

11

运算符重载： 字符串和数组对象

教学目标

- 什么是运算符重载，它如何使程序的可读性更强并使编程更方便
- 重新定义(**重载**)作用在用户自定义类对象上的运算符
- 重载一元和二元运算符的不同之处
- 将一个类的对象转换为另一个类的对象
- 了解运算符重载的时机
- 学习几个使用运算符重载的例子
- 使用标准库**String**类的重载运算符和其他的成员函数
- 使用关键字**explicit**防止编译器采用单参数构造函数进行隐式转换

11.1 简介

- 思考:

- 基本类型的变量可以用系统提供的运算符进行运算

```
int a, b;
```

```
a = a + b;
```

- 那么用户自定义类型（比如：类）的变量（比如：对象）可否使用运算符进行运算呢？

```
Time t1, t2;
```

```
t1 = t1 + t2;
```



11.1 简介

- 本章要介绍怎样把C++中的**运算符**和**类的对象**结合在一起使用

- 这个结合过程称为**运算符重载**

- 通过这种方法使用类的对象，比函数调用更清晰
- 比如：两个复数相加

```
Complex a(1,3), b(2,4), c;
```

```
c = a.add( b );           //原先的实现方法
```

```
c = a + b;               //重载运算符以后的实现方法
```



11.2 运算符重载的基础知识

- 我们接触过的重载运算符
 - 例如，加法运算符(+)对整数、单精度数和双精度数的操作是不相同的
 - 那是因为C++语言本身已经重载了该运算符
- 程序员自己也可以对运算符进行重载
 - 从而把运算符和用户自定义的类型一起使用
 - 但是，**C++不允许建立新的运算符**
 - 仅允许重载（即重新定义）现有的运算符，使它在用于类的对象时具有新类型的含义



11.2 运算符重载的基础知识

- 运算符重载的实现方法：
 - 是通过编写函数定义实现的
 - 为重载运算符定义的函数：
 - 大部分与普通的函数相同
 - 不同之处：
 - 函数名是由关键字operator和其后要重载的运算符符号组成
- 例如：
函数名 `operator+` 是用来重载+运算符 函数

举例 `Complex`类中重载运算符 “+”



11.2 运算符重载的基础知识

- 运算符重载

- 类的对象要使用运算符的话，运算符必须重载

- 编写运算符重载函数的形式通常有两种：

- 有时最好把这些函数定义为类的成员函数（**在类的内部**）
 - 有时最好定义为类的友元函数（**在类的外部**）

- 在极少数情况下，他们可能既不是成员函数，也不是友元函数



软件工程知识11.1

运算符重载提供了C++的可扩展性，这也是C++最吸引人的属性之一。

良好的编程习惯11.1

在完成同样的操作的情况下，如果运算符重载能够比用明确的函数调用使程序更清晰，则应该使用运算符重载。



良好的编程习惯11. 2

不要过度地或不合理地使用运算符重载，因为这样会使程序语义不清且难以阅读。

11.2 运算符重载的基础知识

- 用于类的对象的运算符**必须**重载
 - 但是有三种例外情况
 - 赋值运算符(=)
 - **Memberwise assignment between objects**
 - 取地址运算符(&)
 - **Returns address of object**
 - 逗号运算符(,)
 - 以上三种运算符用于对象时，不需要重载
 - 但并不是不能重载



11.2 运算符重载的基础知识

- 重载举例1:
 - 编写一个**Distance**类表示距离
 - 该类有两个数据成员：米和厘米
 - 该类有一个成员函数：显示距离
 - 以如下格式显示：距离是** 米 ** 厘米
 - 另外，该类要求：
 - 编写一个**add**函数实现距离的相加
 - 对**Distance**类重载加法运算符 `+`
 - `+=`



11.2 运算符重载的基础知识

- 重载举例2:
 - 编写一个Complex类表示复数
 - 该类有两个数据成员：实部和虚部
 - 该类有一个成员函数：显示复数
 - 以如下格式显示：实部 + 虚部 i
 - 另外，该类要求实现
 - 复数的相加和相减
 - ++



11.3 运算符重载的限制

- 不能创建新的运算符，只能重载现有的
- 现有C++中的大部分运算符都可以被重载
 - 但也有些是不能重载的
 - 图11.1列出了可以被重载的运算符
 - 图11.2列出了不能被重载的运算符
- 重载不能改变运算符的优先级
- 重载不能改变运算符的结合律
- 重载不能改变运算符操作数的个数



Operators that can be overloaded

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

Fig. 11.1 | Operators that can be overloaded.



