

面向对象编程基础

本课程入选教育部产学合作协同育人项目

课程主页:<http://cpp.njuer.org>

课程老师：陈明 <http://cv.mchen.org>

面向对象编程基础

构造数据类型

本课程入选教育部产学合作协同育人项目

课程主页:<http://cpp.njuer.org>

课程老师：陈明 <http://cv.mchen.org>

本章内容

- 构造数据类型概述
- 枚举类型
- 数组类型
- 结构类型
- 联合类型
- 指针类型
- 引用类型

构造数据类型

- 有些数据不适合用基本数据类型来表示。
- 语言往往提供了由基本数据类型来构造新类型的手段。
- 构造数据类型属于用户自定义数据类型。

枚举类型

- 如何描述一个星期的每一天这样的数据？如果用int来描述，将会面临：
 - 1表示什么意思？
 - 星期天用什么整数表示？ 0还是7？
 - 如果用0~6表示一个星期的每一天，则对于一个取值为一个星期某一天的int型变量 `day`，如何防止下面的逻辑错误：
 - `day = 10`
 - `day = day*2`
- 在C++中用枚举类型来解决上面的问题。

枚举类型的定义

- 枚举类型是由用户自定义的一种简单数据类型。在定义一个枚举类型时，需要列出其值集中的每个值--枚举值。
- 枚举类型的定义格式为：
 - `enum <枚举类型名> {<枚举值表>;`
 - <枚举值表>为用逗号隔开的若干个整型符号常量。

例如：

- `enum Day {SUN,MON,TUE,WED,THU,FRI,SAT};`
- `enum Color {RED,GREEN,BLUE};`
- `enum Month {JAN,FEB,MAR,APR,MAY,JUN,JUL,
AUG,SEP,OCT,NOV,DEC};`

- 默认情况下，第一个枚举值为0，其它的值为首一个值加1。
- 在定义枚举类型时，也可显式地给枚举值指定值。例如：
 - `enum Day {SUN=7,MON=1,TUE,WED,THU,FRI,SAT};`
 - TUE为2, ...
- `bool`类型可看成是C++语言提供的一个预定义的枚举类型：
 - `enum bool { false, true };`

枚举类型变量的定义

- 先定义枚举类型，再定义枚举类型变量：
 - `enum Day {SUN,MON,TUE,WED,THU,FRI,SAT};`
 - `Day d1;`或
 - `enum Day d1; //C语言的写法`
- 枚举类型和枚举类型变量同时定义：
 - `enum Day {SUN,MON,TUE,WED,THU,FRI,SAT} d1;`或
 - `enum {SUN,MON,TUE,WED,THU,FRI,SAT} d1;`

枚举类型的运算

● 赋值

- 一个枚举类型的变量只能在相应枚举类型的值集中取值。例如：
 - `Day day;`
 - `day = SUN; //OK`
 - `day = 1; //Error`
 - `day = RED; //Error`
- 相同枚举类型之间可以进行赋值操作，例如：
 - `Day d1,d2;`
 - `d2 = d1;`
- 可以把一个枚举值赋值给一个整型变量例如：
 - `int a;`
 - `a = d1; //OK, 将进行类型转换`
- 但不能把一个整型值赋值给枚举类型的变量，
 - `d1 = a; //Error`
 - `d1 = (Day)a; //OK, 但不安全！`

●比较

- 枚举值之间的比较为枚举值所对应的整数之间的比较。

例：

```
MON < TUE (结果为true)
```

●算术运算

- 运算时，枚举值将转换成对应的整型值。
- 对枚举类型进行算术运算的结果类型为算术类型。例如：

```
Day d; int i;
```

```
.....
```

```
i = d+1; //OK
```

```
d = d+1; //Error, 因为d+1的结果为int类型。
```

```
d = (Day)(d+1) //OK
```

- 不能对枚举类型的值直接进行输入，但可以进行输出。例如：

```
Day d;
```

```
cin >> d; //Error
```

```
cout << d; //OK, 将把d转换成int
```

枚举类型输入/输出举例

```
#include <iostream>
using namespace std;
int main()
{ Day d;
  int i;
  cin >> i;
  switch (i)
  {
    case 0: d = SUN;      break;
    case 1: d = MON;      break;
    case 2: d = TUE;      break;
    case 3: d = WED;      break;
    case 4: d = THU;      break;
    case 5: d = FRI;      break;
    case 6: d = SAT;      break;
    default: cout << "Input Error!" << endl; exit(-1);
  }
}
```

.....

switch (d)

```
{    case SUN:    cout << "SUN" << endl;    break;
    case MON:    cout << "MON" << endl;    break;
    case TUE:    cout << "TUE" << endl;    break;
    case WED:    cout << "WED" << endl;    break;
    case THU:    cout << "THU" << endl;    break;
    case FRI:    cout << "FRI" << endl;    break;
    case SAT:    cout << "SAT" << endl;    break;
```

```
}
```

```
return 0;
```

```
}
```

数组类型

- 如何表示一个向量和矩阵这样的复合数据？如果用独立的变量来分别表示它们的元素，则会面临：
 - 变量数量太多（ x_1 、 x_2 、 x_3 、....）。
 - 变量之间缺乏显式的联系。
- C++提供了数组类型来表示上述的数据：
 - 数组类型是一种由**固定**多个**同类型**的元素按一定次序所构成的复合数据类型。
 - 数组类型是一种用户自定义的数据类型。
- 数组类型可分为：
 - 一维数组：表示向量和线性表等
 - 二维数组：表示矩阵等
 - 多维数组（三维及三维以上）

一维数组

- 一维数组用于表示由**固定多个**同类型的具有线性次序关系的数据所构成的复合数据类型。例如：
 - 向量
 - 某门课程的成绩表
 - 学生的姓名表
 -

一维数组类型定义

- 一维数组类型定义格式为：

typedef <元素类型> <一维数组类型名>[<元素个数>;

- <元素类型>为任意C++类型（void除外）
 - <元素个数>为整型常量表达式
- 例如：

```
typedef int A[10]; //由10个int型元素所构成的  
                  //一维数组类型
```

一维数组类型变量定义

- 一维数组类型变量定义格式为：
 - <一维数组类型名> <一维数组变量名>;或
 - <元素类型> <一维数组变量名>[<元素个数>;
 - <元素类型>为任意C++类型（void除外）
 - <元素个数>为整型常量表达式
- 例如：
 - `typedef int A[10];`
 - `A a;` //由10个int型元素所构成的数组。或
 - `int a[10];` //由10个int型元素所构成的数组。

一维数组变量的初始化

- 用一对花括号把元素的初始值括起来。例如：
 - `int a[10]={1,2,3,4,5,6,7,8,9,10};`
- 初始化表中的值可以少于数组元素个数，不足部分的数组元素初始化成0。例如：
 - `int b[10]={1,2,3,4};` //后6个元素初始化为0
- 如果每个元素都进行了初始化，则数组元素个数可以省略。例如：
 - `int c[]={1,2,3};` //隐含着c由三个元素构成

一维数组的操作

- 通常情况下，对数组类型数据的操作要通过其元素来进行。
- 访问一维数组元素
 - 格式：
 <一维数组变量名>[<下标>]
 - <下标>为整型表达式
 - 第一个元素的下标为：0
 - 例如：
 - `int a[10];` //数组a
 - `a[0]、a[1]、...、a[9]` //数组元素
- C++语言不对数组元素下标越界进行检查。如：不检查`a[i]`中`i`的取值是否越界！

- 可把数组的每个元素看成是独立的变量。例如：

```
int a[10];  
int sum=0,i;  
for (i=0; i<10; i++) cin >> a[i];  
for (i=0; i<10; i++) sum += a[i];
```

- 不能对两个数组进行整体赋值，需要通过元素来进行：

```
int a[10],b[10];  
.....  
a = b; //Error  
for (int i=0; i<10; i++) a[i] = b[i]; //OK
```

例：用一维数组实现求第n个费波那契(Fibonacci)数

```
#include <iostream>
using namespace std;
int main()
{ const int MAX_N=40;
  int fibs[MAX_N];
  int n,i;
  cout << "请输入n(1-" << MAX_N << "):";
  cin >> n;
  if (n > MAX_N)
  { cout << "n太大! 应不大于" << MAX_N << endl;
    return -1;
  }
  fibs[0] = fibs[1] = 1; //初始化第1、2个费波那契数
  for (i=2; i<n; i++) //计算其它的费波那契数
    fibs[i] = fibs[i-1] + fibs[i-2];
  cout << "第" << n << "个费波那契数是：" << fibs[n-1] << endl;
  return 0;
}
```

例：从键盘输入10个数，输出其中的最大值

.....

```
int main()
{ int i,a[10];
    for (i=0; i<10; i++)
        cin >> a[i];
    int max=a[0]; //首先假设第0个元素最大
    for (i=1; i<10; i++)
        if (a[i] > max) max = a[i];
    cout << max << endl;
}
```

例：从键盘输入10个数，把它们从小到大排序后输出

- 从n个数中找出最大者，与第n个数交换位置；然后，从剩余的n-1个数中再找出最大者，与第n-1个数交换位置；...，一直到剩下的数只有一个为止。

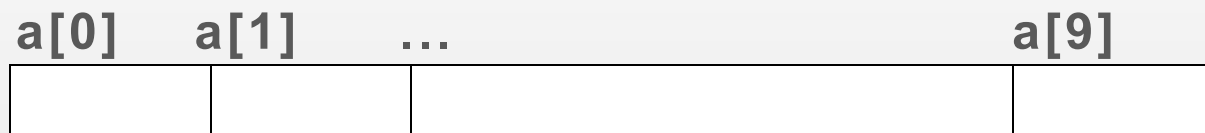
```
int main()
{ int a[10];
  int i;
  for (i=0; i<10; i++) cin >> a[i];
  for (int n=10; n>1; n--) //n为要排序的元素个数
  { int j=0; //用j记住最大元素的下标，首先假设第0个元素最大
    for (i=1; i<n; i++) if (a[i]>a[j]) j = i; //保持j为最大元素的下标
    //交换a[j]和a[n-1]的值
    int temp=a[n-1];
    a[n-1] = a[j];
    a[j] = temp;
  }
  for (i=0; i<10; i++) cout << a[i];
}
```

一维数组的存储分配

- 对于一维数组类型的数据，系统将会在内存中给其分配连续的存储空间来存储数组元素。例如：

```
int a[10];
```

- 其内存空间分配如下：



- 一维数组所占的内存空间大小可以用sizeof操作符来计算。例如：

- `int a[10];`
- `cout << sizeof(a);` //输出数组a所占的内存字节数。

向函数传递一维数组

- 被调用函数的形参一般为不带数组大小的一维数组定义以及数组元素的个数。例如：

```
int max(int x[],int num) //计算x中最大元素的下标
{
    int i,j;
    j = 0; //先假设第0个元素最大
    for (i=1; i<num; i++)
        if (x[i] > x[j]) j = i;
    return j;
}
```


- 调用者需要把一个一维数组变量的名以及数组元素的个数传给被调用函数。
例如：

```
int a[10],b[20],index_max;
```

```
.....
```

```
index_max = max(a,10);
```

```
cout << a[index_max] << index_max << endl;
```

```
index_max = max(b,20);
```

```
cout << b[index_max] << index_max << endl;
```

```
extern int max(int x[],int num);

int main()
{ int a[10];
  int i;
  for (i=0; i<10; i++) cin >> a[i];
  for (int n=10; n>1; n--) //n为要排序的元素个数
  { j = max(a,n); //计算最大元素的下标
    if (j != n-1) //交换a[j]和a[n-1]的值
    { int temp=a[n-1];
      a[n-1] = a[j];
      a[j] = temp;
    }
  }
  for (i=0; i<10; i++) cout << a[i];
  return 0;
}
```

- 为了提高数组传递的效率，数组作为函数参数传递时，C++默认传递的是数组在内存中的首地址，这样，函数的形参数组不再分配内存空间，它共享实参数组的内存空间。
- 注意：函数中通过形参数组能改变实参数组的值！（函数的副作用）

字符串的一种实现 - - 一维字符数组

- C++语言本身没有提供字符串类型。
- 在C++中，通常用元素类型为char的一维数组（字符数组）来表示字符串类型。例如：

`char s[10];` //可表示长度为9的字符串

- **注意：**在定义一个字符数组时，其元素个数应比它实际能够存储的字符串的最大长度**多一个**，因为，通常要在字符串中最后一个字符的后面放置一个表示字符串结束的字符：'\0'。
- 字符串作为函数参数传递时，只要给出一维字符数组这一个参数就够了，不需要给出元素个数（字符串长度）。

例：编写一个函数计算字符串的长度

```
int strlen(char str[])  
{ int i=0;  
    while (str[i] != '\0') i++;  
    return i;  
}  
  
.....  
  
char a[10];  
cin >> a;  
cout << strlen(a) << endl;
```

