Chapter 11

Operator Overloading; String and Array Objects

OBJECTIVES

- What operator overloading is and how it makes programs more readable and programming more convenient.
- To redefine (overload) operators to work with objects of user-defined classes.
- The differences between overloading unary and binary operators.
- To convert objects from one class to another class.
- When to, and when not to, overload operators.

Topics

- 11.1 Introduction
- 11.2 Fundamentals & Restrictions
- 11.3 Operator Functions as Class Members vs. Global Functions
- 11.4 Overloading Stream Insertion and Stream Extraction Operators
- 11.5 Overloading Unary Operators
- 11.6 Overloading Binary Operators
- 11.7 Case Study: String Class
- 11.8 Standard Library Class string

11.1 Introduction

- cout << int_variable; // 整型变量
  cout << ptrInt; // 整型指针
  cout << ptrChar; // 字符指针
- int num = 10; num = num + 1;
- int *pNum = new int[10];
  pNum = pNum + 1;
- String s1(“happy ”), s2(“birthday”);
  s1+=s1; s1=s1+s2;
  cout<<s1;

11.1 Introduction

- Date date1(2011, 1, 30);

  date1 = date1 + 1; //date1 = date1 + 2;

  cout << date1; // 如何实现?
- HugeInt HugeintA, HugeintB;

  HugeintA + HugeintB; // 如何实现?
- operator overloading 运算符重载

11.1 Introduction

- C++语言为了支持基本数据类型数据运算, 内置了多种运算符, 并且其中部分已针对操作数类型的不同进行了重载;
- 当需要将这些运算符用于用户自定义类型时, 用户可以(大部分情况下必须)进行运算符重载.

- 重载运算符的基本概念、限制, 何时选择重载?
- 如何实现重载? 全局vs 成员函数
- 拷贝构造函数/ 转换构造函数
- 自定义String类vs 标准string类

Topics

- 11.1 Introduction
- **11.2 Fundamentals & Restrictions**
- 11.3 Operator Functions as Class Members vs. Global Functions
- 11.4 Overloading Stream Insertion and Stream Extraction Operators
- 11.5 Overloading Unary Operators
- 11.6 Overloading Binary Operators
- 11.7 Case Study: Array Class
- 11.8 Standard Library Class string

11.2 Fundamentals & Restrictions—需求

- To use an operator on class objects, that operator must be overloaded with three exceptions:
  - *assignment* operator (=)
  - *address* (&) and *comma* (,) operators

- 目的: 提高类代码的可用、可读性
- HugeintA.add(HugeintB) vs HugeintA + HugeintB

11.2 Fundamentals & Restrictions—语法

- 运算符重载只是一种“语法上的方便”, 也就是说它只是另一种函数调用的方式. 区别:
  - 定义方式
  - 调用方式
- 定义重载的运算符(可视为特殊函数)就像定义函数(全局/成员), 区别是该函数的名称是

operator@

其中**operator**是关键词, **@**是被重载的运算符, 如:

```
HugeInt operator+(const HugeInt& a);
```

11.2 Fundamentals & Restrictions—语法

- 运算符重载只是一种“语法上的方便”，也就是说它只是另一种函数调用的方式. 区别:
  - 定义方式
  - 调用方式
- 普通函数
  - 全局函数: 函数名(参数列表)
  - 类成员函数: 对象.函数名(参数列表)等
- 重载的运算符: 使用时可以以表达式形式出现
  - **HugeIntA.operator+(HugeIntB)**
  - **HugeIntA + HugeIntB**

11.2 Fundamentals & Restrictions—限制

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

Operators that cannot be overloaded			
.	.*	::	?:

11.2 Fundamentals & Restrictions—限制

- 不能更改Precedence(优先级), Associativity(结合律) 以及Number of Operands(操作数数目)
- 仅能重载现有运算符, 不能创造新运算符
- 仅能重载应用于用户定义数据类型操作数的运算符
- • int + int X
- • Hugeint + Hugeint √
- • Hugeint + int √

Topics

- 11.1 Introduction
- 11.2 Fundamentals & Restrictions
- **11.3 Operator Functions as Class Members vs. Global Functions**
- 11.4 Overloading Stream Insertion and Stream Extraction Operators
- 11.5 Overloading Unary Operators
- 11.6 Overloading Binary Operators
- 11.7 Case Study: Array Class
- 11.8 Standard Library Class string

