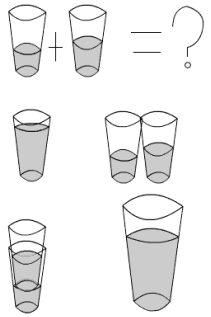
Lecture 8 - Operator Overloading

Meng-Hsun Tsai
CSIE, NCKU



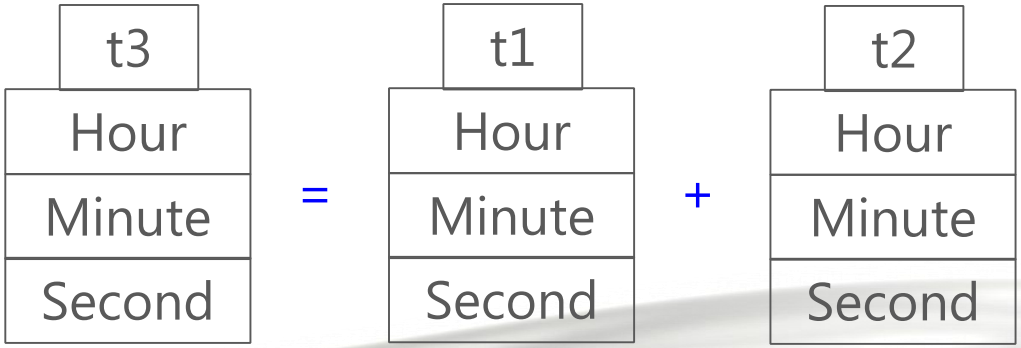
Introduction

- This lecture shows how to enable C++'s operators to work with objects—a process called operator overloading.
- One example of an overloaded operator built into C++ is <<, which is used both as the stream insertion operator and as the bitwise left-shift operator.
- C++ overloads the addition operator (+) and the subtraction operator (-). These operators perform differently, depending on their context in integer, floating-point and pointer arithmetic.
- C++ enables you to overload most operators—the compiler generates the appropriate code based on the context.

Fundamentals of Operator Overloading

- The fundamental types can be used with C++'s rich collection of operators.
- You can use operators with user-defined types as well.
- Although C++ does not allow new operators to be created, it does allow most existing operators to be overloaded so that, when they're used with objects, they have appropriate meaning to those objects.

```
Time t1, t2, t3;
t3 = t1 + t2;
```



Fundamentals of Operator Overloading (cont.)

- Use operator overloading when it makes a program clearer than accomplishing the same operations with function calls.

Which one is clearer ?
1. add(add(x,y),z)
2. x + y + z

- Overloaded operators should mimic the functionality of their built-in counterparts. For example, the + operator should be overloaded to perform addition, not subtraction.
- Avoid excessive or inconsistent use of operator overloading, as this can make a program cryptic and difficult to read.

Fundamentals of Operator Overloading (cont.)

- An operator is overloaded by writing a non-`static` member function definition or global function definition as you normally would, except that the function name now becomes 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 (`+`).
- When operators are overloaded as member functions, they must be non-`static`, because they must be called on an object of the class and operate on that object.

Fundamentals of Operator Overloading (cont.)

- To use an operator on class objects, that operator *must be overloaded—with three exceptions*.
- The assignment operator (=) may be used with every class to perform member-wise assignment of the class's data members.
  - Dangerous for classes with pointer members; we'll explicitly overload the assignment operator for such classes.

```
HugeInt i, j; i = j;
```

- The address (&) and comma (,) operators may also be used with objects of any class without overloading.

```
HugeInt i; HugeInt* ptr = &i;
```

  - The address operator returns a pointer to the object.
  - The comma operator evaluates the expression to its left then the expression to its right, and returns the value of the latter expression.

```
for (HugeInt i = 0, j=1; i<10; k=(i++, j++)) ;
```

Overloading Stream Insertion and Stream Extraction Operators

- You can input and output fundamental-type data using the stream extraction operator `>>` and the stream insertion operator `<<`.
- The C++ class libraries overload these operators to process each fundamental type, including pointers and C-style `char *` strings.
- You can also overload these operators to perform input and output for your own types.
- The next program overloads these operators to input and output `PhoneNumber`.

Storing a Phone Number

```
 1  #include <iostream>
 2  #include "PhoneNumber.h"
 3  using namespace std;
 4
 5  int main()
 6  {
 7     PhoneNumber phone;
 8     cout << "Enter your phone number as (XX) XXXXXXX: ";
 9     cin >> phone;
10     cout << phone << endl;
11     return 0;
12  }
```

```
Enter your phone number as (XX) XXXXXXX: (06) 2757575
(06) 275-7575
```

PhoneNumber.h and *PhoneNumber.cpp*

PhoneNumber.h

```
1  #ifndef PHONENUMBER_H
2  #define PHONENUMBER_H
3  #include <iostream>
4  #include <string>
5  using namespace std;
6  class PhoneNumber {
7    friend ostream &operator <<
     (ostream&, const PhoneNumber&);
8    friend istream &operator >>
     (istream &, PhoneNumber &);
9  private:
10   string areaCode;
11   string exchangeNum;
12   string serialNum;
13 };
14 #endif
```

```
9    cin >> phone;
10   cout << phone << endl;
```

