面向对象

面向过程与面向对象

摇狗尾巴： 面向过程

狗摇尾巴： 面向对象

客观世界与面向对象

客观世界：由各种不同类型的实体构成
面向对象：类型——类；实体——对象

客观世界：不同的实体拥有不同的属性
面向对象：属性——成员变量

客观世界：不同的实体能够执行不同的动作
面向对象：动作——成员函数

面向对象的编程思想能够让程序更加贴近于客观世界。

面向对象的三大特征

封装
继承
多态

封装

从两个方面来理解封装

信息组合

信息隐藏

结构

C语言的结构（struct）已经具备一定程度的信息组合的功能。

setname-1.c

```
struct student
{
    char name[10];
    int number;
    int score;
};
```

```
int main()
{
    struct student zs;

    strcpy(zs.name,"zhangsan");
    zs.number = 1;
    zs.score = 90;

    printf("%s %d %d\n", zs.name, zs.number, zs.score);
    return 0;
}
```

但是，struct的所有字段，都是公开的，任何人都可以随意读写。

这可能带来两个方面的问题：

1. 易用性问题
   使用者需要了解太多struct的细节，才能用好这个struct。
2. 安全性问题
   无法确保使用者按照正确的方式使用struct。

```
strcpy(zs.name,"zhangsan");
zs.number = 1;
zs.score = 95;
```

如何解决?

定义和使用安全的函数：set_name、set_number、set_score ...

setname-2.c

```
void set_name(struct student * stu, char const * text)
{
    if(strlen(text)>=10)
    {
        printf("length of %s >= 10 \n", text);
        exit(0);
    }

    strcpy(stu->name,text);
}
...

set_name(&zs, "zhangsan");
```

但是，仍然不能阻止使用者直接访问struct内部。

```
strcpy(zs.name,"zhangsan");
```

这是因为C语言并不支持“隐藏”。

struct内部的所有细节，对外都是“可见”的。

真正的“隐藏”，需要同时解决上述两个问题：

1. 易用性问题
   实现者只“公开”必要的信息，其它不必要的信息全部“隐藏”起来；
   使用者只需要了解必要的信息。
2. 安全性问题
   实现者只“公开”安全的信息，其它不安全的信息全部“隐藏”起来；
   使用者只能访问安全的信息。

C++的类（class）解决了这两个问题。

类

C++的类（class），把需要公开的部分和需要隐藏的部分区分开来，分别通过public和private说明。

setname-3.cpp

```
class student
{
    private:
    char name[10];

    public:
    int number;
    int score;
```

```
    void set_name(char const * text)
    {
        if(strlen(text)>=10)
        {
            cout << "length of " << text << " >= 10 " << endl;
            exit(0);
        }
        strcpy(name,text);
    }
};
```

```
int main()
{
    student zs;

    zs.set_name("zhangsan");
    zs.number = 1;
    zs.score = 90;
    zs.display();

    return 0;
}
```

name被隐藏起来，不能直接访问，只能通过get_name等函数间接访问。

也可以用同样的方法把number和score隐藏起来。

无论是变量还是函数，只要放到private区，都会被“隐藏”，类的实现者，即class内部的代码，可以访问；类的使用者，即class外部的代码，不能访问。

无论是变量还是函数，只要放到public区，都会被“公开”，类的实现者和使用者都可以访问。

小结

1. 组合：C的struct只能把变量组合起来，每个变量称为一个“字段”。C++的class可以把变量和函数组合起来，每个变量称为一个“成员变量”，每个函数称为一个“成员函数”。“成员变量”和“成员函数”统称为“类成员”。
2. 隐藏：C的struct不具备隐藏的功能，使用者可以访问所有“字段”。C++的class具备隐藏功能，可以把需要公开的“类成员”放到public区，供使用者使用；把需要隐藏的“类成员”放到private区，对使用者透明。

为了提供兼容性，C++也支持struct，但它的功能在C的struct的基础上有所扩展，更加接近于C++的class。

【C++中struct与class的区别与比较】

https://blog.csdn.net/weixin_39640298/article/details/84349171

对象

类（class）是特殊的数据类型，和其它数据类型一样，类可以用来定义变量（variable）。

用数据类型来定义变量，称为该数据类型的实例化。定义出来的变量，称为该数据类型的实例（instance）。

类（class）的实例有一个特殊的名称——对象（object）。

```
int a,b,c;
struct student zs,ls,ww;
```

```
class student { ... }
student zs,ls,ww;
```

