



南方科技大学  
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# C/C++ Program Design

## CS205

**Prof. Shiqi Yu (于仕琪)**

yusq@sustech.edu.cn

<http://faculty.sustech.edu.cn/yusq/>



# Operators for `cv::Mat`



# Function overloading

```
Mat add(Mat& A, Mat& B);  
Mat add(Mat& A, float b);  
Mat add(float a, Mat& B);
```

```
Mat mul(Mat& A, Mat& B);  
Mat mul(Mat& A, float b);  
Mat mul(float a, Mat& B);
```

...

More convenient to code as follows

```
Mat A, B;  
float a, b;  
//...  
Mat C = A + B;  
Mat D = A * B;  
Mat E = a * A;
```



# operators for cv::Mat

```
#include <iostream>
#include <opencv2/opencv.hpp>
```

```
using namespace std;
```

```
int main()
{
```

```
    float a[6]={1.0f, 1.0f, 1.0f, 2.0f, 2.0f, 2.0f};
    float b[6]={1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f};
    cv::Mat A(2, 3, CV_32FC1, a);
    cv::Mat B(3, 2, CV_32FC1, b);
```

```
    cv::Mat C = A * B;
```

```
    cout << "Matrix C = " << endl
         << C << endl;
```

```
    return 0;
```

```
yushiqi@Mac exampleMat % ./matexample
Matrix C =
[9, 12;
 18, 24]
```



# Operator overloading

- Customizes the C++ operators for **operands of user-defined types**.
- Overloaded operators are functions with special function names:

```
std::string s("Hello ");  
s += "C";  
s.operator+=(" and CPP!");
```

`std::basic_string<CharT,Traits,Allocator>::operator+=`

<code>basic_string&amp; operator+=( const basic_string&amp; str );</code>	(1)	(until C++20)
<code>constexpr basic_string&amp; operator+=( const basic_string&amp; str );</code>		(since C++20)
<code>basic_string&amp; operator+=( CharT ch );</code>	(2)	(until C++20)
<code>constexpr basic_string&amp; operator+=( CharT ch );</code>		(since C++20)
<code>basic_string&amp; operator+=( const CharT* s );</code>	(3)	(until C++20)
<code>constexpr basic_string&amp; operator+=( const CharT* s );</code>		(since C++20)
<code>basic_string&amp; operator+=( std::initializer_list&lt;CharT&gt; ilist );</code>	(4)	(since C++11)
<code>constexpr basic_string&amp; operator+=( std::initializer_list&lt;CharT&gt; ilist );</code>		(until C++20)
<code>template &lt; class T &gt;</code>		(since C++17)
<code>basic_string&amp; operator+=( const T&amp; t );</code>	(5)	(until C++20)
<code>template &lt; class T &gt;</code>		
<code>constexpr basic_string&amp; operator+=( const T&amp; t );</code>		(since C++20)

[stringdemo.cpp](#)



南方科技大学  
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# Operator Overloading



# Operator overloading

- Implementation of `operator+()` and `operator+=()`

```
class MyTime
{
    int hours;
    int minutes;
public:
    MyTime(): hours(0), minutes(0){}
    MyTime(int h, int m): hours(h), minutes(m){}

    MyTime operator+(const MyTime & t) const
    {
        MyTime sum;
        sum.minutes = this->minutes + t.minutes;
        sum.hours = this->hours + t.hours;
        sum.hours += sum.minutes / 60;
        sum.minutes %= 60;
        return sum;
    }
    std::string getTime() const;
};
```

```
MyTime t1(2, 40);
MyTime t2(0, 50);
cout << (t1 + t2).getTime() << endl;
```

example1.cpp



# Operator overloading

- If one operand is not MyTime, and is an `int`

```
MyTime t1(2, 40);  
MyTime t2 = t1 + 20;
```

- The function can be

```
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;  
}
```





# Operator overloading

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

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

```
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;  
}
```



# Operator overloading

- Overloaded operators is more user-friendly than functions.
- But , wait ..

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

- How about the expression

```
20 + t1;
```



南方科技大学  
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# friend Functions



# friend Functions

- If we want that operator + can support (`int` + `MyTime`)

```
MyTime t1(2, 40);  
20 + t1;
```

- Let a friend function to help
- Friend functions
  - Declare in a class body
  - Granted class access to members (including private members)
  - But **not** members



# friend Functions

- Again, friend functions are not members! They just declared in the class body.

```
class MyTime
{
    // ...
    public:
        friend MyTime operator+(int m, const MyTime & t)
        {
            return t + m;
        }
};
```



# friend Functions

- A friend function is defined out of the class.
- No `MyTime::` before its function name

```
class MyTime
{
    // ...
    public:
        friend MyTime operator+(int m, const MyTime & t);
};

MyTime operator+(int m, const MyTime & t)
{
    return t + m;
}
```



# friend Functions

- 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

```
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);
```



南方科技大学  
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# User-defined Type Conversion





# 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.
```



# Converting constructor

- Convert another type to the current

```
MyTime(int m): hours(0), minutes(m)
{
    this->hours += this->minutes / 60;
    this->minutes %= 60;
}
```

```
MyTime t2 = 70;
```



# Assignment operator overloading

- Convert another 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;
```



# Be careful

- What is the difference in creating object `t2` and `t3`?

```
MyTime t2 = 80;
```

```
MyTime t3;  
t3 = 80;
```



南方科技大学  
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

# Increment and 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;
}
```



# Operators

- Operators which can be overloaded

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