

1 对象模型

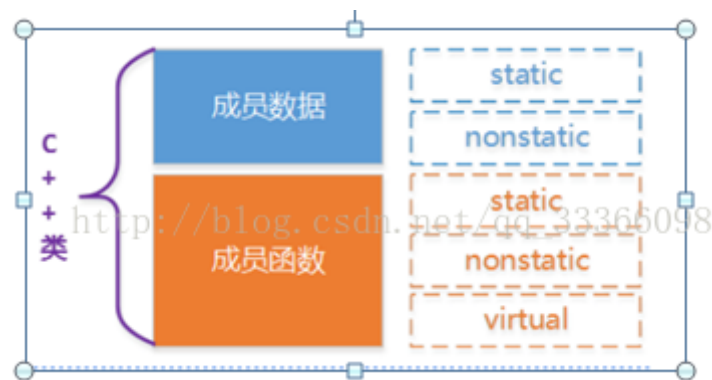
C++对象模型可以概括为以下2部分：

语言中直接支持面向对象程序设计的部分，主要涉及如构造函数、析构函数、虚函数、继承（单继承、多继承、虚继承）、多态等等。

对于各种支持的底层实现机制。

在c语言中，“数据”和“处理数据的操作（函数）”是分开来声明的，也就是说，语言本身并没有支持“数据和函数”之间的关联性。在c++中，通过抽象数据类型（abstractdata type, ADT），在类中定义数据和函数，来实现数据和函数直接的绑定。

概括来说，在C++类中有两种成员数据：static、nonstatic；三种成员函数：static、nonstatic、virtual。



1.1 基础知识

```
struct Person1{
    int a = 10;
    char b = 'b';
};
class Person{
private:
    int a = 10;
};
using namespace std;
int main() {
    Person person;
    cout << "size = " << sizeof(person) << endl;//4个字节
    Person1 person1;
    cout << "size = " << sizeof(person1) << endl;//8个字节
}
```

通过上面的案例，我们可以的得出：

1.2 总结

C++类对象中的成员变量和成员函数是分开存储的

成员变量：

普通成员变量：存储于对象中，与struct变量有相同的内存布局和字节对齐方式

静态成员变量：存储于全局数据区中

成员函数：存储于代码段中。

2 常函数

既要使数据能在一定范围内共享，又要保证它不被任意修改，可使用const。

用const修饰的声明数据成员称为常数据成员

用const修饰的声明成员函数称为常成员函数

用const修饰的定义对象称为常对象

变量或对象被const修饰后其值不能被更新。因此被const修饰的变量或对象必须要进行初始化。

(一) 用const修饰的声明数据成员称为常数据成员

有两种声明形式：

```
const int cctl;  
int const cctl;  
不能省略数据类型，可以添加 public private等访问控制符。
```

说明：

1. 任何函数都不能对常数据成员赋值。
2. 构造函数对常数据成员进行初始化时也只能通过初始化列表进行。（只有此一种方法）
3. 常数据成员在初始化时必须赋值或称其必须初始化。
4. 如果类有多个默认构造函数必须都初始化常数据成员。

通过下面例子来理解常数据成员以上4点。

这样也可以给常数据成员num赋值啊，而且可以不同的对象赋不同的值。

```

#include <iostream>
using namespace std;

class Student
{
private:
    const int num;
    int score;
public:
    Student(int a=1, int b=5):num(a){score = b;}
    void display() ;
};

void Student::display()
{
    cout << num << endl << score << endl;
    cout << "sizeof(int):" << sizeof(int) << endl;
    cout << "sizeof(Student):" << sizeof(Student) << endl;
}

int main()
{
    Student syf(5,5);
    syf.display();
    while (1);
    return 0;
}

```

```

F:\ctest\example1\De
5
5
sizeof<int>:4
sizeof<Student>:8

```

A、请指出下面程序的错误

```

class A
{
private:
    int w,h;
    const int cctlw=5; //错误一
public:
};

void main()
{
    A a; //错误二
    cout << "sss";
    system("pause");
}

