



# 关键字typedef

---

- 将一个复杂类型转换为简单表示

```
typedef struct Node{
    int data;
    struct Node* next;
}Node;

int main(){
    Node n1;
    return 0;
}
```

- 类型重命名，只针对类型.

# 关键字static

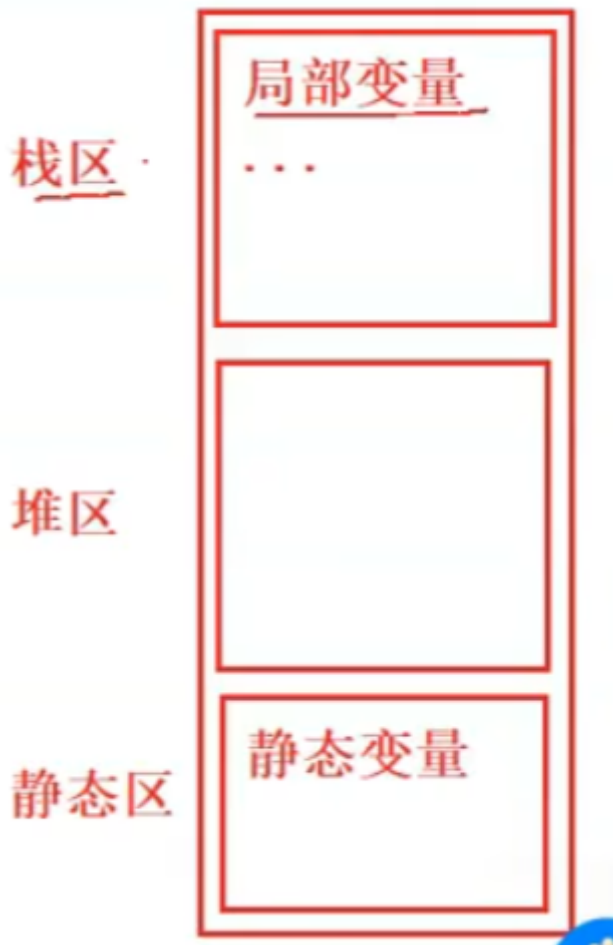
---

- 修饰全局变量
- 修饰局部变量
- 修饰函数（静态函数）

```
//修饰局部变量
#include <stdio.h>

void test(){
    static int a = 1;  //汇编不会编译之后的代码
    a++;
    printf("%d", a);
}

int main(){
    int i = 0;
    while(i<10){
        test();
        i++;
    }
    return 0;
}
```



```
//修饰全局变量  
static int a = 2023;
```

- static修饰全局变量时，使得全局变量的外部链接属性变成了内部连接属性，其他源文件不可以再使用该变量；
- 函数是具有外部链接属性的

## register—寄存器变量

- 将变量放入寄存器中，增加运行速度

## 结构体（结构体的使用）

```
struct Stu{
    char name[20];
    int age;
    char sex[10];
    char tele[10];
}

//结构体的三种调用
//打印结构体信息
struct Stu s = {"张三", "20", "男", "224212098"};

//结构体成员访问
printf("name is %s, age = %d, sex is %s, id is %s",s.name, s.age, s.sex, s.id);
//->操作符
struct Stu *ps = &s;
printf("name is %s, age = %d, sex is %s, id is %s",ps->name, ps->age, ps->sex, ps->id); //地址、指针
```

## 函数的声明和定义

---

```
//函数声明和定义
#include <stdio.h>

//函数的声明
int Add(int x, int y);

int main()
{
    int a = 0;
    int b = 0;
    scanf("%d %d", &a, &b);
    //加法
    int sum = Add(a, b);
    printf("%d\n", sum);

    return 0;
}

//函数的定义
int Add(int x, int y)
{
    return x + y;
}
```

一般来说，函数定义放在.c文件，函数声明放在.h文件。

## 结构体的声明

- 浮点数在内存中不能精确保存
- 结构体可以不完全初始化
- 传结构体以及传结构体指针

```
#include <stdio.h>

int main(){
    int a, b, c;
    a = 5;
    c = ++a; //a=6, c=6
}
```

```
void print2(struct Peo* sp)
{
    printf("%s %s %s %d\n", sp->name, sp->tele, sp->sex, sp->high); //结构体指针->成员变量
}

void print1(struct Peo p)
{
    printf("%s %s %s %d\n", p.name, p.tele, p.sex, p.high); //结构体变量.成员变量
}

int main()
{
    struct Peo p1 = {"张三", "15596668862", "男", 181}; //结构体变量的创建
    struct St s = { {"lisi", "15596668888", "女", 166}, 100, 3.14f};

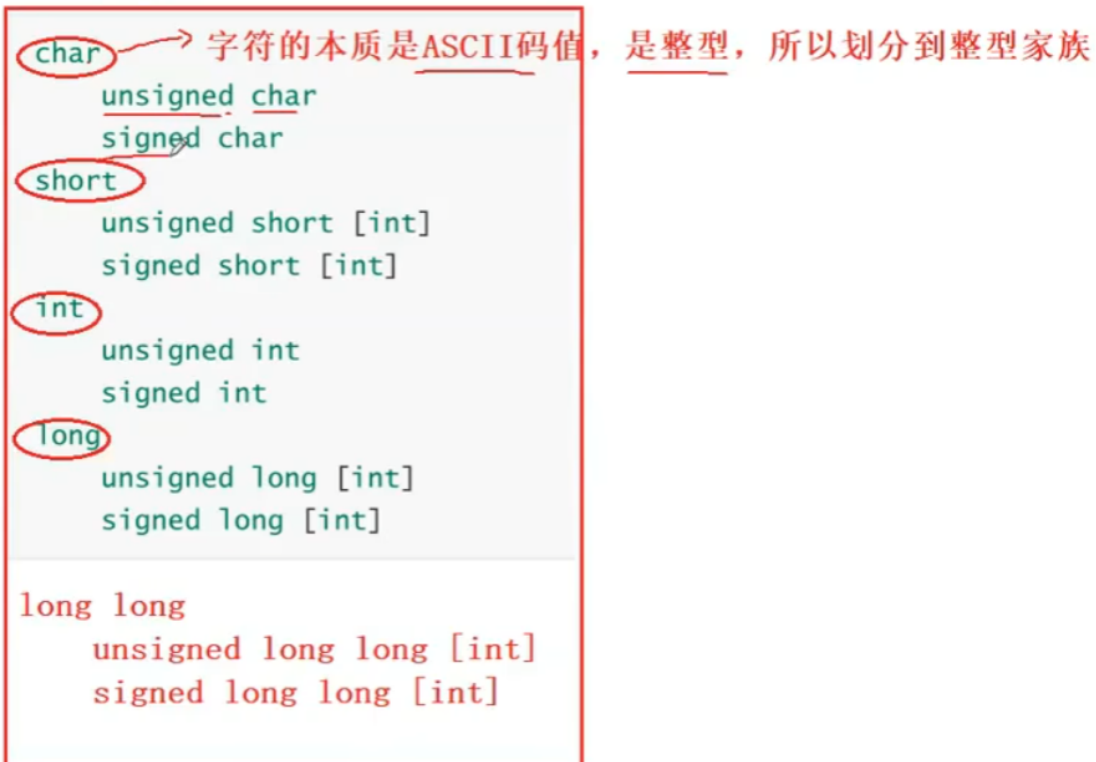
    printf("%s %s %s %d\n", p1.name, p1.tele, p1.sex, p1.high);
    printf("%s %s %s %d %d %f\n", s.p.name, s.p.tele, s.p.sex, s.p.high, s.num, s.f);

    print1(p1);
    print2(&p1);

    return 0;
}
```

