Hello World(第一个程序)

```c
#include <stdio.h>  // 包含另一个文件
int main()      // 定义主函数
{               // 函数体开始
    int num;    // 声明
    num = 1;    // 赋值表达式语句
    printf("Hello World\n"); // 调用函数
    return 0;   // return 语句
}               // 函数结束
```

- 程序分析：
  - `#include`：是预编译指令，<>尖括号，编译器会去系统配置的库环境变量和者用户配置的路径去搜索；""双引号，编译器会先在项目的当前目录查找，找不到后才会去系统配置的库环境变量和用户配置的路径去搜索。
  - `int main()`：前面的int表示此函数的类型是int类型（整型）,main是函数的名字，表示"主函数"。
  - `{···}`：这是花括号，一般而言，所有的C函数都会使用花括号标记函数题的开始与结束。
  - `/*····*/`：C语言中的注释，可同时注释多行。`//`单行注释
  - `int num`：这是声明，声明一个变量一般形式是关键字 标识符，例如`char str`。
  - `num = 1`：赋值表达式语句。
  - `printf()`：函数中双引号内的字符串"Hello World"按原样输出，但是\n是换行符，即在输出"Hello World"后，光标位置移到下一行的开头。
  - `return 0;`：的作用是：当main函数执行结束前将整数0作为函数值返回调用函数处。
- 代码规范
  - 1 程序块要采用缩进风格编写，缩进的空格数为4个。
  - 2 相对独立的程序块之间、变量说明之后 必须加空行。
  - 3 不把多个短语句写在一行中，即一行只写一条语句。
  - 4 对齐缩进为 4个空格字符。
  - 5 标识符的命名要清晰、明了，有明确含义，同时使用完整的单词或大家基本可以理解的缩写，避免使人产生误解，例`temp可缩写为 tmp;`

数据类型

- C数据类型
- 位，字节，字
  - 位（bit）：位是计算机存储的最小单位，简记为b，也称为比特（bit）计算机中用二进制中的0和1来表示数据，一个0或1就代表一位。位数通常指计算机中一次能处理的数据大小。
  - 2、字节（byte）：字节，英文Byte，是计算机用于计量存储容量的一种计量单位，通常情况下一字节等于八位，字节同时也在一些计算机编程语言中表示数据类型和语言字符，在现代计算机中，一个字节等于八位。

- 字 字是表示计算机自然数据单位的术语，在某个特定计算机中，字是其用来一次性处理事务的一个固定长度的位（bit）组，在现代计算机中，一个字等于两个字节。
- 1位=1比特 1字=2字节 1字节=8位
- 1个字长只有8位。从那以后，个人计算机字长增至16位、32位，直到目前的64位，计算机的字长越大，其数据转移越快，允许的内存访问也快的多

类型	存储大小	值范围
char	1 byte	-128 到 127 或 0 到 255
unsigned char	1 byte	0 到 255
signed char	1 byte	-128 到 127
int	4 bytes	-2,147,483,648 到 2,147,483,647
unsigned int	4 bytes	0 到 4,294,967,295
short	2 bytes	-32,768 到 32,767
unsigned short	2 bytes	0 到 65,535
long	4 bytes	-2,147,483,648 到 2,147,483,647
unsigned long	4 bytes	0 到 4,294,967,295

- 无符号(unsigned)与有符号(signed)的区别(int为例子) - 一般的，在你未定义是无符号(unsigned)时，编译器默认的是有符号型(signed) - 两者的区别与数值在内存中的存储有关。(原码、反码和补码) - 举例子为：
  计算机中的计算只有二进制加法。因此，计算机在计算时实际上是取它们的补码进行加法运算。

```
unsigned int a;
int b = -1;
a = b;
printf("a=%u",a);
```

  输出结果：
  a=4294967295
  让我们来分析一下，
  首先int型的-1，对应二进制
  取原码：10000000000000000000000000000001
  取反码：11111111111111111111111111111110
  取补码：11111111111111111111111111111111
  而unsigned int型最大值对应的补码也是它，因此在赋值给a后，就得到了它的最大值。

- 接下来我们测试一下数据类型的存储字节大小

```
#include<stdio.h>
int main()
```

```
{
    printf("Storage size for int : %d \n", sizeof(int));
    printf("Storage size for char : %d \n", sizeof(char));
    return 0;
}
```

