

类与对象



俞智斌

yuzhibin@ouc.edu.cn

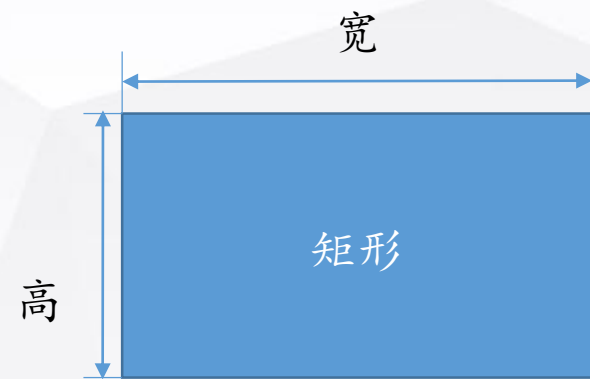
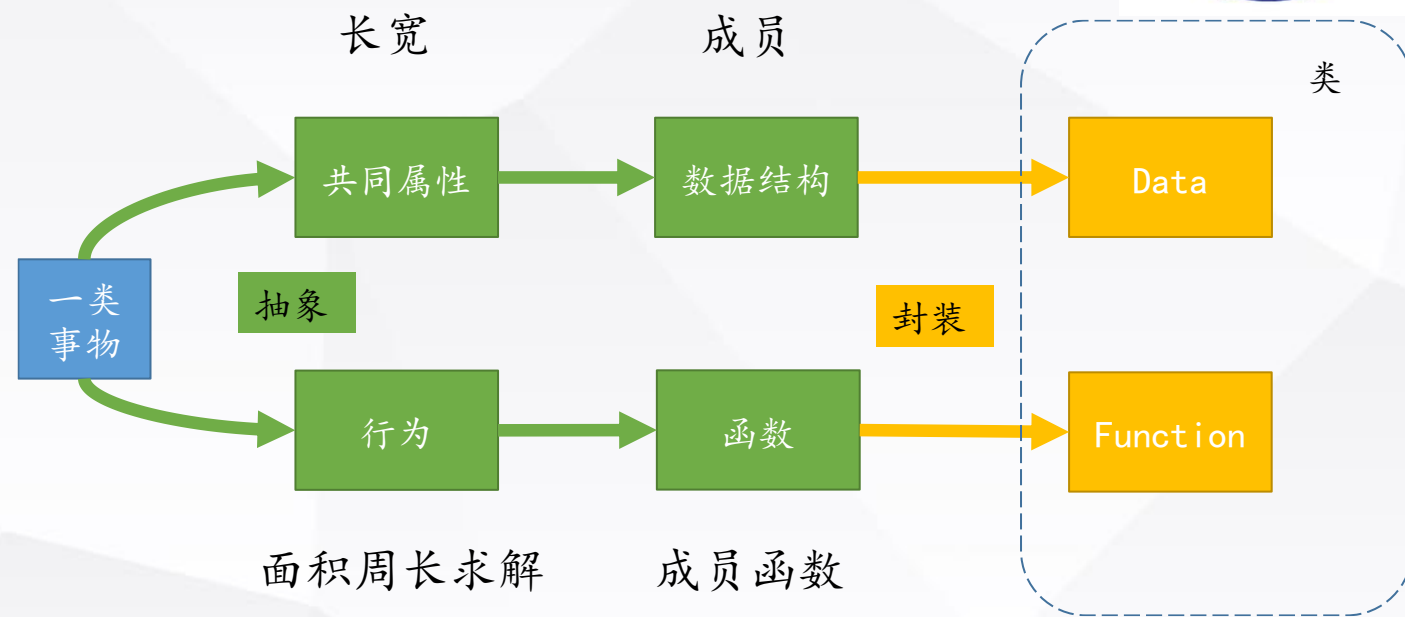
中国海洋大学电子工程系

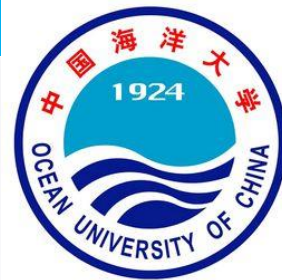


类和对象

• 上节课我们学了

- 从C到C++
- 什么是对象
 - 对象和类
- 什么是面向对象
 - 面向对象与面向过程
- 面向对象编程的特点
- 为什么要用面向对象
- 面向对象编程的设计方法





类和对象

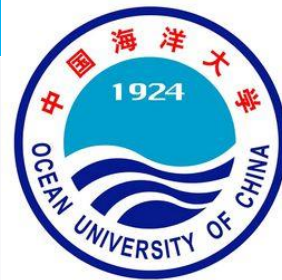
- 类成员的可访问范围
 - 三种成员属性：
 - `private`
 - 私有属性(默认属性)
 - `public`
 - 公有属性
 - `protected`
 - 保护属性（允许继承者调用）



类和对象

- 类成员的可访问范围

```
class Man
{
    int mAge; //私有成员
    char mname[20]; // 私有成员
public:
    void SetName(char * Name)
    {
        strcpy_s(mname, Name);
    }
    void SetAge(int Age)
    {
        mAge = Age;
    }
    void GetName()
    {
        printf("%s", mname);
    }
    void GetAge()
    {
        printf("%d", mAge);
    }
};
```



类和对象

• 类成员的可访问范围

- 类的成员函数 **内部**，可以访问：
 - 当前对象的 **全部** 属性函数
 - 同类其它对象的全部属性
- 类的成员函数 **以外** 的地方，可以访问：
 - 只能够访问该类对象的 **公有成员**
- 设置私有成员的目的
- 强制对成员变量的访问一定要通过成员函数进行
- 设置私有成员的机制 —— **隐藏**

```
#include<stdio.h>
#include <iostream>
int main()
{
    Man m;
    m.SetName("Sarah");
    m.SetAge(20);
    m.GetName();
    m.GetAge();
    return 0;
}
```

```
C:\WINDOWS\system32\cmd.exe
Sarah
20
请按任意键继续. . .
```



类和对象

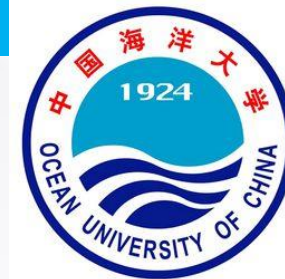
- 内联成员函数

- `inline` + 函数
- 关键字 `inline` 必须与函数定义体放在一起才能使函数成为内联, 仅将 `inline` 放在函数声明前面不起任何作用。
- 如下风格的函数 `Fun` 不能成为内联函数:

```
inline void Fun(int x, int y); // inline 仅与函数声明放在一起  
void Fun(int x, int y) {}
```

