



# Operator Overloading

---

## Lab 7: Operator Overloading for Array Class

Rung-Bin Lin

International Bachelor Program in Informatics  
Yuan Ze University

03/31/2022

# Operator Overloading

- Make the C++ built-in operators available for user-defined (class) objects so that the use of these operators is naturally extended to the objects of user-defined classes.

# Operator Overloading VS. Function Overloading

- **Operator overloading is a kind of function overloading.**
- **Function overloading**
  - **Several functions of the same name can be defined, as long as they have different signatures.**
    - **A signature is a combination of a function's name and its parameter types (in order).**
  - **Overloaded functions can have different return types, but if they do, they must also have different parameter lists.**
  - **Overloaded functions are normally used to perform similar operations that involve different program logic on different data types.**

# Function Overloading

```
1 // Fig. 5.23: fig05_23.cpp
2 // Overloaded functions.
3 #include <iostream>
4 using namespace std;
5
6 // function square for int values
7 int square( int x )
8 {
9     cout << "square of integer " << x << " is ";
10    return x * x;
11 } // end function square with int argument
12
13 // function square for double values
14 double square( double y )
15 {
16    cout << "square of double " << y << " is ";
17    return y * y;
18 } // end function square with double argument
19
```

**Fig. 5.23** | Overloaded square functions. (Part I of 2.)

# Function Overloading cont.

```
20  int main()
21  {
22      cout << square( 7 ); // calls int version
23      cout << endl;
24      cout << square( 7.5 ); // calls double version
25      cout << endl;
26  } // end main
```

square of integer 7 is 49  
square of double 7.5 is 56.25

**Fig. 5.23** | Overloaded square functions. (Part 2 of 2.)

# Restrictions on Operator Overloading <sup>(1)</sup>

- Operators that are overloaded as **non-static member functions**
  - The **leftmost operand** must be an object of the operator's class.
    - An overloaded binary operator + used in **X+Y** will be transformed into a call **X.operator+(Y)**. Hence, **X** must be an object of the operator's class. However, Y may be or may not be an object of the operator's class.
- Operator precedence can not be changed by overloading
- No new operators can be created.

# Restrictions on Operator Overloading (2)

- Operators that are overloaded as **global functions**
  - The leftmost operand may be an object of a different type or a fundamental type.
  - Like stream insertion operator << (or stream extraction operator >>), **cout << X**, if transformed into a function call, becomes **cout.operator<<(X)**. Since **cout** is not an object of the operator<<'s class, we cannot implement operator<< as a member function of the object X's class. Hence, operator<< should be implemented as a global function. Hence, **cout<<X** will be transformed into a call **operator<<(cout, X)**.
  - Usually make **friend** to the class whose objects will use the operator.

# Operators Overloaded as Member Functions

```
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 & ); // assignment operator
```

```
    bool operator==( const Array & ) const; // equality operator
```

```
    // inequality operator; returns opposite of == operator
```

```
    bool operator!=( const Array &right ) const {
```

```
        return ! ( *this == right ); // invokes Array::operator==
```

```
    } // end function operator!=
```

```
    // subscript operator for non-const objects returns modifiable lvalue
```

```
    int &operator[]( int );
```

```
    // subscript operator for const objects returns rvalue
```

```
    int operator[]( int ) const;
```

```
private:
```

```
    int size; // pointer-based array size
```

```
    int *ptr; // pointer to first element of pointer-based array
```

```
}; // end class Array
```



# Operators Overloaded as Global Functions

- Make friend to the class whose data members will be accessed by the operator.

```
class Array {  
    friend ostream &operator<<( ostream &, const Array & );  
    friend istream &operator>>( istream &, Array & );  
public:  
    Array( int = 10 ); // default constructor  
    ...  
    ...  
}
```

It is important to study the code in Fig. 11.6~Fig. 11.8, Fig. 11.9~Fig. 11.11 .

