Object-Oriented Programming
(in C++)

Operator Overloading

Professor Yi-Ping You (游逸平)
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

Outline

- Introduction to Operator Overloading
- Restrictions on Operator Overloading
- Overloading Stream Insertion/Extraction Operators
- Case Study: `Array` Class
- Converting between Types
  - `explicit` Constructors
- Overloading Function Call Operator
- Overloading ++ and `--`
- Proxy Classes

Operator Overloading

- Recall: Function overloading
  - C++ allows the same name to be used for two or more different functions
  - Requirements for overloaded functions
    - They have different signatures
      - i.e., different ordered parameter types

- Operator overloading
  - C++ allows an operator to behave differently based on the types of operands used
  - Same requirements as overloaded functions

<<: A Built-in Overloaded Operator

- <<
  - Bitwise shift-left operator
    - Shifts the bits of the first operand left by the number of bits specified by the second operand
      - Fill from right with 0 bits
    - E.g., `3 << 1 = 6`
  - Stream insertion operator
    - E.g., `cout << "Hello" << endl;`
- Similarly, >> is also overloaded

Fundamentals of Operator Overloading

- Operators for primitive data types cannot be overloaded
- Operators can be overloaded only for user-defined data types (classes and `structs`)
  - Can use existing operators with user-defined types
  - C++ does not allow new operators to be created

Using Operators on Class Objects

- If you want to use an operator on an object, the operator *must* be overloaded, with three exceptions (i.e., the following three operators has default behaviors)
  - = (memberwise assignment operator)
  - & (address-of operator)
  - , (comma operator)
    - Evaluates the expression to its left then the expression to its right, and returns the value of the latter expression

How to Overload An Operator?

1. Write a non-`static` member function definition
   - The operator must be called on an object

   Or write a global function definition
2. Name the function with the reserved word `operator` followed by the symbol for the operator being overloaded
   - E.g., `operator+`

Performing Addition of Objects Using Functions

- Using a member function add

Time.h

```
class Time {
  Time add(Time&) const;
  ...
};
```

client.cpp

```
#include “Time.h”
int main() {
  Time t1, t2;
  Time t3 = t1.add(t2);

  return 0;
}
```

Performing Addition of Objects Using Operators (1/3)

- Using an overloaded operator +

Time.h

```
class Time {
  Time operator+(Time&) const;
  ...
};
```

A non-staic member function

client.cpp

```
#include “Time.h”
int main() {
  Time t1, t2;
  Time t3 = t1 + t2;

  return 0;
}
```

```
<leftObject>.operator<op>(<rightObject>)
= t1.operator+(t2)
= t1 + t2
```

Performing Addition of Objects Using Operators (2/3)

- Using an overloaded operator +

Time.h

```
class Time {
  friend Time operator+(Time&, Time&);
  ...
};
Time operator+(Time&, Time&);
```

Making the function a friend function of Time

A global function

client.cpp

```
#include “Time.h”
int main() {
  Time t1, t2;
  Time t3 = t1 + t2;

  return 0;
}
```

```
operator<op>(<leftObject>,<rightObject>)
= operator+(t1,t2)
= t1 + t2
```

Operators as Global, `friend` Functions

- Global functions are often made `friend`s for performance reasons
- It is possible to overload an operator as a global, non-`friend` function
  - If the function requires access to a class's `private` members, it needs to use `set` or `get` functions
  - The overhead of calling these functions could cause poor performance
  - An alternative solution is to inline the `set` and `get` functions

Arguments of Operator Functions

- Operator functions as member functions
  - Zero arguments for unary, one for binary
  - The object becomes the left-hand argument
- Operator functions as global functions
  - One argument for unary operators, two for binary operators
- E.g.

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

Member function

```
bool operator<(const String &, const String &);
```

Global function

Operators as Member or Global Functions?

