一、

类定义

```
class Score{
    public:
    void setScore(int m, int f);
    void showScore();
    private:
    int mid_exam;
    int fin_exam;
};
//不能在类声明中给数据成员赋初值。
Score op1, op2;
对象名.数据成员名对象名.成员函数名[(参数表)]
op1.setScore(89, 99);
op1.showScore();
```

成员函数

```
返回值类型 类名::成员函数名(参数表){函数体}
void Score::setScore(int m, int f)
{
    mid_exam = m;
    fin_exam = f;
}

void Score::showScore()
```

```
{
    cout << "期中成绩: " << mid_exam << endl;
    cout << "期末成绩: " << fin_exam << endl;
}
```

友元

```
//类内部
friend int getScore(Score &ob);
//外部
int getScore(Score &ob)
{
    return (int)(0.3 * ob.mid_exam + 0.7 *
ob.fin_exam);
}
```

友元必须通过作为**入口参数**传递进来的对象名（或对象指针、对象引用）来访问该对象的数据成员。友元函数提供了不同类的成员函数之间、类的成员函数与一般函数之间进行数据共享的机制。尤其当一个函数需要访问多个类时，友元函数非常有用，普通的成员函数只能访问其所属的类，但是多个类的友元函数能够访问相关的所有类的数据。

一个类的成员函数作为另一个类的友元函数时，必须先定义这个类。并且在声明友元函数时，需要加上成员函数所在类的类名.

```
class Y{
    ...
};
class X{
    friend Y;     //声明类Y为类X的友元类
};
```

当一个类被说明为另一个类的友元类时，它所有的成员函数都成为另一个类的友元函数，这就意味着作为友元类中的所有成员函数都可以访问另一个类中的所有成员。友元关系不具有**交换性**和**传递性**。

静态成员及静态成员函数

在一个类中，若将一个数据成员说明为`static`，则这种成员被称为**静态数据成员**。与一般的数据成员不同，无论建立多少个类的对象，都只有一个静态数据成员的拷贝。从而实现了同一个类的不同对象之间的数据共享。

静态数据成员初始化应在类外单独进行，而且应在定义对象之前进行。一般在 main() 函数之前、类声明之后.

可以使用“`类名::`”访问静态的数据成员。格式如下：`类名::静态数据成员名`。

```
//对象定以后，共有的静态数据成员也可以通过对象进行访问。
对象名.静态数据成员名;
对象指针->静态数据成员名;
```

静态成员函数属于整个类，是该类所有对象共享的成员函数，而不属于类中的某个对象。静态成员函数的作用不是为了对象之间的沟通，而是为了处理静态数据成员。定义静态成员函数的格式如下：

```
static 返回类型 静态成员函数名（参数表）；
//调用公有静态成员函数的一般格式有如下几种：
类名::静态成员函数名(实参表);
对象.静态成员函数名(实参表);
对象指针->静态成员函数名(实参表);
```

静态成员函数只能通过对象名（或对象指针、对象引用）访问该对象的非静态成员

使用静态成员函数的一个原因是，可以用它在建立任何对象之前调用静态成员函数

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

class Score{
private:
    int mid_exam;
    int fin_exam;
    static int count;      //静态数据成员，用于统计学生人数
    static float sum;      //静态数据成员，用于统计期末累加成绩
    static float ave;      //静态数据成员，用于统计期末平均成绩
public:
    Score(int m, int f);
    ~Score();
```

```
    static void show_count_sum_ave();   //静态成员函数
};

Score::Score(int m, int f)
{
    mid_exam = m;
    fin_exam = f;
    ++count;
    sum += fin_exam;
    ave = sum / count;
}

Score::~Score()
{

}

/*** 静态成员初始化 ***/
int Score::count = 0;
float Score::sum = 0.0;
float Score::ave = 0.0;

void Score::show_count_sum_ave()
{
    cout << "学生人数: " << count << endl;
    cout << "期末累加成绩: " << sum << endl;
    cout << "期末平均成绩: " << ave << endl;
}

int main()
{
```

```
    Score sco[3] = {Score(90, 89), Score(78,
99), Score(89, 88)};
    sco[2].show_count_sum_ave();
    Score::show_count_sum_ave();

    return 0;
}
```

构造函数与拷贝构造函数

构造函数是一种特殊的成员函数，它主要用于为对象分配空间，进行初始化。构造函数的名字必须与类名相同，而不能由用户任意命名。它可以有任意类型的参数，但不能具有返回值。它不需要用户来调用，而是在建立对象时自动执行。

