# C语言概述

## 计算机语言的发展

1、机器语言：计算机所能识别的唯一语言（二进制语言）。例如：10100000。计算机刚问世时就是用1和0来编程，编程起来很不方便。

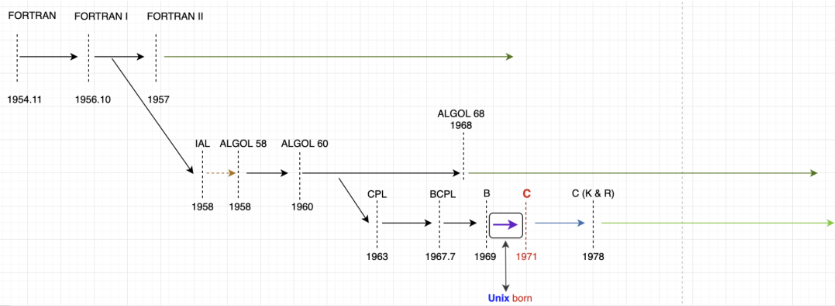
2、汇编语言：因为机器语言很不容易记忆，所以出现了汇编语言，汇编语言相当于给一些经常出现的机器语言起一个英文名字。例如mov，便于使用和记忆。但是使用起来相对高级语言还是很麻烦并且可移植性不好，不同系列的芯片汇编指令不一样。

3、高级语言：把汇编语言再次进行封装。相比于汇编语言使用起来更简单，可移植性更好。例如C语言、C++、VB、java等等。

c语言相比较其他高级语言，更接近于汇编语言，执行效率更高，所以经常用来编写底层软件，例如裸机程序、操作系统等等。

c语言为什么可移植性好？因为c语言编写好的程序最终还是要使用编译器编程成汇编语言，再经过汇编器汇编成机器语言。因为芯片厂商都会提供各自芯片的c语言编译器，所以同样的c语言程序，使用不同的编译器，编译的结果就可以在不同的芯片上运行。

## C语言发展史



## C语言特点

**优点**：简洁、紧凑、灵活；表达方式简练、实用；丰富的数据类型；具有低级语言的特点；是一种结构化语言，适合于大型程序的模块化设计；各种版本的编译系统都提供了预处理命令和预处理程序；可移植性好；生成的目标代码质量高；C语言语法限制不严，程序设计自由度大。**缺点**：C程序的错误更隐蔽；C程序有时会难以理解；C程序有时会难以修改。

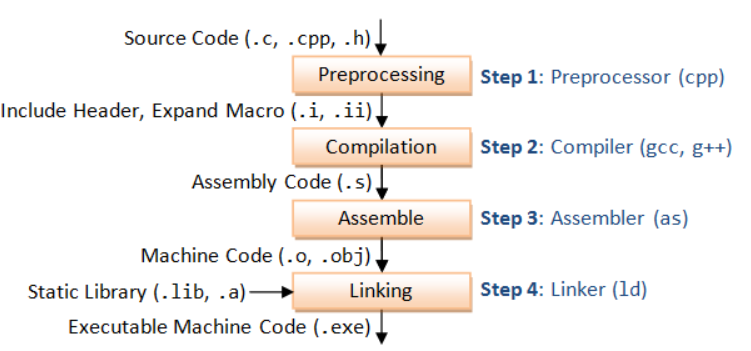
## C语言标准

1978年K&R C；之后C89\C90\C99\C11\C18；未来C2x

## 编程机制

预处理、编译、汇编、链接（gcc hello.c # 编译 ./a.out # 执行）

源文件&库文件→文本文件→汇编文件→机器码目标文件→可执行文件



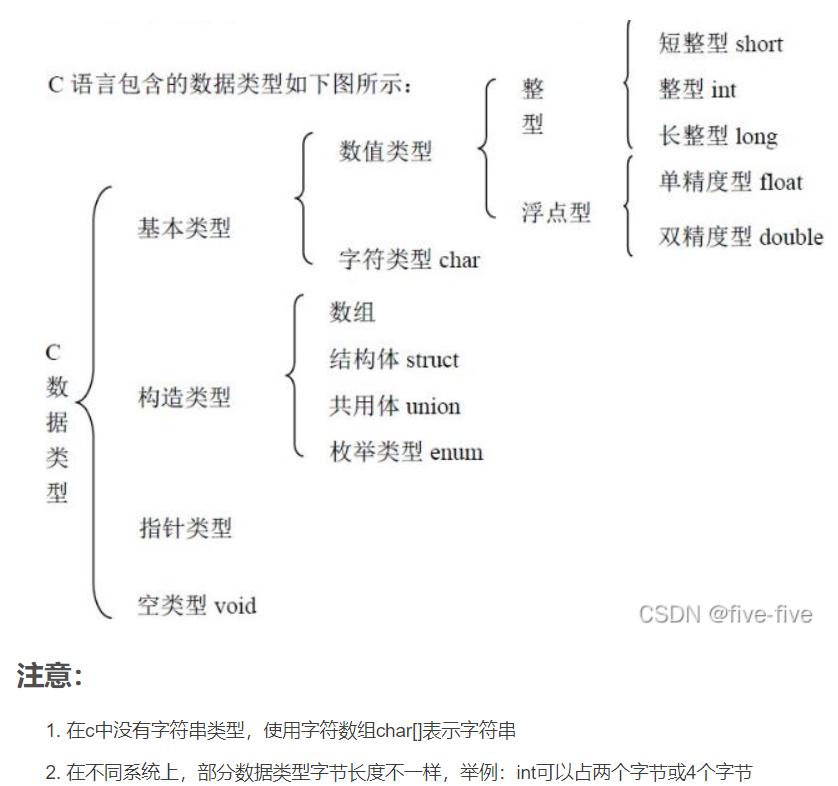
1. **预处理**：将源代码（.c\.cpp\.h）通过（CPP）处理成文本文件（.i\.ii）。预处理用于将所有的#include头文件以及宏定义替换成其真正的内容，预处理之后得到的仍然是文本文件，但文件体积会大很多。（gcc -E -I./inc test.c -o test.i）
2. **编译**：将经过预处理之后的程序转换成特定汇编代码(assembly code)的过程（gcc -S -I./inc test.c -o test.s），其中-E结束后退出，-I库文件，-o目标
3. **汇编**：将上一步的汇编代码转换成机器码(machine code)，这一步产生的文件叫做目标文件，是二进制格式。这一步会为每一个源文件产生一个目标文件。（gcc -c test.s -o test.o）
4. **链接**：将多个目标文以及所需的库文件(.so等)链接成最终的可执行文件(executable file)。（ld -o test.out test.o inc/mymath.o ...libraries...）

# 数据类型

## 变量

**步骤：声明、赋值、使用**

注意：变量表示内存中的一个存储区域（不同的数据类型，占用的空间大小不一样）；该区域有自己的名称 和类型；变量必须先声明，后使用；该区域的数据可以在同一类型范围内不断变化；变量在同一个作用域内不能重名；变量三要素 (变量名+值+数据类型) ，这一点请大家注意。



**整形变量，**各种类型的存储大小与操作系统、 系统位数和编译器有关 ，目前通用的以64位系统为主。在实际工作中，C程序通常运行在linux/unix 操作系统下。一般在windows10(64位) C语言的整型类型分为有符号signed和无符号unsigned 两种，默认是 signed。C程序中整型常声明为int型，除非不足以表示很大的数，才使用long long。bit(位): 计算机中的最小存储单位，byte(字节):计算机中基本存储单元。

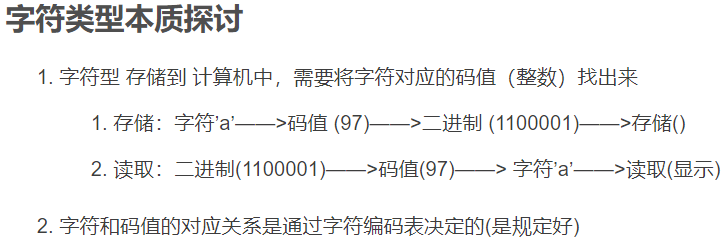
**1byte = 8bit，short在内存中占有2字节，int在内存中占有4个字节。**



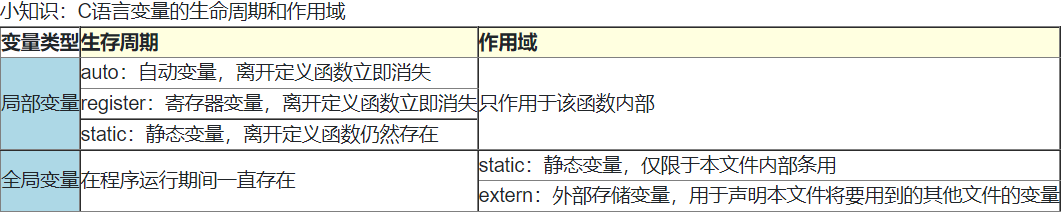
**浮点型：**浮点型变量默认为double型，声明float型变量时，须后加‘f’或‘F’。浮点型变量有两种表示形式：**①**十进制数形式，如5.12、512.0f、.512 (必须有小数点）；**②**科学计数法形式，如5.12e2、5.12E-2。**double型比float型更精确**。printf("d1=%f ", d1)，在输出时默认保留小数点后6位。浮点数=符号位+指数位+尾数位，浮点数是近似值，尾数部分可能丢失，造成精度损失。



**字符型：**字符类型可以表示单个字符，字符类型是char，占用1个字节(可以存字母或数字)。多个字符称为**字符串**，在C语言中使用char[ ]表示，**数组不是基本数据类型而是构造类型**。**字符常量**是用单引号('')括起来的单个字符，例如char c1 ='a'; char c3 = '9'。C中还允许使用转义字符‘\’来将其后的字符转变为特殊字符型常量，例如char c3 = ‘\n’，'\n'表示换行符。**在C语言中char的本质是一个整数**，在输出时显示ASCII码对应的字符。可以直接给char赋一个整数，输出时也会按照对应的ASCII字符输出[97]。char类型是可以像一个整数进行运算，因为它都对应有Unicode码。



**布尔型：**C语言标准(C89)没有定义布尔类型，所以C语言判断真假时以0为假，非0为真 。但这种做法不直观，所以我们可以借助C语言的宏定义。C语言标准(C99)提供了\_Bool 型(仍是整数类型)，但与一般整型不同的是\_Bool变量只能赋值为0或1，非0的值都会被存储为1。C99还提供了一个头文件<stdbool.h> 定义了bool代表\_Bool，true代表1，false代表0。只要导入stdbool.h，就能方便的操作布尔类型了, 比如 bool flag = false与条件控制语句if、循环控制语句while …



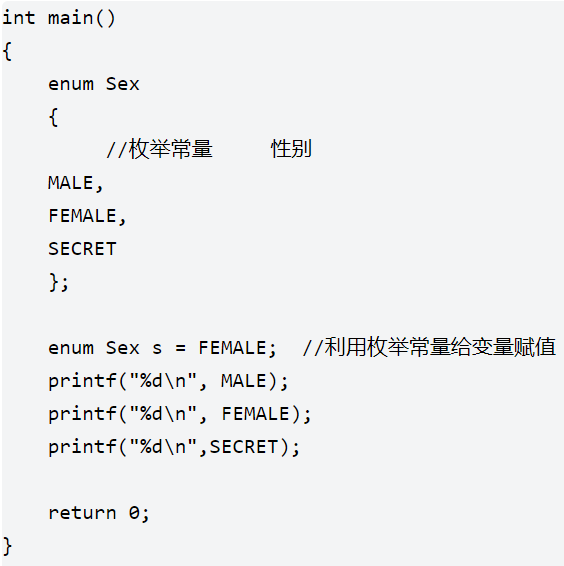
## 常量

**字面常量：**字面常量是直接写出的固定值，它包含C语言中可用的所有数据类型，可分为整型常量、字符常量等。如：9.9，“hello”等就属于这一类常量。

**const修饰的常变量：**有的时候我们希望定义一种变量，值不能被修改并在整个作用域中都维持原值。为了满足用户需求，C语言标准提供const关键字。在定义变量的同时，在变量名之前加上const修饰。const修饰的常变量本质上是变量，但具有常属性不能被修改。在C99标准之前，数组的大小只能是带const修饰的变量，不支持变长数组，如const int n = 20、int arr[n] = { 0 }。

**#define定义的标识符常量：**C语言提供了#define命令定义标识符常量，该标识符常量在程序中是个定值，通常用于代表数组容量或数学常量等。

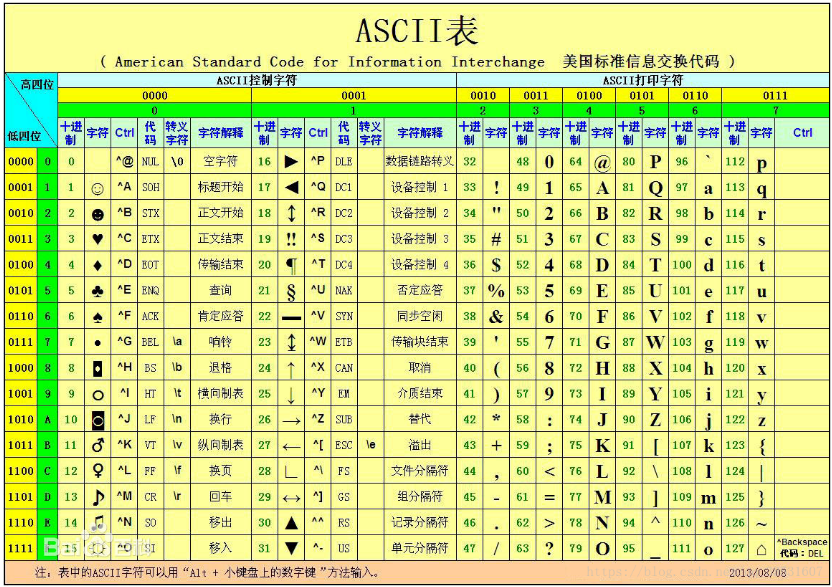
**枚举常量：**日常生活中有一些代表实际意义的常量，特点是数值较小且为非负整数。如一周有7天等。C语言提供了一种枚举（Enum）类型，能够列出所有可能会用到的取值，并给它们取一个名字。不能对枚举常量赋值，只能将它们的值赋给其他的变量。不能再定义与枚举常量名字相同的变量，且不能用&取得它们的地址。



## 基本数据类型

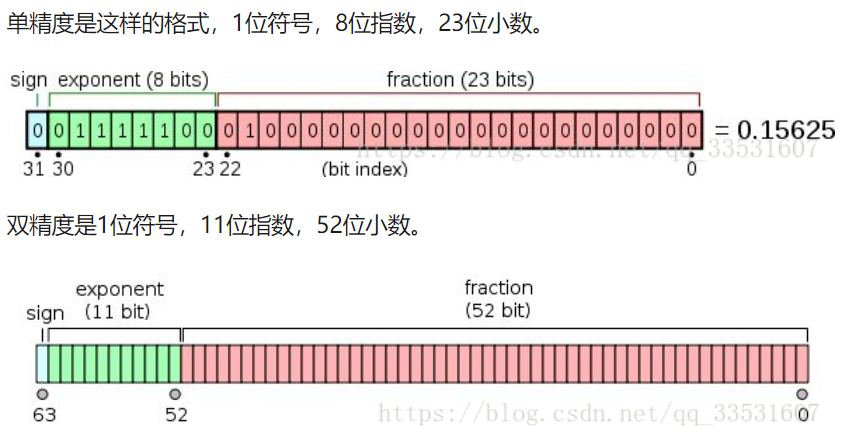
**基本数据类型**

**字符型**：占用1个字节（即8位），存放整数（有符号-128 ~ 127无符号0 ~ 255）。特殊符号例如：'\r'--回车、'\n'--换行、'\b'--退格、'\“'—引号。



**整型：**在32位CPU下int占4个字节，long int占4个字节，short int占2个字节。

**浮点型：**单精度浮点型float占4个字节，双精度浮点型double占8个字节。



# 运算符与表达式

## 常用运算符与表达式

**赋值运算符与表达式：**= += -= \*= /= %=

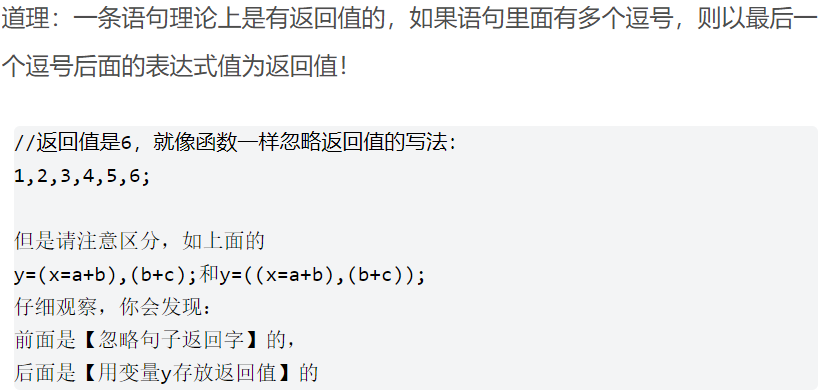
可以int a,b,c; a=b=c=10; 不能int a=b=c=10——不满足先声明后调用

**算数运算符与表达式：**+ - \* / % ++ --

**关系运算符与表达式：**< <= > >= = !=

**逻辑运算符与表达式：**&& || !

## 逗号运算符与表达式



**图片中第一个y=a+b，第二个y=b+c**

## 条件运算符

**表达式1?表达式2:表达式3;** 先执行表达式1，如果表达式1的结果如果为真就执行表达式2，这个运算式最终的结果是表达式2的结果；如果表达式1的结果如果为假就执行表达式3，运算式最终的结果是表达式3的结果。三目运算符可以嵌套，但不建议这样用。

## sizeof运算符

sizeof是C语言的关键字，它用来计算变量（或数据类型）在当前系统中占用内存的字节数。**注：sizeof的写法确实有点像函数，但sizeof真不是函数**，sizeof有**两种写法**：**①**数据类型必须用括号括住：sizeof(数据类型)，例如printf("%d\n",sizeof(char));**②**变量名可以不用括号括住：“sizeof(变量名)、 sizeof 变量名“均可。

**sizeof（结构体）：**理论上讲结构体的各个成员在内存中是连续存放的和数组非常类似，但是**结构体占用内存的总大小不一定等于全部成员变量占用内存大小之和**。因为在编译器底层，为了提高内存寻址的效率，会采用**对其机制**各个成员之间可能会存在缝隙。用sizeof可以得到结构体占用内容在总大小，sizeof(结构体名)或sizeof(结构体变量名)都可以。

**不要对void使用sizeof：**void是无值型或空类型，该类型存储空间大小未知，编译器也不能确定它的大小。void不能声明变量（例如void a），也不能对void使用sizeof。

**不要对指针用sizeof：**如果一个字符指针（如char \*pstr）指向字符串（如char strname[21]）。pstr的值就是数组的地址，如果用sizeof(pstr)得到的不是字符串占用内存的字节数，而是字符指针变量占用内存的字节数（8字节）。

**不要对结构体指针用sizeof：**结构体指针（如struct student \*stuptr）指向结构体（如struct student stu），sizeof(stuptr)的结果也是指针的变量占用的内存数（8字节），正确的用法是用sizeof(struct student)。

## 运算符优先级与求值顺序







# 语句与控制流

## 语句与程序块

一个函数包含声明部分和执行部分。执行部分由语句组成，经编译后产生机器指令**。声明部分不是语句，不产生机器指令**。一个C程序的结构如下：



**分类：控制语句（if/else/for/while/do/continue/break/switch/return/goto）、函数调用语句（printf）、表达式语句、空语句、复合语句(通过大括号{}复合)。**

**输入输出语句：**

**(1). printf** 一般形式为printf(格式控制，输出表列)

d格式符，用来输出有符号十进制整数。%5d表示数据占据5列且数据靠右，若想靠左则为%-5d。**但%+d则表示输出整数前的正负号**。长整型则为%ld，双长整型为%lld，l可放在d、o、x、u前。**而在scanf中%5d则表示只读取前5位整数**。

c格式符，用来输出一个字符。

s格式符，用来输出一个字符串。%5s在printf中表示只输出前5个字符，在scanf中表示只读取前5个字符。

f格式符，用来输出实数。基本型用%f来输出，默认实数部分全部输出小数部分输出6位。**%m.nf指定数据宽度和小数位数，其中m是指包含小数点在内的整个浮点数长度而不单单指整数部分，如果n为0则不会输出小数点**。

e格式符，用来输出指数形式的实数。同样可以用%m.ne形式声明，也可以写成%E此时输出的指数也显示为大写“E“。

**i格式符，在printf中与%d没有区别。但是在scanf中，%d只与十进制形式的整数相匹配，而%i则可以匹配八进制、十进制、十六进制表示的整数**。如输入的数字有前缀0（018、025），%i会把它当作八进制数来处理，如果有前缀0x (如0x54)它将以十六进制来处理。

o格式符，连同符号位一起以八进制整数形式输出。

x格式符，连同符号位一起以十六进制整数形式输出，写成%X时以大写形式显示。

u格式符，用来输出无符号整数。

g格式符，用来输出浮点数，并由系统自动选择为f格式或e格式输出，写成%G时以大写形式显示。

p格式符，用来输出地址(可以写指针)。

**(2). scanf** 一般形式为scanf(格式控制，地址表列)

其中，l可用于输入长整型数据如ld、lo、lx、lu以及double型数据lf、le。h可用于输入短整型数据如hd、ho、hx。在scanf(“a=%d, b=%d”, &a, &b)中输入则必须为a=1, b=2。若为scanf(“a=%d b=%d”, &a, &b)则a=1与b=2之间必须要有1个以上的空白字符。输入的形式要与格式控制严格匹配。scanf和printf函数都有返回值，scanf返回正确读入了多少个变量，printf返回输出了包括换行符在内的有多少个字符。

**(3). sprintf** 一般形式为sprintf(目标字符串，格式控制，地址表列)

sprintf函数用于将字符串输出到目标字符串中(而不是屏幕)，要注意字符串的长度要足够容纳打印的内容，否则会出现内存溢出。

**(4). sscanf** 一般形式为sscanf(源字符串，格式控制，地址表列)

sscanf与scanf类似，都是用于输入的，只是后者以外部设备(stdin)为输入源，前者以固定字符串为输入源。sscanf可用来进行数据类型或进制转换以及按特定规则读取源字符串的特定规则。也即使用正则表达式，但是功能并没有正则表达式强大，并且使用时有较多限制。同理， scanf以及printf中的格式控制都能使用正则表达式。C语言并不直接支持正则表达式，而需要通过函数库regex.h来完成。

几种常见的表达方法：①%[abc]读取a、b、或c的任意一员，[a-z]表示匹配小写字母，[a-z0-9]表示匹配小写字母及数字；②%[^a-z]读取不在a-z之间的字符串，如果碰到 a-z 之间的字符则停止；③%[^=]读取字符串直到碰到=号；④%\*[^=]%s前面带\*号表示不保存变量，跳过字符串中以“=“结尾的子串而读后包括”=“在内的后面部分子串；⑤另外，可用\*匹配变量到格式字符串中，如printf("%\*d\n", a, A)即输出宽度为a的整数A，而printf ("%.\*lf \n", a, A)表示输出带a位小数的浮点数A。

**(5). 字符输入输出**

putchar(c)用于输出字符变量c。若c为整数则当ASCII码处理。

c = getchar()用于输入一个字符，包括控制字符。注意getchar函数返回的不是char类型而是int类型，此时有可能出现潜在错误。一般用法为while((a = getchar() && a!=EOF) != ‘\n’) { }，EOF是指检测文件尾，在头文件中通过宏定义定义值为-1。键盘输入EOF其实是命令行环境遗留下来的，很多系统支持仿真的EOF如用Ctrl+D（Linux系统）或Ctrl+Z（Windows系统）来模拟输入EOF。

## 分支语句

**1.if(…) {…}** 一般形式：if (表达式) {语句;}。用非 0 值表示真，用 0 表示假；if(flag) 相当于 if(flag!=0)；浮点数无法与0比较，只能用近似的值比较；例: 1e-6 等于1x10的-6次方可以看成0来使用。用于单分支选择结构;如含有交叉关系，使用并列的if语句。