- 而如下风格的函数 `Fun` 则成为内联函数:

```
void Fun(int x, int y);  
inline void Fun(int x, int y) // inline 与函数定义体放在一起 {}
```



类和对象

• 内联成员函数

- 整个函数体出现在类定义内部

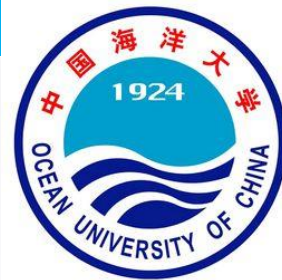
写法1
(不太规范)

```
class A
{
public:
    void Fun(int x, int y) { } // 自动地成为内联函数
}
```

写法2

```
// 头文件
class A
{
public:
    void Fun(int x, int y);
}

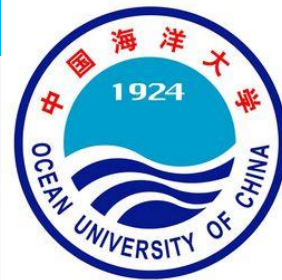
// 定义文件
inline void A::Fun(int x, int y)
{
}
```



类和对象

• 内联成员函数

- 内联函数为什么能够提高效率？
 - 内联是以代码复制为代价，仅仅省去了函数调用的开销，从而提高函数的执行效率。如果执行函数体内代码的时间，相比于函数调用的开销较大，那么效率的收获会很少。另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。
- 什么时候用内联成员函数？
 - 代码量小但频繁调用的场合
 - 没有大量循环语句的场合
- `inline` 是一种“用于实现的关键字”，而不是一种“用于声明的关键字”。

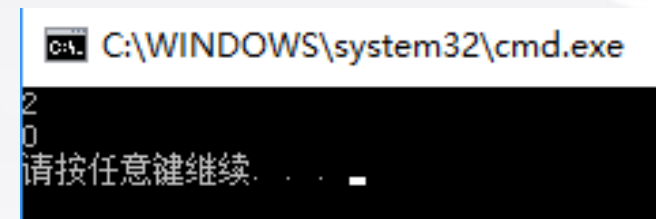


类和对象

- 重载成员函数

```
#include <iostream>
using namespace std;
class Coordinates {
public:
    int cal(int x, int y) { return x - y; }
    double cal(double x, double y) { return x + y; }
};

int main()
{
    Coordinates a;
    cout<<a.cal(1.0, 1.0)<<endl;
    cout << a.cal(1, 1) << endl;
    return 0;
}
```





类和对象

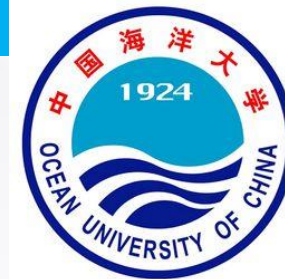
- 缺省参数

```
#include <iostream>
using namespace std;
class Coordinates {
private:
    int x, y;
public:
    void init(int x_ = 1, int y_ = 1) { x = x_, y = y_; }
    void print() { cout << x << endl << y << endl; }
};

void add(int x = 1, int y = 2)
{
    cout << x + y << endl;
}

int main()
{
    Coordinates a;
    a.init();
    a.print();
    add();
    add(5, 6);
    return 0;
}
```

```
C:\WINDOWS\system32\cmd.exe
1
1
3
11
请按任意键继续...
```



类和对象

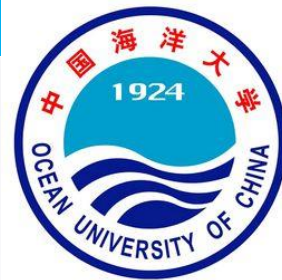
• 缺省参数

- 注意使用缺省参数时要避免发生歧义

```
#include <iostream>
using namespace std;
class Coordinates {
private:
    int x, y;
public:
    void init(int x_ = 1, int y_ = 1) { x = x_, y = y_; }
    void init() { x = 0; y = 0; }
    void print() { cout << x << endl << y << endl; }
};

int main()
{
    Coordinates a;
    a.init();
    a.print();
    return 0;
}
```

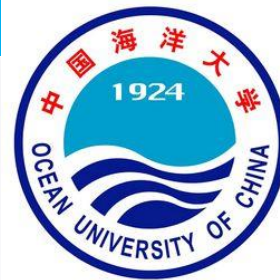
编译器不知道该运行哪个init



类和对象

• 构造函数

- 成员函数的一种名字与类名相同，可以有参数，不能有返回值 (void也不行)
- 作用是对对象进行初始化，如给成员变量赋初值
- 如果定义类时没写构造函数，则编译器生成一个默认的非参数的构造函数
- 默认构造函数无参数，不做任何操作
- 如果定义了构造函数，则编译器不生成默认的非参数的构造函数
- 对象生成时构造函数自动被调用。
- 对象一旦生成，就再也不能在其上执行构造函数
- 一个类可以有多个构造函数



类和对象

• 构造函数

- 如果定义了默认构造函数，则默认的构造函数将不复存在

```
#include <iostream>
using namespace std;
class Coordinates {
private:
    int x, y;
public:
    Coordinates(int x_, int y_) { x = x_, y = y_; }
    Coordinates() { cout << "test"; }
    void print() { cout << x << endl << y << endl; }
};

int main()
{
    Coordinates a;
    Coordinates b(2, 2);
    Coordinates* c = new Coordinates(3, 3);
    Coordinates* d = new Coordinates();
    b.print();
    c->print();
    return 0;
}
```

注意此处没有返回值

```
C:\WINDOWS\system32\cmd.exe
test
test
2
2
3
3
请按任意键继续...
```



类和对象

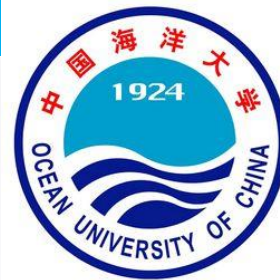
• 构造函数

- 这样也是可以的:

```
#include <iostream>
using namespace std;
class Coordinates {
private:
    int x, y;
public:
    Coordinates(int x_, int y_) { x = x_, y = y_; }
    Coordinates(Coordinates c1, Coordinates c2) { x = c1.x + c1.y; y = c2.x + c2.y; }
    void print() { cout << x << endl << y << endl; }
};
int main()
{
    Coordinates a(1,1);
    Coordinates b(2, 2);
    Coordinates* c = new Coordinates(a, b);
    a.print();
    b.print();
    c->print();
    return 0;
}
```