```
class Score{
public:
    Score(int m, int f);  //构造函数
    void setScore(int m, int f);
    void showScore();
private:
    int mid_exam;
    int fin_exam;
};

Score::Score(int m, int f)
{
    mid_exam = m;
    fin_exam = f;
}


//赋值
```

```
类名 对象名[(实参表)]
Score op1(99, 100);
op1.showScore();
类名 *指针变量名 = new 类名[(实参表)]
Score *p;
p = new Score(99, 100);
p->showScore();
-----------------------
Score *p = new Score(99, 100);
p->showScore();
```

拷贝构造函数是一种特殊的构造函数，其形参是**本类对象的引用**。拷贝构造函数的作用是在建立一个新对象时，使用一个已存在的对象去初始化这个新对象。

- 因为拷贝构造函数也是一种构造函数，所以其函数名与类名相同，并且该函数也没有返回值。
- 拷贝构造函数只有一个参数，并且是同类对象的引用。
- 每个类都必须有一个拷贝构造函数。可以自己定义拷贝构造函数，用于按照需要初始化新对象；如果没有定义类的拷贝构造函数，系统就会自动生成一个**默认拷贝构造函数**，用于复制出与数据成员值完全相同的新对象。
- 调用拷贝构造函数的三种情况
  - 当用类的一个对象去初始化该类的另一个对象时；
  - 当函数的形参是类的对象，调用函数进行形参和实参结合时；
  - 当函数的返回值是对象，函数执行完成返回调用者时。

```
类名::类名(const 类名 &对象名)
{
    拷贝构造函数的函数体;
```

```
}

class Score{
public:
    Score(int m, int f);  //构造函数
    Score();
    Score(const Score &p);  //拷贝构造函数
    ~Score();               //析构函数
    void setScore(int m, int f);
    void showScore();
private:
    int mid_exam;
    int fin_exam;
};

Score::Score(int m, int f)
{
    mid_exam = m;
    fin_exam = f;
}

Score::Score(const Score &p)
{
    mid_exam = p.mid_exam;
    fin_exam = p.fin_exam;
}
```

调用拷贝构造函数的一般形式为：

```
    类名 对象2(对象1);
    类名 对象2 = 对象1;
Score sc1(98, 87);
Score sc2(sc1);    //调用拷贝构造函数
Score sc3 = sc2;   //调用拷贝构造函数
```

析构函数

1. 析构函数与构造函数名字相同，但它前面必须加一个波浪号（~）。
2. 析构函数没有参数和返回值，也不能被重载，因此只有一个。
3. 当撤销对象时，编译系统会自动调用析构函数。

```
class Score{
public:
    Score(int m = 0, int f = 0);
    virtual ~Score();          //析构函数是一个虚函数
private:
    int mid_exam;
    int fin_exam;
};

Score::Score(int m, int f) : mid_exam(m),
fin_exam(f)
{
    cout << "构造函数使用中..." << endl;
}

Score::~Score()
{
    cout << "析构函数使用中..." << endl;
}
```

- 如果定义了一个全局对象，则在程序流程离开其作用域时，调用该全局对象的析构函数。
- 如果一个对象定义在一个函数体内，则当这个函数被调用结束时，该对象应该被释放，析构函数被自动调用。

- 若一个对象是使用`new`运算符创建的，在使用`delete`运算符释放它时，`delete`会自动调用析构函数。

构造函数及析构函数调用次序：

在定义子类对象时，构造函数初始化的调用顺序如下：基类、子对象、子类。

析构函数则相反：子类（自身），子对象，基类。

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

class A{
public:
    A() {
        cout << "A类对象构造中..." << endl;
    }
    ~A() {
        cout << "析构A类对象..." << endl;
    }
};

class B : public A{
public:
    B() {
        cout << "B类对象构造中..." << endl;
    }
    ~B(){
        cout << "析构B类对象..." << endl;
    }
};
```

```
int main() {
    B b;
    return 0;
}

结果:
A类对象构造中...
B类对象构造中...
析构B类对象...
析构A类对象...
```

