

C语言笔记

C语言笔记

C语言发展历程

历史介绍及环境搭建

C的发展与版本-K&R

C的发展与版本-标准

C语言用在哪里？

C语言编译软件（IDE）

C语言基础

第一章 基础语法

- 1、输出函数
- 2、运算符
- 3、变量定义
- 4、变量的名字
- 5、变量初始化
- 6、变量类型
- 7、常量
- 8、转义字符
- 9、输入函数
- 10、运算符优先级和结合性
- 11、一些容易出错的优先级问题
- 12、不同类型数据间的混合运算

第二章、控制语句

1. 字符输入输出函数
2. 选择结构嵌套
3. switch多分支

第三章、数组【一维数组、多维数组】

1. 一维数组
2. 二维数组
3. 字符数组

第四章、函数实现模块化设计

1. 定义函数的方法
2. 调用函数
3. 函数的返回值
4. 对被调用函数的声明和函数原型
5. 数组作为函数参数
6. 局部变量和全局变量
7. 变量的存储方式和生存期
8. 存储类别小结
9. 内部函数和外部函数

第五章、指针【C语言的灵魂】

1. 指针及指针变量【概念、定义】
2. 引用指针变量
3. 通过指针引用数组
4. 通过指针引用字符串
5. 指向函数的指针
6. 返回指针值的函数
7. 指针数组和多重指针
8. 动态内存分配与指向它的指针变量
9. 有关指针的小结

第六章 自定数据类型

1. 定义和使用结构体变量
2. 使用结构体数组

- 3. 结构体指针
 - 4. 用指针处理链表
 - 5. 共同体类型
 - 6. 使用枚举类型
 - 7. 用typedef声明新类型名
- 第七章 对文件的输入输出
- 1. C文件的有关基本知识
 - 2. 打开与关闭文件
 - 3. 顺序读写数据文件

C语言发展历程

历史介绍及环境搭建

- C语言是从B语言发展而来的，B语言是从 BCPL 发展而来的，BCPL 是从 FORTRAN 发展而来的。
- BCPL 和B都支持指针间接方式，所以C也支持了。
- C语言还受到了 PL/1 的影响，还和 PDP-11 的机器语言有很大的关系。
- 1973年3月，第三版的 Unix 上出现了C语言的编译器。
- 1973年11月，第四版的 Unix (System Four) 发布了，这个版本是完全用C语言重新编写的。

C的发展与版本-K&R

- 经典 C ----> 又被叫做 “K&R the C”
- The C Programming Language, by Brian Kernighan and Dennis Ritchie, 2nd Edition, Prentice Hall

C的发展与版本-标准

- 1989年ANSI发布了一个标准——ANSI C
- 1990年ISO接受了ANSI的标准——C89
- C的标准在1995年和1999年两次更新——C95和C99
- 所有的当代编译器都支持C99了

C语言用在哪里？

- 操作系统
- 嵌入式系统
- 驱动程序
- 底层驱动
- 图形引擎、图像处理、声音效果

C语言编译软件 (IDE)

- Dev C++ (4.9 for Win7, 5.0 for Win8)
- MS Visual Studio Express (Windows)
- Xcode (Mac OS X)
- Eclipse-CDT
- Geany (和MinGW一起)
- Sublime (和MinGW一起)
- vim/emacs (和MinGW一起)

C语言基础

编写代码注意：代码里的符号一定要是英文状态下的标点符号！！！，不要用中文！！！！

第一章 基础语法

1、输出函数

```
printf("Hello world!\n");
```

"" 里面的内容叫做“字符串”，printf会把其中的内容原封不动地输出
\\n 表示需要在输出的结果后面换一行。

补充：

- 如果你在使用Dev C++ 4.9.9.2

```
system("pause");
```
- 让程序运行完成后，窗口还能留下
- 不是Dev C++ 4.9.9.2就不需要这个了