# Lab 7: Class Array

- Add an operator `+` into the code in Fig. 11.6, 11.7, 11.8 to concatenate two arrays, said array A and array B, into an array, said C. Place the elements of second array right after the elements of first array.
  - Example,  $A=(1,2,3)$  and  $B=(4,5,6,7)$ , after executing  $A+B$ , C will be  $(1,2,3,4,5,6,7)$
- Add an operator `<<` to shift the elements in an array of n elements to the left by **k** places. If  $k > \text{arraySize}$ , i.e., the size of the array, the element at i-th place will be moved by  $(k \bmod \text{arraySize})$  places. For example,  $A << 1$  means that an array element  $A[i]$  will be moved to  $(i-1)$ th place if  $i-1 \geq 0$ .  $A[0]$  will be placed at  $(\text{arraySize}-1)$ -th place.
  - Example,  $A=(1,2,3,4,5)$ , after performing  $A << 7$ , A will become  $A=(3, 4, 5, 1, 2)$
- Add an operator `+` to add together all the elements in an array.
  - For example, if  $A=(1,2,3,4,5)$ , then  $+A$  will be 15. **For implementing this sort of unary operator, you may refer to the materials in page 512.**

- Add an operator  $+$  to add a constant to every element in an array.
  - For example, if  $A=(1,2,3,4,5)$ , then  $1+A$  will make  $A=(2,3,4,5,6)$ .
- Add an operator  $+$  to add every element in an array with a constant.
  - For example, if  $A=(1,2,3,4,5)$ , then  $A+2$  will make  $A=(3,4,5,6,7)$ .

Note that one of overload operator  $+$  should be implemented as a global function.

# Main() Function

- The main function in Fig. 11.8 should remain the same.
- Before line 63 in the main function in Fig. 11.8, you should add the following statements:

```
cout << "integers2 :\n" << integers2 << endl;
cout << "integers3 :\n" << integers3 << endl;
Array C;
C = integers1 + integers2 + integers3;
cout << "Array C = integers1 + integers2 + integers3: \n" << C ;
int k = 30;
C << k;
cout << "\nShifting the elements of C to the left by " << k << " places:\n" << C << endl;
cout << "The sum of the elements in array C = " << +C << endl;
cout << "Array integers1 = 1+integers1+2 =\n" << 1+integers1+2 << endl;
```

# Output

```
Size of Array integers1 is 7
Array after initialization:
    0      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      0      0

Enter 17 integers:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

After input, the Arrays contain:
integers1:
    0      1      2      3
    4      5      6      7

integers2:
    7      8      9     10
   11     12     13     14
   15     16

Evaluating: integers1 != integers2
integers1 and integers2 are not equal
Call copy constructor!

Size of Array integers3 is 7
Array after initialization:
    0      1      2      3
    4      5      6      7

Assigning integers2 to integers1:
integers1:
    7      8      9     10
   11     12     13     14
   15     16

integers2:
    7      8      9     10
   11     12     13     14
   15     16

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 12
```

```
Assigning 1000 to integers1[5]
integers1:
    7      8      9     10
   11     1000    13     14
   15     16

integers2 :
    7      8      9     10
   11     12     13     14
   15     16

integers3 :
    0      1      2      3
    4      5      6      7

Array C = integers1 + integers2 + integers3:
    7      8      9     10
   11     1000    13     14
   15     16      7      8
    9     10     11     12
   13     14     15     16
    0      1      2      3
    4      5      6      7

Shifting the elements of C to the left by 30 places:
    10     11     1000    13
   14     15     16      7
    8      9     10     11
   12     13     14     15
   16      0      1      2
    3      4      5      6
    7      8      9      7

The sum of the elements in array C = 1239
Array integers1 = 1+integers1+2:
    10     11     12     13
   14     1003    16     17
   18     19

Attempt to assign 1000 to integers1[15]

Error: Subscript 15 out of range
```

The output marked in the red region should be correct.