例：编写一个函数把一个由数字构成的字符串转换成一个整型数

算法： "1234"-->((1*10+2)*10+3)*10+4

```
int str_to_int(char str[])
{ if (str[0] == '\0') return 0;
  int n=str[0]-'0';
  for (int i=1; str[i] != '\0'; i++)
    n = n*10+(str[i]-'0');
  return n;
}
```

```
int str_to_int(char str[])  
{ int n=0;  
  for (int i=0; str[i] != '\0'; i++)  
    n = n*10+(str[i]-'0');  
  return n;  
}
```

字符数组的初始化

```
char s[10]={'h','e','l','l','o','\0'};
```

```
char s[10]="hello";
```

```
char s[10]="hello";
```

```
char s[]="hello";
```

- 在上面的字符数组初始化中，除了第一种形式，其它形式的初始化都会在最后一个字符的后面自动加上'\0'，而对于第一种形式，程序中必须显式地加上'\0'。

标准库中的字符串处理函数（头文件 `cstring` 或 `string.h`）

- 计算字符串的长度

- `int strlen(const char s[]);`

- 字符串复制

- `char *strcpy(char dst[], const char src[]);`

- `char *strncpy(char dst[], const char src[], int n);`

- 字符串拼接

- `char *strcat(char dst[], const char src[]);`

- `char *strncat(char dst[], const char src[], int n);`

- 字符串比较

- `int strcmp(const char s1[], const char s2[]);`

- `int strncmp(const char s1[], const char s2[], int n);`

例：从键盘输入一个字符串，然后把该字符串逆向输出

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{ const int MAX_LEN=100;
  char str[MAX_LEN];
  cin >> str; //?
  int len = strlen(str); //str中的字符个数
  for (int i=0,j=len-1; i<len/2; i++,j--)
  {   char temp;
      temp = str[i];
      str[i] = str[j];
      str[j] = temp;
  }
  cout << str << endl;
  return 0;
}
```

二维数组

- 二维数组通常用于表示由**固定多个**同类型的具有行列结构的数据所构成的复合数据，如矩阵等。
- 二维数组所表示的是一种具有两维结构的数据，第一维称为二维数组的行，第二维称为二维数组的列。
- 二维数组的每个元素由其所在的行和列唯一确定。

二维数组类型定义

- 二维数组类型定义格式：

`typedef <元素类型> <二维数组类型名>[<行数>][<列数>];`

- <元素类型>为任意C++类型（void除外）
- <行数>和<列数>为整型常量表达式
- 例如：

```
typedef int A[10][5]; //由10行、5列int型元素  
                      //所构成的二维数组类型
```

二维数组类型变量的定义

- 二维数组类型变量的定义格式：

- `<二维数组类型名> <二维数组变量名>;`

或

- `<元素类型> <二维数组变量名>[<行数>][<列数>;`

或

- `<一维数组类型名> <二维数组变量名>[<元素个数>]`

- `<元素类型>`为任意C++类型（void除外）

- `<行数>`、`<列数>`和`<元素个数>`为整型常量表达式

- 例如：

- `typedef int A[10][5];`

- `A a;`

或

- `int a[10][5];`

或

- `typedef int B[5];`

- `B a[10];`

二维数组的初始化

```
int a[2][3]={{1,2,3},{4,5,6}};
```

```
int a[2][3]={1,2,3,4,5,6};
```

```
int a[2][3]={1,2,3,4};
```

- 以上初始化按照数组的行来进行

```
int a[2][3]={{1,2},{3,4}};
```

- 每一行的前2个分别初始化为1、2和3、4，其它为0

```
int a[][3]={{1,2,3},{4,5,6},{7,8,9}};
```

- 行数为3

二维数组的操作

- 访问二维数组元素，格式是：

<二维数组变量名>[<下标1>][<下标2>]

- <下标1>和<下标2>为整型表达式，均从0开始。

- 例如：

- int a[10][5];

- a[0][0]、a[0][1]、...、a[9][0]、...、a[9][4]

- 以行为单位访问，例如：

- int a[10][5];

- a[0]、a[1]、...、a[9]

- 上面每一个都为作为一个一维数组，代表二维数组中的一行

- 对二维数组的操作通常是通过其元素来进行。例如：

```
int a[10][5],sum=0;
```

```
.....
```

```
//计算所有元素的和
```

```
for (int i=0; i<10; i++)
```

```
    for (int j=0; j<5; j++)
```

```
        sum += a[i][j];
```

例：从键盘输入一个 $N \times N$ 的矩阵，把它转置后输出

```
#include <iostream>
using namespace std;
int main()
{ const int N=3;
  int a[N][N];
  int i,j;
  //输入矩阵数据
  cout << "请输入" << N << "x" << N << "矩阵：\n";
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      cin >> a[i][j];
```

//矩阵转置： 交换a[i][j]和a[j][i]的值, i=0~N-1,j=i+1~N-1

```
for (i=0; i<N; i++)
```

```
    for (j=i+1; j<N; j++)
```

```
    {        //交换a[i][j]与a[j][i]的值
```

```
        int temp=a[i][j];
```

```
        a[i][j] = a[j][i];
```

```
        a[j][i] = temp;
```

```
    }
```

//输出转置后的矩阵

```
cout <<    "转置后为: \n";
```

```
for (i=0; i<N; i++)
```

```
{    for (j=0; j<N; j++)
```

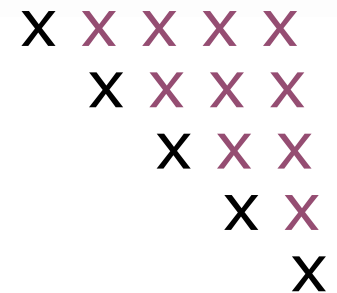
```
        cout << a[i][j] << ' ';
```

```
    cout << endl;
```

```
}
```

```
return 0;
```

```
}
```



```
X X X X X
  X X X X
    X X X
      X X
        X
```

二维数组的存贮

- 在C++中，二维数组元素是按照行来存储的，即先是第一行的元素；再是第二行的元素；...。例如：

- `int a[10][5];`

其内存空间分配如下:

a[0][0] ... a[0][4]			a[1][0] ... a[1][4]			a[9][0] ... a[9][4]		

- 了解二维数组的存储方式，在某些情况下有利于设计出高效的程序。例如，对于一个很大的二维数组a，并且假设该数组是按行存储的，那么下面按列来访问数组元素的程序效率可能会不高：（为什么？）

```
for (int col=0; i<N; i++)  
    for (int lin=0; j<M; j++)  
        ...a[lin][col]...
```

- 另外，有些语言（如FORTRAN）中的二维数组是按列来存储的。在混合语言编程中，如果一个二维数组被各种语言的程序片段所共享，那么就必须给出专门的处理，否则会产生错误的结果。

向函数传递二维数组

- 被调用函数的形参应为不带数组行数的二维数组定义及其行数。例如：

```
int sum(int x[][5], int lin) //对lin行、5列的二维数组求和
{
    int s=0;
    for (int i=0; i<lin; i++)
        for (int j=0; j<5; j++)
            s += x[i][j];
    return s;
}
```

- 注意：作为形参的二维数组的列数必须要写！因为，二维数组作为函数参数传递时实际传递的是数组的首地址，计算 $x[i][j]$ 的内存地址：

- $x[i][j]$ 的地址 = x 的首地址 + $i \times \text{列数} + j$

- 调用者需要提供一个二维数组变量（列数要与形参相同）和行数。例如：

```
int a[10][5],b[20][5];
```

```
.....
```

```
cout << "a的元素之和为：" << sum(a,10) << endl;
```

```
cout << "b的元素之和为：" << sum(b,20) << endl;
```

- 下面的二维数组c就不能调用函数sum来计算其元素的和，因为c的列数与函数sum要求的列数不符：

```
int c[40][20];
```

```
.....
```

```
sum(c,40); //Error
```

二维数组降为一维数组处理

```
int sum(int x[], int num)
{ int s=0;
  for (int i=0; i<num; i++) s += x[i];
  return s;
}
```

.....

```
int a[10][5], b[20][5], c[40][20];
```

.....

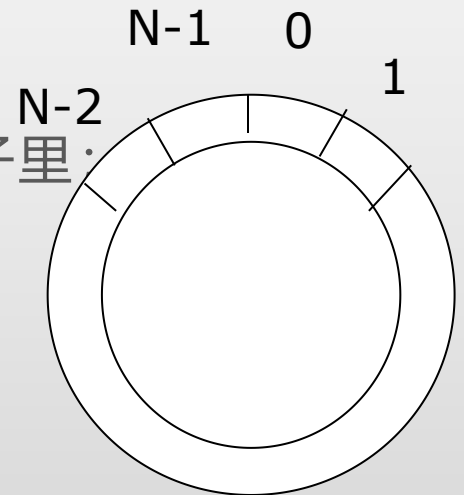
```
cout << sum(a[0], 10*5) << endl;
cout << sum(b[0], 20*5) << endl;
cout << sum(c[0], 40*20) << endl;
```


数组应用：求解约瑟夫（Josephus）问题

- 约瑟夫（Josephus）问题：
 - 有N个小孩（编号为0 ~ N-1）围坐成一圈，从某个小孩开始顺时针报数，报到M的小孩从圈子离开，然后，从下一个小孩开始重新报数，每报到M，相应的小孩从圈子离开，最后一个离开圈子的小孩为胜者，问胜者是哪一个小孩？
- 采用一个一维的循环数组in_circle来表示小孩围成一圈：

```
bool in_circle[N];
```

- in_circle[i]: true表示编号为i的小孩在圈子里；
编号为i的小孩离开了圈子。
- 圈子中i的下一个位置： $j=(i+1)\%N$



- 报数采用下面的方法来实现：
 - 从编号为0的小孩开始报数，用变量index表示要报数的小孩的下标，其初始值为N-1（即将报数的前一个小孩的下标）。
 - 下一个要报数的小孩的下标由下面式子计算：
$$(index+1)\%N$$
 - 用变量count来对**成功的报数**进行计数，
 - 每一轮报数前，count为0，每成功地报一次数，就把count加1，直到M为止。
 - 要使得报数成功，in_circle[index]应为true。
 - 报数成功后，把in_circle[index]设成false。
 - 用变量num_of_children_remained表示圈中剩下的小孩数目，其初始值为N。

//变量num_of_children_remained表示圈中剩下的小孩数目，其初始值为N

//变量count来对成功的报数进行计数

//变量index表示要报数的小孩的位置

```
#include <iostream>
```

```
using namespace std;
```

```
const int N=20,M=5;
```

```
int main()
```

```
{ bool in_circle[N];
```

```
    int num_of_children_remained,index;
```

```
    //初始化数组in_circle。
```

```
    for (index=0; index<N; index++)
```

```
        in_circle[index] = true;
```

//开始报数

index = N-1; //从编号为0的小孩开始报数,

 //index为前一个小孩的位置。

num_of_children_remained = N; //报数前的圈子中小孩个数

while (num_of_children_remained > 1)

{ int count = 0;

 while (count < M) //对成功的报数进行计数, 直到M。

 { index = (index+1)%N; //计算要报数的小孩的编号。

 if (in_circle[index]) count++; //如果编号为index的

 //小孩在圈子中, 该报数为成功的报数。

 }

 in_circle[index] = false; //小孩离开圈子。

 num_of_children_remained--; //圈中小孩数减1。

}

//找最后一个小孩

```
for (index=0; index<N; index++)
```

```
    if (in_circle[index]) break;
```

```
cout << "The winner is No." << index << ".\n";
```

```
return 0;
```

```
}
```

结构类型

- 结构类型用于表示由固定多个类型可以不同的元素（成员）所构成的复合数据，它是一种用户自定义类型。例如，
 - 一个学生数据：学号、姓名、性别、...
- 结构成员之间在逻辑上没有先后次序关系。

结构类型的定义

- 结构类型定义格式：

```
struct <结构类型名> {<成员表>;
```

- <成员表>列出结构类型的元素（成员）及其类型。
- 成员类型可以是任意的C++类型（void和本结构类型除外）。
- 成员的说明次序会影响成员的存储安排。

- 例如，

```
struct Student
{
    int no;
    char name[20];
    Sex sex;
    Date birth_date;
    char birth_place[40];
    Major major;
};
```

```
enum Sex { MALE, FEMALE };
struct Date
{
    int year, month, day;
};
enum Major
{ MATHEMATICS, PHYSICS,
  CHEMISTRY, COMPUTER,
  GEOGRAPHY,
  ASTRONOMY, ENGLISH,
  CHINESE, PHILOSOPHY
};
```

结构类型变量的定义

- 结构类型变量的定义格式如下：
 - <结构类型名> <结构类型变量名>;
 - struct <结构类型名> <结构类型变量名>;
 - struct <结构类型名> {<成员表>} <结构类型变量名>;
 - struct {<成员表>} <结构类型变量名>;

例如：

- Student st;
- struct Student st;
- struct Student { } st;
- struct { } st;

结构类型变量的初始化

- 在定义结构类型的变量时，依次给出成员的初始化，例如：

```
Student some_student={2,"李四", FEMALE,  
    {1970,12,20},"北京", MATHEMATICS};
```

- 在定义一个结构类型时,不能对其成员进行初始化，例如：

```
struct A  
{ int i=1; //Error  
  double d=1.2; //Error  
};
```

结构类型的操作

- 访问结构成员
 - 结构成员的访问要通过结构变量名来 “受限”
 - `<结构类型变量>.<结构成员名>`
 - 例如：
 - `Student st;`
 - `st.no, st.name,`
 - 每个成员都可以看作是一个独立的变量，可以分别操作它们，例如：
 - `st.no = 1;`
 - `strcpy(st.name, "张三");`
 - `st.sex = MALE;`
 - `.....`

结构作用域

- 成员名字的作用域为所在的结构--结构作用域。

```
struct A
{ char name[10]; //OK
};
struct B
{ char name[5]; //OK
};
char name[20]; //OK
int main()
{ A a;
  B b;
  ... a.name ... //结构变量a的成员变量name。
  ... b.name ... //结构变量b的成员变量name。
  ... name ...  //全局变量name。
}
```

- 在C++中，结构类型的名字可以与同一作用域中的其它非结构类型标识符相同。例如，下面的用法是合法的：

```
struct A //结构类型A
{
    .....
};

int A; //整型变量A, OK, 但不好!

