P 21 初识常见关键字

extern 是声明外部符号的
eg：声明全局变量
extern int g_val;
eg：声明外部函数
extern int Add(int x, int y);

P23 初识常见关键字

typedef 类型重定义
eg：typedef unsigned int u

static 静态的

1.修饰局部变量
改变了局部变量的生命周期，本质上是改变了变量的存储类型，变为静态变量
在修饰变量的时候，static 修饰的静态局部变量只执行初始化一次，而且延长了局部变量的生命周期，直到程序运行结束以后才释放

2.修饰全局变量
使得全局变量只能在自己的源文件中使用，声明无效。

全局变量在其他源文件中可以使用，是因为全局变量具有外部链接属性，而被 static 修饰后，就变成了内部链接属性，其他源文件就不能使用。

3.修饰函数
与修饰全局变量一致

P24 常量和宏

define 预处理指令，不是关键字
（include 也是预处理指令）
1.定义常量
#define max 10
2.定义宏
宏是替换
#define ADD（X，Y） X+Y
4*ADD（2，3）4*2+3

#define ADD（X，Y） （（X）+（Y））X，Y 可能是表达式
4*ADD（2，3）4*（2+3）

P26 初识指针

一个内存单元有 1byte

%p 是专门打印地址的
eg：printf（“%p\n”，&a）；
int* pa=&a;

pa 是用来存放地址的，叫做指针变量
int 是说明 pa 指向的对象是 int 类型
eg：char ch=‘w’；
char* pc=&ch;

*pa=20;
*为解引用操作符，找到 pa 里面地址所指的位置

P27 初识结构体

创建结构体
struct stu
{
char name【20】;
int age；
double score；
};

结构体初始化
struct stu s={“张三”，20，80};

结构体指针
struct stu * ps=&s;

访问结构体
s.name　(*ps).name　　ps->name

%lf 是打印双精度浮点型数 double

P30 分支语句 if-else

1.if（）{}
2.if（）{}else{}
3.if(){}else if{}…

P31 分支语句 switch

switch (day)
{
case 1:
printf(“星期一\n”);
break;
……
default：
printf（“输入错误\n”）;
break；
}
字符算整型，switch 里面 day 和 case 里面必须是整型，case 和 default 无顺序

P32 循环语句 while 1

while（判断表达式）
循环语句
break 永久跳出循环
continue 跳出本次循环

EOF　end of file 文件结束标志
getchar（）读取一个字符，遇到结束或错误返回 EOF
ctrl+z 就读取结束，返回 EOF
键盘读取一个字符，中间有缓存区
getchar 的返回类型是整型，EOF 为-1
eg：int ch=getchar（）

putchar（）输出一个字符

P33 循环语句 while 2

数组的数组名本来就是地址，不需要取地址
scanf 和 getchar 等输入语句，是需要经过缓冲区才从键盘获取的，读取不了空格
主要出现在两次连续输入的情况下
解决方法：清除缓存区
1.getchar()清除一个字符
2.while（（tmp=getchar（））！=‘\n‘）
{
;
}清除多个字符

P34 循环语句 for

continue 跳出本次循环，跳到调整部分

注意：循环体内尽量不要改变循环变量
建议 for 语句的循环控制变量的取值采用前闭后开区间的写法
eg：for（i=0；i<10；i++）

判断部分省略，即判断为真，循环变为死循环
初始化部分省略，在多次循环的时候会出错，只会执行一次

P38 循环语句 do while

do
循环体
while（）

break 情况如上
continue 情况和 while 循环一样

char arr【】=“welcome to bit！！！”；
int right=strlen（arr）-1；计算字符串数组最后一个元素的位置
strlen 用来计算字符串的长度，需要引用头文件#include <string.h>

延时 Sleep（1000）延时 1000 毫秒，需要引用头文件#include <windows.h>

system（“cls”）；清空屏幕

P40 循环语句 do while

strcmp 字符串比较函数，需要引用头文件#include <string.h>
eg：strcmp（password，“123456”）==0；
如果两个字符串相等就返回 0
如果第一个字符串小于第二个字符串，返回一个小于 0 的数
如果第一个字符串大于第二个字符串，返回一个大于 0 的数