Operators that cannot be overloaded

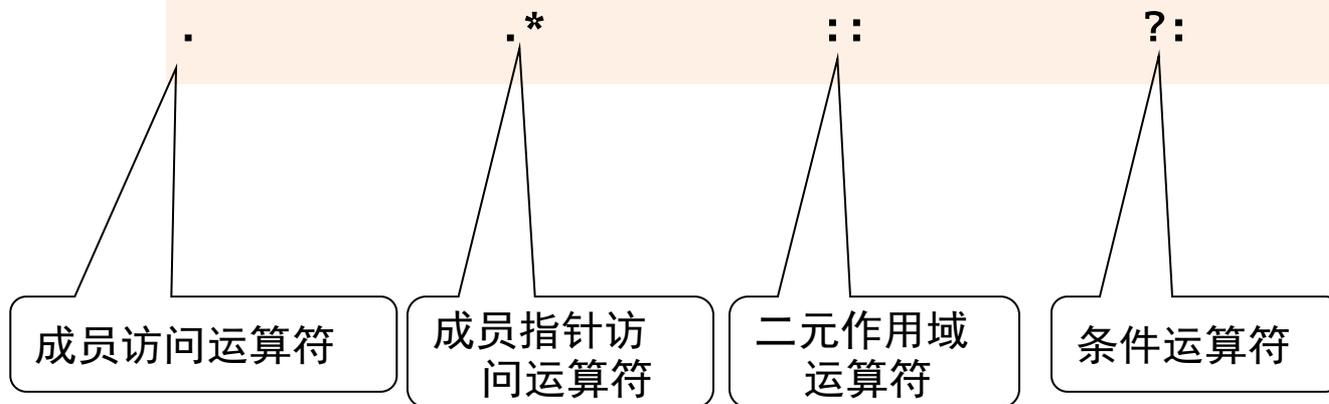


Fig. 11.2 | Operators that cannot be overloaded.



常见编程错误11.2

试图通过运算符重载改变运算符的“元数”将导致编译错误。

常见编程错误11.3

试图通过运算符重载创建新的运算符将导致语法错误。

常见编程错误11.4

试图改变运算符对**基本类型**的对象的作用方式将导致编译错误。

例如：

程序员不能改变运算符 `+` 用于两个整数时的含义。

运算符重载只能和用户自定义类型的对象一起使用，或者用于用户自定义类型的对象和基本类型对象的混合使用时。



软件工程知识11.2

重载运算符函数（作为类的友元函数重载时）的参数至少有一个必须是用户自定义类型的对象或引用。这样使程序员无法改变运算符作用在基本类型的对象上的方式。

例如：下面这样是不正确的

```
int operator+ (int a, int b)
{ return (a-b);}
```



常见的编程错误11.5

下面是常见错误观点：

认为重载了某个运算符(如 “+”)可以自动地重载相关的运算符
(如 “+=”)

或重载了 “==”就自动重载了 “!=”

运算符只能被显式重载(不存在隐式重载)。

要保证相关运算符的一致性，可以用一个运算符实现另一个运算符。

即用重载的运算符 “+”实现重载的运算符 “+=”。

11.4 作为类成员函数和全局函数的运算符函数之比较

• 重载运算符函数

– 作为类的成员函数

- 在重载运算符`()`、`[]`、`→`，或者**赋值运算符**时
- 运算符重载函数必须声明为类的一个成员函数
- 对于其他的运算符，运算符重载函数可以是非成员函数

– 作为类外部的全局函数

- 如果重载的运算符函数不作为成员函数，通常会被定义为类的友元函数
- 因为**重载的运算符函数**肯定需要使用类的数据成员（私有的）



11.4 作为类成员函数和全局函数的运算符函数之比较

- 除了`()`、`[]`、`→`、`=`之外的其他运算符函数，应该定义成成员函数还是友元函数？
- 两种运算符函数的实现原则：
 - 当运算符函数是一个成员函数时，左边的操作数必须是重载运算符的**那个类的一个对象**(或者是对该类对象的引用)。
 - 如果左边的操作数必须是一个不同于重载运算符的**那个类的对象**，或者是一个内部类型的对象，该运算符重载函数必须作为一个非成员函数来实现



11.4 作为类成员函数和全局函数的运算符函数之比较

- 重载的 << 运算符和 >> 运算符

- 对于任何类重载<<运算符，都必须有一个类型为 `ostream&` 的左操作数 `cout`

- 比如：要对 `Time` 类重载 <<，重载后 << 就可以输出 `Time` 类的对象了

`cout << TimeObject`

- 类似地，重载 >> 运算符必须有一个类型为 `istream&` 的左操作数 `cin`

- 所以它也必须是一个 **非成员函数**



性能提示11.1

我们可以把一个运算符作为一个全局、**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`”或者“`get`”函数(即设置数据和读取数据的函数)、调用这些函数的开销会降低性能，因此可以把这些函数写成内联函数以提高性能。



11.4 作为类成员函数和全局函数的运算符函数之比较

- 如果把运算符作为成员函数重载，类的对象必须出现在运算符的左边，这样重载的运算符不具有可交换性
 - 例如，如果我们对复数类提供的重载的加法运算符（作为成员函数）

