New
them

Static class members

- ‘static’ means - associated with a class, rather than with any specific object of that class;
  - Static data members:

```
class Orbiter {
        protected:
                double m_mass;
                XY m_current, m_prior, m_thrust;
        public:
                static int nCount;  //only 1 copy of nCount exists
                …
        };
```

- you must declare global definition code to initialize static data:
  `int Orbiter::nCount = 0;`
- if declared public the data member can be used like this:
  `Orbiter::nCount++;`
- if declared as const static, the data member can be still initialized single time only;

static member-functions

- **if static data member is private, you would need a public, static member function to work with it**
- **another use of static member functions is in constructions. Suppose you need to construct a new orbiting object, but you don't know until run time witch derived Orbiter class you want:**

```
static Orbiter* Orbiter::MakeNew(int select , XY& current, XY& prior, double mass,
                                                    XY& thrust, xY& orientation, Planet* pPlanet)
{
    switch (select){
    case 0:
        return new Planet(current, prior, mass);
    case 1:
        return new SpaceShip(current, prior, thrust, mass, orientation);
    case 2:
        return new Moon(current, prior, mass, pPlanet);
    default:
        return NULL;
    }
}
```

New
them

Operator Overloading

- **What operator overloading is?**
- **Which classes must support operator overloading**
- **What you can and can’t overload**
- **How to implement operator overloads**

What operator overloading is?

You can redefine the function of most built-in operators globally or on a class-by-class basis. Overloaded operators are implemented as functions.The name of an overloaded operator is 'operatorx', where 'x' is the operator.

For example, to overload the addition operator, you define a function called operator+.
Similarly, to overload the addition/assignment operator '+=' - define a function called operator+=.

Operators, provided by C++ work only with built-in types. Not with your own data types.
The compiler doesn't know how to apply operator (like +, == …) to your types.

If you want not to write:

Object3 = object1.Add(object2);

But instead:

Object3 = object1 + object2;

You have to overload operator '+'

So is with complex data type like: Date, String,….
And with

OBJECTS

like operation:
MyBankAccount == youBankAccount;

All Traditional operations can be overloaded.
Not possible to overload some esoteric operations like sizeof or '.'
In managed C++ is not possible to overload →, (), []

The compiler distinguishes between the different meanings of an operator by examining the types of its operands.
The syntax is:

type operator::operator-symbol (parameter-list)

Rules of overloading

- **You can have no new operators!**
- **You can't change the number of operands, taken by an operator;**
- **You can't change the precedence or associativity of operators'!**
  For example * will always take precedence over +, regardless you will.

Overloading operators in managed types

CLS defines a list of operators that .NET languages can support.
That means, when you override operator in a managed C++ type, you don't override t
C++ operator, but instead override the underlying CLS functionality.

What operator overloading is in fact?

having:

```
class ostream{
        .....
        ostream operator<<(char*);
        }
ostream ostream::operator<<(char*)
        {while (*p) buf.sputc(*p++);
         return *this; }
```

we in fact have a definition of operator << as a member of ostream class. This done, the following is possible:

```
s<<p
```

In fact it's interpreted as:

```
s.operator<<(p);
```

where 's' is of type ostream and 'p' is char*.

Operator << is a binary operation!! The syntax

```
operator<<(char*)
```

is looking as 1 parameter operation. Please. Have in mind - this is a second parameter.

Operation '<<' is returning a ostream type. So, the syntax of type:

```
s << p << q
```

Is a possible syntax, equivalent of:

```
(s.operator<<(p)).operator<<(q)
```

Overloading arithmetic operators

```
struct Complex {
        Complex( double r, double i ) : re(r), im(i) {}
        Complex operator+( Complex &other );

        void Display( )
                { cout << re << ", " << im << endl; }

private:
        double re, im;
            };

 // Operator overloaded using a member function
Complex Complex::operator+( Complex &other )
        { return Complex( re + other.re, im + other.im ); }

int main()
{
        Complex a = Complex( 1.2, 3.4 );
        Complex b = Complex( 5.6, 7.8 );
        Complex c = Complex( 0.0, 0.0 );
        c = a + b;
        c.Display();
}
```

