程序设计与算法（三）

C++面向对象程序设计

郭 炜

微信公众号

微博：http://weibo.com/guoweiofpku

学会程序和算法，走遍天下都不怕!

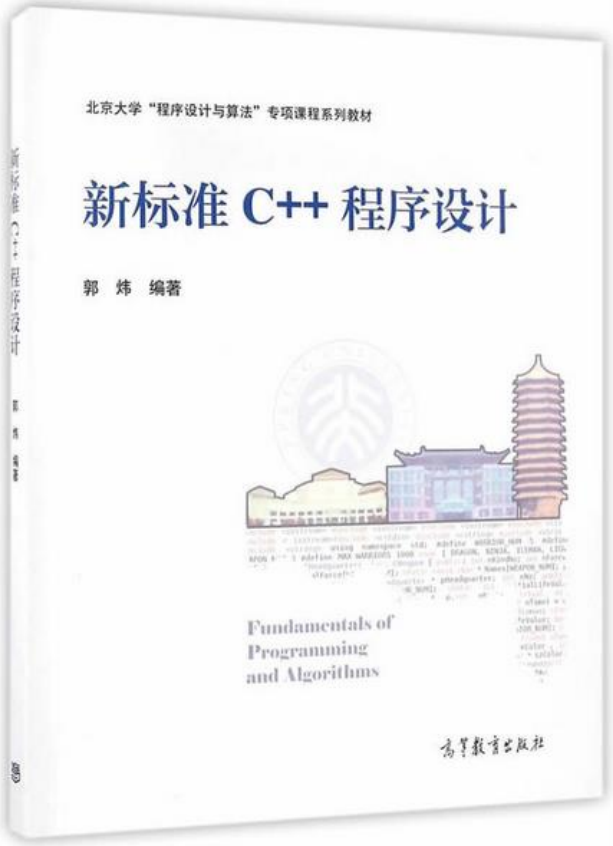
讲义照片均为郭炜拍摄

配套教材：

高等教育出版社

《新标准C++程序设计》

郭炜 编著



this指针

河南云台山

C++程序到C程序的翻译

```
class CCar  {
  public:
            int price;
            void SetPrice(int p);
};
void CCar::SetPrice(int p)
 {   price = p;  }
int main()
{
            CCar car;
            car.SetPrice(20000);
            return 0;
}
```

C++程序到C程序的翻译

```
class CCar  {
  public:
            int price;
            void SetPrice(int p);
};
void CCar::SetPrice(int p)
 {   price = p;  }
int main()
{
            CCar car;
            car.SetPrice(20000);
            return 0;
}
```

C++程序到C程序的翻译

```
class CCar  {
  public:
          int price;
          void SetPrice(int p);
};
void CCar::SetPrice(int p)
 {   price = p;  }
int main()
{
          CCar car;
          car.SetPrice(20000);
          return 0;
}
```

```
struct CCar  {
          int price;
};
```

C++程序到C程序的翻译

```
class CCar  {
  public:
        int price;
        void SetPrice(int p);
};
void CCar::SetPrice(int p)
 {   price = p;  }
int main()
{
        CCar car;
        car.SetPrice(20000);
        return 0;
}
```

```
struct CCar  {
        int price;
};
void SetPrice(struct CCar * this,
        int p)
{    this->price = p;    }
```

C++程序到C程序的翻译

```
class CCar  {
  public:
          int price;
          void SetPrice(int p);
};
void CCar::SetPrice(int p)
 {   price = p;  }
int main()
{
          CCar car;
          car.SetPrice(20000);
          return 0;
}
```

```
struct CCar  {
          int price;
};
void SetPrice(struct CCar * this,
          int p)
{    this->price = p;    }
int main()  {
          struct CCar car;
          SetPrice( & car,
          20000);
          return 0;
}
```

