

# 类与对象(中)

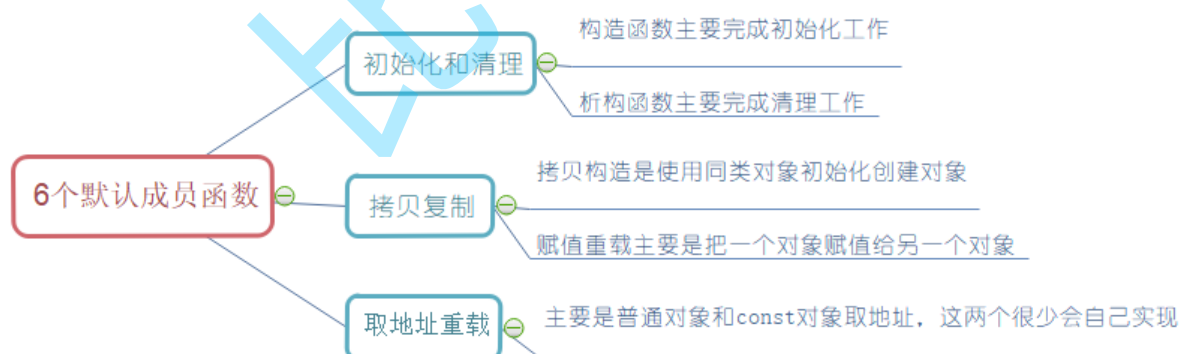
## [本节目标]

- 1. 类的6个默认成员函数
- 2. 构造函数
- 3. 析构函数
- 4. 拷贝构造函数
- 5. 赋值操作符重载
- 6. 默认拷贝构造与赋值运算符重载的问题
- 7. const成员函数
- 8. 取地址及const取地址操作符重载

### 1. 类的6个默认成员函数

如果一个类中什么成员都没有，简称为空类。空类中什么都没有吗？并不是的，任何一个类在我们不写的情况下，都会自动生成下面6个默认成员函数。

```
1 class Date {};
```



### 2. 构造函数

#### 2.1 概念

对于以下的日期类：

```
1 class Date
```

```

2  {
3  public:
4      void SetDate(int year, int month, int day)
5      {
6          _year = year;
7          _month = month;
8          _day = day;
9      }
10
11     void Display()
12     {
13         cout << _year << "-" << _month << "-" << _day << endl;
14     }
15
16 private:
17     int _year;
18     int _month;
19     int _day;
20 };
21
22 int main()
23 {
24     Date d1, d2;
25     d1.SetDate(2018, 5, 1);
26     d1.Display();
27
28     Date d2;
29     d2.SetDate(2018, 7, 1);
30     d2.Display();
31     return 0;
32 }

```

对于Date类，可以通过SetDate公有的方法给对象设置内容，但是如果每次创建对象都调用该方法设置信息，未免有点麻烦，那能否在对象创建时，就将信息设置进去呢？

**构造函数是一个特殊的成员函数，名字与类名相同，创建类类型对象时由编译器自动调用，保证每个数据成员都有一个合适的初始值，并且在对象的生命周期内只调用一次。**

## 2.2 特性

**构造函数是特殊的成员函数，需要注意的是，构造函数的虽然名称叫构造，但是需要注意的是构造函数的主要任务并不是开空间创建对象，而是初始化对象。**

**其特征如下：**

1. 函数名与类名相同。
2. 无返回值。
3. 对象实例化时编译器**自动调用**对应的构造函数。
4. 构造函数可以重载。

```

1  class Date
2  {

```

```

3 public :
4     // 1.无参构造函数
5     Date ()
6     {}
7
8     // 2.带参构造函数
9     Date (int year, int month , int day )
10    {
11        _year = year ;
12        _month = month ;
13        _day = day ;
14    }
15 private :
16     int _year ;
17     int _month ;
18     int _day ;
19 };
20
21 void TestDate()
22 {
23     Date d1; // 调用无参构造函数
24     Date d2 (2015, 1, 1); // 调用带参的构造函数
25
26     // 注意：如果通过无参构造函数创建对象时，对象后面不用跟括号，否则就成了函数声明
27     // 以下代码的函数：声明了d3函数，该函数无参，返回一个日期类型的对象
28     Date d3();
29 }