- `()`, `[]`, `->`, or any of the assignment operators (e.g., `=`, `+=`, `-=`) must be overloaded as member functions
- If the leftmost (or only) operand must be an object of a different class or a fundamental type, this operator function must be implemented as a global function
  - When an operator function is implemented as a member function, the leftmost operand must be an object of the operator's class

Performing Addition of Objects Using Operators (3/3)

- Using an overloaded operator +

Time.h

```
class Time {
  friend Time operator+(int, Time&);
  ...
};
Time operator+(int, Time&);
```

Cannot be implemented as a member function

client.cpp

```
#include “Time.h”
int main() {
  int value;
  Time t1;
  Time t3 = value + t1;
  return 0;
}
```

Operators as Global Functions

- Another reason why one might choose a global function to overload an operator is to enable the operator to be commutative
  - For example, suppose we have an object `number` (of type `long int`) and an object `bigInteger1` (of class `HugeInteger`)
    - `bigInteger1 + number`
      ≡ `number + bigInteger1`

`HugeInteger` is a class in which integers may be arbitrarily large rather than being limited by the machine word size of the underlying hardware

HugeInteger Class

HugeInteger.h

```
class HugeInteger {
  friend HugeInteger operator+(HugeInteger, long int);
  friend HugeInteger operator+(long int, HugeInteger);
public:
  ...
private:
  char digit[41];
};
HugeInteger operator+(HugeInteger&, long int);
HugeInteger operator+(long int, HugeInteger&);
```

HugeInteger.cpp

```
...
HugeInteger operator+(long int a, HugeInteger& b) {
  return b + a;
}
...
```

Outline

- Introduction to Operator Overloading
- Restrictions on Operator Overloading
- Overloading Stream Insertion/Extraction Operators
- Case Study: `Array` Class
- Converting between Types
  - `explicit` Constructors
- Overloading Function Call Operator
- Overloading `++` and `--`

- Proxy Classes

Restrictions on Operator Overloading (1/2)

Operators that can be overloaded

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

Operators that cannot be overloaded

.	.*	::	?:

Restrictions on Operator Overloading (2/2)

- Cannot create new operators
- Cannot change
  - Precedence of operator (order of evaluation)
    - Use parentheses to force order of operators
  - Associativity (left-to-right or right-to-left)
  - Number of operands
    - E.g., & is unary, and it can only act on one operand
  - How operators act on built-in data types (i.e., cannot change integer addition)
- Operators must be overloaded explicitly
  - Overloading + and = does not overload +=

Outline

- Introduction to Operator Overloading
- Restrictions on Operator Overloading
- Overloading Stream Insertion/Extraction Operators
- Case Study: `Array` Class
- Converting between Types
  - `explicit` Constructors
- Overloading Function Call Operator
- Overloading `++` and `--`

- Proxy Classes

Recall: <iostream> Header File

- Declares basic services required for all stream-input/output operations
- Defines input/output classes
  - `istream`
  - `ostream`
- Defines standard stream objects
  - `cin` (an `istream` object)
    - Connected to the standard input device, usually the keyboard
  - `cout` (an `ostream` object)
    - Connected to the standard output device, usually the display screen

Stream Insertion/Extraction Operators

- Stream insertion operator
  - Left-shift operator(<<) is overloaded for stream output
  - Left operand of type `ostream &`
    - Such as **`cout`** object in **`cout << classObject`**
- Stream extraction operator
  - Right-shift operator(>>) is overloaded for stream input
  - Left operand of type `istream&`
    - Such as **`cin`** object in **`cin >> classObject`**
- Already overloaded to process each built-in type, including pointers and C-style `char*` strings

Overloading Stream Insertion/Extraction Operators

classA.h

```
class classA {
  ...
};
```

```
#include ”classA.h"
int main() {
  classA obj;
  cin >> obj;
  cout << obj;
  return 0;
}
```

- Must be overloaded as a global function
  - The left operand has type `istream&` or `ostream&`
    - Could be overloaded as a member function of `istream` or `ostream` class
    - However, you are not allowed to modify `iostream.h` and `iostream.cpp`

An Example of Overloading << and >>

- Suppose we have designed a class, called `PhoneNumber`
- We can overload << and >> to perform input and output for our own types
  - Inputting and outputting objects in the format "`(000) 000-0000`"
- We assume telephone numbers are input correctly