11.3 Operator Functions as Class Members vs. Global Functions

- 选择一: 非静态的类成员函数

```
class String{
public:
    bool operator!() const;
};
```

- 选择二: 全局函数
  - Friend (访问私有数据)
  - Non-friend (不访问私有数据)

```
class PhoneNumber{
    friend ostream& operator<< ( ostream&,
                                 const PhoneNumber & );
};
```

11.3 Operator Functions as Class Members vs. Global Functions

- 1. ( ), [ ], ->和赋值(=, +=, -=等)运算符必须重载为类的成员函数

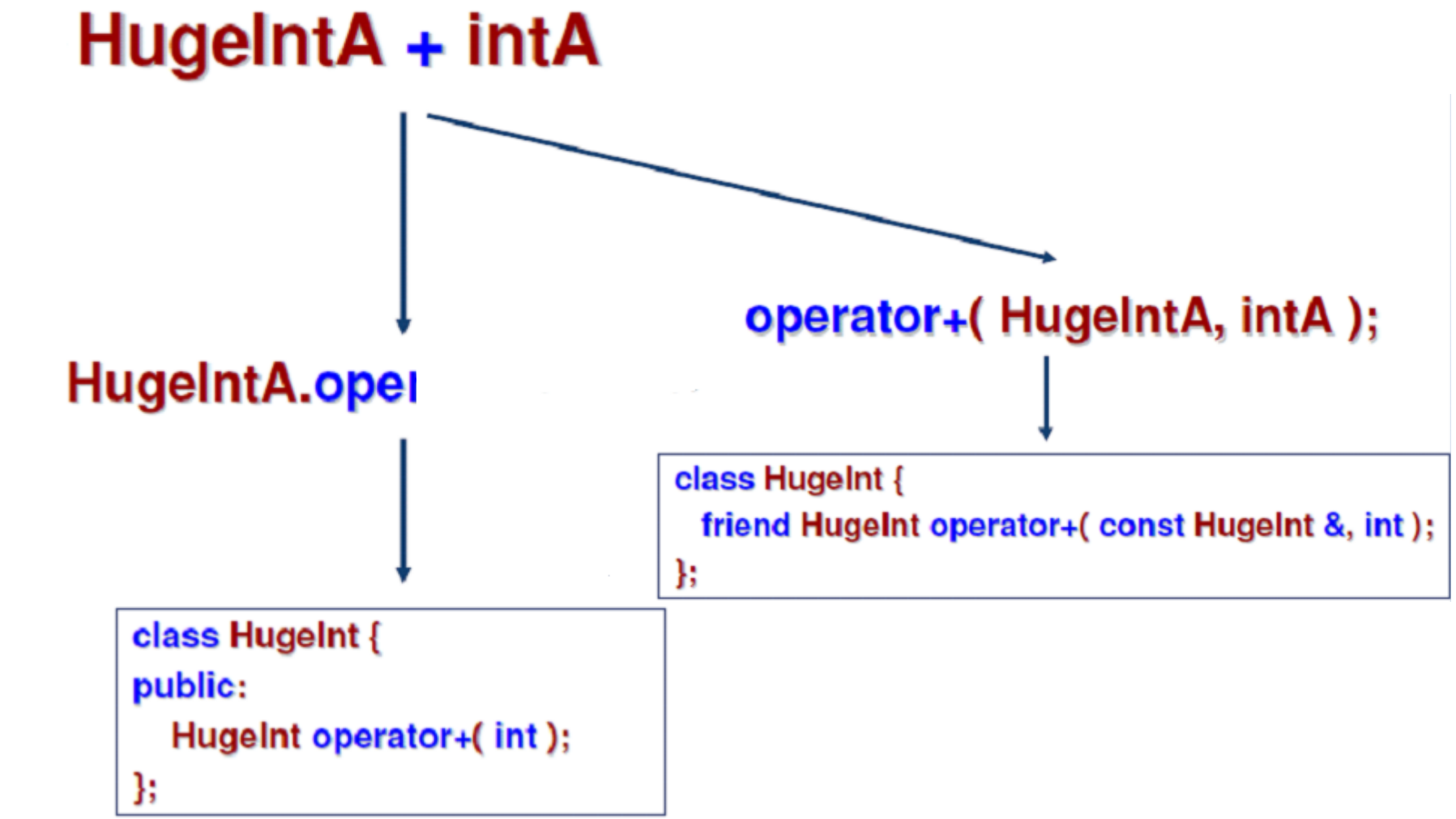
- 2. 其余运算符可以选择重载为成员或全局函数

11.3 Operator Functions as Class Members vs. Global Functions

- 当重载为类的成员函数时
- • 将自动包含该类对象(或其引用)作为操作数，因此函数参数个数等于运算符目数-1
- • 并且, 左操作数(或唯一的操作数)必须为该类对象(或对象引用)

