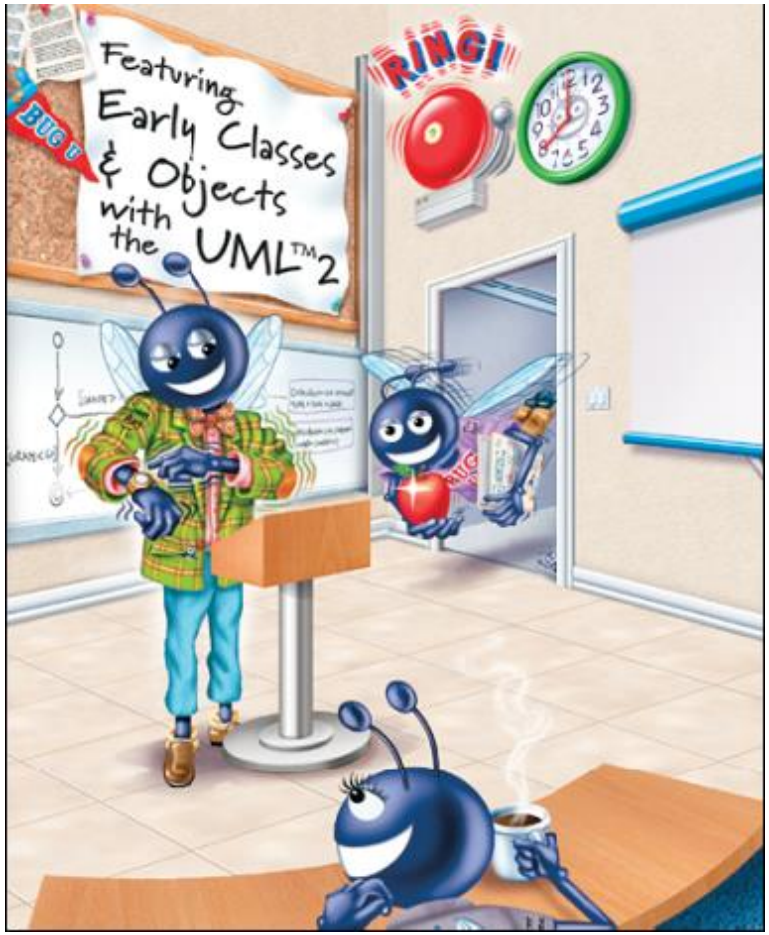
C++程序设计



上节课内容回顾

1. 类成员的访问
2. 访问函数和工具函数
3. 析构函数
4. 默认赋值函数

思考题：

修改 Time 类，包含一个 tick 成员函数，将时间递增 1 秒。在循环中测试 tick 成员函数，以标准时间格式打印。要保证测试到下列情况：

a) 递增到下一分钟

b) 递增到下一小时

c) 递增到下一天

思考题：

```
void Time::setHour( int h )
{
   hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
}

void Time::setMinute( int m )
{
   minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
}

void Time::setSecond( int s )
{
   second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
}
```

思考题：

```
void Time::tick()
{
   setSecond( getSecond() + 1 ); // increment second by 1

   if ( getSecond() == 0 )
   {
      setMinute( getMinute() + 1 ); // increment minute by 1

      if ( getMinute() == 0 )
         setHour( getHour() + 1 ); // increment hour by 1
   }
}
```

```
const int MAX_TICKS = 30;

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

   t.setTime( 23, 59, 57 ); // set time

    for ( int ticks = 1; ticks < MAX_TICKS; ++ticks )
   {
      t.printStandard(); // invokes function printStandard
      cout << endl;
      t.tick(); // invokes function tick
   }

   return 0;
}
```

第九讲 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
- 使用 this 指针
- new 和 delete
- static 数据成员和成员函数
- 容器类、代理类

1. const Objects and const Member Functions

- const 对象
  - 关键字 const
  - 声明对象不能被修改
  - 修改对象会产生编译错误

1. const Objects and const Member Functions

- const 成员函数
  - const 对象只能调用 const 成员函数
  - const 成员函数不能修改对象
  - 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

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。

```
class Time
{
public:
   Time( int = 0, int = 0, int = 0 ); // default constructor
   void setTime( int, int, int ); // set time
   ......
   int getHour() const; // return hour
   int getMinute() const; // return minute
   int getSecond() const; // return second
   void printUniversal() const; // print universal time
   void printStandard(); // print standard time (should be const)
private:
   int hour; // 0 - 23 (24-hour clock format)
   int minute; // 0 – 59
   int second; // 0 – 59
};
```

```
// return hour value
int Time::getHour() const // get functions should be const
{
   return hour;
} // end function getHour

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

```
int main()
{
   Time wakeUp( 6, 45, 0 ); // non-constant object
   const Time noon( 12, 0, 0 ); // constant object

   wakeUp.setHour( 18 );  // non-const   non-const

   noon.setHour( 12 );    // const       non-const

   wakeUp.getHour();      // non-const   const
   noon.getMinute();      // const       const
   noon.printUniversal(); // const       const

   noon.printStandard();  // const       non-const
   return 0;
} // end main
```

1. const Objects and const Member Functions

- **Member initializer**
  - 需要进行初始化
    - const 数据成员
    - 为引用类型的数据成员
  - 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

