

Lecture Notes on C++ Multi-Paradigm Programming

Bachelor of Software Engineering, Spring 2014

Wan Hai

whwanhai@163.com

13512768378

Software School, Sun Yat-sen University, GZ

Classes and Data Abstraction

(类和数据抽象)

Outline

- 1 Abstraction**
- 2 Abstract Data Type**
- 3 Class and Object**
- 4 OOD theory**

Abstraction

- In this complex world, people often use **abstraction** to understand or solve many things.



Abstraction

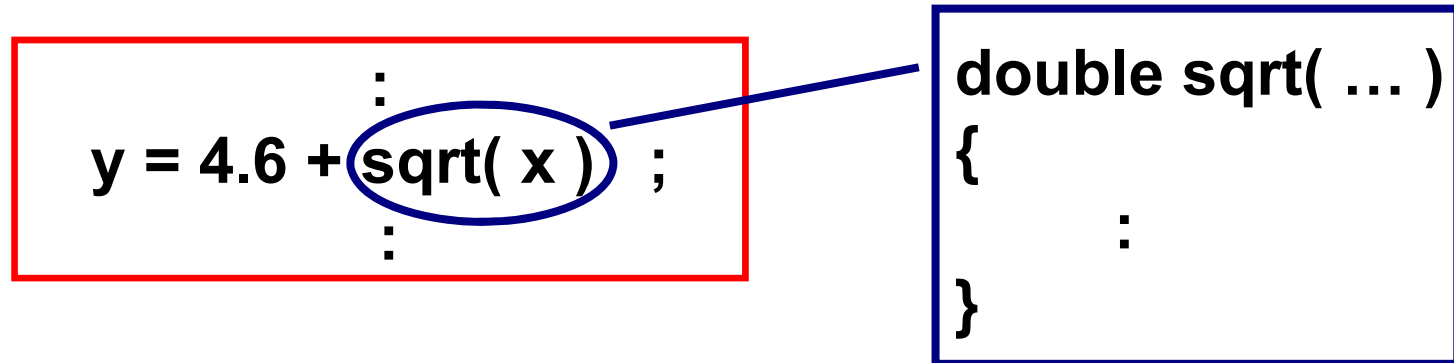
- **Data abstraction:** 只关心该数据“是什么”及“如何使用”，而不关心它是如何运作的。
- **Control abstraction:** 只关心这个行为能够为我们带来什么，而不关心这个行为的具体实现方法。

Abstraction

- **Is the separation of the essential qualities of an object from the details of how it works or is composed.**
- **Focuses on what, not how.**
- **Is necessary for managing large, complex software projects.**

Control Abstraction

- Separates the logical properties of an action from its implementation.



- The function call depends on the function's specification (description), not its implementation (algorithm)

Abstract Data Type（抽象数据类型, ADT）

- 在程序设计中，对于被抽象的数据，称为**抽象数据类型**（**Abstract Data Type, ADT**）。
- 一种**ADT**应具有
 - 1 **说明部分**（说明该该**ADT**是什么及如何使用）：说明部分描述数据值的特性和作用于这些数据之上的操作。
ADT的用户仅须明白这些说明，而无须知晓其内部实现。
 - 2 **实现部分**。

Abstraction

- 用户：抽象是用户的“权利”。
- 设计者：必须关注内部实现。
- 被抽象的对象本身：必须具备说明部分用以向用户说明自身，以及具备具体的实现部分。

An ADT: DATE

ADT Implementation means

- 1) Choosing a specific **data representation** for the abstract data using data types that already exist (built-in or programmer-defined).
- 2) **Writing functions** for each allowable operation.

Type

DATE

Data

Each DATE value is date

in day/month/year

int year, month, day;

Operation

Set the date

Get the date

Increment the date by ~~one day~~

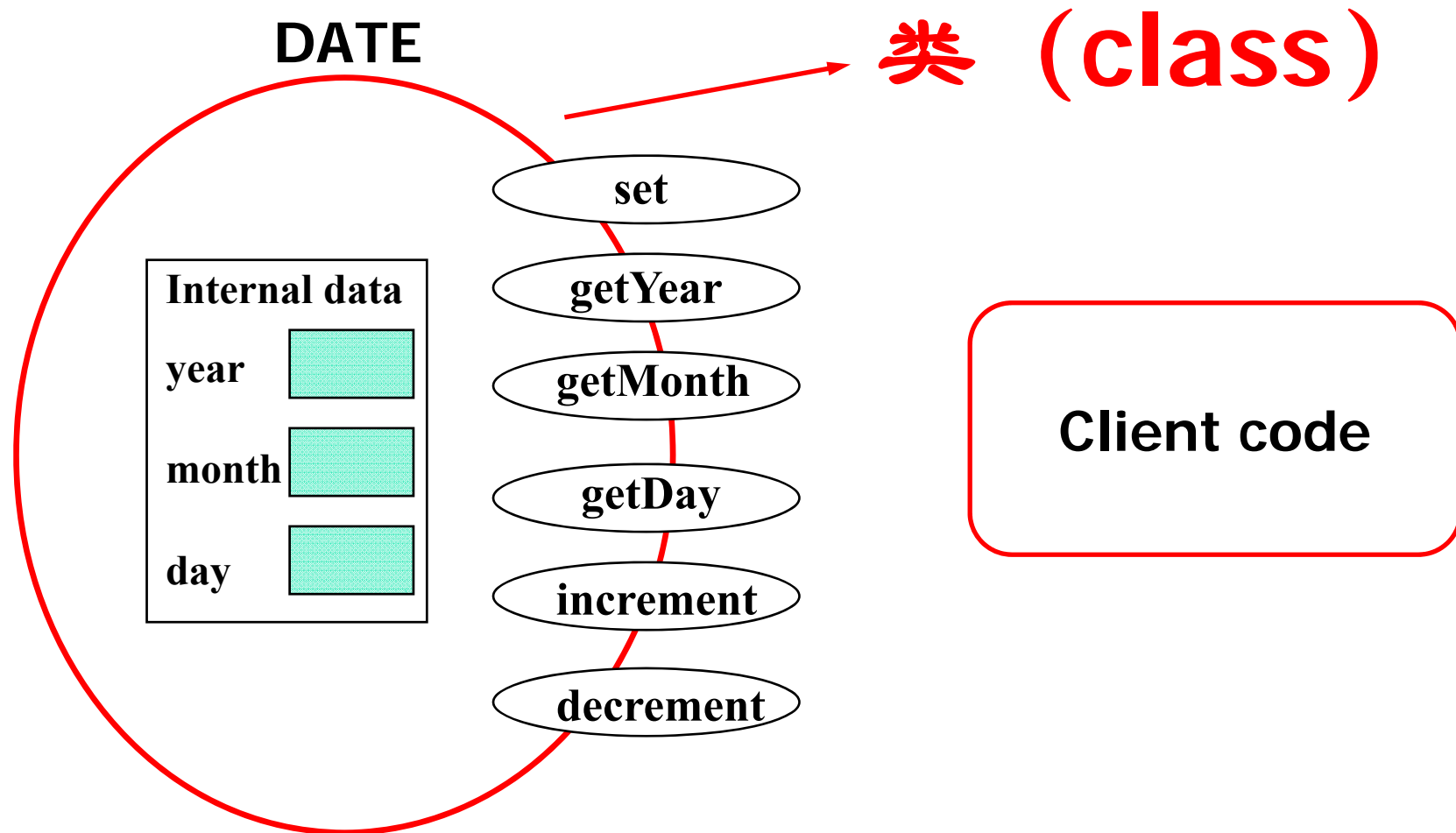
Decrement the date by one day

**set()
get()
increment()
decrement()**

更深入的思考。。。。

- 把**DATE**设计为一种数据类型。
- 内部包含年月日等数据以及在这些数据上可进行的操作。
- 用户利用**DATE**就可以定义多个变量。
- 用户可调用每个变量中公开的操作，但无法直接访问每个变量中被隐藏的内部数据。
- 用户也无需关心变量中各操作的具体实现。
- 于是**DATE**就是一种封装好的数据类型。这就达到了信息隐藏和封装的目的。

Implementing DATE with a class



Class and Object

- **Class:** is a user-defined data type that represents an ADT in C++ that have attributes (data members) and behaviors (member functions) that operate on the data members.
- Variables of the class type are called class **objects(对象)** or class **instances(实例)**.
- Software that uses the class is called a **client(客户代码)**. Client code uses public member functions to handle its class objects.

Implementing DATE with a class

```
class DATE // DATE.h----Specification file of class DATE
{
    public:
        void Set( int, int, int );
        int getMonth() const;
        int getDay() const;
        int getYear() const;
        void Print() const;
        void Increment();
        void Decrement();
    private:
        int month;
        int day;
        int year;
};
```

Implementing DATE with a class

- **class**是保留字，说明**DATE**是**类名**。在{ }中列出类的成员。
- 类的成员包括：
 - **数据成员**：一般说来，数据成员是**需要隐藏**的对象；即外部的程序是不能直接访问这些数据的，应该通过函数成员来访问这些数据。所以一般情况下，数据成员通过关键字**private**声明为私有成员（**private member**）
 - **函数成员**：通过关键字**public**声明为公有成员（**public member**）。外部程序可以访问共有成员，但无法访问私有成员。
- 对于**类的使用者**（即用户代码，简称用户）而言，**只需要获得DATE.h**，即可调用类对象的公有函数访问其内部的数据成员。使用者无法直接访问私有成员，也无需知晓公有函数的内部实现。

Implementing DATE with a class

//DATE.cpp
//the implementation of each
member function of DATE.

```
#include "DATE.h"
#include <iostream>
using namespace std;

int DaysInMonth( int, int );

void DATE::Set(int newYear,
               int newMonth,
               int newDay )
{ }
```

```
int DATE::getMonth() const
{ }
int DATE::getDay() const
{ }
int DATE::getYear() const
{ }
void DATE::Print() const
{ }
void DATE::Increment()
{ }
void DATE::Decrement()
{ }
int DaysInMonth( int mo,
                 int yr )
{ }
```


Implementing DATE with a class

**//DATE.cpp the implementation of each member
//function of DATE.**

```
void DATE::Set(int newYear,  
               int newMonth,  
               int newDay )  
{  
    month = newMonth;  
    day = newDay;  
    year = newYear;  
}
```

Implementing DATE with a class

**//DATE.cpp the implementation of each member function of
//DATE.**

```
int DATE::getMonth() const
```

```
{
```

```
    return month;
```

```
}
```

```
int DATE::getDay() const
```

```
{
```

```
    return day;
```

```
}
```

```
int DATE::getYear() const
```

```
{
```

```
    return year;
```

```
}
```