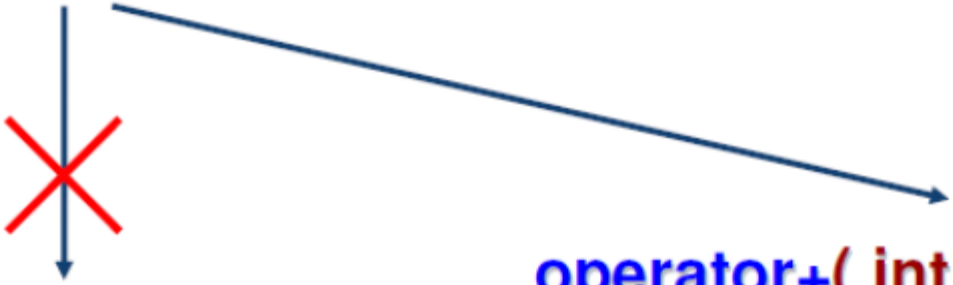
- 当重载为全局函数时
  - • 函数参数个数等于运算符目数

11.3 Operator Functions as Class Members vs. Global Functions



11.3 Operator Functions as Class Members vs. Global Functions

- intB + HugeIntB



intB.operator+(HugeIntB);

operator+(intB, HugeIntB);

```
class HugeInt {
   friend HugeInt operator+( int, const HugeInt & );
};
```

Topics

- 11.1 Introduction
- 11.2 Fundamentals & Restrictions
- 11.3 Operator Functions as Class Members vs. Global Functions
- **11.4 Overloading Stream Insertion and Stream Extraction Operators**
- 11.5 Overloading Unary Operators
- 11.6 Overloading Binary Operators
- 11.7 Case Study: Array Class
- 11.8 Standard Library Class string

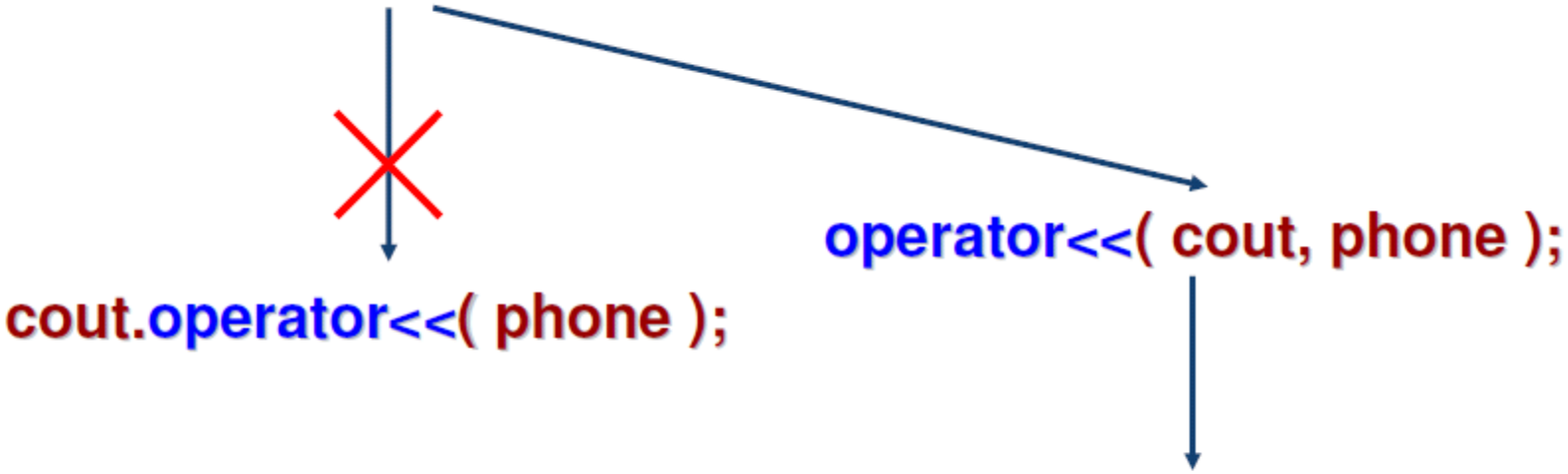
11.4 Overloading Stream Insertion and Stream Extraction Operators

- 需求:
- • cin >> phone; // (123) 456-7890
- • cout << phone; // (123) 456-7890

```
13 class PhoneNumber
14 {
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
```

