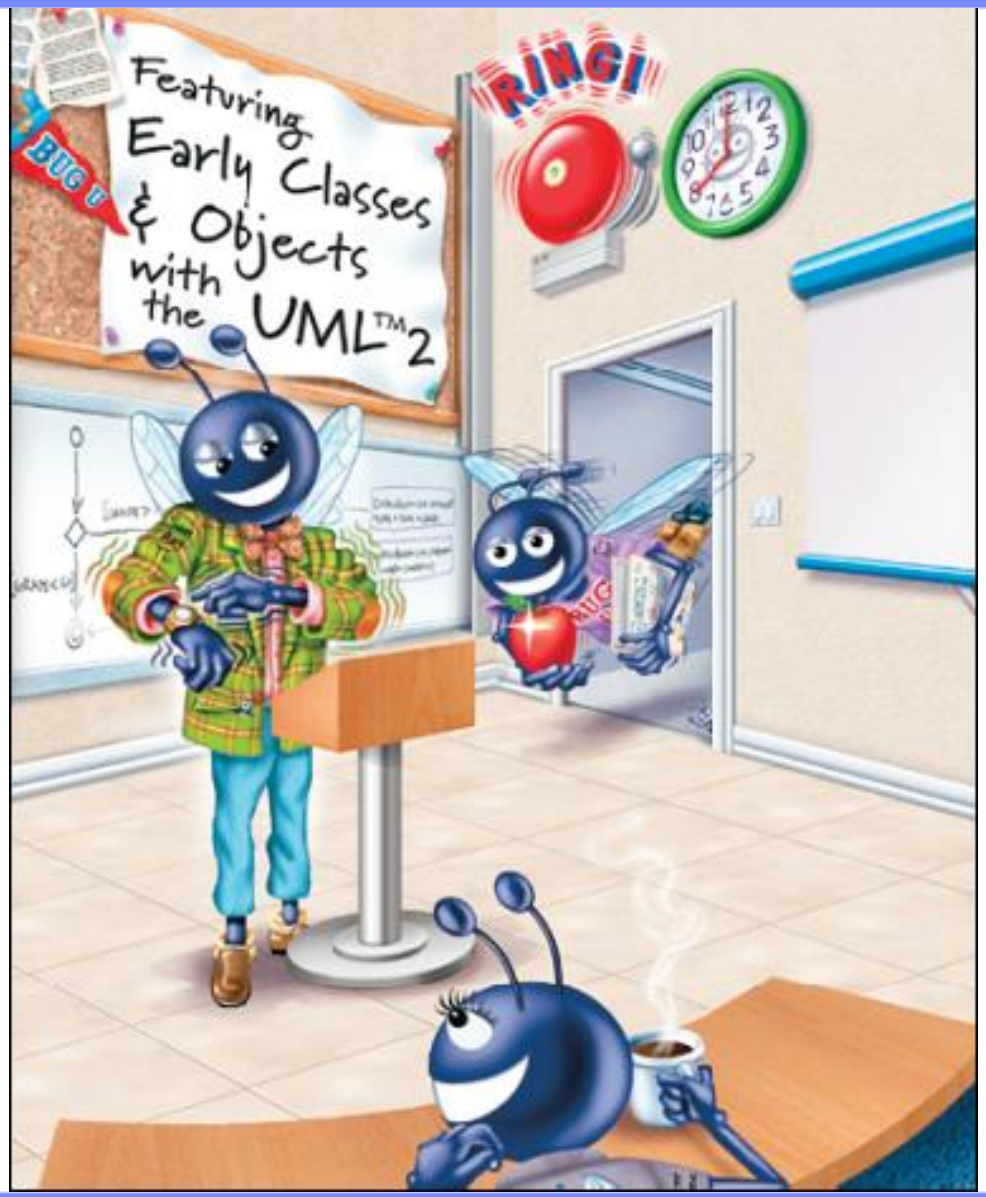


Lecture 6:
类的深入剖析(1)

第九讲 类的深入剖析（I）

学习目标：

- 类成员的访问
- 访问函数和工具函数
- 析构函数
- 默认赋值函数

0 关于#include的问题

- Preprocessor wrappers
  - If the header has been included previously(如果头文件已经被包含过)
    - ✓ 同一个变量或标志符被多次声明，引起编译错误
  - Prevents multiple-definition errors（采用下面方法避免）
    - ✓ `#ifndef` – “if not defined”
      - Skip this code if it has been included already
    - ✓ `#define`
      - Define a name so this code will not be included again
    - ✓ `#endif`
  - Example
    - ✓
```
#ifndef TIME_H
#define TIME_H
… // code
#endif
```

```
1  // Fig. 9.1: Time.h
2  // Declaration of class Time.
3  // Member functions are defined in Time.cpp
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME_H
7  #define TIME_H
8
9  // Time class definition
10 class Time
11 {
12 public:
13    Time(); // constructor
14    void setTime( int, int, int ); // set hour, minute and second
15    void printUniversal(); // print time in universal-time format
16    void printStandard(); // print time in standard-time format
17 private:
18    int hour; // 0 - 23 (24-hour clock format)
19    int minute; // 0 - 59
20    int second; // 0 - 59
21 }; // end class Time
22
23 #endif
```

Preprocessor directive `#ifndef` determines whether a name is defined

Preprocessor directive `#define` defines a name (e.g., `TIME_H`),就如同钓鱼的浮标，起着指示的作用。

它是一个标志符！

函数原型

私有数据成员，
现在不能初始化

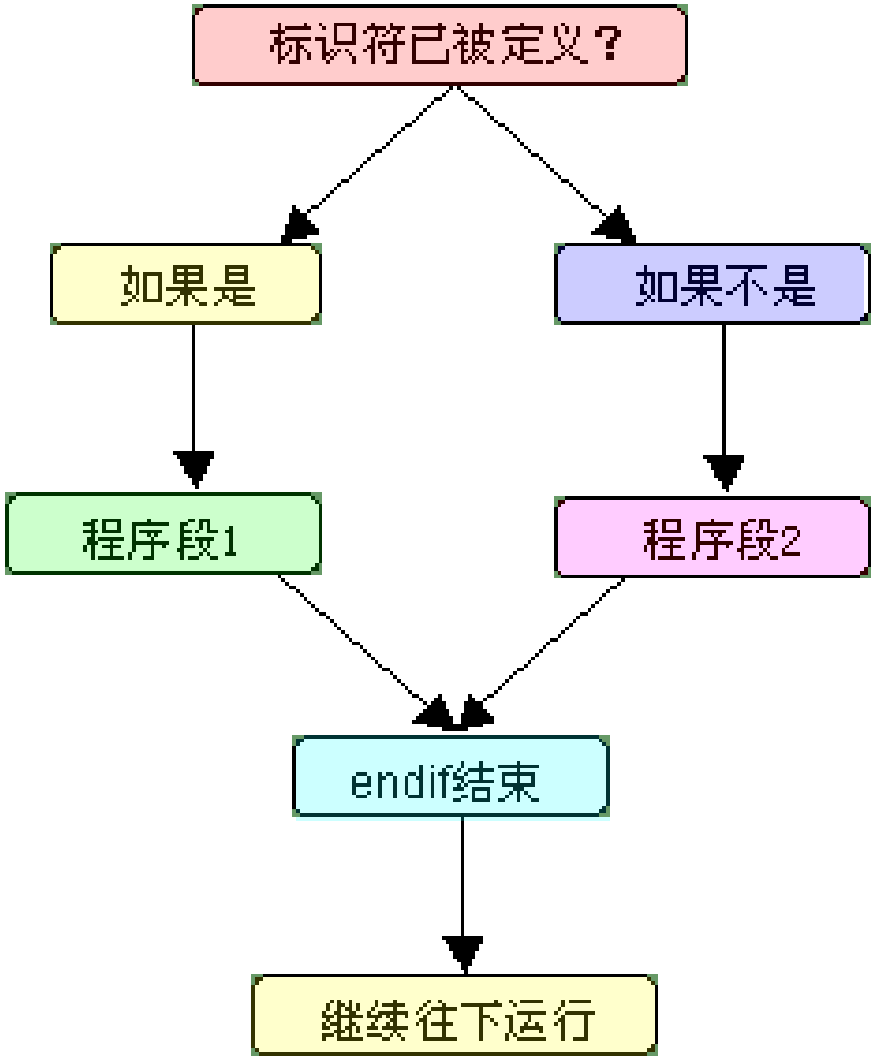
Preprocessor directive `#endif` marks the end of the code that should not be included multiple times