Complex Complex::operator + (Complex &num) ;

重载的运算符“+”左侧应为Complex类的对象，如

sum = c1 + c2;

sum = c2 + c1;

注意：以上两条语句虽然运算的结果是一样的，但是实现的原理完全不同。



11.5 重载流插入运算符和流读取运算符

- << 与 >>
 - 流读取运算符>>和流插入运算符<<可用来输入输出每一种基本数据类型的数据
 - >> 与 << 实际上已经被系统重载了
 - 程序员可以再次重载这两个运算符
 - 用以输入输出用户自定义类型的数据
 - 重载为: **Global, friend functions**
- 图 11.3 例程
 - 类 **PhoneNumber**
 - 用于保存一个电话号码
 - 按照下列格式输出
(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>
10 #include <string>
11 using namespace std;
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
```

```
4 #include <iomanip>
```

```
5 using std::setw;
```

```
6
```

```
7 #include "PhoneNumber.h"
```

```
8
```

```
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<<
```

`cout << phone;`

将被编译器替换为:

`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 (~~

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

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>>

Input each portion of phone number separately

```
1 // Fig. 11.5: fig11_05.cpp
4 #include <iostream>
7 using namespace std;
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



11.5 重载流插入运算符和流读取运算符

- 流读取运算符函数`operator>>`(第27行)含有两个参数
 - 一个是`istream`类的引用(即程序中的`input`)
 - 另一个则是对用户自定义类型`PhoneNumber`类的引用(即程序中的`number`)
 - 函数返回一个`istream`类的引用。
- 在图11.3的程序中，运算符函数`operator>>`用来把下述格式的电话号码输入到类`PhoneNumber`的对象中：
(800) 555 - 1212



11.5 重载流插入运算符和流读取运算符

- 当编译器遇到main()函数中的表达式:

cin >> phone

编译器将生成函数调用:

operator >> (cin, phone);

```
istream &operator>>( istream &input, PhoneNumber &number ) ;
```

- 当执行该调用时:
 - 引用形参input成为cin的一个别名
 - 引用形参number成为phone的一个别名



11.5 重载流插入运算符和流读取运算符

- 流操纵算子 `setw` 限定了读的字符个数。
 - 与 `cin` 及字符串一起使用时，`setw` 把读入的字符个数限定为其参数指定的字符个数，即 `setw(3)` 允许读入3个字符。
- 通过调用 `istream` 类的成员函数 `ignore`
 - 跳过括号、空格、破折号等字符
 - `ignore` 函数丢弃输入流中指定数目的字符，默认个数为1



11.5 重载流插入运算符和流读取运算符

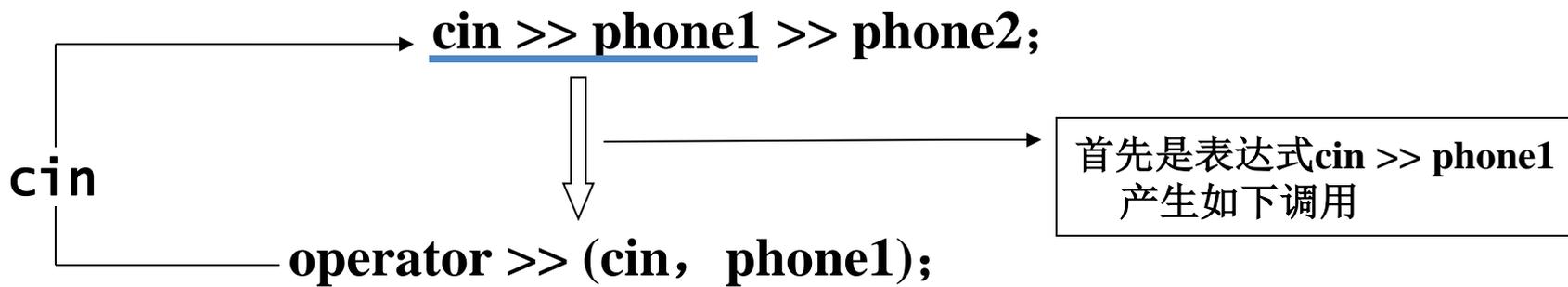
- 函数 `operator>>` 返回对 `istream` 对象的引用 `input`
 - 实际上相当于返回了 `cin`，因为 `cin` 是 `input` 的别名
 - 因而能够在 `PhoneNumber` 对象的输入操作完成后
 - 继续执行对 `PhoneNumber` 的其他对象或者其他数据类型对象的输入操作。
 - 例如：`cin >> phone1 >> phone2 >> x ;`



11.5 重载流插入运算符和流读取运算符

- 例如:

- 可以像下面那样输入两个PhoneNumber对象:



该函数调用的返回值为`cin`，因此表达式的其余部分将被简单地解释为:

`cin >> phone2`



11.5 重载流插入运算符和流读取运算符

- 注意：
 - 函数`operator<<`和`operator>>`在类`PhoneNumber`中被声明为友元函数而不是成员函数。
 - 因为要把类`PhoneNumber`的对象作为运算符的右操作数，所以这些运算符函数必须是全局的友元函数。
- 另外，还要注意：
 - `operator<<` 参数表中引用的`PhoneNumber`是`const`类型
 - 因为只输出`PhoneNumber`
 - `operator>>`参数表中引用的`PhoneNumber`是非`const`类型
 - 由于要在`PhoneNumber`对象中存放输入的电话号码



软件工程知识11.3

无需修改类 `ostream` 和 `istream` 的声明就可以给用户自定义类型添加新的输入/输出能力。

这种方式提高了 C++ 语言的可扩展性，可扩展性是 C++ 的最具吸引力的特点。

11.5 重载流插入运算符和流读取运算符

- 练习：
 - 请为Complex类重载 << 和 >>



11.6 重载一元运算符

- 实现重载运算符的成员函数应为非static
 - 以便访问类的非static数据。
 - 记住，static成员函数只能访问类的static数据成员



11.6 重载一元运算符

- 一元运算符在作用于类的对象时可重载为：
 - 1、一个没有参数的非static成员函数
 - 2、或者带有一个参数的全局函数
 - 参数必须是：
 - 用户自定义类型的对象
 - 或者对该对象的引用



11.6 重载一元运算符

- 本章稍后的例子将要编写一个String类 (11.10)
 - 在String类中我们将要重载 “! ”
 - ! 本身的含义：
 - ! Expression //Expression为普通的变量或表达式
 - 测试Expression的值：
 - 如果为true, 则返回false
 - 如果为false, 则返回true
 - ! 运算符被重载后的含义：
 - ! obj //obj为String类的对象
 - 测试一个字符串是否为空
 - 为空返回 true
 - 不为空返回 false



11.6 重载一元运算符

- 在String类中重载 ! 的两种情况:
 - 如果作为非static的成员函数, 则不需要参数

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

bool String::operator!() const
{
    ...
}
```

- String s;

- !s 相当于 ?



11.6 重载一元运算符

- 在String类中重载 ! 的两种情况:

- 如果作为全局函数, 需要一个参数

```
class String
{
    friend bool operator!(const String &);
    ...
};
```

```
bool operator!( const String & )
{
    ...
}
```

- String s;

- !s 相当于 ?



11.7 重载二元运算符

- 二元运算符在作用于类的对象时可重载为：
 - 带有一个参数的非static成员函数
 - 或者重载为带有两个参数的全局函数
 - 参数之一必须是类的对象或者是对类的对象的引用



11.7 重载二元运算符

- 在String类中重载“<”的两种情况
 - 如果作为非static的成员函数，则需要一个参数

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

– String y, z;

– y < z 相当于

?



11.7 重载二元运算符

- 在String类中重载“<”的两种情况
 - 如果作为全局的非成员函数，则需要两个参数

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

– String y, z;

– y < z 相当于 ?



11.8 实例研究：Array类

- 在C和C++中，基于指针的数组
 - 使用时不检测下标是否超出数组的边界
 - 不能用相等运算符或者关系运算符比较两个数组
 - 数组名可以使用关系运算符比较，但是没有意义
- ```
int x[2],y[2];
if(x > y)
 cout << "ok" << endl;
```
- 不能用赋值运算符把一个数组赋给另一个数组
    - array names are **const** pointers
  - 数组作为一个参数传递给函数时，它的长度必须作为另一参数传递给该函数



## 11.8 实例研究：Array类

- 本节的范例建立了一个具有下列功能的数组类：
  - 它能检测范围以确保数组下标不会越界
  - 允许用赋值运算符把一个数组赋给另外一个数组
  - 数组对象自动知道数组的大小
    - 因而当向函数传递Array数据时，不用将数组的大小传送给函数
  - 可以用流读取运算符和流插入运算符输入输出整个数组
  - 还可以用相等运算符==和!=比较数组
  - 读者还可以增加数组类的其他功能



## 11.8 实例研究：Array类

- 复制构造函数

- 假设我们已经有一个对象 **a**:

**Date a;** //假设Date类有默认构造函数

- 现在要创建一个和对象**a**完全一样的对象**b**，怎么做？

**Date b = a;**

- 如果类Date有**拷贝构造函数**我们就可以用下面语句：

**Date b( a );**

- Date类的拷贝构造函数的原型是怎样的？



## 11.8 实例研究：Array类

- 复制构造函数（Copy constructor）

- 当我们需要拷贝一个对象时，就需要为该对象的类提供这种构造函数：

- 用一个对象的副本初始化另一个同类型的对象
- 下面两条语句的作用是一样的

- ```
Array oldArray;  
Array newArray = oldArray;  
Array newArray( oldArray );
```

- Array类的拷贝构造函数的原形