PhoneNumber.h

```
1  // Fig. 11.3: PhoneNumber.h
2  // PhoneNumber class definition
3  #ifndef PHONENUMBER_H
4  #define PHONENUMBER_H
5
6  #include <iostream>
7  #include <string>
8  using namespace std;
9
10 class PhoneNumber
11 {
12    friend ostream &operator<<( ostream &, const PhoneNumber & );
13    friend istream &operator>>( istream &, PhoneNumber & );
14 private:
15    string areaCode; // 3-digit area code
16    string exchange; // 3-digit exchange
17    string line; // 4-digit line
18 }; // end class PhoneNumber
19
20 #endif
```

Global, `friend` functions

PhoneNumber.cpp

```
4  #include <iomanip>
5  #include "PhoneNumber.h"
6  using namespace std;
7
8  // overloaded stream insertion operator; cannot be
9  // a member function if we would like to invoke it with
10 // cout << somePhoneNumber;
11 ostream &operator<<( ostream &output, const PhoneNumber &number )
12 {
13    output << "(" << number.areaCode << ") "
14       << number.exchange << "-" << number.line;
15    return output; // enables cout << a << b << c;
16 } // end function operator<<
17
18 // overloaded stream extraction operator; cannot be
19 // a member function if we would like to invoke it with
20 // cin >> somePhoneNumber;
21 istream &operator>>( istream &input, PhoneNumber &number )
22 {
23    input.ignore(); // skip (
24    input >> setw( 3 ) >> number.areaCode; // input area code
25    input.ignore( 2 ); // skip ) and space
26    input >> setw( 3 ) >> number.exchange; // input exchange
27    input.ignore(); // skip dash (-)
28    input >> setw( 4 ) >> number.line; // input line
29    return input; // enables cin >> a >> b >> c;
30 } // end function operator>>
```

Allows << to be cascaded

A Client of Class PhoneNumber

```
4  #include <iostream>
5  #include "PhoneNumber.h"
6  using namespace std;
7
8  int main()
9  {
10    PhoneNumber phone; // create object phone
11
12    cout << "Enter phone number in the form (123) 456-7890:" << endl;
13
14    // cin >> phone invokes operator>> by implicitly issuing
15    // the global function call operator>>( cin, phone )
16    cin >> phone;
17
18    cout << "The phone number entered was: ";
19
20    // cout << phone invokes operator<< by implicitly issuing
21    // the global function call operator<<( cout, phone )
22    cout << phone << endl;
23 } // end main
```

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

Outline

- Introduction to Operator Overloading
- Restrictions on Operator Overloading
- Overloading Stream Insertion/Extraction Operators
- Case Study: `Array` Class
- Converting between Types
  - `explicit` Constructors
- Overloading Function Call Operator
- Overloading ++ and `--`
- Proxy Classes

Recall: Pointer-Based vs Object-Based Arrays

- Pointer-based arrays have many problems
  - No boundary check
  - Subscript ranges are limited from `0` to `n - 1` (if the array size is n)
  - An entire array cannot be input or output at once
    - Each array element must be read or written individually (unless it is a null-terminated C string)
  - Two arrays cannot be meaningfully compared with equality or relational operators
  - An array does not know its size
  - One array cannot be assigned to another array with the assignment operator

`Array` Class

- You can develop a class, called `Array`, to address the problems that pointer-based arrays encounter

Array.h

```
3  #ifndef ARRAY_H
4  #define ARRAY_H
5
6  #include <iostream>
7  using namespace std;
8
9  class Array
10 {
11    friend ostream &operator<<( ostream &, const Array & );
12    friend istream &operator>>( istream &, Array & );
13 public:
14    Array( int = 10 ); // default constructor
15    Array( const Array & ); // copy constructor
16    ~Array(); // destructor
17    int getSize() const; // return size
18
19    const Array &operator=( const Array & ); // assignment operator
20    bool operator==( const Array & ) const; // equality operator
21
22    // inequality operator; returns opposite of == operator
23    bool operator!=( const Array &right ) const
24    {
25       return ! ( *this == right ); // invokes Array::operator==
26    } // end function operator!=
27
28    // subscript operator for non-const objects returns modifiable lvalue
29    int &operator[]( int );
30
31    // subscript operator for const objects returns rvalue
32    int operator[]( int ) const;
33 private:
34    int size; // pointer-based array size
35    int *ptr; // pointer to first element of pointer-based array
36 }; // end class Array
37
38 #endif
```

