GCC编译C/C++的四个过程

预处理

```
gcc -E main.c -o main.i
```

编译阶段

```
gcc -S main.i -o main.s
```

汇编阶段

```
gcc -c main.s -o main.o
```

链接阶段

```
gcc main.o -o main.exe
```

关键字

关键字	描述
auto	声明自动变量
double	声明双精度变量或函数
typedef	用以给数据类型取别名
register	声明寄存器变量
short	声明短整型变量或函数
char	声明字符型变量或函数
const	声明只读变量
static	声明静态变量int声明整型变量或函数
struct	声明结构体变量或函数
unsigned	声明无符号类型变量或函数
volatile	说明变量在程序执行中可被隐含地改变long声明长整型变量或函数
union	声明共用数据类型
signed	声明有符号类型变量或函数
void	声明函数无返回值或无参数，声明无类型指针float声明浮点型变量或函数
enum	声明枚举类型
extern	声明变量是在其他文件正声明

关键字	描述
if	条件语句
else	条件语句否定分支（与 if 连用）
switch	用于开关语句
case	开关语句分支
for	一种循环语句do循环语句的循环体
while	循环语句的循环条件
goto	无条件跳转语句
continue	结束当前循环，开始下一轮循环break跳出当前循环
default	开关语句中的"其他"分支
sizeof	计算数据类型长度
return	子程序返回语句（可以带参数，也可不带参数）循环条件

C 中的变量声明

变量声明向编译器保证变量以指定的类型和名称存在，这样编译器在不需要知道变量完整细节的情况下也能继续进一步的编译。变量声明只在编译时有它的意义，在程序连接时编译器需要实际的变量声明。

变量的声明有两种情况：

- 1、一种是需要建立存储空间的。例如：int a 在声明的时候就已经建立了存储空间。
- 2、另一种是不需要建立存储空间的，通过使用extern关键字声明变量名而不定义它。 例如：extern int a 其中变量 a 可以在别的文件中定义的。
- 除非有extern关键字，否则都是变量的定义。

```
extern int i; //声明，不是定义
int i; //声明，也是定义
```

C 中的左值（Lvalues）和右值（Rvalues）

C 中有两种类型的表达式：

1. **左值（lvalue）**：指向内存位置的表达式被称为左值（lvalue）表达式。左值可以出现在赋值号的左边或右边。
2. **右值（rvalue）**：术语右值（rvalue）指的是存储在内存中某些地址的数值。右值是不能对其进行赋值的表达式，也就是说，右值可以出现在赋值号的右边，但不能出现在赋值号的左边。

C 常量

常量是固定值，在程序执行期间不会改变。这些固定的值，又叫做**字面量**。

整数常量

整数常量可以是十进制、八进制或十六进制的常量。前缀指定基数：0x 或 0X 表示十六进制，0 表示八进制，不带前缀则默认表示十进制。

整数常量也可以带一个后缀，后缀是 U 和 L 的组合，U 表示无符号整数（unsigned），L 表示长整数（long）。后缀可以是大写，也可以是小写，U 和 L 的顺序任意。

下面列举几个整数常量的实例：

```
212         /* 合法的 */
215u        /* 合法的 */
0xFeeL      /* 合法的 */
078         /* 非法的：8 不是八进制的数字 */
032UU       /* 非法的：不能重复后缀 */
```

以下是各种类型的整数常量的实例：

```
85         /* 十进制 */
0213       /* 八进制 */
0x4b       /* 十六进制 */
30         /* 整数 */
30u        /* 无符号整数 */
30l        /* 长整数 */
30ul       /* 无符号长整数 */
```

浮点常量

浮点常量由整数部分、小数点、小数部分和指数部分组成。您可以使用小数形式或者指数形式来表示浮点常量。

当使用小数形式表示时，必须包含整数部分、小数部分，或同时包含两者。当使用指数形式表示时， 必须包含小数点、指数，或同时包含两者。带符号的指数是用 e 或 E 引入的。