```

错误一：不能对常数据成员在类中初始化、要通过类的构造函数，只有静态常量才能这样初始化。

错误二：没有合适的默认构造函数可用。因为有常量cctlw没有初始化必须初始化所有常数据成员。

更正后结果如下：

```

class A{
private:
    int w,h;
    const int cctlw;
public:
    const int cctlwcom;//常对象可以是共有私有等访问权限
    A():cctlw(5),cctlwcom(8){};//通过构造函数初始化列表初始化常成员数据。
};

```

B、多个构造函数下的常数据成员

请指出下面程序的错误：

```

class A{
private:
    int w,h;
    const int cctlwl;
public:
    const int cct;
    A():cctlwl(5),cct(6){};

```

```

    A(int x,int y) //错误一
    {
        w=x,h=y;
    }

```

```

};
void main()
{
    A a ;
    A b(3,8);
    cout<< "sss";
    system("pause");
}

```

错误一：每个构造函数都要初始化常数据成员，应改为

```

A(int x,int y):cctlwl(7),cct(8)
{
    w=x,h=y;
}

```

* (亲测，这样是可以的) *

* (二) 用const*修饰的声明声明成员函数称为常成员函数

声明：<类型标志符>函数名（参数表）const；

说明：

1. const是函数类型的一部分，在实现部分也要带该关键字。

2. const关键字可以用于对重载函数的区分。

3. 常成员函数不能更新任何数据成员，也不能调用该类中没有用const修饰的成员函数，只能调用常成员函数和常数据成员。

A、通过例子来理解const是函数类型的一部分，在实现部分也要带该关键字。

（但是我测试的下面的程序可以运行啊，下面的错误在哪？似乎正确的，可能作者改过来之后的）

```

class A{
private:
    int w,h;
public:
    int getValue() const;
    int getValue();
    A(int x,int y)
    {
        w=x,h=y;
    }
    A(){
};
int A::getValue() const //实现部分也带该关键字
{
    return w*h; // ? ? ? ?
}
void main()
{
    A const a(3,4);
    A c(2,6);
    cout<<a.getValue()<<c.getValue()<<"cctwlTest";
    system("pause");
}

```

B、通过例子来理解const关键字的重载

```

class A{
private:
    int w,h;
public:
    int getValue() const
    {
        return w*h;
    }
    int getValue(){
        return w+h;
    }
    A(int x,int y)
    {
        w=x,h=y;
    }
    A(){
};
void main()
{
    A const a(3,4);
    A c(2,6);
    cout<<a.getValue()<<c.getValue()<<"cctwlTest"; //输出12和8
    system("pause");
}

```

C、通过例子来理解常成员函数不能更新任何数据成员

```

class A{
private:
    int w,h;
public:
    int getValue() const;
    int getValue();
    A(int x,int y)
    {
        w=x,h=y;
    }
    A(){}
};

int A::getValue() const
{
    w=10,h=10;//错误, 因为常成员函数不能更新任何数据成员
    return w*h;
}

int A::getValue()
{
    w=10,h=10;//可以更新数据成员
    return w+h;
}

void main()
{
    A const a(3,4);
    A c(2,6);
    cout<<a.getValue()<<endl<<c.getValue()<<"cctwlTest";
    system("pause");
}

```

D、通过例子来理解

- 1、常成员函数可以被其他成员函数调用。
- 2、但是不能调用其他非常成员函数。
- 3、可以调用其他常成员函数。

```

class A{
private:
    int w,h;
public:
    int getValue() const
    {
        return w*h + getValue20;//错误的不能调用其他非常成员函数。
    }
    int getValue20()
    {
        return w+h+getValue0;//正确可以调用常成员函数
    }

    A(int x,int y) http://blog.csdn.net/
    {
        w=x,h=y;
    }
    A0{}
};

void main()
{
    A const a(3,4);
    A    c(2,6);
    cout<<a.getValue0<<endl<<c.getValue20<<"cctwlTest";
    system("pause");
}