2、运算符

算数运算符

C符号	意义
+	加
-	减
*	乘
/	除
%	取余
()	括号
++	自增
--	自减

关系运算符

C符号	意义
>	大于
<	小于
==	等于
>=	大于等于
<=	小于等于
!=	不等于

逻辑运算符

C符号	意义
!	非
&&	与
	或

位运算符：【>>,<<~,|,^,&】

赋值运算符：【=】

条件运算符（三元运算符）：【常量表达式? 返回值1:返回值2;】

逗号运算符：【,】

指针运算符：【*,&】

求字节数运算符：【sizeof】

3、变量定义

变量定义的一般形式就是：<类型名称> <变量名称>;

例如：int a;

4、变量的名字

- 变量需要一个名字，变量的名字是一种“标识符”，意思是它是用来识别这个和那个的不同的名字。
- 标识符有标识符的构造规则。基本的原则是：标识符只能由字母、数字和下划线组成，数字不可以出现在第一个位置上，C语言的关键字（有的地方叫它们保留字）不可以用做标识符。

C语言的保留字：

auto、break、case、char、const、continue、default、do、double、else、enum、extern、float、for、goto、if、int、long、register、return、short、signed、sizeof、static、struct、switch、typedef、union、unsigned、void、volatile、while、inline、restrict

赋值和初始化

```
int price = 0;
```

这一行，定义了一个变量。变量的名字是 price，类型是 int，初始值是 0。

这里的“=”是一个赋值运算符，表示将“=”右边的值赋给左边的变量。

5、变量初始化

如果没有进行初始化变量，直接使用，运算出来的结果会是一个很奇怪的值。

格式为：<类型名称> <变量名称> = <初始值>;

当赋值发生在定义变量的时候，就像程序1中的第7行那样，就是变量的初始化。虽然C语言并没有强制要求所有的变量都在定义的地方做初始化，但是所有的变量在第一次被使用（出现在赋值运算符的右边）之前被应该赋值一次。

6、变量类型

C是一种有类型的语言，所有的变量在使用之前必须定义或声明，所有的变量必须具有确定的数据类型。数据类型表示在变量中可以存放什么样的数据，变量中只能存放指定类型的数据，程序运行过程中也不能改变变量的类型。

六种基本数据类型。

变量类型	说明	字节大小
char	字符型类型	1
short	短整型类型	2
int	整型类型	4
long	长整型类型	4or8
float	单精度浮点类型	4
double	双精度浮点类型	8

- `signed`:有符号, 可省略
- `unsigned`:无符号

7、常量

C99允许使用**常量**, 方法是在定义变量时, 前面加一个关键字 `const`。

符号常量 `#define`: 用法 `#define 常量名 值`, 注意行末没有分号。

8、转义字符

转义字符及其作用

转义字符	含义	ASCII码 (16/10进制)
<code>\0</code>	空字符(NULL)	00H/0
<code>\n</code>	换行符(LF)	0AH/10
<code>\r</code>	回车符(CR)	0DH/13
<code>\t</code>	水平制表符(HT)	09H/9
<code>\v</code>	垂直制表符(VT)	0B/11
<code>\a</code>	响铃(BEL)	07/7
<code>\b</code>	退格符(BS)	08H/8
<code>\f</code>	换页符(FF)	0CH/12
<code>\'</code>	单引号	27H/39
<code>\"</code>	双引号	22H/34
<code>\\</code>	反斜杠	5CH/92
<code>\?</code>	问号字符	3F/63
<code>\ddd</code>	任意字符	三位八进制
<code>\xhh</code>	任意字符	二位十六进制

9、输入函数

`scanf("格式控制字符串", 地址表列);`

格式控制字符串的作用与 `printf` 函数相同, 但不能显示非格式字符串, 也就是不能显示提示字符串。地址表列中给出各变量的地址。地址是由地址运算符 `&` 后跟变量名组成的。

例如:

```
1  #include <stdio.h>
2  int main(void){
3      int a,b,c;
4      printf("input a,b,c\n");
5      scanf("%d%d%d",&a,&b,&c);
6      printf("a=%d,b=%d,c=%d",a,b,c);
7      return 0;
8  }
```

