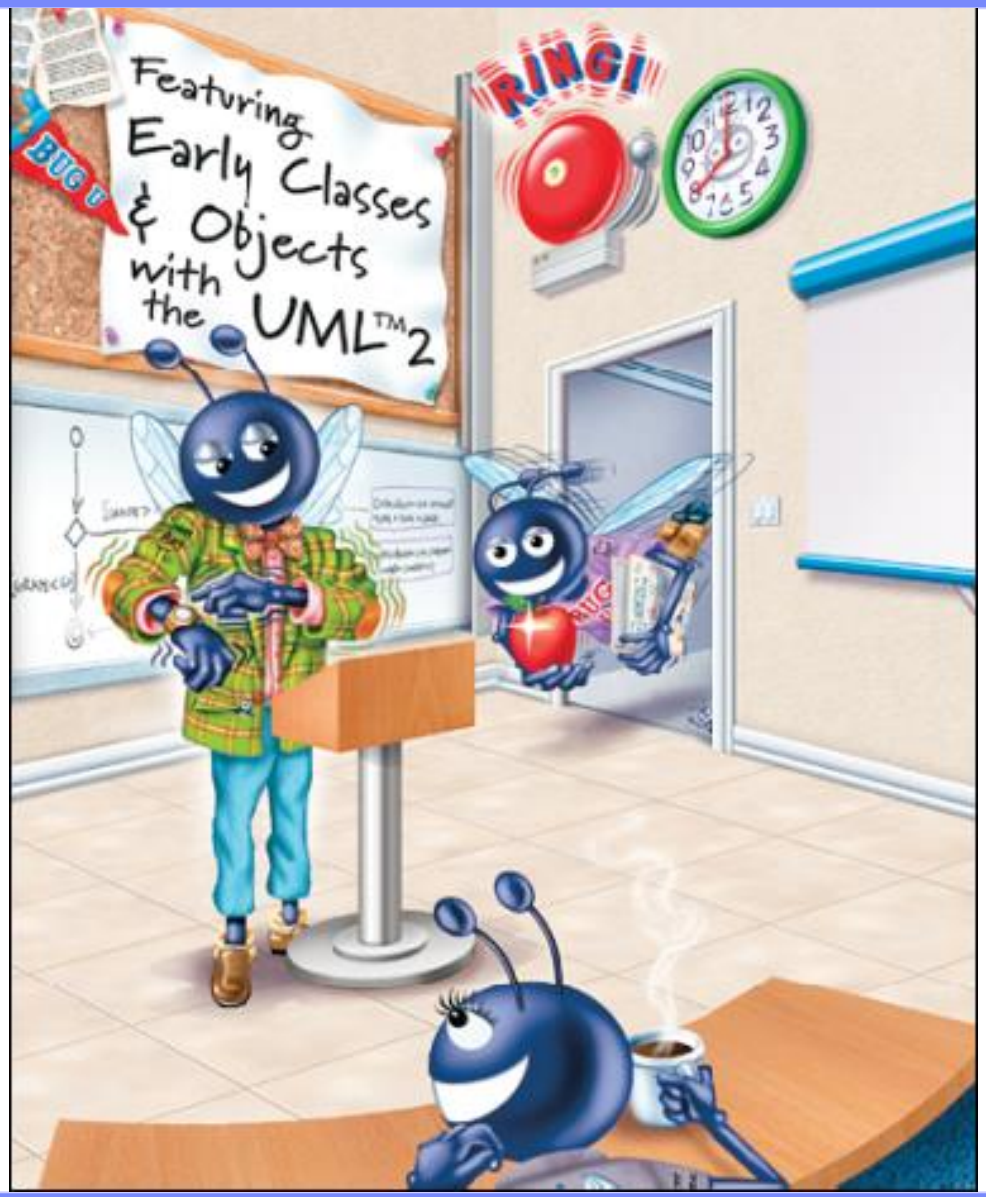


Lecture 7:
类的深入剖析(2)

第十讲 类的深入剖析（II）

学习目标：

- const 对象和 const 成员函数
- 创建由其他对象组成的类
- friend 函数和 friend 类
- this 指针
- new 和 delete
- static 数据成员和成员函数
- 容器类、代理类

1 const 对象和 const 成员函数

- const 对象
  - 标志：关键字 const
  - 目的：声明对象不能被修改
  - 如果const对象被修改会产生编译错误

1 const Objects and const Member Functions

软件工程知识：将对象声明为const有利于实现最低权限原则，试图修改const对象会在编译时发现错误而非等到执行时才发现错误。

性能提示：把变量和对象声明为const，这不仅是一种有效的软件工程原则，而且还能提高性能，因为如今复杂的优化编译器能对常量进行某些优化，但无法对变量进行优化。

1 const Objects and const Member Functions

- const 成员函数
  - const 成员函数的声明和定义都需要用 const 修饰
  - 构造函数和析构函数不能声明为 const
  - const 对象只能调用 const 成员函数
  - const 成员函数不能修改对象

1 const Objects and const Member Functions

- const 成员函数
  - 原型:

ReturnType FunctionName(param1,param2…) const;

  - 定义:

ReturnType FunctionName(param1,param2…) const
{ …}

1 const Objects and const Member Functions

- const 成员函数

```
int A::getValue() const
{
    return privateDataMember;
}
```

1 const Objects and const Member Functions

软件工程知识：可以对 const 成员函数进行非const版本的重载。编译器将根据调用函数的对象性质选择相应的重载函数来使用。如果对象是const的，则编译器使用 const 版本的重载函数；如果对象是非const的，则编译器使用非const版本的重载函数。

```
1  // Fig. 10.1: Time.h
2  // Definition of class Time.
3  // Member functions defined in Time.cpp.
4  #ifndef TIME_H
5  #define TIME_H
6
7  class Time
8  {
9  public:
10    Time( int = 0, int = 0, int = 0 ); // default constructor
11
12    // set functions
13    void setTime( int, int, int ); // set time
14    void setHour( int ); // set hour
15    void setMinute( int ); // set minute
16    void setSecond( int ); // set second
17
18    // get functions (normally declared const)
19    int getHour() const; // return hour
20    int getMinute() const; // return minute
21    int getSecond() const; // return second
```

const 表明这几个成员函数不能修改对象

```
22
23    // print functions (normally declared const)
24    void printUniversal() const; // print universal time
25    void printStandard(); // print standard time (should be const)
26 private:
27    int hour; // 0 - 23 (24-hour clock format)
28    int minute; // 0 - 59
29    int second; // 0 - 59
30 }; // end class Time
31
32 #endif
```

```cpp
// Fig. 10.2: 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

// constructor function to initialize private data;
// calls member function setTime to set variables;
// default values are 0 (see class definition)
Time::Time( int hour, int minute, int second )
{
   setTime( hour, minute, second );
} // end Time constructor

// set hour, minute and second values
void Time::setTime( int hour, int minute, int second )
{
   setHour( hour );
   setMinute( minute );
   setSecond( second );
} // end function setTime
```

