

Linux 命令

1.有问题找男人 man

3: 库函数 printf

2.进程（任务管理器）

查看进程信息: ps top

发送信号 kill

3.后台程序: &或 ctrl+Z

查看后台程序: jobs ps

将后台任务调到前台: fg

4.网络

查看 IP 地址: ifconfig

测试网络是否连通: ping

路由: route

查看 TCP/IP 状态: netstat

寄存器 (CPU 的小太监)

CPU: 运算器/控制器/寄存器

将硬盘中的代码，加载到内存中运算；实际上内存不进行运算，而是交给寄存器运算，寄存器运算完成后，将结果发送给内存。

exit 和 return 区别

return: 终止 return 所在的函数

exit: 结束整个程序（进程）

=====预处理=====

.c 到.o 经历了什么：编译器 gcc/一个程序，生成到可执行文件，经历了哪些过程

gcc hello.c -o demo (是经过：预处理、编译、汇编、链接的过程) :

C语言编译过程:

预处理: gcc -E hello.c -o hello.i
编译: gcc -S hello.i -o hello.s
汇编: gcc -c hello.s -o hello.o
链接: gcc hello.o -o hello_elf

1) 预处理: 宏定义展开、头文件展开、条件编译等，同时将代码中的注释删除，这里并不会检查语法。
2) 编译: 检查语法，将预处理后文件编译生成汇编文件
3) 汇编: 将汇编文件生成目标文件(二进制文件)
4) 链接: C 语言写的程序是需要依赖各种库的，所以编译之后还需要把库链接到最终的可执行程序中去
1dd: 查看依赖的动态库

=====宏定义=====

面试题：用宏定义

求两个数大小 #define max2(a,b) (a)>(b)?(a):(b)

求三个数大小 #define max3(a,b,c) (a)>[max2(b,c)]?(a):[max2(b,c)]

=====条件编译=====

一般情况下，源程序中所有的行都参加编译。但有时希望对部分源程序行只在满足一定条件时才编译，即对这部分源程序行指定编译条件。

测试存在：

```
# ifdef 标识符  
    程序段 1  
# else  
    程序段 2  
# endif
```

测试不存在：

```
# ifndef 标识符  
    程序段 1  
# else  
    程序段 2  
# endif
```

根据表达式定义：

```
# if 表达式  
    程序段 1  
# else  
    程序段 2  
# endif
```

防止头文件.h 多次包含

头文件.h 重复包含：

因为头文件.h 中只是函数声明，并不是函数定义，因此程序不会报错；但是在预处理阶段，头文件会展开多次而使程序占用空间变大。

解决方案：可以使头文件不重复包含

方法 1: #pragma once

方法 2: #ifndef ...#define ...#endif

头文件重复包含的真正含义：

同一个文件包含 N 次头文件，这个头文件只有一次生效。

=====全局变量分文件编程=====

***全局变量分文件编程：在实际开发中，十分常见的错误！

规则：

普通全局变量/普通全局函数在多个.c 文件中也不能重复定义（之前的错误理解：在同一个.c 文件中）

例 1：

在 main.c 和 test.c 两个文件中都定义了全局变量 a，编译时，就会造成 a 重复被定义！

main.c

test.c

```
int a = 0;
```

```
int a = 0;
```

gcc main.c test.c

编译报错：多次定义

例 2：

在 a.h 中定义一个全局变量 a，在 main.c 和 test.c 中引用 a.h 头文件→实质上产生的效果与例 1 是一样的，分析：

main.c 和 test.c 文件在预处理阶段，将会把各自的头文件 a.h 展开，等价于 main.c 和 test.c 文件中都定义了全局变量 a ➔ 同例 1 一样，全局变量重定义的编译错误。

main.c

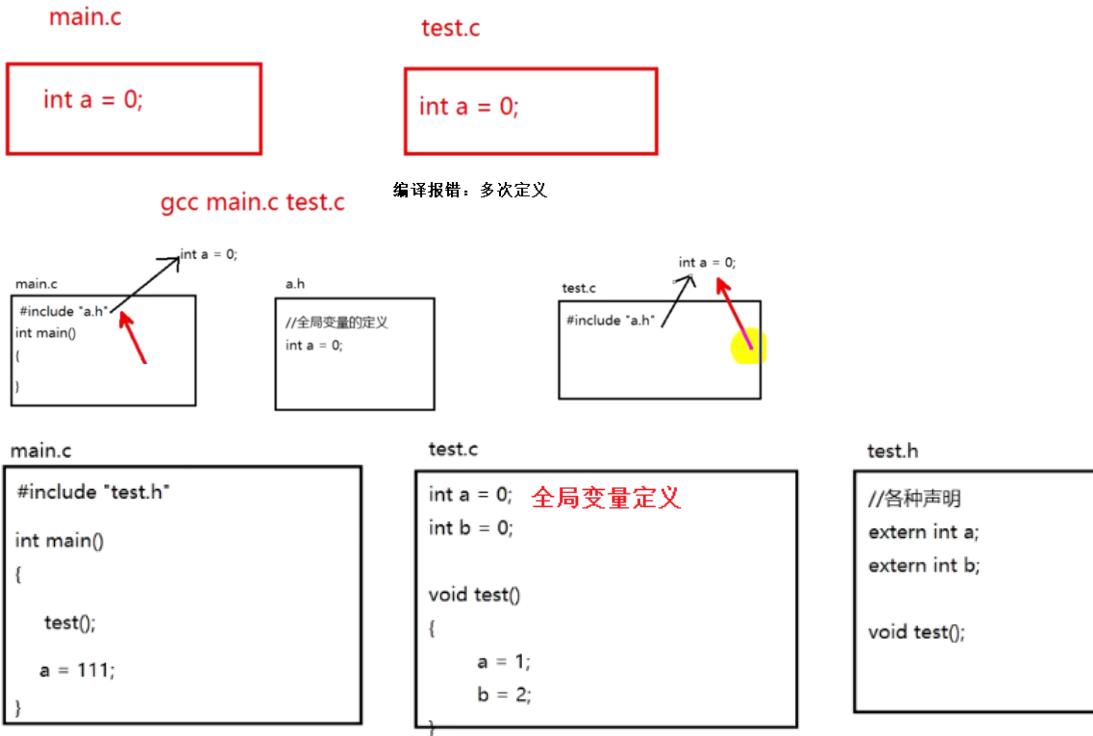
```
#include "a.h"  
int main()  
{  
}
```

a.h

```
//全局变量的定义  
int a = 0;
```

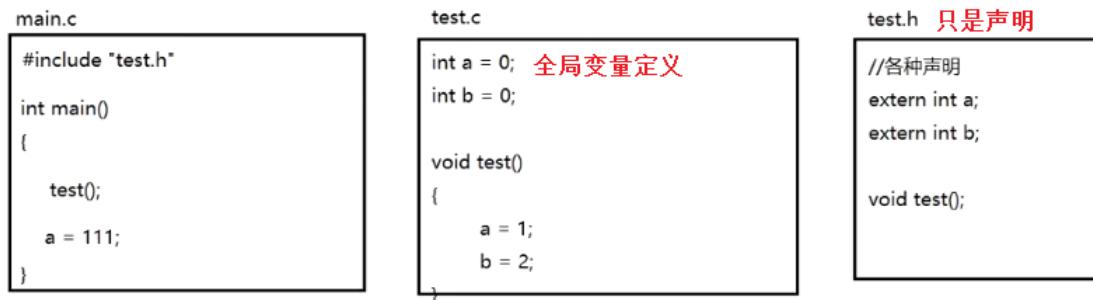
test.c

```
#include "a.h"  
int a = 0;
```



全局变量分文件的结论：

- [1]建立一个全局变量头文件.h: 只包含普通全局变量/普通全局函数的声明 extern
- [2]所有的.c 中每定义一个全局变量，就将该全局变量在.h 头文件中添加声明
- [3]用到头文件中全局变量的.c 文件，直接包含.h 头文件即可



=====兼容 C++ 编译器=====

使 C 语言代码，用 C++ 编译器编译时，按照用 GCC 编译，而不用 G++

如果是 C++ 编译器，用 C 的标准编译

```
2) 让c代码可以在c++编译器编译运行
// __cplusplus是编译器提供好的宏，不是自定义的
#ifndef __cplusplus
extern "C" {
#endif // __cplusplus

// 函数的声明

#ifndef __cplusplus
}
#endif // __cplusplus
```

=====C 语言基础=====

有符号无符号 int : %d 和%u

有符号无符号 char 的范围: 0~255 和-128~127 (注意越界)

**类型限定符: volatile

extern	声明一个变量, extern 声明的变量没有建立存储空间。 extern int a;
const	定义一个常量, 常量的值不能修改。 const int a = 10;
volatile	防止编译器优化代码
register	定义寄存器变量, 提高效率。register 是建议型的指令, 而不是命令型的指令, 如果 CPU 有空闲寄存器, 那么 register 就生效, 如果没有空闲寄存器, 那么 register 无效。

字符数组和字符串

字符串数组初始化: 小心乱码

```
char a[10] = {'a', 'b'}; //后面自动补0  
printf("%s\n", a); //正常
```

```
char a[] = {'a', 'b', 0};  
char a[] = {'a', 'b', '\0'};  
printf("%s\n", a); //正常
```

```
char a[] = {'a', 'b', '0'};  
printf("%s\n", a); //乱码, 没有结束符
```

```
char buf[] = "hello"; //以字符串初始化, 自动隐藏结束符'\0'
```

字符串是字符数组: 只有以\0结尾的字符数组, 才是字符串

```
char a1[] = {'a', 'b', 'c'}; //字符串数组, 不是字符串  
printf("a1 = %s\n", a1); //乱码, 因为没有结束符
```



```
char a2[] = {'a', 'b', 'c', 0}; //字符串  
printf("a2 = %s\n", a2);  
char a3[] = {'a', 'b', 'c', '\0'}; //字符串  
printf("a2 = %s\n", a2);
```



```
char a5[] = {'a', 'b', 'c', '\0', 'h', 'e', '\0'};  
printf("a5 = %s\n", a5); //abc" 发生截断
```

```
char a6[10] = {'a', 'b', 'c', '\0'}; //前面3个字符赋值为a, b, c, 后面自动赋值为0  
printf("a6 = %s\n", a6);
```

```
//1、常用初始化, 使用字符串初始化, 在字符串结尾自动加结束符数字0  
//2、这个结束符, 用户看不到(隐藏), 但是是存在的  
char a7[10] = "abc";  
printf("a7 = %s\n", a7);
```

```
char a8[] = "abc";  
printf("sizeof(a8) = %lu\n", sizeof(a8)); //字符串自动隐藏一个结束符
```

