

第六讲 指针

— 指针

— 持久动态内存分配

潘建瑜@MATH.ECNU

目录页

Contents

1 指针

2 持久动态内存分配

指针

- 指针的定义与运算
- 指针与数组（一维、二维）
- 指针数组与行指针
- 指针与函数

指针

指针变量，简称 **指针**，用来存放其它变量在内存中的 **地址**

指针的定义

类型说明符 * 指针变量名

- 声明一个指针类型的变量，即指针
- 类型说明符 表示该指针 可指向的对象(即目标对象) 的数据类型，
  即指定该指针能存放哪类数据的地址

内存空间的访问方式：1) 变量名；2) 内存地址，即指针。

指针运算

两个指针基本运算

- 提取变量的内存地址：&
- 引用指针所指向的变量 (即目标对象)：*

- 地址运算符：&

&变量名　　// 提取变量在内存中的存放地址

```
int x=3;
int * px;   // 声明指针 px
px = &x;    // 将 x 的地址赋给指针 px，简称 px 指向 x，称 x 为 px 的目标对象
```

注意　指针的类型必须与其指向的对象的类型一致!

指针运算

- 指针运算符：*

*指针变量　// 引用指针的目标对象

```
int x;
int *px; // 声明指针变量
px = &x;
*px = 3;  // 等价于 x = 3，星号后面不能有空格！
```

指针提供了一种访问变量的高效方式，在使用指针时，通常关心的是指针所指向的对象

```
int x = 3;
int * px = &x; // 初始化，指针的值只能是相应类型的变量的地址
```

空指针

void 类型的指针

void * 指针名

- **void** 类型的指针可以存放任何类型数据的地址，即指向任何类型的对象
- 不允许直接通过 **void** 指针来访问其目标对象！
- 若需访问 **void** 类型指针的目标对象，必须通过显式类型转换指明目标对象的类型

```
int x;
void * pv;
pv = &x; // OK, void 型指针指向整型变量
(int *)pv = 3; // OK，使用 void 型指针时需要强制类型转换
```

指针的功能不仅仅是存储内存地址, 它还包含目标对象的更多信息, 如类型、等级等.

指针赋值

指针可能的取值

(1) 空指针：0，NULL 或值为 0，NULL 的常量；

C++11 标准中引入了 nullptr，用于表示空指针，增强了类型安全性。

(2) 类型匹配的对象的地址

(3) 同类型的另一有效指针

(4) 类型匹配的对象的前、后地址（相对位置）

> 没有初始化或赋值的指针是 **无效的指针**（也称 **野指针**），引用无效指针会带来难以预料的后果！

```
int x[3] = {1,2,3};
int * px = &x[0];
int * py = &x[0] + 1;
```

```
int * pz;
pz = nullptr;  // OK
```

指针与常量

- 指向常量的指针

const 类型说明符 * 指针名

```
const int a = 3;
int * pa = &a;  // ERROR, 普通指针不能指向常量
const int * cpa = &a; // OK
```

- 指向 const 对象（常量）的指针必须用 const 声明!
- 这里的 const 限定了指针所指对象的属性

ex06_pointer_const.cpp

```
const int a = 3;
int b = 5;
const int * cpa = &a; // OK
*cpa = 5; // ERROR
cpa = &b; // OK
*cpa = 9; // ERROR
b = 9; // OK
```

- 允许把非 const 对象的地址赋给指向 const 的指针
- 但不允许使用指向 const 的指针来修改其目标对象的值

指针与常量

- 常量指针，简称 **常指针**

  类型说明符 * const 指针名

  - 常量指针：指针本身的值不能修改

```
int a = 3, b = 5;
int * const pa = &a;  // OK
pa = &b;  // ERROR
```