Implementing DATE with a class

**//DATE.cpp the implementation of each member function of
//DATE.**

```
void DATE::Print() const
{
    switch (month)
    {
        case 1 : cout << "January";
                break;
        case 2 : cout << "February";
                break;
                :
        case 12 : cout << "December";
    }
    cout << ' ' << day << ", " << year << endl << endl;
}
```

Implementing DATE with a class

**//DATE.cpp the implementation of each member function of
//DATE.**

```
void DATE::Increment()
{
    day++;
    if (day > DaysInMonth(month, year))
    {
        day = 1;
        month++;
        if (month > 12)
        {
            month = 1;
            year++;
        }
    }
}
```

Implementing DATE with a class

//DATE.cpp the implementation of each member function of DATE.

```
void DATE::Decrement()
{ day--;
  if ( day == 0 )
  {
    if( month == 1 )
    {
      day = 31;
      month = 12;
      year--;
    }
    else
    {
      month--;
      day = DaysInMonth( month, year );
    }
  }
}
```

Implementing DATE with a class

//DATE.cpp the implementation of the auxiliary function

//DaysInMonth.

```
int DaysInMonth( /* in */ int mo, /* in */ int yr )
{
    switch (mo)
    {
        case 1: case 3: case 5: case 7: case 8: case 10: case 12:
            return 31;
        case 4: case 6: case 9: case 11:
            return 30;
        case 2:
            if ((yr % 4 == 0 && yr % 100 != 0) || yr % 400 == 0)
                return 29;
            else
                return 28;
    }
}
```

Implementing DATE with a class

- 在**DATE.cpp**文件开头需要加入预处理命令

#include "DATE.h"

这是因为在**DATE.cpp**中要用到用户自定义的标识符**DATE**，而它的定义在**DATE.h**中。

- 在**DATE.h**中，各函数原型是在**{ }**中的。根据标识符的作用域规则，它们的作用范围仅在类定义中，而不包括**DATE.cpp**。因此在**DATE.cpp**中需要利用作用域解释运算符“**::**”来指明这里的函数是类**DATE**里的成员函数。
- **DATE.cpp**中有时还包括**DATE**内部要使用到的函数，例如**DaysInMonth**。这种函数并非对外公开供用户使用，因此可以将其声明为类的私有成员。若在该函数中没有涉及该类的数据成员，则无需将它们声明为类的成员。

Client Code Using DATE

```
//client.cpp
#include "DATE.h"
#include <iostream>
using namespace std;
int main()
{
    DATE date1, date2; //①
    int tmp;

    date1.Set( 1989, 6, 4 );
    date1.Print();
    date1.Increment();
    date1.Print();
    :
}
```

```
date2.Set( 1997, 7, 1 );
date2.Print();
date2.Decrement();
date2.Print();

tmp = date1.getYear();
tmp++;
date1.Set( tmp, 12, 20 );
date1.Print();

cout << date1.year;
                        //error?

return 0;
}
```


Results

The following items will be displayed on the monitor.

June 4, 1989

June 5, 1989

July 1, 1997

June 30, 1997

December 20, 1990

演示

1.程序DATE1

2.在VC上创建多文件的project

Encapsulation and Information Hiding

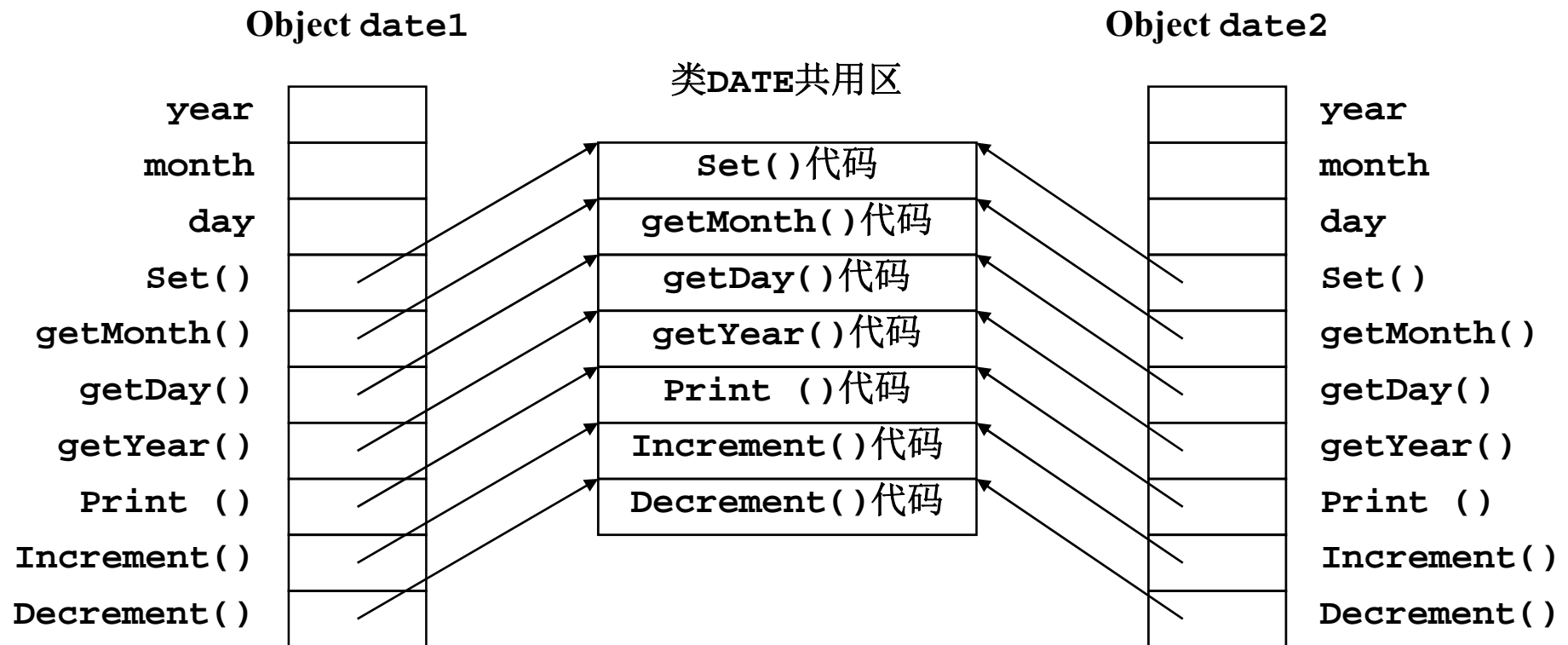
- 年、月、日这些内部数据被隐藏在模块中（这里的模块是文件**DATE.cpp**），客户程序只能通过该模块提供的公开操作来访问这些数据，而不能直接访问这些数据。
- 这种保护措施称为**信息隐藏**（**Information Hiding**）
- 把这这些数据与相关操作组织在一起的方式称为**封装**（**Encapsulation**）。

Encapsulation and Information Hiding

- 客户程序：只**关心****DATE**能够提供那些公开的操作（即提供了哪些函数可调用），并**不关心**这些操作的具体实现。也就是说，用户只**关心****DATE**能“**做什么**”，而**不关心**它内部“**如何做**”。
- 封装就是实现**ADT**的策略，而信息隐藏则是**ADT**的特点。
- 信息隐藏和封装是软件开发的必要技术，亦具商业价值。

Creation of Object

DATE date1, date2; //creation of two DATE objects



Using public members of an object

```
date1.Increment();    //1  
date2.Set(1976,12,20); //2
```

↓
Member selector

- 在成员函数中引用的成员是这个对象中的成员。
- 语句1中所涉及的`year\month\day`等数据成员是`date1`的成员，将修改`date1`这些成员的值；这并不影响另一对象`date2`中的数据成员`year\month\day`。
- 语句2将对象`date2`中的数据成员`year`、`month`和`day`分别设置为1976、12、20，这对另一对象`date1`中的数据成员`year`、`month`和`day`没有任何影响。

What's the Problem with the DATE?

```
//client.cpp
#include "DATE.h"
#include <iostream>
using namespace std;
int main()
{
    DATE date1, date2; //①
    int tmp;

    date1.Set( 1989, 6, 4 );
    date1.Print();
    date1.Increment();
    date1.Print();
    :
}
```

```
date2.Set( 1997, 7, 1 );
date2.Print();
date2.Decrement();
date2.Print();

tmp = date1.getYear();
tmp++;
date1.Set( tmp, 12, 20 );
date1.Print();

cout << date1.year;
                        //error?

return 0;
}
```

Class Constructors (构造函数)

- A class constructor is a member function whose purpose is to **initialize the private data members** of a class object.
- The **name** of a constructor is always the name of the class, and there is no return type for the constructor.
- A class **may have several constructors** with different parameter lists. A constructor with no parameters is the **default constructor**.
- A constructor is implicitly **invoked** when a class object is declared--if there are parameters, their values are listed in parentheses in the declaration.

Class Constructors

- No returned values are allowed in constructors. Even **void** is prohibited.
- Constructors are member functions of a class. Their prototypes should be given in .h and their implementations should be given in .cpp.

Class Constructors

- 如何向构造函数传递参数：在声明类对象时，在对象名后直接写上实参列表，编译器就会根据实参的个数和类型选择调用合适的构造函数。
- 若声明类对象时没有实参列表，则：
 - (1) 若该类有缺省构造函数（即无参构造函数），则调用该构造函数，同时创建该对象（为该对象分配内存空间）。
 - (2) 若该类没有缺省构造函数，则创建该对象。但该对象的私有成员没有得到初始化。

Adding Constructors into Class DATE

```
// DATE.h----类DATE的说明文件
class DATE
{
    public:
        DATE( int, int, int ); //构造函数
        DATE();               //缺省构造函数

        void Set( int, int, int);
        int getMonth() const;
        int getDay() const;
        int getYear() const;
        void Print() const;
        void Increment();
        void Decrement();
        :
```

```
        private:
            int month;
            int day;
            int year;
};
```

Adding Constructors into Class DATE

//DATE.cpp ---- 类DATE中各成员函数的实现

//除增加了构造函数的定义外，其余代码与之前的DATE.cpp相同

:

DATE::DATE(int initYear, int initMonth, int initDay)

{

year = initYear; //在构造函数中进行初始化

month = initMonth;

day = initDay;

}

DATE::DATE()

{

year = 2000;

month = 1;

day = 1;

}

:

Client code using DATE

```
//client.cpp ---- 类DATE中各成员函数的实现
#include "DATE.h"
#include <iostream>
using namespace std;

int main()
{
    DATE date1;                //①自动调用缺省构造函数
    DATE date2( 1976, 12, 20 ); //②自动调用另一个有参数的构造函数

    date1.print();
    date2.print();

    return 0;
}
```