- ```
Array(const Array &);
```



## 11.8 实例研究：Array类

- 何时执行复制构造函数
  - `Date b = a;`
  - `Date b( a );`
- 实际上：
  - 以上两种形式的作用是相同的，但是实现的原理是不同的
  - 第一种形式需要分两种情况讨论：
    - 如果类Date没有重载赋值运算符或提供拷贝构造函数，则执行默认的逐个成员的赋值
    - 如果类Date重载了赋值运算符或提供了拷贝构造函数，则执行重载后的赋值运算符函数
  - 第二种形式会调用拷贝构造函数
  - 另外，如果既没有重载“=”，有没有编写拷贝构造函数，则上面两条语句都会调用默认复制构造函数，即他们是完全一样的



```

1 // Fig. 11.6: Array.h
3 #ifndef ARRAY_H
4 #define ARRAY_H
6 #include <iostream>
8 using namespace std;
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 bool operator!=(const Array &right) const
24 {
25 return ! (*this == right); // invokes Array::operator==
26 } // end function operator!=
27

```

Most operators overloaded as member functions (except << and >>, which must be global functions)

拷贝构造函数

注意：重载 != 时，用到了已经重载的 ==

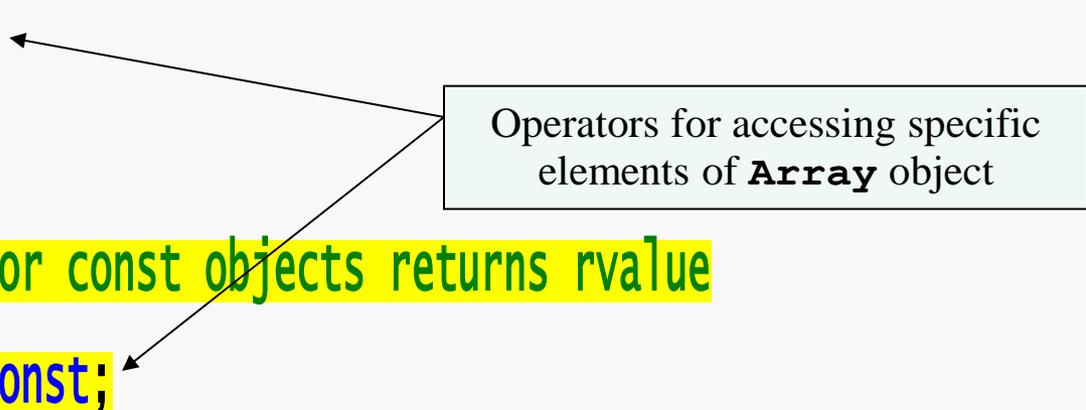
28

29 // subscript operator for non-const objects returns modifiable lvalue

30 int &operator[]( int );

31

Operators for accessing specific elements of **Array** object



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 array

37 }; // end class Array

38

39 #endif

```
1 // Fig 11.7: Array.cpp
2 // Member-function definitions for class Array
3 #include <iostream>
9 #include <iomanip>
12 #include <cstdlib> // exit function prototype
13 using namespace std;
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;

29 Array::Array( const Array &arrayToCopy )

30 : size( arrayToCopy.size )

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 ]; // c

36 } // end Array copy constructor

37

39 Array::~~Array()

40 {

41 delete [] ptr; // release pointer-based array space

42 } // end destructor

43

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

```
49
50 // overloaded assignment operator;
51 // const return avoids: (a1 = a2) = a3
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=
```

Want to avoid self assignment

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

```

71
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
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[]

```

integers1[ 5 ] calls  
integers1.operator[]( 5 )

```
100
103 int 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
118 istream &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
```

```
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;
```

Use overloaded << operator to output

```
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;
```

Use overloaded != operator to test for inequality

```
41 Array integers3(integers1); // invokes copy constructor
```

Use 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
```

Use overloaded = operator to assign

```
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 test for equality

```
56 if (integers1 == integers2)
```

```
57 cout << "integers1 and integers2 are equal" << endl;
```

```
58 // use overloaded subscript operator to create rvalue
```

```
59 cout << "\nintegers1[5] is " << integers1[5];
```

```
60 // use overloaded subscript operator to create lvalue
```

```
61 cout << "\n\nAssigning 1000 to integers1[5]" << endl;
```

```
62 integers1[5] = 1000;
```

```
63 cout << "integers1:\n" << integers1;
```

```
64 // attempt to use out-of-range subscript
```

```
65 cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
```

```
66 integers1[15] = 1000; // ERROR: out of range
```

```
67 return 0;
```

```
68 } // end main
```

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 |   |   |   |

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

## 常见程错误11.6

---

注意复制构造函数要按引用调用，而不是按值调用，否则复制构造函数调用会造成无穷递归(这是个致命逻辑错误)，因为对于按值调用，建立传入复制构造函数的对象副本会造成复制构造函数的递归调用。

