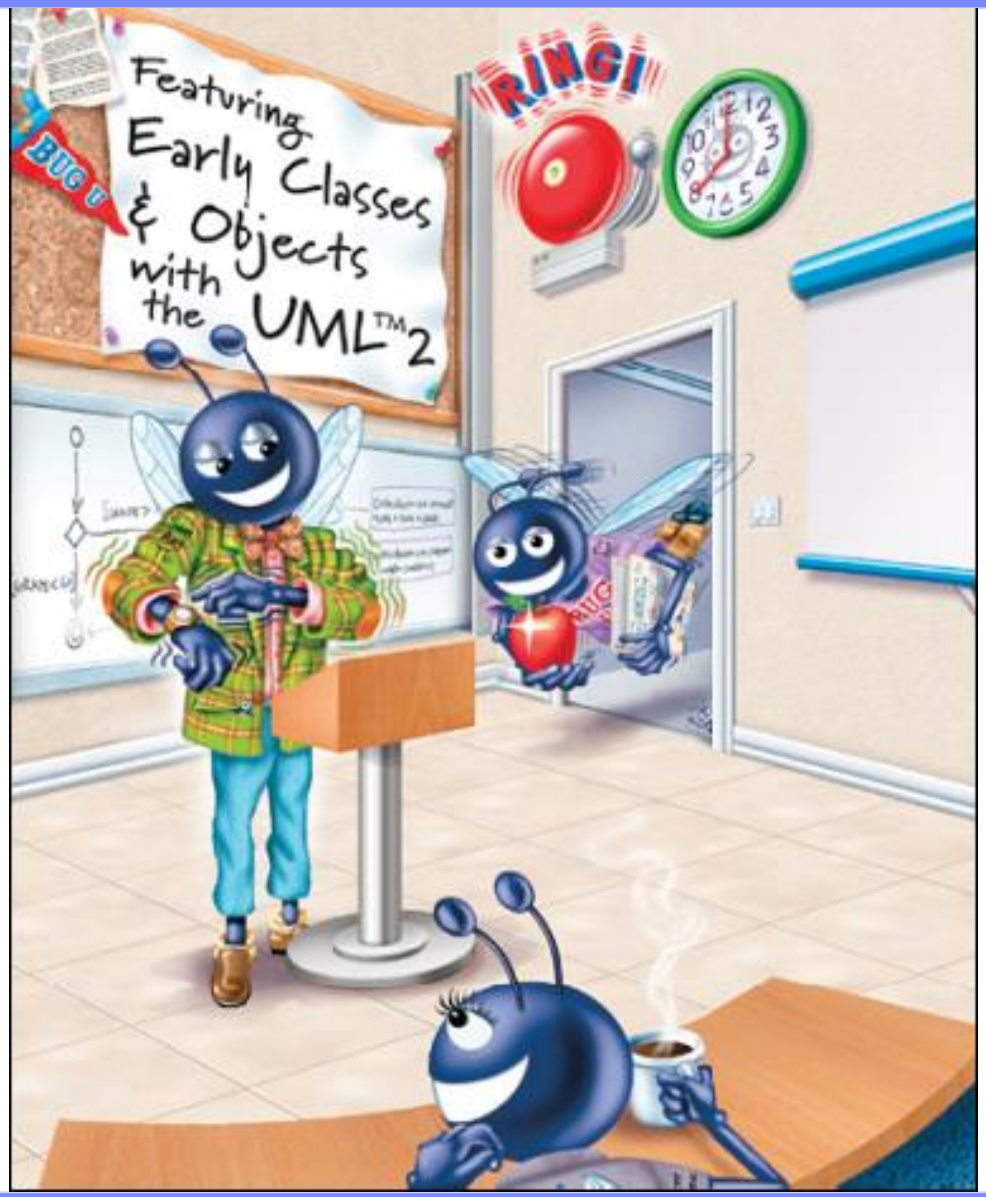


Lecture 8:
运算符重载

第十一讲 运算符重载

学习目标：

- 什么是运算符重载
- 如何进行运算符重载
- 类型转换
- 重载 ++ 和 -- 运算符
- 实例：Array类；String类；Date类

0.问题引入

- 引入运算符重载的原因
  - 在实际中有如下表达式：
    - ✓ int a=1,b=2; float c=1.1,d=2.4;
    - ✓ int e=a+b;
    - ✓ float f=c+d;
    - ✓ float g=f+e;
  - 为什么同一个运算符“+”可以用于完成不同类型的数据的加法运算呢?

  因为：C++语言针对基本数据类型已经对某些运算符做了适当的重载。

续

- 但C++语言提供的**基本数据类型**终究是有限的，我们在解决多种多样的实际问题时，往往需要使用许多的自定义数据类型。
  - ✓ 如在解决科学与工程计算问题时，往往要使用**复数**、**矩阵**等。
  - ✓ 该如何处理这些数据类型之间的运算?

  答案是：**运算符重载**。

- 例：一个定义复数的类：

```
class complex
{    public:
     complex(double r=0.0，double i=0.0)      //构造函数
          {real = r,  imag = i;}               //定义复数的实部与虚部
          void display( )；                    //显示复数的值
     private:
          double real；                        //定义实部
          double imag；                        //定义虚部
};
```

- 实例化对象：

  complex a(10，20), b(5，8)； 即：

  a = 10+20i ; b = 5 + 8i

- 如果需要对a和b进行加法运算，该如何实现呢?
  - 使用“+”运算符，计算表达式“a+b”？
  - 编译的时候会出错，因为编译器不知道该如何完成这个加法。
- 因此：需要自己编写程序来说明“+”在作用于complex类对象时，该实现什么样的功能（显然它的处理方法与基本数据类型的+不完全相同），这就是运算符重载。

运算符重载

使同一个运算符作用与不同类型的数据时导致不同的行为的这种机制称为运算符重载。

运算符重载机制

C++编译器在对运算符进行编译处理时，将一个运算符编译成如下形式：

一元运算符：	@obj	编译成 →	operator @ (obj)
二元运算符：	obj1@obj2	编译成 →	operator @ (obj1,obj2)

其中，关键字operator 加上运算符名的函数称为运算符函数

由于C++中有前置++、--，后置++、--，为了区分它们，C++将后置++、--编译成：

后置 --：	obj--	编译成 →	operator -- (obj，0)
后置++：	obj++	编译成 →	operator ++ (obj，0)

2 Fundamentals of Operator Overloading

- 运算符重载
  - 如果将已有运算符用于用户自定义数据类型时
  - 为类创建特殊的函数----运算符函数

    关键字 operator 后跟要重载的运算符，例如：

    operator+ (参数表)

    表示重载 “ + ” 运算符

运算符重载是对已有的运算符赋予更多的含义，同一个运算符作用于不同类型的数据导致不同类型的行为。

运算符重载的实质就是函数重载。在实现过程中，首先把指定的运算表达式转化为对运算符函数的调用，运算对象转化为运算符函数的实参，然后根据实参的类型来确定需要调用的函数，这个过程是在编译过程中完成的。

运算符重载规则

1.可重载的运算符

C++中的运算符除以下五个运算符之外，其余全部可以被重载。

.	成员选择运算符
.*	成员指针运算符
::	作用域分辨符
?:	三目选择运算符
sizeof	计算数据大小运算符

2.运算符的重载规则

(1) 重载后运算符的优先级与结合性不会改变。
(2) 不能改变原运算符操作数的个数。
(3) 不能重载C++中没有的运算符。
(4) 不能改变运算符的原有语义。

