OPERATOR OVERLOADING AND CASTING OPERATOR

Bùi Tiến Lên

2022

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

Contents

Operator Overloading

- Unary Operators
- Binary Operators
- Function Operator
- Friend functions
- Overloading cin and cout

Operators in C++

- On a syntactical level, there is very little that differentiates an operator from a function, save for the use of the keyword `operator`. An operator declaration looks quite like a function declaration

Syntax

```
return_type operator operator_symbol (...parameter list...);
```

Operators in C++ (cont.)

Usage

- Operator is a global function or a static member function

```
object1 operator_symbol object2
operator operator_symbol (object1, object2)
```

- Operator is the member of a class

```
object1 operator_symbol object2
object1.operator operator_symbol (object2)
```

Operator Overloading

Concept 1

C++ allows us to redefine how standard operators work when used with class objects.

- C++ provides many operators to manipulate data of the primitive data types.
- However, what if you wish to use an operator to manipulate class objects?

Overloading guidelines

- **Do** what users expect for that operator.
- **Define** them if they make logical sense; e.g. *subtraction* of *dates* are ok but not *multiplication* or *division*.
- **Provide** *a complete set* of properly related operators: `a = a + b` and `a += b` have the same effect.

Limitations

- Only built-in operators can be overloaded.
- Cannot overload operators for built-in data types.
- The arity of the operators cannot be changed
- The precedence of the operators remains same.

Operators Cannot Be Overloaded

Operator	Name
`.`	Member selection
`.*`	Pointer-to-member selection
`::`	Scope resolution
`? :`	Conditional ternary operator
`sizeof`	Gets the size of an object/class type

- Operators `=`, `->`, `[]`, `()` can only be overloaded by non-static functions

Unary Operators

- Operators that function on a single operand are called *unary operators.*
- The typical definition of a unary operator implemented as a global function or a static member function is

```
return_type operator operator_type (parameter_type) {
    // ... implementation
}
```

- A unary operator that is the member of a class is defined as

```
return_type operator operator_type () {
    // ... implementation
}
```

The unary operators can be overloaded

Operator	Name
++	Increment
--	Decrement
*	Pointer dereference
->	Member selection
!	Logical NOT
&	Address-of
~	One's complement
+	Unary plus
-	Unary negation
Conversion operators	Conversion operators

Operators (++/--)

- The prefix increment operator (++) within the class declaration

```
CDate& operator ++ () {
    // operator implementation code
    return *this;
}
```

- The postfix increment operator (++) has a different return value and an input parameter (that is not always used):

```
CDate operator ++ (int) {
   // Store a copy of the current state of the object,
   // before incrementing day
   CDate Copy (*this);
   // operator implementation code (that increments this object)
   // Return the state before increment was performed
   return Copy;
}
```

Conversion Operators

- Convert CDate to string

```
class CDate {
private:
  int m_iDay, m_iMonth, m_iYear;
public:
  ...
  operator string() {
    ostringstream formattedDate;
    formattedDate << m_iDay << "/" << m_iMonth << "/" <<
        m_iYear;
    return formattedDate.str();
  }
};
```

Binary Operators

- Operators that function on two operands are called *binary operators*.
- The definition of a binary operator implemented as a global function or a static member function is the following:

```
return_type operator_type (parameter1, parameter2);
```

- The definition of a binary operator implemented as a class member is

```
return_type operator_type (parameter);
```

- The reason the class member version of a binary operator accepts only one parameter is that the second parameter is usually derived from the attributes of the class itself.

The binary operators can be overloaded

<table>
<tr><th>Operator</th><th>Name</th><th>Operator</th><th>Name</th></tr>
<tr><td>,</td><td>Comma</td><td><</td><td>Less than</td></tr>
<tr><td>!=</td><td>Inequality</td><td><<</td><td>Left shift</td></tr>
<tr><td>%</td><td>Modulus</td><td><<=</td><td>Left shift/assignment</td></tr>
<tr><td>%=</td><td>Modulus/assignment</td><td><=</td><td>Less than or equal to</td></tr>
<tr><td>&</td><td>Bitwise AND</td><td>=</td><td>Assignment, Copy Assignment and Move Assignment</td></tr>
<tr><td>&&</td><td>Logical AND</td><td>==</td><td>Equality</td></tr>
<tr><td>&=</td><td>Bitwise AND/assignment</td><td>></td><td>Greater than</td></tr>
<tr><td>*</td><td>Multiplication</td><td>>=</td><td>Greater than or equal to</td></tr>
<tr><td>*=</td><td>Multiplication/assignment</td><td>>></td><td>Right shift</td></tr>
<tr><td>+</td><td>Addition</td><td>>>=</td><td>Right shift/assignment</td></tr>
<tr><td>+=</td><td>Addition/assignment</td><td>^</td><td>Exclusive OR</td></tr>
<tr><td>-</td><td>Subtraction</td><td>^=</td><td>Exclusive OR/assignment</td></tr>
<tr><td>-=</td><td>Subtraction/assignment</td><td>|</td><td>Bitwise inclusive OR</td></tr>
<tr><td>->*</td><td>Pointer-to-member selection</td><td>|=</td><td>Bitwise inclusive OR/assignment</td></tr>
<tr><td>/</td><td>Division</td><td>||</td><td>Logical OR</td></tr>
<tr><td>/=</td><td>Division/assignment</td><td>[]</td><td>Subscript operator</td></tr>
</table>