查看Array的拷贝构造函数，如果把参数中的引用去掉，分析下面语句的执行情况：

```
Array newArray(oldArray);
```



# Array newArray( oldArray );

```
29 Array::Array(const Array arrayToCopy)
30 : size(arrayToCopy.size)
31 {
32 ptr = new int[size];
33
34 for (int i = 0; i < size; i++)
35 ptr[i] = arrayToCopy.ptr[i];
36 } // end Array copy constructor
```

# Array arrayToCopy( oldArray );

在执行拷贝构造函数之前，会传递参数，因为传递的是副本，所以传递参数时会再次调动拷贝构造函数。



## 常见编程错误11.7

如果构造函数简单地将源对象的指针复制到目标对象的指针，则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，从而导致另外一个对象的Ptr没有定义(虚悬指针)，这种情况可能会引起严重的运行时错误。

**考虑如果没有为Array类编写拷贝构造函数，而是使用默认的拷贝构造函数。**

**array1 = array2; 会出现什么情况？**



## 软件工程知识11.5

---

通常要把构造函数、析构函数、重载的赋值运算符以及复制构造函数一起提供给使用**动态内存分配**的类。

## 11.8 实例研究: Array类

- 第16行:

```
const Array &operator= (const Array &);
```

声明了重载的赋值运算符函数。

- 成员函数 `operator=` 测试了这种赋值是否是自我赋值。
- 然后, 测试两个数组长度是否相同
  - 如果是, 则左边Array对象的原始整数数组不重新分配。
  - 否则成员函数`operator=`用`delete`释放目标数组原先动态分配的空间, 将源数组的数据成员`size`复制到目标数组的`size`, 用`new`重新分配目标数组所需的空間并将`new`返回的指针赋给数组的`Ptr`成员
- 成员函数都返回当前对象(即`*this`), 这种处理方式允许诸如`x=y=z`这样的连续赋值



# 常见编程错误11.8

---

当类的对象包含指向动态分配的内存的指针时，如果没有为它提供重载的赋值运算符和复制构造函数则会造成逻辑错误。

# 11.9 类型转换

## • 类型转换

- 对于内部的类型，编译器知道如何转换类型，或用户自己强制转换。
- 但是怎样转换用户自定义类型呢？

## • 这种转换可以用**转换构造函数**实现

- 它是一个可以接收**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。
- 本章稍后（11.10中）要介绍用一个转换构造函数把正常的char\*类型的字符串转换为类String的对象。
- 本节先介绍另一种类型转换方法：**转换运算符**



# 11.9 类型转换

- **转换运算符函数**(也称为强制类型转换运算符)
  - 把某一类的对象转换成另一类的对象，或者转换成基本类型的对象
  - 要求：这种运算符函数必须是一个非static成员函数，而不能是友元函数。
  - 例如，函数原型：  
`A::operator char*() const;`  
上面声明了一个重载的强制类型转换运算符函数，它根据用户自定义类型A的对象转换成一个临时的char\*类型的对象
  - 重载的强制类型转换运算符函数不能指定返回类型
    - 因为这种函数有默认返回类型
    - 返回类型是要转换后的对象类型



## 11.9 类型转换

- 转换运算符

- 可以把一种类的对象转换为其他类的对象
- 或把一种类的对象转换为基本类型的变量

- 例如:

```
A::operator int() const;
```

```
A::operator OtherClass() const;
```



## 11.9 类型转换

- 优点:

- 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。
- 假设我们为类定义了下面的函数

```
String::operator char *() const;
```

即，类 **String** 可以被转化为 **char \***

- 当出现下列语句时:

```
cout << s; // s is a String
```

- 编译器会自动地调用上面定义的函数将s 转化为 **char \***类型
- 这样我们就不需要对类**String**重载 运算符 << 了



## 11.9 类型转换

- 举例:

```
class Complex
{
public:
 ...
 operator int() const
 {
 return this->realPart + this->imaginaryPart;
 }
 ...
};
```



# 11.10 实例研究：String类

- 图 11.9 创建了String类
  - String类可以用来生成和操作字符串
- 转换构造函数
  - 只有一个参数，类型为 **char \***
  - 它能将其他类的对象转换为**自身类**的对象
    - Example
      - `String s1( "happy" );`
        - 从 `char *` 生成一个String



```

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 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 \***

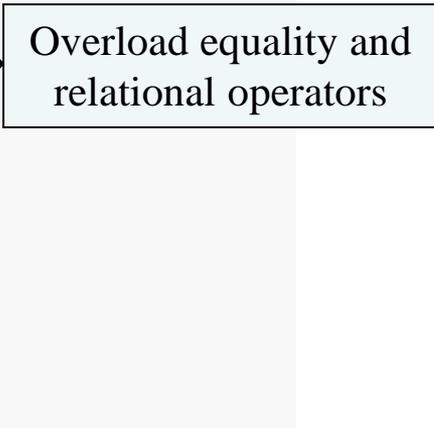
**s1 += s2** will be interpreted as **s1.operator+=(s2)**

Can also concatenate a **String** and a **char \*** because the compiler will cast the **char \*** argument to a **String**