Restrictions on Operator Overloading

能够被重载的运算符							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

2 Fundamentals of Operator Overloading

- 在类的对象上使用运算符
  - 必须对该运算符进行重载（三个例外）
    - ✓赋值运算符 (=)：按成员进行赋值
    - ✓取地址运算符 (&)：返回地址
    - ✓逗号运算符 (,)对象的
      - ◈ 计算逗号左侧表达式然后计算逗号右侧表达式

2 Fundamentals of Operator Overloading

- 运算符重载提供简明的表达

object2 = object1.add(object2); --函数调用方法

vs.

object2 = object2 + object1; --运算符重载方法

哪种方法更直观？显然第二种方法更符合我们常规的使用习惯。

但第二种方法的“+”终将要转化为一个函数调用！

3 Restrictions on Operator Overloading

常见编程错误：误以为重载了某个运算符（如：“+”）可以自动重载相关的运算符（如：“+=”），或重载了“==”就自动重载了“!=”，这将导致错误。运算符只能显式重载（不存在隐式重载）。

4 Operator Functions as Class Members vs. Global Members

- 运算符函数可作为类成员函数，也可作为全局函数
  - 作为类成员函数
    - ✓ 最左侧的操作数应为该类对象
    - ✓ 利用 this 关键字隐式获得最左侧操作数
    - ✓ 运算符 ()，[]，-> 或任何赋值运算符，必须重载为类的成员函数
    - ✓ 当为以下情况将被调用
      - ◈ 二元运算符的左侧操作数为该类对象
      - ◈ 一元运算符的操作数为该类对象

4 Operator Functions as Class Members vs. Global Members

- 作为全局函数
  - ✓需要所有操作数作为参数
  - ✓可以设置为友元来访问 private 或 protected 数据

① 运算符重载为类的成员函数的一般语法形式为：

<函数类型> operator <运算符>（形参表）{ 函数体；}

② 运算符重载为全局函数（类的友元函数）的一般语法形式为：

friend <函数类型> operator <运算符>（形参表）{ 函数体；}

<函数类型> 指出了运算符重载函数的返回类型

operator是定义运算符重载函数的关键字，

<运算符> 给出了要重载的运算符名称。

（形参表）中给出了重载运算符所需要的参数及参数的类型。

4 Operator Functions as Class Members vs. Global Members

性能提示：可以把一个运算符作为一个非成员、非友元函数重载。但是，这样的运算符函数访问类的private和protected数据时必须使用类的public接口中提供的set或get函数，调用这些函数所涉及的开销会降低性能。可以将这些函数内联以提高性能。

4 Operator Functions as Class Members vs. Global Members

- 可交换的运算符
  - 有时需要 + 为可交换的
    - ✓ 即要求 “a + b” 和 “b + a” 均能工作
  - 例如：HugeIntClass + long int
    - ✓ 运算符函数可能为 HugeIntClass 成员函数
    - ✓ 如果需要运算符 ” +“ 成为可交换的，则要求是全局运算符函数才能在两种情况下运行

5 Overloading Stream Insertion and Stream Extraction Operators

- << 和 >> 运算符
  - C++已经将它重载来处理内部数据类型，能输出字符、数字、字符串等
  - 如果需要用来处理用户自定义类型(如数组)，就需要进行重载
    - ✓利用全局，友元函数进行重载

5 Overloading Stream Insertion and Stream Extraction Operators

- 重载 << 与 >> 运算符
  - 重载 << 时左侧操作数为 ostream &
    - ✓例如： cout << classObject
  - 重载 >> 时左侧操作数为 istream &
    - ✓例如： cin >> classObject
  - 需要作为全局函数进行重载

- 重载输出运算符"<<"和输入运算符">>"

```
friend ostream & operator<<(ostream &out, const 用户类型 &obj)
{
    out << obj.item1;
    out << obj.item2;
    ……
    return out;
}
```

```
friend istream & operator>>(istream &in, 用户类型 &obj)
{
    in >> obj.item1;
    in >> obj.item2;
    ……
    return in;
}
```

5 Overloading Stream Insertion and Stream Extraction Operators

- 例：
  - Class PhoneNumber
    - ✓电话号码类
  - 要求自动按指定格式打印
    - ✓(123) 456-7890

```
1  // Fig. 11.3: PhoneNumber.h
2  // PhoneNumber class definition
3  #ifndef PHONENUMBER_H
4  #define PHONENUMBER_H
5
6  #include <iostream>
7  using std::ostream;
8  using std::istream;
9
10 #include <string>
11 using std::string;
12
13 class PhoneNumber
14 {
15    friend ostream &operator<<( ostream &, const PhoneNumber & );
16    friend istream &operator>>( istream &, PhoneNumber & );
17 private:
18    string areaCode; // 3-digit area code
19    string exchange; // 3-digit exchange
20    string line; // 4-digit line
21 }; // end class PhoneNumber
22
23 #endif
```

Notice function prototypes for overloaded operators **>>** and **<<** (must be global, **friend** functions)

```
1  // Fig. 11.4: PhoneNumber.cpp
2  // Overloaded stream insertion and stream extraction operators
3  // for class PhoneNumber.
4  #include <iomanip>
5  using std::setw;
6
7  #include "PhoneNumber.h"
8
9  // overloaded stream insertion operator; cannot be
10 // a member function if we would like to invoke it with
11 // cout << somePhoneNumber;
12 ostream &operator<<( ostream &output, const PhoneNumber &number )
13 {
14    output << "(" << number.areaCode << ") "
15       << number.exchange << "-" << number.line;
16    return output; // enables cout << a << b << c;
17 } // end function operator<<
```

Allows **`cout << phone;`** to be interpreted as: **`operator<<(cout, phone);`**

Display formatted phone number

```
18
19 // overloaded stream extraction operator; cannot be
20 // a member function if we would like to invoke it with
21 // cin >> somePhoneNumber;
22 istream &operator>>( istream &input, PhoneNumber &number )
23 {
24    input.ignore(); // skip (
25    input >> setw( 3 ) >> number.areaCode; // input area code
26    input.ignore( 2 ); // skip ) and space
27    input >> setw( 3 ) >> number.exchange; // input exchange
28    input.ignore(); // skip dash (-)
29    input >> setw( 4 ) >> number.line; // input line
30    return input; // enables cin >> a >> b >> c;
31 } // end function operator>>
```

ignore skips specified number of characters from input (1 by default)

Input each portion of phone number separately

```
1  // Fig. 11.5: fig11_05.cpp
2  // Demonstrating class PhoneNumber's overloaded stream insertion
3  // and stream extraction operators.
4  #include <iostream>
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include "PhoneNumber.h"
10
11 int main()
12 {
13    PhoneNumber phone; // create object phone
14
15    cout << "Enter phone number in the form (123) 456-7890:" << endl;
16
17    // cin >> phone invokes operator>> by implicitly issuing
18    // the global function call operator>>( cin, phone )
19    cin >> phone;
20
21    cout << "The phone number entered was: ";
22
23    // cout << phone invokes operator<< by implicitly issuing
24    // the global function call operator<<( cout, phone )
25    cout << phone << endl;
26    return 0;
27 } // end main
```

Testing overloaded **>>** and **<<** operators to input and output a **PhoneNumber** object

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

6 Overloading Unary Operators

- 重载一元运算符
  - 可以重载为没有参数的非静态成员函数
  - 可以重载为带一个参数的全局函数
    - ✓参数必须为该类对象或引用
  - 注意：静态成员函数只能访问静态数据成员

提示：所谓一元运算符是指具有一个操作数的运算符，如！或 -等。

如果重载为（某对象的）成员函数，则操作数为该对象；

如果重载为全局函数，则操作数由该运算符实现函数的参数决定。

6 Overloading Unary Operators

- 例：重载 “!” 来测试空字符串
  - 重载为非静态成员函数
    - ✓ class String
      {
      public:
        bool operator!() const;
        …
      };
    - ✓ !s 将会调用: s.operator!() -- 没有参数。其运算对象(操作数)为对象 s 本身。s为类的一个对象