# Key Points for Grading

- Check whether the overloaded operators `+` and `>>` are actually implemented.
- Check whether the added code is actually added in the `main()` function.
- Check whether the output marked in the red region is indeed correct.

```

#ifndef ARRAY_H
#define ARRAY_H

#include <iostream>
using namespace std;

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 & ); // assignment operator
    bool operator==( const Array & ) const; // equality operator

    // inequality operator; returns opposite of == operator
    bool operator!=( const Array &right ) const
    {
        return ! ( *this == right ); // invokes Array::operator==
    } // end function operator!=

    // subscript operator for non-const objects returns modifiable lvalue
    int &operator[]( int );

    // subscript operator for const objects returns rvalue
    int operator[]( int ) const;
private:
    int size; // pointer-based array size
    int *ptr; // pointer to first element of pointer-based array
}; // end class Array

#endif

```

```

// default constructor for class Array (default size 10)
Array::Array( int arraySize )
{
    size = ( arraySize > 0 ? arraySize : 10 ); // validate arraySize
    ptr = new int[ size ]; // create space for pointer-based array

    for ( int i = 0; i < size; i++ )
        ptr[ i ] = 0; // set pointer-based array element
} // end Array default constructor

// copy constructor for class Array;
// must receive a reference to prevent infinite recursion
Array::Array( const Array &arrayToCopy )
    : size( arrayToCopy.size )
{
    ptr = new int[ size ]; // create space for pointer-based array

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

// destructor for class Array
Array::~~Array()
{
    delete [] ptr; // release pointer-based array space
} // end destructor

// return number of elements of Array
int Array::getSize() const
{
    return size; // number of elements in Array
} // end function getSize

// overloaded assignment operator;
// const return avoids: ( a1 = a2 ) = a3
const Array &Array::operator=( const Array &right )
{
    if ( &right != this ) // avoid self-assignment
    {
        // for Arrays of different sizes, deallocate original
        // left-side array, then allocate new left-side array
        if ( size != right.size )
        {
            delete [] ptr; // release space

```

---

```

    size = right.size; // resize this object
    ptr = new int[ size ]; // create space for array copy
} // end inner if

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

return *this; // enables x = y = z, for example
} // end function operator=

// 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 number of elements

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

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

// overloaded subscript operator for non-const Arrays;
// reference return creates a modifiable lvalue
int &Array::operator[]( int subscript )
{
    // check for subscript out-of-range error
    if ( subscript < 0 || subscript >= size )
    {
        cerr << "Error: Subscript " << subscript
            << " out of range" << endl;
        exit( 1 ); // terminate program; subscript out of range
    } // end if

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

```

```

int main()
{
    Array integers1( 7 ); // seven-element Array
    Array integers2; // 10-element Array by default

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

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

    // input and print integers1 and integers2
    cout << "\nEnter 17 integers:" << endl;
    cin >> integers1 >> integers2;

    cout << "\nAfter input, the Arrays contain:\n"
        << "integers1:\n" << integers1
        << "integers2:\n" << integers2;

    // use overloaded inequality (!=) operator
    cout << "\nEvaluating: integers1 != integers2" << endl;

    if ( integers1 != integers2 )
        cout << "integers1 and integers2 are not equal" << endl;

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

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

```



```

// use overloaded assignment (=) operator
cout << "\nAssigning integers2 to integers1:" << endl;
integers1 = integers2; // note target Array is smaller

cout << "integers1:\n" << integers1
    << "integers2:\n" << integers2;

// use overloaded equality (==) operator
cout << "\nEvaluating: integers1 == integers2" << endl;

if ( integers1 == integers2 )
    cout << "integers1 and integers2 are equal" << endl;

// use overloaded subscript operator to create rvalue
cout << "\nintegers1[5] is " << integers1[ 5 ];

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

// attempt to use out-of-range subscript
cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
integers1[ 15 ] = 1000; // ERROR: out of range
} // end main

```

Insert the  
statements here



The statements  
being inserted:

```

cout << "integers2 :\n" << integers2 << endl;
cout << "integers3 :\n" << integers3 << endl;
Array C;
C = integers1 + integers2 + integers3;
cout << "Array C = integers1 + integers2 + integers3: \n" << C ;
int k = 30;
C << k;
cout << "\nShifting the elements of C to the left by " << k << " places:\n" << C << endl;
cout << "The sum of the elements in array C = " << +C << endl;
cout << "Array integers1 = 1+integers1+2 =\n" << 1+integers1+2 << endl;

```