C:\WINDOWS\system32\cmd.exe

```
1
1
2
2
2
4
请按任意键继续.
```



类和对象

• 构造函数

- 私有构造函数虽然合法，但用起来不方便

```
#include <iostream>
using namespace std;
class Coordinates {
private:
    int x, y;
    Coordinates(int x_, int y_) { x = x_, y = y_; }
    Coordinates(Coordinates c1, Coordinates c2) { x = c1.x + c1.y; y = c2.x + c2.y; }
    void print() { cout << x << endl << y << endl; }
};

int main()
{
    Coordinates a(1,1);
    Coordinates b(2, 2);
    Coordinates* c = new Coordinates(a, b);
    a.print();
    b.print();
    c->print();
    return 0;
}
```

注意，此处没有
`public`

无法执行

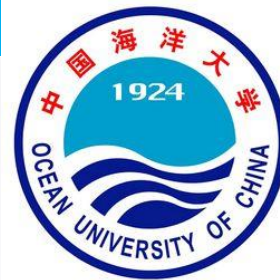


类和对象

• 复制构造函数

- 只有一个参数, 即对同类对象的引用。
- 形如 `X::X(X&)` 或 `X::X(const X &)`, 二者选一。后者能以常量对象作为参数
- 如果没有定义复制构造函数, 那么编译器生成默认复制构造函数。默认的复制构造函数完成复制功能。
- 注意默认复制构造函数一旦定义, 默认复制构造函数将不复存在
- 不允许有形如 `X::X(X)` 的构造函数。

```
int main()
{
    Coordinates a; //使用默认构造函数
    Coordinates b(a); //使用默认复制构造函数
    return 0;
}
```

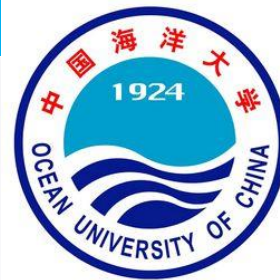



类和对象

- 复制构造函数

```
#include <iostream>
using namespace std;
class Coordinates {
private:
    int x, y;
public:
    Coordinates(int x_, int y_)
    { x = x_, y = y_; }
    Coordinates(const Coordinates & input_c)
    { x = 2, y = 2; }
    void print()
    { cout << x << endl << y << endl; }
};

int main()
{
    Coordinates a(1, 1);
    Coordinates b(a);
    a.print();
    b.print();
    return 0;
}
```



类和对象

• 复制构造函数

- 注意这样写不是赋值而是调用复制构造函数

```
#include <iostream>
using namespace std;
class Coordinates {
private:
    int x, y;
public:
    Coordinates(int x_, int y_)
    { x = x_, y = y_; }
    Coordinates(const Coordinates & input_c)
    { x = 2, y = 2; }
    void print()
    { cout << x << endl << y << endl; }
};
int main()
{
    Coordinates a(1, 1);
    Coordinates b=a;
    a.print();
    b.print();
    return 0;
}
```

```
C:\WINDOWS\system32\cmd.exe
1
1
2
2
请按任意键继续...
```

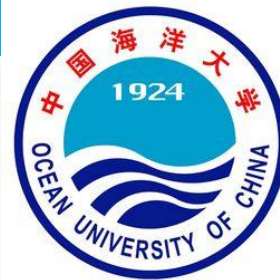


类和对象

• 复制构造函数

- 如果某函数有一个参数是类 A 的对象，那么该函数被调用时，类 A 的复制构造函数将被调用。

```
#include <iostream>
using namespace std;
class Coordinates {
private:
    int x, y;
public:
    Coordinates(int x_, int y_) { x = x_, y = y_; }
    Coordinates(const Coordinates & input_c) { x = 2, y = 2; }
    void print() { cout << x << endl << y << endl; }
};
void Test(Coordinates c)
{
    c.print();
}
int main()
{
    Coordinates a(1, 1);
    Test(a);
    return 0;
}
```



类和对象

• 复制构造函数

- 如果某函数的返回值是类A的对象时，则函数返回时，A的复制构造函数被调用：

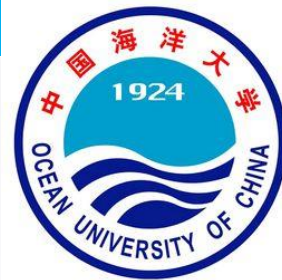
```
#include <iostream>
using namespace std;
class Coordinates {
private:
    int x, y;
public:
    Coordinates(int x_, int y_) { x = x_, y = y_; }
    Coordinates(const Coordinates & input_c) { x = 2, y = 2; }
    void print() { cout << x << endl << y << endl; }
};

Coordinates Test()
{
    Coordinates c(1, 1);
    return c;
}

int main()
{
    Coordinates a=Test();
    a.print();
    return 0;
}
```

C:\WINDOWS\system32\cmd.exe

2
2
请按任意键继续. . .



类和对象

• 类型转换构造函数

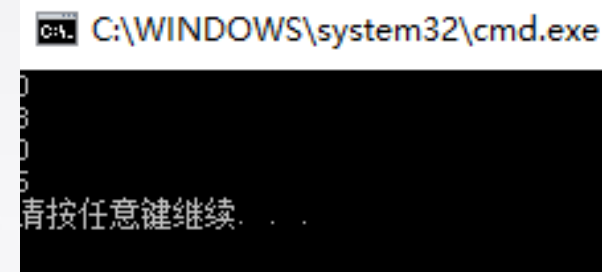
- 目的

- 实现类型的自动转换

- 特点

- 只有一个参数
 - 不是复制构造函数
 - 编译系统会自动调用类型转换构造函数 **建立一个临时对象 / 临时变量**

```
#include <iostream>
using namespace std;
class Coordinates {
private:
    int x, y;
public:
    Coordinates(int x_, int y_) { x = x_, y = y_; }
    Coordinates() {}
    Coordinates(const Coordinates & input_c) { x = 2, y = 2; }
    Coordinates(int input) { x = 0, y = input; print(); }
    void print() { cout << x << endl << y << endl; }
};
int main()
{
    Coordinates a(3);
    Coordinates b=5;
    return 0;
}
```





