程序设计实习

C++ 面向对象程序设计

张勤健
zqj@pku.edu.cn

北京大学信息科学技术学院

2025 年 2 月 21 日

大纲

C++ 程序到 C 程序的翻译

```
1  class CCar {
2  public:
3    int price;
4    void SetPrice(int p);
5  };
6  void CCar::SetPrice(int p) {
7    price = p;
8  }
9  int main() {
10   CCar car;
11   car.SetPrice(20000);
12   return 0;
13 }
```

C++ 程序到 C 程序的翻译

```cpp
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;
}
```

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

C++ 程序到 C 程序的翻译

```
1  class CCar {
2  public:
3    int price;
4    void SetPrice(int p);
5  };
6  void CCar::SetPrice(int p) {
7    price = p;
8  }
9  int main() {
10   CCar car;
11   car.SetPrice(20000);
12   return 0;
13 }
```

```
1  struct CCar {
2    int price;
3  };
```

```
4  void SetPrice(struct CCar *this, int p) {
5    this->price = p;
6  }
```

C++ 程序到 C 程序的翻译

```
1  class CCar {
2  public:
3    int price;
4    void SetPrice(int p);
5  };
6  void CCar::SetPrice(int p) {
7    price = p;
8  }
9  int main() {
10   CCar car;
11   car.SetPrice(20000);
12   return 0;
13 }
```

```
1  struct CCar {
2    int price;
3  };
```

```
4  void SetPrice(struct CCar *this, int p) {
5    this->price = p;
6  }
```

```
7  int main()  {
8    struct CCar car;
9    SetPrice(&car, 20000);
10   return 0;
11 }
```

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

```
4  class A {
5    int i;
6  public:
7    void hello() {
8      cout << "hello" << endl;
9    }
10 };
11 int main() {
12   A *p = NULL;
13   p->hello();
14 }
```

能编译吗？能运行吗？结果是什么？

this 指针作用

```
4  class A {
5    int i;
6  public:
7    void hello() {
8      cout << "hello" << endl;
9    }
10 };
11 int main() {
12   A *p = NULL;
13   p->hello();
14 }
```

能编译吗？能运行吗？结果是什么？

输出：

```
hello
```

this 指针作用

```
4  class A {
5    int i;
6  public:
7    void hello() {
8      cout << "hello" << endl;
9    }
10 };
11 int main() {
12   A *p = NULL;
13   p->hello();
14 }
```

能编译吗？能运行吗？结果是什么？

输出：

```
hello
```

注：本质上上述程序还是存在 undefined behavior。编译时加`-fsanitize=undefined`检查

this 指针作用

```
1  void hello() {
2    cout << "hello" << endl;
3  }
```

->

```
1  void hello(A * this) {
2    cout << "hello" << endl;
3  }
```

```
1  p->Hello();
```

->

```
1  Hello(p);
```

this 指针作用

```
4  class A {
5    int i;
6  public:
7    void hello() {
8      cout << i << "hello" << endl;
9    }
10 };
11 int main() {
12   A *p = NULL;
13   p->hello();
14 }
```

能编译吗？能运行吗？结果是什么？

this 指针作用

```
4  class A {
5    int i;
6  public:
7    void hello() {
8      cout << i << "hello" << endl;
9    }
10 };
11 int main() {
12   A *p = NULL;
13   p->hello();
14 }
```

能编译吗？能运行吗？结果是什么？

```
1  void hello(A *this) {
2    cout << this->i << "hello" << endl;
3  }
```

this 指针作用

```
4  class A {
5    int i;
6  public:
7    void hello() {
8      cout << i << "hello" << endl;
9    }
10 };
11 int main() {
12   A *p = NULL;
13   p->hello();
14 }
```

能编译吗？能运行吗？结果是什么？

```
1  void hello(A *this) {
2    cout << this->i << "hello" << endl;
3  }
```

