Complementos de Programação de Computadores – Aula 4a
Overload de Operadores em C++

Mestrado Integrado em Electrónica Industrial e Computadores

Luís Paulo Reis

lpreis1970@gmail.com / lpreis@dsi.uminho.pt

Professor Associado do Departamento de Sistemas de Informação, Escola de Engenharia, Universidade do Minho, Portugal

(Slides Baseados em P.Deitel e H.Deitel 2010)

Outline

Objectives

- To understand how to redefine (overload) operators to work with new types
- To understand how to convert objects from one class to another class
- To learn when to, and when not to, overload operators
- To study several interesting classes that use overloaded operators
- To create an Array class.

18.1 Introduction

- **Previous Lessons:**
  - ADT's and `classes`
  - Function-call notation is cumbersome for certain kinds of classes, especially mathematical classes
- **In this lesson:**
  - We use C++'s built-in operators to work with class objects
- **Operator overloading**
  - Use traditional operators with user-defined objects
  - Straightforward and natural way to extend C++
  - Requires great care
    - When overloading is misused, programs become difficult to understand

18.1 Introduction

- **Use operator overloading to improve readability**
  - Avoid excessive or inconsistent usage
- **Format**
  - Write function definition as normal
  - Function name is keyword `operator` followed by the symbol for the operator being overloaded.
  - `operator+` would be used to overload the addition operator (+)

18.2 Fundamentals of Operator Overloading

- **Assignment operator (=)**
  - may be used with every class without explicit overloading
  - *memberwise assignment*
  - Same is true for the address operator (&)

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Fig. 18.1 Operators that can be overloaded.

- **Most of C++'s operators can be overloaded**

18.3 Restrictions on Operator Overloading

Operators that cannot be overloaded				
`.`	`.*`	`::`	`?:`	`sizeof`

- Arity (number of operands) cannot be changed
  - Unary operators remain unary, and binary operators remain binary
  - Operators **&**, *****, **+** and **-** each have unary and binary versions
    - Unary and binary versions can be overloaded separately
- No new operators can be created
  - Use only existing operators
- Built-in types
  - Cannot overload operators
  - You cannot change how two integers are added

18.4 Operator Functions as Class Members vs. as friend Functions

- **Operator functions**
  - Can be member or non-member functions
- **Overloading the assignment operators**
  - i.e: `()`, `[]`, `->`, `=`
  - Operator must be a member function
- **Operator functions as member functions**
  - Leftmost operand must be an object (or reference to an object) of the class
  - If left operand of a different type, operator function must be a non-member function
  - A non-member operator function must be a `friend` if `private` or `protected` members of that class are accessed directly

18.4 Operator Functions as Class Members vs. as friend Functions

- **Non-member overloaded operator functions**
  - Enable the operator to be commutative

```
HugeInteger bigInteger1;
long int number;
bigInteger1 = number + bigInteger1;
```

or

```
bigInteger1 = biginteger1 + number;
```

- **Overloaded << and >> operators**
  - Must have left operand of types `ostream &`, `istream &` respectively
  - It must be a non-member function (left operand not an object of the class)
  - It must be a `friend` function if it accesses private data members

```
// Fig. 18.3: fig18_03.cpp
// Overloading the stream-insertion and
// stream-extraction operators.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;
using std::ostream;
using std::istream;

#include <iomanip>

using std::setw;

class PhoneNumber {
   friend ostream &operator<<( ostream&, const PhoneNumber & );
   friend istream &operator>>( istream&, PhoneNumber & );

private:
   char areaCode[ 4 ];  // 3-digit area code and null
   char exchange[ 4 ];  // 3-digit exchange and null
   char line[ 5 ];      // 4-digit line and null
}; // end class PhoneNumber