下面列举几个浮点常量的实例：

```
3.14159       /* 合法的 */
314159E-5L    /* 合法的 */
510E          /* 非法的：不完整的指数 */
210f          /* 非法的：没有小数或指数 */
.e55          /* 非法的：缺少整数或分数 */
```

字符常量

字符常量是括在单引号中，例如，'x' 可以存储在 **char** 类型的简单变量中。

字符常量可以是一个普通的字符（例如 'x'）、一个转义序列（例如 '\t'），或一个通用的字符（例如 '\u02C0'）。

在 C 中，有一些特定的字符，当它们前面有反斜杠时，它们就具有特殊的含义，被用来表示如换行符（\n）或制表符（\t）等。下表列出了一些这样的转义序列码：

转义序列	含义
\	\ 字符
'	' 字符
"	" 字符
\?	? 字符
\a	警报铃声
\b	退格键
\f	换页符
\n	换行符
\r	回车
\t	水平制表符
\v	垂直制表符
\ooo	一到三位的八进制数
\xhh . . .	一个或多个数字的十六进制数

下面的实例显示了一些转义序列字符：

实例

#include <stdio.h> int main() { printf("Hello\tWorld\n\n"); return 0; }

当上面的代码被编译和执行时，它会产生下列结果：

```
Hello   World
```

字符串常量

字符串字面值或常量是括在双引号 "" 中的。一个字符串包含类似于字符常量的字符：普通的字符、转义序列和通用的字符。

您可以使用空格做分隔符，把一个很长的字符串常量进行分行。

下面的实例显示了一些字符串常量。下面这三种形式所显示的字符串是相同的。

```
"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"
```

定义常量

在 C 中，有两种简单的定义常量的方式：

1. 使用 **#define** 预处理器。
2. 使用 **const** 关键字。

C 存储类

---

auto 是局部变量的默认存储类, 限定变量只能在函数内部使用；

register 代表了寄存器变量，不在内存中使用；

static是全局变量的默认存储类,表示变量在程序生命周期内可见；

extern 表示全局变量，即对程序内所有文件可见，类似于Java中的public关键字；

C 语言中全局变量、局部变量、静态全局变量、静态局部变量的区别

从作用域看：

1、全局变量具有全局作用域。全局变量只需在一个源文件中定义，就可以作用于所有的源文件。当然，其他不包含全局变量定义的源文件需要用extern 关键字再次声明这个全局变量。

2、静态局部变量具有局部作用域，它只被初始化一次，自从第一次被初始化直到程序运行结束都一直存在，它和全局变量的区别在于全局变量对所有的函数都是可见的，而静态局部变量只对定义自己的函数体始终可见。

3、局部变量也只有局部作用域，它是自动对象（auto），它在程序运行期间不是一直存在，而是只在函数执行期间存在，函数的一次调用执行结束后，变量被撤销，其所占用的内存也被收回。

4、静态全局变量也具有全局作用域，它与全局变量的区别在于如果程序包含多个文件的话，它作用于定义它的文件里，不能作用到其它文件里，即被static关键字修饰过的变量具有文件作用域。这样即使两个不同的源文件都定义了相同名字的静态全局变量，它们也是不同的变量。

从分配内存空间看：

1、全局变量，静态局部变量，静态全局变量都在静态存储区分配空间，而局部变量在栈里分配空间

2、全局变量本身就是静态存储方式， 静态全局变量当然也是静态存储方式。这两者在存储方式上并无不同。这两者的区别虽在于非静态全局变量的作用域是整个源程序，当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的。而静态全局变量则限制了其作用域，即只在定义该变量的源文件

内有效，在同一源程序的其它源文件中不能使用它。由于静态全局变量的作用域局限于一个源文件内，只能为该源文件内的函数公用，因此可以避免在其它源文件中引起错误。

- 1)静态变量会被放在程序的静态数据存储区(全局可见)中，这样可以在下一次调用的时候还可以保持原来的赋值。这一点是它与堆栈变量和堆变量的区别。
- 2)变量用static告知编译器，自己仅仅在变量的作用范围内可见。这一点是它与全局变量的区别。