11.4 Overloading Stream Insertion and Stream Extraction Operators

- cout << phone;




```
class PhoneNumber{
  friend ostream &operator<<( ostream&, const PhoneNumber & );
};
```

11.4 Overloading Stream Insertion and Stream Extraction Operators

```
12.  ostream &operator<<( ostream &output,
13.                       const PhoneNumber &number )
14.  {
15.     output << "(" << number.areaCode << ") "
16.            << number.exchange << "-" << number.line;
17.     return output; // enables cout << a << b << c;
18.  } // end function operator<<
```

程序解读（P353）

11.4 Overloading Stream Insertion and Stream Extraction Operators

- char s[20];

  cin >> setw(n) >> s; // n = 20

  指定最多读入n-1个字符, 并在尾部自动添加null character

- string s;

  cin >> setw(n) >> s;

  指定读入n个字符赋值给s

Topics

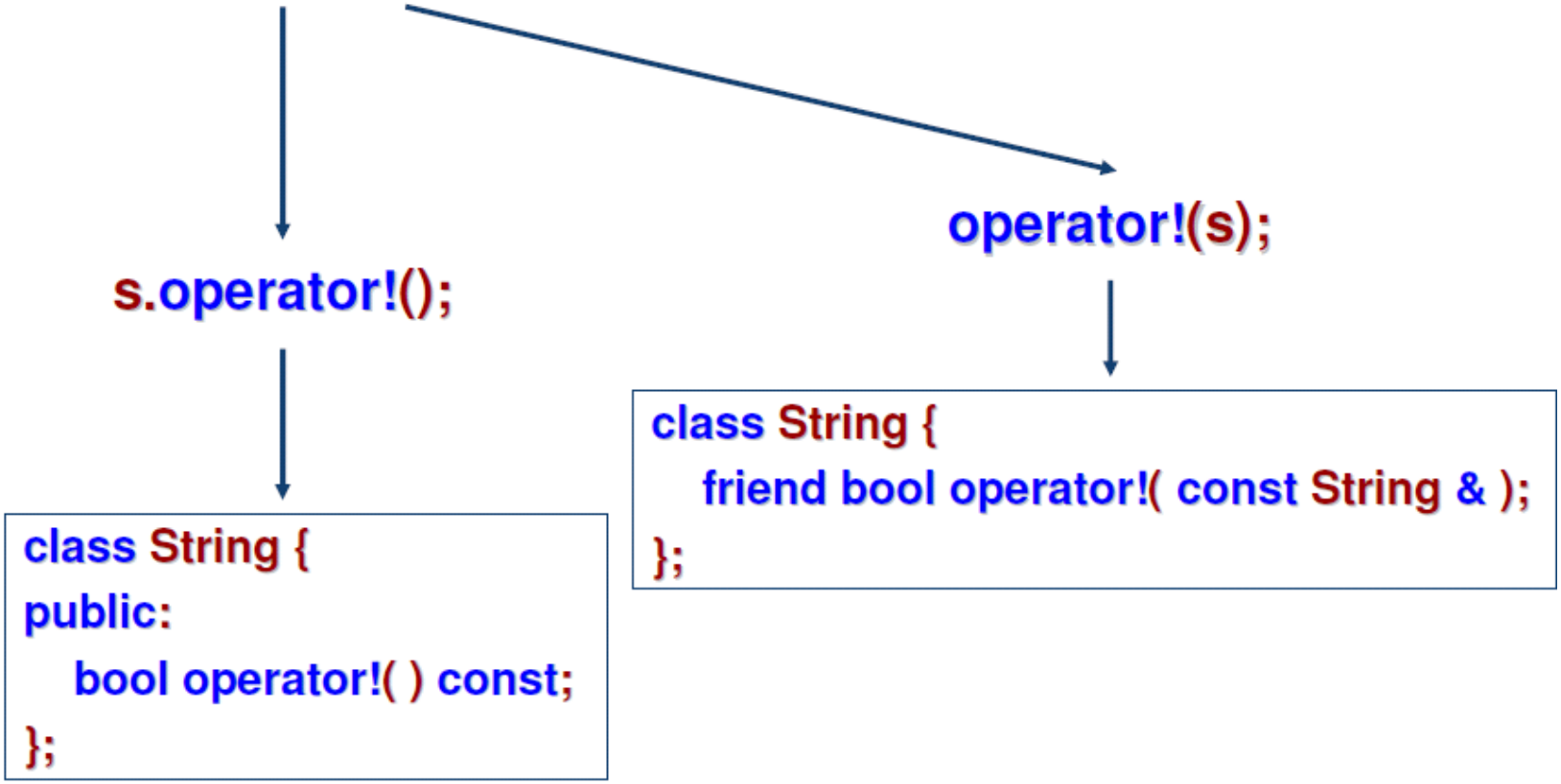
- 11.1 Introduction
- 11.2 Fundamentals & Restrictions
- 11.3 Operator Functions as Class Members vs. Global Functions
- 11.4 Overloading Stream Insertion and Stream Extraction Operators
- **11.5 Overloading Unary Operators**
- 11.6 Overloading Binary Operators
- 11.7 Case Study: Array Class
- 11.8 Standard Library Class string