结果：运行错误

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。

```
4  class Complex {
5  public:
6    double real, imag;
7    void print() {
8      cout << real << "," << imag;
9    }
10   Complex(double r, double i) : real(r), imag(i) {}
11   Complex addOne() {
12     this->real++;  //等价于 real++;
13     this->print(); //等价于 print()
14     return *this; // 返回对象本身
15   }
16 };
17 int main() {
18   Complex c1(1, 1), c2(0, 0);
19   c2 = c1.addOne();
20   return 0;
21 }
```

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。

```
4  class Complex {
5  public:
6    double real, imag;
7    void print() {
8      cout << real << "," << imag;
9    }
10   Complex(double r, double i) : real(r), imag(i) {}
11   Complex addOne() {
12     this->real++;  //等价于 real++;
13     this->print(); //等价于 print()
14     return *this; // 返回对象本身
15   }
16 };
17 int main() {
18   Complex c1(1, 1), c2(0, 0);
19   c2 = c1.addOne();
20   return 0;
21 }
```

结果：输出 2,1

静态成员-基本概念

静态成员：在定义前面加了 static 关键字的成员。

```
1  class CRectangle {
2  private:
3    int w;
4    int h;
5    static int nTotalArea; //静态成员变量
6    static int nTotalNumber;
7  public:
8    CRectangle(int w_, int h_);
9    ~CRectangle();
10   static void printTotal(); //静态成员函数
11 };
```

静态成员-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。

静态成员-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。

```
1  class CMyclass {
2    int n;
3    static int s;
4  };
```

静态成员-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。

```
1  class CMyclass {
2    int n;
3    static int s;
4  };
```

sizeof(CMyclass) 等于 4

静态成员-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。

```
1  class CMyclass {
2    int n;
3    static int s;
4  };
```

sizeof(CMyclass) 等于 4

- 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。

静态成员-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。

```
1  class CMyclass {
2    int n;
3    static int s;
4  };
```

sizeof(CMyclass) 等于 4

- 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

```
1  CRectangle::printTotal();
```

如何访问静态成员

1. 类名:: 成员名

```
1 CRectangle::printTotal();
```

2. 对象名. 成员名

```
1 CRectangle r;
2 r.printTotal();
```

如何访问静态成员

1. 类名:: 成员名

```
1 CRectangle::printTotal();
```

2. 对象名. 成员名

```
1 CRectangle r;
2 r.printTotal();
```

3. 指针-> 成员名

```
1 CRectangle *p = &r;
2 p->printTotal();
```

如何访问静态成员

1. 类名:: 成员名

```
1 CRectangle::printTotal();
```

2. 对象名. 成员名

```
1 CRectangle r;
2 r.printTotal();
```

3. 指针-> 成员名

```
1 CRectangle *p = &r;
2 p->printTotal();
```

4. 引用. 成员名

```
1 CRectangle &ref = r;
2 int n = ref.nTotalNumber;
```

静态成员-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-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。

静态成员-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

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

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

静态成员示例

```
4  class CRectangle {
5    int w, h;
6    static int nTotalArea;
7    static int nTotalNumber;
8  public:
9    CRectangle(int w_, int h_);
10   ~CRectangle();
11   static void printTotal();
12 };
13 CRectangle::CRectangle(int w_, int h_) {
14   w = w_;
15   h = h_;
16   nTotalNumber++;
17   nTotalArea += w * h;
18 }
19 CRectangle::~CRectangle() {
20   nTotalNumber--;
21   nTotalArea -= w * h;
22 }
23 void CRectangle::printTotal() {
24   cout << nTotalNumber << "," << nTotalArea << endl;
25 }
```

静态成员示例

```
26 int CRectangle::nTotalNumber = 0;
27 int CRectangle::nTotalArea = 0;
28 // 必须在定义类的文件中对静态成员变量进行一次说明
29 //或初始化。否则编译能通过，链接不能通过。
30 int main() {
31   CRectangle r1(3, 3), r2(2, 2);
32   // cout << CRectangle::nTotalNumber; // Wrong , 私有
33   CRectangle::printTotal();
34   r1.printTotal();
35   return 0;
36 }
```