6 Overloading Unary Operators

- 重载为全局函数，需要一个参数
  - ✓ bool operator!(const String &)
  - ✓ s! 调用形式： operator!(s) -- 1个参数

7 Overloading Binary Operators

- 重载二元运算符
  - 重载为带一个参数的非静态成员函数
  - 重载为带两个参数的全局函数
    - ✓一个参数必须为类的对象或引用

提示：所谓二元运算符是指具有二个操作数的运算符，如 + 。

如果重载为（某对象的）成员函数，则一个操作数为该对象本身，另一个操作数为函数所带；

如果重载为全局函数，则操作数由该运算符实现函数的二个参数决定。

7 Overloading Binary Operators

- 例：重载“+=”
  - 如果为非静态成员函数
    - ✓ class String
      {
      public:
        const String & operator+=(const String &);
        …
      };
    - ✓ y += z 将调用： y.operator+=(z) -- 1个参数。

    其中 y 为类的对象。

7 Overloading Binary Operators

- 如果为全局函数
  - ✓ const String &operator+=(String &, const String &);
  - ✓ y += z 变换成 operator+=(y, z) -- 2个参数

8 Case Study: Array Class

- C++ 中基于指针的数组
  - 无边界检查
  - 不能利用 == 进行比较
  - 不能进行数组间赋值
  - 如果数组作为参数传递给函数，一般必须将数组的大小作为参数同时传递

8 Case Study: Array Class

- 后面例子中的 Array 类实现了
  - 边界检查
  - 数组赋值
  - 数组知道自己的大小
  - 利用 << 和 >> 进行数组的输入输出
  - 利用 == 和 != 进行数组比较

8 Case Study: Array Class

- 拷贝构造函数
  - 用另一对象来初始化当前对象
    - ✓ 值传递 （返回对象或将对象作为参数）
    - ✓ 如：
      - ◈ Array newArray(oldArray); 或
        Array newArray = oldArray
      - ◈ 用oldArray的值初始化
        newArray,oldArray要先调用构造函数进行实例化

8 Case Study: Array Class

- 拷贝构造函数
  - `Array( const Array & );`
    - ✓ 参数必须为对象的引用，否则为值传递，将会继续调用拷贝构造函数，变为无限循环

```
1  // Fig. 11.6: Array.h
2  // Array class for storing arrays of integers.
3  #ifndef ARRAY_H
4  #define ARRAY_H
5
6  #include <iostream>
7  using std::ostream;
8  using std::istream;
9
10 class Array
11 {
12    friend ostream &operator<<( ostream &, const Array & );
13    friend istream &operator>>( istream &, Array & );
14 public:
15    Array( int = 10 ); // default constructor
16    Array( const Array & ); // copy constructor
17    ~Array(); // destructor
18    int getSize() const; // return size
19
20    const Array &operator=( const Array & ); // assignment operator
21    bool operator==( const Array & ) const; // equality operator
22
23    // inequality operator; returns opposite of == operator
24    bool operator!=( const Array &right ) const
25    {
26       return ! ( *this == right ); // invokes Array::operator==
27    } // end function operator!=
```

重载<< 和 >>并定义为本类的友元。以便使用：cin >> Array对象.

Operator>>(cin, arrayObject)

Prototype for copy constructor

该处的!= 实现实际上重载了上面定义的==，以减少程序的复杂性

```
28
29    // subscript operator for non-const objects returns modifiable lvalue
30    int &operator[]( int );
31
32    // subscript operator for const objects returns rvalue
33    int operator[]( int ) const;
34 private:
35    int size; // pointer-based array size
36    int *ptr; // pointer to first element of pointer-based ar
37 }; // end class Array
38
39 #endif
```

分别用于**非const对象**与**const对象**的下标取值。

非const版本返回引用，可作为左值被修改；

const版本返回值的副本，只能是右值，不能被修改

```
1  // Fig 11.7: Array.cpp
2  // Member-function definitions for class Array
3  #include <iostream>
4  using std::cerr;
5  using std::cout;
6  using std::cin;
7  using std::endl;
8
9  #include <iomanip>
10 using std::setw;
11
12 #include <cstdlib> // exit function prototype
13 using std::exit;
14
15 #include "Array.h" // Array class definition
16
17 // default constructor for class Array (default size 10)
18 Array::Array( int arraySize )
19 {
20    size = ( arraySize > 0 ? arraySize : 10 ); // validate arraySize
21    ptr = new int[ size ]; // create space for pointer-based array
22
23    for ( int i = 0; i < size; i++ )
24       ptr[ i ] = 0; // set pointer-based array element
25 } // end Array default constructor
```

```
26
27 // copy constructor for class Array;
28 // must receive a reference to prevent infinite recursion
29 Array::Array( const Array &arrayToCopy )
30    : size( arrayToCopy.size )          //初始化器给 size 赋值
31 {
32    ptr = new int[ size ]; // create space for pointer-based array
33
34    for ( int i = 0; i < size; i++ )
35       ptr[ i ] = arrayToCopy.ptr[ i ]; // copy into object
36 } // end Array copy constructor
37
38 // destructor for class Array
39 Array::~Array()
40 {
41    delete [] ptr; // release pointer-based array space
42 } // end destructor
43
44 // return number of elements of Array
45 int Array::getSize() const
46 {
47    return size; // number of elements in Array
48 } // end function getSize
```

We must declare a new integer array so the objects do not point to the same memory.

否则通过引用联结的两个对象数组指向同一数据块，这会引起严重错误！

```
49
50 // overloaded assignment operator;
51 // const return avoids: ( a1 = a2 ) = a3   将参数指向的数组元素赋给本对象指向数组空间
52 const Array &Array::operator=( const Array &right )
53 {
54    if ( &right != this ) // avoid self-assignment
55    {
56       // for Arrays of different sizes, deallocate original
57       // left-side array, then allocate new left-side array
58       if ( size != right.size )
59       {
60          delete [] ptr; // release space
61          size = right.size; // resize this object
62          ptr = new int[ size ]; // create space for array copy
63       } // end inner if
64
65       for ( int i = 0; i < size; i++ )
66          ptr[ i ] = right.ptr[ i ]; // copy array into object
67    } // end outer if
68
69    return *this; // enables x = y = z, for example
70 } // end function operator=
```

避免**自赋值**的情况

This would be dangerous if **this** is the same **Array** as **right**

```
71
72 // determine if two Arrays are equal and
73 // return true, otherwise return false
74 bool Array::operator==( const Array &right ) const
75 {
76    if ( size != right.size )
77       return false; // arrays of different number of elements
78
79    for ( int i = 0; i < size; i++ )
80       if ( ptr[ i ] != right.ptr[ i ] )
81          return false; // Array contents are not equal
82
83    return true; // Arrays are equal
84 } // end function operator==
85
86 // overloaded subscript operator for non-const Arrays;
87 // reference return creates a modifiable lvalue
88 int &Array::operator[]( int subscript )
89 {
90    // check for subscript out-of-range error
91    if ( subscript < 0 || subscript >= size )
92    {
93       cerr << "\nError: Subscript " << subscript
94          << " out of range" << endl;
95       exit( 1 ); // terminate program; subscript out of range
96    } // end if
97
98    return ptr[ subscript ]; // reference return
99 } // end function operator[]
```