PhoneNumber.cpp

```
1  #include <iomanip>
2  #include "PhoneNumber.h"
3  using namespace std;
4  ostream & operator <<(ostream &out,
       const PhoneNumber &num)
5  {
6    out << "(" << num.areaCode << ") "
         << num.exchangeNum << "-"
         << num.serialNum;
7    return out;
8  }
9  istream &operator >> (istream & in,
           PhoneNumber & num)
10 {
11   in.ignore(); // skip (
12   in >> setw(2) >> num.areaCode;
13   in.ignore(2); // skip ) and space
14   in >> setw(3) >> num.exchangeNum;
15   in >> setw(4) >> num.serialNum;
16   return in;
17 }
```

Stream Extraction Operator >>

- The stream extraction operator function `operator>>` takes `istream` reference `in` and `PhoneNumber` reference `num` as arguments and returns an `istream` reference.
- Operator function `operator>>` inputs phone numbers of the form
  - `(06) 2757575`
- When the compiler sees the expression
  - `cin >> phone`
- it generates the global function call
  - `operator>>( cin, phone );`
- When this call executes, reference parameter `in` becomes an alias for `cin` and reference parameter `num` becomes an alias for `phone`.

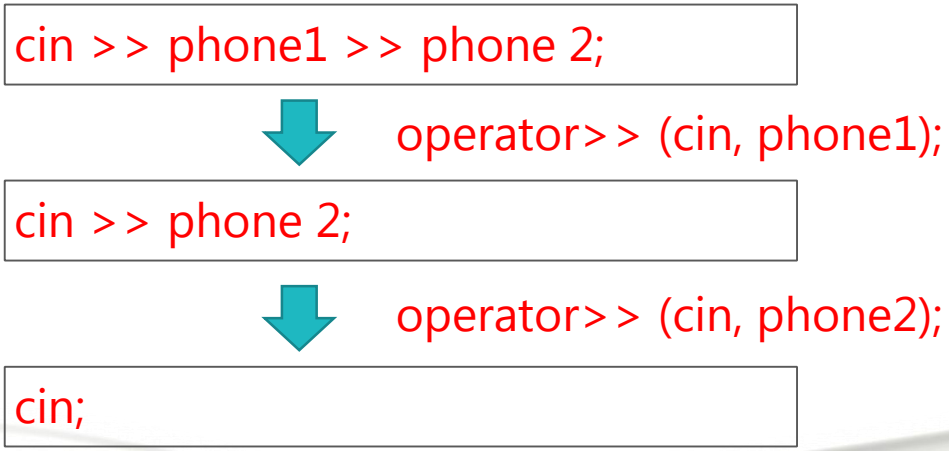
Stream Extraction Operator >> (cont.)

- The operator function reads as `string`s the three parts of the telephone number.
- Stream manipulator `setw` limits the number of characters read into each `string`.
- The parentheses and space characters are skipped by calling `istream` member function `ignore`, which discards the specified number of characters in the input stream (one character by default).

```
11    in.ignore(); // skip (
12    in >> setw(2) >> num.areaCode;
13    in.ignore(2); // skip ) and space
```

Stream Extraction Operator >> (cont.)

- Function `operator>>` returns `istream` reference `in` (i.e., `cin`).
- This enables input operations on `PhoneNumber` objects to be cascaded with input operations on other `PhoneNumber` objects or on objects of other data types.



Stream Insertion Operator <<

- The stream insertion operator function takes an `ostream` reference (`out`) and a `const PhoneNumber` reference (`num`) as arguments and returns an `ostream` reference.
- Function `operator<<` displays objects of type `PhoneNumber`.
- When the compiler sees the expression
  - `cout << phone`

  it generates the global function call
  - `operator<<( cout, phone );`
- Function `operator<<` displays the parts of the telephone number as `string`s, because they're stored as `string` objects.

Stream Insertion Operator << (cont.)

- The functions `operator>>` and `operator<<` are declared in `PhoneNumber` as global, `friend` functions
  - global functions because the object of class `PhoneNumber` is the operator's right operand.

```
 6  class PhoneNumber {
 7    friend ostream &operator <<
      (ostream&, const PhoneNumber&);
 8    friend istream &operator >>
      (istream &, PhoneNumber &);
    ...
13  };
```

```
 4  ostream & operator <<(ostream
&out,
          const PhoneNumber &num)
 5  {
 6    out << "(" << num.areaCode << ")
"
            << num.exchangeNum << "-"
            << num.serialNum;
    ...
 8  }
 9  istream &operator >> (istream & in,
              PhoneNumber & num)
10  {
    ...
15    in >> setw(4) >> num.serialNum;
```

Restrictions on Operator Overloading

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

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

Operators that can not be overloaded			
.	.*	::	?:

Restrictions on Operator Overloading (cont.)

- The precedence of an operator cannot be changed by overloading.
- The associativity of an operator (i.e., whether the operator is applied right-to-left or left-to-right) cannot be changed by overloading.
- It isn't possible to change the "arity" of an operator (i.e., the number of operands an operator takes): Overloaded unary operators remain unary operators; overloaded binary operators remain binary operators.
- Attempting to change the "arity" of an operator via operator overloading is a compilation error.

Restrictions on Operator Overloading (cont.)

- C++'s only ternary operator (`?:`) cannot be overloaded.
- Operators `&`, `*`, `+` and `-` all have both unary and binary versions; these unary and binary versions can each be overloaded.
- It isn't possible to create new operators; only existing operators can be overloaded.
- Attempting to create new operators via operator overloading is a syntax error.