生成一个 1～100 的随机数

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
void game()
{
int ret=rand()%100+1;
printf(“%d\n”,ret);
}
int main()
{
srand((unsigned int)time(NULL));强制转换 time 的返回值的格式
game();
}
```

rand()生成一个 0～32767 之间的一个随机数，但需要在调用前使用 srand（）生成一个初始点，其中参数的格式为 unsigned int，使用时需要引用头文件#include<stdlib.h>
time 是引用一个时间戳，其返回值是一个整型，使用时需要引用头文件#include<time.h>

P41 分支循环作业讲解

F10～逐过程执行，会跳过其他函数
F11～逐语句执行，其他函数内部的语句也会逐一执行

P43 分支循环作业讲解

sqrt（）开方，需要引用头文件#include<math.h>
system()系统命令，需要引用头文件#include<stdlib.h>
system(“shutdown -s -t 60”);
60 秒后电脑关机
system(“shutdown -a”);取消关机

again：…

...
goto again；
goto 跳到指定位置

P45 函数讲解

库函数的学习网站：www.cplusplus.com
http://en.cppreference.com
strcpy(arr1,arr2);把 arr2 中的内容复制到 arr1，两个参数为指针格式，\0 也可以复制过去

memset 内存设置
eg：char arr[]=“hello bit”；
memset(arr,’x’,5);arr 中就会变成 xxxxx bit

P46 函数

函数调用时所传递的参数叫实际参数，简称实参，实参可以是：常量，变量，表达式，函数等，但在函数进行调用时，它们都必须有确定的值

函数定义时的参数叫做形式参数，简称形参，形参只有在被调用的时候会分配空间，当函数调用完毕之后，形参就自动销毁了，因此形参只在函数中有效

P48 函数的调用

传值调用
函数的形参和实参分别占有不同的内存块，对形参修改不会影响实参

传址调用
让形参和实参真正的联系起来

函数传递参数时，实参传递的数组只是数组首地址，形参接收的类型实际上是指针形式，所以函数内部求不了数组的元素个数

P50 函数的嵌套调用和链式访问

函数可以嵌套调用，不能嵌套定义
链式访问，把一个函数的返回值作为另外一个函数的参数

printf 的返回值是打印在屏幕上的字节数
eg：printf(“%d”,printf(“%d”,printf(“%d”,43)));
打印结果为 4321

函数的声明
eg：int Add（int x, int y); x 和 y 可以省略
函数的声明只需要告诉函数的返回类型，函数名，函数的参数类型就好
函数的声明必须在函数调用之前
函数的声明一般放在头文件中，函数的定义一般放在同名的.c 文件中
在用到所定义的函数时，要引用相应的头文件
eg：#include “sub.h”

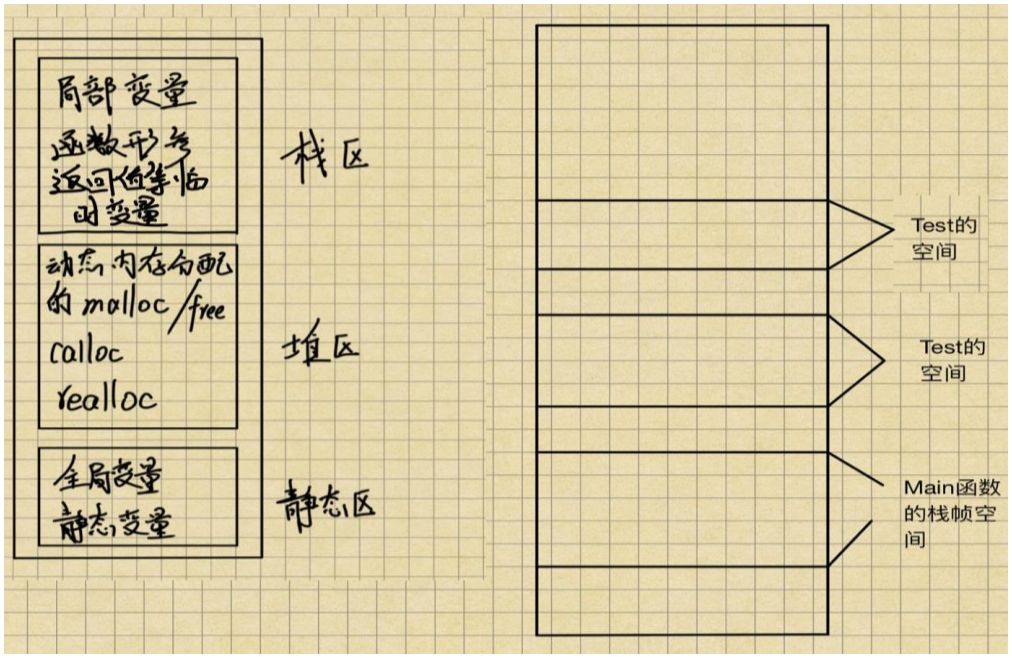
代码的隐藏，将文件的属性中 exe 改为 lib 静态库，然后只需要相应的头文件和 lib 文件就可以使用该函数，使用时应先导入静态库
#pragma comment(lib,”sub.lib”)

P51 函数递归

自己调用自己，把大事化小的思想
eg：输入 1234，打印 1 2 3 4

递归的两个必要条件：
1.存在限制条件，当满足这个条件的时候，递归便不再继续
2.每次递归调用后越来越接近这个限制条件

Stack overflow 栈溢出



栈溢出是因为不断调用函数所造成的
写递归的时候递归次数不能太多，不然容易栈溢出

P53 函数递归

递归的时候尽量不用 a++，容易出错
写递归的时候可能出现栈溢出，效率低的问题，有时候可以用非递归的方式来写，例如迭代

P55 函数递归

函数可以没有参数，例如 main 函数
函数设计时应该尽量少的定义全局变量
主函数的位置可以在任意位置

P56 函数作业讲解

函数的处理结果的两个数据返回主函数，可以通过
1.形参用数组
2.形参用两个指针
3.用两个全局变量

逗号表达式
eg：（a，b）会依次从左到右执行，此条指令的结果是 b

P60 一维数组

c99 才支持变长数组，即数组长度为变量，一般都是常值表达式

数组初始化
完全初始化，把数组中所有的元素全部初始化
不完全初始化，只初始化数组中的一部分元素，其余元素全部为 0，如果是字符数组，则全部是/0

数组未指定长度的话，就根据数组初始化的长度定义
eg：char ch5[]=“bit”;里面有四个元素，b，i，t，/0
字符串长度为 4
char ch6[]={‘b’,’i’,’t’};里面有三个元素，b，i，t
字符串长度为随机值，但数组长度还是 3
打印出来是

bit
bit 汤燕的见 itit

%p 是专门打印地址的,按地址的格式打印——16 进制
eg:printf(“%p\n”,&arr[0]);
printf(“%x\n”,0x12);
printf(“%p\n”,0x12);
结果：
12
0000 0012 十六进制

1.一维数组在内存中是连续存放的
2.随着数组下标的增长，地址是由低到高变化的！

打印类型
%o，以八进制无符号形式输出整数（不输出前导符 0）
%x 和%X，以十六进制无符号形式输出整数（不输出前导符 0），大写打印大写，小写打印小写
%u，以十进制无符号形式输出整数
%m.nf，指定数据宽度和小数位数
%-m.nf，数组长度不超过 m 时，数据向左靠，右端补空格

P61 二维数组

创建二维数组
int arr[3][4];
char ch[3][10];
初始化—创建的时候同时赋值
int arr[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};完全初始化
int arr[3][4]={1,2,3,4,5,6};不完全初始化
int arr[3][4]={{1,2},{5,6},{3,4}};可看成 3 个一维数组
int arr[][4]={{1,2},{5,6},{3,4}};二维数组的行可以省略，列不行
二维数组在内存中是连续存放的
int arr[3][4]={{1,2},{5,6},{3,4}};
二维数组名：arr
二维数组第一行一维数组名：arr[0]

P62 数组作为函数参数

数组名是表示数组首元素地址
但有 2 个意外
1.sizeof（数组名）—数组名表示整个数组—计算的是整个数组的大小，单位是字节

2.&数组名—数组名表示整个数组—取出的是整个数组的地址，与数组首地址一样，但意义不一样
&arr+1 相对于&arr 是加了一个 arr 数组长度的地址
arr+1 相对于 arr 是只加了一个元素的长度

P72 操作符详解

取模操作符%两端都为整数

移位操作符移动的都是内存中存的数，即反码
右移操作符>>
1.算术右移
丢弃右边，左边补原符号
2.逻辑右移
丢弃右边，左边补 0
一般都是算术右移

位操作符
按位与&　按位或｜ 按位异或^（相同为 0，不同为 1）

对于正整数来说，原码补码反码是相同的
对于负数来说，
原码：数值的二进制
反码：原码符号位不变，其他位按位取反
补码：反码加一

异或交换数
a=3,b=5
a=a^b　　0^a=a
b=a^b　　a^a=0
a=a^b

P74 操作符详解

连续赋值
a=x=y+1　从右往左执行
sizeof 是单目操作符，因此可以省略变量两侧的括号，但不能省略类型两侧的括号
sizeof a

eg：
short s=5;
int a=10;
printf(“%d”,sizeof(s=a+2));　2 此处的 s 运算是在编译器里运行，所以 s 还是 5

printf(“%d”,s);　　5

sizeof 括号中的运算是不参与运算的

～按位取反，包含符号位

P76 操作符详解

int i=0,a=1,b=2,c=3,d=4;
i=a++&&++b&&d++;a 为 0，逻辑与后面不管是什么 i 都是 0，所以++b 和 d++不进行计算
i=a++||++b||d++;

逗号表达式，就是用逗号隔开多个表达式，从左依次向右进行计算。整个表达式的结果是最后一个表达式的结果。

P77 操作符详解

.操作符　　结构体.成员名
->操作符　结构体指针->成员名

表达式求值
表达式求值的顺序一部分是由操作符的优先级和结合性决定。
同样，有些表达式的操作数在求值的过程中可能需要转换为其他类型。
隐式类型转换
c 的整型算术运算总是以缺省整型类型的精度来进行的。
为了获得这个精度，表达式中的字符和短整型操作数在使用前被转换为普通整型，这种转换称为整型提升。
整型提升
整型提升只有不足四个字节时，即 char 类型（1 字节）和 short（2 字节）类型，运算时才会有整型提升。

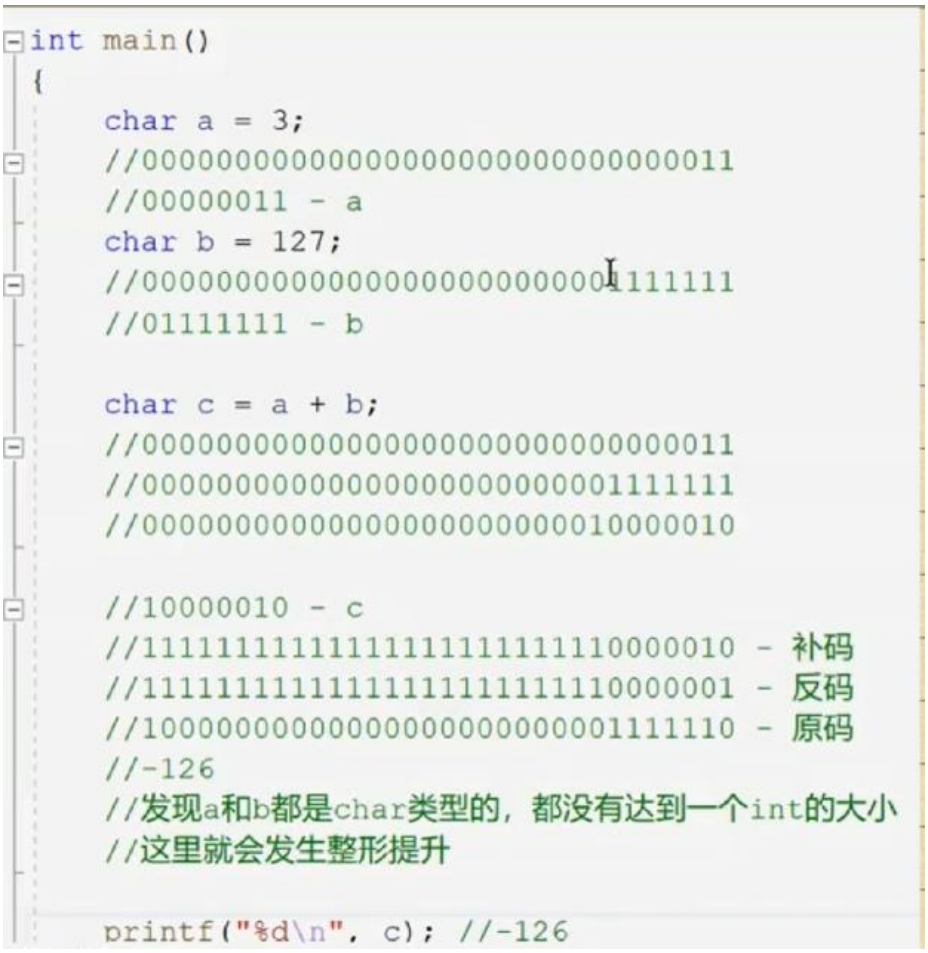
整型提升的意义：

> 表达式的整型运算要在CPU的相应运算器件内执行，CPU内整型运算器(ALU)的操作数的字节长度一般就是int的字节长度，同时也是CPU的通用寄存器的长度。
>
> 因此，即使两个char类型的相加，在CPU执行时实际上也要先转换为CPU内整型操作数的标准长度。
>
> 通用CPU（general-purpose CPU）是难以直接实现两个8比特字节直接相加运算（虽然机器指令中可能有这种字节相加指令）。所以，表达式中各种长度可能小于int长度的整型值，都必须先转换为int或unsigned int，然后才能送入CPU去执行运算。

eg：
char a,b,c;
a=b+c;

b 和 c 的值被提升为普通整型，然后再执行加法运算
加法运算结束后，结果被截断，然后存储在 a 中

%u 打印一个无符号数



P79 操作符详解

sizeof 返回的是无符号整数，用%u 输出

算术转换
如果某个操作符的各个操作数属于不同的类型，那么除非其中一个操作数转换为另外一个操作数的类型，否则无法进行运算。下面的层次体系称为寻常算术转换。
long double
double
float
unsigned long int
long int
unsigned int
int

sizeof 计算的是存储空间的大小，包括\0，计算变量/类型所占内存大小，单位是字节

strlen 计算的是字符串长度，不包括\0，即找到\0 之前出现了几个字符，单位是字节

P82 指针初阶

指针类型的意义
1.指针类型决定了，指针解引用的权限有多大
2.指针类型决定了，指针走一步，能走多远（步长）

P83 指针初阶

野指针的产生
1.指针未初始化
2.指针越界访问
3.指针指向的空间释放

如何规避野指针
1.指针初始化
当不知道指针具体地址时，可以将指针初始化为 0 或者 NULL（void *)
2.小心指针越界
3.指针指向空间释放及时置 NULL
当指针所指的空间已经还给系统时，将指针置 NULL
4.指针使用之前检查有效性
指针置为 NULL 时也不是有效指针，所以使用前必须检查其有效性