11.5 Overloading Unary Operators

- 一元运算符重载
- • As a non-static member function with no arguments (无参数的非静态成员函数)
- • As a global function with one argument (一个参数的全局函数)
- • Argument must be either an object of the class or a reference to an object of the class (参数必须是对象或者对象的引用)

11.5 Overloading Unary Operators

- String s; // 设计!s 判断是否为空字符串



Topics

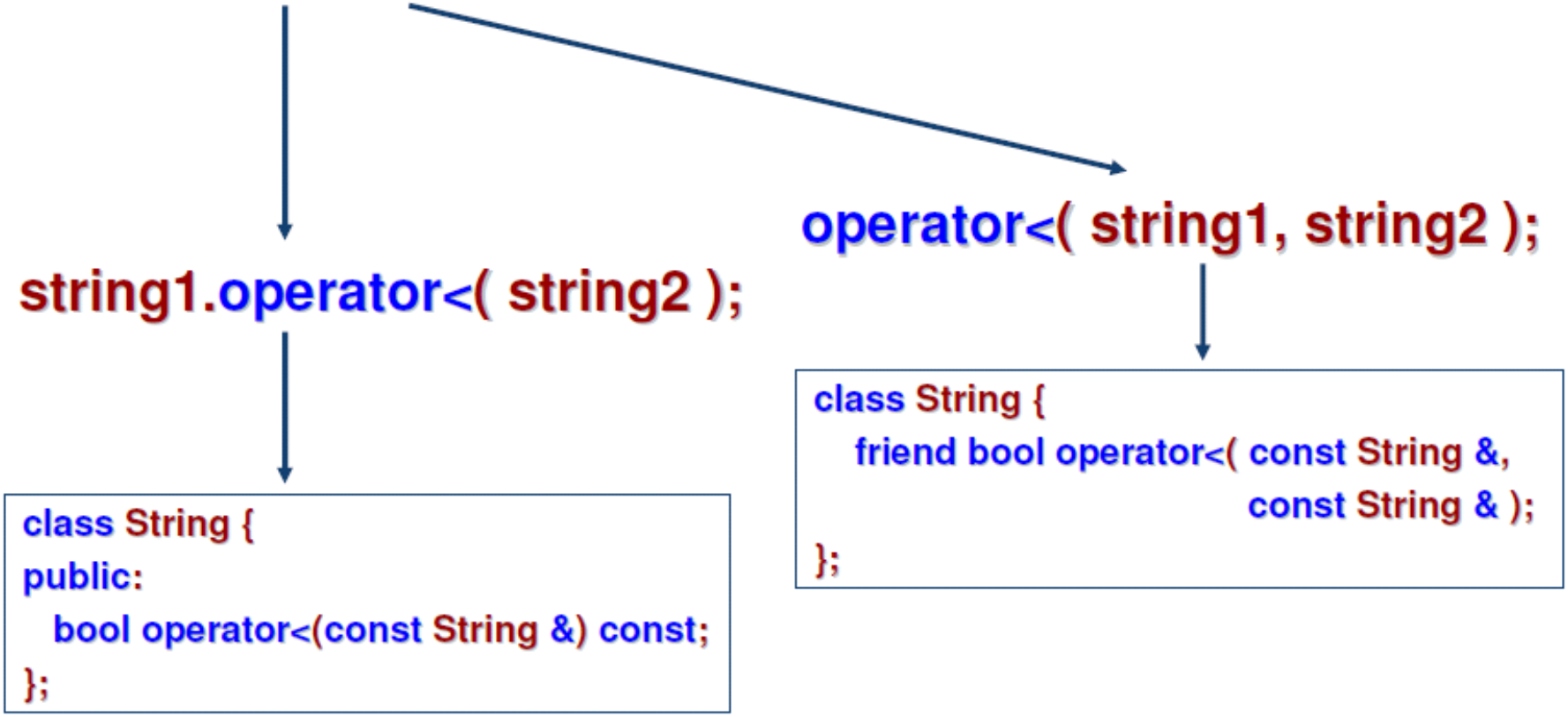
- 11.1 Introduction
- 11.2 Fundamentals & Restrictions
- 11.3 Operator Functions as Class Members vs. Global Functions
- 11.4 Overloading Stream Insertion and Stream Extraction Operators
- 11.5 Overloading Unary Operators
- **11.6 Overloading Binary Operators**
- 11.7 Case Study: Array Class
- 11.8 Standard Library Class string

