Предефиниране на оператори

- Припокриване на оператори. Същност, ограничения. Припокриване на аритметични операции.
- Преобразувания и операции - преобразувания.

Предефиниране на оператори

- В езика C++ е предвидена възможност операторите да бъдат дефинирани за потребителските типове.
- Има само няколко оператора, които не могат да се предефинират от потребителя:

  :: – оператор за избор на област на видимост;

  . – оператор за избор на член;

  .* – оператор за избор на член чрез указател към член;

  sizeof – оператор за размер на обект;

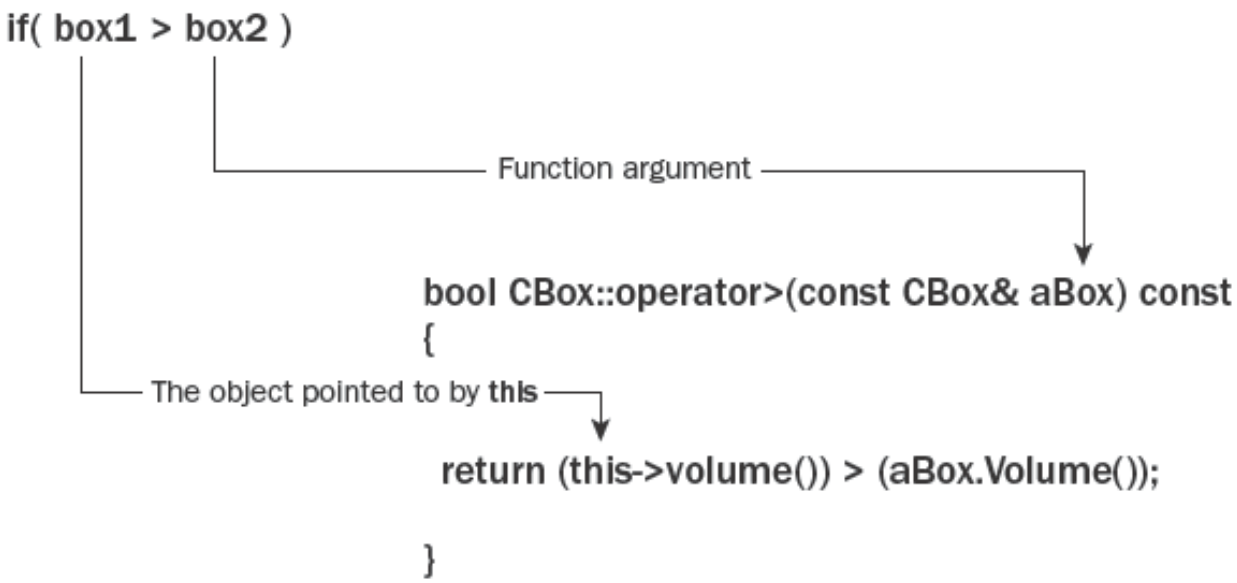
  typeid – оператор за идентификация на типа;

  ?: – оператора за условен избор;
- Всички останали оператори могат да се предефинират.

Предефиниране на оператори

- Дава възможност стандартните оператори в C++, като +, -, * и др. да работят върху обекти от ваши класове.
- За да се имплементира предифиниране на оператор се създава специална функция, чрез ключовата дума **operator** и символа, който искате да предефинирате

Пример


if(box1 > box2)
Function argument
bool CBox::operator>(const CBox& aBox) const
{
The object pointed to by this
return (this->volume()) > (aBox.Volume());
}

Бинарни и унарни оператори

- Бинарен оператор се нарича оператор, който действа върху два аргумента. Унарен е оператор, който действа върху един аргумент.
- Примери за бинарни оператори са операторите + (a+b), * (a*b), - (a-b), / (a/b) и т.н.
- Примери за унарни оператори са операторите - (-a), ! (!a), ~ (~a), ++ (a++) и т.н.
- Видът на оператора определя начините, по които той може да бъде предефиниран.

Бинарни оператори

- Бинарните оператори могат да се дефинират по два начина:
  - Като нестатична член-функция на класа, която приема един аргумент – например:

Point Point :: operator +(cons t Point & p)

  - Като функция, която не е член на класа и приема два аргумента – например:

Point ope r a tor +(cons t Point & p1 , cons t Point & p2)

Пример 1/2

```
#include <iostream>
using namespace std;

class Point {
  double x_, y_;
public:
  Point(double x=0,double y=0)
    : x_(x), y_(y)
  {}
```

```
  double get_x() const {return x_;}
  double get_y() const {return y_;}
  Point operator+(const Point& p) const;
};
Point Point::operator+(const Point& p) const {
  Point result(get_x()+p.get_x(),
               get_y()+p.get_y());
  return result;
}
```

Пример 2/2

```
1 int main(void) {
2   Point p1(1.0,1.0), p2(2.0,2.0), p3;
3
4   p3=p1+p2;
5   cout<<"p3=("
6       <<p3.get_x()<<",␣"
7       <<p3.get_y()<<")"<<endl;
8   return 0;
9 }
```