- 指向常量的常量指针

  const 类型说明符 * const 指针名

  - 指针本身的值不能修改，也不能通过它来修改其指向的对象

指针算术运算

指针可以和整数或整型变量进行加减运算，运算规则与指针的类型有关！

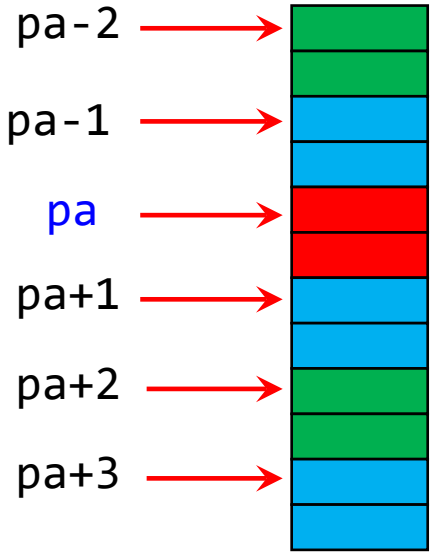
```
int * pa; int k;
pa + k   // pa 所指的当前位置之后第 k 个元素的地址
pa - k   // pa 所指的当前位置之前第 k 个元素的地址
```

指针的算术运算通常是与数组的使用相联系的

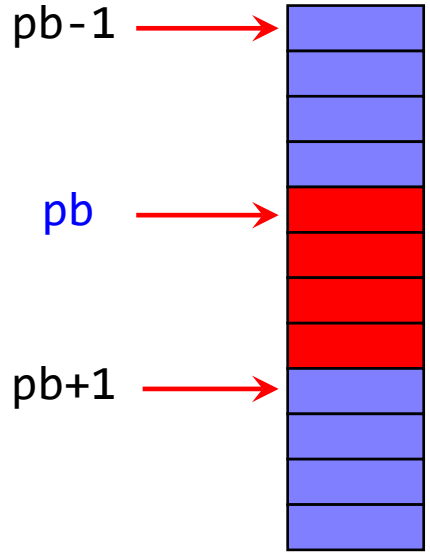
```
int x[5] = {0,1,2,3,4};
int * px = &x[0]; // px 指向 x[0]
cout << *px << endl;       // 输出 x[0] 的值
cout << *(px+1) << endl;   // 输出的 x[1] 值
px = px + 2;   // px 改为指向 x[2]
```

指针算术运算

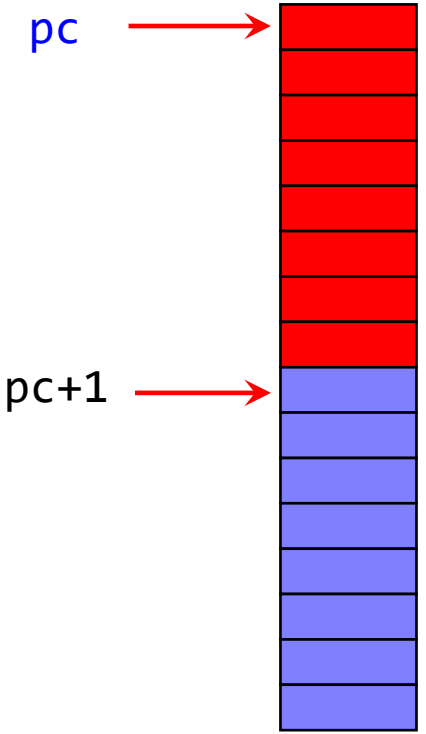
```
short * pa
    // 每个元素占 2 个字节
```




```
int * pb
    // 每个元素占 4 个字节
```




```
double * pc
    // 每个元素占 8 个字节
```



指针数组

- 指针数组：数组的元素都是指针变量

类型说明符 * 指针数组名[n]

ex06_pointer_02.cpp

```
int A[3][4] = {{0,2,4,8}, {1,3,5,7}, {2,3,5,8}};
int *pa[3] = {A[0],A[1],A[2]}; // 声明一个有三个元素的指针数组
                               // 并初始化：pa[0]=A[0]，pa[1]=A[1]，pa[2]=A[2]
```

