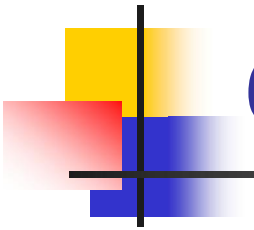
第六章　类成员(一)

面向对象程序设计(C++)



6 类成员（一）

6.1 成员变量与成员函数

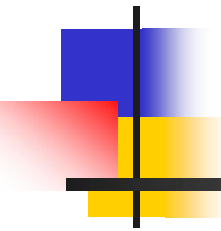
6.2 this指针

6.3 成员对象

6.4 const(常量)

6.5 const对象与const成员函数

6.6 静态成员变量与静态成员函数



6.1 成员变量与成员函数

6.1.1 成员变量

- 可以是基本类型、构造类型，甚至是一个另一个类的对象、对象指针或引用；

```
class X{
private:
    int  basetypemem;     // 基本类型
    char str[20];         // 构造类型
    Cstack   obj_stack;   // 另一个类的对象,stack必须
                          //在X之前定义
    CTime   *obj_time;    // 另一个类的对象指针
    CTime    & obj_time;  // 另一个类的对象引用
    …;
}
```

6.1.1 成员变量(续)

- 提前声明：一个类提前声明之后，这个类的对象指针、引用可以作为另一个类的成员变量。但这个类的对象不能作另一个类的成员变量。

```
class Stack;
class StackofStack{
   int   topStack;
    Stack *stackstorage;
 };
```

6.1.2 成员函数

- 在类的定义体外定义：

```
class Stack {
    int empty();
    void pop();
    int  top();
    void push(int);
};

int Stack::empty ()
{
     return (nb_elements==0);
}
```

6.1.2 成员函数(续)

- 在定义体内定义

```
class Stack {
    int empty() {  return (nb_elements==0); }
    void pop();
    int  top();
    void push(int);
};
```

在类的定义体内定义时，自动成为内联函数。

6.1.3 对象与类

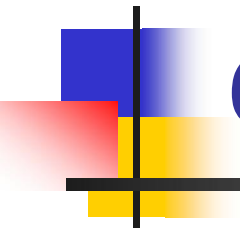
- 创建对象

  类　对象1，对象2，对象n；

  Stack intstack(10), stringstack;

- 存储分配

  一个类所有的对象的成员变量(除了静态变量)都有独立空间，但同一个类的所有对象成员共享同一组成员函数。



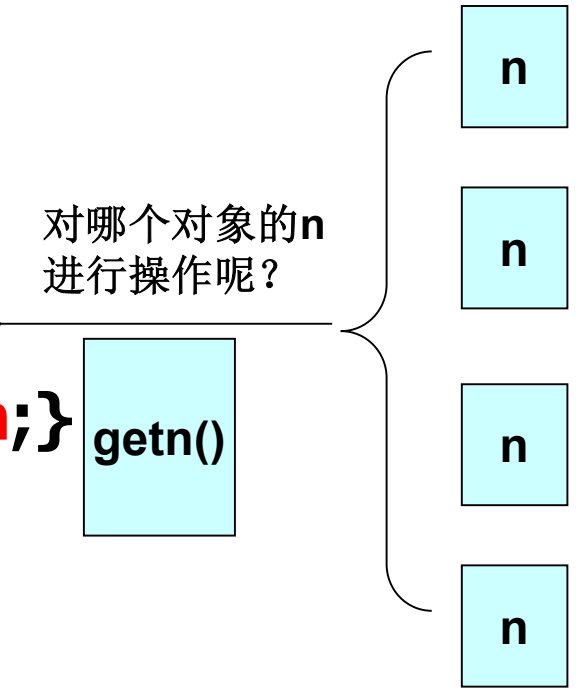
6.2 this指针

6.2 this指针

- 问题：**当对象的多个实例同时存在时，在一个类的成员函数内部引用一个数据成员，并没有指明一个类的具体实例。**

  编译器如何决定引用哪个对象的数据成员呢？

```
Eg:  class  CTest{
        private:
           int n;
        public:
           int getn() {   return n;}
        }
```