- Изразът в ред 4 е еквивалентен на следното:

```
p3=p1.operator+(p2);
```

Унарни оператори

- Унарните оператори могат да се дефинират по два начина:
  - Като нестатична член-функция на класа, която не приема аргументи – например:

Point Point :: operator -(void)

  - Като функция, която не е член на класа и приема един аргумент – например:

Point operator -(cons t Point & p)

Пример 1/2

- Нека разгледаме втория вариант за предефиниране на унарен оператор. Като пример отново ще използваме класа `Point` и унарния оператор `-`:

```
1 #include <iostream>
2 using namespace std;
3 class Point {
4   double x_, y_;
5 public:
6   Point(double x=0,double y=0)
7     : x_(x), y_(y)
8   {}
9   double get_x() const {return x_;}
10  double get_y() const {return y_;}
11 };
```

Пример 2/2

```
1  Point operator-(const Point& p) {
2    Point result(-p.get_x(),-p.get_y());
3    return result;
4  }
5  int main(void) {
6    Point p1(1.0,1.0), p2;
7
8    p2=-p1;
9    cout<<"p2=("
10       <<p2.get_x()<<",␣"
11       <<p2.get_y()<<")"<<endl;
12   return 0;
13 }
```

- Изразът в ред 8 е еквивалентен на следното:

```
p2=operator-(p1);
```

- Резултатът от изпълнението на тази програма е:

```
p2=(-1, -1)
```

Предефиниране на оператори

- Всеки оператор може да се дефинира само за синтаксиса, който е определен за него в спецификацията на езика. Например:
  - Не може да се дефинира унарен оператор за делене /, тъй като в спецификацията на езика този оператор е дефиниран като бинарен.
  - Не може да се дефинира бинарен оператор за логическо отрицание !, тъй като в спецификацията на езика този оператор е дефиниран като унарен.
  - Операторът -, обаче, може да бъде предефиниран като унарен и като бинарен оператор, тъй като в спецификацията на езика са дефинирани и двата варианта на оператора.

Предефиниране на оператора за изход <<

- Операторът за изход << е бинарен оператор. Първият аргумент на оператора за изход задължително трябва да бъде от типа ostream.
- Типичният начин за предефиниране на оператора за изход е той да бъде дефиниран като функция извън рамките на класа по следният начин:

```
ostream & operator <<( ostream & out , cons t Point & p);
```

Пример

```
1 ostream& operator<<(ostream& out,
2                      const Point& p) {
3   out << "point(" << p.get_x() << ",␣"
4       << p.get_y() << ")";
5   return out;
6 }
```

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;
}
```

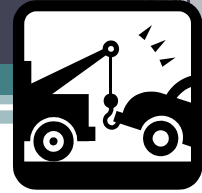
++ distance; //OK
or:

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:

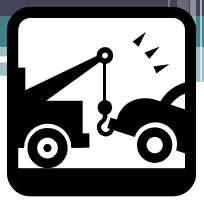
```
d3 = d1 + d2;          // for Dbl types
```

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;
}
```

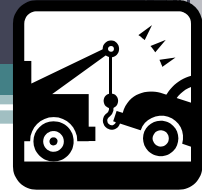
See the following list For CLS functions

Redefinable Operators



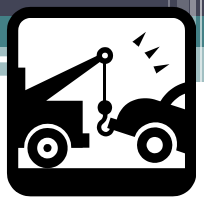
Operator	Name	Type
,	Comma	Binary
!	Logical NOT	Unary
!=	Inequality	Binary
%	Modulus	Binary
%=	Modulus assignment	Binary
&	Bitwise AND	Binary
&	Address-of	Unary
&&	Logical AND	Binary
&=	Bitwise AND assignment	Binary
( )	Function call	—
( )	Cast Operator	Unary
*	Multiplication	Binary
*	Pointer dereference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

Redefinable Operators



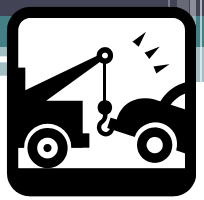
–	Subtraction	Binary
–	Unary negation	Unary
––	Decrement1	Unary
–=	Subtraction assignment	Binary
–>	Member selection	Binary
–>*	Pointer-to-member selection	Binary
/	Division	Binary
/=	Division assignment	Binary
<	Less than	Binary
<<	Left shift	Binary
<<=	Left shift assignment	Binary
<=	Less than or equal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

Redefinable Operators



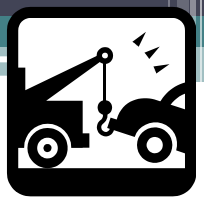
[ ]	Array subscript	—
^	Exclusive OR	Binary
^=	Exclusive OR assignment	Binary
|	Bitwise inclusive OR	Binary
|=	Bitwise inclusive OR assignment	Binary
||	Logical OR	Binary
~	One's complement	Unary
delete	Delete	—
new	New	—
conversion operators	conversion operators	Unary

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	^	…

.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;
        }
```