```
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>=
```

Overload equality and relational operators



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

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

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



```
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
```



```
29 // copy constructor
30 String::String(const String ©)
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[]
107
108 // return reference to character in String as rvalue
109 char 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
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>>
```



```
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
```

Use overloaded stream insertion operator for **Strings**

Use overloaded equality and relational operators for **Strings**



Use overloaded negation operator for **Strings**

Use overloaded assignment operator for **Strings**

Use overloaded addition assignment operator for **Strings**

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

Use overloaded function call operator for **Strings**

```

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";

```



```
61 // test assignment (=) operator with self-assignment
62 cout << "assigning *s4Ptr to *s4Ptr" << endl;
63 *s4Ptr = *s4Ptr; // test overloaded assignment
64 cout << "*s4Ptr = " << *s4Ptr << endl;
65
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 **Strings**

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

```

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

```

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

```

*(continued at top of next 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
```



# 软件工程知识11.8

当使用转换构造函数实现隐式转换时，C++只会使用一个隐式的构造函数调用来试图满足重载赋值运算符的需要。

通过执行一系列隐式的、用户自定义的类型转换来满足重载运算符的需要是不可能的。

即，编译器不能连续的自动转换很多次。

## 性能提示11.2

与先执行隐式类型转换然后再执行连接操作相比，使重载的连接运算符+=只有一个const char\*类型参数的执行效率更高。

```
String s1("yes"), s2("no");
s1 = s1 + s2;
```

```
String s1("yes");
s1 += "no"; //参考String类的第45行代码
思考：s1与“no”能相加吗？他们类型不一致！
```

```
s1 += "no";
```

```
char * sptr="no";
s1 += sptr;
```

```
s1.operator+=(sptr);
```

```
s1.operator+=(String(sptr));
```

```
char *sptr = "no";
{
 String temp(sptr);
 s1.operator+=(temp);
}
```

sptr的类型与operator+=的参数类型不一致  
编译器会为他们自动调用转换构造函数，完成类型的自动转换



## 软件工程知识11.10

---

通过用前面定义的成员函数实现新的成员函数，  
可以有效的复用代码，从而减少要编写的代码量。

## 11.11 重载++和--运算符

- 本节介绍编译器如何识别前置和后置的自增及自减运算符
- 要重载既能允许前置又能允许后置的自增运算符
  - 每个重载的运算符函数必须有一个明确的特征以使编译器能确定要使用的++版本
  - 前置++ 和 后置++ 是两个不同的运算符，因此重载运算符函数也应该不相同



# 11.11 重载++和--运算符

## • 重载 前置++

- 重载前置++的方法与重载其他前置一元运算符一样。
- 假设我们要对Date重载 前置++，可以采用两种方式：
  - 作为成员函数时，重载运算符函数的原形为：
    - `Date &operator++()`;
    - `++d1` 相当于 `d1.operator++()`
  - 作为友员函数时，重载运算符函数的原形为：
    - `friend Date &operator++( Date & );`
    - `++d1` 相当于 `operator++( d1 )`



# 11.11 重载++和--运算符

## • 重载 后置++

- 重载后置自增运算符必须有别于前置运算符，区别在于参数表：
  - An `int` with value `0`
- 作为成员函数时，重载运算符函数的原形为：
  - `Date operator++( int );`
  - `d1++` 相当于 `d1.operator++( 0 )`
- 作为友员函数时，重载运算符函数的原形为：
  - `Date operator++( Date &, int );`
  - `d1++` 相当于 `operator++( d1, 0 )`



## 11.11 重载++和--运算符

- 注意：
  - 后置的自增运算符按值返回Date对象
  - 而前置的自增运算符按引用返回Date对象。



## 11.12 实例研究：Date类

- 举例 **Date class** 对重载运算符的应用
  - 重载了自增运算符
    - 用于增加 **day**
  - 重载了 **+=** 运算符



## 11.12 实例研究：Date类

- 类Date的Public接口提供了以下成员函数：
  - 一个重载的流插入运算符
  - 一个默认的构造函数
  - 一个setDate函数
  - 一个重载的前置自增运算符函数
  - 一个重载的后置自增运算符函数
  - 一个重载的加法赋值运算符(+ =)
  - 一个检测闰年的函数
  - 一个判断是否为每月最后一天的函数。



# 11.12 实例研究: Date类

```
class Date
```

```
{
```

```
.....
```

```
Date &operator++();
```

```
void setDate();
```

```
int getYear();
```

```
....;
```

```
int year;
```

```
int month;
```

```
int day;
```

```
};
```

```
void Date::setDate()
```

```
{
```