6.2.1 隐藏的this 指针

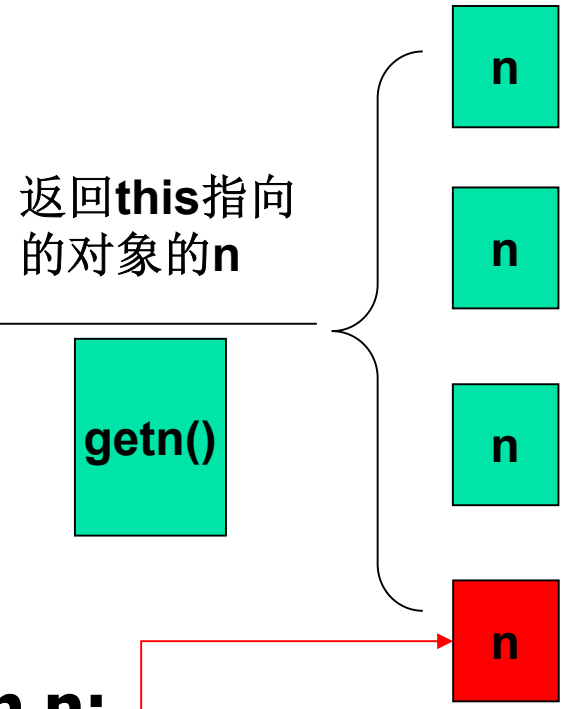
- 解决办法：编译器给成员函数传递一个隐藏的对象指针参数，该指针总是指向当前要引用的对象。称为“**this指针**”

- 实现方法：类**X**中的每个成员函数都隐式地声明**this**指针；

```
int CTest::getn(X *const this)
{
    return this->n ; //等价于return n;
 }
```

操作的是 this->n！

```
class  CTest{
        private:
            int n;
        public:
            int getn();
        }
int CTest::getn(X *const this)
{
    return this->n ; //等价于return n;
 }
```



例1：

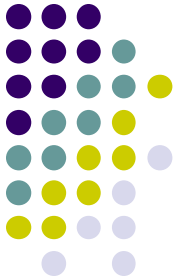
```
class Test {
public:
   Test( int = 0 );              // default constructor
   void print() const;
private:
   int x;
};

Test::Test( int a ) { x = a; }  // constructor
void Test::print() const
{
   cout << "        x = " << x
        << "\n  this->x = " << this->x
        << "\n(*this).x = " << ( *this ).x << endl;
}
```

例1：

```
int main()
{
   Test testObject( 12 );
    testObject.print();
}
```

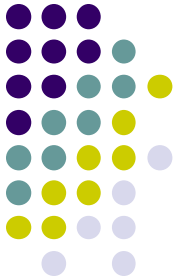
运行结果：

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



例2：

```
Struct Link{
   Link* pre;
   Link* suc;
   int data;

   Link* insert(int x)
   {
      return pre=new Link{pre,this,x};
   }

   void remove()
   {
      if (pre) pre->suc=suc;
      if (suc) suc->pre=pre;
      delete this;
   }
};
```



6.2.2 this是一个常量指针

- **X * const this**：不能修改**this**指针的值；

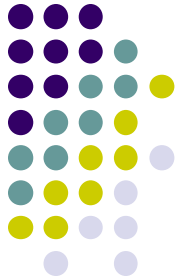
```
int Ctest::setn()
{   Ctest t1;
    this = new t1;   //非法操作，不能改变this指针
}
```