```

5. 如果类中没有显式定义构造函数，则C++编译器会自动生成一个无参的默认构造函数，一旦用户显式定义编译器将不再生成。

```

1 class Date
2 {
3 public:
4     /*
5     // 如果用户显式定义了构造函数，编译器将不再生成
6     Date (int year, int month, int day)
7     {
8         _year = year;
9         _month = month;
10        _day = day;
11    }
12    */
13 private:
14     int _year;
15     int _month;
16     int _day;
17 };
18
19 void Test()
20 {
21     // 没有定义构造函数，对象也可以创建成功，因此此处调用的是编译器生成的默认构造函数

```

```

22     Date d;
23 }

```

6. 无参的构造函数和全缺省的构造函数都称为默认构造函数，并且默认构造函数只能有一个。注意：无参构造函数、全缺省构造函数、我们没写编译器默认生成的构造函数，都可以认为是默认成员函数。

```

1  // 默认构造函数
2  class Date
3  {
4  public:
5      Date()
6      {
7          _year = 1900 ;
8          _month = 1 ;
9          _day = 1;
10     }
11
12     Date (int year = 1900, int month = 1, int day = 1)
13     {
14         _year = year;
15         _month = month;
16         _day = day;
17     }
18
19 private :
20     int _year ;
21     int _month ;
22     int _day ;
23 };
24
25 // 以下测试函数能通过编译吗?
26 void Test()
27 {
28     Date d1;
29 }

```

7. 关于编译器生成的默认成员函数，很多童鞋会有疑惑：在我们不实现构造函数的情况下，编译器会生成默认的构造函数。但是看起来默认构造函数又没什么用？d对象调用了编译器生成的默认构造函数，但是d对象year/month/\_day，依旧是随机值。也就说在这里**编译器生成的默认构造函数并没有什么卵用？？**

解答：C++把类型分成内置类型(基本类型)和自定义类型。内置类型就是语法已经定义好的类型：如int/char...，自定义类型就是我们使用class/struct/union自己定义的类型，看看下面的程序，就会发现编译器生成默认的构造函数会对自定义类型成员\_t调用的它的默认成员函数

```

1  class Time
2  {
3  public:
4      Time()
5      {
6          cout << "Time()" << endl;

```

```

7     _hour = 0;
8     _minute = 0;
9     _second = 0;
10 }
11 private:
12     int _hour;
13     int _minute;
14     int _second;
15 };
16
17 class Date
18 {
19 private:
20     // 基本类型(内置类型)
21     int _year;
22     int _month;
23     int _day;
24
25     // 自定义类型
26     Time _t;
27 };
28
29 int main()
30 {
31     Date d;
32     return 0;
33 }

```

## 8. 成员变量的命名风格

```

1 // 我们看看这个函数，是不是很僵硬？
2 class Date
3 {
4 public:
5     Date(int year)
6     {
7         // 这里的year到底是成员变量，还是函数形参？
8         year = year;
9     }
10 private:
11     int year;
12 };
13
14 // 所以我们一般都建议这样
15 class Date
16 {
17 public:
18     Date(int year)
19     {
20         _year = year;
21     }

```

```

22 private:
23     int _year;
24 };
25
26 // 或者这样。
27 class Date
28 {
29 public:
30     Date(int year)
31     {
32         m_year = year;
33     }
34 private:
35     int m_year;
36 };
37
38 // 其他方式也可以的，主要看公司要求。一般都是加个前缀或者后缀标识区分就行。

```

### 3.析构函数

#### 3.1 概念

前面通过构造函数的学习，我们知道一个对象时怎么来的，那一个对象又是怎么没的呢？

析构函数：与构造函数功能相反，析构函数不是完成对象的销毁，局部对象销毁工作是由编译器完成的。而对象在销毁时会自动调用析构函数，完成类的一些资源清理工作。

#### 3.2 特性

析构函数是特殊的成员函数。

其特征如下：

1. 析构函数名是在类名前加上字符~。
2. 无参数无返回值。
3. 一个类有且只有一个析构函数。若未显式定义，系统会自动生成默认的析构函数。
4. 对象生命周期结束时，C++编译系统系统自动调用析构函数。

```

1  typedef int DataType;
2  class SeqList
3  {
4  public :
5      SeqList (int capacity = 10)
6      {
7          _pData = (DataType*)malloc(capacity * sizeof(DataType));
8          assert(_pData);
9
10         _size = 0;
11         _capacity = capacity;

```

```

12     }
13
14     ~SeqList()
15     {
16         if (_pData)
17         {
18             free(_pData );    // 释放堆上的空间
19             _pData = NULL;    // 将指针置为空
20             _capacity = 0;
21             _size = 0;
22         }
23     }
24
25 private :
26     int* _pData ;
27     size_t _size;
28     size_t _capacity;
29 };