**2. if(…) {…} else {…}** 一般形式： if(表达式) {语句1; } else { 语句2;}。用于双分支控制的条件语句；用于非此即彼的关系时。

**3. if(…) {…} else if(…) {…} … else if(…) {…} else {…}** 一般形式：if(表达式1) { 语句1;}else if(表达式2) {语句2;}…else if(表达式n){语句n;}else{语句n+1;}用于多分支的控制的条件语句。

**4. switch() {case …: …; case …: …; … default: …;}** 一般形式：switch(表达式){case 常量1:语句1;case 常量2:语句2;…case 常量n:语句n;default: 语句n+1;}，用于多路选择的语句。

switch语句相当于多个if-else语句；(表达式)只能是char型或者int型；case 后面至少要有一个空格，常量后面是冒号；(表达式)与常量类型要保持一致；**在需要跳出的时候，在语句后面加上break。**



## 循环语句

**while语句**while语句是控制表达式在循环体之前的循环语句，条件为真执行循环体。无限循环程序应包含break、goto、return或者导致程序终止的函数（如exit等），以在适当的时机终止程序。

**do while语句**do while语句与while语句非常相似，只不过do while语句在每次执行完循环体之后对控制表达式进行判断的，因此do while语句的循环体至少会执行一次，而while语句的循环体可能一次都不被执行，条件为真执行循环体。

**for语句**for语句的格式如下： for (表达式1; 表达式2; 表达式3） 语句

表达式1是初始化步骤，只执行一次，表达式2是控制表达式，控制循环的终止，表达式3在每次循环的最后被执行。

for语句的执行步骤是：首先执行表达式1进行初始化，然后判断表达式2的值是否为真，若为真，则执行循环体，然后执行表达式3.随后再次对表达式2的值进行判断，若为真，则再次执行循环体和表达式3，依次循环直到表达式2的值为0为止。

for语句的3个表达式都可以省略，若省略表达式1，则在执行循环前没有初始化的操作；若省略第三个表达式，则执行完循环体后，没有再需要执行的语句，循环体确保循环能够在有限时间内终止；若省略第二个表达式，则每次判断都默认为真，除非在循环体内使用break,goto和程序终止函数外，循环不会停止。

**break语句**break语句在循环体的中间设置退出点，用以跳出while，do while和for循环，直接执行循环语句后面的语句。除此之外，break语句还用于switch语句，用以跳出switch语句而直接执行switch语句后面的语句。对于嵌套的循环语句和（或）switch语句，break只能跳过其所在的那层循环或switch语句。

**continue语句**continue语句并**不跳出循环**，而是将程序执行正好转移到循环体末尾处，跳过本次循环中循环体余下的内容。continue语句只能用于循环体中。

**goto语句**可以用在程序的任何位置。与continue和break不同，它可以跳转到同一个函数中任何有标记的语句处。goto依赖于定义在语句开头的标识符，其形式为：标识符 : 语句。goto语句的形式为（goto 标识符;）。

goto易于造成程序混乱，代码难以阅读，并且其他跳转语句、return和exit语句能够满足大多数需要goto语句的情况。除非万不得已，尽量不要使用。

# 函数与程序结构【\*】

## 函数的声明与定义

**函数声明**

在C语言中，函数的定义顺序是有讲究的：默认情况下，只有在后面定义的函数中才可以调用前面定义过的函数。反过来写在标准C编译环境是不合法的，但是在GCC编译器下只是个警告。如果想把函数的定义写在main函数后面，而且main函数能正常调用这些函数，那就必须在main函数的前面进行函数的声明。

函数声明格式“返回值类型 函数名 (参数1, 参数2, ...)”，例如“int sum(int, int);”。函数声明只是证明函数的存在。而且只要知道函数名、函数的返回值、函数接收多少个参数、每个参数是什么类型的，就能够调用这个函数了，因此声明函数可以省略参数名称。如果只有函数的声明，而没有函数的定义，那么程序将会在链接时出错。这个程序是可以编译成功的，因为我们在main函数前面声明了sum函数(函数的声明和定义是两码事)，这个函数声明可以理解为：在语法上，骗一下main函数，告诉它sum函数是存在的，所以从语法的角度上main函数是可以调用sum函数的。究竟这个sum函数存不存在呢，有没有被定义呢？编译器是不管的。在编译阶段，编译器并不检测函数有没有定义，只有在链接的时候才会检测这个函数存不存在，也就是检测函数有没有被定义。

**多源文件开发**

为什么要多源文件开发？代码比较少时可以都保存在一个.c源文件中，但是，在实际开发过程中，项目做大了，源代码肯定非常多，很容易就上万行代码了，甚至上十万、百万都有可能。这个时候如果把所有的代码都写到一个.c源文件中，那么这个文件将会非常庞大，也非常恶心，不要说调试程序了，连阅读代码都非常困难。而且，公司里面都是以团队开发为主，如果多个开发人员同时修改一个源文件，那就会带来很多麻烦的问题，比如张三修改的代码很有可能会抹掉李四之前添加的代码。因此，为了模块化开发，一般会将不同的功能写到不同的.c源文件中，这样的话，每个开发人员都负责修改不同的源文件，达到分工合作的目的，能够大大提高开发效率。也就是说，一个正常的C语言项目是由多个.c源文件构成。

将子函数写到其他源文件中。在主要的源代码文件中加声明，例如“int sum(int, int);”，来骗过编译器。【不用头文件】

编译源文件，一起编译“cc -c main.c sum.c”，或者单独编译“cc -c main.c”、“cc -c sum.c”。

链接所有目标文件，一定要同时链接两个文件，不能单独链接一个文件。①如果只是链接main.o文件：cc main.o，错误信息是：在main.o中找到不到sum这个标识符，其实就是找不到sum函数的定义。因为sum函数的定义在sum.o文件中，main.o中只有sum函数的声明。②如果只是链接sum.o文件：cc sum.o，错误信息是：找不到main函数。一个C程序的入口点就是main函数，main函数定义在main.o中，sum.o中并没有定义main函数，连入口都没有，怎么能链接成功、生成可执行文件呢？③可以看出，main.o和sum.o有密不可分的关系，其实链接的目的就是将所有相关联的目标文件和C语言函数库组合在一起，生成可执行文件。cc main.o sum.o -o sum.out生成可执行文件

之后用命令./sum.out即可运行。从中也可以得出一个结论：**只要知道某个函数的声明，就可以调用这个函数，编译就能成功**。不过想要这个程序能够**运行成功，必须保证在链接的时候能找到函数的定义**。

**#include**

#include的作用：#include是C语言的预处理指令之一，所谓预处理，就是在编译之前做的处理，预处理指令一般以#开头。#include 指令后面会跟着一个文件名，预处理器发现 #include 指令后，就会根据文件名去查找文件，并把这个文件的内容包含到当前文件中。被包含文件中的文本将替换源文件中的 #include 指令，就像你把被包含文件中的全部内容拷贝到这个 #include 指令所在的位置一样。所以第一行指令的作用是将stdio.h文件里面的所有内容拷贝到第一行中。如果被包含的文件拓展名为.h，我们称之为"头文件"(Header File)，头文件可以用来声明函数，要想使用这些函数，就必须先用 #include 指令包含函数所在的头文件。#include 指令不仅仅限于.h头文件，可以包含任何编译器能识别的C/C++代码文件，包括.c、.hpp、.cpp等，甚至.txt、.abc等等都可以。如下图所示，编译链接后，程序还是可以照常运行的，因为 #include 的功能就是将文件内容完全拷贝到 #include 指令所在的位置。说明：这里用txt文件纯属演示，平时做项目不会这样做，除非吃饱了撑着，才会把代码都写到txt中去。



#include可以使用绝对路径，上面的#include "my.txt"使用的是相对路径，其实也可以使用绝对路径。比如#include "/Users/apple/Desktop/my.txt"。

**#include <>和#include ""的区别**：当被include的文件路径不是绝对路径的时候，有不同的搜索顺序。①对于使用双引号""来include文件，搜索的时候按以下顺序：先在这条include指令的父文件所在文件夹内搜索，所谓的父文件，就是这条include指令所在的文件；如果上一步找不到，则在父文件的父文件所在文件夹内搜索；如果上一步找不到，则在编译器设置的include路径内搜索；如果上一步找不到，则在系统的INCLUDE环境变量内搜索。②对于使用尖括号<>来include文件，搜索的时候按以下顺序：在编译器设置的include路径内搜索；如果上一步找不到，则在系统的INCLUDE环境变量内搜索。

**stdio.h**：stdio.h 是C语言函数库中的一个头文件，里面声明了一些常用的输入输出函数，比如往屏幕上输出内容的printf函数；注意：stdio.h里面只有printf函数的声明。前面已经提到：只要知道函数的声明，就可以调用这个函数，就能编译成功。不过想要这个程序能够运行成功，必须保证在链接的时候能找到函数的定义。其实链接除了会将所有的目标文件组合在一起，还会关联C语言的函数库，函数库中就有printf函数的定义。因此前面的程序是可以链接成功的。

**头文件.h和源文件.c的分工**：函数声明放在.h头文件中，函数定义放在.c源文件中。其实sum.h和sum.c的文件名不一样要相同，可以随便写，只要文件名是合法的。但还是建议写成一样，因为一看文件名就知道sum.h和sum.c是有联系的。

**运行步骤分析**：

1> 在编译之前，预编译器会将sum.h文件中的内容拷贝到main.c中。

2> 接着编译main.c和sum.c两个源文件，生成目标文件main.o和sum.o，这2个文件是不能被单独执行的，原因很简单：sum.o中不存在main函数，肯定不可以被执行；\* main.o中虽然有main函数，但是它在main函数中调用了一个sum函数，而sum函数的定义却存在于sum.o中，因此main.o依赖于sum.o。

3> 把main.o、sum.o链接在一起，生成可执行文件。

4> 运行程序。

**可不可以在main.c中包含sum.c文件，不要sum.h文件了？**

这么一看，语法上是绝对没有问题的，main.c、sum.c都能编译成功，分别生成sum.o、main.o文件。但是当我们同时链接main.o和sum.o时会出错。原因：当链接这两个文件时链接器会发现sum.o和main.o里面都有sum函数的定义，于是报"标识符重复"的错误，也就是说sum函数被重复定义了。默认情况下，C语言不允许两个函数的名字相同。因此，不要尝试去#include那些.c源文件。**一般不建议include源代码文件(.c)，但也不是绝对不行。**

**注意：如果源文件sum.c中没有函数调用本文件内未声明的函数，则不需要#include “sum.h”，反之则需要。所以，源文件sum.c中不是必须#include “sum.h”。在gcc编译器中，如果函数调用了本文件内未声明的函数，并未没有#include “sum.h”，虽然可以正常编译、链接、运行，但是编译过程中会给出一个警告。**

## 函数的参数

**形式参数与实际参数**

形参出现在被调函数当中，在整个函数体内都可以使用。形参在定义时编译系统并不分配存储空间，只有在调用该函数时才分配内存单元。调用结束内存单元被释放，故形参只有在函数调用时有效，调用结束时不能再使用。

实参出现在主调函数当中，当函数调用时，主调函数把实参的值传送给被调函数的形参，从而实现函数间的数据传递。传递方式有两种：值传递和地址传递方式。

**变量作为函数参数**

当形参定义为变量时，实参可以是常量、变量和表达式，这种函数间的参数传递为值传递方式。值传递的特点是参数的“单向传递”；并不会因为形参改变而影响实参。

**数组作为函数参数**

数组元素作为函数参数：数组元素又称为下标变量，它具有普通变量的一切性质，因此数组元素作为函数的实参进行数据传递是与普通变量没有任何区别，也是值传递。例如定义“int swap(int a,int b)”，调用“swap(a[0],b[0]);”。

一维数组名作为函数参数：数组名是一个地址，是数组的首地址，因此用数组名作为函数的参数进行数据传递时，执行的是地址传递方式。所谓地址传递，顾名思义实参传递的不是数据本身，而是数据存在的地址。函数调用时，把数组名即数组的首地址作为实参传递给形参（必须是可接受地址的数组名或者指针变量），形参数组名取得首地址后就有了实在的数组，这时实质上实参和形参是同一个数组，指向同一段存储空间，实参的改变就是对形参的改变，所以传址方式可看成是数据进行了“双向传递”。

数组指针（即数组元素的地址）作为函数参数：由于数组元素的地址的本质仍然为地址，所以属于地址传递方式。例如定义“int swap(int \*a,int \*b)”，调用“int arr[] = {1,2}; int \*a = &arr[0]; int \*b = &arr[1]; swap(a,b);”。

数组元素（下标变量）作为函数的参数进行的数据传递是值传递方式，数组名（数组首地址）、数组元素的地址（&arr[0]）作为函数参数进行的数据传递是地址传递方式。

实参为数组名时，形参接收时可以有三种形式：带下标的数组名(arr[0])。不带下标的数组名（arr）、可接收地址值的指针变量名（\*a）。由于是参数组和形参数组都指向同一段内存单元，故它们操作的是同一批数据，所以形参的改变就是改变了实参中的数据。

**参数传递的是类型后面的部分，(int \*)算是类型名。例如int sum(int a,int \*b)，参数传递时是a=XXX，b=XXX。不是\*b=XXX。**

## 函数的返回值

没有返回值的函数为空类型，用void表示。一旦函数的返回值类型被定义为 void，就不能再接收它的值了。为了使程序有良好的可读性并减少出错， 凡不要求返回值的函数都应定义为 void 类型。

return 语句可以有多个，可以出现在函数体的任意位置，但是每次调用函数只能有一个 return 语句被执行，所以只有一个返回值（少数的编程语言支持多个返回值，例如Go语言）。

函数一旦遇到 return 语句就立即返回，后面的所有语句都不会被执行到了。从这个角度看，return 语句还有强制结束函数执行的作用。

return 语句是提前结束函数的唯一办法。return 后面可以跟一份数据，表示将这份数据返回到函数外面；return 后面也可以不跟任何数据，表示什么也不返回，仅仅用来结束函数。无返回值类型(void)函数中return 后面不能带任何数据，直接写分号即可。

## 函数的调用

**标准库函数**

分为：I/O函数，字符串，字符处理函数，数学函数，接口函数，时间转换和操作函数，动态地址分配函数，目录函数，过程控制函数，字符屏幕和图形功能函数。

这些库函数在不同的头文件中声明。比如：①math.h头文件中有：sin(x),cos(x),exp(x)（求e^x），fabs(x)（求x的绝对值）等库函数。②stdio.h头文件中有：scanf(), printf(), gets(), puts(), getchar(), putchar()等库函数。③string.h头文件中有：strcmp(),strcpy(),strcat(),strlen等库函数。

**函数的定义**

无参数：“类型名 函数名（）{ 函数体 }” 或者“类型名 函数名（void）{ 函数体 }”

有参数：“类型名 函数名（形式参数）{ 函数体 }”

空函数：“类型名 函数名（）{ }”

**函数调用时的参数传递**

函数间通过参数来传递数据，即通过主调函数中的实际参数（实参）向被调用函数中的形式参数（形参）进行传递。实参向形参传递数据的方式：实参将值单向传递给形参，形参的变化不影响实参值。

分析：值传递是单向传递，形参数据变化，而实参保持原数据不变。

在函数调用时，形参被临时分配相应的内存，调用结束后，形参单元被释放，而实参单元保留并维持原值。

实参是表达式，负责向对应的形参标识的内存单元传递数据，实参向形参的数据传递是“值传递”。

实参与形参必须个数相，对应的形参和实参的类型必须一致。

**函数调用的返回值**

1. 函数的返回值通过函数中的return语句获得。如果需要从调用函数带回一个函数值（供主函数使用），被调函数中需包含return语句。
2. 在定义函数时要指定函数值的类型，函数类型决定返回值的类型。

**函数调用的嵌套**

定义函数时不能定义另一个函数，但是可以进行嵌套调用函数。主函数调用一级子函数，调用次级子函数……

**递归的概念**

函数的递归调用是指：一个函数在他的函数体内直接或间接地调用它自身。分为：直接递归（函数直接调用自身）和间接递归（函数通过其他函数调用自身）。可分为“回溯”和“递推”两个阶段。

**递归的一般形式**

反回值类型 递归函数名（参数说明表）

{

if(递归终止条件)

返回值=递归终止值；

else

返回值=递归调用（...）的表达式；

return 返回值；

}

**函数递归举例**

用递归求n的阶乘、汉诺塔问题等。

分析：将n个盘子从A座上移到C座上需要三步：

（1）将A上的n-1个盘子借助C座先移到B座上；

（2）把A座上剩下的一个盘子移到C座上；

（3）将n-1个盘从B座上借助于A移到C座上。

## 函数的递归

**函数递归优缺点**

程序调用自身的行为就是递归。可以直接或间接的调用，本质是把复杂的问题转化为一个规模小的问题。递归一般只需少量的代码就可描绘出多次重复计算。其主要思考方式在于大事化小。

优点是为具有某些特征的编程问题提供了最简单的策略，缺点是层层调用，算法的复杂度可能过高，以致于快速耗干了计算机的内存资源，不方便阅读和维护等。

**递归使用场景**

能够要求转化为新的问题，且二者解决方法相同，所处理的对象存在规律变化；非递归比较麻烦，而递归很简单；有模板或是公式可以直接套用，不会出现明显问题。

**递归的必要条件**

明确存在限制条件，每次递归越来越逼近条件。

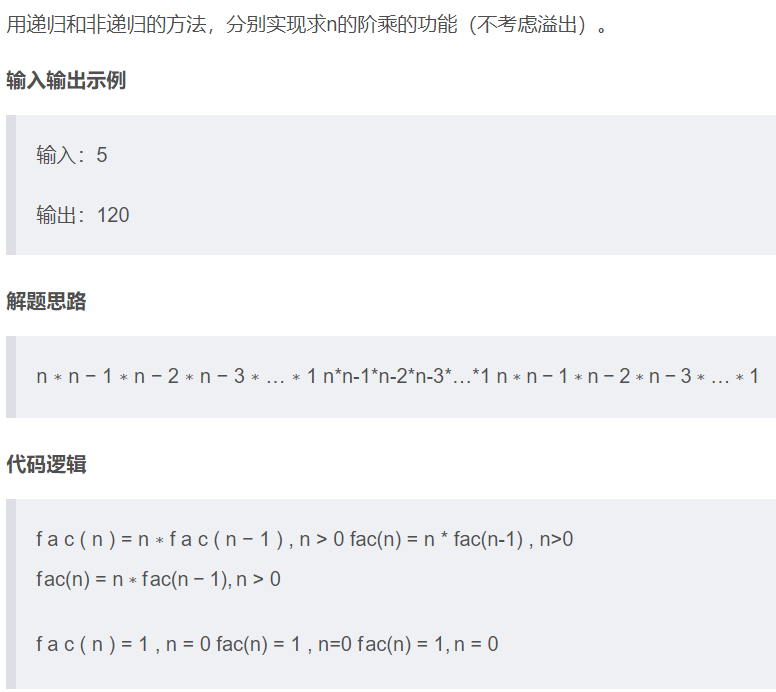
**递归的细节说明**

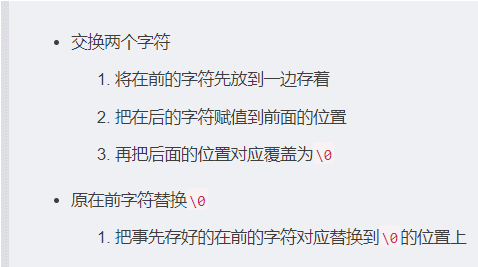
①每级递归都有自己的变量，可能名称相同，但是其值不同。（递归调用时，系统自动保留当前函数的参数变量。每次调用系统都会为函数开辟相应的空间）②次调用都要返回值，递归执行结束后，控制权传回到上一级函数。调用结束后，系统释放本次调用所开辟的空间，程序返回到上一次的调用点，同时获得初进该级调用的参数。每级递归必须逐级返回，不可跳跃或间断。③函数中递归语句之前的代码，按被调函数的顺序执行，递归之后的代码，与被调函数相反的顺序执行。

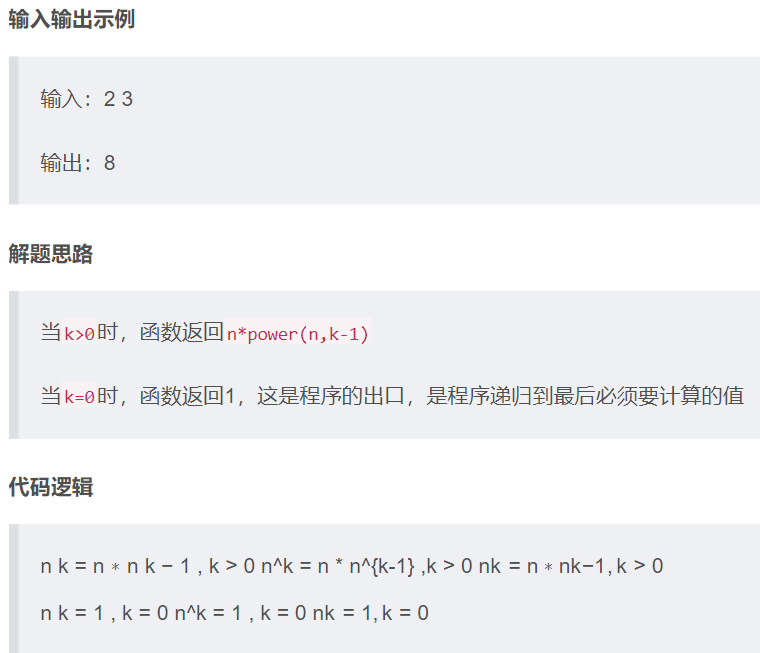
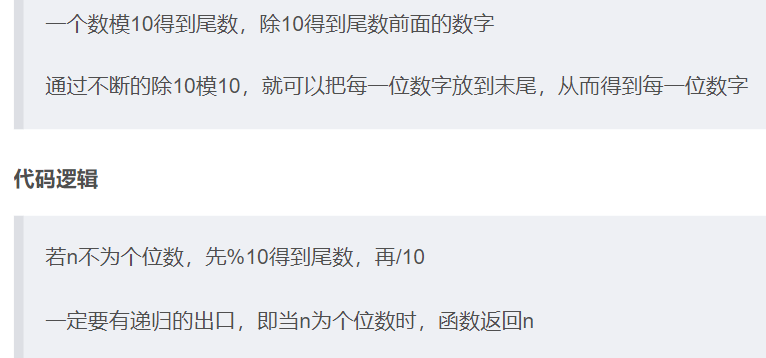
**递归例子说明**