输出：

```
Storage size for int : 4
Storage size for char : 1
```

- 运算符（主要）
  - 逻辑运算符

运算符	说明
&&	与运算，双目，对应数学中的“且”
!	非运算，单目，对应数学的“非”
||	或运算,双目，对应数学中的“或”

  - 算术运算符
  - 关系运算符
  - 条件运算符：?:例题 `int a = (3>5)? 3:5`
- 格式字符

  %d, %f, %c, %s

选择结构

- if-else语句

```
if(判断条件)
{
    // 如果条件为真，执行这个语句
}
else
{
    // 如果条件为假，执行这个语句
}
```

- switch语句

```
switch(表达式)
{
    case 常量1: 语句1;
    case 常量2: 语句2;
```

```
    case 常量3：语句3;
    .    .    .
    .    .    .
    .    .    .
    case 常量n：语句n;
    default: 语句n+1;
}
```

注意在每次的**case**语句后加上**break;**即在执行一个**case**子句后，应当用**break**语句跳出**switch**结构，终止**switch()**的执行

- 三目运算符:判断条件 ？ 真 ： 假;

循环结构

- while语句实现循环

```
while(表达式)
{
    // to do
}
```

while语句特点是先判断条件表达式，后执行循环体语句

- do…while语句

```
do{
    // to do
}while(表达式);
```

do…while语句的特点是先无条件的执行循环体再判断循环条件是否成立(注意在dowhile语句后加分号)

- for语句
  - 表达式1：循环变量的初始化
  - 表达式2：循环判断条件
  - 表达式3：循环递增条件

```
for(表达式1; 表达式2; 表达式3)
{
  // to-do
}
```

for循环中的“表达式**1**（循环变量赋初值）”、“表达式**2(**循环条件**)**”和“表达式**3**（循环变量增量）”都是选择项，即可以缺省，但分号**(;)**不能缺省

  - 例题 计算从1加到100的值

```c
#include <stdio.h>
int main()
{
    int sum = 0;
    for (int i = 1; i<=100; i++)
    {
        sum += i;
    }
    printf("%d\n", sum);
}
```

- 三种循环一般情况下可以互相代替
  - 什么时候用while或者do-while
    a.循环次数不确定
    b.循环的改变不是简单的递增或递减
- continue与break的区别
  - continue是只结束本次循环，而不是终止整个循环的执行
    break是终止整个循环过程，不再判断执行循环的条件是否成立

```c
#include <stdio.h>
int main()
{
    int i,j,n = 0;
    for (int i = 1; i <= 4; i++)
        for (int j = 1; j <= 5; j++,n++)
        {
            if (n % 5 == 0)
                printf("\n");
            if (i == 3 && j == 1)
                break;
            printf("%d\t", i * j);
        }
    printf("\n");
    return 0;
}
```

输出结果为

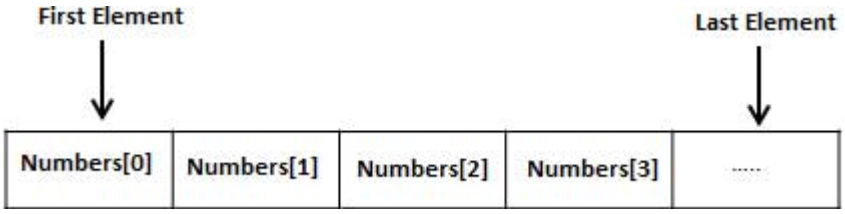
```
1       2       3       4       5
2       4       6       8       10

4       8       12      16      20
```

之后把break换成continue再运行一次
题目1：打印九九乘法表
题目2：打印左直角三角形

```
*
* *
* * *
* * * *
* * * * *
```

数组

- 什么是数组？
  - 数组是一组有序数据的集合，下标代表的就是数据在数组中的序号
  - 数组中的每一个元素都属于同一数据类型
  - 用一个数组名与有数字的方括号就可以确定数组中元素
- 一维数组

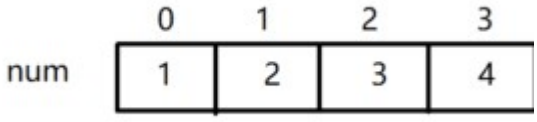


  - 一维数组的定义：数组在声明时必须指定其大小，因为数组的内存分配是在编译期间进行的

```
数据类型 数组名[整型变量表达式];
type arrayName[arraySize];
```