非const版本，可以作为左值被修改

integers1[5] 实际调用函数 integers1.operator[](5)

```
100
101// overloaded subscript operator for const Arrays
102// const reference return creates an rvalue
103int Array::operator[]( int subscript ) const
104{
105   // check for subscript out-of-range error
106   if ( subscript < 0 || subscript >= size )
107   {
108      cerr << "\nError: Subscript " << subscript
109         << " out of range" << endl;
110      exit( 1 ); // terminate program; subscript out of range
111   } // end if
112
113   return ptr[ subscript ]; // returns copy of this element
114} // end function operator[]
115
116// overloaded input operator for class Array;
117// inputs values for entire Array
118istream &operator>>( istream &input, Array &a )
119{
120   for ( int i = 0; i < a.size; i++ )
121      input >> a.ptr[ i ];
122
123   return input; // enables cin >> x >> y;
124} // end function
```

Const版本，只能作为右值，不能被修改

```
125
126// overloaded output operator for class Array
127ostream &operator<<( ostream &output, const Array &a )
128{
129   int i;
130
131   // output private ptr-based array
132   for ( i = 0; i < a.size; i++ )
133   {
134      output << setw( 12 ) << a.ptr[ i ];
135
136      if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
137         output << endl;
138   } // end for
139
140   if ( i % 4 != 0 ) // end last line of output
141      output << endl;
142
143   return output; // enables cout << x << y;
144} // end function operator<<
```

```
1  // Fig. 11.8: fig11_08.cpp
2  // Array class test program.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  #include "Array.h"
9
10 int main()
11 {
12    Array integers1( 7 ); // seven-element Array
13    Array integers2; // 10-element Array by default
14
15    // print integers1 size and contents
16    cout << "Size of Array integers1 is "
17       << integers1.getSize()
18       << "\nArray after initialization:\n" << integers1;
19
20    // print integers2 size and contents
21    cout << "\nSize of Array integers2 is "
22       << integers2.getSize()
23       << "\nArray after initialization:\n" << integers2;
24
25    // input and print integers1 and integers2
26    cout << "\nEnter 17 integers:" << endl;
27    cin >> integers1 >> integers2;
```

Retrieve number of elements in **Array**

Use overloaded **>>** operator to input

```
28
29    cout << "\nAfter input, the Arrays contain:\n"
30       << "integers1:\n" << integers1
31       << "integers2:\n" << integers2;
32
33    // use overloaded inequality (!=) operator
34    cout << "\nEvaluating: integers1 != integers2" << endl;
35
36    if ( integers1 != integers2 )
37       cout << "integers1 and integers2 are not equal" << endl;
38
39    // create Array integers3 using integers1 as an
40    // initializer; print size and contents
41    Array integers3( integers1 ); // invokes copy constructor
42
43    cout << "\nSize of Array integers3 is "
44       << integers3.getSize()
45       << "\nArray after initialization:\n" << integers3;
46
47    // use overloaded assignment (=) operator
48    cout << "\nAssigning integers2 to integers1:" << endl;
49    integers1 = integers2; // note target Array is smaller
50
51    cout << "integers1:\n" << integers1
52       << "integers2:\n" << integers2;
53
54    // use overloaded equality (==) operator
55    cout << "\nEvaluating: integers1 == integers2" << endl;
```

Use overloaded **<<** operator to output

Use overloaded **!=** operator to test for inequality

Use copy constructor

Use overloaded **=** operator to assign

```
56
57    if ( integers1 == integers2 )
58       cout << "integers1 and integers2 are equal" << endl;
59
60    // use overloaded subscript operator to create rvalue
61    cout << "\nintegers1[5] is " << integers1[ 5 ];
62
63    // use overloaded subscript operator to create lvalue
64    cout << "\n\nAssigning 1000 to integers1[5]" << endl;
65    integers1[ 5 ] = 1000;
66    cout << "integers1:\n" << integers1;
67
68    // attempt to use out-of-range subscript
69    cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
70    integers1[ 15 ] = 1000; // ERROR: out of range
71    return 0;
72 } // end main
```

Use overloaded == operator to test for equality

Use overloaded [] operator to access individual integers, with range-checking

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

Size of Array integers2 is 10
Array after initialization:
           0           0           0           0
           0           0           0           0
           0           0

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

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

Evaluating: integers1 != integers2
integers1 and integers2 are not equal
```

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

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

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13

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

Attempt to assign 1000 to integers1[15]

Error: Subscript 15 out of range
```

8 Case Study: Array Class

常见编程错误：拷贝构造函数应使用按引用传递接受参数，而不是按值传递。

常见编程错误：如果拷贝构造函数只把源对象的指针复制给目标对象的指针，这两个对象将指向同一块动态分配的内存块，执行析构函数时将释放该内存块，结果导致另一个对象的指针悬空（这种指针被称为“危险指针”），如果这时去使用该指针，会引起运行时错误（例如程序过早地终止等）。

8 Case Study: Array Class

软件工程知识：通常要把构造函数、析构函数、重载的赋值运算符以及拷贝构造函数一起提供给使用动态内存分配的类。

软件工程知识：当类的对象包含指向动态分配的内存的指针时，如果不为其提供重载的赋值运算符和复制的构造函数会造成逻辑错误。

软件工程知识：防止一个类对象赋值给另一个类对象是可以实现的，具体做法是将赋值操作声明为该对象的private成员。

9 Converting between Types

- 类型转换
  - 例如：将 int 转换为 floats
  - 用户自定义类型之间的转换

- 对于基本数据类型
  - 编译器知道如何转换类型
  - 程序员也可以用强制类型转换运算符实现内部类型之间的强制转换
- 对于用户自定义类型之间，用户自定义类型和内部类型之间
  - 程序员必须明确地指明如何转换——转换构造函数，也就是使用单个参数的构造函数，这种函数仅仅把其他类型(包括内部类型)的对象转换为某个特定类的对象。
  - 强制类型转换运算符
- 转换运算符(也称为强制类型转换运算符)可以把一种类的对象转换为其他类的对象或内部类型的对象。
- 必须是一个非static成员函数，而不能是友元函数。

- 函数原型：　A::operator char *() const;
  - 声明了一个重载的强制类型转换运算符函数
  - 用户自定义类型A的对象→临时的char*类型的对象
  - const :没有修改原始对象
  - 重载的强制类型转换运算符函数不能指定返回类型
    - ✓ 返回类型是要转换后的对象类型

- 标准C++中有四个类型转换符
  - static_cast
  - dynamic_cast
  - reinterpret_cast
  - const_cast

- static_cast < type-id > (expression)

  该运算符把expression转换为type-id类型，但没有运行时类型检查来保证转换的安全性。它主要有如下几种用法：

  - 用于类层次结构中基类和子类之间指针或引用的转换。进行上行转换（把子类的指针或引用转换成基类表示）是安全的；进行下行转换（把基类指针或引用转换成子 类表示）时，由于没有动态类型检查，所以是不安全的。
  - 用于基本数据类型之间的转换，如把int转换成char，把int转换成enum。这种转换的安全性也要开发人员来保证。
  - 把空指针转换成目标类型的空指针。
  - 把任何类型的表达式转换成void类型。

- static_cast<char *>(s)
  - 编译器会产生函数调用s.operator char*(),
  - 操作数s是调用成员函数operator char*的类对象s。

  - A::operator int()const；
  - A::operator otherClass()const；

- 强制类型转换运算符和转换构造函数一个很好的特点就是：当需要的时候，编译器可以为建立一个临时对象而自动地调用这些函数
- 如果用户自定义的类String的某个对象s出现在程序中需要使用char*类型的对象的位置上，例如：

```
cout << s;
```

编译器调用重载的强制类型转换运算符函数operator char*将对象转换为char*类型，并在表达式中使用转换后的char*类型的结果。String类提供该转换运算符后，不需要重载流插入运算符用cout输出String。