静态成员示例

```
26 int CRectangle::nTotalNumber = 0;
27 int CRectangle::nTotalArea = 0;
28 // 必须在定义类的文件中对静态成员变量进行一次说明
29 //或初始化。否则编译能通过，链接不能通过。
30 int main() {
31   CRectangle r1(3, 3), r2(2, 2);
32   // cout << CRectangle::nTotalNumber; // Wrong , 私有
33   CRectangle::printTotal();
34   r1.printTotal();
35   return 0;
36 }
```

输出结果：

```
2,13
2,13
```

静态成员-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。

```
1  void CRectangle::printTotal() {
2    cout << w << "," << nTotalNumber << "," << nTotalArea << endl;  //wrong
3  }
```

静态成员-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。

```
1  void CRectangle::printTotal() {
2    cout << w << "," << nTotalNumber << "," << nTotalArea << endl;  //wrong
3  }
```

- 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!

```
4  class CRectangle {
5    int w, h;
6    static int nTotalArea;
7    static int nTotalNumber;
8  public:
9    CRectangle(int w_, int h_);
10   ~CRectangle();
11   static void printTotal();
12 };
13 CRectangle::CRectangle(int w_, int h_) {
14   w = w_;
15   h = h_;
16   nTotalNumber++;
17   nTotalArea += w * h;
18 }
19 CRectangle::~CRectangle() {
20   nTotalNumber--;
21   nTotalArea -= w * h;
22 }
23 void CRectangle::printTotal() {
24   cout << nTotalNumber << "," << nTotalArea << endl;
25 }
```

此 CRectangle 类写法，有没有缺陷?

- 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
  - 当用一个对象去初始化同类的另一个对象时
  - 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
  - 当用一个对象去初始化同类的另一个对象时
  - 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。

- 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
  - 当用一个对象去初始化同类的另一个对象时
  - 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。

- 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
  - 当用一个对象去初始化同类的另一个对象时
  - 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。

```
1  CRectangle::CRectangle(const CRectangle &r) {
2    w = r.w;
3    h = r.h;
4    nTotalNumber ++;
5    nTotalArea += w * h;
6  }
```

单选题

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

单选题

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

单选题

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

单选题

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

有成员对象的类叫封闭（enclosing) 类。

成员对象和封闭类

有成员对象的类叫封闭（enclosing）类。

```
1  class CTyre { //轮胎类
2  private:
3    int radius; //半径
4    int width;  //宽度
5  public:
6    CTyre(int r, int w) : radius(r), width(w) {}
7  };
8  class CEngine {}; //引擎类
9  class CCar { //汽车类
10 private:
11   int price; //价格
12   CTyre tyre;
13   CEngine engine;
14 public:
15   CCar(int p, int tr, int tw);
16 };
17 CCar::CCar(int p, int tr, int w) : price(p), tyre(tr, w) {}
18 int main() {
19   CCar car(20000, 17, 225);
20   return 0;
21 }
```

成员对象和封闭类

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：

```
1  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。

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。

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。

成员对象和封闭类

```
4  class CTyre {
5  public:
6    CTyre() { cout << "CTyre contructor" << endl; }
7    ~CTyre() { cout << "CTyre destructor" << endl; }
8  };
9  class CEngine {
10 public:
11   CEngine() { cout << "CEngine contructor" << endl; }
12   ~CEngine() { cout << "CEngine destructor" << endl; }
13 };
14 class CCar {
15 private:
16   CEngine engine;
17   CTyre tyre;
18 public:
19   CCar() { cout << "CCar contructor" << endl; }
20   ~CCar() { cout << "CCar destructor" << endl; }
21 };
22 int main() {
23   CCar car;
24   return 0;
25 }
```

成员对象和封闭类

```
4  class CTyre {
5  public:
6    CTyre() { cout << "CTyre contructor" << endl; }
7    ~CTyre() { cout << "CTyre destructor" << endl; }
8  };
9  class CEngine {
10 public:
11   CEngine() { cout << "CEngine contructor" << endl; }
12   ~CEngine() { cout << "CEngine destructor" << endl; }
13 };
14 class CCar {
15 private:
16   CEngine engine;
17   CTyre tyre;
18 public:
19   CCar() { cout << "CCar contructor" << endl; }
20   ~CCar() { cout << "CCar destructor" << endl; }
21 };
22 int main() {
23   CCar car;
24   return 0;
25 }
```

输出结果

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

单选题

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

单选题

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

封闭类的复制构造函数

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。

封闭类的复制构造函数

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。