需要指出的是，这里的 pa 是数组名，不是指针

指针与一维数组

数组元素在内存中是连续存放的，因此使用指针可以非常方便地处理数组元素！

```
int a[] = {0,2,4,8};
int * pa = a; // OK, 数组名代表数组的首地址，即 pa = &a[0]
*pa = 3; // OK，等价于 a[0] = 3
*(pa+2) = 5;  // OK，等价于 a[2] = 5
```

指针与一维数组

访问一维数组的四种方式

(1) 数组名+下标，如：a[0]
(2) 数组名+指针运算，如：*(a+1)
(3) 指针+指针运算，如：int * pa = a; *pa
(4) 指针+下标，如：int * pa = a; pa[0]

若 pa 是指针，k 是整型数值，则
*(pa+k) 可以写成 pa[k]
*(pa-k) 可以写成 pa[-k]

ex06_pointer_array_01.cpp

```
// 第一种方式：数组名+下标
for (int i = 0; i < n; i++)
    cout << a[i] << "," ;
```

```
// 第二种方式：数组名+指针运算
for (int i = 0; i < n; i++)
    cout << *(a+i) << "," ;
```

```
// 第三种方式：指针+指针运算
for (int* pa = a; pa < a + n; pa++)
    cout << *pa << "," ;
```

```
// 第四种方式：指针+下标
for (int* pa = a, i = 0; i < n; i++)
    cout << pa[i] << "," ;
```

指针与一维数组名

一维数组名等同于指向数组第一个元素的 **常指针**!

```
*(a+2) = 5;  // OK, 等价于 a[2] = 5
```

ex06_pointer_array_02.cpp

```
*(pa++) = 3;   // OK, 等价于 a[0] = 3; pa = pa+1;
*(a++) = 3;   // ERROR! a 是数组名, 等效于常指针!
```

小结

一维数组 a[n] 与指针 pa = a

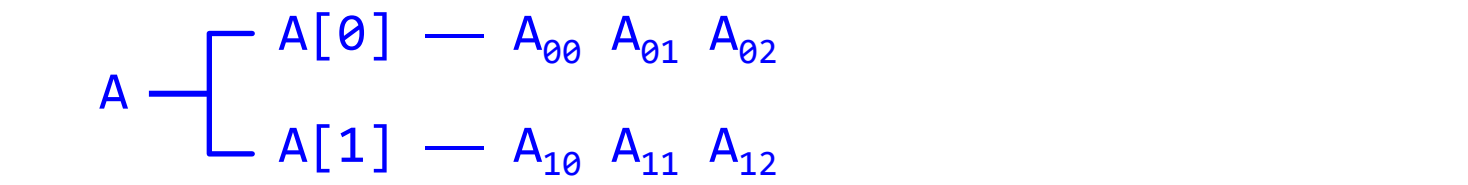
- 数组名 a 是地址常量，数组名 a 与 &a[0] 等价；
- a+i 是 a[i] 的地址，a[i] 与 *(a+i) 等价；
- 数组元素的下标访问方式也是按地址进行的；
- 可以通过指针 pa 访问数组的任何元素，且更加灵活，比如形参；
- pa++ 或 ++pa 合法，但 a++ 不合法；
- *(pa+i) 与 pa[i] 等价；

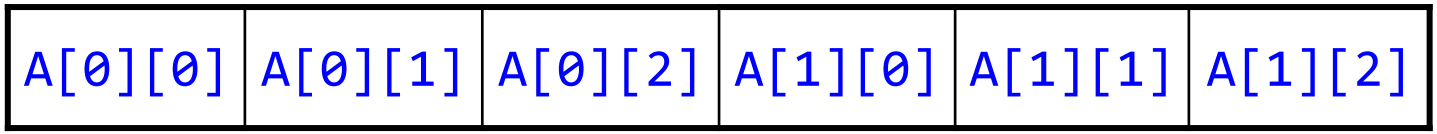
```
a[i]  <=> pa[i]  <=> *(pa+i)  <=> *(a+i)
```

