

# C/C++ Programming Language

CS205 Spring

Feng Zheng

Week 11



南方科技大学  
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY



# Content

- Operator Overloading
  - Operator function
  - Friends
  - Overloading the << operator
- Automatic Conversions and Type Casts for Classes
  - Type cast from single augment to an object
    - ✓ Implicit constructor
    - ✓ Explicit constructor
  - Conversion function

# Brief Review



# Review

- Objects and classes

- Two programming styles
- Classes in C++
- Access control
- Function implementations
- Constructors and destructors
- This pointer *static member function*
- Class scope *没有 this pointer*



# Operator Overloading



# Overloading

- Function overloading
  - Let you use **multiple functions** sharing **the same name**
  - Relationship to others:
    - ✓ **Default** arguments
    - ✓ Function **templates**
- \* operator (An operator overloading **example**)
  - Applied to **an address**, yield the value stored at that address
  - Applied **two numbers**, yield the product of the values



# Operator Function

- Operator function

- Keyword: **operator** for C++
- To overload an operator, you use a **special function**
- Function header has the form:

`operatorop(argument-list)`

- ✓ `operator+()` overloads the `+` operator
- ✓ `operator*()` overloads the `*` operator
- ✓ `operator[]()` overloads the `[]` operator
- The compiler, recognizing the **operands** as belonging to the class, replaces the **operator** with the corresponding **operator function**



# Example: Time on Our Hands

- Developing an operator overloading

- 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
```

- Run mytime1.h mytime1.cpp usetime1.cpp

- The name of the **operator+()** function allows it to be invoked by using either **function notation** or **operator** notation
- Add **more than two** objects
- **Left-to-right** operator

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

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

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





# Overloading Restrictions

- Restrictions
  - Must have at least one operand that is a user-defined type
  - Can't violate the syntax rules for the original operator
  - Can't alter operator precedence
  - Can't create new operator symbols
- Member or nonmember operator overloading functions
  - Use only **member functions** to overload these operators

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



# Overloading Restrictions

- **Cannot** overload the following operators

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



# Operators That Can Be Overloaded

- Operators that **can** be overloaded

---

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

---

- Run `mytime2.h mytime2.cpp usetime2.cpp`

- More overloaded operators
  - ✓ **Subtract** one time from another
  - ✓ **Multiply** a time by a factor

Friends



# Introducing Friends

- Access control
  - Can access the **public portions directly**
  - Can access the **private members** of an object only by using the **public member functions**
  - Problems:
    - ✓ Public class methods serve as **the only access**
    - ✓ This restriction is **too rigid to fit particular problems**
- C++ provides another form of access: the **friend**
  - **Friend functions**
  - Friend **classes** - will be introduced later
  - Friend **member functions** - will be introduced later



# Motivation for Friend Functions

- Problem: overloading a binary operator
  - **Left operand** is the invoking object
    - ✓ `Time Time::operator*(double mult) const`
    - ✓ `A = B * 2.75; A = B.operator*(2.75);`
    - ✓ `A = 2.75 * B;` // cannot correspond to a member function
- Solution: using a **nonmember** function
  - A nonmember function is **NOT** invoked by an **object**
  - Any values it uses, including objects, are **explicit arguments**
- Problem: ordinary nonmember functions **can't directly access** private data in a class



# Creating Friends

- Making a function **a friend to a class**
  - Allow the function the **same access privileges** that a member function of the class has
- First step: 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 three implications
  - ✓ **Not a member function**
  - ✓ **Isn't invoked** by using the membership operator
  - ✓ Has the **same access rights** as a member function



# Creating Friends

- The second step is to write the function definition
  - Not a member function you **don't** use the **:: qualifier**
  - **Don't** use the **friend** keyword in the definition

```
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;
}
```

- A friend function to a class is a **nonmember** function that has the **same** access rights as a **member** function





# More About Friends

- Summary

- Only a **class declaration** can decide which functions are friends, so the class declaration still controls which functions access private data
- Class methods and friends are simply two different mechanisms for **expressing a class interface**
- Overload an operator for a class and use the operator with a **nonclass term as the first operand**
- Uncomment: mytime2.h mytime2.cpp usetime2.cpp

```
Time operator*(double m, const Time & t)
{
    return t * m;    // use t.operator*(m)
}
```



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

- A question?

- Suppose trip is a **Time object**

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

- Suppose fun is a **function name**

- ✓ `cout << "(fun)= " << (void*)fun << endl;` // how about no (void\*)

- Print the address of a **string**

- ✓ `const char *pszStr = "this is a string"; cout << pszStr << endl;`

- Heavily overloaded

- << : **bit manipulation operators** - shifts bits left in a value

- << : **output tool** - recognize all the basic C++ types.



# The First Version of Overloading <<

- Use a Time **member function** to overload <<, the Time object would come first

- Be confusing

- The problem is the same to the overloaded **\* operator**



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

- Use a **friend function** to overload the operator:

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

- **Enable** to have the original syntax of cout

`cout << trip;`



# The Second Version of Overloading

<<

- Consider the example:
  - Read the output statement from **left to right**
  - Return **a reference** to the invoking object—cout

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

(cout << x) << y;

↑  
cout

- **Problem:**

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

- Return **a reference** to an **ostream** object
- Run mytime3.h mytime3.cpp  
usetime3.cpp

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



# Overloaded Operators: Member Versus Nonmember Functions

- Have a choice between using **member** functions or **nonmember** functions to implement operator overloading

- One: has the **prototype** in the **Time class declaration**

✓ **One operand** is passed implicitly via the `this` pointer

