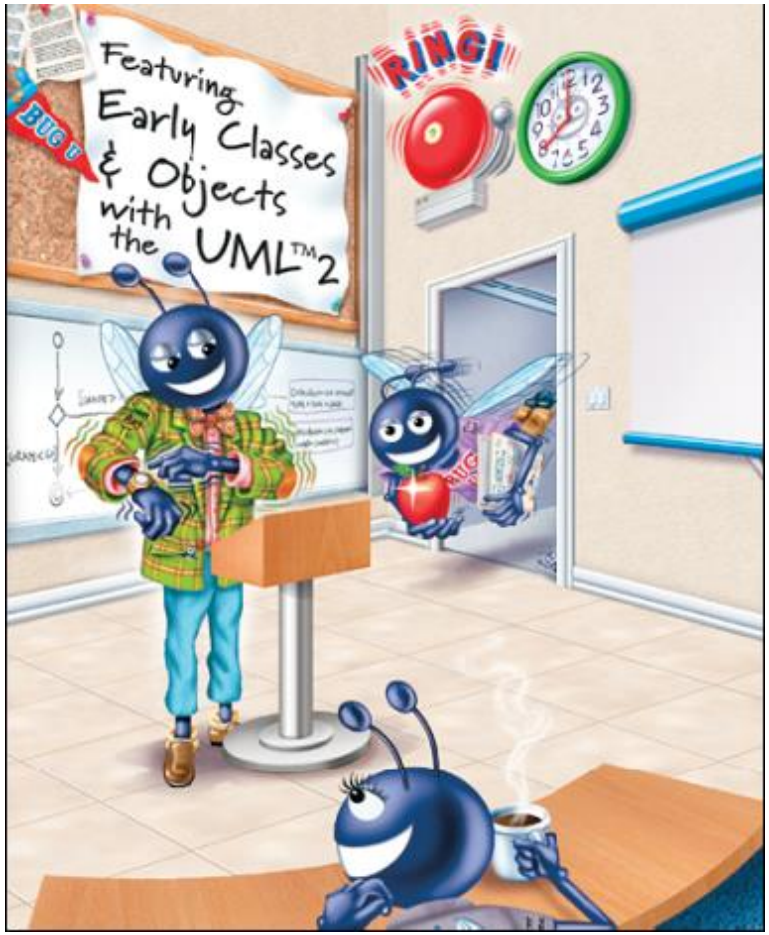
C++程序设计



上节课内容回顾

1. 指针和引用的异同
2. 指针作为参数传递给函数
3. 基于指针的 C 风格的字符串
4. 指针和数组的关系
5. 函数指针

第八讲 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

1. Time Class Case Study

- **Prevents multiple-definition errors**

```
#ifndef TIME_H
#define TIME_H
... // code
#endif
```

```
#ifndef TIME_H
#define TIME_H
// Time class definition
class Time
{
public:
   Time(); // constructor
   void setTime( int, int, int ); // set hour, minute and second
   void printUniversal(); // print time in universal-time format
   void printStandard(); // print time in standard-time format
private:
   int hour; // 0 - 23 (24-hour clock format)
   int minute; // 0 – 59
   int second; // 0 – 59
}; // end class Time
#endif
```

```
Time::Time()
{
   hour = minute = second = 0;
}

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

```
// print Time in universal-time format (HH:MM:SS)
void Time::printUniversal()
{
   cout << setfill( '0' ) << setw( 2 ) << hour << ":"
      << setw( 2 ) << minute << ":" << setw( 2 ) << second;
} // end function printUniversal

// print Time in standard-time format (HH:MM:SS AM or PM)
void Time::printStandard()
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << ":"
      << setfill( '0' ) << setw( 2 ) << minute << ":" << setw( 2 )
      << second << ( hour < 12 ? " AM" : " PM" );
} // end function printStandard
```

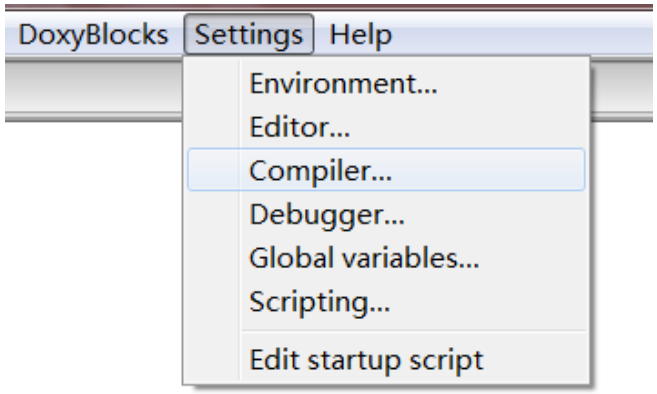
```
#include "Time.h" // include definition of class Time from Time.h