## 数据储存

## 整型家族：



- \
- 数据存储形式——0x1122 3344

大端储存：11223344（顺序存放，从低地址到高地址）

小端储存：44332211（倒序存放，从高地址到低地址）

```
//判断大小端储存
int main(){
    //int占用4个字节，只要将首个字节拿出来，就可以判断是大端还是小端
    int a = 1;
    if(*(char*)&a == 1){
        printf("小端");
    }
    else{
        print("大端");
    }
}
return 0;
}
```

- 整型提升

## 指针进阶

1. 字符指针
2. 数组指针
3. 指针数组
4. 数组传参和指针传参
5. 函数指针
6. 函数指针数组
7. 指向函数指针数组的指针
8. 回调函数
9. 指针和数组面试题的解析

```
int main()
{
    const char* p1 = "abcdef";
    const char* p2 = "abcdef";

    char arr1[] = "abcdef";
    char arr2[] = "abcdef";

    if (p1 == p2)
        printf("p1==p2\n");
    else
        printf("p1!=p2\n");

    if (arr1 == arr2)
        printf("arr1 == arr2\n");
    else
        printf("arr1 != arr2\n");

    return 0;
}
```

需要源码课件等，请三联后加鹏哥微

- 指针固定大小为4/8字节（32位/64位）
- 指针是由类型的

## 字符指针



```
#include <stdio.h>
int main(){
    char* p = "abcdefg";    //把字符串首地址fi
}
```

## 指针数组


- 用来存放指针的数组

```
int * arr1[6];
char *arr2[2];

int main(){
    int arr1[] = {1,2,3,4,5};
    int arr2[] = {1,2,3,4,5};
    int arr3[] = {1,2,3,4,5};

    int* parr[] = {arr1,arr2,arr3};
    for(int i=0; i<3; i++){
        for(int j=0; j<5; j++){
            printf(parr[i] + j);
            printf(parr[i][j]);    //等价
        }
    }
}
```

## 数组指针(指向数组的指针)



```
int *p1[10];    p1是指针数组
int (*p2)[10];   p2是数组指针, p2可以指向一个数
//p1, p2分别是什么? 组, 该数组有10个元素, 每个元素是
                        int类型。
```

```

int main()
{
    int arr[10] = { 0 };
    printf("%p\n", arr);
    printf("%p\n", &arr[0]);

    int sz = sizeof(arr);
    printf("%d\n", sz);

    return 0;
}

```

- 数组名通常表示数组首元素的地址
- 有两个例外：
  - (1) sizeof(数组名), 本数组名表示整个数组, 计算整个数组的大小
  - (2) &数组名, 表示整个数组

```

int main(){
    int arr[] = {1,2,3,4,5,6,7,8,9,0};

    int *p = arr;
    for(int i=0; i<10; i++){
        printf("%d",*(p+i));
    }
}输出为1, 2, 3, 4, 5, 6, 7, 8, 9, 0