类和对象

• 析构函数

• 特点

- 成员函数的一种
- 名字与类名相同
- 在前面加 ~
- 没有参数和返回值
- 一个类最多只有一个析构函数
- 对象消亡时将自动被调用
- 在对象消亡前做善后工作
- 释放分配的空间等
- 定义类时没写析构函数，则编译器生成缺省析构函数，其中不涉及释放用户申请的内存释放等清理工作
- 定义了析构函数，则编译器不生成缺省析构函数



类和对象

- 析构造函数

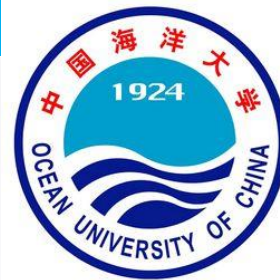
- 例子

```
#include <iostream>
using namespace std;
class Coordinates {
private:
    int x, y; char* p;
public:
    Coordinates(int x_, int y_) { x = x_, y = y_; }
    Coordinates() {}
    Coordinates(int input)
    { x = 0, y = input; p = new char[10]; }
    void print() { cout << x << endl << y << endl; }
    ~Coordinates() { cout << "goodbye!" << endl; delete[] p; }
};

int main()
{
    Coordinates a(3);
    a.print();
    return 0;
}
```

C:\WINDOWS\system32\cmd.exe

```
0
3
goodbye!
请按任意键继续. . .
```



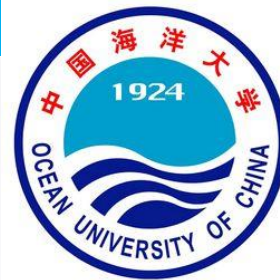
类和对象

• 析构函数

- 释放类数组时，数组中每个对象的析构函数都会被调用

```
#include <iostream>
using namespace std;
class Coordinates {
private:
    int x, y; char* p;
public:
    Coordinates(int x_, int y_) { x = x_, y = y_; }
    Coordinates() {}
    Coordinates(int input) { x = 0, y = input; p = new char[10]; }
    void print() { cout << x << endl << y << endl; }
    ~Coordinates() { cout << "goodbye!" << endl; }
};
int main()
{
    Coordinates a[2] = { { 1, 2 }, {3, 4} };
    a[0].print();
    a[1].print();
    return 0;
}
```

```
C:\WINDOWS\system32\cmd.exe
1
2
3
4
goodbye!
goodbye!
请按任意键继续. . .
```

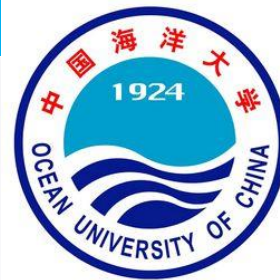
类和对象

• 析构函数

- delete运算符将导致析构函数立刻被使用

```
#include <iostream>
using namespace std;
class Coordinates {
private:
int x, y; char* p;
public:
    Coordinates(int x_, int y_) { x = x_, y = y_; }
    Coordinates() { p = new char[10]; }
    void print() { cout << x << endl << y << endl; }
    ~Coordinates()
    {
        cout << "goodbye!" << endl;
        delete[] p;
    }
};
int main()
{
    Coordinates* a = new Coordinates();
    delete a;
    cout << "end" << endl;
    return 0;
}
```

```
C:\WINDOWS\system32\cmd.exe
goodbye!
end
请按任意键继续...
```



类和对象

• 析构函数

- delete[] 将调用每个数组元素中的析构函数

```
#include <iostream>
using namespace std;
class Coordinates {
private:
int x, y; char* p;
public:
Coordinates(int x_, int y_) { x = x_, y = y_; }
Coordinates() { p = new char[10]; }
void print() { cout << x << endl << y << endl; }
~Coordinates()
{
cout << "goodbye!" << endl;
delete[] p;
};
int main()
{
Coordinates* a = new Coordinates[2];
delete[] a;
cout << "end" << endl;
return 0;
}
```

C:\WINDOWS\system32\cmd.exe

```
goodbye!
goodbye!
end
请按任意键继续 . . .
```



类和对象

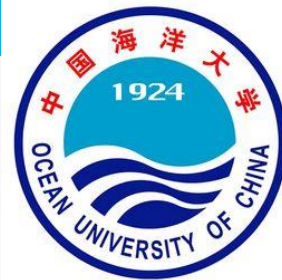
• 课堂练习

• 1. 对于通过 new 运算符生成的对象

- ☐ 执行 delete 操作时才能析构
- ☐ 在程序结束时自动析构
- ☐ 在包含该 new 语句的函数返回时自动析构
- ☐ 在执行 delete 操作时会析构，如果没有执行delete操作，则在程序结束时自动析构

• 2. 以下说法中正确的是：

- ☐ 一个类只能定义一个析构函数，但可以定义多个构造函数
- ☐ 一个类一定会有无参构造函数
- ☐ 构造函数的返回值类型是 void
- ☐ 一个类只能定义一个构造函数，但可以定义多个析构函数



类和对象

• 课堂练习

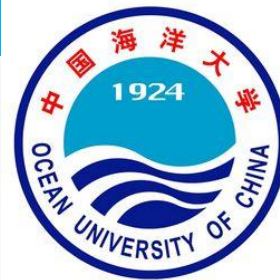
• 1. 对于通过 new 运算符生成的对象

- ☐ 执行 delete 操作时才能析构
- ☐ 在程序结束时自动析构
- ☐ 在包含该 new 语句的函数返回时自动析构

✓ 在执行 delete 操作时会析构，如果没有执行 delete 操作，则在程序结束时自动析构

• 2. 以下说法中正确的是：

- ✓ 一个类只能定义一个析构函数，但可以定义多个构造函数
- ☐ 一个类一定会有无参构造函数
- ☐ 构造函数的返回值类型是 void
- ☐ 一个类只能定义一个构造函数，但可以定义多个析构函数



类和对象

• 课堂练习

- 1. 以下程序，哪个是不正确的？



```
int main() {  
    class A { int v; };  
    A a; a.v = 3; return 0;}
```



```
int main() {  
    class A { public: int v; A * p; };  
    A a; a.p = &a; return 0;  
}
```



```
int main() {  
    class A { public: int v; };  
    A * p = new A;  
    p->v = 4; delete p;  
    return 0;}
```



```
int main() {  
    class A { public: int v; A * p; };  
    A a; a.p = new A; delete a.p;  
    return 0;  
}
```