int main()
{
   Time t; // instantiate object t of class Time

   t.printUniversal(); // 00:00:00
   t.printStandard(); // 12:00:00 AM

   t.setTime( 13, 27, 6 ); // change time
   ......
   t.setTime( 99, 99, 99 ); // attempt invalid settings
   ......
}
```

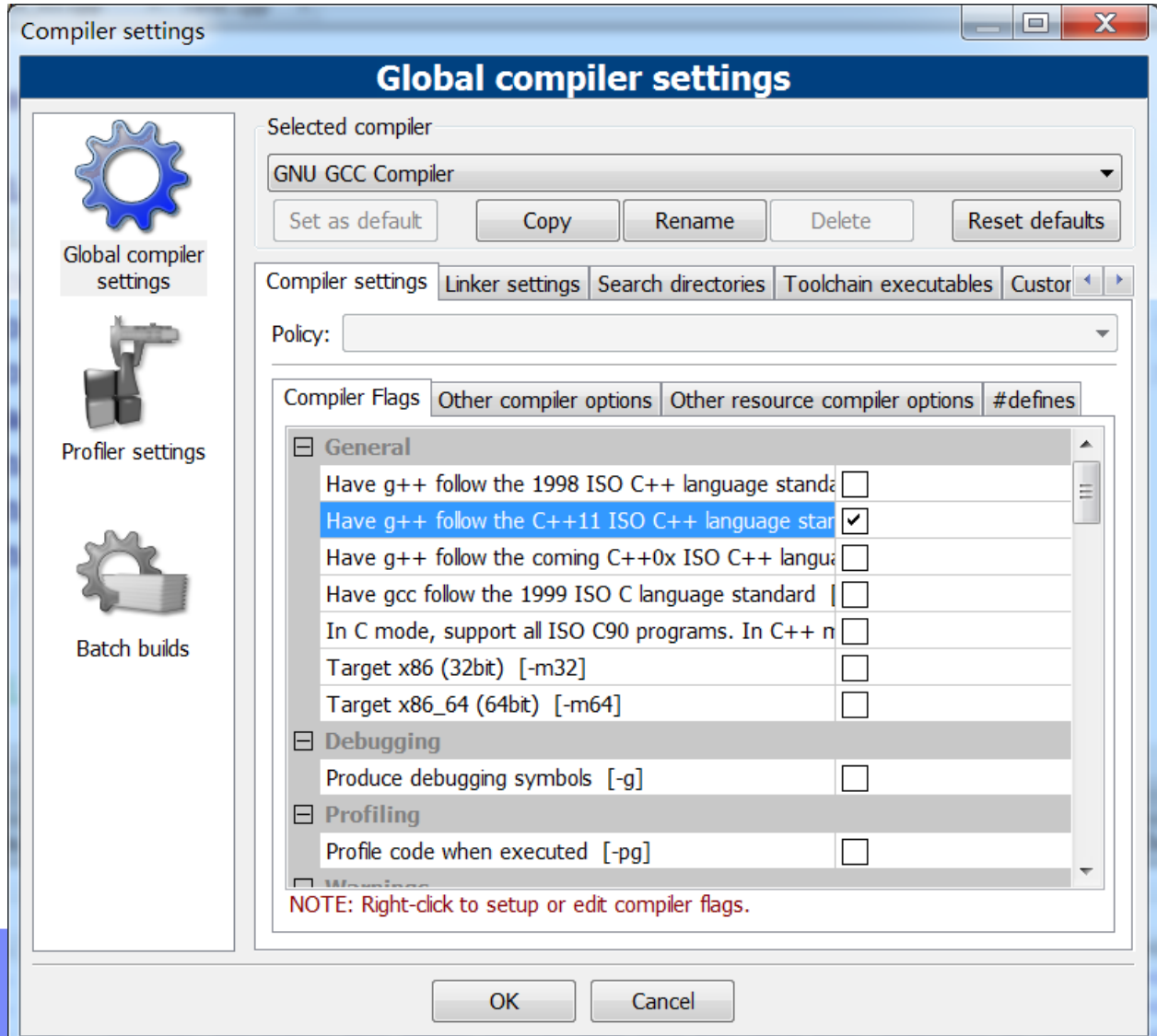
Non-Static Data Member Initializers

```
class Time
{
public:
    Time(); // constructor
    void setTime( int , int, int );
    void printUniversal(); // print
    void printStandard(); // print t
private:
    int hour = 0; // 0 - 23 (24-hour
    int minute = 0; // 0 - 59
    int second = 0; // 0 - 59
}; // end class Time
```

```
warning: non-static data member initializers only available with -std=c++11 or -std=gnu++11
warning: non-static data member initializers only available with -std=c++11 or -std=gnu++11
warning: non-static data member initializers only available with -std=c++11 or -std=gnu++11
```

Non-Static Data Member Initializers





Non-Static Data Member Initializers

As a simple example,