```

//常用二维数组

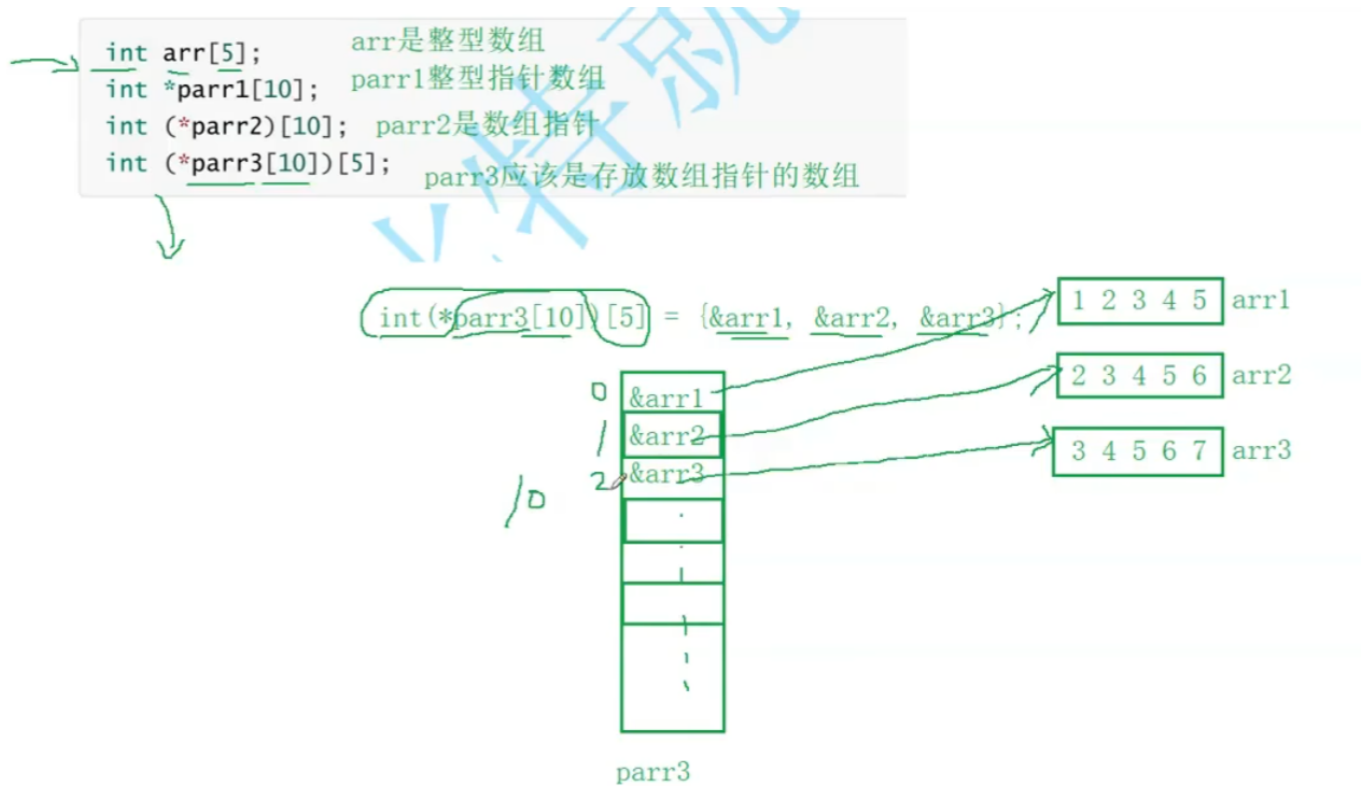
//二维数组的首元素是他的第一行

```

void print(int (*p)[5], r, c){
    int i = 0;
    for(int i=0; i<r; i++){
        for(int j=0; j<c; j++){
            print(*(p+i)+j));
        }
    }
}

int main(){
    int arr[3][5] = {1,2,3,4,5,,2,3,4,5,6,3,4,5,6,7};
    print(arr,3,5);
    return 0;
}

```



## 数组参数和指针参数

- 一维数组传参

```
//test的写法
void test(int arr[])
void test(int arr[10])
void test(int * arr) //数组名相当于数组元素首地址, 所以san'zhong

int main(){
    int arr[10] = {0};
    test(arr);
}
```

```
void test(int *arr[20]);
void test(int **arr);

int main(){
    int *arr[20] = {0};          //int* int * int* int* .....
    test(arr);
}
```

- 二维数组传参

```
void test(int arr[3][5]);
void test(int arr[][5]);
//二维数组传参，只能省略第一个维度的数字
```

```
//传指针
void test(int (*arr)[5]);
void test()
```

```
int main(){
    int arr[3][5] = {0};
    test(arr);
}
```

## 数组指针的常见用法

---

```
void print1(int arr[3][5], int r, int c)
{
    int i = 0;
    for(i=0; i<r; i++){
        int j = 0;
        for(j=0; j<c; j++){
            printf("%d", arr[i][j]);
        }
        printf("\n");
    }
}

int main(){
    int arr[3][5] = {1,2,3,4,5,2,3,4,5,6,3,4,5,6,7};
    print1(arr,3,5);
    print2(arr,3,5);
    return 0;
}
```

- 数组名是首元素的地址
- 二维数组的首元素是数组第一行

```
void print2(int (*p)[5], int r, int c){
    int i,j;a
    for(i=0; i<r; i++){
        for(j=0; j<c; j++){
            printf("%d", *((p+i)+j));
        }
        printf("\n");
    }
}
```

# 函数指针

---

```
int main(){
    //指针数组
    int* arr[4];
    char* ch[5];
    //数组指针
    int arr[5];
    int (*pa)[5] = &arr;

    char* arr[6];
    char* (*p3)[6] = &arr;

    return 0;
}
```

## 函数指针

---

```
int test(const char* str){
    return 0;
}
int main(){
    //函数指针，是指向函数的指针
    printf("%p\n",test);
    printf("%p\n",&test);    //二者等价
    int (*pf)(const char*) = test;
    (*pf)("abc"); //实际上传的是a的地址
}
```

- 回调函数：通过函数指针调用的函数，把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用来调用其所指向的函数时，称为回调函数。

## 函数指针的用途

//写一个计算器，加减乘除

```
void menu(){
    printf("*****\n");
    printf("*****    1.Add    2.Sub    *****\n");
    printf("*****    3.Mul    4.Div    *****\n");
    printf("*****    0.Exit      *****\n");
    printf("*****\n");
}

int Add(int x, int y){
    return x+y;
}
int Sub(){
}
int Mul(){
}
int Div(){
}
int Exit(){
}

void calc(int (*pf)(int, int)){
    int x = 0;
    int y = 0;
    int ret = 0;

    printf("请输入两个操作数:>\n");
    scanf("%d %d", &x, &y);
    ret = pf(x,y);
    printf("%d\n", ret);
}

int main(){

    int input = 0;

    do{
        menu();
        printf("请选择>");
        scanf("%d", &input);
        switch(input){
            case 1:
                calc(Add);
                break;
            case 2:
                calc(Sub);
                break;
            case 3:
                calc(Mul);
                break;
            case 4:
                calc(Div);
                break;
            case 5:
                calc(Exit);
                break;
        }
    }while(input);
    return 0;
}
```

# 函数指针数组

函数指针也是一种指针，把函数和指针放在数组中，就是函数指针数组

```
int main(){
    int (*pf)(int, int) = Add;    //pf是函数指针
    int (*arr[4])(int, int)      = {Add, Sub, Mul, Div};    //arr就是函数指针数组

    return 0;
}
```