```

(三) 用const修饰的定义对象称为常对象

常对象是指对象的数据成员的值在对象被调用时不能被改变。常对象必须进行初始化，且不能被更新。不能通过常对象调用普通成员函数，但是可以通过普通对象调用常成员函数。常对象只能调用常成员函数。

常对象的声明如下：

const <类名> <对象名>

<类名> const <对象名>

两种声明完全一样没有任何区别。

1、通过下面例子来理解常对象：

A、请指出下面程序的错误

```

class A{
private:
    int w,h;
public:
    int getArea() const
    {
        return w*h;
    }
    int getW(){ return w;}

    void setWH(int x,int y) {w=x,h=y;}
    A(int x,int y){w=x,h=y;}
    A(){//本例中不能省略
};
void main()
{
    A a;//普通对象可以不初始化
    a.setWH(3,9);
    A const b; //常对象必须声明的同时初始化，正确的是 A const b(3,6)。
    b.setWH(3,7);//假如上面改正后用这一句，还是错误应为b是常对象不能调用非常成员函数,切其值调用时不能改变。
    cout<< a.getArea()<<endl<< b.getArea()<<c.getArea();
    system("pause");
}

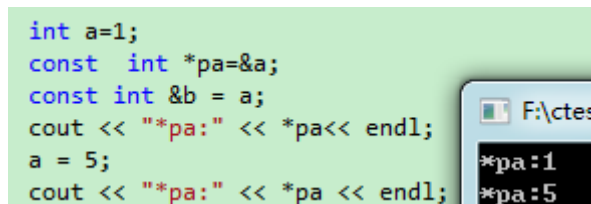
```

<http://blog.csdn.net/>

PS:

- 常成员函数可以引用const数据成员，也可以引用非const的数据成员；
- const数据成员可以被const成员函数引用，也可以被非const的成员函数引用；
- 常成员函数不能调用另一个非const成员函数。

记住定义为const后，其值不能改变即可。对于常对象、常成员函数，肯定也不能调用能改变其他值得函数。（自己添加）



```

int a=1;
const int *pa=&a;
const int &b = a;
cout << "*pa:" << *pa<< endl;
a = 5;
cout << "*pa:" << *pa << endl;

```

既要使数据能在一定范围内共享，又要保证它不被任意修改，这时可以使用const。对于const int pa，指的是不能通过改变pa的值来改变pa指向的变量的值，但可以通过改变pa指向的变量的值来改变pa的值。设置为常引用，还是可以改变实际的值。（但不可用**pa或b来改变a的值）

指向对象的常指针，其指向始终不变。

指向常变量的指针变量：const 类型名 *指针变量名；（把变量换成对象，即指向常对象的指针变量同下）

- 1、如果一个变量已被声明为常变量，只能用指向常变量的指针变量指向它，而不能用一般的（指向非const型变量的）指针变量指向它。
- 2、指向常变量的指针变量除了可以指向常变量外，还可以指向未被声明为const的变量。此时不能通过此指针变量改变该变量的值
- 3、如果函数的形参是指向非const型变量的指针，实参只能用指向非const变量的指针，而不能用指向const变量的指针

在C++面向对象程序设计中，经常用常指针和常引用作函数参数。这样既能保证数据安全，使数据不能被随意修改，在调用函数时又不必建立实参的拷贝。用常指针和常引用作函数参数，可以提高程序运行效率。