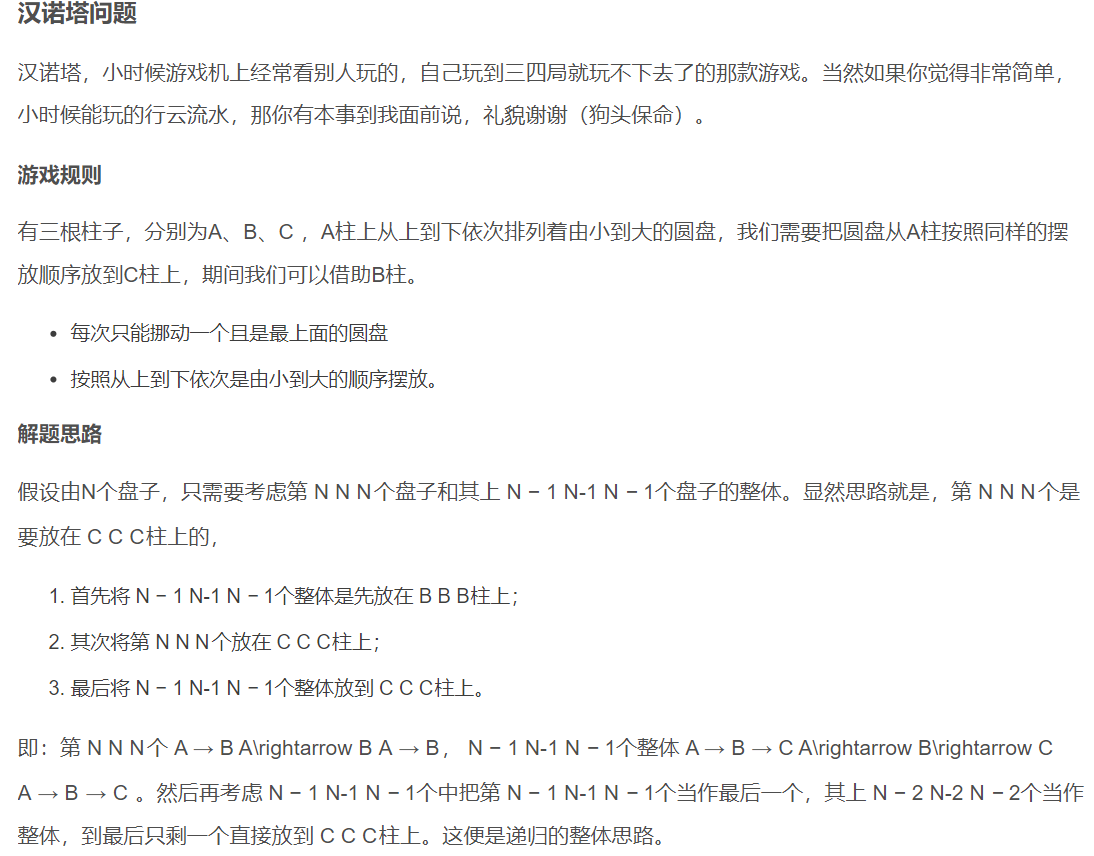
打印整数每一位，n的阶乘，strlen函数模拟，逆序字符串，

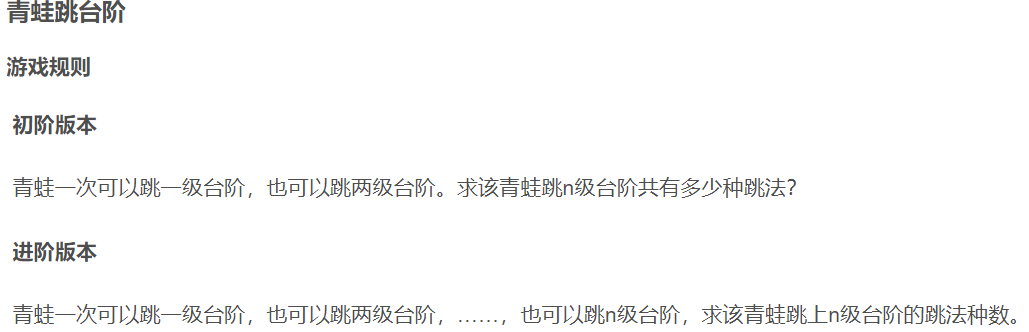
数字各位之和，n的k次幂，斐波那契数列













## 局部变量和全局变量——作用域

**局部变量**

1. 函数的开头定义，在本句以下的函数内有效；

2. 函数内的复合语句内定义，只能在本复合语句内切本行以下有效；

3. 形式参数，只在此函数有效；

4. 函数中间（非开头）定义，只能在本行以下的函数内有效；

注意：程序执行到某个函数时，这个函数内部的局部变量将会被分配内存空间；局部变量在函数执行结束后，变量所占内存将会被释放。

**全局变量**

1. 源文件开头定义，在此行以下的所有定义的函数都有效；

2. 函数前与函数后，此全局变量在此行之前是无效，只能在此之下的函数中有效；

3. 文件结尾，程序结尾的全局变量是没有意义的；

注意：为了区别全局变量和局部变量，往往大家在写程序的时候都喜欢将全局变量的首字母大写，而局部变量的首字母小写。

**全局变量的优点和缺点**

优点：C语言的函数，每次最多只能返回一个值，但是如果定义了全局变量，那么在这个变量的有效范围内，很多函数都能改变这个变量的值，所以增加了函数之间的联系，通过函数的调用可以得到一个或一个以上的值。

缺点：（大量使用全局变量的情况下）①占内存：全局变量所占的内存空间不会像局部变量一样会被释放；②降低程序清晰性：无法随时确定定义的全局变量的值的大小；③降低通用性：程序设计时要求函数的“内聚性”强，函数与函数之间“耦合性”弱；定义全局变是一定要注意在有效范围内变量不能重名，并且当全局变量被跨文件调用的函数调用时，不能出现全局变量与所跨文件中存在重名变量，否则有可能会出错；所以，为了提高程序的可靠性，可移植性和可读性等，全局变量尽量少用。

## 头文件

**#include的作用**

在include的地方，把头文件里的内容原封不动的复制到引用该头文件的地方。

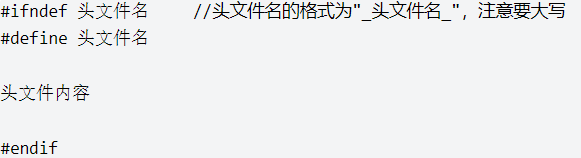
**头文件的作用**

头文件引用有两种形式：#include < stdio.h> 和 include "main.h “。用< >引用的一般是编译器提供的头文件，编译时会在指定的目录中去查找头文件。具体是哪个目录，编译器知道，我们不用关心。用” “引用的一般是自己写的头文件，编译时，编译器会在项目所在的文件夹中进行查找，如果还才存在子文件夹，则在Makefile中用-I（大写i）来指定头文件搜索目录（库的路径）。总结：系统提供的头文件用< >引用， 自己写的用” "引用。

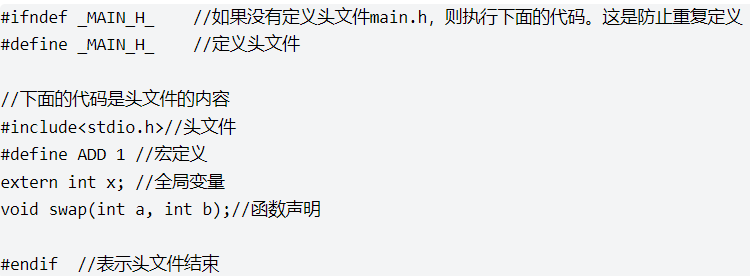
**头文件的内容**

**头文件里一般包括宏定义，全局变量，函数原型声明**。

**头文件的格式**



**示例代码:头文件main.h**

****

**include包含.c源文件**

include可以包含.c源文件，在某些工程里可以看到，但这样的做法不常见也不推荐。

include关键字包含.c源文件和.h头文件，意义都是一样的，在原地将引用的文件展开。

## 内部函数和外部函数

C语言中，根据函数能否被其他源文件调用，分为内部函数（可以被其他源文件调用的函数）和外部函数 （只在定义的文件中有效）。

**外部函数——extern**

开发大型项目，可能包含很多源文件来分别实现，最终，再整合在一起，有时，一个源文件中，需要调用其他源文件中的函数。调用外部函数之前，需要在**当前源文件中**定义外部函数，定义外部函数的方式是在函数的返回值类型前面添加extern关键字。例如，extern int add(int x,int y);编译器通过extern关键字会知道，add()函数是定义在其他文件中的外部函数。真正定义外部函数时，可以省略关键字extern。

由函数的返回类型、函数名和参数列表组成，这类格式的代码被称为函数原型（声明）。当代码中包含函数原型时，可能有两种情况：①程序员希望编译器自动从其他文件中，查找该函数的定义，②程序员先定义未实现的空函数，然后，在其他文件中具体实现。

**注意：声明外部函数时，无论有没有关键字extern，外部函数与原函数定义的返回值类型、函数名称和参数列表必须一致。**

**内部函数——static(也称为静态函数)**

当多人开发时，可能出现函数重名的情况，不同源文件中的同名函数会相互干扰。此时，就需要一些特殊函数，只在定义的文件中有效，这类函数称为内部函数。

定义内部函数：在定义内部函数时，需要在函数的返回值类型前面添加static关键字，也称静态函数。

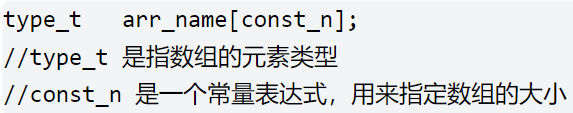
函数调用时如果出现函数名冲突，优先调用内部函数。如果没声明内部函数的两个函数同名，将出现BUG。

**注意：如果已经#include了一个文件，编译过程第一步预处理已经把该文件内容已经拷贝到当前文件，再用静态static标识也就没用了**。

# 数组

## 数组简介

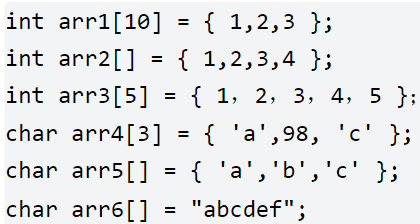
**一维数组的创建** 数组创建，C99标准之前[ ]中要给一个**常量**才可以，不能使用变量。



**一维数组的初始化**

数组的初始化是指，在创建数组的同时给数组的内容一些合理初始值（初始化）。

数组在创建的时候如果想不指定数组的确定的大小就得初始化。数组的元素个数根据初始化的内容来确定。但是对于下面的代码要区分，内存中如何分配。

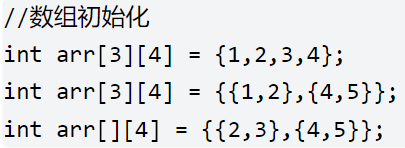


**一维数组的使用：**数组是使用下标来访问的，下标是从0开始。数组的大小可以通过sizeof(arr)计算得到。

**一维数组在内存中的存储：**随着数组下标的增长，元素的地址，也在有规律的递增。由此可以得出结论：数组在内存中是连续存放的。

**二维数组的创建：**type\_t arr\_name[const\_m] [const\_n];

**二维数组的初始化**

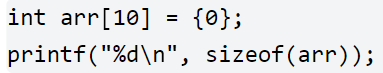


**二维数组的使用：**也是通过下标

**二维数组在内存中的存储：**二维数组在内存中也是连续存储的

**数组越界**

数组的下标是有范围限制的。数组的下规定是从0开始的，如果输入有n个元素，最后一个元素的下标就是n-1。所以数组的下标如果小于0，或者大于n-1，就是数组越界访问了，超出了数组合法空间的访问。C语言本身是不做数组下标的越界检查，编译器也不一定报错，但是编译器不报错，并不意味着程序就是正确的，所以程序员写代码时，最好自己做越界的检查。二维数组的行和列也可能存在越界。



输出结果是40，sizeof(数组名)，计算整个数组的大小，一个int占4。

arr = &arr[0] 都是数组地址，也就是数组首元素地址。

\*arr = arr[0] 都是数组中存放的第一个元素的值

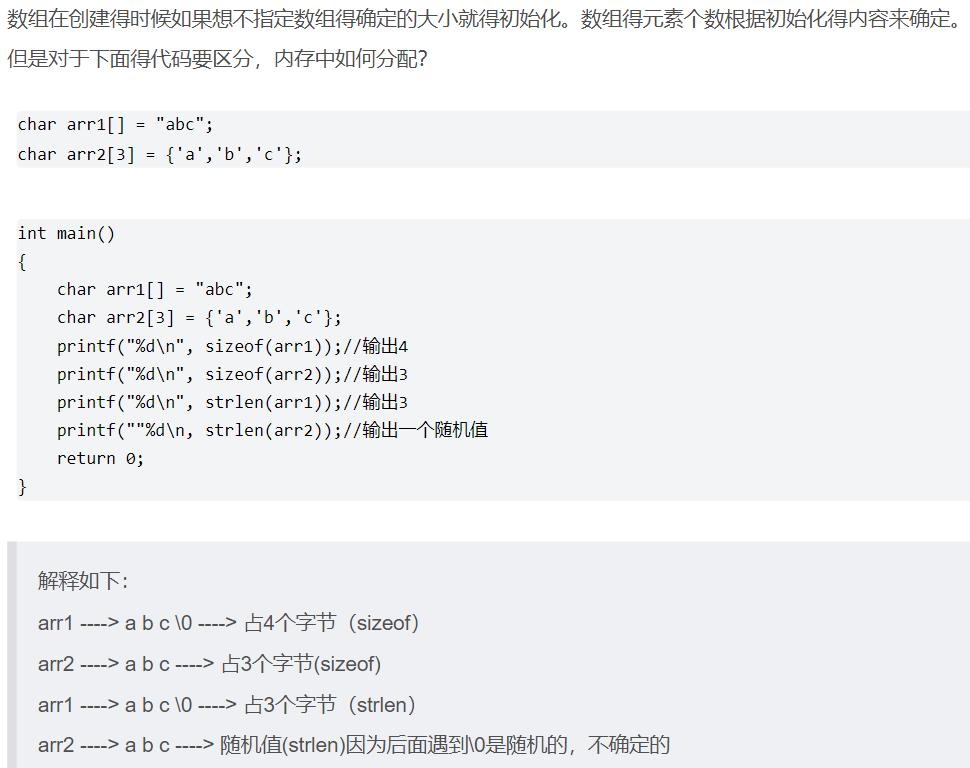
## 一维数组

int arr[10] = {1,2,3}是不完全初始化，剩下的元素默认初始化为0。

数组在创建的时候如果想不指定数组得确定的大小就得初始化。数组得元素个数根据初始化得内容来确定。但是对于下面得代码要区分，内存中如何分配？

char arr1[ ]=”abc”；给数组arr1分配了4个空间，默认在arr1[3]添加结束符“\0”。

char arr2[ ]={‘a’,’b’,’c’}；则从存储器分配了3个空间给arr2，所以不知道何时才能遇到结束符“\0”。而strlen（）是检测字符串长度的函数，遇到结束符“\0”才会停止，所以随机。



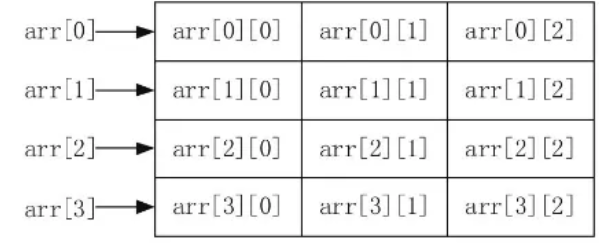
strlen 和 sizeof 没有什么关联

strlen 是求字符串长度的 - 只能针对字符串求长度 - 库函数 - 使用得引用头文件

sizeof 计算变量、数组、类型得大小 - 单位是字节 -操作符

## 二维数组

与一维数组的定义唯一的不同是多了一个常量表达式2。其中，常量表达式1为第一维的长度（行），常量表达式2为第二维的长度（列）。通常在处理二维数组的时候，为了便于理解，都将数组视为一个矩阵，常量表达式1表示矩阵的行数，而常量表达式2表示矩阵的列数。与一维数组一样，在定义二维数组时，常量表达式同样不能为变量。**前行后列。**



数组中各个元素在矩阵中对应的位置由二维数组的两个下标决定。我们可以将定义的二维数组int arr[4][3]拆分理解，原先有个arr[3]的一维数组，把这种一维数组作为新的元素，4个这种元素组成了新的数组。即，一维数组ARR[4]放着四个元素arr[3]。

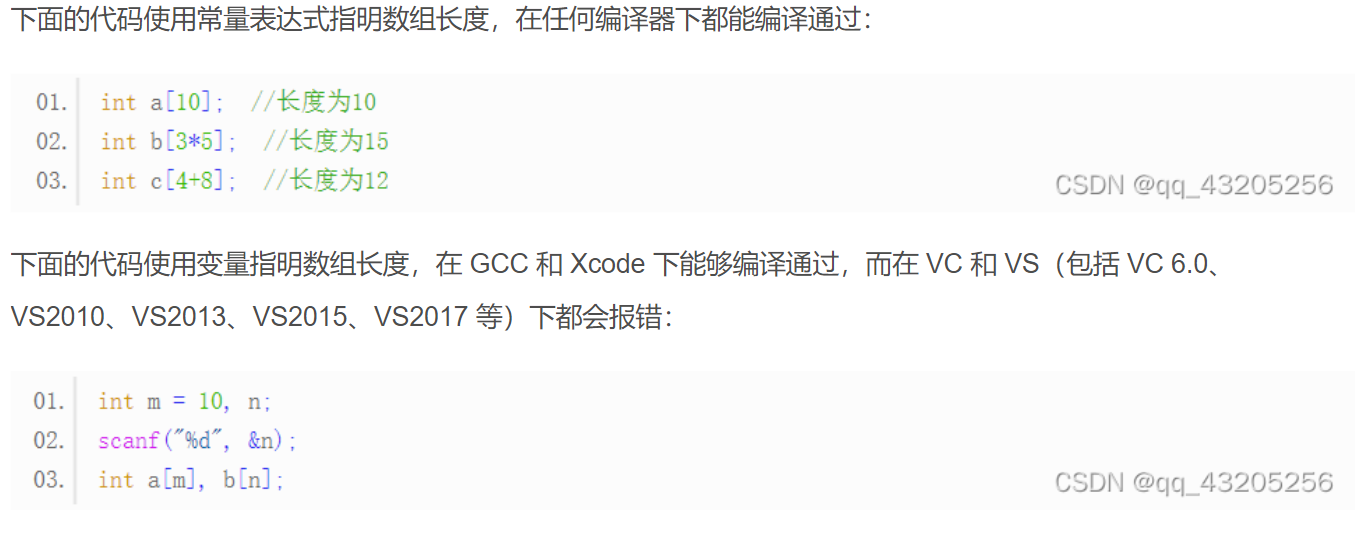
通过上述二维数组在内存中的存储结构图可以发现，二维数组中的所有元素都存储在一片连续的内存单元中，所占用的内存大小为元素类型所占用的内存大小乘以第一维及第二维的长度。如果以矩阵的方式来分析二维数组的存储方式，那么先从矩阵第一行从左往右依次存储完所有元素，然后按照同样的方法存储第二行的所有元素，直到存储完所有数组元素为止。

## 变长数组

在《C语言的三套标准：C89、C99和C11》一节中我们讲到，目前经常使用的C语言有三个版本，分别是 C89、C99 和 C11。C89（也称 ANSI C）是较早的版本，也是最经典的版本，国内大学几乎都是以该版本为基础进行授课。C99 和 C11 是后来对 C89 的升级，增添了一些新内容（不多），语法更加灵活了，同时兼容 C89。

各种编译器都能很好地支持 C89 标准，但对 C99 的支持却不同：开源组织的 GCC 和 Xcode 使用的 LLVM/Clang 已经支持了大部分（几乎全部）的 C99 标准，而微软的 VC、VS 对 C99 却不感兴趣，直到后来的 VS2013、VS2015、VS2017 才慢慢支持，而且支持得还不好。

C89 和 C99 对数组做出了不同的规定：在 C89 中，必须使用常量表达式指明数组长度；也就是说，数组长度中不能包含变量，不管该变量有没有初始化。而在 C99 中，可以使用变量指明数组长度。





变量的值在编译期间并不能确定，只有等到程序运行后，根据计算结果才能知道它的值到底是什么，所以数组长度中一旦包含了变量，那么数组长度在编译期间就不能确定了，也就不能为数组分配内存了，只有等到程序运行后，得到了变量的值，确定了具体的长度，才能给数组分配内存，我们将这样的数组称为**变长数组**(VLA, Variable Length Array)。

普通数组（固定长度的数组）是在编译期间分配内存的，而变长数组是在运行期间分配内存的。**变长数组仍然是静态数组。**注意，变长数组是说数组的长度在**定义之前**可以改变，一旦定义了，就不能再改变了，所以变长数组的容量也是不能扩大或缩小的，它仍然是静态数组。以上面的代码为例，第 8 行代码是数组定义，此时就确定了数组的长度，在此之前长度可以随意改变，在此之后长度就固定了。从你定义数组的那一刻起（也就是第二行代码），数组的长度就确定下来了，以后再也不会改变了；改变 n 的值并不会影响数组长度，它们之间没有任何“联动”关系。用 scanf() 读入 n 的值，影响的也仅仅是 n 本身，不会影响数组。在定义数组之前就输入 n 的值，这样输入的值才有用武之地。

**变长数组不能在定义的时候进行初始化；变长数组必须是自动存储类别，即不能使用extern或static存储类别说明符。**

# 指针【\*】

## 指针与地址

同C语言中其他变量一样，把指针也看成是一种变量即可，这种变量专门存储地址值。变量和指针的类型必须相同。如：int a=100; int\* b=&a;。

C程序在调用函数时使用“**按值调用**”，这意味着变量的值直接从调用函数复制到被调函数的实参中，“按值调用”是C语言中调用函数的唯一方式。**值传递**传过去的是变量数值，被调函数会创建形参来存储该数值，形参改变实参不变，变量数值也不变。**地址传递**传过去的是变量的地址，被调函数会创建形参来存储该地址，从而可以改变该地址对应的变量，故变量数值改变。

指针变量声明：int\* iptr; char\* cptr; double\* dptr;

指针变量先声明再初始化：

int a = 100; char b = 'o'; double c = 100.5;

int\* iptr; char\* cptr; double\* dptr;

iptr = &a; cptr = &b; dptr = &c;

指针变量直接声明并初始化：

int a = 100; int\* iptr = &a; int b = \*iptr;

**注：Int\* a,\*b;同时声明两个int指针，而int\* a,b;声明了指针\*a和变量b。**

一个函数只能访问自己的栈帧。但是，通过指针，一个函数可以间接地读写另一个栈帧中的变量值。值得注意的是，函数之间利用指针读写是有方向的。如果函数F1调用F2，则F2可以读写F1中的变量值，反之不行。很好理解，F2得栈帧先出栈，出栈后变量都没有了，F1还读写什么呢？

## 指针与函数参数

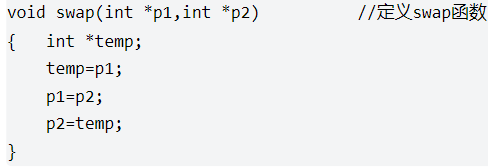
指针变量作为函数参数：函数的参数不仅可以是整型、浮点型、字符型等数据，还可以是指针类型。它的作用是将一个变量的地址传送到另一个函数中。



在函数调用时，将实参变量的值传送给形参变量，因此 p1 的值为 &a，p2 的值为 &b。这时，p1 和pointer\_1 都指向变量 a，p2 和 pointer\_2 都指向变量 b，执行 swap函数体后，\*p1 和 \*p2 的值互换，相当于 a 与 b 的值互换。函数调用结束后，形参 p1 和 p2 不复存在 (被释放)，此时，a 与 b 已经是交换后的值。

如果swap函数写成int \*temp; \*temp=\*p1; \*p1=\*p2; \*p2=\*temp;就会出问题。因为没有对\*temp指针初始化，因此temp中的值是不可预见的，所以 temp 所指向的单元也是不可预见的。所以，对 \*temp 赋值就是向一个未知的存储单元赋值，而这个存储单元可能存在着一个有用的数据，这样就可能破坏系统的正常工作状况。

**不能企图通过改变指针形参的值而使指针实参的值改变。**



如果swap函数如上图所示，该操作只是把p1和p2指针的值对调（即原先p1和pointer\_1指向a，现在p1指向b但是pointer\_1指向a，且现在变量a的值依旧不变；p2同理）。该操作并没有改变pointer\_1与pointer\_2指针以及a、b变量。

C语言中实参变量和形参变量之间的数据传递是单向的值传递方式。指针变量做函数参数同样要遵循这一规则。不可能通过执行调用函数来改变实参指针变量的值，但是可以改变实参指针变量所指变量的值。

 函数的调用可以(而且只可以)得到得到一个返回值，而使用指针变量做参数，可以得到多个变化了的值。

## 指针与数组

**通过指针引用数组元素**引用一个数组元素，可以用以下两种方法：①下标法，如 a[i] 形式；②指针法，如 \*(a+i)或 \*(p+i)。其中 a 是数组名，p是指向数组元素的指针变量，其初值为 p=a。

具体：①int arr[5]，输出a[i]；②int arr[5]，输出\*(arr+i)；③int arr[i],\*p，且p=arr输出\*(p+i)；④int arr[i],\*p，且p=&arr[0]输出\*(p+i)；⑤int arr[i],\*p，且p=arr输出\*(p++)。

**注：要时刻注意指针的位置。**

## 地址算数运算

当C语言中的指针执行数组元素时，C语言允许对指针进行**算术运算**。

C语言支持**3种**格式的指针算术运算，包括指针加上整数，指针减去整数和两个指针相减**：①**指针p加上整数j表示指针向后移动j个单位（每个单位大小为指针p的类型所占的字节数），指向p原先指向的元素后的第j个元素。若p指向数组a[i],则p+j指向a[i+j]。**②**指针p减去整数j表示指针向前移动j个单位，指向p原先指向的元素前的第j个元素。若p指向数组a[i],则p-j指向a[i-j]。**③**两个指针相减，得到的是指针之间元素的个数。若指针p指向a[i]，指针q指向a[j],则p-q等于i-j。