  - 内存布局与边界：一维数组在内存中连续存储的，下标是从0开始至arraySize-1,
  - 初始化数组：`int a[4] = {1,2,3,4};`,如果数据未满，剩余值默认为0
  - 或者直接利用数组的下标进行赋值

```
int main()
{
    int num[4];
    for (int i = 0; i < 4; i++)
    {
        num[i] = i;
    }
    return 0;
}
```



题：写一个冒泡排序 冒泡排序可视化

- 二维数组

a[0]
a[1]
a[2]
a[3]
a[4]

一维数组

a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]
a[3][0]	a[3][1]	a[3][2]	a[3][3]	a[3][4]
a[4][0]	a[4][1]	a[4][2]	a[4][3]	a[4][4]

二维数组

- 二维数组的定义

```
数据类型 数组名[整型变量表达式][整型变量表达式];
type arrayName[arraySize][arraySize];
int a[3][2];
```

- 初始化数组：
  - 分行赋初值：`int a[3][4] = {{1,2,3,1},{1,2,3},{1,2,4}};`
  - 将所有的数据写到一个花括号内:`int a[3][4] = {1,2,3,4,5,6};`
  - 对部分元素赋值：`int a[3][4] = {{1},{1,2},{1,2,3}};`
  - 对全部元素都赋值：`int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};`

    例题：将一个2*3的数组行列互换,存到另一个二维数组中

- 字符数组
  - 
  - 字符数组定义(与定义数值型数组类似)

    `char a[3] = {'a','b','c'};`
  - 字符串与字符数组的区别 c语言中字符串是作为字符数组来处理的，在用字符数组存储字符串时自动加一个`'\0'`作为结束符
  - 字符数组的输入输出
    - 逐个字符输入输出，用格式符`"%c"`
    - 将整个字符串一次输入或输出，用格式符`%s`

      1.输出的字符中不包含结束符`'\0'`

      2.用`"%s"`格式符输出字符串时，`printf`函数的输出项时字符数组名，而不是数组元素名
  - 字符串处理函数
    - 输出字符串函数-puts函数
    - 输入字符串的函数-gets函数
    -
  - 实现一个测字符串长度的函数

```
int main()
{
    char  a[] = "20 C Train";
    int i = 0;
    for (;a[i]; i++);
    printf("%d\n", i);
}
```

c语言培训下

- 指针
  - 指针及其使用方法
    - 初识指针
    - 指针能干什么
    - 指针的基本运算
  - 指向数组的指针
    - 指向一维数组的指针
    - 指向二维数组的指针
  - 指针数组
  - 数组指针
  - 二级指针
  - 指针在函数中的作用
    - 指针作为函数参数
    - 指针作为函数的返回值
    - 指向函数的指针

指针

指针及其使用方法

初识指针

指针是什么？

如果在程序中定义了一个变量，在对程序进行编译时，系统就会给这个变量分配内存单元。编译系统根据程序中定义的变量类型，分配一定长度的空间。内存区的每一个字节有一个编号，这就是“**地址**”。

由于通过地址能找到所需的变量单元，可以说，地址指向该变量单元，将地址形象化地称为“**指针**”。

如何定义指针变量？

- 指针本身就是一个变量，它存储的是数据在内存中的地址而不是数据的值。

```
数据类型* 变量名 或 数据类型 *变量名 //两者均可

int *pointer;
char *name;
```

- 这样就定义了两个指针变量，int 和 char 是这两个指针变量的数据类型，*表示这是指针变量
- **指针变量的初始化：** 每一个变量都有一个内存位置，每一个内存位置都定义了可使用连字号（&）运算符访问的地址，它表示了在内存中的一个地址。

```
int a = 10,*p;
p = &a;
```

```
//或者
int a = 10;
int *p = &a;
```

- 第一种我们可以理解为要定义一个int 型的指针 p，然后再对这个指针p取a的地址。
- 第二种方法其实和第一种的意思是一样的，这种方法可以理解为在定义指针变量p的同时将a的地址赋给p。
- 备注：在变量声明的时候，如果指针变量没有确切的地址可以赋值，为指针变量赋值NULL，养成一个良好的编程习惯。被赋值为NULL的指针被称为空指针。NULL指针是定义在标准库中的值为零的常量。

```
int  *ptr = NULL;
printf("ptr 的地址是 %p\n", ptr  );
return 0;
/*
结果:
ptr 的地址是 0x0
*/
```

