Advanced Programming

Implementing Your Own Operators

Zeinab Zali

References: (1) "C++ How to program" Deitel&Deitel, (2) "A Tour of C++" Bjarne Stroustrup, (3) Other useful learning pages such as geeksforgeeks and tutorialpoints

ECE Department, Isfahan University of Technology

Built-in Arrays in C++

built in arrays in C++ have some limitations, so we want to define a new Array class.

- C++ does not check whether subscripts fall outside the range of the array.
- Built-in arrays of size n must number their elements 0,..., n – 1; alternate subscript ranges are not allowed.
- An entire built-in array cannot be input or output at once.

Built-in Arrays in C++

- Two built-in arrays cannot be meaningfully compared with equality or relational operators
- When a built-in array is passed to a general-purpose function designed to handle arrays of any size, the array's size must be passed as an additional argument.
- One built-in array cannot be assigned to another with the assignment operator(s).

Array Class

```
1   class Array {
2   private:
3       size_t size; // pointer-based array size
4       int* ptr; // pointer to first element of pointer-
    based array

6   public:
7       explicit Array(int = 10); // default constructor
8       Array(const Array&); // copy constructor
9       ~Array(); // destructor
10      size_t getSize() const; // return size
11  };
```

Using the Array Class

Assume we want to use the array class with some operators

```
1#include "Array1.h"
2using namespace std;
3int main() {
4     Array integers1{7}; // seven-element Array
5     Array integers2; // 10-element Array by default

6     // input and print integers1 and integers2
7     cout << "\nEnter 17 integers:" << endl;
8     cin >> integers1 >> integers2;
9     cout << "\nThe Arrays contain:\n"
10    << "integers1: " << integers1
11    << "integers2: " << integers2 ;
```

Using the Array Class

```
12      // use inequality (!=) operator
13      cout << "\nEvaluating: integers1 != integers2" << endl;
14      if ( integers1 != integers2 ) {
15           cout << "integers1 and integers2 are not equal" <<
        endl;
16      }
17      // create Array integers3 using integers1 as an
18      // initializer; print size and contents
19      Array integers3{integers1}; // invokes copy constructor
20      cout << "\nSize of Array integers3 is " << integers3.
        getSize()
21      << "\nArray after initialization: " << integers3;

22      // use assignment (=) operator
23      cout << "\nAssigning integers2 to integers1:" << endl;
24      integers1 = integers2; // note target Array is smaller
25      cout << "integers1: " << integers1 << "integers2: " <<
        integers2 ;
```

Using the Array Class

```
26      // use equality (==) operator
27      cout << "\nEvaluating: integers1 == integers2" << endl;
28      if ( integers1 == integers2 ) {
29           cout << "integers1 and integers2 are equal" << endl
        ;
30      }

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

33      // use subscript operator to create lvalue
34      cout << "\n\nAssigning 1000 to integers1[5]" << endl;
35      integers1[5] = 1000;
36      cout << "integers1: " << integers1 ;
```

Using the Array Class

```
37      // attempt to use out-of-range subscript
38      try {
39           cout << "\nAttempt to assign 1000 to integers1[15]"
         << endl;
40           integers1[15] = 1000; // ERROR: subscript out of
        range
41      }
42      catch (out_of_range& ex) {
43           cout << "An exception occurred: " << ex.what() <<
        endl;
44      }
```

Operators of Array Class

As you see, we used some operators on objects of Array class:

- **<<** : stream insertion (print)
- **>>** : stream input (read)
- **=** : assignment
- **!=** : inequality
- **==** : equality
- **[]** : rvalue subscript
- **[]** : lvalue subscript

Operators of Array Class

- Who specifies **what operators** can be used on objects of array class?
- Who specifies **how operators** operates on objects of array class?

Case Study: String Class

For a customized String class, we may use some operators:

- **<<**: inserting an string on the output stream
- **>>**: reading an string from the input stream
- **+** : concatenating two strings
- **<** : comparing two strings (if the first one lower than second one)
- **many other operators**

Case Study: List Class

For a Linked List class, we may use some operators:

- <<: inserting all the elements of a list on the output stream
- + : concatenating two lists

What other operators do you propose for the List class?

Operator Overloading, What?

Operator Overloading is the process of enabling C++'s operators (+, -, *, +=, !=, ...) to work with objects of our new defined classes

- One Operator may operate differently on different classes of objects.
- You can overload most operators to be used with class objects—the compiler generates the appropriate code based on the types of the operands.
  - Ex: + on two integers performs addition of them to get their sum, + on two strings performs concatenation on them.

Operator Overloading, How?

To use an operator on an object of a class, you must define overloaded operator functions for that class.