```
4  class A {
5  public:
6    A() { cout << "default" << endl; }
7    A(const A &a) { cout << "copy" << endl; }
8  };
9  class B {
10   A a;
11 };
12 int main() {
13   B b1, b2(b1);
14   return 0;
15 }
```

封闭类的复制构造函数

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。

```
4  class A {
5  public:
6    A() { cout << "default" << endl; }
7    A(const A &a) { cout << "copy" << endl; }
8  };
9  class B {
10   A a;
11 };
12 int main() {
13   B b1, b2(b1);
14   return 0;
15 }
```

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。

友元

友元分为友元函数和友元类两种

友元函数

友元函数: 一个类的友元函数可以访问该类的私有成员.

友元函数

友元函数: 一个类的友元函数可以访问该类的私有成员.

```
4   class CCar; //提前声明 CCar 类, 以便后面的 CDriver 类使用
5   class CDriver {
6   public:
7     void modifyCar(CCar *pCar); //改装汽车
8   };
9   class CCar {
10  private:
11    int price;
12    friend int mostExpensiveCar(CCar cars[], int total); //声明友元
13    friend void CDriver::modifyCar(CCar *pCar);          //声明友元
14    //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。
15  };
16  void CDriver::modifyCar(CCar *pCar) {
17    pCar->price += 1000; //汽车改装后价值增加
18  }
19  int mostExpensiveCar(CCar cars[], int total) { //求最贵汽车的价格
20    int tmpMax = -1;
21    for (int i = 0; i < total; ++i)
22      if (cars[i].price > tmpMax)
23        tmpMax = cars[i].price;
24    return tmpMax;
25  }
26  int main() {
27    return 0;
28  }
```

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。

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。

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

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

友元类之间的关系不能传递，不能继承

单选题

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

单选题

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

常量对象

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。

常量对象

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。

```
1  class Sample {
2  private:
3    int value;
4  public:
5    Sample() {}
6    void SetValue() {}
7  };
8  const Sample Obj; // 常量对象
9  Obj.SetValue();   //错误 。
```

常量成员函数

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。

常量成员函数

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。

```
1  class Sample {
2  private:
3    int value;
4  public:
5    void func(){};
6    Sample() {}
7    void SetValue() const {
8      value = 0; // wrong
9      func();    // wrong
10   }
11 };
12 const Sample Obj;
13 Obj.SetValue(); //常量对象上可以使用常量成员函数
```

常量成员函数

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。

```
1  class Sample {
2  private:
3    int value;
4  public:
5    void func(){};
6    Sample() {}
7    void SetValue() const {
8      value = 0; // wrong
9      func();    // wrong
10   }
11 };
12 const Sample Obj;
13 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。

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

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。

```
4  class CTest {
5    int n;
6  public:
7    CTest() { n = 1; }
8    int getValue() const {
9      return n;
10   }
11   int getValue() {
12     return 2 * n;
13   }
14 };
15 int main() {
16   const CTest objTest1;
17   CTest objTest2;
18   cout << objTest1.getValue() << "," << objTest2.getValue();
19   return 0;
20 }
```

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。

```
4  class CTest {
5    int n;
6  public:
7    CTest() { n = 1; }
8    int getValue() const {
9      return n;
10   }
11   int getValue() {
12     return 2 * n;
13   }
14 };
15 int main() {
16   const CTest objTest1;
17   CTest objTest2;
18   cout << objTest1.getValue() << "," << objTest2.getValue();
19   return 0;
20 }
```

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。

mutable 成员变量

可以在 const 成员函数中修改的成员变量

mutable 成员变量

可以在 const 成员函数中修改的成员变量

```
1  class CTest {
2  public:
3    bool getData() const {
4      m_n1++;
5      return m_b2;
6    }
7  private:
8    mutable int m_n1;
9    bool m_b2;
10 };
```