理解“&”和“*”（取地址和指针运算符）

```
int a=100,b=10;
//定义整型变量a,b，并初始化
int *pointer_1,*pointer_2;
//定义指向整型数据的指针变量pointer_1, pointer_2
pointer_1=&a;   //把变量a的地址赋给指针变量pointer_1
pointer_2=&b;   //把变量b的地址赋给指针变量pointer_2
printf("a=%d,b=%d\n",a,b);      //输出变量a和b的值
printf("*pointer_1=%d,*pointer_2=%d\n",*pointer_1,*pointer_2);

/*
a=100,b==10
*pointer_1=100,*pointer_2=10
*/
```

指针基本运算

指针就是地址，地址在内存中也是以数的形式存在，所以指针也能进行基本运算。

```
int a;
int *p = &a;
printf("%p\n",p);
p++;
printf("%p\n",p);
p -= 2;
```

```
printf("%p\n",p);
return 0;
/*
000000000062FE44
000000000062FE48
000000000062FE40
*/
```

指向数组的指针

指向一维数组的指针

- 数组中的每个数据都会保存在一个储存单元里面，只要是储存单元就会有地址，所以就可以用指针保存数组储存单元的地址。

为指针赋数组数据的地址

```
int *p = NULL;
int num[5] = {1,2,3,4,5};
for(int i = 0;i < 5; i++)
{
    p = &num[i];
    printf("%d ",*p);
}
```

使用数组名为指针赋值

- 第一种

```
int num[5] = {1,2,3,4,5};
int *p;
p = &num[0];
```

- 第二种：直接将数组名赋予指针，指针需要储存的数据就是地址，而num就代表该数组的首地址。

```
p = num;
```

- 对于*(num + i),num 是数组的首地址，指向数组的首元素，而num + i则是数组的第i个元素的地址，再加上指针运算符* 就得到了该元素的值
- array每次加一的时候，它的值都会增加sizeof(int)，加i的时候就增加i *sizeof(int)

```

```

```
int num[5] = {2,4,6,8,10};
for(int i = 0;i < 5;i ++)
{
    //通过数组下标遍历数组
    printf("%d",num[i]);
    //通过指针变量遍历数组
    printf("%d",*(num + i));
}
```

指向二维数组的指针

跟一维数组相似

```
int num[3][2] = {{1,2},{3,4},{5,6}};
int *p = &num[0][0];
```

- **注意：** 不能为指针直接赋予二维数组的数组名，即上面的代码不能写成： int *p = num;
- 假设定义一个二维数组：num[m][n];一个指针p指向了这个二维数组的首地址，那么对于数组的数据num[i][j](0 <= i < m,0 <= j < n) ，指针变量p要想指向这个数据,那么指针变量 p = p + n * i + j;

```
double arr[4][3] = {
{78.4,72.1,41.2},
{56.4,12.4,45.1},
{12.5,14.6,20.4},
{23.5,34.6,67.8}
}
double *p_d = &arr[0][0]; //指针变量的类型必须要跟数组类型一致
printf("二维数组中arr[3][2]位置上的数据为：%6.11f\n",*(p_d + 3 * 3 + 2));
```

指针数组

顾名思义： 保存指针的数组

- 一维指针数组的定义形式为：类型名 *数组名p[数组长度];

```
int *prt_array[10];
```

- 括号里面ptr_array[10]表示的是一个长度为10的数组，然后括号外面的* 说明数组的元素类型是 int* 的指针类型
- 实例

```

```

```c
int main()
{
 int a = 16, b = 932, c = 100;
//定义一个指针数组
 int *arr[3] = {&a, &b, &c};
printf("%d %d %d\n", *arr[0], *arr[1], *arr[2]);
return 0;
}
/*
16 932 100
*/
```

数组指针

顾名思义： 指向数组的指针

如果一个指针指向了数组，就称它为数组指针。

```c
int a[4][3] = {{0,2,3},{1,5,6},{2,3,4},{7,8,9}};
```

在概念上的矩阵是像这种矩阵的样子：

```
0 2 3
1 5 6
2 3 4
7 8 9
```

但实际上它在内存中是链式存储的：

```
0 2 3 1 5 6 2 3 4 7 8 9
```

二维数组可以分解成多个一维数组，a[0]包括a[0][0]、 a[0][1]、a[0][2] 三个元素

```
      a[][0]  a[][1]  a[][2]
a[0]    0       2       3
a[1]    1       5       6
a[2]    2       3       4
a[3]    7       8       9
```