随机数的产生 srand/rand: 面试题→产生四位随机数****

srand(time(NULL))用时间设置种子, rand 产生随机数

```
#include <time.h>  
time_t time(time_t *t);  
功能: 获取当前系统时间  
参数: 常设置为 NULL  
返回值: 当前系统时间, time_ 相当于 long 类型, 单位为毫秒
```



```
#include <stdlib.h>  
void srand(unsigned int seed);  
功能: 用来设置 rand() 产生随机数时的随机种子  
参数: 如果每次 seed 相等, rand() 产生随机数相等  
返回值: 无
```



```
#include <stdlib.h>  
int rand(void);  
功能: 返回一个随机数值  
参数: 无  
返回值: 随机数
```

//先设置种子, 种子设置一次即可
//如果srand()参数一样, 随机数就一样
//srand(100);

//time (NULL) 功能获取系统当前时间, 由于时间会变, srand() 也会改变
srand((unsigned int)time (NULL));

```
int i = 0;  
int num;  
for(i = 0; i < 10; i++)  
{  
    num = rand(); //rand() 产生随机数  
    printf("num = %d\n", num);  
}
```

=====格式化输入/输出=====

printf 格式化输出到屏幕 sprintf 格式化输出到指定数组 dst
printf("a = %d, b = %c ,d = %s\n",a,b,c);
sprintf(dst, "a = %d, b = %c ,d = %s\n",a,b,c);

scanf 从屏幕输入值到变量 sscanf 从字符串输入值到变量
scanf("%d %d %d", &a,&b,&d); //默认分割符是空格

```
sscanf
int a, b, c;
char buf[] = "1 2 3";
//从buf中以指定的格式提取内容
sscanf(buf, "%d %d %d", &a, &b, &c);
//提取整形变量是最方便的
char str[] = "a = 1, b = 2, c = 3";
sscanf(str, "a = %d, b = %d, c = %d", &a, &b, &c);
```

=====字符串处理函数: stdin 键盘/stdout 屏幕=====

0-重点: 空格和\0 不是一个东西, 之前理解的误区(之前一直一位空格就是\0 字符)

例如:

带空格的字符串, 打印时不会被截断, 空格依然会被打印出来
strlen 并不会在空格处结束; 会在\0 处结束
strcpy/strcat 一样可以将 src 中的空格复制和追加到 dst

1-字符串长度: size_t strlen(const char* s) 与 sizeof

[1]sizeof: 计算一个数据的大小, 单位为字节, 返回值 size_t(unsigned int)

sizeof() 测数据类型大小, 不会因为结束符提前结束

[2]计算字符串 s 长度, 不包含字符串结束符(遇到\0 结束)

sizeof 和 strlen 的区别

sizeof 是测数据类型长度, 不会因为结束符\0 提前结束
strlen 会以结束符\0 结束

简单的说, 只有两种情况:

如果 s 中间没有\0, sizeof(s) = strlen(s)+1

如果 s 中有\0, sizeof(s)不变, strlen(s)=\0 之前的长度

面试题 1: 手撕 strlen

```
int my_strlen(char str[])
{
    int i;
    for (i = 0; str[i] != '\0'; i++);
    return i;
}
```

2-字符串拷贝 strcpy/strncpy

介绍:

strcpy: 将 src 中全部字节, 拷贝到 dst

strncpy: 将 src 中, 指定 len 长度的字节, 拷贝到 dst

**strcpy/strncpy 区别:

strcpy 拷贝\0 之前的数据(不包括\0); 拷贝完成后, 在 dst 末尾自动补\0

strncpy 拷贝\0 以及\0 之前的数据; 拷贝完成后, 不在 dst 末尾补\0

面试题 2：手撕 strcpy

```
void my strcpy(char* dst, char* src)
{
    int i = 0;
    while (src[i] != '\0')
    {
        dst[i] = src[i];
        i++;
    }
    dst[i] = '\0';
}
```

3-字符串追加 strcat/strncat

4-字符串比较 strcmp/strncmp

5-字符串查询 strchr/strstr

strchr: 在 str 中查找一个字符 ch, 查询成功返回第一次出现 ch 的地址

strstr: 在 str 中查找一个字符串 str, 查询成功返回第一次出现 str 的地址

面试题 3：查找字符串中子字符串出现的次数：strstr

```
int f(char* const str, int n, char* substr, int m)
{
    int cnt = 0;
    char* p = str; //为了不修改str, 用p操作str
    char* res = NULL; //res: strstr返回的地址
    while (1)
    {
        res = strstr(p, substr);
        if (res == NULL) //如果查找失败
            break;
        //如果查找成功
        p = res + m; //p指针移动到res+m位置, 进行下一次查找
        cnt++;
    }
    return cnt;
}
```

6-字符串切割 strtok: 会破坏原来的字符串

char* strtok(源字符串, **切割字符**);

[1]功能: 当 str 中发现 ch 时, 就会将 ch 字符改成\0 (当连续出现多个时, 只替换第一个 ch)

[2]返回值: 切割后的字符串 (调用 strtok 成功, 源字符串被破坏)

使用方法:

第一次调用, 第一个参数: 切割字符串

之后每次调用, 第一个参数: NULL

代码举例:

```
int main()
{
    char buf[10] = "ABC,DF,XY";
    char* p = NULL;
    p = strtok(buf, ","); //第一次切割
    while (p != NULL) //切割成功
    {
        printf("%s\n", p);
        p = strtok(NULL, ","); //第二次起切割, 第一个参数写NULL
    }
    system("pause");
}
```



面试题 4：数组的每个字段都是随机 int 数，求和

```
//每个字段都是一个整数，字段的数量随机，字段之间用,分割，求所有整数的和  
char a[100] = "12,43,65,13,97,54,19";  
  
int get_sun(char* str)  
{  
    int res = 0;  
    int main()  
    {  
        char str[] = "1,2,3,4";  
        int n = get_sun(str);  
        system("pause");  
    }  
    ...  
    char* numstr = strtok(str, ",");  
    while (1)  
    {  
        if (numstr == NULL)  
            break;  
        int tmp = atoi(numstr);  
        res += tmp;  
        numstr = strtok(NULL, ",");  
    }  
    return res;  
}
```

7-atoi/atof/atol 将字符串转换成 int/float/long

[1]将字符串转换成 int/float/long

内部原理：

扫描字符串，跳过前面空格字符，直到遇到数字或正负号才开始转换，遇到非数字或\0转换结束，将结果返回。

[2]将 int/float/long 转换成字符串：sprintf

面试题 5：手撕 atoi

```
int my_atoi(char* str)  
{  
    int flag = 0; //正负标志位  
    char* p = str;  
    switch (p[0])  
    {  
        case '-':  
            p++; flag = 0;  
            break; //如果有+-号，跳过  
        case '+':  
            p++; flag = 1;  
            break;  
        default:  
            flag = 1;  
    }  
    int res = 0;  
    while (*p != '\0')  
    {  
        res = [上一次值×10] + [(*p - '0')];  
        p++; //当前的值  
    }  
    if (flag == 1)  
        return res;  
    else  
        return -res;  
}
```

```
int main()  
{  
    char str[] = "-123333";  
    int n = my_atoi(str);  
    system("pause");  
}
```

面试题 6：两头堵

去掉两边的无效字符，获取剩下有效字符的个数，并把结果拷贝出去使用。
即：
 获取ABCDABCD长度
 将ABCDABCD拷贝出去

```
int fun(char* str, int n, char* buf)
{
    int cnt = 0;
    char* start = str; //首元素地址
    char* end = str + strlen(str) - 1; //尾元素地址
    while (*start == ' ' && start != '\0') //从左到右
        start++;
    while (*end == ' ' && end != str) //从右到左
        end--;
    cnt = end - start + 1;
    //将结果拷贝出去使用
    strncpy(buf, start, cnt);
    buf[cnt] = '\0'; //结束符
    return cnt;
}
```

char *p = "aaabbbddddd";
char *start = p;
char *end = p + strlen(p) - 1;

```
int main()
{
    char str[100] = "ABCDABCD ";
    int len = strlen(str);
    char buf[1024];
    int n = fun(str, len, buf);
    system("pause");
}
```

=====指针=====

一. 指针

0.作用：通过函数改变实参，必须地址传递

```
void swap2(int *m, int *n)
{
    int tmp;
    tmp = *m;
    *m = *n;
    *n = tmp;
}
```

1.指针大小

32位：4字节

64位：8字节

2.方括号[]不是数组的专属：p[i]实质上等于*(p+i)

3.万能指针 void*：指向任何类型的指针

任何类型的指针变量，都能赋值给 void* 变量

4.const 与指针：const 与*的位置

const 修饰*还是修饰变量

const int *p 和 int const *p 指针变量 p 指向的地址不能修改

int* const p 指针变量 p 的值不能修改

5.返回局部变量的地址 返回全局变量的地址

(1) 返回局部变量的地址

规定：Linux 不允许返回局部变量的地址

```
int* fun()
{
    int a;
    return &a; //当函数返回后，a自动释放
}
int main()
{
    int* p = fun(); //不合法：p接受到无效的地址（a空间被释放）
    *p = 200;
```

(2) 返回全局变量的地址

```
int a; //全局变量
int* fun()
{
    return &a; //当函数返回后, a不释放
}
int main()
{
    //方法1
    int* p = fun(); //合法: p指向a的地址
    *p = 200; //用*p取出a地址中的变量a, 将其改成200, 此时a=200
    ...
    //方法2
    *(fun()) = 300; /*(fun()): 取出a地址中的变量a, 修改为300, 此时a=300
```

二. 指针和数组

1. 数组名

数组名就是数组首元素的地址&A[0]; 数组名是常量, 不能被修改

```
char A[] = "ABCDE"; //末尾自动补充'\0'
int n = sizeof(A) / sizeof(*A); //包括最后一个\0

char* head = A; //指向数组首元素地址
char* tail = A + n; //指向数组尾元素地址
while (head != tail)
{
    printf("%c\n", *head);
    head++;
}
```