- An operator is overloaded by writing a (1) non-static member function definition or (2) non-member function definition
- The function name starts with the keyword **operator** followed by the symbol for the operator being overloaded.
  - For example, the function name **operator+** would be used to overload the addition operator (+) for use with objects of a particular user-defined type.

C++ Default Operators

There are three default operators in C++ for objects of each new defined class, however you can also overload them.

- **The assignment operator (=)** may be used with most classes to perform memberwise assignment of the data members—each data member is assigned from the assignment's "source" object (on the right) to the "target" object (on the left).
  - Memberwise assignment is dangerous for classes with pointer member, so you must explicitly overload the assignment operator for such classes.
- **The address (&) operator** returns a pointer to the object; this operator also can be overloaded.
- **The comma operator** evaluates the expression to its left then the expression to its right, and returns the value of the latter expression.

Binary Overloaded Operators as Member Functions

Overloaded operator functions for binary operators can be member functions only when the left operand is an object of the class in which the function is a member, for examples

- operator+ for adddition of two strings that returns a new string as the result
- operator== for comparing two Array class that returns a bool value as the result
- operator

Equality Operator for Array Class (Member Function)

When overloading binary operator == as a non-static member function, if y and z are Array objects, then y == z is treated by the compiler as if y.operator==(z) had been written, so invoking the operator == member function with one argument.

```
1   // determine if two Arrays are equal and
2   // return true, otherwise return false
3   bool Array::operator==(const Array& right) const {
4       // arrays of different number of elements
5       if (size != right.size) {
6           return false;
7       }
8       for (size_t i{0}; i < size; ++i) {
9           if (ptr[i] != right.ptr[i]) {
10              // Array contents are not equal
11              return false;
12          }
13      }
14      return true; // Arrays are equal
15  }
```

Binary Overloaded Operators as Non-Member Functions

- As a non-member function, a binary operator must take two arguments—one of which must be an object (or a reference to an object) of the class that the overloaded operator is associated with.
- If y and z are Array objects or references to Array objects, then y == z is treated as if the call operator==(y, z) had been written in the program, invoking function operator== with two argumanents.

Equality Operator for Array Class (Non-member Function)

```
1   // determine if two Arrays are equal and
2   // return true, otherwise return false
3   bool operator==(const Array& left, const Array& right){
4       // arrays of different number of elements
5       if (left.size != right.size) {
6           return false;
7       }
8       for (size_t i{0}; i < left.size; ++i) {
9           if (left.ptr[i] != right.ptr[i]) {
10              // Array contents are not equal
11              return false;
12          }
13      }
14      return true; // Arrays are equal
15  }
```

Equality Operator for Array Class (Non-member Function)

Encapsulation Violation! How the non-member function operator== could access private members of Array class? such as size or ptr?

Freind Function and Class

A **friend function/class** of a class is a non-member function/other class that has the right to access the public and non-public class members

- Standalone functions, entire classes or member functions of other classes may be declared to be friends of another class
- Friendship is granted, not taken—for class B to be a friend of class A, class A must explicitly declare that class B is its friend

Freind Function and Class

- To declare a non-member function as a friend of a class, place the function prototype in the class definition and precede it with the keyword friend
- To declare all member functions of class B as friends of class A, place a declaration of the form "friend class B;" in the definition of class A.
- The friend declaration(s) can appear anywhere in a class and are not affected by access specifiers public or private.

Equality Operator for Array Class (Non-member Function)

Encapsulation Violation! How the non-member function operator== could access private members of Array class? such as size or ptr?

- We must define the function operator as the friend of Array class

```
1class Array {
2     friend bool operator==(const Array&, const Array&);
```

Binary Stream Insertion and Extraction

Stream Insertion(«) and Extraction operators(») operate on each object of a new defined class as right operand and an istream or ostream objects as left operand.

- So, we must overload these operators as non-member functions.

```
1class Array {
2     friend std::ostream& operator<<(std::ostream&, const
      Array&);
3     friend std::istream& operator>>(std::istream&, Array&);
```

Binary Stream Insertion and Extraction

```
1   // overloaded input operator for class Array;
2   // inputs values for entire Array
3   istream& operator>>(istream& input, Array& a) {
4       for (size_t i{0}; i < a.size; ++i) {
5           input >> a.ptr[i];
6       }
7       return input; // enables cin >> x >> y;
8   }

9   // overloaded output operator for class Array
10  ostream& operator<<(ostream& output, const Array& a) {
11      // output private ptr-based array
12      for (size_t i{0}; i < a.size; ++i) {
13          output << a.ptr[i] << " ";
14      }
15      output << endl;
16      return output; // enables cout << x << y;
17  }
```

Unary Operators as Member Function or Non-member Function

A unary operator for a class can be overloaded as a non-static member function with no arguments or as a non-member function with one argument that must be an object (or a reference to an object) of the class.