这里的a 就是那四个一维数组的组名，接着 定义一个数组指针

```c
int (*p)[3] = a;
```

*括号里面的*代表p是一个指针，[3]代表这个 指针 p指向了类型为int[3]的数组*

- p指向数组a的开头，就是指向数组的第0行元素，p + 1 指向数组的第一行元素
- 所以 *(p+1) 就表示数组的第一行元素的值，有多个数据
- *(p+1) + 1表示第一行的第一个数据的地址

二级指针

顾名思义： 指向指针的指针

假设有一个 int 类型的变量 a，p1是指向 a 的指针变量，p2 又是指向 p1 的指针变量，它们的关系如下图所示：

用代码形式展现就是：

```
int a = 100;
int *p1 = &a;
int **p2 = &p1;
```

指针变量也是一种变量，也会占用存储空间，也可以使用&获取它的地址。C语言不限制指针的级数，每增加一级指针，在定义指针变量时就得增加一个星号*。p1 是一级指针，指向普通类型的数据，定义时有一个*；p2 是二级指针，指向一级指针 p1，定义时有两个*。

同样的道理也会有三级指针、四级（司机）指针等等

指针在函数中的作用

指针作为函数的参数

在c语言中实参和形参之间的数据传输是单向的“值传递”方式，也就是实参可以影响形参，而形参不能影响实参。指针变量作为参数也不例外，但是可以改变实参指针变量所指向的变量的值。

```
void swap2(int *px,int *py){
    int t;
    t=*px;
    *px=*py;
    *py=t;
}

int main()
{
    int a=1,b=2;
    int *pa=&a,*pb=&b;

    swap2(pa,pb);
    printf("a=%d,b=%d\n",a,b);
```

```
}

/*
a=2,b=1
*/
```

指针作为函数的返回值

指针函数

C语言允许函数的返回值是一个指针（地址），我们将这样的函数称为指针函数。下面的例子定义了一个函数strlong()，用来返回两个字符串中较长的一个：

```
#include <stdio.h>
#include <string.h>
char *strlong(char *str1, char *str2){
    if(strlen(str1) >= strlen(str2)){
        return str1;
    }else{
        return str2;
    }
}
int main(){
    char str1[30], str2[30], *str;
    gets(str1);
    gets(str2);
    str = strlong(str1, str2);
    printf("Longer string: %s\n", str);
    return 0;
}
/*
android
android-lab
Longer string: android-lab

*/
```

需要注意的是：函数运行结束后会销毁在它内部定义的所有局部数据，包括局部变量、局部数组和形式参数。

~~ ~~

- 结构体
  - 初始结构体
  - 结构体的使用
    - 结构体与函数
  - 结构体的应用——链表
    - 结构体变量指针
    - 链表概述

        - 链表操作
- 文件
    - 为什么要有文件
    - 文件基本操作

结构体

初识结构体

结构体是什么？

就比如说，你要写一个个人信息的录入系统，它会包含很多内容（姓名、性别、身高、体重、身份证号、年龄、财产、爱好、性取向 等），这个时候你如果一个一个定义就会很麻烦。

```
#include "stdio.h"
int main()
{
    char name[20] = "lier";
    int height = 175;
    int weight = 70;
    char sex = 'm';
    short age = 19;
    long  wealth  = 300000;
    printf(“李二的个人信息：\n");
    printf("姓名：%s,身高：%d,性别：%c,年龄：%d,财产：
%d\n",name,height,sex,age,wealth);
    return 0;
}
```

这个时候结构体就能发挥它的优势

```
struct Stu{   // struct 关键字   Stu名称
    int height;//身高
    int weight;//体重
    char sex;//性别
    int age; //年龄
    long wealth;
};
```

*这里说一下**typedef**这是一个重命名的关键字，如果结构体这样写*

```
typedef struct Stu{   // struct 关键字   Stu名称
    int height;//身高
    int weight;//体重
    char sex;//性别
    int age; //年龄
    long wealth;
}N;
```

```
typedef + 数据类型 + 你想要重命名的英文
```