还可以用关系运算符进行指针比较。指针的比较依赖于指针所指向的两个元素的相对位置。若指针p指向a[i]，指针q指向a[j],p和q的比较结果由i与j的大小决定。

因此适用于指针运算的运算符包括算术运算符+、-，赋值运算符和复合赋值运算符（=，+=,-=，++，--)和所有的关系运算符。



**指针和数组**

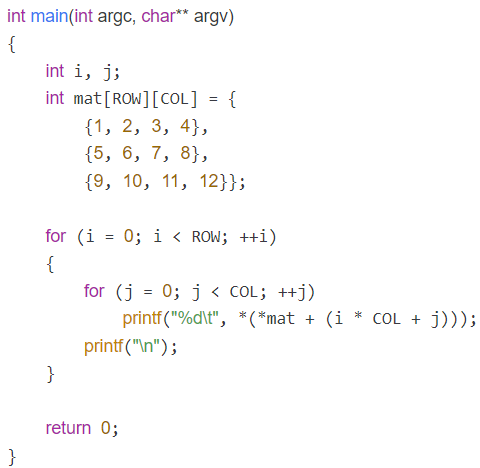
**可以用数组的名字作为指向数组第一个元素的指针。但数组名是指针常量，不能赋值。例如int arr[5]，可以写\*arr=0等价于arr[0]=5，不能arr++。**

**同样，指针也可以作为数组名一样使用，指针p的表达式p[i]等价于\*(p+i)。**

**函数中数组形式的形式参数传入的都是指针，为了明确指针形式，可以将数组型形式参数改为指针，两者是完全一样的。**

多维数组的元素在内存中是依序存储的，获得第一个元素的指针，就可以通过重复自增指针的方式访问到每一个元素。

C语言中n维数组是被当成一个一维数组，每个数组元素是一个n-1维数组，依次类推。因此对于数组int a[N1][N2]...[Nm]来说，a是一个一维数组的指针，其元素是一个n维数组，因此a指针增减的移动单位是n维数组，a[N1]是一个一维数组的指针，其元素是一个n-1维数组，因此a[N1]指针增减的移动单位是n-1维数组，依次类推，a[N1][N2]... [Nm-1]是一个一维数组的指针，其指针增加的移动单位是一个int型长度，指针指向的是int型的数组元素。【这部分讲的不是很明白，可以先往后看】

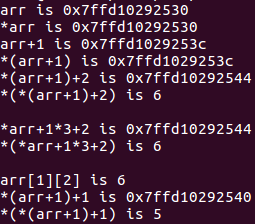


如上图所示，mat可以理解为一个一维数组的指针，其元素是一个一维数组。\*mat也是一个一维数组指针，其元素是一个int数值。

数组表示mat[i][j]可以表示第i行j列的元素。

指针表示\*(\*(mat+i)+j)表示： &mat是指向二维数组的指针，mat是指向二维数中第0个一维向量的指针，(mat+i)是指向二维数组第i个一维向量的指针，\*(mat+i)是指向第i个一维向量中第0个元素的指针，\*(mat+i)+j是指向第i个一维向量中第j个元素的指针。**\*(\*(mat+i)+j)是第i个一维数组中第j个元素**。

指针表示\*(\*mat+(i\*col+j))表示：&mat是指向二维数组的指针，mat是指向二维数中第0个一维向量的指针， \*mat是指向第0个一维向量中第0个元素的指针，(\*mat+(i\*col+j)) 是指向第0个一维向量中第i\*col+j个元素的指针。**\*(\*mat+(i\*col+j))是第0个一维向量中的第i\*col+j个元素。**

****

**总结：不带\*是指向二维数组某个一维向量的指针，带一个\*是指向某个元素的指针，只有带两个\*才是具体元素。**

## 指针数组以及指向指针的指针 【\*】

**数组**

指针也是一种数据类型，指针的步长就看他指向内存空间的类型int a[10]; //其中a代表的是数组首元素的地址。a+1 步长 是4，但是&a代表整个数组的地址，内存空间加1就是地址偏移40，所以&a+1 步长 40 ；从数值上看a和&a数值是一样的。

**为什么需要指针?**

指针解决了一些编程中基本的问题。

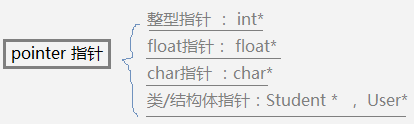
第一，指针的使用使得不同区域的代码可以轻易的共享内存数据。当然你也可以通过数据的复制达到相同的效果，但是这样往往效率不太好，因为诸如结构体等大型数据，占用的字节数多，复制很消耗性能。但使用指针就可以很好的避免这个问题，因为任何类型的指针占用的字节数都是一样的（根据平台不同，有4字节或者8字节或者其他可能）。

第二，指针使得一些复杂的链接性的数据结构的构建成为可能，比如链表，链式二叉树等等。

第三，有些操作必须使用指针。如操作申请堆内存。还有：C语言中的一切函数调用中，值传递都是“按值传递”的，如果我们要在函数中修改被传递过来的对象，就必须通过这个对象的指针来完成。

**指针是什么？**

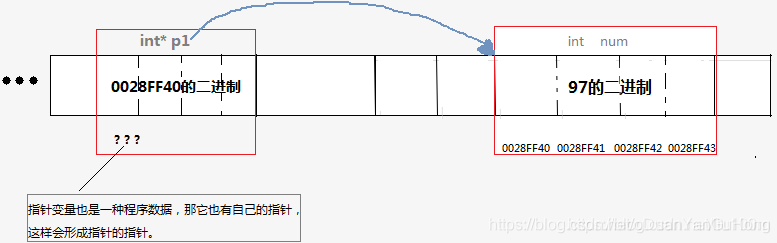
指针是程序数据在内存中的地址，而指针变量是用来保存这些地址的变量。



由于内存中的每一个字节都有一个唯一的编号，因此，在程序中使用的变量，常量，甚至数函数等数据，当他们被载入到内存中后，都有自己唯一的一个编号，这个编号就是这个数据的地址。指针就是这样形成的。char占一个字节，int占4个字节。

**指针与变量的关系**

用来保存指针的变量，就是指针变量。如果指针变量p1保存了变量 num的地址，则就说：p1指向了变量num，也可以说p1指向了num所在的内存块 ，这种指向关系，在图中一般用箭头表示。



指针的值：很好理解，如上面的num 变量 ，其地址的值就是0028FF40 ，因此 p1的值就是0028FF40。数据的地址，用于在内存中定位和标识这个数据，因为任何2个内存不重叠的不同数据的地址都是不同的。

指针的类型：指针的类型决定了这个指针指向的内存的字节数并如何解释这些字节信息。一般指针变量的类型要和它指向的数据的类型匹配。

由于num的地址是0028FF40，因此p1 和 p2的值都是0028FF40

int \*p1 : 将从地址0028FF40 开始解析，因为p1是int类型指针，int占4字节，因此向后连续取4个字节，并将这4个字节的二进制数据解析为一个整数 97。

char \*p2 : 将从地址0028FF40 开始解析，因为p2是char类型指针，char占1字节，因此向后连续取1个字节，并将这1个字节的二进制数据解析为一个字符，即’a’。

同样的地址，因为指针的类型不同，对它指向的内存的解释就不同，得到的就是不同的数据。

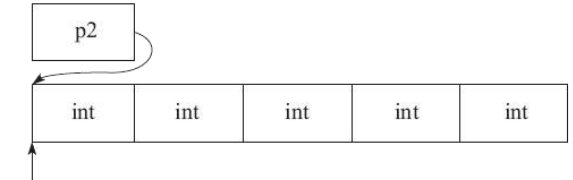
**数组指针和指针数组**

首先，需要明确一个优先级顺序：()>[]>\*。

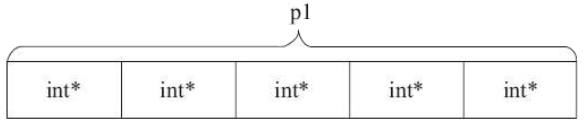
数组指针的概念，数组的指针：是一个指针，指向数组的指针。（\*p)[n]：根据优先级，先看括号内，则p是一个指针，这个指针指向一个一维数组，数组长度为n，这是“数组的指针”，即数组指针。

指针数组的概念，即指针的数组：是一个数组，装着指针的数组。\*p[n]：根据优先级，先看[]，则p是一个数组，再结合，这个数组的元素是指针类型，共n个元素，这是“指针的数组”，即指针数组。

根据上面两个分析，可以看出，p是什么，则词组的中心词就是什么，即数组“指针”和指针“数组”。



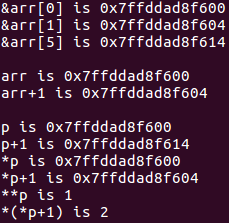
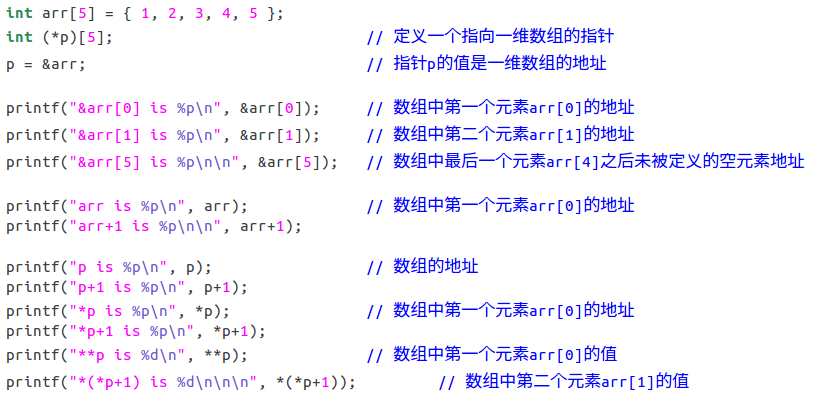
数组指针的结构int (\*p2)[5]；对于语句int (\* p2)[5]，“()”的优先级比“[]”高，“\*”号和 p2 构成一个指针的定义，指针变量名为 p2，而 int 修饰的是数组的内容，即数组的每个元素。也就是说，p2 是一个指针，它指向一个包含 5 个 int 类型数据的数组，如图 2 所示。很显然，它是一个数组指针，数组在这里并没有名字，是个匿名数组。

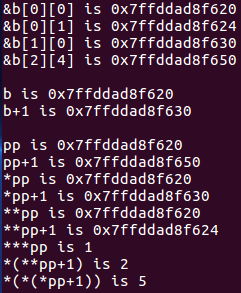


指针数组的结构 int \*p1[5]；对于语句int\* p1[5]，因为“[]”的优先级要比 \* 要高，所以 p1 先与“[]”结合，构成一个数组的定义，数组名为 p1，而“int\*”修饰的是数组的内容，即数组的每个元素。也就是说，该数组包含 5 个指向 int 类型数据的指针，如图 1 所示，因此，它是一个指针数组。

总结：①由此可见，对数组指针来说，首先它是一个指针，它指向一个数组，也就是说它是指向数组的指针，在 32 位系统下永远占 4 字节，至于它指向的数组占多少字节，这个不能够确定，要看具体情况；②而对指针数组来说，首先它是一个数组，数组的元素都是指针，也就是说该数组存储的是指针，数组占多少个字节由数组本身决定。

数组指针的定义与初始化：int a[5] = { 1, 2, 3, 4, 5 }；int (\*p)[5]；p = &a；





**注：数组指针本质上是个指针，不是数组，所以p[0]就是p，p[1]、p[2]等都是没意义的。**

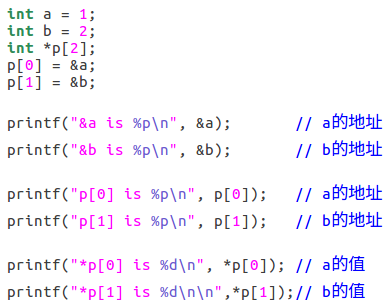
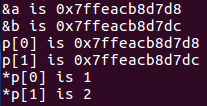
**注：数组名代表的是低一级的地址，一维数组数组名arr代表数组中第一个元素的地址，&arr代表一维数组的地址，二者值相同；二维数组的数组名b代表二维数组中第一个一维向量的地址，&b代表二维数组的地址。**

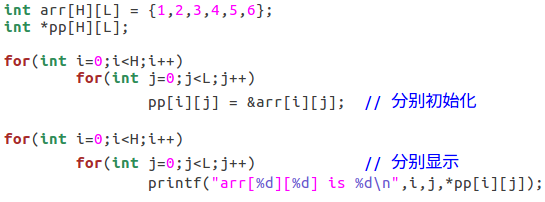
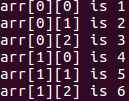
**注：创建数组指针（指向数组的指针），类型要和数组相同，指针值是数组地址。类比于变量指针，指针值是变量地址。**

**注：指针类型决定指针移动步长，和指针值无关。不同类型的指针，指针值可能相同，但指针移动步长不同。**

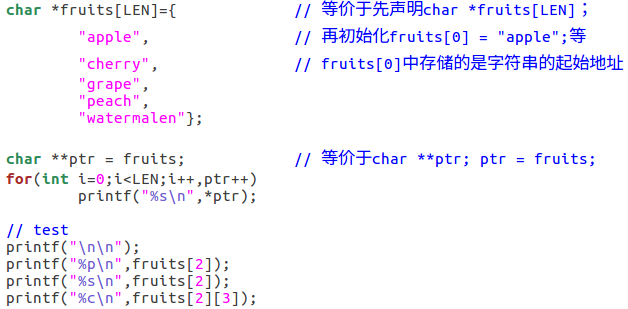
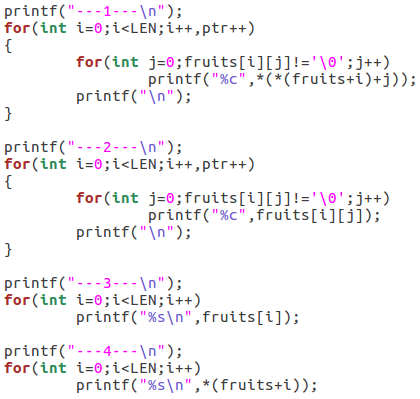
在C语言几乎所有使用数组的表达式中，数组名的值也就是数组第一个元素的地址【低一级】，他是个指针常量可以参与运算大不能被赋值，类型取决于数组元素的类型。

指针数组的定义与初始化：int a = 1；int b = 2；int \*p[2]；p[0] = &a；p[1] = &b；指针数组中的每个元素要单独初始化。

下面看一种新鲜的写法：下面五种方法均可显示。

将字符串看作字符的数组。

ptr是指向指针的指针, ptr-->指针-->字符，该指针也是字符串首地址。所以fruits也是指向指针的指针，本质上等价于ptr【ptr = fruits】。

fruits[]是指针数组，所以fruits[i]是指针也是字符串首地址，等价于&fruits[i][0]。 &fruits[i][j]是某个字符的地址，fruits[i][j]是字符。

## 指向函数的指针【\*】

**函数指针类型的通用形式:** 返回类型 (\* 可选限定符 函数指针名) (形参列表)。

**声明指向函数的指针：**①void print(int num)，声明函数；②void (\*funptr)(int) = print，声明指向函数的指针, 该函数接受一个int参数, 返回void, 并用print函数的地址初始化。

**函数名是什么？**

函数名被使用时总是由编译器将它转换为该函数类型的函数指针, 比如 print 的类型就是 void () (int) 所以可以用 print 初始化 funptr. 当使用 & 时只是显示的说明了编译器的转换操作, 所以 print 与 &print 都是 void () (int) 类型。

注意: 函数名与数组名在这点上的区别。函数名本质上就是个指针，不管什么类型的函数名都是指针，都仅占用一个字节。所以指针funptr与\*funptr在运算上也毫无差别，都是地址加一。

**如何通过指向函数的指针调用函数?**事实上有三种调用方式可选:

print(10); (1) 使用函数名调用print函数

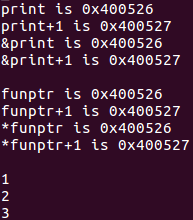
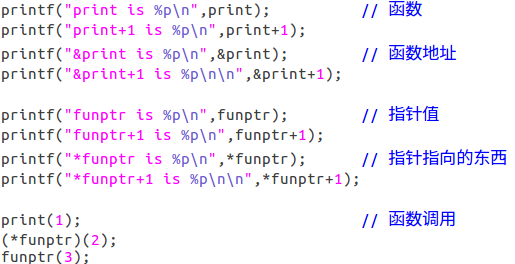
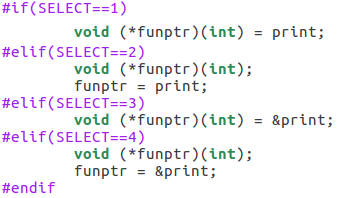
(\*funptr)(10); (2) 使用指向函数指针解引调用print函数`

funptr(10); (3) 使用指向函数指针直接调用print函数

(1)的执行过程是函数名print首先被转换成一个指向函数的指针, 该指针指向print函数在内存中的位置, 然后调用操作符调用print函数, 执行开始于这个位置的代码。

(2)的执行过程是解引操作将funptr转换为函数名, 然后执行(1)的操作, 显然解引操作不是必需的, 函数调用操作符需要的是一个指向函数的指针。

(3)的执行过程是函数调用操作符直接调用函数。



**从上图可知，四种初始化方式完全相同，三种调用方式完全相同。**

**一般选用3、4声明方式，选用1、2调用方式，以此来与数组指针写法保持一致，也更容易理解。**

**声明 “指向函数的指针” 的指针**

指向函数的指针本质上仍然是一种指针, 我们可以按照二级指针声明指向它的指针。

****

**声明元素类型为 “指向函数的指针” 的数组**

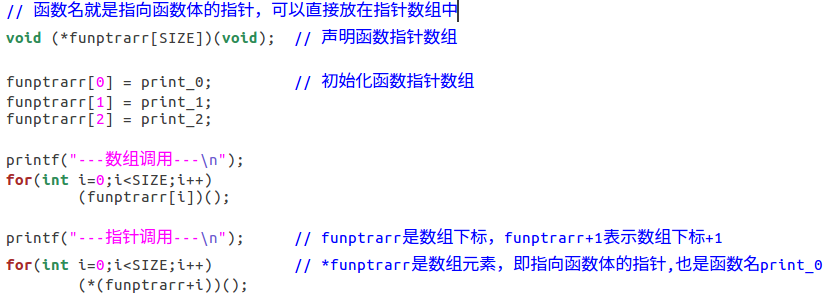
指针数组：装着指针的数组。函数指针：指向函数的指针。**函数指针数组**，数组中装着指向函数的指针。**名字从前往后念，写法优先级从后往前写**。最优先是本质。

同样, 如果具有多个类型一样的指向函数的指针, 我们可以将它们放在一个数组中。①void print(int num)， 声明函数；②void (\*funptr)(int) = &print， 声明指向函数的指针；③void (\*funptrarray[])(int) = funptr， 声明元素类型为"指向函数的指针"的数组, 数组大小为1, funptrarray[0]用funptr初始化；④(\*funptrarray[0])(10)，调用print；⑤(\*\*funptrarray)(10)， 等价的调用print。

方法一：



方法二：



上述两种方法得出的结果完全一致，并且都可以解释的通。

**注：下标是特殊的地址。**

# 字符串

## 字符串输入与输出

**字符串输入，**如果想把一个字符串读取到程序中，必须首先预留存储字符串的空间，然后使用输入函数来获取这个字符串，C库提供了三个读取字符串的函数：scanf()、gets()和fgets()。

**创建存储空间，**可以通过数组(char name[10])来分配存储空间，也可以通过C语言动态内存分配函数来动态分配存储空间。

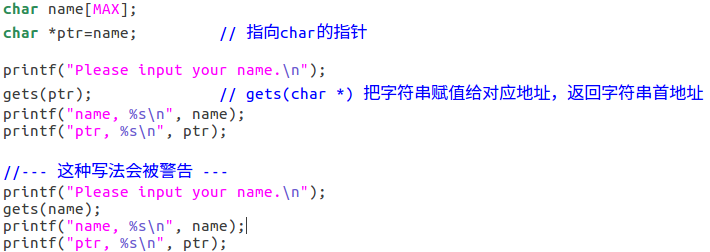
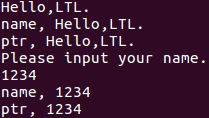
char \*name；scanf("%s", name)；如果通过上面的代码来创建存储空间，可能会通过编译器，但是在读入name的时候，name会覆盖程序中的数据和代码，并可能导致程序异常终止。这个是因为scanf()把信息复制到由name指定的地址中，而在这种情况下，参数是个未被初始化的指针，name可能指向任何地方。**所以不能用这种方式，要用数组或动态内存分配函数。**

**初始化，见下方代码中的写法。**

**gets()函数**

gets()(get string)函数对于交互式程序非常方便，它从系统的标准输入设备(通常是键盘)获得一个字符串。因为字符串没有预定的长度，所以gets()函数通过判断遇到的第一个换行符(\n)结束输入，按回车键可以产生这个字符。它读取换行符之前(不包括换行符)的所有字符，并在这些字符后添加一个空字符(\0)。gets()函数不检查存储区是否能够容纳实际输入的数据。

如果在gets()函数在读取字符串时出错或者遇到文件结尾，它就返回一个空(或0)地址，这个空地址被称为空指针，并且stdio.h里面定义的常量 NULL来表示，可以用while(get(name) != NULL)来进行一些错误检测。也可以通过 while((ch = getchar()) != EOF)来完成上面的错误检测。

上图中数组名name和ptr都是指向字符的指针，区别在于name是常量而ptr是变量。不需要指针直接gets（name）是可以用的，但是会被警告。下面是动态分配的方式：

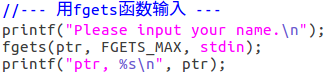


**综上所述共三种初始化：①数组+指针；②数组；③指针动态分配。**

**注：**空指针和空字符是不一样的，不要混淆。空指针是一个地址，而空字符是一个char类型的数据对象其值为0，数字上都可以用0表示，但是概念不同：NULL是一个指针，而0是一个char类型的常量。

**fgets()函数**

因为gets()函数不会检查存储区是否能够容纳实际输入的数据，多出来的字符简单地溢出到相邻的内存区，所以上面的代码在编译的时候会有warning。fgets()函数和gets()函数的不同：它需要第二个参数来说明最大读入字符数。如果这个参数值为n，fgets()就会读取最多n-1个字符或者读完一个换行符为止(因为会自动添加一个空字符(\n))，由这两者中最先满足的那个结束输入。



如果fgets()读取到换行符，就会把它存到字符串里，而不是像gets()那样丢弃。

它还需要第三个参数来说明读哪一个文件，从键盘上读取数据时，可以使用stdin(代表standard input)作为参数，这个标识符在stdio.h中定义。

**scanf()函数**

scanf()函数可以使用%s格式来读入一个字符串，scanf()函数和gets()函数的主要区别在于如何决定字符串何时结束。scanf()函数更基于获得单词(get word)而不是获取字符串(get string)。scanf()函数有两种方法决定输入结束，无论哪一种都是遇到的第一个非空白字符开始：**①**如果使用%s格式，字符串读取到(但不包括)下一个空白字符(比如空格、制表符或换行符)结束；**②**如果指定了字段宽度，比如%10s，scanf()函数就会读取10个字符或者直到遇到第一个空白字符，由二者最先满足的那一个终止输入。

**字符串输出，**C语言有三个输出字符串的标准库函数：puts()、fputs()和printf()。

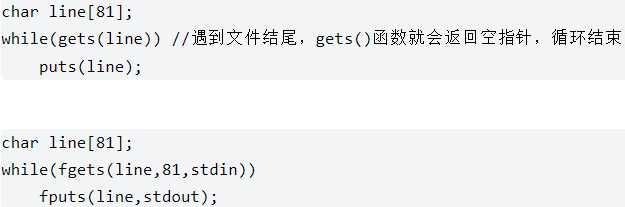
**puts()函数**

puts()函数使用很简单，只需要给出字符串参数的地址，**它遇到空字符(\0)就会结束输出(所以必须要有空字符)**。puts()函数在显示字符串的时候，**会自动在其后添加一个换行符(\n)。**