指针与二维数组

二维数组 **按行** 顺序存放，是由一维数组组成的一维数组

例：`int A[2][3]={{1,2,3},{7,8,9}};`





`A[0]`（第一行的首地址）　　`A[1]`（第二行的首地址）

`A[0]`, `A[1]` 有时也称为 行数组

指针与二维数组

```
int A[2][3] = {{1,2,3},{7,8,9}};
int * pa = A[0];  // 行数组A[0]的首地址, 等价于 pa = &A[0][0]
int * pa1 = A[1]; // 行数组A[1]的首地址，即 &A[1][0]
*pa = 11;      // 等价于 A[0][0] = 11
*(pa+2) = 12; // 等价于 A[0][2] = 12
*(pa+4) = 13; // 等价于 A[1][1] = 13
```

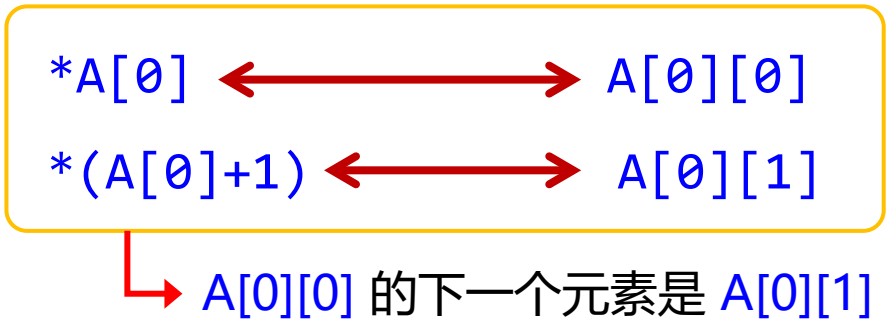
设指针 p=&A[0][0]，则 A[i][j] <==> *(p+n*i+j)

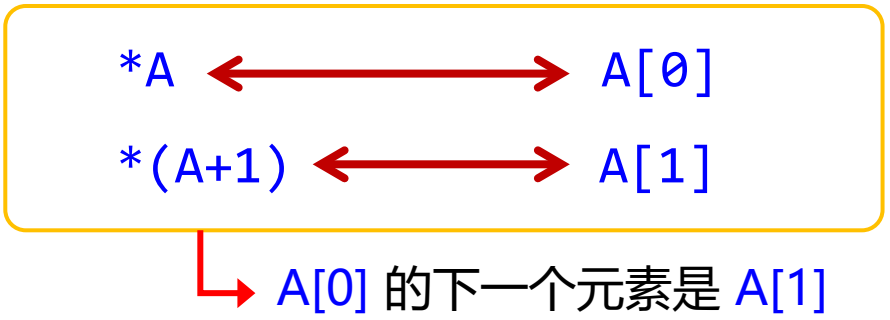
注：n 是列数

ex06_pointer_array_2D.cpp

二维数组名

- 对于二维数组 A，虽然 A、A[0] 都是代表数组首地址，但二者指向的对象不同：
  - A[0] 是一维数组的名字，它指向的是行数组 A[0] 的首元素，即 A[0][0]
  - 而 A 是一个二维数组的名字，它指向的是它的首元素，即 A[0]，即行数组






```
int A[2][3]={{1,2,3},{7,8,9}};
int * p = A; // ERROR!
int * p = A[0]; // OK
```

一维数组名等同于普通指针，
但二维数组名与普通指针不等同！

指针与引用

引用与指针

- 引用是变量的别名，只针对变量，函数没有引用
- 引用必须初始化，且不能修改
- 用引用能实现的功能，用指针都能实现
- 传递大量数据时（数组），建议使用指针

```
int a = 3;
int * pa = &a; // 指针
int & ra = a;  // 引用
```

引用作为函数参数的优点

- 效果与指针类似，但可读性强；
- 函数调用简单、安全；

C++ 有引用，C 语言没有引用

行指针

- 行指针：指向行数组的指针

  类型说明符 (* 指针名)[n]