```
if (p！=NULL)
*p=10;
```

指针运算
指针+-整数
指针比较大小
指针+-指针
指针和指针相减的前提是，两个指针指向同一个空间，两个指针相减得到的是两个指针所指的两个空间中间的元素个数

P85 指针和数组

指针的比较
标准规定：允许指向数组元素的指针与指向数组最后一个元素后面的那个内存位置指针比较，但是不允许与指向第一个元素之前的那个内存位置的指针比较
向前不行，向后可以

```
int* p=arr;
```

arr[2]<==>*(arr+2)<==>*(p+2)<==>*(2+p)<==>*(2+arr)<==>2[arr]
arr[2]==p[2]

二级指针
int a=10;
int* pa=&a;
int* *ppa=&pa;
*pa==a *ppa==pa ——>**ppa==a

指针数组
int* parr[5];整型指针数组，存放整型指针的数组

P87 结构体初阶

s1 和 s2 是全局变量

```
struct B
{
    char c;
    short s;
    double d;
};
struct Stu
{
    //成员变量
    struct B sb;
    char name[20];//名字
    int age;//年龄
    char id[20];
} s1, s2;//s1和s2也是结构体变量
int main()
{
    //s是局部变量
    struct Stu s = { {'w', 20, 3.14}, "张三", 30, "202005034" };//对象

    return 0;
```

结构体成员访问
eg：s.name
struct Stu* pa=&s;
(*pa).name pa->name

结构体传参
如果传递的结构体参数为 s，则接收的为 struct Stu t，传值调用
如果传递的结构体参数为&s,则接收的为 struct Stu* t, 传址调用
从空间和时间上来讲，传址调用较好，节省时间和空间，效率高，而且可以改变原来的结构体

函数传参时通常是从右向左传递
函数传参时，参数是需要压栈的，如果传递一个结构体对象时，结构体过大，参数压栈的系统开销比较大，所以会导致性能的下降。
所以结构体传参时，优先选择传址调用

P93 模拟实现字符串相关函数

assert 断言
eg：assert（src！=NULL）
假会提醒，真则无事发生
需要引用头文件 include<assert.h>

const 修饰变量，这个变量就被称为常变量，不能被修改，但本质上还是变量
const 修饰指针变量时，const 如果放在*左边，修饰的是*p，表示指针指向的内容是不能通过指针来改变的，但 p 是可以改变的
const 修饰指针变量时，const 如果放在*右边，修饰的是指针变量 p，表示指针变量不可以改变，但指针所指的内容可以被改变

P98 c 语言初阶考试

scanf 函数输入的时候以空格结尾，即输入 abs ghj，只会接受到 abs
想要输入空格，用 gets 函数，即 gets（arr）

P99 数据的存储

大端字节序
把数据的低位字节序的内容存放在高地址处，高位字节的内容存放在低地址处
小端字节序
把数据的高位字节序的内容存放在高地址处，低位字节的内容存放在低地址处

P101 数据的存储

```
char a = -1;
//10000000000000000000000000000001
//11111111111111111111111111111110
//11111111111111111111111111111111
//11111111
//11111111111111111111111111111111
signed char b = -1;
//11111111

unsigned char c = -1;
//11111111
//00000000000000000000000011111111
//
printf("a=%d,b=%d,c=%d", a, b, c);
//-1 -1 255
```

截断与整型提升

注意

1.char 类型是 signed char 还是 unsigned char 类型，c 语言没有明确的规定，取决于编译器，一般大多数都是 signed char

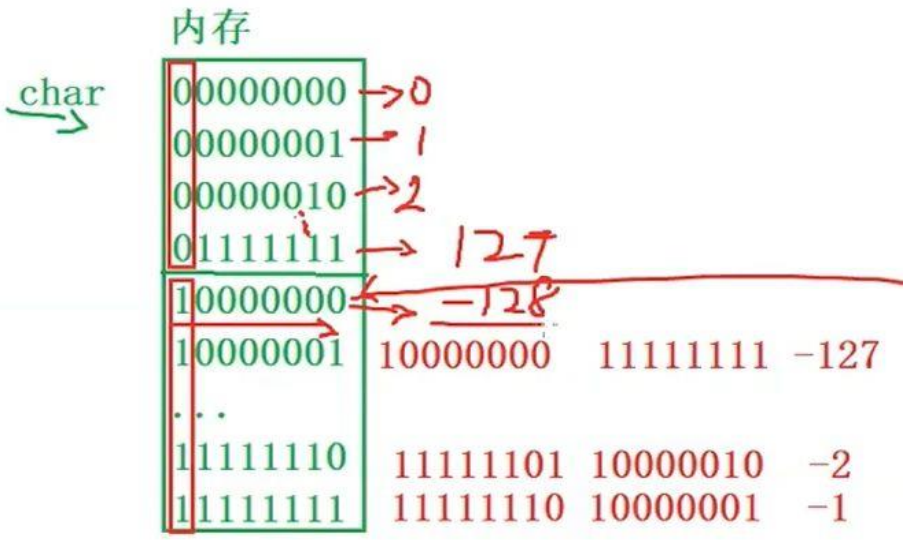
而 int 和 short 是有明确规定的，都是有符号的

char 类型的取值范围是-128～127

127+1=-128

-1+1=0

-128-1=127



%d 打印的是有符号的整型数

P103 数据的存储

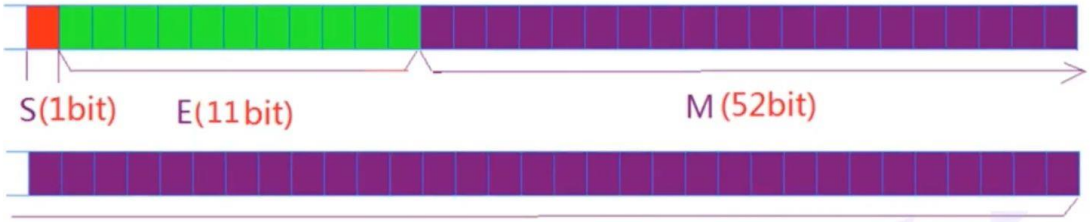
strlen 函数，求字符串的长度，包含多少个字符，遇到 0 时截止，0 不计算在内

P104 浮点型的存储

浮点型的存储与整型的存储方式不相同
对于 float 形式，占 4 个字节，32 位 bit 位

单精度浮点数存储模型

对于 double 形式，占 8 个字节，64 位 bit 位



双精度浮点数存储模型

5.5 的 float 存储
转换为二进制 101.1,用科学计数法表示 1.011*2^2
S=0 表示正负
E 为无符号数，表示幂，即 2，表示为 2+127,即 10000001
M 为 011
所以 5.5 的存储为 01000000100000000000000000000011

读取的时候需要注意 E 有三种情况

E不全为0或不全为1

> 这时，浮点数就采用下面的规则表示，即指数E的计算值减去127（或1023），得到真实值，再将有效数字M前加上第一位的1。比如：0.5（1/2）的二进制形式为0.1，由于规定正数部分必须为1，即将小数点右移1位，则为1.0*2^(-1)，其阶码为-1+127=126，表示为01111110，而尾数1.0去掉整数部分为0，补齐0到23位00000000000000000000000，则其二进制表示形式为:

E全为0

> 这时，浮点数的指数E等于1-127（或者1-1023）即为真实值，有效数字M不再加上第一位的1，而是还原为0.xxxxxx的小数。这样做是为了表示±0，以及接近于0的很小的数字。

E全为1

> 这时，如果有效数字M全为0，表示±无穷大（正负取决于符号位s）；

例子：

```
//浮点型的存储
int main()
{
    int n = 9;//以整型的方式存入9
    float* pFloat = (float*)&n;//强制转换为float形式
    printf("%d\n", n);//以整型的方式输出，为9
    printf("%f\n", *pFloat);//以浮点型的方式输出，为0.000000

    *pFloat = 9.0;//以浮点型的方式存入9.0
    printf("%d\n", n);//以整型的方式输出，为1091567616
    printf("%f\n", *pFloat);//以浮点型的方式输出，为9.000000
    return 0;
}
```

P106 指针进阶

字符指针也可以指向一个字符串
eg：char* ps=“hello bit”；此处的字符串为常量字符串，不能修改
本质上是把字符串的首字符的地址存储到了 ps 中
打印的时候可以全部打印出来
eg：printf（“%s”，ps）；可以打印出整个字符串

指针数组
用来存储指针变量的数组

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

int* arr[3] = {a,b,c};
int i = 0;
for (i = 0; i < 3; i++)
{
    int j = 0;
    for (j = 0; j < 5; j++)
    {
        //printf("%d ", *(arr[i] + j));
        printf("%d ", arr[i][j]);
    }
    printf("\n");
}
```