11.6 Overloading Binary Operators

- 二元运算符重载
- • as a non-static member function with one argument (一个参数的非静态成员函数)
- • as a global function with two arguments (两个参数的全局函数)
- • one of those arguments must be either a class object or a reference to a class object (至少一个参数必须是对象或者对象的引用)

11.6 Overloading Binary Operators

- string1 < string2



Topics

- 11.1 Introduction
- 11.2 Fundamentals & Restrictions
- 11.3 Operator Functions as Class Members vs. Global Functions
- 11.4 Overloading Stream Insertion and Stream Extraction Operators
- 11.5 Overloading Unary Operators
- 11.6 Overloading Binary Operators
- **11.7 Case Study: Array Class**
- 11.8 Standard Library Class string

11.7 Case Study: Array Class

- Array Class 需求
  - Array a1(7), a2;
  - cout<<a1.getSize;
  - cin>>a1>>a2; cout<<a1<<a2;
  - if(a1!=a2) //if(a1==a2)
  - a1[2]=12;
  - cout<<a1[2];
  - Array a3(a1); //Array a3=a1;
  - a2=a1;

```
Array::Array(int arraySize)
{
    if(arraySize>0)
        size=arraySize;
    else
        size=10;
    ptr=new int[size];
    for(int i=0;i<size;i++)
        ptr[i]=0;
}
Array::~Array()
{
    delete []ptr;
}
```

11.7 Case Study: Array Class

- Array Class 需求
  - Array a1(7), a2;
  - cout<<a1.getSiz
  - cin>>a1>>a2; cc
  - if(a1!=a2) //if(a1==a2)
  - a1[2]=12;
  - cout<<a1[2];
  - Array a3(a1); //Array a3=a1;
  - a2=a1;

```
bool Array::operator==(const Array &right) const
{
    if(size!=right.size)
        return false;
    for(int i=0;i<size;i++)
        if(ptr[i]!=right.ptr[i])
            return false;
    return true;
}
```

11.7 Case Study: Array Class

- Array Class 需求
  - Array a1(7), a2;
  - cout<<a1.getSize;
  - cin>>a1>>a2; cout<<
  - if(a1!=a2) //if(a1==a2)
  - a1[2]=12;
  - cout<<a1[2];
  - Array a3(a1); //Array a3=a1;
  - a2=a1;

```
int Array::operator[](int subscript) const
{
    return ptr[subscript];
}

int &Array::operator[](int subscript)
{
    return ptr[subscript];
}
```

11.7 Case Study: Array Class

- 拷贝构造函数Copy Constructor

```
int a = 10; // 初始化
a = 100; // 赋值

Num b;
Num a = b; // 拷贝构造
a = b; // 赋值
```

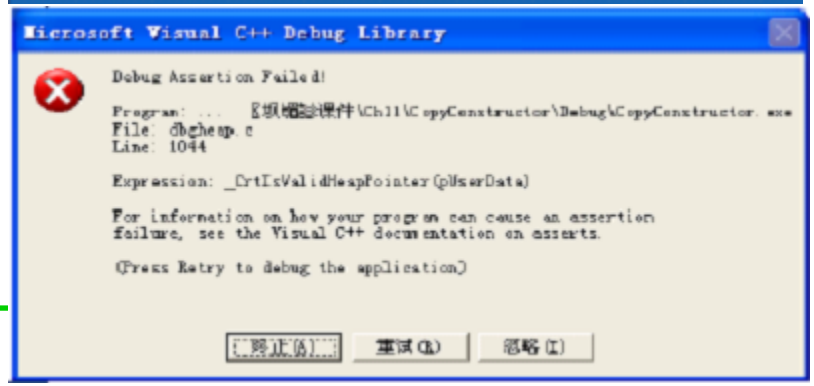
11.7 Case Study: Array Class

```
class Num{
public:
  Num(){
    nums = new int[10];
    for( int i=0; i<10; i++) nums[i] = i;
  }
  void setvalue( int i, int v ){ nums[i] = v; }
  void print(){
    cout << nums << ": ";
    for( int i=0; i<10; i++ )
      cout << nums[i] << " ";
    cout << endl;
  }
  ~Num(){ delete [] nums; }
private:
  int *nums;
};
```

```
int main()
{
  Num a;
  a.setvalue( 0, 100 );
  a.print();

  Num b = a;
  b.print();
  return 0;
};
```

```
00031090: 100 1 2 3 4 5 6 7 8 9
00031090: 100 1 2 3 4 5 6 7 8 9
```