6.2.3 this指针的用途之一

- 通过返回*this的引用，可以实现成员函数的链式调用

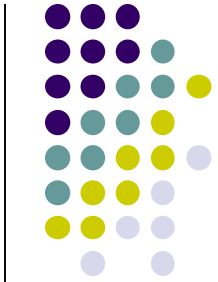
```
Class X{
      X &assign(){ …; return(*this)} ;
      X &setvalue(){…; return(*this)};
      …;
  }

X objX;
objX.assign().setvalue().assign();
```

（. 从左向右结合）

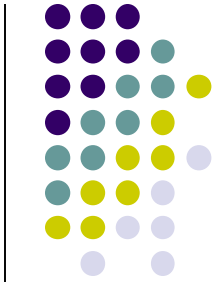


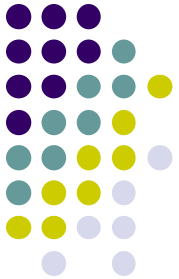
```
// Fig. 7.8: time6.h
class Time {
public:
   Time( int = 0, int = 0, int = 0 );  // default constructor
   Time &setTime( int, int, int );    // set hour, minute,
      second
   Time &setHour( int );                // set hour
   Time &setMinute( int );              // set minute
   Time &setSecond( int );  // set second
   int getHour() const;     // return hour
   int getMinute() const;   // return minute
   int getSecond() const;   // return second
   void printMilitary() const;  // print military time
   void printStandard() const;  // print standard time
```

```
private:
   int hour;              // 0 - 23
   int minute;             // 0 - 59
   int second;             // 0 - 59
};


// Fig. 7.8: time.cpp
#include "time6.h"
#include <iostream.h>

Time::Time( int hr, int min, int sec )
   { setTime( hr, min, sec ); }

Time &Time::setTime( int h, int m, int s )
{
```



```
   setHour( h );
      setMinute( m );
      setSecond( s );
   return *this;   // enables cascading
}

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

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

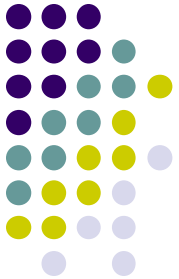




```
Time &Time::setSecond( int s )
{
   second = ( s >= 0 && s < 60 ) ? s : 0;
   return *this;   // enables cascading
}

int Time::getHour() const { return hour; }
int Time::getMinute() const { return minute; }
int Time::getSecond() const { return second; }

void Time::printMilitary() const
{
cout << ( hour < 10 ? "0" : "" ) << hour << ":"
        << ( minute < 10 ? "0" : "" ) << minute;
}
```

```
void Time::printStandard() const
{
   cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
        << ":" << ( minute < 10 ? "0" : "" ) << minute
        << ":" << ( second < 10 ? "0" : "" ) << second
        << ( hour < 12 ? " AM" : " PM" );
}

// Fig. 7.8: fig07_08.cpp
#include <iostream.h>
#include "time6.h"
int main()
{
   Time t;
   t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
   cout << "Military time: ";
```

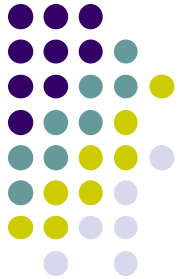
```
t.printMilitary();
   cout << "\nStandard time: ";
   t.printStandard();

cout << "\n\nNew standard time: ";
   t.setTime( 20, 20, 20 ).printStandard();
   cout << endl;

   return 0;
}
```



运行结果：

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

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

- 思考：**假如按值返回*this，问程序的输出是什么?**



- 思考：**假如按值返回*this，问程序的输出是什么?**
- 解答：

Military time: 18:00

Standard time: 6:00:00 PM

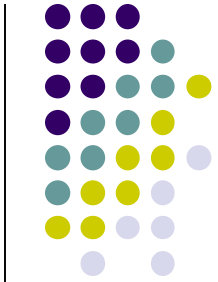
New standard time: 8:20:20 PM

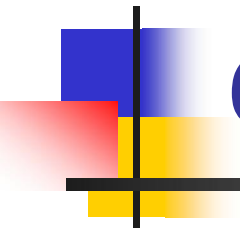
6.2.4 this指针的重要用途之二（略）

- 防止自身赋值：