**fputs()函数**

fputs()函数是puts()函数面向文件版本，两者主要的区别是：①fputs()函数需要第二个参数来说明要写的文件，可以使用stdout(standard output)作为参数来进行输出显示。②与puts()函数不同，fputs()函数并不为输出自动添加换行符。

读取一行并把它回显在下一行，用下面的两种循环都可以办到。



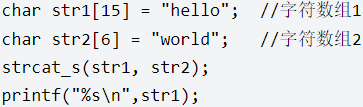
**printf()函数**

如同puts()函数一样，printf()函数在输出字符串的时候同样需要一个字符串地址作为参数，但是printf()函数没有puts()函数方便，但是它可以格式化多种数据类型，输出的时候也不自动添加换行符。

## 字符串函数【\*】

**输入输出函数：**scanf(格式控制，地址列表)、printf(格式控制，地址列表)；gets(字符数组名)、puts(字符数组名)；fgets(字符数组名，长度，标志参数)、fputs(字符数组名，长度，标志参数)；getch(字符数组名)、putch(字符数组名)……gets()和puts()函数只能输入和输出一个字符串，不可输出多个，而scanf()和printf()可输入输出多个。

**字符串连接函数：**strcat(字符数组1，字符数组2)。字符串2连接到字符串1的后面，将连接后的结果放在字符数组1中，最后得到字符数组1的地址。两个字符串后面均有结束符’\0’，在连接时字符串1后面的结束符会被取消，只在新串最后保留结束符。



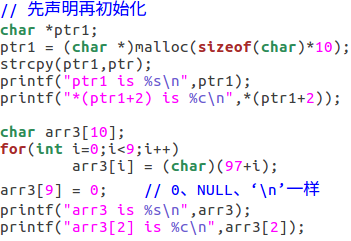
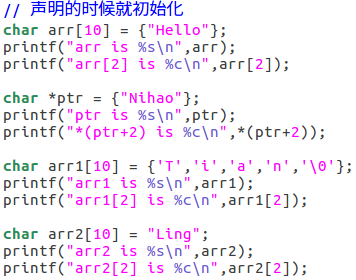
**字符串赋值函数：**strcpy(字符数组1，字符数串2或者字符数组名)、strncpy(字符数组1，字符串2或者字符数组名，长度)。**注：①**若开始没有对字符数组1进行初始化或者复制，str1中的内容是无法预知的，复制时str2将内容和结束符一起复制到str1中，而此时str1字符数组结束符后面的剩余空间的字节内容不一定是’\0’；**②**不能用赋值语句将字符串常量或者字符数组直接赋值给字符数组，如str1=“world”;这写法是错误的，必须要用strcpy()函数，用赋值语句将一个字符赋值给字符型变量或者字符数组是可以的。

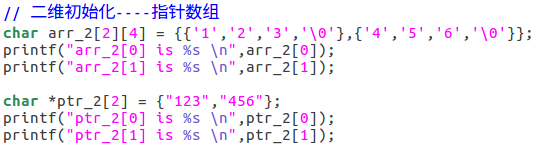
**字符串比较函数：**strcmp(字符串1或者字符数组，字符数串2或者字符数组)。**①**两个字符串从左到右按照 ACSII 码值大小比较，直到出现不同的字符或者遇到’\0’为止；**②**若出现不相同的字符，则以第一对不相同的字符的比较结果为准；**③**字符串1等于字符串2，函数值为0，字符串1大于字符串2，函数值为正整数，字符串1小于字符串2，函数值为负整数；**④**两个字符串不能直接用str1>str2这种方式比较，必须用strcmp()函数。

**字符串长度函数：**strlen(字符串或者字符数组)。**①**strlen()函数在测试字符串长度时不包括结束符；**②**为了测试字符串的实际长度，C语言规定了一个字符串结束标志，以字符’\0’作为结束标志，因此虽然字符数组大小为15，但是程序运行结果为5，因为碰到’\0’自动结束。

**字符串转换大小写函数：①**大写转小写：strlwr(字符数组)；**②**小写转大写：strupr(字符数组)。

**字符串初始化整合：双引号字符串，单引号字符。字符串先声明再赋值要用函数。**

****

****

# 结构体【\*】

## 结构体简介

结构体，怎么理解？

你可以把它想象成一个桌面上的文件夹，这个文件夹里面可以有各种各样的文件，当然也还可以再有文件夹的存在，文件夹里面再放文件……。如果你要修改其中一个文件的内容，就是首先通过桌面上的那个文件夹作为入口，然后一个一个的进入文件夹去寻找你需要的文件，找到之后就可以随你修改了。

long、unsigned int 、short、char（相当于各种文件类型，比如 .txt、.c、.h）这些关键字是否很熟悉？这都是 C 语言定义好的数据类型，直接拿来用就行了。但是我想自定义一个别的类型的数据怎么办？就靠 struct 了。结构体，顾名思义，就是将一个个数据类型构成一个数据类型以方便使用。

比如说一个 24 位的像素，有 R、G、B 三种颜色，每种颜色都用 8 bit 表示，如果使用一般的方法怎么表示呢？这样表示当然没有问题，也能解决你的需求，但是你在使用的时候会发现，通常他们使用场合一样，只是有的时候需要使用 Red 值，有的时候需要 Green，有的时候可能又要 Blue，但它们的共同点是都是用来表示一个像素的，那么有没有办法把这些数据类型组合起来方便调用呢？当然有，就是今天的主角，struct。

我们先看看使用 struct 如何表示一个像素：Pixel 中文表示像素，这样就通过这个结构体将三个数据结合在一起了（用文件夹装在一起），并且这个新组合的数据类型就叫 Pixel，和 int、char 等类似。

那么我们如何像使用 int、char 一样定义一个代表像素的数据类型呢？就是通过 struct + 结构体名 定义了。这里定义了两个像素，每个像素下都有 Red、Green、Blue 这三个字节数据，也就是说共有六个字节的空间：那么对于这些数据如何使用呢？比如说要设置 Red=100，Green=120，Blue=210：这样就行了，是不是很简单呢！如果你的 MDK 开启了输入补充功能，那么写这些代码就更容易了：可以看到你在敲完 Pixel1. 的圆点后，结构体的成员立马出来了，这时候你就能自己选择哪一个成员了，是不是很方便呢。不然一个结构体那么多成员，怎么记得住啊。如果你用这个结构体数据类型定义了一个结构体指针，那么就通过 -> 箭头调用，相当方便的。

而且如果你想对整个结构体进行赋值也是很方便的事情：这样一条语句就将三个成员变量的值进行了修改，和通过关键字定义的变量并没什么区别。但还有一点，每次定义一个结构体变量都要敲 struct 关键字还是很麻烦的事情，所以这个时候可以使用 **typedef** 这个关键字了：这样声明之后，每次要定义一个新的 Pixel 结构体，只要使用 Pixel 就行了，而不必加入 struct 来声明这是一个结构体。而为了让自己知道这是一个**自己定义的数据类型**，**一般会在名称后面加 \_t 或者 TypeDef 等**。比如 GPIO 结构体：并且结构体（文件夹）里面还可以套结构体（文件夹），被套的结构体里面也可能有结构体……。不仅能套结构体，指针、联合体、枚举、数组（各种文件）也都是一样的，而常规的 char、int 等更不用说了，完全按你的心意随意组合就是了（比如上面的结构体套了 uint16\_t 和 两个结构体）。

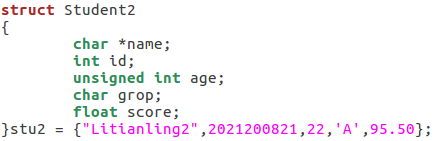
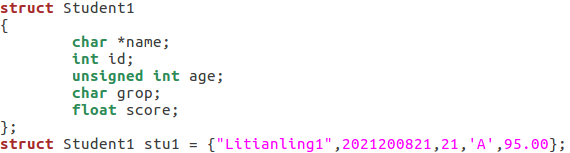
这些数据类型都可以通过结构体形成一个数据结构类型，是不是感觉特别方便啊。小项目可能结构体用的不多，但是大项目如果不用结构体，那么操作数据类型是一件很麻烦的事情，所以一定要学会使用结构体。

当然了，套用的结构体多了，对运行效率还是有一些影响的，对一些性能要求比较高的地方可以不用结构体，或者通过一些方法提高效率，不然你套的深了，寻找其中的成员变量还是需要不少指令消耗的，这一点需要引起注意。另外，编译器为了优化读写效率，可能会对数据类型进行填充：这个存储空间应该是 1+1+1+4 = 7，但是实际上是 8 （你可以通过 sizeof() 测试），就是因为 STM32 的处理字长为 4 个字节，是最快的读写长度，所以对变量 Reserve 进行了 4 字节对齐。它的存放位置如下：当然这个是可以通过对齐方式进行修改的，但最好不要，因为这样会降低读写效率，除非是那种定义好的通信协议，那没办法，只能改了（这个坑千万要注意）。

有的时候可能需要**获取结构体的偏移地址**，此时就可用通过以下方法获取：结构体偏移量计算宏（非常规方法）：**#define OFFSET(TYPE, MEMBER) ((unsigned long)(&(((TYPE \*)0)->MEMBER)))**，比如说要获取 Green 在结构体 Pixel 的偏移地址，就可以通过上面的宏进行计算：Offset 的结果就是 1，因为前一个 Red 共占用了一个字节空间，所以它的偏移地址就是 1。

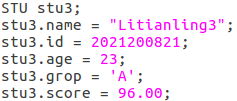
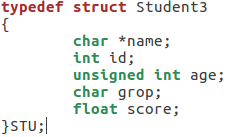
最后**结构体的初始化，就是对成员变量顺序初始化**：struct xxxx xxx = {xxxxx, xxxxx, xxxxx};而在 C99 模式下可以使用以下方式对结构体进行初始化，这样就能方便的观察每一个变量的初始值：struct xxxx xxx = {.x1 = xxxxx,.x2 = xxxxx,.x3 = xxxxx}。

**结构体的声明与初始化**

****

上述两方式完全相同stu1与stu2都是全局变量，若要声明新的结构体，则要在main函数中声明赋值。方式①：struct Student1 stu1 = {"Litianling1",2021200821,21,'A',95.00};直接声明并初始化。方式②：STU stu3; stu3.name = "Litianling3"; stu3.id = 2021200821; tu3.age = 23; stu3.grop = 'A'; stu3.score = 96.00;先声明再初始化。

**还可以定义新结构体类型，在main函数中可以直接用STU声明不需要struct关键字**

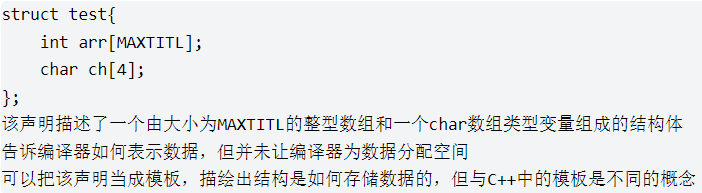


## 结构体数组

**结构体**

在C语言中，可以使用结构体来存放一组不同的数据类型，提高数据的表示能力。结构体是一种集合，可以包含多个变量或数组，类型可以先相同也可以不同，包含的变量或数组称为结构体的成员。

**结构体声明**：描述了一个结构的组织布局



**定义结构体变量**见上一节。

**访问结构体成员**，使用结构成员运算符——点(.)访问结构体中的成员。即“结构体变量名.结构体成员”。

**结构体初始化——列表初始化【只能在声明的同时初始化】**

struct Student1 stu1 = {"Litianling1",2021200821,21,'A',95.00};

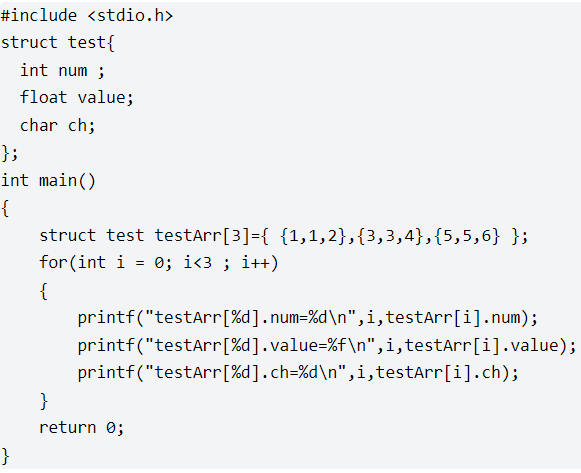
**结构体初始化——指定初始化方式1**

stu3.name = "Litianling3"; stu3.id = 2021200821; tu3.age = 23; stu3.grop = 'A';

**结构体初始化——指定初始化方式2**

stu3 = {.name = "Litianling3",.id = 2021200821, .age = 23,.grop = 'A'};

**结构体数组**，结构体数据就是指数组中的每一个元素都是结构体。

****

**结构体指针**

当一个指针变量指向结构体时，称为结构体指针：struct 结构体名 \*变量名；

结构体变量名与数组名不一样，数组名在表达式中会被转换成数组指针，而结构体变量名不会，无论在任何表达式中标表示的都是整个结构体本身，要获取结构体变量的地址，则必须使用取地址符&。

**结构体指针获取结构体成员**

**通过结构体指针获取结构体成员的方式：①(\*ptr).structMember，这种方式相当于“结构体.元素”；②ptr->structMember，这种方式是直接引用。**

****

注：.运算符优先级高于\*，所以(\*ptr)括号不能少。

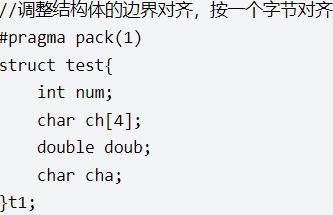
**结构体指针作为函数参数**

结构体变量名代表整个结构体变量本身，当把结构体变量本身作为参数传递时，结构体内部的结构体成员较多时会造成时间和空间开销大，影响程序的运行效率。

使用结构体指针作为参数传递时，因为传递的是地址，会比原本含有较多结构体的结构体变量更加块，但是也会造成结构体变量数据被修改的可能。

**字节对齐**

结构体中的各个成员理论上在内存中都是连续存储的，但**实际上是根据字节对齐规则进行存储的。**#pragma pack(n)，指定按照n字节对齐规则进行存储。



**字节对齐的基本规则**

①结构体内部任何K字节的基本对象相对于结构体首地址的偏移，必须是K的整数倍

②结构体变量的首地址能够被其最宽基本类型成员的大小所整除

③结构体的总大小为结构体最宽基本类型成员大小的整数倍

序：计算机内存是以字节（Byte）为单位划分的，理论上CPU可以访问任意编号的字节，但实际情况并非如此。CPU 通过地址总线来访问内存，一次能处理几个字节的数据，就命令地址总线读取几个字节的数据。32 位的 CPU 一次可以处理4个字节的数据，那么每次就从内存读取4个字节的数据。

**对齐的作用和原因**

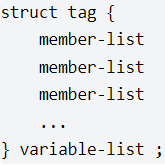
各个硬件平台对存储空间的处理上有很大的不同。一些平台对某些特定类型的数据只能从某些特定地址开始存取。比如有些架构的CPU在访问一个没有进行对齐的变量的时候会发生错误,那么在这种架构下编程必须保证字节对齐.其他平台可能没有这种情况，但是最常见的是如果不按照适合其平台要求对数据存放进行对齐，会在存取效率上带来损失。比如有些平台每次读都是从偶地址开始，如果一个int型（假设为32位系统）如果存放在偶地址开始的地方，那么一个读周期就可以读出这32bit，而如果存放在奇地址开始的地方，就需要2个读周期，并对两次读出的结果的高低字节进行拼凑才能得到该32bit数据。显然在读取效率上下降很多。

将一个数据尽量放在一个步长之内，避免跨步长存储，这称为内存对齐。在32位编译模式下，默认以4字节对齐；在64位编译模式下，默认以8字节对齐。

**总结**

结构体是一种自定义的数据类型，是创建变量的模板，不占用内存空间，真正需要开辟内存空间来存储的是结构体成员变量，结构体中的各个成员理论上在内存中都是连续存储的，但实际上是根据字节对齐规则进行存储的。

## 结构体指针【\*】

 struct 语句的格式如下：tag 是结构体标签。member-list 是标准的变量定义，比如 int i; 或者 float f，或者其他有效的变量定义。variable-list 结构变量，定义在结构的末尾，最后一个分号之前，您可以指定一个或多个结构变量。在一般情况下，tag、member-list、variable-list 这 3 部分至少要出现 2 个。

**结构体占空间吗？**

int 、char 是变量类型，不是变量；变量才分配内存；那么struct xxx 占空间吗？也不占空间；结构体是我们自定义的数据类型，不是变量；只有将**类型实例化才占内存空间**。实例化的结构体占多少内存由结构体成员决定，并且要考虑字节对齐。

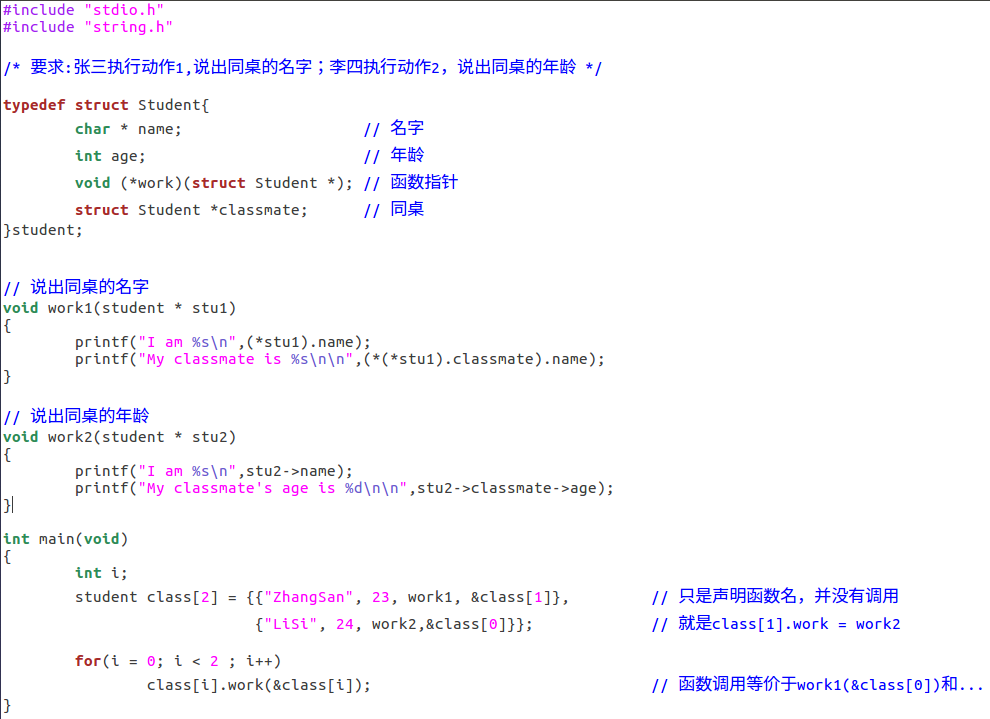
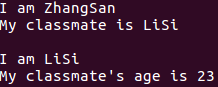
结构体指针初始化与应用见上面

## 结构体与函数【\*】

**结构体做函数参数有三种传递方式：**一是传递结构体变量，这是值传递，二是传递结构体指针，这是地址传递，三是传递结构体成员，当然这也分为值传递和地址传递。

**结构体+函数指针**

函数指针是指向函数的指针变量。通常我们说的指针变量是指向一个整型、字符型或数组等变量，而函数指针是指向函数。函数指针可以像一般函数一样，用于调用函数、传递参数。



**上面这张图是重中之重，结构体、指针、数组、函数，全都套在一起了。**

## 链式结构【\*】

链表是一种常见的基础数据结构，结构体指针在这里得到了充分的利用。**链表可以动态的进行存储分配**，也就是说，链表是一个功能极为强大的数组，他可以在节点中定义多种数据类型，还可以根据需要随意增添，删除，插入节点。链表都有一个**头指针**，一般以**head**来表示，存放的是一个**地址**。链表中的节点分为两类，**头结点和一般节点**，**头结点没有数据域**。链表中每个节点都分为两部分，一个**数据域**，一个是**指针域**。说到这里你应该就明白了，链表就如同车链子一样，**head指向第一个元素**：第一个元素又指向第二个元素；……，直到最后一个元素，该元素不再指向其它元素，它称为“表尾”，**表尾的地址部分放一个“NULL”**（表示“空地址”），链表到此结束。

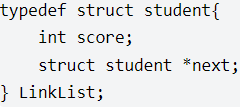
作为有强大功能的链表，对他的操作当然有许多，比如：链表的**创建，修改，删除，插入，输出，排序，反序，清空链表的元素，求链表的长度**等等。

初学链表，一般从单向链表开始。

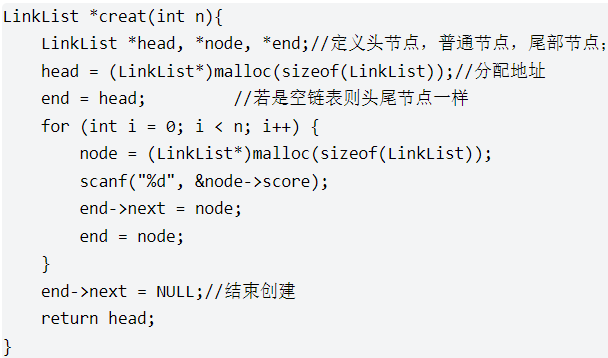
 

空链表 有n个节点的链表

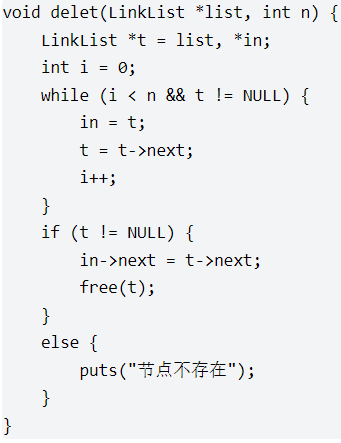
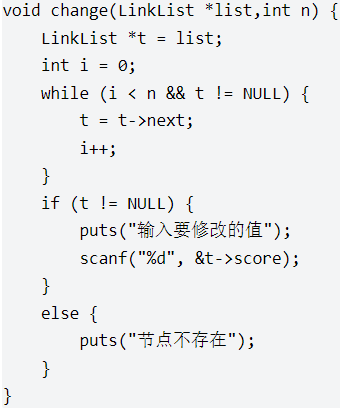
**（1）创建链表结构**



**（2）初始化一个链表**，n为链表节点个数。

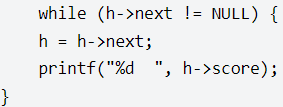


**（3）修改链表节点值**，修改第n个节点



**（4）删除链表的元素**也就是把前节点的指针域越过要删除的节点指向下下个节点。即：p->next = q->next;然后放出q节点的空间，即free(q);

**（5）插入链表节点，**插入节点就是用插入前节点的指针域链接上插入节点的数据域，再把插入节点的指针域链接上插入后节点的数据域。根据图，插入节点也就是：e->next = head->next; head->next = e;增加链表节点用到了两个结构体指针和一个int数据。



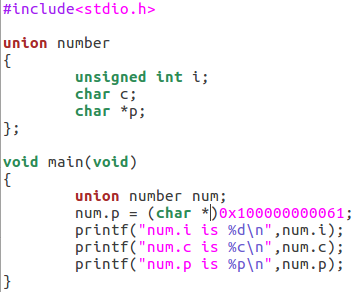
**（6）输出链表**，输出链表很简单，边遍历边输出就行了。

# 联合体与枚举类型【\*】

## 联合体union

**联合体的概念**

**联合体的声明和结构体类似**，但他的行为却和结构体不同。联合体所有成员引用的是内存中相同位置。当你想在不同时刻把不同东西存在同一个位置时可以使用联合体。



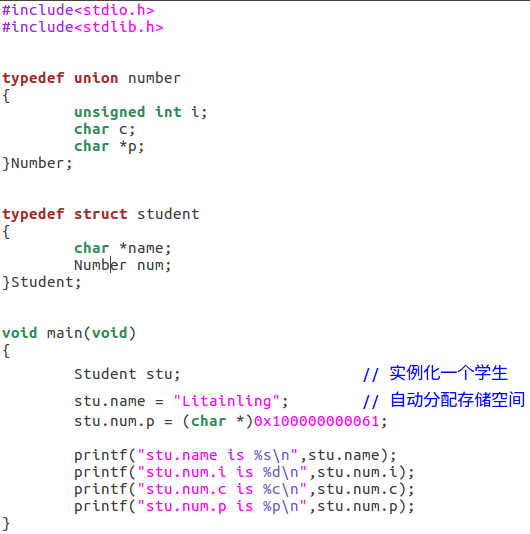
如上图所示，联合体就是特殊的结构体，如果其的各个成员具有不同的变量，联合的长度等于最长成员的长度。例如上图中“unsigned int”占用4个字节即32位，“char”占1个字节即8位，“char \*”是指针类型，指针的值char就是系统地址，在64位操作系统中占8个字节即64位。所以联合体占8个字节。