Array.cpp: Constructor

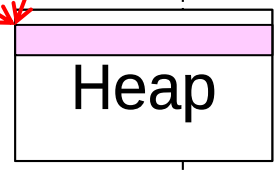
```
1  // Fig 11.7: Array.cpp
2  // Array class member- and friend-function definitions.
3  #include <iostream>
4  #include <iomanip>
5  #include <cstdlib> // exit function prototype
6  #include "Array.h" // Array class definition
7  using namespace std;
8
9  // default constructor for class Array (default size 10)
10 Array::Array( int arraySize )
11 {
12    size = ( arraySize > 0 ? arraySize : 10 ); // validate arraySize
13    ptr = new int[ size ]; // create space for pointer-based array
14
15    for ( int i = 0; i < size; i++ )
16       ptr[ i ] = 0; // set pointer-based array element
17 } // end Array default constructor
18
```

Array.cpp: Copy Constructor, Destructor, and getSize()

```
19 // copy constructor for class Array;
20 // must receive a reference to prevent infinite recursion
21 Array::Array( const Array &arrayToCopy )
22    : size( arrayToCopy.size )
23 {
24    ptr = new int[ size ]; // create sp
25
26    for ( int i = 0; i < size; i++ )
27       ptr[ i ] = arrayToCopy.ptr[ i ]; // copy into object
28 } // end Array copy constructor
29
30 // destructor for class Array
31 Array::~Array()
32 {
33    delete [] ptr; // release pointer-based array space
34 } // end destructor
35
36 // return number of elements of Array
37 int Array::getSize() const
38 {
39    return size; // number of elements in Array
40 } // end function getSize
41
```

Default copy constructor

```
size = arrayToCopy.size;
ptr = arrayToCopy.ptr;
```



Array.cpp: Copy Constructor, Destructor, and getSize()

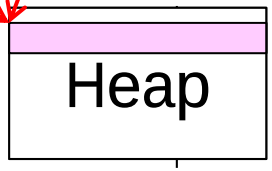
```
#include "Array.h"
int main() {
   Array array1;
   Array array2(array1);

   return 0;
}
```

```
t infinite recursion
y )
```

Default copy constructor

```
size = arrayToCopy.size;
ptr = arrayToCopy.ptr;
; // copy into object
```

array2 array1




```
30  // destructor for class Array
31  Array::~Array()
32  {
33     delete [] ptr; // release pointer-based array space
34  } // end destructor
35
36  // return number of elements of Array
37  int Array::getSize() const
38  {
39     return size; // number of elements in
40  } // end function getSize
41
```

A dangling pointer appears when array1 and array2 are destroyed

Array.cpp: = Operator

```
42  // overloaded assignment operator;
43  // const return avoids: ( a1 = a2 ) = a3
44  const Array &Array::operator=( const Array &right )
45  {
46     if ( &right != this ) // avoid self-assignment
47     {
48        // for Arrays of different sizes,
49        // left-side array, then allocate
50        if ( size != right.size )
51        {
52           delete [] ptr; // release space
53           size = right.size; // resize this object
54           ptr = new int[ size ]; // cre
55        } // end inner if
56
57        for ( int i = 0; i < size; i++ )
58           ptr[ i ] = right.ptr[ i ]; //
59     } // end outer if
60
61     return *this; // enables x = y = z,
62  } // end function operator=
63
```

a1=a2=a3 is allowed

Default assignment operator

```
size = right.size;
ptr = right.ptr;
```

```
#include "Array.h"
int main() {
   Array array1, array2;
   array2 = array1;
   // array2.operator=(array1);

   return 0;
}
```

Reading: http://www.parashift.com/c++-faq/self-assignment-why.html

Array.cpp: == Operator

```
64  // determine if two Arrays are equal and
65  // return true, otherwise return false
66  bool Array::operator==( const Array &right ) const
67  {
68     if ( size != right.size )
69        return false; // arrays of different number of elements
70
71     for ( int i = 0; i < size; i++ )
72        if ( ptr[ i ] != right.ptr[ i ] )
73           return false; // Array contents are not equal
74
75     return true; // Arrays are equal
76  } // end function operator==
```

```
#include ”Array.h"
int main() {
  Array array1, array2;
  while (array1 == array2); // while (array1.operator==(array2));
  return 0;
}
```