Overloading arithmetic operators- full example 1/3

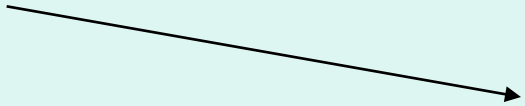
.h

```
// The FeetInches class holds distances or measurements  expressed in feet and inches.

class FeetInches
{private:
   int feet;                                    // To hold a number of feet
   int inches;                                  // To hold a number of inches
   void simplify();                             // Defined in FeetInches.cpp
public:
   // Constructor
   FeetInches(int f = 0, int i = 0)
     { feet = f;
       inches = i;
       simplify(); }

   // Mutator functions
   void setFeet(int f)          { feet = f; }

   void setInches(int i)
     { inches = i;
       simplify(); }

   // Accessor functions
   int getFeet() const          { return feet; }

   int getInches() const        { return inches; }

   // Overloaded operator functions
   FeetInches operator + (const FeetInches &);      // Overloaded +
   FeetInches operator - (const FeetInches &);      // Overloaded –
   FeetInches operator ++ ();                       // Prefix ++
   FeetInches operator ++ (int);                    // Postfix ++
};
```

Overloading arithmetic operators- full example 2/3

.cpp

```
// Implementation file for the FeetInches class
#include "FeetInches.h"
// Definition of member function simplify(). This function     *
// checks for values in the inches member greater than       *
// twelve or less than zero. If such a value is found,       *
// the numbers in feet and inches are adjusted to conform    *
// to a standard feet&inches expression. For example,      *
// 3 feet 14 inches would be adjusted to 4 feet 2 inches and *
// 5 feet -2 inches would be adjusted to 4 feet 10 inches.   *
void FeetInches::simplify()
{   if (inches >= 12)             {
      feet += (inches / 12);
      inches = inches % 12;   }
   else if (inches < 0)
   {      feet -= ((abs(inches) / 12) + 1);
      inches = 12 - (abs(inches) % 12);   }
}
// Overloaded binary + operator.              *
FeetInches FeetInches::operator + (const FeetInches &right)
{   FeetInches temp;
   temp.inches = inches + right.inches;
   temp.feet = feet + right.feet;
   temp.simplify();
   return temp;
}
// Overloaded binary - operator.              *
FeetInches FeetInches::operator - (const FeetInches &right)
{   FeetInches temp;

   temp.inches = inches - right.inches;
   temp.feet = feet - right.feet;
   temp.simplify();
   return temp;}
```



Overloading arithmetic operators- full example 3/3

.cpp

```
//*************************************************************
// Overloaded prefix ++ operator. Causes the inches member to *
// be incremented. Returns the incremented object.          *
//affect only the object, so – no need for a parameter !
//*************************************************************

FeetInches FeetInches::operator++()
{
        ++inches;
        simplify();
        return *this;
}

//*************************************************************
// Overloaded postfix ++ operator. Causes the inches member to  *
// be incremented. Returns the value of the object before the   *
// increment.                                                   *
//*************************************************************

FeetInches FeetInches::operator++(int)
{
        FeetInches temp(feet, inches);

        inches++;
        simplify();
        return temp;
}
```

++ distance; //OK
or:

distance2 = ++ distance1; //OK
Is equivalent to:
distance2 = distance1.operator++();

Dummy parameter. (nameless) . So the function will be used in postfix mode

Temporary object – hold the value before increment.
This value will be returned after.
So, following is correct:
distance2 = distanse1++;

In first .NET - overloading arithmetic operators (working on value types)

We have the type:

```
__value struct Dbl
{
        double val;
Public:
        Dbl(double v) { val = v;}
        Double getVal() { return val;}
}
```

If you want to implement:

```
d3 = d1 + d2;          // for Dbl types
```

You have to implement +operator. Add the following code to the class definition:

```
static Dbl op_Addition(Dbl 1st, Dbl second)
{
Dbl result(1st.val +second.val);
return result;
}
```

See the following list For CLS functions

Redefinable Operators

Operator	Name	Type
,	Comma	Binary
!	Logical NOT	Unary
!=	Inequality	Binary
%	Modulus	Binary
%=	Modulus assignment	Binary
&	Bitwise AND	Binary
&	Address-of	Unary
&&	Logical AND	Binary
&=	Bitwise AND assignment	Binary
( )	Function call	—
( )	Cast Operator	Unary
*	Multiplication	Binary
*	Pointer dereference	Unary
*=	Multiplication assignment	Binary
+	Addition	Binary
+	Unary Plus	Unary
++	Increment1	Unary
+=	Addition assignment	Binary