类和对象

• 课堂练习

- 1. 以下程序，哪个是不正确的？



```
int main() {  
    class A { int v; };  
    A a; a.v = 3; return 0;}
```



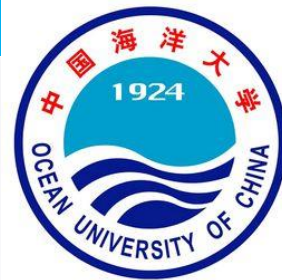
```
int main() {  
    class A { public: int v; A * p; };  
    A a; a.p = &a; return 0;  
}
```



```
int main() {  
    class A { public: int v; };  
    A * p = new A;  
    p->v = 4; delete p;  
    return 0;}
```



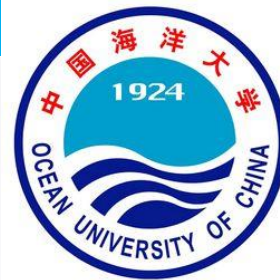
```
int main() {  
    class A { public: int v; A * p; };  
    A a; a.p = new A; delete a.p;  
    return 0;  
}
```



类和对象

• 课堂练习

```
#include <iostream>
using namespace std;
class Demo {
    int id;
public:
    Demo(int i)
    {id = i;
    cout << "id = " << id << " Constructed" << endl;}
    ~Demo() {cout << "id = " << id << " Deleted" << endl;}
};
Demo d1(1);
void Func() {
    static Demo d2(2);
    Demo d3(3);
    cout << "Func" << endl;
}
int main() {
    Demo d4(4);
    d4 = 6;
    cout << "main" << endl;
    { Demo d5(5); }
    Func();
    cout << "main ends" << endl;
    return 0;
}
```



类和对象

• 课堂练习

```
#include <iostream>
using namespace std;
class Demo {
    int id;
public:
    Demo(int i)
    {id = i;
    cout << "id = " << id << " Constructed" << endl;}
    ~Demo() {cout << "id = " << id << " Deleted" << endl;}
};
Demo d1(1);
void Func() {
    static Demo d2(2);
    Demo d3(3);
    cout << "Func" << endl;
}
int main() {
    Demo d4(4);
    d4 = 6;
    cout << "main" << endl;
    { Demo d5(5); }
    Func();
    cout << "main ends" << endl;
    return 0;
}
```

```
C:\WINDOWS\system32\cmd.exe
id = 1 Constructed
id = 4 Constructed
id = 6 Constructed
id = 6 Deleted
main
id = 5 Constructed
id = 5 Deleted
id = 2 Constructed
id = 3 Constructed
Func
id = 3 Deleted
main ends
id = 6 Deleted
id = 2 Deleted
id = 1 Deleted
请按任意键继续...
```




类和对象

• 静态成员变量和静态成员函数

- 静态成员：在说明前面加了 `static` 关键字的成员。
 - 静态成员变量/常量
 - 静态成员函数

```
#include <iostream>
using namespace std;
class Coordinates {
private:
    int x, y;
public:
    Coordinates(int x_, int y_) { x = x_, y = y_; }
    static int total_points;
    static void set_total_points(int input) { total_points = input; }
    static void print() { cout << total_points << endl; }
};
```



类和对象

• 静态成员变量和静态成员函数

- 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，但是使用前**必须初始化**。

```
#include <iostream>
using namespace std;
class Coordinates {
private:
    int x, y;
public:
    Coordinates(int x_, int y_) { x = x_, y = y_; }
    static int total_points;
    static void set_total_points(int input) { total_points = input; }
    static void print() { cout << total_points << endl; }
};
int Coordinates::total_points = 0;           //定义并初始化静态数据成员
void main()
{
    cout << sizeof(Coordinates) << endl;
}
```

C:\WINDOWS\system32\cmd.exe

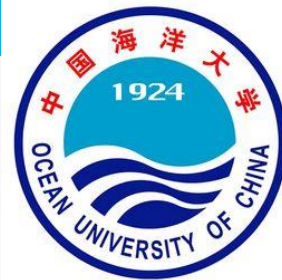
8
请按任意键继续. . .



类和对象

• 静态成员变量和静态成员函数

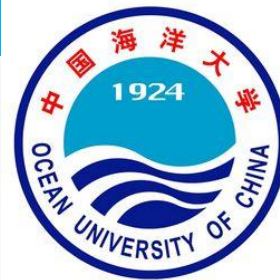
```
#include <iostream>
using namespace std;
class Coordinates {
private:
    int x, y;
public:
    Coordinates(int x_, int y_) { x = x_, y = y_; }
    static int total_points;
    static void set_total_points(int input) { total_points =
        input; }
    static void print() { cout << total_points << endl; }
};
int Coordinates::total_points = 0;           //定义并初始化静态数据成员
void main()
{
    Coordinates c(10, 20);
    c.set_total_points(int (100));           //使用静态成员函数
    Coordinates::print();                   //使用静态成员函数
    Coordinates::set_total_points(200);     //使用静态成员函数
    c.print();
    c.total_points = 300;                    //改写静态成员变量
    Coordinates d(1, 2);
    cout << d.total_points << endl; //该类所有对象共享静态成员
}
```



类和对象

• 静态成员变量和静态成员函数

- 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。
- 如果考虑一个需要随时知道对象（比如坐标点）总数的图形处理程序
- 可以用全局变量来记录总数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



类和对象

- 静态成员变量和静态成员函数

```
#include <iostream>
using namespace std;
class Coordinates {
private:
    int x, y;
public:
    Coordinates(int x_, int y_) { x = x_, y = y_; total_points++; }
    static int total_points;
    static void set_total_points(int input) { total_points = input; }
    static void print() { cout << total_points << endl; }
};

int Coordinates::total_points = 0;           //定义并初始化静态数据成员
void main()
{
    Coordinates c(10, 20);
    Coordinates d(2, 3);
    Coordinates e(2, 3);
    e.print();
}
```

C:\WINDOWS\system32\cmd.exe

3
请按任意键继续. . .