.....

struct A a; //定义一个结构类型A的变量a
A = 1; //把1赋值给整型变量A
```

- 结构赋值

- 可以对结构类型数据进行整体赋值，例如：

- Student st1,st2;

- st1 = st2; //OK

- 不同的结构类型之间不能相互赋值，例如：

- Student st;

- Date today;

- st = today; //Error

结构类型的存储

- 结构类型的变量在内存中占用一块连续的存储空间，其各个元素依它们在结构类型中的定义次序存储在这块内存空间中。例如：
 - `Student st;`
 - 其内存空间安排如下：

`st.no` `st.name` `st.sex` `st.birth_date` `st.birth_place` `st.major`

--	--	--	--	--	--

- 可用`sizeof`计算结构类型的大小。

- 在C++实现中，为了提高计算机指令对整个结构及某些成员的访问效率，在为结构类型的变量分配内存空间时，往往要对整个结构以及某些结构成员所占的空间按某种方式进行地址对齐。例如，

```
struct A//对齐到地址为4的倍数的边界上
{
    char ch1;
    int i1; //对齐到地址为4的倍数的边界上
    char ch2;
    int i2; //对齐到地址为4的倍数的边界上
};
```

- 这样就会使得结构成员的内存空间可能会存在“空隙”。这时，整个结构类型的内存空间就会比各个成员内存空间之和要大。（假设int为4个字节，上述结构要占用16个字节）。怎么处置这个问题！

向函数传递结构数据

- 可作为参数传给函数，默认参数传递方式为值传递，例如：

```
void f(Student st)
{ ... st.name ...
}
```

.....

```
Student st1;
```

.....

```
f(st1);
```

- 可作为函数返回值返回给调用者，例如：

```
Student g()
{ Student st;
```

.....

```
    return st;
```

```
}
```

.....

```
Student st1=g();
```


例：名表

- 名表是指一个由一系列名字及其相关信息所构成的表。
- 名表可以用一个一维数组来表示，每个元素用一个结构来表示。

```
const int NAME_LEN=20;
const int TABLE_LEN=100;
struct TableItem
{ char name[NAME_LEN];
  ..... //其它信息
};
TableItem name_table[TABLE_LEN];
```

名表的查找（检索）

- 名表查找是指根据某个名字在名表中查找与该名字相关的信息。
 - 顺序查找
 - 折半查找（二分法）

名表查找（顺序）

```
#include <cstring>
using namespace std;
int linear_search(char key[], //关键词
                  TableItem t[], //名表
                  int num_of_items) //表的长度
{ int index;
  for (index=0; index<num_of_items; index++)
    if (strcmp(key,t[index].name) == 0) break;
  if (index < num_of_items)
    return index;
  else
    return -1;
}
```

```
const int NAME_LEN=20;
const int TABLE_LEN=100;
struct TableItem
{ char name[NAME_LEN];
  ..... //其它信息
};
TableItem name_table[TABLE_LEN];
int main()
{ int n; //名表元素的个数（长度）
  ..... //名表元素数据的获取
  char name[NAME_LEN]; //待查找的名字
  ..... //待查找的名字获取
  int result = linear_search(name,name_table,n);
  if (result == -1)
  { cout << "Not found\n";
    return -1;
  }
  ..... //使用name_table[result]的值
  return 0;
}
```

名表查找（二分法）

- 如果名表的元素已经按名字大小排了序，则可以采用二分法（折半）查找：
 - 首先用要查找的值与名表中间位置上的元素与进行比较
 - 若相等，则找到，
 - 若大于中间位置上的元素，则在名表的后半部分中继续进行查找；
 - 若小于中间位置上的元素，则在名表的前半部分中继续进行查找。
 - 在前半部分或后半部分中查找时，仍然采用折半查找，直到找到或表中元素比较完为止。

```
int binary_search(char key[], TableItem t[], int
    num_of_items)
{ int index,first,last;
  first = 0;  last = num_of_items-1;
  while (first <= last)
  {    index = (first+last)/2;
    int r=strcmp(key,t[index].name);
    if (r == 0) // key等于t[index].name
        return index;
    else if (r > 0) // key大于t[index].name
        first = index+1;
    else //key小于t[index].name
        last = index-1;
  }
  return -1;
}
```

算法分析

- 顺序查找
 - 最好情况：比较1次
 - 最坏情况：比较N次
 - 平均情况： $(1+2+\dots+N)/N = (N+1)/2$ 次
- 二分法查找
 - 最好情况：比较1次
 - 最坏情况：比较 $\log_2(N+1)$ 次
 - 平均情况： $\log_2(N+1)-1$ 次

联合（union）类型

- 如何表示一组图形数据（圆、线段、矩形等）？
 - ？ `figures[100];` //元素类型怎么定义？
- 联合类型：
 - 用一个类型表示多种类型的数据。
 - 例如，下面的联合类型A，可以用于描述int、char或double类型的数据：

```
union A
{ int i;
  char c;
  double d;
};
```


- 对于一个联合类型的变量，在程序中将会分阶段地把它作为不同的类型来使用，而不会同时把它作为几种类型来用。例如：

```
A a;
```

```
a.i = 1; //给变量a赋一个int型的值
```

```
... a.i ... //把a当作int型来用
```

```
a.c = 'A'; //给变量a赋一个char型的值
```

```
... a.c ... //把a当作char型来用。
```

```
a.d = 2.0; //给变量a赋一个double型的值
```

```
... a.d ... //把a当作double型来用。
```

- 当给一个联合类型的变量赋了一个某种类型的值之后，如果以另外一种类型来使用这个值，将得不到原来的值。例如：

```
a.i = 12;
```

```
cout << a.d; //输出什么呢？
```

- 实际上，联合类型的所有成员占有同一块内存空间，该内存空间的大小为其最大成员所需要的内存空间的大小。
 - `cout << sizeof(a);` //输出：8（假设double占8个字节）
- 也可利用联合类型来实现多种数据共享内存空间。例如：

```
union AB
{ int a[100];
  double b[100];
};
AB buffer;
... buffer.a ... //使用数组a
.....
... buffer.b ... //使用数组b
.....
```

例：从键盘输入一组图形数据，然后输出相应的图形。其中的图形可以是：线段、矩形和圆。

- 一组图形数据可用一个一维数组表示：

```
const int MAX_NUM_OF_FIGURES=100;  
Figure figures[MAX_NUM_OF_FIGURES];
```

- 数组元素的类型Figure是一个联合类型。

```
struct Line
{ double x1,y1,x2,y2;
};
struct Rectangle
{ double left,top,right,bottom;
};
struct Circle
{ double x,y,r;
};
union Figure
{ Line line;
  Rectangle rect;
  Circle circle;
};
Figure figures[MAX_NUM_OF_FIGURES];
```

- 该表示存在问题：
 - 无法区分存储在figures[i]中的是什么图形!

●解决上面问题的一种办法:

```
struct Line
{ double x1,y1,x2,y2;
};
struct Rectangle
{ double left,top,right,bottom;
};
struct Circle
{ double x,y,r;
};
union Figure
{ Line line;
  Rectangle rect;
  Circle circle;
};
enum FigureShape { LINE, RECTANGLE, CIRCLE };
struct TaggedFigure
{ FigureShape shape;
  Figure figure;
};
TaggedFigure figures[MAX_NUM_OF_FIGURES];
```

- 给一个数组元素figures[i]赋值时，除了图形的几何数据外，还需给出它为何种图形。例如：

```
figures[i].shape = LINE;  
figures[i].figure.line.x1 = 10;  
figures[i].figure.line.y1 = 20;  
figures[i].figure.line.x2 = 100;  
figures[i].figure.line.y2 = 200;
```

- 访问数组元素figures[i]时，通过figures[i].shape的值就可知道figures[i]存储的是什么图形。

● 图形数据的输入：

```
int count;
for (count=0; count<MAX_NUM_OF_FIGURES; count++)
{   int shape;
    do
    {   cout << "请输入图形的种类(0:线段,1:矩形,2:圆,-1:结束):";
        cin >> shape;
    } while (shape < -1 || shape > 2);
    if (shape == -1) break;
    switch (shape)
    {   case 0: //线
        figures[count].shape = LINE;
        cout << "请输入线段的起点和终点坐标 (x1,y1,x2,y2) :";
        cin >> figures[count].figure.line.x1
            >> figures[count].figure.line.y1
            >> figures[count].figure.line.x2
            >> figures[count].figure.line.y2;
        break;
```

```
case 1: //矩形
```

```
    figures[count].shape = RECTANGLE;
```

```
    cout << "请输入矩形的左上角和右下角坐标(x1,y1,x2,y2):";
```

```
    cin >> figures[count]. figure.rect.left
```

```
        >> figures[count]. figure.rect.top
```

```
        >> figures[count]. figure.rect.right
```

```
        >> figures[count]. figure.rect.bottom;
```

```
    break;
```

```
case 2: //圆形
```

```
    figures[count].shape = CIRCLE;
```

```
    cout << "请输入圆的圆心坐标和半径 (x,y,r) : ";
```

```
    cin >> figures[count]. figure.circle.x
```

```
        >> figures[count]. figure.circle.y
```

```
        >> figures[count]. figure.circle.r;
```

```
    break;
```

```
    } //end of switch
```

```
    } //end of for
```


- 图形的输出：

```
for (int i=0; i<count; i++)  
{ switch (figures[i].shape)  
  { case LINE:  
    draw_line(figures[i]. figure.line);  
    break;  
  case RECTANGLE:  
    draw_rectangle(figures[i]. figure.rect);  
    break;  
  case CIRCLE:  
    draw_circle(figures[i]. figure.circle);  
    break;  
  }  
}
```

- 解决上面图形表示问题的另一种办法是：

```
enum FigureShape { LINE, RECTANGLE, CIRCLE };  
struct Line  
{ FigureShape place_holder;  
  double x1,y1,x2,y2;  
};  
struct Rectangle  
{ FigureShape place_holder;  
  double left,top,right,bottom;  
};  
struct Circle  
{ FigureShape place_holder;  
  double x,y,r;  
};  
union Figure  
{ FigureShape shape;  
  Line line;  
  Rectangle rect;  
  Circle circle;  
};
```

指针类型

- 需求：
 - 如何把数据的地址传给一个函数，以提高参数传递的效率？相应参数的类型应是什么？
 - 如何使用元素个数可变的数组？
- 指针为上述问题提供了解决方案。
 - 指针是内存地址的抽象表示，一个指针代表了一个内存地址。
 - 获取变量的地址：&<变量名>
- 指针类型是一种用户自定义的简单类型，它的值集是由一些内存地址（指针）构成。

指针类型的定义

- 指针类型的定义格式为：

```
typedef <类型> *<指针类型名>;
```

- 其中，<指针类型名>表示一个指针类型，其值集为<类型>所表示的数据的地址。
- 例如，下面定义了一个指针类型Pointer，其值集为所有int变量的地址。

```
typedef int *Pointer;
```

指针类型变量的定义

- 指针类型变量（简称：指针变量）的定义格式：
 - <指针类型名> <指针变量名>;
 - 或
 - <类型> *<指针变量名>; // 指针类型与变量定义合一
 - 例如，下面定义了一个指针类型变量p：
 - typedef int *Pointer;
 - Pointer p; // p为一个指向整数类型数据的指针变量
 - 或
 - int *p; // p为一个指向整数类型数据的指针变量
- p的取值可以是下面int型变量x的地址：
- int x;
 - p = &x;