客观世界中存在各种不同类型的实体，每种类型对应一个类（class），每个实体对应一个对象（object）。

所以，类和对象是对客观世界的类型和实体的抽象描述，使用面向对象的方法，可以设计出更加接近于客观世界的程序代码。

例如：旺财是一只狗，狗这种类型都拥有尾巴这个属性，并且能够执行摇尾巴这个动作，所以，狗是一个类，旺财是一个对象。

dog.cpp

```
class dog {
    private:
    char tail[10]="\n~~~\n";    /* 私有成员，“别人”不能碰狗的尾巴 */

    public:
    void wag()                  /* 公有成员，“别人”可以叫狗摇尾巴 */
    {
        cout << tail;
    }
};

dog wangcai;
cout << wangcai.tail;           /* 错误：访问私有成员 */
wangcai.wag();                  /* 正确：访问公有成员 */
```

构造函数和析构函数

初始化

每个变量都对应一段内存空间，在定义变量的时候，可以对这段内存空间进行初始化。

初始化只有一次机会，就是在定义这个变量的时候。

如果在定义变量的时候不进行初始化，则该变量（对应的内存空间）的值是随机的，直到第一次为该变量赋值。

```
int a;
int b;
cout << a << " " << b << endl;

a=1;
b=2;
cout << a << " " << b << endl;
```

在定义变量的时候，可以使用括号或赋值符号对变量进行初始化操作。

```
int a(0);
int b=0;
cout << a << " " << b << endl;

a=1;
b=2;
cout << a << " " << b << endl;
```

同样，如果在定义对象的时候不进行初始化，则该对象的各个成员变量（对应的内存空间）的值是随机的，直到第一次为该对象的各个成员变量赋值。

```
student zs;
zs.display();

zs.set_name("zhangsan");
zs.number = 1;
zs.score = 90;

zs.display();
```

在定义对象的时候，可以使用括号或赋值符号对这个对象进行初始化。
可以使用已有的对象，对新的对象进行初始化。
这样的方式初始化出来的新对象，和原对象是一模一样的。

```
student ls(zs);
ls.display();

student ww=zs;
ls.display();
```

构造函数

为满足不同的初始化需求，我们可以自己定义“构造函数”，实现其它的初始化方法。

构造函数，是一种特殊的成员函数，函数名和类名完全相同，没有返回类型。构造函数的主要任务，是对对象进行“初始化”操作。

可以根据需求定义多个构造函数，它们之间通过不同的参数区分（重载构造函数）。

```
class student
{
    public:
    student()
    {
        strcpy(name,"none");
        number=0;
        score=0;
    }
```

```
    student(char const * text)
    {
        strcpy(name,text);
        number=0;
        score=0;
    }

    student(char const * text, int n, int s)
    {
        strcpy(name,text);
        number=n;
        score=s;
    }
}
```

构造函数在对象初始化时调用，如果有多个构造函数，编译器根据参数的个数和类型确定选用哪一个构造函数。

```
student zs;                    /* 注: 不能是student zs(); */
student ls("lisi",2,80);
student ww("wangwu");
```

以下书写方式和上面等效。

```
student zs={};
student ls={"lisi",2,80};
student ww={"wangwu"};
```

如果没有自定义的构造函数，编译器会提供一个默认的不带任何参数的构造函数，这个构造函数不会对成员变量进行任何操作。一旦有了自定义的构造函数，编译器就不会提供这个默认的构造函数了。

思考：以下代码会出现什么问题？

constructor-1.cpp

```
student zs;
```

拷贝构造函数

另外，编译器还会提供另一个默认的构造函数，叫做“拷贝构造函数”或“复制构造函数”。

当使用一个已有对象对新对象进行初始化时，就会调用默认的“拷贝构造函数”。

默认的“拷贝构造函数”会采用“位拷贝”的方式来初始化新对象，所以初始化出来的新对象和原对象是一模一样的。

```
student ls(zs);
ls.display();

student ww=zs;
ls.display();
```