设置year、  
month和day的值

```
}
```

```
int Date::getYear()
```

```
{
```

取year的值

```
}
```



```

1 // Fig. 11.12: Date.h
3 #ifndef DATE_H
4 #define DATE_H
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,
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
3 #include <iostream>
4 #include "Date.h"
5
7 const int Date::days[] =
8 { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
9
11 Date::Date(int m, int d, int y)
12 {
13 setDate(m, d, y);
14 } // end Date constructor
15
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
```

```

30 Date &Date::operator++()
31 {
32 helpIncrement(); // increment date
33 return *this; // reference return to create an lvalue
34 } // end function operator++
35
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
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+=

```

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()
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<<
```

```

1 // Fig. 11.14: fig11_14.cpp
2 // Date class test program.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Date.h" // Date class definition
8
9 int main()
10 {
11 Date d1; // defaults to January 1, 1900
12 Date d2(12, 27, 1992); // December 27, 1992
13 Date d3(0, 99, 8045); // invalid date
14
15 cout << "d1 is " << d1 << "\nd2 is " << d2 << "\nd3 is " << d3;
16 cout << "\n\n d2 += 7 is " << (d2 += 7);
17
18 d3.setDate(2, 28, 1992);
19 cout << "\n\n d3 is " << d3;
20 cout << "\n++d3 is " << ++d3 << " (leap year allows 29th)";
21
22 Date d4(7, 13, 2002);
23
24 cout << "\n\nTesting the prefix increment operator:\n"
25 << " d4 is " << d4 << endl;
26 cout << "++d4 is " << ++d4 << endl;
27 cout << " d4 is " << d4;
28

```

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
```

# 11.14 explicit构造函数

- 隐式转换

- 任何单参数的构造函数都可以被编译器用来执行隐式转换
- 有时，隐式转换是不受欢迎的，或者会导致错误
  - 关键字 **explicit**
    - 禁止由转换构造函数完成的隐式转换



```
1 // Fig. 11.16: Fig11_16.cpp
2 // Driver for simple class Array.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Array.h"
8
9 void outputArray(const Array &); // prototype
10
11 int main()
12 {
13 Array integers1(7); // 7-element array
14 outputArray(integers1); // output Array integers1
15
16 return 0;
17 } // end main
```

```
18
19 // print Array contents
20 void outputArray(const Array &arrayToOutput)
21 {
22 cout << "The Array received has " << arrayToOutput.getSize()
23 << " elements. The contents are:\n" << arrayToOutput << endl;
24 } // end outputArray
```

```
The Array received has 7 elements. The contents are:
 0 0 0 0
 0 0 0
The Array received has 3 elements. The contents are:
 0 0 0
```



```

1 // Fig. 11.17: 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 explicit 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

```

Use **explicit** keyword to avoid implicit conversions when inappropriate

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!=

28

29 // subscript operator for non-const objects returns lvalue

30 int &operator[]( int );

31

32 // subscript operator for const objects returns rvalue

33 const int &operator[]( int ) const;

34 private:

35 int size; // pointer-based array size

36 int \*ptr; // pointer to first element of pointer-based array

37 }; // end class Array

38

39 #endif

# 常见的编程错误11.10

---

试图调用 **explicit constructor** 以进行隐式转换会导致编译错误。

# 常见的编程错误11.11

---

关键字**explicit**只能用在单参数的构造函数前，如果用在数据成员或者其他成员函数前，则会导致编译错误。



```
1 // Fig. 11.18: Fig11_18.cpp
2 // Driver for simple class Array.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Array.h"
8
9 void outputArray(const Array &); // prototype
10
11 int main()
12 {
13 Array integers1(7); // 7-element array
14 outputArray(integers1); // output Array integers1
15 outputArray(3); // convert 3 to an Array and output Array's contents
16 outputArray(Array(3)); // explicit single-argument constructor call
17 return 0;
18 } // end main
```

Using keyword **explicit** on the conversion constructor disallows this line to erroneously call the conversion constructor

An explicit call to the conversion constructor is still allowed

19

20 // print array contents

21 void outputArray( const Array &arrayToOutput )

22 {

23 cout << "The Array received has " << arrayToOutput.getSize()

24 << " elements. The contents are:\n" << arrayToOutput << endl;

25 } // end outputArray

c:\cpphttp5\_examples\ch11\Fig11\_17\_18\Fig11\_18.cpp(15) : error C2664:  
'outputArray' : cannot convert parameter 1 from 'int' to 'const Array &'  
Reason: cannot convert from 'int' to 'const Array'  
Constructor for class 'Array' is declared 'explicit'



## 错误预防技巧11.3

---

对于那些不应该被编译器用来执行隐式转换的单参数构造函数，请务必记住使用关键字 **explicit**.