Line 6～23 定义了类

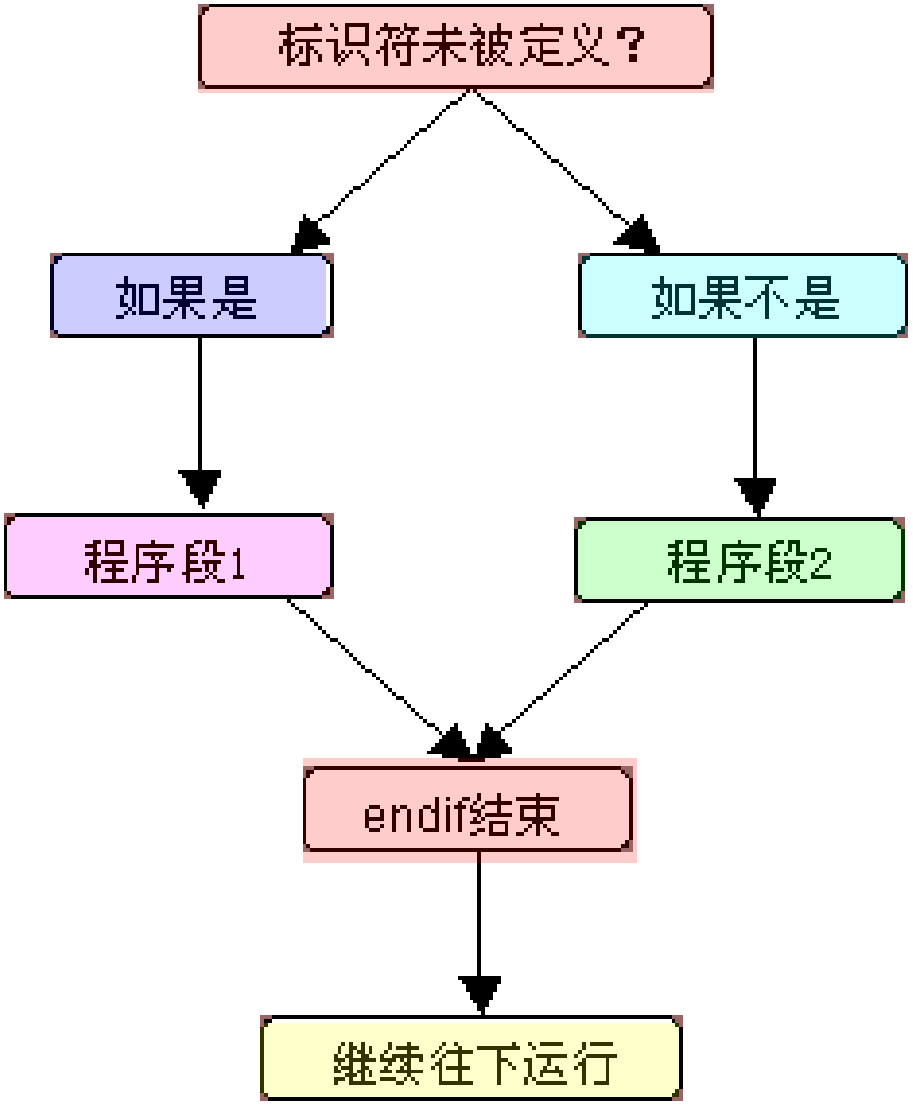
```
6  #ifndef TIME_H
7  #define TIME_H
.....
23 #endif
```

- 建立大程序时，其他定义和声明也放在头文件中。上述预处理指令使得如果发现已经定义了TIME_H名字时，就不再包含#ifndef和#endif之间的代码。如果文件中原先没有包含头文件，则TIME_H名字显然也没有被定义过，就继续执行下面的定义。
- 多次包含头文件语句通常发生在大程序中，有时头文件A本身已经包含其他头文件B、C等，则定义A后，B与C就不需要再次定义。
- 注意：预处理指令中符号化常量名使用的一般规则是：
  - 头文件名用大写形式
  - 头文件名中圆点(．)换成下划线。

- #ifdef 标识符
  程序段1
  #else
  程序段2
  #endif



- #ifndef 标识符
  程序段1
  #else
  程序段2
  #endif



2. 内联函数

- 方式一：程序设计的最初阶段是程序从上写到下
  - 优点：没有函数调用，没有程序执行控制权的转移，执行效率较高
  - 缺点：重复实现相同功能的模块（如：求最大值模块）；如果需要修改就要修改多处
- 方式二：C程序开始以函数调用的形式进行堆砌
  - 优点：按功能进行模块划分，条理清晰，重复实现的功能函数只写一份，修改简单；
  - 缺点：函数间的控制权转移、函数环境保存、参数传递等额外工作使执行效率下降
- 方式三：有没有同时具有方式一与方式二优点的程序实现方法？

C++提供了一种提高效率的方法，即在编译时将所用函数的代码嵌入到主调用函数中，这种被嵌入到主调用函数中的函数称为内联函数（inline function）。

指定内联函数的方法很简单，方法之一是在函数首行的左端加一个关键字inline即可。

- 内联函数不在调用时发生控制转移，
- 只是在编译时将函数体嵌入到每一个调用语句处。这样就节省了参数传递、控制转移等开销。

内嵌函数的缺点： 虽然程序执行效率提高了，但可执行程序规模变大了。实际应用中要权衡效率和规模之间的关系。

例： 定义求三个数的最大数的函数，并将该函数指定为内联函数.

```
#include<iostream>
using std::cout ;
using std::endl ;
inline int max(int a, int b, int c)
{    if(b>a)   a=b;
     if(c>a)   a=c;
     return  a;
}
 int main( )
{  cout<<" max=" << max(10,20,30) << endl ;
}
```

由于在定义函数时，指定它为内联函数，因此，编译系统在遇到函数调用max(10,20,30)时，就用max( )函数体内的代码代替max(10,20,30)，同时将实参代替形参。

说明:

1. 内联函数告诉编译器，用函数体代码替换源代码中的每一个内联函数调用。

2. 内联函数执行较快，因为处理器时间没有浪费在函数的跳转、执行和值的返回上。但内联函数增加了目标程序的长度，因此，inline限定符应该只用于经常使用的小函数。

- 设定内联函数的方法
  - 在类定义体内实现的成员函数
  - 显式使用关键字inline指定定义在类定义体外的成员函数
    - 关键字inline必须与函数定义体放在一起才能使函数成为内联，仅将inline放在函数声明前面不起任何作用

```
class A
{
public:
    void Foo(int x, int y)
        { … }
        // 自动地成为内联函数
};
```

```
class A
{
public:
        void Foo(int x, int y);
};

inline void A::Foo(int x, int y)
{…}
```

- 类的成员函数可以在类定义中声明，同时加以实现
- 类的成员函数也可以在类定义中声明，在类定义体外加以实现
  - 尝试定义为**内联方式**是一种有效的方法

软件工程知识：通过函数原型在类定义中声明成员函数，在类定义之外实现这些函数，可以区分类的接口与实现方法。这样可以实现良好的软件工程。

常见编程错误：在类定义外部实现类的成员函数时，忽略函数名中类名和作用域运算符是错误的。
正确方法：**类名::函数名(参数列表)**

性能提示：在类定义中定义小的成员函数将自动使该成员函数成为内联函数（一般编译器都是如此处理），可以提高运行的性能。

软件工程知识：在类定义的内部实现小的成员函数并不能提高软件的通用性，因为使用该类的客户程序能够看到函数的实现方法，这个实现方法随同客户程序一起进行编译，如果该小函数的实现发生了改变，客户程序也就必须重新编译。

```
1  // Fig. 9.2: Time.cpp
2  // Member-function definitions for class Time.
3  #include <iostream>
4  using std::cout;
5
6  #include <iomanip>
7  using std::setfill;
8  using std::setw;
9
10 #include "Time.h" // include definition of class Time from Time.h
11
12 // Time constructor initializes each data member to zero.
13 // Ensures all Time objects start in a consistent state.
14 Time::Time()
15 {
16    hour = minute = second = 0;
17 } // end Time constructor
18
19 // set new Time value using universal time; ensure that
20 // the data remains consistent by setting invalid values to zero
21 void Time::setTime( int h, int m, int s )
22 {
23    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
24    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
25    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
26 } // end function setTime
```

构造函数，初始化各变量为0

Ensure that **hour**, **minute** and **second** values remain valid

```
27
28 // print Time in universal-time format (HH:MM:SS)
29 void Time::printUniversal()
30 {
31    cout << setfill( '0' ) << setw( 2 ) << hour << ":"
32       << setw( 2 ) << minute << ":" << setw( 2 ) << second;
33 } // end function printUniversal
34
35 // print Time in standard-time format (HH:MM:SS AM or PM)
36 void Time::printStandard()
37 {
38    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << ":"
39       << setfill( '0' ) << setw( 2 ) << minute << ":" << setw( 2 )
40       << second << ( hour < 12 ? " AM" : " PM" );
41 } // end function printStandard
```

