

Day01

一. 指针

1. 什么是指针?

内存单元地址称为指针

`./a.out`

4G虚拟空间 1G内核 3G用户

1个内存单元 占 1个字节

`int * p;` 4个字节

2. 用指针修改变量的值

修改变量的值有两种方法

```
int a = 10;
```

1. 直接修改

```
a = 20;
```

2. 间接修改

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

```
*p = 20;
```

//笔试题

```
#include <stdio.h>
```

//函数调用，实参初始化形参

```
//int* m = p;
```

```
//int* n = q;
```

//执行完 函数体内的三行代码后，本质是交换了指针变量 m和n的指向

//执行完m 指向了 q, n指向了 p

```
void fun(int* m, int* n)
```

```
{
```

```
    int* temp = m;
```

```
    m = n;
```

```
    n = temp;
```

```
}
```

```
int main()
```

```
{
```

```
    int a = 3, b = 5;
```

```
    int* p = &a, *q = &b;
```

```
    fun(p,q);
```

```
    printf("a is %d    b is %d\n",a,b); // ? ? 3 5
```

```
    return 0;
```

```
}
```

3. 指针所占内存空间的大小

```
int* p; //sizeof(p) ? 4
char* p; //sizeof(p) ? 4
float* p; //sizeof(p) ? 4
```

4. 指针、数组、函数

```
//将一个一维数组传递给一个函数
void showArray(int* p, int n)
{
    int i;
    for(i = 0; i < n; i++)
    {
        printf("%d ",p[i]);
    }
    printf("\n");
}

int main()
{
    int a[5] = {1,2,3,4,5};
    int *p = a;

    showArray(a, 5); //传递整型数组，通常需要数组的首地址和元素个数n
    return 0;
}
```

如何定义一个指针，保存一个一维数组的首地址

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

int* p = a; //数组的名字a就是数组的首地址
int* q = &a[0]; //数组第一个元素的地址就是数组的首地址
```

值: $a[i] == p[i] == *(p+i) == *(a+i)$

地址: $\&a[i] == \&p[i] == p+i == a+i$

$p++$; //指针向后移动一个位置

$a++$; //语法错误

$a+=2$; //语法错误

4.1 传递数组的不同写法

方法一: 直接写成指针

```
#include <stdio.h>

void showArray(int* p, int n)
{
    int i;
    for(i = 0; i < n; i++)
    {
        printf("%d ",p[i]);
    }
    printf("\n");
}
```

```
int main()
{
    int a[5] = {1,2,3,4,5};
    showArray(a, 5); //传递整型数组，通常需要数组的首地址和元素个数n
    return 0;
}
```

方法二: int a[5]

```
#include <stdio.h>

void showArray(int a[5])//你这么写,本质上就是写了 int* a;,a是指针变量 void
showArray(int* a)
{
    int i;
    printf("sizeof(a)/sizeof(a[0]) is %d\n",sizeof(a)/sizeof(a[0]));
    //a是一个指针变量,占4个字节,所以是 4 / 4 == 1,无法的到元素的个数
    for(i = 0; i < sizeof(a)/sizeof(a[0]); i++)
    {
        printf("%d ",a[i]);
    }
    printf("\n");
}

int main()
{
    int a[5] = {1,2,3,4,5};
    showArray(a);
    return 0;
}
```

```
linux@ubuntu:~$ ./a.out
sizeof(a)/sizeof(a[0]) is 1
1 → 只循环了一次，只打印了 数组中的元素a[0]
```

方法三:int a[]

```
#include <stdio.h>

void showArray(int a[])//你这么写,本质上就是写了 int* a; void showArray(int* a)
{
    int i;
    printf("sizeof(a)/sizeof(a[0]) is %d\n",sizeof(a)/sizeof(a[0]));
    //a是一个指针变量,占4个字节,所以是 4 / 4 == 1,无法的到元素的个数
    for(i = 0; i < sizeof(a)/sizeof(a[0]); i++)
    {
        printf("%d ",a[i]);
    }
    printf("\n");
}

int main()
{
    int a[5] = {1,2,3,4,5};
    showArray(a);
    return 0;
}
```

```
}
```

```
linux@ubuntu:~$ ./a.out
sizeof(a)/sizeof(a[0]) is 1
1 ──> 只循环了一次，只打印了 数组中的元素a[0]
```

5. 指针、字符串、字符数组

```
//请说出下面的语法正确与否
char a[] = "hello"; //正确
char b[6] = "hello"; //正确
char c[5] = "hello"; // 错，字符串组成包含 '\0', 'h' 'e' 'l' 'l' 'o' '\0', 共6个字符，数组越界
char d[100] = "hello"; //正确
char* p = "hallo"; //正确
char* q = a; //
//请问a的类型是什么？ a是字符数组 也可以看成char* ,因为数组名字就是数组的首地址 &a[0] == a
== char*
//请问a[0]的类型是什么？ char
//请问&a[0]的类型是什么？ char*

char a[] = "hello";
char *p = a;
char b[100] = "hello";
char *q = b;
//sizeof(数组名) 求的是整个数组所占内存空间的大小 元素个数 * 单个元素的大小
//求字符串的长度，不包含 '\0', 只要有了字符串的首地址，就可以求长度
//指针变量大小，和类型没关系，占4个字节
sizeof(a) ? 6, 元素省略6不写，元素个数是6， a是数组，所以 6*sizeof(char) == 6
strlen(a) ? 5, 字符串长度不包含 '\0'
sizeof(p) ? 4, p是指针变量，占4个字节
strlen(p) ? 5, 字符串长度不包含 '\0'
sizeof(b) ? 100, a是数组，所以 100*sizeof(char) == 100
sizeof(q) ? 4, q是指针变量，占4个字节
strlen(q) ? 5, 字符串长度不包含 '\0'
strlen(b) ? 5, 字符串长度不包含 '\0'
```

#练习1:

```
//本质是自己编写一个函数，实现库函数strcat的功能
写一个字符串连接函数mystrcat(); //参数自己定义
将"hello " 和 "world" 合并成一个字符串
定义一个mystrcat函数实现 将两个字符串连接在一起
char a[100] = "helloworld\0";
char b[] = "world";
mystrcat(a,b); //将b数组中的字符串连接在a数组的后面
puts(a) //helloworld
//strlen, strcpy, strcmp, strcat
012345
//"hello "
//"world"
//连接的时候是从a数组中字符串 '\0' 位置的下标开始被赋值，连接
a[5] = b[0] //j = 0 i+j = 5
a[6] = b[1] //j = 1 i+j = 6
a[7] = b[2] //j = 2 i+j = 7
a[i+j] = b[j]
```

```

或者5
a[5] = b[0];
i++,j++;

.....
.....

```

```

#include <stdio.h>

void myStrcat(char* dest, char* src)
{
    //1.先找到a数组中字符串'\0'的位置下标
    int i,j;
    for(i = 0; dest[i] != '\0'; i++);
    //上面的循环结束,dest[i] == '\0',i就是'\0'位置的下标
    //2. 从'\0'开始执行字符串拷贝的思想,逐个赋值过去
    for(j = 0; src[j] != '\0'; j++)
    {
        dest[i] = src[j];
        i++;
    }
    dest[i] = '\0';//注意此处一定要拷贝'\0'
}

int main(int argc, const char *argv[])
{
    char a[100] = "hello";
    char b[] = "world";
    myStrcat(a,b);
    puts(a);
    return 0;
}

```

```

linux@ubuntu:~$ gcc test.c
linux@ubuntu:~$ ./a.out
helloworld

```

#练习2:

0123456789....

编写一个函数实现功能: 将字符串"Computer Science"
 然后从第一个字母开始间隔的输出该字符串, 用指针完成。
 打印输出: "Cmue cec"
 //注意实现打印输出效果即可, 不需要覆盖删除操作

```

#include <stdio.h>

void showString(char* s)//char* s = p;
{
    int i = 0;
    while(s[i] != '\0')

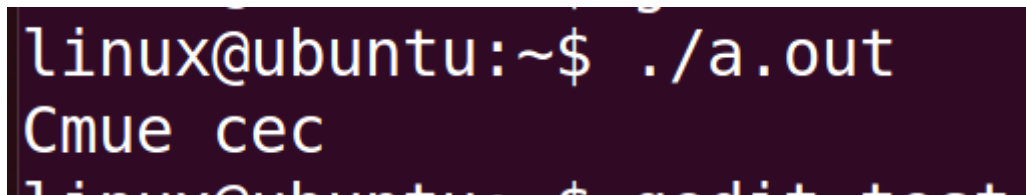
```

```

    {
        if(i % 2 == 0)
            printf("%c",s[i]);
        i++;
    }
    printf("\n");
}

int main()
{
    char* p = "Computer Science";
    showString(p);
    return 0;
}

```



```

linux@ubuntu:~$ ./a.out
Cmue cec

```

二. 数组指针

1. 数组指针本质?

概念：指向数组的指针 传递二维数组

本质是指针，重头戏在后面

有了本质，我们才能确定研究问题的角度

将本质确定为指针之后，研究角度

1. 指针的类型是什么??

2. 指针指向的类型是什么??