也可以自己定义“拷贝构造函数”，该函数需要：

1. 带一个参数
2. 参数类型是该类的引用类型
   一旦有了自定义的“拷贝构造函数”，编译器就不会提供默认的“拷贝构造函数”了。

```
student(student & s)
{
    strcpy(name,s.name);
    number=s.number;
    score=s.score;
}
```

思考

默认的“拷贝构造函数”已经能够完成拷贝对象的工作了，为什么还需要自定义的“拷贝构造函数”？

一个函数中如果存在指针变量，在函数退出时，系统会自动释放指针变量的内存空间（4或8字节），但不会自动释放它指向的内存空间。

delete-1.cpp

```
void func(int i)
{
    char *p;
    p=new char[num];
    ...
}
```

一个对象中如果存在指针类型的成员变量，在释放该对象时，系统会自动释放指针变量的内存空间，但不会自动释放它指向的内存空间。

delete-2.cpp

```
void func(int i)
{
    student zs;
    cout  << i << endl;
}
```

```
void func(int i)
{
    student * zs;
    zs = new student;
    cout  << i << endl;
    delete zs;
}
```

所以，需要在释放对象的时候执行一些“扫尾”工作，以释放动态分配的内存空间。这个“扫尾”工作就是由“析构函数”完成的。

析构函数

析构函数也是一种特殊的成员函数，函数名是在类名前面加一个~号，没有返回类型，也没有参数。析构函数不能重载，每个类只有一个析构函数。析构函数在释放对象时调用，可用于释放动态申请的内存空间或其它“扫尾”工作。

delete-3.cpp

```
~student()
{
    delete[] resume;
}
```

每个对象在释放的时候，都会调用自己的析构函数。

```
void func(int i)
{
    student zs;
    student ls;
    cout  << i << endl;
}
```

思考：以下代码会出现什么问题？

delete-4.cpp

```
void func(int i)
{
    student zs;
    student ls(zs);
    cout  << i << endl;
}
```

浅拷贝 vs 深拷贝

普通变量的拷贝：

```
int a;
a=1;
cout << a << endl;

int b;
b=a;
cout << b << endl;
```

指针的拷贝：

```
char *p1;
p1=new char[1000000];
strcpy(p1,"hello");
cout << p1 << endl;
```

```
char *p2;
p2=p1;
cout << p2 << endl;
```

```
char *p2;
p2=new char[1000000];
strcpy(p2,p1);
cout << p2 << endl;
```

两者都实现了指针的拷贝，但前者是拷贝指针本身的值，并没有拷贝指针指向的内容（浅拷贝）；后者是拷贝指针指向的内容（深拷贝）。对浅拷贝来说，对一个指针指向的内容的进行操作，会对另一个指针产生影响；对深拷贝来说，对一个指针指向的内容进行操作，不会对另一个指针造成影响。

```
strcpy(p1,"world");
cout << p2 << endl;
```

在释放内存时，浅拷贝可能造成更大的影响。

```
delete[] p1;
delete[] p2;
cout << "OK" << endl;
```

默认的“拷贝构造函数”采用“位拷贝”的方式来初始化新的对象，如果成员变量中有指针类型的变量，则只能达到“浅拷贝”的效果。

如要达到“深拷贝”的效果，就必须自己定义“拷贝构造函数”。

delete-5.cpp

```
student(student & s)
{
    strcpy(name,s.name);
    number=s.number;
    score=s.score;
    resume=new char[num]();
    strcpy(resume,s.resume);
}
```

临时对象

一般情况下，构造函数是在定义一个对象时，由系统自动调用的。

特殊情况下，也可以直接调用构造函数。

直接调用构造函数，会生成一个临时对象。

临时对象的生存期是由系统自动控制的，不同的编译器可能有不同的处理方式，通常情况下都会把临时对象作为一个“右值”看待。

取地址

```
int a=1;
int * p = &(a+2);
```

```
class student() {...};
student * p = &(student());
```

引用的初始化

```
int & r = a+2;
```

```
student & r = student();
```

参数传递

```
void func(int & x) { ... }
int main()
{
    int a=1;
    func(a+2);
    return 0;
}
```

```
void func(student & x) { ... }
int main()
{
    func(student());
    return 0;
}
```

思考

以下代码会出现什么问题？

temporary-1.cpp

```
student zs=student();
zs.display();

student ls=student("lisi");
ls.display();

student ww=student("wangwu",3,80);
ww.display();
```