访问方式与二维数组可相同
但只是模拟出二维数组
与二维数组不同

```
int* arr1[10]; //整形指针的数组
char *arr2[4]; //一级字符指针的数组
char **arr3[5];//二级字符指针的数组
```

P108 指针进阶

数组指针—指向的是一个数组
eg：

```
int arr[10] = { 1,2,3,4,5 };
int* (*parr)[10] = &arr;//arr是数组的首元素的地址，&arr是数组的地址
//parr是一个数组指针，里面存放的是数组的地址
double* d[5];//创建一个指针数组
double* (*pa)[5] = &d;//pa是一个数组指针，里面存放指针数组d的地址
//double*是表示数组里面内容的格式，*pa是表示pa是一个指针，[5]表示所指数组有5个元素
```

arr 数组的元素类型为 int
arr 数组的类型为 int[10]

arr 和&arr 的区别
两者的值相同，但意义不一样
arr 是数组首元素的地址，加一之后是第二个元素的地址
&arr 是数组的地址，加一之后是跳过了这个数组

数组名是首元素的地址
但是有两个例外
1.sizeof（数组名）—数组名表示整个数组，计算的是整个数组的大小，单位是字节
2.&数组名—数组名表示整个数组，取出的是整个数组的地址

数组指针的应用
1.访问一维数组
不常用

```c
//数组指针访问一维数组
int main()
{
    int arr[10] = { 0,1,2,3,4,5,6,7,8,9 };
    int(*parr)[10] = &arr;
    int i = 0;
    for (i = 0; i < 10; i++)
    {
        printf("%d ", *((*parr) + i));//*parr相当于数组名，也可写作*(*parr + i)
    }
    return 0;
}
```

2.访问二维数组

二维数组的首元素为第一行

```c
//p是一个数组指针
void print2(int(*p)[5], int r, int c)
{
    int i = 0;
    int j = 0;
    for (i = 0; i < r; i++)
    {
        for (j = 0; j < c; j++)
        {
            printf("%d ", *(*(p + i) + j));
        }
        printf("\n");
    }
}

int main()
{
    //int a[5];  &a
    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);//arr数组名，表示数组首元素的地址
    return 0;
}
```

int arr[5]

表示一个数组，可以存放 5 个元素，每个元素为 int 类型

int *parr1[10]

表示一个指针数组，可以存放 10 个指针，每个指针指向的内容为 int 类型

int (*parr2)[10]

表示一个数组指针，指向一个数组，数组可以存放 10 个元素，每个元素为 int 类型

int (*parr3[10])[5]

表示一个存放数组指针的数组，数组可以存放 10 个数组指针，每个指针都指向一个数组，每个数组包含 5 个元素，每个元素为 int 类型

一维数组传参

int arr1[10]={0};

```
int* arr2[10]={0};
test1(arr1);
test2(arr2);

1. void test1(int arr1[10])
2. void test1(int arr1[ ])
3. void test1(int* arr)
4. void test2(int* arr[10])
5. void test2(int**arr)
```

二维数组传参

```
int arr [3][5]={0};
int arr [ ] [5]={0};   //行可以省略，列不可以省略
test(arr);

1. void test(int arr[3][5])
2. void test(int arr[ ][5])
3. void test(int (*arr)[5])
```

P109 指针初阶等作业讲解

全局变量不初始化时，默认为 0
sizeof 操作符的结果为无符号的整型数
pow（a，b）是求 a 的 b 次方，需要引用头文件 math.h
P111 调试技巧作业讲解
c 程序中常见的错误有编译错误，链接错误，运行时错误，栈溢出属于运行时错误。
F5 是开始调试
ctrl+F5 是开始执行不调试

P115 指针进阶

一级指针传参，一级指针接收
二级指针传参，二级指针接收
传一级指针的地址，也可以传递存放一级指针的数组

函数指针：指向函数的指针 ，存放函数地址的指针
注意：
数组名！=&数组名（数值相同，意义不同）
函数名==&函数名

```c
//函数指针
int ADD(int x, int y)
{
    return x + y;
}
int main()
{
    int (*pf)(int, int) = &ADD;//pf是一个函数指针
    //int (*pf)(int, int) = ADD;//&ADD==ADD==pf
    printf("%p\n", ADD);
    printf("%p\n", &ADD);//两者意义数字都相同
    int ret = (*pf)(3, 5);//函数指针调用函数
    int ret = pf(3, 5);//函数指针调用函数,*是为了理解，没有实际的意义
    int ret = ADD(3, 5);//函数指针调用函数
    printf("%d\n", ret);
    return 0;
}
```

P117 指针进阶

```c
//代码解读
int main()
{
    //代码1
    (*(void (*)())0)();
    //调用0地址处的函数
    //该函数无参数，返回类型为void
    //1.void(*)()--函数指针类型
    //  void(*p)()--函数指针变量
    //2.(void(*)())0--对0进行强制类型转化,转化为函数指针类型
    //3.*(void(*)())0--对0地址进行了解引用操作
    //4..(*(void(*)())0)()--调用0地址处的函数
    return 0;
}
//代码2
void (*signal(int, void(*)(int)))(int);//函数声明
//等价于   void(*)(int)  signal(int,void(*)(int))
//函数signal的函数参数类型分别是int类型和void(*)(int)函数指针类型，返回类型是void(*)(int)
//但语法要求不能这样写，所以将函数放*一起
//
//对代码2进行简化，如下
//typedef void(*pfun_t)(int);--重新定义void(*)(int)函数指针类型为pfun_t
//pfun_t signal(int,void(*)(int));
```

函数指针数组—存放函数指针的数组

```c
//函数指针数组
int Add(int x, int y)
{
    return x + y;
}
int Sub(int x, int y)
{
    return x - y;
}
int main()
{
    int(*pf1)(int, int) = Add;//定义函数指针变量pf1
    int(*pf2)(int, int) = Sub;//定义函数指针变量pf2
    int(*pfarr[2])(int, int) = { Add,Sub };//pfarr就是函数指针数组
    return 0;
}
```

P119 指针进阶

函数指针的应用

```c
//2
int add(int x, int y)
{
    return x + y;
}
int sub(int x, int y)
{
    return x - y;
}
int mul(int x, int y)
{
    return x * y;
}
int div(int x, int y)
{
    return x / y;
}
void meau()
{
    printf("*****************************************\n");
    printf("**********   1.add     2.sub   **********\n");
    printf("**********   3.mul     4.div   **********\n");
    printf("**********      0.退出系统      **********\n");
    printf("*****************************************\n");
}
int main()
{
    int input = 0;
    int x = 0;
    int y = 0;
    int ret = 0;
    //转移表
    int(*p[5])(int, int) = { 0,add,sub,mul,div };
    do
    {
        meau();
        printf("请选择：");
        scanf("%d", &input);
        if (input >= 1 && input < 5)
        {
            printf("请输入两个操作数:");
            scanf("%d %d", &x, &y);
            ret = (*p[input])(x, y);
            printf("ret=%d\n", ret);
        }
        else if (input == 0)
        {
            printf("退出系统\n");
            break;
        }
        else
        {
            printf("输入错误，请重新输入\n");
            break;
        }
        printf("\n\n");
    } while (input);
    return 0;
}
```

函数回调

```c
void cal(int(*p)(int, int))
{
    int x = 0;
    int y = 0;
    int ret = 0;
    printf("请输入两个操作数:");
    scanf("%d %d", &x, &y);
    ret = (*p)(x, y);
    printf("ret=%d\n", ret);
}
int add(int x, int y)
{
    return x + y;
}
int sub(int x, int y)
{
    return x - y;
}
int mul(int x, int y)
{
    return x * y;
}
int div(int x, int y)
{
    return x / y;
}
void meau()
{
    printf("*****************************************\n");
    printf("**********   1.add     2.sub   **********\n");
    printf("**********   3.mul     4.div   **********\n");
    printf("**********      0.退出系统     **********\n");
    printf("*****************************************\n");
}
int main()
{
    int input = 0;
    int x = 0;
    int y = 0;
    int ret = 0;
    do
    {
        meau();
        printf("请选择：");
        scanf("%d", &input);

        switch (input)
        {
        case 1:
            cal(add);
            break;
        case 2:
            cal(sub);
            break;
        case 3:
            cal(mul);
            break;
        case 4:
            cal(div);
            break;
        case 0:
            printf("退出系统\n");
            break;
        default:
            printf("输入错误，请重新输入\n");
            break;
        }
        printf("\n\n");
    } while (input);
    return 0;
```

回调函数就是一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另外一个函数，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的，而是在特定的事件或条件发生时由另外一方调用的，用于对该事件或条件进行响应。