```
int* p;
```

//p的类型是什么?? int* , 用手挡住p

//p指向的类型是什么?? int, 用手挡住*p

```
int a[3][4];
```

```
int (*p)[4] =a;
```

//p的类型是什么?? 用手挡住p int (*)[4]

//p指向的类型是什么?? 用手挡住(*p), int [4]

整型指针，指向的类型是整型

字符指针，指向的类型是字符

数组指针，指向的类型是数组

```
int b[2][5];
```

定义一个数组指针指向b

```
int(*p)[5];
```

```

#include <stdio.h>
int main(int argc, const char *argv[])
{

```

```

int a[5] = {1,2,3,4,5};
//a的类型是什么 int [5]
//int [5] 代表的是元素个数为5个int的一维数组类型'
//a是一个普通变量的名字
int b;
//sizeof(数据类型) 和 sizeof(变量名) 等价
printf("%d %d\n",sizeof(int), sizeof(b));
printf("%d %d\n",sizeof(int [5]), sizeof(a));

return 0;
}

```

```

linux@ubuntu:~$ ./a.out
4 4
20 20

```

抛出问题：如何传递二维数组

```

#include <stdio.h>

void showArray(int* p, int row, int column)
{
    int i,j;
    for(i = 0; i < row; i++)
    {
        for(j = 0; j < column; j++)
        {
            printf("%d ",p[i][j]);
        }
        printf("\n");
    }
}

int main(int argc, const char *argv[])
{
    int a[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
    showArray(a,3,4);
    return 0;
}

```

```

linux@ubuntu:~$ gcc test.c
test.c: In function 'showArray':
test.c:10:21: error: subscripted value is neither array nor pointer nor vector
test.c: In function 'main':
test.c:19:2: warning: passing argument 1 of 'showArray' from incompatible pointer type [enabled by default]
test.c:3:6: note: expected 'int *' but argument is of type 'int (*)[4]'
linux@ubuntu:~$

```

函数形参是int* p,需要的int*, 但是给的类型是 int (*)[4]

2. 传递二维数组给函数

```

#include <stdio.h>

```

```

void showArray(int (*p)[4], int n)
{
    int i,j;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < 4; j++)
        {
            printf("%d ",p[i][j]);
        }
        printf("\n");
    }
}

int main(int argc, const char *argv[])
{
    //传递整型的一维数组,元素的类型是int, 所以函数的形参用的int* ,来保存数组首地址, 指向元素类型的指针
    //传递整型的二维数组,元素的类型是一维数组int [4],所以函数的形参用的 int(*) [4], 指向元素类型的指针
    int a[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
    showArray(a,3);
    return 0;
}

```

#练习3:

在main函数中定义一个二维数组

```

int a[2][3] = {{12,34,5},{23,4,34}};

void getMax()//参数自己定义,将main函数中的二维数组传递给getMax函数
{

}

int main()
{
    int a[2][3] = {{12,34,5},{23,4,34}};
    //调用getMax函数,得到二维数组中的最大值,打印输出
    return 0;
}

```

```

#include <stdio.h>
void getMax(int (*p)[3], int n, int* q)
{
    int i,j;
    //先假设最大值
    *q = p[0][0]; //q保存的是main函数中max的首地址,所以*q代表的就是main函数中的max
    //逐个比较找最大值
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < 3; j++)
        {
            if(p[i][j] > *q)
            {
                *q = p[i][j];
            }
        }
    }
}

```

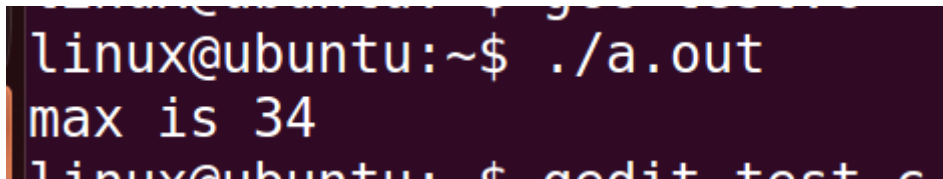


```

    }
}
}

int main()
{
    int max; //用来保存最大值
    int a[2][3] = {{12,34,5},{23,4,34}};
    getMax(a, 2, &max);
    printf("max is %d\n",max);
    return 0;
}

```



```

linux@ubuntu:~$ ./a.out
max is 34
linux@ubuntu:~$ gcc test.c

```

4. 行指针和列指针

此处的行指针和列指针并不是C语言里面的指针的标准概念(内存单元地址)

比如

```

int a[5] = {1,2,3,4,5};
int i = 2; //i是整型, i是几, 就对应的下标为几的元素, 此时把i可以看成指针(非C语言意义的指针),
           指向每个元素
printf("%d\n",a[i]);

```

```

//i 称为行指针
//j 称为列指针
#include <stdio.h>

int main()
{
    int a[3][4] = {{1,2,3,4},{3,3,3,3},{7,8,9,10}};
    int (*p)[4] = a; //定义一个数组指针, 保存二维数组的首地址
    int i,j;
    for(i = 0; i < 3; i++)
    {
        for(j = 0; j < 4; j++)
        {
            //p[i] == a[i] == *(p+i)
            //p[i][j] == a[i][j] == (*(p+i)+j)
            printf("%d ",*(p+i)+j);
        }
        printf("\n");
    }

    return 0;
}

```

```

int a[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
//二维数组由一维数组组成
//二维数组有几行，那么就有几个一维数组
//二维数组有3个元素个数4个int的一维数组组成
// int [4] //代表的就是元素个数为4个int的一维数组
// int a[3][4] 二维数组的元素类型 int [4]
a[0][0] a[0][1] a[0][2] a[0][3] // 第0行一维数组的名字a[0]
a[1][0] a[1][1] a[1][2] a[1][3] // 第1行一维数组的名字a[1]
a[2][0] a[2][1] a[2][2] a[2][3] // 第2行一维数组的名字a[2]

int b[5] = {1,2,3,4};
*(b+0) == b[0]
*(b+1) == b[1]

a[0][0] a[0][1] a[0][2] a[0][3] // 第0行一维数组的名字a[0]
*(a[0] + 0) --->a[0][0]
*(a[0] + 1) --->a[0][1]

*(a[0] + j) ---> a[0][j]

*(a[i] + j) ---> a[i][j]

将a[i]替换成*(a+i)

*(*(a+i) + j) == a[i][j]

int (*p)[4] = a;
p+1 //地址量增加多少?? 16 ,地址量增加多少，取决于p的类型 p的类型是int [4]

p+i //i代表的是行指针

*(p+i) //代表的是每一行一维数组的名字
*(p+1)+2 === &p[1][2]

*(*(p+1)+2) = p[1][2]

```

三. 指针数组

1. 指针数组的本质?

本质是数组，重头戏在后面

有了本质，我们才能确定研究问题的角度

将本质确定为数组之后，研究角度

1. 数组元素的类型是什么??

2. 数组的元素个数是几个??

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

//数组元素的类型是什么?? 用手挡住数组的名字和元素个数 挡住a[5],剩下的就是元素类型 int

//数组的元素个数是几个?? 5

整型数组，数组的元素类型是整型

字符数组，数组的元素类型是字符

指针数组，数组的元素类型是指针

2. 字符指针数组

//定义字符指针数组

//字符指针数组中的每个元素都是字符指针

//相当于一次性定义了5个字符指针(char*)变量 name[0] name[1] name[2] name[3] name[4]

```
char* name[5] = {"zhangsan", "xiaosi", "asan", "xiaoming", "xixi"};
```

name的类型?? 字符指针数组

name[0]的类型?? char* 字符指针 name[0]是数组的元素

sizeof(name[0]) ----> 4, 因为name[0]是一个指针变量，占4个字节

sizeof(name) ----> 元素个数*元素大小 == 5*sizeof(char*) == 5*4 == 20

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    char* name[5] = {"zhangsan", "xiaosi", "asan", "xiaoming", "xixi"};
```

```
    int i;
```

```
    for(i = 0; i < 5; i++)
```

```
        printf("%s\n", name[i]); //name[i]是指针，保存的是字符串的首地址，可以用%s直接将字符串打印输出
```

```
    return 0;
```

```
}
```

```
linux@ubuntu:~$ gcc test.c
linux@ubuntu:~$ ./a.out
zhangsan
xiaosi
asan
xiaoming
xixi
```

3. 整型指针数组

```
#include <stdio.h>
```

```
int main()
{
    int a = 3, b = 5, c = 7;
    //数组中的元素类型是int*
    //sizeof(d) == 12  3*sizeof(int*) == 3 * 4 == 12
    int* d[3] = {&a, &b, &c};
    int i;
    for(i = 0; i < 3; i++)
    {
        printf("%d\n", *d[i]); // [] 优先级 高于 * 所以 d[i]是一个整体, 代表的是数组的元
        素, int*
    }
}
```

```
linux@ubuntu:~$ gcc test.c
linux@ubuntu:~$ ./a.out
3
5
7
linux@ubuntu:~$ gedit test.c
```

```
int (*p)[4];
int* p[4];
```

四. 值传递和地址传递

问题

编写一个函数, 修改变量a的值

```
void setA() //将a的值修改为200
{

}

int main()
{
    int a= 10;

}
```

1. 值传递

值传递不能修改实参变量的值

```
#include <stdio.h>

//值传递
```

```

void setA(int a)//函数调用的时候main函数中实参变量a的值, 给到了形参变量a
{
    a = 200;//将形参变量a的值修改为200, 和main函数中的实参变量a无关
}

int main(int argc, const char *argv[])
{
    int a = 10;
    setA(a);
    printf("a is %d\n", a);//a is 10
    return 0;
}

```

2. 地址传递

地址传递可以修改实参变量的值

```

#include <stdio.h>

//地址传递
void setA(int* p)//int* p = &a;
{
    *p = 200;
}

int main(int argc, const char *argv[])
{
    int a = 10;
    setA(&a);
    printf("a is %d\n", a);//a is 200
    return 0;
}

```

3. 函数传递值得方式

一个函数想要给调用者, 传递值, 有几种方式

两种

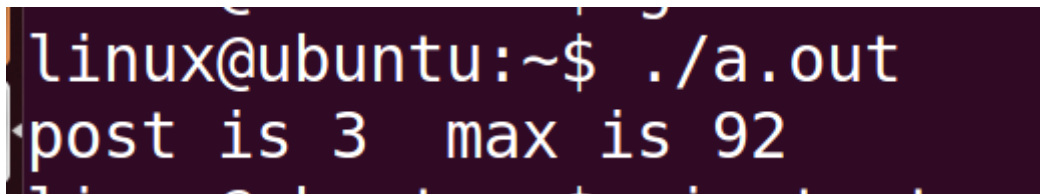
1. 返回值 return max;
2. 参数上的地址传递 getMax(a, 5, &max)