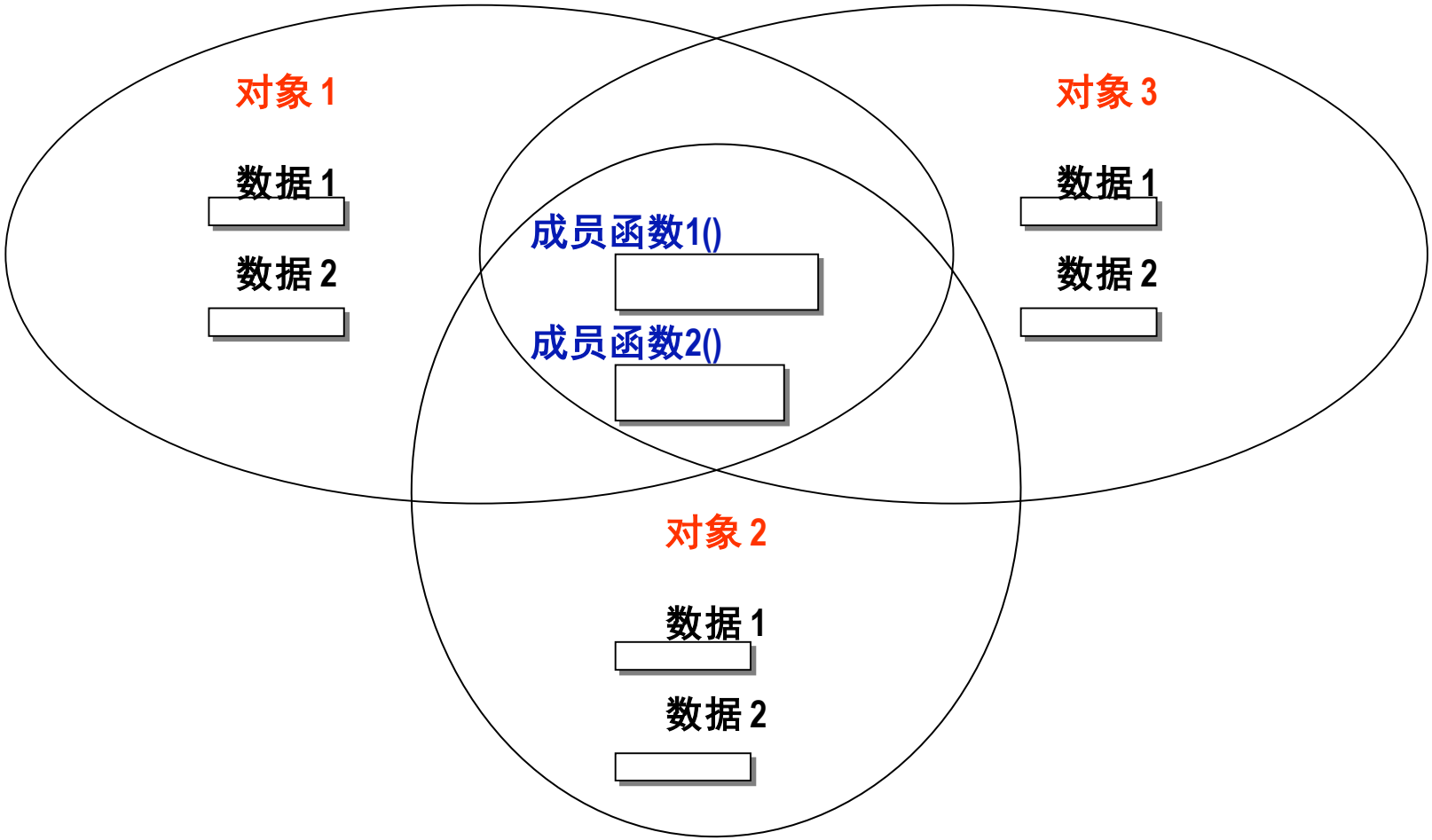
Using **setfill** stream manipulator to specify a fill character

- 类的成员函数在类定义中声明原型，在类定义外实现
- 类的数据成员不能在类中声明时进行初始化。只能由构造函数或相应的成员函数（如set函数）进行赋值。

```
6 #include <iomanip>
7 using std::setfill;
8 using std::setw;
```

- 主要是对cin,cout之类的一些操纵运算子，比如setfill, setw, setbase, setprecision等。它是I/O流控制头文件,就像C里面的格式化输出一样.
- “粘性”——一旦设定，就会对后面一直起作用，直到你重新设置。
  - *setfill*(char):设置填充字符 ,默认的是空格
  - *setprecision*(int):控制输出流显示浮点数的数字个数
- “非粘性”—— 只对紧接着的显示的值起作用
  - setw():设置显示宽度

- 类中定义了数据成员和成员函数
- 类实例化成对象后，每个对象对类中的数据成员都有其自己的副本
- 但该类的所有对象都使用同一份成员函数副本
- 因此，对象只包含数据成员。
  - sizeof(某类名)=sizeof(某类任一对象)
    =某类定义中数据成员字节之和


对象 1
数据 1
数据 2
成员函数1()
成员函数2()
对象 3
数据 1
数据 2
对象 2
数据 1
数据 2

- Using class Time
  - 可以通过以下方式访问类成员
    - ✓ Time sunset;
    - ✓ Time arrayOfTimes[5];
    - ✓ Time &dinnerTime = sunset;
    - ✓ Time *timePtr = &dinnerTime;

    能够写成：Time *timePtr = &Time; ？ ？

```
1 // Fig. 9.3: fig09_03.cpp
2 // Program to test class Time.
3 // NOTE: This file must be compiled with Time.cpp.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Time.h" // include definition of class Time from Time.h
9
10 int main()
11 {
12    Time t; // instantiate object t of class Time, 自动调用构造函数
13
14    // output Time object t's initial values
15    cout << "The initial universal time is ";
16    t.printUniversal(); // 00:00:00
17    cout << "\nThe initial standard time is ";
18    t.printStandard(); // 12:00:00 AM
19
20    t.setTime( 13, 27, 6 ); // change time
21
22    // output Time object t's new values
23    cout << "\n\nUniversal time after setTime is ";
24    t.printUniversal(); // 13:27:06
25    cout << "\nStandard time after setTime is ";
26    t.printStandard(); // 1:27:06 PM
27
28    t.setTime( 99, 99, 99 ); // attempt invalid settings
```

```
16    hour = minute = second = 0;
```

```
23    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
24    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
25    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
```

```
29
30    // output t's values after specifying invalid values
31    cout << "\n\nAfter attempting invalid settings:"
32       << "\nUniversal time: ";
33    t.printUniversal(); // 00:00:00
34    cout << "\nStandard time: ";
35    t.printStandard(); // 12:00:00 AM
36    cout << endl;
37    return 0;
38 } // end main
```

```
The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM
```

性能提示：对象只包含数据。对类名或该类的对象采用sizeof运算符时，只能得到类的数据的长度。编译器生成独立于所有类的对象的成员函数的副本(只有一份)。类的所有对象共享这个成员函数唯一副本。当然，每个对象都需要自己的类数据副本，因为对象中的数据是不同的。

2 Class Scope and Accessing Class Members

- 类作用域包括
  - 数据成员作用域
    - ✓在类定义中声明的变量
  - 成员函数作用域
    - ✓在类定义中声明的函数

2 Class Scope and Accessing Class Members

- 在类作用域内
  - 类成员可以被所有成员函数访问
- 在类作用域外
  - public 类成员可以通过句柄来引用。句柄有：
    - ✓ 对象名
    - ✓ 对象引用
    - ✓ 对象指针
  - private类成员则不能被直接访问

2 Class Scope and Accessing Class Members

- 成员函数中声明的变量
  - 具有块作用域
  - 仅在该函数中有效
- 隐藏的类作用域变量
  - 如果在成员函数中定义一个与类的数据成员同名的变量
  - 在成员函数中可以通过作用域运算符(::)来访问被隐藏的数据成员

单目作用域运算符的用法

```
#include <iostream>
using namespace std;
int value = 1;
int main( )
{    int  value=5;
     cout <<"value (local)  ="<<    value <<endl;
     cout <<"value (global)="<< ::value<<endl;
}
```

运行结果：
value (local)=5
value (global)=1

2 Class Scope and Accessing Class Members

- Dot member selection operator (.)
  - 在句柄为对象名或是对象引用时使用
- Arrow member selection operator (->)
  - 在句柄为对象指针时使用