从以上分析可以看出， 把局部变量改变为静态变量后是改变了它的存储方式即改变了它的生存期。把全局变量改变为静态变量后是改变了它的作用域，限制了它的使用范围。因此static 这个说明符在不同的地方所起的作用是不同的。应予以注意。

Tips:

- A.若全局变量仅在单个C文件中访问，则可以将这个变量修改为静态全局变量，以降低模块间的耦合度；
- B.若全局变量仅由单个函数访问，则可以将这个变量改为该函数的静态局部变量，以降低模块间的耦合度；
- C.设计和使用访问动态全局变量、静态全局变量、静态局部变量的函数时，需要考虑重入问题，因为他们都放在静态数据存储区，全局可见；
- D.如果我们需要一个可重入的函数，那么，我们一定要避免函数中使用static变量(这样的函数被称为：带"内部存储器"功能的的函数)
- E.函数中必须要使用static变量情况:比如当某函数的返回值为指针类型时，则必须是static的局部变量的地址作为返回值，若为auto类型，则返回为错指针

C 作用域规则

任何一种编程中，作用域是程序中定义的变量所存在的区域，超过该区域变量就不能被访问。C 语言中有三个地方可以声明变量：

1. 在函数或块内部的**局部**变量
2. 在所有函数外部的**全局**变量
3. 在**形式**参数的函数参数定义中

C 头文件

头文件是扩展名为 **.h** 的文件，包含了 C 函数声明和宏定义，被多个源文件中引用共享

只引用一次头文件

如果一个头文件被引用两次，编译器会处理两次头文件的内容，这将产生错误。为了防止这种情况，标准的做法是把文件的整个内容放在条件编译语句中，如下：

```
#ifndef HEADER_FILE
#define HEADER_FILE

the entire header file file

#endif
```

这种结构就是通常所说的包装器 **#ifndef**。当再次引用头文件时，条件为假，因为 HEADER_FILE 已定义。此时，预处理器会跳过文件的整个内容，编译器会忽略它。

有条件引用

有时需要从多个不同的头文件中选择一个引用到程序中。例如，需要指定在不同的操作系统上使用的配置参数。您可以通过一系列条件来实现这点，如下：

```
#if SYSTEM_1
   # include "system_1.h"
#elif SYSTEM_2
   # include "system_2.h"
#elif SYSTEM_3
   ...
#endif
```

但是如果头文件比较多的时候，这么做是很不妥当的，预处理器使用宏来定义头文件的名称。这就是所谓的**有条件引用**。它不是用头文件的名称作为 **#include** 的直接参数，您只需要使用宏名称代替即可：

```
 #define SYSTEM_H "system_1.h"
 ...
 #include SYSTEM_H
```

SYSTEM_H 会扩展，预处理器会查找 system_1.h，就像 **#include** 最初编写的那样。SYSTEM_H 可通过 -D 选项被您的 Makefile 定义。

C 预处理器

C 预处理器不是编译器的组成部分，但是它是编译过程中一个单独的步骤。简言之，C 预处理器只不过是一个文本替换工具而已，它们会指示编译器在实际编译之前完成所需的预处理。我们将把 C 预处理器（C Preprocessor）简写为 CPP。

所有的预处理器命令都是以井号（#）开头。它必须是第一个非空字符，为了增强可读性，预处理器指令应从第一列开始。下面列出了所有重要的预处理器指令：

指令	描述
#define	定义宏
#include	包含一个源代码文件
#undef	取消已定义的宏
#ifdef	如果宏已经定义，则返回真
#ifndef	如果宏没有定义，则返回真
#if	如果给定条件为真，则编译下面代码
#else	#if 的替代方案
#elif	如果前面的 #if 给定条件不为真，当前条件为真，则编译下面代码
#endif	结束一个 #if……#else 条件编译块
#error	当遇到标准错误时，输出错误消息
#pragma	使用标准化方法，向编译器发布特殊的命令到编译器中