```
int *p,*q; //p和q均为指针变量
```

```
int *p,q; //p为指针变量, q为int型变量
```

```
int* p,q; //p为指针变量, q为int型变量
```

```
typedef int* Pointer;
```

```
Pointer p,q; //p和q均为指针类型的变量
```

```
void *p; //p可以指向任意类型的数据
```

- **注意：** 指针变量拥有自己的内存空间，在该空间中存储的是另一个数据的内存地址，例如：

```
int x=1;
```

```
int *p=&x;
```



指针类型的基本操作

- 赋值
- 间接访问
- 指针运算

指针赋值操作

```
int x,*p,*p1;
```

```
double y,*q;
```

```
p = &x; //OK, p指向x。
```

```
q = &y; //OK, q指向y。
```

```
p = &y; //Error, 类型不一致。
```

```
q = &x; //Error, 类型不一致。
```

```
p1 = p; //OK, p1指向p所指向的变量。
```

```
p1 = q; //Error, 类型不一致。
```

```
p = 0; //OK, 使得p不指向任何变量。
```

```
p = 120; //Error, 120为int型。
```

```
p = (int *)120; //OK, 不建议使用。
```

```
void *any_pointer;
```

```
any_pointer = &x; //OK
```

```
any_pointer = &y; //OK
```

间接访问操作(*和->)

- 可以通过 “*” 来访问一个指针变量指向的变量，例如：

```
int x;
```

```
int *p;
```

```
p = &x;
```

```
x = 1;
```

```
*p = 2; //等价于： x = 2;
```

指针间接访问操作的例子

```
int *p;  
int x;  
x = 1;  
p = &x;  
*p = 2;
```

- 执行操作：“x = 1;”前，（假设120和124分别代表变量x和p的内存地址）

	x	p
120:	?	124: ?

- 执行操作：“x = 1;”后：

	x	p
120:	1	124: ?

- 执行操作：“p = &x; ” 后：

	x	p
120:	1	124: 120

- 执行操作：“*p = 2;”后，

	x	p
120:	2	124: 120

- 对于一个指向结构类型变量的指针变量，如果通过该指针变量来访问相应结构变量的成员，则可以写成：

- (*<指针变量>).<结构成员>

或

- <指针变量>-><结构成员>

例如：

```
struct A
```

```
{ int i;
```

```
  double d;
```

```
  char ch;
```

```
};
```

```
A a;
```

```
A *p=&a;
```

```
.....
```

```
cout << (*p).i << p->d << endl; //输出a.i和a.d
```

- 请注意下面的问题：

- `int *p;`

- `*p = 1;` //1赋值到哪里去了？

指针的运算

- 一个指针加上或减去一个整型值

- 实际加（或减）的值由该指针所指向的数据类型来定。例如：

```
int x;
```

```
int *p;
```

```
p = &x + 2; //p的值为x的地址加上sizeof(int)*2
```

- 该操作通常用于以指针方式来访问数组元素。例如：

```
int a[10];
```

```
int *p;
```

```
p = &a[0]; //或 p = a;
```

访问数组a的元素可采用：

- a[0]、a[1]、...、a[9]

- *p、*(p+1)、...、*(p+9)

- p[0]、p[1]、...、p[9]

```
int a[10];  
int sum=0;
```

.....

```
for (int i=0; i<10; i++)  
    sum += a[i];
```

或者

```
int *p=&a[0];  
for (int i=0; i<10; i++)  
{ sum += *p;  
  p++;  
}
```

或者

```
int *p=&a[0];  
for (int i=0; i<10; i++)  
    sum += p[i];
```

●两个同类型的指针相减

- 实际结果由指针所指向的类型来定。例如：

```
int a[10];  
int *p = &a[0];  
int *q = &a[3];  
cout << q-p << endl; //输出3
```

●两个同类型的指针比较

- 比较它们所对应的内存地址的大小。例如：

```
int a[10],sum,*p,*q;  
.....  
for (p=&a[0],q=&a[9],sum=0; p<=q; p++)  
    sum += *p;
```


指针的输出

```
int x=1;  
int *p=&x;  
cout << *p; //输出p指向的值 (x的值)  
cout << p; //输出p的值(x的地址)
```

- 特殊情况:

```
char str[]="ABCD";  
char *q=&str[0];  
cout << *q; //输出q指向的字符: A  
cout << q; //输出q指向的字符串: ABCD  
cout << (void *)q //输出p的值, 即字符串"ABCD"的内存首地址
```

下面指针的用法好吗？

```
int x=0,y;
```

```
int *p=&x;
```

```
*p = 1;
```

```
y = *p+3*x;
```

指针的主要用途

- 作为函数形参的类型
- 实现动态数据结构
- 高效访问数组元素

指针作为形参类型

```
int f(SomeType *p)
{ ... *p ... //通过p间接访问传进来的数据
}

.....

SomeType a;
f(&a);
```

- 指针作为形参的类型可以产生两个效果：
 - 提高参数传递效率：大（量）数据的参数传递。
 - 通过形参改变实参的值：把函数的计算结果通过参数返回给调用者。

●向函数传递大型的结构类型数据

```
struct A
```

```
{ int no;
```

```
  char name[20];
```

```
  .....
```

```
};
```

```
void f(A *p) //p为指向结构类型的指针
```

```
{ .....
```

```
  ... p->no .... //或者, (*p).no
```

```
  ... p->name ... //或者, (*p).name
```

```
  .....
```

```
}
```

```
int main()
```

```
{ A a;
```

```
  .....
```

```
  f(&a); //把结构变量的地址传给函数f。
```

```
  .....
```

```
}
```

- 在C++中，数组参数的默认传递方式是把实数组的首地址传给函数，以提高参数传递效率。
- 实际上，对于下面的函数定义和调用：

```
int max(int x[],int num)
{ .....
  ... x[i] ...
  .....
}
```

```
int main()
{ int a[10];
  .....
  ...max(a,10)...
  ....
}
```

- 编译程序将按下面的方式来实现：

```
int max(int *x,int num)
{ .....
  ... *(x+i) ...
  .....
}
```

```
int main()
{ int a[10];
  .....
  ...max(&a[0],10)...
  ....
}
```

通过形参改变实参的值

- 编写一个能交换两个变量值的函数

```
void swap(int x, int y)
{   int t=x;
    x = y;
    y = t;
}
int main()
{   int a=0,b=1;
    swap(a,b);
    cout << "a=" << a << ",b=" << b << endl;
    return 0;
}
```

输出：a=0,b=1

- 上述函数swap无法实现所需功能！

●解决方案：

```
void swap(int *px, int *py)
{
    //交换px和py所指向的变量的值。
    int t=*px;
    *px = *py;
    *py = t;
}

int main()
{
    int a=0,b=1;
    swap(&a,&b); //把变量a和b的地址传给函数swap的形参px
    和py。
    cout << "a=" << a << ",b=" << b << endl;
    return 0;
}
```

输出： a=1,b=0

避免指针参数带来的不必要的副作用

- 通过指针类型的形参可以改变实参的值，从而导致函数的副作用。
- 如何避免指针参数带来的不必要的副作用？

```
int f(int *p)
{   int y = (*p)*2; //使用p指向的值
    (*p)++; //改变了p指向的值
    return y;
}

int main()
{   int x=10;
    cout << x+f(&x) << endl; //输出： 31
    return 0;
}
```

指向常量的指针

const int *p; //p为指向常量的指针变量

int *q;

const int x=0;

int y;

p = &x; //OK

*p = 1; //Error

q = &y; //OK

*q = 1; //OK

q = &x; //Error

p = &y; //? OK

指向常量的指针作为函数形参类型

- 把形参定义为指向常量的指针，可避免不必要的副作用。例如：

```
void g(const A *p) //A为一个结构类型
{
    .....
    p->no = ... //Error, 不能改变p所指向的数据。
}
```

- 再例如：

```
void f(const int p[], int num)
//或者, void f(const int *p, int num)
{
    .....
    p[i] = ... //Error, 不能改变p所指向的数据。
    .....
}
```

- **结论：** 如果只需要提高参数效率，而不想有副作用，则需要把形参定义为指向常量的指针类型！

指针类型的常量

```
int x,y;
```

```
int *const p=&x; //定义了一个指针类型的常量p,  
                //它指向的是变量
```

```
*p = 1; //OK, *p是一个变量
```

```
p = &y; //Error, p是一个常量, 其值不能被修改
```

```
.....
```

```
void f(int *p)
{ .....
  ... *p ... //访问实参吗?
  .....
}
```

```
void f(int *p)
{ int m;
  p = &m; //OK
  ... *p ... //访问m!
  .....
}
```

```
void f(int *const p)
{ .....
  ... *p ... //访问实参?
  .....
}
```

```
void f(int *const p)
{ int m;
  p = &m; //Error
  ... *p ... //访问实参!
  .....
}
```

- 实际上，对于下面的函数定义：

```
int max(int x[],int num)
{
    .....
    x = ...; //Error
    ... x[i] ...
    .....
}
```

- 编译程序将精确地按下面的方式来解释：

```
int max(int *const x,int num)
{
    .....
    x = ...; //Error
    ... *(x+i) ...
    .....
}
```

指向常量的指针常量

```
const int x=0;
```

```
const int y=1;
```

```
const int * const p=&x; //p是一个指向常量的指针常量
```

```
*p = 1; //Error
```

```
p = &y; //Error
```

```
void f(const int *const p)
{ int m;
  p = &m; //Error
  *p = 10; //Error
  .....
}
```

指针作为函数返回值类型

- 函数的返回值类型可以是一个指针类型。例如：

```
int *max(const int x[], int num)
{ int max_index=0;
  for (int i=1; i<num; i++)
      if (x[i] > x[max_index]) max_index = i;
  return (int*)&x[max_index];
}

int main()
{ int a[100];
  .....
  cout << *max(a,100) << endl;
  return 0;
}
```


- 不能把局部量的地址返回给调用者。例如：

```
int *f()
{ int i=0;
  return &i;
}
int *g()
{ int j=1;
  return &j;
}
int main()
{ int x ;
  int *p=f();
  int *q=g();
  x=*p+*q;
  cout << x << endl; //输出什么?
  return 0;
}
```

指针与动态变量

- 对输入的100个数进行排序：

```
int a[100];  
for (int i=0; i<100; i++) cin >> a[i];  
sort(a,100);
```

- 对输入时指定的若干个数进行排序，下面的做法可行吗？

```
int n;  
cin >> n; //数的个数  
int a[n]; //?  
for (int i=0; i<n; i++) cin >> a[i];  
sort(a,n);
```

动态变量

- **动态变量**是指在程序**运行**中，由程序根据需要所创建的变量。例如：

- `int *p1;`

- `p1 = new int;` //创建了一个int型动态变量，p1指向之。

或

- `p1 = (int *)malloc(sizeof(int));` //#include <cstdlib>

- 再例如：

- `int *p2;`

- `p2 = new int[n];` //创建了一个由n（n可以是**变量**）个
//int型元素所构成的动态数组变量，
//p2指向其第一个元素。

或

- `p2 = (int *)malloc(sizeof(int)*n);`

- 对于一个动态的n维数组，除了第一维的大小外，其它维的大小必须是常量或常量表达式。例如：

- `int (*q)[20];` //q为一个指向由20个int型元素所构成的或者

- `typedef int A[20];`

- `A *q;`

- `int n;`

- `.....`

- `q = new int[n][20];` //创建一个n行、20列的二维动态
//数组，返回第一行的地址。 等价于：`q = new A[n];`

- `... q[i][j] ...` //访问q指向的二维数组的第i行、第j列的元素

- 如何创建一个m行、n列的动态数组？

- 用一维数组实现：`int *p=new int[m*n];`

- 第i行、第j列元素：`*(p+i*n+j)`

通过指针访问动态变量

- 动态变量没有名字，对动态变量的访问需要通过指向动态变量的指针变量来进行（间接访问）。例如：

```
int *p,*q;  
p = new int;  
...*p... //访问上面创建的int型动态变量  
q = new int[n];  
...(q+3)... //或...q[3]..., 访问上面创建的  
                //动态数组中的第4个元素
```

- 在C++中，动态变量需要由程序显式地撤消（使之消亡）。例如：

- `delete p;` //撤消p指向的int型动态变量

或

- `free(p);`

- 再例如：

- `delete []q;` //撤消q指向的动态数组

或

- `free(q);`

- 一般来说，用new创建的动态变量需要用delete来撤销；用malloc创建的动态变量则需要用free撤销。

- 用delete和free只能撤消动态变量！
 - `int x,*p;`
 - `p = &x;`
 - `delete p; //Error`
 - 用delete和free撤消动态数组时，其中的指针变量必须指向数组的第一个元素！
 - `int *p=new int[n];`
 - `p++;`
 - `delete []p; //Error`
- 或
- `int *p=(int *)malloc(sizeof(int)*n);`
 - `p++;`
 - `free(p); //Error`

动态变量的生存期

- 动态变量具有动态生存期：
 - **动态生存期**是指从new操作或函数调用malloc后开始，到delete操作或函数调用free时结束的时间段。
 - 一个动态变量创建后，只要不对它进行delete操作或函数调用free，它就一直存在，直到程序执行结束。
- 动态变量的空间是在堆区中分配。

“内存泄漏” 问题

- 内存泄漏 (memory leak) :

- 没有撤消动态变量，而把指向它的指针变量指向了别处或指向它的指针变量的生存期结束了，这时，这个动态变量存在但不可访问（这个动态变量已成为一个“孤儿”），从而浪费空间。例如：