**** 格式字符串****

格式字符串的一般形式为：

`%[*][输入数据宽度][长度]类型`

其中有方括号[]的项为任选项。各项的意义如下。

类型

表示输入数据的类型，其格式符和意义如下表所示。

格式	字符意义
%d	输入十进制整数
%o	输入八进制整数
%x	输入十六进制整数
%u	输入无符号十进制整数
%f或%e	输入实型数(用小数形式或指数形式)
%c	输入单个字符
%s	输入字符串

使用scanf函数还须注意以下几点：

- scanf函数中没有精度控制，如：scanf("%5.2f",&a);是非法的。不能企图用此语句输入小数为2位的实数。
- scanf中要求给出变量地址，如给出变量名则会出错。如 scanf("%d",a);是非法的，应改为 scanf("%d",&a);才是合法的。
- 在输入多个数值数据时，若格式控制串中没有非格式字符作输入数据之间的间隔则可用空格，TAB或回车作间隔。C编译在碰到空格，TAB，回车或非法数据(如对"%d"输入"12A"时，A即为非法数据)时即认为该数据结束。
- 在输入字符数据时，若格式控制串中无非格式字符，则认为所有输入的字符均为有效字符。

printf格式附加字符

字符	说明
l	长整型整数，可加在格式符d、o、x、u前面
m(代表一个正整数)	数据最小宽度
n(代表一个正整数)	对实数，表示输出n位小数；对字符串，表示截取的字符个数；
-	输出的数字或字符域内向左靠

一般形式为：`% 附加字符 格式字符`

scanf格式附加字符

字符	说明
l	输入长整型数据（可用%ld，%lo，%lx，%lu）以及double型数据（用%lf或%le）
h	输入短整型数据（可用%hd，%ho，%hx）
域宽	指定输入数据所占宽度（列数），宽域应为正整数
*	本输入项在读入后不赋给相应变量

10、运算符优先级和结合性

优先级	运算符	名称或含义	使用形式	结合方向	说明
1	[]	数组下标	数组名[常量表达式]	左到右	
	()	圆括号	(表达式) 函数名(形参表)		
	.	成员选择（对象）	对象.成员名		
	->	成员选择（指针）	对象指针->成员名		
2	-	负号运算符	-表达式	右到左	单目运算符
	(类型)	强制类型转换	(数据类型)表达式		
	++	自增运算符	++变量名 变量名++		单目运算符
	--	自减运算符	--变量名 变量名--		单目运算符
	*	取值运算符	*指针变量		单目运算符
	&	取地址运算符	&变量名		单目运算符
	!	逻辑非运算符	!表达式		单目运算符
	~	按位取反运算符	~表达式		单目运算符
	sizeof	长度运算符	sizeof(表达式)		
3	/	除	表达式 / 表达式	左到右	双目运算符
	*	乘	表达式*表达式		双目运算符
	%	余数（取模）	整型表达式%整型表达式		双目运算符
4	+	加	表达式+表达式	左到右	双目运算符
	-	减	表达式-表达式		双目运算符
5	<<	左移	变量<<表达式	左到右	双目运算符
	>>	右移	变量>>表达式		双目运算符

6	>	大于	表达式>表达式	左到右	双目运算符
	>=	大于等于	表达式>=表达式		双目运算符
	<	小于	表达式<表达式		双目运算符
	<=	小于等于	表达式<=表达式		双目运算符
7	==	等于	表达式==表达式	左到右	双目运算符
	!=	不等于	表达式!=表达式		双目运算符
8	&	按位与	表达式&表达式	左到右	双目运算符
9	^	按位异或	表达式^表达式	左到右	双目运算符
10		按位或	表达式 表达式	左到右	双目运算符