```
class Increment
{
public:
   Increment( int c = 0, int i = 1 ); // default constructor

   void addIncrement()
   {
      count += increment;
   } // end function addIncrement

   void print() const; // prints count and increment
private:
   int count;
   const int increment; // const data member
}; // end class Increment
```

```
// constructor
Increment::Increment( int c, int i )
   : count( c ), // initializer for non-const member
     increment( i ) // required initializer for const member
{
   // empty body
} // end constructor Increment
```

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是语法错误。

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值来初始化，不能用赋值语句。

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。

2. Composition: Objects as Members of Classes

- **Composition（组合）**
  - 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

```
class Date
{
public:
   Date( int = 1, int = 1, int = 1900 ); // default constructor
   void print() const; // print date in month/day/year format
   ~Date(); // provided to confirm destruction order
private:
   int month; // 1-12 (January-December)
   int day; // 1-31 based on month
   int year; // any year

   // utility function to check if day is proper for month and year
   int checkDay( int ) const;
}; // end class Date
```

```
class Employee
{
public:
   Employee( const char * const, const char * const,
      const Date &, const Date & );
   void print() const;
   ~Employee(); // provided to confirm destruction order
private:
   char firstName[ 25 ];
   char lastName[ 25 ];
   const Date birthDate; // composition: member object
   const Date hireDate; // composition: member object
}; // end class Employee
```

```
// constructor uses member initializer list to pass initializer
// values to constructors of member objects birthDate and hireDate
// [Note: This invokes the so-called "default copy constructor" which the
// C++ compiler provides implicitly.]
Employee::Employee( const char * const first, const char * const last,
   const Date &dateOfBirth, const Date &dateOfHire )
   : birthDate( dateOfBirth ), // initialize birthDate
     hireDate( dateOfHire ) // initialize hireDate
{
   // copy first into firstName and be sure that it fits
   int length = strlen( first );
   length = ( length < 25 ? length : 24 );
   ......
}
```

```
int main()
{
   Date birth( 7, 24, 1949 );
   Date hire( 3, 12, 1988 );
   Employee manager( "Bob", "Blue", birth, hire );

   cout << endl;
   manager.print();

   cout << "\nTest Date constructor with invalid values:\n";
   Date lastDayOff( 14, 35, 1994 ); // invalid month and day
   cout << endl;
   return 0;
} // end main
```

```
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

- 字符串拷贝
- birthDate, hireDate 的构建
- 初始化器的使用
- 引用与值传递
- 拷贝构造函数

思考题：

- 写一个拷贝构造函数，观察使用初始化器初始化成员对象时，成员对象的构造过程

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），则将产生一个编译错误。

2. Composition: Objects as Members of Classes

性能提示：通过成员初始化值显式初始化成员对象，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，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
  - 可以提高性能
  - 适用于成员函数无法完成某些操作时

3. friend Functions and friend Classes

- 声明一个函数为一个类的友元
  - 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;

    ClassTwo 的所有成员函数成为 ClassOne 的友元

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 必须显式声明 class B 为它的友元
- 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 的友元或者 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。

```
class Count
{
   friend void setX( Count &, int ); // friend declaration
public:
   Count() : x( 0 )
   {
      // empty body
   } // end constructor Count

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

```
void setX( Count &c, int val )
{
   c.x = val; // allowed because setX is a friend of Count
} // end function setX

int main()
{
   Count counter; // create Count object

   cout << "counter.x after instantiation: ";
   counter.print();

   setX( counter, 8 ); // set x using a friend function
   cout << "counter.x after call to setX friend function: ";
   counter.print();
   return 0;
} // end main
```

4. Using the this Pointer

- 成员函数知道在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 指针不是对象的一部分
  - 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隐式使用
  - 当使用关键字 this 时显式使用
  - 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

```
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
```

```
void Test::print() const
{
   // implicitly use the this pointer to access the member x
   cout << "        x = " << x;

   // explicitly use the this pointer and the arrow operator
   // to access the member x
   cout << "\n  this->x = " << this->x;

   // explicitly use the dereferenced this pointer and
   // the dot operator to access the member x
   cout << "\n(*this).x = " << ( *this ).x << endl;
} // end function print
```

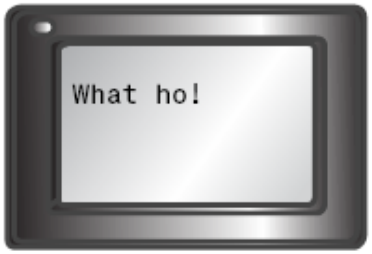
4. Using the this Pointer

- 级联成员函数调用
  - 在同一条语句上进行多个函数调用
  - 使成员函数返回解引用的 this 指针
  - 例如：
    - t.setMinute(30).setSecond(22);
      - 调用 t.setMinute(30);
      - 然后调用 t.setSecond(22);

4. Using the this Pointer

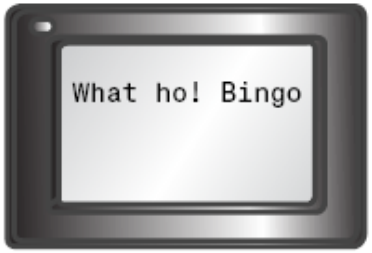
```
char * name = "Bingo"
cout << "What ho! " << name << "!\n";
```

sends What ho!
to output buffer and
returns cout



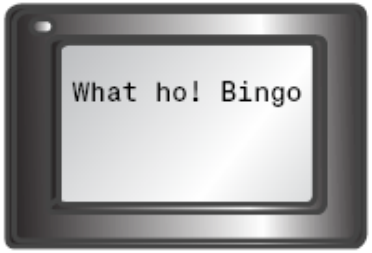