#练习4:

```
//请按照给定的函数类型，找到数组中的最大值的下标，在main函数中打印最大值和最大值的下标
//注意：main函数只要有了最大值的下标，就可以将最大值打印输出
//要求：不允许有返回值
void findMax()//函数的参数，自己思考
{

}
int main()
{
    int a[8] = {75, 28, 31, 92, 89, 73, 12, 84};
    return 0;
}
```

```
#include <stdio.h>
void findMax(int* p, int n, int* q)//int* q 参数上地址传递,得到最大值的下标
{
    int i;
    *q = 0;//因为假设的是第0个元素为最大值
    for(i = 1; i < n; i++)
    {
        if(p[i] > p[*q])
            *q = i;//记录当前最大值下标
    }
}
int main()
{
    int post;//用来保存最大值的位置下标
    int a[8] = {75, 28, 31, 92, 89, 73, 12, 84};
    findMax(a, 8, &post);
    printf("post is %d max is %d\n",post, a[post]);
    return 0;
}
```



```
linux@ubuntu:~$ ./a.out
post is 3 max is 92
```

五. 动态内存分配

动态内存分配：在程序运行的时候确定内存大小

在堆区手动申请，必须手动释放

malloc和free

malloc函数功能：在堆区申请一段连续的内存空间，返回值是申请空间的首地址，申请失败返回值是NULL

```
int* p = malloc(10*sizeof(int));
free函数，手动释放 free(p)
```

使用注意事项：

1. 手动申请之后，必须手动释放，malloc和free成对出现
2. malloc申请，存在失败的情况，必须对返回值做出判断

```
int n;
```

```
scanf("%d",&n);
int a[n] = { 0 };//错误，定义数组，长度必须是常量，n是变量，语法错误

int n;
scanf("%d",&n);
int* p = malloc(n*sizeof(int));

if(p == NULL)
{
    printf("malloc failed!!\n");
    return -1;
}
free(p);//手动释放
```

```
int a = 10;
int* p = &a;
*p = 200; //通过指针，修改了栈区,4个字节的存储空间，里面存储的是200

int* p = malloc(sizeof(int));
*p = 200; //通过指针，修改了堆区,4个字节的存储空间，里面存储的是200

int a[10];
int* p = a; //保存栈区连续空间的首地址40个字节
int* p = malloc(10*sizeof(int)); //保存堆区连续空间的首地址40个字节
```

#练习5:

```
struct student
{
    char name[20];
    int age;
    int score;
};
请编写程序，输入n个学生信息，将成绩最高的学生信息打印输出
//要求:存储n个学生信息空间，来自动态内存分配
```

```
#include <stdio.h>
#include <stdlib.h>

struct student
{
    char name[20];
    int age;
    int score;
};

//输入学生信息
void setStudentInof(struct student* p, int n)
{
    int i;
    for(i = 0; i < n; i++)
    {
        printf("请输入第%d个学生信息 姓名 年龄 成绩:\n",i+1);
        scanf("%s%d%d",p[i].name, &p[i].age,&p[i].score);
    }
}
```

```

}
//打印最高分学生信息
void showMax(struct student* p, int n)
{
    //找最大值
    int max = p[0].score;
    int i;
    for(i = 1; i < n; i++)
    {
        if(p[i].score > max)
            max = p[i].score;
    }
    //循环结束后找到了最大值
    for(i = 0; i < n; i++)
    {
        if(max == p[i].score)
            printf("最高分:%s %d %d\n",p[i].name,p[i].age, p[i].score);
    }
}

int main(int argc, const char *argv[])
{
    int n;
    struct student* p = NULL;
    printf("请您输入班级的人数:\n");
    scanf("%d", &n);
    //在堆区手动申请n学生信息的结构体空间
    p = (struct student*)malloc(n*sizeof(struct student));
    if(p == NULL)
    {
        printf("malloc failed!!\n");
        return -1;
    }

    setStudentInof(p, n);
    showMax(p, n);

    //手动释放
    free(p);
    return 0;
}

```



```
linux@ubuntu:~$ gcc test.c
linux@ubuntu:~$ ./a.out
请您输入班级的人数：
3
请输入第1个学生信息 姓名 年龄 成绩：
asan 19 100
请输入第2个学生信息 姓名 年龄 成绩：
lisi 28 97
请输入第3个学生信息 姓名 年龄 成绩：
haha 17 100
最高分:asan 19 100
最高分:haha 17 100
linux@ubuntu:~$ gedit test.c
```

作业1:

必须自己能够独立编写出自定义的字符串处理函数：
strlen() strcpy() strcat() strcmp()

作业2:

已知数组a[10]和b[10]中元素的值递增有序，用指针实现将两个数组中的元素按递增的顺序输出。

```
int a[10] = {1,6,8,13,24,35,38,45,49,56};
int b[10] = {3,9,18,25,29,35,39,41,51,52};
```

```
#include <stdio.h>

int main(int argc, const char *argv[])
{
    int a[10] = {1,6,8,13,24,35,38,45,49,56};
    int b[10] = {3,9,18,25,29,35,39,41,51,52};

    //同时遍历两个一维数组,谁小就打印谁
    int* pa = a;
    int* pb = b;

    while(pa < a+10 && pb < b+10)//一假即假 同假
    {
        if(*pa < *pb)
        {
            printf("%d ",*pa);
            pa++;//打印之后,指向下一个数组,pb指针不动,下一轮循环与pa的再比较
        }
        else/*pa >= *pb
        {
            printf("%d ",*pb);
            pb++;
        }
    }
```

```

}
//上面的循环结束之后,必然有一个数组已经遍历完成
if(pa == a+10)//说明b数组有剩余
{
    //遍历pb剩余部分
    while(pb < b+10)
    {
        printf("%d ",*pb);
        pb++;
    }
}
else//说明pa数组有剩余
{
    //遍历pa剩余部分
    while(pa < a+10)
    {
        printf("%d ",*pa);
        pa++;
    }
}
printf("\n");

return 0;
}

```

```

linux@ubuntu:~$ ./a.out
1 3 6 8 9 13 18 24 25 29 35 35 38 39 41 45 49 51 52 56
linux@ubuntu:~$ vi test.c

```

作业3:

编写一个函数is_within().它接受两个参数,一个是字符,另一个是字符串指针。其功能如果是字符在字符串中。就返回1(真);如果字符不在字符串中,就返回0(假)。

```

#include <stdio.h>

int is_within(char* s, char c)
{
    int i;
    for(i = 0; s[i] != '\0'; i++)
    {
        if(s[i] == c)
            return 1;//找到了
    }
    //上面的循环都结束了,都没有执行return 1,说明不存在
    return 0;
}

int main(int argc, const char *argv[])
{
    int ret = is_within("hello",'e');
    if(ret)//if(ret == 1)
        printf("存在!!\n");
    else
        printf("不存在!!\n");
}

```

```
return 0;
}
```

```
linux@ubuntu:~$ ./a.out
存在!!
```

作业4:

在main函数中定义二维数组，传递给函数 `showYangHui()`，实现打印杨辉三角形，
传递参数几，就打印几阶杨辉三角形

```
a[i][j] = a[i-1][j] + a[i-1][j-1];
```

@复习

1. 数组指针:本质是指针，指向数组的指针

数组指针语法目的: 传递二维数组，用数组指针

```
int a[5][8];
```

//定义一个数组指针，保存二维数组的首地址

```
int (*p)[8] = a; //p是一个指针变量 sizeof(p) == 4
```

p的类型是 用手挡住p `int (*)[8]`

p指向的类型 用手挡住 (*p), `int [8]`

```
p[i][j] == a[i][j] == *(p[i]+j) == *(*p+i)+j)
```

2. 指针数组:本质是数组，元素类型为指针的数组

数组中每个元素的类型都是指针 `int* char* double*`

指针数组:本质上就是一次性定义多个指针变量

```
char* p,*q,*r,*t,*m; //定义5个字符指针变量
```

```
char* a[5] = {"aa","bb","cc","dd","ee"}; //定义5个字符指针变量
```

//元素个数5个，用手挡住 a[5]剩下就是数组的元素类型 `char*`

a //字符指针数组

a[0]//类型 `char*` 字符指针

```
sizeof(a) ----> 5*sizeof(char*) == 20
```

Day02

一. 命令行传参

命令行传参: 在程序运行的时候, 给程序传递参数
shell脚本命令行传递参数 系统预设变量

```
bash test.sh aa bb cc
$0 ---> test.sh
$1 ---> aa
$2 ---> bb
$3 ---> cc
$# ---> 3
```

```
#include <stdio.h>
//标准main函数的格式需要背下来
//arg ---> argument 参数
//c count --> 参数个数
//v value --> 参数的值
//char* argv[] 等价于 char** argv
//为什么故意写成 char* argv[],函数参数,需要的是一个字符指针数组的首地址
// "./a.out" "aa" "bb" "cc"
//          argv[0]   argv[1]  argv[2]  argv[3]
// char* argv[] = {"./a.out", "aa",    "bb",    "cc" };//字符指针数组
// argv //字符指针数组 argv[0]类型 char*,是一个字符指针变量

int main(int argc, char* argv[])
{
    int i;
    printf("命令行参数个数: %d\n",argc);
    //将命令行上的所有参数打印删除,包括"./a.out"
    for(i = 0; i < argc; i++)
    {
        printf("argv[%d] -----> %s\n",i,argv[i]);
        //argv[i]的类型,是字符指针数组中的元素,类型是char*, 字符指针变量
    }
    return 0;
}
```

```
linux@ubuntu:~$ ./a.out aa bb cc
命令行参数个数: 4
argv[0] -----> ./a.out
argv[1] -----> aa
argv[2] -----> bb
argv[3] -----> cc
```

```
File Edit View Search Terminal Help
linux@ubuntu:~$ ./a.out
命令行参数个数: 1
argv[0] -----> ./a.out
linux@ubuntu:~$ ./a.out aa bb
命令行参数个数: 3
argv[0] -----> ./a.out
argv[1] -----> aa
argv[2] -----> bb
linux@ubuntu:~$ ./a.out aa bb cc dd ee ff
命令行参数个数: 7
argv[0] -----> ./a.out
argv[1] -----> aa
argv[2] -----> bb
argv[3] -----> cc
argv[4] -----> dd
argv[5] -----> ee
argv[6] -----> ff
linux@ubuntu:~$
```

#练习1:

利用命令行传递参数实现小型计算器

```
./a.out 35 - 5
30
./a.out 35 + 5
40
./a.out 35 / 5
7
./a.out 35 * 5
7
```

```
#include <stdio.h>
#include <stdlib.h>
//      0      1 2 3
// ./a.out 23 + 45
// "23" ---> 23
int main(int argc, char* argv[])
{
    //容错判断
    if(argc != 4)
    {
        printf("忘记传递参数了!! ./a.out 32 + 45\n ");
        return -1;
    }
    int num1 = atoi(argv[1]); //将命令行上的"23" --> 23
    int num2 = atoi(argv[3]); //将命令行上的"45" --> 45
```

```

char oper = *argv[2];
//argv[2] 保存的是" + "的首地址
//"+ " 由 '+' 和 '\0' 组成
//char* p = "+ ";
//printf("%c", *p); //打印输出 '+', 因为p保存的是字符串的首地址, 也就是第一个 '+' 的首地址
//argv[2]保存的是" + "的首地址, 也就是第一个字符的首地址
//*argv[2] 代表的就是第一个字符 '+'
switch(oper)
{
case '+':
    printf("%d + %d = %d\n", num1, num2, num1+num2);
    break;
case '-':
    printf("%d - %d = %d\n", num1, num2, num1-num2);
    break;
case '*':
    printf("%d * %d = %d\n", num1, num2, num1*num2);
    break;
case '/':
    printf("%d / %d = %d\n", num1, num2, num1/num2);
    break;
}
return 0;
}

```

```

linux@ubuntu:~$ ./a.out 23 + 45
23 + 45 = 68
linux@ubuntu:~$ ./a.out 23 - 45
23 - 45 = -22
linux@ubuntu:~$ ./a.out 23 / 45
23 / 45 = 0
linux@ubuntu:~$ ./a.out 23 * 45
忘记传递参数了!! ./a.out 32 + 45
linux@ubuntu:~$ ./a.out 23 \* 45
23 * 45 = 1035
linux@ubuntu:~$ gedit test.c
linux@ubuntu:~$ 注意乘法 传参

```

二. 递归函数

1. 什么是递归函数

一个函数直接或者间接的调用自己

2. 递归有两种方式

(1) 直接调用自己

```
#include <stdio.h>

//直接调用自己
void fun()
{
    int a[100000]; //加速程序, 栈空间不足
    printf("hello world!!\n");
    fun();
    printf("11111111\n");
    //11111111不会打印, 因为一直在调用自己, 每次调用fun函数都没有结束, 不会走到
    //fun();的下一行代码
}

int main(int argc, const char *argv[])
{
    fun();
    return 0;
}

(2) 间接调用自己
#include <stdio.h>

void fun2();
//间接调用自己
void fun1()
{
    printf("hello world!!\n");
    fun2();
}

void fun2()
{
    fun1();
}

int main(int argc, const char *argv[])
{
    fun1();
    return 0;
}
```

3. 递归函数注意事项

递归函数占空间较大, 容易造成死循环, 一定要有递归的结束条件

4. 递归案例

用递归函数, 求 $1+2+3+\dots+100$ 的和

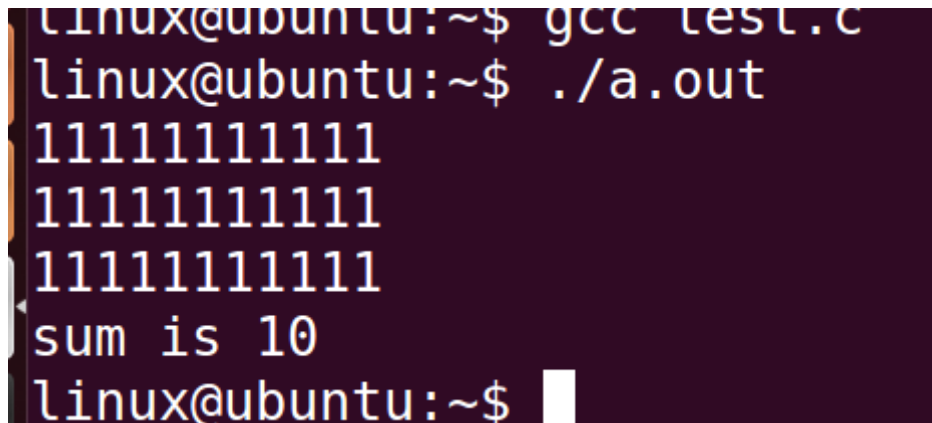
```
#include <stdio.h>

int getSum(int num)
{
    int result;
    //递归函数需要递归的结束条件
    if(num == 1)
        return 1;
```

```

    result = getSum(num-1) + num;
    printf("1111111111\n");
    return result;
}
// int sum = getSum(4)
// getSum(4) = getSum(3) + 4
// getSum(3) = getSum(2) + 3
// getSum(2) = getSum(1) + 2
// getSum(1) = 1
// getSum(2) = 1 + 2 == 3
// getSum(3) = 3 + 3 == 6
// getSum(4) = 6 + 4 == 10
int main()
{
    int sum = getSum(4);
    printf("sum is %d\n",sum); //sum is 10
    return 0;
}

```



```

linux@ubuntu:~$ gcc test.c
linux@ubuntu:~$ ./a.out
1111111111
1111111111
1111111111
sum is 10
linux@ubuntu:~$

```

#练习2:

编写一个递归函数实现函数式 $F(n)$ 是满足下面的结果
斐波那契数列

```

F(0) = 0
F(1) = 1
F(2) = 1
F(3) = 2
F(4) = 3
F(5) = 5
F(6) = 8
...
...
...

```

```

#include <stdio.h>

int F(int n)
{
    //递归结束条件
    if(n == 0 || n == 1)
        return n;
}

```



```

        //前两项的和 = 第三项
        return F(n-1) + F(n-2);

    }

    int main()
    {
        int i;
        for(i = 0; i < 20; i++)
        {
            printf("%d ",F(i));
        }
        printf("\n");
        return 0;
    }

```

```

linux@ubuntu:~$ ./a.out
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181
linux@ubuntu:~$ vi test.c

```

三. 二级指针

1. 什么是二级指针

二级指针：指向指针的指针

2. 定义及使用

```

int a = 10;
int* p = &a;
int** q = &p;

a == *p == **q //类型 int
&a == p == *q  //类型 int*
&p == q        //类型 int**
&q             //类型 int***

```

```

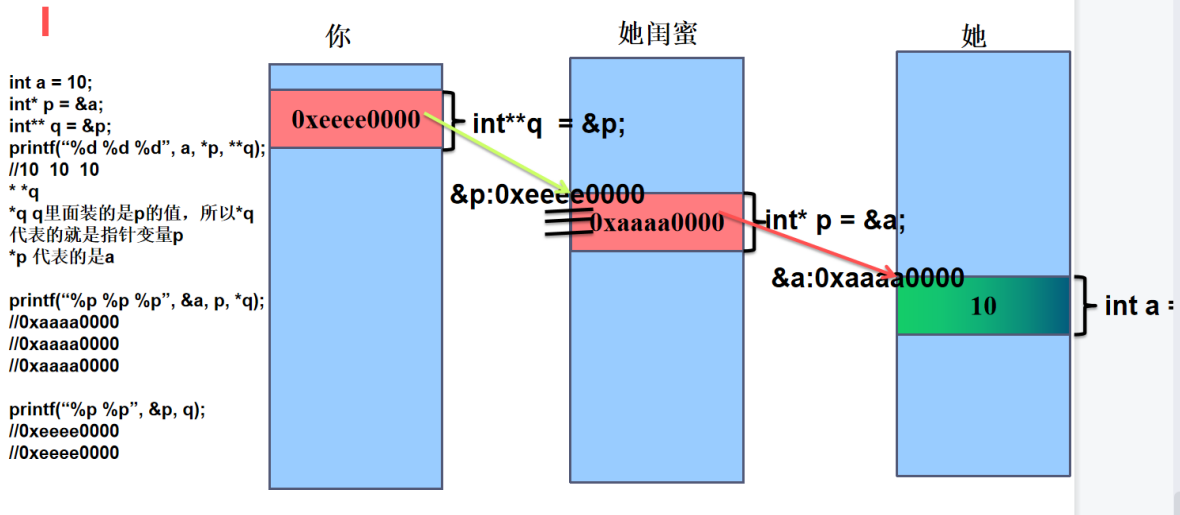
#include <stdio.h>

int main(int argc, const char *argv[])
{
    int a = 10;
    int* p = &a;
    int** q = &p; //对一级指针变量取地址,就是二级指针 int**
    printf("%d %d %d\n",a, *p, **q);
    printf("%p %p %p\n",&a, p, *q);
    printf("%p %p\n",&p, q);
    return 0;
}

```

```
linux@ubuntu:~$ gcc test.c
linux@ubuntu:~$ ./a.out
10 10 10
0xbfd8a3d4 0xbfd8a3d4 0xbfd8a3d4
0xbfd8a3d8 0xbfd8a3d8
```

二级指针: q是二级指针, 指向指针的指针



请说出下面代表的类型

- q ? // int**
- *q ? // int*
- **q ? // int
- &q ? // int***
- p ? // int*
- *p ? // int
- &p ? // int**
- a ? // int
- &a ? // int*

#笔试题:

```
请问下面的程序有问题吗? 如果有请指正
#include <stdio.h>
#include <stdlib.h>
void get_memory(int *q)
{
    q = malloc(10 * sizeof(int));
}

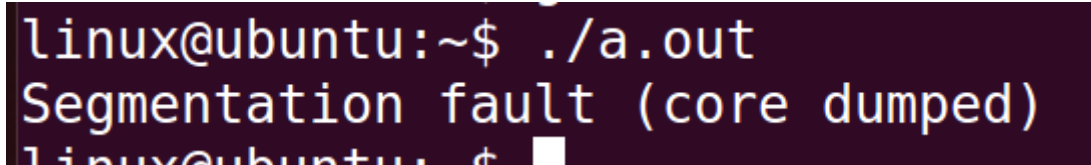
int main()
{
    int i;
    int *p = NULL;
    get_memory(p);
    for(i = 0; i < 10; i++)
    {
        p[i] = i;
    }
    for(i = 0; i < 10; i++)
```

```

{
    printf("%d\n", p[i]);
}
}