```

5. 关于编译器自动生成的析构函数，是否会完成一些事情呢？下面的程序我们会看到，编译器生成的默认析构函数，对会自定类型成员调用它的析构函数。

```

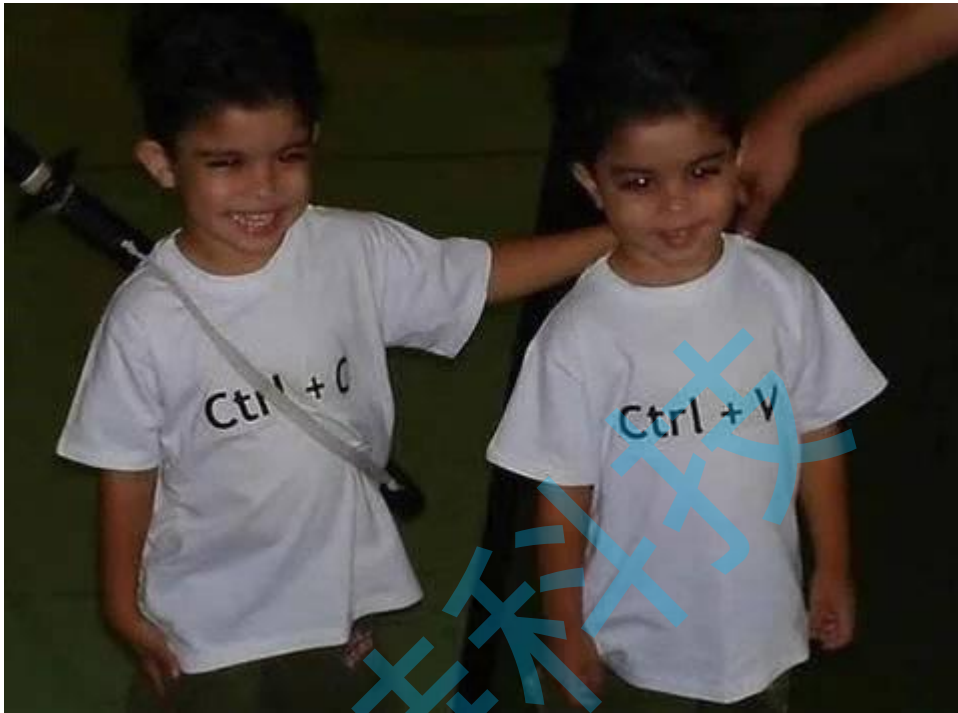
1  class String
2  {
3  public:
4      String(const char* str = "jack")
5      {
6          _str = (char*)malloc(strlen(str) + 1);
7          strcpy(_str, str);
8      }
9
10     ~String()
11     {
12         cout << "~String()" << endl;
13         free(_str);
14     }
15 private:
16     char* _str;
17 };
18
19 class Person
20 {
21 private:
22     String _name;
23     int _age;
24 };
25
26 int main()
27 {
28     Person p;
29     return 0;
30 }