Restrictions on Operator Overloading (cont.)

- The meaning of how an operator works on fundamental types cannot be changed by operator overloading.
  - You cannot, for example, change the meaning of how + adds two integers.
- Operator overloading works only with objects of user-defined types or with a mixture of an object of a user-defined type and an object of a fundamental type. That is, at least one argument of an operator function must be an object or reference of a user-defined type.

Restrictions on Operator Overloading (cont.)

- Overloading an assignment operator and an addition operator to allow statements like
  - `object2 = object2 + object1;`
- does not imply that the += operator is also overloaded to allow statements such as
  - `object2 += object1;`
- Such behavior can be achieved only by explicitly overloading operator += for that class.

Operator Functions as Class Members vs. Global Functions

- Operator functions can be member functions or global functions.
  - Global functions are often made `friend`s for performance reasons.
- Arguments for both operands of a binary operator must be explicitly listed in a global function call.
- When overloading `()`, `[]`, `->` or any of the assignment operators, the operator overloading function must be declared as a member function.
- For the other operators, the operator overloading functions can be class members or standalone functions.

Operator Functions as Class Members vs. Global Functions (cont.)

- Whether an operator function is implemented as a member function or as a global function, the operator is still used the same way in expressions.
- When an operator function is implemented as a member function, the leftmost (or only) operand must be an object (or a reference to an object) of the operator's class.
- If the left operand must be an object of a different class or a fundamental type, this operator function must be implemented as a global function (as we'll do with << and >>).

Operator Functions as Class Members vs. Global Functions (cont.)

- The overloaded stream insertion operator (`<<`) is used in an expression in which the left operand has type `ostream &`, as in `cout << classObject`.
- To use the operator in this manner where the right operand is an object of a user-defined class, it must be overloaded as a global function.
- Similarly, the overloaded stream extraction operator (`>>`) is used in an expression in which the left operand has type `istream &`, as in `cin >> classObject`, and the right operand is an object of a user-defined class.

Operator Functions as Class Members vs. Global Functions (cont.)

- You might choose a global function to overload an operator to enable the operator to be commutative, so an object of the class can appear on the right side of a binary operator.
- The `operator+` function, which deals with an object of the class on the left, can still be a member function.
- The global function simply swaps its arguments and calls the member function.

member function

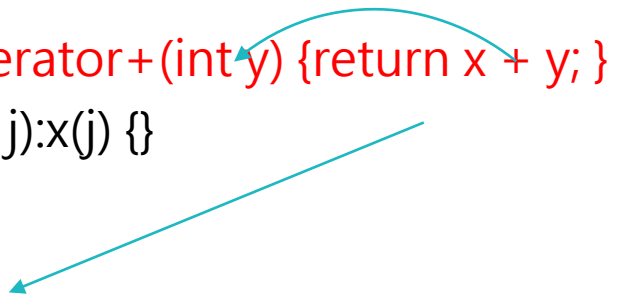
```
int Cls::operator+(int x)
{
    ...
}
```

global function

```
int operator+(int x, Cls obj)
{
    return obj + x;
}
```

Making + Commutative

```
1 class Cls {
2       friend int operator+(int b, Cls & anObj);
3 public:
4       int operator+(int y) {return x + y; }
5       Cls(int j):x(j) {}
6 private:
7       int x;
8 };
9 int operator+(int b, Cls & anObj)
10 {      return anObj + b;          }
```



```
11 int main()
12 {
13       int a = 5;
14       Cls obj(3);
15       cout << obj + a << endl;
16       cout << a + obj ;
17       return 0;
18 }
```

Output:
8
8

Dynamic Memory Management

- Sometimes it's useful to determine the size of an array dynamically at execution time and then create the array.
- C++ enables you to control the allocation and deallocation of memory in a program for objects and for arrays of any built-in or user-defined type.
  - Known as dynamic memory management; performed with new and delete.
- You can use the new operator to dynamically allocate (i.e., reserve) the exact amount of memory required to hold an object or array at execution time.
- The object or array is created in the free store (also called the heap)—a region of memory assigned to each program for storing dynamically allocated objects.

Dynamic Memory Management (cont.)

- Once memory is allocated in the free store, you can access it via the pointer that operator `new` returns.
- You can return memory to the free store by using the `delete` operator to deallocate it.
- The `new` operator allocates storage of the proper size for an object of the specified type, calls the constructor to initialize the object and returns a pointer to the type specified.
- If `new` is unable to find sufficient space in memory for the object, it indicates that an error occurred by "throwing an exception."

Dynamic Memory Management (cont.)

- To destroy a dynamically allocated object and free the space for the object, use the `delete` operator as follows:
  - `delete ptr;`
- This statement first calls the destructor for the object to which `ptr` points, then deallocates the memory associated with the object, returning the memory to the free store.
- Not releasing dynamically allocated memory when it's no longer needed can cause the system to run out of memory prematurely. This is sometimes called a "memory leak."