# 回调函数

- 一个通过函数指针调用的函数，如果把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用来调用其所指向的函数时，我们就说这是回调函数，回调函数不是由该函数的实现方直接调用，而是在特定的时间或者条件发生时由另外一方调用的，用于对该事件或者条件进行响应。

```
//经典案例
void qsort(void* base,        /待排序数据起始位置
           size_t num,
           size_t width, //数据大小 (单位是字节)
           int(* cmp)(const void* e1, const void* e2) //函数指针-是一个比较函数
)
int main(){
    qsort(arr, size, sizeof(arr[0]), cmp);
}
```

```
int main()
{
    int a = 10;
    int* pa = &a;
    *pa = 20; //
    printf("%d\n", a); //20
    return 0;
}
```

- 指针类型：（1）决定+1/-1操作跳过多少个字节  
（2）解引用操作时的权限
- 指针数组：本质上就是数组，不过数组中存放的是指针（地址）

```
int* pa;
int* pb;
int* pc;

int* arr[3] = {pa, pb, pc};
```

## 数组名

数组名在大部分条件下表示数组首元素的地址，但是有两个例外：

- （1）sizeof（数组名）
- （2）&数组名----->取出数组的地址

数组指针：指向数组的指针

函数名表示该函数的地址，Add和&Add等效

```
int arr[10] = {1,2,3};
int (*parr)[10] = &arr;    //数组指针，实际上是指针
```

在int (pfArr[4])(int, int)中，pfArr的每个元素类型为：int(\*) (int, int)表示为函数指针类型

```
int mian(){
    int a[] = {1, 2, 3, 4};
    printf("%d\n", sizeof(a)); //16
    printf("%d\n", sizeof(a + 0)); 4 //首元素地址大小，占用4个字节大小
    printf("%d\n", sizeof(*a)); 4 //a是数组首元素地址，*a就是地址的解引用，找到a的首元素
    printf("%d\n", sizeof(&a)); 4 //表示整个数组的地址大小，还是4
    printf("%d\n", sizeof(*&a)); 16 //对整个数组地址进行解引用
    printf("%d\n", sizeof(&a + 1)); 4/8 //地址加1
    printf("%d\n", sizeof(&a【0】)); 4/8 //地址大小
    printf("%d\n", sizeof(&a[0] + 1)); 4/8 //地址大小
}
```



```
int main()
{
    char arr[] = { 'a','b','c','d','e','f' };
    printf("%d\n", sizeof(arr)); //6
    //sizeof(数组名)
    printf("%d\n", sizeof(arr + 0)); //4/8
    //arr + 0 是数组首元素的地址
    printf("%d\n", sizeof(*arr)); //1
    //*arr就是数组的首元素，大小是1字节
    //*arr --> arr[0]
    /*(arr+0) --> arr[0]
    printf("%d\n", sizeof(arr[1])); //1
    printf("%d\n", sizeof(&arr)); //4/8
    //&arr是数组的地址，是地址就是4/8个字节
    printf("%d\n", sizeof(&arr + 1)); //4/8
    //&arr + 1是数组后的地址
    //
    printf("%d\n", sizeof(&arr[0] + 1)); //4/8
    //&arr[0] + 1是第二个元素的地址
    //发生了整型提升
    return 0;
}
```

```

int main()
{
    //char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f' };

    char arr[] = "abcdef";

    //没有结束符" \0 "
    //所以用strlen时输出随机值, strlen要求输入一个地址参数

    //strlen是求字符串长度的, 关注的是字符串中的\0, 计算的是\0之前出现的字符的个数
    //strlen是库函数, 只针对字符串
    //sizeof只关注占用内存空间的大小, 不在乎内存中放的是什么
    //sizeof是操作符
    //
    //[a b c d e f \0]
    printf("%d\n", strlen(arr));//6
    printf("%d\n", strlen(arr + 0));//6
    //printf("%d\n", strlen(*arr));//err, 指向一个野指针
    //printf("%d\n", strlen(arr[1]));//err, 指向一个野指针
    printf("%d\n", strlen(&arr));//6
    printf("%d\n", strlen(&arr + 1));//随机值
    printf("%d\n", strlen(&arr[0] + 1));//5

    //[a b c d e f \0]
    //printf("%d\n", sizeof(arr));//7
    //printf("%d\n", sizeof(arr + 0));//4/8
    //printf("%d\n", sizeof(*arr));//1
    //printf("%d\n", sizeof(arr[1]));//1
    //printf("%d\n", sizeof(&arr));//4/8
    //printf("%d\n", sizeof(&arr + 1));//4/8
    //printf("%d\n", sizeof(&arr[0] + 1));//4/8

    return 0;
}

```

```

int main()
//{
//    char* p = "abcdef"; //指向a的地址
//    printf("%d\n", sizeof(p));    4/8
//    printf("%d\n", sizeof(p + 1)); 4/8
//    printf("%d\n", sizeof(*p));    1
//    printf("%d\n", sizeof(p[0]));    1
//    printf("%d\n", sizeof(&p));    4/8
//    printf("%d\n", sizeof(&p + 1)); 4/8
//    printf("%d\n", sizeof(&p[0] + 1)); 4/8
//
//    printf("%d\n", strlen(p));    6
//    printf("%d\n", strlen(p + 1)); 5
//    printf("%d\n", strlen(*p));    error
//    printf("%d\n", strlen(p[0]));    error
//    printf("%d\n", strlen(&p));    随机值
//    printf("%d\n", strlen(&p + 1)); 随机值
//    printf("%d\n", strlen(&p[0] + 1)); 5
//
//    return 0;
//}