1. 一般对象的成员表示：

<对象名>.<数据成员名>　//表示数据成员；

或

<对象名>.<成员函数名>（<参数表>）//表示成员函数；

这里．是成员运算符，该运算符的功能是表示对象的成员。

t.setTime (13,27,6)表示 Time类的t对象的成员函数

```
//tdate.h
class tdate
{
 public:
  void set(int,int,int);
  int isleapyear( );
  void print( );
 private:
  int month;
  int day;
  int year;
};
```

```
//tdate.cpp
#include “iostream.h”
#include “tdate.h”
void tdate::set(int m,int d,int y)
{month=m;
 day=d;
 year=y;
}

int tdate::isleapyear( )
{return
     (year%4==0&&year%100!=0)||(year%400
     ==0);
}

void tdate::print( )
{cout<<month<<”/”<<day<<”/”<<year<<endl;}
```

```
#include <iostream.h>
#include “tdate.h”

void func( )
{
 month=10;          //error：非成员函数不能访问私有成员

 tdate::month=10;   //error：未声明对象，不能通过类访问成员

tdate::set(1,15,1998);
}                   //error：未声明对象，不能通过类访问成员函数
```

```
#include <iostream.h>
#include “tdate.h”
void func( )
{
 tdate oneday;
   oneday.set(2,15,1998);
   oneday.print( );
}
```

2. 指向对象的指针的成员表示：

〈对象指针名〉➡〈数据成员名〉

或

<对象指针名>→<成员函数名>（<参数表>）

这里，→是指向运算符，它与·运算符的区别是：

→用来表示指向对象的指针的成员，

·用来表示一般对象的成员。

```
tdate s , *pdate;

pdate = &s;

pdate →year  // 表示对象指针pdate的year 数据成员;
```

```
pdate→setdate(int y, int m, int d)

// 表示对象指针pdate的setdate( ) 成员函数。
```

```
#include <iostream.h>
#include “tdate.h”
void somefunc( tdate *ps)
{
 ps->print( );
 if(ps->isleapyear( ))
   cout<<“leap year\n”;
 else
   cout<<“not leap year\n”;
}
int main( )
{
 tdate s;
 s.set(2,15,1998);   // 采用 . 操作
 somefunc( &s );     // 采用 -> 操作
}
```

```
#include <iostream.h>
#include "tdata.h"
void somefunc ( tdate &refs)
{
 refs.print( );
 if(refs.isleapyear( ))
  cout<<"yes\n";
 else
  cout<<"no\n";
}
int main( )
{
 tdate s;
 s.set(2,15,1998);
 somefunc( s );
}
```

引用对象的成员表示与一般对象的成员表示相同。

```
1  // Fig. 9.4: fig09_04.cpp
2  // Demonstrating the class member access operators . and ->
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // class Count definition
8  class Count
9  {
10 public: // public data is dangerous
11    // sets the value of private data member x
12    void setX( int value )
13    {
14       x = value;
15    } // end function setX
16
17    // prints the value of private data member x
18    void print()
19    {
20       cout << x << endl;
21    } // end function print
22
23 private:
24    int x;
25 }; // end class Count
```

```
26
27 int main()
28 {
29    Count counter; // create counter object
30    Count *counterPtr = &counter; // create pointer to counter
31    Count &counterRef = counter; // create reference to counter
32
33    cout << "Set x to 1
34    counter.setX( 1 ); // set data member x to 1
35    counter.print(); // call member function print
36
37    cout << "Set x to 2 and
38    counterRef.setX( 2 ); // set data member x to 2
39    counterRef.print(); // call member function print
40
41    cout << "Set x to 3 and
42    counterPtr->setX( 3 ); // set data member x to 3
43    counterPtr->print(); // call member function print
44    return 0;
45 } // end main
```

Using the dot member selection operator with an object

Using the dot member selection operator with a reference

Using the arrow member selection operator with a pointer

```
Set x to 1 and print using the object's name: 1
Set x to 2 and print using a reference to an object: 2
Set x to 3 and print using a pointer to an object: 3
```

3 Separating Interface from Implementation

- 将类定义与类成员函数实现相分离的优点
  - 使得程序易于修改
    - ✓只要类的接口保持不变，改变类的实现将不会影响到客户
  - 头文件包含部分实现内容和相关提示
    - ✓内联函数需要定义在头文件中
    - ✓私有成员在头文件类定义中出现

- 类的客户使用类时不需要访问类的源代码，但客户需要连接类的目标码（类编译之后的版本）。
- 对类接口很重要的信息应放在头文件中。只在类内部使用而类的客户不需要的信息应放在不发表的源文件中。这是最低权限原则的又一个例子。

3 Separating Interface from Implementation

软件工程知识：类客户端如果要使用类，不需要访问类的源代码。但是，客户端需要连接类的对象代码（即编译后的类）。这有利于由独立软件供应商(ISV)提供类库进行销售或发放许可证。ISV在自己的产品中只提供头文件和目标模块，不提供专属信息（例如源代码）。

软件工程知识：将类声明放在使用该类的任何客户端的头文件(.h)中，形成类的public接口。将类成员函数的实现放在源文件(.cpp)中，形成类的实现方法。

4 Access Functions and Utility Functions

- Access functions（访问函数）
  - 可以读取和显示数据
  - 可以测试条件的真假
    - ✓例如, isEmpty 函数
- Utility functions (工具函数)
  - 私有成员函数用来支持公有成员函数的操作
  - 不是类的公共接口的一部分

```
1  // Fig. 9.5: SalesPerson.h
2  // SalesPerson class definition.
3  // Member functions defined in SalesPerson.cpp.
4  #ifndef SALESP_H
5  #define SALESP_H
6
7  class SalesPerson
8  {
9  public:
10    SalesPerson(); // constructor
11    void getSalesFromUser(); // input sales from keyboard
12    void setSales( int, double ); // set sales for a specific month
13    void printAnnualSales(); // summarize and print sales
14 private:
15    double totalAnnualSales(); // prototype for utility function
16    double sales[ 12 ]; // 12 monthly sales figures
17 }; // end class SalesPerson
18
19 #endif
```

Prototype for a **private** utility function

```
1  // Fig. 9.6: SalesPerson.cpp
2  // Member functions for class SalesPerson.
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7  using std::fixed;
8
9  #include <iomanip>
10 using std::setprecision;
11
12 #include "SalesPerson.h" // include SalesPerson class definition
13
14 // constructor :  initialize elements of array sales to 0.0
15 SalesPerson::SalesPerson()
16 {
17    for ( int i = 0; i < 12; i++ )
18       sales[ i ] = 0.0;
19 } // end SalesPerson constructor
```

```
20
21 // get 12 sales figures from the user at the keyboard
22 void SalesPerson::getSalesFromUser()
23 {
24    double salesFigure;
25
26    for ( int i = 1; i <= 12; i++ )
27    {
28       cout << "Enter sales amount for month " << i << ": ";
29       cin >> salesFigure;              // 循环输入 12 次
30       setSales( i, salesFigure );      // 实际是向数组中赋值，这种方法有些矫情
31    } // end for
32 } // end function getSalesFromUser
33
34 // set one of the 12 monthly sales figures; function subtracts
35 // one from month value for proper subscript in sales array
36 void SalesPerson::setSales( int month, double amount )
37 {
38    // test for valid month and amount values
39    if ( month >= 1 && month <= 12 && amount > 0 )
40       sales[ month - 1 ] = amount; // adjust for subscripts 0-11
41    else // invalid month or amount value
42       cout << "Invalid month or sales figure" << endl;
43 } // end function setSales
```

```
44
45 // print total annual sales (with the help of utility function)
46 void SalesPerson::printAnnualSales()
47 {
48    cout << setprecision( 2 ) << fixed
49       << "\nThe total annual sales are: $"
50       << totalAnnualSales() << endl; // call utility function
51 } // end function printAnnualSales
52
53 // private utility function to total annual sales
54 double SalesPerson::totalAnnualSales()
55 {
56    double total = 0.0; // initialize total
57
58    for ( int i = 0; i < 12; i++ ) // summarize sales results
59       total += sales[ i ]; // add month i sales to total
60
61    return total;
62 } // end function totalAnnualSales
```

Calling a `private` utility function

Definition of a `private` utility function

```cpp
// Fig. 9.7: fig09_07.cpp
// Demonstrating a utility function.
// Compile this program with SalesPerson.cpp

// include SalesPerson class definition from SalesPerson.h
#include "SalesPerson.h"

int main()
{
   SalesPerson s; // create SalesPerson object s

   s.getSalesFromUser(); // note simple sequential
   s.printAnnualSales(); // no control statements
   return 0;
} // end main
```

能写：s. totalAnnualSales() 吗？

```
Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03
Enter sales amount for month 5: 4376.34
Enter sales amount for month 6: 5698.45
Enter sales amount for month 7: 4439.22
Enter sales amount for month 8: 5893.57
Enter sales amount for month 9: 4909.67
Enter sales amount for month 10: 5123.45
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92

The total annual sales are: $60120.59
```

5 Constructors with Default Arguments

- 构造函数可以声明默认参数
  - 构造函数作用：对数据成员初始化
  - 所有参数均为默认参数的构造函数称为**默认构造函数**
    - ✓可以不加参数调用
    - ✓一个类最多有一个默认构造函数