2. 数组名和指针

```
void fun(int A[]);      void fun(int A[3][4]);
void fun(int* A);        void fun(int** A); //二维数组不是二级指针
void fun(int *A[4]);
```

3. 数组名作形参, 退化为指针

=====字符串常量: data 区 (文字常量区) =====

1. 字符串常量



结论:

指针 p 指向字符串常量，不能对该 p 指向的内存 p[i]

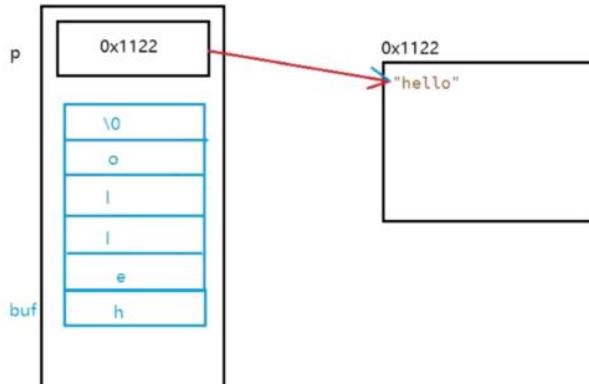
将字符串常量的值，一个一个赋值给 A 的每个元素，可以该 A[i]

2. 字符串常量初始化问题

```
int main(int argc, char *argv[])
{
    //1. p指针保存了"hello"的地址
    //2. 指针所指向的内存不能修改
    char *p = "hello";

    //1. 把"hello"一个一个字符放在buf数组中
    //2. 数组的元素可以修改
    char buf[] = "hello";

    return 0;
}
```



=====内存管理：局部/全局/static=====

1. 普通函数和 static 函数的区别

普通局部变量和 static 局部变量区别：

1、内存分配和释放

- a) 普通局部变量只有执行到定义变量的语句才分配空间
- b) static 局部变量在编译阶段（函数还没有执行），变量的空间已经分配
- c) 普通局部变量离开作用域 {}，自动释放
- d) static 局部变量只有在整个程序结束才自动释放

2、初始化

- a) 普通局部变量不初始化，值为随机数
- b) static 局部变量不初始化，值为 0
- c) static 局部变量初始化语句只有第一次执行时有效
- d) static 局部变量只能用常量初始化

面试题：为什么用局部变量给 static 变量初始化会失败？

答案：因为 static 在编译阶段就分配内存，而局部变量是在执行到定义语句才分配内存。因此局部变量不存在，怎么可能初始化 static 变量！

2. 普通全局变量：所有文件只能有一个普通全局变量的定义！

```
//1、在{}外面（函数外面）定义的变量为全局变量
//2、只有定义了全局变量，任何地方都能使用此变量
//3、如果使用变量时，在前面找不到此全局变量的定义，需要声明后才能使用
//4、全局变量不初始化，默认赋值为0
//5、声明只是针对全局变量，不是针对局部变量
//6、全局变量只能定义一次，可以声明多次
//7、全局变量在编译阶段已经分配空间（函数没有执行前），只有在整个程序结束，才自动释放
//8、所有文件，全局变量只能定义一次，可以声明多次
```

全局变量的缺陷

```
int a;
int a;
int a = 0; //定义，其它是声明
int a;
int a;

//有1次是定义，有3次是声明
int b;
int b;
int b;
int b;

//1、如果定义一个全局变量，没有赋值（初始化），无法确定是定义，还是声明
//2、如果定义一个全局变量，同时初始化，这个肯定是定义
```

解决缺陷的方案

```
//1、定义一个全局变量，建议初始化  
int a = 10;  
  
//2、如果声明一个全局变量，建议加extern  
extern int a;
```

***全局变量分文件编程：在实际开发中，十分常见的错误！

规则：

普通全局变量/普通全局函数在多个.c 文件中也不能重复定义（之前的错误理解：在同一个.c 文件中）

例 1：

在 main.c 和 test.c 两个文件中都定义了全局变量 a，编译时，就会造成 a 重复被定义！

main.c

test.c

int a = 0;

int a = 0;

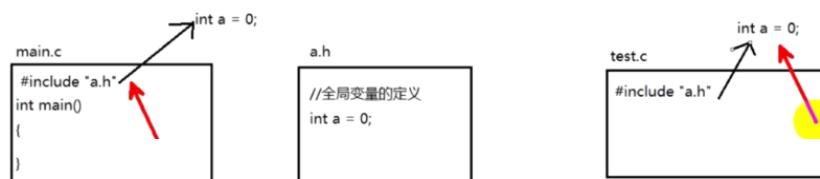
gcc main.c test.c

编译报错：多次定义

例 2：

在 a.h 中定义一个全局变量 a，在 main.c 和 test.c 中引用 a.h 头文件→实质上产生的效果与例 1 是一样的，分析：

main.c 和 test.c 文件在预处理阶段，将会把各自的头文件 a.h 展开，等价于 main.c 和 test.c 文件中都定义了全局变量 a → 同例 1 一样，全局变量重定义的编译错误。



main.c

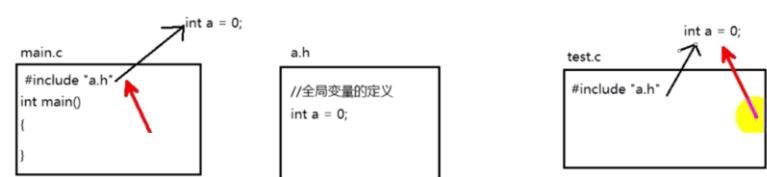
test.c

int a = 0;

int a = 0;

gcc main.c test.c

编译报错：多次定义



main.c <pre>#include "test.h" int main() { test(); a = 111; }</pre>	test.c <pre>int a = 0; 全局变量定义 int b = 0; void test() { a = 1; b = 2; }</pre>	test.h <pre>//各种声明 extern int a; extern int b; void test();</pre>
--	---	--

全局变量分文件的结论：

- [1]建立一个全局变量头文件.h: 只包含普通全局变量/普通全局函数的声明 `extern`
- [2]所有的.c 中每定义一个全局变量，就将该全局变量在.h 头文件中添加声明
- [3]用到头文件中全局变量的.c 文件，直接包含.h 头文件即可

main.c <pre>#include "test.h" int main() { test(); a = 111; }</pre>	test.c <pre>int a = 0; 全局变量定义 int b = 0; void test() { a = 1; b = 2; }</pre>	test.h 只是声明 <pre>//各种声明 extern int a; extern int b; void test();</pre>
--	---	---

3.static 全局变量：文件作用域

- a) static全局变量和普通全局变量的区别就是作用域不一样（文件作用域）
- b) `extern`关键字只适用于普通全局变量
- c) 普通全局变量，所有文件都能使用，前提需要声明
- d) static全局变量只能本文件使用，别的文件不能使用
- e) 不同文件只能出现一个普通全局变量的定义
- d) 一个文件只能有一个static全局变量的定义，不同文件间的static全局变量，就算名字相同，也是没有关系的2个变量
- e) static将全局变量限制在一个文件中，另一个文件即使使用`extern`扩展，也会失败。

4.普通(全局)函数和 static(全局)函数的区别：文件作用域

- a) 所有文件只能有一次普通函数的定义
- b) 一个文件可以有一个static函数的定义
- c) 普通函数所有文件都能调用，前提是使用前声明
- d) static函数只能在定义所在的文件中使用

5.总结：作用域/生命周期

类型	作用域	生命周期
auto 变量	一对 {} 内	当前函数
static 局部变量	I 一对 {} 内	整个程序运行期
extern 变量	整个程序	整个程序运行期
static 全局变量	当前文件	整个程序运行期
extern 函数	整个程序	整个程序运行期
static 函数	当前文件	整个程序运行期
register 变量	一对 {} 内	当前函数

=====内存布局 text/data/bss/堆/栈=====

1.面试题：内存分区：程序运行前/后

编译好

的可执

在程序没有执行前，行程序 的几个内存分区已经确定，虽然分区确定，但是没有加载内存，程序只有运行时才加载内存：

text(代码区)：只读，函数

data：初始化的数据，全局变量，static变量，文字常量区（只读）

bss：没有初始化的数据，全局变量，static变量

当运行程序，加载内存，首先根据前面确定的内存分区(text, data, bss)先加载，然后额外加载2个区

text(代码区)：只读，函数

data：初始化的数据，全局变量，static变量，文字常量区（只读）

bss：没有初始化的数据，全局变量，static变量

stack(栈区)：普通局部变量，自动管理内存，先进后出的特点

heap(堆区)：手动申请空间，手动释放，整个程序结束，系统也会自动回收；

总结：

类型	作用域	生命周期	存储位置
auto 变量	一对 0 内	当前函数	栈区
static 局部变量	一对 0 内	整个程序运行期	初始化在 data 段，未初始化在 BSS 段
extern 变量	整个程序	整个程序运行期	初始化在 data 段，未初始化在 BSS 段
static 全局变量	当前文件	整个程序运行期	初始化在 data 段，未初始化在 BSS 段
extern 函数	整个程序	整个程序运行期	代码区
static 函数	当前文件	整个程序运行期	代码区
register 变量	一对 0 内	当前函数	运行时存储在 CPU 寄存器
字符串常量	当前文件	整个程序运行期	data 段

栈越界：函数递归调用

栈区：用完就释放，用完就释放，比较小

堆区：大

[1]举例：

```
int main(int argc, char *argv[])
{
    //语法上没有问题，栈区分配很大的内存，越界了，导致段错误
    int a[100000000000] = {0};
```

[2]查看栈空间大小：

```
edu@edu:~/share/c_code/day10$ ulimit -a
core file size          (blocks, -c) 0
data seg size            (kbytes, -d) unlimited
cheduling priority       (-e) 0
file size                (blocks, -f) unlimited
pending signals          (-i) 7781
max locked memory        (kbytes, -l) 64
max memory size          (kbytes, -m) unlimited
open files               (-n) 1024
pipe size                (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority        (-r) 0
stack size               (kbytes, -s) 8192
cpu time                 (seconds, -t) unlimited
max user processes        (-u) 7781
virtual memory            (kbytes, -v) unlimited
file locks               (-x) unlimited
```