Array.cpp: [] Operator

```
78  // overloaded subscript operator for non-const Arrays;
79  // reference return creates a modifiable lvalue
80  int &Array::operator[]( int subscript )
81  {
82     // check for subscript out-of-range error
83     if ( subscript < 0 || subscript >= size )
84     {
85        cerr << "\nError: Subscript " << subscript
86           << " out of range" << endl;
87        exit( 1 ); // terminate program; subscript out of range
88     } // end if
89
90     return ptr[ subscript ]; // reference return
91  } // end function operator[]
```

```
#include ”Array.h"
int main() {
  Array array1;
  array1[5] = 1000; // array1.operator[](5) = 1000;
  return 0;
}
```

Array.cpp: [] and >> Operators

```
93  // overloaded subscript operator for const Arrays
94  // const reference return creates an rvalue
95  int Array::operator[]( int subscript ) const
96  {
97     // check for subscript out-of-range error
98     if ( subscript < 0 || subscript >= size )
99     {
100       cerr << "\nError: Subscript " << subscript
101          << " out of range" << endl;
102       exit( 1 ); // terminate program; sub
103    } // end if
104
105    return ptr[ subscript ]; // returns cop
106 } // end function operator[]
107
108 // overloaded input operator for class Arr
109 // inputs values for entire Array
110 istream &operator>>( istream &input, Array &a )
111 {
112    for ( int i = 0; i < a.size; i++ )
113       input >> a.ptr[ i ];
114
115    return input; // enables cin >> x >> y;
116 } // end function
```

```
#include "Array.h"
int main() {
  const Array array1;
  cout << array1[3];
  // cout << array1.operator[](3);

  return 0;
}
```

Array.cpp: << Operator

```
118 // overloaded output operator for class Array
119 ostream &operator<<( ostream &output, const Array &a )
120 {
121    int i;
122
123    // output private ptr-based array
124    for ( i = 0; i < a.size; i++ )
125    {
126       output << setw( 12 ) << a.ptr[ i ];
127
128       if ( ( i + 1 ) % 4 == 0 ) // 4 numbers pe
129          output << endl;
130    } // end for
131
132    if ( i % 4 != 0 ) // end last line of output
133       output << endl;
134
135    return output; // enables cout << x << y;
136 } // end function operator<<
```

```
#include "Array.h"
int main() {
  Array array1;
  cout << array1;

  return 0;
}
```

```
           0           0           0           0
           0           0           0           0
           0           0
```

Outline

- Introduction to Operator Overloading
- Restrictions on Operator Overloading
- Overloading Stream Insertion/Extraction Operators
- Case Study: `Array` Class
- Converting between Types
  - `explicit` Constructors
- Overloading Function Call Operator
- Overloading `++` and `--`
- Proxy Classes

Conversion Constructors and Operators

- The compiler knows how to perform certain conversions among fundamental types, but what about user-defined types?
- Conversion constructors (single-argument constructors)
  - Converting objects of other types into objects of user-defined types
- Conversion operators (also called cast operators)
  - Converting objects of user-defined types into objects of other types

An Example of Conversion Constructor (1/2)

String.h

```
#ifndef STRING_H
#define STRING_H

class String {
public:
   // conversion/default constructor
   String( const char * = "" );
   ...
private:
   int length; // string length, not counting null terminator
   char *sPtr; // pointer to start of pointer-based string
   void setString( const char * ); // utility function
};
#endif
```

```
#include "String.h"

int main() {
  String s;
  s = "hello”;
  // s = String("hello");
}
```

An Example of Conversion Constructor (2/2)

String.cpp

```
#include "String.h" // String class definition
#include <cstring>  // contains strlen and strcpy prototypes

// conversion/default constructor
String::String( const char *s )
   : length( ( s != 0 ) ? strlen( s ) : 0 ) {
  setString( s ); // call utility function
}

void String::setString( const char *string2 ) {
   sPtr = new char[ length + 1 ]; // allocate memory
   if ( string2 != 0 )
      strcpy( sPtr, string2 ); // copy literal to object
   else
      sPtr[ 0 ] = '\0'; // empty string
}
```

Conversion (Cast) Operators

```
classA::operator char *() const;
classA::operator int() const;
classA::operator OtherClass() const;
```