```



李亚楠-4

为什么 strlen(&p) 是随机值

```
char* p = "abcdef";
```



二维数组

```

int main()
{
    int a[3][4] = { 0 };
    printf("%d\n", sizeof(a));          48
    printf("%d\n", sizeof(a[0][0])); 4
    printf("%d\n", sizeof(a[0])); 16
    //a[0]是第一行这个一维数组的数组名，单独放在sizeof内部，a[0]表示第一个整个这个一维数组
    //sizeof(a[0])计算的就是第一行的大小
    printf("%d\n", sizeof(a[0] + 1)); 4
    //a[0]并没有单独放在sizeof内部，也没取地址，a[0]就表示首元素的地址
    //就是第一行这个一维数组的第一个元素的地址，a[0] + 1就是第一行第二个元素的地址
    printf("%d\n", sizeof(*(a[0] + 1)));
    //a[0] + 1就是第一行第二个元素的地址 4
    //*(a[0] + 1)就是第一行第二个元素
    printf("%d\n", sizeof(a + 1)); //4/8
    //a虽然是二维数组的地址，但是并没有单独放在sizeof内部，也没取地址
    //a表示首元素的地址，二维数组的首元素是它的第一行，a就是第一行的地址
    //a+1就是跳过第一行，表示第二行的地址
    printf("%d\n", sizeof(*(a + 1))); //16
    //*(a + 1)是对第二行地址的解引用，拿到的是第二行
    //*(a+1)-->a[1]
    //sizeof(*(a+1))-->sizeof(a[1])
    //
    printf("%d\n", sizeof(&a[0] + 1)); //4/8
    //&a[0] - 对第一行的数组名取地址，拿出的是第一行的地址
    //&a[0]+1 - 得到的是第二行的地址
    //
    printf("%d\n", sizeof(&a[0] + 1)); //16
    printf("%d\n", sizeof(*a)); //16
    //a表示首元素的地址，就是第一行的地址
    //*a就是对第一行地址的解引用，拿到的就是第一行
    //
    printf("%d\n", sizeof(a[3])); //16
    printf("%d\n", sizeof(a[0])); //16

    //int a = 10;
    //sizeof(int);
    //sizeof(a);
    return 0;
}

```

## 顶级题目

---

```

{
    int a[4] = { 1, 2, 3, 4 };
    int* ptr1 = (int*)&a + 1;
    int* ptr2 = (int*)((int)a + 1);
    printf("%x,%x", ptr1[-1], *ptr2);
    return 0;
}

```

$ptr1[-1]$   
 $*(ptr1+(-1))$   
 $*(ptr1-1)$

$a=0x0012ff40$   
 $a+1 \rightarrow 0x0012ff44$   
 $(int)a+1 \rightarrow 0x0012ff41$

$1 \rightarrow 0x\ 00\ 00\ 00\ 01$   
 低 高  
 ptr2  
 小端存储模式  
 $02\ 00\ 00\ 00$   
 $00\ 00\ 00\ 04$

```

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

```

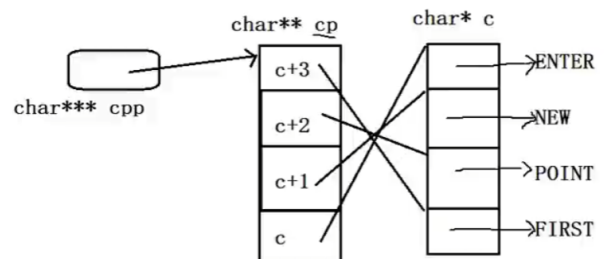
- 逗号表达式, a实际上是[1,3; 5,0; 0,0]

```

int main()
{
    char* c[] = { "ENTER", "NEW", "POINT", "FIRST" };
    char** cp[] = { c + 3, c + 2, c + 1, c };
    char*** cpp = cp;

    printf("%s\n", **++cpp);
    printf("%s\n", *-- * ++cpp + 3);
    printf("%s\n", *cpp[-2] + 3);
    printf("%s\n", cpp[-1][-1] + 1);
    return 0;
}

```



- strlen返回值是一个无符号整型
- 模拟实现(无符号整型通常占用4个字节)

```

size_t my_strlen(const char* str)
{
    size_t count = 0;
    assert(str);
    while(*str != '\0')
    {
        count++;
        str++;
    }
    return count;
}

int main(){
    char arr[] = "abcdef";
    size_t n = my_strlen(arr);
    printf("%u\n", n)
    return 0;
}

```

## strcpy自主实现

```

void my_strcpy(char* dest, char* src)
{
    assert(dest);
    assert(src);
    char* ret = dest;
    while(*src);
    {
        *dest++ = *src++;
    }
    *dest = *src;
    return ret;
}

int main(){
    char arr1[] = "abcdef";
    char arr2[20] = {0};
    my_strcpy(arr2, arr1);
    printf("%s\n", arr2);
    return 0;
}

```

## 字符串追加

```

char* my_strcat(char* dest, char* src)
{
    char* ret = dest;
    assert(dest && src);
    while(*dest != '\0')
    {
        dest++;
    }
    while(*dest++ = *src++)
    {;}
    return ret;
}
int main(){
    char arr1[20] = "hello";
    char arr2[20] = "world";
    my_strcat(arr1, arr2);    //hello world
    return 0;
}

```

## memcpy(内存拷贝)

```
void * memcpy ( void * destination, const void * source, size_t num );
```

- 函数memcpy从source的位置开始向后复制num个字节的数据到destination的内存位置。
- 这个函数在遇到 '\0' 的时候并不会停下来。
- 如果source和destination有任何的重叠，复制的结果都是未定义的。

## 什么是泛型指针：没有具体类型的指针(void\*)，可以是任意类型

```

int main(){

    int arr1[] = {1,2,3,4,5,6,7};
    int arr2[20] = {0};
    memcpy(arr2, arr1, 28);    4*7个字节

}

```

## 自定义memcpy

```

void* my_memcpy(void* dest, const void* src, size_t num)
{
    assert(dest && src);
    void* ret = dest;
    while(num--)
    {
        *(char*)dest = *(char*)src;
        dest = (char*)dest + 1;
        src = (char*)src + 1;
    }
    return ret;
}

```

## 基本知识:

- 一个数的原码是该数的二进制表示
- 反码是原码的二进制符号位不变，其他按位取反
- 补码是反码的二进制加1
- 原码、反码、补码最高位0表示正数，最高位1表示负数
- 整型提升

```

int main()
{
    //char  -128~127
    //unsigned char 0~255
    unsigned char a = 200;
    //00000000000000000000000011001000
    //11001000 -a 截断
    unsigned char b = 100;
    //0000000000000000000000001100100
    //01100100 - b 截断
    unsigned char c = 0;

    c = a + b;
    //整型提升
    //00000000000000000000000011001000
    //0000000000000000000000001100100
    //000000000000000000000000100101100
    //00101100-c 截断
    //000000000000000000000000101100
    //
    printf("%d %d", a + b, c);
    //300 44
    return 0;
}