二、类内成员的访问控制，继承的访问控制（public、protected、private）；

基类中的成员	继承方式	基类在派生类中的访问属性
private	public|protected|private	不可直接访问
public	public|protected|private	public|protected|private
protected	public|protected|private	protected|protected|private

基类的成员可以有 public、protected、private3 中访问属性，基类的成员函数可以访问基类中其他成员，但是在类外通过基类的对象，就只能访问该基类的公有成员。同样，派生类的成员也可以有 public、protected、private3 种访问属性，派生类的成员函数可以访问派生类中自己增加的成员，但是在派生类外通过派生类的对象，就只能访问该派生类的公有成员。

比如 protected 经过 public 继承方式后仍然是 protected，可以把基类内的私有对象换成保护对象，这样就可以继承到下

派生类对基类成员的访问形式主要有以下两种：

- **内部访问**：由派生类中新增的成员函数对基类继承来的成员的访问。
- **对象访问**：在派生类外部，通过派生类的对象对从基类继承来的成员的访问。

链表及链表类的基本操作

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

class node
{
public:
    int data;
    node *next;
};

//建立链表
node* createlist(int n)
{
    node *temp, *head = NULL, *tail = NULL;
    int num;
    cin >> num;
    head = new node;
    if (head == NULL)
```

```
    {
        cout << "No memory available!";
        return NULL;
    }
    else
    {
        head->data = num;
        head->next = NULL;
        tail = head;
    }
    for (int i = 0; i < n - 1; i++)
    {
        cin >> num;
        temp = new node;
        if (temp == NULL)
        {
            cout << "No memory available!";
            return NULL;
        }
        else
        {
            temp->data = num;
            temp->next = NULL;
            tail->next = temp;
            tail = temp;
        }
    }
    return head;
}

//遍历链表
void outputList(node *head)
{
```

```
    cout << "List: ";
    node *curNode = head;
    while (curNode)
    {
        cout << curNode->data;
        if (curNode->next)
            cout << "->";
        curNode = curNode->next;
    }
    cout << endl;
}

//查找指定整数
node* findData(int n, node *head)
{
    node *curNode = head;
    while (curNode)
    {
        if (curNode->data == n)
        {
            cout << "Find " << n << " in the
list." << endl;
            return curNode;
        }
        curNode = curNode->next;
    }
    cout << "can't find " << n << " int the
list." << endl;
    return NULL;
}

//将输入的整数从小到大插入链表
node* insertData(int n, node *head)
```

```
{
    node *curNode = head;//插入点的后节点
    node *preNode = NULL;//插入点的前节点
    node *newNode = NULL;//新节点
    while ((curNode != NULL) && (curNode->data
< n))
    {
        preNode = curNode;
        curNode = curNode->next;
    }
    newNode = new node;
    if (newNode == NULL)
    {
        cout << "No memory available!";
        return head;
    }
    newNode->data = n;
    if (preNode == NULL)//插入到链表头
    {
        newNode->next = curNode;
        return newNode;
    }
    else
    {
        preNode->next = newNode;
        newNode->next = curNode;
        return head;
    }

}

//删除节点
node *deleteData(int n, node* head)
```

```
{
    node *curNode = head;//指向当前节点
    node *preNode = NULL;//指向前节点
    while (curNode && curNode->data != n)
    {
        preNode = curNode;//当前节点变前节点
        curNode = curNode->next;
    }
    if (curNode == NULL)
    {
        cout << "Can't find " << n << " in the
list" << endl;
        return head;
    }
    if (preNode == NULL)//删除首节点
        head = head->next;
    else
        preNode->next = curNode->next;

    delete curNode;
    return head;

}

int main()
{
    int n;
    node *ListHead = NULL;
    cout << "please enter the number of nodes:"
<< endl;
    cin >> n;
    if (n > 0)
        ListHead = createlist(n);
```