构造函数调用一般函数完成，没有重新写

```
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() const // get functions should be const
48 {
49    return hour;
50 } // end function getHour
```

const 在函数原型与函数实现中都需要得到体现

```
51
52 // return minute value
53 int Time::getMinute() const
54 {
55    return minute;
56 } // end function getMinute
57
58 // return second value
59 int Time::getSecond() const
60 {
61    return second;
62 } // end function getSecond
63
64 // print Time in universal-time format (HH:MM:SS)
65 void Time::printUniversal() const
66 {
67    cout << setfill( '0' ) << setw( 2 ) << hour << ":"
68       << setw( 2 ) << minute << ":" << setw( 2 ) << second;
69 } // end function printUniversal
70
71 // print Time in standard-time format (HH:MM:SS AM or PM)
72 void Time::printStandard() // note lack of const declaration
73 {
74    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
75       << ":" << setfill( '0' ) << setw( 2 ) << minute
76       << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
77 } // end function printStandard
```

```
1  // Fig. 10.3: fig10_03.cpp
2  // Attempting to access a const object with non-const member functions.
3  #include "Time.h" // include Time class definition
4
5  int main()
6  {
7     Time wakeUp( 6, 45, 0 ); // non-constant object
8     const Time noon( 12, 0, 0 ); // constant object,  const写在最前面
9
10                              // OBJECT      MEMBER FUNCTION
11    wakeUp.setHour( 18 );  // non-const   non-const
12
13    noon.setHour( 12 );    // const       non-const
14
15    wakeUp.getHour();      // non-const   const
16
17    noon.getMinute();      // const       const
18    noon.printUniversal(); // const       const
19
20    noon.printStandard();  // const       non-const
21    return 0;
22 } // end main
```

Cannot invoke non-**const** member functions on a **const** object

Borland C++ command-line compiler error messages:

```
Warning W8037 fig10_03.cpp 13: Non-const function Time::setHour(int)
   called for const object in function main()
Warning W8037 fig10_03.cpp 20: Non-const function Time::printStandard()
   called for const object in function main()
```

Microsoft Visual C++.NET compiler error messages:

```
C:\cpphtp5_examples\ch10\Fig10_01_03\fig10_03.cpp(13) : error C2662:
   'Time::setHour' : cannot convert 'this' pointer from 'const Time' to
   'Time &'
        Conversion loses qualifiers
C:\cpphtp5_examples\ch10\Fig10_01_03\fig10_03.cpp(20) : error C2662:
   'Time::printStandard' : cannot convert 'this' pointer from 'const Time' to
   'Time &'
        Conversion loses qualifiers
```

GNU C++ compiler error messages:

```
fig10_03.cpp:13: error: passing `const Time' as `this' argument of
   `void Time::setHour(int)' discards qualifiers
fig10_03.cpp:20: error: passing `const Time' as `this' argument of
   `void Time::printStandard()' discards qualifiers
```

问题：

- 前面的例子中，类的私有成员均为非const类型，因此可以正常地调用构造函数进行初始化

- 如果该私有成员为const类型，则要求定义时就赋值，显然在类定义中直接赋值是错误的，而又不能像非const类型成员那样通过构造函数赋值语句来赋值，那该怎么办呢？

答案：成员初始化器

1 const Objects and const Member Functions

- Member initializer （成员初始化器）
  - 对特定类型的数据成员进行初始化
    - ✓const 数据成员
    - ✓引用类型的数据成员
  - 也可以用于任何数据成员

1 const Objects and const Member Functions

- Member initializer list
  - 出现在构造函数参数列表后，函数体的左花括号前
  - 用冒号 (:) 与参数列表相分隔
  - 数据成员名后跟括号，括号内包含初始值
  - 多个数据成员用逗号分隔
  - 初始化在构造函数执行前执行

```
1  // Fig. 10.4: Increment.h
2  // Definition of class Increment.
3  #ifndef INCREMENT_H
4  #define INCREMENT_H
5
6  class Increment
7  {
8  public:
9     Increment( int c = 0, int i = 1 ); // default constructor
10
11    // function addIncrement definition
12    void addIncrement()
13    {
14       count += increment;
15    } // end function addIncrement
16
17    void print() const; // prints count and increment
18 private:
19    int count;
20    const int increment; // const data member
21 }; // end class Increment
22
23 #endif
```

const data member that must be initialized using a member initializer

```
1  // Fig. 10.5: Increment.cpp
2  // Member-function definitions for class Increment demonstrate using a
3  // member initializer to initialize a constant of a built-in data type.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "Increment.h" // include definition of class Increment
9
10 // constructor
11 Increment::Increment( int c, int i )
12    : count( c ), // initializer for non-const member
13      increment( i ) // required initializer for const member
14 {
15    // empty body
16 } // end constructor Increment
17
18 // print count and increment values
19 void Increment::print() const
20 {
21    cout << "count = " << count << ", increment = " << increment << endl;
22 } // end function print
```

Colon (:) marks the start of a member initializer list

Member initializer for non-**const** member **count**

Required member initializer for **const** member **increment**

真正需要用初始化器进行初始化的只有 **increment**

```
11 Increment::Increment( int c, int i )
12    : increment( i ) // required initializer for const member
13
14 {
15    count = c ;
16 }// end constructor Increment
```

```cpp
// Fig. 10.6: fig10_06.cpp
// Program to test class Increment.
#include <iostream>
using std::cout;

#include "Increment.h" // include definition of class Increment

int main()
{
   Increment value( 10, 5 );  // count = 10, increment = 5

   cout << "Before incrementing: ";
   value.print();

   for ( int j = 1; j <= 3; j++ )
   {
      value.addIncrement();
      cout << "After increment " << j << ": ";
      value.print();
   } // end for

   return 0;
} // end main
```

```
Before incrementing: count = 10, increment = 5
After increment 1: count = 15, increment = 5
After increment 2: count = 20, increment = 5
After increment 3: count = 25, increment = 5
```

1 const Objects and const Member Functions

常见编程错误：不给常量数据成员提供成员初始化值会引起语法错误。

软件工程知识：如果成员函数不修改对象，最好将所有类成员函数声明为const。

软件工程知识：常量数据成员(const 对象和const“变量”）和引用数据成员要用**成员初始化器**来初始化，不能用赋值语句。

```
1  // Fig. 10.7: Increment.h
2  // Definition of class Increment.
3  #ifndef INCREMENT_H
4  #define INCREMENT_H
5
6  class Increment
7  {
8  public:
9     Increment( int c = 0, int i = 1 ); // default constructor
10
11    // function addIncrement definition
12    void addIncrement()
13    {
14       count += increment;
15    } // end function addIncrement
16
17    void print() const; // prints count and increment
18 private:
19    int count;
20    const int increment; // const data member
21 }; // end class Increment
22
23 #endif
```

Member function declared **`const`** to prevent errors in situations where an **`Increment`** object is treated as a **`const`** object

```cpp
// Fig. 10.8: Increment.cpp
// Attempting to initialize a constant of
// a built-in data type with an assignment.
#include <iostream>
using std::cout;
using std::endl;

#include "Increment.h" // include definition of class Increment

// constructor; constant member 'increment' is not initialized
Increment::Increment( int c, int i )
{
   count = c; // allowed because count is not constant
   increment = i; // ERROR: Cannot modify a const object
} // end constructor Increment

// print count and increment values
void Increment::print() const
{
   cout << "count = " << count << ", increment = " << increment << endl;
} // end function print
```