1. 在C++中，允许在函数的声明或定义时给一个或多个参数指定默认值。但是要求在一个指定了默认值的参数的右边，不能出现没有指定默认值的参数。

2. 函数调用时，编译器按从左至右的顺序将实参与形参结合，当实参的数目不足时，编译器将按同样的顺序用说明中或定义中的默认值来补足所缺少的实参。

分析下列程序的运行结果：

```
#include <iostream>
using namespace std;
void fun (int a=1, int b=3, int c=5)
 {  cout <<"a=" <<a<< ", b=" << b <<", c=" << c << endl;  }
int main( )
 {    fun( );
      fun(7);
      fun(7,9);
      fun(7,9,11);
}
```

运行结果：
a=1, b=3, c=5
a=7, b=3, c=5
a=7, b=9, c=5
a=7, b=9, c=11

```
void ShMess (char*Text, int Length = -1, int Color = 0);
```

调用ShMess函数时，可指定一个、二个或三个参数：

```
ShMess("Helloo");

ShMess("Helloo",5);

ShMess(“Helloo”, 5, 8);

ShMess();  // 合法吗？
```

注意：说明缺省参数类似，如果调用函数时省略缺省参数，必须省略所有后续参数，例如，下面的调用是错误的：

```
ShMess("Helloo",  , 8);
```

使用默认参数可简化函数调用的编写。常见的程序设计错误是指定并使用默认参数不是最右边的参数（即没有把它右边的参数同时指定默认参数）。

例，函数可声明为(从右到左缺省）：

```
void f0(float x, int y, char z);
void f1(float x, int y, char z=`B`);
void f2(float x, int y=4, char z=`B`);
void f3(float x=1, int y=4, char z=`B`);
```

调用时（从右到左补充）：

```
float a=2.1; int b= 5; char c=`C`;
f3(a, b, c);   //三参数值为：a, b, c
f3(a,b);       //三参数值为：a, b, `B`
f3( );         //三参数值为：1, 4, `B`
f1(a,b);       //三参数值为：a, b, `B`
f0(a,b,c);     //三参数值为：a, b, c
f0(a,b); f1(a); f2( ); //错误
```

为什么要用默认参数函数?

1. 对函数的实参数实行初始化.
2. 使函数的定义和调用更具有一般性
3. 降低编程的复杂性, 降低程序出错的可能性

```
1  // Fig. 9.8: Time.h
2  // Declaration of class Time.
3  // Member functions defined in Time.cpp.
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME_H
7  #define TIME_H
8
9  // Time abstract data type definition
10 class Time
11 {
12 public:
13    Time( int = 0, int = 0, int = 0 ); // default constructor
14
15    // set functions
16    void setTime( int, int, int ); // set hour, minute, second
17    void setHour( int ); // set hour (after validation)
18    void setMinute( int ); // set minute (after validation)
19    void setSecond( int ); // set second (after validation)
```

Prototype of a constructor with default arguments

```
20
21    // get functions
22    int getHour(); // return hour
23    int getMinute(); // return minute
24    int getSecond(); // return second
25
26    void printUniversal(); // output time in universal-time format
27    void printStandard(); // output time in standard-time format
28 private:
29    int hour; // 0 - 23 (24-hour clock format)
30    int minute; // 0 - 59
31    int second; // 0 - 59
32 }; // end class Time
33
34 #endif
```

```cpp
// Fig. 9.9: Time.cpp
// Member-function definitions for class Time.
#include <iostream>
using std::cout;

#include <iomanip>
using std::setfill;
using std::setw;

#include "Time.h" // include definition of class Time from Time.h

// Time constructor initializes each data member to zero;
// ensures that Time objects start in a consistent state
Time::Time( int hr, int min, int sec )
{
   setTime( hr, min, sec ); // validate and set time
} // end Time constructor

// set new Time value using universal time; ensure that
// the data remains consistent by setting invalid values to zero
void Time::setTime( int h, int m, int s )
{
   setHour( h ); // set private field hour
   setMinute( m ); // set private field minute
   setSecond( s ); // set private field second
} // end function setTime
```

Parameters could receive the default values

Time t1;

Time t2(11);

Time t3(11,11);

Time t4(11,11,11);　　哪个是正确的？

```
27
28 // set hour value
29 void Time::setHour( int h )
30 {
31    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
32 } // end function setHour
33
34 // set minute value
35 void Time::setMinute( int m )
36 {
37    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
38 } // end function setMinute
39
40 // set second value
41 void Time::setSecond( int s )
42 {
43    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
44 } // end function setSecond
45
46 // return hour value
47 int Time::getHour()
48 {
49    return hour;
50 } // end function getHour
51
52 // return minute value
53 int Time::getMinute()
54 {
55    return minute;
56 } // end function getMinute
```

```
57
58 // return second value
59 int Time::getSecond()
60 {
61    return second;
62 } // end function getSecond
63
64 // print Time in universal-time format (HH:MM:SS)
65 void Time::printUniversal()
66 {
67    cout << setfill( '0' ) << setw( 2 ) << getHour() << ":"
68       << setw( 2 ) << getMinute() << ":" << setw( 2 ) << getSecond();
69 } // end function printUniversal
70
71 // print Time in standard-time format (HH:MM:SS AM or PM)
72 void Time::printStandard()
73 {
74    cout << ( ( getHour() == 0 || getHour() == 12 ) ? 12 : getHour() % 12 )
75       << ":" << setfill( '0' ) << setw( 2 ) << getMinute()
76       << ":" << setw( 2 ) << getSecond() << ( hour < 12 ? " AM" : " PM" );
77 } // end function printStandard
```

```
67   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
68      << setw( 2 ) << minute << ":" << setw( 2 ) << second;  ??
```

- 如果类的成员函数已经提供类的构造函数(或其他成员函数)所需功能的所有部分，则从构造函数(或其他成员函数)调用这个成员函数。这样可以简化代码维护和减少修改代码实现方法时的出错机会。因此就形成了一个一般原则：避免重复代码。
- 只在头文件内的类定义的函数原型中声明默认函数参数值。
  - 错误：在头文件和成员函数定义中指定同一成员函数的默认初始化值。

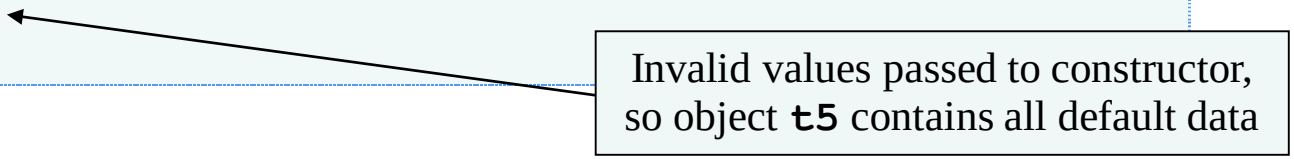
```
1  // Fig. 9.10: fig09_10.cpp
2  // Demonstrating a default constructor for class Time.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Time.h" // include definition of class Time from Time.h
8
9  int main()
10 {
11    Time t1; // all arguments defaulted
12    Time t2( 2 ); // hour specified; minute and second defaulted
13    Time t3( 21, 34 ); // hour and minute specified; second defaulted
14    Time t4( 12, 25, 42 ); // hour, minute and second specified
15    Time t5( 27, 74, 99 ); // all bad values specified
16
17    cout << "Constructed with:\n\nt1: all arguments defaulted\n  ";
18    t1.printUniversal(); // 00:00:00
19    cout << "\n  ";
20    t1.printStandard(); // 12:00:00 AM
21
22    cout << "\n\nt2: hour specified; minute and second defaulted\n  ";
23    t2.printUniversal(); // 02:00:00
24    cout << "\n  ";
25    t2.printStandard(); // 2:00:00 AM
```

Initializing **Time** objects using 0, 1, 2 and 3 arguments

可以加入吗： cout << setfill('0') << setw(2) << hour << ":" << setw(2) << minute << ":" << setw(2) << second; ??

```
26
27    cout << "\n\nt3: hour and minute specified; second defaulted\n  ";
28    t3.printUniversal(); // 21:34:00
29    cout << "\n  ";
30    t3.printStandard(); // 9:34:00 PM
31
32    cout << "\n\nt4: hour, minute and second specified\n  ";
33    t4.printUniversal(); // 12:25:42
34    cout << "\n  ";
35    t4.printStandard(); // 12:25:42 PM
36
37    cout << "\n\nt5: all invalid values specified\n  ";
38    t5.printUniversal(); // 00:00:00
39    cout << "\n  ";
40    t5.printStandard(); // 12:00:00 AM
41    cout << endl;
42    return 0;
43 } // end main
```

```
Constructed with:

t1: all arguments defaulted
  00:00:00
  12:00:00 AM

t2: hour specified; minute and second defaulted
  02:00:00
  2:00:00 AM

t3: hour and minute specified; second defaulted
  21:34:00
  9:34:00 PM

t4: hour, minute and second specified
  12:25:42
  12:25:42 PM

t5: all invalid values specified
  00:00:00
  12:00:00 AM
```