```

- 大端储存是把数据的低字节内容存放到高地址
- 小端储存是把数据的低字节内容存放到低地址

编程题探究:



```

int main() {
    int a = 0;
    int b = 0;
    int c = 0;
    int d = 0;
    int e = 0;
    for (a = 1; a <= 5; a++)
    {
        for (b = 0; b <= 5; b++)
        {
            for (c = 0; c <= 5; c++)
            {
                for (d = 0; d <= 5; d++)
                {
                    for (e = 0; e <= 5; e++)
                    {
                        if (((b == 2) + (a == 3) == 1)
                            && ((b == 2) + (a == 3) == 1)
                            && ((b == 2) + (a == 3) == 1)
                            && ((b == 2) + (a == 3) == 1)
                            && ((b == 2) + (a == 3) == 1))
                        {
                            if (a * b * c * d * e == 120)
                            {
                                printf("a=%d b=%d c=%d e=%d f=%d", a, b, c, d, e);
                            }
                        }
                    }
                }
            }
        }
    }
    return 0;
}

```

```

int main()
{
    int killer = 0;
    for (killer = 'a'; killer <= 'd'; killer++)
    {
        if ((killer != 'a') + (killer == 'c') + (killer == 'd') + (killer != 'd') == 3)
        {
            printf("%c\n", killer);
        }
    }
    return 0;
}

```

```

void left_rotate(char arr[], int k)
{
    int i = 0;
    int len = strlen(arr);
    k %= len;
    for (i = 0; i < k; i++)
    {
        //每次旋转一个字符
        char tmp = arr[0];
        int j = 0;
        for (j = 0; j < len - 1; j++)
        {
            arr[j] = arr[j + 1];
        }
        arr[len - 1] = tmp;
    }
}

```

```

#include <assert.h>
void reverse(char* left, char* right)
{
    assert(left && right);
    while (left < right)
    {
        char tmp = *left;
        *left = *right;
        *right = tmp;
        left++;
        right--;
    }
}

```

## 杨氏矩阵（从左到右递增，从上到下递增）

# 结构体和联合体

## 一、结构体

各成员拥有自己的内存，各自互不干涉同时存在，遵循内存对齐原则，一个struct变量的总长度等于所有成员长度之和。

## 二、联合体

各成员共用一块内存空间，并且同时只有一个成员可以得到这块内存的使用权(对该内存的读写)，各变量共用一个内存首地址。因而，联合体比结构体更节约内存。一个union变量的总长度至少能容纳最大的成员变量，而且要满足是所有成员变量类型大小的整数倍。不允许对联合体变量名U2直接赋值或其他操作。

```

#include<stdio.h>
//结构体
struct u    //u表示结构体类型名
{
    char a;    //a表示结构体成员名
    int b;
    short c;
}U1;
//U1表示结构体变量名
//访问该结构体内部成员时可以采用U1.a=1;其中"点"表示结构体成员运算符

//联合体
union u1    //u1表示联合体类型名
{
    char a;    //a表示联合体成员名
    int b;
    short c;
}U2;
//U2表示联合体变量名
//访问该联合体内部成员时可以采用U2.a=1;其中"点"表示联合体成员运算符

//主函数
int main(){
    printf("%d\n",sizeof(U1));
    printf("%d\n",sizeof(U2));
    return 0;
}

/*程序运行结果是:
12
4*/

```

## Union成员赋值

```

//联合体
union u1
{
    char a;
    int b;
    short c;
}U2;

//主函数
int main(){
    U2.a='a';
    printf("%c%c\n",U2.b,U2.c);//输出aa
    U2.a='b';
    printf("%c%c\n",U2.b,U2.c);//输出bb
    U2.b=0x4241;
    printf("%c%c\n",U2.a,U2.c);//输出AA
    return 0;
}

```

## union大小计算准则：1、至少要容纳最大的成员变量 2、必须是所有成员变量类型大小的整数倍

代码中U3至少容纳最大e[5]=20字节，同时变量类型最大值是整数倍，即使double(字节数是8)的整数倍，因而sizeof(U3)=24。

```
#include<stdio.h>

//联合体
union u2
{
    char a;
    int b;
    short c;
    double d;
    int e[5];
}U3;

//主函数
int main(){
    printf("%d\n",sizeof(U3)); //输出24
    return 0;
}
```

U5中a四个字节，后面b和c加起来3个字节，正好补1个字节对齐；U6中b1个字节，要和后面的a对齐，需要补3个字节对齐，c也要补1个字节对齐，因而最终U6为12个字节。另外，要想改变这种默认对齐设置，可以用

`#pragma pack (2)` /指定按2字节对齐/

`#pragma pack ()` /取消指定对齐，恢复缺省对齐/

```
#include<stdio.h>

//联合体
struct u4
{
    int a;
    char b;
    short c;
}U5;

struct u5
{
    char b;
    int a;
    short c;
}U6;

//主函数
int main(){
    printf("%d\n",sizeof(U5));
    printf("%d\n",sizeof(U6));
    return 0;
}

//输出为
//8
//12
```

### 第 3 题 (编程题)

题目名称:

字符串旋转结果

题目内容:

写一个函数，判断一个字符串是否为另外一个字符串旋转之后的字符串。

例如：给定s1 = AABCD和s2 = BCDAA，返回1

给定s1=abcd和s2=ACBD，返回0.