It is an error to modify a **`const`** data member; data member **`increment`** must be initialized with a member initializer

```cpp
Increment::Increment( int c, int i )
   : increment( i ) // required initializer for const member

{
   count = c ;
}// end constructor Increment
```

```cpp
// Fig. 10.9: fig10_09.cpp
// Program to test class Increment.
#include <iostream>
using std::cout;

#include "Increment.h" // include definition of class Increment

int main()
{
   Increment value( 10, 5 );

   cout << "Before incrementing: ";
   value.print();

   for ( int j = 1; j <= 3; j++ )
   {
      value.addIncrement();
      cout << "After increment " << j << ": ";
      value.print();
   } // end for

   return 0;
} // end main
```

Borland C++ command-line compiler error message:

```
Error E2024 Increment.cpp 14: Cannot modify a const object in function
   Increment::Increment(int,int)
```

Microsoft Visual C++.NET compiler error messages:

```
C:\cpphtp5_examples\ch10\Fig10_07_09\Increment.cpp(12) : error C2758:
   'Increment::increment' : must be initialized in constructor
   base/member initializer list
         C:\cpphtp5_examples\ch10\Fig10_07_09\Increment.h(20) :
            see declaration of 'Increment::increment'
C:\cpphtp5_examples\ch10\Fig10_07_09\Increment.cpp(14) : error C2166:
   l-value specifies const object
```

GNU C++ compiler error messages:

```
Increment.cpp:12: error: uninitialized member 'Increment::increment' with
   'const' type 'const int'
Increment.cpp:14: error: assignment of read-only data-member
   `Increment::increment'
```

2 Composition: Objects as Members of Classes

- Composition (组合)(对象作为类的成员)
  - 是一种 *has-a* 关系
  - 一个类可以将其他类的对象作为成员
  - Example
    - ✓AlarmClock 对象将 Time 对象作为成员

2 Composition: Objects as Members of Classes

- 初始化成员对象
  - 成员初始化器从对象的构造函数向成员对象的构造函数传递参数
  - 成员对象按照它们在类定义中出现的顺序进行构造，而不是按照在初始化列表中出现的顺序
    - ✓ 在宿主对象构造之前进行构造
  - 如果不提供初始化器
    - ✓ 成员对象的默认构造函数被隐式调用

```
1  // Fig. 10.10: Date.h
2  // Date class definition; Member functions defined in Date.cpp
3  #ifndef DATE_H
4  #define DATE_H
5
6  class Date
7  {
8  public:
9     Date( int = 1, int = 1, int = 1900 ); // default constructor
10    void print() const; // print date in month/day/year format
11    ~Date(); // provided to confirm destruction order
12 private:
13    int month; // 1-12 (January-December)
14    int day; // 1-31 based on month
15    int year; // any year
16
17    // utility function to check if day is proper for month and year
18    int checkDay( int ) const;
19 }; // end class Date
20
21 #endif
```

```
1  // Fig. 10.11: Date.cpp
2  // Member-function definitions for class Date.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Date.h" // include Date class definition
8
9  // constructor confirms proper value for month; calls
10 // utility function checkDay to confirm proper value for day
11 Date::Date( int mn, int dy, int yr )
12 {
13    if ( mn > 0 && mn <= 12 ) // validate the month
14       month = mn;
15    else
16    {
17       month = 1; // invalid month set to 1
18       cout << "Invalid month (" << mn << ") set to 1.\n";
19    } // end else
20
21    year = yr; // could validate yr
22    day = checkDay( dy ); // validate the day
23
24    // output Date object to show when its constructor is called
25    cout << "Date object constructor for date ";
26    print();
27    cout << endl;
28 } // end Date constructor
```

```
29
30 // print Date object in form month/day/year
31 void Date::print() const
32 {
33    cout << month << '/' << day << '/' << year;
34 } // end function print
35
36 // output Date object to show when its destructor is called
37 Date::~Date()
38 {
39    cout << "Date object destructor for date ";
40    print();
41    cout << endl;
42 } // end ~Date destructor
```

```
43
44 // utility function to confirm proper day value based on
45 // month and year; handles leap years, too
46 int Date::checkDay( int testDay ) const
47 {
48    static const int daysPerMonth[ 13 ] =
49       { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
50
51    // determine whether testDay is valid for specified month
52    if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
53       return testDay;
54
55    // February 29 check for leap year
56    if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
57       ( year % 4 == 0 && year % 100 != 0 ) ) )
58       return testDay;
59
60    cout << "Invalid day (" << testDay << ") set to 1.\n";
61    return 1; // leave object in consistent state if bad value
62 } // end function checkDay
```

```
1  // Fig. 10.12: Employee.h
2  // Employee class definition.
3  // Member functions defined in Employee.cpp.
4  #ifndef EMPLOYEE_H
5  #define EMPLOYEE_H
6
7  #include "Date.h" // include Date class definition
8
9  class Employee
10 {
11 public:
12    Employee( const char * const, const char * const,
13       const Date &, const Date & );
14    void print() const;
15    ~Employee(); // provided to confirm destruction order
16 private:
17    char firstName[ 25 ];
18    char lastName[ 25 ];
19    const Date birthDate; // composition: member object
20    const Date hireDate; // composition: member object
21 }; // end class Employee
22
23 #endif
```

Parameters to be passed via member initializers to the constructor for class **Date**

const objects of class **Date** as members

```
1  // Fig. 10.13: Employee.cpp
2  // Member-function definitions for class Employee.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // strlen and strncpy prototypes
8  using std::strlen;
9  using std::strncpy;
10
11 #include "Employee.h" // Employee class definition
12 #include "Date.h" // Date class definition
13
14 // constructor uses member initializer list to pass initializer
15 // values to constructors of member objects birthDate and hireDate
16 // [Note: This invokes the so-called "default copy constructor" which the
17 // C++ compiler provides implicitly.]
18 Employee::Employee( const char * const first, const char * const last,
19    const Date &dateOfBirth, const Date &dateOfHire )
20    : birthDate( dateOfBirth ), // initialize birthDate
21      hireDate( dateOfHire ) // initialize hireDate
22 {
23    // copy first into firstName and be sure that it fits
24    int length = strlen( first );
25    length = ( length < 25 ? length : 24 );
26    strncpy( firstName, first, length );
27    firstName[ length ] = '\0';
```