2. 内存操作函数 (4个)

之前的四个函数，只能操作 `char ch[]` 和 `string` 类型的数据
而内存操作函数，都可以操作。

`memset()` //主要用于清空

```
普通类型清0
int a;
memset(&a, 0, sizeof(a)); //常用
//中间参数虽然是整型，但是以字符处理
memset(&a, 97, sizeof(a));
```

```
数组清0
int b[10];
memset(b, 0, sizeof(b));
memset(b, 0, 10 * sizeof(int)); //数组赋初值
char str[10];
memset(str, 'a', sizeof(str));
```

`memcpy()` //拷贝

有了 `strncpy` 为什么使用 `memcpy`: 因为 `strncpy` 不能拷贝\0

```
char p[] = "hello\0mike"; //以字符串初始化，自动在默认隐藏一个结束符'\0';
char buf[100];
strncpy(buf, p, sizeof(p)); 遇见\0, 停止拷贝
printf("buf1 = %s\n", buf); //buf1 = hello
printf("buf2 = %s\n", buf + strlen("hello") + 1); //buf2 =
memset(buf, 0, sizeof(buf));
memcpy(buf, p, sizeof(p)); 遇见\0, 依然可以拷贝
printf("buf3 = %s\n", buf); //buf3 = hello
printf("buf4 = %s\n", buf + strlen("hello") + 1); //buf4 = mike
```

面试：`memcpy` 内存重叠问题

```
memset(&A[2], A, 5*sizeof(A[0]));
```

解决方案：`memmove`

如果出现内存重叠，使用 `memmove()`，用法和 `memcpy` 一样

`memmove` //不存在内存重叠问题

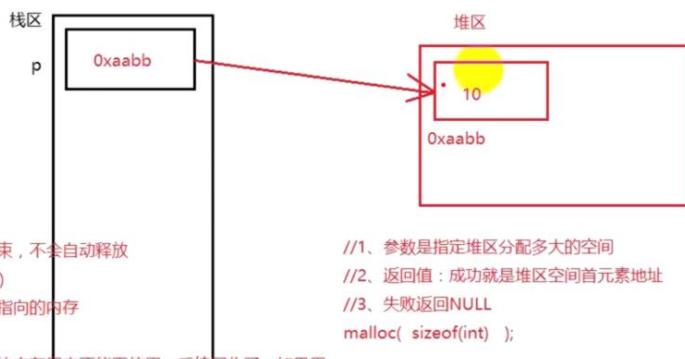
`memcmp` //比较：和 `strcmp` 一样

```
int flag = memcmp(a, b, 10 * sizeof(int));
```

3. 指针指向堆区空间

[0]介绍

```
int main(int argc, char *argv[])
{
    int *p;
    p = (int *)malloc(sizeof(int));
    *p = 10;
    printf("%p = %d\n", *p);
    1. 动态分配的空间，如果程序没有结束，不会自动释放
    return 0;
} //1. 参数是指定堆区分配多大的空间
//2. 返回值：成功就是堆区空间首元素地址
//3. 失败返回NULL
//malloc( sizeof(int) );
```



[1]空指针：指向 `NULL` 的指针（没有指向地址的指针）

[2]野指针：指向非法地址的指针

例 1：

```
int* p; //没有初始化的指针，会自动给 p 初始化，p 指向一个非法地址  
*p = 100; //表示给 p 指向的非法地址，赋值 100→出现段错误
```

例 2：

```
int* p = (int*)malloc(sizeof(int)); //p 指向堆区分配的内存  
free(p); //内存释放后，p 指向无效的地址  
*p = 100; //使用 p 指向的内存，编译器并不会检测到错误；但是会带来很大的隐患，  
出现未知的错误
```

[3]避免重复释放内存

说明：同一块内存不能被释放两次，否则会 done

```
if(p!=NULL) //释放前：判断p不为空，才释放  
{  
    free(p);  
    p = NULL; //释放后：将p设为NULL  
}
```

[4]堆区越界

```
int main(int argc, char *argv[])
{
    char *p = NULL;

    p = (char *)malloc(0); //分配0个空间给p
    if(p == NULL)p有值，导致p!=NULL
    {
        printf("分配失败\n");
        return 0;
    }

    strcpy(p, "mikejiang"); 可怕的是，依然可以赋值
    printf("%s\n", p); 也能打印结果：mikejiang

    free(p); 也可以释放
    return 0; 但是：程序就是错误的 ----> 将代码放到VS
               中运行，出现如图所示结果
}
```

程序分析：
在linux下程序运行没有错误，就很可怕了！

结论：
很明显程序是错误的，堆越界----运行正确的错误代码是
非常可怕的



4.面试：

(1)返回栈区地址

```
int *fun()
{
    int a = 10;
    return &a; //函数调用完毕，a释放
}

int main(int argc, char *argv[])
{
    int *p = NULL;
    p = fun();
    *p = 100; //操作野指针指向的内存，err

    return 0;
}
```

edu@edu:~/share/c_code/day11/01_代码分析\$./a.out
段错误 (核心已转储)

(2)返回 Data 区地址

```
int *fun()
{
    static int a = 10;
    return &a; //函数调用完毕，a不释放
}

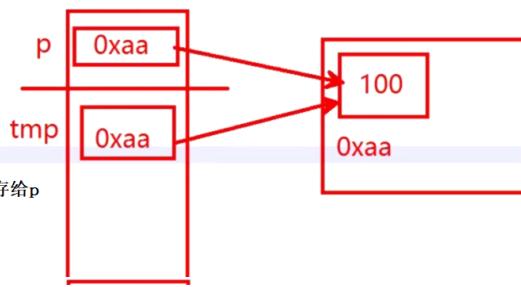
int main(int argc, char *argv[])
{
    int *p = NULL;
    p = fun();
    *p = 100; //ok
    printf("p = %d\n", *p);

    return 0;
}
```

(3)值传递 1

正确！

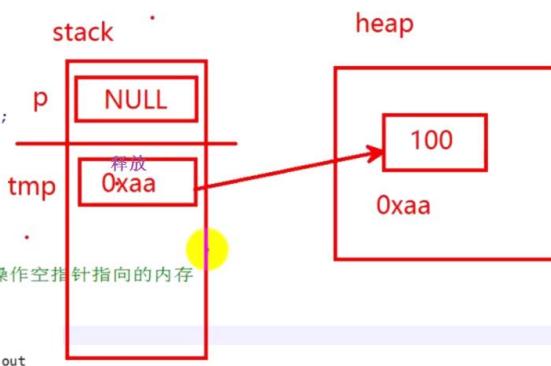
```
void fun(int *tmp)
{
    *tmp = 100; //函数执行完，tmp释放，但是堆空间不释放
}
int main(int argc, char *argv[])
{
    int *p = NULL;
    p = (int *)malloc(sizeof(int)); 分配内存给p
    fun(p); //值传递
    printf("%d\n", *p); //ok, *p为100
    return 0; *p操作堆空间内存，执行正确
}
```



(4)值传递 2

错误！

```
void fun(int *tmp)
{
    tmp = (int *)malloc(sizeof(int));
    *tmp = 100; //执行完毕，tmp释放
}
int main(int argc, char *argv[])
{
    int *p = NULL;
    fun(p); //值传递
    printf("%d\n", *p); //err, 操作空指针指向的内存
    return 0; 操作*p，出现段错误
}
```



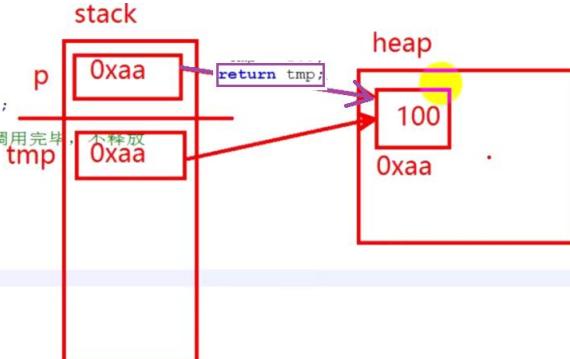
(5)返回堆区地址---将(4)修改

```
#include <stdlib.h>

int *fun()
{
    int *tmp = NULL;
    tmp = (int *)malloc(sizeof(int));
    *tmp = 100;
    return tmp; //返回堆区地址，函数调用完毕
}

int main(int argc, char *argv[])
{
    int *p = NULL;
    p = fun();
    printf("%d\n", *p); //ok

    //堆区空间，使用完毕，手动释放
    if (p != NULL)
    {
        free(p);
        p = NULL;
    }
    return 0;
}
```



(6)二级指针作形参（下面两种写法等价，第二种更加简单）

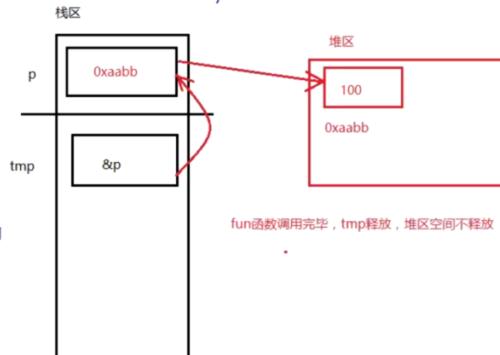
```
#include <stdio.h>
#include <stdlib.h>

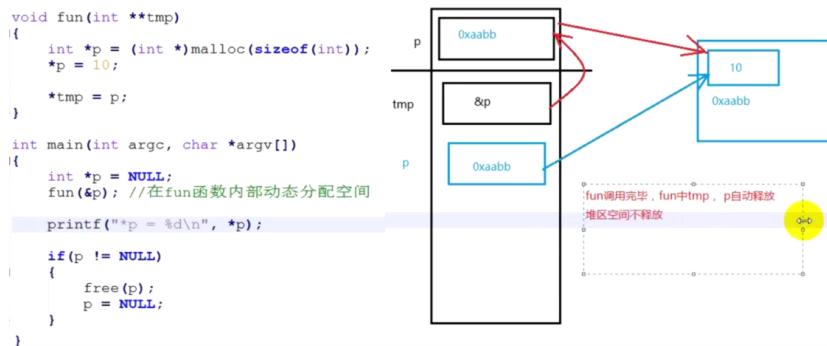
void fun(int **tmp)
{
    *tmp = (int *)malloc(sizeof(int));
    **tmp = 100;
}

int main(int argc, char *argv[])
{
    int *p = NULL;
    fun(&p); //在fun函数内部动态分配空间

    printf("%d\n", *p);

    return 0;
}
```