this指针作用

- 其作用就是指向成员函数所作用的对象

this指针作用

➢非静态成员函数中可以直接使用this来代表指向该函数作用的对象的指针。

```
class Complex {
  public:
    double real, imag;
    void Print() {  cout << real << "," << imag ;  }
    Complex(double r,double i):real(r),imag(i)
    {        }
    Complex AddOne()        {
        this->real ++;  //等价于 real ++;
        this->Print();    //等价于 Print
        return * this;
    }
};
```

```
int main()  {
    Complex c1(1,1),c2(0,0);
    c2 = c1.AddOne();
    return 0;
}
```

this指针作用

➢非静态成员函数中可以直接使用this来代表指向该函数作用的对象的指针。

```
class Complex {
  public:
    double real, imag;
    void Print() {  cout << real << "," << imag ;  }
    Complex(double r,double i):real(r),imag(i)
    {        }
    Complex AddOne()        {
        this->real ++;  //等价于 real ++;
        this->Print();    //等价于 Print
        return * this;
    }
};
```

```
int main()  {
    Complex c1(1,1),c2(0,0);
    c2 = c1.AddOne();
    return 0;
}
```

this指针作用

➢非静态成员函数中可以直接使用this来代表指向该函数作用的对象的指针。

```
class Complex {
  public:
    double real, imag;
    void Print() {  cout << real << "," << imag ;  }
    Complex(double r,double i):real(r),imag(i)
    {        }
    Complex AddOne()        {
        this->real ++;  //等价于 real ++;
        this->Print();    //等价于 Print
        return * this;
    }
};
```

```
int main()  {
    Complex c1(1,1),c2(0,0);
    c2 = c1.AddOne();
    return 0;
} //输出 2,1
```

this指针作用

```
class A
{
        int i;
  public:
        void Hello() { cout << "hello" << endl; }
};
int main()
{
        A * p = NULL;
        p->Hello();  //结果会怎样？
}
```

this指针作用

```
class A
{
        int i;
  public:
        void Hello() { cout << "hello" << endl; }
};
int main()
{
        A * p = NULL;
        p->Hello();
} // 输出：hello
```

this指针作用

```
class A
{
        int i;
  public:
        void Hello() { cout << "hello" << endl; }
};   → void Hello(A * this ) { cout << "hello" << endl; }

int main()
{
        A * p = NULL;
        p->Hello();
} // 输出：hello
```

this指针作用

```
class A
{
        int i;
  public:
        void Hello() { cout << "hello" << endl; }
};   → void Hello(A * this ) { cout << "hello" << endl; }

int main()
{
        A * p = NULL;
        p->Hello(); → Hello(p);
} // 输出：hello
```

this指针作用

```
class A
{
        int i;
  public:
        void Hello() { cout << i << "hello" << endl; }
};

int main()
{
        A * p = NULL;
        p->Hello(); → Hello(p);
} // 输出：hello
```

this指针作用

```
class A
{
        int i;
  public:
        void Hello() { cout << i << "hello" << endl; }
}; → void Hello(A * this ) { cout << this->i << "hello"
        << endl; }

int main()
{
        A * p = NULL;
        p->Hello(); → Hello(p);
} // 输出：hello
```

this指针作用

```
class A
{
        int i;
  public:
        void Hello() { cout << i << "hello" << endl; }
}; → void Hello(A * this ) { cout << this->i << "hello"
        << endl; }
 //this若为NULL，则出错！！
int main()
{
        A * p = NULL;
        p->Hello(); → Hello(p);
} // 输出：hello
```

this指针和静态成员函数

静态成员函数中不能使用 this 指针！

因为静态成员函数并不具体作用与某个对象！

因此，静态成员函数的真实的参数的个数，就是程序中写出的参数个数！

以下说法不正确的是：

A) 静态成员函数中不能使用this指针

B) this指针就是指向成员函数所作用的对象的指针

C) 每个对象的空间中都存放着一个this指针

D) 类的非静态成员函数，真实的参数比所写的参数多1

以下说法不正确的是：

A) 静态成员函数中不能使用this指针

B) this指针就是指向成员函数所作用的对象的指针

C) 每个对象的空间中都存放着一个this指针

D) 类的非静态成员函数，真实的参数比所写的参数多1

答案：C

静态成员

瑞士马特洪峰

基本概念(教材P191)