```

//上面程序目的，调用get_memory函数，实现将malloc函数申请空间的首地址，传递
 //给main函数中的指针变量p中保存
 //上面的程序运行会报段错误，因为值传递的方式不能修改实参变量p的值，调用get_memory函数之后，p依然是空指针



```

linux@ubuntu:~$ ./a.out
Segmentation fault (core dumped)
linux@ubuntu:~$

```

程序修正：通过地址传递的方式，来修改是参变量p

(1)参数上地址传递修改代码

```

#include <stdio.h>
#include <stdlib.h>
void get_memory(int** q)//int** q = &p;
{
    *q = malloc(10 * sizeof(int));//因为q中装的是main函数中p的地址,所以*q代表的就是main
    函数中的p
    if(*q == NULL)
    {
        printf("malloc failed!!\n");
        return;
    }
}

int main()
{
    int i;
    int *p = NULL;
    get_memory(&p);
    for(i = 0; i < 10; i++)
    {
        p[i] = i;
    }
    for(i = 0; i < 10; i++)
    {
        printf("%d\n", p[i]);
    }
    //手动释放
    free(p);
    return 0;
}

```

(2)返回值的形式修改代码

```
#include <stdio.h>
#include <stdlib.h>
int* get_memory()
{
    int* q = malloc(10*sizeof(int));
    if(q == NULL)
    {
        printf("malloc failed!!\n");
        return NULL;
    }

    return q; //将malloc申请空间的首地址返回
}
//get_memory函数,通过参数上地址传递的方式,给main函数一个地址
int main()
{
    int i;
    int *p = NULL;
    p = get_memory(); //返回值形式
    for(i = 0; i < 10; i++)
    {
        p[i] = i;
    }
    for(i = 0; i < 10; i++)
    {
        printf("%d\n", p[i]);
    }
    //手动释放
    free(p);
    return 0;
}
```

四. 指针函数

1. 什么是指针函数

本质是函数，返回值是指针类型(char* int* float* double* short* int**)的函数，就是指针函数

指针函数，本质是函数，研究问题的角度什么？？

1. 函数的参数有几个，类型是什么
2. 函数的返回值类型是什么

2. 指针函数举例

```
//下面的函数都是指针函数
int* get_memory()
void* malloc(int size)
char *strcpy(char *dest, const char *src);
char *strcat(char *dest, const char *src);
```

#练习3:

请编写一个函数，查找一个字符串中是否包含某个字符，如果存在请将第一次出现的字符的首地址返回，不存在，返回值NULL

```
"hello" 'e'
```

```
//返回值和参数类型自己定义
```

```
?? iswithin(??, ??)
```

```
{
```

```
}
```

```
#include <stdio.h>
```

```
//char* s 保存被查找字符串的首地址
```

```
//char c 保存查找的字符
```

```
char* iswithin(char* s, char c)
```

```
{
```

```
    /*
```

```
    //方法一
```

```
    //遍历字符串查找与c比较
```

```
    int i;
```

```
    for(i = 0; s[i] != '\0'; i++)
```

```
    {
```

```
        if(s[i] == c)
```

```
            return &s[i]; //return s+i
```

```
            //&a[i] == a+i == &p[i] == p+i
```

```
    }
```

```
    //如果上面的循环自然结束,一直没有执行return &s[i]
```

```
    //说明字符不存在
```

```
    return NULL;
```

```
    */
```

```
    //"hello"
```

```
    //方法二
```

```
    while(*s != '\0')
```

```
    {
```

```
        if(*s == c)
```

```
            return s;
```

```
            s++; //s = s + 1;
```

```
    }
```

```
    return NULL;
```

```
}
```

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

```
{
```

```
    char* ret = iswithin("hello", 'e');
```

```
    if(ret == NULL)
```

```
    {
```

```
        printf("e不存在!!\n");
```

```
    }
```

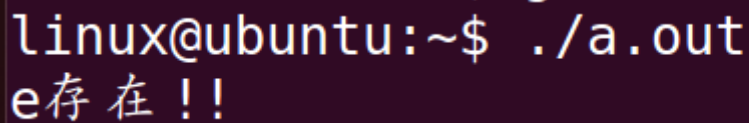
```
    else
```

```
    {
```

```
        printf("%c存在!!\n", *ret);
```

```
    }
```

```
return 0;
}
```



```
linux@ubuntu:~$ ./a.out
e存在!!
```

五. 函数指针

1. 什么是函数指针

函数指针：指向函数的指针

本质是指针，重头戏在后面

有了本质，我们才能确定研究问题的角度

将本质确定为指针之后，研究角度

1. 指针的类型是什么??

2. 指针指向的类型是什么??

```
int* p;
```

//p的类型是什么?? int* , 用手挡住p

//p指向的类型是什么?? int, 用手挡住*p

整型指针，指向的类型是整型

字符指针，指向的类型是字符

数组指针，指向的类型是数组

函数指针，指向的类型是函数

2. 函数类型

请说下面函数的类型

```
void fun(void); // void (void)
```

```
int getMax(int a, int b); // int (int,int)
```

```
char* strcpy(char* dest, char* src); // char* (char*,char*)
```

3. 定义函数指针指向函数

//如何定义一个函数指针，指向一个函数

```
void fun(void);
```

//定义函数指针，复制函数声明,将函数的名字，替换成 (*p)

```
void (*p)(void); //定义一个函数指针变量p
```

p = fun; //给函数指针变量p赋值，函数的名字就是函数在内存当中的首地址

```
void(*p)(void) = fun; //天生丽质，初始化函数指针变量p
```

```
int getMax(int a, int b);
```

```
int (*p)(int,int) = getMax;
```

#练习4:

```
void *malloc(int size);  
定义一个函数指针，可以指向malloc  
void* (*p)(int) = malloc;  
  
char *is_within(char *p, char c); //is_within函数的类型 char* (char*,char)  
定义一个函数指针，可以指向is_within  
char *(*p)(char *, char ) = is_within;  
  
int (*p)[4];  
p的类型 int (*)[4]  
p指向的类型 int[4]  
p的类型是什么？用手挡住的p char* (*)(char*, char)  
p指向的类型是什么？用手挡住的是(*p) char* (char*,char)
```

4. 用函数指针调用函数

```
#include <stdio.h>  
  
int add(int a, int b)  
{  
    return a + b;  
}  
int sub(int a, int b)  
{  
    return a - b;  
}  
int mul(int a, int b)  
{  
    return a * b;  
}  
int dev(int a, int b)  
{  
    return a / b;  
}  
  
int main(int argc, const char *argv[])  
{  
    //add sub mul dev 四个函数都是 int (int,int),属于同一类型的函数  
    //1. 定义一个函数指针变量,名字叫p  
    int (*p)(int, int);  
    //2. 给函数指针变量赋值, 函数的名字就是函数的首地址  
    p = add; //p指向了add函数  
    //3. 通过函数指针来调用函数,直接将指针变量p,当做函数的名字来使用,进行调用函数  
    int result = p(3, 5);  
    printf("result is %d\n", result);  
    p = mul;  
    result = p(3, 5);  
    printf("result is %d\n", result);  
    //p里面装的是哪个函数的地址,在调用函数的时候,调用的就是那个函数  
    return 0;  
}
```

```
linux@ubuntu:~$ ./a.out
result is 8
result is 15
linux@ubuntu:~$
```

5. 回调函数

回调函数：本质是函数，函数的参数是函数指针变量的函数，称为回调函数

```
#include <stdio.h>

int add(int a, int b)
{
    return a + b;
}
int sub(int a, int b)
{
    return a - b;
}
int mul(int a, int b)
{
    return a * b;
}
int dev(int a, int b)
{
    return a / b;
}

//回调函数:函数的参数有函数指针变量的函数,称为回调函数
//函数参数列表
//第1参数:函数指针变量 int (*p)(int,int)
//第2,3参数:int a, int b
//为了增加函数功能的灵活性,在通过函数指针调用函数的时候,需要两个int类型的参数
int process( int (*p)(int,int), int a, int b)// int (*p)(int,int) = add;
{
    int result = p(a,b);
    return result;
}

int main(int argc, const char *argv[])
{
    int result = process(add, 3, 5);
    printf("result add is %d\n",result);
    result = process(sub, 8, 7);
    printf("result sub is %d\n",result);
    result = process(mul, 3, 7);
    printf("result mul is %d\n",result);
    result = process(dev, 5, 2);
    printf("result dev is %d\n",result);
    return 0;
}
```



```
linux@ubuntu:~$ ./a.out
result add is 8
result sub is 1
result mul is 21
result dev is 2
```

6. 函数指针数组

本质是数组，数组中的每个元素都是函数指针类型

本质：一次定义多个函数指针变量

```
int (*p)(int, int);
int (*q)(int, int);
int (*r)(int, int);
int (*t)(int, int);
p = add;
q = sub;
r = mul;
t = dev;
//定义4个函数指针变量，名字叫 p q r t
int (*p)(int, int);
p的类型：挡住p， int (*)(int,int)
p指向的类型：挡住(*p) int (int,int)
```

```
int (*a[4])(int,int);//[] 优先级高于*,所以a和[]结合在一起，本质是数组
```

```
int a[5];//用手挡住数组名字和元素个数a[5]，剩下的就是元素的类型 int
int (*a[4])(int,int);//用手挡住数组名字和元素个数a[4]，剩下的就是元素的类型 int (*)
(int,int)
//一次性定义4个函数指针变量，名字分别叫做 a[0] a[1] a[2] a[3]
//整容手法
a[0] = add;
a[1] = sub;
a[2] = mul;
a[3] = dev;
//天生丽质
int (*a[4])(int,int) = {add, sub, mul, dev};
```

```
#include <stdio.h>

int add(int a, int b)
{
    return a + b;
}
int sub(int a, int b)
{
    return a - b;
}
int mul(int a, int b)
{
    return a * b;
```

```

}
int dev(int a, int b)
{
    return a / b;
}

//回调函数:函数的参数有函数指针变量的函数,称为回调函数
//函数参数列表
//第1参数:函数指针变量 int (*p)(int,int)
//第2,3参数:int a, int b
//为了增加函数功能的灵活型,在通过函数指针调用函数的时候,需要两个int类型的参数
int process( int (*p)(int,int), int a, int b)// int (*p)(int,int) = add;
                                                // int (*p)(int,int) = a[0];
{
    int result = p(a,b);
    return result;
}

int main(int argc, const char *argv[])
{
    //定义一个函数指针数组,初始化
    //[]优先级高于* ,所以a和[4]结合在一起,a是数组的名字,本质是数组
    //元素类型:用手挡住a[4]剩下的就是元素类型 int (*)(int,int)
    //元素个数:4个
    int (*a[4])(int,int) = {add, sub, mul, dev};
    int i;
    for(i = 0; i < 4; i++)
    {
        printf("result is %d\n", process(a[i],3, 5));
        printf("result is %d\n", a[i](3,5));//a[i]是一个函数指针变量,直接用函数指针变量
调用函数
    }

    return 0;
}