运行结果，屏幕上显示：
January 1, 2000
December 20, 1976

Constructor

- 如果设计的类没有构造函数，C++编译器会自动为该类型建立一个缺省构造函数。该构造函数没有任何形参，且函数体为空。
- 应该养成编写构造函数的习惯。

Constructors with default parameters

- **Default arguments**
 - **Set in default constructor function prototype.**
 - **Do not set defaults in the function definition, outside of a class**

Constructors with default parameters

```
class DATE // DATE.h
{
    public:
        DATE( int = 2000, int = 1, int = 1 );
        :
};
```

```
DATE::DATE( int initYear, int initMonth, int initDay ) //DATE.cpp
{
    year = initYear; month = initMonth; day = initDay;
}
```

```
int main() //client.cpp
{
    DATE date1; //initYear, initMonth, initDay分别为2000, 1, 1
    DATE date2( 1976 ); //initYear, initMonth, initDay分别为1976, 1, 1
    DATE date2( 1976, 12 ); //initYear, initMonth, initDay分别为1976, 12, 1

    DATE date2( 1976, 12, 20 ); //initYear, initMonth, initDay分别为1976, 12, 10
}
```


Adding Constructors into Class DATE

```
// DATE.h----类DATE的说明文件
class DATE
{
    public:
        DATE( int, int, int ); //构造函数
        DATE();                //缺省构造函数

        void Set( int, int, int);
        int getMonth() const;
        int getDay() const;
        int getYear() const;
        void Print() const;
        void Increment();
        void Decrement();
        :
```

```
        private:
            int month;
            int day;
            int year;
};
```

Adding Constructors into Class DATE

//DATE.cpp ---- 类DATE中各成员函数的实现

//除增加了构造函数的定义外，其余代码与之前的DATE.cpp相同

:

DATE::DATE(int initYear, int initMonth, int initDay)

{

year = initYear; //在构造函数中进行初始化

month = initMonth;

day = initDay;

}

DATE::DATE()

{

year = 2000;

month = 1;

day = 1;

}

:

使用new和delete动态分配和释放内存空间

```
int *p;  
int Length, i;  
  
cout << "Enter the lenght you want: ";  
cin >> Length;  
p = new int[ Length ];  
  
cout << "Enter a[0]~a[" << Length-1 << "]:" << endl;  
for( i = 0 ; i < Length; i++ )  
    cin >> *(p+i);  
  
inv( p , Length );  
  
for( i = 0 ; i < Length; i++ )  
    cout << *(p+i) << " ";  
  
delete []p;
```

new 数据类型[元素个数]

//程序14

也可仅动态分配一个变量

```
int *p;  
p = new int;  
  
cout << "Please enter an integer value: ";  
cin >> *p;  
cout << "The value you enter is: " << *p;  
  
delete p;
```

new运算符的使用

指针=new 类型名; //动态创建一个变量

指针=new 类型名[数组长度]; //用于动态分配数组

指针=new 类型名（初始化表）; //动态创建对象

- 初始化表及其括号为可选
- 类型可为基本类型，也可类类型，若为类类型，则初始化表相当于将实际参数传递给该类的构造函数
- new运算返回一个指针，指向分配到的内存空间
- 若内存分配失败，则返回**NULL（0）**。

注意

- 动态分配的内存空间**使用完毕后**，应该“**释放**”掉这块空间，即使得这块内存空间可以被操作系统**回收**以作它用。
- 假如程序中动态分配了很多内存空间，但使用完毕后都不释放，则这些空间无法用于存储别的数据，造成严重的**内存浪费**。

Destructor (析构函数)

- 也是类的**成员函数**。**作用**是在对象撤销时执行一些清理任务。
- C++语言规定析构函数名是**类名前加波浪号“~”**，以别于构造函数。
- 析构函数**不能**有任何返回类型，这点与构造函数相同。但同时析构函数还不能带任何参数，也就是说析构函数一定是**无参函数**。
- 在客户代码中，可以**通过“.”显式调用**析构函数；但更多的情况下，是在对象生存期结束时**自动被调用**的。

Destructor (析构函数)

```
// *****  
// demoClass.h 类DemoClass的说明文件  
// *****  
  
#include <iostream>  
#include <string>  
using namespace std;  
  
class DemoClass {  
public:  
    DemoClass();  
    ~DemoClass();  
};
```


Destructor (析构函数)

```
// DemoClass.cpp 类DemoClass的实现文件

#include "demoClass.h"
#include <iostream>
using namespace std;

DemoClass::DemoClass()
{
    cout << "Now in Constructor. The object is being created." <<
endl;
}

DemoClass::~~DemoClass()
{
    cout << "Now in Deconstructor. The object is deleted." << endl;
}
```

Destructor (析构函数)

```
#include "demoClass.h"
#include <iostream>
using namespace std;

int main()
{
    DemoClass obj;

    cout << "Now in function main." << endl;

    return 0;
}
```

运行结果，屏幕上显示：

Now in Constructor. The object is being created.

Now in function main.

Now in Destructor. The object is deleted.

Destructor (析构函数)

//ARRAY.h-----类ARRAY的说明文件

```
class ARRAY
{
    public:
        ARRAY(int initLenght); //构造函数
        ~ARRAY(); //析构函数
    private:
        int Length;
        float* p;
};
```

Destructor (析构函数)

```
//ARRAY.cpp----类ARRAY的实现文件
#include <iostream>
#include "ARRAY.h"
using namespace std;

ARRAY::ARRAY( int initLength )
{
    Length = initLength;
    p = new float[ Length ];
    cout << Length << "bytes have already allocated." << endl;
}

ARRAY::~~ARRAY()
{
    delete []p;
    cout << Length << "bytes have been released. Bye." << endl;
}
```

Destructor (析构函数)

//DEMO.cpp

```
#include <iostream>
#include "ARRAY.h"
using namespace std;
```

```
int main()
{
```

```
    ARRAY arr( 100 );
```

```
    cout << "This is the end of the program"
         << " and the object will be destroyed." << endl;
```

```
    return 0;
```

```
}
```

运行结果，屏幕上显示：
100 bytes have already allocated.
This is the end of the program and the
object will be destroyed.
100 bytes have been released. Bye.

对象指针

```
class DATE
{
public:
    DATE( int initYear, int initMonth, int initDay ); //构造函数
    DATE(); //缺省构造函数

    void Set( int newMonth, int newDay, int newYear );
    int getMonth() const;
    int getDay() const;
    int getYear() const;
    void Print() const;
    void Increment();
    void Decrement();

private:
    int month;
    int day;
    int year;
};
```

对象指针

//程序31

```
DATE* date_ptr;  
DATE date(1989, 6, 4);
```

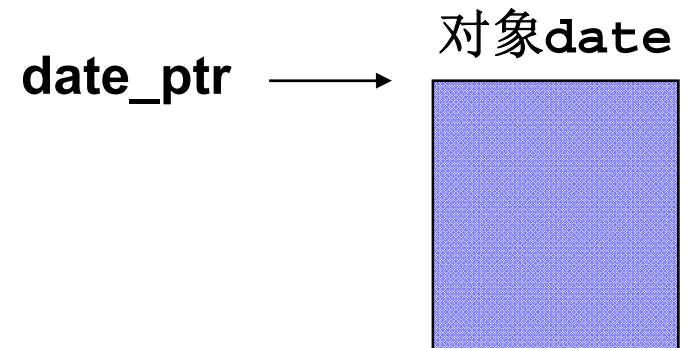
```
date_ptr = &date;
```

```
date.Print();  
date_ptr->Print();
```

运行结果:

June 4, 1989

June 4, 1989



对象指针

- 对象是一种复合型的数据，往往占据比较多内存空间；如果程序中需要使用很多对象，可能容易造成内存紧张。
- 解决方法：在程序需要对象时**创建**对象，在对象使用完毕后**撤销**这个对象。
- 实现这一方法就要使用**指向对象的指针**。

对象的动态创建和撤销

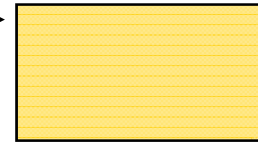
//程序31

```
DATE* date_ptr;  
date_ptr = new DATE(1976, 12 ,20);
```

```
if( date_ptr == NULL )  
{  
    cout << “内存分配失败” ;  
    return 1;  
}  
else  
{  
    date_ptr -> Increment();  
    date_ptr -> Print();  
}
```

```
delete date_ptr;
```

date_ptr →



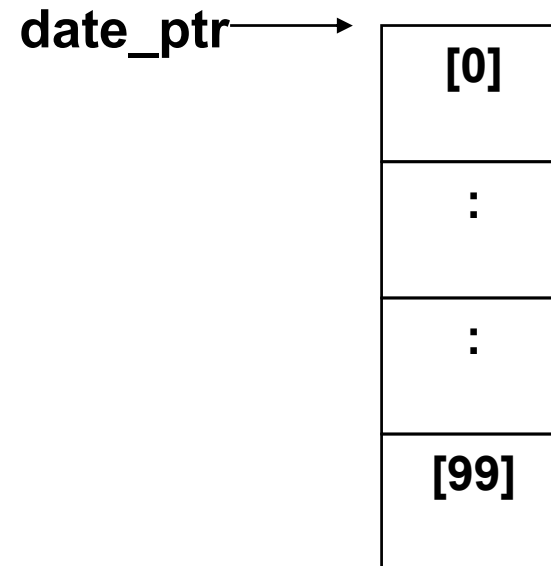
对象指针 = new 类名(初始化列表);

运行后输出结果:

December 21, 1976

对象的动态创建和撤销

```
DATE* date_ptr;  
int k = 100;  
date_ptr = new DATE[k];  
  
if( date_ptr == NULL )  
{  
    cout << “内存分配失败” ;  
    return 1;  
}  
else  
{  
    date_ptr -> Increment();  
    date_ptr -> Print();  
}
```



delete []date_ptr; // []将令所有元素都调用各自的析构函数

关于delete

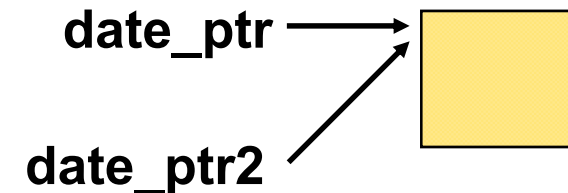
- 用**new**分配的内存存在不再使用时，要用**delete**释放。
- **delete**释放的是指针所指对象占据的内存。
- 用**delete**释放空间后，指针的值仍是原来指向的地址，但指针已无效（重复释放将出错）。
- **delete**对象指针，会调用该对象的析构函数。
- 若使用**new**运算分配的是数组（尤其是类类型对象的数组），则用**delete**释放时必须注意加上方括号。