5 Constructors with Default Arguments

软件工程知识：如果类的成员函数已经提供类构造函数所需的全部或部分功能，构造函数就可以调用成员函数来完成。作为一个普遍原则，应该尽量避免重复代码(相同内容在不同函数中重复出现多次)。

软件工程知识：只在头文件类定义内部的函数原型中声明默认函数参数值。

6 Destructors

在一个程序块中，所定义的变量有作用域限定，当变量超出其作用域时，系统将自动释放为该变量分配的内存空间。

类似地，在一个程序块中，所定义的对象也有作用域限定，当对象超出其作用域时，系统将隐式调用类的**析构函数**。

说明：当撤销一个类的对象时，该类的析构函数将自动被调用。析构函数本身并不会破坏这个对象，它只是在系统回收对象所占用的内存空间之前做一些清理工作。

常见的程序设计错误是给析构函数传递参数,或从析构函数返回值,或重载析构函数。

析构函数的特点：

1. 析构函数也是成员函数，函数体可写在类体内，也可以写在类体外。

2. 析构函数的名字同类名，并在前面加“~”字符，用来与构造函数加以区别。析构函数不指定数据类型，并且也没有参数。

3. 一个类中只可能定义一个析构函数，不能重载。

4. 如果用户没有编写析构函数，编译系统会自动生成一个缺省的“空的”析构函数。

何时调用构造函数与析构函数

- 构造函数与析构函数是自动隐式调用的。
- 函数的调用顺序取决于执行过程进入和离开实例化对象范围的顺序。
- 析构函数的调用顺序与构造函数相反。
- 对象存储类可以改变析构函数的调用顺序。

6 Destructors

- 因此，析构函数
  - 特殊的成员函数，名字为：波浪线+类名，如： ~Time
  - 当对象被销毁时隐式调用
  - 并没有真正释放对象内存
    - ✓执行收尾工作
    - ✓系统重新声明对象内存
      - ◈使得内存可以被其他对象使用

6 Destructors

- 无参数，无返回值
- 一个类只能有一个析构函数
  - ✓析构函数不允许重载
- 如果程序没有显式提供析构函数，编译器会提供一个空的析构函数

6 Destructors

常见编程错误：向析构函数传递参数，指定析构函数的返回值类型（即使指定了void）以及从析构函数返回数值或重载析构函数都是语法错误。

7 构造函数与析构函数调用时间

- 对于在全局作用域定义的对象
  - 构造函数在任何其他函数之前调用
  - 相应的析构函数在 main 函数终止后调用
    - ✓ exit 函数
      - ◇ 迫使程序立即终止
      - ◇ 不执行自动对象的析构函数
      - ◇ 通常用来在程序检测到错误时终止程序

7 When Constructors and Destructors Are Called

- 对于在全局作用域定义的对象
  - ✓ abort 函数
    - ◇ 与 exit 的功能相似
      - ◇ 但是迫使程序立即终止，并且不允许任何对象的析构函数被调用
    - ◇ 通常用来指示程序的异常终止

7 When Constructors and Destructors Are Called

- 对于局部对象
  - 构造函数在对象被定义时调用
  - 相应得析构函数在对象离开其作用域时被调用
  - 如果程序使用 exit 或 abort 函数终止，对象的析构函数将不会被调用

7 When Constructors and Destructors Are Called

- 对于静态局部对象
  - 构造函数仅被调用一次（当对象第一次被定义时）
  - 析构函数在 main 函数退出或程序通过调用 exit 终止时被调用
    - ✓ 如果程序通过调用 abort 来终止，析构函数将不会被调用

```
1  // Fig. 9.11: CreateAndDestroy.h
2  // Definition of class CreateAndDestroy.
3  // Member functions defined in CreateAndDestroy.cpp.
4  #include <string>
5  using std::string;
6
7  #ifndef CREATE_H
8  #define CREATE_H
9
10 class CreateAndDestroy
11 {
12 public:
13    CreateAndDestroy( int, string ); // constructor
14    ~CreateAndDestroy(); // destructor
15 private:
16    int objectID; // ID number for object
17    string message; // message describing object
18 }; // end class CreateAndDestroy
19
20 #endif
```

Prototype for destructor

```cpp
// Fig. 9.12: CreateAndDestroy.cpp
// Member-function definitions for class CreateAndDestroy.
#include <iostream>
using std::cout;
using std::endl;

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

// constructor
CreateAndDestroy::CreateAndDestroy( int ID, string messageString )
{
   objectID = ID; // set object's ID number
   message = messageString; // set object's descriptive message

   cout << "Object " << objectID << "   constructor runs   "
      << message << endl;
} // end CreateAndDestroy constructor

// destructor
CreateAndDestroy::~CreateAndDestroy()
{
   // output newline for certain objects; helps readability
   cout << ( objectID == 1 || objectID == 6 ? "\n" : "" );

   cout << "Object " << objectID << "   destructor runs   "
      << message << endl;
} // end ~CreateAndDestroy destructor
```

Defining the class's destructor

```cpp
// Fig. 9.13: fig09_13.cpp
// Demonstrating the order in which constructors and
// destructors are called.
#include <iostream>
using std::cout;
using std::endl;

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

void create( void ); // prototype

CreateAndDestroy first( 1, "(global before main)" ); // global object

int main()
{
   cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
   CreateAndDestroy second( 2, "(local automatic in main)" );
   static CreateAndDestroy third( 3, "(local automatic in main)" );

   create(); // call function to create objects

   cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
   CreateAndDestroy fourth( 4, "(local automatic in main)" );
   cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
   return 0;
} // end main
```

全局范围中定义的对象的构造函数在文件中的任何其他函数(包括**main**)执行之前调用

Object created outside of **main**

当程序执行到对象定义时，调用自动局部对象的构造函数。

Local automatic object created in **main**

Local **static** object created in **main**

Local automatic object created in **main**

```
27
28 // function to create objects
29 void create( void )
30 {
31    cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
32    CreateAndDestroy fifth( 5, "(local automatic in create)" );
33    static CreateAndDestroy sixth( 6, "(local st
34    CreateAndDestroy seventh( 7, "(local automatic in create)" );
35    cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl
36 } // end function create
```

Local automatic object created in **create**

Local **static** object created in **create**

Local automatic object created in **create**

static局部对象的构造函数只在程序执行首次到达对象定义时调用一次，对应的析构函数在main终止或调用exit函数时调用。

自动对象的构造函数与析构函数在每次对象进入和离开范围时调用。

析构函数的调用顺序与构造函数相反

```
Object 1   constructor runs   (global before main)

MAIN FUNCTION: EXECUTION BEGINS
Object 2   constructor runs   (local automatic in main)
Object 3   constructor runs   (local static in main)

CREATE FUNCTION: EXECUTION BEGINS
Object 5   constructor runs   (local automatic in create)
Object 6   constructor runs   (local static in create)
Object 7   constructor runs   (local automatic in create)

CREATE FUNCTION: EXECUTION ENDS
Object 7   destructor runs    (local automatic in create)
Object 5   destructor runs    (local automatic in create)

MAIN FUNCTION: EXECUTION RESUMES
Object 4   constructor runs   (local automatic in main)

MAIN FUNCTION: EXECUTION ENDS
Object 4   destructor runs    (local automatic in main)
Object 2   destructor runs    (local automatic in main)

Object 6   destructor runs    (local static in create)
Object 3   destructor runs    (local static in main)

Object 1   destructor runs    (global before main)
```

8 A Subtle Trap—Returning a Reference to a private Data Member

- 返回一个私有对象的引用
  - 一种危险的用法
    - ✓类的公有成员函数返回一个该类私有数据成员的引用
      - ◇客户代码可以改变类的私有数据成员
      - ◇返回私有数据成员的指针会造成同样问题

```
1  // Fig. 9.14: Time.h
2  // Declaration of class Time.
3  // Member functions defined in Time.cpp
4
5  // prevent multiple inclusions of header file
6  #ifndef TIME_H
7  #define TIME_H
8
9  class Time
10 {
11 public:
12    Time( int = 0, int = 0, int = 0 );
13    void setTime( int, int, int );
14    int getHour();
15    int &badSetHour( int ); // DANGEROUS reference return
16 private:
17    int hour;
18    int minute;
19    int second;
20 }; // end class Time
21
22 #endif
```