数组

数组的定义

普通数组

```
int a[3];
```

对象数组

```
student stu[3];
```

数组的使用

普通数组

```
a[0]=0;
for(int i=0; i<3; i++)
{
        cout << a[i] << endl;
}
```

对象数组

```
stu[0].set_name("zhangsan");
stu[0].number = 1;
stu[0].score = 90;

for(int i=0; i<3; i++)
{
        stu[i].display();
}
```

数组的初始化

普通数组

```
int a[3] = { 0, 1, 2 };
for(int i=0; i<3; i++)
{
        cout << a[i] << endl;
}
```

对象数组

```
student stu[3] = { {}, {"lisi",2,80}, {"wangwu"} };
for(int i=0; i<3; i++)
{
        stu[i].display();
}
```

或者

```
student stu[3] = { student(), student("lisi",2,80), student("wangwu") };
for(int i=0; i<3; i++)
{
        stu[i].display();
}
```

指针

指针的定义

普通指针

```
int *ip;
```

对象指针

```
student *sp;
```

指针的使用

普通指针

```
int i(100);
ip = &i;
cout << *ip << endl;
```

对象指针

```
student zs("zhangsan",1,90);
sp = &zs;
sp->display();
(*sp).display();
```

指向动态分配的内存

普通指针

```
ip = new int(200);
cout << *ip << endl;
delete ip;
```

对象指针

```
sp = new student("lisi",2,80);
sp->display();
(*sp).display();
delete sp;
```

指向数组元素

以指针的形式访问

普通指针

```
int ia[3]={ 1, 2, 3 };
ip=ia;
for(int i=0; i<3; i++)
{
    cout << *ip << endl;
    ip++; // ip = ip + 1;
}
cout << *(ip-3) << endl;
```

对象指针

```
student stu[3] = { {"zhangsan",1,90 }, {"lisi",2,80}, {"wangwu",3,70} };
sp = stu;
for(int i=0; i<3; i++)
{
    sp->display(); // (*sp).display();
    sp++; // sp = sp + 1;
}
(sp-3)->display();
```

以数组的形式访问

普通指针

```
int ia[3]={ 1, 2, 3 };
ip=ia;
for(int i=0; i<3; i++)
{
    cout << ip[i] << endl;
}
```

对象指针

```
student stu[3] = { {"zhangsan",1,90 }, {"lisi",2,80}, {"wangwu",3,70} };
sp = stu;
for(int i=0; i<3; i++)
{
    sp[i].display();
}
```

this指针

上述实例，都是从一个对象的外部，通过指针访问该对象。

有的时候，需要从一个对象的内部，通过指针访问该对象。

这是通过this指针实现的。

```
student(char const * text, int n, int s)
{
    strcpy(this->name,text);
    this->number=n;
    this->score=s;
}
```

默认情况下，编译器会自动补充this指针（定义参数和访问对象成员的时候），所以一般不需要明确的写出来。但是，在有的情况下，需要把this指针明确的写出来。

```
student(char const * name, int number, int score)
{
    strcpy(this->name,name);
    this->number=number;
    this->score=score;
}
```

通过调试器可观察到编译器自动补充的this指针。

```
(gdb) bt
#0  student::student (this=0x61fea8,
    name=0x40405a <std::piecewise_construct+22> "zhangsan", number=1, score=90) at test.cpp:48
#1  0x004015f8 in main () at test.cpp:90

(gdb) bt
#0  student::student (this=0x61fe90,
    text=0x404063 <std::piecewise_construct+31> "lisi") at test.cpp:35
#1  0x00401616 in main () at test.cpp:93
```

参数传递

普通参数

整型变量

```
void swap(int x, int y)
{
    int tmp;
    tmp=x;
    x=y;
    y=tmp;
}
```

对象

```
void swap_number(student s1, student s2)
{
    int tmp;
    tmp = s1.number;
    s1.number = s2.number;
    s2.number = tmp;
}
```

```
int main()
{
    student zs("zhangsan",1,90);
    zs.display();

    student ls("lisi", 2, 80);
    ls.display();

    swap_number(zs, ls);
    zs.display();
    ls.display();
    return 0;
}
```

指针参数

整型指针

```
void swap(int * x, int * y)
{
    int tmp;
    tmp=*x;
    *x=*y;
    *y=tmp;
}
```