```
cout << name << "!\n";
```

sends Bingo
to output buffer and
returns cout




```
cout << "!\n";
```

sends !\n to output
buffer and returns
cout (the return
value is unused)



```
Time &Time::setHour( int h ) // note Time & return
{
   hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
   return *this; // enables cascading
} // end function setHour

Time &Time::setMinute( int m ) // note Time & return
{
   minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
   return *this; // enables cascading
} // end function setMinute

Time &Time::setSecond( int s ) // note Time & return
{
   second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
   return *this; // enables cascading
} // end function setSecond
```

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

   // cascaded function calls
   t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );

   ......
}
```

4. Using the this Pointer

- 级联成员函数调用
  - 如不返回引用，结果如何？
  - 此时return *this如何工作？

思考题：

- 观察不返回引用的时候，程序的输出结果是怎么样的？分析原因。

5. Dynamic Memory Management with Operators new and delete

- **Operator new**
  - 在运行期间为对象分配内存
  - 调用构造函数初始化对象
  - 返回 new 右侧所声明的类型指针

5. Dynamic Memory Management with Operators new and delete

- **Free store**
  - 有时称为堆（heap）
  - 为程序运行期间分配的对象存储区域

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
  - 例如：
    - double *ptr = new double(3.14159);
  - 例如：
    - Time *timePtr = new Time(12, 45, 0);

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];

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

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
    - 类信息
  - 以关键字 static 声明

6. static Class Members

- static 数据成员
  - 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
  - 基本类型的 static 成员
    - 默认被初始化为 0
    - 如果需要不同的初始值，static 数据成员只能被初始化一次
  - 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
  - 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
    - 当对象不存在时访问 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
  - 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

```
class Employee
{
public:
   Employee( const char * const, const char * const ); // constructor
   ~Employee(); // destructor
   const char *getFirstName() const; // return first name
   const char *getLastName() const; // return last name

   static int getCount(); // return number of objects instantiated
private:
   char *firstName;
   char *lastName;

   static int count; // number of objects instantiated
}; // end class Employee
```

```
// 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
```

```
// constructor dynamically allocates space for first and last name and
// uses strcpy to copy first and last names into the object
Employee::Employee( const char * const first, const char * const last )
{
   firstName = new char[ strlen( first ) + 1 ];
   strcpy( firstName, first );

   lastName = new char[ strlen( last ) + 1 ];
   strcpy( lastName, last );

   count++; // increment static count of employees

   cout << "Employee constructor for " << firstName
      << ' ' << lastName << " called." << endl;
} // end Employee constructor
```

```
// destructor deallocates dynamically allocated memory
Employee::~Employee()
{
   cout << "~Employee() called for " << firstName
      << ' ' << lastName << endl;

   delete [] firstName; // release memory
   delete [] lastName; // release memory

   count--; // decrement static count of employees
} // end ~Employee destructor
```

```
int main()
{
   cout << "Number of employees before instantiation of any objects is "
      << Employee::getCount() << endl; // use class name

   Employee *e1Ptr = new Employee( "Susan", "Baker" );
   Employee *e2Ptr = new Employee( "Robert", "Jones" );

   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";
```

```
   delete e1Ptr; // deallocate memory
   e1Ptr = 0; // disconnect pointer from free-store space
   delete e2Ptr; // deallocate memory
   e2Ptr = 0; // disconnect pointer from free-store space

   // no objects exist, so call static member function getCount again
   // using the class name and the binary scope resolution operator
   cout << "Number of employees after objects are deleted is "
      << Employee::getCount() << endl;
   return 0;
} // end main
```