9 Converting between Types

- 类型转换后无需重载一些运算符
  - 假设类 String 可以被转换为 char *
  - cout << s; //该处 s is a String
    - ✓编译器隐式的将 s 转换为 char * 进行输出
    - ✓无需重载 <<

10 Case Study: String Class

- class String
  - 类似于标准库中的 string 类
- 转换构造函数
  - 任何单参数的构造函数
    - ✓例如：String s1("happy");
      - ◈ 从 char * 创建 String

10 Case Study: String Class

- 重载函数调用运算符
  - 函数可以带有任意长度，复杂的参数列表

```
1  // Fig. 11.9: String.h
2  // String class definition.
3  #ifndef STRING_H
4  #define STRING_H
5
6  #include <iostream>
7  using std::ostream;
8  using std::istream;
9
10 class String
11 {
12    friend ostream &operator<<( ostream &, const String & );
13    friend istream &operator>>( istream &, String & );
14 public:
15    String( const char * = "" ); // conversion/default constructor
16    String( const String & ); // copy constructor
17    ~String(); // destructor
18
19    const String &operator=( const String & ); // assignment ope
20    const String &operator+=( const String & ); // concatenation
21
22    bool operator!() const; // 检测串是否为空?
23    bool operator==( const String & ) const; // test s1 == s2
24    bool operator<( const String & ) const; // test s1 < s2
25
```

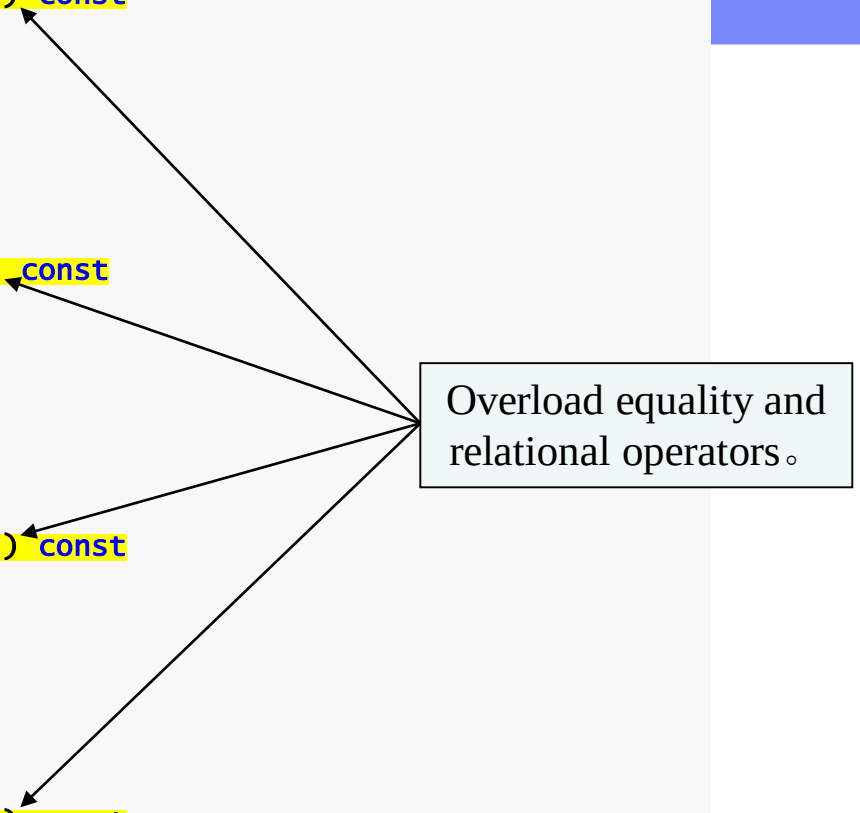
Conversion constructor to make a **String** from a **char ***

s1 += s2 will be interpreted as **s1.operator+=(s2)**

如果s2是C风格的”xxx”怎办?

编译器只能将 **char *** 转换为**string**类型

```
26    // test s1 != s2
27    bool operator!=( const String &right ) const
28    {
29       return !( *this == right );
30    } // end function operator!=
31
32    // test s1 > s2
33    bool operator>( const String &right ) const
34    {
35       return right < *this;
36    } // end function operator>
37
38    // test s1 <= s2
39    bool operator<=( const String &right ) const
40    {
41       return !( right < *this );
42    } // end function operator <=
43
44    // test s1 >= s2
45    bool operator>=( const String &right ) const
46    {
47       return !( *this < right );
48    } // end function operator>=
```



```
49
50    char &operator[]( int ); // subscript operator (modifiable lvalue)
51    char operator[]( int ) const; // subscript operator (rvalue)
52    String operator()( int, int = 0 ) const; // return a substring
53    int getLength() const; // return string length
54 private:
55    int length; // string length (not counting null terminator)
56    char *sPtr; // pointer to start of pointer-based string
57
58    void setString( const char * ); // utility function
59 }; // end class String
60
61 #endif
```

const 和 non-**const**两个版本的重载

重载 **()** 运算符返回规定的子串

```
1  // Fig. 11.10: String.cpp
2  // Member-function definitions for class String.
3  #include <iostream>
4  using std::cerr;
5  using std::cout;
6  using std::endl;
7
8  #include <iomanip>
9  using std::setw;
10
11 #include <cstring> // strcpy and strcat prototypes
12 using std::strcmp;
13 using std::strcpy;
14 using std::strcat;
15
16 #include <cstdlib> // exit prototype
17 using std::exit;
18
19 #include "String.h" // String class definition
20
21 // conversion (and default) constructor converts char * to String
22 String::String( const char *s )
23    : length( ( s != 0 ) ? strlen( s ) : 0 )
24 {
25    cout << "Conversion (and default) constructor: " << s << endl;
26    setString( s ); // call utility function
27 } // end String conversion constructor
28
```