AABCD左旋一个字符得到ABCDA

AABCD左旋两个字符得到BCDAA

AABCD右旋一个字符得到DAABC

```

int is_left_rotate(char arr1[],char arr2[])
{
    int len = strlen(arr1);
    int i = 0;
    for (i < 0; i < len; i++)
    {
        char tmp = arr1[0];
        int j = 0;
        for (j = 0; j < len - 1; j++)
        {
            arr1[j] = arr1[j + 1];
        }
        arr1[len - 1] = tmp;
        if (strcmp(arr2, arr1) == 0)
        {
            return 1;
        }
        return 0;
    }
}

int main()
{
    char arr1[] = "abcdef";
    char arr2[] = "defabc";

    //判断语句
    int ret = is_left_rotate(arr1, arr2);
    if (ret == 1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

```

## 结构体

- 值的集合，被称为成员变量
- 结构体的定义

```

struct Stu{
    //相关属性
    int a;
    char b;
    float c;
};

```

## 链表节点

```
struct Node{
    int data;
    struct Node* next;
};
```

```
struct Point{
    int x;
    int y;
}p1 = {2, 3};

struct Stu{
    char name[20];
    int age;
    struct score s;
}
struct score{
    int n;
    char ch;
}
int main(){
    struct Point p2 = {3, 4};
    struct Stu s1 = {"zahngsan", 16, {100, 'q'}};
    printf("%s %d %d %c\n", s1.name, s1.age, s1.s.n, s1.s.ch);
    return 0;
}
```

## 结构体内存对齐

首先得掌握结构体的对齐规则：

1. 第一个成员在与结构体变量偏移量为0的地址处。
2. 其他成员变量要对齐到某个数字（对齐数）的整数倍的地址处（偏移量）。 对齐数 = 编译器默认的一个对齐数 与 该成员大小的较小值。 VS中默认值为8
3. 结构体总大小为最大对齐数（每个成员变量都有一个对齐数）的整数倍。
4. 如果嵌套了结构体的情况，嵌套的结构体对齐到自己的最大对齐数的整数倍处，结构体的整体大小就是所有最大对齐数（含嵌套结构体的对齐数）的整数倍。

```
struct S1
{
    char c1;
    int i;
    char c2;
};
printf("%d\n", sizeof(struct S1)); 输出为12
```

## 结构体传参



```

struct S1
{
    int data[100];
    int num;
};

void print1(struct S1 ss)
{
    printf("%d", ss.data[1]);
}

void print2(struct S* ps)
{
    printf("%d", ps->data[1]);
}

int main() {
    struct S1 s = { {1,2,3}, 100 };
    print1(s); //传值调用
    print2(&s); //传地址调用
    return 0;
}

```

## 位段

位段的声明和结构是类似的，有两个不同：

- 1.位段的成员必须是 int、unsigned int 或signed int 。
- 2.位段的成员名后边有一个冒号和一个数字。

```

struct A
{
    int _a:2;
    int _b:5;
    int _c:10;
    int _d:30;
};

```

## 枚举

---

```
enum Day//星期
{
    Mon,
    Tues,
    Wed,
    Thur,
    Fri,
    Sat,
    Sun
};
enum Sex//性别
{
    MALE,
    FEMALE,
    SECRET
};
enum Color//颜色
{
    RED,
    GREEN,
    BLUE
};
```

以上定义的 `enum Day` , `enum Sex` , `enum Color` 都是枚举类型。 `{}` 中的内容是枚举类型的可能取值, 也叫 枚举常量。

这些可能取值都是有值的, 默认从0开始, 一次递增1, 当然在定义的时候也可以赋初值。 例如:

```
enum Color//颜色
{
    RED=1,
    GREEN=2,
    BLUE=4
};
```

## 判断大小端储存

```

int check_sys
{
    union
    {
        char c;
        int i;
    }u;
    u.i = 1;
    return u.c; //共享内存机制
}
int main()
{
    int a = 1; //0x 00 00 00 01
    //低----->高
    //01 00 00 00 小端储存
    //00 00 00 01 大端储存
    int ret = check_sys();
    if(ret == 1){
        小端
    }
    else 大端

    return 0;
}

```

## C语言动态内存开辟

```

#include <errno.h>
#include <string.h>
#include <stdlib.h>

int main()
{
    int arr[10] = { 0 };
    //动态内存开辟
    int* p = (int*)malloc(40);
    if (p == NULL)
    {
        printf("%s\n", strerror(errno));
        return 1;
    }
    //使用
    int i = 0;
    for (i = 0; i < 10; i++)
    {
        *(p + i) = i;
    }
    for (i = 0; i < 10; i++)
    {
        printf("%d\n", *(p + i));
    }
    free(p);
    p = NULL;
    return 0;
}

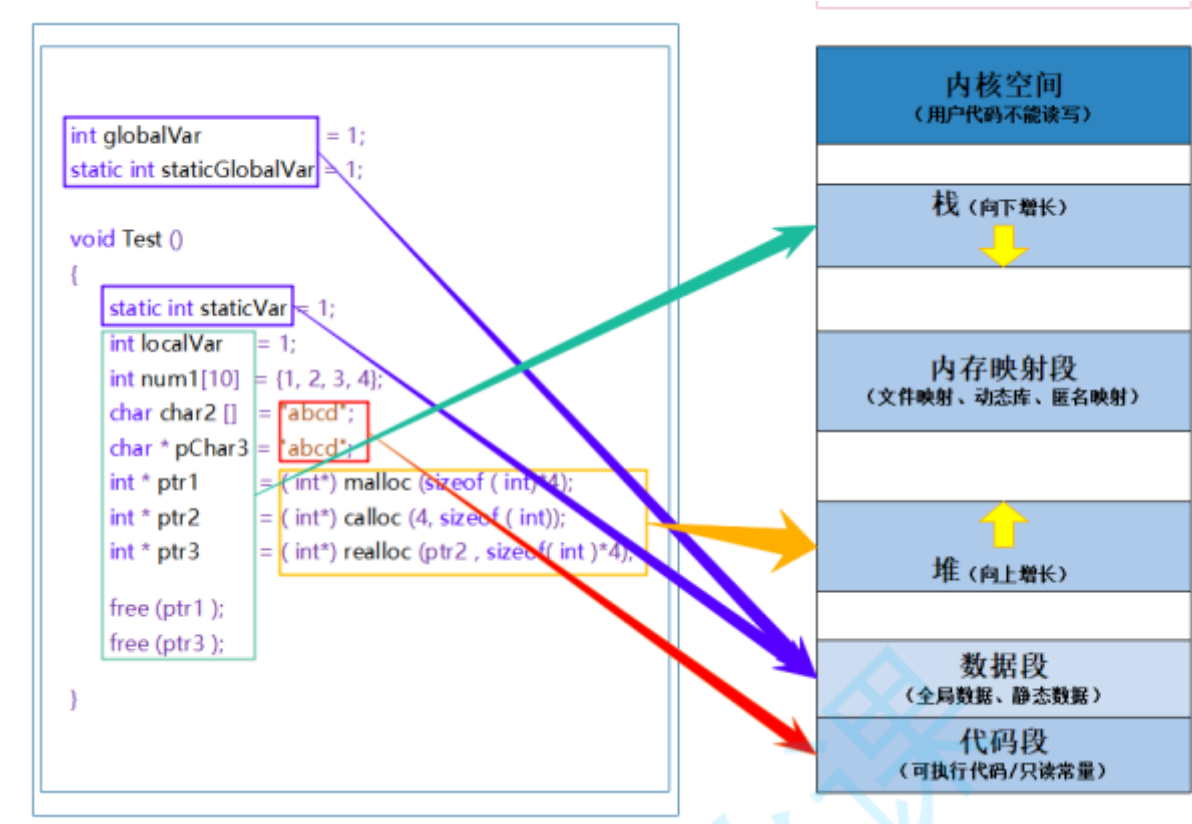
```