P120 指针进阶

void*可以接收任何形式的指针
库函数的查找网站 cplusplus

```c
//qsort函数排序
//void qsort(void* base,size_t num,size_t size,int (*cmp)(const void* e1,const void* e2));//函数声明
//base中存放的是待排序数据中的第一个对象的地址
//num是排序元素的个数
//size是一个排序元素的大小，单位是字节
//int (*cmp)(const void* e1,const void* e2)是用来比较待排序数据中的2个元素的函数
#include<stdlib.h>
#include<string.h>
void print(int* arr, int sz_1)
{
    int i = 0;
    for (i = 0; i < sz_1; i++)
    {
        printf("%d ", arr[i]);
    }
}

int cmp_int(const void* e1, const void* e2)
{
    return *(int*)e1 - *(int*)e2;
}

//整型数据排序
void test1()
{
    int arr[] = { 9,8,7,6,5,4,3,2,1,0 };
    int sz = sizeof(arr) / sizeof(arr[0]);
    //排序
    qsort(arr, sz,sizeof(arr[0]), cmp_int);
    //打印
    print(arr, sz);
}

//结构体数据排序
//定义结构体类型
struct Stu
{
    char name[20];
    int age;
};

int sort_by_age(const void* e1, const void* e2)
{
    return (*(struct Stu*)e1).age - (*(struct Stu*)e2).age;
}

int sort_by_name(const void* e1, const void* e2)
{
    return strcmp((*(struct Stu*)e1).name ,(*(struct Stu*)e2).name);
}

void test2()
{
    struct Stu s[3] = { {"zhangsan",30},{"lisi",34},{"wangwu",20} };
    //按照年龄来排序
    int sz = sizeof(s) / sizeof(s[0]);
    //qsort(s, sz, sizeof(s[0]), sort_by_age);
    //按照姓名排序
    qsort(s, sz, sizeof(s[0]), sort_by_name);
}

int main()
{
    //test1();//整型数据排序
    test2();//结构体数据排序
    return 0;
}
```

P122 指针进阶

void*类型的地址不能进行加减操作
void*类型可以存放任何类型的指针
但 void*类型的指针不能进行解引用操作，因为不知道需要操作几个字节的空间
同时，void*类型的指针是不能加减操作的

P124 指针进阶

数组名只有在两种情况下表示整个数组

1. sizeof（a）操作数只有单独的一个数组名，有其余的符号时，数组名表示的便是数组第一个元素的地址
2. &a 表示的是取数组 a 的地址

Sizeof（）内部的表达式不计算

P127 指针进阶

%x 打印的是 16 进制数
%#x 打印 16 进制数时会加上 0x，例如 0x00001000

P[0]可以写成*（p+0）

P129 指针练习

Printf 语句中的表达式不计算，即不影响变量的内容，只是单纯的计算出来数并输出
但是++，--可以改变变量的内容
Eg：

```c
int main()
{
    int a = 5;
    printf("%d\n", a + 1);//6
    printf("%d\n", ++a);//6
    printf("%d\n", a);//6
    return 0;
}
```

P130 指针作业讲解

动态内存开辟：
申请：malloc　　eg：p=malloc( )
使用
释放：free（p）；释放后的指针需要手动置空　p=NULL；
Free（p）；是把 p 指向的空间还给系统，但 p 里面还是存放的原来的地址，但此时通过 p 来访问该地址的时候就会报错，所以在释放空间之后，还需要手动置空

[ ]的优先级高于*

P134 指针作业讲解

char *strcat(char *dest, const char *src) 把 **src** 所指向的字符串追加到 **dest** 所指向的字符串的结尾。
strcat 函数不能自己追加自己，**strncat** 可以追加自己
char *strncat(char *dest, const char *src, size_t n) 把 **src** 所指向的字符串追加到 **dest** 所指向的字符串的结尾，直到 **n** 字符长度为止

char *strcpy(char *dest, const char *src) 把 **src** 所指向的字符串复制到 **dest**。
需要注意的是如果目标数组 dest 不够大，而源字符串的长度又太长，可能会造成缓冲溢出的情况。

char *strstr(const char *haystack, const char *needle) 在字符串 **haystack** 中查找第一次出现字符串 **needle** 的位置，不包含终止符 '\0'。

P137 字符串函数及模拟实现 strlen&&strcpy&&strcat

函数介绍
size_t strlen(const char* str)

- 字符串以'\0'作为结束标志，strlen 函数返回的是字符串中'\0'前面出现的字符个数（不包括'\0'）。
- 参数指向的字符串必须要以'\0'结束。
- 注意函数的返回值为 size_t，是无符号的整型。
- 学会 strlen 的模拟实现。

长度不受限制的字符串函数
char *strcpy(char *dest, const char *src) 把 **src** 所指向的字符串复制到 **dest**。

- 需要注意的是如果目标数组 dest 不够大，而源字符串的长度又太长，可能会造成缓冲溢出的情况。
- 源字符串 src 必须以'\0'结束。

- 会将源字符串中的'\0'拷贝到目标空间。
- 目标空间必须足够大，以确保能存放源字符串。
- 目标空间必须可修改，即不能是常量字符串。
- 学会模拟实现。

char *strcat(char *dest, const char *src) 把 **src** 所指向的字符串追加到 **dest** 所指向的字符串的结尾。

- 源字符串必须以'\0'结束
- 目标空间必须有足够大，能够容纳下源字符串的内容
- 目标空间必须可修改
- 不能自己后面追加自己，因为追加的时候把自己的'\0'覆盖掉了，没有'\0'来截至
- 模拟实现

int strcmp(const char *str1, const char *str2) 把 **str1** 所指向的字符串和 **str2** 所指向的字符串进行比较。

该函数返回值如下：

- 如果返回值小于 0，则表示 str1 小于 str2。
- 如果返回值大于 0，则表示 str1 大于 str2。
- 如果返回值等于 0，则表示 str1 等于 str2。

长度受限制的字符串函数

char *strncat(char *dest, const char *src, size_t n) 把 **src** 所指向的字符串追加到 **dest** 所指向的字符串的结尾，直到 **n** 字符长度为止。

char *strncpy(char *dest, const char *src, size_t n) 把 **src** 所指向的字符串复制到 **dest**，最多复制 **n** 个字符。当 src 的长度小于 n 时，dest 的剩余部分将用空字节填充。

int strncmp(const char *str1, const char *str2, size_t n) 把 **str1** 和 **str2** 进行比较，最多比较前 **n** 个字节。

P139 字符串函数及模拟实现 strlen&&strcpy&&strcat

char *strstr(const char *haystack, const char *needle) 在字符串 **haystack** 中查找第一次出现字符串 **needle** 的位置，不包含终止符 '\0'。

char *strtok(char *str, const char *sep) 分解字符串 **str** 为一组字符串，**sep** 为分隔符。

- sep 参数是个字符串，定义了用作分隔的字符集合
- 第一个参数指定一个字符串，它包含了 0 个或者多个由 sep 字符串中的一个或者多个分隔符分割的标记
- strtok 函数找到 str 中的下一个标记，并将其用'\0'结尾，返回一个指向这个标记的指针。（注意：strtok 函数会改变被操作的字符串，所以在使用 strtok 函数切分的字符串一般都

是临时拷贝的内容并且可修改。）

- strtok 函数的第一个参数不为 NULL，函数将找到 str 中第一个标记，strtok 函数将保存它在字符串的位置（保存位置为下一次切割的首地址）。（strtok 函数有记忆功能，那么他就是用静态局部变量（static）来实现的，用全局变量也可以实现，但是依赖全局变量的话该函数不够独立）
- strtok 函数的第一个参数为 NULL，函数将在同一个字符串中被保存的位置开始，查找下一个标记。
- 如果字符串中不存在更多的标记，则返回 NULL 指针

char *strerror(int errnum) 从内部数组中搜索错误号 **errnum**，并返回一个指向错误消息字符串的指针。**strerror** 生成的错误字符串取决于开发平台和编译器。

- **errnum** -- 错误号，通常是 **errno**。

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

int main ()
{
   FILE *fp;

   fp = fopen("file.txt","r");
   if( fp == NULL )
   {
      printf("Error: %s\n", strerror(errno));
   }
   Fclose(pf);
  return(0);
}
```

让我们编译并运行上面的程序，这将产生以下结果，因为我们尝试打开一个不存在的文件：

```
Error: No such file or directory
```

P140 内存函数

strerror 是把错误码转换成错误信息，perror 是直接打印除错误信息

C 库函数 **void perror(const char *str)** 把一个描述性错误消息输出到标准错误 stderr。首先输出字符串 **str**，后跟一个冒号，然后是一个空格。

1. 首先把错误码转换成错误信息
2. 打印错误信息（包含自定义信息）

字符转换函数和字符分类函数
Eg：tolower() isdigit()

内存操作函数
1.内存拷贝函数
void *memcpy(void *str1, const void *str2, size_t n) 从存储区 **str2** 复制 **n** 个字节到存储区 **str1**。
该函数返回一个指向目标存储区 str1 的指针。

Void*类型的指针不能解引用和加一操作。

P142 内存函数

memcpy 函数只能拷贝不重叠的内存，在 vs 中可以实现重叠内存拷贝。
memmove 函数可以处理重叠内存
memcmp 函数是内存比较函数，返回值的类型与 strcmp 相同
memset 内存设置函数，以字节为单位设置内存
void *memset(void *str, int c, size_t n) 复制字符 **c**（一个无符号字符）到参数 **str** 所指向的字符串的前 **n** 个字符。

P144 内存函数+自定义类型

匿名结构体类型：只能使用一次

```
struct
{
    char a;
    int b;
    double c;
}s;
```

结构体类型的自引用
引用的时候只能引用和自己类型相同的指针变量(链表)