函数实现在159行

```
29 // copy constructor
30 String::String( const String &copy )
31    : length( copy.length )
32 {
33    cout << "Copy constructor: " << copy.sPtr << endl;
34    setString( copy.sPtr ); // call utility function
35 } // end String copy constructor
36
37 // Destructor
38 String::~String()
39 {
40    cout << "Destructor: " << sPtr << endl;
41    delete [] sPtr; // release pointer-based string memory
42 } // end ~String destructor
43
44 // overloaded = operator; avoids self assignment
45 const String &String::operator=( const String &right )
46 {
47    cout << "operator= called" << endl;
48
49    if ( &right != this ) // avoid self assignment
50    {
51       delete [] sPtr; // prevents memory leak
52       length = right.length; // new String length
53       setString( right.sPtr ); // call utility function
54    } // end if
55    else
56       cout << "Attempted assignment of a String to itself" << endl;
57
```

```
58    return *this; // enables cascaded assignments
59 } // end function operator=
60
61 // concatenate right operand to this object and store in this object
62 const String &String::operator+=( const String &right )
63 {
64    size_t newLength = length + right.length; // new length
65    char *tempPtr = new char[ newLength + 1 ]; // create memory
66
67    strcpy( tempPtr, sPtr ); // copy sPtr
68    strcpy( tempPtr + length, right.sPtr ); // copy right.sPtr
69
70    delete [] sPtr; // reclaim old space
71    sPtr = tempPtr; // assign new array to sPtr
72    length = newLength; // assign new length to length
73    return *this; // enables cascaded calls
74 } // end function operator+=
75
76 // is this String empty?
77 bool String::operator!() const
78 {
79    return length == 0;
80 } // end function operator!
81
82 // Is this String equal to right String?
83 bool String::operator==( const String &right ) const
84 {
85    return strcmp( sPtr, right.sPtr ) == 0;
86 } // end function operator==
87
```

```
88 // Is this String less than right String?
89 bool String::operator<( const String &right ) const
90 {
91    return strcmp( sPtr, right.sPtr ) < 0;
92 } // end function operator<
93
94 // return reference to character in String as a modifiable lvalue
95 char &String::operator[]( int subscript )
96 {
97    // test for subscript out of range
98    if ( subscript < 0 || subscript >= length )
99    {
100      cerr << "Error: Subscript " << subscript
101         << " out of range" << endl;
102      exit( 1 ); // terminate program
103   } // end if
104
105   return sPtr[ subscript ]; // non-const return; modifiable lvalue
106} // end function operator[]    可以作为左侧被修改
107
108// return reference to character in String as rvalue
109char String::operator[]( int subscript ) const
110{
111   // test for subscript out of range
112   if ( subscript < 0 || subscript >= length )
113   {
114      cerr << "Error: Subscript " << subscript
115           << " out of range" << endl;
116      exit( 1 ); // terminate program
117   } // end if
```

```
118
119   return sPtr[ subscript ];   // returns copy of this
      element. 返回值的副本，不能作为左值
120   120    } // end function operator[]
```

```
118
119   return sPtr[ subscript ]; // returns copy of this element
120} // end function operator[]
121
122// return a substring beginning at index and of length subLength
123String String::operator()( int index, int subLength ) const
124{
125   // if index is out of range or substring length < 0,
126   // return an empty String object
127   if ( index < 0 || index >= length || subLength < 0 )
128      return ""; // converted to a String object automatically
129
130   // determine length of substring
131   int len;
132
133   if ( ( subLength == 0 ) || ( index + subLength > length ) )
134      len = length - index;
135   else
136      len = subLength;
137
138   // allocate temporary array for substring and
139   // terminating null character
140   char *tempPtr = new char[ len + 1 ];
141
142   // copy substring into char array and terminate string
143   strncpy( tempPtr, &sPtr[ index ], len );
144   tempPtr[ len ] = '\0';
```

```
145
146   // create temporary String object containing the substring
147   String tempString( tempPtr );
148   delete [] tempPtr; // delete temporary array
149   return tempString; // return copy of the temporary String
150} // end function operator()
151
152// return string length
153int String::getLength() const
154{
155   return length;
156} // end function getLength
157
158// utility function called by constructors and operator=
159void String::setString( const char *string2 )
160{
161   sPtr = new char[ length + 1 ]; // allocate memory
162
163   if ( string2 != 0 ) // if string2 is not null pointer, copy contents
164      strcpy( sPtr, string2 ); // copy literal to object
165   else // if string2 is a null pointer, make this an empty string
166      sPtr[ 0 ] = '\0'; // empty string
167} // end function setString
168
169// overloaded output operator
170ostream &operator<<( ostream &output, const String &s )
171{
172   output << s.sPtr;
173   return output; // enables cascading
174} // end function operator<<
```

```
175
176// overloaded input operator
177istream &operator>>( istream &input, String &s )
178{
179   char temp[ 100 ]; // buffer to store input
180   input >> setw( 100 ) >> temp;
181   s = temp; // use String class assignment operator
182   return input; // enables cascading
183} // end function operator>>
```

补：

- char *strncpy

  (char* strDest,const char* strSour,size_t count);

- 拷贝一个字符串到另外一个字符串
  - strDest是目标字符串地址
  - strSour是源字符串地址
  - size_t是要拷贝的字符个数
    - ✓ count<=strlen(strDest)， strDest的最后不会加\0
    - ✓ count> strlen(strDest)，strDest最后会自动追加count - strlen(strSour)个\0，
  - 当strDest==strSour，既要拷贝的和要被拷贝的是同一个地址时，将不处理。

```cpp
// Fig. 11.11: fig11_11.cpp
// String class test program.
#include <iostream>
using std::cout;
using std::endl;
using std::boolalpha;

#include "String.h"

int main()
{
   String s1( "happy" );
   String s2( " birthday" );
   String s3;

   // test overloaded equality and relational operators
   cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
      << "\"; s3 is \"" << s3 << '\"'
      << boolalpha << "\n\nThe results of comparing s2 and s1:"
      << "\ns2 == s1 yields " << ( s2 == s1 )
      << "\ns2 != s1 yields " << ( s2 != s1 )
      << "\ns2 >  s1 yields " << ( s2 > s1 )
      << "\ns2 <  s1 yields " << ( s2 < s1 )
      << "\ns2 >= s1 yields " << ( s2 >= s1 )
      << "\ns2 <= s1 yields " << ( s2 <= s1 );


   // test overloaded String empty (!) operator
   cout << "\n\nTesting !s3:" << endl;

```

Use overloaded stream insertion operator for **String**s

Use overloaded equality and relational operators for **String**s

```
31    if ( !s3 )
32    {
33       cout << "s3 is empty; assigning s1 to s3;" << endl;
34       s3 = s1; // test overloaded assignment
35       cout << "s3 is \"" << s3 << "\"";
36    } // end if
37
38    // test overloaded String concatenation operator
39    cout << "\n\ns1 += s2 yields s1 = ";
40    s1 += s2; // test overloaded concatenation
41    cout << s1;
42
43    // test conversion constructor
44    cout << "\n\ns1 += \" to you\" yields" << endl;
45    s1 += " to you"; // test conversion constructor
46    cout << "s1 = " << s1 << "\n\n";
47
48    // test overloaded function call operator () for substring
49    cout << "The substring of s1 starting at\n"
50       << "location 0 for 14 characters, s1(0, 14), is:\n"
51       << s1( 0, 14 ) << "\n\n";
52
53    // test substring "to-end-of-String" option
54    cout << "The substring of s1 starting at\n"
55       << "location 15, s1(15), is: "
56       << s1( 15 ) << "\n\n";
57
58    // test copy constructor
59    String *s4Ptr = new String( s1 );
60    cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";
```

Use overloaded negation operator for **String**s

Use overloaded assignment operator for **String**s

Use overloaded addition assignment operator for **String**s

char * string is converted to a **String** before using the overloaded addition assignment operator

Use overloaded function call operator for **String**s

```
61
62    // test assignment (=) operator with self-assignment
63    cout << "assigning *s4Ptr to *s4Ptr" << endl;
64    *s4Ptr = *s4Ptr; // test overloaded assignment
65    cout << "*s4Ptr = " << *s4Ptr << endl;
66
67    // test destructor
68    delete s4Ptr;
69
70    // test using subscript operator to create a modifiable lvalue
71    s1[ 0 ] = 'H';
72    s1[ 6 ] = 'B';
73    cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
74       << s1 << "\n\n";
75
76    // test subscript out of range
77    cout << "Attempt to assign 'd' to s1[30] yields:" << endl;
78    s1[ 30 ] = 'd'; // ERROR: subscript out of range
79    return 0;
80 } // end main
```

Use overloaded subscript operator for **`String`**s

Attempt to access a subscript outside of the valid range

```
Conversion (and default) constructor: happy
Conversion (and default) constructor:  birthday
Conversion (and default) constructor:
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing !s3:
s3 is empty; assigning s1 to s3;
operator= called
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
Conversion (and default) constructor:  to you
Destructor:  to you
s1 = happy birthday to you

Conversion (and default) constructor: happy birthday
Copy constructor: happy birthday
Destructor: happy birthday
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is:
happy birthday
```

The constructor and destructor are called for the temporary **`String`**

(continued at top of next slide...)