Sample Program with Memory Leak

```
 1 #include <iostream>
 2 using namespace std;
 3 int main()
 4 {
 5        int x, *y;
 6        for (x = 0; x < 10000000; x++)
 7             y = new int;
 8        cin >> x;
 9        return 0;
10 }
```

Output of top:

```
  PID USERNAME  THR PRI NICE   SIZE    RES STATE  TIME   WCPU   COMMAND
71685    Meng     1     5     0   157M   156M ttyin   0:02  42.78%  memory_leak
```

Operator *new*

- You can provide an initializer for a newly created fundamental-type variable, as in
  - `double *ptr = new double( 3.14159 );`
- The same syntax can be used to specify a comma-separated list of arguments to the constructor of an object.

Operator *new [ ]*

- You can also use the `new` operator to allocate arrays dynamically.
- For example, a 10-element integer array can be allocated and assigned to `gradesArray` as follows:
  - `int *gradesArray = new int[ 10 ];`
- A dynamically allocated array's size can be specified using any non-negative integral expression that can be evaluated at execution time.
- Also, when allocating an array of objects dynamically, you cannot pass arguments to each object's constructor—each object is initialized by its default constructor.

Operator *delete [ ]*

- To deallocate a dynamically allocated array, use the statement
  - `delete [] ptr;`
- If the pointer points to an array of objects, the statement first calls the destructor for every object in the array, then deallocates the memory.
- Using `delete` on a null pointer (i.e., a pointer with the value 0) has no effect.

Common Errors on Using Operator *delete* and *delete [ ]*

- Using `delete` instead of `delete []` for arrays of objects can lead to runtime logic errors. To ensure that every object in the array receives a destructor call, always delete memory allocated as an array with operator `delete []` .
- Similarly, always delete memory allocated as an individual element with operator `delete` — the result of deleting a single object with operator `delete []` is undefined.

runtime logic error

```
Cls *arr_p = new Cls[10];
delete arr_p;
```

undefined

```
Cls *obj_p = new Cls;
delete [] obj_p;
```

Error: Using Operator *delete* to Delete an Array

```
 1 #include <iostream>
 2 using namespace std;
 3 class Cls {
 4 public:
 5     ~Cls() { cout << "Destructor" << endl; }
 6 private:
 7     int x;
 8 };
 9 int main()
10 {
11     Cls * ptr = new Cls[10];
12     delete ptr;
13     return 0;
14 }
```

Output:
Destructor

Error: Using Operator *delete [ ]* to Delete an Object (cont.)

```
 1 #include <iostream>
 2 using namespace std;
 3 class Cls {
 4 public:
 5     ~Cls() { cout << "Destructor" << endl; }
 6 private:
 7     int x;
 8 };
 9 int main()
10 {
11     Cls * ptr = new Cls;
12     delete [] ptr;
13     return 0;
14 }
```

Output:

```
Destructor
Destructor
Destructor
Destructor
Destructor
Destructor
Destructor
Destructor
Destructor
Destructor
Destructor
Destructor
Destructor
Destructor
Aborted (core dumped)
```

Case Study: `Array` Class

- Pointer-based arrays have many problems, including:
  - A program can easily "walk off" either end of an array, because C++ does not check whether subscripts fall outside the range of an array.
  - One array cannot be assigned to another with the assignment operator.
  - When an array is passed to a function designed to handle arrays of any size, the array's size must be passed as an additional argument.
  - An entire array cannot be input or output at once.
  - Two arrays cannot be meaningfully compared with equality or relational operators.
  - Arrays of size *n* must number their elements 0, …, *n* – 1; alternate subscript ranges are not allowed.

Case Study: `Array` Class (cont.)

- C++ provides the means to implement more robust array capabilities via classes and operator overloading.
- In this example, we create a powerful `Array` class:
  - Performs range checking.
  - Allows one array object to be assigned to another with the assignment operator.
  - Objects know their own size (as member data).
  - Input or output entire arrays with the stream extraction and stream insertion operators, respectively.
  - Can compare `Array`s with the equality operators `==` and `!=`.
- C++ Standard Library class template `vector` provides many of these capabilities as well.

myArray.cpp

```
 1 #include <iostream>
 2 #include "Array.h"
 3 using namespace std;
 4 int main()
 5 {
 6    Array ints1(7);
 7    Array ints2;
 8
 9    cout << "Size of ints1 = " << ints1.getSize();
10    cout << "\ncontent = " << ints1;
11    cout << "Size of ints2 = " << ints2.getSize();
12    cout << "\ncontent = " << ints2;
13
14    cout << "\nEnter 17 numbers: " << endl;
15    cin >> ints1 >> ints2;
16
17    cout << "ints1: " << ints1 << endl;
18    cout << "ints2: " << ints2 << endl;
19
20    if (ints1 != ints2)
21       cout << "ints1 != ints2" << endl;
22
23    Array ints3(ints1);
24    cout << "ints3: " << ints3 << endl;
25
26    ints1 = ints2;
27    cout << "ints1: " << ints1 << endl;
28    cout << "ints2: " << ints2 << endl;
29
30    cout << ints1[5] << endl;
31    ints1[5] = 50;
32    cout << "ints1: " << ints1 << endl;
33
34    ints1[20] = 60;
35    return 0;
36 }
```

