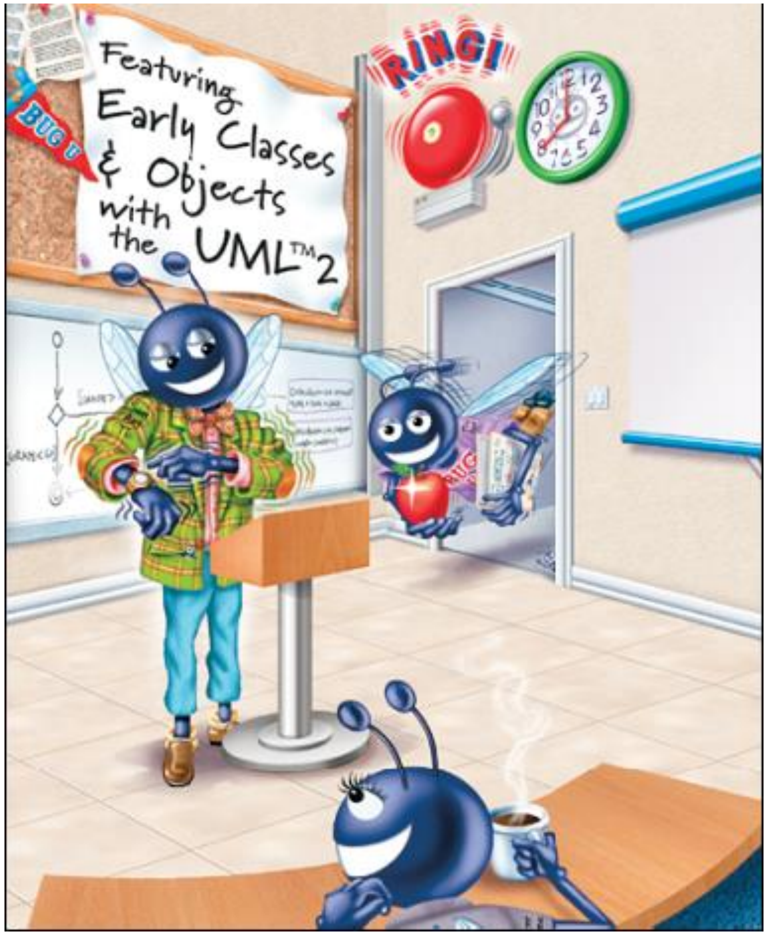
C++程序设计



上节课内容回顾

1. const 对象和 const 成员函数
2. 创建由其他对象组成的类
3. friend 函数和 friend 类
4. 使用 this 指针
5. new 和 delete
6. static 数据成员和成员函数
7. 容器类、代理类

第十讲 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

1. Introduction

- 将运算符应用于对象（运算符重载）
  - 比函数调用更清晰
  - 运算符对上下文敏感
- 例如：
  - <<：流插入，左移位
  - +：执行多种数据类型的算术运算

2. Fundamentals of Operator Overloading

- 运算符重载
  - 可以将已有运算符用于用户自定义数据类型
  - 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+ 重载 “ + ” 运算符

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 (&)：返回对象的地址
    - 逗号运算符 (,)
      - 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
  - ➢ object3 = object1.add(object2);

    vs.

    object3 = object1 + object2;

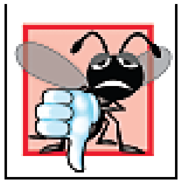
3. Restrictions on Operator Overloading

- 不能改变
  - 运算符优先级顺序
  - 运算符结合顺序（从左到右）
  - 操作数个数
  - 运算符通常意义下的功能

3. Restrictions on Operator Overloading

- 不能创建新运算符
- 运算符必须被显式重载
  - 重载 + 和 = 不意味着重载了 +=

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。运算符只能显示重载（不存在隐式重载）。

3. Restrictions on Operator Overloading

能够被重载的运算符

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

3. Restrictions on Operator Overloading

不能被重载的运算符			
.	.*	::	?:

4. Operator Functions as Class Members vs. Global Members

- 运算符函数
  - 作为成员函数
    - 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

- 运算符函数
  - ➢ 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

4. Operator Functions as Class Members vs. Global Members

- 重载 << 运算符
  - ➢ 左侧操作数为 ostream &
    - ◈例如： cout << classObject
  - ➢ 同样， 重载 >> 时左侧操作数为 istream &
  - ➢ 因此需要作为全局函数进行重载