11	&&	逻辑与	表达式&&表达式	左到右	双目运算符
12		逻辑或	表达式 表达式	左到右	双目运算符
13	?:	条件运算符	表达式1? 表达式2: 表达式3	右到左	三目运算符
14	=	赋值运算符	变量=表达式	右到左	
	/=	除后赋值	变量/=表达式		
	=	乘后赋值	变量=表达式		
	%=	取模后赋值	变量%=表达式		
	+=	加后赋值	变量+=表达式		
	-=	减后赋值	变量-=表达式		
	<<=	左移后赋值	变量<<=表达式		
	>>=	右移后赋值	变量>>=表达式		
	&=	按位与后赋值	变量&=表达式		
	^=	按位异或后赋值	变量^=表达式		
	=	按位或后赋值	变量 =表达式		
15	,	逗号运算符	表达式,表达式,...	左到右	

上表中可以总结出如下规律：

1. 结合方向只有三个是从右往左，其余都是从左往右。
2. 所有双目运算符中只有赋值运算符的结合方向是从右往左。
3. 另外两个从右往左结合的运算符也很好记，因为它们很特殊：一个是单目运算符，一个是三目运算符。

4. C语言中有且只有一个三目运算符。
5. 逗号运算符的优先级最低，要记住。
6. 此外要记住，对于优先级：算术运算符 > 关系运算符 > 逻辑运算符 > 赋值运算符。逻辑运算符中“逻辑非 !”除外。

11、一些容易出错的优先级问题

上表中，优先级同为1的几种运算符如果同时出现，那怎么确定表达式的优先级呢？这是很多初学者迷糊的地方。下表就整理了这些容易出错的情况：

优先级问题	表达式	经常误认为的结果	实际结果
. 的优先级高于 * (-> 操作符用于消除这个问题)	<code>*p.f</code>	p 所指对象的字段 f，等价于： <code>(*p).f</code>	对 p 取 f 偏移，作为指针，然后进行解除引用操作，等价于： <code>*(p.f)</code>
<code>[]</code> 高于 <code>*</code>	<code>int</code> <code>*ap[]</code>	ap 是个指向 int 数组的指针，等价于： <code>int (*ap)[]</code>	ap 是个元素为 int 指针的数组，等价于： <code>int *(ap[])</code>
函数 () 高于 <code>*</code>	<code>int *fp()</code>	fp 是个函数指针，所指函数返回 int，等价于： <code>int (*fp)()</code>	fp 是个函数，返回 int <code>*</code> ，等价于： <code>int*(fp())</code>
<code>==</code> 和 <code>!=</code> 高于位操作	<code>(val & mask != 0)</code>	<code>(val & mask) != 0</code>	<code>val & (mask != 0)</code>
<code>==</code> 和 <code>!=</code> 高于赋值符	<code>c =</code> <code>getchar()</code> <code>!= EOF</code>	<code>(c = getchar()) != EOF</code>	<code>c = (getchar() != EOF)</code>
算术运算符高于位移运算符	<code>msb << 4 + lsb</code>	<code>(msb << 4) + lsb</code>	<code>msb << (4 + lsb)</code>
逗号运算符在所有运算符中优先级最低	<code>i = 1, 2</code>	<code>i = (1,2)</code>	<code>(i = 1), 2</code>

12、不同类型数据间的混合运算

- (1) `+`、`-`、`*`、`/`、运算符两侧中有一个为 float 或 double 型，结果都为 double 型数据。
- (2) 如果 int 型与 float 型数据进行运算，会先把 int 型和 float 型数据转换为 double 型，然后再进行运算，结果是 double 型
- (3) 字符 (char) 型数据与整形数据进行运算，就是把字符型数据的 ASCII 代码与整形数据进行运算。如： `12+'A'` 等效于 `12+65` 结果为 77，字符型数据与实型数据进行运算，则会将字符型的 ASCII 代码转换为 double 型数据然后再进行运算。

以上的转换都是由编译器自动完成转换的，知道其转换的原理即可，不用自己进行转换。

第二章、控制语句

- if...else... 条件语句
- for()... 循环语句
- while()... 循环语句
- do...while() 循环语句
- continue 结束本次循环

- break 终止执行switch或循环语句
- switch 多分支语句
- return 从函数返回语句
- goto 转向语句，在结构化程序中基本不用goto语句