```
class String{
private:
    char * str;
public:
    String & assign(const String & s) {
        int tmplen;
        if(this == &s) {
            error("自身赋值" );
            return(*this);
        }
        else{
```

```
            tmplen= strlen(s.str);
            delete ( str)
            str = new char[tmplen+1];
            strcpy(str,s.str);
        }
    }
    … ; //
};

String   s1(“How are you?”), s2(“Fine”);
s1.assign(s2);
s2.assign(s2);
```





6.3 成员对象

6.3.1 成员对象（对象成员）

- 类的数据成员可以是另一个类的对象，这种数据成员称为“成员对象”。

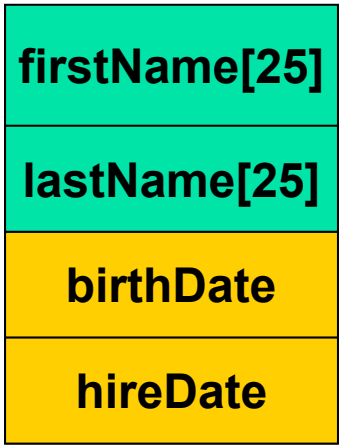
```
class Employee {
private:
   char firstName[ 25 ];
   char lastName[ 25 ];
   const Date birthDate;
   const Date hireDate;
   …; // 以下省略

};
```

6.3.2 初始化次序

- 构造函数的执行次序：

  对象成员的构造函数先初始化，然后才是包含它的类的构造函数。有多个对象成员时，按照在类的定义中声明的次序初始化。

Employee:



先执行：

birthDate.Date(…)

hireDate.Date(…)

再执行：

e.Employee(…)

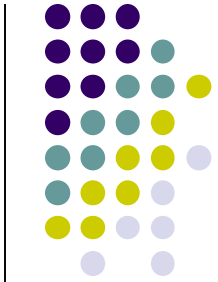
6.3.3 成员初始化表

- 如何初始化？成员初始化表：

类的构造函数名(类构造函数参数列表+成员构造函数参数列表):成员对象1(参数列表),成员对象2(参数列表)

```
Employee::Employee(
                    char *fname, char *lname,
                    int bmonth, int bday, int byear,
                    int hmonth, int hday, int hyear )
                  : birthDate( bmonth, bday, byear ),
                    hireDate( hmonth, hday, hyear )
```

例：成员对象(对象成员)的初始化



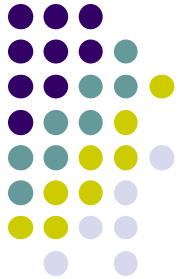
```
1.  #include "date1.h“
2.  class Employee {
3.  public:
4.     Employee( char *, char *, int, int, int, int, int, int );
5.     void print() const;
6.     ~Employee();  // provided to confirm destruction
    order
7.  private:
8.     char firstName[ 25 ];
9.     char lastName[ 25 ];
10.    const Date birthDate;
11.    const Date hireDate;
12. };
```

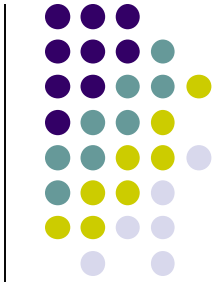
```
13. Employee::Employee( char *fname, char *lname,
14.                     int bmonth, int bday, int byear,
15.                     int hmonth, int hday, int hyear )
16.    : birthDate( bmonth, bday, byear ),
17.      hireDate( hmonth, hday, hyear )
18. {
19.    …; // 构造函数体;
20. }

21. void Employee::print() const
22. {
23.  …; // 略
24. }

25. // Destructor: provided to confirm destruction order
26. Employee::~Employee()
```

构造函数的参数表

成员对象初始化表

```
27. {
28.    cout << "Employee object destructor: "
29.         << lastName << ", " << firstName << endl;
30. }