关于delete

- **注意：** 由于**delete**一个指针即等同于释放掉该指针所指向的内存空间，而在**C++**中同一内存空间的释放只能进行一次；已**delete**过的指针再次**delete**会产生错误。因此，最好**不要让多个指针指向同一个对象**（多个指针同时指向同一内存空间的现象称为“**指针别名**”），看下页例子。

指针别名

```
DATE* date_ptr;  
DATE* date_ptr2;  
date_ptr = new DATE(1976, 12, 20);
```



```
date_ptr2 = date_ptr;
```

```
date_ptr -> Print(); //正确
```

```
date_ptr2 -> Print(); //正确
```

```
delete date_ptr; //将使date_ptr和date_ptr2同时失效
```

```
date_ptr -> Print(); //错误, date_ptr已失效
```

```
date_ptr2 -> Print(); //错误, date_ptr2也已失效
```

```
delete date_ptr2; //错误, 不能再次释放
```

//程序32

指针别名

//对于指向简单数据类型的指针，也有相同的问题。 程序33

```
int* p1;  
int* p2;
```

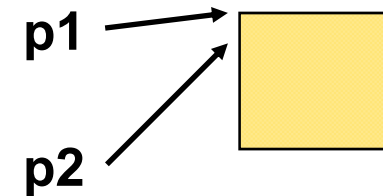
```
p1 = new int;  
p2 = p1;
```

```
*p1 = 10;  
cout << *p2 << endl;
```

```
delete p1;    //将使p1r和p2同时失效
```

```
cout << *p1;  //错误, p1已失效  
cout << *p2;  //错误, p2也已失效
```

```
delete p2;    //错误, 不能再次释放
```



内存垃圾

- 程序不再需要动态创建的对象时，一定要记住释放掉这些对象。否则这些空间会一直被占用，无法分配给别程序使用；如果指向这些空间的指针指向了别处，将无法回收这些内存空间。
- 这些无法回收的内存空间称为 **内存垃圾**（**garbage**）。内存垃圾不断增加会消耗掉大量内存空间，有时会导致系统崩溃。

对象的复制

- 在C++中，一个对象可以直接赋值给相同类型的另一个对象，这就是对象的复制。例如：

```
DATE date1, date2;
```

```
...
```

```
date2 = date1; //复制
```

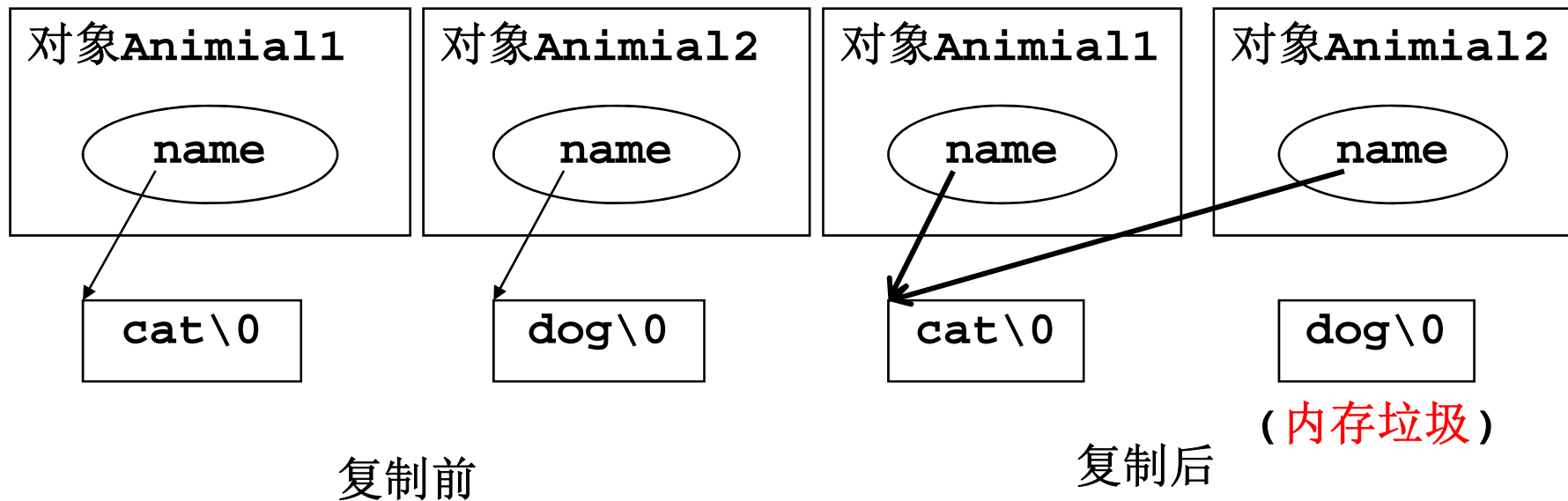
- C++提供的这种复制策略称为浅复制（**shallow copy**），在数据成员中不出现指针时，是没有任何问题的。但在数据成员中出现指针时，就很可能出现问题。

“指针别名” 和 “内存垃圾” 的现象

```
class ANIMAL
{
public:
    ANIMAL( char* str )
    {
        name = new char[ strlen( str ) + 1 ];
        strcpy( name, str );
    }
    ~ANIMAL()
    {
        delete name;
    }
private:
    char* name;
};
```

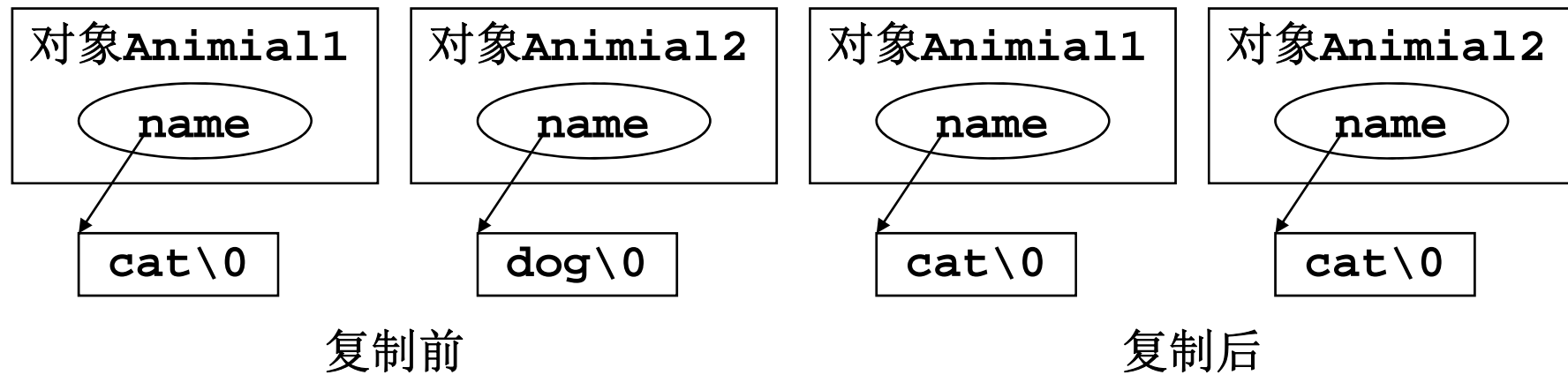
“指针别名”和“内存垃圾”的现象

```
ANIMAL Animal1("cat");  
ANIMAL Animal2("dog");  
Animal2 = Animal1;
```



浅复制 (**shallow copy**)

我们希望



深复制 (deep copy)

Program's Structure (1)

```
#include <iostream> //DATE.h
using namespace std;

class DATE {
    public:
        void Set( int newMonth, int newDay, int newYear );
        int getMonth() const;
        :
    private:
        int month;
        int day;
        int year;
};
```

```
#include <DATE.h> //DATE.cpp

void DATE::Set(int newYear, int newMonth, int newDay )
{ ... }

int DATE::getMonth() const
{ ... }

:
```

```
//Client.cpp
#include <DATE.h>

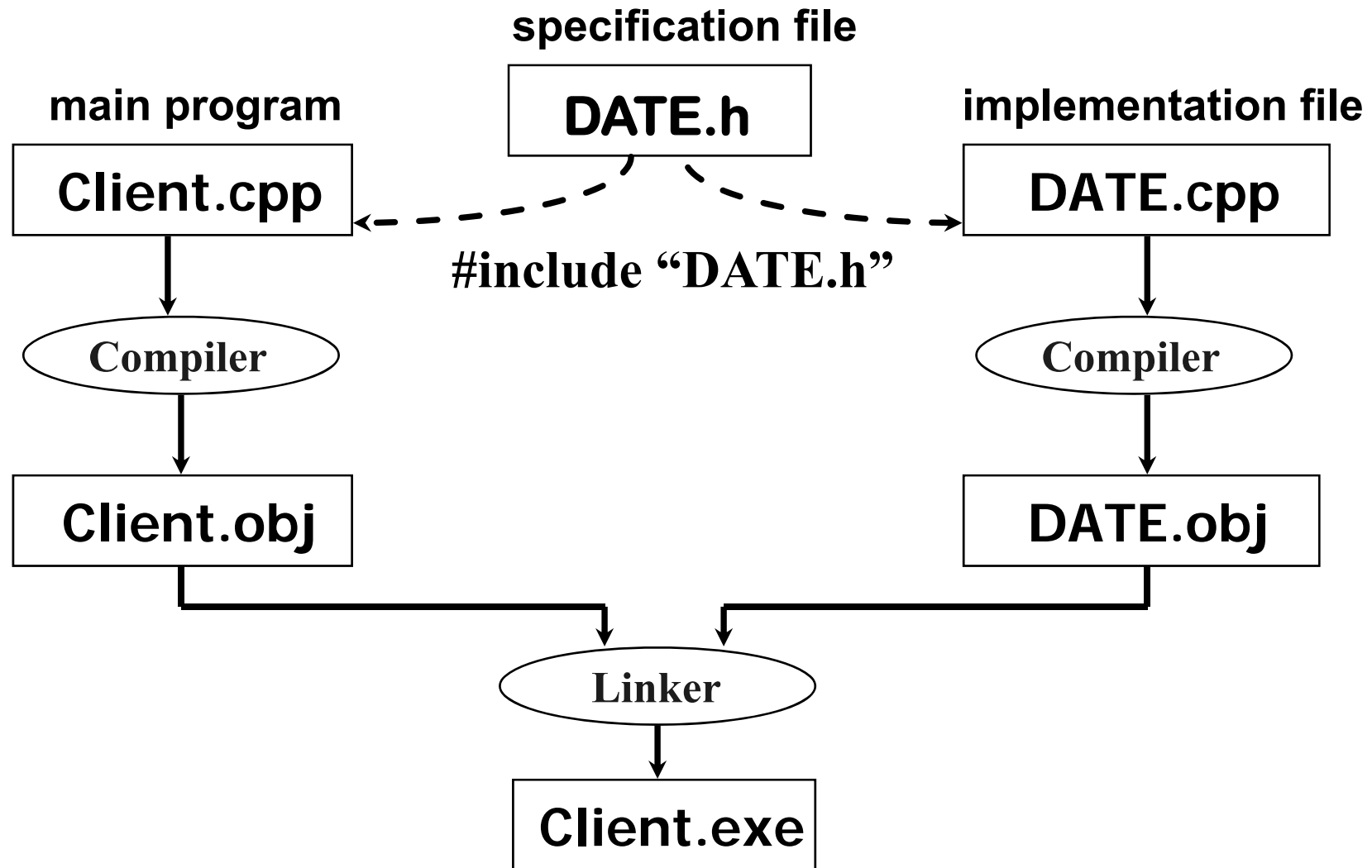
int main()
{
    DATE date1, date2;
    :
    date1.Print();
    :
}
```