```

```

linux@ubuntu:~$ ./a.out
result is 8
result is 8
result is -2
result is -2
result is 15
result is 15
result is 0
result is 0
linux@ubuntu:~$ vim test.c

```

六. const关键字

const关键字：常量化意思

5 = 3; //5是常量，不可改，语法错误

```
#include <stdio.h>

int main(int argc, const char *argv[])
{
    const int a = 10; //变量a,经过const修饰后,变为常量,不可改
    a = 200; //此行代码报错, a是常量
    printf("a is %d\n", a);
    return 0;
}
```

```
linux@ubuntu:~$ gcc test.c
test.c: In function 'main': 编译报错, 因为变量a被const修饰
test.c:7:2: error: assignment of read-only variable 'a'
```

1. const修饰*p

```
#include <stdio.h>

int main(int argc, const char *argv[])
{
    const int a = 20;
    //const在*p前面,*p不可变 也就是p指向的内存空间里存储的值不可变
    //const int* p = &a;
    int const* p = &a;
    *p = 200; //此行报错, *p是常量
    printf("a is %d\n", a);
    return 0;
}
```

```
linux@ubuntu:~$ gcc test.c
test.c: In function 'main':
test.c:10:2: error: assignment of read-only location '*p'
```

2. const修饰p

```
#include <stdio.h>

int main(int argc, const char *argv[])
{
    int a = 20;
    int b;
    //const在p前修饰,p是常量,p里面存储的地址不可变
    int* const p = &a;
    *p = 200; //此行正确
}
```

```

    p = &b; //此行编译报错, 因为p是常量
    printf("a is %d\n", a);
    return 0;
}

```

```

linux@ubuntu:~$ gcc test.c
test.c: In function 'main':
test.c:11:2: error: assignment of read-only variable 'p'
linux@ubuntu:~$ vim test.c

```

3.const修饰p和*p

```

#include <stdio.h>

int main(int argc, const char *argv[])
{
    int a = 20;
    int b;
    //p和*p前都有const, 说明p和*p都是常量
    const int* const p = &a;
    //int const* const p = &a;
    *p = 200; //此行编译报错, 因为*p是常量
    p = &b; //此行编译报错, 因为p是常量
    printf("a is %d\n", a);
    return 0;
}

```

```

linux@ubuntu:~$ gcc test.c
test.c: In function 'main':
test.c:11:2: error: assignment of read-only location '*p'
test.c:12:2: error: assignment of read-only variable 'p'
linux@ubuntu:~$ gedit test.c

```

```

//const修饰, 不需要修改的指针上
void mystrcpy(char* dest, const char* src)
{
}

void mystrcat(char* dest, const char* src)
{
}

int strcmp(const char* s1, const char* s2)
{
}

```

七. 宏定义

1. 宏定义优点

- (1) 见名知意
- (2) 一改统改

2. 宏定义案例

案例1:

```
#define N 3.14
```

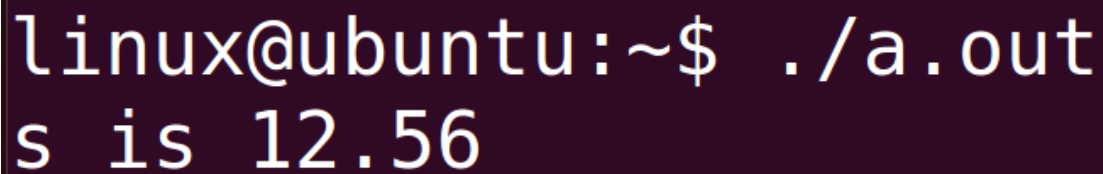
案例2:

```
#define MAIN ;}
```

```
#include <stdio.h>

#define N 3.14
#define R ;}

int main(int argc, const char *argv[])
{
    float s = N * 2 * 2;
    printf("s is %.2f\n", s);
    return 0
    R //无脑替换为 ;}, 编译通过
```



```
linux@ubuntu:~$ ./a.out
s is 12.56
```

案例3:

```
#define N 2
#define M N + 1
```

```
#include <stdio.h>

#define N 2
#define M N+1

int main(int argc, const char *argv[])
{
    //无脑替换
    printf("result is %d\n", M*M);
    //M*M == N+1*N+1 == 2+1*2+1 == 5
    return 0;
}
```

```
linux@ubuntu:~$ ./a.out
result is 5
```

案例4:

宏值如果有多行，用"`\`"连接

```
#include <stdio.h>

#define SHOW_STAR printf("  *\n");\
    printf(" ***\n");\
    printf("*****\n");

int main(int argc, const char *argv[])
{
    SHOW_STAR
    SHOW_STAR
    return 0;
}
```

```
linux@ubuntu:~$ ./a.out
  *
 ***
*****
  *
 ***
*****
```

3. 带参宏

案例:带参数宏()`sq`求一个数的平方

```
#include <stdio.h>

//带参宏依然是无脑替换
//sqr带参宏,求一个数的平方
#define sqr(x) (x)*(x)

int main(int argc, const char *argv[])
{
    int result = sqr(2+1);
    printf("result is %d\n", result); //result is 9
    return 0;
}
```

#练习5:

实现一个宏MAX，能求两个数的较大值然后写main函数测试

```
#include <stdio.h>

#define MAX(a,b) (a) > (b) ? (a) : (b)

int main(int argc, const char *argv[])
{
    int result = MAX(3,5);
    printf("result is %d\n", result);
    return 0;
}
```

```
linux@ubuntu:~$ ./a.out
result is 5
```

4. 带参宏与函数的区别

<p>带参宏</p> <p>宏是在编译的时候替换</p> <p>如果代码量小，调用频率又很高</p> <p>带参宏会导致代码量大</p>	<p>函数</p> <p>运行时调用函数，要分配栈空间</p>
---	--

```
#define N 10
int a[N]; //正确
```

八. 条件编译

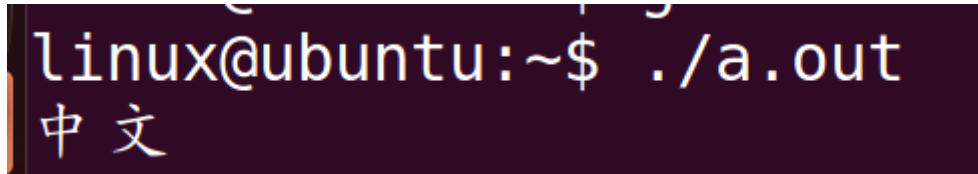
案例1:

```
#define DEBUG
#else
#endif
```

```
#include <stdio.h>

#define CHINESE

int main(int argc, const char *argv[])
{
#ifdef CHINESE
    printf("中文\n"); //如果定义了CHINESE宏，此行代码，参加编译
#else
    printf("English\n"); //如果没有定义CHINESE宏，此行代码，参加编译
#endif
    return 0;
}
```




```
linux@ubuntu:~$ ./a.out
中文
```

案例2:

在编译程序的时候，可以通过-D参数，定义一个宏
gcc -D CHINESE test.c

```
#include <stdio.h>
//注意此处没有定义CHINESE宏
int main(int argc, const char *argv[])
{
#ifdef CHINESE
    printf("中文\n");
#else
    printf("English\n");
#endif
    return 0;
}
```



```
linux@ubuntu:~$ gcc test.c
linux@ubuntu:~$ ./a.out
English
linux@ubuntu:~$ gcc -D CHINESE test.c 在编译程序的时候，定义一个宏CHINESE
linux@ubuntu:~$ ./a.out
中文
linux@ubuntu:~$
```

案例3:

```
#if 1 //1真
#endif

#if 0 //0假
#endif
```



```
File Edit View Search Terminal Help
1 #include <stdio.h>
2
3 int main(int argc, const char *argv[])
4 {
5     printf("1111111111\n");
6     printf("1111111111\n");
7     printf("1111111111\n");
8 #if 0          if 0 endif中间的代码不参加编译，相当于注释
9     printf("1111111111\n");
10    printf("1111111111\n");
11    printf("1111111111\n");
12 #endif
13    printf("1111111111\n");
14    printf("1111111111\n");
15    return 0;    改成if 1 可以参加编译
16 }
```

九.避免头文件重复包含

面试问题：请问如何避免头文件的重复包含??

```
ifndef ***** (宏)
#define ***** (宏)
    代码
#endif
```

.c结尾 C程序源文件
.h结尾 head 头文件

工作中，全局变量，要放在.c文件中，不要放在.h文件中
工作中 .h文件里面放什么

(1)函数的声明

```
extern int getMax(int,int);
```

(2)全局变量的声明

```
extern int max;
```

(3)结构体的定义

a.h

```
#ifndef _A_H
#define _A_H
int max = 200; //原则上，全局变量，不要放在.h文件中'
#endif
```

b.h

```
#ifndef _B_H
#define _B_H
#include "a.h"
#endif
```

test.c

```
#include <stdio.h>
#include "a.h"
#include "b.h"

int main()
{
    printf("max is %d\n",max);
    return 0;
}
```