```

## 4. 拷贝构造函数

### 4.1 概念

在现实生活中，可能存在一个与你一样的自己，我们称其为双胞胎。



那在创建对象时，可否创建一个与一个对象一模一样的新对象呢？

**构造函数：**只有单个形参，该形参是对本类类型对象的引用(一般常用const修饰)，在用已存在的类类型对象创建新对象时由编译器自动调用。

### 4.2 特征

拷贝构造函数也是特殊的成员函数，其特征如下：

1. 拷贝构造函数是构造函数的一个重载形式。
2. 拷贝构造函数的参数只有一个且必须使用引用传参，使用传值方式会引发无穷递归调用。

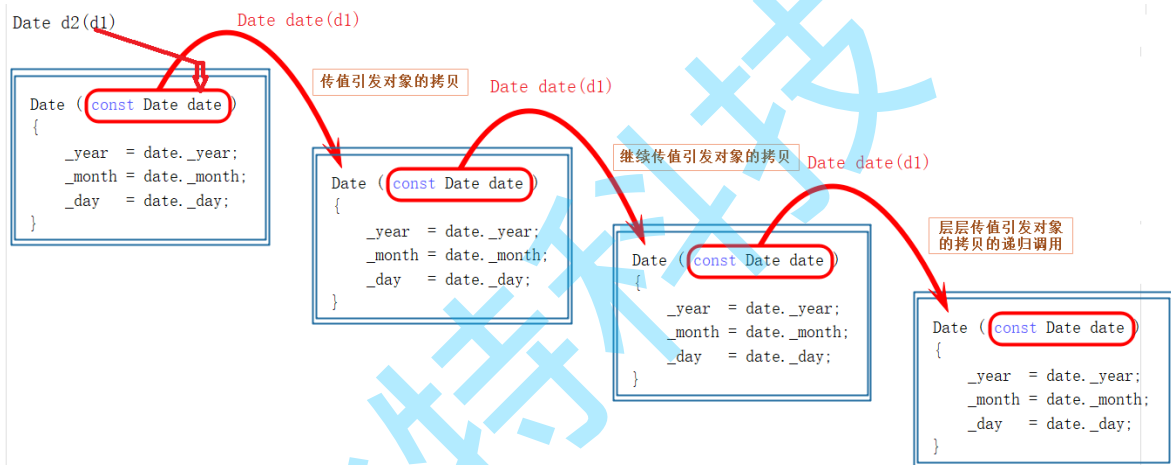
```
1  class Date
2  {
3  public:
4      Date(int year = 1900, int month = 1, int day = 1)
5      {
6          _year = year;
7          _month = month;
8          _day = day;
9      }
10
11     Date(const Date& d)
12     {
```



```

13     _year = d._year;
14     _month = d._month;
15     _day = d._day;
16 }
17 private:
18     int _year;
19     int _month;
20     int _day;
21 };
22
23 int main()
24 {
25     Date d1;
26     Date d2(d1);
27
28     return 0;
29 }

```



3. 若未显示定义，系统生成默认的拷贝构造函数。默认的拷贝构造函数对象按内存存储按字节序完成拷贝，这种拷贝我们叫做浅拷贝，或者值拷贝。

```

1  class Date
2  {
3  public:
4      Date(int year = 1900, int month = 1, int day = 1)
5      {
6          _year = year;
7          _month = month;
8          _day = day;
9      }
10 private:
11     int _year;
12     int _month;
13     int _day;
14 };
15
16 int main()
17 {
18     Date d1;

```

```

19 // 这里d2调用的默认拷贝构造完成拷贝，d2和d1的值也是一样的。
20 Date d2(d1);
21
22 return 0;
23 }

```

4. 那么编译器生成的默认拷贝构造函数已经可以完成字节序的值拷贝了，我们还需要自己实现吗？当然像日期类这样的类是没必要的。那么下面的类呢？验证一下试试？

```

1 // 这里会发现下面的程序会崩溃掉？这里就需要我们以后讲的深拷贝去解决。
2 class String
3 {
4 public:
5     String(const char* str = "jack")
6     {
7         _str = (char*)malloc(strlen(str) + 1);
8         strcpy(_str, str);
9     }
10
11     ~String()
12     {
13         cout << "~String()" << endl;
14         free(_str);
15     }
16 private:
17     char* _str;
18 };
19
20 int main()
21 {
22     String s1("hello");
23     String s2(s1);
24 }

```

## 5. 赋值运算符重载

### 5.1 运算符重载

C++为了增强代码的可读性引入了运算符重载，运算符重载是具有特殊函数名的函数，也具有其返回值类型，函数名字以及参数列表，其返回值类型与参数列表与普通的函数类似。

函数名字为：关键字operator后面接需要重载的运算符符号。

函数原型：返回值类型 operator操作符(参数列表)

注意：

- 不能通过连接其他符号来创建新的操作符：比如operator@
- 重载操作符必须有一个类类型或者枚举类型的操作数

- 用于内置类型的操作符，其含义不能改变，例如：内置的整型+，不能改变其含义
- 作为类成员的重载函数时，其形参看起来比操作数数目少1成员函数的操作符有一个默认的形参this，限定为第一个形参
- .\*、::、sizeof、?:、. 注意以上5个运算符不能重载。这个经常在笔试选择题中出现。

```

1 // 全局的operator==
2 class Date
3 {
4 public:
5     Date(int year = 1900, int month = 1, int day = 1)
6     {
7         _year = year;
8         _month = month;
9         _day = day;
10    }
11 //private:
12     int _year;
13     int _month;
14     int _day;
15 };
16
17 // 这里会发现运算符重载成全局的就需要成员变量是共有的，那么问题来了，封装性如何保证？
18 // 这里其实可以用我们后面学习的友元解决，或者干脆重载成成员函数。
19 bool operator==(const Date& d1, const Date& d2)
20 {
21     return d1._year == d2._year;
22         && d1._month == d2._month
23         && d1._day == d2._day;
24 }
25
26 void Test ()
27 {
28     Date d1(2018, 9, 26);
29     Date d2(2018, 9, 27);
30     cout<<(d1 == d2)<<endl;
31 }