预处理器运算符

C 预处理器提供了下列的运算符来帮助您创建宏：

宏延续运算符（\）

一个宏通常写在一个单行上。但是如果宏太长，一个单行容纳不下，则使用宏延续运算符（\）。例如：

```
#define  message_for(a, b)  \
    printf(#a " and " #b ": We love you!\n")
```

字符串常量化运算符（#）

在宏定义中，当需要把一个宏的参数转换为字符串常量时，则使用字符串常量化运算符（#）。在宏中使用的该运算符有一个特定的参数或参数列表。例如：

```
#include <stdio.h>

#define  message_for(a, b)  \
    printf(#a " and " #b ": We love you!\n")

int main(void)
{
   message_for(Carole, Debra);
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Carole and Debra: We love you!
```

标记粘贴运算符（##）

宏定义内的标记粘贴运算符（##）会合并两个参数。它允许在宏定义中两个独立的标记被合并为一个标记。例如：

```
#include <stdio.h>

#define tokenpaster(n) printf ("token" #n " = %d",
token##n)

int main(void)
{
   int token34 = 40;

   tokenpaster(34);
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
token34 = 40
```

这是怎么发生的，因为这个实例会从编译器产生下列的实际输出：

```
printf ("token34 = %d", token34);
```

这个实例演示了 token##n 会连接到 token34 中，在这里，我们使用了**字符串常量化运算符（#）**和**标记粘贴运算符（##）**。

defined() 运算符

预处理器 **defined** 运算符是用在常量表达式中的，用来确定一个标识符是否已经使用 #define 定义过。如果指定的标识符已定义，则值为真（非零）。如果指定的标识符未定义，则值为假（零）。下面的实例演示了 defined() 运算符的用法：

```
#include <stdio.h>

#if !defined (MESSAGE)
   #define MESSAGE "You wish!"
#endif

int main(void)
{
   printf("Here is the message: %s\n", MESSAGE);
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Here is the message: You wish!
```

参数化的宏

CPP 一个强大的功能是可以使用参数化的宏来模拟函数。例如，下面的代码是计算一个数的平方：

```
int square(int x) {
   return x * x;
}
```

我们可以使用宏重写上面的代码，如下：

```
#define square(x) ((x) * (x))
```

在使用带有参数的宏之前，必须使用 **#define** 指令定义。参数列表是括在圆括号内，且必须紧跟在宏名称的后边。宏名称和左圆括号之间不允许有空格。例如：

```
#include <stdio.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void)
{
   printf("Max between 20 and 10 is %d\n", MAX(10, 20));
   return 0;
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Max between 20 and 10 is 20
```

C typedef

C 语言提供了 **typedef** 关键字，您可以使用它来为类型取一个新的名字。下面的实例为单字节数字定义了一个术语 **BYTE**：

```
typedef unsigned char BYTE;
```

在这个类型定义之后，标识符 BYTE 可作为类型 **unsigned char** 的缩写，例如：

```
BYTE  b1, b2;
```

typedef vs #define

#define 是 C 指令，用于为各种数据类型定义别名，与 **typedef** 类似，但是它们有以下几点不同：

- **typedef** 仅限于为类型定义符号名称，**#define** 不仅可以为类型定义别名，也能为数值定义别名，比如您可以定义 1 为 ONE。
- **typedef** 是由编译器执行解释的，**#define** 语句是由预编译器进行处理的。

C 中的运算符优先级

类别	运算符	结合性
后缀	() [] -> . ++ - -	从左到右
一元	+ - ! ~ ++ - - (type)* & sizeof	从右到左
乘除	* / %	从左到右
加减	+ -	从左到右
移位	<< >>	从左到右
关系	< <= > >=	从左到右
相等	== !=	从左到右
位与 AND	&	从左到右
位异或 XOR	^	从左到右
位或 OR	|	从左到右
逻辑与 AND	&&	从左到右
逻辑或 OR	||	从左到右
条件	?:	从右到左
赋值	= += -= *= /= %=>>= <<= &= ^= |=	从右到左
逗号	,	从左到右