Member initializers that pass arguments to **Date**'s implicit default copy constructor

```
28
29    // copy last into lastName and be sure that it fits
30    length = strlen( last );
31    length = ( length < 25 ? length : 24 );
32    strncpy( lastName, last, length );
33    lastName[ length ] = '\0';
34
35    // output Employee object to show when constructor is called
36    cout << "Employee object constructor: "
37       << firstName << ' ' << lastName << endl;
38 } // end Employee constructor
39 //========================================================================
40 // print Employee object
41 void Employee::print() const
42 {
43    cout << lastName << ", " << firstName << "  Hired: ";
44    hireDate.print();
45    cout << "  Birthday: ";
46    birthDate.print();
47    cout << endl;
48 } // end function print
49
50 // output Employee object to show when its destructor is called
51 Employee::~Employee()
52 {
53    cout << "Employee object destructor: "
54       << lastName << ", " << firstName << endl;
55 } // end ~Employee destructor
```

```
1  // Fig. 10.14: fig10_14.cpp
2  // Demonstrating composition--an object with member objects.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Employee.h" // Employee class definition
8
9  int main()
10 {
11    Date birth( 7, 24, 1949 );
12    Date hire( 3, 12, 1988 );
13    Employee manager( "Bob", "Blue", birth, hire );
14
15    cout << endl;
16    manager.print();
17
18    cout << "\nTest Date constructor with invalid values:\n";
19    Date lastDayOff( 14, 35, 1994 ); // invalid month and day
20    cout << endl;
21    return 0;
22 } // end main
```

Passing objects to a host object constructor

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Blue

Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949

Test Date constructor with invalid values:
Invalid month (14) set to 1.
Invalid day (35) set to 1.
Date object constructor for date 1/1/1994

Date object destructor for date 1/1/1994
Employee object destructor: Blue, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
```

2 Composition: Objects as Members of Classes

常见编程错误：如果成员对象不是用成员初始化器形式进行初始化，并且成员对象的类没有提供默认的构造函数（换言之，成员对象的类定义了一个或多个构造函数，但是没有一个是默认的构造函数），则会产生一个编译错误。

2 Composition: Objects as Members of Classes

性能提示：通过成员初始化器显式地初始化成员对象，可以避免两次初始化成员对象的开销：一次是调用成员对象的默认构造函数，一次是调用成员对象的赋值函数。

软件工程知识：如果一个类将其他类的对象作为其成员，即使将这个成员对象指定为public，也不会破坏该成员对象private成员的封装与隐藏。

3 friend Functions and friend Classes

- 类的友元函数
  - 在类的作用域外定义，不是类的成员函数
  - 具有访问该类非 public 成员的权力
  - 单独的函数或整个类均可声明为其他类的友元
  - 适用于成员函数无法完成某些操作时
  - 可以提高性能

3 friend Functions and friend Classes

- 声明一个函数为另一个类的友元
  - 在类定义中提供函数原型并在前面加上关键字 friend
- 声明一个类为另一个类的友元
  - 如：在ClassOne的定义中放置如下声明

```
friend class ClassTwo;
```

即定义ClassTwo是自己的朋友，因此：

允许ClassTwo访问ClassOne的成员变量，就如同ClassOne自己的函数一样。

3 friend Functions and friend Classes

- 友元需要被授予，而不是索取
  - class B要想成为class A的友元，class A必须显式同意，即在class A中声明class B为自己的友元
  - 或者说如果Class A同意Class B的成员访问自己的成员变量就要主动声明Class B是自己的友元
- 友元关系不是对称的，并且不能传递
  - class A 是 class B 的友元，且 class B 又是 class C 的友元，不能认为 B 是 A 的友元、C 是 B 的友元或 A 是 C 的友元

3 friend Functions and friend Classes

软件工程知识：尽管类定义中有友元函数的原型，但友元函数仍然不是成员函数。

软件工程知识：private、protected和public的成员访问符号与友元关系的声明无关，因此友元关系声明可以放在类定义中的任何位置。

```
1  // Fig. 10.15: fig10_15.cpp
2  // Friends can access private members of a class.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // Count class definition
8  class Count
9  {
10    friend void setX( Count &, int ); // friend declaration
11 public:
12    // constructor
13    Count()
14       : x( 0 ) // initialize x to 0
15    {
16       // empty body
17    } // end constructor Count
18
19    // output x
20    void print() const
21    {
22       cout << x << endl;
23    } // end function print
24 private:
25    int x; // data member
26 }; // end class Count
```

friend function declaration (can appear anywhere in the class)

表示允许setx()函数可以访问类Count的成员函数

使用初始化器对成员变量进行赋值

```
27
28 // function setX can modify private data of Count
29 // because setX is declared as a friend of Count (line 10)
30 void setX( Count &c, int val )
31 {
32    c.x = val; // allowed because setX is a friend of Count.
33 } // end function setX    如果不是友元就不能访问类Count的 Private 成员
34
35 int main()
36 {
37    Count counter; // create Count object
38
39    cout << "counter.x after instantiation: ";
40    counter.print();     // 打印出 x 的值应该是初始化的值 0
41
42    setX( counter, 8 ); // set x using a friend function
43    cout << "counter.x after call to setX friend function: ";
44    counter.print();    // 打印出 x 的值应该是经过友元函数设置后的值 8
45    return 0;
46 } // end main
```

friend function can modify **Count**'s **private** data

Calling a **friend** function; note that we pass the **Count** object to the function

除友元外，类Count没有提供修改其private数据成员的函数，不使用友元就修改不了其private成员x

```
counter.x after instantiation: 0
counter.x after call to setX friend function: 8
```

```cpp
// Fig. 10.16: fig10_16.cpp
// Non-friend/non-member functions cannot access private data of a class.
#include <iostream>
using std::cout;
using std::endl;

// Count class definition (note that there is no friendship declaration)
class Count
{
public:
   // constructor
   Count()
      : x( 0 ) // initialize x to 0
   {
      // empty body
   } // end constructor Count

   // output x
   void print() const
   {
      cout << x << endl;
   } // end function print
private:
   int x; // data member
}; // end class Count
```

```
26
27 // function cannotSetX tries to modify private data of Count,
28 // but cannot because the function is not a friend of Count
29 void cannotSetX( Count &c, int val )
30 {
31    c.x = val; // ERROR: cannot access private member in Count
32 } // end function cannotSetX
33
34 int main()
35 {
36    Count counter; // create Count object
37
38    cannotSetX( counter, 3 ); // cannotSetX is not a friend
39    return 0;
40 } // end main
```

Non-**friend** function cannot access the class's **private** data.

如果让你改错，使该程序正常运行，你知道修改哪里吗？

Borland C++ command-line compiler error message:

```
Error E2247 Fig10_16/fig10_16.cpp 31: 'Count::x' is not accessible in
   function cannotSetX(Count &,int)
```

Microsoft Visual C++.NET compiler error messages:

```
C:\cpphtp5_examples\ch10\Fig10_16\fig10_16.cpp(31) : error C2248: 'Count::x'
   : cannot access private member declared in class 'Count'
        C:\cpphtp5_examples\ch10\Fig10_16\fig10_16.cpp(24) : see declaration
           of 'Count::x'
        C:\cpphtp5_examples\ch10\Fig10_16\fig10_16.cpp(9) : see declaration
           of 'Count'
```

GNU C++ compiler error messages:

```
fig10_16.cpp:24: error: 'int Count::x' is private
fig10_16.cpp:31: error: within this context
```

4 Using the this Pointer

- 成员函数需要知道该处理哪个对象的数据成员
  - 每个对象可以通过一个称为 `this` 的指针来访问它自己的地址
  - 对象的 `this` 指针不是该对象自身的一部分
  - `this` 被编译器作为隐式参数传递给对象的非静态成员函数

4 Using the this Pointer

- 对象可以隐式或显式地使用 this 指针
  - 当直接访问数据成员时表示隐式使用
  - 当使用关键字this时表示显式使用
  - this指针的类型依赖于对象的类型和成员函数是否被声明为const

```cpp
// Fig. 10.17: fig10_17.cpp
// Using the this pointer to refer to object members.
#include <iostream>
using std::cout;
using std::endl;

class Test
{
public:
   Test( int = 0 ); // default constructor
   void print() const;
private:
   int x;
}; // end class Test

// constructor
Test::Test( int value )
   : x( value ) // initialize x to value
{
   // empty body
} // end constructor Test
```

```
22
23 // print x using implicit and explicit this pointers;
24 // the parentheses around *this are required
25 void Test::print() const
26 {
27    // implicitly use the this pointer to access the member x
28    cout << "        x = " << x;
29
30    // explicitly use the this pointer and the arrow operator
31    // to access the member x
32    cout << "\n  this->x = " << this->x;
33
34    // explicitly use the dereferenced this pointer and
35    // the dot operator to access the member x
36    cout << "\n(*this).x = " << ( *this ).x << endl;
37 } // end function print
38
39 int main()
40 {
41    Test testObject( 12 ); // instantiate and initialize testObject
42
43    testObject.print();
44    return 0;
45 } // end main
```

Implicitly(隐) using the **this** pointer to access member **x**

Explicitly(显) using the **this** pointer to access member **x**

Using the dereferenced **this** pointer and the dot operator

```
        x = 12
  this->x = 12
(*this).x = 12
```

4 Using the this Pointer

- 级联成员函数调用
  - 在同一条语句上进行多个函数调用
  - 要求成员函数返回解引用(dereferenced)的this指针时才能使用
  - 例如:
    - ✓ `t.setMinute( 30 ).setSecond( 22 );`
      - ◈ 调用 `t.setMinute( 30 );//显然该处应该返回this指针`
      - ◈ 然后调用 `t.setSecond( 22 );`

```
1  // Fig. 10.18: Time.h
2  // Cascading member function calls.
3
4  // Time class definition.
5  // Member functions defined in Time.cpp.
6  #ifndef TIME_H
7  #define TIME_H
8
9  class Time
10 {
11 public:
12    Time( int = 0, int = 0, int = 0 ); // default constructor
13
14    // set functions (the Time & return types enable cascading)
15    Time &setTime( int, int, int ); // set hour, minute, second
16    Time &setHour( int ); // set hour
17    Time &setMinute( int ); // set minute
18    Time &setSecond( int ); // set second
```

set functions return **`Time &`** to enable cascading

```
15    void setTime( int, int, int ); // set time
16    void setHour( int ); // set hour
17    void setMinute( int ); // set minute
18    void setSecond( int ); // set second
```

```
19
20    // get functions (normally declared const)
21    int getHour() const; // return hour
22    int getMinute() const; // return minute
23    int getSecond() const; // return second
24
25    // print functions (normally declared const)
26    void printUniversal() const; // print universal time
27    void printStandard() const; // print standard time
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. 10.19: Time.cpp
// Member-function definitions for Time class.
#include <iostream>
using std::cout;

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

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

// constructor function to initialize private data;
// calls member function setTime to set variables;
// default values are 0 (see class definition)
Time::Time( int hr, int min, int sec )
{
   setTime( hr, min, sec );
} // end Time constructor

// set values of hour, minute, and second
Time &Time::setTime( int h, int m, int s ) // note Time & return
{
   setHour( h );
   setMinute( m );
   setSecond( s );
   return *this; // enables cascading
} // end function setTime
```

Returning dereferenced **`this`** pointer enables cascading

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

```
29 // set hour value
30 void Time::setHour( int h )
31 {
32    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
33 } // end function setHour
```

```
55
56 // get minute value
57 int Time::getMinute() const
58 {
59    return minute;
60 } // end function getMinute
61
62 // get second value
63 int Time::getSecond() const
64 {
65    return second;
66 } // end function getSecond
67
68 // print Time in universal-time format (HH:MM:SS)
69 void Time::printUniversal() const
70 {
71    cout << setfill( '0' ) << setw( 2 ) << hour << ":"
72       << setw( 2 ) << minute << ":" << setw( 2 ) << second;
73 } // end function printUniversal
74
75 // print Time in standard-time format (HH:MM:SS AM or PM)
76 void Time::printStandard() const
77 {
78    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
79       << ":" << setfill( '0' ) << setw( 2 ) << minute
80       << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
81 } // end function printStandard
```

```
1  // Fig. 10.20: fig10_20.cpp
2  // Cascading member function calls with the this pointer.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Time.h" // Time class definition
8
9  int main()
10 {
11    Time t; // create Time object
12
13    // cascaded function calls
14    t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
15
16    // output time in universal and standard formats
17    cout << "Universal time: ";
18    t.printUniversal();
19
20    cout << "\nStandard time: ";
21    t.printStandard();
22
23    cout << "\n\nNew standard time: ";
24
25    // cascaded function calls
26    t.setTime( 20, 20, 20 ).printStandard();
27    cout << endl;
28    return 0;
29 } // end main
```

t.setHour(18);

t.setMinute(30);

t.setSecond(22);

Cascaded function calls using the reference returned by one function call to invoke the next

注意该处级联的写法，**printStandard()函数不返回 t类型的引用，所以不能写到前面**