类和对象

• 静态成员变量和静态成员函数

- 注意:
- 静态成员函数中不能出现非静态成员变量!
- 非静态成员函数调用静态成员是允许的

```
class Coordinates {  
private:  
int x, y;  
public:  
Coordinates(int x_, int y_) { x = x_, y = y_; }  
static int distance;  
static void set_total_points(int input) { distance = input; }  
static void print() { cout << distance << endl; x = 1; } 非法使用  
};
```



类和对象

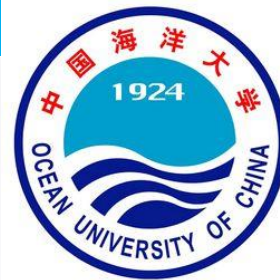
• 静态成员变量和静态成员函数

- 这样写有什么问题？

```
#include <iostream>
using namespace std;
class Coordinates {
private:
    int x, y;
public:
    static int total_points;
    Coordinates::Coordinates(int x_, int y_) { x = x_, y = y_; total_points++; } //构造函数负责把
    总数加一
    static void print() { cout << total_points << endl; } //打印总数（静态成员）
    Coordinates::~Coordinates() { total_points--; } //析构函数负责把总数减一
};
int Coordinates::total_points = 0; //定义并初始化静态数据成员
void main()
{
    Coordinates c(10, 20);
    Coordinates d(2, 3);
    Coordinates e(1, 3); //定义3个点
    Coordinates dd[2] = { { 1, 2 }, { 3, 4 } }; //一次性定义2个点
    Coordinates* f = new Coordinates(2, 3); //用指针定义1个点
    delete f; //释放指针内的成员
    e.print();
}
```

C:\WINDOWS\system32\cmd.exe

5
请按任意键继续. . .

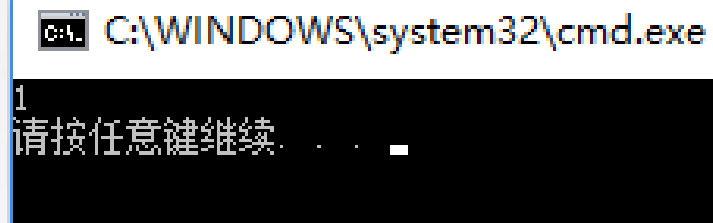


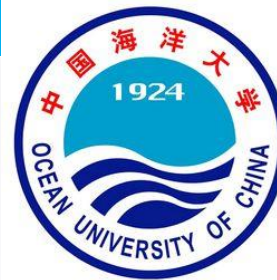
类和对象

• 静态成员变量和静态成员函数

- 这样就有问题了

```
#include <iostream>
using namespace std;
class Coordinates {
private:
    int x, y;
public:
    static int total_points;
    Coordinates(int x_, int y_) { x = x_, y = y_; total_points++; } //构造函数负责把总数加一
    static void print() { cout << total_points << endl; } //打印总数（静态成员）
    Coordinates::~Coordinates() { total_points--; } //析构函数负责把总数减一
};
int Coordinates::total_points = 0; //定义并初始化静态数据成员
void main()
{
    Coordinates c(10, 20);
    Coordinates d(c);
    Coordinates e(d); //定义3个点
    Coordinates dd[2] = { c, d }; //一次性定义2个点
    Coordinates* f = new Coordinates(2, 3); //用指针定义1个点
    delete f; //释放指针内的成员
    e.print();
}
```





类和对象

• 静态成员变量和静态成员函数

- 这样就解决问题了

```
#include <iostream>
using namespace std;
class Coordinates {
private:
    int x, y;
public:
    static int total_points;
    Coordinates(int x_, int y_) { x = x_, y = y_; total_points++; } //构造函数负责把总数加一
    static void print() { cout << total_points << endl; } //打印总数（静态成员）
    ~Coordinates() { total_points--; } //析构函数负责把总数减一
    Coordinates(Coordinates & input) { total_points++; } //重载复制构造函数
};
int Coordinates::total_points = 0; //定义并初始化静态数据成员
void main()
{
    Coordinates c(10, 20);
    Coordinates d(c);
    Coordinates e(d); //定义3个点
    Coordinates dd[2] = { c, d }; //一次性定义2个点
    Coordinates* f = new Coordinates(2, 3); //用指针定义1个点
    delete f; //释放指针内的成员
    e.print();
}
```

C:\WINDOWS\system32\cmd.exe

5
请按任意键继续...



类和对象

- 成员对象和封闭类

- 成员对象：一个类的成员变量是另一个类的对象
- 包含 成员对象 的类叫 封闭类 (Enclosing)

```
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
};
```

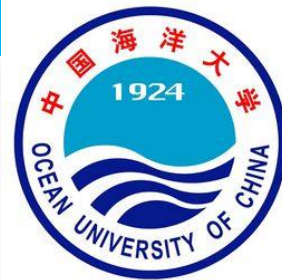


类和对象

• 成员对象和封闭类

- 成员对象：一个类的成员变量是另一个类的对象
- 包含 成员对象 的类叫 封闭类 (Enclosing)

```
class CCar { //汽车类      “封闭类”
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;
}
```



类和对象

• 成员对象和封闭类

- 成员对象：一个类的成员变量是另一个类的对象
- 包含 成员对象 的类叫 封闭类 (Enclosing)

```
class CCar { //汽车类      “封闭类”
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);
    CCar car; //可行吗?
    return 0;
}
```



类和对象

• 封闭类构造函数的初始化列表

- 定义封闭类的构造函数时，添加初始化列表：
 - 类名::构造函数(参数表):成员变量1(参数表), 成员变量2(参数表), ...
 - {
 - ...
 - }

• 调用顺序

- 当封闭类对象生成时，
 - Step1: 执行所有成员对象的构造函数
 - Step2: 执行封闭类的构造函数
- 成员对象的构造函数调用顺序
 - 和成员对象在类中的说明顺序一致
 - 与在成员初始化列表中出现的顺序无关
- 当封闭类的对象消亡时，
 - Step1 : 先执行封闭类的析构函数
 - Step2 : 执行成员对象的析构函数
- 析构函数顺序和构造函数的调用顺序相反

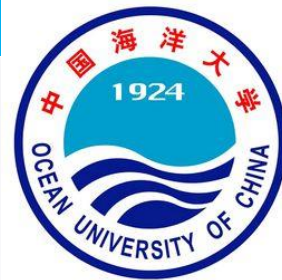


类和对象

```
#include <iostream>
using namespace std;
class CTyre {
public:
    CTyre() { cout << "CTyre constructor" << endl; }
    ~CTyre() { cout << "CTyre destructor" << endl; }
};
class CEngine {
public:
    CEngine() { cout << "CEngine constructor" << endl; }
    ~CEngine() { cout << "CEngine destructor" << endl; }
};
class CCar {
private:
    CEngine engine;
    CTyre tyre;
public:
    CCar() { cout << "CCar constructor" << endl; }
    ~CCar() { cout << "CCar destructor" << endl; }
};
int main() {
    CCar car;
    return 0;
}
```