```

fig18_03.cpp (1 of 3)

```cpp
// Overloaded stream-insertion operator (cannot be
// a member function if we would like to invoke it with
// cout << somePhoneNumber;).
ostream &operator<<( ostream &output, const PhoneNumber &num )
{
   output << "(" << num.areaCode << ") "
          << num.exchange << "-" << num.line;
   return output;     // enables cout << a << b << c;
} // end operator<< function

istream &operator>>( istream &input, PhoneNumber &num )
{
   input.ignore();                      // skip (
   input >> setw( 4 ) >> num.areaCode; // input area code
   input.ignore( 2 );                   // skip ) and space
   input >> setw( 4 ) >> num.exchange; // input exchange
   input.ignore();                      // skip dash (-)
   input >> setw( 5 ) >> num.line;      // input line
   return input;      // enables cin >> a >> b >> c;
} // end operator>> function

int main()
{
   PhoneNumber phone; // create object phone

```

fig18_03.cpp (2 of 3)

```cpp
   cout << "Enter phone number in the form (123) 456-7890:\n";

   // cin >> phone invokes operator>> function by
   // issuing the call operator>>( cin, phone ).
   cin >> phone;

   // cout << phone invokes operator<< function by
   // issuing the call operator<<( cout, phone ).
   cout << "The phone number entered was: " << phone << endl;
   return 0;
} // end function main
```

```
Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212
```

fig18_03.cpp (3 of 3)

18.6 Overloading Unary Operators

- **Overloading unary operators**
  - Avoid `friend` functions and `friend` classes unless absolutely necessary.
  - Use of `friends` violates the encapsulation of a class.
  - As a member function:

```
class String {
  public:
     bool operator!() const;
     ...
  };
```

18.7 Overloading Binary Operators

- **Overloaded binary operators**
  - Non-`static` member function, one argument
  - Non-member function, two arguments

```
class String {
public:
   const String &operator+=( const String & );
   ...
}; // end class String
```

```
  y += z;
```

equivalent to

```
y.operator+=( z );
```

18.7 Overloading Binary Operators

- **Example**

```
class String {
   friend const String &operator+=( String &,
                                    const String & );
   ...
}; // end class String
```

```
   y += z;
```

equivalent to

```
 operator+=( y, z );
```

18.8 Case Study: An Array class

- **This example Implements an Array class with:**
  - Range checking
  - Array assignment
  - Arrays that know their size
  - Outputting/inputting entire arrays with **<<** and **>>**
  - Array comparisons with **==** and **!=**

array1.h (1 of 2)

```cpp
// Fig. 18.4: array1.h
// Simple class Array (for integers)
#ifndef ARRAY1_H
#define ARRAY1_H

#include <iostream>

using std::ostream;
using std::istream;

class Array {
   friend ostream &operator<<( ostream &, const Array & );
   friend istream &operator>>( istream &, Array & );
public:
   Array( int = 10 );                    // default constructor
   Array( const Array & );               // copy constructor
   ~Array();                             // destructor
   int getSize() const;                  // return size
   const Array &operator=( const Array & ); // assign arrays
   bool operator==( const Array & ) const;  // compare equal

   // Determine if two arrays are not equal and
   // return true, otherwise return false (uses operator==).
   bool operator!=( const Array &right ) const
      { return ! ( *this == right ); }

```

array1.h (2 of 2)

```cpp
   int &operator[]( int );              // subscript operator
   const int &operator[]( int ) const;  // subscript operator
   static int getArrayCount();          // Return count of
                                        // arrays instantiated.
private:
   int size; // size of the array
   int *ptr; // pointer to first element of array
   static int arrayCount;  // # of Arrays instantiated
}; // end class Array

#endif
```

array1.cpp (1 of 6)

```cpp
// Fig 18.4: array1.cpp
// Member function definitions for class Array
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include <iomanip>

using std::setw;

#include <cstdlib>
#include <cassert>
#include "array1.h"

```

```cpp
// Initialize static data member at file scope
int Array::arrayCount = 0;   // no objects yet

// Default constructor for class Array (default size 10)
Array::Array( int arraySize )
{
   size = ( arraySize > 0 ? arraySize : 10 );
   ptr = new int[ size ]; // create space for array
   assert( ptr != 0 );    // terminate if memory not allocated
   ++arrayCount;          // count one more object

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = 0;          // initialize array
} // end Array constructor

// Copy constructor for class Array
// must receive a reference to prevent infinite recursion
Array::Array( const Array &init ) : size( init.size )
{
   ptr = new int[ size ]; // create space for array
   assert( ptr != 0 );    // terminate if memory not allocated
   ++arrayCount;          // count one more object

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = init.ptr[ i ];  // copy init into object
} // end Array constructor