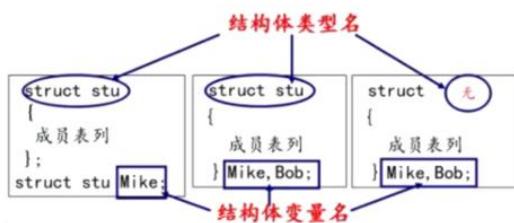


====结构体(复合类型/自定义类型)=====

1. 定义结构体变量

定义结构体变量的方式：

- 先声明结构体类型再定义变量名
- 在声明类型的同时定义变量
- 直接定义结构体类型变量（无类型名）



```

//定义结构体变量
//1、类型名 变量名
struct Student stu; //别忘了struct关键字

//1、结构体变量初始化，和数组一样，要使用大括号
//2、只有在定义是才能初始化
struct Student stu2 = {18, "mike", 59};

struct Student *p;

```

结构体数组

```

struct Student a[5] =
{
    {18, "mike", 59},
    {22, "jiang", 66},
    {33, "xiaobai", 87},
    {28, "lily", 77},
    {30, "lucy", 68}
};

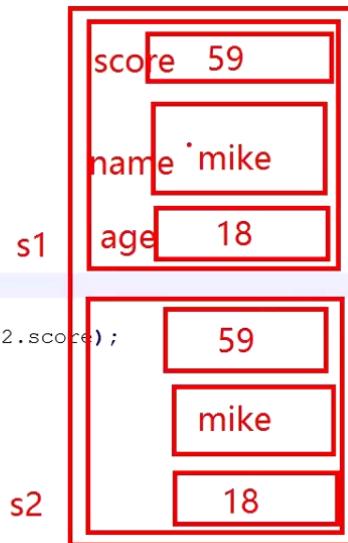
```

2. 结构体变量相互赋值，存在浅拷贝的风险

```
int main()
{
    int a = 10;
    int b;
    //1、把a的值给了b
    //2、a和b有关系吗
    b = a;

    //相同类型的2个结构体变量可以相互赋值
    struct Student s1 = {18, "mike", 59};
    struct Student s2;
    s2 = s1;
    printf("%d, %s, %d\n", s2.age, s2.name, s2.score);

    return 0;
}
```



3. 值传递：同样不会更改成功

```
void setStu(struct Student tmp)
{
    tmp.age = 22;
    strcpy(tmp.name, "jiang");
    tmp.score = 77;
    printf("setStu %d, %s, %d\n", tmp.age, tmp.name, tmp.score);
}

int main(int argc, char *argv[])
{
    struct Student s1 = {18, "mike", 59};

    setStu(s1); //通过函数修改成员内容
    printf("%d, %s, %d\n", s1.age, s1.name, s1.score); 结论：没有修改成功
}
```

edu@edu:~/share/c_code/day11\$./a.out
setStu 22, jiang, 77
18, mike, 59

4. 结构体套一级指针

```
struct Test
{
    char *str; //str没有分配内存
};

int main(int argc, char *argv[])
{
    struct Test obj;
    strcpy(obj.str, "mike"); 向str（没有分配内存）中拷贝数据
    printf("str = %s\n", obj.str);
}
```

段错误

解决方案 1: ****成员变量指针指向→Data

```
struct Student
{
    int age;
    char *name;
    int score;
};

int main()
{
    struct Student s;
    s.age = 18;
    str.name = "mike"; //成员变量指针指向文件常量区的字符串
    s.score = 59;

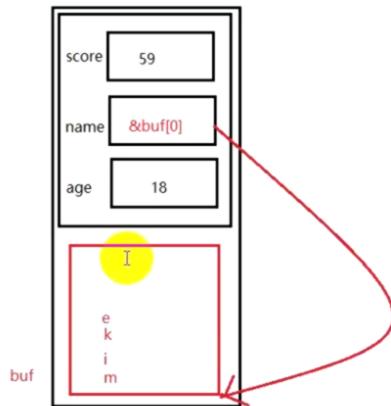
    return 0;
}
```

解决方案 2: 成员变量指针指向→栈

```
//成员变量指针指向栈区空间
int main()
{
    struct Student s;
    s.age = 18;
    {
        char buf[100];
        s.name = buf; //指向栈区空间
        strcpy(str.name, "mike");
        s.score = 59;

        printf("buf = %s\n", buf);

        return 0;
    }
}
```



解决方案 3: 成员变量指针指向→堆

```
//成员变量指针指向堆区空间
int main()
{
    struct Student s;
    s.age = 18;
    //s.name = (char *)malloc(strlen("mikejiangabcsb") + 1) + sizeof(char);
    s.name = (char *)malloc(strlen("mikejiangabcsb") + 1); 0xaa
    strcpy(s.name, "mikejiangabcsb");
    s.score = 59;

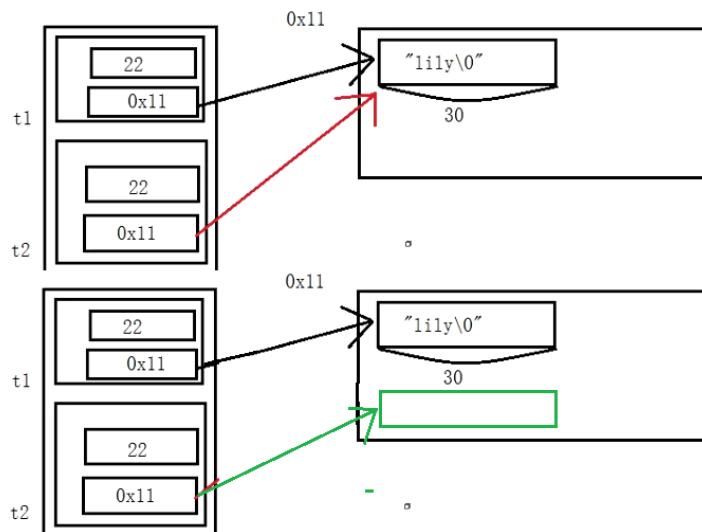
    printf("%d, %s, %d\n", s.age, s.name, s.score);
}
```

5. 深拷贝浅拷贝

```
Teacher t1;
t1.name = (char *)malloc(30);
strcpy(t1.name, "lily");
t1.age = 22;
```

```
Teacher t2;
t2 = t1; 浅拷贝
```

```
Teacher t2;
t2 = t1;
//深拷贝，人为增加内容，重新拷贝一下
t2.name = (char *)malloc(30);
strcpy(t2.name, t1.name);
```



面试 1：字节对齐（偏移量）——空间换时间，效率更高

原则 1：数据成员的对齐规则(以最大的类型字节为单位)。

结构体(struct)的数据成员，第一个数据成员放在 offset 为 0 的地方，以后每个数据成员存放在 offset 为该数据成员大小的整数倍的地方(比如 int 在 32 位机为 4 字节，则要从 4 的整数倍地址开始存储)

原则 2：结构体作为成员的对齐规则。

如果一个结构体 B 里嵌套另一个结构体 A，则结构体 A 应从 offset 为 A 内部最大成员的整数倍的地方开始存储。(struct B 里存有 struct A，A 里有 char, int, double 等成员，那 A 应该从 8 的整数倍开始存储。)，结构体 A 中的成员的对齐规则仍满足原则 1、原则 2。

注意：

1. 结构体 A 所占的大小为该结构体成员内部最大元素的整数倍，不足补齐。
2. 不是直接将结构体 A 的成员直接移动到结构体 B 中

原则 3：收尾工作

结构体的总大小，也就是 `sizeof` 的结果，必须是其内部最大成员的整数倍，不足的要补齐。

struct A{ int a; short b; double c; }; /* aaaabb** ccccccc */	struct A{ short b; int a; double c; }; /* bbbaaa** ccccccc */	struct A{ double c; int a; short b; }; /* ccccccc aaaabb** */	struct A{ double c; short b; int a; }; /* ccccccc bb**aaaa */	struct A { // 结构体嵌套 struct B { char e[2]; int f; double g; short h; struct A j; }; /* aaaa**** bbbbbbb cccc*** */
---	---	---	---	--

指定对齐单位

偏移量大于对齐单位，就换行

```
#pragma pack(2) //指定对齐单位为2
struct A
{
    int a;
    char b;
    short c;
    char d;      总结:
};           当且仅当指定对
/*          齐单位<最大偏移量
   aa
   aa
   b*
   cc
   d*
*/
```

面试 2：位段*****

/*

规则：

指定对其单位是8，但是由于8>最大偏移量，
因此

最大对齐数目 = 最大偏移量

*/

```
#pragma pack(8) //指定对齐单位为8
```

```
struct A
```

```
{
    int a; //最大对齐数目4
    char b;
    short c;
    char d;
```

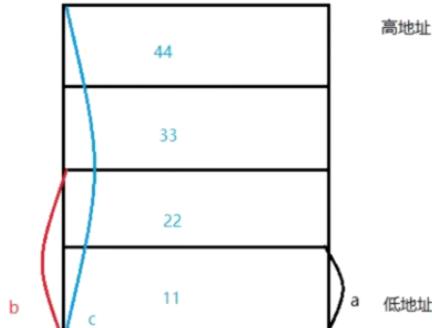
```
};           /*
*/           aaaa
               b*cc
               d***
```

=====共用体 union=====

obj.c = 0x44332211;

```
union Test
{
    unsigned char a;
    unsigned short b;
    unsigned int c;
};

//左边是高位，右边是低位
//高位放高地址，低位放低地址（小端）
```



//2、共用体的大小为最大成员的大小
printf("%lu\n", sizeof(union Test));

//3、共用体公有一块内存，所有成员的地址都一样

```
union Test obj;
printf("%p, %p, %p\n", &obj, &obj.a, &obj.b, &obj.c);
```