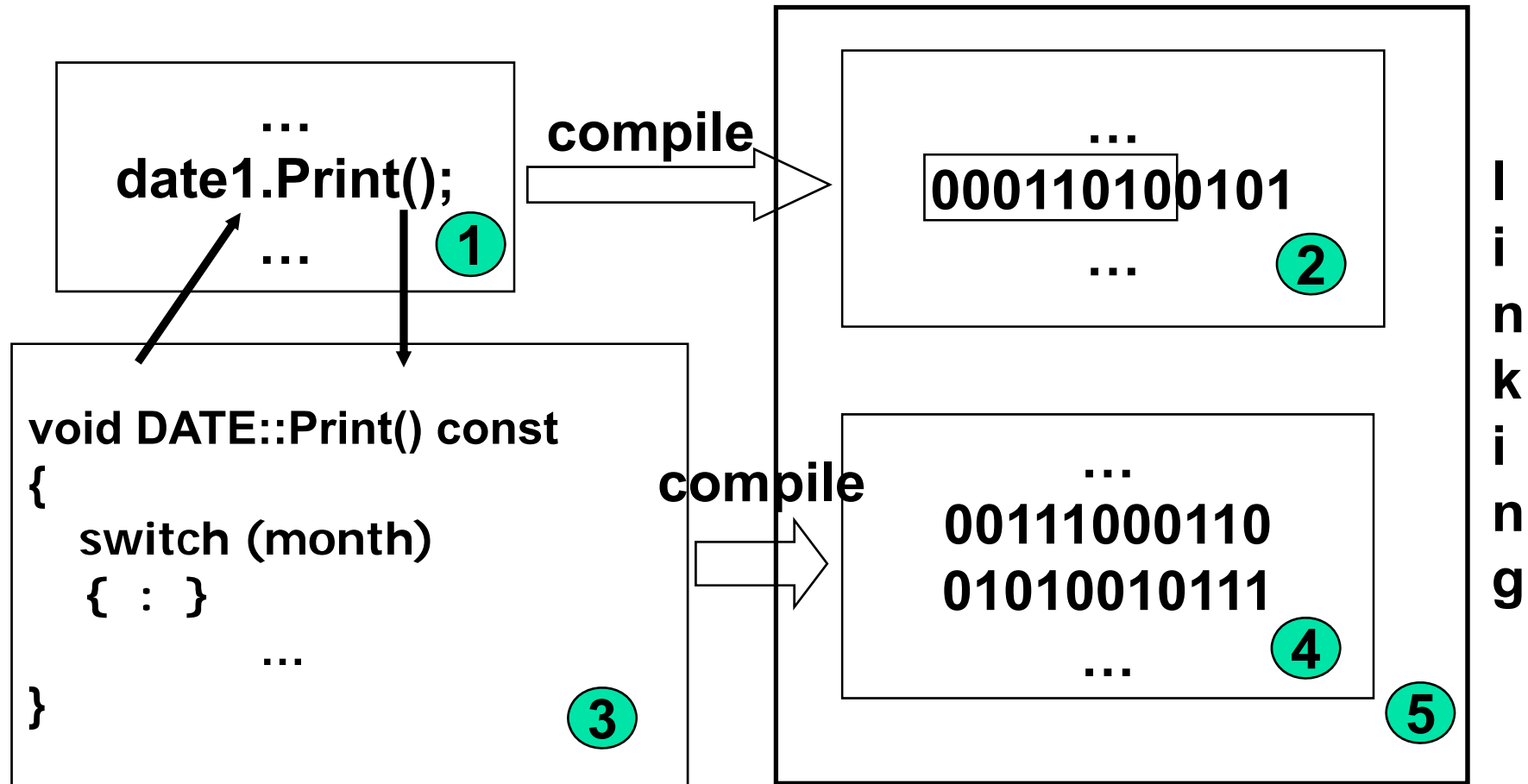
Separate Compilation and Linking of Files

- .cpp被编译成.obj文件，同一程序中的各个obj文件被链接成.exe可执行文件。.h文件是不会被编译的。
- 在C++中，多文件程序中的各.cpp文件不但被单独编译（separate compilation），而且可以在不同的时刻编译。
- 对于一个类，例如DATE，其.h及.obj文件都应该可以被用户使用。前者使用户知晓对象的功能和如何使用该对象；用户也需要后者链接到他自己的程序上，以便创建可执行文件。

Separate Compilation and Linking of Files



Separate Compilation and Linking of Files



1 client.cpp 2 client.obj 3 DATE.cpp
4 DATE.obj 5 client.exe

Program's Structure (1)

```
#include <iostream> //DATE.h
using namespace std;

class DATE {
    public:
        void Set( int newMonth, int newDay, int newYear );
        int getMonth() const;
        :
    private:
        int month;
        int day;
        int year;
};
```

```
#include <DATE.h> //DATE.cpp

void DATE::Set(int newYear, int newMonth, int newDay )
{ ... }

int DATE::getMonth() const
{ ... }

:
```

```
//Client.cpp
#include <DATE.h>

int main()
{
    DATE date1, date2;
    :
}
```



Program's Structure (2)


```
#include <iostream> //DATE.h
using namespace std;

class DATE {
    public:
        void Set( int newMonth, int newDay, int newYear );
        int getMonth() const;
        :
    private:
        int month;
        int day;
        int year;
};

void DATE::Set(int newYear, int newMonth, int newDay )
{ ... }

int DATE::getMonth() const
{ ... }

int DATE::getDay() const
{ ... }
:
```



```
//client.cpp

#include <DATE.h>

int main()
{
    DATE date1, date2;
    :
}
```

Program's Structure (3)

```
#include <iostream>    //DATE.h
using namespace std;

class DATE {
    public:
        void Set( int newMonth,  int newDay, int newYear )
        { ... }

        int getMonth() const
        { ...}

        int getDay() const
        { ...}
        :
    private:
        int month;
        int day;
        int year;
};
```

```
//Client.cpp

#include <DATE.h>

int main()
{
    DATE date1, date2;
    :
}
```



Program's Structure (4)

//client.cpp

```
#include <iostream>
using namespace std;
```

```
class DATE {
    public:
        void Set( int newMonth,
                  int newDay, int newYear );
        int getMonth() const;
        int getDay() const;
        int getYear() const;
        void Print() const;
        void Increment();
        void Decrement();
    private:
        int month;
        int day;
        int year;
};
```

//client.cpp

```
void DATE::Set(int newYear,
               int newMonth,
               int newDay )
{ ... }

int DATE::getMonth() const
{ ... }

int DATE::getDay() const
{ ...}

:

int main()
{
    DATE date1, date2;
    :
}
```

Program's Structure (5)

```
#include <iostream> //client.cpp
using namespace std;
class DATE {
    public:
        void Set( int newMonth,  int newDay, int newYear )
        { ... }

        int getMonth() const
        { ...}

        int getDay() const
        { ...}

        :
    private:
        int month;
        int day;
        int year;
};
int main()
{
    DATE date1, date2;
    :
}
```



const Member Functions

- **Data members can not be modified in a const member functions.**
- **'const' appears in both function prototype and function definition.**
- **'const' is used for readability and credibility.**

const Member Functions

```
class DATE // DATE.h----Specification file of class DATE
{
    public:
        void Set( int newMonth, int newDay, int newYear );
        int getMonth() const;
        int getDay() const;
        int getYear() const;
        void Print() const;
        void Increment();
        void Decrement();
    private:
        int month;
        int day;
        int year;
};
```

const Member Functions

```
//DATE.cpp  
//the implementation of each  
member function of DATE.
```

```
#include "DATE.h"  
#include <iostream>  
using namespace std;
```

```
int DaysInMonth( int, int );  
  
void DATE::Set(int newYear,  
               int newMonth,  
               int newDay )  
{ }
```

```
int DATE::getMonth() const  
{ }  
int DATE::getDay()   const  
{ }  
int DATE::getYear()  const  
{ }  
void DATE::Print()    const  
{ }  
void DATE::Increment()  
{ }  
void DATE::Decrement()  
{ }  
int DaysInMonth( int mo,  
                 int yr )  
{ }
```

const Member Functions

**//DATE.cpp the implementation of each member
//function of DATE.**

```
void DATE::Set(int newYear,  
               int newMonth,  
               int newDay )  
{  
    month = newMonth;  
    day = newDay;  
    year = newYear;  
}
```


const Member Functions

**//DATE.cpp the implementation of each member function of
//DATE.**

```
int DATE::getMonth() const
```

```
{
```

```
    return month;
```

```
}
```

```
int DATE::getDay() const
```

```
{
```

```
    return day;
```

```
}
```

```
int DATE::getYear() const
```

```
{
```

```
    return year;
```

```
}
```

const Member Data （常量数据成员）

- 常量数据成员的声明与符号常量（命名常量）的声明类似。
- 常量数据成员是对象的数据成员，不能在声明时初始化，且常量一旦声明之后就不能再作为左值，所以只能在构造函数的初始化列表中对常量数据成员进行初始化。

类的常量成员

```
class Demo {
public:
    Demo(): data1(0) // 常量数据成员只能在构造函数初始化列表中初始化
    {
        // data1 = 0; × // 此处不能对常量数据成员data1赋值

        data = 0;
    }

private:
    int data; // 一般的数据成员
    const int data1; // 常量数据成员
};
```

const object

//client.cpp ---- 类DATE中各成员函数的实现

```
#include "DATE.h"
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    DATE date1;
```

```
    const DATE date2( 1976, 12, 20 );
```

```
    date1.Set( 2011, 3, 14 ); ✓
```

```
    date2.Set( 2011, 3, 14 ); ✗
```

```
    date2.Print(); ✓
```

```
    return 0;
```

```
}
```