```
Universal time: 18:30:22
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM
```

5 Dynamic Memory Management with Operators new and delete (使用new与delete进行动态内存管理)

- 问题的来源
  - 数组等数据结构在定义时需要明确大小，数组长度不能是变量。如果定义短了则不能完整存放数据，定义长了会造成浪费，能否采用可变长的方法来处理这种结构？

C++内存格局

- 全局数据区 (data area)
- 代码区 (code area)
- 栈区 (stack area)
- 堆区 (heap area)

- 全局变量、静态数据、常量存放在全局数据区;
- 所有类成员函数和非成员函数代码存放在代码区;
- 为运行函数而分配的局部变量、函数参数、返回数据、返回地址等存放在栈区;
- 其余的空间都被称为堆区。 (动态内存管理只能是堆区)

堆内存的分配与释放

当程序运行到需要一个动态分配的变量或对象时，必须向系统申请取得**堆**中的一块大小足够的存贮空间，用于存贮该变量或对象。当不再使用该变量或对象时，也就是它的生命结束时，要**显式释放**它所占用的存贮空间，这样系统就能对该堆空间进行再次分配，做到重复使用有限的资源。

在C++中，申请和释放堆中分配的存贮空间，分别使用new和delete的两个运算符来完成，其使用的格式如下：

指针变量名=new 类型名(初始化式);

delete 指针名;

new 运算符返回的是一个指向所分配类型变量（对象）的指针。对所创建的变量或对象，都是通过该指针来间接操作的，而动态创建的对象本身没有名字。

堆内存的分配与释放

new 类型名T（初值列表）

功能：在程序执行期间，申请用于存放T类型对象的内存空间，
并依据初值列表调用合适的构造函数。

结果值：成功：T类型的指针，指向新分配的内存。

失败：0（NULL）

5 Dynamic Memory Management with Operators new and delete

- Operator delete
  - 动态销毁分配的对象内存
  - 调用对象的析构函数
  - 释放的内存可供程序的其他对象来使用

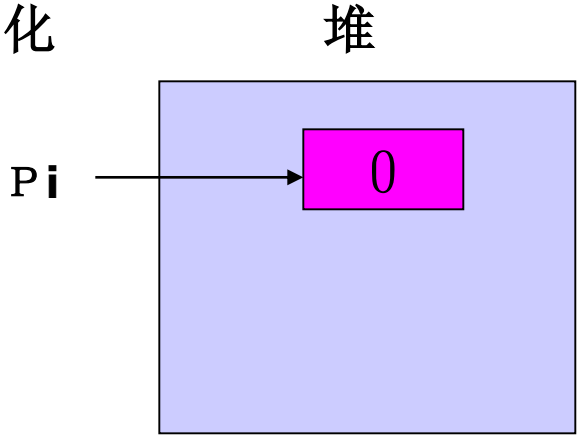
5 Dynamic Memory Management with Operators new and delete

- 通过 new 来初始化分配的对象
  - 为新建立的基本类型变量提供初始值。例如：
    - `double *ptr = new double( 3.14159 );`
  - 为新建立的对象进行初始化。例如：
    - `Time *timePtr = new Time( 12, 45, 0 );`

1．用初始化式(initializer)来显式初始化

例如：

```
int *pi=new int(0);
```



2．当pi生命周期结束时，必须释放pi所指向的目标：

```
delete pi;
```

注意这时释放了pi所指的目标的内存空间，也就是撤销了该目标，称动态内存释放（dynamic memory deallocation），但指针pi本身并没有撤销，它自己仍然存在，该指针所占内存空间并未释放。

5 Dynamic Memory Management with Operators new and delete

- new 运算符可以用来动态分配数组
  - int *gradesArray = new int[10];
  - int *gradesArray = new int[m];
    - ✓因为是动态产生，因此m可以在该句执行之前计算而定，而不需要在编译时就确定

5 Dynamic Memory Management with Operators new and delete

- 删除动态分配的数组
  - `delete [] gradesArray;`
  - 如果 `gradesArray` 为对象数组
    - ✓ 首先调用每个对象的析构函数
    - ✓ 然后释放内存
  - 如果语句中没有包括（`[]`）并且 `gradesArray` 指向一个对象数组
    - ✓ 只有第一个对象的析构函数被调用

动态分配的三个特点

- 首先，变量n在编译时没有确定的值，而是在运行中输入，按运行时所需分配堆空间，这一点是动态分配的优点，可克服数组“大开小用”的弊端。delete []pc是将n个字符的空间释放，而用delete pc则只释放了一个字符的空间；
- 如果有一个char *pc1，令pc1=p，同样可用delete [] pc1来释放该空间。尽管C++不对数组作边界检查，但在堆空间分配时，对数组分配空间大小是纪录在案的。
- 没有初始化式（initializer），不可对数组初始化。

指针使用的4个问题

- 动态分配失败。返回一个空指针（NULL），表示发生了异常，堆资源不足，分配失败。
- 指针删除与堆空间释放。删除一个指针p（delete p;）实际意思是删除了p所指的目标（变量或对象等），释放了它所占的堆空间，而不是删除p本身，释放堆空间后，p成了空悬指针（p仍占有空间，但不知指向哪里）

续

- 内存泄漏（memory leak）和重复释放。
  - new与delete 是配对使用的， delete只能释放堆空间。
  - 如果new返回的指针值丢失，则所分配的堆空间无法回收，称内存泄漏；
  - 同一空间重复释放也是危险的，所以必须妥善保存new返回的指针，以保证不发生内存泄漏，也必须保证不会重复释放堆内存空间。
- 动态分配的变量或对象的生命期。
  - 无名对象，它的生命期并不依赖于建立它的作用域，比如在函数中建立的动态对象在函数返回后仍可使用。我们也称堆空间为自由空间（free store）就是这个原因。

5 Dynamic Memory Management with Operators new and delete

常见编程错误：删除数组时，用delete代替delete[ ]将导致运行时的逻辑错误。为保证数组中的每个对象都接受一个析构函数调用，数组生成的内存空间要用delete[ ]运算符删除。各个元素生成的内存空间则用delete运算符删除。

6 static Class Members

- static 数据成员
  - 所有对象共用一份数据拷贝，而不是以前介绍的每个对象一份自己的数据信息
  - 这一份数据是“整个类范围上”的信息
  - 以关键字 `static` 声明
  - 可以看作全局变量，但是属于类作用域
  - 可以被声明为 `public`，`private` 或者 `protected`

6 static Class Members

- static 数据成员
  - 如果是基本类型的 static 成员
    - ✓ 默认被初始化为 0
    - ✓ static 数据成员只能被初始化一次，不能根据不同需要初始化为不同的值
  - const static int 或 enum 类型的数据成员
    - ✓ 可以在类定义中声明时初始化

6 static Class Members

- static 数据成员
  - 如果是其他 static 数据成员
    - ✓ 必须在文件作用域内定义（即：在类定义外）
    - ✓ 在定义的同时需要初始化
  - 具有默认构造函数的 static 成员对象
    - ✓ 因为它们的默认构造函数会被调用，所以不必初始化

6 static Class Members

- static 数据成员
  - 对象不存在时就已存在(早于构造函数)
    - ✓当对象不存在时就可以访问 public static 数据成员
      - ◇例如: Martian::martianCount
  - 也可以通过该类的对象进行访问
    - ◇例如: myMartian.martianCount

6 static Class Members

- 声明一个成员函数为 static，下面都是正确的：
  - 如果它不访问类的 non-static 数据成员或 non-static 成员函数
  - 一个 static 成员函数没有 this 指针
  - static 数据成员和 static 成员函数独立于类的任何对象存在
  - 当一个 static 成员函数被调用时，内存中可能没有任何对象

```
1  // Fig. 10.21: Employee.h
2  // Employee class definition.
3  #ifndef EMPLOYEE_H
4  #define EMPLOYEE_H
5
6  class Employee
7  {
8  public:
9     Employee( const char * const, const char * const ); // constructor
10    ~Employee(); // destructor
11    const char *getFirstName() const; // return first name
12    const char *getLastName() const; // return last name
13
14    // static member function
15    static int getCount(); // return number of objects instantiated
16 private:
17    char *firstName;
18    char *lastName;
19
20    // static data
21    static int count; // number of objects instantiated
22 }; // end class Employee
23
24 #endif
```

静态的成员函数只能访问静态的数据成员

静态的数据成员为所有对象所调用

```cpp
// Fig. 10.22: Employee.cpp
// Member-function definitions for class Employee.
#include <iostream>
using std::cout;
using std::endl;

#include <cstring> // strlen and strcpy prototypes
using std::strlen;
using std::strcpy;