```
struct N
{
    int a;
    struct N* s;
}
```

P145 自定义类型

结构体内存对齐：

1. 结构体的第一个成员放在结构体变量在内存中存储位置的 0 偏移处开始
2. 从第二个成员往后所有的成员，都放在一个对齐书（成员的大小和默认对齐数的较小值）的整数的整数倍的地址处
3. 结构体的总大小是结构体的所有成员的对齐数中最大的那个对齐数的整数倍
4. 如果嵌套了结构体的情况，嵌套的结构体对齐到自己最大对齐数的整数倍处，结构体的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

P147 自定义类型

修改默认对齐数

#pragma pack(2);//修改默认对齐数为 2

#pragma pack();//取消默认对齐数设置

默认对齐数表示最大储存空间，不能超过默认对齐数

百度笔试题：

写一个宏，计算结构体中某变量相对于首地址的偏移量，并给出说明

考察内容：offsetof 宏的实现

offsetof(type, member-designator) 会生成一个类型为 **size_t** 的整型常量，它是一个结构成员相对于结构开头的字节偏移量。成员是由 member-designator 给定的，结构的名称是在 type 中给定的。

注：需要引用头文件 stddef.h

结构体传参的时候要传地址，节省空间和时间

位段

位段的声明和结构体是类似的，有两个不同：

1. 位段的成员必须是 int、unsigned int 或者 signed int。也有 char 类型
2. 位段的成员名后面有一个冒号和数字

位段的内存分配：

1. 位段空间上是按 4 个字节（int）或者 1 个字节（char）的方式来开辟的
2. 位段涉及很多不确定的因素，位段是不跨平台，注重可移植的程序应该避免使用位段

eg：

```
struct A
{
    //4 个字节—32bit
    int _a:2;   //_a 成员占 2 个 bit 位
    int _b:5;   //_b 成员占 5 个 bit 位
    int _c:10;   //_c 成员占 10 个 bit 位
    //4 个字节—32bit
    int _d:30;   //_d 成员占 30 个 bit 位
}
```

位段跨平台的问题

1. int 位段被当成有符号数还是无符号数是不确定的
2. 位段中最大位的数目不能确定（16 位机器最大是 16，32 位机器最大是 32，写成 27，16 位机器会出现问题）
3. 位段中的成员在内存中从左向右分配，还是从右向左分配，标准尚未定义
4. 当一个结构包含两个位段，第二个位段成员比较大，无法容纳第一个位段剩余的位时，是舍弃剩余还是利用，这是不确定的

总结：位段可以节省空间，但有跨平台的问题

P149 自定义类型

枚举类型的定义：

```
enum Color
{
    REG，//0
    GREEN，//1
    BLUE //2            //默认递增
};
```

P150 联合体

联合体类型定义

联合也是一种特殊的自定义类型

这种类型定义的变量也包含一系列的成员，特征是这些成员共用同一块空间（所以联合也叫共用体）

eg：

```
union   Un   un
{
    int i;
    char c;
}
```

占 4 个字节

P151 联合体的计算

联合大小计算

1. 联合的大小至少是最大成员的大小。
2. 当最大成员大小不是最大对齐数的整数倍的时候，就要对齐到最大对齐数的整数倍

eg：

```
union Un
{
   char a[5];//对齐数按 5 个 char 类型计算，即为 1
   int i;//对齐数是 4
};
```

最大对齐数是 4，但最大空间是 5，不是最大对齐数的整数倍，所以对齐至 8

P151 通讯录

结构体可以直接用等号赋值。

P156 通讯录

动态内存函数

内存：

栈区：局部变量，函数形参

堆区：动态内存开辟：malloc，calloc，free，realloc

静态区：全局变量，静态变量

1. **void *malloc(size_t size)** 分配所需的内存空间，并返回一个指向它的指针。
   注：开辟完空间要检查是否开辟成功。头文件 stdlib.h

2. **void free(void *ptr)** 释放之前调用 calloc、malloc 或 realloc 所分配的内存空间。
   注：free 释放空间后，ptr 存放的还是原来空间的地址，需要手动变成 Null。
   free 只能释放动态开辟的空间

3. **void *calloc(size_t nitems, size_t size)** 分配所需的内存空间，并返回一个指向它的指针。
   nitems 指要被分配的元素个数。size 指元素的大小。
   区别：
   malloc 和 **calloc** 之间的不同点是，malloc 不会设置内存为零，而 calloc 会设置分配的内存为零。
   calloc 函数会将开辟空间初始化为 0，malloc 不会初始化。

4. **void *realloc(void *ptr, size_t size)** 尝试重新调整之前调用 **malloc** 或 **calloc** 所分配的 **ptr** 所指向的内存块的大小。
   ptr -- 指针指向一个要重新分配内存的内存块，该内存块之前是通过调用 malloc、calloc 或 realloc 进行分配内存的。如果为空指针，则会分配一个新的内存块，且函数返回一个指向它的指针。

size -- 内存块的新的大小，以字节为单位。如果大小为 0，且 ptr 指向一个已存在的内存块，则 ptr 所指向的内存块会被释放，并返回一个空指针。

注意：realloc 重新分配空间的时候，是在原有的空间后面增加空间，返回整片空间的地址，如果原有空间后面没有足够的空间，那么就如下实现：

1. 重新找一片空间
2. 将原来空间中的内容复制到新空间
3. 释放原来的空间
4. .返回新空间的地址

如果新找空间都没有位置，则返回 null

所以，重新分配空间之后的返回值要新建立一个指针存储，判断与原空间地址是否相同

P158 动态内存分配 1

动态内存开辟常见的错误

1. 对 NULL 指针解引用操作
2. 对动态空间的越界访问
3. 使用 free 释放非动态开辟的空间
4. 使用 free 释放动态空间的一部分
5. 对同一块动态开辟的空间多次释放
6. 动态开辟的空间忘记释放---内存泄漏

   注：释放空间的两种方式：

   （1）：主动释放动态空间

   （2）：程序结束（不是函数结束，是整个程序结束）被动释放

P160 动态内存分配 2

printf（地址）可以直接打印该地址的内容

栈区存放的是局部变量，形参等临时变量，会被销毁的

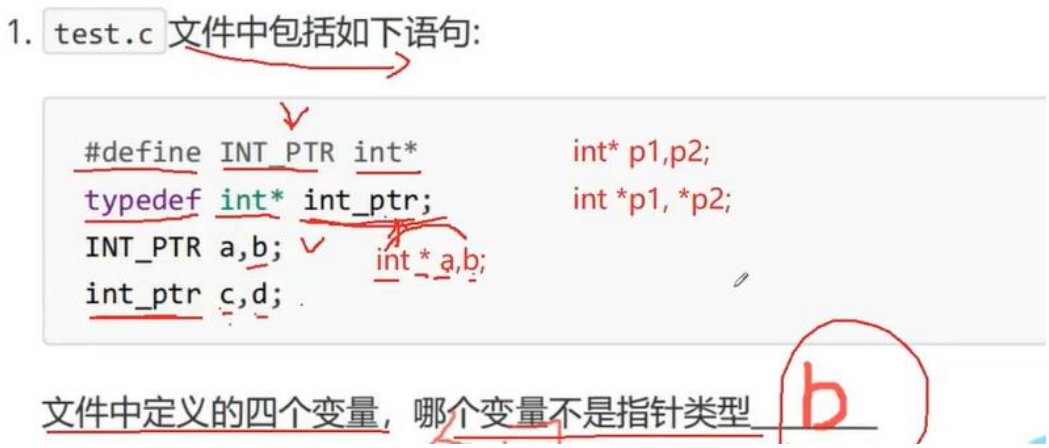
堆区存放的是动态开辟的空间，不会被销毁

P162 柔性数组

指针定义的时候不能连续定义

eg：int* p1,p2　　*给了 p1，p2 没有*，所以 p1 是指针变量，p2 只是整型

改正：int* p1，*p2



柔性数组

c99 中，结构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 s
{
    int i;
    int a[ ];//柔性数组，也可以写成 a[0],具体看编译器支持哪一种
}
```

柔性数组的特点

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，已适应柔性数组的预期大小

柔性数组可以通过对结构体开辟内存，再对结构体中的指针开辟内存达到同样的效果，但是这种两次开辟动态内存的方法释放空间的时候就必须先释放结构体中指针变量的内存，再释放结构体的内存。所以还是柔性数组好用

多次开辟动态内存的时候，释放空间的顺序

一般不会多次开辟动态内存，因为会出现内存碎片，降低对内存的利用率，降低对内存的访问速度

P167 文件 1

文件的分类

按功能分：程序文件，数据文件

程序文件：源程序文件：.c 文件　　目标文件：windows 为.obj 文件　可执行文件：windows 为.exe 文件

文件指针

每个被使用的文件都在内存中开辟了一个相应的文件信息区，用来存放文件的相关信息（如文

件的名字，文件的状态及文件当前的位置等）。这些信息是保存在一个结构体变量中的。该结构体类型是有系统声明的，取名 FILE。不同编译器 FILE 结构体不同。

FILE *fopen(const char *filename, const char *mode) 使用给定的模式 **mode** 打开 **filename** 所指向的文件

参数：

- **filename** -- 字符串，表示要打开的文件名称。
- **mode** -- 字符串，表示文件的访问模式，可以是以下表格中的值：

mode 有下列几种形态字符串:

- r 以只读方式打开文件，该文件必须存在。
- r+ 以可读写方式打开文件，该文件必须存在。
- rb+ 读写打开一个二进制文件，允许读数据。
- rw+ 读写打开一个文本文件，允许读和写。
- w 打开只写文件，若文件存在则文件长度清为 0，即该文件内容会消失。若文件不存在则建立该文件。
- w+ 打开可读写文件，若文件存在则文件长度清为零，即该文件内容会消失。若文件不存在则建立该文件。
- a 以附加的方式打开只写文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留。（EOF 符保留）
- a+ 以附加方式打开可读写的文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾后，即文件原先的内容会被保留。（原来的 EOF 符不保留）
- wb 只写打开或新建一个二进制文件；只允许写数据。
- wb+ 读写打开或建立一个二进制文件，允许读和写。
- ab+ 读写打开一个二进制文件，允许读或在文件末追加数据。
- at+ 打开一个叫 string 的文件，a 表示 append,就是说写入处理的时候是接着原来文件已有内容写入，不是从头写入覆盖掉，t 表示打开文件的类型是文本文件，+号表示对文件既可以读也可以写。