Array.h

```
 1 #ifndef ARRAY_H
 2 #define ARRAY_H
 3 #include <iostream>
 4 using namespace std;
 5 class Array{
 6    friend ostream & operator<<
          (ostream &, const Array &);
 7    friend istream & operator>>
          (istream &, Array &);
 8 public:
 9    Array (int = 10);
10    Array (const Array &);
11    ~Array();
12    int getSize() const;
13
14    const Array & operator = (const Array &);
15    bool operator== (const Array &) const;
16    bool operator!=(const Array & right) const
17    {
18       return !(*this == right);
19    }
20    int & operator[](int);
21    int operator[](int) const;
22 private:
23    int size;
24    int *ptr;
25 };
26 #endif
```

Array.cpp

```
 1 #include <iostream>
 2 #include <iomanip>
 3 #include <cstdlib>  // for exit()
 4 #include "Array.h"
 5 using namespace std;
 6
 7 Array::Array(int arrSize)
 8 {
 9    size = (arrSize>0?arrSize:10);
10    ptr = new int[size];
11    for(int i=0;i<size;++i)
12       ptr[i] = 0;
13 }
14 Array::Array(const Array & arrToCpy)
          :size(arrToCpy.size)
15 {
16    ptr = new int[size];
17    for(int i=0;i<size;++i)
18       ptr[i] = arrToCpy.ptr[i];
19 }
20 int Array::getSize() const
21 {
22    return size;
23 }
24 const Array & Array::operator=
          (const Array &right)
25 {
26    if(&right != this)
27    {
28       if(size != right.size)
29       {
30          delete [] ptr;
31          size = right.size;
32          ptr = new int[size];
33       }
34       for(int i=0;i<size;++i)
35          ptr[i] = right.ptr[i];
36    }
37    return *this;
38 }
```

Array.cpp (cont.)

```
39 bool Array::operator==
       (const Array &right) const
40 {
41    if(size != right.size)
42       return false;
43    for(int i=0;i<size;++i)
44       if(ptr[i] != right.ptr[i])
45          return false;
46    return true;
47 }
48 int & Array::operator[](int subscr)
49 {
50    if(subscr<0 || subscr>=size)
51    {
52       cerr << "Error: subscript "<< subscr
53          << " out of range" << endl;
54       exit(1);
55    }
56    return ptr[subscr];
57 }
```

```
58 int Array::operator[](int subscr) const
59 {
60    if(subscr<0 || subscr>=size)
61    {
62       cerr << "Error: subscript "<< subscr
63          << " out of range" << endl;
64       exit(1);
65    }
66    return ptr[subscr];
67 }
68 Array::~Array()
69 {
70    delete [] ptr;
71 }
72 istream & operator>>
       (istream &in, Array &a)
73 {
74    for(int i=0;i<a.size;++i)
75       in >> a.ptr[i];
76    return in;
77 }
```

Array.cpp (cont.)

```
78 ostream & operator<<
       (ostream &out, const Array &a)
79 {
80    int i;
81    for(int i=0;i<a.size;++i)
82       out << setw(3) << a.ptr[i];
83    out << endl;
84    return out;
85 }
```

```
Size of ints1 = 7
content =   0  0  0  0  0  0  0
Size of ints2 = 10
content =   0  0  0  0  0  0  0  0  0  0

Enter 17 numbers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
ints1:   1  2  3  4  5  6  7
ints2:   8  9 10 11 12 13 14 15 16 17

ints1 != ints2
ints3:   1  2  3  4  5  6  7

ints1:   8  9 10 11 12 13 14 15 16 17
ints2:   8  9 10 11 12 13 14 15 16 17
13
ints1:   8  9 10 11 12 50 14 15 16 17

Error: subscript 20 out of range
```

`Array`'s Default Constructor

- Line 9 of `Array.h` declares the default constructor for the class and specifies a default size of 10 elements.
- The default constructor validates and assigns the argument to data member `size`, uses `new` to obtain the memory for the internal pointer-based representation of this array and assigns the pointer returned by `new` to data member `ptr`.
- Then the constructor uses a `for` statement to set all the elements of the array to zero.

Array's Copy Constructor

- Line 10 of `Array.h` declares a copy constructor that initializes an `Array` by making a copy of an existing `Array` object.
- Such copying must be done carefully to avoid the pitfall of leaving both `Array` objects pointing to the same dynamically allocated memory.
- This is exactly the problem that would occur with default memberwise copying, if the compiler is allowed to define a default copy constructor for this class.
- Copy constructors are invoked whenever a copy of an object is needed, such as in passing an object by value to a function, returning an object by value from a function or initializing an object with a copy of another object of the same class.

Array's Copy Constructor (cont.)

- The `Array` copy constructor copies the elements of one `Array` into another.
- The copy constructor can also be invoked by writing as follows:
  - `Array ints3 = ints1;`
- The equal sign in the preceding statement is *not* the assignment operator.
- When an equal sign appears in the declaration of an object, it invokes a constructor for that object.
- This form can be used to pass only a single argument to a constructor.

Array's Copy Constructor (cont.)

- The argument to a copy constructor should be a `const` reference to allow a `const` object to be copied.
- A copy constructor must receive its argument by reference, not by value. Otherwise, the copy constructor call results in infinite recursion (a fatal logic error) because receiving an object by value requires the copy constructor to make a copy of the argument object.

```
10    Array (const Array &);
```

Error: Pass-by-value in Copy Constructor

```
 1 #include <iostream>
 2 using namespace std;
 3 class Cls {
 4 public:
 5       Cls():x(5) {}
 6       Cls(Cls obj) {   x = obj.x;    }
 7 private:
 8       int x;
 9 };
10 int main()
11 {
12       Cls obj1;
13       Cls obj2 (obj1);
14       return 0;
15 }
```


```
$ g++ -o copy_constructor copy_constructor.cpp
copy_constructor.cpp:6: error: invalid constructor; you
probably meant `Cls (const Cls&)'
copy_constructor.cpp:6: error: invalid member function
declaration
```

Array's Copy Constructor (cont.)