3友元函数

私有成员对于类外部的所有程序部分而言都是隐藏的，访问它们需要调用一个公共成员函数，

但有时也可能会需要创建该规则的一项例外。

友元函数是一个不属于类成员的函数，但它可以访问该类的私有成员。换句话说，友元函数被视为好像是该类的一个成员。友元函数可以是常规的独立函数，

也可以是其他类的成员。实际上，整个类都可以声明为另一个类的友元。

为了使一个函数或类成为另一个类的友元，必须由授予它访问权限的类来声明。类保留了它们的朋友的"名单"，

只有名字出现在列表中的外部函数或类才被授予访问权限。通过将关键字 friend 放置在函数的原型之前，即可将函数声明为友元。

友元函数的一般格式如下：

```
friend <return type><function name> (<parameter type list>);
```

友元分为：友元函数和友元类

友元提供了一种突破封装的方式，有时提供了便利。但是友元会增加耦合度，破坏了封装，所以友元不宜多用。

3.1 全局函数做友元函数

1. 全局函数写到类中做声明 并且最前面写关键字 friend
2. 友元函数可访问类的私有成员，但不是类的成员函数
3. 友元函数不能用const修饰 友元函数可以在类定义的任何地方声明，不受类访问限定符限制
4. 一个函数可以是多个类的友元函数
5. 友元函数的调用与普通函数的调用和原理相同

```
#include <iostream>
#include "MyArray.h"
#include "string"
using namespace std;
class Person{
//
    friend void visit(Person *person);
public:
    int a = 10;
    const int b = 12;
    mutable int c = 100;
public:
    string name = "keting";

private:
```

```

        int age =18;

};

void visit(Person *person){
    cout << "name " << person->name << endl;
    cout << "name " << person->age << endl; //访问不到
}
//常对象
int main() {
    Person *person =new Person();
    visit(person);
}

```

类作为友元类

1. 友元关系不能被继承
2. 友元关系是单向的，类A是类B的朋友，但类B不一定是类A的朋友。
3. 友元关系不具有传递性，类B是类A的朋友，类C是类B的朋友，但类C不一定是类A的朋友

```

#include
#include
using namespace std;
class Building;

class goodGay
{
public:
    goodGay();

    void visit();

private:
    Building * building;
};

class Building
{
    //让好基友类 作为Building的好朋友
    friend class goodGay;
public:
    Building();

public:
    string m_SiteingRoom;//客厅
private:
    string m_BedRoom;//卧室
};

goodGay::goodGay()
{

```

```

        building = new Building;
    }

    void goodGay::visit()
    {
        cout << "好基友正在访问: " << this->building->m_SiteingRoom << endl;
        cout << "好基友正在访问: " << this->building->m_BedRoom<< endl;
    }

    Building::Building()
    {
        this->m_SiteingRoom = "客厅";
        this->m_BedRoom = "卧室";
    }

    void test01()
    {
        goodGay gg;
        gg.visit();
    }

    int main()
    {
        test01();
        system("pause");
        return 0;
    }

```

成员函数作为友元函数

```

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

//只让visit可以作为Buildingde的好朋友，visit2不可以访问私有属性
class Building;
class goodGay
{
public:
    goodGay();

    void visit();

    void visit2();

private:
    Building * building;
};

class Building
{
    //让成员函数visit作为友元函数

```

```

        friend void goodGay:: visit();
public:
    Building();

public:
    string m_SiteingRoom;//客厅
private:
    string m_BedRoom;//卧室
};

goodGay::goodGay()
{
    building = new Building;
}

void goodGay::visit()
{
    cout << "好基友正在访问: " << this->building->m_SiteingRoom << endl;
    cout << "好基友正在访问: " << this->building->m_BedRoom<< endl;
}

void goodGay::visit2()
{
    cout << "好基友正在访问: " << this->building->m_SiteingRoom << endl;
    //cout << "好基友正在访问: " << this->building->m_BedRoom << endl;
}

Building::Building()
{
    this->m_SiteingRoom = "客厅";
    this->m_BedRoom = "卧室";
}

void test01()
{
    goodGay gg;
    gg.visit();
    gg.visit2();
}

int main()
{
    test01();
    system("pause");
    return 0;
}

```