```
    outputList(ListHead);//遍历链表
    cin >> n;
    node *Findn;
    Findn = findData(n, ListHead);//查找整数n
    cin >> n;
    ListHead = insertData(n, ListHead);//从小到大插入整数n
    outputList(ListHead);//遍历链表

    system("pause");
    return 0;
}
```

子类的构造函数和析构函数及调用次序；

- 基类的析构函数必须为**虚函数**，这样通过多态调用的时候才能保证派生类的析构函数被调用；
- 指向子类的基类指针或者引用会覆盖基类对应虚函数的实现；
- 如果父类里声明了某函数为虚函数，则在子类此函数的声明里不管有没有 virtual 关键字，都是虚函数，即使访问权限发生变化；
- 如果基类和子类的函数名相同，参数也相同，那么在子类中无论加不加 override，都是 override 的。注意，如果一个函数有 const 修饰，另一个没有，则不是override。换句话说，除了函数体不同，其他相同则是 override；

```
#include<iostream>
using namespace std;
// 基类
class B {
public:
  B() { cout << "B0::B()" << endl; }
  B(int a) { cout << "B1::B()" <<" "<< "a=" <<
a << endl; }
  B(const B& b) {cout << "B2::B()" << endl;} //
拷贝构造
  ~B() { cout << "~B()" << endl; }
};
// 派生类
class D :public B {
public:
  D() { cout << "D0::D()" << endl; }
  D(int a) { cout << "D1::D()" <<" "<< "a=" <<
a << endl; }
  ~D() { cout << "~D()" << endl; }
};


int main() {
  B b;
  return 0;
}
// 先构造再析构
// B0::B()
// ~B()

int main() {
  D d;
  return 0;
```

```
}
// 基类构造-子类构造-子类析构-基类析构
// B0::B()
// D0::D()
// ~D()
// ~B()

int main() {
  B b = B();                //调用的是B()构造函数
  B b1 = B(1);              //调用的是B(int a)构造函数
  return 0;// 使用构造函数创建对象的顺序与使用析构函数释放对象的顺序相反
// 即后构造的先析构（栈：先进后出）
}
// B0::B()
// B1::B() a=1
// ~B()    b1
// ~B()    b


int main() {
  D d = D();        //只有一个D的对象，先调用B()，再调用D()
  D d1 = D(1);      //只有一个D的对象，先调用B()，再调用D(int a)
  return 0;
}
// B0::B()
// D0::D()
// B0::B()
// D1::D() a=1
// ~D()    d1
// ~B()    d1
```

```
// ~D()    d
// ~B()    d


int main() {
  B b = D();    //先调用B()，再调用D()，会调用B的拷贝构造函数
  return 0;
}
// B0::B()
// D0::D()
// B2::B()
// ~D() d
// ~B() d
// ~B() b


int main() {
  B b;
  D();
  return 0;
}
// B0::B()
// B0::B()
// D0::D()
// ~D()
// ~B()
// ~B()



int main() {
```

```
  B b = D(1);  //先调用B()，再调用D(int a)，D(int a)会调用B的拷贝构造函数
  return 0;
}
// B0::B()
// D1::D() a=1
// B2::B()
// ~D()
// ~B()
// ~B()


int main() {
  B* d = new B;  //调用B()
  delete d;      //有此行代码才会执行~B()
  return 0;
}
// B0::B()
// ~B()

int main() {
  B* d = new D;    //先调用B()，再调用D()
  delete d;
  return 0;
}
// B0::B()
// D0::D()
// ~B() // 注意：此行代码会调用~B()，但不会调用~D()，因为基类的析构函数未加上virtual，造成内存泄漏

// 注意：如果基类的析构函数加上virtual，则delete时基类和子类都会被释放：
// B0::B()
```

```
// D0::D()
// ~D()
// ~B()


int main() {
  D* d = new D;     //先调用B()，再调用D()
  delete d;
  return 0;
}

// B0::B()
// D0::D()
// ~D()
// ~B()