- The copy constructor for `Array` uses a member initializer to copy the `size` of the initializer `Array` into data member `size`, uses `new` to obtain the memory for the internal pointer-based representation of this `Array` and assigns the pointer returned by `new` to data member `ptr`.
- Then the copy constructor uses a `for` statement to copy all the elements of the initializer `Array` into the new `Array` object.
- An object of a class can look at the `private` data of any other object of that class (using a handle that indicates which object to access).

Dangling Pointer

- If the copy constructor simply copied the pointer in the source object to the target object's pointer, then both objects would point to the same dynamically allocated memory.
- The first destructor to execute would then delete the dynamically allocated memory, and the other object's ptr would be undefined, a situation called a dangling pointer — this would likely result in a serious run-time error when the pointer was used.

Array's operators >> and >>

- When the compiler sees an expression like `cout << arrayObject`, it invokes global function `operator<<` with the call
  - `operator<<( cout, arrayObject )`
- When the compiler sees an expression like `cin >> arrayObject`, it invokes global function `operator>>` with the call
  - `operator>>( cin, arrayObject )`
- These stream insertion and stream extraction operator functions cannot be member functions of class `Array`, because the `Array` object is always mentioned on the right side of the stream insertion operator and the stream extraction operator.

Array's Operator =

- Line 14 of `Array.h` declares the overloaded assignment operator function for the class.
- When the compiler sees the expression `ints1 = ints2` in line 26 of `myArray.cpp`, the compiler invokes member function `operator=` with the call
  - `ints1.operator=( ints2 )`
- Member function `operator=`'s implementation tests for self-assignment in which an `Array` object is being assigned to itself.
- When `this` is equal to the `right` operand's address, a self-assignment is being attempted, so the assignment is skipped.

Array's Operator = (cont.)

- `operator=` determines whether the sizes of the two arrays are identical; in that case, the original array of integers in the left-side `Array` object is not reallocated.
- Otherwise, `operator=` uses `delete` to release the memory, copies the `size` of the source array to the `size` of the target array, uses `new` to allocate memory for the target array and places the pointer returned by `new` into the array's `ptr` member.
- Regardless of whether this is a self-assignment, the member function returns the current object (i.e., `*this`) as a constant reference; this enables cascaded `Array` assignments such as `x = y = z`, but prevents ones like `(x = y) = z` because `z` cannot be assigned to the `const Array&` reference that is returned by `(x = y)`.

Notice on Class with Dynamically Allocated Memory

- A copy constructor, a destructor and an overloaded assignment operator are usually provided as a group for any class that uses dynamically allocated memory.
- It's possible to prevent class objects from being copied; to do this, simply make both the overloaded assignment operator and the copy constructor of that class `private`.

```
 5 class Array{
     ...
 8 private:
10    Array (const Array &);
     ...
14    const Array & operator = (const Array
&);
```

```
23    Array ints3(ints1);
 ...
26    ints1 = ints2;
```

Array's Operator ==

- When the compiler sees the expression `ints1 == ints2`, the compiler invokes member function `operator==` with the call
  - `ints1.operator==( ints2 )`
- Member function `operator==` immediately returns `false` if the `size` members of the arrays are not equal.
- Otherwise, `operator==` compares each pair of elements.
- If they're all equal, the function returns `true`.

Array's Operator !=

- Member function `operator!=` uses the overloaded `operator==` function to determine whether one `Array` is equal to another, then returns the opposite of that result.
- Writing `operator!=` in this manner enables you to reuse `operator==`, which reduces the amount of code that must be written in the class.

```
 5 class Array{
    ...
15    bool operator== (const Array &) const;
16    bool operator!=(const Array & right)
const
17    {
18       return !(*this == right);
19    }
```

Array's operator []

- The array subscript operator `[]` is not restricted for use only with arrays; it also can be used, for example, to select elements from other kinds of container classes, such as linked lists, strings and dictionaries.
- Also, when `operator[]` functions are defined, subscripts no longer have to be integers—characters, strings, floats or even objects of user-defined classes also could be used.
- Each `Array` object consists of a `size` member indicating the number of elements in the `Array` and an `int` pointer—`ptr`—that points to the dynamically allocated pointer-based array of integers managed by the `Array` object.

Array's operator [] (cont.)

- When the compiler sees the expression `ints1[5]`, it invokes the appropriate overloaded `operator[]` member function by generating the call
  - `ints1.operator[]( 5 )`
- The compiler creates a call to the `const` version of `operator[]` when the subscript operator is used on a `const Array` object.
- If the subscript is in range, the non-`const` version of `operator[]` returns the appropriate array element as a reference so that it may be used as a modifiable *lvalue*.
- If the subscript is in range, the `const` version of `operator[]` returns a copy of the appropriate element of the array.

Overloading ++ and −−

- The prefix and postfix versions of the increment and decrement operators can all be overloaded.
- To overload the increment operator to allow both prefix and postfix increment usage, each overloaded operator function must have a distinct signature, so that the compiler will be able to determine which version of ++ is intended.