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

// define and initialize static data member at file scope
int Employee::count = 0;

// define static member function that returns number of
// Employee objects instantiated (declared static in Employee.h)
int Employee::getCount()
{
   return count;
} // end static function getCount
```

该处赋值，表示在文件的范围内有效，即在该cpp文件范围内有效

static member function can access only **static** data, because the function might be called when no objects exist

```
22
23 // constructor dynamically allocates space for first and last name and
24 // uses strcpy to copy first and last names into the object
25 Employee::Employee( const char * const first, const char * const last )
26 {
27    firstName = new char[ strlen( first ) + 1 ];
28    strcpy( firstName, first );
29
30    lastName = new char[ strlen( last ) + 1 ];
31    strcpy( lastName, last );
32
33    count++; // increment static count of employees   每构造一次就加 1
34
35    cout << "Employee constructor for " << firstName
36       << ' ' << lastName << " called." << endl;
37 } // end Employee constructor
38 //-------------------------------------------------------------------------------------------
39 // destructor deallocates dynamically allocated memory
40 Employee::~Employee()
41 {
42    cout << "~Employee() called for " << firstName
43       << ' ' << lastName << endl;
44
45    delete [] firstName; // release memory
46    delete [] lastName; // release memory
47
48    count--; // decrement static count of employees   每析构一次就减 1
49 } // end ~Employee destructor
```

Dynamically allocating **char** arrays

Non-**static** member function (i.e., constructor) can modify the class's **static** data members

Deallocating memory reserved for arrays

```
50
51 // return first name of employee
52 const char *Employee::getFirstName() const
53 {
54    // const before return type prevents client from modifying
55    // private data; client should copy returned string before
56    // destructor deletes storage to prevent undefined pointer
57    return firstName;
58 } // end function getFirstName
59
60 // return last name of employee
61 const char *Employee::getLastName() const
62 {
63    // const before return type prevents client from modifying
64    // private data; client should copy returned string before
65    // destructor deletes storage to prevent undefined pointer
66    return lastName;
67 } // end function getLastName
```

返回类型前加const表示该函数不能用于修改 private 类型的变量

```cpp
// Fig. 10.23: fig10_23.cpp
// Driver to test class Employee.
#include <iostream>
using std::cout;
using std::endl;

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

int main()
{
   // use class name and binary scope resolution operato
   // access static number function getCount
   cout << "Number of employees before instantiation of any objects is "
      << Employee::getCount() << endl; // use class name

   // use new to dynamically create two new Employees
   // operator new also calls the object's constructor
   Employee *e1Ptr = new Employee( "Susan", "Baker" );
   Employee *e2Ptr = new Employee( "Robert", "Jones" );

   // call getCount on first Employee object
   cout << "Number of employees after objects are instantiated is "
      << e1Ptr->getCount();

   cout << "\n\nEmployee 1: "
      << e1Ptr->getFirstName() << " " << e1Ptr->getLastName()
      << "\nEmployee 2: "
      << e2Ptr->getFirstName() << " " << e2Ptr->getLastName() << "\n\n";
```

Calling **static** member function using class name and binary scope resolution operator。

此时还没有实例化后的对象

Dynamically creating **Employee**s with **new**

Calling a **static** member function through a pointer to an object of the class

```
29
30    delete e1Ptr; // deallocate memory
31    e1Ptr = 0; // disconnect pointer from free-store space
32    delete e2Ptr; // deallocate memory
33    e2Ptr = 0; // disconnect pointer from free-store space
34
35    // no objects exist, so call static member function getCount again
36    // using the class name and the binary scope resolution operator
37    cout << "Number of employees after objects are deleted is "
38       << Employee::getCount() << endl;
39    return 0;
40 } // end main
```

Releasing memory to which a pointer points

Disconnecting a pointer from any space in memory

```
Number of employees before instantiation of any objects is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after objects are instantiated is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Susan Baker
~Employee() called for Robert Jones
Number of employees after objects are deleted is 0
```

6 static Class Members

软件工程知识：即使还没有实例化任何对象，类的静态数据成员和成员函数就已经存在并可使用。

良好编程习惯：删除动态分配的内存后，将指向该内存的指针设置为0，以切断指针与前面已分配内存的连接。

试问：

- const 与 static 有什么区别？
  - const 定义的变量或对象不能被改变
  - const 的作用范围与对象中的定义有关
  - const 的public变量由构造函数初始化；private变量使用成员初始化器进行初始化
  - static 定义的变量在所有对象中只有一份，但可以被改变也可以被访问
  - static 的作用范围在“类”级
  - static 的变量可以在构造函数外被初始化。

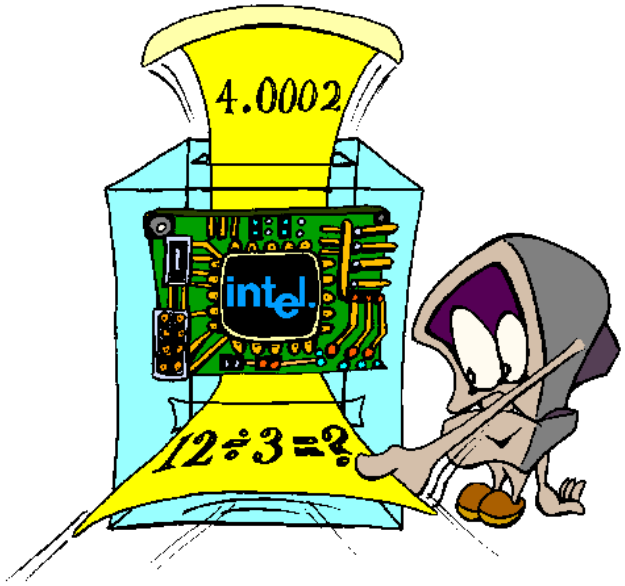
7 Data Abstraction and Information Hiding

- 数据抽象
  - 客户关心类提供的功能，而不关心功能是如何实现的。就如同一般人只关心自行车提供的功能而不关心是如何实现这些功能的，比如复杂的链条传动机制等。
    - ✓ 例如：堆栈类的客户无需关心堆栈的实现
  - 程序员不应该编写依赖于实现细节的代码
- 信息隐藏
  - 一个类通常对客户隐藏实现细节


4
12÷3=?


4
12÷3=?


4
12÷3=?


4.0002
intel
12÷3=?

8 Container Classes and Iterators

- 容器类（也称为集合类）
  - 为保存一组对象而设计的类
  - 通常提供诸如：插入，删除，查找，排序等服务
  - 数组，堆栈，队列，树，链表等都是容器类

8 Container Classes and Iterators

- 迭代器对象(迭代器)
  - 通常与容器类相关
  - 遍历(“穿行于”)集合，返回下一个元素或对下一个元素进行一些操作，替用户完成遍历的工作
  - 一个容器类可以有几个迭代器
  - 每个迭代器维护自身的位置信息

9 Proxy Classes（代理类）

- 软件工程两个基本原则
  - 接口与实现的分离
  - 隐藏实现细节
- 头文件包含了部分类的实现和提示信息
  - 类的私有数据成员(private)出现在头文件中
  - 潜在的向客户暴露了专有信息

9 Proxy Classes

- 代理类
  - 向客户隐藏包含私有数据成员在内的信息
  - 客户只知道类提供的公有接口
  - 使得客户在使用类的服务时无法访问到类的实现细节