Prototype for function that returns a reference

```
1  // Fig. 9.15: Time.cpp
2  // Member-function definitions for Time class.
3  #include "Time.h" // include definition of class Time
4
5  // constructor function to initialize private data;
6  // calls member function setTime to set variables;
7  // default values are 0 (see class definition)
8  Time::Time( int hr, int min, int sec )
9  {
10    setTime( hr, min, sec );
11 } // end Time constructor
12
13 // set values of hour, minute and second
14 void Time::setTime( int h, int m, int s )
15 {
16    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
17    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
18    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
19 } // end function setTime
```

```
20
21 // return hour value
22 int Time::getHour()
23 {
24    return hour;
25 } // end function getHour
26
27 // POOR PROGRAMMING PRACTICE:
28 // Returning a reference to a private data member.
29 int &Time::badSetHour( int hh )
30 {
31    hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
32    return hour; // DANGEROUS reference return
33 } // end function badSetHour
```

Returning a reference to a **`private`** data member = DANGEROUS!

```
1  // Fig. 9.16: fig09_16.cpp
2  // Demonstrating a public member function that
3  // returns a reference to a private data member.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "Time.h" // include definition of class Time
9
10 int main()
11 {
12    Time t; // create Time object
13
14    // initialize hourRef with the reference returned by badSetHour
15    int &hourRef = t.badSetHour( 20 ); // 20 is a valid hour
16
17    cout << "Valid hour before modification: " << hourRef;
18    hourRef = 30; // use hourRef to set invalid value in Time object t
19    cout << "\nInvalid hour after modification: " << t.getHour();
```

Modifying a **`private`** data member through a returned reference

```
20
21    // Dangerous: Function call that returns
22    // a reference can be used as an lvalue!
23    t.badSetHour( 12 ) = 74; // assign another invalid value to hour
24
25    cout << "\n\n*********************************************
26       << "POOR PROGRAMMING PRACTICE!!!!!!!\n"
27       << "t.badSetHour( 12 ) as an lvalue, invalid hour: "
28       << t.getHour()
29       << "\n*************************************************" << endl;
30    return 0;
31 } // end main
```

Modifying **`private`** data by using a function call as an *lvalue*

```
Valid hour before modification: 20
Invalid hour after modification: 30

*************************************************
POOR PROGRAMMING PRACTICE!!!!!!!
t.badSetHour( 12 ) as an lvalue, invalid hour: 74
*************************************************
```

8 A Subtle Trap—Returning a Reference to a private Data Member

错误预防技巧：绝不要让类的public成员函数返回对该类private数据成员的非常量引用（或指针）。返回这种引用会破坏类的封装，也是很危险的，应该避免。

9 Default Memberwise Assignment

- 默认逐个成员赋值
  - 赋值运算符 (=)
    - ✓ 可以进行同一类型对象之间的赋值
      - ◈等号右侧对象的每个数据成员被赋值到等号左侧的对象中
    - ✓ 当数据成员包含指针指向动态分配的内存中会导致严重的问题

9 Default Memberwise Assignment

- 拷贝构造函数 (Copy constructor)
  - 使得对象可以按值传递
    - ✓ 拷贝原始对象的值到传递给函数或函数返回的新对象中
  - 编译器提供默认拷贝构造函数
    - ✓ 拷贝对象每个每个数据成员到新对象中 (即： 逐个拷贝或赋值)
  - 当数据成员包含指向动态内存的指针时同样存在严重问题

拷贝构造函数 (Copy constructor)

拷贝构造函数是与类名相同，形参是本类的对象的引用的函数，在用已存在对象初始化新建立对象时调用。

类的拷贝构造函数一般由用户定义，如果用户没有定义拷贝构造函数，系统就会自动生成一个默认函数，这个默认拷贝构造函数的功能是把初始值对象的每个数据成员的值依次复制到新建立的对象中。因此，也可以说是完成了同类对象的克隆(Clone)。这样得到的对象和原对象具有完全相同的数据成员，即完全相同的属性。

拷贝构造函数 (Copy constructor)

用户可以也可以根据实际问题的需要定义特定的拷贝构造函数来改变缺省拷贝构造函数的行为，以实现同类对象之间数据成员的传递。如果用户自定义了拷贝构造函数，则在用一个类的对象初始化该类的另外一个对象时，自动调用自定义的拷贝构造函数。

定义一个拷贝构造函数的一般形式为：

```
类名（类名& 对象名）
{
   …
};
```

拷贝构造函数在用类的一个对象去初始化该类的另一个对象时调用，以下三种情况相当于用一个已存在的对象去初始化新建立的对象，此时，调用拷贝构造函数：

① 当用类的一个对象去初始化该类的另一个对象时。

② 如果函数的形参是类的对象，调用函数时，将对象作为函数实参传递给函数的形参时。

③ 如果函数的返回值是类的对象，函数执行完成，将返回值返回时。

```
1   /*****************************
2   *   Clock.cpp               *
3   *  构造拷贝构造函数         *
4   *****************************/
5   #include <iostream>
6   using namespace std;
7   class Clock {
8       private:
9         int H,M,S;
10      public:
11      Clock(int h=0,int m=0,int s=0)
12      {
13              H=h,M=m,S=s;
14              cout<<"constructor:"<<H<<":"<<M<<":"<<S<<endl;
15      }
16      ~Clock()
17      {
18              cout<<"destructor:"<<H<<":"<<M<<":"<<S<<endl;
19      }
```

浅拷贝与深拷贝

在默认的拷贝构造函数中，拷贝的策略是直接将原对象的数据成员值依次拷贝给新对象中对应的数据成员，如前面示例p6_4.cpp中定义的拷贝函数所示，那么我们为何不直接使用系统默认的拷贝构造函数，何必又自己定义一个拷贝构造函数呢？但是，有些情况下使用默认的拷贝构造函数却会出现**意想不到的问题**。

例如，使用下列程序中定义的String类，执行系统就会**出错**

```
class String {
    private:
            char *Str;
            int len;
    public:
            void ShowStr();
            String();
};
        int  main() {
           String s1("123456");
           String s2=s1;
        }
```

为什么会出错呢?

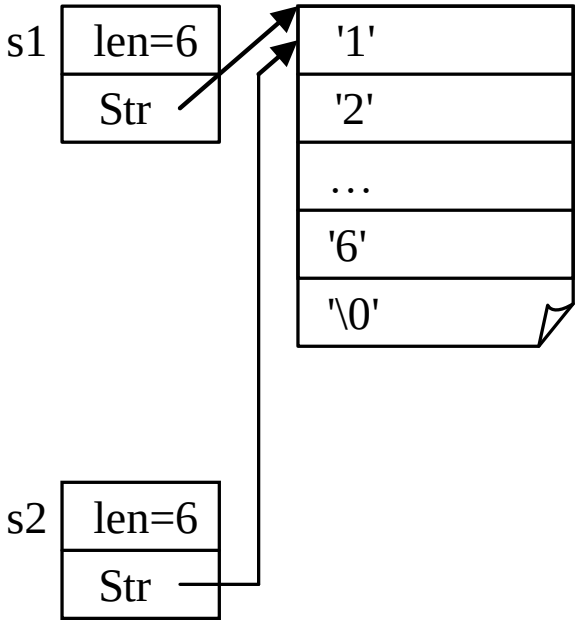
如果对象的数据成员是指针，
当执行String s2=s1时，默认的浅拷贝构造函数进行的是下列操作：

s2.len=s1.len;
s2.Str=s1.Str;

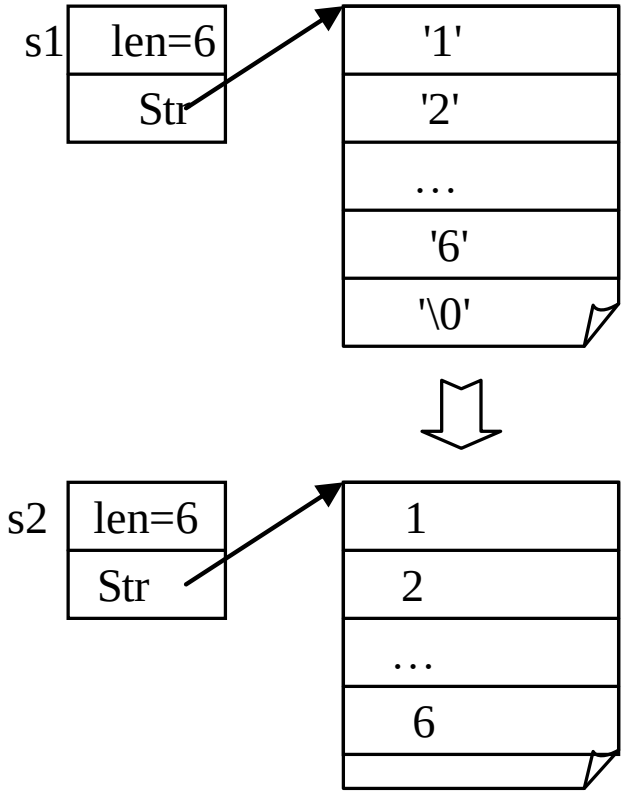
实际上是**将s1.Str的地址赋给了s2.Str，两个指针对象实际上指向的是同一块内存空间。并没有为s2.Str分配内存**，执行String s2=s1;后，对象s2析构，释放内存，然后对象s1析构，由于s1.Str 和s2.Str所占用的是同一块内存，而**同一块内存不可能释放两次**，所以当对象s1析构时，程序出现**异常**，无法正常执行和结束。