1. 字符输入输出函数

putchar 输出一个字符

getchar 输入一个字符

2. 选择结构嵌套

```
1  if(){
2      if() 语句;
3      else() 语句;
4  }
5  else{
6      if() 语句;
7      else() 语句;
8  }
```

3. switch多分支

```
1  switch(表达式)
2  {
3      case 常量1: 语句1; break;
4      case 常量2: 语句2; break;
5      .....
6      case 常量n: 语句n; break;
7      default : 语句 n+1; break;
8  }
```

由于用法基本一致，其余不做详细介绍。请参考：《C语言程序设计（第五版）》——谭浩强【第五章-循环结构-110页】

第三章、数组【一维数组、多维数组】

1. 一维数组

一维数组的定义方式为：

类型说明符 数组名 [常量表达式];

其中，类型说明符是任一种基本数据类型或构造数据类型。数组名是用户定义的数组标识符。方括号中的常量表达式表示数据元素的个数，也称为数组的长度。例如：

```
1  int a[10];           /* 说明整型数组a，有10个元素 */
2  float b[10], c[20]; /* 说明实型数组b，有10个元素，实型数组c，有20个元素 */
3  char ch[20];        /* 说明字符数组ch，有20个元素 */
```

对于数组类型说明应注意以下几点：

- 1) 数组的类型实际上是指数组元素的取值类型。对于同一个数组，其所有元素的数据类型都是相同的。
- 2) 数组名的书写规则应符合标识符的书写规定。
- 3) 数组名不能与其它变量名相同。例如：

```
1 | int a;  
2 | float a[10];
```

是错误的。

4) 方括号中常量表达式表示数组元素的个数，如a[5]表示数组a有5个元素。但是其下标从0开始计算。因此5个元素分别为a[0], a[1], a[2], a[3], a[4]。

5) 不能在方括号中用变量来表示元素的个数，但是可以是符号常数或常量表达式。例如：

```
1 | #define FD 5      // 宏定义，FD为常量（值不可改变）  
2 | int a[3+2], b[7+FD];
```

是合法的。但是下述说明方式是错误的。

```
1 | int n=5;  
2 | int a[n];
```

6) 允许在同一个类型说明中，说明多个数组和多个变量。例如：

```
1 | int a, b, c, d, k1[10], k2[20];
```

一维数组元素的引用

数组元素是组成数组的基本单元。数组元素也是一种变量，其标识方法为数组名后跟一个下标。下标表示了元素在数组中的序号。数组元素的一般形式为：

数组名[下标]

其中下标只能为整型常量或整型表达式。如为小数时，C编译将自动取整。例如：

a[5]
a[i+j]
a[i++]

都是合法的数组元素。

数组元素通常也称为下标变量。必须先定义数组，才能使用下标变量。在C语言中只能逐个地使用下标变量，而不能一次引用整个数组。

一维数组的初始化

一维数组的初始化可以使用以下方法实现：

1) 定义数组时给所有元素赋初值，这叫“完全初始化”。例如：

```
1 | int a[5] = {1, 2, 3, 4, 5};
```

通过将数组元素的初值依次放在一对花括号中，如此初始化之后，a[0]=1; a[1]=2; a[2]=3; a[3]=4; a[4]=5，即从左到右依次赋给每个元素。需要注意的是，初始化时各元素间是用逗号隔开的，不是用分号。

2) 可以只给一部分元素赋值，这叫“不完全初始化”。例如：

```
1 | int a[5] = {1, 2};
```

定义的数组a有5个元素，但花括号内只提供两个初值，这表示只给前面两个元素a[0]、a[1]初始化，而后面三个元素都没有被初始化。不完全初始化时，没有被初始化的元素自动为0。

需要注意的是，“不完全初始化”和“完全不初始化”不一样。如果“完全不初始化”，即只定义“int a[5];”而不初始化，那么各个元素的值就不是0了，所有元素都是垃圾值。

你也不能写成“int a[5]={};”。如果大括号中什么都不写，那就是极其严重的语法错误。大括号中最少要写一个数。比如“int a[5]={0};”，这时就是给数组“清零”，此时数组中每个元素都是零。此外，如果定义的数组的长度比花括号中所提供的初值的个数少，也是语法错误，如“a[2]={1, 2, 3, 4, 5};”。