Redefinable Operators

–	Subtraction	Binary
–	Unary negation	Unary
—	Decrement1	Unary
–=	Subtraction assignment	Binary
–>	Member selection	Binary
–>*	Pointer-to-member selection	Binary
/	Division	Binary
/=	Division assignment	Binary
<	Less than	Binary
<<	Left shift	Binary
<<=	Left shift assignment	Binary
<=	Less than or equal to	Binary
=	Assignment	Binary
==	Equality	Binary
>	Greater than	Binary
>=	Greater than or equal to	Binary
>>	Right shift	Binary
>>=	Right shift assignment	Binary

Redefinable Operators

[ ]	Array subscript	—
^	Exclusive OR	Binary
^=	Exclusive OR assignment	Binary
|	Bitwise inclusive OR	Binary
|=	Bitwise inclusive OR assignment	Binary
||	Logical OR	Binary
~	One's complement	Unary
delete	Delete	—
new	New	—
conversion operators	conversion operators	Unary

Nonredefinable Operators

Operator	Name
.	Member selection
.*	Pointer-to-member selection
::	Scope resolution
? :	Conditional

Although overloaded operators are usually called implicitly by the compiler when they are encountered in code, they can be invoked explicitly the same way as any member or nonmember function is called.

.NET

CLS supports the following functions that can be overloaded:

Operation	C++ equivalent	CLS function name
Decrement	--	op_Decrement
Negate	!	Op_Negetion
Unary plus	+	op_UnaryPlus
Multiplication	*	op_Multiply
	/	….
	-	
	%	
	=	
	==	
	<=	
	>	
	>=	
Logical AND	&&	
	||	
	<<	
	>>	
	&	
	|	
Exclusive OR	^	…

Operator Overloading (second example 1/2)

- You must use ‘operator’ keyword (for example: ‘operator+’). This function can be either member functions of the type or global functions that are not member of any type.

- many overloaded operators are implemented as class member functions. As an example: many times XY objects are used as xy-coordinate pairs and it’s obvious to have ‘+’ and ‘–’ operations on them:

```
XY XY::operator +(const XY& xy) const
    {
        return XY(x + xy.x, y + xy.y);
    }

XY XY::operator -(const XY& xy) const
    {
        return XY(x - xy.x, y - xy.y);
    }
```

- To use them:

XY new_xy = xy1 + xy2; // xy1 and xy2 are defined as of XY type.

Operator Overloading

- other operations with coordinates:

```
XY XY::operator -() const {                          //unary minus
      return XY(-x, -y);
}
XY operator *(double mult) {                         //scalar multiply
      return XY( s * mult, y * mult);
}
const XY&operator *=(const double mult) {            // operation *=
      x *= mult;
      y *= mult;
      return this;
}
```

- It's now possible to code:

```
new_xy = xy1 * 2.7;
xy *= 5.0;
```

.NET: Overloading operator functions in managed code

You can also overload the operator functions (mentioned before) also.
Suppose you want to redefine 'operator+' to work not only on 2 Dbl's operands, but on Dbl + int:

D3 = d1 + 5;

You have to override op_Addition function:

```
static Dbl op_Addition(Dbl first, int second)
        {
        Dbl result(first.val + second)
        return result;
        }
```

And also:

```
static Dbl op_Addition(int first, Dbl second)
        {
        Dbl result(first + second.val)
        return result;
        }
```

.NET: Overloading reference types in managed code

Reference types are accessed using pointers, which means that the arguments for operator functions always have to be pointers:

```
static MyRef* op_Addition(MyRef * first, MyRef * second)
        {
        MyRef * result = new MyRef(first.val + second.val);
        return result;
        }
```

In first .NET versions we can't call implicitly overloaded operators on reference types:

```
MyRef* r3 = first + second;                    //doesn't work

MyRef* r3 = MyRef::op_Addition(first, second);    // but this will work

        // in new .NET version this limitation is removed
```

New
them

Conversion operators

- classical *automatic conversion*:

```
int m =3;
double l = atan(m);  // atan expects a double argument, so the compiler converts int
                     //to double before passing it to function
```

- conversion for own classes:

1. you must write the code yourself to predefine the operation:

```
String::operator const char*() const
{
    return (const char*) m_pch;
}
```

2. You can now use a String argument anywhere the compiler expects a const char*

- MFC class CString has the same overloaded const char* conversion operator. You can use this operator