11.7 Case Study: Array Class

```
class Num{
public:    1. 拷贝构造函数: 参数为同类对象引用的构造函数!
    ..............
    Num (const Num & n){
        nums = new int[10];
        for( int i=0; i<10; i++ )
            nums[i] = n.nums[i];
        cout << "Copy constructor called." << endl;
    }
    ..............
};
```

```
00031090: 100 1 2 3 4 5 6 7 8 9
Copy constructor called.
00031168: 100 1 2 3 4 5 6 7 8 9
```

11.7 Case Study: Array Class

- 拷贝构造函数Copy Constructor, 何时被调用:
  - 传值方式传递对象参数
  - 函数返回对象
  - 使用同类对象来初始化对象

- 总结: 当类中含有需要动态分配内存的指针数据成员, 应提供拷贝构造函数并重载赋值运算符, 以避免缺省拷贝和赋值.

11.7 Case Study: Array Class

```
class Num{
public:
  Num(){
    nums = new int[10];
    for( int i=0; i<10; i++) nums[i] = i;
  }
  Num (const Num & n){
    nums = new int[10];
    for( int i=0; i<10; i++ )
       nums[i] = n.nums[i];
    cout << "Copy constructor called." << endl;
  }
  void setvalue( int i, int v ){ nums[i] = v; }
  void print(){
    cout << nums << ": ";
    for( int i=0; i<10; i++ )
       cout << nums[i] << " ";
    cout << endl;
  }
  ~Num(){ delete [] nums; }
private:
  int *nums;
};
```

```
int main()
{
  Num a;
  a.setvalue( 0, 100 );
  a.print();

  Num b = a;
  b.print();
  return 0;
};
```

```
00031090: 100 1 2 3 4 5 6 7 8 9
Copy constructor called.
00031168: 100 1 2 3 4 5 6 7 8 9
```

11.7 Case Study: Array Class

- Array Class 需求
  - Array a1(7), a2;
  - cout<<a1.getSize;

```
Array::Array(const Array &arrayToCopy):size(arrayToCopy.size)
{
    ptr=new int[size];
    for(int i=0;i<size;i++)
        ptr[i]=arrayToCopy.prt[i];
}
```

  - cout<<a1[2];
  - Array a3(a1); //A
  - a2=a1;

```
const Array &Array::operator=(const Array &right)
{
    if(&right!=this)
    {
        if(size!=right.size)
        {
            delete []ptr;
            size=right.size;
            ptr=new int[size];
        }
        for(int i=0;i<size;i++)
            ptr[i]=right.prt[i];
    }
    return *this;
}
```

11.7 Case Study: Array Class

- 转换构造函数Conversion Constructor 单参数的构造函数, 一般用于将其他类型的对象(包括基本数据类型)转换为当前类的对象
- **Any constructor that receives a single argument is considered to be a conversion constructor**
- 目的: 使编译器执行自动类型转化!

11.7 Case Study: Array Class

```
class One{
public:
    One() { cout << "One Constructor called." << endl; }
    ~One() { cout << "One Destructor called." << endl; }
};
class Two{
public:
    Two( const One & ) { cout << "Two Conversion Constructor called." << endl; }
    ~Two() { cout << "Two Destructor called." << endl; }
};
void f( Two t ){ cout << "Function f called." << endl; }

int main()
{
    One one;
    f( one );
    cout << "Check whether Two has be
    return 0;
}
```

2.转换构造函数: 参数为其它数据类型!