Overloading Copy Assignment Operator (=)

- To ensure deeper copies, as with the copy constructor, we need to specify a copy assignment operator

```
ClassType& operator= (const ClassType& CopySource)
{
    if(this != &copySource)  // do not copy itself
    {
        // Assignment operator implementation
    }
    return *this;
}
```

Overloading Equality (==) and Inequality (!=) Operators

- It is a good practice to define the comparison operators. A generic expression of the equality operator is the following:

```
bool operator== (const ClassType& compareTo) {
    // comparison code here, return true if equal else false

}
```

- The inequality operator can reuse the equality operator:

```
bool operator!= (const ClassType& compareTo) {
    // comparison code here, return true if inequal else
        false

}
```

Subscript Operator ([])

- The operator that allow array-style [] access to a class is called *subscript operator*.
- The typical syntax of a subscript operator is:

```
return_type& operator [] (subscript_type& subscript);
```

- We can implement two subscript operators—one as a const function and the other as a `non-const` one

```
// use to write / change Buffer at Index
char& operator [] (int nIndex);
// used only for accessing char at Index
char& operator [] (int nIndex) const;
```

Function Operator ()

- The operator () that make objects behave like a function is called a function operator.
- They find application in the **standard template library** (STL) and are typically used in STL algorithms.

```
#include <iostream>
#include <string>
using namespace std;
class CDisplay {
public:
   void operator () (string Input) const {
     cout << Input << endl;
  }
};
int main () {
  CDisplay mDisplayFuncObject;
  // equivalent to mDisplayFuncObject.operator () ("Display this string!");
  mDisplayFuncObject ("Display this string!");
  return 0;
}
```

Friend functions

- With the keyword friend, we grant access to other functions or classes
- Use member functions if you can. Only choose friend functions when you have to.
- Sometimes, friend functions are good: Cannot modify original class, e.g. `ostream`

```
class CSample {
private:
  int m_a, m_b;
public:
  ...
  friend int Compute(CSample x);
}
```

```
int Compute(CSample x) {
  return x.m_a+x.m_b;
}
int main() {
  CSample x;
  ...
  cout<<"The result is:"<<Compute(x);
}
```

Overloading `cin` and `cout`

- We cannot access to the `istream` or `ostream` code → cannot overload << or >> as member functions
- They cannot be members of the user-defined class because the first parameter must be an object of that type
- Operators << and >> must be non-members, but it needs to access to private data members make them friend functions

Example

```
class CFraction {
private:
  int numerator, denominator;
public:
  ...
  friend ostream& operator << (ostream&, const CFraction&);
}
ostream& operator << (ostream& out, const CFraction& x) {
  out << x.numerator << " / " << x.denominator;
  return out;
}
int main() {
  CFraction a;
  ...
  cout << a;
  return 0;
}
```

Casting Operators

- The Need for Casting
- C-Style Casting
- The C++ Casting Operators
- Problems with the C++ Casting Operators

The Need for Casting

- In a perfectly **type-safe** and **type-strong** world comprising well-written C++ applications, there should be no need for casting and for casting operators.
- However, we live in a real world where modules programmed by a lot of different people and vendors often using different environments have to work together.
- To make this happen, compilers very often need to be instructed to interpret data in ways that make them compile and the application function correctly.

C-Style Casting

The usage syntax of the C-Style casting

```
destination_type dest = (destination_type)src;
```

- Most C++ compilers won't even let you get away with this

```
char* pszString = "Hello World!";
// error: cannot convert char* to int*
int* pBuf = pszString;
```

- C++ compilers still do see the need to be backward compliant to keep old and legacy code building

```
// Cast one problem away, create another
int* pBuf = (int*)pszString;
```

The C++ Casting Operators

The four C++ casting operators are

- `static_cast`
- `dynamic_cast`
- `reinterpret_cast`
- `const_cast`

The usage syntax of the casting operators is consistent:

```
destination_type  o=cast_type<destination_type>(object_to_be_casted);
```

```
destination_type* p=cast_type<destination_type*>(pointer_to_be_casted);
```

```
destination_type& r=cast_type<destination_type&>(reference_to_be_casted);
```

Using `static_cast`

- `static_cast` is a mechanism that can be used to convert pointers between related types, and perform explicit type conversions for standard data types that would otherwise happen automatically or implicitly.
- `static_cast` implements a basic compile-time check to ensure that the pointer is being cast to a related type.
- Using `static_cast`, a pointer can be upcasted to the base type, or can be down-casted to the derived type

```
Base* pBase = new Derived ();     // construct a Derived object
Derived* pDerived = static_cast<Derived*>(pBase);     // ok!

// CUnrelated is not related to Base via any inheritance hierarchy
CUnrelated* pUnrelated = static_cast<CUnrelated*>(pBase); // Error
// The cast above is not permitted as types are unrelated
```