- 静态成员：在定义前面加了static关键字的成员。

基本概念

- 静态成员：在说明前面加了static关键字的成员。

```
class CRectangle
{
    private:
       int w, h;
         static int nTotalArea;    //静态成员变量
         static int nTotalNumber;
    public:
          CRectangle(int w_,int h_);
        ~CRectangle();
          static void PrintTotal();    //静态成员函数
};
```

基本概念

- 普通成员变量每个对象有各自的一份，而静态成员变量一共就一份，为所有对象共享。

基本概念

- 普通成员变量每个对象有各自的一份，而静态成员变量一共就一份，为所有对象共享。

sizeof 运算符不会计算静态成员变量。

```
class CMyclass  {
    int n;
    static int s;
};
```

基本概念

- 普通成员变量每个对象有各自的一份，而静态成员变量一共就一份，为所有对象共享。

sizeof 运算符不会计算静态成员变量。

```
class CMyclass  {
    int n;
    static int s;
};
```

则　sizeof(CMyclass) 等于 4

基本概念

- 普通成员变量每个对象有各自的一份，而静态成员变量一共就一份，为所有对象共享。
- 普通成员函数必须具体作用于某个对象，而静态成员函数并不具体作用于某个对象。

基本概念

- 普通成员变量每个对象有各自的一份，而静态成员变量一共就一份，为所有对象共享。
- 普通成员函数必须具体作用于某个对象，而静态成员函数并不具体作用于某个对象。
- 因此静态成员不需要通过对象就能访问。

如何访问静态成员

1） 类名::成员名

CRectangle::PrintTotal();

如何访问静态成员

1） 类名::成员名

```
CRectangle::PrintTotal();
```

2） 对象名.成员名

```
CRectangle r; r.PrintTotal();
```

如何访问静态成员

1） 类名::成员名

CRectangle::PrintTotal();

2） 对象名.成员名

CRectangle r; r.PrintTotal();

3） 指针->成员名

CRectangle * p = &r; p->PrintTotal();

如何访问静态成员

1） 类名::成员名

CRectangle::PrintTotal();

2） 对象名.成员名

CRectangle r; r.PrintTotal();

3） 指针->成员名

CRectangle * p = &r; p->PrintTotal();

4） 引用.成员名

CRectangle & ref = r; int n = ref.nTotalNumber;

基本概念

- 静态成员变量本质上是全局变量，哪怕一个对象都不存在，类的静态成员变量也存在。

基本概念

- 静态成员变量本质上是全局变量，哪怕一个对象都不存在，类的静态成员变量也存在。
- 静态成员函数本质上是全局函数。

基本概念

- 静态成员变量本质上是全局变量，哪怕一个对象都不存在，类的静态成员变量也存在。
- 静态成员函数本质上是全局函数。
- 设置静态成员这种机制的目的是将和某些类紧密相关的全局变量和函数写到类里面，看上去像一个整体，易于维护和理解。