4. Operator Functions as Class Members vs. Global Members

- 可交换的运算符
  - ➢ 可能需要 + 称为可交换的
    - ◈即 “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，则需要全局运算符函数

5. Overloading Stream Insertion and Stream Extraction Operators

- **<< 和 >> 运算符**
  - 已经被重载来处理内部数据类型
  - 也可用来处理用户自定义类型
    - ◈利用全局，友元函数进行重载

5. Overloading Stream Insertion and Stream Extraction Operators

- 例子程序
  - Class PhoneNumber
    - 包含电话号码
  - 自动的按格式来打印电话号码
    - (123) 456-7890

```
class PhoneNumber
{
   friend ostream &operator<<( ostream &, const PhoneNumber & );
   friend istream &operator>>( istream &, PhoneNumber & );
private:
   string areaCode; // 3-digit area code
   string exchange; // 3-digit exchange
   string line; // 4-digit line
};
```

```
// overloaded stream insertion operator; cannot be
// a member function if we would like to invoke it with
// cout << somePhoneNumber;
ostream &operator<<( ostream &output, const PhoneNumber &number )
{
  output << "(" << number.areaCode << ") "
    << number.exchange << "-" << number.line;
  return output; // enables cout << a << b << c;
}
```

```
// overloaded stream extraction operator; cannot be
// a member function if we would like to invoke it with
// cin >> somePhoneNumber;
istream &operator>>( istream &input, PhoneNumber &number )
{
   input.ignore(); // skip (
   input >> setw( 3 ) >> number.areaCode; // input area code
   input.ignore( 2 ); // skip ) and space
   input >> setw( 3 ) >> number.exchange; // input exchange
   input.ignore(); // skip dash (-)
   input >> setw( 4 ) >> number.line; // input line
   return input; // enables cin >> a >> b >> c;
}
```

```
int main()
{
   PhoneNumber phone; // create object phone

   // cin >> phone invokes operator>> by implicitly issuing
   // the global function call operator>>( cin, phone )
   cin >> phone;

   // cout << phone invokes operator<< by implicitly issuing
   // the global function call operator<<( cout, phone )
   cout << phone << endl;
   return 0;
} // end main
```

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

6. Overloading Unary Operators

- 后面的例子中将重载 “!” 来测试空字符串
  - 如果为非静态成员函数
    - class String

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

    - !s 将会调用: s.operator!()

6. Overloading Unary Operators

- 后面的例子中将重载 “!” 来测试空字符串
  - 如果为全局函数，需要一个参数
    - bool operator!(const String &)
    - !s 将会调用： operator!(s)

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

7. Overloading Binary Operators

- 在后面的例子中将重载： “+=”
  - 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)

7. Overloading Binary Operators

- 在后面的例子中将重载： “+=”
  - 如果为全局函数
    - const String &operator+=(String &, const String &);
    - y += z becomes operator+=(y, z)

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，必须将数组的大小作为参数同时传递

8. Case Study: Array Class

- 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
  - 当对象被拷贝时需要调用:
    - 值传递 （返回对象或将对象作为参数）
    - 用另一对象来初始化当前对象
      - Array newArray(oldArray); 或
        Array newArray = oldArray

