员函数重载

Friend Function

A=B*2.75；A=B.operator*(2.75) 都可以，但 A=2.75*B 对应的 2.75无成员函数.

在定义一个类的时候，可以把一些函数（包括全局函数和其他类的成员函数）声明为“友元”，这样那些函数就成为该类的友元函数，在友元函数内部就可以访问该类对象的私有成员了。

将全局函数声明为友元的写法如下：

```
friend  返回值类型  函数名(参数表);
```

将其他类的成员函数声明为友元的写法如下：

```
friend  返回值类型  其他类的类名::成员函数名(参数表);
```

但是，不能把其他类的私有成员函数声明为友元。

因此在对许多运算符的重载中常用友元函数.

友元函数创建：① 声明：

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

不能用对象.operator*调用，与成员函数有一样的访问权限

② 实现：

```
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;
}
```

不需要再加friend关键字，也不用加类名::

- Friend functions
  - Declare in a class body
  - Granted class access to members (including private members)
  - But not members

注意：① 友元函数一定要在类中声明
② 友元函数能访问成员(包括私有变量)
③ 友元函数不是成员，不能通过this调用，函数名前无类名::

- Operator << can also be overloaded.
- But in (cout << t1;) , the first operand is std::ostream, not MyTime.
- To modify the definition of std::ostream? No!
- Use a friend function

其中一个操作数是要重载运算符的对象.

```
friend std::ostream & operator<<(std::ostream & os, const MyTime & t)
{
    std::string str = std::to_string(t.hours) + " hours and "
                + std::to_string(t.minutes) + " minutes.";
    os << str;
    return os;
}
friend std::istream & operator>>(std::istream & is, MyTime & t);
```

友元类

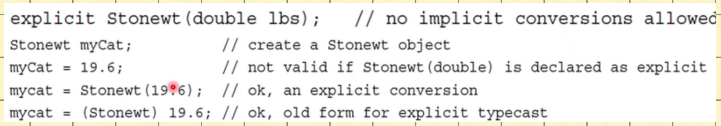
一个类 A 可以将另一个类 B 声明为自己的友元，类 B 的所有成员函数就都可以访问类 A 对象的私有成员。在类定义中声明友元类的写法如下：

```
friend  class  类名;
```

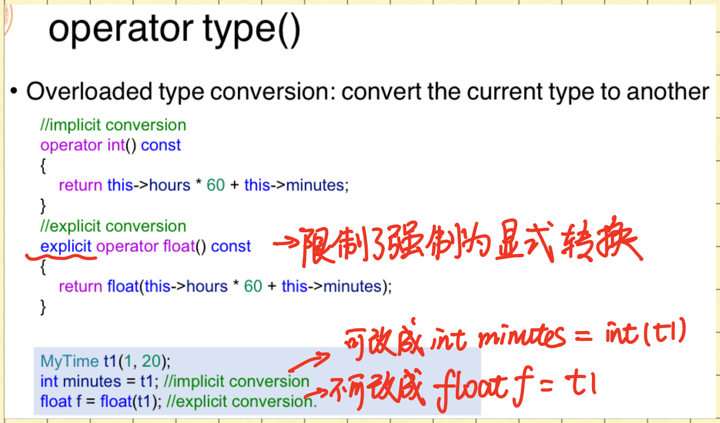
关于友元类的注意事项：

(1) 友元关系不能被继承。
(2) 友元关系是单向的，不具有交换性。若类B是类A的友元，类A不一定是类B的友元，要看在类中是否有相应的声明。
(3) 友元关系不具有传递性。若类B是类A的友元，类C是B的友元，类C不一定是类A的友元，同样要看类中是否有相应的申明。

Type Conversion



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



operator type()

- Overloaded type conversion: convert the current type to another

```
//implicit conversion
operator int() const
{
    return this->hours * 60 + this->minutes;
}
//explicit conversion
explicit operator float() const
{
    return float(this->hours * 60 + this->minutes);
}

MyTime t1(1, 20);
int minutes = t1; //implicit conversion
float f = float(t1); //explicit conversion.
```

→限制了强制为显式转换

可改成 int minutes = int(t1)

不可改成 float f = t1

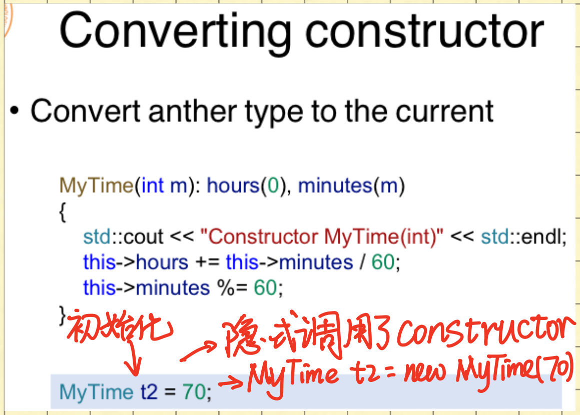
explicit自C++11始.