十、static关键字作用

1. **static**修饰局部变量，此时的局部变量称为静态变量，作用域不变，生命周期直到整个程序的结束
静态变量多次调用，只初始化一次，未初始化，默认是0，存储在全局区
2. **static**修饰全局变量和函数的时候，全局变量和函数只能在本文件中使用，不能被其他外部文件调用

```
#include <stdio.h>

void fun()
{
    //静态变量，多次调用，只初始化一次，生命周期直到整个程序的结束
    static int x = 1; //加上static修饰后，此时的局部变量x称为静态变量
    x++;
    printf("x is %d\n",x);
}

int main(int argc, const char *argv[])
{
    //printf("main x is %d\n",x); //编译报错，因为x的作用域依然是局部作用域
    fun(); //x is 2
    fun(); //x is 3
    fun(); //x is 4
    return 0;
}
```

```
linux@ubuntu:~/0202$ ./a.out
x is 2
x is 3
x is 4
```

a.c

```
#include <stdio.h>

//一个.c文件想要使用另一个.c中的全局变量
//需要加上外部引用声明
extern int max; //作用告诉编译器,全局变量max在其他位置
extern void showMax();

int main(int argc, const char *argv[])
{
    printf("max is %d\n",max);
    showMax();
    return 0;
}
```

b.c

```
#include <stdio.h>
//全局变量
//static修饰全局变量max,max只能在本文件b.c中使用,不能被其他文件调用
static int max = 200;

//static修饰函数,函数只能在本文件中使用,不能被其他文件调用
static void showMax()
{
    printf("showMax(): max is %d\n",max);
}
```

编译: gcc a.c b.c

```
linux@ubuntu:~/0202$ gcc a.c b.c
/tmp/cc4jhyA6.o: In function `main':
a.c:(.text+0xb): undefined reference to `max'
a.c:(.text+0x21): undefined reference to `showMax'
collect2: ld returned 1 exit status
linux@ubuntu:~/0202$
```

作业:

思考下面p代表的意义,先说本质,再说具体研究问题角度
解题方法,先确定本质,就可确定用手挡住谁

//先确定本质，有了本质我们就知道研究问题的角度，有了研究问题的角度，就知道用手挡住谁

1. 见到`(*p)`的语法，立刻本质定义为指针
2. 如果是指针，挡住`p`，去找`p`的类型，挡住`*p`或`(*p)`去找`p`指向的类型
3. `[]`优先级高于`*`，来确定本质是数组
4. 如果是数组，`[]`里面的数字就是元素的个数，挡住数组名和元素个数 `a[5]`剩下的就是元素的类型
5. `()`优先级高于`*`，来确定本质是函数
如果是函数，用手挡住函数的名字，剩下的就是函数的类型，`()`内是参数个数和类型，`()`外返回值类型

本质：指针、数组、函数

```
(1) int p[3];           //本质数组 整型数组 3个元素， 元素类型int， 3个int类型的整型数组
(2) int (*p)[3];       //本质指针 数组指针 p的类型: int (*)[3], p指向类型 int [3],
                        //指向元素个数为3个int的一维数组类型的数组指针
(3) int* p[3];         //本质数组 指针数组 p的类型: int*,元素个数为3个int*的指针数组
(4) int *p(int);       //本质函数 指针函数 函数名字p,函数类型 int* (int)
(5) int (*p)(int);     //本质指针 函数指针 p的类型: int (*)(int), p指向类型 int (int)
(6) int *(*p)(int);    //本质指针 函数指针 p的类型: int *(*)(int), p指向类型 int *
(int)
(7) int (*p[3])(int);  //本质数组 函数指针数组 3个元素， 元素类型int (*)(int),
                        //3个int (*)(int)类型的函数指针数组
(8) int *(*p[3])(int); //数组 函数指针数组 3个元素， 元素类型int *(*)(int)
                        //3个int *(*)(int)类型的函数指针数组
                        //元素类型是函数指针，指向的函数是指针函数 int*
(int)
```

@复习

1. 命令行传递参数

标准main函数的格式

//argc 命令行参数的个数，包含 ./a.out

//argv 用来保存字符指针数组的首地址

```
int main(int argc, char** argv)
{
    //将命令行所有参数打印输出
    int i;
    //argv[i]的类型是 char*
    for(i = 0; i < n; i++)
        printf("%s\n",argv[i]);
}
```

//二级指针： 传递指针数组的时候，用二级指针

```
void showName(char** p, int n)
{
    int i;
    for(i = 0; i < n; i++)
        puts(p[i]);
}

int main()
{
    char* name[3] = {"asan","aaa","ccc"};
    //name[0]类型 char*
    //&name[0]类型 char**
    //&name[0] == name 数组的首地址
    showName(name, 3);
    return 0;
}
```

```
}
```

2. 递归函数

递归浪费空间，一定要注意递归的结束条件(容易造成死循环)

3. 二级指针：指向指针的指针

```
int a = 10;
int* p = &a;
int** q = &p;
a == *p == **q //int
&a == p == *q  //int*
&p == q        //int**
```

4. 宏定义：无脑替换，见名知意，一改统该

```
#define sqr(x) x*x
```

```
sqr(2+1) ==== 5
```

带参宏，不是函数，本质上还是无脑替换

Day03

一. 枚举

1. 数据类型

基本数据类型：char int short long float double 指针

构造数据类型：数组 结构体 共用体

空类型：void

2. 枚举类型定义

定义枚举类型的关键字是 enum

枚举就是整型

```
enum week
{
    Mon, //默认从0开始
    Tues,
    wed,
    Thurs,
    Fri,
    Sat,
    Sun
};
```

```
#include <stdio.h>
```

```
enum week
{
    Mon,
    Tues,
    wed,
    Thurs,
    Fri,
    Sat,
    Sun
};

int main()
{
    printf("%d %d %d %d %d %d %d\n", Mon, Tues, wed, Thurs, Fri, Sat, Sun);
    int n;
    scanf("%d", &n);
    switch(n)
    {
        case Mon:
            printf("星期一\n");
            break;
        case Tues:
            printf("星期二\n");
            break;
        case wed:
            printf("星期三\n");
            break;
        case Thurs:
            printf("星期四\n");
            break;
    }
    return 0;
}
```

```
linux@ubuntu:~$ gcc test.c
linux@ubuntu:~$ ./a.out
0 1 2 3 4 5 6
0
星期一
```

```
enum week
{
    Mon,
    Tues = 3,
    wed,
    Thurs = 3,
    Fri = 5,
    Sat,
    Sun
};

printf("%d %d %d %d %d %d %d\n", Mon, Tues, wed, Thurs, Fri, Sat, Sun);
    0 3 4 3 5 6 7
```

3. 枚举注意点

- (1) 枚举类型中，声明的第一个枚举成员默认值为0
- (2) 以后每个没有被赋值的枚举成员值将是前一个枚举成员的值加1得到的。
- (3) 定义枚举类型时，可以为枚举成员显示赋值。允许多个枚举成员有相同的值。
- (4) 没有显示赋值的枚举成员的值，总是前一个枚举成员的值+1

二. 共用体

共用体是多个成员变量，共用同一块内存空间，所有的成员变量，起始地址一样任意时刻，只能保证一个成员变量存储的数据是有效的

1. 共用体定义

```
共用体定义关键字 union
union A
{
    int a;
    double b;
    char c;
};
```

```
#include <stdio.h>

union data
{
    int a;
    double b;
    char c;
};

int main(int argc, const char *argv[])
{
    //定义一个共用体变量,名字叫s
    union data s;
    s.a = 10;
    printf("s.a is %d\n", s.a);
    s.b = 888.88;
    printf("s.b is %lf\n", s.b);
```

```
printf("s.a is %d\n",s.a);//s.a受到 s.b赋值的影响,因为共用体公用的是同一块内存,起始地址一样.相互干扰
return 0;
}
```

```
linux@ubuntu:~$ ./a.out
s.a is 10
s.b is 888.880000
s.a is 1030792151
```

b成员赋值后,影响了a成员原来的值,因为共用空间

2. 共用体占内存大小

```
//共用体的大小为最大的成员变量的大小
union data
{
    int a;
    double b;
    char c;
};

union data s;
sizeof(s) ----> 8 //共用体的大小为最大的成员变量的大小
```

```
#include <stdio.h>

union data
{
    int a;
    double b;
    char c;
    char name[21];
};

int main(int argc, const char *argv[])
{
    //定义一个共用体变量,名字叫s
    union data s;
    printf("sizeof(s) is %d\n",sizeof(s)); // is 24,共用体大小,为最大的成员变量的大小,和结构体一样,有内部对齐
    return 0;
}
```

```
linux@ubuntu:~$ ./a.out
sizeof(s) is 24
```


三. 位运算

1. 按位与&

按位与，同真为真，一假即假 1真 0假

```
int a = 11;
int b = 6;
```

```
a   1011
&
b   0110
    0010 == 2
```

2. 按位或|

按位或，同假为假，一真即真 1真 0假

```
int a = 11;
int b = 6;
```

```
a   1011
|
b   0110
    1111 == 15
```

```
#include <stdio.h>

int main(int argc, const char *argv[])
{
    int a = 11;
    int b = 17;
    // 01011
    // 10001
    //& 00001 //1
    //| 11011 //27
    //^ 11010 //26
    printf("a & b is %d\n", a & b);
    printf("a | b is %d\n", a | b);
    printf("a ^ b is %d\n", a ^ b);
    return 0;
}
```

```
linux@ubuntu:~$ gcc test.c
linux@ubuntu:~$ ./a.out
a & b is 1
a | b is 27
a ^ b is 26
```