运算符优先级：

```
括号成员是老大;        // 括号运算符 []() 成员运算符.  ->

全体单目排老二;        // 所有的单目运算符比如++、 --、 +(正)、 -(负) 、指针运算*、&

乘除余三,加减四;      // 这个"余"是指取余运算即%

移位五，关系六;       // 移位运算符：<< >> ，关系：> < >= <= 等

等与不等排行七;       // 即 == 和 !=

位与异或和位或;       // 这几个都是位运算：位与(&)异或(^)位或(|)


"三分天下"八九十;
```

```
逻辑与，逻辑或；      // 逻辑运算符：|| 和 &&

十一十二紧挨着；      // 注意顺序：优先级(||)  底于  优先级(&&)

条件只比赋值高，      // 三目运算符优先级排到 13 位只比赋值运算符和 "," 高

逗号运算最低级！      //逗号运算符优先级最低
```

指针和数组

C语言标准对此作了说明：

规则1：表达式中的数组名被编译器当做一个指向该数组第一个元素的指针；

注：下面几种情况例外

1)数组名作为sizeof的操作数

2)使用&取数组的地址

规则2：下标总是与指针的偏移量相同；

规则3：在函数参数的声明中，数组名被编译器当做指向该数组第一个元素的指针。

规则1和规则2结合在一起理解，就是对数组下标的引用总是可以写成“一个指向数组的起始地址的指针加上偏移量”。如a[i]总是被编译器解析为*(a+i)的形式。

&p，p，a，&a

&p：表示取存储指针变量p的内存单元的地址； sizeof(&p)=4;

p：表示取指针变量p存储的地址； sizeof(p)=4;

a：表示取数组第一个元素的地址； sizeof(a)=3*4=12;

&a：表示取整个数组的首地址； sizeof(&a)=4

```
 int array[5] = {0};
    printf("        array = %p\n", array);
    printf("       &array = %p\n", &array);
    printf("    array + 1 = %p\n", array + 1);
    printf("&array[0] + 1 = %p\n", &array[0] + 1);
    printf("   &array + 1 = %p\n", &array + 1);
    printf("\n");
    printf(" sizeof(int*) = %d\n", sizeof(int*));
    printf(" sizeof(array[0]) = %d\n", sizeof(&array[0]));
    printf(" sizeof(0xffffcc04) = %d\n",
sizeof(0xffffcc04));
    printf(" sizeof(array) = %d\n", sizeof(array));//数组类
型的长度
    printf("sizeof(&array) = %d\n", sizeof(&array));//计算
的是指针的长度
```

从两个例子分析C语言的声明

在C语言中，声明的形式和使用的形式相似，这种用法可能是C语言的独创，K & R也承认"C语言声明的语法有时候会带来严重的问题"。C语言的声明存在的最大问题是你无法以一种人们所习惯的自然方式从左到右阅读一个声明。下面看一个例子：

```
char * const *(*next)();
```

如果在第一眼就能看出这个声明要表达的意思，那么证明你的C语言功底已经到了一定的程度。《C专家编程》一书中给出的识别步骤为：

1)从变量名next开始，并注意到它直接被括号括住；

2)所以先把括号里的东西作为一个整体，得出"next”是一个指向....的指针；

3)然后考虑括号外面的东西，在星号前缀和括号后缀之间做一个选择；

4)根据C语言声明的优先级规则(后面会给出)，优先级较高的是右边的函数括号，所以得出"next"是一个函数指针，指向一个返回...的函数；

5)然后，，处理前缀"*"，得出指针所指的内容；

6)最后，把"char * const *"解释为指向字符串的常量指针。

把上述结果加以概括，这个声明表示"next是一个指针，它指向一个函数，这个函数返回另一个指针，该指针指向一个类型为char的常量指针"。这个问题便迎刃而解了.