Preincrementing ++

- Suppose, for example, that we want to add 1 to the day in `Date` object `d1`.
- When the compiler sees the preincrementing expression `++d1`, the compiler generates the member function call
  - `d1.operator++()`
- The prototype for this operator function would be
  - `Date &operator++();`

Preincrementing ++ (cont.)

- If the prefix increment operator is implemented as a global function, then, when the compiler sees the expression `++d1`, the compiler generates the function call
  - `operator++( d1 )`
- The prototype for this operator function would be declared in the `Date` class as
  - `Date &operator++( Date & );`
- 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.

Postincrementing ++

- The convention that has been adopted in C++ is that, when the compiler sees the postincrementing expression `d1++`, it generates the member function call
  - `d1.operator++( 0 )`
- The prototype for this function is
  - `Date operator++( int )`
- The argument `0` is strictly a "dummy value" that enables the compiler to distinguish between the prefix and postfix increment operator functions.
- The same syntax is used to differentiate between the prefix and postfix decrement operator functions.

Postincrementing ++ (cont.)

- If the postfix increment is implemented as a global function, then, when the compiler sees the expression `d1++`, the compiler generates the function call
  - `operator++( d1, 0 )`
- The prototype for this function would be
  - `Date operator++( Date &, int );`
- Once again, the `0` argument is used by the compiler to distinguish between the prefix and postfix increment operators implemented as global functions.

Postincrementing ++ (cont.)

- The postfix increment operator returns `Date` objects by value, whereas the prefix increment operator returns `Date` objects by reference, because the postfix increment operator typically returns a temporary object that contains the original value of the object before the increment occurred.
- The extra object that is created by the postfix increment (or decrement) operator can result in a significant performance problem—especially when the operator is used in a loop. For this reason, you should use the postfix increment (or decrement) operator only when the logic of the program requires postincrementing (or postdecrementing).

Date.h

```
 1 #ifndef DATE_H
 2 #define DATE_H
 3 #include <iostream>
 4 using namespace std;
 5 class Date {
 6    friend ostream & operator<<(ostream&,const Date&);
 7 public:
 8    Date(int m=1, int d=1, int y=1900);
 9    void setDate(int, int , int);
10    Date &operator++();     // for ++d
11    Date operator++(int);    // for d++
12    const Date &operator+=(int);  // for d += n
13    static bool leapYear(int);
14    bool endOfMonth(int) const;
15 private:
16    int month, day, year;
17    static const int days[];
18    void helpIncrement();
19 };
20 #endif
```

Date.cpp

```
 1 #include <iostream>
 2 #include <string>
 3 #include "Date.h"
 4 using namespace std;
 5 const int Date::days[] =
 6   {0,31,28,31,30,31,30,31,31,30,31,30,31};
 7 Date::Date(int m, int d, int y)
 8 {
 9    setDate(m,d,y);
10 }
11 void Date::setDate(int mm, int dd, int yy)
12 {
13    month = (mm>=1 && mm<=12)?mm:1;
14    year = (yy>=1900 && yy<=2100)?yy:1900;
15    if(month == 2 && leapYear(year))
16       day = (dd>=1 && dd<=29)?dd:1;
17    else
18       day = (dd>=1 && dd<=days[month])?dd:1;
19 }
```

```
20 Date& Date::operator++()
21 {
22    helpIncrement();
23    return *this;
24 }
25 Date Date::operator++(int)
26 {
27    Date temp = *this;
28    helpIncrement();
29    return temp;
30 }
31 const Date& Date::operator+=
       (int addDays)
32 {
33    for(int i=0;i<addDays;++i)
34       helpIncrement();
35    return *this;
36 }
```

Date.cpp (cont.)

```
37 bool Date::leapYear(int y)
38 {
39     if(y % 400 == 0 ||
40        (y % 100 != 0 && y % 4 == 0))
41         return true;
42     else
43         return false;
44 }
45 bool Date::endOfMonth(int d) const
46 {
47     if(month == 2 && leapYear(year))
48         return d == 29;
49     else
50         return d == days[month];
51 }
52 void Date::helpIncrement()
53 {
54     if(!endOfMonth(day))
55         ++day;
56     else
57         if(month<12)
58         {
59             ++month;
60             day=1;
61         }
62         else
63         {
64             ++year;
65             month=1;
66             day=1;
67         }
68 }
```

Date.cpp (cont.)

```
69 ostream& operator<<(ostream & out,const Date & d)
70 {
71     static string monthName[13] =
72         {"", "January", "February",
73          "March", "April", "May",
74          "June", "July", "August",
75          "September", "October",
76          "November", "December" };
77     out << monthName[d.month] << ' ' << d.day << ", " << d.year;
78     return out;
79 }
```