Converting a `classA` object into a `char *` object

- Non-static member functions
- Specify no return type
  - The return type is the type to which the object is being converted
- Suppose `s` is a class object
  - When the compiler sees the expressions

    ```
    static_cast<char *>(s)
    ```

  - The compiler generates the call

    ```
    s.operator char *()
    ```

An Example of Cast Operator

String.h

```
#ifndef STRING_H
#define STRING_H

class String {
public:
   // conversion (cast) operator
   operator char *() {
     sPtr[length] = '\0';
     return sPtr;
   }
   ...
private:
   int length; // string length
   char *sPtr; // pointer to start of pointer-based string
};
#endif
```

```
#include "String.h"

int main() {
  String s("hello");
  char *str;
  str = s;
  // str =
  //   s.operator char *();

  return 0;
}
```

Using Conversion Operators

- Suppose an object `s` is a user-defined `String` class
- We can just develop a cast operator for the `String` class to make the following statement compilable and executable, without overloading the stream insertion operator

```
cout << s;
```

Outline

- Introduction to Operator Overloading
- Restrictions on Operator Overloading
- Overloading Stream Insertion/Extraction Operators
- Case Study: `Array` Class
- Converting between Types
  - `explicit` Constructors
- Overloading Function Call Operator
- Overloading `++` and `--`

- Proxy Classes

Misusing a Constructor as a Conversion Constructor

Array.h

```
class Array {
public:
  Array(int = 10); // default constructor
  ...
private:
   int size; // pointer-based array size
   int *ptr; // pointer to start of pointer-based array
};
```

The compiler might treat it as a conversion constructor that converts an `int` to an `array` object

Array.cpp

```
 9  // default constructor for class Array (default size 10)
10  Array::Array( int arraySize )
11  {
12     size = ( arraySize > 0 ? arraySize : 10 ); // validate arraySize
13     ptr = new int[ size ]; // create space for pointer-based array
14
15     for ( int i = 0; i < size; i++ )
16        ptr[ i ] = 0; // set pointer-based array element
17  } // end Array default constructor
```

- This constructor can be misused by the compiler to perform an implicit conversion

Accidentally Using a Constructor as a Conversion Constructor

```
 3  #include <iostream>
 4  #include "Array.h"
 5  using namespace std;
 6
 7  void outputArray( const Array & ); // prototype
 8
 9  int main()
10  {
11     Array integers1( 7 ); // 7-element array
12     outputArray( integers1 ); // output Array integers1
13     outputArray( 3 ); // convert 3 to an Array and output Array's contents
14  }  // end main
15
16  // print Array contents
17  void outputArray( const Array &arrayToOutput )
18  {
19     cout << "The Array received has " << arrayToOutput.getSize()
20        << " elements. The contents are:\n" << arrayToOutput << endl;
21  } // end outputArray
```

- The compiler thought there is a conversion constructor that converts 3 into an `array` object

`explicit` Constructors

- C++ provides the keyword `explicit` to suppress implicit conversions via conversion constructors
  - A constructor that is declared `explicit` cannot be used in an implicit conversion

Array.h

```
class Array {
public:
  explicit Array(int = 10); // default constructor
  ...
private:
   int length; // string length
   char *sPtr; // pointer to start of pointer-based string
};
```

    - The compiler will then report a compilation error that the argument of `outputArray` is incorrect in the previous example

Outline

- Introduction to Operator Overloading
- Restrictions on Operator Overloading
- Overloading Stream Insertion/Extraction Operators
- Case Study: `Array` Class
- Converting between Types
  - `explicit` Constructors
- Overloading Function Call Operator
- Overloading ++ and `--`
- Proxy Classes

Overloading Function Call Operator (1/2)

String.h

```
#ifndef STRING_H
#define STRING_H

class String {
public:
   // return a substring
   String operator()( int, int = 0 ) const;
   ...
private:
   int length; // string length, not counting null terminator
   char *sPtr; // pointer to start of pointer-based string
};
#endif
```

Overloading Function Call Operator (2/2)