31. #include <iostream.h>
32. #include "emply1.h"
33. int main()
34. {
35.    Employee e( "Bob", "Jones", 7, 24, 1949, 3, 12,
    1988 );
36.    return 0;
37. }
```





输出：

Date object constructor for date 7/24/1949

Date object constructor for date 3/12/1988

Employee object constructor Bob Jones

….

Employee object destructor Bob Jones

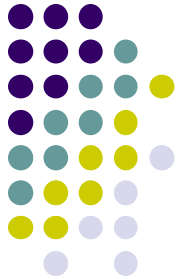
Date object destructor for date 3/12/1988

Date object destructor for date 7/24/1949

即：由内向外建立，由外向内删除

例1：

```
class C {
public:
   C( int i ){ x=i;
                cout<<“C”<<endl;
              }
private:
   int x;
};

class C2 {
public:
   C2():C1(1) {
                 cout<<“C1”<<endl;
               }
private:
   C C1;
};

C2 object;
```



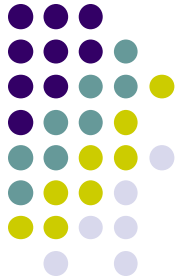
例1：

```
class C {
public:
   C( int i ){ x=i;
               cout<<“C”<<endl;
             }
private:
   int x;
};

class C2 {
public:
   C2():C1(1) {
                cout<<“C1”<<endl;
              }
private:
   C C1;
};

C2 object;   // 初始化列表在构造函数执行前执行，后者覆盖前者
```

6.3.4 注意

- 初始化表只能出现在构造函数的定义体中，不能出现在构造函数的声明中。
  (MS VC++没有这个限制。)
- 若不提供成员初始化值，则编译器隐式调用成员对象的默认构造函数。
- 初始化列表比构造函数体效率高
- const 成员必须在成员初始化表中初始化

例2：



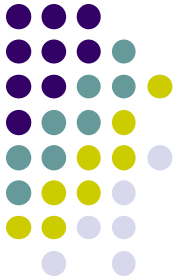
```
class C {
private:
   const int a;
public:
   C():a(10){};
};
```

或者

```
class C {
private:
   int &a;
public:
   C(int i):a(i){};   // 引用初始化必须赋值
};
```

例3：

```
class C {
private:
   int x;
public:
   C(int i):{x=i;}
};

class C1: public C {
public:
   C1():C(100){}   // 子类初始化基类的私有成员
};
```

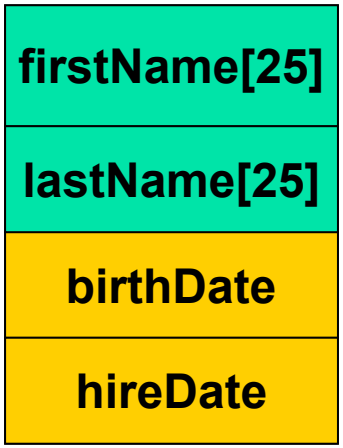


6.3.4 析构次序

- 析构函数的执行次序：

  先执行对象的析构函数，然后才执行对象成员的析构函数。有多个对象成员时，按照在类的定义中声明的次序相反的次序析构对象成员。

Employee:



先删除
employee对象；
然后删除
hireDate对象
最后删除
birthDate对象

1. 指出下列程序中 this指针的错误用法，并说明理由：

```
class X{
      int i;
public:
      X(){ i=0;}
      void use_this(){
            X x;
            this = &x;
            *this.i=2;
            X * This;
            This = this;
            This = &x;
       }
 };
```

```
int main(){
            X x;
            this -> x =12;
       }
```

1. 指出下列程序中 this指针的错误用法，并说明理由：

```
class X{
      int i;
public:
      X(){ i=0;}
      void use_this(){
            X x;
            this = &x; //error
            *this.i=2;  //error
            X * This;
            This = this;
            This = &x;
       }
 };
```

```
int main(){
            X x;
            this -> x =12;  //error
      }
```