下面再看一个例子：

```
char *(* c[10])(int **p);

void (*signal(int sig,void(*func)(int)))(int)
```

首先，从变量名c开始，然后处理后缀"[]"，表明c是一个数组，接着处理前缀"*"，表示c是一个指针数组。然后处理后面的括号，表明数组c中的指针类型是指向一个函数的指针，并且这个函数的参数有且仅有一个：为指向指针的指针，该函数的返回值为一个指向字符串的指针。归纳在一起，为：

"c是一个数组[0...9]，它的元素类型是函数指针，其所指向的函数返回值是一个指向字符串的指针，并且把一个指向指针的指针作为唯一的参数"。

以下是《C专家编程》一书中提到的C语言声明的优先级规则，摘自第64页。

eg

```
数组指针：
    int (*p)[5] 这里的p是一个指针，指向一个具有5个元素的数组指针
指针数组：
    int *p[5] 这里的p是一个数组，数组中的元素类型是int*
```

C语言——常量指针和指针常量的区别

常量指针：表示const修饰的为所申明的类型。 例如：

```
//注意char const *p与const char *p效果相同。
void consttest(const char *p)
{
    printf("p[1]=%c\n",p[1]);
    p=1;//正确
    *(p+1)='a';//错误
}
```

因为const修饰的是char，所以就是说：p所指向的内存地址所对应的值，是const，因此不可修改。但指针所指向的内存地址是可以修改的，因为其并不是const类型。

指针常量：表示const修改的指针。

例如：

```
void testconst(char *const p)
{
    char *tmp="13213";
    p=1;;//错误
    p=tmp;;//错误
    p[1]='a';//正确
    *(p+1)='a';//正确
}
```

因为const修饰的是指针p，也就是说：指针所指向的内存地址是const，不可修改。但p所指向内存地址所对应的值是可以修改的，因为其并不是const类型。

指向常量的指针常量: const同时修饰类型和指针。只读 例如：

```
void consttestconst(const char *const p) //引用
{
    p=1;//错误
    p[1]='a';//错误
}
```

因为const同时修饰这类型和指针，也就是说：指针所指向的内存地址不可修改同时内存地址所对应的值也不可修改。

总结：主要看的就是const所处的位置。

> *1)const 在前：表示const修饰的为所申明的类型。常量指针*
>
> *2)const 在后：表示const修饰的为指针。指针常量*
>
> *3)前*后均有：表示const同时修改类型和指针。指向常量的指针常量*

const的优点 在C/C++中关键字const用来定义一个只读的变量或者对象，有如下优点：

> *（1）便于类型检查，如函数的函数fun（const int a） a的值不允许变，这样便于保护实参。（2）功能类似与宏定义，方便参数的修改和调整。如const int max = 100；（3）节省空间，如果再定义a = max,b=max。。。就不用在为max分配空间了，而用宏定义的话就一直进行宏替换并为变量分配空间（4）为函数重载提供参考*

常量指针和指针常量的区别 下面通过一个例子来解析 常量指针 和 指针常量，我们先总结一下 常量指针 和 指针常量 的区别

首先一定要明白哪种定义方式是常量指针，哪种是指针常量，这里可以记住三句话加深记忆：

> *（指针）和const（常量） 谁在前先读谁；*象征着地址，const象征着内容；谁在前面谁就不允许改变。*

好吧，让我们来看这个例子：

```
int a =3;
int b = 1;
int c = 2;
int const *p1 = &b;//const 在前，定义为常量指针
int *const p2 = &c;//*在前，定义为指针常量
```