6. static Class Members

- 与前面例子中字符串处理的区别？
- Static 数据成员的声明和定义
- 构造和析构函数的调用

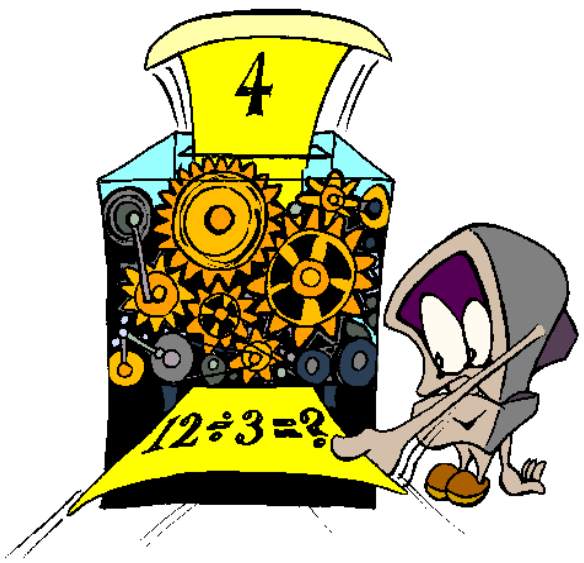
6. static Class Members

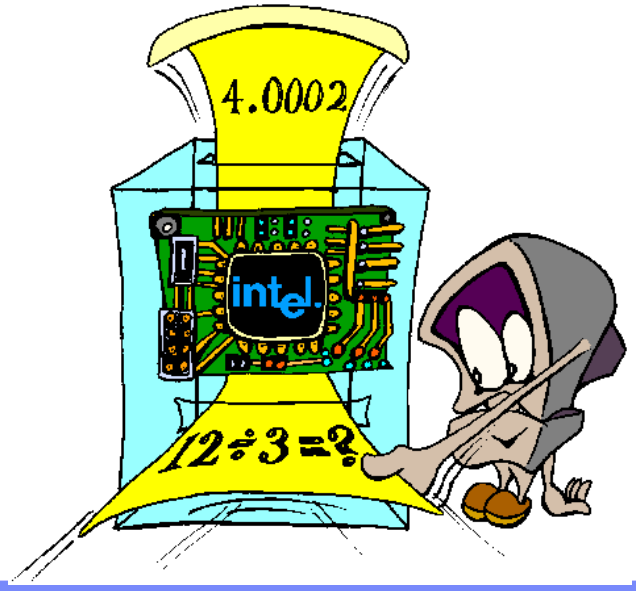
软件工程知识：即使还没有实例化任何对象，类的静态数据成员和成员函数就已经存在并可使用。

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。

7. Data Abstraction and Information Hiding

- 信息隐藏
  - 一个类通常对客户隐藏实现细节
- 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
    - ◈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





8. Container Classes and Iterators

- 容器类（也称为集合类）
  - 类用来设计包含对象的集合
  - 通常提供诸如：插入，删除，查找，排序等服务
  - 例如：数组，堆栈，队列，树，链表

8. Container Classes and Iterators

- 迭代对象或简单的迭代器
  - 通常与容器类相关
  - 遍历集合，返回下一个元素或对下一个元素进行一些操作
  - 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

- 头文件包含了部分类的实现和提示信息
  - 例如，类的私有数据成员出现在头文件中
  - 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

```
class Implementation
{
public:
   Implementation( int v ) : value( v )
   {
   } // end constructor Implementation
   void setValue( int v )
   {
      value = v; // should validate v
   } // end function setValue
   int getValue() const
   {
      return value;
   } // end function getValue
private:
   int value; // data that we would like to hide from the client
}; // end class Implementation
```

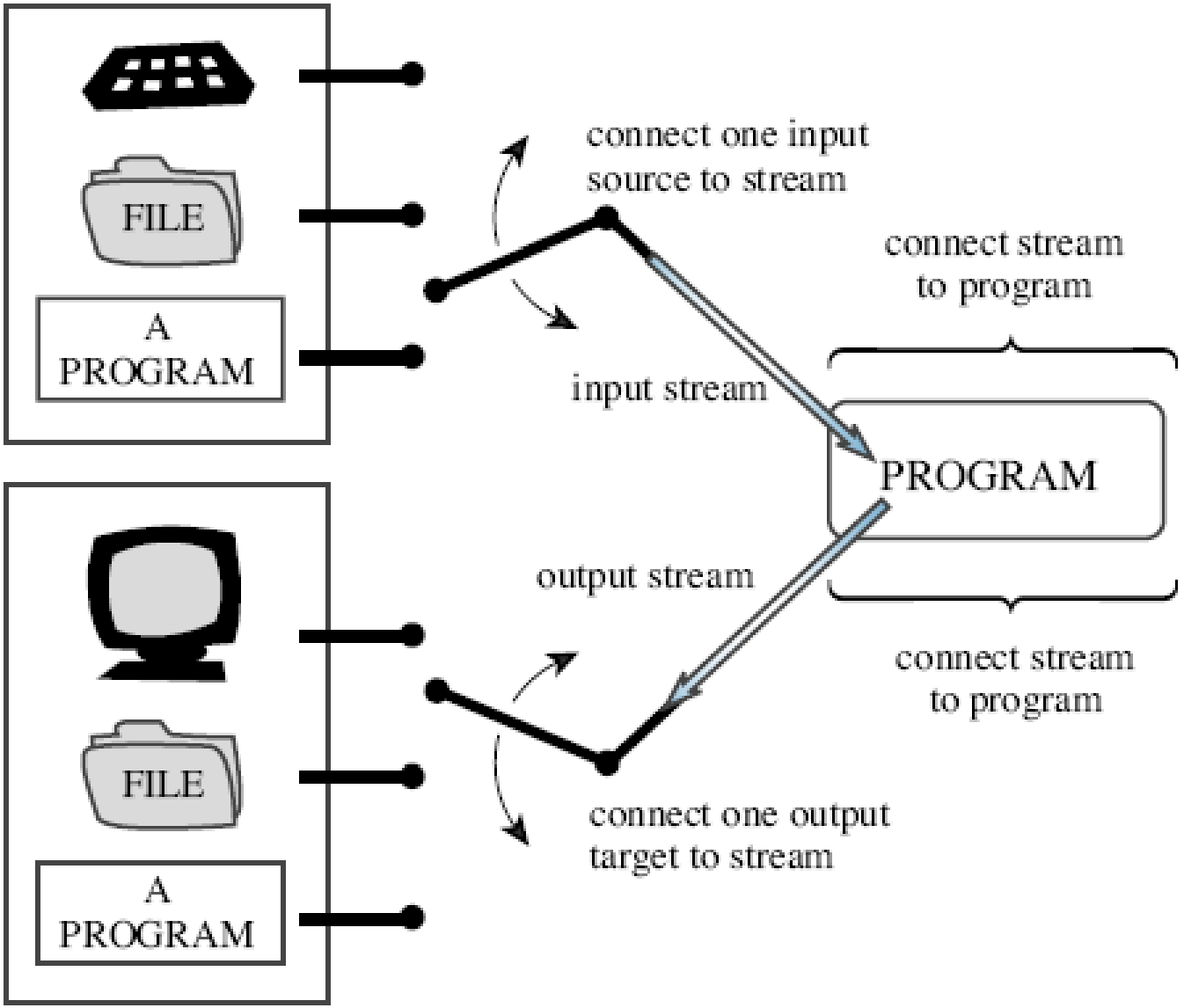
```
class Implementation; // forward class declaration

class Interface
{
public:
   Interface( int ); // constructor
   void setValue( int ); // same public interface as
   int getValue() const; // class Implementation has
   ~Interface(); // destructor
private:
   // requires previous forward declaration
   Implementation *ptr;
}; // end class Interface
```