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

```
class Array
{
   friend ostream &operator<<( ostream &, const Array & );
   friend istream &operator>>( istream &, Array & );
public:
   Array( int = 10 ); // default constructor
   Array( const Array & ); // copy constructor
   ~Array(); // destructor
   int getSize() const; // return size

   const Array &operator=( const Array & ); // assignment operator
   bool operator==( const Array & ) const; // equality operator

   bool operator!=( const Array &right ) const
   {
      return ! ( *this == right ); // invokes Array::operator==
   } // end function operator!=
   // subscript operator for non-const objects returns modifiable lvalue
   int &operator[]( int );


   // subscript operator for const objects returns rvalue
   int operator[]( int ) const;
private:
   int size; // pointer-based array size
   int *ptr; // pointer to first element of pointer-based array
}; // end class Array
```

```
// default constructor for class Array (default size 10)
Array::Array( int arraySize )
{
   size = ( arraySize > 0 ? arraySize : 10 ); // validate arraySize
   ptr = new int[ size ]; // create space for pointer-based array

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = 0; // set pointer-based array element
} // end Array default constructor
```

```
// copy constructor for class Array;
// must receive a reference to prevent infinite recursion
Array::Array( const Array &arrayToCopy )
   : size( arrayToCopy.size )
{
   ptr = new int[ size ]; // create space for pointer-based array

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = arrayToCopy.ptr[ i ]; // copy into object
} // end Array copy constructor
```

```
// destructor for class Array
Array::~Array()
{
   delete [] ptr; // release pointer-based array space
} // end destructor

// return number of elements of Array
int Array::getSize() const
{
   return size; // number of elements in Array
} // end function getSize
```

```
// overloaded assignment operator; const return avoids: ( a1 = a2 ) = a3
const Array &Array::operator=( const Array &right )
{
  if ( &right != this ) // avoid self-assignment
  {
    if ( size != right.size )
    {
      delete [] ptr; // release space
      size = right.size; // resize this object
      ptr = new int[ size ]; // create space for array copy
    } // end inner if
    for ( int i = 0; i < size; i++ )
      ptr[ i ] = right.ptr[ i ]; // copy array into object
  } // end outer if
  return *this; // enables x = y = z, for example
} // end function operator=
// determine if two Arrays are equal and
// return true, otherwise return false
bool Array::operator==( const Array &right ) const
{
   if ( size != right.size )
      return false; // arrays of different number of elements

   for ( int i = 0; i < size; i++ )
      if ( ptr[ i ] != right.ptr[ i ] )
         return false; // Array contents are not equal
   return true; // Arrays are equal
} // end function operator==
// overloaded subscript operator for non-const Arrays;
// reference return creates a modifiable lvalue
int &Array::operator[]( int subscript )
{
  // check for subscript out-of-range error
  if ( subscript < 0 || subscript >= size )
  {
    cerr << "\nError: Subscript " << subscript << " out of range" << endl;
    exit( 1 ); // terminate program; subscript out of range
  } // end if
  return ptr[ subscript ]; // reference return
} // end function operator[]
```

```
// overloaded subscript operator for const Arrays
// const reference return creates an rvalue
int Array::operator[]( int subscript ) const
 {
   // check for subscript out-of-range error
   if ( subscript < 0 || subscript >= size )
   {
     cerr << "\nError: Subscript " << subscript << " out of range" << endl;
     exit( 1 ); // terminate program; subscript out of range
   } // end if

   return ptr[ subscript ]; // returns copy of this element
} // end function operator[]
```

```
// overloaded input operator for class Array;
// inputs values for entire Array
istream &operator>>( istream &input, Array &a )
{
   for ( int i = 0; i < a.size; i++ )
      input >> a.ptr[ i ];

  return input; // enables cin >> x >> y;
} // end function
```

```cpp
// overloaded output operator for class Array
ostream &operator<<( ostream &output, const Array &a )
{
   int i;
   for ( i = 0; i < a.size; i++ )
   {
      output << setw( 12 ) << a.ptr[ i ];
      if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
         output << endl;
   } // end for

   if ( i % 4 != 0 ) // end last line of output
      output << endl;

   return output; // enables cout << x << y;
} // end function operator<<
```

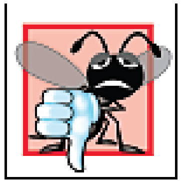
```
int main()
{
  Array integers1( 7 ); // seven-element Array
  Array integers2; // 10-element Array by default

  cout << "Size of Array integers1 is " << integers1.getSize()
     << "\nArray after initialization:\n" << integers1;

  cout << "\nSize of Array integers2 is " << integers2.getSize()
     << "\nArray after initialization:\n" << integers2;

  cout << "\nEnter 17 integers:" << endl;
  cin >> integers1 >> integers2;
```

```
cout << "\nAfter input, the Arrays contain:\n"
   << "integers1:\n" << integers1
   << "integers2:\n" << integers2;


// use overloaded inequality (!=) operator
cout << "\nEvaluating: integers1 != integers2" << endl;


if ( integers1 != integers2 )
   cout << "integers1 and integers2 are not equal" << endl;
```

```
// create Array integers3 using integers1 as an
// initializer; print size and contents
Array integers3( integers1 ); // invokes copy constructor

cout << "\nSize of Array integers3 is "<< integers3.getSize()
   << "\nArray after initialization:\n" << integers3;
// use overloaded assignment (=) operator
cout << "\nAssigning integers2 to integers1:" << endl;
integers1 = integers2; // note target Array is smaller

cout << "integers1:\n" << integers1<< "integers2:\n" << integers2;

// use overloaded equality (==) operator
cout << "\nEvaluating: integers1 == integers2" << endl;
```