常量指针p1：指向的地址可以变，但内容不可以重新赋值，内容的改变只能通过修改地址指向后变换。

> *p1 = &a是正确的，但 *p1 = a是错误的。*

指针常量p2：指向的地址不可以重新赋值，但内容可以改变，必须初始化，地址跟随一生。

> *p2= &a是错误的，而*p2 = a 是正确的。*

关于常量指针,指针常量的知识点补充 1.概念： 常量指针是指–指向常量的指针，顾名思义，就是指针指向的是常量，即，它不能指向变量，它指向的内容不能被改变，不能通过指针来修改它指向的内容，但是指针自身不是常量，它自身的值可以改变，从而指向另一个常量。 指针常量是指–指针本身是常量。它指向的地址是不可改变的，但地址里的内容可以通过指针改变。它指向的地址将伴其一生，直到生命周期结束。有一点需要注意的是，指针常量在定义时必须同时赋初值。 注：也有人将这两个名称的定义与含义反过来认为：“指针常量：顾名思义它的中心词是“常量”这是重点，指针就是一个修饰的作用。所以这里的指针还是一个变量，它的内容存放的是常量的地址。常量指针：关键字是指针，它是不能被改变的，因为指针总是指向地址的，所以它的意思是它指向的地址是不能被改变的”。但我个人认为后者不合理，所以使用前者。 2.使用方法： 使用时写法上的区别：常量指针：const在之前 指针常量：const在之后。 当然我们也可以定义常量指针常量，那就需要加上两个const，一前一后！以上只是从定义上给出两者的本质上的不同，在具体使用上，还有很多变化，但万变不离其宗，我们可以根据它的原理分析出各种复杂用法的实质。

3.使用技巧 使用指针常量可以增加代码的可靠性和执行效率。 如：

```
Int a;
Int * const p =&a;
```

增加可靠性：不用担心p被修改或释放导致非预期结果； 增加执行效率：不用在子函数中对p做为空检查可以提高效率。

C 文件读写

打开文件

```
FILE *fopen( const char * filename, const char * mode );
```

模式	描述
r	打开一个已有的文本文件，允许读取文件。
w	打开一个文本文件，允许写入文件。如果文件不存在，则会创建一个新文件。在这里，您的程序会从文件的开头写入内容。如果文件存在，则该会被截断为零长度，重新写入。
a	打开一个文本文件，以追加模式写入文件。如果文件不存在，则会创建一个新文件。在这里，您的程序会在已有的文件内容中追加内容。
r+	打开一个文本文件，允许读写文件。

模式	描述
w+	打开一个文本文件，允许读写文件。如果文件已存在，则文件会被截断为零长度，如果文件不存在，则会创建一个新文件。
a+	打开一个文本文件，允许读写文件。如果文件不存在，则会创建一个新文件。读取会从文件的开头开始，写入则只能是追加模式。

> *b:以二进制方式打开文件*
>
> *t:以文本方式打开文件(默认方式下以文本方式打开文件)*

字符读写

字符读写主要使用两个函数fputc和fgetc，两个函数的原型是:

```
int fputc(int ch,FILE *fp);//若写入成功则返回写入的字符，否则返回-1
int fgetc(FILE *fp);  //若读取成功则返回读取的字符，否则返回-1
```

字符串读写

```
//将字符串写入文件，若写入成功则返回一个非负值，否则返回-1;
int fputs(const char *s,FILE *fp);
//从文件中读取不超过n-1个字符到字符数组中(若文件中字符少于n-1个，则只读取文件中存在的字符)，系统在字符数组末尾自动添加一个'\0'，返回字符数组的首地址
//对于fgets函数，在读取过程中，若读取到字符'\n'，则读取过程提前结束
char *fgets(char *s,int n,FILE *fp);
```

块读写

```
//从文件读取一组数据存放在首地址为buffer的内存空间中，size为一个数据块的大小，n为要读取的数据块的个数，若读取成功，则返回读取的数据的数据块的个数，否则返回0.
unsigned int fread(void *buffer,unsigned int size,unsigned int n,FILE *fp);
//向文件中写入数据，写入成功返回写入数据块的个数，否则返回0.
 unsigned int fwrite(const void *buffer,unsigned int size,unsigned int n,FILE *fp);
```

格式化读写

- 格式化读写和其他几种读写有很大的不同。格式化读写是以我们人所能识别的格式将数据写入文件，即若以格式化方式写入一个整型数值65，则其实是写入的两个字符'6'和'5'，即占2字节，而不是4字节，但是若以块写方式写入，则其占4字节。即在使用格式化读写时系统自动进行了一些转换。