C:\WINDOWS\system32\cmd.exe


```
CEngine constructor
CTyre constructor
CCar constructor
CCar destructor
CTyre destructor
CEngine destructor
请按任意键继续...
```



类和对象

- 友元 (Friend)
 - 友元函数
 - 一个类的友元函数可以访问该类的私有成员
 - 友元类
 - A是B的友元类 ->A的成员函数可以访问B的私有成员

将一个类的函数通过友元函数传递给另一个类：



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





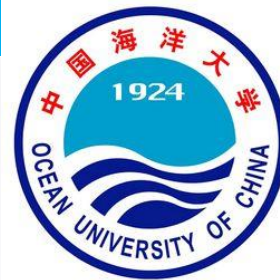
类和对象

- 友元 (Friend)
 - 友元函数
 - 一个类的友元函数可以访问该类的**私有**成员

```
using namespace std;
#include <iostream>
class CCar; //提前声明 CCar类, 以便后面CDriver类使用
class CDriver {
public:
    void ModifyCar(CCar * pCar); //改装汽车
};
class CCar {
public:
    CCar::CCar(int p) { price = p; }
private:
    int price;
    friend void CDriver::ModifyCar(CCar * pCar); //
    声明友元函数
};
```

3/20/2017

```
void CDriver::ModifyCar(CCar * pCar)
{
    pCar->price += 1000; //汽车改装后价值增加
    cout << pCar->price<<endl; //输出增加后的价格
}
int main()
{
    CCar c(10000); //定义一辆价值10000的汽车
    CDriver cd; //定义一件引擎
    cd.ModifyCar(&c); //用引擎改装汽车, 使得汽车
    价值增加
    return 0;
}
```

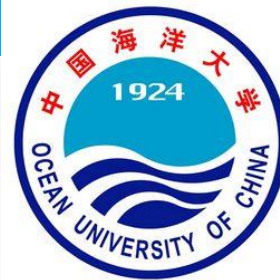
类和对象

- 友元 (Friend)
 - 友元类
 - 一个类的友元函数可以访问该类的私有成员

```
using namespace std;
#include <iostream>
class CCar; //提前声明 CCar类, 以便后面CDriver类使用
class CDriver {
public:
    void ModifyCar(CCar * pCar); //改装汽车
};
class CCar {
public:
    CCar(int p) { price = p; }
private:
    int price;
    friend class CDriver; //声明友元类;
```

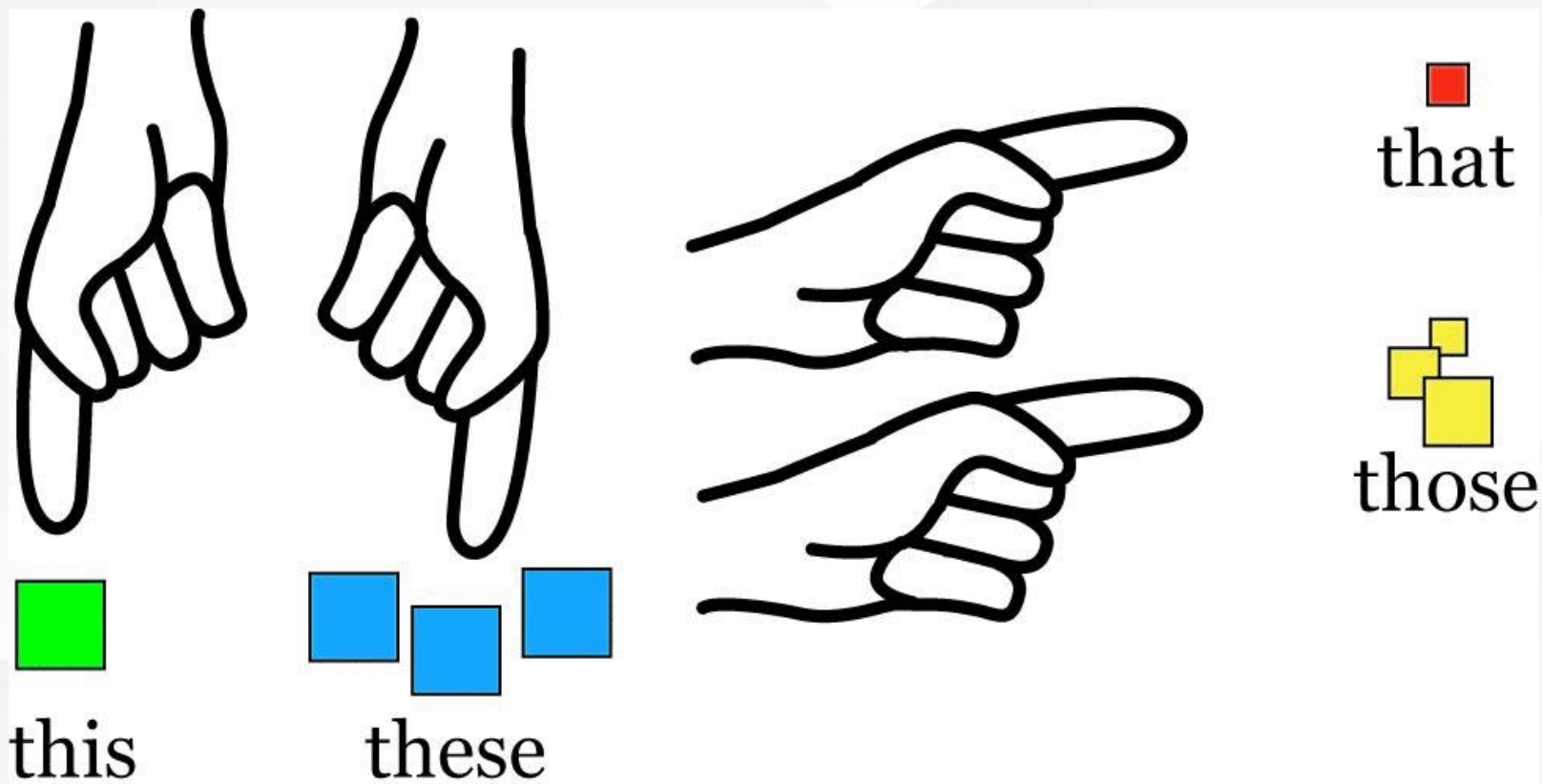
```
void CDriver::ModifyCar(CCar * pCar)
{
    pCar->price += 1000; //汽车改装后价值增加
    cout << pCar->price<<endl; //输出增加后的价格
}
int main()
{
    CCar c(10000); //定义一辆价值10000的汽车
    Engine e; //定义一件引擎
    cd.ModifyCar(&c); //用引擎改装汽车, 使得汽车
    return 0;
}
```

注意! 友元之间的关系不能被传递和继承!