String.cpp

```
#include “String.h"
#include <cstring>

// return a substring beginning at index and of length subLength
String String::operator()( int index, int subLength ) const {
   // if index is out of range or substring length < 0,
   // return an empty String object
   if ( index < 0 || index >= length || subLength < 0 )
      return ""; // converted to a String object automatically

   // determine length of substring
   int len;
   if ( ( subLength == 0 ) || ( index + subLength > length ) )
      len = length – index;
   else
      len = subLength;

   // allocate temporary array for substring and
   // terminating null character
   char *tempPtr = new char[ len + 1 ];

   // copy substring into char array and terminate string
   strncpy( tempPtr, &sPtr[ index ], len );
   tempPtr[ len ] = '\0';

   // create temporary String object containing the substring
   String tempString( tempPtr );
   delete [] tempPtr; // delete temporary array

   return tempString; // return copy of the temporary String
} // end function operator()
```

```
#include "String.h"

int main() {
  String str(“Hello World”);
  cout << str(6, 5) << endl;
  return 0;
}
```

```
World
```

Outline

- Introduction to Operator Overloading
- Restrictions on Operator Overloading
- Overloading Stream Insertion/Extraction Operators
- Case Study: `Array` Class
- Converting between Types
  - `explicit` Constructors
- Overloading Function Call Operator
- Overloading ++ and `--`
- Proxy Classes

Overloading ++ and --

- Operators as member functions
  - Overloading the prefix ++ and --

```
classA &classA::operator++();
```

  - Overloading the postfix ++ and --

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

- Operators as global functions
  - Overloading the prefix ++ and --

```
classA &operator++(classA &);
```

  - Overloading the postfix ++ and --

```
classA operator++(classA &, int);
```

How the Compiler Deals with ++ and --?

```
#include "classA.h”

int main() {
  classA obj;

  ++obj; // compiler generates obj.operator++()
         // or operator++(obj)

  obj++; // compiler generates obj.operator++(0)
         // or operator++(obj, 0)

  return 0;
}
```

A “dummy value”

An Example of ++ and - -

Date.h

```
3  #ifndef DATE_H
4  #define DATE_H
5
6  #include <iostream>
7  using namespace std;
8
9  class Date
10 {
11    friend ostream &operator<<( ostream &, const Date & );
12 public:
13    Date( int m = 1, int d = 1, int y = 1900 ); // default constructor
14    void setDate( int, int, int ); // set month, day, year
15    Date &operator++(); // prefix increment operator
16    Date operator++( int ); // postfix increment operator
17    const Date &operator+=( int ); // add days, modify object
18    static bool leapYear( int ); // is date in a leap year?
19    bool endOfMonth( int ) const; // is date at the end of month?
20 private:
21    int month;
22    int day;
23    int year;
24
25    static const int days[]; // array of days per month
26    void helpIncrement(); // utility function for incrementing date
27 }; // end class Date
28
29 #endif
```

Partial Codes of Date.cpp

```
31  // overloaded prefix increment operator
32  Date &Date::operator++()
33  {
34     helpIncrement(); // increment date
35     return *this; // reference return to create an lvalue
36  } // end function operator++
37
38  // overloaded postfix increment operator; note that the
39  // dummy integer parameter does not have a parameter name
40  Date Date::operator++( int )
41  {
42     Date temp = *this; //
43     helpIncrement();
44
45     // return unincrement
46     return temp; // value
47     // end function operat
```

Returning the reference to `temp` is a logic error

```
77  // function to help increment the date
78  void Date::helpIncrement()
79  {
80     // day is not end of month
81     if ( !endOfMonth( day ) )
82        day++; // increment day
83     else
84        if ( month < 12 ) // day is end of month and month < 12
85        {
86           month++; // increment month
87           day = 1; // first day of new month
88        } // end if
89        else // last day of year
90        {
91           year++; // increment year
92           month = 1; // first month of new year
93           day = 1; // first day of new month
94        } // end else
95  } // end function helpIncrement
```