就表明将这个定义的结构体重新命名为**N**

- 结构体变量的初始化

结构体也是一种数据类型，从某种程度上说与int等类似，属于同级，所以定义变量的方式也是一样的。

```
struct Stu stu1,stu2;  //这里定义了量的Stu类型的变量
```

- 结构体成员的赋值

结构体成员的获取形式为：

```
结构体变量名.成员名;
```

例如

```
Stu stu1;
stu1.age = 19;
stu1.height = 175;
stu1.sex = 'm';
stu1.wealth = 30000;
stu1.weight = 70;
printf("身高：%d,性别：%c,年龄：%d,财产：
%d\n",stu1.height,stu1.sex,stu1.age,stu1.wealth);
```

结构体的使用

- 结构体与数组

  结构体中的成员变量可以是数组，这样可以省略很多定义，这就是数组的用处了，在这里就不赘述了

- 结构体与指针

  结构体可以作为函数的参数传进子函数中，然后再子函数中使用

下面是一个输出函数

Node 是一个结构体，**print()**是一个子函数，这个字函数有一个**Node**类型的参数

```
void print(Node *head)
{
```

```
    Node *p = head;
    if (!p){
            printf("\n链表是空的! ");
    }else {
            while (p) {
                    printf("%d->", p->info);
                p = p->next;
            }
    }
    printf("\n");
}
```

函数的应用———链表

结构体变量指针

- 结构体变量指向自身

也就是说定义了一个结构体类型的指针，这个指针指向了结构体本身

```c
struct table{
    int i;
    char c;
    struct table *st;    //定义的结构体指针指向了本身
};
```

- 指向其它结构变量

  即将定义的两个结构体变量，比方说定义了 st1 和 st2两个结构体变量，只需要将st2 的地址 赋给 st1 的指针域，这样 st1 的指针就指向了 st2

```
int main()
{
    table st1 = {1,'a'};
    table st2 = {2,'b'};
    st1.st = &st2;

    //使用结构体变量输出st1自身的2个成员的值
    printf("%d %c\n",st1.i,st1.c);

    //使用结构体指针域所指向的结构体输出数值,即 st2 中的数值
    printf("%d %c\n",st1.st->i,st1.st->c);

    //使用结构体变量输出st2自身的2个成员的值
    printf("%d %c\n",st2.i,st2.c);
```

```
    return 0;
}
/*
1 a
2 b
2 b
*/
```

链表

链表的最小单元———节点

```
struct table
{
    int i;
    char c;
    struct table *next;
}
strcut table st1 = {1,'a'};
struct table st2 = {2,'b'};
st1.next = &st2;
```

动态创建链表

- 构造一个结构类型，此结构类型必须包含至少一个成员指针，此指针要指向此结构类型，
- 定义3个结构体类型的指针，按照用途可以命名为，p_head,p_rail,p_new
- 动态生成新的结点，为各成员变量赋值，最后加到链表当中

  struct node { short i; 数据域 char c; ///数据域 struct node *next; //指针域，用于指向下一个结点 }

定义结构体指针，不一定要在main函数中定义

```
struct node *p_head,*p_rail,*p_new ;  //如果加上typedef就更完美了
```

使用**malloc**函数冬天申请存储空间，声明形式

```
p_head = (struct node*)malloc(sizeof(struct node));
```

- (struct node*)类型
- malloc()申请空间
- sizeof() 申请的大小

- 需要注意的是，在使用完这个结构体以后要将申请的空间释放，调用的函数为free();

实例

构造结构体

```
struct node {
    short i;
    char c;
    struct node *next;
};
```

定义变量

```
struct node node1 = {1,'A'};
struct node node2 = {2,'B'};
struct node node3 = {3,'C'};
node1.next = &node2;
node2.next = &node3;
```

动态申请节点并添加到链表中

```
struct node *p_new;
p_new = (struct node *)malloc(sizeof(struct node));
p_new->i = 4;
p_new->c = 'd';
node3.next = p_new;
```

链表操作

插入

- 插入节点到第一个数据前面

```
struct node p_new = (struct node *)malloc(sizeof(struct node));  //创建新结点，并为其开辟空间
scanf("%d%c",&(p_new->i),&(p_new->c));  //录入结点数据
//插入节点
p_new->next = p_head-next;
p_head->next = p_new;
```