myDate.cpp

```
 1 #include <iostream>
 2 #include "Date.h"
 3 using namespace std;
 4 int main()
 5 {
 6     Date d1;
 7     Date d2(12,27,1992);
 8     Date d3(0,99,8045);
 9     cout << "d1: "<< d1 << endl;
10     cout << "d2: "<< d2 << endl;
11     cout << "d3: "<< d3 << endl;
12     cout << "d2 += 7: "<< (d2+=7) << endl;
13
14     d3.setDate(2,28,1992);
15     cout << "d3: "<< d3 << endl;
16     cout << "++d3: "<< ++d3 << endl;
17
18     Date d4(7,13,2002);
19     cout << "d4: "<< d4 << endl;
20     cout << "++d4: "<< ++d4 << endl;
21     cout << "d4: "<< d4 << endl;
22
23     cout << "d4: "<< d4 << endl;
24     cout << "d4++: "<< d4++ << endl;
25     cout << "d4: "<< d4 << endl;
26
27     return 0;
28 }
```

```
d1: January 1, 1900
d2: December 27, 1992
d3: January 1, 1900
d2 += 7: January 3, 1993
d3: February 28, 1992
++d3: February 29, 1992
d4: July 13, 2002
++d4: July 14, 2002
d4: July 14, 2002
d4: July 14, 2002
d4++: July 14, 2002
d4: July 15, 2002
```

Case Study: A `Date` Class

- The overloaded prefix increment operator returns a reference to the current `Date` object (i.e., the one that was just incremented).
- This occurs because the current object, `*this`, is returned as a `Date &`.
  - Enables a preincremented `Date` object to be used as an *lvalue*, which is how the built-in prefix increment operator works for fundamental types.

Case Study: A `Date` Class (cont.)

- To emulate the effect of the postincrement, we must return an unincremented copy of the `Date` object.
- On entry to `operator++`, we save the current object (`*this`) in `temp`.
- Next, we call `helpIncrement` to increment the current `Date` object.
- Then, line 29 returns the unincremented copy of the object previously stored in `temp`.
- This function cannot return a reference to the local `Date` object `temp`, because a local variable is destroyed when the function in which it's declared exits.

Standard Library Class `string`

- The following program demonstrates many of class `string`'s overloaded operators, it's conversion constructor for C strings and several other useful member functions, including `empty`, `substr` and `at`.
- Function `empty` determines whether a `string` is empty, function `substr` returns a `string` that represents a portion of an existing `string` and function `at` returns the character at a specific index in a `string` (after checking that the index is in range).

Using Operators on *string* Objects

```
 1 #include <iostream>
 2 #include <string>
 3 using namespace std;
 4 int main()
 5 {
 6    string s1("happy");
 7    string s2("birthday");
 8    string s3;
 9    cout << "s1: " << s1;
10    cout << "\ns2: " << s2;
11    cout << "\ns3: " << s3;
12    cout << "\ns2 == s1?: " << ((s2==s1)?"true":"false");
13    cout << "\ns2 != s1?: " << ((s2!=s1)?"true":"false");
14    cout << "\ns2 > s1?: " << ((s2>s1)?"true":"false");
15    cout << "\ns2 < s1?: " << ((s2<s1)?"true":"false");
16    cout << "\ns2 >= s1?: " << ((s2>=s1)?"true":"false");
17    cout << "\ns2 <= s1?: " << ((s2<=s1)?"true":"false");
18
19    cout <<   "\nIs s3 empty?: "   << ((s3.empty())?"true":"false");
20    s3=s1;
21    cout << "\ns3: " << s3;
22
23    s1+=s2;
24    cout << "\ns1: " << s1;
25    s1+= "to you";
26    cout << "\ns1: " << s1;
27    cout << "\ns1.substr(0,14): " << s1.substr(0,14);
28    cout << "\ns1.substr(15): " << s1.substr(15);
29
```

Using Operators on *string* Objects (cont.)

```
30    string s4(s1);
31    cout << "\ns4: " << s4;
32    s4=s4;
33    cout << "\ns4: " << s4;
34
35    s1[0]='H';
36    s1[6]='B';
37    cout << "\ns1: " << s1;
38    s1.at(30)='D';
39    return 0;
40 }
```

```
s1: happy
s2: birthday
s3:
s2 == s1?: false
s2 != s1?: true
s2 > s1?: false
s2 < s1?: true
s2 >= s1?: false
s2 <= s1?: true
Is s3 empty?: true
s3: happy
s1: happy birthday
s1: happy birthdayto you
s1.substr(0,14): happy birthday
s1.substr(15): o you
s4: happy birthdayto you
s4: happy birthdayto you
terminate called after throwing an instance of
'std::out_of_range'
  what():  basic_string::at
s1: Happy Birthdayto youAbort (core dumped)
```

Using Operators on *string* Objects (cont.)

- Class `string`'s overloaded equality and relational operators perform lexicographical comparisons using the numerical values of the characters (ASCII code) in each `string`.
- Class `string` provides member function empty to determine whether a `string` is empty.
- Class `string`'s overloaded += operator performs string concatenation.
- Class `string`'s member function substr returns a portion of a string as a `string` object.
  - When the second argument is not specified, `substr` returns the remainder of the `string` on which it's called.

Using Operators on *string* Objects (cont.)

- Class `string`'s overloaded `[]` operator can create *lvalues that enable new characters to replace existing characters in a* `string`.
  - *Class string's overloaded [] operator does not perform any bounds checking*.
- Class `string` provides bounds checking in its member function `at`, which "throws an exception" if its argument is an invalid subscript.
  - By default, this causes a C++ program to terminate and display a system-specific error message.
  - Function `at` returns the character at the specified location as a modifiable *lvalue* or an unmodifiable *lvalue* (i.e., a `const` reference), depending on the context in which the call appears.