```

array1.cpp (2 of 6)

```cpp
// Destructor for class Array
Array::~Array()
{
   delete [] ptr;            // reclaim space for array
   --arrayCount;             // one fewer object
} // end Array destructor

// Get the size of the array
int Array::getSize() const { return size; }

// Overloaded assignment operator
// const return avoids: ( a1 = a2 ) = a3
const Array &Array::operator=( const Array &right )
{
   if ( &right != this ) {  // check for self-assignment

      // for arrays of different sizes, deallocate original
      // left side array, then allocate new left side array.
      if ( size != right.size ) {
         delete [] ptr;         // reclaim space
         size = right.size;     // resize this object
         ptr = new int[ size ]; // create space for array copy
         assert( ptr != 0 );    // terminate if not allocated
      } // end if

      for ( int i = 0; i < size; i++ )
         ptr[ i ] = right.ptr[ i ];  // copy array into object
   } // end if

```

array1.cpp (3 of 6)

```cpp
   return *this;   // enables x = y = z;
} // end operator= function

// Determine if two arrays are equal and
// return true, otherwise return false.
bool Array::operator==( const Array &right ) const
{
   if ( size != right.size )
      return false;    // arrays of different sizes

   for ( int i = 0; i < size; i++ )
      if ( ptr[ i ] != right.ptr[ i ] )
         return false; // arrays are not equal

   return true;        // arrays are equal
} // end operator== function

// Overloaded subscript operator for non-const Arrays
// reference return creates an lvalue
int &Array::operator[]( int subscript )
{
   // check for subscript out of range error
   assert( 0 <= subscript && subscript < size );

   return ptr[ subscript ]; // reference return
} // end operator[] function

```

array1.cpp (4 of 6)

```cpp
// Overloaded subscript operator for const Arrays
// const reference return creates an rvalue
const int &Array::operator[]( int subscript ) const
{
   // check for subscript out of range error
   assert( 0 <= subscript && subscript < size );

   return ptr[ subscript ]; // const reference return
} // end operator[] function

// Return the number of Array objects instantiated
// static functions cannot be const
int Array::getArrayCount() { return arrayCount; }

// Overloaded input operator for class Array;
// inputs values for entire array.
istream &operator>>( istream &input, Array &a )
{
   for ( int i = 0; i < a.size; i++ )
      input >> a.ptr[ i ];

   return input;   // enables cin >> x >> y;
} // end operator>> function

```

array1.cpp (5 of 6)

```cpp
// Overloaded output operator for class Array
ostream &operator<<( ostream &output, const Array &a )
{
   int i;

   for ( i = 0; i < a.size; i++ ) {
      output << setw( 12 ) << a.ptr[ i ];

      if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
         output << endl;
   } // end for

   if ( i % 4 != 0 )
      output << endl;

   return output;   // enables cout << x << y;
} // end operator<< function
```

array1.cpp (6 of 6)

```cpp
// Fig. 18.4: fig18_04.cpp
// Driver for simple class Array
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include "array1.h"

```

fig18_04.cpp (1 of 4)

```cpp
int main()
{
   // no objects yet
   cout << "# of arrays instantiated = "
        << Array::getArrayCount() << '\n';

   // create two arrays and print Array count
   Array integers1( 7 ), integers2;
   cout << "# of arrays instantiated = "
        << Array::getArrayCount() << "\n\n";

   // print integers1 size and contents
   cout << "Size of array integers1 is "
        << integers1.getSize()
        << "\nArray after initialization:\n"
        << integers1 << '\n';

   // print integers2 size and contents
   cout << "Size of array integers2 is "
        << integers2.getSize()
        << "\nArray after initialization:\n"
        << integers2 << '\n';