(... continued from bottom of previous slide)

```
Destructor: happy birthday
Conversion (and default) constructor: to you
Copy constructor: to you
Destructor: to you
The substring of s1 starting at
location 15, s1(15), is: to you

Destructor: to you
Copy constructor: happy birthday to you

*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
operator= called
Attempted assignment of a String to itself


*s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1[30] yields:
Error: Subscript 30 out of range
```

Boolalpha（补）

- *When the boolalpha format flag is set, bool values are insterted/extracted as their names: true and false instead of integral values.*

  This flag can be unset with the noboolalpha manipulator.

```
int main ()
{ bool b; b=true;
 cout << boolalpha << b << endl;
cout << noboolalpha << b << endl;
 return 0; }
```

输出结果

```
true
1
```

11 Standard Library Class string

- string 类
  - <string>, namespace std
  - 可以初始化：string s1("hi");
  - 重载了 << (as in cout << s1)
  - 重载了关系运算符：==, !=, >=, >, <=, <
  - 重载了赋值运算符 =
  - 重载了 +=

11 Standard Library Class string

- string 类
  - substr 成员函数
    - ✓s1.substr(0, 14);//从位置 0 取 14 个字符
    - ✓s1.substr(15);//取从位置 15 开始到结束

11 Standard Library Class string

- string 类
  - 重载了 []
    - ✓ 访问一个字符
    - ✓ 无边界检查
  - at 成员函数
    - ✓ 访问一个字符：s1.at(10);
    - ✓ 具有边界检查，如果下标越界将抛出异常

12 Overloading ++ and --

- ++/-- 运算符可以被重载
  - 假设我们想对 Date 对象进行加 1 操作
  - 成员函数原型
    - Date &operator++();
    - ++d1 变为 d1.operator++()
  - 全局函数原型
    - Date &operator++(Date &);
    - ++d1 变为 operator++(d1)

12 Overloading ++ and --

- 区分前加和后加
  - 后加带有一个空参数 (int 型, 值为 0)
  - 成员函数原型
    - Date operator++(int);
    - d1++ 变为 d1.operator++(0)
  - 全局函数原型
    - Date operator++(Date &, int);
    - d1++ 变为 operator++(d1, 0)

12 Overloading ++ and --

- 返回值
  - 前加
    - ✓返回引用（Date &），可以作为左值
  - 后加
    - ✓返回值：返回具有原来值的临时对象
    - ✓*右值（不能出现在等号左侧）*
- 以上规定同样适用于 -- 操作

13 Case Study: A Date Class

- Date 类
  - 重载 ++ 运算符来改变年/月/日
  - 重载 += 运算符
  - 检测闰年
  - 检测月末最后一天

```
1  // Fig. 11.12: Date.h
2  // Date class definition.
3  #ifndef DATE_H
4  #define DATE_H
5
6  #include <iostream>
7  using std::ostream;
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    bool leapYear( int ) const; // 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
```

Note the difference between prefix and postfix increment

```
1  // Fig. 11.13: Date.cpp
2  // Date class member-function definitions.
3  #include <iostream>
4  #include "Date.h"
5
6  // initialize static member at file scope; one classwide copy
7  const int Date::days[] =
8     { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
9
10 // Date constructor
11 Date::Date( int m, int d, int y )
12 {
13    setDate( m, d, y );
14 } // end Date constructor
15
16 // set month, day and year
17 void Date::setDate( int mm, int dd, int yy )
18 {
19    month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
20    year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;
21
22    // test for a leap year
23    if ( month == 2 && leapYear( year ) )
24       day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
25    else
26       day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
27 } // end function setDate
```

```
28
29 // overloaded prefix increment operator
30 Date &Date::operator++()
31 {
32    helpIncrement(); // increment date
33    return *this; // reference return to create an lvalue
34 } // end function operator++
35
36 // overloaded postfix increment operator; note that the
37 // dummy integer parameter does not have a parameter name
38 Date Date::operator++( int )
39 {
40    Date temp = *this; // hold current state of object
41    helpIncrement();
42
43    // return unincremented, saved, temporary object
44    return temp; // value return; not a reference return
45 } // end function operator++
46
47 // add specified number of days to date
48 const Date &Date::operator+=( int additionalDays )
49 {
50    for ( int i = 0; i < additionalDays; i++ )
51       helpIncrement();
52
53    return *this; // enables cascading
54 } // end function operator+=
55
```

Postfix increment updates object and returns a copy of the original

Do not return a reference to `temp`, because it is a local variable that will be destroyed

```
56 // if the year is a leap year, return true; otherwise, return false
57 bool Date::leapYear( int testYear ) const
58 {
59    if ( testYear % 400 == 0 ||
60       ( testYear % 100 != 0 && testYear % 4 == 0 ) )
61       return true; // a leap year
62    else
63       return false; // not a leap year
64 } // end function leapYear
65
66 // determine whether the day is the last day of the month
67 bool Date::endOfMonth( int testDay ) const
68 {
69    if ( month == 2 && leapYear( year ) )
70       return testDay == 29; // last day of Feb. in leap year
71    else
72       return testDay == days[ month ];
73 } // end function endOfMonth
74
```

```
75 // function to help increment the date
76 void Date::helpIncrement()           // 真正实现加 1 天
77 {
78    // day is not end of month
79    if ( !endOfMonth( day ) )
80       day++; // increment day
81    else
82       if ( month < 12 ) // day is end of month and month < 12
83       {
84          month++; // increment month
85          day = 1; // first day of new month
86       } // end if
87       else // last day of year
88       {
89          year++; // increment year
90          month = 1; // first month of new year
91          day = 1; // first day of new month
92       } // end else
93 } // end function helpIncrement
94 //-------------------------------------------------------------------------
95 // overloaded output operator
96 ostream &operator<<( ostream &output, const Date &d )
97 {
98    static char *monthName[ 13 ] = { "", "January", "February",
99       "March", "April", "May", "June", "July", "August",
100      "September", "October", "November", "December" };
101   output << monthName[ d.month ] << ' ' << d.day << ", " << d.year;
102   return output; // enables cascading
103} // end function operator<<
```

```cpp
// Fig. 11.14: fig11_14.cpp
// Date class test program.
#include <iostream>
using std::cout;
using std::endl;

#include "Date.h" // Date class definition

int main()
{
   Date d1; // defaults to January 1, 1900
   Date d2( 12, 27, 1992 ); // December 27, 1992
   Date d3( 0, 99, 8045 ); // invalid date

   cout << "d1 is " << d1 << "\nd2 is " << d2 << "\nd3 is " << d3;
   cout << "\n\nd2 += 7 is " << ( d2 += 7 );

   d3.setDate( 2, 28, 1992 );
   cout << "\n\n  d3 is " << d3;
   cout << "\n++d3 is " << ++d3 << " (leap year allows 29th)";

   Date d4( 7, 13, 2002 );

   cout << "\n\nTesting the prefix increment operator:\n"
      << "  d4 is " << d4 << endl;
   cout << "++d4 is " << ++d4 << endl;
   cout << "  d4 is " << d4;

}
```

Demonstrate prefix increment

```
29    cout << "\n\nTesting the postfix increment operator:\n"
30       << "  d4 is " << d4 << endl;
31    cout << "d4++ is " << d4++ << endl;
32    cout << "  d4 is " << d4 << endl;
33    return 0;
34 } // end main
```

Demonstrate postfix increment

```
d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993

  d3 is February 28, 1992
++d3 is February 29, 1992 (leap year allows 29th)

Testing the prefix increment operator:
  d4 is July 13, 2002
++d4 is July 14, 2002
  d4 is July 14, 2002

Testing the postfix increment operator:
  d4 is July 14, 2002
d4++ is July 14, 2002
  d4 is July 15, 2002
```