// 不管基类的析构函数是否是虚函数(即是否加virtual关键词)，delete时基类和子类都会被释放；
```

五、参数默认值、函数重载、函数模板、类模板；

带默认参数的构造函数

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

class Score{
public:
```

```
    Score(int m = 0, int f = 0);    //带默认参数
的构造函数
    void setScore(int m, int f);
    void showScore();
private:
    int mid_exam;
    int fin_exam;
};

Score::Score(int m, int f) : mid_exam(m),
fin_exam(f)
{
    cout << "构造函数使用中..." << endl;
}

void Score::setScore(int m, int f)
{
    mid_exam = m;
    fin_exam = f;
}

void Score::showScore()
{
    cout << "期中成绩: " << mid_exam << endl;
    cout << "期末成绩: " << fin_exam << endl;
}

int main()
{
    Score op1(99, 100);
    Score op2(88);
    Score op3;
    op1.showScore();
```

```
    op2.showScore();
    op3.showScore();
}
```

函数重载：这意味着，在同一作用域，只要函数参数的类型不同，或者参数的个数不同，或者二者兼而有之，两个或者两个以上的函数可以使用相同的函数名。

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

int add(int x, int y)
{
    return x + y;
}

double add(double x, double y)
{
    return x + y;
}

int add(int x, int y, int z)
{
    return x + y + z;
}

int main()
{
    int a = 3, b = 5, c = 7;
    double x = 10.334, y = 8.9003;
    cout << add(a, b) << endl;
    cout << add(x, y) << endl;
    cout << add(a, b, c) << endl;
```

```
    return 0;
}
```

```
//函数模板
#include <iostream>
using namespace std;

template <typename T>
T Max(T *array, int size = 0) {
    T max = array[0];
    for (int i = 1  ; i < size; i++) {
        if (array[i] > max) max = array[i];
    }
    return max;
}

int main() {
    int array_int[] = {783, 78, 234, 34, 90,
1};
    double array_double[] = {99.02, 21.9,
23.90, 12.89, 1.09, 34.9};
    int imax = Max(array_int, 6);
    double dmax = Max(array_double, 6);
    cout << "整型数组的最大值是: " << imax << endl;
    cout << "双精度型数组的最大值是: " << dmax <<
endl;
    return 0;
}

//函数模板的重载
#include <iostream>
using namespace std;
```

```
template <class Type>
Type Max(Type x, Type y) {
    return x > y ? x : y;
}

template <class Type>
Type Max(Type x, Type y, Type z) {
    Type t = x > y ? x : y;
    t = t > z ? t : z;
    return t;
}

int main() {
    cout << "33,66中最大值为 " << Max(33, 66) <<
endl;
    cout << "33,66,44中最大值为 " << Max(33, 66,
44) << endl;
    return 0;
}
```

运算符重载（常见运算符的重载，如=、==、[]、<<、>>、+、-、+=、-=、
前置++、后置++、--、取负）

(缺)

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

class Complex{
private:
    double real, imag;
public:
```

```
    Complex(double r = 0.0, double i = 0.0):
real(r), imag(i) { }
    friend Complex operator+(Complex& a,
Complex& b) {
        Complex temp;
        temp.real = a.real + b.real;
        temp.imag = a.imag + b.imag;
        return temp;
    }
    void display() {
        cout << real;
        if (imag > 0) cout << "+";
        if (imag != 0) cout << imag << "i" <<
endl;
    }
};

int main()
{
    Complex a(2.3, 4.6), b(3.6, 2.8), c;
    a.display();
    b.display();
    c = a + b;
    c.display();
    c = operator+(a, b);
    c.display();

    return 0;
}
```

7、类的模板（基本的模板定义）；

所谓类模板，实际上就是建立一个通用类，其数据成员、成员函数的返回类型和形参类型不具体指定，用一个虚拟的类型来代表。使用类模板定义对象时，系统会根据实参的类型来取代类模板中虚拟类型，从而实现不同类的功能。

```
template <typename T>
class Three{
private:
    T x, y, z;
public:
    Three(T a, T b, T c) {
        x = a; y = b; z = c;
    }
    T sum() {
        return x + y + z;
    }
}
```