类的静态成员

- 静态（**static**）成员是类的组成部分但不是任何对象的组成部分
- 通过在成员声明前加上保留字**static**将成员设为**static**（在数据成员的类型前加保留字**static**声明静态数据成员；在成员函数的返回类型前加保留字**static**声明静态成员函数）
- **static**成员遵循正常的公有/私有访问规则。C++程序中，如果访问控制允许的话，可在类作用域外直接（不通过对象）访问静态成员（需加上类名和::）

类的静态成员（续）

- 静态数据成员具有静态生存期，是类的所有对象共享的存储空间，是整个类的所有对象的属性，而不是某个对象的属性。
- 与非静态数据成员不同，静态数据成员不是通过构造函数进行初始化，而是必须在类定义体的外部再定义一次，且恰好一次，通常是在类的实现文件中再声明一次，而且此时不能再用**static**修饰。

类的静态成员（续）

- 静态成员函数不属于任何对象
- 静态成员函数没有this指针
- 静态成员函数不能直接访问类的非静态数据成员，只能直接访问类的静态数据成员

类的静态成员

```
class DATE           // DATE.h
{
    public:
        DATE( int =2000, int =1, int = 1);

        static void getCount( );

        void Set( int, int, int);
        int getMonth() const;
        int getDay() const;
        int getYear() const;
        void Print() const;
        void Increment();
        void Decrement();
        :
```

```
        :
        private:
            int month;
            int day;
            int year;
            static int count;

};
```


类的静态成员

//DATE.cpp

StaticMember

int DATE::count = 0; //必须在类定义体的外部再定义一次

DATE::DATE(int initYear, int initMonth, int initDay)

```
{  
    year = initYear;  
    month = initMonth;  
    day = initDay;  
    count++;  
}
```

void DATE::getCount()

```
{  
    cout << "There are " << count << " objects now" << endl;  
}
```

void main()

```
{  
    DATE DATE1;  
    DATE DATE2( 1976, 12, 20 );  
  
    DATE::getCount();  
}
```

this指针的使用

```
class DATE // DATE.h----Specification file of class DATE
{
    public:
        void Set( int, int, int );
        int getMonth() const;
        int getDay() const;
        int getYear() const;
        void Print() const;
        void Increment();
        void Decrement();
    private:
        int month;
        int day;
        int year;
};
```

//DATE3

this指针的使用

```
//DATE.cpp  
//the implementation of each  
member function of DATE.
```

```
#include "DATE.h"  
#include <iostream>  
using namespace std;  
  
int DaysInMonth( int, int );  
  
void DATE::Set(int Year,  
               int Month,  
               int Day )  
{ }
```

```
int DATE::getMonth() const  
{ }  
int DATE::getDay() const  
{ }  
int DATE::getYear() const  
{ }  
void DATE::Print() const  
{ }  
void DATE::Increment()  
{ }  
void DATE::Decrement()  
{ }  
int DaysInMonth( int mo,  
                 int yr )  
{ }
```

this指针的使用

//在成员函数中直接使用数据成员时，已隐含地使用了**this**。
//当然，也可以明确写出。

```
void DATE::Set(int newYear,  
               int newMonth,  
               int newDay )
```

```
{
```

```
✓ month = newMonth; //this->month = newMonth ✓
```

```
✓ day = newDay;      //this->day = newDay ✓
```

```
✓ year = newYear;    //this->year = newYear ✓
```

```
}
```

//client code

```
DATE date1, date2;  
date1.set(2011,1,1);  
date2.set(2012,2,2);
```

this指针的使用

//在成员函数中需要区别形参与数据成员时，就需要明确
//写出**this**

```
void DATE::Set(int year,  
               int month,  
               int day )  
{  
    this->month = month;  
    this->day   = day;  
    this->year  = year;  
}
```

//client code

```
DATE date1, date2;  
date1.set(2011,1,1);  
date2.set(2012,2,2);
```

this指针的使用

```
class DATE // DATE.h----Specification file of class DATE
```

```
{ public:
```

```
    DATE( int, int, int );
```

```
    DATE();
```

```
    DATE& Set( int, int, int );
```

```
    int getMonth() const;
```

```
    int getDay() const;
```

```
    int getYear() const;
```

```
    void Print() const;
```

```
    void Increment();
```

```
    void Decrement();
```

```
private:
```

```
    int month;
```

```
    int day;
```

```
    int year;
```

```
};
```

```
//DATE3
```

this指针的使用

```
DATE& DATE::Set(int newYear, int newMonth, int newDay )  
//使用引用返回  
{  
    month = newMonth; day = newDay; year = newYear;  
  
    return *this;  
}                                     //Date.cpp
```

```
DATE date1;                                     //client code  
DATE date2( 1976, 12, 20 );  
  
date1.Set(2011,1,1).Print();  
date2.Set(2012,2,2).Set(2013,3,3).Print();  
  
date2.Print(); // what is the output
```

this指针的使用

```
DATE DATE::Set(int newYear, int newMonth, int newDay )
```

```
//没有使用引用返回
```

```
{
```

```
    month = newMonth; day = newDay; year = newYear;
```

```
    return *this;
```

```
}
```

```
//Date.cpp
```

```
DATE date1;
```

```
DATE date2( 1976, 12, 20 );
```

```
//client code
```

```
date1.Set(2011,1,1).Print();
```

```
date2.Set(2012,2,2).Set(2013,3,3).Print();
```

```
date2.Print(); // what is the output?
```


Constructors

- Three kinds of constructor:
 - 1、 default constructor: constructor with no parameters.
 - 2、 normal constructor: data members are initialized by the parameters.
 - 3、 copy constructor(拷贝构造函数): the object being created is initialized by an existing object.

拷贝构造函数

- 形参类型为该类类型本身且参数传递方式为按引用传递。
- 用一个已存在的该类对象初始化新创建的对象。
- 每个类都必须有拷贝构造函数：
 - 用户可根据自己的需要显式定义拷贝构造函数。
 - 若用户未提供，则该类使用由系统提供的缺省拷贝构造函数。
 - 缺省拷贝构造函数使用逐位复制方式利用已存在的对象来初始化新创建的对象（相当于赋值=）。

拷贝构造函数的一般形式

- 用类类型本身作形式参数。
- 该参数传递方式为按引用传递，避免在函数调用过程中生成形参副本。
- 该形参一般声明为**const**，以确保在拷贝构造函数中不修改实参的值

C::C(const C& obj);

例如: **COMPLEX(const COMPLEX& other);**

拷贝构造函数的作用（1）

- 在声明语句中用一个对象初始化另一个对象。

//假定**C**为已定义的类，则

C obja; //调用**C**的缺省构造函数

C obja(1,2) //调用**C**的有参普通构造函数

/*调用**C**的拷贝构造函数用对象**obja**初始化对象**objb**。如果有为**C**类明确定义拷贝构造函数，将调用这个拷贝构造函数；如果没有为**C**类定义拷贝构造函数，将调用缺省的拷贝构造函数。
*/

C objb(obja); //或

C objb = obja ; //两者等价

拷贝构造函数的作用（2）

- 将一个对象作为实参，以按值调用方式传递给被调函数的形参对象。

假定**C**为已定义的类，**obja**为**C**类对象，且

```
void fun(C temp)
```

```
{ ...
```

```
}
```

则

```
fun(obja);
```

用**obja**来初始化**temp**。如果有为**C**类明确定义拷贝构造函数，将调用这个拷贝构造函数；如果没有为**C**类定义拷贝构造函数，将调用缺省的拷贝构造函数。

//**obja**传递给**fun**函数，创建形参对象**temp**时，调用**C**的拷贝构造函数用对象**obja**初始化对象**temp**，**temp**生存期结束被撤销时，调用析构函数

拷贝构造函数的作用（3）

- 生成一个临时对象作为函数的返回结果：当函数返回一对象时，系统将自动创建一个临时对象来保存函数的返回值。创建此临时对象时调用拷贝构造函数，当函数调用表达式结束后，撤销该临时对象时，调用析构函数。

```
C fun2()      //假定C为已定义类，且  
{  
    C t;  
    ...  
    return t ;  
}
```

则：

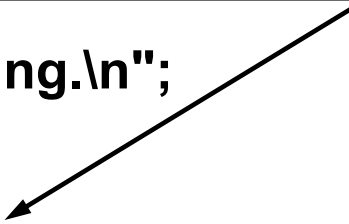
```
b = fun2();
```

t	-->	临时对象	-->	b
拷贝构造函数		赋值运算符		

例

```
class FOO {  
public:
```

形参类型为该类类型本身且参数传递方式为按引用传递,
形参一般声明为const



```
    FOO(int i)  
    {    cout << "Constructing.\n";  
        member = i;  
    }  
    FOO(const FOO& other)  
    {    cout << "Copy constructing.\n";  
        member = other.member;  
    }  
    ~FOO()  
    {    cout << "Destructing.\n"; }  
    int get()  
    {    return member;    }  
};
```

```
private:
```

```
    int member;
```

```
};
```

例（续）

```
void display ( FOO obj )
{
    cout << obj.get() << "\n";
}

FOO get_foo( )
{
    int value;

    cout<< "Input an integer:";
    cin>>value;
    FOO obj(value);

    return obj;
}
```


例（续）

```
int main()
{
    FOO obj1(15);    // 调用一般的构造函数

    FOO obj2 = obj1; // 调用拷贝构造函数

    display(obj2);   // 调用拷贝构造函数用实参obj2初始化形参obj

    obj2=get_foo(); // 调用拷贝构造函数创建作为返回结果的临时对象

    display(obj2);

    return 0;
}                                     //程序FOO
```

运行结果及分析

Constructing.

Copy constructing.

Copy constructing.

15

Destructing.

Input an integer: 7↵

Constructing.

Copy constructing.

Destructing.

Destructing.

Copy constructing.

7

Destructing.

Destructing.

Destructing.