**其程序运行、翻译过程：①向num.p写十六进制的0x1000 0000 0000 0061；②输出为整型数据时，后四个字节0x0000 0061翻译成十进制大的97输出；③输出为字符数据时，最后的一个字节0x61翻译成对应的字符‘a’输出。**简单说就是同一块内存，被翻译成不同类型的数据，如何翻译取决于定义的数据类型。

**联合体的作用**

为什么使用联合体，当你只想看浮点型数如何存储在一块特定的存储器上，可以用联合体。例如：联合体中定义float与char \*，向float写值然后输出指针的值，即可看到存储器实际存储的十六进制值。

再者，如果一个结构体中（或者结构体某一部分）同一时间只有一个元素使用，那么就会浪费其他的存储空间，用结构体就能大大改善。**并且结构体与联合体还可以嵌套。**

****

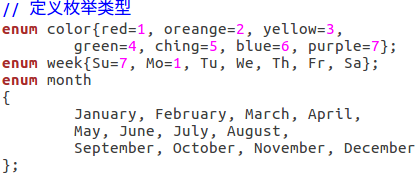
以上两种方式完全相同。

## 枚举类型enum

**基本概念**

枚举enum就是英文enumerate的缩写，也就是列举、排列说明的意思。枚举类型是C语言中的一种特殊类型，枚举类型可以让我们的程序使用一些固定长度和固定数值的变量值范围。定义枚举类型：enum 枚举类型 {枚举值列表};

**定义枚举类型，默认从0开始**，January = 0



如果在定义枚举类型时指定元素的值，也可以改变枚举元素的值：enum weekday { sun=7,mon=1,tue,wed,thu,fri,sat}day；这时，sun 为 7，mon 为 1，以后元素顺次加 1，所以 sat 就是 6 了。

**调用枚举类型**



**定义枚举类型的变量并初始化**



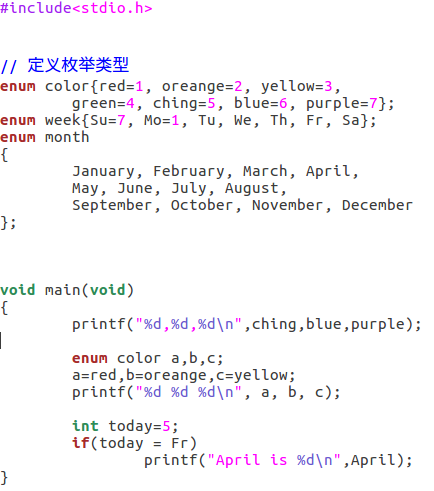
**调用枚举类型的变量**



**枚举值可以用来作判断**



**整体效果如下：**



**总结：定义方式与结构体相同，存储方式类似结构体但不同之处在于共享存储，调用方式完全不同于结构体（不能color.red）但类似于变量。**

# 位运算

## 位和字节

**位**

　　二进制数系统中，每个0或1就是一个位(bit)，位是数据存储的最小单位。其中8 bit就称为一个字节（Byte）。计算机中的CPU位数指的是CPU一次能处理的最大位数，例如32位计算机的CPU一次最多能处理32位数据，计算机中的CPU位数也成为机器字长，和数据总线（CPU与内部存储器之间连接的用于传输数据的线的根数）的概念是统一的。

注：关于地址总线、数据总线、控制总线的概念与原理在另一篇博文里面来讲述。

**比特**

　　1) 计算机专业术语，是信息量单位，是由英文BIT音译而来。二进制数的一位所包含的信息就是一比特，如二进制数0101就是4比特。

2)二进制数字中的位，信息量的度量单位，为信息量的最小单位。数字化音响中用电脉冲表达音频信号，“1”代表有脉冲，“0”代表脉冲间隔。如果波形上每个点的信息用四位一组的代码表示，则称4比特，比特数越高，表达模拟信号就越精确，对音频信号信号还原能力越强。

**字节：**1字节（byte） = 8 比特（bit）

注：这个字节与比特的关系是规定的，记住就好，通用于任何场景，容易混淆的是字长和字节，字长指的是cpu一次性能够运算的数据的位数，不同的计算机可能不一样，但是字节这个概念是恒久不变的。

一个英文字符和英文标点占用一个字节，一个中文字符和中文标点占用两个字节。

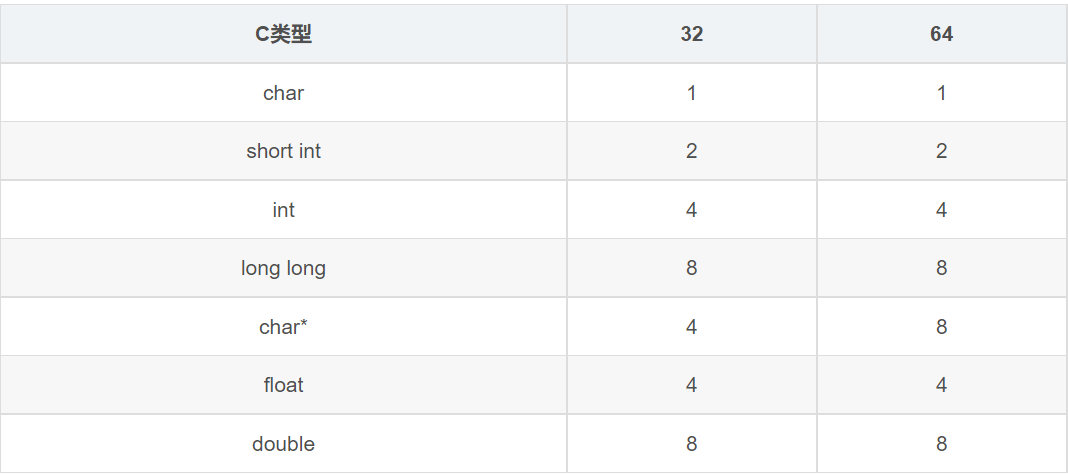
**半字：**1半字 = 0.5 字，长度随字长而定

**字**

在计算机中，一串数码作为一个整体来处理或运算的，称为一个计算机字，简称宇。字通常分为若干个字节(每个字节一般是8位)。在存储器中，通常每个单元存储一个字，因此每个字都是可以寻址的。字的长度用位数来表示。

在计算机的运算器、控制器中，通常都是以字为单位进行传送的。宇出现在不问的地址其含义是不相同。例如，送往控制器去的字是指令，而送往运算器去的字就是一个数。

一个字由若干个字节组成，不同的计算机系统的字长是不同的，常见的有8位、16位、32位、64位等，字长越长，计算机一次处理的信息位就越多，精度就越高。**64位的计算机，1字=8字节=64位，半字=4字节=32位。32位的计算机，1字=4字节=32位，半字=2字节=16位。16位微型计算机，1字=2字节=16位，半字=1字节=8位。**



## 进制转换

**绪论**

十六进制（Hexadecimal）：在数学中是一种逢16进1的进位制。一般用数字0到9和字母A到F（或a~f）表示，其中:A~F表示10~15。

十进制（Decimal System）：每相邻的两个计数单位之间的进率都为十；十进制是中华民族的一项杰出创造，在世界数学史上有重要意义。著名的英国科学史学家李约瑟教授曾对中国商代记数法予以很高的评价，"如果没有这种十进制，就几乎不可能出现我们现在这个统一化的世界了"，李约瑟说："总的说来，商代的数字系统比同一时代的古巴比伦和古埃及更为先进更为科学。"

八进制（Octal）：一种以8为基数的计数法，采用0，1，2，3，4，5，6，7八个数字，逢八进1。一些编程语言中常常以数字0开始表明该数字是八进制。八进制的数和二进制数可以按位对应（八进制一位对应二进制三位），因此常应用在计算机语言中。

**十进制与二进制转换**

二进制（binary）：在数学和数字电路中指以2为基数的记数系统，以2为基数代表系统是二进位制的。这一系统中，通常用两个不同的符号0（代表零）和1（代表一）来表示。

十进制转二进制：①把十进数除以2，记下余数（余数保存在字符串中），现用商除以2，再记下余数，如此循环，直到商为0；②把保存余数的字符串反过来，就是结果。

二进制转十进制：把二进制字符串从最高位（左边第一位）开始用商乘以2再加余数（该位的数字），如此循环，左边第一位的商肯定是0。

**进制之间转换原则**

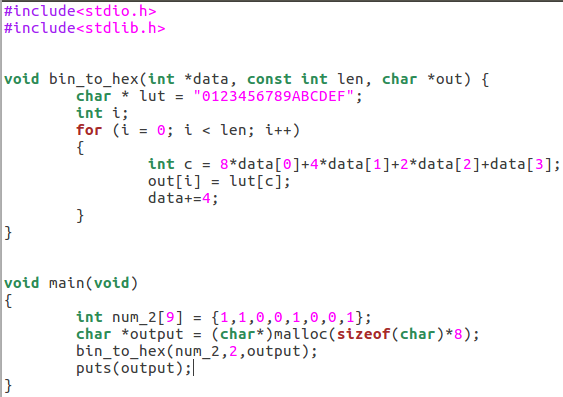
转换原则：不同进制之间的转换本质就是确定各个不同权值位置上的数码。转换正整数的进制的有一个简单算法，就是通过用目标基数作长除法；余数给出从最低位开始的“数字”

基于上述原则详细解释十进制转换成二进制：

十进制整数部分转换：用2去除十进制整数，可以得到一个商和余数；再用2去除商，又会得到一个商和余数，如此进行，直到商为零时为止，然后把先得到的余数作为二进制数的低位有效位，后得到的余数作为二进制数的高位有效位，依次排列起来。

十进制小数部分转换：用2乘十进制小数，可以得到积，将积的整数部分取出，再用2乘余下的小数部分，又得到一个积，再将积的整数部分取出，如此进行，直到积中的小数部分为零，或者达到所要求的精度为止。

十进制和二进制转换，二进制再转换为8或16进制。



## 位逻辑运算符

**&按位与**，参加运算的两个数据，按二进制位进行“与”运算。如果两个相应的二进制位都为１，则该位的结果值为1；否则为0。这里的1可以理解为逻辑中的true,0可以理解为逻辑中的false。按位与其实与逻辑上“与”的运算规则一致。逻辑上的“与”，要求运算数全真，结果才为真。若，A=true,B=true,则A∩B=true。按位与的用途：①清零，若想对一个存储单元清零，即使其全部二进制位为0，只要找一个二进制数，其中各个位符合条件——原来的数中为1的位，新数中相应位为0。然后使二者进行&运算，即可达到清零目的；②取一个数中某些指定位，若有一个整数a(2byte),想要取其中的低字节，只需要将a与8个1按位与即可；③保留指定位，与一个数进行“按位与”运算，此数在该位取1。

**| 按位或**，两个相应的二进制位中只要有一个为1，该位的结果值为1。借用逻辑学中或运算的话来说就是，一真为真。应用：按位或运算常用来对一个数据的某些位定值为1

**~按位取反**，他是一元运算符，用于求整数的二进制反码，即分别将操作数各二进制位上的1变为0，0变为1。

**^按位异或运算**，相同为0不同为1（同或相同为1不同为0）。

**<<左移运算符**，左移运算符是用来将一个数的各二进制位左移若干位，移动的位数由右操作数指定（右操作数必须是非负值），其右边空出的位用0填补，高位左移溢出则舍弃该高位。左移1位相当于该数乘以2，左移2位相当于该数乘以2\*2＝4,15＜＜2=60，即乘了４。但此结论只适用于该数左移时被溢出舍弃的高位中不包含1的情况。

**>>右移运算符**，右移运算符是用来将一个数的各二进制位右移若干位，移动的位数由右操作数指定（右操作数必须是非负值），移到右端的低位被舍弃，对于无符号数，高位补0。对于有符号数，某些指令将对左边空出的部分用符号位填补（即“算术移位”），而另一些指令则对左边空出的部分用0填补（即“逻辑移位”）。注意：对无符号数,右移时左边高位移入0；对于有符号的值,如果原来符号位为0(该数为正),则左边也是移入0。如果符号位原来为1(即负数),则左边移入0还是1,要取决于所用的计算机系统。有的系统移入0,有的系统移入1。移入0的称为“逻辑移位”,即简单移位；移入1的称为“算术移位”。

**位运算赋值运算符**，位运算符与赋值运算符可以组成复合赋值运算符。例如: &=, |=, >>=, <<=, ∧=。a & = b相当于 a = a & b，a << =2相当于a = a << 2。

## 对其特性

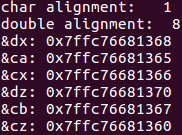
位操作之对齐特性：\_Alignas()&&\_Alignof()

对齐指的设置变量在内存中的存储位置。

\_Alignof运算符可以求出存储某个类型所需的字节数，在关键字\_Aligof后面的圆括号中写上类型名即可：\_Alignof(int)的返回值是4，意思int类型变量用4个字节存储。

可以使用\_Alignas说明符设置变量或类型的对齐值，但设置的对齐值不能小于存储该类型所需的字节数。①\_Alignas(double) char c1;②\_Alignas(8) char c2;③unsigned char \_Alignas(long double) c\_arr[sizeof(long double)]。

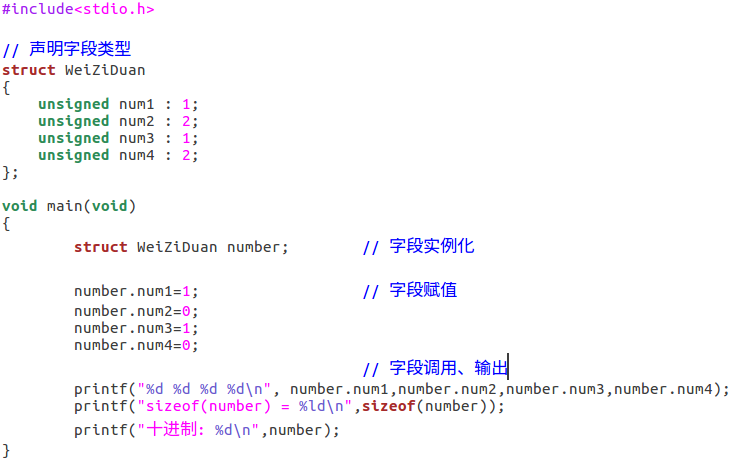
注意：不同版本的要求不同，有的要求\_Alignas(type)说明符在类型说明符的后面，有的在前面，无论哪一种都能够识别。



## 位字段

**位字段**（bit filed）是C语言中一种存储结构，不同于一般结构体的是它在定义成员的时候需要指定成员所占的**位数**。位字段是一个signed int或unsigned int类型变量中一组相邻的位（C99和C11新增了Bool类型的位字段）。位字段通过一个结构声明来建立，该结构声明为每个字段提供标签，并确定该字段的宽度。

本质：一种特殊的结构体，成员默认是int类型只需要指定是有符号的还是无符号的，需要单独指明每个成员占用几个比特位来存储。



**如上图所示，number实际上只用了1+2+1+2=6个比特位进行存储，但计算机默认最小对其单元是4个字节，所以sizeof(number)=4实际占用4个字节。number在存储器中的存储实际上是：1（1比特）00（2比特）1（1比特）00（2比特），在实际上输出number的时候只读取到了“1001”，转换成十进制就是9。**

# 预处理器

## 宏定义

**预处理机制**

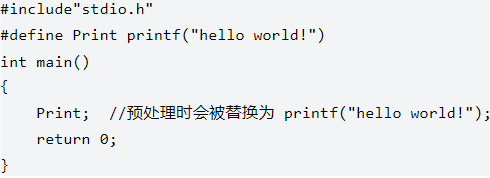
编译一个C语言程序的第一步骤就是预处理阶段，这一阶段就是宏发挥作用的阶段。C预处理器在源代码编译之前对其进行一些文本性质的操作，主要任务包括删除注释、插入被#include进来的文件内容、定义和替换由#define 定义的符号以及确定代码部分内容是否根据条件编译（#if )来进行编译。”文本性质”的操作，就是指一段文本替换成另外一段文本，而不考虑其中任何的语义内容。宏仅仅是在C预处理阶段的一种文本替换工具，编译完之后对二进制代码不可见。

**宏定义的用法——①宏常量**

我们最常使用到的#define的用法就是用#define来定义一个符号常量，而要修改时,只需修改#define这条语句就行了,不必每处代码都修改。

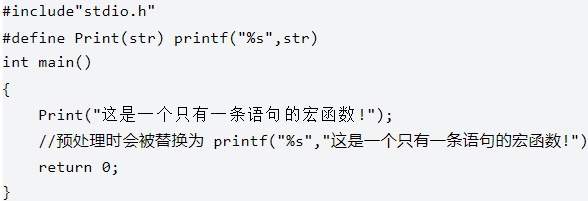
**宏定义的用法——②宏语句**

我们还可以用宏定义一条或多条语句。



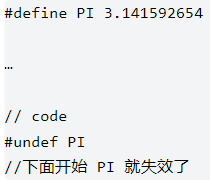
**宏定义的用法——③宏函数**

我还可以用宏来定义函数，宏定义也可以带参数。

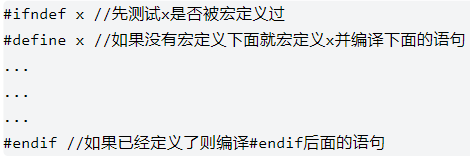
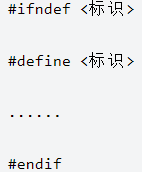


**宏定义的用法——④其他**

#undef 是用来撤销宏定义的，用法如下：



使用ifndef防止头文件被重复包含和编译，他是条件指示符中的一种。这是宏定义的一种，它可以根据是否已经定义了一个变量来进行分支选择，一般用于调试等等。实际上确切的说这应该是预处理功能中三种（宏定义，文件包含和条件编译）中的一种----条件编译。 条件指示符#ifndef检查预编译常量在前面是否已经被宏定义。如果在前面没有被宏定义，则条件指示符的值为真，于是从#ifndef到#endif之间的所有语句都被包含进来进行编译处理。相反，如果#ifndef指示符的值为假，则它与#endif指示符之间的行将被忽略。

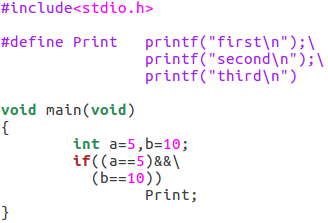
 

标识符命名规则：<标识>在理论上来说可以是自由命名的，但每个头文件的这个“标识”都应该是唯一的。标识的命名规则一般是头文件名全大写，前后加下划线，并把文件名中的“.”也变成下划线，如：头文件stdio.h的标识符是\_STDIO\_H\_

千万不要忽略了头件的中的#ifdef，这是一个很关键的东西。比如你有两个C文件，这两个C文件都include了同一个头文件。而编译时，这两个C文件要一同编译成一个可运行文件，于是问题来了，大量的声明冲突。所以还是把头文件的内容都放在#ifndef和#endif中吧。不管你的头文件会不会被多个文件引用，你都要加上这个。

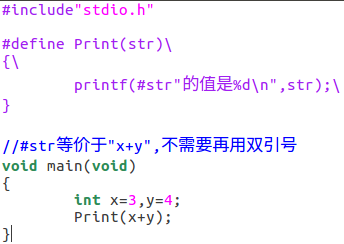
**宏定义相关作用符——①换行符号“\”**

我们定义宏语句或者宏函数时不可能总是一条语句呀,那要是有很多条语句时怎么办?都写在一行吗?这样显然代码就不美观,可读性不好,所以有多条语句时,我们就在每行末尾(除了最后一行)加上"\",代表换行的意思。【也能在主函数中用，例如if的条件过长】

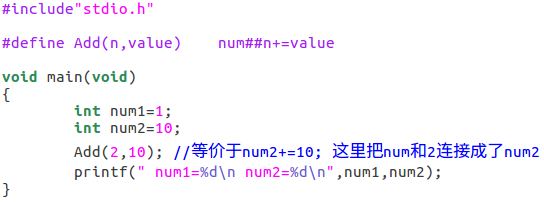
**宏定义相关作用符——②字符串化符号“#”**

"#"是“字符串化”的意思，在宏定义中的#是把跟在后面的参数转换成一个字符串。



**宏定义相关作用符——③片段连接符号“##”**

“##”是一种分隔连接方式，它的作用是先分隔，然后进行强制连接。在普通的宏定义中，预处理器一般把空格解释成分段标志，对于每一段和前面比较，相同的就被替换。但是这样做的结果是，被替换段之间存在一些空格。如果我们不希望出现这些空格，就可以通过添加一些##来替代空格。



**宏函数的妙用——①类型传递**

我们知道函数虽然可以传递参数,但是却不能把类型作为参数传递,有时我们为了实现函数的复用性，就要使用STL模板，但是我们这个时候还有另外一种选择，就是写成宏函数。例如一个开辟内存的函数：

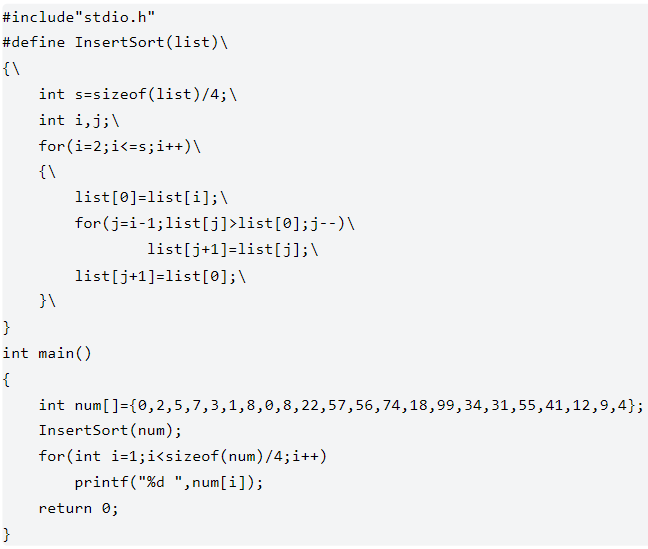


这个时候，我们就能把类型为参数进行传递了，可以开辟各种类型的内存。



**宏函数的妙用——②传递数组**

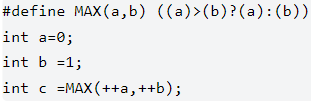
由数组作为函数参数传递时，会失去其数组特性，也就是无法使用sizeof()函数计算出数组的大小，比如我们写一个排序函数，排序时我们不仅需要知道数组的首地址，还需要知道数组的大小，但是仅仅把数组名作为参数传递时，无法得知其数组大小，这时我们的函数就需要传递第二个参数，也就是数组的大小，于是函数就要写成Sort(int \*a,int size)。但宏函数就可以解决这个问题。例:



**注意事项——①运算符优先级**

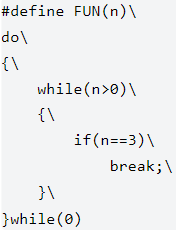
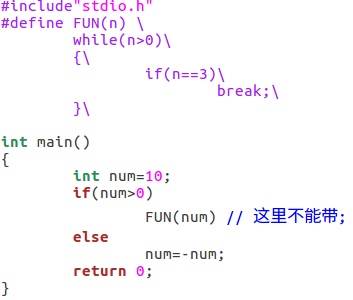
#define MULTIPLY(x, y) x \* y，这是一个很简单的乘法函数，计算MULTIPLY(10, 10)结果是100，但是如果计算MULTIPLY(5+5, 10)时，结果还是100吗?当然不是，MULTIPLY (5+5 , 10)=5+5\*10=55，所以结果是55，所以写宏函数时要特别注意运算符的优先级，这里稳妥一点的写法应该是#define MULTIPLY(x, y) ((x)\*(y))。