# 堆栈和静态区

堆：由malloc系列函数或者new操作符分配的内存，其生命周期由free或者delete决定，在没有释放内存之前一直存在，直到程序结束，其特点是使用灵活，空间比较大，但是容易出错。

栈：保存局部变量。栈上的内容只在函数的范围内存在，当函数运行结束时，这些内容会被自动销毁。其特点是效率高，但是空间大小有限。

静态区：保存自动全局变量和static变量（static全局变量和局部变量）。静态区的内容在整个程序的生命周期内都存在，由编译器在编译时候分配。



- C/C++程序内存分配的几个区域：
1. 栈区 (stack)：在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。栈区主要存放运行函数而分配的局部变量、函数参数、返回数据、返回地址等。
  2. 堆区 (heap)：一般由程序员分配释放，若程序员不释放，程序结束时可能由OS回收。分配方式类似于链表。
  3. 数据段 (静态区) (static) 存放全局变量、静态数据。程序结束后由系统释放。
  4. 代码段：存放函数体 (类成员函数和全局函数) 的二进制代码。

有了这幅图，我们就可以更好的理解在《C语言初识》中讲的static关键字修饰局部变量的例子了。实际上普通的局部变量是在栈区分配空间的，栈区的特点是在上面创建的变量出了作用域就销毁。但是被static修饰的变量存放在数据段（静态区），数据段的特点是在上面创建的变量，直到程序结束才销毁 所以生命周期变长。

# 柔性数组

结构体中的最后一个元素允许是未知大小的数组，这就叫做『柔性数组』成员。

```
typedef struct st_type
{
    int i;
    int a[]; //柔性数组成员
}type_a;
```

- 结构中的柔性数组成员前面必须至少一个其他成员。
- sizeof 返回的这种结构大小不包括柔性数组的内存。
- 包含柔性数组成员的结构用 `malloc ()` 函数进行内存的动态分配，并且分配的内存应该大于结构的大小，以适应柔性数组的预期大小。

```
struct S
{
    int n;
    int arr[]; //柔性数组成员
};

int main()
{
    int sz = sizeof(struct S);
    printf("%d\n", sz);
    return 0;
} //错误使用，输出为4
```

## 正确使用

```

#include <stdio.h>

struct S
{
    int n;
    int arr[];    //柔性数组成员
};

int main()
{
    /*int sz = sizeof(struct S);
    printf("%d\n", sz);*/
    struct S* ps=(struct S*)malloc(sizeof(struct S) + 40); //malloc返回地址空间，强转为结构体指针类型

    ps->n = 100;
    int i = 0;
    for (i = 0; i < 10; i++)
    {
        ps->arr[i] = i;
    }
    for (i = 0; i < 10; i++)
    {
        printf("%d", ps->arr[i]);
    }
    struct S* ptr=(struct S*)realloc(ps,sizeof(struct S)+80);
    if (ptr != NULL)
    {
        ps = ptr;
        ptr = NULL;
    }
    free(ps);
    ps = NULL;
    return 0;
}

```

## C语言结构体里的成员数组和指针

```

#include <stdio.h>
struct str{
    int len;
    char s[0];
};

struct foo {
    struct str *a;
};

int main(int argc, char** argv) {
    struct foo f={0};
    if (f.a->s) {
        printf( f.a->s);
    }
    return 0;
}

```

# 程序环境和预处理

---

## 宏替换

---

在程序中扩展#define定义符号和宏时，需要涉及几个步骤。

1. 在调用宏时，首先对参数进行检查，看看是否包含任何由#define定义的符号。如果是，它们首先被替换。
2. 替换文本随后被插入到程序中原来文本的位置。对于宏，参数名被他们的值替换。
3. 最后，再次对结果文件进行扫描，看看它是否包含任何由#define定义的符号。如果是，就重复上述处理过程。

注意：

1. 宏参数和#define 定义中可以出现其他#define定义的变量。但是对于宏，不能出现递归。
2. 当预处理器搜索#define定义的符号的时候，字符串常量的内容并不被搜索。

## 查找策略

<>：直接去库目录下查找

""：先去代码所在目录下查找：再去库目录下查找

## 原码、反码、补码

---

- 计算机运算器只有加法，所以减法转化为加法运算

原码：

14：00001110

21：10010101（黄色表示符号位）

反码：

正数原码反码补码都相同；负数反码为符号位不变，其余位数字取反

补码：

就是反码加1

# 原码 反码 补码

## 4、计算机内计算方式

$$14 + (-21) = -7$$

补码 + 补码

补码 → 反码 → 原码

标志位不变，其它取反

0	0	0	0	1	1	1	0	
+	1	1	1	0	1	0	1	1
<hr/>								
1	1	1	1	1	0	0	1	-7补
-							1	
<hr/>								
1	1	1	1	1	0	0	0	-7反
<hr/>								
1	0	0	0	0	1	1	1	-7原