八、多态（虚函数、纯虚函数、抽象类、虚析构函数、动态绑定）；

在基类中的某个成员函数被声明为虚函数后，此虚函数就可以在一个或多个派生类中被重新定义。虚函数在派生类中重新定义时，其函数原型，包括返回类型、函数名、参数个数、参数类型的顺序，都必须与基类中的原型完全相同。

虚函数的作用是**允许在派生类中重新定义与基类同名的函数，并且可以通过基类指针或引用来访问基类和派生类中的同名函数。**

```
#include <iostream>
```

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

class Family{
private:
    string flower;
public:
    Family(string name = "鲜花"): flower(name) {
}
    string getName() {
        return flower;
    }
    virtual void like() {
        cout << "家人喜欢不同的花: " << endl;
    }
};

class Mother: public Family{
public:
    Mother(string name = "月季"): Family(name) {
}
    void like() {
        cout << "妈妈喜欢" << getName() << endl;
    }
};

class Daughter: public Family{
public:
    Daughter(string name = "百合"): Family(name)
{ }
    void like() {
        cout << "女儿喜欢" << getName() << endl;
    }
```

```
};

int main() {
    Family *p;
    Family f;
    Mother mom;
    Daughter dau;
    p = &f;
    p->like();
    p = &mom;
    p->like();
    p = &dau;
    p->like();

    return 0;
}
```

```
家人喜欢不同的花:
妈妈喜欢月季
女儿喜欢百合
```

- 派生类必须从它的基类公有派生。
- 首先在基类中定义虚函数;
- 虚函数必须是其所在类的成员函数
- 构造函数不能是虚函数，但是析构函数是虚函数

九、基本文件操作（只要求文本文件的读写，读写指针不作要求）

在 C++ 中进行文件操作的一般步骤如下：

1. 为要进行操作的文件定义一个流对象。
2. 建立（或打开）文件。如果文件不存在，则建立该文件。如果磁盘上已存在该文件，则打开它。
3. 进行读写操作。在建立（或打开）的文件基础上执行所要求的输入 / 输出操作。
4. 关闭文件。当完成输入 / 输出操作时，应把已打开的文件关闭。

```
//把字符串 “I am a student.” 写入磁盘文件 text.txt 中。
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    ofstream fout("../test.txt", ios::out);
    if (!fout) {
        cout << "Cannot open output file." << endl;
        exit(1);
    }
    fout << "I am a student.";
    fout.close();

    return 0;
}
```

```
//把磁盘文件 test1.dat 中的内容读出并显示在屏幕上。
#include <iostream>
#include <fstream>
using namespace std;
```

```
int main() {
    ifstream fin("../test.txt", ios::in);
    if (!fin) {
        cout << "Cannot open output file." <<
endl;
        exit(1);
    }
    char str[80];
    fin.getline(str , 80);
    cout << str <<endl;
    fin.close();

    return 0;
}
```

十、异常处理（难度参考教材例题，try\catch\throw 组合使用）

抛出（Throw）--> 检测（Try） --> 捕获（Catch）

```
try{
    // 可能抛出异常的语句
}catch(exceptionType variable/*异常信息类型[变量
名]*/){
    // 处理异常的语句
}
```

`try`和`catch`都是 C++ 中的关键字，后跟语句块，不能省略`{}`。try 中包含可能会抛出异常的语句，一旦有异常抛出就会被后面的 catch 捕获。从 try 的意思可以看出，它只是“检测”语句块有没有异常，如果没有发生异常，它就“检测”不到。catch 是“抓住”的意思，用来捕获并处理 try 检测到的异常；

如果 try 语句块没有检测到异常（没有异常抛出），那么就不会执行 catch 中的语句。

这就好比，catch 告诉 try：你去检测一下程序有没有错误，有错误的话就告诉我，我来处理，没有的话就不要理我！可以一个 try 多个 catch。

throw 后立即离开本级函数。

十一、

const

```
class Date{
private:
    int year;
    int month;
    int day;
public:
    Date(int y, int m, int d) : year(y),
month(m), day(d){

    }
    void showDate();
    void showDate() const;
};
//常对象函数不能更新对象的数据成员，也不能调用该类的普通成员函数，这就保证了在常成员函数中绝不会更新数据成员的值。
```

引用