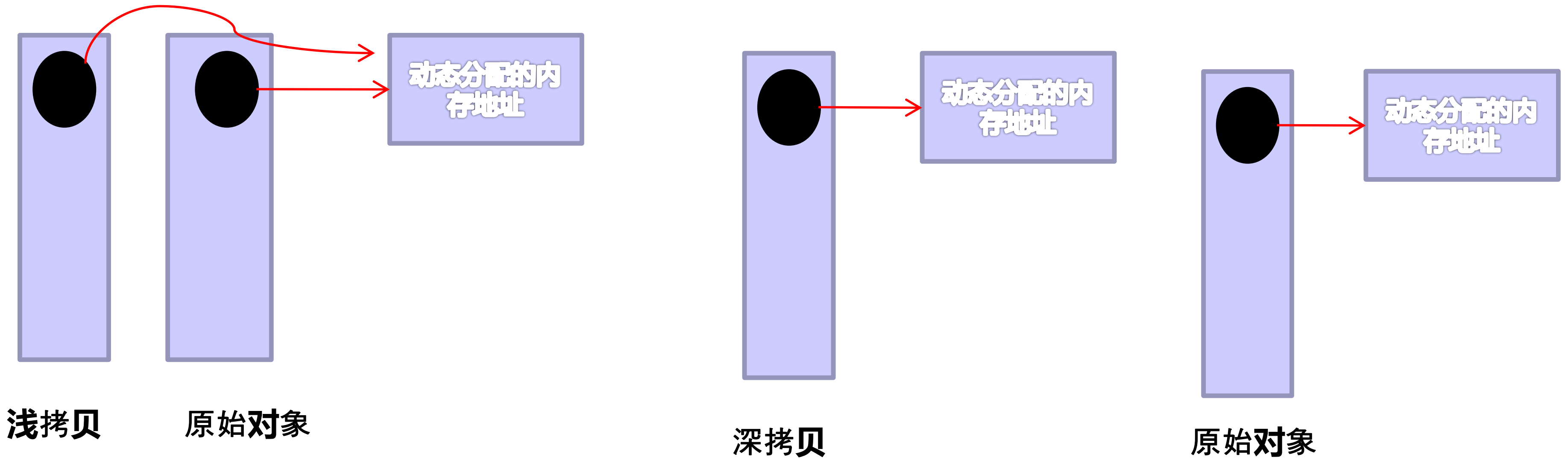
```
   if ( integers1 == integers2 )
      cout << "integers1 and integers2 are equal" << endl;

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

   // use overloaded subscript operator to create lvalue
   cout << "\n\nAssigning 1000 to integers1[5]" << endl;
   integers1[ 5 ] = 1000;
   cout << "integers1:\n" << integers1;

   // attempt to use out-of-range subscript
   cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
   integers1[ 15 ] = 1000; // ERROR: out of range
   return 0;
} // end main
```

8. Case Study: Array Class

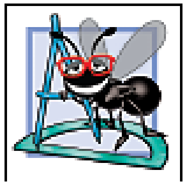


常见编程错误：如果拷贝构造函数只把源对象的指针复制到目标对象的指针，这两个对象将指向同一块动态分配的内存块，执行析构函数时将释放该内存块，结果导致另一个对象的ptr没有定义（这种指针称为“危险指针”），如果在这种情况下使用该指针，可能会引起运行时错误（例如程序过早地终止）。

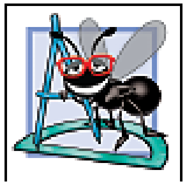
8. Case Study: Array Class



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拷贝构造函数会造成逻辑错误。

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

9. Converting between Types

- 类型转换运算符
  - 从一个类到另一个类的转换
  - 类和基本数据类型之间的转换
  - 必须为非静态成员函数
  - 无需声明返回类型
    - ◈隐式地返回转换后的类型