11.7 Case Study: Array Class

- 转换构造函数Conversion Constructor

- 基本数据类型之间 X
- 抽象数据类型之间 √
- 抽象数据类型和基本数据类型之间 √

- 总结: 单参数构造函数
  - 相同数据类型: 拷贝构造函数
  - 不同数据类型: 转换构造函数

11.7 Case Study: String Class

- **String Class** *P451 11.9-11.11*
  - • conversion constructor
  - • copy constructor

- • 重载13个运算符<<, >>, =, +=, !, ==, <, !=, >, <=, >=, [ ], ()

```
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 * = "" ); //
16    String( const String & ); // co
17    ~String(); // destructor
18
19    const String &operator=( const String & ); // assignment operator
20    const String &operator+=( const String & ); // concatenation operator
21
22    bool operator!() const; // is String empty?
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 *`**

```
89 bool String::operator<( const String &right ) const
90 {
91    return strcmp( sPtr, right.sPtr ) < 0;
92 } // end function operator<
93
```

```
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>=
```

11.7 Case Study: Strin

```
49
50    char &operator[]( int ); // subscript operator (modifia
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 ter
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
```

Two overloaded subscript operators, for **const** and non-**const** objects

Overload the function call operator **()** to return a substring

- **()**: 函数调用运算符**function call operator**, **can take arbitrarily long and complex parameter lists.**

对象名(参数列表);

Outline

String.cpp

(1 of 7)

```
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 proto
17 using std::exit;
18
19 #include "String.h" // String cl
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
```

```
void String::setString( const char *string2 )
{
   sPtr = new char[ length + 1 ]; // allocate memory

   if ( string2 != 0 ) // if string2 is not null pointer, copy contents
      strcpy( sPtr, string2 ); // copy literal to object
   else // if string2 is a null pointer, make this an empty string
      sPtr[ 0 ] = '\0'; // empty string
} // end function setString
```

Outline

String.cpp

(2 of 7)

```
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=
```

- 赋值运算符：为何返回const?

  const String &operator=(const String &);

  避免(a = b) = c;

```
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==
```

```
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 retur
106} // end function operator[]
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
119   return sPtr[ subscript ]; // returns copy of this element
120} // end function operator[]
```

Outline

String.cpp

(4 of 7)

- 下标运算符：为何给两个函数？
  - ✓ char &operator[ ](int); // modifiable lvalue
    String s1 = “abc”; s1[1] = ‘z’;
  - ✓ char operator[ ](int) const; // rvalue
    const String s2 = “abc”; cout << s2[1];

Outline

String.cpp

```
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
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 a
139   // terminating null character
140   char *tempPtr = new char[ len + 1 ];
141
142   // copy substring into char array and termi
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()
```

```
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";
```

```
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
```

```
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>>
```

Outline

String.cpp

(7 of 7)

Outline

fig11_11.cpp

(1 of 5)

```
1  // Fig. 11.11: fig11_11.cpp
2  // String class test program.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::boolalpha;
7
8  #include "String.h"
9
10 int main()
11 {
12    String s1( "happy" );
13    String s2( " birthday" );
14    String s3;
15
16    // test overloaded equality and relational operators
17    cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
18       << "\"; s3 is \"" << s3 << '\"'
19       << boolalpha << "\n\nThe results of comparing s2 and s1:"
20       << "\ns2 == s1 yields " << ( s2 == s1 )
21       << "\ns2 != s1 yields " << ( s2 != s1 )
22       << "\ns2 >  s1 yields " << ( s2 > s1 )
23       << "\ns2 <  s1 yields " << ( s2 < s1 )
24       << "\ns2 >= s1 yields " << ( s2 >= s1 )
25       << "\ns2 <= s1 yields " << ( s2 <= s1 );
26
27
28    // test overloaded String empty (!) operator
29    cout << "\n\nTesting !s3:" << endl;
30
```

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

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

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

```
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
```

```
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),
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 **Strings**

Use overloaded assignment operator for **Strings**

Use overloaded addition assignment operator for **Strings**

```
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

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
```

```
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
```

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

fig11_11.cpp

(3 of 5)

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

Attempt to access a subscript outside of the valid range

```
*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
```

Topics

- 11.1 Introduction
- 11.2 Fundamentals & Restrictions
- 11.3 Operator Functions as Class Members vs. Global Functions
- 11.4 Overloading Stream Insertion and Stream Extraction Operators
- 11.5 Overloading Unary Operators
- 11.6 Overloading Binary Operators
- 11.7 Case Study: Array Class
- **11.8 Standard Library Class string**

11.8 Standard Library Class string

- Header file: <string>

程序解读 P465 11.15

Summary

- 哪些运算符可以重载？何时需要重载？有何限制？如何重载？
- 成员函数vs 全局函数
- 拷贝构造函数和转换构造函数

Homework

- 实验必选题目：

  11. 8-11

- 实验任选题目：

- 作业题目(Homework)：