类和对象

- this指针
 - 现实中的this





类和对象

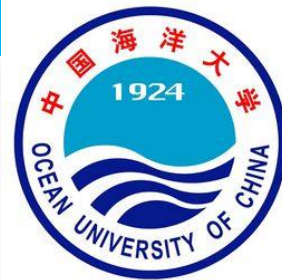
- this指针
 - C中的this

```
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++

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

C



类和对象

- this指针
 - C++中的this
 - 其作用就是指向成员函数所作用的**对象**
 - **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指针。

```
using namespace std;
#include <iostream>
class Complex {
public:
    double real, imag;
    void Print() { cout << real << ", " << imag << endl; }
    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
```

C:\Windows\system32\cmd.exe

2,1
请按任意键继续. . .



类和对象

- this指针
 - C++中的this
 - 其作用就是指向成员函数所作用的对象
 - 但前提是这个对象不是空的!

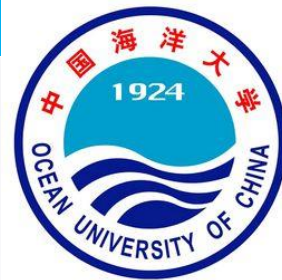
```
using namespace std;
#include <iostream>
class A
{
    int i;
public:
    void Hello() { cout << this->i << endl; }
};
int main()
{
    A * p = NULL;
    p->Hello(); //结果会怎样?
}
```



类和对象

- this指针
 - C++中的this
 - 其作用就是指向成员函数所作用的对象
 - 但前提是这个对象不是空的!

```
using namespace std;
#include <iostream>
class A
{
    int i;
public:
    void Hello() { cout << this->i << endl; }
};
int main()
{
    A * p=new A();           //改成这样就可以
    p->Hello();               //结果会怎样?
}
```



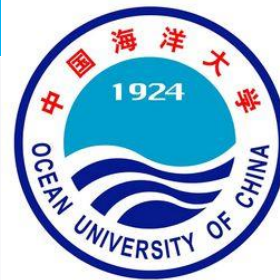
类和对象

- 常量对象、常量成员函数和常引用
 - 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 Demo
{
private:
    int value;
public:
    void SetValue() { }
};

const Demo Obj; // 常量对象

void main()
{
    Obj.SetValue(1); // 可以直接使用常量对象
}
```



类和对象

- 常量对象、常量成员函数和常引用
 - 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。
 - 常量成员函数执行期间不应修改其所作用的对象。因此，在常量成员函数中不能修改成员变量的值（静态成员变量除外），也不能调用同类的非常量成员函数（静态成员函数除外）。

```
class Sample
{
public:
    int value;
    void GetValue() const;
    void func() { };
    Sample() { }
};

void Sample::GetValue() const
{
    value = 0; // 错误，常量函数内
               // 不应修改其作用的对象
    func(); // 错误
}
```

```
int main() {
    const Sample o;
    o.value = 100; // 错误，常量对象不可被修改
    o.func(); // 错误，常量对象上面不能执行非常量成员函数
    o.GetValue(); // 可行，常量对象上可以执行常量成员函数
    return 0;
}
```




类和对象

- 常量对象、常量成员函数和常引用
 - 两个成员函数，名字和参数表都一样，但是一个是const，一个不是，算重载。

```
using namespace std;
#include <iostream>
class CTest {
private:
    int n;
public:
    CTest() { n = 1; }
    int GetValue() const { return n; }
    int GetValue() { return 2 * n; }
};
int main() {
    const CTest objTest1;
    CTest objTest2;
    cout << objTest1.GetValue() << ", " <<
    objTest2.GetValue();
    return 0;
}
```



类和对象

- 常量对象、常量成员函数和常引用
 - 引用前面可以加`const`关键字，成为常引用。
 - 不能通过常引用，修改其引用的变量。

```
int main() {  
    int n = 0;  
    const int & r = n;  
    r = 5; //错误  
    n = 4; //可行  
    return 0;  
}
```

- 为什么要用常引用？
- 我们知道对象作为函数的参数时，生成该参数需要调用复制构造函数，效率比较低。用指针作参数，代码又不好看……这可以通过引用解决

```
class Sample {  
    ...  
};  
void PrintfObj(Sample & o)  
{  
    .....  
}
```

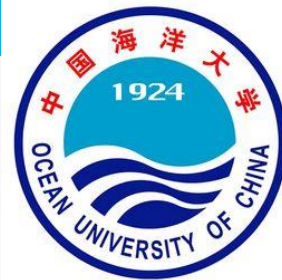


类和对象

- 常量对象、常量成员函数和常引用
 - 但……
 - 对象引用作为函数的参数有一定**风险性**，若函数中不小心修改了形参，则实参也跟着变，这可能不是我们想要的。如何避免？
 - ——通过常引用

```
#include <iostream>
using namespace std;
class Sample {
public:
    int a = 0;
};
void PrintfObj(const Sample & o)
{
    cout << o.a << endl;
}
void main()
{
    Sample s;
    s.a = 1;
    PrintfObj(s);
}
```

```
#include <iostream>
using namespace std;
class Sample {
public:
    int a = 0;
};
void PrintfObj(Sample & o)
{
    o.a=2;
    cout << o.a << endl;
}
void main()
{
    Sample s;
    s.a = 1;
    PrintfObj(s);
}
```



类和对象

• 小结

- 类成员的访问范围
- 内联成员函数
- 重载成员函数
- 构造函数
 - 复制构造函数
 - 类型转换构造函数
- 析构函数
- 静态成员变量和静态成员函数
- 封闭类和友元
- `this`指针
- 常量对象、常量成员函数和常引用

*Thank
You!*