**注意事项——②宏参数重复调用**



这里不是c=MAX(1,2)=2；而实际上上面代码等价于int c =((++a)>(++b)?(++a):(++b));可以看到实际上a b都各自加了两次,所以c=1>2?2:3=3,所以结果是3。

**注意事项——③分号吞噬**



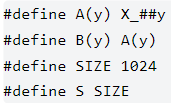
如果加上分号，看似代码没有问题但是一编译就报错，编译器显示"error: ‘else’ without a previous ‘if’"，原来是因为FUN函数是一个代码块，然后if(num>0) FUN(num); 就等价于if(num>0) {…}; 这不就是在大括号后面打分号了吗？这样else当然就缺少if了。这个时候我们可以用do{…}while(0)来解决这个问题，因为while()后面正好需要一个分号。也可以直接去掉函数调用的分号。

**注意事项——④递归调用**

#define NUM (4 + NUM)，按前面的理解，(4 + NUM)会展开成(4 + (4 + NUM))，然后一直展开下去，直至内存耗尽。但是，预处理器采取的策略是只展开一次。也就是说，NUM只会展开成(4 + NUM)，而展开之后NUM的含义就要根据上下文来确定了。

**注意事项——⑤宏参数预处理**

宏参数中若包含另外的宏，那么宏参数在被代入到宏体之前会做一次完全的展开，除非宏体中含有#或##。

A(S)会被展开成X\_S。因为宏体中含有##，宏参数直接代入宏体。B(S)会被展开成X\_1024。因为B(S)的宏体是A(S)，并没有#或##，所以S在代入前会被完全展开成1024，然后才代入宏体，变成X\_1024。

## #include 指令

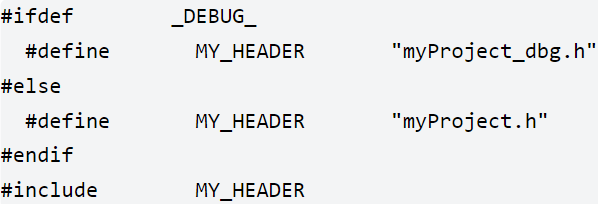
**#include 命令介绍**：#include 命令是预处理命令的一种。预处理命令可以将别的源代码内容插入到所指定的位置；可以标识出只有在特定条件下才会被编译的某一段程序代码；可以定义类似标识符功能的宏，在编译时预处理器会用别的文本取代该宏。

**插入头文件的内容**

#include 命令告诉预处理器将指定头文件的内容插入到预处理器命令的相应位置。有两种方式可以指定插入头文件：#include <文件名>、#include "文件名"。

如果需要包含标准库头文件，应该使用第一种格式。如果需要包含针对程序所开发的自定义源文件，则应该使用第二种格式。采用 #include 命令所插入的文件，通常文件扩展名是 .h，文件包括函数原型、宏定义和类型定义，只要使用 #include 命令，这些定义就可被任何源文件使用。

注：可以在 #include 命令中使用宏。如果使用宏，宏替换后的结果必须确保生成正确的 #include 命令。



预处理上述程序时，如果 DEBUG 宏已被定义，那么预处理器会插入 myProject\_dbg.h 的内容；如果还没定义，则插入 myProject.h 的内容。

**预处理器如何找到头文件**

由编译器的 C 语言版本决定 #include 命令所指定文件的搜索路径，同时也由其决定文件名是否区分大小写。**①**对于命令中使用尖括号指定的文件<文件名>，预处理器通常会在特定的系统路径下搜索。例如，在 Unix 系统中会搜索路径 /usr/local/include 与 /usr/include。**②**对于命令中用双引号指定的文件“文件名“，预处理器通常首先在当前目录下寻找，也就是包含该程序其他源文件的目录。如果在当前目录下没有找到，那么预处理器也会搜索系统的 include 路径。文件名中可以包含路径，如果文件名中包含了路径，则预处理器只会到该目录下寻找。

还可以通过使用编译器命令行选项，或在环境变量（该变量通常称为 INCLUDE）中加入搜索路径，为 #include 命令指定自己的搜索路径。具体的做法请参考采用的编译器的说明文档。

**嵌套的#include命令**

#include 命令可以嵌套使用；也就是说，通过 #include 命令插入的源文件本身也可以包含另一个 #include 命令。预处理器最多允许 15 层的嵌套包含。

因为头文件有时候会包含另一个头文件，很容易发生相同的一个文件被多次包含的情况。例如，假设myProject.h文件中有如下代码#include <stdio.h>，假设main.c源文件中有#include <stdio.h>与#include "myProject.h"，源文件就会两次包含stdio.h，一次是直接包含，另一次是间接包含。

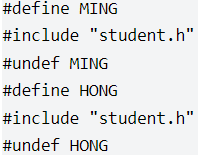
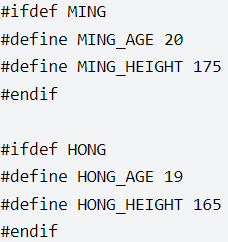
可以采用条件编译命令**避免多次包含相同的文件**，如将myProject.h文件改写成“#ifndef \_INCFILE\_H\_ #define \_INCFILE\_H\_ /\* ...实际的头文件incfile.h的内容 ... \*/ #endif。第一次出现包含 incfile.h 的命令时，\_INCFILE\_H\_ 宏是没有定义的，因此插入#ifndef和#endif之间的内容，这段内容包含了对 INCFILE\_H\_ 宏的定义。嵌入 incfile.h 文件之后，#ifndef 条件就会为false，预处理器会忽略 #ifndef 和 #endif 之间的内容。

## #undef指令

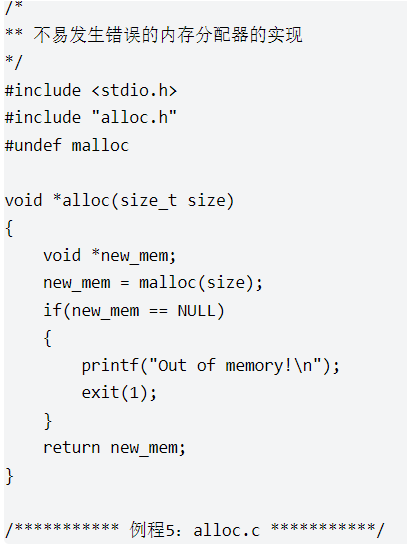
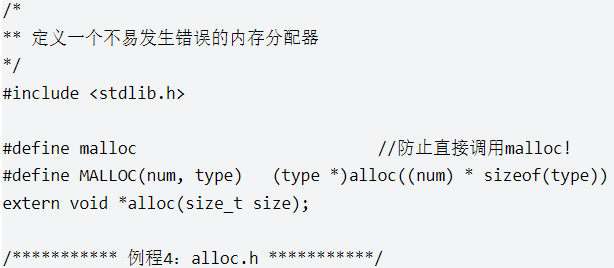
C语言中#undef的语法定义是：#undef 标识符，用来将前面定义的宏标识符取消定义。

**防止宏定义冲突**，在一个程序块中用完宏定义后，为防止后面标识符冲突需要取消其宏定义。在一个程序段中使用完宏定义后立即将其取消，防止在后面程序段中用到同样的名字而产生冲突。

**增强代码可读性**，在同一个头文件中定义结构类型相似的对象，根据宏定义不同获取不同的对象，主要用于增强代码的可读性。在一个头文件里定义的两个对象与分别在两个头文件里定义效果相同，但如果将相似的对象只用一个头文件申明，可以增强源代码的可读性。



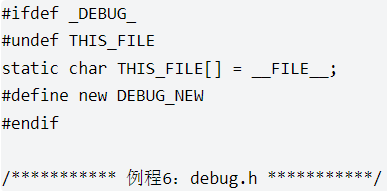
**自定义接口**，将某个库函数包装成自定义接口，而只允许用户调用自定义接口，禁止直接调用库函数。例如，自定义安全的内存分配器接口：



其中“#define malloc”是为了防止用户直接调用库函数malloc。**①**只要用户源文件包含了头文件alloc.h，就不能直接调用库函数malloc，而只能调用自定义函数MALLOC，如果用户要调用库函数malloc编译器会发生错误。**②**在alloc.c实现自定义函数需要用到库函数malloc，所以需要用这一句“#undef malloc”取消alloc.h中对malloc的宏定义。

这种技巧还是比较有意思的，用于对已经存在的库函数进行封装。而且如果包含了自定义接口文件，就不能直接调用库函数，而只能调用自定义封装的函数。

**用于调试头文件中**，偶然看到这样一个代码用到了#undef。



## 条件编译

使用与平台有关的C语言函数，可能会使得程序不具有可移植性。比如Socket编程、多线程编程等是与平台有关的。若想将程序做成平台无关的就需要用到与平台相关的条件编译。基本概念，条件编译的行为类似于C语言中的if…else… 条件编译是预编译指示命令,用于控制是否编译某段代码。

**GCC编译器**：①#ifdef \_\_GNUC\_\_ ②#if \_\_GNUC\_\_ >= 3 // GCC3.0以上

Visual C++编译器：①#ifdef \_MSC\_VER(非VC编译器很多地方也有定义) ②#if \_MSC\_VER >=1000 // VC++4.0以上 ③#if \_MSC\_VER >=1100 // VC++5.0以上 ④#if \_MSC\_VER >=1200 // VC++6.0以上 ⑤#if \_MSC\_VER >=1300 // VC2003以上 ⑥#if \_MSC\_VER >=1400 // VC2005以上

**Borland C++编译器**：#ifdef \_\_BORLANDC\_\_

**UNIX操作系统**：#ifdef \_\_unix 或 #ifdef \_\_unix\_\_

**Linux操作系统**：#ifdef \_\_linux 或 #ifdef \_\_linux\_\_

**FreeBSD操作系统**：#ifdef \_\_FreeBSD\_\_

**NetBSD操作系统**：#ifdef \_\_NetBSD\_\_

**32位Windows操作系统**：#ifdef \_WIN32 或 #ifdef WIN32

**64位Windows操作系统**：#ifdef \_WIN64

**GUI App**：#ifdef \_WINDOWS

**CUI App**：#ifdef \_CONSOLE

**Windows版本的WINVER** PC机Windows(95/98/Me/NT/2000/XP/Vista)和Windows CE都定义了：①#if (WINVER >= 0x030a) // Windows 3.1以上 ②#if (WINVER >= 0x0400) // Windows 95/NT4.0以上 ③#if (WINVER >= 0x0410) // Windows 98以上 ④#if (WINVER >= 0x0500) // Windows Me/2000以上 ⑤#if (WINVER >= 0x0501) // Windows XP以上 ⑥#if (WINVER >= 0x0600) // Windows Vista以上

**Windows 95/98/Me版本的\_WIN32\_WINDOWS** MFC App、PC机上(Windows CE没有定义)：①#ifdef \_WIN32\_WINDOWS ②#if (\_WIN32\_WINDOWS >= 0x0400) // Windows 95以上 ③#if (\_WIN32\_WINDOWS >= 0x0410) // Windows 98以上 ④#if (\_WIN32\_WINDOWS >= 0x0500) // Windows Me以上

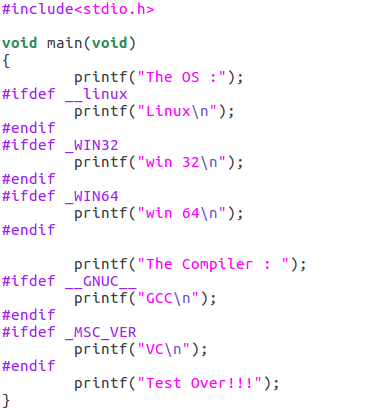
**Windows NT版本的\_WIN32\_WINNT**：①#if (\_WIN32\_WINNT >= 0x0500) // Windows 2000以上 ②#if (\_WIN32\_WINNT >= 0x0501) // Windows XP以上 ③#if (\_WIN32\_WINNT >= 0x0600) // Windows Vista以上

**Windows CE(PocketPC)** ：#ifdef \_WIN32\_WCE

**Internet Explorer版本的\_WIN32\_IE Cygwin**：#ifdef \_\_CYGWIN\_\_

**32bit版Cygwin(现在好像还没有64bit版)** ：#ifdef \_\_CYGWIN32\_\_

**MinGW(-mno-cygwin指定)** ：#ifdef \_\_MINGW32\_\_

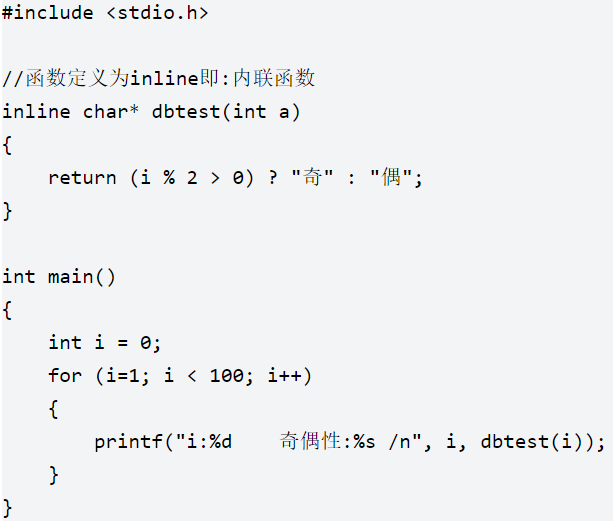


C语言条件编译(#if，#ifdef，#ifndef，#endif，#else，#elif，#end)。C语言由源代码生成的各阶段如下: C源程序->编译预处理->编译->优化程序->汇编程序->链接程序->可执行文件。

## 内联函数

**什么是内联函数**

在C语言中，如果一些函数被频繁调用，不断地有函数入栈到函数栈中，会大量消耗栈空间或栈内存。为了解决这个问题，专门引入了inline修饰符来表示内联函数。栈空间就是放置程序的局部数据空间，也就是函数内数据的内存空间，系统中栈空间是有限的，假如频繁大量的使用就会因栈空间不足造成程序出错，函数的死循环、递归调用等最终就会导致栈内存空间枯竭。



上面的例子就是标准的内联函数的用法，使用inline修饰带来的好处我们表面看不出来，其实在内部的工作就是在每个for循环的内部任何调用dbtest(i)的地方都换成了(i%2> 0)?"奇":"偶"这样就避免了频繁调用函数对栈内存重复开辟所带来的消耗。其实这种有点类似动态库和静态库的问题，使 dbtest 函数中的代码直接被放到main 函数中，执行for 循环时，会不断调用这段代码，而不是不断地开辟一个函数栈。

**内联函数的编程风格**

关键字inline 必须与函数定义体放在一起才能使函数成为内联，仅将inline 放在函数声明前面不起任何作用。【**申明不带，实现带**】所以说，inline 是一种 “用于实现的关键字” ，而不是“用于声明的关键字”。一般地，用户可以阅读函数的声明，但是看不到函数的定义。尽管在大多数教科书中内联函数的声明、定义体前面都加了inline 关键字，但我认为inline 不应该出现在函数的声明中。这个细节虽然不会影响函数的功能，但是体现了高质量C++/C 程序设计风格的一个基本原则：声明与定义不可混为一谈，用户没有必要、也不应该知道函数是否需要内联。【上层开发无需看到底层】

inline的使用是有所限制的：**inline只适合函数体内代码简单的函数数使用**，不能包含复杂的结构控制语句例如while、switch，**并且内联函数本身不能是直接递归函数**(自己内部还调用自己的函数)。

**慎用内联**

内联能提高函数的执行效率，为什么不把所有的函数都定义成内联函数？

内联是以代码膨胀（复制）为代价，仅仅省去了函数调用的开销，从而提高函数的执行效率。如果执行函数体内代码的时间，相比于函数调用的开销较大，那么效率的收获会很少。另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。一个好的编译器将会根据函数的定义体，自动地取消不值得的内联（这进一步说明了inline 不应该出现在函数的声明中）。

以下情况不宜使用内联：①如果函数体内的代码比较长，使用内联将导致内存消耗代价较高。②如果函数体内出现循环，那么执行函数体内代码的时间要比函数调用的开销大。

**总结**：将内联函数放在头文件里实现是合适的，省却你为每个文件实现一次的麻烦。而声明跟定义要一致其实是指，如果在每个文件里都实现一次该内联函数的话，最好保证每个定义都是一样的，否则将会引起未定义的行为。如果不是每个文件里的定义都一样，那么，编译器展开的是哪一个，那要看具体的编译器而定。所以最好将内联函数定义放在头文件中。

# 文件

## 文件的基本操作

文件是当今计算机系统不可或缺的部分。文件用于存储程序，文档，数据，书信，表格，图形，照片，等其它种类的信息，作为程序员，必须会编写创建文件和从文件中读写数据。

**什么是文件？**

文件通常是指在磁盘或固态是的一段已经命名的缓冲区，例如我们经常使用的stdio.h就是一个文件，c语言将文件看成一段连续的字节，每个字节都可被单独读输，c提供了两种模式：文本模式和二进制模式。

**打开文件**

在c语言中我们一般利用fopen函数来打开文件，该函数声明在<stdio.h>库中。调用该函数的一般形式：FILE \* fp; fp = fopen(const char \*filename, const char \*mode); 其中filename指向一个字符串，该字符串为该文件的文件名（即路径），第二个参数为c语言规定的模式字符串。



**注：如果使用w模式打开一个现有文件，该文件的内容会被删除。**

程序成功打开文件后，fopen函数返回文件指针，其他I/O函数可以使用这个指针指定该文件，即上述代码中的fp，FILE是c语言的一种结构体，它的定义在stdio.h库中。其中包含了缓冲区信息等，这里不多赘述。若是我们打开文件失败，这里fopen（）函数会返回一个空指针。

**关闭文件**

我们在对文件进行完操作之后要关闭文件，防止文件内容被破坏。在c语言中我们一般用fclose（）函数关闭文件，形式如下：fclose(fp);

其中fp为文件指针，必要时会刷新缓冲区。当磁盘已满、移动硬盘被拔出或者出现I/O错误时文件会关闭失败，所以对于比较正式的程序我们应该检查文件是否关闭，当文件成功关闭时fclose函数返回0，否则返回EOF（这里eof为文件结尾）。

判断代码如下：if (fclose(fp) != 0) printf("Error in closing file %s\n", filename);

**文件的读写操作——输入输出字符**

在c语言中我们利用getc()和putc()函数进行文件里字符的输入和输出，语法如下所示：ch = getc(fp); putc(ch, fp); getc函数是读取文件中的一个字符赋给ch。putc函数是将ch写入文件中。

**文件的读写操作——输入输出字符串**

在文件中我们要进行字符串的输入和输出可以利用fgets函数和fputs函数，语法如下所示：

①fgets(buf, stlen, fp); fgets函数的第一个参数和gets函数相同表示存储输入的地址（如上述代码中的buf是一个char型数组的名称），第二个参数是一个整数，表示带输入字符串的大小，最后一个参数为文件指针。**fgets函数读取输入直到第一个换行符的后面，或者读到文件的末尾——EOF，或者读取到stlen-1个字符，然后fgets函数在末尾添加一个空字符使之成为一个字符串，故我们读取的字符串长度为字符数加上一个空字符**。 注意，如果fgets函数在读到字符上限之前已经读完一行，它为将换行符放到空字符前面。fgets在遇见EOF即文件结尾时会返回null。

②fputs(buf, fp); fputs函数的两个参数实际意义与fgets相同，它的作用是将字符串写入指定的文件中，其中由于fgets函数会保留换行符，所以fputs函数不会像puts函数一样自动添加换行符。

**文件的读写操作——格式化输入与输出**

文件的格式化输入与输出函数为fscanf，fprintf函数其工作方式与scanf，printf类似。语法如下所示：fprintf(fp, "%s", str1); fscanf(fp, "%d", &a);

**文件的读写操作——文件读写移动操作**

在我们对文件进行读取时，经常需要改变读取的位置，在c语言中有两个函数可以便于我们进行这种操作。

①tell函数：用来取文件当前位置。其一般形式为:long n；n = ftell(fp); 它的返回值时一个长整型，表示当前的读写位置（从文件开始处到现在的字节数），调用正确返回当前读写位置，错误返回-1L。

②fseek函数：用来改变文件指针的当前位置。其一般形式为:fseek(fp，offset，from); offset:位移量，类型为long型，表示以from为起点移动的量相对值（字节数）。from:移动的起始位置。from是”SEEK\_SET”偏移量的起始点是“文件开始处”，from是”SEEK\_CUR”偏移量的起始点是“当前位置”，from是”SEEK\_END”偏移量的起始点是“文件末尾”。

## 文件的读写

**一、打开文件句柄**。参数1：文件路径，参数2：文件打开模式，函数执行成功返回文件流指针，错误返回NULL。FILE \*fopen(const char \*path, const char \*mode);

模式 操作 区别 文件要求

r 读 从文件头开始 文件需存在

r+ 读写 从文件头开始 文件需存在

w 写 从文件头开始 文件不存在则创建，存在则清空

w+ 读写 从文件头开始 文件不存在则创建，存在则清空

a 写 从文件尾开始 文件不存在进行创建，存在则追加

a+ 读写 从文件头读取，从文件尾写入 文件不存在进行创建，存在则追加

**二、文件关闭**。参数1：文件流，int fclose(FILE \*fp);

**三、文件写入**

**字符写入**。fputc(); 参数1：写入的字符，参数2：文件流。作用：将单个字符写入到文件中，返回值：成功时，返回写入字符的ascii码值，错误返回EOF（-1）。int fputc(int c, FILE \*stream);

**字符串写入**。fputs(); 参数1：写入的字符串，参数2：文件流。作用：将字符串写入文件中，返回值：返回一个非负值，如果发生错误则返回 EOF(-1)。int fputs(const char \*s, FILE \*stream);

**数据块写入。**fwrite(); 参数1：要获取的数据的地址，参数2：要写入内容的单字节数，参数3：要写入size字节的数据项的个数，参数4：目标文件指针。返回值：返回实际写入的数据块的数目，作用：向文件写入数据块，以二进制形式对文件进行操作，不局限于文本文件。size\_t fwrite(const void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);

**格式化写入**。fprintf(); 参数1：目标文件指针，参数2：指定的格式控制字符串，参数3：各种输出项，与格式控制字符串中的字段一起写到文件中。返回值：执行成功返回实际写入文件的字符个数；执行失败，返回负数。作用：用来将输出项按指定的格式写入到指定的文本文件中。int fprintf(FILE \*stream, const char \*format, ...);

**四、文件读取**

**字符读取**。fgetc()；参数1：目标文件指针。返回值：执行成功返回读取的字符，读取错误或者遇到结束标志EOF，返回EOF。作用：从指定的文件中读取一个字符。int fgetc(FILE \*stream);

**字符串读取**。fgets()；参数1：存储读取的数据，参数2：存储数据的大小，参数3：要读取的文件流。返回值：成功则返回读取的buf，失败则返回NULL，这是，buf中的数据不确定。作用：读取指定场长度的字符串存到字符数组中。每次只读取一行，每次最多读bufsize-1个字符（第bufsize个字符赋'\0'）。char \*fgets(char \*buf, int bufsize, FILE \*stream);

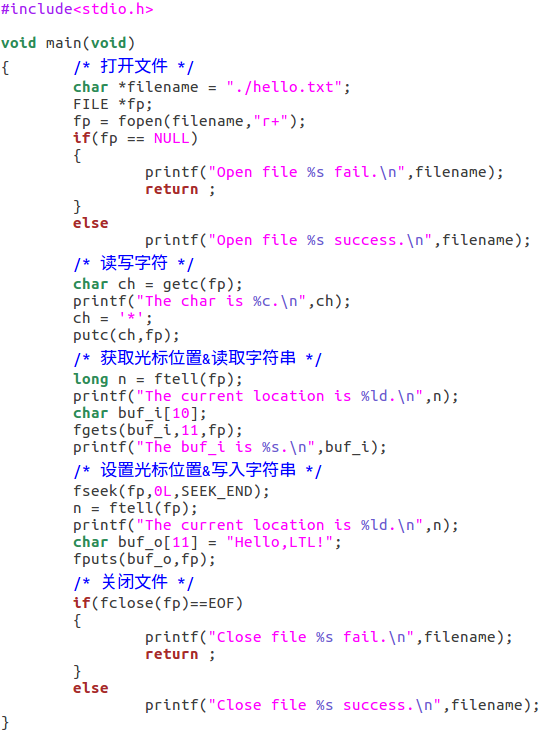
**数据块读取**。fread()；参数1：存储读取的数据，参数2：要读取的每个数据项的字节数，参数3：要读取的数据项的个数，参数4：读取的文件流。返回值：返回真实读取的数据项count数，错误时返回0。作用：一次读取文件中由若干个数据项组成的数据块，数据块的大小取决于数据项的大小和数据项的个数。size\_t fread ( void \*buffer, size\_t size, size\_t count, FILE \*stream) ;