- Example: we can overload the increment operator for Array class to enable incrementing all the elements of an Array object at once.

Prefix Operator++: Unary Operator

Prefix operator++ for an Array object can be overloaded in two ways:

- if operator++ is overloaded as member function, ++a is invoked as a.operator++()

```
1    Array& Array::operator++(){
2        for(int i=0;i<size; i++)
3            ptr[i]++;
4        return *this;
5    }
```

- if operator++ is overloaded as non-member function, ++a is invoked as operator++(a)

```
1    Array& operator++(Array& a){
2        for(int i=0;i<a.size; i++)
3            a.ptr[i]++;
4        return a;
5    }
```

Postfix Operator++: Binary Operator

Overloading the postfix increment operator presents a challenge, because the compiler must be able to distinguish between the signatures of the overloaded prefix and postfix increment operator functions.

- Solution: a dummy value is used that enables the compiler to distinguish between the prefix and postfix increment operator functions

Postfix Operator++: Binary Operator

- if operator++ is overloaded as member function, when the compiler sees the postincrementing expression a++ , it generates the member-function call a.operator++(0) (Array operator++(int))
- if operator++ is overloaded as non-member function, when the compiler sees the expression a++ ,the compiler generates the function call operator++(a,0) (Array operator++(Array&,int)

Question: Why we declare return type of prefix++ as reference, but the return type of postfix++ as non reference?
Exercise: Implement the overloaded postfix++ for Array class?

Conversion Operators

A conversion operator (also called a cast operator) can be used to convert an object of one class to another type

- It must be a non-static member function.
- The function prototype *textMyClass::operator string() const;* declares an overloaded cast operator function for converting an object of class *MyClass* into a temporary *string* object
- It is called when compiler sees static_cast conversion

Static Cast

- static_cast is a compile-time cast and is the simplest type of cast that can be used
- It can be used explicitly.

```
1    string s1 = static_cast<string>(integers1);
2    string s2 = (string)integers1;
3    string s3 = string(integers1);
```

- It also does implicit conversions between types.

```
1    string s4 = integers1;
```

explicit Constructors

- In some situations, implicit conversions are undesirable or error-prone
- Any constructor that can be called with a single argument and is not declared explicit can be used by the compiler to perform an implicit conversion.
  - The constructor's argument is converted to an object of the class in which the constructor is defined.
  - if this constructor is declared explicit, it could not be misused by the compiler to perform an implicit conversion
  - if class A has a constructor with one argument and is not declared explicit, the compiler assumes the constructor is a **conversion constructor** and uses it to convert the argument into an object of class A.

explicit Constructors

content...

Assignment Operator

When the compiler sees the expression integers1 = integers2, the compiler invokes member function operator= with the call of assignment operator

- Default memberwise assignment assigns each data member of the object on the right of the assignment individually to the same data member in the object on the left of the assignment operator

Common Programming Error 10.4

Not providing a copy constructor and overloaded assignment operator for a class when objects of that class contain pointers to dynamically allocated memory is a potential logic error.

Assignment Operator for Array Class

```
1     // overloaded assignment operator;
2     // const return avoids: (a1 = a2) = a3
3     const Array& Array::operator=(const Array& right) {
4          if (&right != this) { // avoid self-assignment
5               // for Arrays of different sizes, deallocate
    original
6               // left-side Array, then allocate new left-side
     Array
7               if (size != right.size) {
8                    delete[] ptr; // release space
9                    size = right.size; // resize this object
10                   ptr = new int[size]; // create space for
    Array copy
11              }
12              for (size_t i{0}; i < size; ++i) {
13                   ptr[i] = right.ptr[i]; // copy array into
    object
14              }
15         }
16         return *this; // enables x = y = z, for example
17    }
```

Commutative Operators

Suppose we want to overload a binary operator which operates on two operands with different class type, for enabling Commutative property for such an operator, we must overload it both as member function and non-member function. For example, assume s is an String and c is a char:

- s + c (operator+ is overloaded as the member function of String)
- c + s (operator+ is overloaded as non-member function)

Restrictions on Operator Overloading

- You cannot change the precedence and associativity of an operator by overloading.
- You cannot change the "arity" of an operator (i.e., the number of operands an operator takes).
- You cannot create new operators—only existing operators can be overloaded.
- You cannot change the meaning of how an operator works on objects of fundamental types.

Subscript Operator

Exercise: Try to overload subscript operator for Array class, pay attention that subscription on a const Array object can not be used as lvalue in any statement.