用explicit能防止隐式自动转换.

一般加在构造函数前.

对参数≥2的构造函数, explicit失效.

但若参数中存在默认参数且只有一个参数无默认值, explicit仍有效



Converting constructor

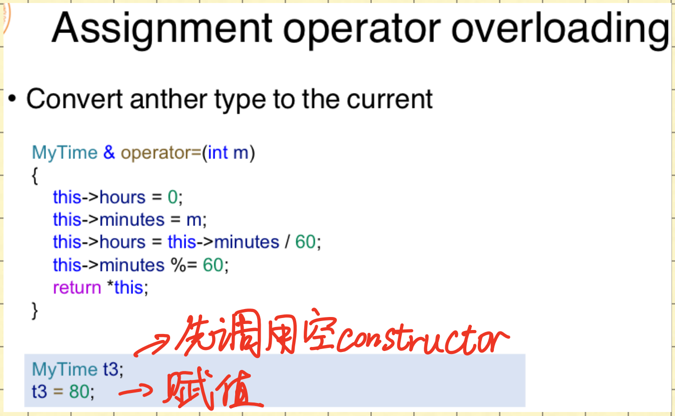
- Convert anther type to the current

```
MyTime(int m): hours(0), minutes(m)
{
    std::cout << "Constructor MyTime(int)" << std::endl;
    this->hours += this->minutes / 60;
    this->minutes %= 60;
}

MyTime t2 = 70;
```

初始化

→隐式调用了constructor

→MyTime t2 = new MyTime(70)



Assignment operator overloading

- Convert anther type to the current

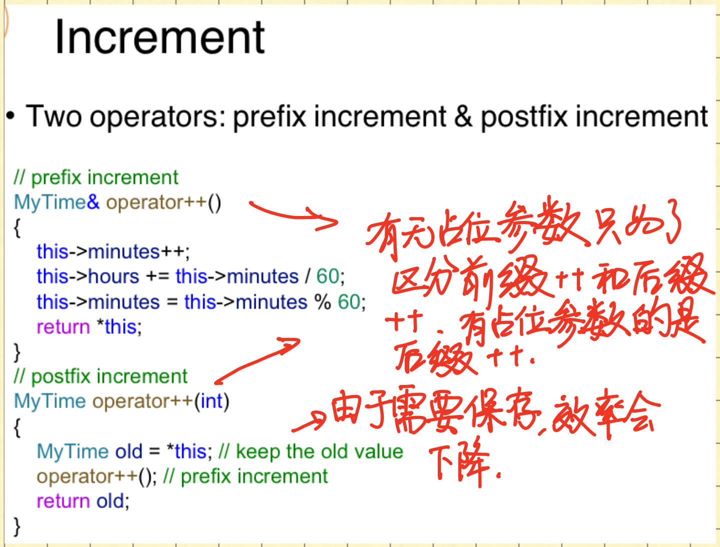
```
MyTime & operator=(int m)
{
    this->hours = 0;
    this->minutes = m;
    this->hours = this->minutes / 60;
    this->minutes %= 60;
    return *this;
}

MyTime t3;
t3 = 80;
```

→先调用空constructor

→赋值

若该类中未重载operator=(), 则这里第二行实际上又调用了一次单参数的constructor, 而且先前的空构造生成的对象会一直存在直至程序终止, 事实上造成了内存泄露.

Increment & Decrement operators



Increment

- Two operators: prefix increment & postfix increment

```
// prefix increment
MyTime& operator++()
{
    this->minutes++;
    this->hours += this->minutes / 60;
    this->minutes = this->minutes % 60;
    return *this;
}
// postfix increment
MyTime operator++(int)
{
    MyTime old = *this; // keep the old value
    operator++(); // prefix increment
    return old;
}
```

→有无占位参数只为了区分前缀++和后缀++. 有占位参数的是后缀++.

→由于需要保存, 效率会下降.