//4、给某个成员赋值，会影响到另外的成员

=====枚举 enum=====

```
//第一个成员如果没有赋值， 默认为0， 下一个成员比上一个多1
//枚举类型 enum Color
//成员： 枚举成员， 枚举常量
enum Color
{
    pink, red, green=10, white, blue, yellow
};

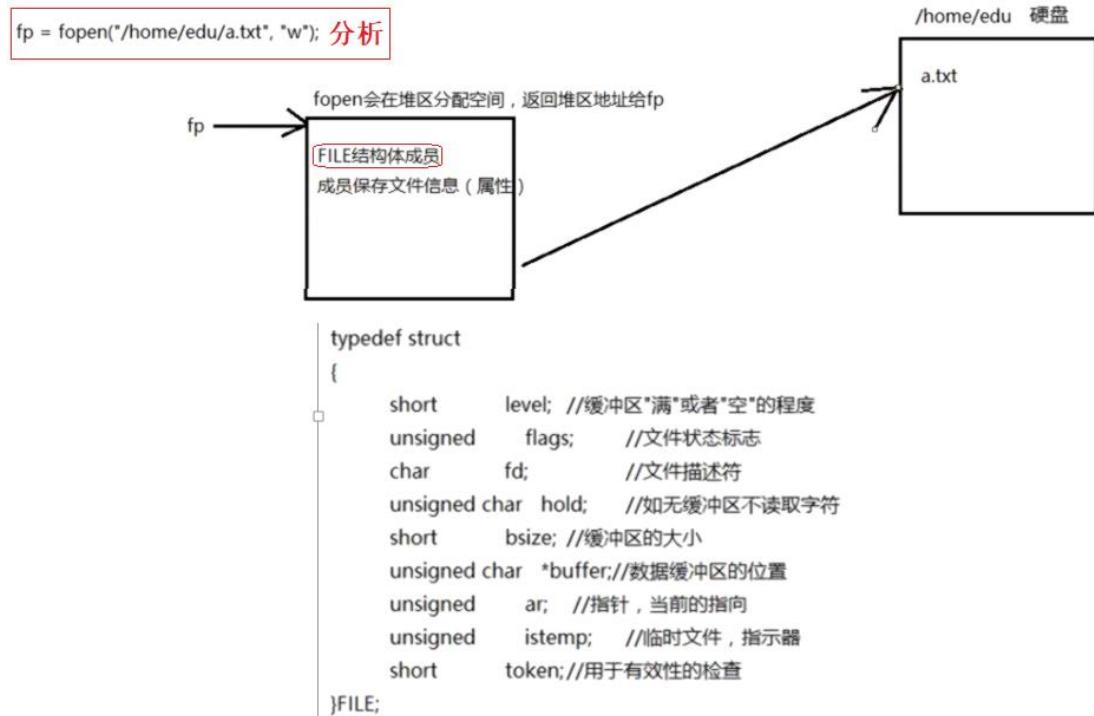
//枚举变量flag
enum Color flag;
// 可以使用枚举成员给flag 赋值
flag = pink; //等价于pink = 0
```

=====文件 FILE=====

1.FILE* fd 指针介绍

执行 `FILE* fp = fopen("a.txt", "w")` 之后，将会分配内存给 `sizeof(FILE)` 大小的空间，返回给 `fp` 指针 → 因此，`fp` 指针指向 `FILE` 结构体，而不是指向文件 → 因此不能直接用 `fp` 指针操作文件！

但是 `fp` 指针指向的 `FILE` 结构体中的成员指针变量，指向文件，可以用它操作文件。在实际的使用时，使用库函数直接操作 `fp` 指针，来间接修改 `fp` 指向的 `FILE` 结构体，进而完成对文件的操作。



2.文件分类: 设备文件/磁盘文件

设备文件：键盘，鼠标

磁盘文件：放在硬盘中的文件（文本文件/二进制文件）

3. 标准设备的三个文件指针：stdin/stdout/stderr

解读：**printf/puts** **scanf/gets** 的本质。

(1) 实际上是将内容，写入到标准输入/标准输出文件中。

(2) 关闭 **fclose(stdin)** **fclose(stdout)** **fclose(stderr)**，后，在调用 **scanf/puts** 等函数，函数执行失效。

C 语言中有三个特殊的文件指针由系统默认打开，**用户无需定义即可直接使用**：

- **stdin**: 标准输入，默认为当前终端（键盘），我们使用的 **scanf**、**getchar** 函数默认从此终端获得数据。
- **stdout**: 标准输出，默认为当前终端（屏幕），我们使用的 **printf**、**puts** 函数默认输出信息到此终端。
- **stderr**: 标准出错，默认为当前终端（屏幕），我们使用的 **perror** 函数默认输出信息到此终端。

4. 相对路径

1、在linux，相对路径相对于可执行程序

2、VS

- 编译同时运行程序，相对路径，相对于xxxx.vcxproj（项目文件）所在的路径
- 如果直接运行程序，相对路径相对于可执行程序

3、Qt

- 编译同时运行程序，相对路径，相对于debug所在的路径
- 如果直接运行程序，相对路径相对于可执行程序

5. 文件 API

[1] **fopen/fclose** 打开/关闭

// "w"，如果文件不存在，新建，如果文件存在，清空内容再打开
// "r"，如果文件不存在，打开失败
// "a"，如果文件不存在，新建，如果文件存在，光标自动放在文件末尾

[2] 从键盘逐字读取：**fgetc/getc** 向 stream 逐字写出：**fputc/putc**

从 stream，逐字读取 char 给返回值 将 char，逐字写入 stream

1 int fgetc(FILE *stream); 从流 stream 中取字符	int putc(int char, FILE *stream)	把参数 char 指定的字符（一个无符号字符）写入流 stream 中，并把位置标识符往前移动。
1 int getc(FILE *stream); 从流 stream 中取字符	int fputc(int char, FILE *stream)	

【参数】参数 stream 为从中读取字符的文件流。
【返回值】该函数执行成功后，将返回所读取的字符。读到文件尾而无数据时便返回 EOF。

```
FILE* fp = fopen("test.txt", "r+");  
while ((ch = fgetc(fp)) != EOF) // 从 fd 中，逐字符读取  
    putc(ch, stdout); printf("%c", ch);  
  
while ((ch = fgetc(stdin)) != 'Q') // 从键盘 stdin 中，逐字符读取  
    putc(ch, stdout); printf("%c", ch);
```

char ch;
while (1)
{
 ch = fgetc(fp);
 if (feof(fp))
 break;
 fputc(ch, stdout); // printf("%c", ch);
}

[3] fgets/fputs gets/puts

```
从stream中，按行最多读取bufsize-1个字符，存到buf中  
char *fgets(char *buf, int bufsize, FILE *stream);  
//一次最多读取bufsize-1个字符：如果用gets(str1,6,file1);去  
读取则执行后str1 = "Love,"，读取了6-1=5个字符  
  
char buf[1024];  
FILE* fp = fopen("test.txt", "r+");  
while (1)  
{  
    fgets(buf, sizeof(buf), fp);  
    fputs(buf, stdout); //printf("%s", buf);  
    if (feof(fp))  
        break;  
}
```

重点：fgets读取完数据后，将会在后面添加\n字符
比如：
fgets(buf, sizeof(buf), stdin); //在键盘输入100后
结果：buf = gjw\n
可用sscanf只取出gjw，保存到num中---> int num;
sscanf(buf, "%d\n", &num);

```
将str, 写入stream中  
int fputs(const char *str, FILE *stream); 将str, 写入stream  
char buf[1024];  
FILE* fp = fopen("src.txt", "r");  
FILE* newfd = fopen("dst.txt", "w+");  
while (1)  
{  
    fgets(buf, sizeof(buf), fp);  
    fputs(buf, newfd); //printf("%s", buf);  
    if (feof(fp))  
        break;  
}
```

```
char * gets ( char * str ); 从标准输入设备  
                           备读字符串函数  
char str[100];  
gets(str);  
cout<<str<<endl;
```

```
int puts(const char *string); 把字符串输出到标准输出设备，  
                               将'\0'转换为回车换行  
char string[] = "This is an example output string\n";  
puts(string);
```

[4] fprintf fscanf 文件的格式化输入/输出

```
格式化输出到文件  
num = rand()%100;  
//num放在%d的位置，"10\n"，然后此字符串写在（显示）在屏幕  
//printf("%d\n", num);  
//num放在%d的位置，"10\n"，然后此字符串写在（显示）在fp所关联的文件  
fprintf(fp, "%d\n", num);
```

格式化读文件，到变量
while(1)
{
 fscanf(fp, "%d\n", &num);
 printf("num = %d\n", num);

 if(feof(fp))
 {
 break;
 }
}

[5] fread/fwrite //按照块读写

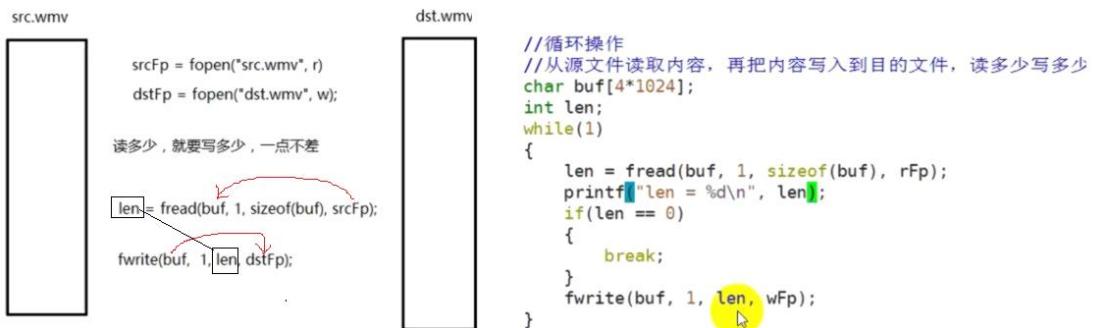
```
#include <stdio.h>  
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream); 以数据块的方式给文件写入内容  
块数据大小      块数  
返回值：  
成功：实际成功写入文件数据的块数目，此值和 nmemb 相等  
失败：0
```

//写文件
Student s[4] = {
 18, "mike", 59,
 22, "jiang", 66,
 33, "lily", 77,
 44, "lucy", 88
};

一次性将s全部写入fp
int ret = fwrite(&s[0], 1, sizeof(s), fp);
printf("ret = %d\n", ret);

//读文件
Student s[10];
int i = 0;
int ret = 0;
while(1)
{
 ret = fread(&s[i], 1, sizeof(Student), fp);
 if(ret == 0) //读取结束
 {
 break;
 }
 i++;
}
printf("i = %d\n", i); //i就是读取的元素个数