```
int x,*p;
```

```
p = new int[10]; //动态数组
```

```
p = &x; //之后，上面的动态数组就访问不到了！
```

“悬浮指针”问题

- 悬浮指针 (dangling pointer) :
 - 用delete或free撤消动态变量后，C++编译程序一般会把指向它的指针变量的值赋为0，这时该指针指向一个无效空间。例如：

```
int *p;  
p = new int;  
.....  
delete p; //撤销了p所指向的动态变量  
.....  
*p = 1; //?
```

动态变量的应用——动态数组

- 对输入的若干个数进行排序，如果输入时先输入数的个数，然后再输入各个数，则可用下面的动态数组来表示这些数：

```
int n;  
int *p;  
cin >> n;  
p = new int[n];  
for (int i=0; i<n; i++) cin >> p[i];  
sort(p,n);  
.....  
delete []p;
```

- 对输入的若干个数进行排序，在输入时，先输入各个数，最后输入一个结束标记（如：-1），这时，可以按以下方式实现：

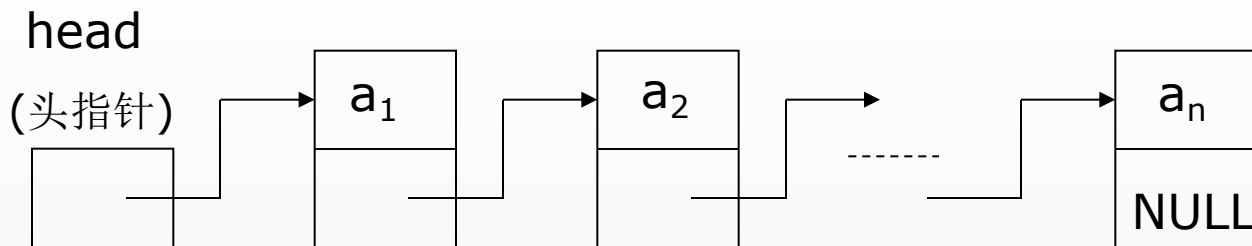
```
const int INCREMENT=10;
int max_len=20,count=0,n,*p=new int[max_len];
cin >> n;
while (n != -1)
{ if (count >= max_len)
    { max_len += INCREMENT;
      int *q=new int[max_len];
      for (int i=0; i<count; i++) q[i] = p[i];
      delete []p;
      p = q;
    }
    p[count] = n;
    count++;
    cin >> n;
}
sort(p,count);
.....
delete []p;
```

- 上面的实现方法虽然可行，但是，
 - 当数组空间不够时，它需要重新申请空间、进行数据转移以及释放原有的空间，这样做比较麻烦并且效率有时不高。
 - 当需要在数组中增加或删除元素时，它还将会面临数组元素的大量移动问题。
- 链表可以避免数组的上述问题！

动态变量的应用——链表

- 链表用于表示由若干（个数不定）同类型的元素所构成的具有线性结构的复合数据。
- 链表元素在内存中不必存放在连续的空间内。
- 链表中的每一个元素除了本身的数据外，还包含一个（或多个）指针，它（们）指向链表中下一个（和其它）元素。
- 如果每个元素只包含一个指针，则称为单链表，否则称为多链表。

单链表



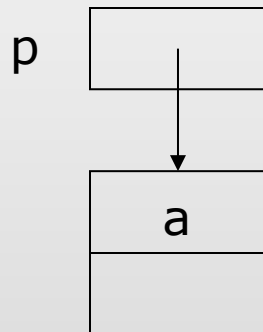
- 单链表的每个元素只包含一个指针。
- 需要一个**头指针**，指向第一个元素。
- 单链表中的结点类型和表头指针变量可定义如下：

```
struct Node //结点的类型定义
{ int content; //代表结点的数据
  Node *next; //代表后一个结点的地址
};
```

```
Node *head=NULL; //头指针变量定义，初始状态下
                  //为空值。NULL在cstdio中定义为0
```

在链表中插入一个结点

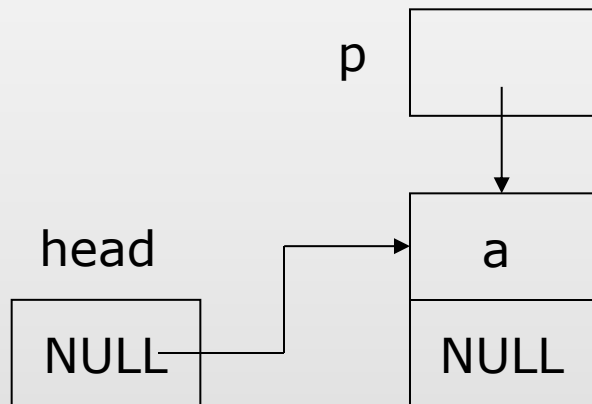
- 首先产生一个新结点：
 - `Node *p=new Node; //产生一个动态变量来表示新结点`
 - `p->content = a; //把a赋给新结点中表示结点值的成员`
- 图示为：



- 如果链表为空（创建第一个结点时），则进行下面的操作：

```
if (head == NULL) //表头指针为空
{ head = p; //头指针指向新结点。
  p->next = NULL; //或, head->next = NULL;
                  //把新结点的next成员置为NULL。
}
```

- 图示为：



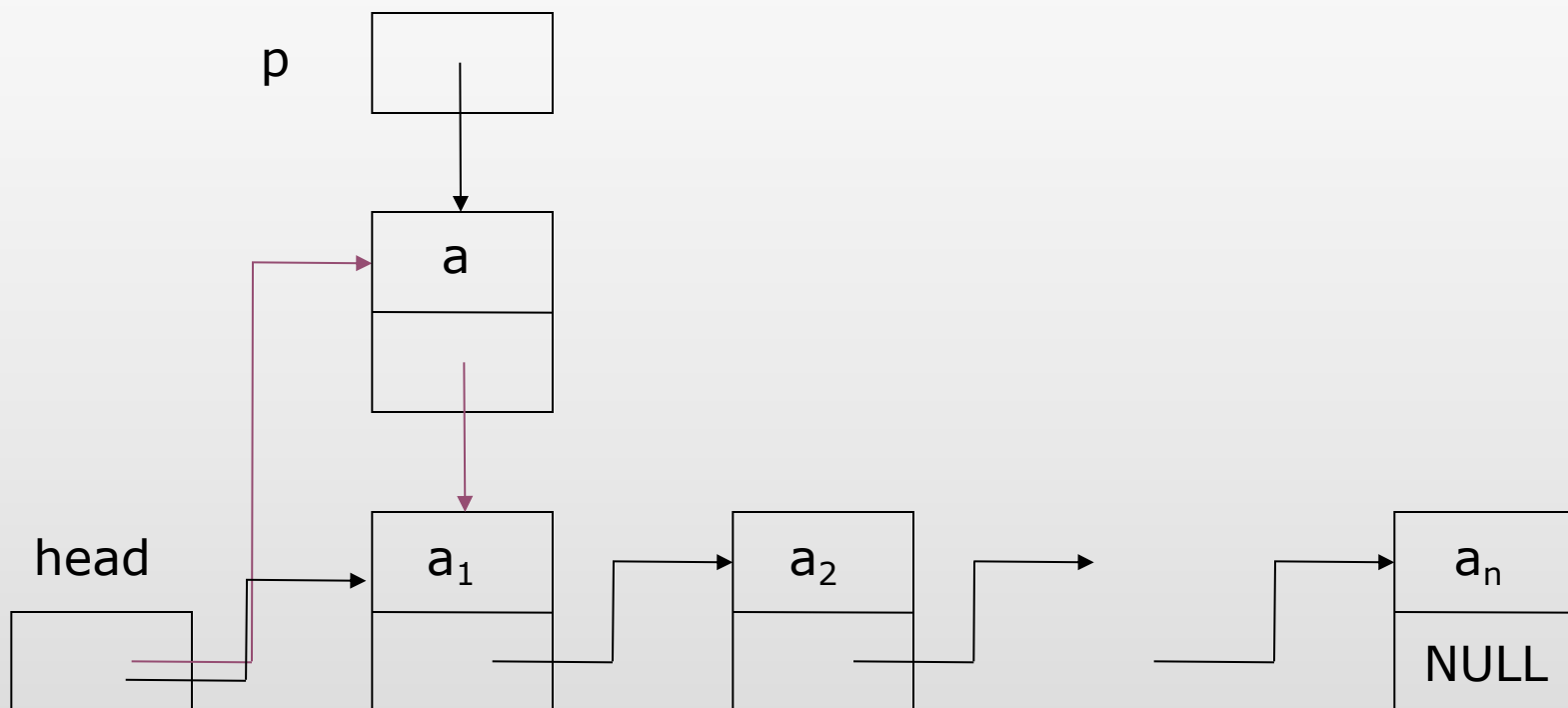
以下操作均假设链表不为空！

- 如果新结点插在表头，则进行下面的操作：

`p->next = head;` //把新结点的下一个结点指定为
//链表原来的第一个结点。

`head = p;` //表头指针指向新结点。

- 图示为：



- 如果新结点插在表尾，则进行下面的操作：

Node *q=head; //q指向第一个结点

while (q->next != NULL) //循环查找最后一个结点

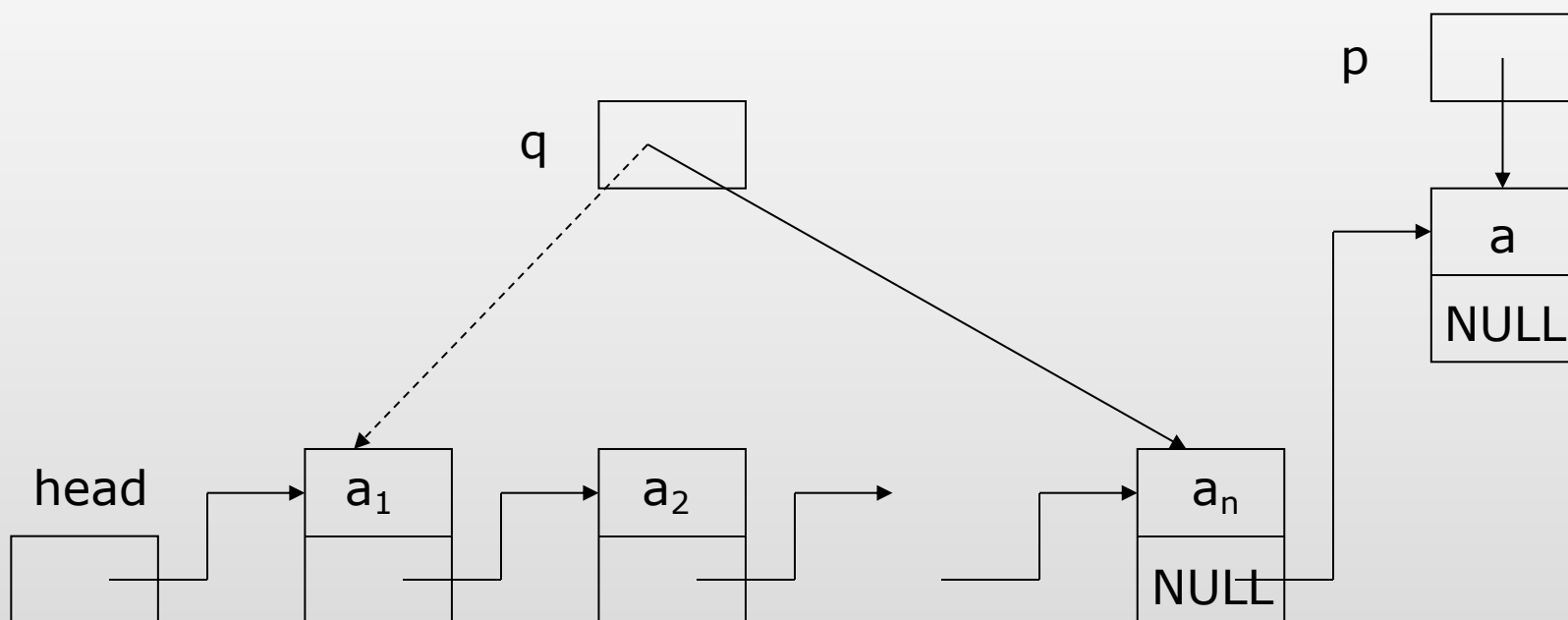
q = q->next;

//循环结束后，q指向链表最后一个结点

q->next = p; //把新结点加到链表的尾部。

p->next = NULL; //把新结点的next成员置为NULL。

- 图示为：



- 如果新结点插在链表中第 i ($i > 0$) 个结点 (a_i) 的后面, 则进行下面的操作:

```
Node *q=head; //q指向第一个结点。
```

```
int j=1; //当前结点的序号, 初始化为1
```

```
while (j < i && q->next != NULL) //循环查找第i个结点。
```

```
{ q = q->next; //q指向下一个结点
```

```
  j++; //结点序号增加1
```

```
}
```

```
//循环结束时, q或者指向第i个结点, 或者指向最后一个结点  
  (结点数不够i时)。
```

```
if (j == i) //q指向第i个结点。
```

```
{ p->next = q->next; //把q所指向结点的下一个结点指定为  
  //新结点的下一个结点。
```

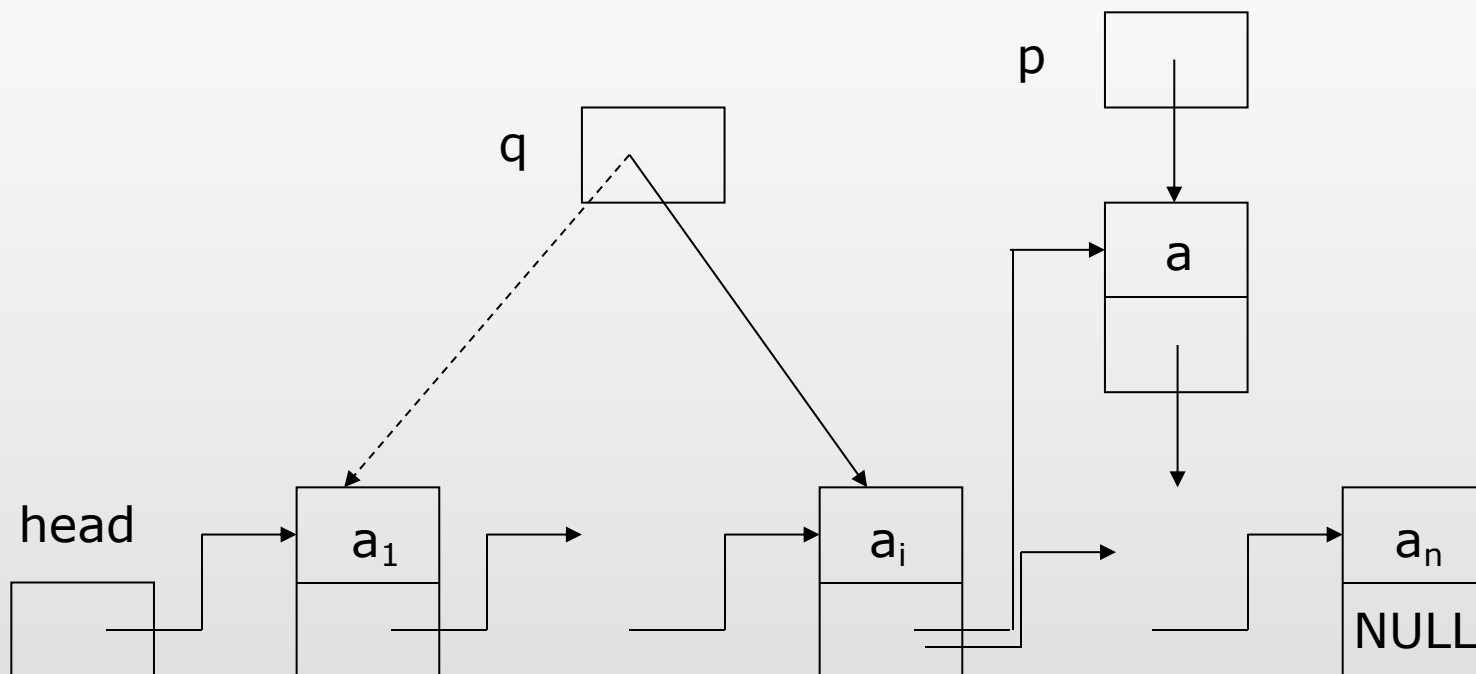
```
  q->next = p; //把新结点指定为q所指向结点的下一个结点。
```

```
}
```

```
else //链表中没有第i个结点。
```

```
  cout << "没有第" << i << "个结点\n";
```

- 图示为：



在链表中删除一个结点

下面的操作假设链表不为空，即：head != NULL

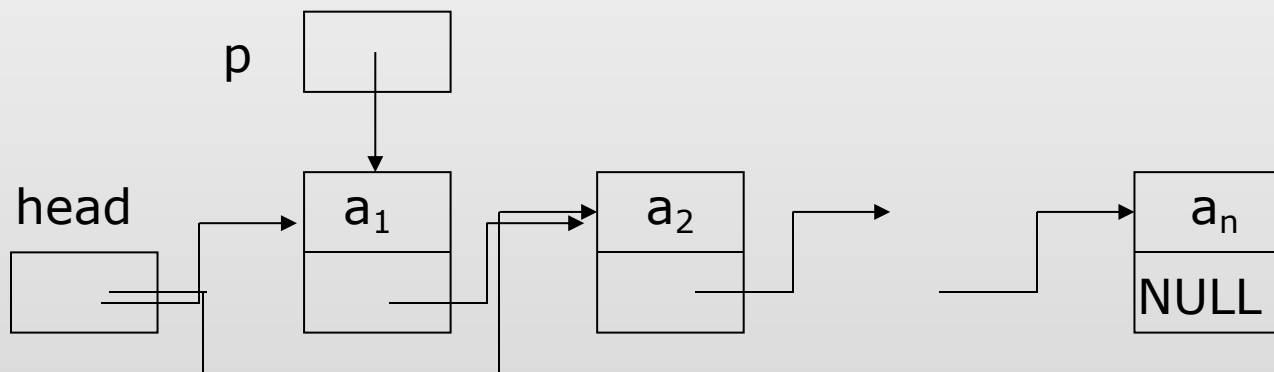
- 如果删除链表中第一个结点，则进行下面的操作：

Node *p=head; //p指向第一个结点。

head = head->next; //头指针指向第一个结点的下一个结点。

delete p; //归还删除结点的空间。

- 图示为：



- 如果删除链表的最后一个结点，则进行下面的操作：

```
Node *q1=NULL,*q2=head;
```

```
//循环查找最后一个结点，找到后，q2指向它，q1指向它的前一个结点。
```

```
while (q2->next != NULL)
```

```
{ q1 = q2;
```

```
    q2 = q2->next;
```

```
}
```

```
if (q1 == NULL) //链表中只有一个结点。
```

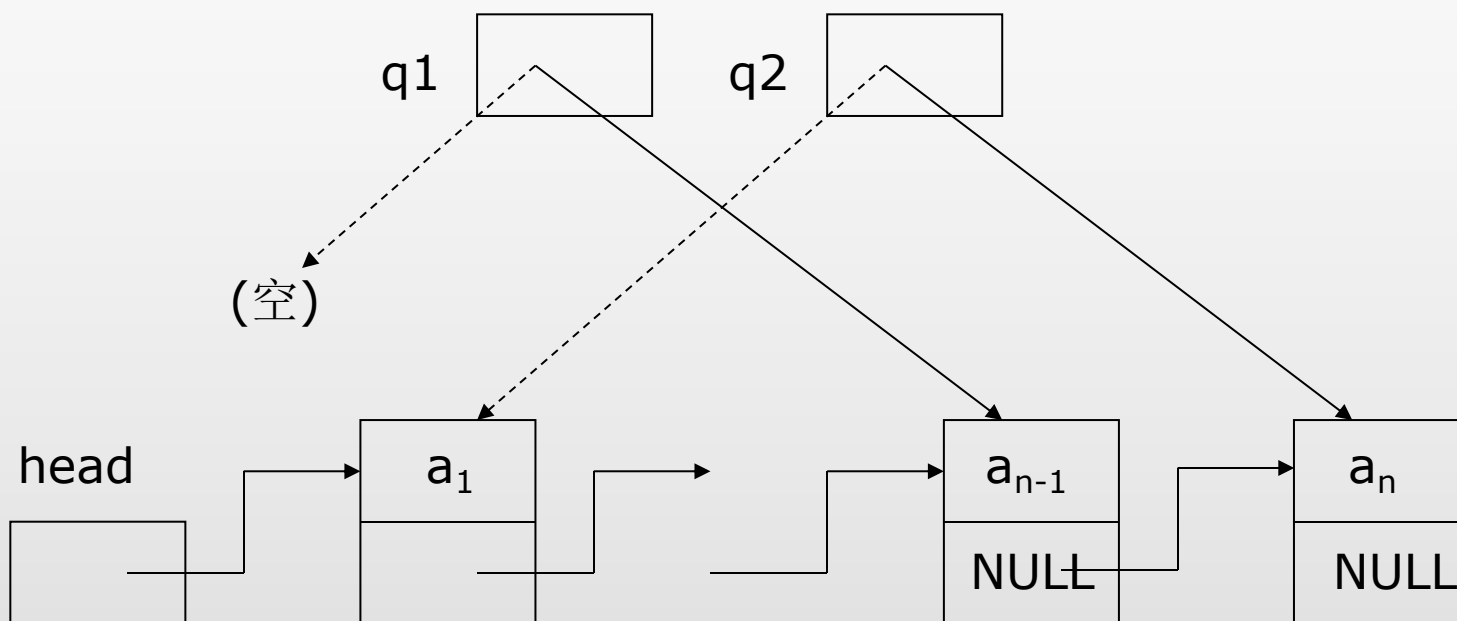
```
    head = NULL; //把头指针置为NULL。
```

```
else //存在倒数第二个结点。
```

```
    q1->next = NULL; //把倒数第二个结点的next置为NULL。
```

```
delete q2; //归还删除结点的空间。
```


- 图示为：



- 如果删除链表中第 i ($i > 0$) 个结点 a_i , 则进行下面的操作:

```
if (i == 1) //要删除的结点是链表的第一个结点。
```

```
{ Node *p=head; //p指向第一个结点。
```

```
    head = head->next; //head指向第一个结点
```

```
                //的下一个结点。
```

```
    delete p; //归还删除结点的空间。
```

```
}
```

```
else //要删除的结点不是链表的第一个结点。
{ Node *p=head; //p指向第一个结点。
  int j=1; //当前结点的序号，初始化为1
  while (j < i-1) //循环查找第i-1个结点。
  {   if (p->next == NULL)
      break; //当没有下一个结点时，退出循环
      p = p->next; //p指向下一个结点
      j++; //结点序号加1
  }
  if (p->next != NULL) //链表中存在第i个结点。
  {   Node *q=p->next; //q指向第i个结点。
      p->next = q->next; //把第i-1个结点的next
                          //改成第i个结点的next
      delete q; //归还第i个结点的空间。
  }
  else //链表中没有第i个结点。
      cout << "没有第" << i << "个结点\n";
}
```

在链表中检索某个值a

```
int index=0;//用于记住结点的序号，初始化为0
//从第一个结点开始遍历链表的每个结点查找值为a的结点。
for (Node *p=head; p!=NULL; p=p->next)
{ index++; //记住结点的序号，下面输出时需要。
  if (p->content == a) break;
}
if (p != NULL) //找到了
  cout << "第" << index << "个结点的值为：" << a
  << endl;
else //未找到
  cout << "没有找到值为" << a << "的结点\n";
```

对输入的若干个数进行排序，在输入时，先输入各个数，最后输入一个结束标记（如：-1）

```
struct Node
{ int content; //代表结点的数据
  Node *next; //代表后一个结点的地址
};
extern Node *input(); //输入数据，建立链表，返回链表的头指针
extern void sort(Node *h); //排序
extern void output(Node *h); //输出数据
extern void remove(Node *h); //删除链表
int main()
{ Node *head;
  head = input();
  sort(head);
  output(head);
  remove(head);
  return 0;
}
```

```

#include <iostream>
#include <cstdio>
using namespace std;
Node *input() //从表尾插入数据
{ Node *head=NULL, //头指针
    *tail=NULL; //尾指针

    int x;
    cin >> x;
    while (x != -1)
    { Node *p=new Node;
      p->content = x;
      p->next = NULL;
      if (head == NULL)
        head = p;
      else
        tail->next = p;
      tail = p;
      cin >> x;
    }
    return head;
}

```

```

#include <iostream>
#include <cstdio>
using namespace std;
Node *input() //从表头插入数据
{ Node *head=NULL; //头指针
    int x;
    cin >> x;
    while (x != -1)
    { Node *p=new Node;
      p->content = x;
      p->next = head;
      head = p;
      cin >> x;
    }
    return head;
}

```

```

void sort(Node *h) //采用选择排序，小的往前放
{ if (h == NULL || h->next == NULL) return;
  //从链表头开始逐步缩小链表的范围
  for (Node *p1=h; p1->next != NULL; p1 = p1->next)
  { Node *p_min=p1; //p_min指向最小的结点，初始化为p1
    //从p1的下一个开始与p_min进行比较
    for (Node *p2=p1->next; p2 != NULL; p2=p2->next)
      if (p2->content < p_min->content) p_min = p2;
    if (p_min != p1)
    { int temp = p1->content;
      p1->content = p_min->content;
      p_min->content = temp;
    }
  }
}

void output(Node *h)
{ for (Node *p=h; p!=NULL; p=p->next)
  cout << p->content << ',';
  cout << endl;
}

```