  - 该指针以长度为 n 的一维数组为基本单位

```
int A[2][3]={{1,2,3},{7,8,9}};
int (*p)[3]; // 声明行指针，注意 n 与列数相等
p = A; // OK
```

ex06_pointer_row.cpp

一维数组名等同于普通指针，二维数组名等同于行指针

指针作为函数的参数

- 以 **地址方式** 传递数据，形参和实参指向共同的内存地址，是同一个数据
- 形参是指针时，实参可以是指针或地址
- 可以传递函数代码的首地址，即将函数作为参数

```
void split(double x, int * n, double * f)
```

```
split(x, &x1, &x2)
```

ex06_pointer_arguments.cpp

函数间需要传递大量数据（数组）时，通常是利用指针传递数据的首地址

如果被调函数不需要修改实参的值，建议使用 **指向常量的指针**

指针型函数

当函数的返回值是 地址/指针 时，该函数就是 指针型函数

指针型函数的定义

```
数据说明符 * 函数名(形参列表)
{
   函数体
}
```

注：不要返回局部变量的地址，或指向局部变量的指针。

指向函数的指针

- 在程序运行过程中，不仅数据要占用内存空间，函数也要占用一定的内存空间
- 函数名就代表函数在内存空间中的首地址
- 用来存放这个地址的指针就是指向该函数的指针

函数指针的定义

类型说明符 (* 函数指针名)(形参列表)

- 这里的 类型说明符 和 形参列表 应与其指向的函数相同
- 可以象使用函数名一样使用函数指针

函数名除了表示函数的首地址外，还包括函数的返回值类型，形参个数、类型、顺序等信息

指向函数的指针

ex06_pointer_fun.cpp

```
int Gcd(int x, int y);
int Lcm(int x, int y);

int (* pf)(int, int); // 声明函数指针

pf = Gcd; // pf 指向函数 Gcd
cout << "最大公约数: " << pf(a,b) << endl;

pf = Lcm; // pf 指向函数 Lcm
cout << "最小公倍数: " << pf(a,b) << endl;
```

持久动态内存分配

- 动态内存申请 --- new
- 动态内存释放 --- delete
- 动态数组的申请与释放
- 智能指针

持久动态内存分配

在程序运行过程中，可以根据实际需要 动态分配 内存空间

- 持久动态内存分配相关函数
  - 申请内存空间：**new**
  - 释放内存空间：**delete**

申请和释放单个存储单元

- 申请单个存储单元

```
px = new 类型说明符;
```

```
px = new 类型说明符(初始值);
```

  - 申请用于存放指定数据类型数据的内存空间
  - 若申请成功，则返回该内存空间的地址，否则返回 0 或 NULL

- 持久动态内存的释放

```
delete px; // 释放由 new 申请的内存空间
```

  - px 必须是由 new 操作的返回值
  - 持久动态内存申请后必须手工释放，否则可能会造成内存泄漏

申请和释放动态数组

- 创建一维动态数组

```
px = new 类型说明符[数组长度];
```

```
px = new 类型说明符[数组长度](); // 赋初值 0
```

```
px = new 类型说明符[数组长度]{初值列表}; // C++11
```

ex06_new_01.cpp

- 创建多维动态数组

```
px = new 类型说明符[n1][n2]...
```

- 动态数组的释放

```
delete[] px; // 释放由 new 申请的动态存储空间
```

持久动态内存举例：素数

例：给定一个正整数 N，求出 N 个最小的素数

```
int main()
{
   int N, i, flag, k = 0, n = 2;
   cout << "Input N: ";  cin >> N;
   int * pa = new int[N]; // 申请内存空间，长度可根据需要改变
   while(k < N)    // 寻找前 N 个素数
   {
      flag = 0;
      for(i = 2; i <= sqrt(n); i++)
          if (n%i == 0) {flag = 1; break;}
      if (flag == 0) { pa[k] = n; k++; }
      n++;
    }
}
```

```
for(i = 0; i < k && pa[i] <= sqrt(n); i++)
   if (n%pa[i] == 0) {flag = 1; break;}
```