3) 如果定义数组时就给数组中所有元素赋初值，那么就可以不指定数组的长度，因为此时元素的个数已经确定了。编程时我们经常都会使用这种写法，因为方便，既不会出问题，也不用自己计算有几个元素，系统会自动分配空间。例如：

```
1 | int a[5] = {1, 2, 3, 4, 5};
```

可以写成：

```
1 | int a[] = {1, 2, 3, 4, 5};
```

第二种写法的花括号中有5个数，所以系统会自动定义数组a的长度为5。但是要注意，只有在定义数组时就初始化才可以这样写。如果定义数组时不初始化，那么省略数组长度就是语法错误。比如：

```
1 | int a[];
```

那么编译时就会提示错误，编译器会提示你没有指定数组的长度。

2. 二维数组

二维数组定义的一般形式是：

类型说明符 数组名[常量表达式1][常量表达式2]

其中常量表达式1表示第一维下标的长度，常量表达式2表示第二维下标的长度。例如：

```
1 | int a[3][4];
```

说明了一个三行四列的数组，数组名为a，其下标变量的类型为整型。

该数组的下标变量共有3×4个，即：

```
1 | a[0][0], a[0][1], a[0][2], a[0][3]
2 | a[1][0], a[1][1], a[1][2], a[1][3]
3 | a[2][0], a[2][1], a[2][2], a[2][3]
```

二维数组在概念上是二维的，即是说其下标在两个方向上变化，下标变量在数组中的位置也处于一个平面之中，而不是象一维数组只是一个向量。但是，实际的硬件存储器却是连续编址的，也就是说存储器单元是按一维线性排列的。如何在一维存储器中存放二维数组，可有两种方式：一种是按行排列，即放完一行之后顺次放入第二行。另一种是按列排列，即放完一列之后再顺次放入第二列。

在C语言中，二维数组是按行排列的。即，先存放a[0]行，再存放a[1]行，最后存放a[2]行。每行中有四个元素也是依次存放。由于数组a说明为int类型，该类型占两个字节的内存空间，所以每个元素均占有两个字节。

二维数组元素的引用

二维数组的元素也称为双下标变量，其表示的形式为：

数组名[下标][下标]

其中下标应为整型常量或整型表达式。例如：

`a[3][4]`

表示a数组三行四列的元素。

下标变量和数组说明在形式中有些相似，但这两者具有完全不同的含义。数组说明的方括号中给出的是某一维的长度，即可取下标的最大值；而数组元素中的下标是该元素在数组中的位置标识。前者只能是常量，后者可以是常量，变量或表达式。

二维数组元素的初始化

二维数组初始化也是在类型说明时给各下标变量赋以初值。二维数组可按行分段赋值，也可按行连续赋值。

对于二维数组初始化赋值还有以下说明：

1) 可以只对部分元素赋初值，未赋初值的元素自动取0值。例如：

```
1 | int a[3][3]={1},{2},{3};
```

是对每一行的第一列元素赋值，未赋值的元素取0值。赋值后各元素的值为：

```
1 0 0
2 0 0
3 0 0
```

```
1 | int a [3][3]={0,1},{0,0,2},{3};
```

赋值后的元素值为：

```
0 1 0
0 0 2
3 0 0
```

2) 如对全部元素赋初值，则第一维的长度可以不给出。例如：

```
1 | int a[3][3]={1,2,3,4,5,6,7,8,9};
```

可以写为：

```
1 | int a[][3]={1,2,3,4,5,6,7,8,9};
```

3) 数组是一种构造类型的数据。二维数组可以看作是由一维数组的嵌套而构成的。设一维数组的每个元素都又是一个数组，就组成了二维数组。当然，前提是各元素类型必须相同。根据这样的分析，一个二维数组也可以分解为多个一维数组。C语言允许这种分解。