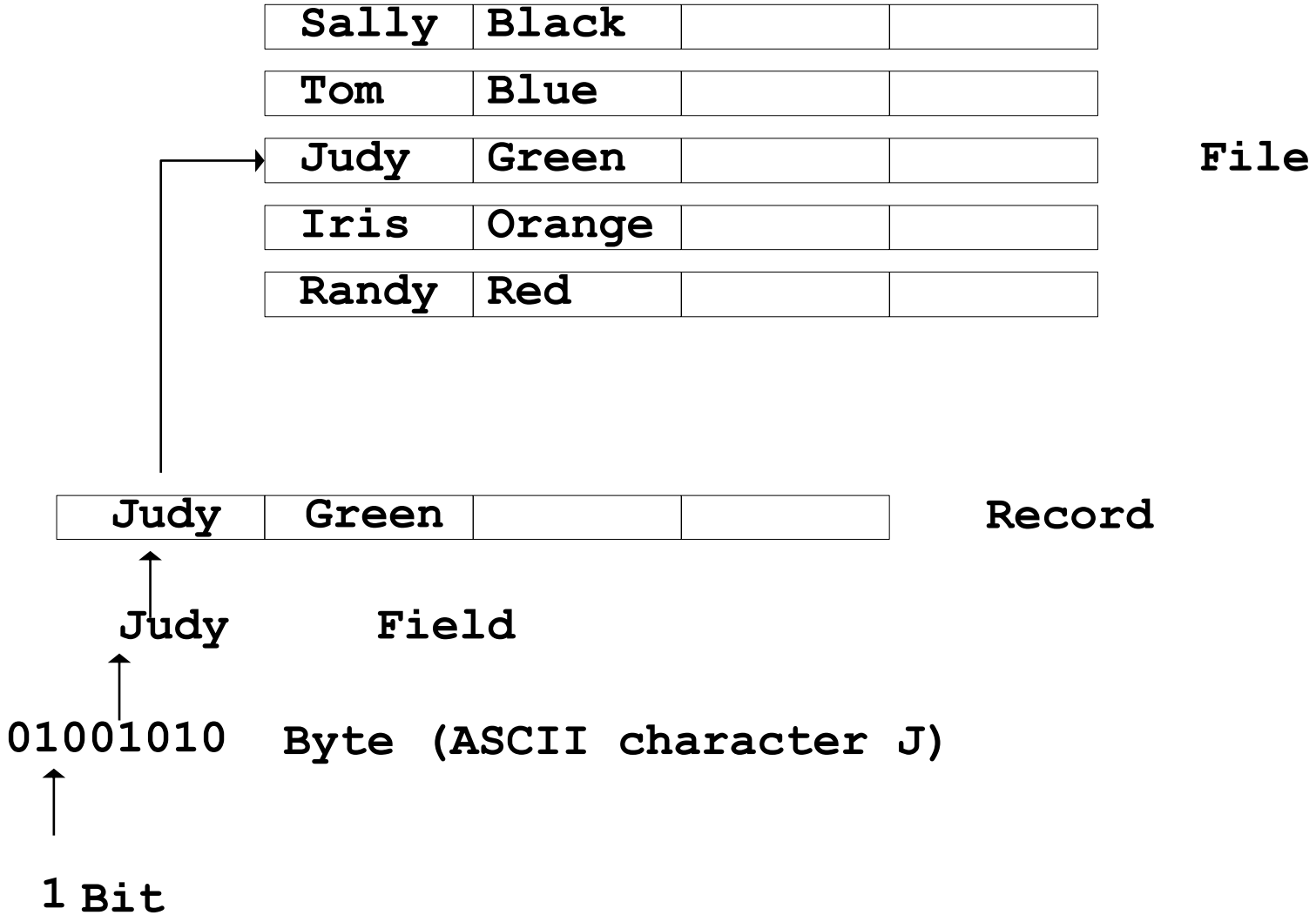
```
// Implementation of class Interface--client receives this file only
// as precompiled object code, keeping the implementation hidden.
#include "Interface.h" // Interface class definition
#include "Implementation.h" // Implementation class definition

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