- fprintf和fscanf函数一般成对出现，若数据是用fprintf进行写入的，则最好使用fscanf进行读取。
- 在使用fprintf函数写入时，若文件是以文本方式打开，如果参数format中包含了'\n'，则最后文件中会被写入换行符；而若文件以二进制方式打开，则文件中不会被写入换行符

```
//用于从文件格式化读取数据，若读取成功，则返回读取的数据个数，否则返回-1
int fscanf(FILE *fp,const char *format[,argument]....);
//用于向文件格式化写入数据，若写入成功，则返回写入的字符个数，否则返回-1
int fprintf(FILE *fp,const char *format[,argument]....);
//从字符串读取格式化输入。
int sscanf(const char *str, const char *format, ...)
//发送格式化输出到 str 所指向的字符串
int sprintf(char *str, const char *format, ...)
```

其他几个常见的操作

移动位置指针的函数

```
// 将位置指针移动到文件首
void rewind(FILE *fp);
//将位置指针移动到距离origin的offset字节数的位置
int fseek(FILE *fp,long int offset,int origin);
```

> *origin的值有三个: SEEK_SET(0) 文件首, SEEK_CUR(1) 当前位置, SEEK_END(2) 文件尾 若文件是以**a**追加方式打开，则当进行写操作时，这两个函数是不起作用的，无论将位置指针移动哪个位置，始终将添加的数据追加到文件末尾*
>
> *只有用**r+** 模式打开文件才能插入内容，**w** 或 **w+** 模式都会清空掉原来文件的内容再来写，**a** 或 **a+** 模式即总会在文件最尾添加内容，哪怕用 fseek() 移动了文件指针位置*

stat函数

```
//通过文件名filename获取文件信息，并保存在buf所指的结构体stat中
#include<sys/stat.h>
int stat(const char *filename,struct stat *_stat)

//返回的结构体
struct stat {
        mode_t     st_mode;       //文件对应的模式，文件，目录等
        ino_t      st_ino;       //inode节点号
        dev_t      st_dev;        //设备号码
        dev_t      st_rdev;       //特殊设备号码
        nlink_t    st_nlink;      //文件的连接数
        uid_t      st_uid;        //文件所有者
```

```
    gid_t     st_gid;       //文件所有者对应的组
    off_t     st_size;      //普通文件，对应的文件字节数
    time_t    st_atime;     //文件最后被访问的时间
    time_t    st_mtime;     //文件内容最后被修改的时间
    time_t    st_ctime;     //文件状态改变时间
    blksize_t st_blksize;   //文件内容对应的块大小
    blkcnt_t  st_blocks;    //伟建内容对应的块数量
};
```

ftell函数

```
//计算当前位置指针距文件首的字节数，若出错，则返回-1L。
//利用ftell函数可以计算出文件的大小。
long int ftell(FILE *fp);
```

feof函数

```
//检测当前位置指针是否到达文件末尾，若到达文件末尾，则返回一个非零值，否则返回0。
int feof(FILE *fp);
```

ferror函数

```
//检测文件操作过程中是否出错，若出错，则返回一个非零值，否则返回0
int ferror(FILE *fp);
```

remove函数

```
//删除文件，若删除成功，则返回0，否则返回非零值
int remove(const char *filename);
```

rename函数

```
//将文件重命名，重命名成功则返回0，否则返回非零值。
int rename(const char *oldname,const char *newname);
```

freopen函数

```
//实现重定向输入输出。此函数在测试数据时用得比较多。
FILE* freopen(const char *filename,const char *mode,FILE *stream);
```

fclose函数

```
//关闭一个流，若成功，则返回0，否则返回-1.注意每次对文件操作完之后需关闭流，否则可能会造成数据丢失。
int fclose(FILE *stream);
```

fflush函数

- 由于fflush是实时的将缓冲区的内容写入磁盘，所以不要大量去使用，
- 只要特别敏感的数据，需要及时写入防止丢失,比如密码

```
//可以将缓冲区中任何未写入的数据写入文件中,成功返回0，失败返回EOF
int fflush(FILE *_file);
```