由此可见，在某些情况下，浅拷贝会带来数据安全方面的隐患。

当类的数据成员中有指针类型时，我们就必须定义一个特定的拷贝构造函数，该拷贝构造函数不仅可以实现原对象和新对象之间数据成员的拷贝，而且可以为新的对象**分配单独的内存资源**，这就是**深拷贝构造函数**。



执行 String s2=s1;进行浅拷贝



执行 String s2=s1；进行深拷贝

【例】 带深拷贝构造函数的字符串类。

在String类中，加入下列拷贝构造函数，构成了带深拷贝函数的字符串类。

```
String( String & r)
{
    len=r.len;
    if(len!=0)
    {
        Str=new char[len+1];
        strcpy(Str,r.Str);
    }
}
```

下列程序能正常运行。

```
int main(){
    String s1("123456");
    String s2=s1;
    s1.ShowStr();
    s2.ShowStr();
    return 0
}
```

注意：

- 在重新定义拷贝构造函数后，默认拷贝构造函数与默认构造函数就不存在了，如果在此时调用默认构造函数就会出错。
- 在重新定义构造函数后，默认构造函数就不存在了，但默认拷贝构造函数还存　在。
- 在对象进行赋值时，拷贝构造函数不被调用。此时进行的是结构式的拷贝。

```
20      Clock(Clock & p)
21      {
22          H=p.H;
23          M=p.M;
24          S=p.S;
25          cout<<“after copy constructor called: "<<H<<":"<<M<<":"<<S<<endl;
26      }
27      void ShowTime()
28      {
29          cout<<H<<":"<<M<<":"<<S<<endl;
30      }
31  };
32    Clock fun(Clock C)
33    {
34        return C;
35    }
36  void main(void)
37  {
38      Clock C1(8,0,0);
39      Clock C2(9,0,0);
40
41
42
43
44  }
```

运行结果：

```
constructor:8:0:0
constructor:9:0:0
after copy constructor called: 8:0:0
after copy constructor called: 9:0:0
after copy constructor called: 9:0:0
destructor:9:0:0
destructor:9:0:0
constructor:0:0:0
```

☞ 注意：

- 拷贝构造函数只是在用一个已存在的对象去初始化新建立的对象时调用，在对象进行赋值时，拷贝构造函数不被调用。
- 用一个常量初始化新建立的对象时，调用构造函数，不调用拷贝构造函数。
- 建立对象时，构造函数与拷贝构造函数有且仅有一个被调用。

9 Default Memberwise Assignment

性能提示： 按值传递对象的安全性较高，因为被调用函数无法访问原始对象。但如果要复制大对象，按值传递则可能降低性能。对象按引用传递时可以按指针或引用传递。按引用传递有性能优势，但安全性较低，因为被调用函数可以访问原始对象。按常量引用传递则既安全，又有性能上的优势。

```
1  // Fig. 9.17: Date.h
2  // Declaration of class Date.
3  // Member functions are defined in Date.cpp
4
5  // prevent multiple inclusions of header file
6  #ifndef DATE_H
7  #define DATE_H
8
9  // class Date definition
10 class Date
11 {
12 public:
13    Date( int = 1, int = 1, int = 2000 ); // default constructor
14    void print();
15 private:
16    int month;
17    int day;
18    int year;
19 }; // end class Date
20
21 #endif
```

```cpp
// Fig. 9.18: Date.cpp
// Member-function definitions for class Date.
#include <iostream>
using std::cout;
using std::endl;

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

// Date constructor (should do range checking)
Date::Date( int m, int d, int y )
{
   month = m;
   day = d;
   year = y;
} // end constructor Date

// print Date in the format mm/dd/yyyy
void Date::print()
{
   cout << month << '/' << day << '/' << year;
} // end function print
```

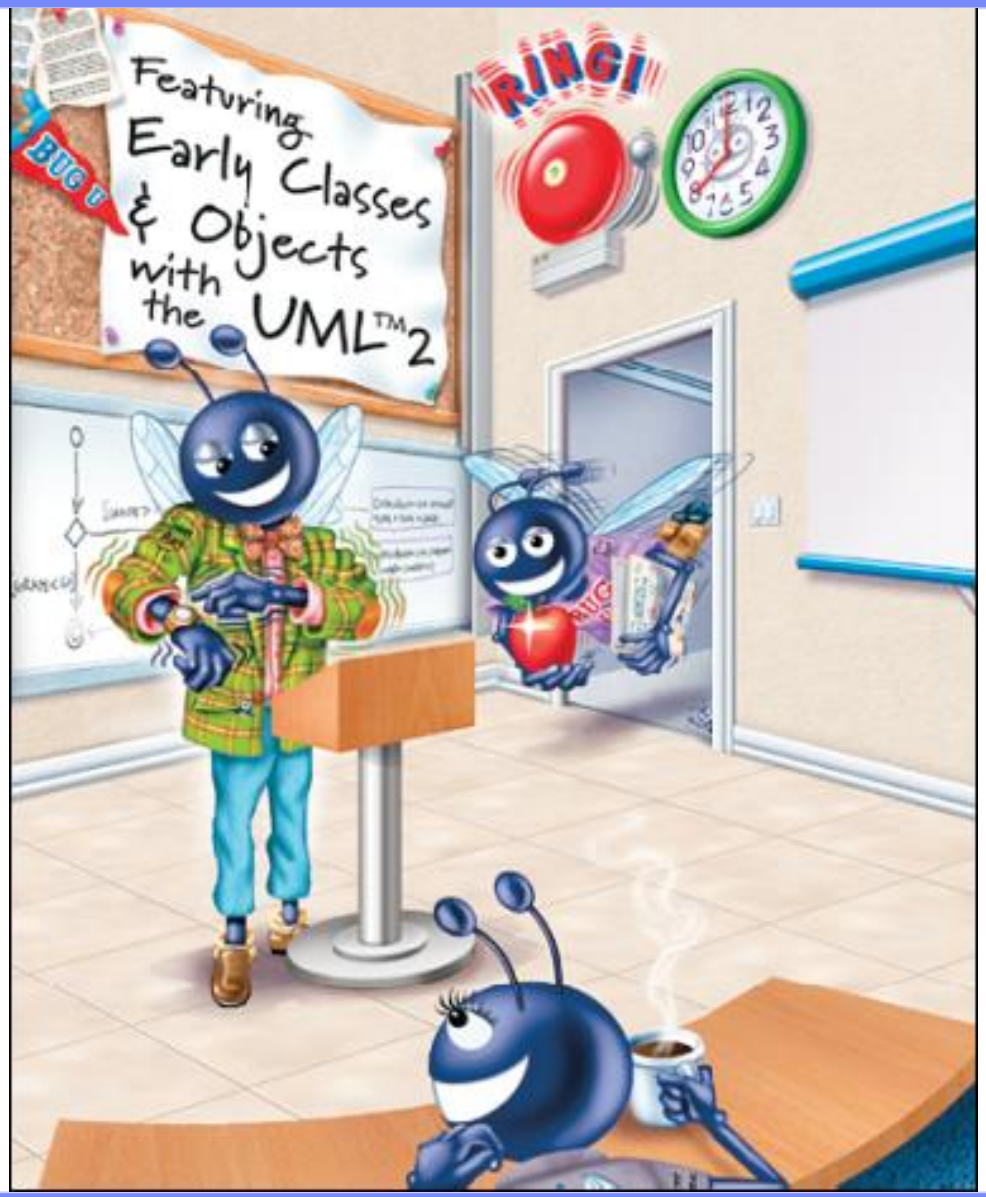
```
1  // Fig. 9.19: fig09_19.cpp
2  // Demonstrating that class objects can be assigned
3  // to each other using default memberwise assignment.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "Date.h" // include definition of class Date from Date.h
9
10 int main()
11 {
12    Date date1( 7, 4, 2004 );
13    Date date2; // date2 defaults to 1/1/2000
14
15    cout << "date1 = ";
16    date1.print();
17    cout << "\ndate2 = ";
18    date2.print();
19
20    date2 = date1; // default memberwise assignment
21
22    cout << "\n\nAfter default memberwise assignment, date2 = ";
23    date2.print();
24    cout << endl;
25    return 0;
26 } // end main
```

Memberwise assignment assigns data members of **date1** to **date2**

date2 now stores the same date as **date1**

```
date1 = 7/4/2004
date2 = 1/1/2000

After default memberwise assignment, date2 = 7/4/2004
```

END!



Thank you!