代码实战: cp 命令



[6]fseek 文件的随机读写 rewind 移动到开头 ftell 获取光标到开头的字节数

```
int fseek(FILE *stream, long offset, int whence);

SEEK_SET: 从文件开头移动offset个字节
SEEK_CUR: 从当前位置移动offset个字节
SEEK_END: 从文件末尾移动offset个字节

//开头不能往左移动，末尾可以往后移动

fseek(fp, 0, SEEK_SET); 在开头偏移0个字节, 回到开头 rewind(fp)
fseek(fp, 100, SEEK_SET); 在开头向右偏移100个字节, 回到开头

fseek(fp, 0, SEEK_CUR); 在当前位置偏移0个字节
fseek(fp, 100, SEEK_CUR); 在当前位置向右偏移100个字节
fseek(fp, -100, SEEK_CUR); 在当前位置向左偏移100个字节

fseek(fp, 0, SEEK_END); 在结尾位置偏移0个字节, 移动到最后

//光标移动末尾
fseek(fp, 0, SEEK_END);
long size = ftell(fp); 获取文件长度
```

[7]删除文件/重命名文件

#include <stdio.h>	#include <stdio.h>
int remove(const char *pathname);	int rename(const char *oldpath, const char *newpath);
功能: 删除文件	功能: 把 oldpath 的文件名改为 newpath
参数:	参数:
pathname: 文件名	oldpath: 旧文件名
返回值 : 成功: 0	newpath: 新文件名
失败: -1	失败: -1

[8]stat 获取文件状态，存放到第二个参数中

```
int stat(const char *path, struct stat *buf);

struct stat {
    dev_t     st_dev;   //文件的设备编号
    ino_t     st_ino;   //节点
    mode_t    st_mode;  //文件的类型和存取的权限
    nlink_t   st_nlink; //连到该文件的硬连接数目，刚建立的文件值为1
    uid_t     st_uid;   //用户 ID
    gid_t     st_gid;   //组 ID
    dev_t     st_rdev;  //设备类型若此文件为设备文件，则为其设备编号
    off_t     st_size;  //文件字节数(文件大小)
    unsigned long st_blksize; //块大小(文件系统的 I/O 缓冲区大小)
    unsigned long st_blocks; //块数
    time_t    st_atime; //最后一次访问时间
    time_t    st_mtime; //最后一次修改时间
    time_t    st_ctime; //最后一次改变时间(指属性)
};
```

[9]int fflush(FILE* stream) //缓冲区不满，也立即将缓冲区中的内容写到文件中

文件缓冲区（提高读写效率）一例：班长拿东西给班主任

ANSI C 标准采用“缓冲文件系统”处理数据文件。

所谓缓冲文件系统是指系统自动地在内存区为程序中每一个正在使用的文件开辟一个文件缓冲区从内存向磁盘输出数据必须先送到内存中的缓冲区，装满缓冲区后才一起送到磁盘去。

如果从磁盘向计算机读入数据，则一次从磁盘文件将一批数据输入到内存缓冲区（充满缓冲区），然后再从缓冲区逐个地将数据送到程序数据区（给程序变量）。



```
int fflush(FILE *stream);  
功能: 更新缓冲区, 让缓冲区的数据立马写到文件中。  
参数:  
    stream: 文件指针  
返回值:  
    成功: 0  
    失败: -1
```

```
int main(int argc, char *argv[])
{
    FILE *fp = fopen("12.txt", "w");
    fputs("这是最后一天, 好开森", fp);

    //1、默认情况, 程序没有结束, 也没有关闭, 文件缓冲区满, 自动刷新缓冲区
    //fclose(fp);

    //3、文件不关闭, 程序没有结束, 实时刷新, 调用fflush
    fflush(fp);

    //4、程序正常关闭, 缓冲区的内容也会写入文件
    while(1)
    {
        NULL; //目的不让程序结束
    }
    return 0;
}
```

作业：结构体中有指针变量的读写文件

(1)问题核心：结构体中有指针变量*name，如何将结构体写入 fwrite 文件中？

1. 将以下结构体类型变量，以二进制的形式存放到文件中，并且可以实现读的接口，打印读出来信息。

```
typedef struct Student
{
    char *name;      //名字
    int id;
    int name_len;   //名字长度
}Stu;

Stu s; //定义结构体变量
//结构体成员赋值
s.id = 1;
s.name_len = strlen("mike"); //名字长度
s.name = (char *)malloc(s.name_len + 1);
strcpy(s.name, "mike");
```

(2)分析：错误解法

```
typedef struct Student
{
    //如果是指针变量, 写文件时, 只是把指针的值写入文件
    char *name;      //名字
    int id;
    int name_len;   //名字长度
}Stu;

Stu s; //定义结构体变量
//结构体成员赋值
s.id = 1;
s.name_len = strlen("mike"); //名字长度
s.name = (char *)malloc(s.name_len + 1);
strcpy(s.name, "mike");
```

结构体中的指针变量*sname, 用 fwrite 写入文件时, 写入的是 name 的值(即 name 所指向的分配的堆空间的地址 0xaabb)

当程序结束后, 堆空间释放, 因此 name 指向的堆空间释放(失效), 再用 fread 读取到 name 的值后, 调用 printf("%s", s.name) 打印 name 里面存放的值时, 肯定会崩溃(因为此时的 name 中的值是之前释放的堆地址, 它是无效地址)

(3)解决方案

(3-1)写入结构体数据

在写入结构体时:

先将**结构体 s 的数据**写入文件: 其中结构体 s 中变量 name_len 是 name 的长度

还要再将 **s.name 的数据**也写入文件

如下图所示:

```
//1、先将结构体写入文件
s.name_len = strlen("mike"); //名字长度
s.name = (char *)malloc(s.name_len + 1);
strcpy(s.name, "mike");
fwrite(&s, sizeof(s), 1, fp);

//2、再将name所指向的字符串存到文件中
ret = fwrite(s.name, s.name_len, 1, fp);
if (ret < 0)
{
    perror("fwrite name");
    return;
}
```

写完结构体, 单独把name*指向的数组写一遍

结构体s的数据

s. name的数据

(3-2)读取结构体数据

```
//1、先将结构体读出来
fread(&s, sizeof(s), 1, fp);
//2、再读字符串
s.name = (char *)malloc(s.name_len + 1); //开辟空间接受字符串
memset(s.name, 0, s.name_len+1);
fread(s.name, s.name_len, 1, fp); //fp读到s.name中, 读取长度s.name_len
//打印结构体成员
printf("name = %s, id = %d, len = %d\n", s.name, s.id, s.name_len);
```

6.dup***

****执行 printf, 将内容写入文件: dup

```
int main()
{
    printf("before aaaaaaaaaa\n"); //printf函数的内部实现, 往标准输出设备(1)写内容
    //fclose(stdout);
    close(1); //1代表是标准输出设备, 关闭了, 1就是空闲的数字

    int fd = open("01.txt", O_WRONLY, 0777);
    dup(1);

    //printf函数的内部实现, 往(1)写内容, 但是1现在和01.txt关联, printf的内容写到01.txt
    printf("after bbbbbbbb\n");
    printf("after cccccccc\n");
    printf("after dddddddd\n");
}
```

作业: 加密解密文件

=====指针强化=====

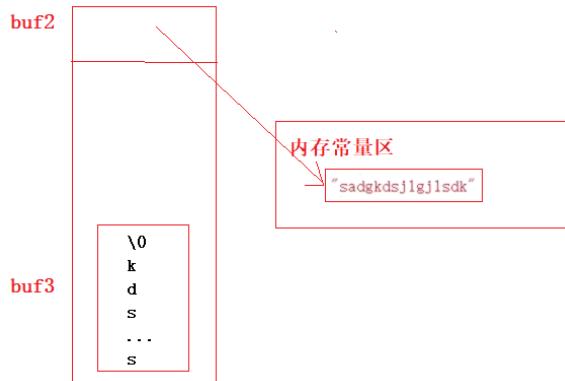
1. void 类型

```
/* void, 无类型
1、函数参数为空, 定义函数时, 可以用void修饰: int fun(void)
2、函数没有返回值: void fun(void);
3、不能定义void类型的普通变量: void a; //err, 无法确定类型, 不同类型分配空间不一样
4、可以定义void *变量: void *p; //ok, 32永远4字节, 64永远8字节
5、数据类型本质: 固定内存块大小别名
6、void *p万能指针, 函数返回值, 函数参数
```

2. 指针可写?

```
//写内存时, 一定要确保内存可写
char *buf2 = "sadgkdsjlgjlsdk"; //文字常量区, 内存不可改
//buf2[2] = '1'; //err

char buf3[] = "sadgkdsjlgjlsdk"
buf3[1] = '1'; //ok
```



3. 数组初始化

//不指定长度, 没有0结束符, 有多少个元素就有多长 char buf1[] = { 'a', 'b', 'c' }; buf中只有abc printf("buf = %s\n", buf); abc烫烫贪燥 鳐	//指定长度, 后面没有赋值的元素, 自动补0 char buf2[100] = { 'a', 'b', 'c' }; printf("buf2 = %s\n", buf2); abc
//所有元素赋值为0 char buf3[100] = { 0 }; char buf5[50] = { '1', 'a', 'b', 0, '7' }; printf("buf5 = %s\n", buf5); 1ab07	//char buf4[2] = { '1', '2', '3' }; //数组越界 char buf6[50] = { '1', 'a', 'b', 0, '7' }; printf("buf6 = %s\n", buf6); 1ab char buf7[50] = { '1', 'a', 'b', '\0', '7' }; printf("buf7 = %s\n", buf7); 1ab

```
//使用字符串初始化, 常用 //strlen: 测字符串长度, 不包含数字0, 字符'\0'  
//sizeof: 测数组长度, 包含数字0, 字符'\0'

char buf8[] = "agjdslgjlsdjq"; 14
printf("strlen = %d, sizeof = %d\n", strlen(buf8), sizeof(buf8));
char buf9[100] = "agjdslgjlsdjq"; 100
printf("strlen = %d, sizeof = %d\n", strlen(buf9), sizeof(buf9));
```