```
Time operator+(const Time & t) const; // member version
```

- Two: use the prototype with keyword: **friend**

```
// nonmember version
```

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

- Compiler can convert the statement

```
T1 = T2 + T3;      T1 = T2.operator+(T3);      // member function  
                  T1 = operator+(T2, T3);    // nonmember function
```

- Must choose **one form**

# Automatic Conversions and Type Casts for Classes



# Conversions for Built-in Types

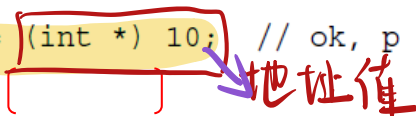
- Generate **numeric** type conversions

```
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
```

- **Recognize** that the diverse numeric types all represent the same basic thing—a **number**
- Incorporate **built-in rules** for making the conversions
- **Does not** automatically convert types that are **not compatible**

```
int * p = 10;  // type clash
```

```
int * p = (int *) 10;  // ok, p and (int *) 10 both pointers
```





# Define a class related to **a** basic type or to **another** class

- See header file in `stonewt.h`

- Have **three** constructors

```
Stonewt blossom(132.5);    // weight = 132.5 pounds
Stonewt buttercup(10, 2);  // weight = 10 stone, 2 pounds
Stonewt bubbles;           // weight = default value
```

- Write code like the following

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

- ✓ Implicit conversion: happen automatically, **no need explicit** type cast

- ✓ Provided a **default value** for the second parameter

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

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

- ✓ Explicit conversions: **turn off** the automatic aspect (keyword: **explicit**)

➡ **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
```





# When does the compiler use the Stonewt(double) function?

- Argument-matching process provided by function prototyping
  - If the **explicit** is used in the declaration, Stonewt(double) is used **only** for an **explicit type cast**
  - Otherwise, it is used for the following implicit conversions, when
    - ✓ **initialize** a Stonewt object using a **type double value**
    - ✓ **assign** a **type double value** to a Stonewt object
    - ✓ **pass** a **type double value** to a function that expects a Stonewt argument
    - ✓ a function that's **declared to return** a Stonewt value **tries to return** a double value
    - ✓ any of the preceding situations use a **built-in type** that can unambiguously be **converted** to **type double**
- Only a constructor that can be used with just one argument works as a conversion function



# Conversion Functions

- Question: Can we do the **reverse**?

```
Stonewt wolfe(285.7);
```

→ 

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

- **Yes, conversion function**

- User-defined type casts

```
Stonewt wolfe(285.7);
```

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

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

- Use a conversion function in this form

`operator typeName();`

- ✓ Must **be** a **class method**
- ✓ Must **not** specify a **return type**
- ✓ Must have **no arguments**

```
Stonewt::operator double()const  
{  
    return pounds;  
}
```



No return type but has return value



# Applying Type Conversions Automatically

- Problem: when **omit** the explicit type cast

```
cout << "Poppins: " << int(poppins) << " pounds.\n";  
Stonewt poppins(9,2.8); cout << "Poppins: " << poppins << " pounds.\n";  
  
long gone = poppins;    // ambiguous ✕
```

- The compiler complains about using an **ambiguous conversion**
- The class has defined **two** (double and int) conversion functions
- Use an **explicit type cast**, when the class defines **two or more** conversions

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



# Solutions and Summary

- Solutions

- Declare a conversion operator as **explicit**

```
class Stonewt
{
    ...
    // conversion functions
    explicit operator int() const;
    explicit operator double() const;
};
```

- Replace a conversion function with a **nonconversion** function

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



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

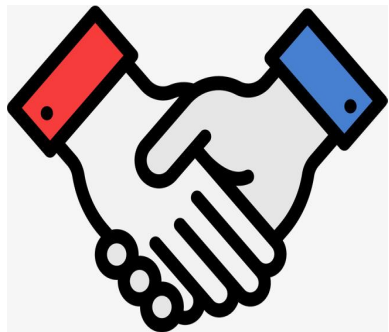
- Summary

- A **class constructor** that has but a **single argument** serves as an instruction for converting a value of the argument type to the class type
- **Conversion function** serves as an instruction for converting a class object to some other types



# Summary

- Operator Overloading
  - Operator function
  - Friends
  - Example: overloading the << operator
- Automatic Conversions and Type Casts for Classes (=)
  - Type cast from **single** augment to an object
    - ✓ Implicit constructor
    - ✓ Explicit constructor
  - Conversion function



Thanks



[zhengf@sustech.edu.cn](mailto:zhengf@sustech.edu.cn)