```

```

1 class Date
2 {
3 public:
4     Date(int year = 1900, int month = 1, int day = 1)
5     {
6         _year = year;
7         _month = month;
8         _day = day;
9     }
10
11     // bool operator==(Date* this, const Date& d2)
12     // 这里需要注意的是，左操作数是this指向的调用函数的对象
13     bool operator==(const Date& d2)
14     {

```

```

15         return _year == d2._year;
16         && _month == d2._month
17         && _day == d2._day;
18     }
19 private:
20     int _year;
21     int _month;
22     int _day;
23 };
24
25 void Test ()
26 {
27     Date d1(2018, 9, 26);
28     Date d2(2018, 9, 27);
29     cout<<(d1 == d2)<<endl;
30 }

```

## 5.2 赋值运算符重载

```

1  class Date
2  {
3  public :
4      Date(int year = 1900, int month = 1, int day = 1)
5      {
6          _year = year;
7          _month = month;
8          _day = day;
9      }
10
11     Date (const Date& d)
12     {
13         _year = d._year;
14         _month = d._month;
15         _day = d._day;
16     }
17
18     Date& operator=(const Date& d)
19     {
20         if(this != &d)
21         {
22             _year = d._year;
23             _month = d._month;
24             _day = d._day;
25         }
26     }
27 private:
28     int _year ;
29     int _month ;
30     int _day ;
31 };

```

赋值运算符主要有四点：

1. 参数类型
2. 返回值
3. 检测是否自己给自己赋值
4. 返回\*this
5. 一个类如果没有显式定义赋值运算符重载，编译器也会生成一个，完成对象按字节序的值拷贝。

```
1  class Date
2  {
3  public:
4      Date(int year = 1900, int month = 1, int day = 1)
5      {
6          _year = year;
7          _month = month;
8          _day = day;
9      }
10 private:
11     int _year;
12     int _month;
13     int _day;
14 };
15
16 int main()
17 {
18     Date d1;
19     Date d2(2018, 10, 1);
20
21     // 这里d1调用的编译器生成operator=完成拷贝，d2和d1的值也是一样的。
22     d1 = d2;
23
24     return 0;
25 }
```

那么编译器生成的默认赋值重载函数已经可以完成字节序的值拷贝了，我们还需要自己实现吗？当然像日期类这样的类是没必要的。那么下面的类呢？验证一下试试？

```
1  // 这里会发现下面的程序会崩溃掉？这里就需要我们以后讲的深拷贝去解决。
2  class String
3  {
4  public:
5      String(const char* str = "")
6      {
7          _str = (char*)malloc(strlen(str) + 1);
8          strcpy(_str, str);
9      }
10
11     ~String()
12     {
13         cout << "~String()" << endl;
14         free(_str);
15     }
```

```

16 private:
17     char* _str;
18 };
19
20 int main()
21 {
22     String s1("hello");
23     String s2("world");
24
25     s1 = s2;
26 }

```

## 6.日期类的实现

```

1  class Date
2  {
3  public:
4      // 获取某年某月的天数
5      int GetMonthDay(int year, int month)
6      {
7          static int days[13] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
8          int day = days[month];
9          if (month == 2
10             &&((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)))
11             {
12                 day += 1;
13             }
14             return day;
15         }
16
17         // 全缺省的构造函数
18         Date(int year = 1900, int month = 1, int day = 1);
19
20         // 拷贝构造函数
21         // d2(d1)
22         Date(const Date& d);
23
24         // 赋值运算符重载
25         // d2 = d3 -> d2.operator=(d2, d3)
26         Date& operator=(const Date& d);
27
28         // 析构函数
29         ~Date();
30
31         // 日期+=天数
32         Date& operator+=(int day);
33
34         // 日期+天数
35         Date operator+(int day);
36
37         // 日期-天数
38         Date operator-(int day);