如二维数组 `a[3][4]`，可分解为三个一维数组，其数组名分别为：

```
a[0]
a[1]
a[2]
```

对这三个一维数组不需另作说明即可使用。这三个一维数组都有4个元素，例如：一维数组 `a[0]` 的元素为 `a[0][0]`, `a[0][1]`, `a[0][2]`, `a[0][3]`。必须强调的是，`a[0]`, `a[1]`, `a[2]` 不能当作下标变量使用，它们是数组名，不是一个单纯的下标变量。

3. 字符数组

1、字符数组的定义

```
char word[] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
```

以0（整数0）结尾的一串字符

0或 `'\0'` 是一样的，但是和 `'0'` 不同

0标志字符串的结束，但它不是字符串的一部分

计算字符串长度的时候不包含这个0

字符串以数组的形式存在，以数组或指针的形式访问

更多的是以指针的形式

2、字符数组与字符串

- `"Hello"`
- `"Hello"` 会被编译器变成一个字符数组放在某处，这个数组的长度是6，结尾还有表示结束的0
- 两个相邻的字符串常量会被自动连接起来
- 行末的\表示下一行还是这个字符串常量

-
- C语言的字符串是以字符数组的形态存在的
 - 不能用运算符对字符串做运算
 - 通过数组的方式可以遍历字符串
 - 唯一特殊的地方是字符串字面量可以用来初始化字符数组
 - 以及标准库提供了一系列字符串函数

```
char *s = "Hello, world!";
```

- s 是一个指针，初始化为指向一个字符串常量
- 由于这个常量所在的地方，所以实际上s是 const char * s，但是由于历史的原因，编译器接受不带 const的写法
- 但是试图对s所指的字符串做写入会导致严重的后果
- 如果需要修改字符串，应该用数组：`char s[] = "Hello, world!";`

指针还是数组？

- `char *str = "Hello";`
- `char word[] = "Hello";`
- 数组：这个字符串在这里
- 作为本地变量空间自动被回收
- 指针：这个字符串不知道在哪里
- 处理参数
- 动态分配空间
 - 如果要构造一个字符串—>数组
 - 如果要处理一个字符串—>指针

3、字符串的表示形式

在C语言中，可以用两种方法表示和存放字符串：

(1) 用字符数组存放一个字符串

```
char str[]="I love China";
```

(2) 用字符指针指向一个字符串

```
char* str="I love China";
```

对于第二种表示方法，有人认为str是一个字符串变量，以为定义时把字符串常量"I love China"直接赋给该字符串变量，这是不对的。

C语言对字符串常量是按字符数组处理的，在内存中开辟了一个字符数组用来存放字符串常量，程序在定义字符串指针变量str时只是把字符串首地址（即存放字符串的字符数组的首地址）赋给str。

两种表示方式的字符串输出都用

```
printf("%s\n",str);
```

%s表示输出一个字符串，给出字符指针变量名str（对于第一种表示方法，字符数组名即是字符数组的首地址，与第二种中的指针意义是一致的），则系统先输出它所指向的一个字符数据，然后自动使str自动加1，使之指向下一个字符...，如此，直到遇到字符串结束标识符 "`\0`"。

• 字符串可以表达为char的形式 • char不一定是字符串 • 本意是指向字符的指针，可能指向的是字符的数组（就像int*一样） • 只有它所指的字符数组有结尾的0，才能说它所指的是字符串

4、对使用字符指针变量和字符数组两种方法表示字符串的讨论

虽然用字符数组和字符指针变量都能实现字符串的存储和运算，但它们二者之间是有区别的，不应混为一谈。

4.1、字符数组由若干个元素组成，每个元素放一个字符；而字符指针变量中存放的是地址（字符串/字符数组的首地址），绝不是将字符串放到字符指针变量中（是字符串首地址）

4.2、**赋值方式：** 对字符数组只能对各个元素赋值，不能用以下方法对字符数组赋值

```
1 char str[14];
2 str="I love China";
3 //（但在字符数组**初始化**时可以，即char str[14]="I love China";）
```