持久动态内存举例：素数

例：将前面的例子写成函数形式

ex06_new_02.cpp

```
int * find_Nprime(int N)
{
   int * pa = new int[N]; // 申请动态数组

   int i, flag, k = 0, n = 2;
   while (k < N)
   {
      flag = 0;
      for(i = 0; i < k && pa[i] <= sqrt(n); i++)
         if (n%pa[i] == 0) {flag = 1; break;}
      if (flag==0) { pa[k] = n; k++; }
      n++;
   }
   return pa;
}
```

```
int main()
{
   int N;
   cout << "Input N: ";  cin >> N;

   int *pa = find_Nprime(N);

   cout << "前" << N << "个素数为: \n";
   for(int i = 0; i < N; i++)
        cout << *(pa+i) << " ";

   delete[] pa;  // 释放动态数组 pa

   return 0;
}
```

持久动态内存举例：矩阵

ex06_new_03_matrix.cpp

```
int main()
{
    int i, j, n = 5;

    int *pa = new int[n*n];  // 建议使用一维数组，方便灵活

    for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
        *(pa+i*n+j) = i+j+1;

    for(i = 0; i < n; i++)
    {  for(j = 0; j < n; j++)
        cout << setw(6) << *(pa+i*n+j);
    cout << endl;
    }

    delete[] pa;
}
```

持久动态内存举例：矩阵

ex06_new_04_matrix.cpp

```
void set_matrix(int * a, int n);
void print_matrix(int * pa, int n);

int main()
{
    int n = 5;
    int * pa = new int[n*n]; // 申请存储空间

    set_matrix(pa, n);
    print_matrix(pa, n);

    delete[] pa; // 释放存储空间
    return 0;
}
```

```
void set_matrix(int * pa, int n)
{
    int i, j;
    for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
            *(pa+i*n+j) = i+j+1;
}
```

持久动态内存举例：矩阵

ex06_new_05_matrix.cpp

```
int * set_matrix(int n);
void print_matrix(int * pa, int n);

int main()
{
    int n = 5;

    int * pa = set_matrix(pa, n);
    print_matrix(pa, n);

    delete[] pa; // 释放存储空间
    return 0;
}
```

```
int * set_matrix(int n)
{   int i, j;
    pa = new int[n*n];
    for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
            *(pa+i*n+j) = i+j+1;

    return pa;
}
```

#include <memory>

智能指针

- 智能指针是 C++ 中用于自动管理内存的一种工具，它通过封装原始指针，确保在适当的时候释放内存，从而避免内存泄漏，主要用于持久动态内存分配。
- 智能指针包括：unique_ptr，shared_ptr，weak_ptr，auto_ptr (已弃用)

unique_ptr

ex06_pointer_unique.cpp

- 独占资源所有权，同一时间只能有一个 unique_ptr 指向特定内存
- 离开 unique_ptr 对象的作用域时，会自动释放资源
- 不能将 unique_ptr 直接赋值给其他指针，但可以转移，转移后不再拥有所有权

智能指针

shared_ptr

- 共享内存资源的所有权，多个 shared_ptr 可以指向同一内存，当最后一个 shared_ptr 失效时内存资源被自动释放
- 可以理解为对内存资源做引用计数, 当引用计数为 0 时, 自动释放资源
- C++ 20 开始，shared_ptr 支持指向数组

ex06_pointer_shared.cpp

智能指针

week_ptr

- 与 shared_ptr 一起使用， 一个 weak_ptr 对象看做是 shared_ptr 管理的资源的观察者，它不影响共享资源的生命周期。
- 需要使用 weak_ptr 正在观察的资源，可以将 weak_ptr 提升为 shared_ptr。
- 当 shared_ptr 管理的资源被释放时，weak_ptr 会自动变成 nullptr

第六讲上机作业

1、编写函数，交换两个双精度变量的值，分别用引用和指针实现。

函数分别为 swap_ref 和 swap_pointer，并在主函数中定义两个双精度变量，从键盘接受输入，并将交换后的值在屏幕上输出。

（程序取名 **hw06_01.cpp**）

```
void swap_ref(double & ra, double & rb);
void swap_pointer(double * pa, double * pb);
```