对象指针

```
void swap_number(student * s1, student  * s2)
{
    int tmp;
    tmp = s1->number;
    s1->number = s2->number;
    s2->number = tmp;
}
```

```
int main()
{
    student zs("zhangsan",1,90);
    zs.display();

    student ls("lisi", 2, 80);
    ls.display();

    swap_number(&zs, &ls);
    zs.display();
    ls.display();
    return 0;
}
```

引用参数

整型引用

```
void swap(int & x, int & y)
{
    int tmp;
    tmp=x;
    x=y;
    y=tmp;
}
```

对象引用

```
void swap_number(student & s1, student  & s2)
{
    int tmp;
    tmp = s1.number;
    s1.number = s2.number;
    s2.number = tmp;
}
```

```
int main()
{
    student zs("zhangsan",1,90);
    zs.display();

    student ls("lisi", 2, 80);
    ls.display();

    swap_number(zs, ls);
    zs.display();
    ls.display();
    return 0;
}
```

友元

封装实现了：1、组合；2、隐藏

解决了：安全性问题

但是：安全性 vs 方便性，权衡？

友元打破了类的权限规则。为一个类设置友元后，该类的所有成员对该友元都是可见的。友元机制提高了方便性，但降低了安全性。

有两种类型的友元：1、友元函数；2、友元类。

友元函数

```
class student
{
    ...
    friend void display_name(student s);
};

void display_name(student s)
{
    cout << s.name << endl;
}
```

```
int main()
{
    student zs("zhangsan",1,90);
    display_name(zs);
    return 0;
}
```

友元类

```
class student
{
    ...
    friend class teacher;
};

class teacher
{
    void check(student s)
    {
        cout << s.name << endl;
        cout << s.number << endl;
        cout << s.score << endl;
    }
};
```

```
int main()
{
    student zs("zhangsan",1,90);
    teacher t;
    t.check(zs);
    return 0;
}
```

类的组合

一个类的成员变量，可以是普通的数据类型，也可以是类类型。

如果一个成员变量是类类型，称为“类的组合”。

类的组合体现了对象的包含关系。

在使用类的组合时，要特别注意一个对象和它包含的对象的初始化问题。

一个对象的构造函数中可以定义一个成员初始化列表，在该列表中，可以对该对象的成员（包括对象成员）进行初始化。

```
#include <iostream>
#include <string.h>
using namespace std;

class point
{
    public:
    float x,y;
    point(float x, float y)
    {
        this->x = x;
        this->y = y;
    }
    // or
    point(float x, float y):x(x),y(y)
    {
    }
};
```

```
class circle
{
    public:
    point center;
    float radius;
    circle(float x, float y, float r) : center(x,y)
    {
        radius = r;
    }
    // or
    circle(float x, float y, float r) : center(x,y),radius(r)
    {
    }
};
```

```
int main()
{
    circle c(50,60,20);
    cout << c.center.x << endl;
    cout << c.center.y << endl;
    cout << c.radius << endl;

    return 0;
}
```

构造函数的调用顺序：先成员，后自己。

析构函数的调用顺序：先自己，后成员。

如果有多个成员，要注意成员之间的先后顺序。

情况很复杂，方法很简单：

```
point()
{
    cout << "point" << endl;
}

~point()
{
    cout << "~point" << endl;
}

circle()
{
    cout << "circle" << endl;
}

~circle()
{
    cout << "~circle" << endl;
}
```

常对象和常成员

常对象

常量：初始化之后，就不能再修改的变量。

特别注意：初始化 ≠ 修改

```
int a=1;
a=2;

int const b=1;
b=2;  // error
```

常对象：初始化之后，就不能再修改的对象。

```
student zs("zhangsan",1,90);
zs.score=100;

student const ls("lisi");
ls.score=100;  // error
```

对于成员函数，是否会修改对象？

```
student zs("zhangsan",1,90);
zs.display();

student const ls("lisi");
ls.display();  // ???
```