```
void Interface::setValue( int v )
{
   ptr->setValue( v );
} // end function setValue

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

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

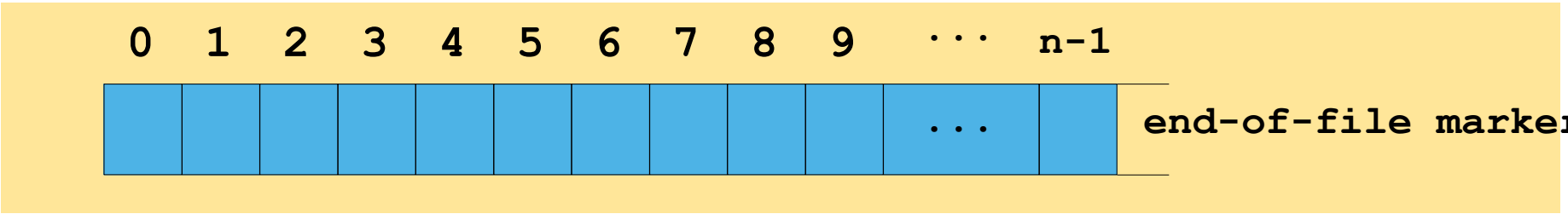
```
#include "Interface.h" // Interface class definition

int main()
{
   Interface i( 5 ); // create Interface object

   cout << "Interface contains: " << i.getValue()
      << " before setValue" << endl;

   i.setValue( 10 );

   cout << "Interface contains: " << i.getValue()
      << " after setValue" << endl;
   return 0;
} // end main
```

文件操作





- C++ views file as sequence of bytes
  - Ends with *end-of-file* marker



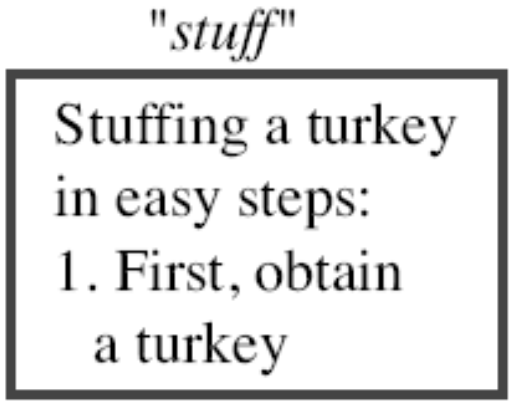
- To open file, create objects
  - Constructors take *file name* and *file-open mode*
    - ofstream outClientFile("filename", *fileOpenMode*);
    - A filename
      - If the file doe not exist, it is first created
    - A file-open mode
      - ios::out – the default mode
        - Overwrites preexisting data in the file
      - ios::app
        - Appends data to the end of the file

- To open file, create objects
  - To attach a file later
    - ofstream outClientFile;
    - outClientFile.open("filename", *fileOpenMode*);

File-open modes

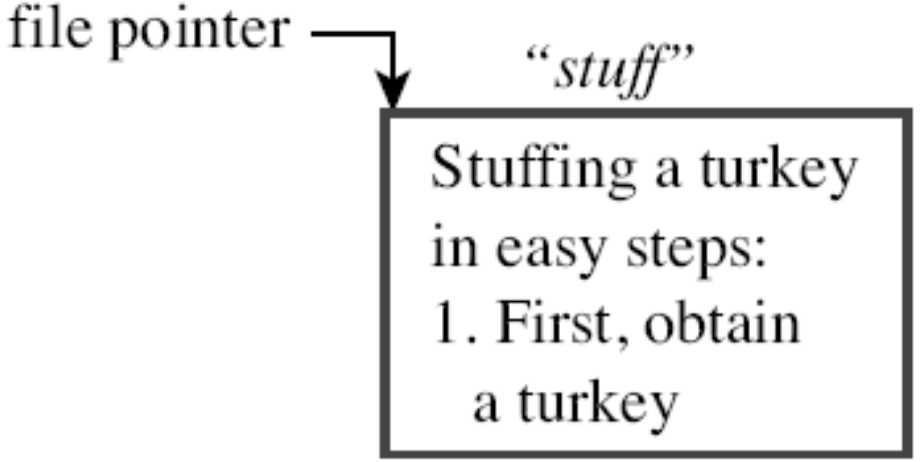
Mode	Description
ios::app	Write all output to the end of the file.
ios::ate	Open a file for output and move to the end of the file (normally used to append data to a file). Data can be written anywhere in the file.
ios::in	Open a file for input.
ios::out	Open a file for output.
ios::trunc	Discard the file's contents if it exists (this is also the default action for **ios::out**)
ios::binary	Open a file for binary (i.e., non-text) input or output.