上述的形态字符串都可以再加一个 b 字符，如 rb、w+b 或 ab+等组合，加入 b 字符用来告诉函数库以二进制模式打开文件。如果不加 b，表示默认加了 t，即 rt,wt,其中 t 表示以文本模式打开文件。由 fopen()所建立的新文件会具有 S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH(0666)权限，此文件权限也会参考 umask 值。

有些 C 编译系统可能不完全提供所有这些功能，有的 C 版本不用"r+","w+","a+",而用"rw","wr","ar"等，读者注意所用系统的规定。

二进制和文本模式的区别

1.在 windows 系统中，文本模式下，文件以"\r\n"代表换行。若以文本模式打开文件，并用 fputs 等函数写入换行符"\n"时，函数会自动在"\n"前面加上"\r"。即实际写入文件的是"\r\n" 。

2.在类 Unix/Linux 系统中文本模式下，文件以"\n"代表换行。所以 Linux 系统中在文本模式和二进制模式下并无区别。

c 语言程序，只要运行起来，就默认打开了 3 个流
stdin-标准输入流-键盘
stdout-标准输出流-屏幕
stderr-标准错误流-屏幕

功能	函数名	适用于
字符输入函数	fgetc	所有输入流
字符输出函数	fputc	所有输出流
文本行输入函数	fgets	所有输入流
文本行输出函数	fputs	所有输出流
格式化输入函数	fscanf	所有输入流
格式化输出函数	fprintf	所有输出流
二进制输入	fread	文件
二进制输出	fwrite	文件

int fgetc(FILE *stream) 从指定的流 stream 获取下一个字符（一个无符号字符），并把位置标识符往前移动。
int fputc(int char, FILE *stream) 把参数 **char** 指定的字符（一个无符号字符）写入到指定的流 stream 中，并把位置标识符往前移动。

char *fgets(char *str, int n, FILE *stream) 从指定的流 stream 读取一行，并把它存储在 **str** 所指向的字符串内。当读取 **(n-1)** 个字符时（最后放一个\0），或者读取到换行符时，或者到达文件末尾时，它会停止，具体视情况而定。
int fputs(const char *str, FILE *stream) 把字符串写入到指定的流 stream 中，但不包括空字符。

int fscanf(FILE *stream, const char *format, ...) 从流 stream 读取格式化输入。
int fprintf(FILE *stream, const char *format, ...) 发送格式化输出到流 stream 中。

P169 文件 2

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream) 从给定流 **stream** 读取数据到 **ptr** 所指向的数组中。
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream) 把 **ptr** 所指向的数组中的数据写入到给定流 **stream** 中。返回值为成功写的元素个数。

P170 文件 3

scanf　针对标准输入的格式化的输入语句-stdin
fscanf　针对所有输入流的格式化输入语句-stdin、文件
sscanf　从一个字符串中读取一个格式化数据

printf　针对标准输出的格式化输出语句-stdout
fprintf　针对所有输出流的格式化输出语句-stdout、文件
sprintf　把一个格式化的数据，转化成字符串

int sscanf(const char *str, const char *format, ...) 从字符串读取格式化输入
int sprintf(char *str, const char *format, ...) 发送格式化输出到 **str** 所指向的字符串

文件随机读写
int fseek(FILE *stream, long int offset, int whence) 设置流 **stream** 的文件位置为给定的偏移 **offset**，参数 offset 意味着从给定的 **whence** 位置查找的字节数。
参数：
stream -- 这是指向 FILE 对象的指针，该 FILE 对象标识了流。
offset -- 这是相对 whence 的偏移量，以字节为单位。
whence -- 这是表示开始添加偏移 offset 的位置。它一般指定为下列常量之一：

常量	描述
SEEK_SET	文件的开头
SEEK_CUR	文件指针的当前位置
SEEK_END	文件的末尾

返回值：如果成功，则该函数返回零，否则返回非零值。

long int ftell(FILE *stream) 返回给定流 stream 的当前文件位置
void rewind(FILE *stream) 设置文件位置为给定流 **stream** 的文件的开头

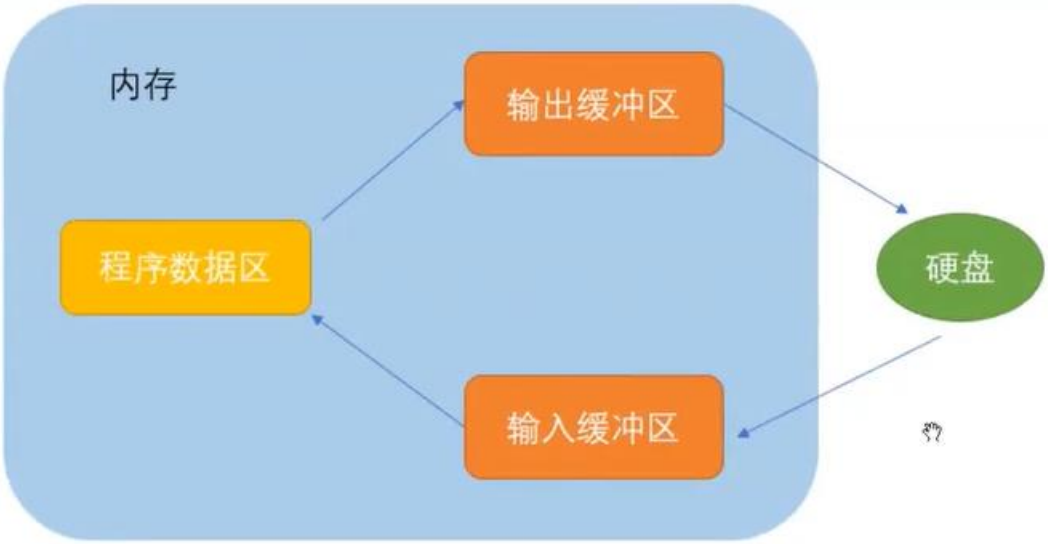
int feof(FILE *stream) 函数
当设置了与流关联的文件结束标识符时，该函数返回一个非零值，否则返回零
**在文件读取过程中，不能用 feof 函数的返回值直接用来判断文件是否结束
而是应用于当文件读取结束时，判断是读取失败结束，还是遇到文件尾结束。

int ferror(FILE *stream) 测试给定流 stream 的错误标识符。
如果设置了与流关联的错误标识符，该函数返回一个非零值，否则返回一个零值。