编译器默认会认为所有成员函数都会修改对象。

除非明确指出：本成员函数不会修改对象——常成员函数。

常成员函数

把函数声明为常成员函数。

```
void display() const
{
    ...
}
```

作弊会被抓到!

```
void display() const
{
    ...
    score = 100;  // error
}
```

常成员变量

学号是一个学生入学就确定了的（初始化），以后不能再修改。

```
int const number;
```

思考： 以下代码会出什么问题？

const-member-1.cpp

常成员变量必须初始化，且只能在构造函数的成员初始化列表中进行初始化。合理使用常对象和常成员，能够增强程序的安全性和可控性。

静态成员

静态成员变量

所有对象共享的数据，可设为静态变量。静态变量只有一份副本，所以，静态变量属于“类”，而非静态变量属于“对象”。

对于静态成员变量，类的内部只是“申明”，类的外部才是“定义”。

在定义静态成员变量时，可以进行初始化。

```
class student
{
    public:
    ...
    static char school[10];
    ...
}

char student::school[10] = "UESTC";
```

```
int main()
{
    student zs("zhangsan",1,90);
    zs.display();
    cout << zs.school << endl;  // cout << student::school << endl;

    student ls("lisi", 2, 80);
    ls.display();
    cout << ls.school << endl;  // cout << student::school << endl;

    return 0;
}
```

通过调试器观察静态成员变量的地址。

```
(gdb) p &student::school
$2 = (char (*)[10]) 0x403008 <student::school>
(gdb) p sizeof(student::school)
$3 = 10
(gdb) p &zs
$4 = (student *) 0x61fea8
(gdb) p sizeof(zs)
$5 = 24
(gdb) p &ls
$6 = (student *) 0x61fe90
(gdb) p sizeof(ls)
$7 = 24
```

所以，静态成员变量不属于“对象”，而属于“类”。没有对象，也可以访问静态成员变量。访问的方式是通过“类”。

```
int main()
{
    strcpy(student::school, "uestc");
    cout << student::school << endl;
    return 0;
}
```

静态成员函数

静态成员函数是没有this指针的成员函数。

编译器不会为静态成员函数自动补充this指针。

static-member.cpp

通过调试器检查编译器是否为静态成员函数补充this指针。

```
(gdb) bt
#0  student::set_school (
    text=0x404045 <std::piecewise_construct+1> "uestc") at test.cpp:90
#1  0x004015e2 in main () at test.cpp:98
```

所以，在不指定对象的情况下，静态成员函数是无法访问对象的普通成员变量的。但是，在不指定对象的情况下，静态成员函数仍然可以访问静态成员变量，因为静态成员变量属于“类”，而不属于“对象”。

```
static void set_school(char const * text)
{
    strcpy(school, text);
    cout << school << endl;
    cout << name << endl; // error
    cout << number << endl; // error
    cout << score << endl; // error
}
```

如果希望用静态成员函数访问普通成员变量，必须指定明确的对象。

```
static void set_school(char const * text)
{
    strcpy(school, text);
    cout << school;

    student s;
    cout << s.name;
    cout << s.number;
    cout << s.score;
}
```

静态成员的应用

单件模式：一种常见的设计模式。

一般情况下，一个类可以有多个实例（对象）

```
student zs("zhangsan", 1, 90);
zs.display();

student ls("lisi", 2, 80);
ls.display();

student ww("wangwu ", 3, 70);
ww.display();
```

在某些应用场景中，一个类最多只允许一个实例，如何实现?

每创建一个实例，就会调用一次构造函数。

上述构造函数是从类外调用的 or 从类内调用的?

类外调用：使用者调用，不可控。

类内调用：实现者调用，可控。

只允许调用一次，所以不能从类外调用 —— 隐藏构造函数。

```
class student
{
    private:
    student() { ... }
    ...
}
```

隐藏构造函数后，不能从类外调用构造函数。
只能从类内（成员函数中）调用构造函数。

```
class student
{
    private:
    student() { ... }
    ...
    new_student() { ... student() ... }
    ...
}
```

问题1：调用student()前，对象是否存在?
问题2：new_student()属于类 or 对象?
问题3：new_student()应该公开 or 隐藏?
问题4：如何控制student()最多只调用一次?

完整的代码如下。

student-vip.cpp

```
student *zs = student::new_student("zhangsan", 1, 90);
zs->display();

student *ls = student::new_student("lisi", 2, 80);
ls->display();

student *ww = student::new_student("wangwu ", 3, 70);
ww->display();
```

课后复习

课程MOOC

www.icourse163.org/course/UESTC-1001774006

第八章 类与对象

码图作业

matu.uestc.edu.cn

第8章 作业1

第8章 作业2