```

```

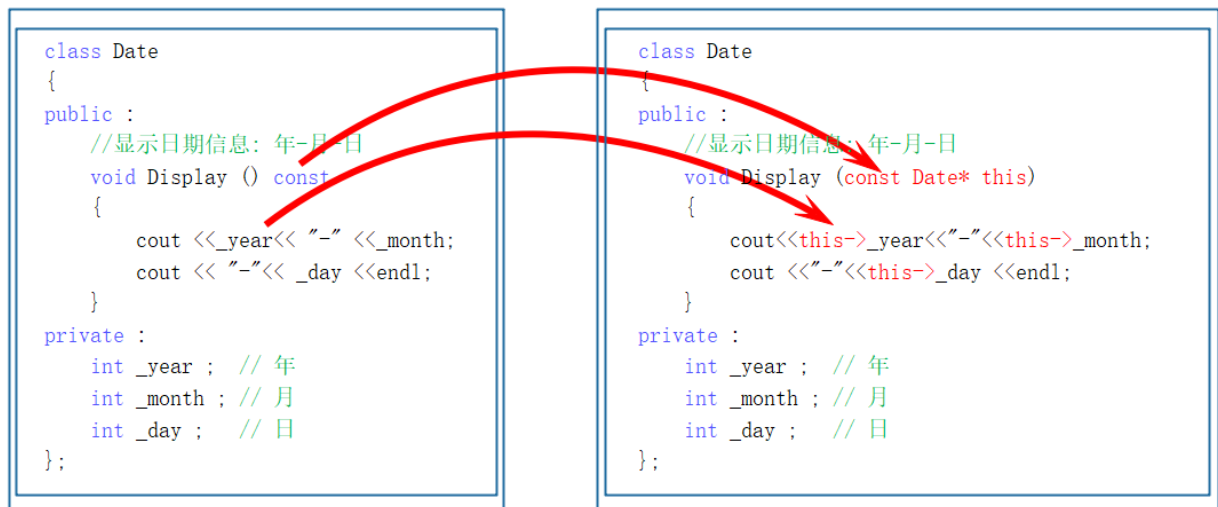
39
40     // 日期-=天数
41     Date& operator-=(int day);
42
43     // 前置++
44     Date& operator++();
45
46     // 后置++
47     Date operator++(int);
48
49     // 后置--
50     Date operator--(int);
51
52     // 前置--
53     Date& operator--();
54
55     // >运算符重载
56     bool operator>(const Date& d);
57
58     // ==运算符重载
59     bool operator==(const Date& d);
60
61     // >=运算符重载
62     inline bool operator >= (const Date& d);
63
64     // <运算符重载
65     bool operator < (const Date& d);
66
67     // <=运算符重载
68     bool operator <= (const Date& d);
69
70     // !=运算符重载
71     bool operator != (const Date& d);
72
73     // 日期-日期 返回天数
74     int operator-(const Date& d);
75 private:
76     int _year;
77     int _month;
78     int _day;
79 };

```

## 7.const成员

### 7.1 const修饰类的成员函数

将const修饰的类成员函数称之为const成员函数，const修饰类成员函数，实际修饰该成员函数隐含的this指针，表明在该成员函数中不能对类的任何成员进行修改。



编译器对const成员函数的处理

我们来看看下面的代码

```
1  class Date
2  {
3  public :
4      void Display ()
5      {
6          cout<<"Display ()" <<endl;
7          cout<<"year:" <<_year<< endl;
8          cout<<"month:" <<_month<< endl;
9          cout<<"day:" <<_day<< endl<<endl ;
10     }
11     void Display () const
12     {
13         cout<<"Display () const" <<endl;
14         cout<<"year:" <<_year<< endl;
15         cout<<"month:" <<_month<< endl;
16         cout<<"day:" <<_day<< endl<<endl;
17     }
18 private :
19     int _year ; // 年
20     int _month ; // 月
21     int _day ; // 日
22 };
23 void Test ()
24 {
25     Date d1 ;
26     d1.Display ();
27
28     const Date d2;
29     d2.Display ();
30 }
```

请思考下面的几个问题:



1. const对象可以调用非const成员函数吗?
2. 非const对象可以调用const成员函数吗?
3. const成员函数内可以调用其它的非const成员函数吗?
4. 非const成员函数内可以调用其它的const成员函数吗?

## 8.取地址及const取地址操作符重载

这两个默认成员函数一般不用重新定义，编译器默认会生成。

```
1  class Date
2  {
3  public :
4      Date* operator&()
5      {
6          return this ;
7      }
8
9      const Date* operator&()const
10     {
11         return this ;
12     }
13 private :
14     int _year ; // 年
15     int _month ; // 月
16     int _day ; // 日
17 };
```

这两个运算符一般不需要重载，使用编译器生成的默认取地址的重载即可，只有特殊情况，才需要重载，比如想让别人获取到指定的内容！