```
class A {
public:
    int a = 7;
};
```

would be equivalent to

```
class A {
public:
    A() : a(7) {}
};
```

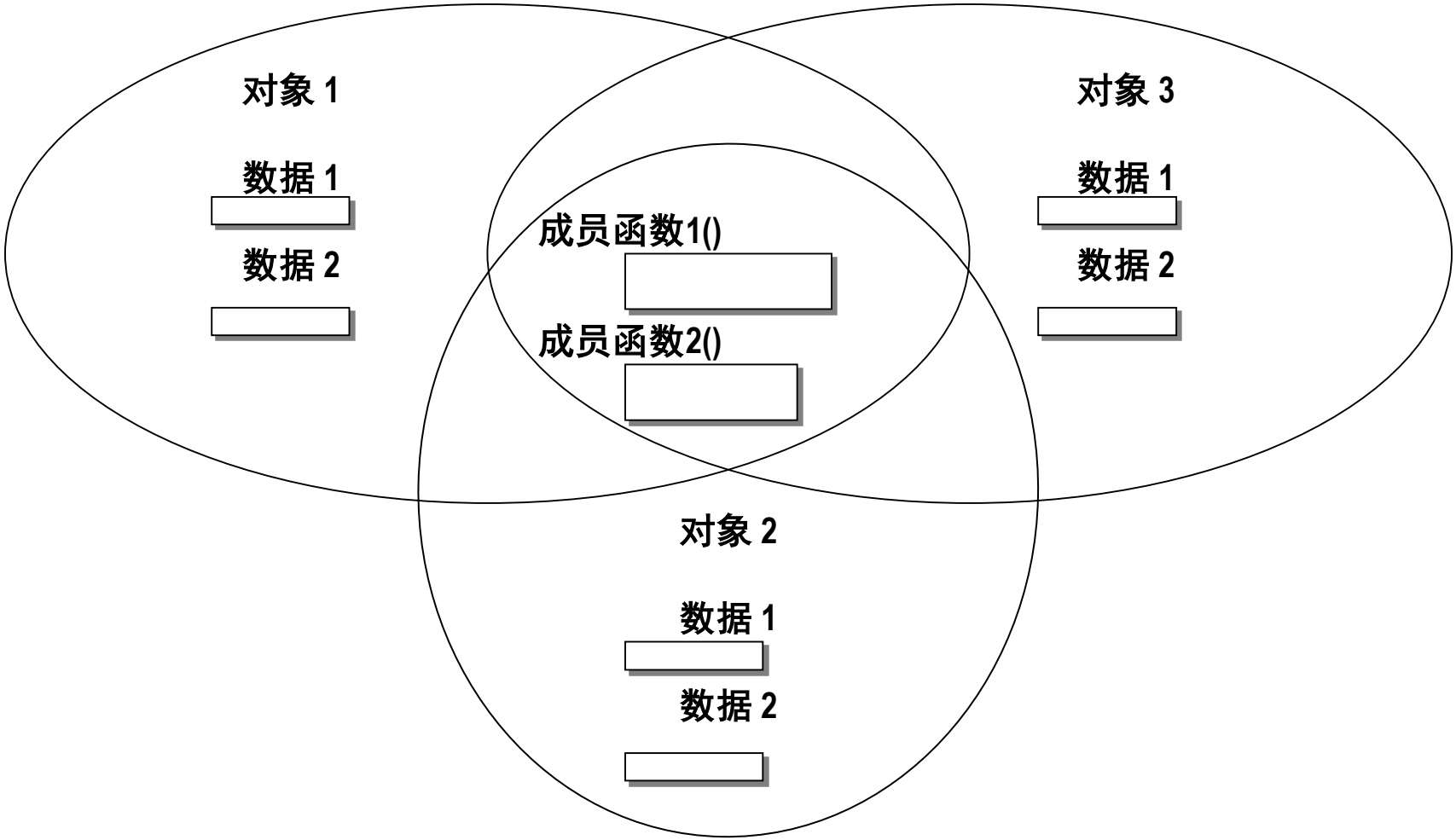
1. Time Class Case Study

- 成员函数在类定义中声明，在类定义外实现
- 成员函数在类定义中声明并实现
  - C++编译器尝试用内联方式调用成员函数

1. Time Class Case Study

- 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;

1. Time Class Case Study



2. Class Scope and Accessing Class Members

- 类作用域包括
  - 数据成员
    - ◈在类定义中声明的变量
  - 成员函数
    - ◈在类定义中声明的函数
- 非成员函数定义在文件作用域内

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
    - ◈对象名
    - ◈对象引用
    - ◈对象指针

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

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

3. Separating Interface from Implementation

- 将类定义与类成员函数定义相分离
  - 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

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）。

4. Access Functions and Utility Functions

- Access functions
  - 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 (also called helper functions)
  - 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