```
void remove(Node *h)
```

```
{ while (h != NULL)
```

```
    { Node *p=h;
```

```
      h = h->next;
```

```
      delete p;
```

```
    }
```

```
}
```


用链表实现求解约瑟夫问题

```
#include <iostream>
using namespace std;
struct Node
{ int no; //小孩的编号
  Node *next; //指向下一个小孩的指针
};
int main()
{ int m, //用于存储要报的数
  n, //用于存储小孩的个数
  num_of_children_remained; //用于存储圈子里
                              //剩下的小孩个数
  cout << "请输入小孩的个数和要报的数： ";
  cin >> n >> m;
```

//构建圈子

Node *first,*last; //first和last用于分别指向第一个和
//最后一个小孩

first = last = new Node; //生成第一个结点

first->no = 0; //第一个小孩的编号为0

for (int i=1; i<n; i++) //循环构建其它小孩结点

{ Node *p=new Node; //生成一个小孩结点

p->no = i; //新的小孩结点的编号为i

last->next = p; //最后一个小孩的next指向新生成
//的小孩结点

last = p; //把新生成的小孩结点成为最后一个结点

}

last->next = first; //把最后一个小孩的下一个小孩
//设为第一个小孩

//开始报数

num_of_children_remained = n; //报数前的圈子中小孩个数

Node *previous=last; //previous指向开始报数的前一个小孩

while (num_of_children_remained > 1)

{ for (int count=1; count<m; count++) //循环m-1次

previous = previous->next;

//循环结束时，previous指向将要离开圈子的小孩的前一个小孩

Node *p=previous->next; //p指向将要离圈的小孩结点

previous->next = p->next; //小孩离开圈子

delete p; //释放离圈小孩结点的空间

num_of_children_remained--; //圈中小孩数减1

}

//输出胜利者的编号

cout << "The winner is No." << previous->no << "\n";

delete previous; //释放胜利者结点的空间

return 0;

}

指针与数组

- 使用指针来访问数组元素能提高效率。例如：

- 用下标访问数组元素

```
const int N=100;
int a[N];
for (int i=0; i<N; i++)
{ ... a[i] ... //这里需要计算a[i]的地址：
                // (char *)&a[0]+i*sizeof(int)
} //N次乘法 + 2N次加法
```

- 用指针访问数组元素

```
for (int *p=&a[0],*q=&a[N-1]; p<=q; p++)
{ ... *p ...
} //N次加法
```

获取数组的首地址

- 一维数组的首地址

int a[10]; //等价于： typedef int A[10]; A a;

- 通过数组首元素来获得。例如：

```
int *p;
```

```
p = &a[0];
```

或

```
p = a; //把一维数组a隐式类型转换成第一个元素的地址： &a[0]
```

```
p++; //加： sizeof(int)
```

- 通过整个数组获得。例如：

- A *q; //或int (*q)[10];

- q = &a; //整个数组的地址，它与&a[0]值相同，但类型不同

- q++; //加： 10×sizeof(int)

- 用于按行来访问二维数组

- 当创建一个动态的一维数组时，得到的是第一个元素的地址。例如：
 - `int n;`
 - `int *p;`
 - `.....`
 - `p = new int[n];` //创建一个由n个int型元素
态数组，返回第一个元素的
//地址，其类型为：`int *` //构成的一维动

●二维数组的首地址

`int b[5][10];` //等价于: `typedef int A[10]; A b[5];`
//或 `typedef int B[5][10]; B b;`

●通过第一行、第一列元素来获得。例如:

- `int *p;`
- `p = &b[0][0];` //或 `p = b[0];` (自动转换成 `&b[0][0]`)
- `p++;` //加: `sizeof(int)`

●通过第一行的一维数组获得。例如:

- `A *q;` //或 `int (*q)[10];`
- `q = &b[0];` //或 `q = b;` (自动转换成 `&b[0]`)
- `q++;` //加: `10 × sizeof(int)`, `q` 指向下一行

●通过整个数组获得 (在三维数组中使用)。例如:

- `B *r;` //或 `int (*r)[5][10];`
- `r = &b;`
- `r++;` //加: `5 × 10 × sizeof(int)`
- 在三维数组中使用

- 对于一个动态的n维数组，实际上是按一维动态数组来创建的，返回的首地址类型是去掉第一维后的数组指针类型。例如，下面创建一个动态的二维数组：

- `typedef int A[10];` //A表示一个由10个int型元素
//所构成的一维数组类型

- `int m;`

- `A *q;` //或：`int (*q)[10];`

- `q = new int[m][10];`

- //创建一个由m行10列的二维数组，

- //返回第一行的地址（类型为：`A *`）。

或

- `q = new A[m];`

- 当把一个二维数组传给一个函数时，编译器将把它转换成第一行的地址。例如：

```
void f(int p[][10], int lines)
//解释成： void g(int (*p)[10], int lines)
{ ... p[i][j]... //解释成：  *(*p+i)+j)
}
int main()
{ int b[20][10];
  g(b,20); //转换成： g(&b[0],20);
  .....
}
```

```
int sum(int *p, int n) //一维数组元素求和
{ int s=0;
  for (int *q=p+n; p<q; p++)
    s += *p;
  return s;
}
```

.....

```
int a[10],b[5][10];
```

.....

//a的所有元素的和

```
sum(a,10);
```

//b的第0行所有元素的和

```
sum(b[0],10);
```

//b的所有元素的和

```
sum(b[0],5*10);
```

```
int sum2(int (*p)[10], int n) //对行为10个int型  
                                //的二维数组求和
```

```
{ int s=0;  
  for (int (*q)[10]=p+n; p<q; p++)  
    s += sum(*p,10);  
  return s;  
}
```

.....

```
int a[10],b[5][10],c[4][20];
```

.....

```
sum2(&b[0],5); //或: sum2(b,5);
```

```
sum2(&a,1);
```

```
sum2(&c[0],4); //?
```

```
int sum3(int *p, int lin, int col) //任意二维数组
{ int s=0;
  for (int *q=p+lin*col; p<q; p++)
    s += *p;
  return s;
}
```

.....

```
int b[20][10],c[4][20];;
...sum3(?)... //参数如何写?
...sum3(&b[0][0],20,10)...
...sum3(&c[0][0],4,20)...
```

```
void f(int *p[], int n)
```

```
{ .....
```

```
}
```

```
.....
```

f(?) //能提供什么样的参数?

```
int *a[10];
```

```
.....
```

```
f(a,10);
```

函数main的参数

- 可以给函数main定义参数，其定义格式为：

```
int main(int argc, char *argv[]);
```

- argc表示传给函数main的参数的个数，
 - argv表示各个参数，它是一个一维数组，其每个元素为一个指向字符串的指针。
- 以“copy file1 file2”执行程序copy时，copy的函数main将得到参数：
 - argc: 3
 - argv[0]: "copy"
 - argv[1]: "file1"
 - argv[2]: "file2"

函数指针

- C++中可以定义一个指针变量，使其指向一个函数。
例如：
 - `double (*fp)(int);` //fp是一个指向函数的指针变量
或者
 - `typedef double (*FP)(int);`
 - `FP fp;`
- 对于一个函数，可以用取地址操作符&来获得它的内存地址，或直接用函数名来表示。例如：
 - `double f(int x) { ... }`
 - `fp = &f;` //或者，`fp = f;`
- 通过函数指针调用函数可采用下面的形式：
 - `(*fp)(10);` //或者，`fp(10);`

例：编写一个程序，根据输入的要求执行在一个函数表中定义的某个函数。

```
#include <iostream>
#include <cmath>
using namespace std;
const int MAX_LEN=8;
typedef double (*FP)(double);
FP func_list[MAX_LEN]={sin,cos,tan,asin,acos,atan,log,log10};
int main()
{ int index;
  double x;
  do //循环以获得正确的输入
  {      cout << "请输入要计算的函数(0:sin 1:cos 2:tan 3:asin\n"
          << "4:acos 5: atan 6:log 7:log10):";
        cin >> index;
  } while (index < 0 || index > 7);
  cout << "请输入参数： ";
  cin >> x;
  cout << "结果为： " << (*func_list[index])(x) << endl;
  return 0;
}
```


向函数传递函数

- 函数的形参定义为一个函数指针类型，调用时的实参为一个函数的地址。例如：

```
int f(int);
int g(int);
int func(int (*fp)(int x)) //参数为一个函数指针类型。
{ int i;
  .....
  ...(*fp)(i)... //或fp(i); 调用形参fp所指向的函数。
  .....
}
int main()
{ .....
  ...func(&f)... //或func(f); 调用函数func, 把f作为参数传给它。
  ...func(&g)... //或func(g); 调用函数func, 把g作为参数传给它。
  .....
}
```

- 再例如，下面的函数integrate计算任意一个一元可积函数在一个区间上的定积分：

```
double integrate(double (*f)(double),  
                 double a, double b)  
{ .....  
}
```

- 下面的函数调用分别用于计算函数my_func、sin和cos在区间[1,10]、[0,1]和[1,2]上的定积分：

```
integrate(my_func,1,10);  
integrate(sin,0,1);  
integrate(cos,1,2);
```

多级指针

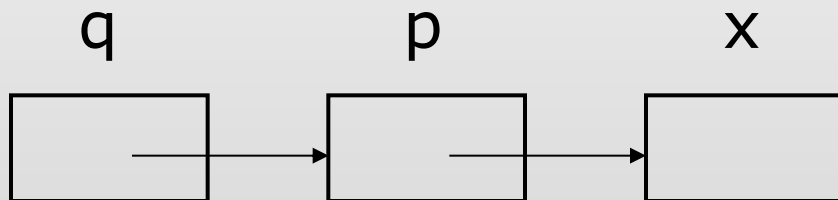
- 如果一个指针变量所指向的数据的类型为指针类型，则为多级指针，例如：

```
int x=0;
```

```
int *p=&x;
```

```
int **q=&p; //q是个多级指针
```

- 访问变量x：x、*p、**q
- 访问变量p：p、*q



交换两个指针变量值的函数

```
#include <iostream>
using namespace std;
void swap(int **x, int **y)
{ int *t;
  t = *x;
  *x = *y;
  *y = t;
}
int main()
{ int a=0,b=1;
  int *p=&a,*q=&b;
  cout << *p << ',' << *q << endl; //p指向a, q指向b。输出： 0,1
  swap(&p,&q);
  cout << *p << ',' << *q << endl; //p指向b, q指向a。输出： 1,0
  return 0;
}
```

.....

void input(Node **h) //从表头插入数据建立链表, h返回头指针

```
{    int x;
    cin >> x;
    while (x != -1)
    { Node *p=new Node;
      p->content = x;
      p->next = *h;
      *h = p;
      cin >> x;
    }
}
```

```
int main()
{ Node *head=NULL;
  input(&head); //函数调用完之后, head指向新建链表的头吗?
  .....
```

```
}
```

C++的引用类型

- 引用类型用于给一个变量取一个别名。例如：

```
int x=0;  
int &y=x; //y为引用类型的变量，可以看成是x的别名  
cout << x << ',' << y << endl; //结果为： 0,0  
y = 1;  
cout << x << ',' << y << endl; //结果为： 1,1
```

- 在语法上，
 - 对引用类型变量的访问与非引用类型相同。
- 在语义上，
 - 对引用类型变量的访问实际访问的是另一个变量（被引用的变量）。
 - 效果与通过指针间接访问另一个变量相同。

- 对引用类型需要注意下面几点：

- 定义引用类型变量时，应在变量名加上符号 “&”，以区别于普通变量。

- `int &y=x;`

- 定义引用变量时必须要有初始化，并且引用变量和被引用变量应具有相同的类型。

- `int x;`

- `int &y=x;`

- 引用类型的变量定义之后，它不能再引用其它变量。

- `int x1,x2;`

- `int &y=x1;`

- `.....`

- `y = &x2; //Error`

引用类型作为函数的参数类型

- 提高参数传递的效率。例如：

```
struct A
{ int i;
  .....
};
void f(A &x) //x引用相应的实参
{ .....
  ... x.i ... //访问实参
  .....
}
int main()
{ A a;
  .....
  f(a); //引用传递，提高参数传递效率
  .....
}
```


- 通过形参改变实参的值。例如：

```
#include <iostream>
using namespace std;
void swap(int &x, int &y) //交换两个int型变量的值
{ int t;
  t = x;
  x = y;
  y = t;
}
int main()
{ int a=0,b=1;
  cout << a << ',' << b << endl; //结果为： 0,1
  swap(a,b);
  cout << a << ',' << b << endl; //结果为： 1,0
  return 0;
}
```

```
#include <iostream>
using namespace std;
void swap(int *&x, int *&y) //交换两个int *型指针变量的值
{ int *t;
  t = x;
  x = y;
  y = t;
}
int main()
{ int a=0,b=1;
  int *p=&a,*q=&b;
  cout << *p << ',' << *q << endl; //p指向a, q指向b; 输出: 0,1
  swap(p,q);
  cout << *p << ',' << *q << endl; //p指向b, q指向a; 输出: 1,0
  return 0;
}
```

.....

void input(Node *&h) //从表头插入数据，建立链表，h返回头指针