静态成员示例

考虑一个需要随时知道矩形总数和总面积的图形处理程序

可以用全局变量来记录总数和总面积

用静态成员将这两个变量封装进类中，就更容易理解和维护

```
class CRectangle
{
    private:
            int w, h;
            static int nTotalArea;
            static int nTotalNumber;
    public:
        CRectangle(int w_,int h_);
                ~CRectangle();
        static void PrintTotal();
};
```

```
CRectangle::CRectangle(int w_,int h_)
{
    w = w_;
    h = h_;
    nTotalNumber ++;
    nTotalArea += w * h;
}
CRectangle::~CRectangle()
{
    nTotalNumber --;
    nTotalArea -= w * h;
}
void CRectangle::PrintTotal()
{
    cout << nTotalNumber << "," <<   nTotalArea << endl;
}
```

```
int CRectangle::nTotalNumber = 0;
int CRectangle::nTotalArea = 0;
// 必须在定义类的文件中对静态成员变量进行一次说明
//或初始化。否则编译能通过，链接不能通过。
int main()
{
        CRectangle r1(3,3), r2(2,2);
        //cout << CRectangle::nTotalNumber; // Wrong , 私有
        CRectangle::PrintTotal();
        r1.PrintTotal();
        return 0;
}
```

```
int CRectangle::nTotalNumber = 0;
int CRectangle::nTotalArea = 0;
// 必须在定义类的文件中对静态成员变量进行一次说明
//或初始化。否则编译能通过，链接不能通过。
int main()
{
        CRectangle r1(3,3), r2(2,2);
        //cout << CRectangle::nTotalNumber; // Wrong , 私有
        CRectangle::PrintTotal();
        r1.PrintTotal();
        return 0;
}
```

输出结果:

2,13

2,13

注意事项

- 在静态成员函数中，不能访问非静态成员变量，也不能调用非静态成员函数。

```
void CRectangle::PrintTotal()
{
        cout << w << "," << nTotalNumber << "," <<
        nTotalArea << endl;  //wrong
}
```

注意事项

➢ 在静态成员函数中，不能访问非静态成员变量，也不能调用非静态成员函数。

```
void CRectangle::PrintTotal()
{
        cout << w << "," << nTotalNumber << "," <<
        nTotalArea << endl;   //wrong
}
```

CRetangle::PrintTotal(); //解释不通，w 到底是属于那个对象的？

```
CRectangle::CRectangle(int w_,int h_)
{
    w = w_;
    h = h_;
    nTotalNumber ++;
    nTotalArea += w * h;
}
CRectangle::~CRectangle()
{
    nTotalNumber --;
    nTotalArea -= w * h;
}
void CRectangle::PrintTotal()
{
    cout << nTotalNumber << "," <<   nTotalArea << endl;
}
```

此CRectangle类写法，有何缺陷？

➢在使用CRectangle类时，有时会调用复制构造函数生成临时的隐藏的CRectangle对象

➢在使用CRectangle类时，有时会调用复制构造函数生成临时的隐藏的CRectangle对象

调用一个以CRectangle类对象作为参数的函数时，

调用一个以CRectangle类对象作为返回值的函数时

➢在使用CRectangle类时，有时会调用复制构造函数生成临时的隐藏的CRectangle对象

调用一个以CRectangle类对象作为参数的函数时，
调用一个以CRectangle类对象作为返回值的函数时

➢ 临时对象在消亡时会调用析构函数，减少nTotalNumber 和nTotalArea的值，可是这些临时对象在生成时却没有增加nTotalNumber 和 nTotalArea的值。

- 解决办法：为CRectangle类写一个复制构造函数。

➢ 解决办法：为CRectangle类写一个复制构造函数。

```
CRectangle :: CRectangle(CRectangle & r )
{
        w = r.w;    h = r.h;
        nTotalNumber ++;
        nTotalArea += w * h;
}
```

下面说法哪个不正确？

A) 静态成员函数内部不能访问同类的非静态成员变量，也不能调用同类的非静态成员函数

B) 非静态成员函数不能访问静态成员变量

C) 静态成员变量被所有对象所共享

D) 在没有任何对象存在的情况下，也可以访问类的静态成员

下面说法哪个不正确？

A) 静态成员函数内部不能访问同类的非静态成员变量，也不能调用同类的非静态成员函数

B) 非静态成员函数不能访问静态成员变量

C) 静态成员变量被所有对象所共享

D) 在没有任何对象存在的情况下，也可以访问类的静态成员

答案：B

成员对象和封闭类

瑞士阿尔卑斯山少女峰

成员对象和封闭类（P196)