创建obj1

创建obj2，用obj1对其进行初始化

创建obj，obj2初始化obj(display里面)

输出obj的值（display函数里）

撤销obj（执行离开display函数时）

输入7

创建obj（get_foo函数里）

创建作为返回值的临时对象

撤销temp_obj

撤销作为返回值的临时对象

创建obj，obj2初始化obj(display里面)

输出obj的值

撤销obj

撤销obj2

撤销obj1

例：对象作为函数返回值产生的问题

```
//Name.h  
class NAME {  
public:  
    NAME();  
    ~NAME();  
    void show();  
    void set(char* s);  
private:  
    char* str;  
};
```

例：对象作为函数返回值产生的问题

```
NAME::NAME() //Name.cpp
{
    str = NULL;
    cout << "Constructing.\n";
}
NAME::~~NAME()
{
    cout << "Destructing.\n";
    if (str != NULL)
        delete []str;
}

void NAME::show()
{
    cout << str << "\n";
}
```

```
void set(char* s)
{
    if (str!=NULL)
        delete []str;

    str = new char[strlen(s) + 1];

    if (str!=NULL)
        strcpy(str, s);
}
```

分析

```
NAME get_name()
{
    NAME obj; //①
    char temp_str[250]; //②
    cout << "Input your name: ";
    cin >> temp_str;
    obj.set(temp_str); //③
    return obj; //④
} //⑤
```

```
void main()
{
    NAME myname;
    myname = get_name();
    myname.show();
} //程序Name1
```

1. 创建myname

myname.str

NULL

2. 调用get_name

①创建局部对象obj

obj.str

2000H

②读入temp_str

L e e \0

③调用set函数, new、复制2000H

L e e \0

④ get_name函数返回, 创建临时对象, 调用缺省的拷贝构造函数

临时对象.str

2000H

分析（续）

⑤撤销局部变量**obj**：由**new**运算分配的空间（地址为**2000H**）通过**delete**运算被释放。

3. 临时对象赋给**myname**，调用系统提供的缺省赋值运算符，实施浅复制，从而**myname.str**的值置为**2000H**。

4. 撤销临时对象，释放其**str**成员指向的内存空间（地址为**2000H**）。因为该内存空间已释放，再次释放会出现问题。

5. **myname.show()**输出对象**myname**的**str**成员指向的内存空间（地址为**2000H**）中的内容，该内容不确定。

6. 撤销对象**myname**，释放其**str**成员指向的内存空间（地址为**2000H**）。**Delete**运算出现问题。

程序运行实例

Constructing.

创建myname

Constructing.

创建obj

Input your name: Lee↵

输入Lee

Destructing.

撤销obj, delete str

Destructing.

撤销函数返回的临时对象

Null pointer assignment delete出问题

××× （此行输出的字符串是不确定的！）

Destructing.

撤销myname

Null pointer assignment delete出问题

- 以上是在TC3运行环境中的一个运行实例，在有的运行环境（VC）中，程序运行到输入字符串后即出现内存错误，不能继续运行。

修改：加入拷贝构造函数的定义

```
NAME::NAME(NAME& other)
{
    cout << "Copy Constructing.\n";
    if (other.str == NULL)
    {
        str=NULL;
        return;
    }
    str=new char[strlen(other.str)+1];
    if (str!=NULL)
        strcpy(str, other.str);
}
```

④ **get_name**函数返回，调用拷贝构造函数创建临时对象：

obj.str

2000H → L | e | e | \0

临时对象.str

3000H → L | e | e | \0

```
void main()
{
    NAME myname;
    myname = get_name();
    myname.show();
}
```


修改：加入赋值运算符重载函数的定义

```
NAME& operator=(const NAME& other)    //程序Name2
```

```
{
```

```
    if (other.str == NULL)
```

```
    {    str=NULL;
        return *this;
    }
```

```
    if (str!=NULL) delete []str;
```

```
    str=new char[strlen(other.str)+1];
```

```
    if (str!=NULL) strcpy(str, other.str);
```

```
    return *this; /*若函数的返回值类型为NAME，则此语句会引起
                  对拷贝构造函数的调用;若函数的返回值类型为
                  NAME&，则不调用拷贝构造函数*/
```

```
}
```

```
void main()
```

```
{
```

```
    NAME myname;
    myname = get_name();
    myname.show();
```

```
}
```

修改后程序的运行实例

Constructing.

Constructing.

Input your name: Lee↵

Copy Constructing.

Destructing.

Destructing.

Lee

Destructing.

关于拷贝构造函数的提示

- 对于不含指针成员类，使用系统提供（编译器合成）的缺省拷贝构造函数即可。
- 缺省拷贝构造函数使用浅复制策略，因此对含指针成员类并不能满足需要。
- 含指针成员类通常应在构造函数（及拷贝构造函数）中分配内存，在析构函数中释放内存。

Composition(组合)

- **Composition**
 - **Class has objects of other classes as members.**
- **Construction of objects**
 - **Member objects constructed in order declared, NOT in order of constructor's member initializer list**
 - **Constructed before their enclosing class objects (host objects)**
 - **Constructors called inside out**
 - **Destructors called outside in**

Composition(组合)

```
#include "date.h"
class Employee
{
public:
    Employee( char *, char *, int, int, int, int, int, int );
    void print() const;
    ~Employee();
private:
    char firstName[ 25 ];
    char lastName[ 25 ];
    Date birthDate;
    Date hireDate;
};
```

Composition(组合)

```
Employee::Employee( char *fname, char *lname,           //Employee.cpp
                   int bmonth, int bday, int byear,
                   int hmonth, int hday, int hyear )
    : hireDate( hmonth, hday, hyear )
      birthDate( bmonth, bday, byear ),
{
    int length = strlen( fname );
    length = ( length < 25 ? length : 24 );
    strncpy( firstName, fname, length );
    firstName[ length ] = '\0';
    length = strlen( lname );
    length = ( length < 25 ? length : 24 );
    strncpy( lastName, lname, length );
    lastName[ length ] = '\0';
}
```

Composition(组合)

```
void Employee::print() const  
{  
    cout << lastName << ", " << firstName << "\nHired: ";  
    hireDate.print();  
    cout << " Birth date: ";  
    birthDate.print();  
}
```

```
Employee::~~Employee()  
{  
    cout << "Employee object destructor: "  
        << lastName << ", " << firstName << endl;  
}
```

Composition(组合)

```
//Client.cpp
#include "employ1.h"
int main()
{
    Employee e( "Bob", "Jones", 7, 24, 1949, 3, 12, 1988 );
    e.print();
    return 0;
}
```


对象成员的初始化

- 对象成员：在类中声明的具有类类型的数据成员
- 为了初始化对象成员，类的构造函数必须调用对象成员所属类的构造函数，如：

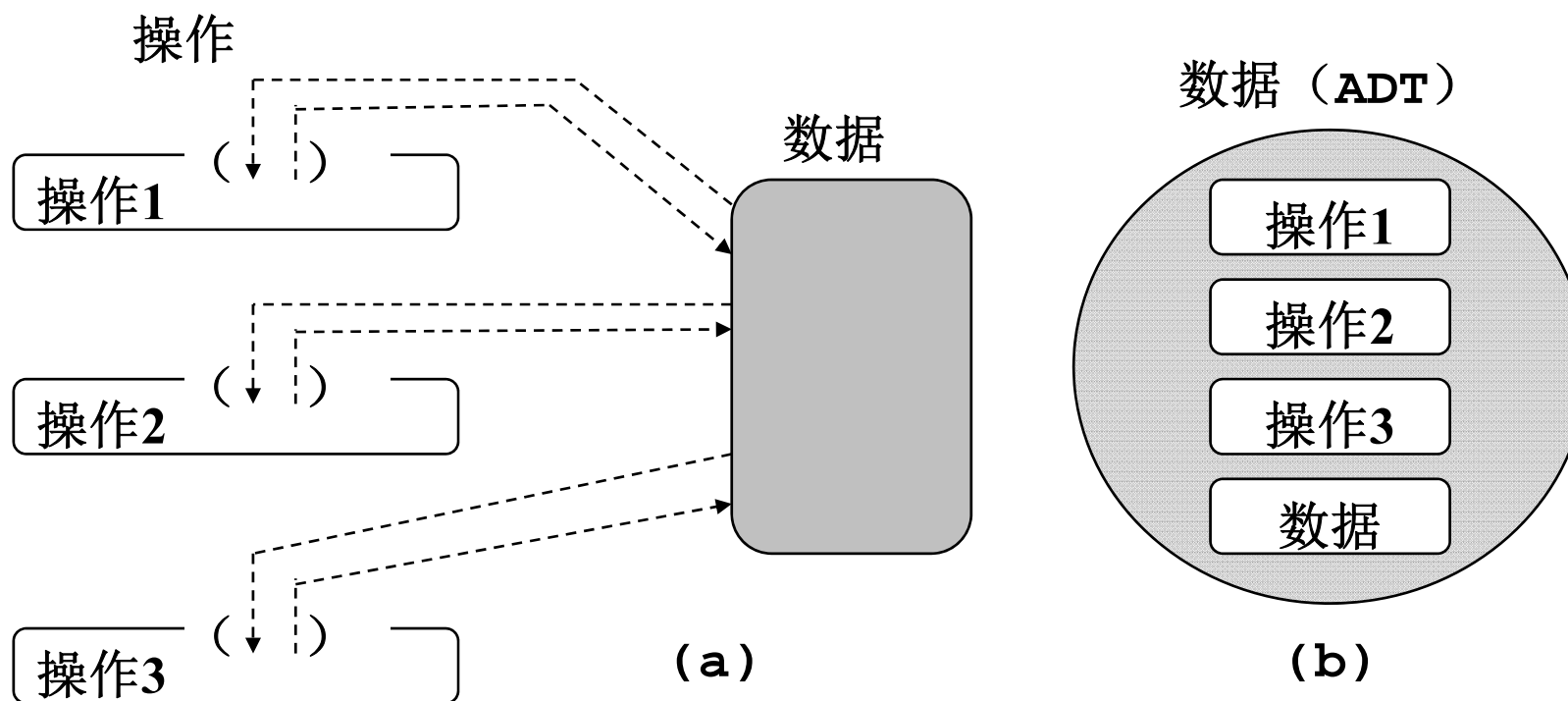
```
class C  
{  
    CLASS_1 obj1;  
    CLASS_2 obj2;  
    ...  
    CLASS_N objn;  
};
```

```
C::C(参数表0): obj1(参数表1), obj2(参数表2), ..., objn(参数表n)  
{    ...    }
```

关于面向对象程序设计的若干基本问题

- 面向对象程序设计
 - 是一种理念（idea）：思维和方法论的问题。
 - 是某种语言里面支持面对对象的具体机制：程序语言的运用问题
 - （类和对象、继承、多态性和模板等）。