**格式化读取**。fscanf()；参数1：读取的文件流，参数2：读取的字符格式，参数3：读取的各个输入项。返回值：成功则返回读入的参数个数，失败返回EOF。作用：根据数据格式format从文件流中读取数据，遇到空格或换行结束。int fscanf(FILE\*stream,const char\*format,[argument...]);

**五、文件读写结合**

**①**fgetc()与fputc()结合使用，以字符格式读取文件，再以字符格式写入文件，适用于文本文件。**②**fgets()与fputs()结合使用，从文件中按行读取字符串，再以字符串写入文件，适用于文本文件，优点是按行读取很方便。**③**fread()与fwrite()结合使用，以数据块格式读取，再以数据块格式写入到文件中，可以读取二进制文件，优点是读取二进制文件使用。**④**fprintf()与fscanf()结合使用，以格式化的方式读取，遇到空格或换行就结束，再将读取的文件写入到文件中，优点是可以指定写入的文件格式。

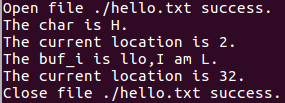
**六、例程**



原始的文件hello.txt如下：



程序运行结果如下：



处理后的文件如下：



# 存储管理

## 存储类别

C语言中的四种存储类别： 自动变量 （auto）、静态变量（static）、寄存器变量（register）、外部变量（extern）。

**自动变量**：也称为**局部变量**，通常在自定义函数内或代码段中（用“{}”括起来的）定义的变量，除了加了static关键字修饰的都是自动变量。自动变量数据存储在动态存储区中，都是动态地分配存储空间的。函数中的形参和在函数中定义的变量(包括在复合语句中定义的变量)都属此类，在调用该函数时系统会给它们分配存储空间，在函数调用结束时就自动释放这些存储空间。**自动变量用关键字 auto 进行存储类别的声明，例如“auto int num;”，函数内不加任何修饰词定义的变量都是自动变量**。

**外部变量**：也称为**全局变量**，是在函数的外部定义的，它的作用域为从变量定义处开始到本程序文件的末尾。**①**如果外部变量不在文件的开头定义，其有效的作用范围只限于定义处到文件末尾。如果在定义点之前的函数想引用该外部变量，则应该在引用之前用关键字 extern 对该变量进行“外部变量声明”，表示该变量是一个已经定义的外部变量，有了此声明就可以从“声明”处起合法地使用该外部变量。**②**我们如何和获取外部文件的某个变量呢？当然也可以通过extern关键字来进行声明。**总之**：我们可以用 extern 声明外部变量，达到扩展程序文件中的作用域效果，在哪里调用就在其前方再次用“extern”声明。

**静态变量**：有时希望函数中的局部变量的值在函数调用结束后不消失而保留原值，这时就应该指定局部变量为静态局部变量，用关键字 static 进行声明。**注：static关键字也可以用来标明静态函数防止函数名同名的冲突。详情请见《函数与程序结构》一章。**

**寄存器变量**：为提高效率，C语言允许将局部变量的值存放在 CPU 的寄存器中，这种变量叫做寄存器变量，用关键字register声明。使用寄存器变量需要注意以下几点：**①**只有局部自动变量和形式参数可以作为寄存器变量。**②**一个计算机系统中的寄存器数目有限，不能定义任意多个寄存器变量。**③**不能使用取地址运算符“&”求寄存器变量的地址。

**相关概念——作用域(scope)**

作用域描述了程序中合法访问一个标识符的区域。一个C变量的作用域可以是：代码块作用域(block scope)、函数原型作用域(function prototype scope)、文件作用域(file scope)。

**相关概念——链接(linkage)**

跟作用域类似，变量的链接是一个空间概念，描述了程序中合法访问一个标识符的区域。一个C变量的链接类型可以是：外部链接(external linkage)、内部链接(internal linkage)、空链接(no linkage)。

**相关概念——存储器(storage duration)**

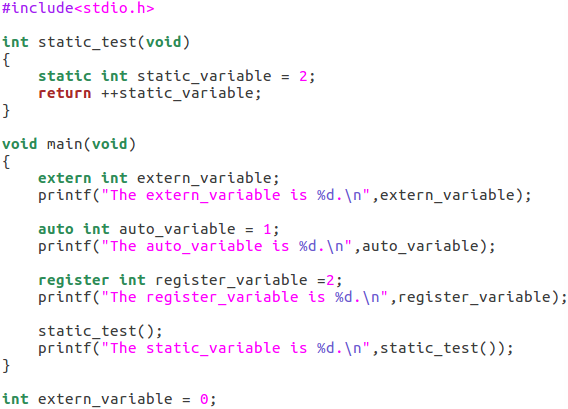
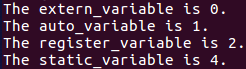
变量的声明周期，描述了一个C变量在程序执行期间的存在时间。一个C变量的存储期可以是：静态存储期(static storage duration)、自动存储期(automatic storage duration)、动态存储期(dynamic storage duration)。

**相关概念——存储类说明符**

①auto：声明一个自动变量。②static：声明一个静态变量，或声明一个内部链接函数和全局变量。③register：声明一个寄存器存储类变量。④extern：声明一个外部存储变量⑤typedef：语法意义上的存储类，与实际存储类型无关。



程序代码示例：



## 内存动态管理

**为什么要动态内存分配？**

前面用到的内存开辟方式为在栈上开辟空间，例如：int val = 20就是在栈空间上开辟四个字节的存储空间，char arr[10] = {0}就是在栈空间上开辟10个字节的连续空间。这种开辟空间的方式有两个特点：**①**空间开辟大小是固定的。**②**数组在声明的时候，必须指定数组的长度，它所需要的内存在编译时分配。

但是对于空间的需求，不仅仅是上述的情况。有时候我们需要的空间大小在程序运行的时候才能知道，那数组编译时开辟空间的方式就不能满足需求了，就只能试试动态存开辟了。

**动态内存函数——malloc**

**malloc用来开辟动态内存，void \* malloc(size\_t size)**。一般常用(强制类型转换)malloc(sizeof(类型)\*数量)，这个函数向内存申请一块连续可用的空间，并返回指向这块空间的指针。如果开辟成功，则返回一个指向开辟好空间的指针。如果开辟失败，则返回一个NULL指针，因此malloc的返回值一定要做检查。

返回值的类型是 void\* ，所以malloc函数并不知道开辟空间的类型，具体在使用的时候使用者自己来决定，需要强制类型转换。注意：参数 size 不要为0。

**动态内存函数——free**

**free函数用来释放动态开辟的内存，void free (void\* ptr)**。**①**如果参数 ptr 指向的空间不是动态开辟的，那free函数的行为是未被定义的。**②**如果参数 ptr 是NULL指针，则函数什么事都不做。**③**free前后指向的地址不发生任何变化，改变的只是指针和对应的内存的管理关系，一般再添加一句代码令指针为NULL如ptr=NULL。**④**动态申请时，指针包含有其它信息会浪费一定空间，所以使用malloc申请小块空间的成本较大，不推荐动态申请小块内存。

注：malloc和free都声明在 stdlib.h 头文件中。

注：要保证指针ptr是动态开辟的，并且开辟成功不是NULL。

**动态内存函数——calloc**

C语言还提供了一个函数叫 calloc ， calloc 函数也用来动态内存分配。原型如下：**void\* calloc (size\_t num, size\_t size)**，函数的功能是为 num 个大小为 size 的元素开辟一块空间，并且把空间的每个字节初始化为0。**与函数 malloc 的区别**只在于 calloc 会在返回地址之前把申请的空间的每个字节初始化为全0。

**动态内存函数——realloc**

realloc函数的出现让动态内存管理更加灵活。有时会我们发现过去申请的空间太小了，有时候我们又会觉得申请的空间过大了，为了合理使用内存，我们一定会对内存的大小做**灵活的调整**。realloc 函数就可以做到对动态开辟内存大小的调整。 函数原型：**void \* realloc(void \* ptr, size\_t size)**，ptr是要调整的内存地址，size调整之后新大小，返回值为调整之后的内存起始位置。

这个函数调整原内存空间大小的基础上，还会将原来内存中的数据移动到新的空间。realloc在调整内存空间的是存在两种情况：**①**原有空间之后有足够大的空间，要扩展内存就在原有内存之后直接追加空间，原来空间的数据不发生变化。②原有空间之后没有足够多的空间，扩展方法是在堆空间上另找一个合适大小的连续空间来使用，这样函数返回的是一个新的内存地址。要注意realloc函数返回的内存地址是否一致，如果realloc申请失败返回NULL指针老空间ptr被置空，如果无保险措施将造成内存泄漏，无法找到原有的内存地址。

**实际内存占用查看函数——malloc\_usable\_size，略**。

**常见动态内存错误——对NULL指针的解引用操作**，动态分配后要先确定是否分配成功，如果分配不成功就不能赋值。

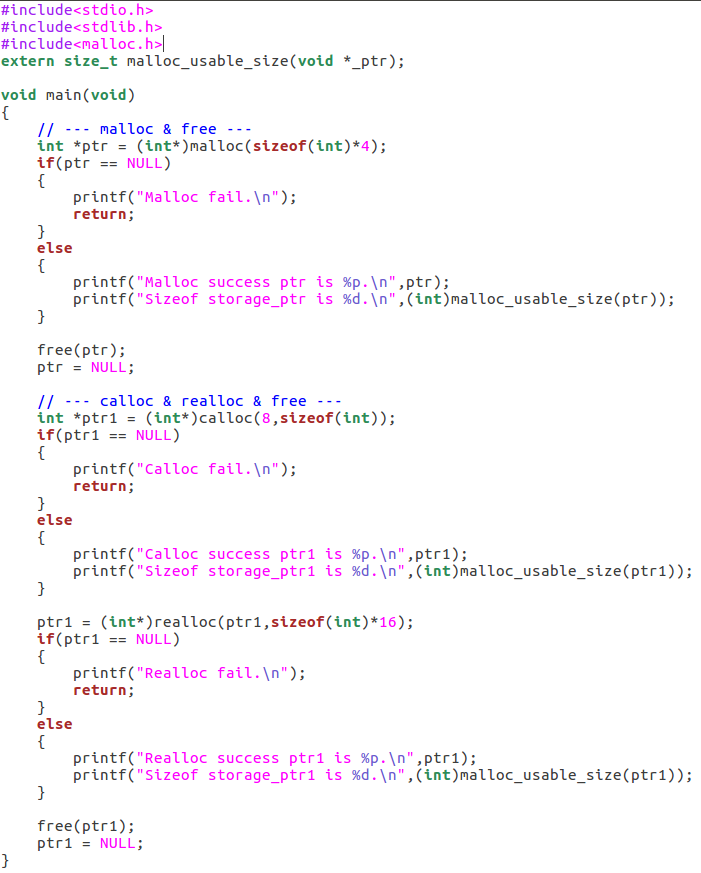
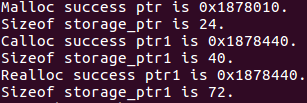
**常见动态内存错误——对动态开辟空间的越界访问**，如果分配了10个大小为sizeof(int)的空间就不能访问第11个。

**常见动态内存错误——对非动态开辟内存使用free释放**，如果创建了一个定长数组，它不是动态分配的不能释放。

**常见动态内存错误——使用free释放一块动态开辟内存的一部分**，当free(ptr)中的ptr没有指向动态分配空间的首地址就不能释放。

**常见动态内存错误——对同一块动态内存多次释放**。

**常见动态内存错误——动态开辟内存忘记释放（内存泄漏）**。



上述代码中，第一次分配了8+4\*4=24个字节的空间后释放，第二次分配了8+4\*8=40个字节的存储空间，之后调整为8+16\*4=72字节的空间最后释放。总的来说，实际占用的空间要比分配的空间多八个字节，用来存储动态分配空间大小等信息。

# 标准库函数

## 数学库

**三角函数 ：**sin() 正弦函数、cos() 余弦函数、tan() 正切函数、asin() 反正弦函数、acos() 反余弦函数、atan() 反正切函数、atan2() 带两个参数的反正切函数。

**双曲函数 ：**sinh()双曲正弦函数、cosh()双曲余弦函数、tanh()双曲正切函数。

**指数函数与对数函数 ：**exp () 以 e 为底数的指数函数、frexp(param，n) 二进制浮点数表示param\*2^n、log(x) 计算x的自然对数 、log10() 以10为底的对数、modf() 返回x的小数部分（其符号与x相同）但是参数中可以添加整数部分的变量、exp2() 返回2的x次方、log2(x) 计算x的二进制对数。

**幂函数 ：**pow(base, power) 幂函数、sqrt(x) 计算x的平方根、cbrt(x) 计算x的立方根、hypot(x,y) 计算直角三角形的斜边。

**误差与伽马函数：**erf(x)误差函数、erfc(x)余差函数erfc(x) = 1-erf(x)是误差函数的补函数、tgamma(x)伽马函数、lgamma(x) log伽马函数。 ​

**四舍五入与余数函数**：ceil(x) x上取整函数、floor(x) x的下取整函数、fmod(y, x) y/x的余数、round(x) x的四舍五入值 ​。

**绝对值、最小、最大：**fabs(x) x的绝对值函数、abs(x) x的绝对值、fmax(x, y) 两个参数中的最大值、fmin(x, y) 两个参数中的最小值。

## 通用工具库



**stdlib .h 头文件定义了四个变量类型、一些宏和各种通用工具函数。**

1、double atof(const char \*str)。函数原型 double atof(const char \*str)。函数功能：把参数 str 所指向的字符串转换为一个浮点数（类型为 double 型）。

2、int atoi(const char \*str)。函数原型 int atoi(const char \*str)。函数功能： 把参数 str 所指向的字符串转换为一个整数（类型为 int 型）。

3、long int atol(const char \*str)。函数原型 long int atol(const char \*str)。函数功能： 把参数 str 所指向的字符串转换为一个长整数（类型为 long int 型）。

4、double strtod(const char \*str, char \*\*endptr)。函数原型 double strtod(const char \*str, char \*endptr)。函数功能： 把参数 str 所指向的字符串转换为一个浮点数（类型为 double 型）。str – 要转换为双精度浮点数的字符串。endptr – 对类型为 char 的对象的引用，其值由函数设置为 str 中数值后的下一个字符。这个当第二个参数为空时和 double atof(const char \*str)是相同，但是当而第二形参引用后，第二个指针会指向存储字符串位置的地址。

5、long int strtol(const char \*str, char \*\*endptr, int base)。函数原型 long int strtol(const char \*str, char \*endptr, int base)。函数功能： 把参数 str 所指向的字符串转换为一个长整数（类型为 long int 型）。str – 要转换为长整数的字符串。endptr – 对类型为 char 的对象的引用，其值由函数设置为 str 中数值后的下一个字符。base – 基数，必须介于 2 和 36（包含）之间，或者是特殊值 0，进制数。

6、unsigned long int strtoul(const char \*str, char \*\*endptr, int base)。函数原型 unsigned long int strtoul(const char \*str, char \*\*endptr, int base)。函数功能： 把参数 str 所指向的字符串转换为一个无符号长整数（类型为 unsigned long int 型）。

7、void \*calloc(size\_t nitems, size\_t size)。函数原型 void \*calloc(size\_t nitems, size\_t size)。函数功能： 分配所需的内存空间，并返回一个指向它的指针。malloc 和 calloc 之间的不同点是，malloc 不会设置内存为零。

8、void free(void \*ptr)。函数原型 void free(void \*ptr)。函数功能：分配所需的内存空间，该函数不返回任何值。

9、void \*malloc(size\_t size)。函数原型 void \*malloc(size\_t size)。函数功能：分配所需的内存空间，并返回一个指向它的指针。该函数返回一个指针 ，指向已分配大小的内存。如果请求失败，则返回 NULL。

10、void \*realloc(void \*ptr, size\_t size)。函数原型 void \*realloc(void \*ptr, size\_t size)。函数功能： 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。该函数返回一个指针 ，指向重新分配大小的内存。如果请求失败，则返回 NULL。

11、void abort(void)。函数原型 void abort(void)。函数功能：使一个异常程序终止。

12、int atexit(void (\*func)(void))。函数原型 int atexit(void (\*func)(void))。函数功能：当程序正常终止时，调用指定的函数 func。

13、void exit(int status)函数原型 void exit(int status)。函数功能：使程序正常终止。

14、char \*getenv(const char \*name)。函数原型 char \*getenv(const char \*name)。函数功能：搜索 name 所指向的环境字符串，并返回相关的值给字符串。

15、int system(const char \*string)。函数原型 int system(const char \*string)。函数功能： 由 string 指定的命令传给要被命令处理器执行的主机环境。

16、void \*bsearch(const void \*key, const void \*base, size\_t nitems, size\_t size, int (\*compar)(const void \*, const void \*))。函数原型 void \*bsearch(const void \*key, const void \*base, size\_t nitems, size\_t size, int (\*compar)(const void , const void ))。key – 指向要查找的元素的指针，类型转换为 void 。base – 指向进行查找的数组的第一个对象的指针，类型转换为 void 。nitems – base 所指向的数组中元素的个数。size – 数组中每个元素的大小，以字节为单位。compar – 用来比较两个元素的函数。函数功能：执行二分查找。

17、void qsort(void \*base, size\_t nitems, size\_t size, int (\*compar)(const void , const void))。函数原型 void qsort(void \*base, size\_t nitems, size\_t size, int (\*compar)(const void , const void))。base – 指向要排序的数组的第一个元素的指针。nitems – 由 base 指向的数组中元素的个数。size – 数组中每个元素的大小，以字节为单位。compar – 用来比较两个元素的函数。函数功能：数组排序。

18、int abs(int x)。函数原型 int abs(int x)。函数功能：返回 x 的绝对值。

19、div\_t div(int numer, int denom)。函数原型 div\_t div(int numer, int denom)。函数功能：分子除以分母。

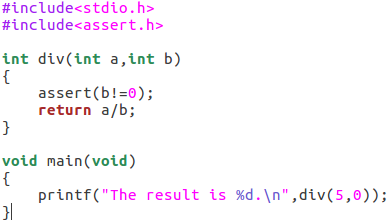
20、int rand(void)生成随机数。

21、void srand(unsigned int seed)。函数原型：int rand(void)。函数原型：void srand(unsigned int seed)。函数功能：返回一个范围在 0 到 RAND\_MAX 之间的伪随机数。函数功能：该函数播种由函数 rand 使用的随机数发生器。

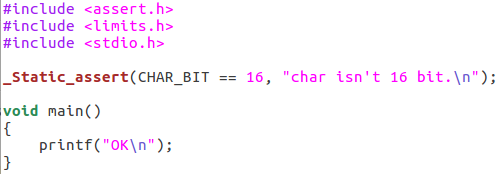
## 断言库

assert.h是一个用于辅助调试程序的小型库。

assert()宏接收一个整型表达式作为参数，如果表达式为假，就在标准错误stderr中写入一条错误信息，如测试名，文件名，行号等，并调用abort()终止程序。



**\_Static\_assert(C11)**，assert是在运行时检查，C11标准准新增\_Static\_assert声明，可以在编译时检查assert()表达式，assert会导致程序的终止，\_Static\_assert会导致编译的失败 \_Static\_assert接收两个参数，第一个参数是整型常量表达式，第二个参数是一个字符串，如果第一个表达式求值为假(0或\_False)，编译器会显示字符串，且不通过编译。



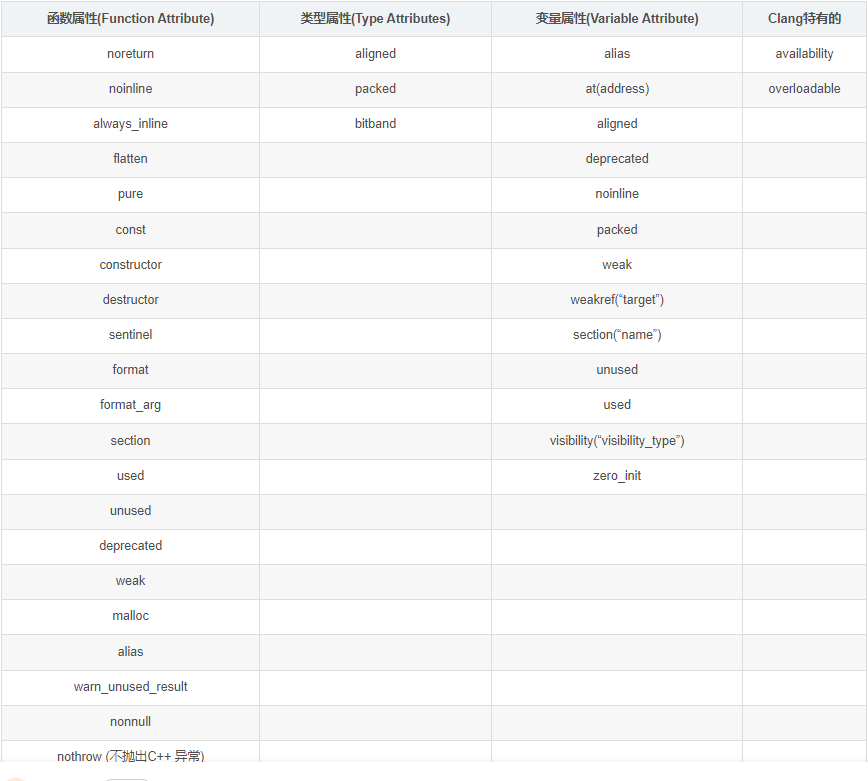
# 其他

## \_\_attribute\_\_()机制

**\_\_attribute 语法的来源，**GNU C 的一大特色就是\_\_attribute\_\_ 机制。attribute 可以设置函数属性（Function Attribute）、变量属性（Variable Attribute）和类型属性（Type Attribute）。

其位置约束为：放于声明的尾部“;” 之前。attribute 书写特征为: attribute 前后都有两个下划线，并切后面会紧跟一对原括弧，括弧里面是相应的\_\_attribute\_\_ 参数。attribute 语法格式为: attribute ((attribute-list))。当\_\_attribute\_\_ 用于修饰对象时，它就如同C 语言语法体系结构的类型限定符，跟const , volatile , restrict 等属一类。当\_\_attribute\_\_ 用于修饰函数时，它就相当于一个函数说明符，跟inline，Noreturn 属同一类。当\_\_attribute\_ 用于修饰一个结构体，联合体或者枚举类型，该限定符只能放在类型标识符之前。

**\_\_attribute 所支持的类型，**当我们需要识别当前编译器能否支持GNU 语法拓展，我们可以使用 \_\_GNU \_\_ 宏作为区分。



[https://blog.csdn.net/weaiken/article/details/88085360?ops\_request\_misc=%257B%2522request%255Fid%2522%253A%2522165678571316781818722535%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request\_id=165678571316781818722535&biz\_id=0&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~sobaiduend~default-2-88085360-null-null.142^v30^control,185^v2^control&utm\_term=\_\_attribute\_\_&spm=1018.2226.3001.4187](https://blog.csdn.net/weaiken/article/details/88085360?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522165678571316781818722535%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=165678571316781818722535&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduend~default-2-88085360-null-null.142%5ev30%5econtrol,185%5ev2%5econtrol&utm_term=__attribute__&spm=1018.2226.3001.4187)

## using namespace std

C++中的using namespace std是什么意思呢？为了理解其中的含义，我们首先需要学习一个概念叫做命名空间。

  我们都知道，C语言规定变量名不能与关键字冲突，但是并没有规定变量名不能与库中的变量名冲突，但这在实际的操作的过程中就会产生意想不到的错误：例如，将一个time变量放在main函数中，根据局部优先的原则，time变量就会被理解为一个局部变量。但当time作为一个全局变量出现的时候，由于time库中也含有一个名叫time的函数名，头文件被展开后，此time被理解为变量名还是函数名就会产生歧义。

using name std的作用是释放std命名空间中的变量名，函数名以及类型名，其中std是C++标准库的命名空间。