File-open modes

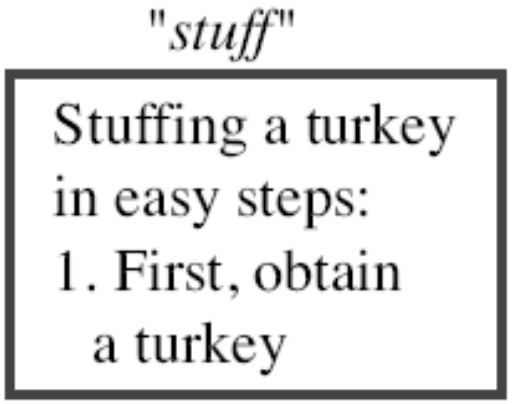


opening the file for input

```
ifstream fin("stuff");
```

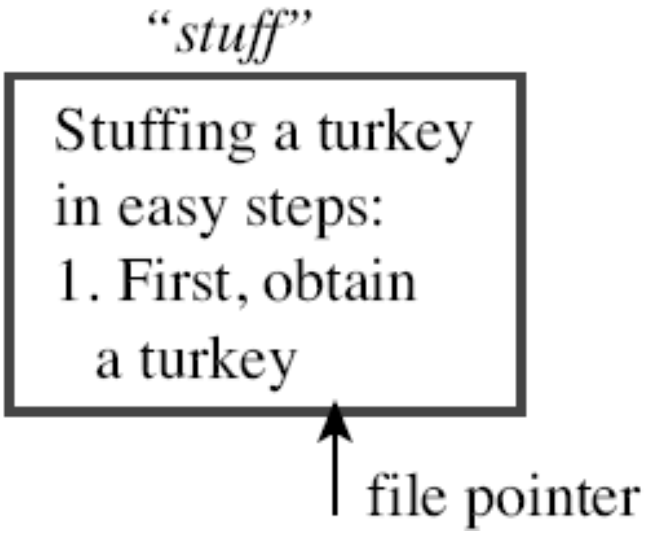


File-open modes

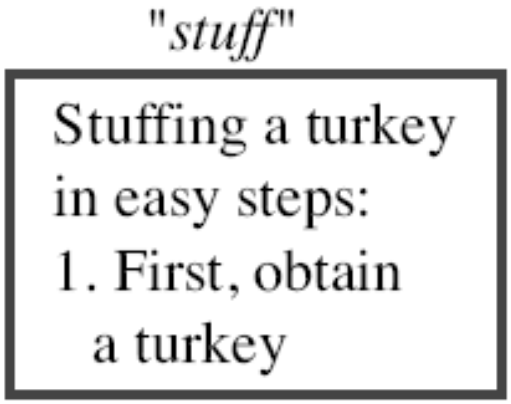


opening the file for appending

```
ofstream fout("stuff", ios::out¦ios:app);
```