➢有成员对象的类叫 封闭（enclosing)类。

```
class CTyre //轮胎类
{
        private:
             int radius;   //半径
             int width;     //宽度
        public:
       CTyre(int r,int w):radius(r),width(w)  {    }
};
class CEngine //引擎类
{
};
```

```
class CCar {  //汽车类
        private:
                int price;  //价格
                CTyre tyre;
                CEngine engine;
        public:
                CCar(int p,int tr,int tw );
};
CCar::CCar(int p,int tr,int w):price(p),tyre(tr, w)
{
};
int main()
{
        CCar car(20000,17,225);
        return 0;
}
```

上例中，如果 CCar类不定义构造函数， 则下面的语句会编译出错：

```
CCar car;
```

因为编译器不明白 car.tyre该如何初始化。car.engine 的初始化没问题，用默认构造函数即可。

任何生成封闭类对象的语句，都要让编译器明白，对象中的成员对象，是如何初始化的。

具体的做法就是：通过封闭类的构造函数的初始化列表。

成员对象初始化列表中的参数可以是任意复杂的表达式，可以包括函数，变量，只要表达式中的函数或变量有定义就行。

封闭类构造函数和析构函数的执行顺序

●封闭类对象生成时，先执行所有对象成员的构造函数，然后才执行封闭类的构造函数。

●对象成员的构造函数调用次序和对象成员在类中的说明次序一致，与它们在成员初始化列表中出现的次序无关。

●当封闭类的对象消亡时，先执行封闭类的析构函数，然后再执行成员对象的析构函数。次序和构造函数的调用次序相反。

封闭类实例

```
class CTyre {
     public:
        CTyre() {  cout << "CTyre contructor" << endl; }
        ~CTyre() { cout << "CTyre destructor" << endl; }
};
class CEngine {
     public:
        CEngine() { cout << "CEngine contructor" << endl; }
        ~CEngine() { cout << "CEngine destructor" << endl; }
};
class CCar {
        private:
                CEngine engine;
                CTyre tyre;
        public:
        CCar( ) { cout << "CCar contructor" << endl; }
        ~CCar() { cout << "CCar destructor" << endl; }
};
```

封闭类实例

```
int main(){
        CCar car;
        return 0;
}
```

输出结果：

```
CEngine contructor
CTyre contructor
CCar contructor
CCar destructor
CTyre destructor
CEngine destructor
```

封闭类的复制构造函数(P198)

封闭类的对象，如果是用默认复制构造函数初始化的，那么它里面包含的成员对象，也会用复制构造函数初始化。

```
class A
{
public:
    A() { cout << "default" << endl; }
    A(A & a) { cout << "copy" << endl;}
};
class B { A a; };
int main()
{
       B b1,b2(b1);
       return 0;
}
```

输出：

default
Copy

说明b2.a是用类A的复制构造函数初始化的。而且调用复制构造函数时的实参就是b1.a。

以下说法正确的是：

A) 成员对象都是用无参构造函数初始化的

B) 封闭类中成员对象的构造函数先于封闭类的构造函数被调用

C) 封闭类中成员对象的析构函数先于封闭类的析构函数被调用

D) 若封闭类有多个成员对象，则它们的初始化顺序取决于封闭类构造函数中的成员初始化列表

以下说法正确的是：

A) 成员对象都是用无参构造函数初始化的

B) 封闭类中成员对象的构造函数先于封闭类的构造函数被调用

C) 封闭类中成员对象的析构函数先于封闭类的析构函数被调用

D) 若封闭类有多个成员对象，则它们的初始化顺序取决于封闭类构造函数中的成员初始化列表

答案：B

友元
（friends)

美国马蹄湾

友元（friends,P199）

友元分为友元函数和友元类两种

1）友元函数：一个类的友元函数可以访问该类的私有成员.

```
class CCar ; //提前声明 CCar类，以便后面的CDriver类使用
class CDriver
{
        public:
           void ModifyCar( CCar * pCar) ; //改装汽车
};
class CCar
{
          private:
                  int price;
       friend int MostExpensiveCar( CCar cars[], int total); //声明友元
       friend void CDriver::ModifyCar(CCar * pCar);        //声明友元
};
```

```
void CDriver::ModifyCar( CCar * pCar)
{
        pCar->price += 1000; //汽车改装后价值增加
}
int MostExpensiveCar( CCar cars[],int total)
//求最贵汽车的价格
{
        int tmpMax = -1;
        for( int i = 0;i < total; ++i )
                if( cars[i].price > tmpMax)
                        tmpMax = cars[i].price;
        return tmpMax;
}
int main()
{
        return 0;
}
```