```

fig18_04.cpp (2 of 4)

```
   // input and print integers1 and integers2
   cout << "Input 17 integers:\n";
   cin >> integers1 >> integers2;
   cout << "After input, the arrays contain:\n"
        << "integers1:\n" << integers1
        << "integers2:\n" << integers2 << '\n';

   // use overloaded inequality (!=) operator
   cout << "Evaluating: integers1 != integers2\n";
   if ( integers1 != integers2 )
      cout << "They are not equal\n";

   // create array integers3 using integers1 as an
   // initializer; print size and contents
   Array integers3( integers1 );

   cout << "\nSize of array integers3 is "
        << integers3.getSize()
        << "\nArray after initialization:\n"
        << integers3 << '\n';

   // use overloaded assignment (=) operator
   cout << "Assigning integers2 to integers1:\n";
   integers1 = integers2;
   cout << "integers1:\n" << integers1
        << "integers2:\n" << integers2 << '\n';

```

fig18_04.cpp (3 of 4)

```
   // use overloaded equality (==) operator
   cout << "Evaluating: integers1 == integers2\n";
   if ( integers1 == integers2 )
      cout << "They are equal\n\n";

   // use overloaded subscript operator to create rvalue
   cout << "integers1[5] is " << integers1[ 5 ] << '\n';

   // use overloaded subscript operator to create lvalue
   cout << "Assigning 1000 to integers1[5]\n";
   integers1[ 5 ] = 1000;
   cout << "integers1:\n" << integers1 << '\n';

   // attempt to use out of range subscript
   cout << "Attempt to assign 1000 to integers1[15]" << endl;
   integers1[ 15 ] = 1000;  // ERROR: out of range

   return 0;
} // end function main
```

fig18_04.cpp (4 of 4)

Array output (1 of 1)

```
# of arrays instantiated = 0
# of arrays instantiated = 2
Size of array integers1 is 7
Array after initialization:
           0           0           0           0
           0           0           0
Size of array integers2 is 10
Array after initialization:
           0           0           0           0
           0           0           0           0
           0           0

Input 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
After input, the arrays contain:
integers1:
           1           2           3           4
           5           6           7
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 != integers2
They are not equal
Size of array integers3 is 7
Array after initialization:
           1           2           3           4
           5           6           7
```

Array output (2 of 2)

```
Assigning integers2 to integers1:
integers1:
           8           9          10          11
          12          13          14          15
          16          17
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 == integers2
They are equal

integers1[5] is 13
Assigning 1000 to integers1[5]
integers1:
           8           9          10          11
          12        1000          14          15
          16          17

Attempt to assign 1000 to integers1[15]
Assertion failed: 0 <= subscript && subscript < size,
file Array1.cpp, line 95 abnormal program termination
```

18.9 Converting between Types

- **Cast operator**
  - Convert objects into built-in types or other objects
  - Conversion operator must be a non-`static` member function
  - Cannot be a `friend` function
  - Do not specify return type

  For user-defined class `A`:

```
A::operator char *() const;
A::operator int() const;
A::operator otherClass() const;
```

  - When compiler sees `(char *) s` it calls: `s.operator char*()`

- **The compiler can call these functions to create temporary objects**
  - If s is not of type `char *`

  Calls `A::operator char *() const;` for

```
cout << s;
```

18.10 Overloading ++ and --

- **Pre/post-incrementing/decrementing operators**
  - Can be overloaded
  - How does the compiler distinguish between the two?
  - Prefix versions overloaded same as any other prefix unary operator would be. i.e. `d1.operator++();` for `++d1;`
- **Postfix versions**
  - When compiler sees postincrementing expression, such as `d1++;`
    - Generates the member-function call:

```
d1.operator++( 0 );
```

  - Prototype:

```
Date::operator++( int );
```

Escola de Engenharia
Universidade do Minho

Complementos de Programação de Computadores – Aula 4a
Overload de Operadores em C++

Mestrado Integrado em Electrónica Industrial e Computadores

Luís Paulo Reis

lpreis1970@gmail.com / lpreis@dsi.uminho.pt

Professor Associado do Departamento de Sistemas de Informação, Escola de Engenharia, Universidade do Minho, Portugal

(Slides Baseados em P.Deitel e H.Deitel 2010)