2、给定两个一维数组 a 和 b，其中 a 中的数据是无序的，而 b 中的数据按升序排列试统计 a 的所有元素中，大于 b 的第 k 个元素且小于第 k+1 个元素的数据个数。其中

a=[98,12,34,71,43,54,28,33,65,56], b=[10,30,50,80,100]

要求将结果存放在数组 c 中，其中 c[k] 表示数组 a 中大于 b[k] 而小于 b[k+1] 的元素个数。

（程序取名 **hw06_02.cpp**）

第六讲上机作业

3、有 17 人围成一圈，编号 1～17，从 1 号开始报数，报到 3 的倍数的人离开，一直数下去，直到最后只剩一人，求此人编号。（程序取名 **hw06_03.cpp**）

4、二进制转十进制。编写函数，将一个用字符串表示的二进制数转化为十进制数，如“10001”所对应的十进制数为17，在主函数中用“**1100110011001100**”测试。（程序取名 **hw06_04.cpp**）

```
int bin2dec(const char * const str);
```

提示：将一个字符转化成数字，可借助字符加减运算（推荐）或字符串函数。

5、字符易位破译。 编写函数，测试两个字符串是否字符异位相等，即两个字符串中包含的字母是相同的，但次序可以不同，如 "**silent**" 和 "**listen**" 是字符异位相等，但 "**baac**" 与 "**abcc**" 不是。（程序取名 **hw06_05.cpp**）

```
bool isAnagram(const char * const str1,
               const char * const str2);
```

提示：可以先对字符串进行排序，然后再比较。

第六讲上机作业

6、编写函数，实现两个有序（升序）数组的快速合并，使得合并后的数组仍然有序，并在主函数测试：

取 x=[3,5,8,11,23,39], y=[1,5,12,13,18,41,58,68]（程序取名 **hw06_06.cpp**）

```
void vec_merge(int * px, int m, int * py, int n, int * pz);
```

注意：m 和 n 分别是数组 x 和 y 的长度，合并后的数组存放在 z 中。

7、找出前 100 个素数，存放在数组 p 中，并分别使用下列方式在屏幕上输出 p，每行输出 10 个。

方式一：数组名+下标运算；（程序取名 **hw06_07.cpp**）

方式二：数组名+指针运算；

方式三：指针+指针运算

第六讲上机作业

8、编写函数，计算两个矩阵的乘积 `Z=X*Y`，其中 $X \in \mathbf{R}^{m\times p}$，$Y \in \mathbf{R}^{p\times n}$，$Z \in \mathbf{R}^{m\times n}$。

要求对任意正整数 `m, p, n` 都能实现矩阵相乘，在主函数中以 `m=3, n=4, p=2` 为例，生成两个随机矩阵 `X, Y`，元素为 `0` 到 `9` 之间的正整数，计算 `Z`，并将输出。（程序取名 **hw06_08.cpp**）

```
void matrix_prod(double * px, double * py, double * pz,
                 int m, int p, int n);
```

提示：这里 `px, py, pz` 分别指向 `X[0][0], Y[0][0]` 和 `Z[0][0]`

9、（可选题）

编写函数，实现矩阵乘积的 **Strassen** 算法，在主函数中生成两个 n 阶随机整数矩阵进行测试，比较与普通矩阵乘积的计算效率（消耗的时间），分别取：$n = 2^{10}, 2^{12}, 2^{14}$。

（程序取名 **hw06_09.cpp**）

```
void matrix_prod_strassen(double * px, double * py, double * pz, int n);
```

第六讲上机作业

10、（可选题）

编写函数, 实现求解线性方程组的 **Gauss** 消去法, 并在主函数中进行测试:

求解线性方程组 $\boldsymbol{Hx} = \boldsymbol{b}$, 其中 是 8 阶 **Hilbert** 矩阵，$\boldsymbol{b} = [1,1, ..., 1]^T$。（程序取名 **hw06_10.cpp**）

```
void GE(double* pa, double* b, double* x, int n)
```

提示：这里 pa 是指向 A[0][0] 的指针