Using dynamic_cast

- Dynamic casting actually executes the cast at runtime.
- Dynamic casting works with only **polymorphic type**
- The result of a dynamic_cast operation can be checked to see whether the attempt at casting succeeded.
- The typical usage syntax of the dynamic_cast operator is

```
destination_type* pDest = dynamic_cast<destination_type*>(pSource);
// Check for success of the casting operation before using pointer
if (pDest)
    pDest->CallFunc();
```

Listing 1

- Using dynamic casting to tell whether an fish object is a tuna or a carp

```
#include <iostream>
using namespace std;
class Fish {
public:
    virtual void Swim() {
        cout << "Fish swims in water" << endl;
    }
    // base class should always have virtual destructor
    virtual ~Fish() {}
};
class Tuna: public Fish {
public:
    void Swim() {
        cout << "Tuna swims real fast in the sea" << endl;
    }
    void BecomeDinner() {
        cout << "Tuna became dinner in Sushi" << endl;
```

Listing 1 (cont.)

```
    }
};
class Carp: public Fish {
public:
    void Swim() {
        cout << "Carp swims real slow in the lake" << endl;
    }
    void Talk() {
        cout << "Carp talked crap" << endl;
    }
};
void DetectFishType(Fish* InputFish) {
    Tuna* pIsTuna = dynamic_cast <Tuna*>(InputFish);
    if (pIsTuna) {
        cout << "Detected Tuna. Making Tuna dinner: " << endl;
        pIsTuna->BecomeDinner();    // calling Tuna::BecomeDinner
    }
    Carp* pIsCarp = dynamic_cast <Carp*>(InputFish);
    if(pIsCarp) {
```

Listing 1 (cont.)

```
        cout << "Detected Carp. Making carp talk: " << endl;
        pIsCarp->Talk();  // calling Carp::Talk
    }
    cout << "Verifying type using virtual Fish::Swim: " << endl;
    InputFish->Swim(); // calling virtual function Swim
}
int main() {
    Carp myLunch;
    Tuna myDinner;
    DetectFishType(&myDinner);
    cout << endl;
    DetectFishType(&myLunch);
    return 0;
}
```

Using `reinterpret_cast`

- `reinterpret_cast` is the closest a C++ casting operator gets to the C-style cast.
- It really does allow the programmer to cast one object type to another, regardless of whether or not the types are related; that is, it forces a reinterpretation of type using a syntax as seen in the following sample:

```
Base * pBase = new Base ();
CUnrelated * pUnrelated = reinterpret_cast<CUnrelated*>(pBase);
// The code above was not good programming,
// even when it compiles!
```

Using const_cast

- const_cast enables you to turn off the const access modifier to an object.
- Consider a problem

```
class SomeClass {
public:
    // ...
    void DisplayMembers ();
};
void DisplayAllData (const SomeClass& mData) {
    mData.DisplayMembers ();  // Compile failure
    // reason for failure: call to a non-const member
    // using a const reference
}
```

Using const_cast (cont.)

- Solution

```
void DisplayAllData (const SomeClass& mData) {
    SomeClass& refData = const_cast <SomeClass&>(mData);
    refData.DisplayMembers();     // Allowed!
}
```

- Note also that const_cast can be used with pointers

```
void DisplayAllData (const SomeClass* pData) {
    // pData->DisplayMembers(); Error
    // attempt to invoke a non-const function!
    SomeClass* pCastedData = const_cast <SomeClass*>(pData);
    pCastedData->DisplayMembers();     // Allowed!
}
```

Problems with the C++ Casting Operators

- The syntax is cumbersome and non-intuitive to being redundant
- Let's simply compare this code

```
double dPi = 3.14159265;

// C++ style cast: static_cast
int Num = static_cast <int>(dPi); // result: nNum is 3

// C-style cast
int Num2 = (int)dPi;              // result: Num2 is 3

// leave casting to the compiler
int Num3 = dPi;                   // result: Num3 is 3
```

Workshop

Quiz

1. How does the compiler know whether an overloaded ++ operator should be used in prefix or postfix mode?

   . . . .

   . . . .

   . . . .

2. What is passed to the parameter of a class's operator = function?

   . . . .

   . . . .

   . . . .

3. Why shouldn't a class's overloaded = operator be implemented with a void operator function?

   . . . .

   . . . .

   . . . .

Exercises

1. Implement a `CFraction` class with
   - basic arithmetic operators: +, −, ∗, /
     Remember to handle

```
CFraction x, y;
y = x + 5;
y = 5 + x;
```

   - prefix and postfix increment operators `x++` and `++x`

References

Deitel, P. (2016).
C++: How to program.
Pearson.

Gaddis, T. (2014).
Starting Out with C++ from Control Structures to Objects.
Addison-Wesley Professional, 8th edition.

Jones, B. (2014).
Sams teach yourself C++ in one hour a day.
Sams.