14 explicit Constructors

- 隐式转换
  - 由编译器执行单参数的构造函数
  - 有时候，隐式转换是不希望发生的，容易出错的
    - ✓关键字 explicit
      - ◈使得不能通过转换构造函数进行隐式转换

```cpp
// Fig. 11.15: fig11_15.cpp
// Standard Library string class test program.
#include <iostream>
using std::cout;
using std::endl;

#include <string>
using std::string;

int main()
{
   string s1( "happy" );
   string s2( " birthday" );
   string s3;

   // test overloaded equality and relational operators
   cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
      << "\"; s3 is \"" << s3 << '\"'
      << "\n\nThe results of comparing s2 and s1:"
      << "\ns2 == s1 yields " << ( s2 == s1 ? "true" : "false" )
      << "\ns2 != s1 yields " << ( s2 != s1 ? "true" : "false" )
      << "\ns2 >  s1 yields " << ( s2 > s1 ? "true" : "false" )
      << "\ns2 <  s1 yields " << ( s2 < s1 ? "true" : "false" )
      << "\ns2 >= s1 yields " << ( s2 >= s1 ? "true" : "false" )
      << "\ns2 <= s1 yields " << ( s2 <= s1 ? "true" : "false" );

   // test string member-function empty
   cout << "\n\nTesting s3.empty():" << endl;
```

Passing strings to the `string` constructor

Create empty `string`

```
29
30    if ( s3.empty() )
31    {
32       cout << "s3 is empty; assigning s1 to s3;" << endl;
33       s3 = s1; // assign s1 to s3
34       cout << "s3 is \"" << s3 << "\"";
35    } // end if
36
37    // test overloaded string concatenation operator
38    cout << "\n\ns1 += s2 yields s1 = ";
39    s1 += s2; // test overloaded concatenation
40    cout << s1;
41
42    // test overloaded string concatenation operator with C-style string
43    cout << "\n\ns1 += \" to you\" yields" << endl;
44    s1 += " to you";
45    cout << "s1 = " << s1 << "\n\n";
46
47    // test string member function substr
48    cout << "The substring of s1 starting at location 0 for\n"
49       << "14 characters, s1.substr(0, 14), is:\n"
50       << s1.substr( 0, 14 ) << "\n\n";
51
52    // test substr "to-end-of-string" option
53    cout << "The substring of s1 starting at\n"
54       << "location 15, s1.substr(15), is:\n"
55       << s1.substr( 15 ) << endl;
```

Member function **empty** tests if the **string** is empty

Member function **substr** obtains a substring from the **string**

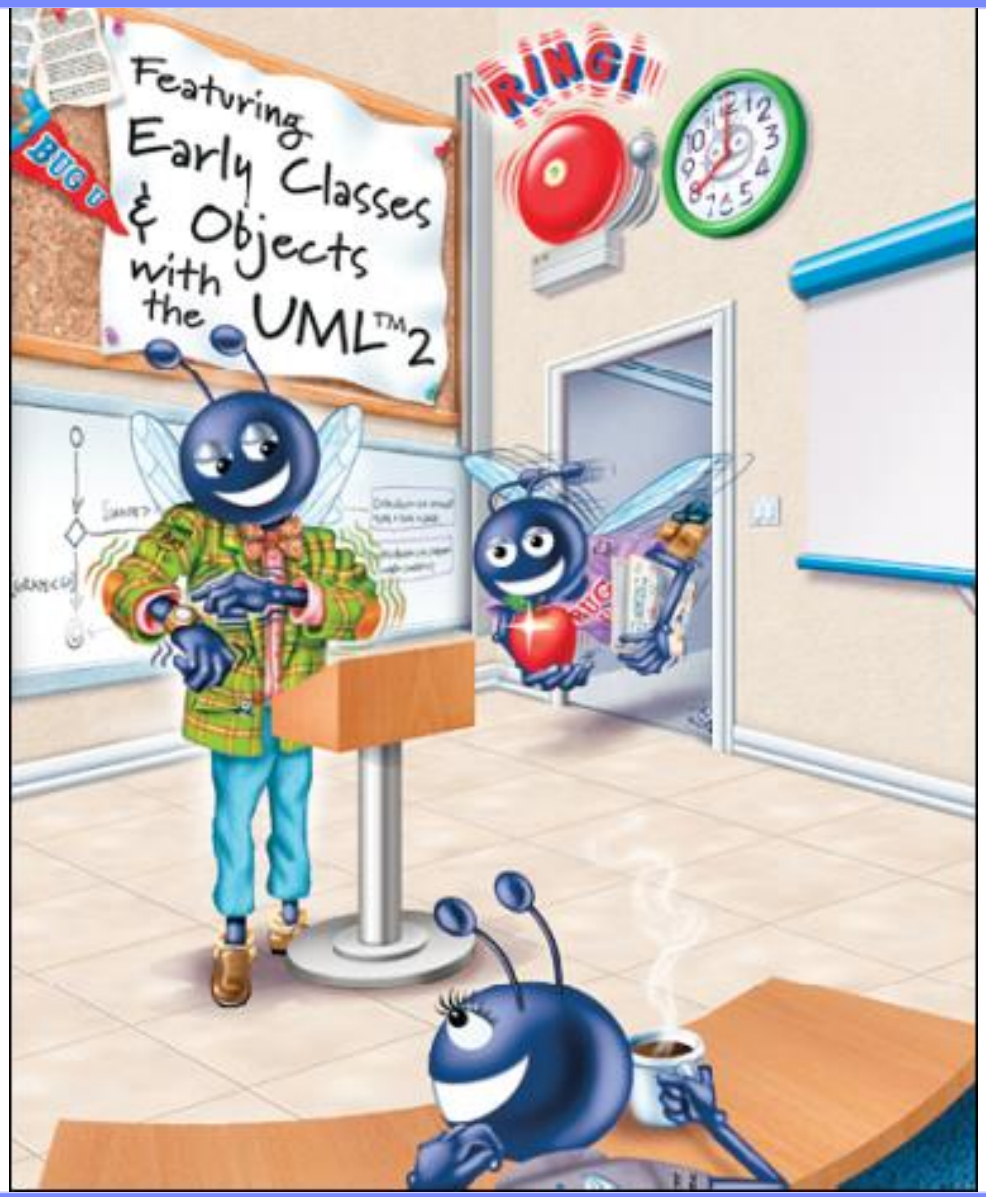
```
56
57    // test copy constructor
58    string *s4Ptr = new string( s1 );
59    cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";
60
61    // test assignment (=) operator with self-assignment
62    cout << "assigning *s4Ptr to *s4Ptr" << endl;
63    *s4Ptr = *s4Ptr;
64    cout << "*s4Ptr = " << *s4Ptr << endl;
65
66    // test destructor
67    delete s4Ptr;
68
69    // test using subscript operator to create lvalue
70    s1[ 0 ] = 'H';
71    s1[ 6 ] = 'B';
72    cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
73       << s1 << "\n\n";
74
75    // test subscript out of range with string member function "at"
76    cout << "Attempt to assign 'd' to s1.at( 30 ) yields:" << endl;
77    s1.at( 30 ) = 'd'; // ERROR: subscript out of range
78    return 0;
79 } // end main
```

Accessing specific character in **`string`**

Member function **`at`** provides range checking

```
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing s3.empty():
s3 is empty; assigning s1 to s3;
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
s1 = happy birthday to you

The substring of s1 starting at location 0 for
14 characters, s1.substr(0, 14), is:
happy birthday

The substring of s1 starting at
location 15, s1.substr(15), is:
to you

*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
*s4Ptr = happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1.at( 30 ) yields:

abnormal program termination
```

END!



Thank you!