- 插入节点到链表中间

```
struct node p_new = (struct node *)malloc(sizeof(struct node));  //创建新结点，并为其开辟空间
p_new->i = 2;
p_new->c = 'B';

struct node *p_front = p_head->next;
p_new->next = p_front->next;
p_front->next = p_new;
```

- 插入节点到末尾

```
while(1)
{
if(p-next == NULL)
{
    p_rail = p;
    break;
}
    p = p->next;
}
p_rail->next = p_new;
p_tail = p_new;
```

- 删除链表中的结点

```
void del_list(struct node *p_head,int pos)
{
    strct node *p_front,*p_del;
    p_front = p_head;
    for(int i = 0;i <= pos - 1;i ++)
    {
        p_front = p_front->next;
    }
    p_del = p_front->next;
    p_front->next = p_del->next;
    free(p_del);
}
```

文件

为什么要有文件操作

两个没有解决的问题

- 不得不再次运行程序
    - 我们运行计算机上的程序，然后不断输入数据给程序，然后得到程序对程序的处理结果，如果关掉程序的话，再想看到那些数据，就不得不再次运行程序。而且如果数据量过大的话，没办法留住这些数据 不得不重新输入数据
    - 每次在循行程序的时候吗，每运行一次都要重新的从简盘再次录入数据，而文件的运用帮我们解决了这个繁琐的问题

文件操作

写入数据

想要让程序在文件中写入文件，在程序与文件建立关联的时候，必须保证打开方是可写的，有 4 中方式可以将数据写入文件当中

- 字符方式
- 格式化方式
- 字符串方式
- 二进制方式

1. 字符方式

   程序可以以字符为单位，一个字符一个字符的将数据写入到文件当中，需要的函数是 fputc(),声明如下：

   ```
   int fputc(char c,FILE *stream);
   ```

    - 参数c代表将要被写进去的字符
    - 参数stream是一个文件指针，只想被写入的文件

   ```
   #include <stdio.h>
   #include <stdlib.h>
   int main()
   {
       char ch; //定义一个字符串
       int i = 0;
       FILE *fp;
       fp = fopen("D:\\File\\MarkdownFile\\test.txt","w");
       while((ch = getchar()) != '\n')
       {
           i = fputc(ch,fp);  // 以字符为单位，写入到text.txt文件
           if(i == -1)
           {
               puts("字符写入失败！");
               exit(0);
           }
       }
   ```

```
    return 0;
}
```

2. 格式化方式

如果写入文件的内容有特定的格式要求，可以使用格式化的方式将数据写入到文本

stdio.h 提供了一个库函数 fprintf(),可以达到这个目的，声明如下：

```
int fprintf(FILE *stream,const char *format[, argument ] ...);
```

和 printf()的使用方法一致

- 参数的 stream 是指向将要被写入数据的文件的文件指针
- 参数 format 是格式化的字符串
- 参数 argument 是可选的，如果 format 中有格式符，argument 就是对应的变量
- 函数 fprintf() 返回实际输入到文件中的字符的个数

```
struct info
{
    short no;
    char name[10];
    char sex[6];
};

struct info info_st[3] ={
    {1,"baoqianyue","men"},
    {2,"lihao","men"},
    {3,"wanghao","men"}
};
for(int i = 0;i < 3;i ++)
{
    fprintf(fp,"No = %d\tname = %-8s\tsex = %-6s\n",info_st[i].no,
        info_st[i].name,info_st[i].sex);
}
```

3. 字符串方式

```
char c[100];
gets(c);
int value = fputs(c,fp);  //fp指向文件
if(value == -1)
{c
    puts("字符串写入失败！\n");
```

```
    exit(0);
}
```

4. 二进制方式

```
struct info
{
    short no;
    char name[10];
    char sex[6];
};

struct info info_st[3] ={
    {1,"baoqianyue","men"},
    {2,"lihao","men"},
    {3,"wanghao","men"}
};
```

int count = fwrite(info_st,sizeof(struct info),3,fp); //写入数据到文件

- info_st 结构体类型指针
- sizeof 大小
- 3 count 有几条数据
- fp 指向文件

读取数据

1. 字符方式：fgets()函数
2. 格式化方式：fscanf()函数
3. 字符串方式：fgets()函数
4. 二进制方式：fread()函数