```
类型 &引用名 = 已定义的变量名
//引用与其所代表的变量共享同一内存单元，系统并不为引用另外分配存储空间。实际上，编译系统使引用和其代表的变量具有相同的地址。
#include <iostream>
using namespace std;
int main()
{
    int i = 10;
    int &j = i;
    cout << "i = " << i << " j = " << j << endl;
    cout << "i的地址为 " << &i << endl;
    cout << "j的地址为 " << &j << endl;
    return 0;
}
//输出 i 和 j 的值相同，地址也相同。
/*
引用并不是一种独立的数据类型，它必须与某一种类型的变量相联系。在声明引用时，必须立即对它进行初始化，不能声明完成后再赋值。
为引用提供的初始值，可以是一个变量或者另一个引用。
指针是通过地址间接访问某个变量，而引用则是通过别名直接访问某个变量。
*/
//引用作为函数参数、使用引用返回函数值
#include <iostream>
using namespace std;

void swap(int &a, int &b)
{
```

```
    int t = a;
    a = b;
    b = t;
}

int a[] = {1, 3, 5, 7, 9};

int& index(int i)
{
    return a[i];
}

int main()
{
    int a = 5, b = 10;
    //交换数字a和b
    swap(a, b);
    cout << "a = " << a << " b = " << b <<
endl;
    cout << index(2) << endl;    //等价于输出元素
a[2]的值
    index(2) = 100;              //等价于将a[2]的
值赋为100;
    cout << index(2) << endl;

    return 0;
}
```

指针

```
void Sample::copy(Sample& xy)
{
    if (this == &xy) return;
    *this = xy;
}
//this指针保存当前对象的地址，称为自引用指针。
```

对象指针和对象数组

```
类名 数组名[下标表达式]
//用只有一个参数的构造函数给对象数组赋值
Exam ob[4] = {89, 97, 79, 88};
//用不带参数和带一个参数的构造函数给对象数组赋值
Exam ob[4] = {89, 90};
//用带有多个参数的构造函数给对象数组赋值
Score rec[3] = {Score(33, 99), Score(87, 78),
Score(99, 100)};

Score score;
Score *p;
p = &score;
p->成员函数();

Score score[2];
score[0].setScore(90, 99);
score[1].setScore(67, 89);

Score *p;
p = score;   //将对象score的地址赋值给p
p->showScore();
p++;    //对象指针变量加1
```

```
p->showSccore();

Score *q;
q =&score[1];    //将第二个数组元素的地址赋值给对象指针变量q
```

传参数

```
class Point{
public:
    int x;
    int y;
    Point(int x1, int y1) : x(x1), y(y1)  //成员初始化列表
    { }
    int getDistance()
    {
        return x * x + y * y;
    }
};

void changePoint1(Point point)    //使用对象作为函数参数，但不影响调用该函数的对象（实参本身）。
{
    point.x += 1;
    point.y -= 1;
}

void changePoint2(Point *point)    //使用对象指针作为函数参数，可以影响实际参数的值
{
    point->x += 1;
    point->y -= 1;
```

```
}

void changePoint3(Point &point)  //使用对象引用作为函数参数，可以影响参数的值，好用
{
    point.x += 1;
    point.y -= 1;
}


int main()
{
    Point point[3] = {Point(1, 1), Point(2, 2), Point(3, 3)};
    Point *p = point;
    changePoint1(*p);
    cout << "the distance is " << p[0].getDistance() << endl;
    p++;
    changePoint2(p);
    cout << "the distance is " << p->getDistance() << endl;
    changePoint3(point[2]);
    cout << "the distance is " << point[2].getDistance() << endl;

    return 0;
}
```

指针数组

指向字符串的指针

对象的动态建立与释放、类成员数据的动态建立与释放。

```
指针变量名 = new 类型
    int *p;
    p = new int;
delete 指针变量名
    delete p;
// new 分配的空间，使用结束后应该用也只能用 delete 显式地释放


指针变量名 = new 类型名[下标表达式];
int *p = new int[10];
delete []指针变量名;
delete p;

//new 可在为简单变量分配空间的同时，进行初始化
指针变量名 = new 类型名(初值);
int *p;
p = new int(99);
...
delete p;
```