```
#ifndef SALESP_H
#define SALESP_H
class SalesPerson
{
public:
   SalesPerson(); // constructor
   void getSalesFromUser(); // input sales from keyboard
   void setSales( int, double ); // set sales for a specific month
   void printAnnualSales(); // summarize and print sales
private:
   double totalAnnualSales(); // prototype for utility function
   double sales[ 12 ]; // 12 monthly sales figures
}; // end class SalesPerson
#endif
```

```
void SalesPerson::printAnnualSales()
{
   cout << setprecision( 2 ) << fixed
      << "\nThe total annual sales are: $"
      << totalAnnualSales() << endl; // call utility function
} // end function printAnnualSales

double SalesPerson::totalAnnualSales()
{
   double total = 0.0; // initialize total

   for ( int i = 0; i < 12; i++ ) // summarize sales results
      total += sales[ i ]; // add month i sales to total

   return total;
} // end function totalAnnualSales
```

5. Constructors with Default Arguments

- 构造函数可以声明默认参数
  - 对数据成员初始化
  - 所有参数均为默认参数的构造函数也是默认构造函数
    - 可以不加参数调用
    - 一个类最多有一个默认构造函数

```
#ifndef TIME_H
#define TIME_H

// Time abstract data type definition
class Time
{
public:
   Time( int = 0, int = 0, int = 0 ); // default constructor

   // set functions
   void setTime( int, int, int ); // set hour, minute, second
   void setHour( int ); // set hour (after validation)
.......
};
```

```
#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
```

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

int main()
{
   Time t1; // all arguments defaulted
   Time t2( 2 ); // hour specified; minute and second defaulted
   Time t3( 21, 34 ); // hour and minute specified; second defaulted
   Time t4( 12, 25, 42 ); // hour, minute and second specified
   Time t5( 27, 74, 99 ); // all bad values specified

   ......
}
```

6. Destructors

- 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

- 析构函数
  - 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

7. When Constructors and Destructors Are Called

- 构造函数和析构函数
  - 编译器隐式调用
    - ◈调用顺序取决于对象何时进入或离开其作用域
  - 通常
    - ◈析构函数按照调用构造函数相反的顺序调用
  - 但是
    - ◈对象的存储类别会改变析构函数的调用顺序

7. When Constructors and Destructors Are Called

- 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
  - 相应的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

```
#ifndef CREATE_H
#define CREATE_H

class CreateAndDestroy
{
public:
   CreateAndDestroy( int, string ); // constructor
   ~CreateAndDestroy(); // destructor
private:
   int objectID; // ID number for object
   string message; // message describing object
}; // end class CreateAndDestroy

#endif
```

```
#include "CreateAndDestroy.h"


// 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
```

```
#include "CreateAndDestroy.h"
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 static 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
```

```
// function to create objects
void create( void )
{
   cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
   CreateAndDestroy fifth( 5, "(local automatic in create)" );
   static CreateAndDestroy sixth( 6, "(local static in create)" );
   CreateAndDestroy seventh( 7, "(local automatic in create)" );
   cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
} // end function create
```

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

- 返回一个对象的引用
  - 对象的别名：可以作为左值
    - ◈若返回常量引用，不能作为左值

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

- 返回一个对象的引用
  - 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

```
class Time
{
public:
   Time( int = 0, int = 0, int = 0 );
   void setTime( int, int, int );
   int getHour();
   int &badSetHour( int ); // DANGEROUS reference return
private:
   int hour;
   int minute;
   int second;
}; // end class Time
```

```
// POOR PROGRAMMING PRACTICE:
// Returning a reference to a private data member.
int &Time::badSetHour( int hh )
{
   hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
   return hour; // DANGEROUS reference return
} // end function badSetHour
```

```
int main()
{
   Time t; // create Time object

   // initialize hourRef with the reference returned by badSetHour
   int &hourRef = t.badSetHour( 20 ); // 20 is a valid hour

   cout << "Valid hour before modification: " << hourRef;
   hourRef = 30; // use hourRef to set invalid value in Time object t
   cout << "\nInvalid hour after modification: " << t.getHour();
   // Dangerous: Function call that returns
   // a reference can be used as an lvalue!
   t.badSetHour( 12 ) = 74; // assign another invalid value to hour
   return 0;
} // end main
```

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

思考题：

- 修改Time类，使它包含一个tick成员函数，该函数将存放在Time对象中的时间递增1秒。编写一个程序，在循环中测试tick成员函数。在每次循环迭代中都以标准时间格式打印时间，以显示tick成员函数是否工作正常。

  a) 递增到下一分钟；b) 递增到下一小时；c) 递增到下一天