```
1  // Fig. 10.24: Implementation.h
2  // Header file for class Implementation
3
4  class Implementation
5  {
6  public:
7     // constructor
8     Implementation( int v )
9        : value( v ) // initialize value with v
10    {
11       // empty body
12    } // end constructor Implementation
13
14    // set value to v
15    void setValue( int v )
16    {
17       value = v; // should validate v
18    } // end function setValue
19
20    // return value
21    int getValue() const
22    {
23       return value;
24    } // end function getValue
25 private:
26    int value; // data that we would like to hide from the client
27 }; // end class Implementation
```

Class definition for the class that contains the proprietary implementation we would like to hide

The data we would like to hide from the client

```
1  // Fig. 10.25: Interface.h
2  // Header file for class Interface
3  // Client sees this source code, but the source code does not reveal
4  // the data layout of class Implementation.
5
6  class Implementation; // forward class declaration required by line 17
7
8  class Interface
9  {
10 public:
11    Interface( int ); // constructor
12    void setValue( int ); // same public interface as
13    int getValue() const; // class Implementation has
14    ~Interface(); // destructor
15 private:
16    // requires previous forward declaration (line 6)
17    Implementation *ptr;
18 }; // end class Interface
```

Declares **Implementation** as a data type without including the class's complete header file

public interface between client and hidden class

Using a pointer allows us to hide implementation details of class **Implementation**

```cpp
// Fig. 10.26: Interface.cpp
// Implementation of class Interface--client receives this file only
// as precompiled object code, keeping the implementation hidden.
#include "Interface.h" // Interface class definition
#include "Implementation.h" // Implementation class definition

// constructor
Interface::Interface( int v )
   : ptr ( new Implementation( v ) ) // initialize ptr to point to
{                                    // a new Implementation object
   // empty body
} // end Interface constructor

// call Implementation's setValue function
void Interface::setValue( int v )
{
   ptr->setValue( v );
} // end function setValue

// call Implementation's getValue function
int Interface::getValue() const
{
   return ptr->getValue();
} // end function getValue

// destructor
Interface::~Interface()
{
   delete ptr;
} // end ~Interface destructor
```

Only location where **`Implementation.h`** is included with **`#include`**

Setting the value of the hidden data via a pointer

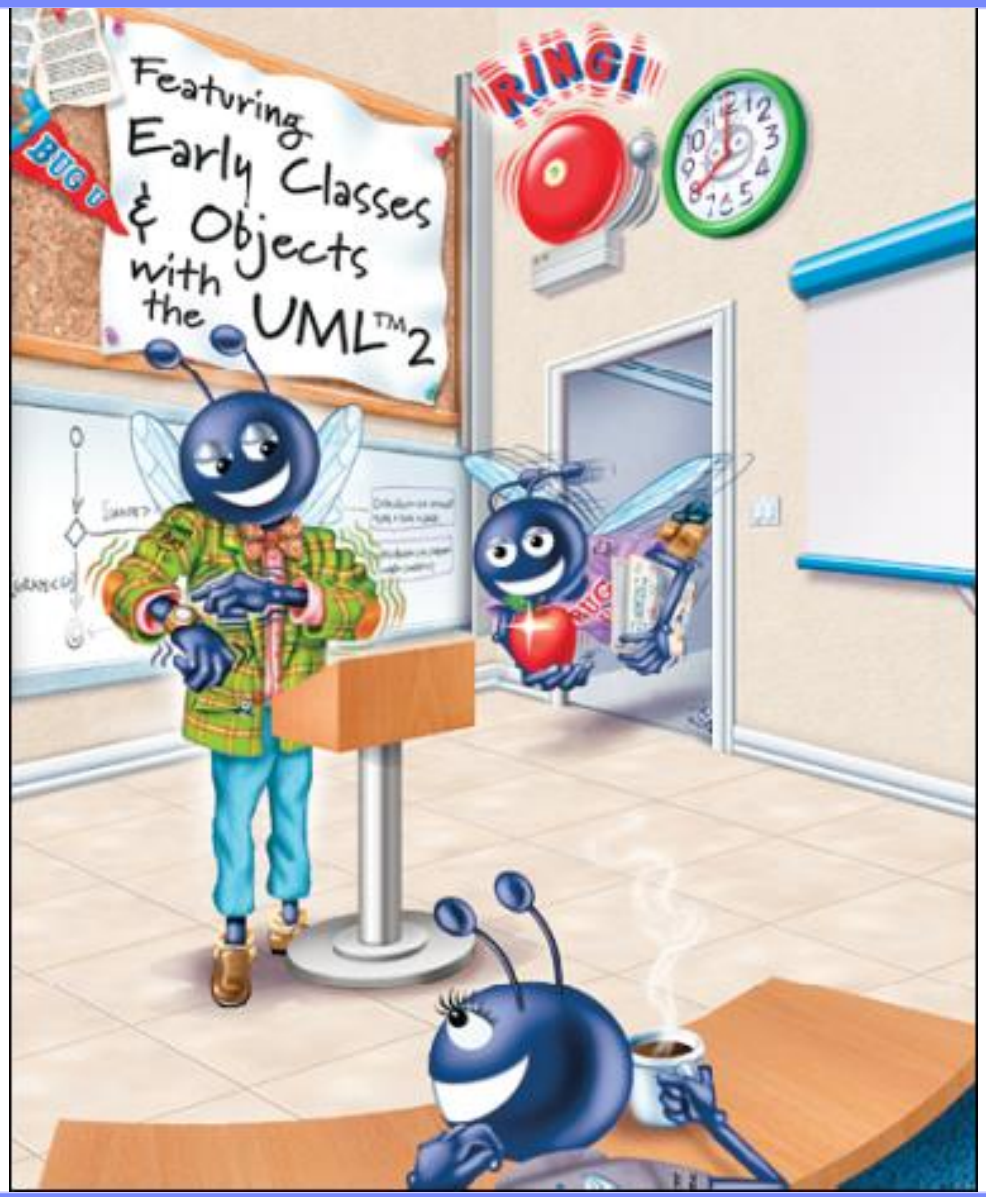
Getting the value of the hidden data via a pointer

```
1  // Fig. 10.27: fig10_27.cpp
2  // Hiding a class's private data with a proxy class.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Interface.h" // Interface class definition
8
9  int main()
10 {
11    Interface i( 5 ); // create Interface object
12
13    cout << "Interface contains: " << i.getValue()
14       << " before setValue" << endl;
15
16    i.setValue( 10 );
17
18    cout << "Interface contains: " << i.getValue()
19       << " after setValue" << endl;
20    return 0;
21 } // end main
```

Only the header file for **Interface** is included in the client code—no mention of the existence of a separate class called **Implementation**

```
Interface contains: 5 before setValue
Interface contains: 10 after setValue
```

END !



Thank you!