A Client of `Date` Class

```
 3  #include <iostream>
 4  #include "Date.h" // Date class definition
 5  using namespace std;
 6
 7  int main()
 8  {
 9     Date d1; // defaults to January 1, 1900
10     Date d2( 12, 27, 1992 ); // December 27, 1992
11     Date d3( 0, 99, 8045 ); // invalid date
12
13     cout << "d1 is " << d1 << "\nd2 is " << d2 << "\nd3 is " << d3;
14     cout << "\n\nd2 += 7 is " << ( d2 += 7 );
15
16     d3.setDate( 2, 28, 1992 );
17     cout << "\n\n  d3 is " << d3;
18     cout << "\n++d3 is " << ++d3 << " (leap year allows 29th)";
19
20     Date d4( 7, 13, 2002 );
21
22     cout << "\n\nTesting the prefix increment operator:\n"
23        << "  d4 is " << d4 << endl;
24     cout << "++d4 is " << ++d4 << endl;
25     cout << "  d4 is " << d4;
26
27     cout << "\n\nTesting the postfix increment operator:\n"
28        << "  d4 is " << d4 << endl;
29     cout << "d4++ is " << d4++ << endl;
30     cout << "  d4 is " << d4 << endl;
31  } // end main
```

Outline

- Introduction to Operator Overloading
- Restrictions on Operator Overloading
- Overloading Stream Insertion/Extraction Operators
- Case Study: `Array` Class
- Converting between Types
  - `explicit` Constructors
- Overloading Function Call Operator
- Overloading ++ and `--`

- Proxy Classes

Proxy Classes

- Good software practice: separate interface (.h) from implementation (and hide implementation details in .cpp)
- BUT header file does contain names of private variables
- Solution: proxy class
  - Have a wrapper class (Interface) that contains all functions of class you are wrapping (Implementation), and only 1 `private` data member—a pointer to the class you are wrapping
  - Each function in Interface class just calls `ImplementationObj->functionName();`
  - Client only sees wrapper class—the Interface .h file

An Example of Proxy Classes (Implementation)

Implementation.h

```
 4  class Implementation
 5  {
 6  public:
 7     // constructor
 8     Implementation( int v )
 9        : value( v ) // initialize value with v
10     {
11        // empty body
12     } // end constructor Implementation
13
14     // set value to v
15     void setValue( int v )
16     {
17        value = v; // should validate v
18     } // end function setValue
19
20     // return value
21     int getValue() const
22     {
23        return value;
24     } // end function getValue
25  private:
26     int value; // data that we would like to hide from the client
27  }; // end class Implementation
```

An Example of Proxy Classes (Interface)

Interface.h

```
1  // Fig. 11.17: Interface.h
2  // Proxy class Interface definition.
3  // Client sees this source code, but the source code does not reveal
4  // the data layout of class Implementation.
5
6  class Implementation; // forward class declaration required by line 17
7
8  class Interface
9  {
10 public:
11    Interface( int ); // constructor
12    void setValue( int ); // same public interface as
13    int getValue() const; // class Implementation has
14    ~Interface(); // destructor
15 private:
16    // requires previous forward declaration (line 6)
17    Implementation *ptr;
18 }; // end class Interface
```

An Example of Proxy Classes (Interface)

Interface.cpp

```
 4  #include "Interface.h" // Interface class defini
 5  #include "Implementation.h" // Implementation cl
 6
 7  // constructor
 8  Interface::Interface( int v )
 9     : ptr ( new Implementation( v ) ) // initialize ptr to point to
10  {                                    // a new Implementation object
11     // empty body
12  } // end Interface constructor
13
14  // call Implementation's setValue function
15  void Interface::setValue( int v )
16  {
17     ptr->setValue( v );
18  } // end function setValue
19
20  // call Implementation's getValue function
21  int Interface::getValue() const
22  {
23     return ptr->getValue();
24  } // end function getValue
25
26  // destructor
27  Interface::~Interface()
28  {
29     delete ptr;
30  } // end ~Interface destructor
```

The proxy class imposes an extra “layer” of function calls as “price to pay” for hiding the private data

An Example of Proxy Classes (Client)

```
 3  #include <iostream>
 4  #include "Interface.h" // Interface class definition
 5  using namespace std;
 6
 7  int main()
 8  {
 9     Interface i( 5 ); // create Interface object
10
11     cout << "Interface contains: " << i.getValue()
12        << " before setValue" << endl;
13
14     i.setValue( 10 );
15
16     cout << "Interface contains: " << i.getValue()
17        << " after setValue" << endl;
18  } // end main
```

```
Interface contains: 5 before setValue
Interface contains: 10 after setValue
```