1.文本文件读取是否结束，判断返回值是否为 EOF（fgetc），或者 NULL（fgets）
2.二进制文件的读取结束判断，判断返回值是否小于实际要读个数

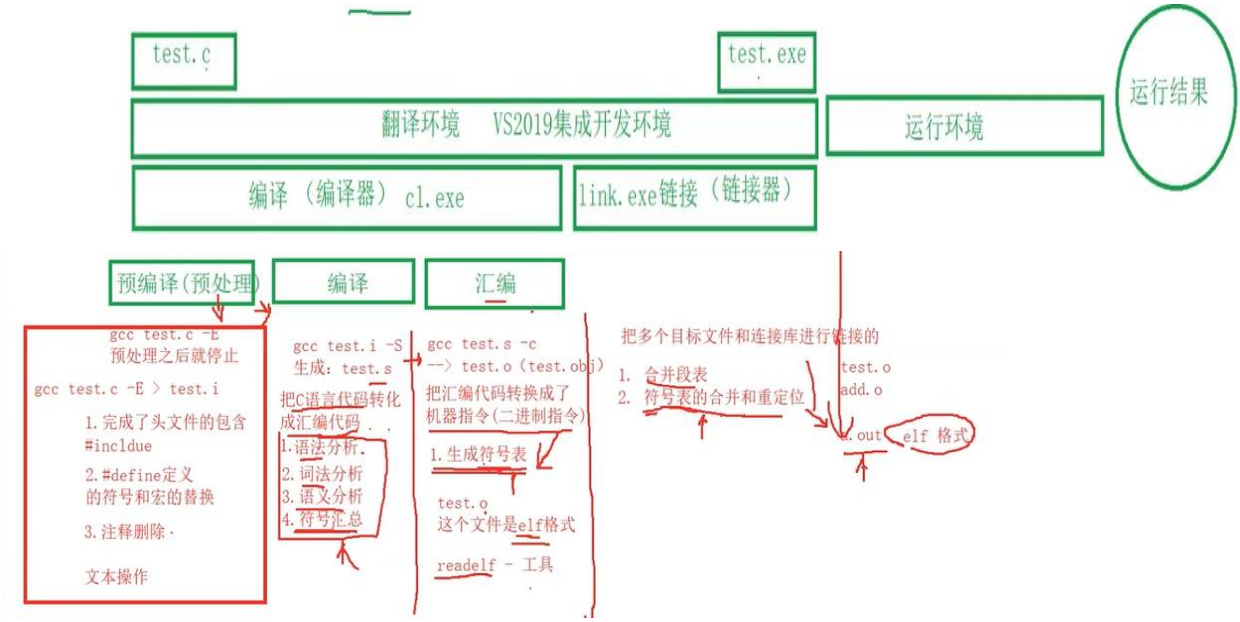
文件缓冲区

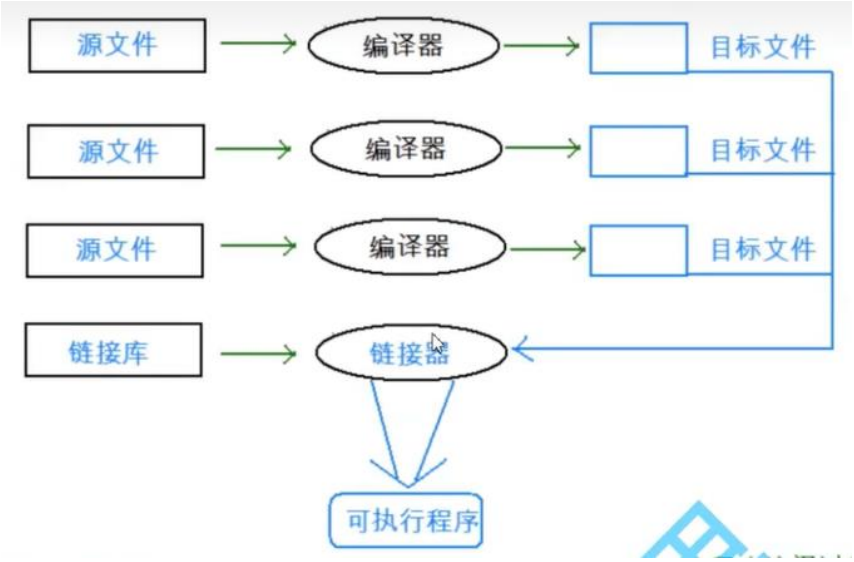
ANSIC 标准采用**"缓冲文件系统"**处理的数据文件的，所谓缓冲文件系统是指系统自动地在内存中为程序中每一个正在使用的文件开辟一块**"文件缓冲区"**。从内存向磁盘输出数据会先送到内存中的缓冲区，装满缓冲区后才一起送到磁盘上。如果从磁盘向计算机读入数据，则从磁盘文件中读取数据输入到内存缓冲区（充满缓冲区），然后再从缓冲区逐个地将数据送到程序数据区（程序变量等）。缓冲区的大小根据C编译系统决定的。



P172 预处理 1and 预处理 2

程序环境和预处理





P175 预处理 3

\——续行符

#define 定义宏
#define 机制包括一个规定，允许把参数替换到文本中，这种实现通常称为宏或者定义宏

宏的声明方式
#define name(parament-list) stuff
其中的 parament-list 是一个由逗号隔开的符号表，它们可能出现在 stuff 中

注意：
参数列表的左括号必须与 name 紧邻
如果两者之间有任何空白存在，参数列表就会被解释为 stuff 的一部分

#define square(x) x*x
printf(“%d\n”,square(3));
替换后为 printf(“%d\n”,3*3); 结果为 9
printf(“%d\n”,square(3+1));
替换后为 printf(“%d\n”,3+1*3+1); 结果为 7
所以宏的参数是完全替换，为了防止这种情况，通常把宏定义如下
#define square(x) ((x)*(x))

错误例子
#define double(x) (x)+(x)
printf(“%d\n”,10*double(4));
替换后 printf(“%d\n”,10*4+4); 结果等于 44

#define 的替换规则

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。

#--在字符串中插入字符串

eg：

```
#define PRINT(X, FORMAT) printf("the value of "#X" is "FORMAT"\n", X);

int main()
{
    //printf("hello world\n");
    //printf("hello " "world\n");

    int a = 10;
    PRINT(a, "%d");
    printf("the value of ""a"" is %d\n", a);
    //the value of a is 10

    int b = 20;
    PRINT(b, "%d");
    printf("the value of ""b"" is %d\n", b);

    //the value of b is 20

    int c = 30;
    PRINT(c, "%d");
    //the value of c is 30

    float f = 5.5f;
    PRINT(f, "%f");


    return 0;
}
```

##--连接两个字符串

##可以把位于它两边的符号合成一个符号

它允许宏定义从分离的文本片段创建标识符

```
#define CAT(X,Y) X##Y

int main()
{
    int class101 = 100;
    printf("%d\n", CAT(class, 101));

    //printf("%d\n", class101);


    return 0;
}
```

注：这样连接必须产生一个合法的标识符，否则其结果就是未定义

P177 预处理 4

带副作用的宏参数

```c
#define MAX(X,Y) ((X)>(Y)?(X):(Y))

int main()
{
    int a = 5;
    int b = 8;
    int m = MAX(a++, b++);
    printf("a=%d b=%d\n", a, b);

    //int m = ((a++) > (b++) ? (a++) : (b++));

    printf("m = %d\n", m);//?

    return 0;
}
```

运行结果：m=9，a=6，b=10

宏和函数的区别

1. 用于调用函数和从函数返回的代码可能比实际执行这个小型计算工作所需要的时间更多。**所以宏比函数在程序的规模和速度方面更胜一筹。**
2. 更为重要的是函数的参数必须声明为特定的类型。所以函数只能在类型合适的表达式上使用。反之这个宏怎可以适用于整形、长整型、浮点型等可以用于>来比较的类型。**宏是类型无关的。**

当然和宏相比函数也有劣势的地方：

1. 每次使用宏的时候，一份宏定义的代码将插入到程序中。除非宏比较短，否则可能大幅度增加程序的长度。
2. 宏是没法调试的。
3. 宏由于类型无关，也就不够严谨。
4. 宏可能会带来运算符优先级的问题，导致程容易出现错。

宏有时候可以做函数做不到的事情。比如：宏的参数可以出现**类型**，但是函数做不到。

宏和函数的一个对比

属性	#define定义宏	函数
代码长度	每次使用时，宏代码都会被插入到程序中。除了非常小的宏之外，程序的长度会大幅度增长	函数代码只出现于一个地方；每次使用这个函数时，都调用那个地方的同一份代码
执行速度	更快	存在函数的调用和返回的额外开销，所以相对慢一些
操作符优先级	宏参数的求值是在所有周围表达式的上下文环境里，除非加上括号，否则邻近操作符的优先级可能会产生不可预料的后果，所以建议宏在书写的时候多些括号。	函数参数只在函数调用的时候求值一次，它的结果值传递给函数。表达式的求值结果更容易预测。

带有副作用的参数	参数可能被替换到宏体中的多个位置，所以带有副作用的参数求值可能会产生不可预料的结果。	函数参数只在传参的时候求值一次，结果更容易控制。
参数类型	宏的参数与类型无关，只要对参数的操作是合法的，它就可以使用于任何参数类型。	函数的参数是与类型有关的，如果参数的类型不同，就需要不同的函数，即使他们执行的任务是不同的。
调试	宏是不方便调试的	函数是可以逐语句调试的
递归	宏是不能递归的	函数是可以递归的

#undef 取消宏定义

3.4 命令行定义

许多C 的编译器提供了一种能力，允许在命令行中定义符号。用于启动编译过程。
例如：当我们根据同一个源文件要编译出不同的一个程序的不同版本的时候，这个特性有点用处。（假定某个程序中声明了一个某个长度的数组，如果机器内存有限，我们需要一个很小的数组，但是另外一个机器内存大写，我们需要一个数组能够大写。）

命令行定义看看就行

条件编译

在编译一个程序的时候我们如果要将一条语句（一组语句）编译或者放弃是很方便的。因为我们有条件编译指令。

比如说：

> 调试性的代码，删除可惜，保留又碍事，所以我们可以选择性的编译。

#ifdef === #if defined
#ifndef === #if !defined

嵌套式

```
4.嵌套指令
#if defined(OS_UNIX)
    #ifdef OPTION1
        unix_version_option1();
    #endif
    #ifdef OPTION2
        unix_version_option2();
    #endif
#elif defined(OS_MSDOS)
    #ifdef OPTION2
        msdos_version_option2();
    #endif
#endif
```

P179 预处理 5

头文件的包含

用" "包含头文件，那么就是包含的是本地文件

用<>包含头文件，包含的就是库文件

两者查找策略不同：" "先在当前目录查找，找不到在标准位置（库函数的头文件下查找）

<>直接在标准位置查找

防止头文件被包含两次

方法 1：

#pragma once 放在头文件中，表示该头文件只会被包含 1 次

方法 2：

条件编译

P180 结构体大小计算作业

```
int main()
{
    unsigned char puc[4];
    struct tagPIM
    {
        unsigned char ucPim1;
        unsigned char ucData0 : 1;
        unsigned char ucData1 : 2;
        unsigned char ucData2 : 3;
    }*pstPimData;
    pstPimData = (struct tagPIM*)puc;
    memset(puc,0,4);
    pstPimData->ucPim1 = 2;
    pstPimData->ucData0 = 3;
    pstPimData->ucData1 = 4;
    pstPimData->ucData2 = 5;
    printf("%02x %02x %02x %02x\n",puc[0], puc[1], puc[2], puc[3]);
    return 0;
}
```

题目内容：

A.02 03 04 05

B.02 29 00 00

C.02 25 00 00

D.02 29 04 00

B

一个典型的C程序编译管道，包含预处理、编译、汇编、链接四个环节。

.c 文件----->预处理产生.i 文件----->编译产生.s 文件----->汇编产生.o 文件----->链接产生.exe 文件