➢可以将一个类的成员函数(包括构造、析构函数)说明为另一个类的友元。

```
class B {
  public:
        void function();
};

class A {
      friend void B::function();
};
```

2)友元类：　如果A是B的友元类，那么A的成员函数可以访问B的私有成员。

```
class CCar
{
        private:
                int price;
        friend class CDriver; //声明CDriver为友元类
};
class CDriver
{
        public:
        CCar myCar;
           void ModifyCar()   {//改装汽车
                myCar.price += 1000;//因CDriver是CCar的友元类，
                                    //故此处可以访问其私有成员
           }
};
int main(){    return 0; }
```

✓友元类之间的关系不能传递，不能继承。

以下关于友元的说法哪个是不正确的？

A) 一个类的友元函数中可以访问该类对象的私有成员

B) 友元类关系是相互的，即若类A是类B的友元，则类B也是类A的友元

C) 在一个类中可以将另一个类的成员函数声明为友元

D) 类之间的友元关系不能传递

以下关于友元的说法哪个是不正确的？

A) 一个类的友元函数中可以访问该类对象的私有成员

B) 友元类关系是相互的，即若类A是类B的友元，则类B也是类A的友元

C) 在一个类中可以将另一个类的成员函数声明为友元

D) 类之间的友元关系不能传递

答案：B

常量成员函数

泰国普吉岛

常量成员函数

➢如果不希望某个对象的值被改变，则定义该对象的时候可以在前面加 const关键字。

```
class Sample {
        private :
                int value;
        public:
                Sample() { }
                void SetValue() {              }
};
const Sample Obj;   // 常量对象
Obj.SetValue ();    //错误 。常量对象只能使用构造函数、析构函数和 有const 说明的函数(常量方法）
```

➢在类的成员函数说明后面可以加const关键字，则该成员函数成为常量成员函数。

➢常量成员函数内部不能改变属性的值，也不能调用非常量成员函数。

```
class Sample {
        private :
                int value;
        public:
                void func() { };
                Sample() { }
                void SetValue()  const {
                        value = 0; // wrong
                        func(); //wrong
                }
};
const Sample Obj;
Obj.SetValue (); //常量对象上可以使用常量成员函数
```

➢在定义常量成员函数和声明常量成员函数时都应该使用const 关键字。

```
class Sample {
      private :
            int value;
      public:
            void PrintValue()  const;
};
void Sample::PrintValue()  const  { //此处不使用const会
                                    //导致编译出错
      cout << value;
}
void Print(const Sample & o) {
      o.PrintValue(); //若 PrintValue非const则编译错
}
```

常量成员函数

如果一个成员函数中没有调用非常量成员函数，也没有修改成员变量的值，那么，最好将其写成常量成员函数。

常量成员函数的重载

➢两个函数，名字和参数表都一样，但是一个是const，一个不是，算重载。

```
#include <iostream>
using namespace std;
class CTest {
        private :
                int n;
        public:
                CTest() { n = 1 ; }
                int GetValue()  const  { return n ; }
                int GetValue() { return 2 * n ; }
};
```

常量成员函数的重载

➢两个函数，名字和参数表都一样，但是一个是const，一个不是，算重载。

```
int main()  {
              const CTest objTest1;
              CTest objTest2;
              cout << objTest1.GetValue() << "," <<
                      objTest2.GetValue() ;
              return 0;
}
```

=>1,2

mutable成员变量

➢可以在const成员函数中修改的成员变量

```
class CTest
{
public:

    bool GetData() const
    {
        m_n1++;
        return m_b2;
    }
private:
    mutable int  m_n1;
    bool m_b2;
};
```