3. 按位取反~

按位取反: 1变为0,0变为1

```
#include <stdio.h>

int main(int argc, const char *argv[])
{
    int a = 10;
    printf("~a is %d\n", ~a); //打印输出的是~a原码的值 ~a is -11
    //~运算符, 将所有的位进行取反
    //a      0000 0000 0000 0000 0000 0000 0000 1010 补码
    //~a     1111 1111 1111 1111 1111 1111 1111 0101 补码
    //通过~a的补码取求~a的原码
    //~a     1000 0000 0000 0000 0000 0000 0000 1010 符号位不变, 其他位取反
    //~a     1000 0000 0000 0000 0000 0000 0000 1011 +1 -11
    //有了~a的补码, 求出~a的原码
    //负数, 原码求补码 : 符号位不变, 其他位取反, 再+1
    //负数, 补码求原码 : 符号位不变, 其他位取反, 再+1
    return 0;
}
```

```
#include <stdio.h>

int main(int argc, const char *argv[])
{
    int a = -11;
    printf("~a is %d\n", ~a); //打印输出的是~a原码的值 ~a is 10
    //~运算符, 将所有的位进行取反
    //a      1000 0000 0000 0000 0000 0000 0000 1011 原码
    //~a     1111 1111 1111 1111 1111 1111 1111 0100 反码 符号位不变, 其他位取反
    //~a     1111 1111 1111 1111 1111 1111 1111 0101 补码 反码+1

    //~a     0000 0000 0000 0000 0000 0000 0000 1010 补码
    // 发现~a最高位是0, 所以 原码 == 反码 == 补码
    //~a     0000 0000 0000 0000 0000 0000 0000 1010 原码 10
    return 0;
}
```

4. 异或^

异或：相同为0，不同为1

```
int a = 11;
int b = 6;
```

```
a    1011
^
b    0110
1101 == 13
```

5. <<左移

```
int a = 9;
printf("a << 3 is %d\n", a << 3); // a << 3 is 72
```

```
9    0000 0000 0000 0000 0000 0000 0000 1001
9<<3 0000 0000 0000 0000 0000 0000 0100 1000 //72
```

a << 3 相当于整体向左移动3个二进制位，空出的位置有0填充

a << 3 相当于 $a * 2^3$ == $9 * 8 = 72$

a << 8 相当于 $a * 2^8$

```
8 ---> 1<<3
16 ---> 1<<4
4 ---> 1<<2
2 ---> 1<<1
```

6. >>右移

```
int a = 9;
printf("a >> 3 is %d\n", a >> 3); // a >> 3 is 1
```

```
9    0000 0000 0000 0000 0000 0000 0000 1001
9>>3 0000 0000 0000 0000 0000 0000 0000 0001 将多出去的3位删除
```

a >> 3 相当于整体向右移动3个二进制位，空出的位置有0填充，右侧多出的3位，删除

a >> 3 相当于 $a / 2^3$ == $9 / 8 = 1$

四. 原码 反码 补码

```
signed int a = 10; //有符号的整型变量a
    a = -2;
unsigned int a = 10; //无符号的整型变量a  unsigned 无符号
```

对于有符号的变量，最高位代表的是符号位

最高位是1，是负数

最高位是0，是正数

正数： 原码 == 反码 == 补码

负数： 原码 == 反码(符号位不变，其他位取反)， == 补码(反码+1)

```
1
原码
```

```

0000 0000 0000 0000 0000 0000 0000 0001
反码
0000 0000 0000 0000 0000 0000 0000 0001
补码
0000 0000 0000 0000 0000 0000 0000 0001

-1
原码
1000 0000 0000 0000 0000 0000 0000 0001
反码(符号位不变, 其他位取反)
1111 1111 1111 1111 1111 1111 1111 1110
补码(反码+1)
1111 1111 1111 1111 1111 1111 1111 1111

1+ -1 == 0
1补码
0000 0000 0000 0000 0000 0000 0000 0001
-1补码
1111 1111 1111 1111 1111 1111 1111 1111
0000 0000 0000 0000 0000 0000 0000 0000

```

五. memcpy与strcpy

1. 内存拷贝函数

```

#include <string.h>
void *memcpy(void *dest, const void *src, size_t n);
#功能: 内存拷贝, 将一块内存中的数据, 拷贝到另一块内存中
#参数: void *dest 目的 目的内存空间的首地址
        const void *src 源 被拷贝的内存空间的首地址
        //将src指向的内存空间 拷贝 到 dest指向的内存空间
        size_t n //以字节为单位, 将src指向的内存空间的前n个字节, 拷贝到desc指向的内存空间
#返回值: 返回值的 dest里面的首地址

```

```

#include <string.h>
#include <stdio.h>

int main(int argc, const char *argv[])
{
    char a[100] = "11111111111111111111111111111111";
    char b[] = "hello";
    //内存拷贝函数
    memcpy(a, b, 5); //将b数组的前5个字节, 拷贝到a数组中, 没有拷贝'\0'
    printf("a is %s\n", a);

    //字符串拷贝函数
    char c[100] = "11111111111111111111111111111111";
    char d[] = "hello";
    strcpy(c, d); //字符串拷贝函数只能 处理字符串
    printf("c is %s\n", c);
}

```



```

    }
}

int main(int argc, const char **argv)
{
    char* name[5] = {"zhangsan", "lisi", "wangwu", "zhaoliu", "maqi"};
    showName(name, 5);
    //name[0] 类型 char*
    //&name[0] 类型 char**
    //数组名字就是数组的首地址 name 等价于 &name[0]，实参name的类型char**
    return 0;
}

```

@作业2

char* name[5] = {"zhangsan", "lisi", "wangwu", "zhaoliu", "maqi"};
 写一个函数sortName()用冒泡排序实现对名字进行排序 按照电话本顺序进行排序，交换位置的时候，调用swap()函数

```

#include <string.h>
#include <stdio.h>

//遍历数组
void showName(char** p, int n)
{
    int i;
    for(i = 0; i < n; i++)
    {
        puts(p[i]); //p[i] == name[i] == char*
        //printf("%s\n", p[i]); //p[i] 保存的是每个字符串的首地址
    }
}

void swap(char** p, char** q)
{
    char* temp = *p;
    *p = *q;
    *q = temp;
}

//对数组进行冒泡排序
void sortName(char** p, int n)
{
    int i, j;
    for(i = 0; i < n-1; i++)
    {
        for(j = 0; j < n-1-i; j++)
        {
            if(strcmp(p[j], p[j+1]) == 1) //代表左边的字符串>右边
            {
                swap(&p[j], &p[j+1]); //p[j]是char*, 所以&p[j]是char** 地址传递
            }
        }
    }
}

#endif
//交换p[j] 和 p[j+1]的指向

```

```

        //因为p[j] 和 p[j+1]的类型是char*,交换两个变量的指向所以
        //第三个变量类型是char*
        char* temp = p[j];
        p[j] = p[j+1];
        p[j+1] = temp;
    }
}

int main(int argc, const char **argv)
{
    char* name[5] = {"zhangsan", "lisi", "wangwu", "zhaoliu", "maqi"};
    showName(name, 5);
    sortName(name, 5);
    printf("-----\n");
    showName(name, 5);
    //name[0] 类型 char*
    //&name[0] 类型 char**
    //数组名字就是数组的首地址 name 等价于 &name[0], 实参name的类型char**
    return 0;
}

```

```

linux@ubuntu:~$ ./a.out
zhangsan
lisi
wangwu
zhaoliu
maqi
-----
lisi
maqi
wangwu
zhangsan
zhaoliu

```

@复习

1. 指针函数:本质是函数, 返回值为指针类型的函数

```
int* get_memory(); //指针函数
```

2. 函数指针:本质是指针, 指向函数的指针

函数类型 `int getMax(int a, int b)`

`getMax`函数的类型 `int (int,int)`

指向这个类型的指针类型 `int (*)(int,int)`

定义一个函数指针变量

```

int (*p)(int,int);
p = getMax; //指向这个函数
p(3,5); //通过函数指针调用函数

```

3. 回调函数：将函数指针变量当做函数参数的函数称为 回调函数

```

int process(int (*p)(int,int), int a, int b)
{

    return p(a,b);
}
int main()
{
    int result = process(add, 3, 5);

    return 0;
}

```

4. 函数指针数组：本质一次性定义多个函数指针变量

```

int (*a[4])(int,int) = {add, sub, mul, dev};

```

5. const关键字:常量

```

const int a = 10; //a 变为常量
//const在*p前
int a = 10, b = 20;
const int* p = &a; //等价于 int const* p = &a
*p = 200; //此行报错,const在*p前, *p不可改
p = &b;

//const在p前
int a = 10, b = 20;
int * const p = &a;
*p = 200;
p = &b; //此行报错,const在p前, p不可改

//const在*p和p前
int a = 10, b = 20;
const int * const p = &a; //等价于 int const* const p = &a;
*p = 200; //此行报错,const在*p前, *p不可改
p = &b; //此行报错,const在p前, p不可改

```

6. 条件编译

```

#ifdef DEBUG
#else
#endif

//在编译的时候, 添加一宏
gcc -D DEBUG test.c

#if 1 //1保留, 0注释
#endif

```

7. 枚举：枚举就是整型

```

//默认枚举成员变量从0开始
//如果显示赋值, 成员变量就是赋值的值
//如果没有显示赋值, 他的值是前一个成员变量的值+1
//枚举成员变量的值可以出现重复
enum week
{
    Mon, //0
    Tues,
    wed
};

```


8. 共用体

```
union data
{
    int a;
    char b;
    double;
};
sizeof(union data) == 8
```

所有的成员变量，共用一块内存空间，起始地址都相同

在使用共用体的时候，相互成员变量之间会影响

任意时刻，只有一个成员变量保存的数据是有效的

共用体的大小：最大的成员变量的大小（注意内存对齐的问题）

```
union data
{
    int a;
    char b;
    double;
    char name[21];
};
sizeof(union data) == 24 //内存对齐问题
```

9. & | ^ (相同为假0，不同为真1)

```
<< //放大 a << 3    a*2的3次幂    a<<n    a*2的n次幂
>> //缩小 a >> 3    a/2的3次幂    a>>n    a/2的n次幂
~ //按位取反 1变为0,0变为1
```

10. 原码 反码 补码

```
signed int a; //等价于 int a; 有符号的整型变量a
unsigned int a; //无符号的整型变量a
```

最高位是1，代表负数

最高位是0，代表正数

正数：原码 == 反码 == 补码

负数：原码 ---> 反码（原码 符号位不变，其他位取反，得到反码） ---> 补码（反码+1）

负数：原码求补码 符号位不变，其他位取反再+1

负数：补码求原码 符号位不变，其他位取反再+1

11. 如何避免头文件的重复包含

```
#ifndef A_H
#define A_H
    //代码
#endif
```