数据vs.操作

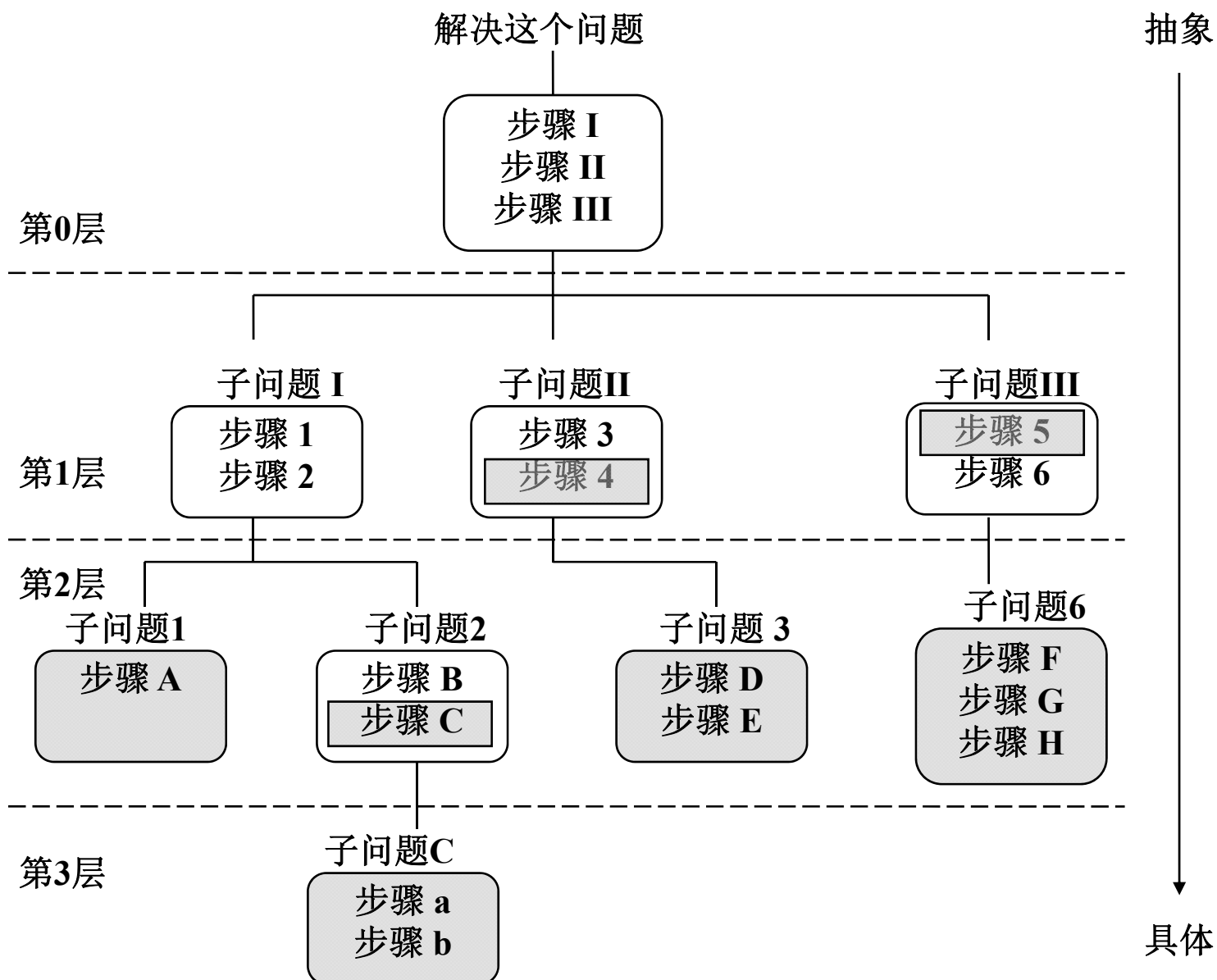


谁主动谁被动？

数据vs.操作

- 数据与操作“主动与被动”的关系反映出人类思考问题、寻求解决方案的两大思路：
 - 面向过程的思考方式
 - 面向对象的思考方式
- 思考方式不同就会导致算法结构不同，从而最终导致程序的不同。

面向过程设计



面向过程设计

- 利用这种方式思考问题时，我们把问题看成是什么？
- 在构建大型软件系统时，面向过程的设计往往导致程序有两大致命缺陷：
 - 导致程序结构不灵活。若高层算法需要修改，那么可能底层的算法也因此需要修改。
 - 导致代码难以复用。

面向对象的程序设计

- 面向对象的程序设计（**object oriented design, OOD**）已被证明是开发和维护大型软件的更好的设计方式，它在程序结构、代码复用、封装隐藏等方面，有着面向过程设计难以企及的优势。

面向对象的程序设计

- Q: 什么是对象?
 - A: 一般说来, **任何事物**都可以看成对象。我们要考察或研究现实或思维世界中的某个实体, 那么它就成为我们的对象。

- Q: OOD把问题视作什么?
 - A: 是把问题视作各类**实体（对象）**的组合。

- Q: 它关注对象的什么?
 - A: 它关注对象中包含的**数据**及作用于这些数据之上的**操作**, 也需要关注对象之间的**关系和相互作用**。

面向对象的程序设计

- Q: 如何确认问题中的对象?
 - 一般来说，我们需要在问题域中寻求对象，即仔细研究问题的定义，从中搜索各重要的名词和动词。名词很可能就是对象，而动词可能就是对象的操作。

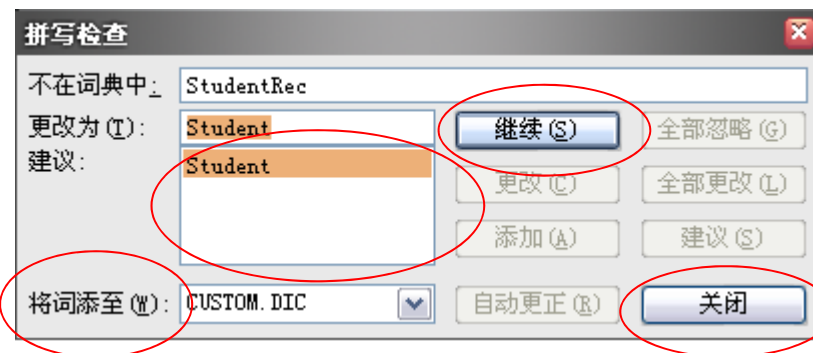
面向对象的程序设计

- 下面给出问题的一部分：

“。。。。。。程序必须解决学生的图书馆帐户。允许学生通过帐户借书、还书，允许学生往帐户中存钱，逾期罚款从帐户中扣除。。。。。”
- 重要名词：学生 帐户
- 重要动词：借书、还书、存款、扣除
- 这个问题可能存在两类对象：学生**student**和帐户**account**。
- **account**的数据属性应该包括：借书信息、余额、每次存款、取（罚）款的时间等。操作应该包括借书**BorrowBooks**、还书**ReturnBooks**、存钱**Deposit**、罚款**Fine**。
- **student**应该有哪些数据和操作，需要这个问题的更多信息才能确定。

面向对象的程序设计

- 存在两类对象：
 - 反映现实和思维世界中的事物或概念的对象。例如“房子”、“银行帐号”、“纳税人”、“计数器”等；
 - 用以实现程序的各种工具对象：各种按钮、对话框、浏览器等。



面向对象的程序设计

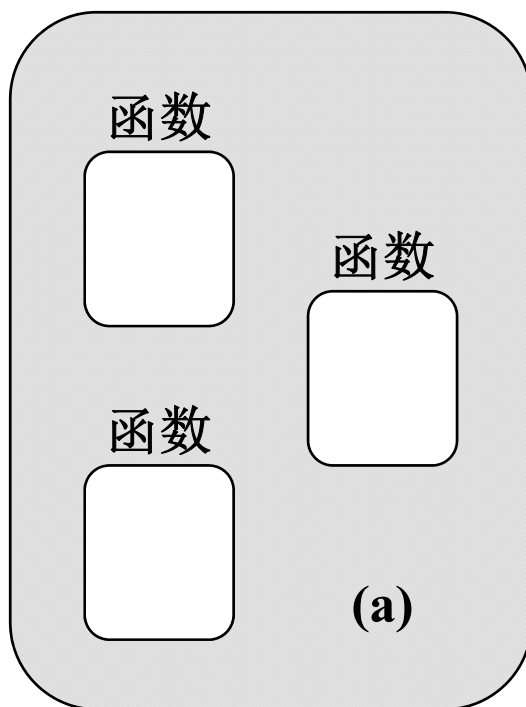
- 对于第[1]类的对象，要将其翻译为程序算法中的对象（即确定其数据和操作），往往会发生**剧烈的变异**。
- 虽然有一些抽象概念是有确切的物质形态与之相对应，但往往我们要转化的仍是这些抽象概念。
- 有一些对象确实存在于物质世界中，但如果要转化为程序中的对象，就要抽取我们需要的属性。
- 运用面向对象时，人类思维的往往需要**跳跃式突变**，很多时候绝无面向过程中的“自然而然”。这也是初学者**感到困难**的地方。

面向对象的程序设计

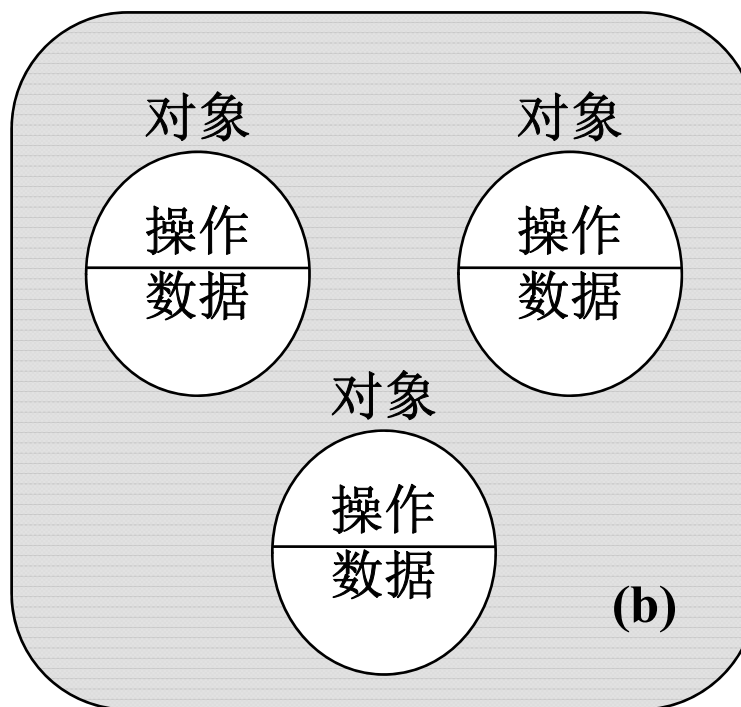
- 模拟现实世界或思维世界，**不能奴隶式**的追随我们之所见；而是需要有**洞察力**，用思维的力量直达**对象的本质**，才能确定对象及其操作。
- **不断实践**，是提高面向对象程序设计能力的唯一途径。

面向过程 vs. 面向对象

程序



程序



术语