4. const 是一个冒牌货

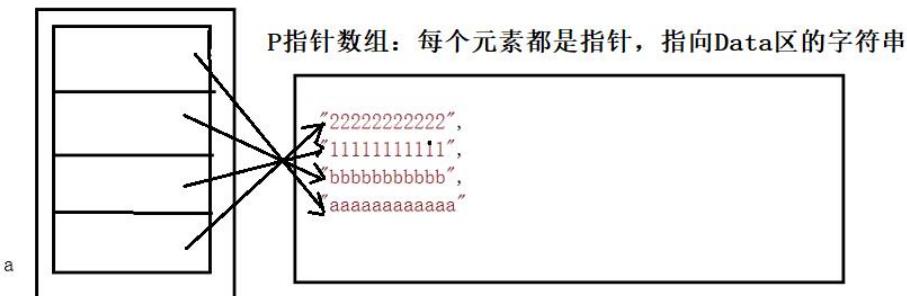
```
const int b = 10;
//b = 100; //err
int *q = &b;
*q = 22;
printf("%d, %d\n", b, *q);
```

3. 指针数组: int* p[10]; → 数组, 每个元素都是指针

```
char a[4][30] = { "222222222222", "111111111111", "bbbbbbbbbbbb", "aaaaaaaaaaaaaa" };
```



```
char *p[] = { "1111111111", "0000000000", "bbbbbbbbbb", "aaaaaaaaaa" };
//char **q = { "1111111111", "0000000000", "bbbbbbbbbb", "aaaaaaaaaa" }; //err ↴
```



4. 数组指针: int (*p)[10]; → 指向某一个数组的首地址, 而不是指向首元素地址

(1) **整个数组首地址/行首地址/行首元素地址

二维数组数组名: 第0行首地址 a+i: 第i行首地址	
1 要想把首地址转化为首元素地址, 加*: *(a+i)	
2 要想得到某个元素地址, 加偏移量: *(a+i)+0, *(a+i)+1, *(a+i)+j → &a[i][0], &a[i]+1, &a[i][j]	
3 要想得到某个元素的值, 是这个元素的地址基础上加*	
((a+i)+j) → *(&a[i][j]) → a[i][j]	

char A[10]; printf("%d,%d,%d\n", A, A + 1, &A + 1); /* A = &A[0] = &(*(A+0)) 首元素地址 A+i = &(*(A+i)) &A = 整个数组的首地址	char B[10][10]; printf("%d,%d,%d,%d\n", B, B + 1, *(B+1)+1, &B + 1); /* B = &B[0] = &(*(B+0)) 第0行的首地址 (相当于一维数组的首地址) B+i = &(*(B+i)) 第i行的首地址 (相当于一维数组的首地址) *(B+i)+j 第i行, 第j列元素的地址 &B = 整个数组的首地址
	int a[10]; int b[5][10]; I int (*p)[10]; p = &a; //为何加 & p = b; //为何不用加 &

char A[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 }; typedef char(*Ptr)[10]; Ptr p; p = &A; //p = A错误 数组指针, 指向一维数组整个数组的首地址 而不是指向一维数组首元素的地址 for (int i = 0; i < 10; i++) { /*p = *A = A; ---> (*p+i) = *(A+i) = A[i]; printf("%d ", *(p + i));	char A[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } }; typedef char(*Ptr)[3]; //数组指针类型 Ptr p; //定义数组指针类型的变量 p = A; //A代表&A[0], 表示第0行首地址 (一维数组的首地址) int n = sizeof(A) / sizeof(A[0]); int nj = sizeof(A[0]) / sizeof(A[0][0]); for (int i = 0; i < n; i++) for (int j = 0; j < nj; j++) { printf("%d ", *(p + i) + j); }
---	---

(2) 数组指针变量的定义 (3 种形式)

```
//方式1  
typedef int A[10]; //声明数组类型A  
A* a; //定义  
  
//方式2  
typedef int (*Ptr)[10]; //声明数组指针类型Ptr  
Ptr ptr; //定义  
  
//方式2  
int (*P)[10]; //直接定义数组指针变量P
```

(3) 二维数组的三种参数模型

```
int A[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };  
  
void print01(int A[2][3]); /*5K*/  
void print03(int A[][3]); /*7K*/  
void print02(int (*A)[3]); /*9K*/ void print12(int *A[3]); //error *  
  
void print11(int **A); // error
```

(4) 指针数组/二维数组 <=====> 字符串

```
char* keyWord[] = { "AAA", "BBB", "CCC" };  
  
keyWord[2] → AAA  
keyWord[1] → BBB  
keyWord[0] → CCC
```

```
char* keyWord[] = { "AAA", "BBB", "CCC" };  
void print00(char** keyWord)  
void print00(char* keyWord[])  
{  
    for (int i = 0; i < 3; i++)  
        printf("%s ", keyWord[i]);  
}
```

```
char keyWord1[3][100] = { "AAA", "BBB", "CCC" };  
  
keyWord1[2] → AAA  
keyWord1[1] → BBB  
keyWord1[0] → CCC
```

```
void print01(char (*keyWord)[100])  
//void print01(char(*keyWord)[]) error, 必须指定步长  
{  
    for (int i = 0; i < 3; i++)  
        printf("%s ", keyWord[i]);  
}
```

下面例题：

使用到 `char* buf[]` 和 `char buf[M][N]` 保存字符串

```

//两个递增有序的数组: char* buf1[], char(*buf2)[30] ==> 合并成一个char **p3
int getMergeStr(char* buf1[], int len1, char(*buf2)[30], int len2, char*** dst, int *len3)
{
    if (buf1 == NULL || buf2 == NULL || len1 == 0 || len2 == 0)
        return -1;

    *len3 = len1 + len2;
    *dst = (char**)malloc(sizeof(char)*(len1 + len2));

    int i = 0, j = 0;
    int k = 0;
    while (i < len1 && j < len2)
    {
        if (strcmp(buf1[i], buf2[j]) < 0)
        {
            (*dst)[k] = (char*)malloc(sizeof(buf1[i]));
            strcpy(((*dst)[k++]), buf1[i++]);
        }
        else
        {
            (*dst)[k] = (char*)malloc(sizeof(buf2[j]));
            strcpy(((*dst)[k++]), buf2[j++]);
        }
    }

    while (i < len1)
    {
        (*dst)[k] = (char*)malloc(sizeof(buf1[i]));
        strcpy(((*dst)[k++]), buf1[i++]);
    }

    while (j < len2)
    {
        (*dst)[k] = (char*)malloc(sizeof(buf2[j]));
        strcpy(((*dst)[k++]), buf2[j++]);
    }

    return 0;
}

void print(char** p3, int len3)
{
    for (int i = 0; i < len3; i++)
        printf("%s ", p3[i]);
}
}

int free_buf(char*** p3, int len3)
{
    char** tmp = *p3;
    if (tmp == NULL)
        return -1;
    for (int i = 0; i < len3; i++)
    {
        if (tmp[i] != NULL)
        {
            free(tmp[i]);
            tmp[i] = NULL;
        }
    }
    free(tmp);
    tmp = NULL;
}

int main()
{
    char* buf1[] = { "11", "33", "55" };
    char buf2[][30] = { "22", "44", "66" };
    int len1 = sizeof(buf1) / sizeof(buf1[0]);
    int len2 = sizeof(buf2) / sizeof(buf2[0]);

    char **p3 = NULL;
    int len3 = 0;

    int res = getMergeStr(buf1, len1, buf2, len2,
                          &p3, &len3);
    if (res == -1)
        return -1;
    print(p3, len3);

    free_buf(&p3, len3);
    system("pause");
}

```

(5) sizeof / 指针数组 / 二维数组

```

char* p1[] = { "A", "AAAAAA", "A", "A" };
printf("%d,%d\n", sizeof(p1), sizeof(p1[0])); //16, 4
//p1是数组, 每个元素都是char*的指针, 占4个字节, 共4个sizeof(p1)=4*4
//p1[i]是char*, 占4个字节

char* p2[10] = { "A", "AAAAAA", "A", "A" };
printf("%d,%d\n", sizeof(p2), sizeof(p2[0])); //40, 4
//p2是数组, 每个元素都是char*的指针, 占4个字节, 共10个sizeof(p1)=4*10
//p2[i]是char*, 占4个字节

char p3[][10] = { "A", "AAAAAA", "A", "A" };
printf("%d,%d\n", sizeof(p3), sizeof(p3[0])); //40, 10
//sizeof(p3) = sizeof(char)*4*10
//p3[i]是char类型的一维数组, 数组长度是10, sizeof(p3[i]) = sizeof(char)*10

char p4[10][10] = { "A", "AAAAAA", "A", "A" };
printf("%d,%d\n", sizeof(p4), sizeof(p4[0])); //100, 10
//sizeof(p4) = sizeof(char)*10*10
//p4[i]是char类型的一维数组, 数组长度是10, sizeof(p4[i])=sizeof(char)*10

```

5. 函数指针 ==> 回调函数

```
typedef void(*pFunType)(int, int); // 定义函数指针类型

void add(int a, int b)
{
    printf("a + b = %d\n", a + b);
}

void sub(int a, int b)
{
    printf("a - b = %d\n", a - b);
}

void execute(int a, int b, pFunType p) // 函数指针变量作形参
{
    p(a, b); // 回调函数
}

=====递归=====
```

实现字符串反转

```
void reserve(char *str, int n)
{
    int i = 0, j = n-1;
    while (i < j)
    {
        char t = str[i];
        str[i] = str[j];
        str[j] = t;
        i++;
        j--;
    }
}

int main()
{
    char str[] = "ABCD"; // char* str = "ABCD"; --- error
    int len = strlen(str);
    reserve(str, len);
    printf("%s\n", str);
    system("pause");
}
```

=====动态库封装与使用=====

1. 创建DLL工程



2. 写代码

```
test.h* ② x
全局范围
1 #include<stdio.h>
2
3 int add(int a, int b);

test.c* ② x
全局范围
1 #declspec(dllexport) int add(int a, int b)
2 {
3     return a + b;
4 }
```

3. 编译后，生成dll和lib



4. 使用编译好的动态库

- 需要三个文件：dll, lib, 头文件.h
(1) .h放到测试项目中
(2) lib添加到连接器的附加依赖项中
(3) dll放到测试项目生成的exe中

=====日志文件=====

=====内存泄漏检测=====