9. Converting between Types

- 类型转换运算符
  - 例如：
    - 原型： A::operator char *() const;
      - 将类 A 转换为临时的 char *
      - static_cast< char * >(s) 调用： s.operator char *()
    - 同样：
      - A::operator int() const;
      - A::operator OtherClass() const;

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; // 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

```
class String
{
   friend ostream &operator<<( ostream &, const String & );
   friend istream &operator>>( istream &, String & );
public:
   String( const char * = "" ); // conversion/default constructor
   String( const String & ); // copy constructor
   ~String(); // destructor

   const String &operator=( const String & ); // assignment operator
   const String &operator+=( const String & ); // concatenation operator

   bool operator!() const; // is String empty?
   bool operator==( const String & ) const; // test s1 == s2
   bool operator<( const String & ) const; // test s1 < s2
   bool operator!=( const String &right ) const
   {
      return !( *this == right );
   } // end function operator!=

   bool operator>( const String &right ) const
   {
      return right < *this;
   } // end function operator>

   bool operator<=( const String &right ) const
   {
      return !( right < *this );
   } // end function operator <=

   bool operator>=( const String &right ) const
   {
      return !( *this < right );
   } // end function operator>=
   char &operator[]( int ); // subscript operator (modifiable lvalue)
   char operator[]( int ) const; // subscript operator (rvalue)
   String operator()( int, int = 0 ) const; // return a substring
   int getLength() const; // return string length
private:
   int length; // string length (not counting null terminator)
   char *sPtr; // pointer to start of pointer-based string

   void setString( const char * ); // utility function
}; // end class String
```

```
// conversion (and default) constructor converts char * to String
String::String( const char *s )
   : length( ( s != 0 ) ? strlen( s ) : 0 )
{
   cout << "Conversion (and default) constructor: " << s << endl;
   setString( s ); // call utility function
} // end String conversion constructor
```

```
// copy constructor
String::String( const String &copy )
   : length( copy.length )
 {
   cout << "Copy constructor: " << copy.sPtr << endl;
   setString( copy.sPtr ); // call utility function
} // end String copy constructor

// Destructor
String::~String()
 {
   cout << "Destructor: " << sPtr << endl;
   delete [] sPtr; // release pointer-based string memory
} // end ~String destructor
```

```
// overloaded = operator; avoids self assignment
const String &String::operator=( const String &right )
 {
   cout << "operator= called" << endl;

   if ( &right != this ) // avoid self assignment
   {
     delete [] sPtr; // prevents memory leak
     length = right.length; // new String length
     setString( right.sPtr ); // call utility function
   } // end if
   else
     cout << "Attempted assignment of a String to itself" << endl;
   return *this
}
```

```
// concatenate right operand to this object and store in this object
const String &String::operator+=( const String &right )
{
   size_t newLength = length + right.length; // new length
   char *tempPtr = new char[ newLength + 1 ]; // create memory

   strcpy( tempPtr, sPtr ); // copy sPtr
   strcpy( tempPtr + length, right.sPtr ); // copy right.sPtr

   delete [] sPtr; // reclaim old space
   sPtr = tempPtr; // assign new array to sPtr
   length = newLength; // assign new length to length
   return *this; // enables cascaded calls
} // end function operator+=
// is this String empty?
bool String::operator!() const
{
   return length == 0;
} // end function operator!

// Is this String equal to right String?
bool String::operator==( const String &right ) const
{
   return strcmp( sPtr, right.sPtr ) == 0;
} // end function operator==
// Is this String less than right String?
bool String::operator<( const String &right ) const
{
   return strcmp( sPtr, right.sPtr ) < 0;
} // end function operator<

// return reference to character in String as a modifiable lvalue
char &String::operator[]( int subscript )
{
   if ( subscript < 0 || subscript >= length )
   {
      cerr << "Error: Subscript " << subscript << " out of range" << endl;
      exit( 1 ); // terminate program
   } // end if

   return sPtr[ subscript ]; // non-const return; modifiable lvalue
} // end function operator[]
```

```
// return reference to character in String as rvalue
char String::operator[]( int subscript ) const
{
  // test for subscript out of range
  if ( subscript < 0 || subscript >= length )
  {
    cerr << "Error: Subscript " << subscript
         << " out of range" << endl;
    exit( 1 ); // terminate program
  }
  return sPtr[ subscript ]; // returns copy of this element
}
```