```
{    int x;
    cin >> x;
    while (x != -1)
    { Node *p=new Node;
      p->content = x;
      p->next = h;
      h = p;
      cin >> x;
    }
```

}

int main()

```
{ Node *head=NULL;
  input( head);
```

.....

}

常量的引用

- 通过把形参定义成对常量的引用，可以防止在函数中通过引用类型的形参改变实参的值。

```
struct A
{ int i;
  .....
};
void f(const A &x)
{ x.i = 1; //Error
  .....
}
int main()
{ A a;
  .....
  f(a);
  .....
}
```

```
void f(int *p)
{ .....
    int m;
    p = &m; //OK,
    ... *p ... //通过p可以访问实参以外的数据
}

void g(int &x)
{ int m;
    .....
    x = &m; //Error
    ... x ... //通过x只能访问实参
}

int main()
{ int a;
    f(&a);
    g(a);
}
```

```
void f(int *const p)
{ .....
    int m;

    p = &m; //Error
    ... *p ... //通过p只能访问实参
}
```

引用类型与指针类型的区别

- 引用类型和指针类型都可以实现通过一个变量访问另一个变量，但在语法上，
 - 引用是采用直接访问形式
 - 指针则需要采用间接访问形式
- 在作为函数参数类型时，
 - 引用类型参数的实参是一个变量的名字
 - 指针类型参数的实参是一个变量的地址
- 在定义时初始化以后，
 - 引用类型变量不能再引用其它变量
 - 指针类型变量可以指向其它的变量
- 引用类型一般作为指针类型来实现（有时又把引用类型称作**隐蔽的指针**，hidden pointer)
- 能够用引用实现的指针功能，尽量用引用！

END

实践课

- 编程题三道
- 本场景将使用一台Alibaba Cloud Elastic Compute Service ECS实例（云服务器），使用Vim编辑markdown文档，使用git进行版本控制，使用VIM编辑cpp源文件，使用g++编译。
 - 云服务器（Elastic Compute Service，简称ECS）
 - Vim是文本编辑器。
 - markdown 是一种文档格式，扩展名为.md，可用任意编辑器打开并编辑。
 - Git是版本控制工具。最常用的命令 `git clone [GITHUB上的项目地址]`
 - g++是C++编译器。使用命令 `g++ ./test1.cpp -o test1` 编译；`./test1` 执行。

实践课 SHELL界面

【测试场景】南京大学《面向对象编程基础》实验课 cloud shell...

详情

剩余体验时间:

03:58:48

结束体验

体验手册

云产品资源

实验 < >

体验云账号，创建资源后生成

收起 ^

子用户名称: u-c2p6zcqf@1010471854...

子用户密码: Th1Wu8Vh2Im6Ws5A

AK ID: LTAI5tLuUbHwq1yRd7vf...

AK Secret: FAIGzEmaAiFwcOJTy1Kkx...

注意:
若登录子账号，请打开隐私窗口进行登录。

一键复制子账号登录链接

1 实验流程

上一页

下一页

>_ 华东2(上海)i-uf69vzokp3y3fthfgnxa u-c2p6zcqf root@106.15.52.237C

>_ 2. root@iZuf69vzokp3y3fthfgnxaZ:~ ×

Welcome to Alibaba Cloud Elastic Compute Service !

Last login: Thu Oct 7 21:20:38 2021 from 118.31.243.221
[root@iZuf69vzokp3y3fthfgnxaZ ~]#

default

+

>_ 命令终端 已连接 华东2(上海) i-uf69vzokp3y3fthfgnxa 106.15.52.237:22 91wwmp1xm4 14 pt 5,35 24

实践课 IDE界面

developer.aliyun.com/adscscenario/exp/ea37002044c0427988bfaa4746ba2908

【测试场景】南京大学-陈明-《面向对象编程基础》实验课 IDE界... 详情 剩余体验时间: 03 : 41 : 59

结束体验

文件 编辑 选择 视图 帮助

资源管理器

- 打开的编辑器
 - test.cpp test.cpp
- ADC_421_1675128494019369_IE3S
 - .vscode
 - a.out
 - README.md
 - test.cpp
- 大纲
 - main()

test.cpp

```
1  #include<iostream>
2  using namespace std;
3  int main()
4  {
5      cout<<"hello"<<endl;
6      return 0;
7  }
```

终端 sh sh +

```
[admin@c9c8943c-4512-4b6c-86c2-1d31aa55e522-7bodd6477-xdtkx /home/admin/workspace/codeup
.aliyun.com/60bf16267db6c7317ae89f2c/workbench/repo 2021-10-07 2021100701373377]
$

[admin@c9c8943c-4512-4b6c-86c2-1d31aa55e522-7bodd6477-xdtkx /home/admin/workspace/codeup
.aliyun.com/60bf16267db6c7317ae89f2c/workbench/repo 2021-10-07 2021100701373377]
$
```

输出 调试控制台 终端

master 21 预览

行7, 列2 空格: 2 LF UTF8 C++ Linux

实践课

- 习题1

- 实现strcpy函数

- `char *strcpy(char* dest, const char *src)`

- 习题2

- $N \times N$ 幻方。（打印 $N=9$ 幻方） N 为奇数。在 $N \times N$ 方阵中，填入 $1, 2, \dots, N \times N$ 各个数，使得行、列、两对角线和相等。
 - 当 N 为奇数时，我们可以通过以下方法构建一个幻方：
 - 首先将1写在第一行的中间。之后，按如下方式从小到大依次填写每个数 $K (K=2, 3, \dots, N \times N)$ ：
 - 1.若 $(K-1)$ 在第一行但不在最后一列，则将 K 填在最后一行， $(K-1)$ 所在列的右一列；
 - 2.若 $(K-1)$ 在最后一列但不在第一行，则将 K 填在第一列， $(K-1)$ 所在行的上一行；
 - 3.若 $(K-1)$ 在第一行最后一列，则将 K 填在 $(K-1)$ 的正下方；
 - 4.若 $(K-1)$ 既不在第一行，也不在最后一列，如果 $(K-1)$ 的右上方还未填数，则将 K 填在 $(K-1)$ 的右上方，否则将 K 填在 $(K-1)$ 的正下方。

- 习题3

- 编程解决八皇后问题。（ 8×8 棋盘上放置8个皇后，任意横竖斜线上不能有两个皇后）

实践课

- 选择开始体验 （提示注册阿里云账号并登录）
- 单击屏幕右侧创建资源 （本次实验2小时）
- 资源创建完毕后， 使用命令安装git g++ vim工具
 - `yum install -y git gcc-c++ vim`
- 使用git工具进行版本控制，使用VIM编辑一个markdown文档，可参照左侧git说明帮助
 - `mkdir test` 建立文件夹test
 - `cd test` 进入文件夹test
 - `git init` 表示文件夹版本库初始化
 - `vim test1.cpp` 按i键入程序源文件（习题1），按ESC 再按：wq退出
 - `g++ ./test1.cpp -o test1` 编译test1程序
 - `./test1` 执行习题1程序
 - `vim test2.cpp` 按i键入程序源文件（习题2），按ESC 再按：wq退出
 - `g++ ./test2.cpp -o test2` 编译test2程序
 - `./test2` 执行习题2程序
 - `vim test3.cpp` 按i键入程序源文件（习题3），按ESC 再按：wq退出
 - `g++ ./test3.cpp -o test3` 编译test3程序
 - `./test3` 执行习题3程序
 - `vim README.md` 键入新内容（本次实验心得），
 - `git add .` 加入当前文件夹下所有文件到暂存区
 - `git config --global user.email "you@example.com"`
 - `git config --global user.name "Your Name"`

附加题

- 反转链表 【简单】

- 输入一个长度为 n 链表，反转链表后，输出新链表。
- 例如：链表 1 为 $9 \rightarrow 3 \rightarrow 7$ ，反转后，链表为 $7 \rightarrow 3 \rightarrow 9$

- 两个链表生成相加链表 【中等】

- 假设链表中每一个节点的值都在 $0 - 9$ 之间，那么链表整体就可以代表一个整数。
- 给定两个这种链表，请生成代表两个整数相加值的结果链表。
- 例如：链表 1 为 $9 \rightarrow 3 \rightarrow 7$ ，链表 2 为 $6 \rightarrow 3$ ，最后生成新的结果链表为 $1 \rightarrow 0 \rightarrow 0 \rightarrow 0$ 。

- N皇后问题 【较难】

- N 皇后问题是指在 $n * n$ 的棋盘上要摆 n 个皇后，
- 要求：任何两个皇后不同行，不同列也不再同一条斜线上，
- 求给一个整数 n ，返回 n 皇后的摆法数。
- 例如：8皇后摆法92

附：Markdown文档语法

一级标题

二级标题

斜体 **粗体**

- 列表项

- 子列表项

> 引用

[超链接](http://asdf.com)

![图片名](<http://asdf.com/a.jpg>)

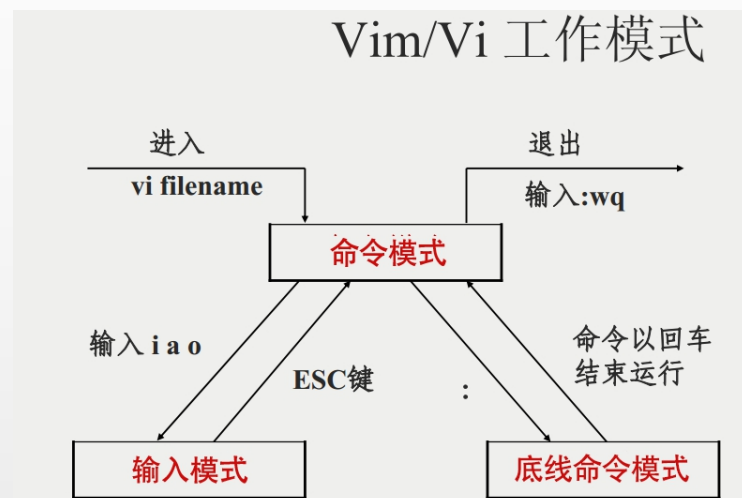
|表格标题1|表格标题2|

|-|-|

|内容1|内容2|

vim 的使用

- vim 共分为三种模式
 - 命令模式
 - 刚启动 vim，便进入了命令模式。按 ESC，可切换回命令模式
 - i 切换到输入模式，以输入字符。
 - x 删除当前光标所在处的字符。
 - : 切换到底线命令模式，可输入命令。
 - 输入模式
 - 命令模式下按下 i 就进入了输入模式。
 - ESC，退出输入模式，切换到命令模式
 - 底线命令模式
 - 命令模式下按下：（英文冒号）就进入了底线命令模式。
 - q 退出程序
 - w 保存文件



vim 常用按键说明

- 除了 `i`, `Esc`, `:wq` 之外, 其实 `vim` 还有非常多的按键可以使用。
 - 命令模式下:
 - **光标移动**
 - `j`下 `k`上 `h`左 `l`右
 - `w`前进一个词 `b`后退一个词
 - `Ctrl+d` 向下半屏 `ctrl+u` 向上半屏
 - `G` 移动到最后一行 `gg` 第一行 `ngg` 第n行
 - **复制粘贴**
 - `dd` 删一行 `ndd` 删n行
 - `yy` 复制一行 `nyy`复制n行
 - `p`将复制的数据粘贴在下一行 `P`粘贴到上一行
 - `u`恢复到前一个动作 `ctrl+r`重做上一个动作
 - **搜索替换**
 - `/word` 向下找word `? word` 向上找
 - `n`重复搜索 `N`反向搜索
 - `:1, s/word1/word2/g`从第一行到最后一行寻找 `word1` 字符串, 并将该字符串取代为 `word2`

vim 常用按键说明

- 除了 `i`, `Esc`, `:wq` 之外, 其实 `vim` 还有非常多的按键可以使用。
 - 底线命令模式下:
 - `:set nu` 显示行号
 - `:set nonu` 取消行号
 - `:set paste` 粘贴代码不乱序
- 【注: 把caps lock按键映射为ctrl, 能提高编辑效率。】

END

大作业：【任选一】

- 使用C++语言（可再结合其它语言），完成编程大作业。（部分）代码能跑在ECS平台上。
 - - 编程刷题100道 leetcode难度。
 - - 信息管理系统。（学生\图书）管理系统等等
 - - 游戏。五子棋、连连看等等
 - - 智能家居。与智能硬件（arduino、树莓派、舵机等等）相结合，使用C++语言（可再结合其它语言和其它现有框架），打造智能宿舍（声控\刷卡开关宿舍大门等等）
 - - 其它感兴趣或有意义的编程项目（比如经典项目-代理服务器等等）
- 提交：期末考试当日之前邮件提交完全部资料到13675128506@qq.com
 - - 项目报告（概要设计、详细设计、类图、流程图、用例图等等。项目运行截图或视频）。若选择leetcode刷题给出每道题的结题报告。
 - - 项目代码，git log记录。（代码仓库可放在git.nju.edu.cn上）
 - - 项目汇报PPT（本学期17周，倒数第3周提交PPT）