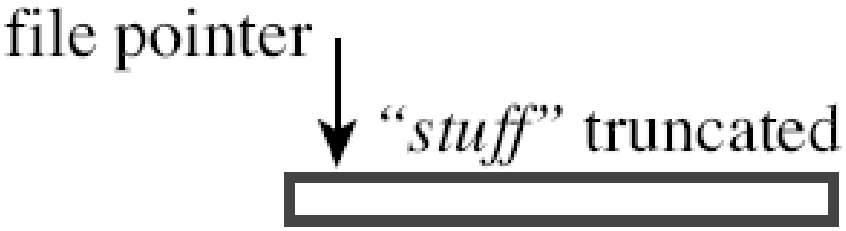


File-open modes



opening the file for output

```
ofstream fout("stuff");
```



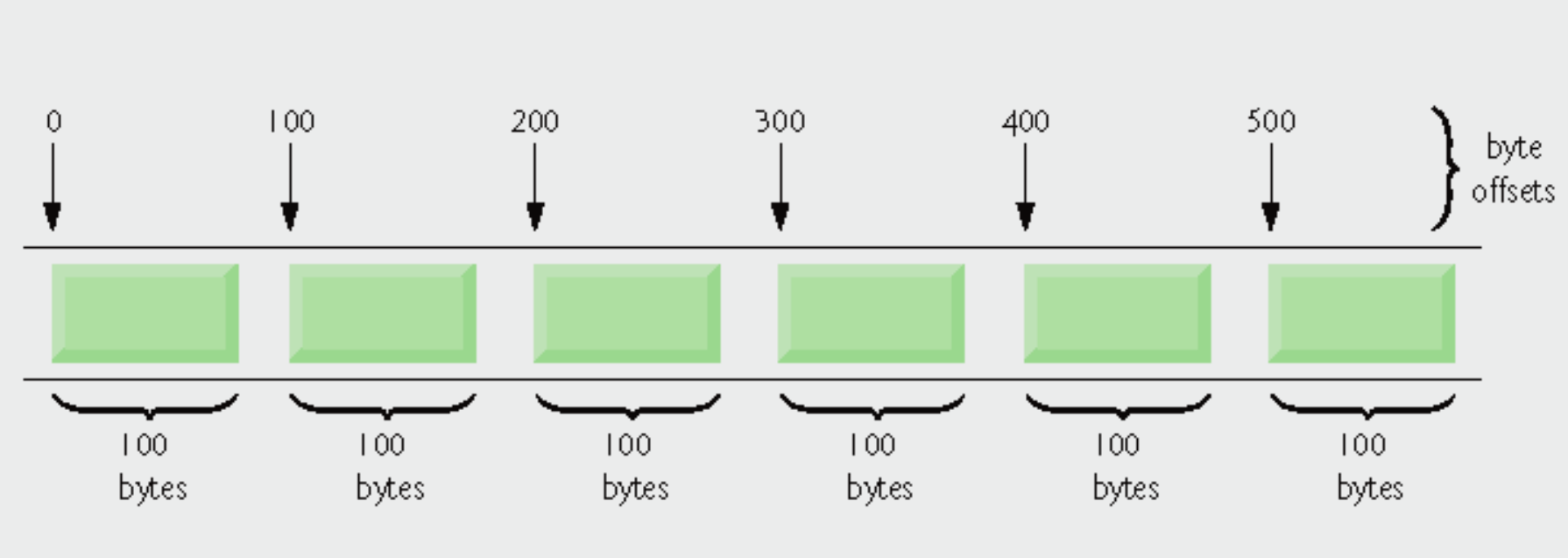
- To open file, create objects
  - Opens a file for input
    - ifstream inClientFile;
    - inClientFile.open("filename", *fileOpenMode*);
      - A filename
      - A file-open mode
        - ios::in – the default mode
        - Can only read from the file

- File-position pointer
  - Member functions seekg and seekp (of istream and ostream, respectively)
    - ios::beg – the default
      - Positioning relative to the beginning
    - ios::cur
      - Positioning relative to the current position
    - ios::end
      - Positioning relative to the end

- File-position pointer
  - Examples
    - fileObject.seekg(n);
      - Position to the nth byte of fileObject
    - fileObject.seekg(n, ios::cur);
      - Position n bytes forward in fileobject
    - fileObject.seekg(n, ios::end);
      - Position n bytes back from end of fileObject
    - fileObject.seekg(0, ios::end);
      - Position at end of fileObject

- File-position pointer
  - Member functions tellg and tellp (of istream and ostream, respectively)
    - Returns current position of the file-position pointer as type long
    - Example
      - Location = fileObject.tellg();

- **Random-access files**
  - A record can be inserted, deleted or modified without affecting other records
  - Require that all records be of the same length, arranged in the order of the record keys
    - Program can calculate the exact location of any record
      - Base on the record size and record key



- ostream member function write
  - Writes a number of bytes from a location in memory to the stream
  - First argument
    - A const char * pointing to bytes in memory
  - Second argument
    - A size_t specifying the number of bytes to write
  - Example
    - outFile.write(reinterpret_cast< const char * >(&number), sizeof(number));

- **Writing data randomly**
  - Opening for input and output in binary mode
    - Use an fstream object
    - Combine file-open modes ios::in, ios::out and ios::binary
  - Use function seekp to set the “put” file-position pointer to the specific position
    - Example calculation
      - (n – 1) * sizeof(ClientData)
  - Use function write to output the data

Sequentially reading a random-access file

- ifstream member function read
  - Inputs a number of bytes from the current file position in the stream into an object
  - First argument
    - A char * pointing to the object in memory
  - Second argument
    - A size_t specifying the number of bytes to input
  - ifstream inCredit("credit.dat", ios::in);
  - inCredit.read(reinterpret_cast< char * >(&client), sizeof(ClientData));

Example 01：

```
// writing on a text file
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile ("example.txt");
  if (myfile.is_open())
  {
    myfile << "This is a line.\n";
    myfile << "This is another line.\n";
    myfile.close();
  }
  else cout << "Unable to open file";
  return 0;
}
```

Example 01：

```
[file example.txt]
This is a line.
This is another line.
```

Example 02：

```
// reading a text file
…
int main () {
  string line;
  ifstream myfile ("example.txt");
  if (myfile.is_open())
  {
    while (! myfile.eof() )
    {
      getline (myfile,line);
      cout << line << endl;
    }
    myfile.close();
  }
  else cout << "Unable to open file";
  return 0;
}
```

Example 03：

```
// obtaining file size
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  long begin,end;
  ifstream myfile ("example.txt");
  begin = myfile.tellg();
  myfile.seekg (0, ios::end);
  end = myfile.tellg();
  myfile.close();
  cout << "size is: " << (end-begin) << " bytes.\n";
  return 0;
}
```

思考题：

- 修改FileExample中的程序，添加一个显示文件内容的选项