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

   int len;

   if ( ( subLength == 0 ) || ( index + subLength > length ) )
      len = length - index;
   else
      len = subLength;

   char *tempPtr = new char[ len + 1 ];

   strncpy( tempPtr, &sPtr[ index ], len );
   tempPtr[ len ] = '\0';
   // create temporary String object containing the substring
   String tempString( tempPtr );
   delete [] tempPtr; // delete temporary array
   return tempString; // return copy of the temporary String
} // end function operator()

// return string length
int String::getLength() const
{
   return length;
} // end function getLength
```

```
// utility function called by constructors and operator=
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
```

```cpp
// overloaded output operator
ostream &operator<<( ostream &output, const String &s )
 {
   output << s.sPtr;
   return output; // enables cascading
} // end function operator<<
```

```cpp
// overloaded input operator
istream &operator>>( istream &input, String &s )
{
   char temp[ 100 ]; // buffer to store input
   input >> setw( 100 ) >> temp;
   s = temp; // use String class assignment operator
   return input; // enables cascading
} // end function operator>>
```

10. Case Study: String Class

- s3的构造函数
- !s3的结果
- s1 += s2   and   s1 += “ to you”
- 重载的函数调用运算符
- 重载的[]运算符
- 析构函数

11. 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)

11. 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)

11. 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**

12. 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

```cpp
class Date
 {
   friend ostream &operator<<( ostream &, const Date & );
public:
   Date( int m = 1, int d = 1, int y = 1900 ); // default constructor
   void setDate( int, int, int ); // set month, day, year
   Date &operator++(); // prefix increment operator
   Date operator++( int ); // postfix increment operator
   const Date &operator+=( int ); // add days, modify object
   bool leapYear( int ) const; // is date in a leap year?
   bool endOfMonth( int ) const; // is date at the end of month?
private:
   int month;
   int day;
   int year;
   static const int days[]; // array of days per month
   void helpIncrement(); // utility function for incrementing date
}; // end class Date
```

```
// overloaded prefix increment operator
Date &Date::operator++()
{
  helpIncrement(); // increment date
  return *this; // reference return to create an lvalue
} // end function operator++

// overloaded postfix increment operator; note that the
// dummy integer parameter does not have a parameter name
Date Date::operator++( int )
{
  Date temp = *this; // hold current state of object
  helpIncrement();

  // return unincremented, saved, temporary object
  return temp; // value return; not a reference return
} // end function operator++
```

```cpp
// if the year is a leap year, return true; otherwise, return false
bool Date::leapYear( int testYear ) const
{
   if ( testYear % 400 == 0 || ( testYear % 100 != 0 && testYear % 4 == 0 ) )
      return true; // a leap year
   else
      return false; // not a leap year
} // end function leapYear

// determine whether the day is the last day of the month
bool Date::endOfMonth( int testDay ) const
{
   if ( month == 2 && leapYear( year ) )
      return testDay == 29; // last day of Feb. in leap year
   else
      return testDay == days[ month ];
} // end function endOfMonth
```

```
void Date::helpIncrement()
{
  if ( !endOfMonth( day ) )
    day++; // increment day
  else
    if ( month < 12 ) // day is end of month and month < 12
    {
      month++; // increment month
      day = 1; // first day of new month
    } // end if
    else // last day of year
    {
      year++; // increment year
      month = 1; // first month of new year
      day = 1; // first day of new month
    } // end else
} // end function helpIncrement
```

13. Standard Library Class string

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

13. 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

13. 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

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

```
#include "Array.h"

void outputArray( const Array & ); // prototype

int main()
 {
   Array integers1( 7 ); // 7-element array
   outputArray( integers1 ); // output Array integers1
   outputArray( 3 ); // convert 3 to an Array and output Array’s contents
  return 0;
}  // end main
```

```
class Array
{
  friend ostream &operator<<( ostream &, const Array & );
  friend istream &operator>>( istream &, Array & );
public:
  explicit Array( int = 10 ); // default constructor
  Array( const Array & ); // copy constructor
  ~Array(); // destructor
  int getSize() const; // return size

  const Array &operator=( const Array & ); // assignment operator
  ……
}
```