而对字符指针变量，采用下面方法赋值：

```
1 char* a;
2 a="I love China";
```

或者是 `char* a="I love China";` 都可以

4.3、对字符指针变量赋初值（**初始化**）：

```
char* a="I love China";
```

等价于：

```
1 char* a;
2 a="I love China";
```

而对于字符数组的初始化

```
char str[14]="I love China";
```

不能等价于：

```
1 char str[14];
2 str="I love China"; （这种不是初始化，而是赋值，而对数组这样赋值是不对的）
```

4.4、如果定义了一个字符数组，那么它有确定的内存地址；而定义一个字符指针变量时，它并未指向个确定的字符数据，并且可以多次赋值。

5、字符串处理函数

注意：在使用字符串处理函数函数时应当在程序文件的开头用 `#include <string.h>`。

- puts函数
 - 输出字符串：`puts(字符数组)`
- gets函数

- 输入字符串: `gets(字符数组)`
- `strcat`函数
 - 字符串连接函数: `strcat(字符串数组1, 字符串数组2)`
- `strcpy`函数
 - 字符串复制函数: `strcpy(字符串数组1, 字符串数组2)`
- `strncpy`函数
 - 字符串复制函数: `strncpy(字符串数组1, 字符串数组2, n)`, `n` 为常数
 - 把字符串数组2中前面 `n` 个字符复制到字符串数组1中
- `strcmp`函数
 - 字符串比较函数: `strcmp(字符串数组1, 字符串数组2)`
 - 比较规则: 将两个字符串自左向右逐个字符相比 (按ASCII码值大小比较), 直到出现不同的字符或遇到 `'\0'` 为止。

- 1.如果字符串数组1与字符串数组2相同, 则返回函数值为0。
 - 2.如果字符串数组1>字符串数组2相同, 则返回函数值为一个正整数。
 - 3.如果字符串数组1<字符串数组2相同, 则返回函数值为一个负整数。
- `strlen`函数
 - 测字符串长度: `strlen(字符串数组)`
 - 函数的值为字符串中的实际长度 (不包括 `'\0'` 在内)
- `strlwr`函数
 - 转换为小写: `strlwr(字符串数组)`
- `strupr`函数
 - 转换为小写: `strupr(字符串数组)`

第四章、函数实现模块化设计

1. 定义函数的方法

函数体包括**声明部分**和**语句部分**

- 定义无参函数

一般形式为:

```

1  类型名 函数名(){
2      函数体
3  }
4  或
5  类型名 函数名(void){
6      函数体
7  }
```

- 定义有参函数

一般形式为:

```

1  类型名 函数名(形式参数列表){
2      函数体
3  }
```


- 定义空参函数
 - 函数体是空的。调用此函数时，什么工作也不做，没有任何实际作用。

一般形式为：

```
1 类型名 函数名()  
2  {}
```

2. 调用函数

调用函数的形式

一般的调用形式为：

```
1 函数名(实参表列);
```

函数调用语句：把函数调用单独作为一个语句。

函数表达式：函数出现在另一个表达式中。

函数参数：函数调用作为另外一个函数调用时的参数。

函数作为参数时的数据传递

【函数形式参数和实际参数】

函数的参数分为两种，分别是形式参数与实际参数。

①形式参数：

在定义函数时函数名后面括号中的变量名称称为形式参数（简称形参），即形参出现在函数定义中。形参变量只有在被调用时才会为其分配内存单元，在调用结束时，即刻释放所分配的内存单元。因此，形参只在函数内部有效，只有当函数被调用时，系统才为形参分配存储单元，并完成实参与形参的数据传递。在函数未被调用时，函数的形参并不占用实际的存储单元，也没有实际值。

②实际参数：

主调函数中调用一个函数时，函数名后面括号中的参数称为实际参数（简称实参），即实参出现在主调函数中。

实参可以是常量，变量，表达式，函数等，无论实参是何种类型的量，在进行函数调用时，它们都必须具有确定的值，以便把这些值传递给形参。因此应预先用赋值，输入等办法使实参获得确定值。

说明：在被定义的函数中，必须指定形参的类型。实参与形参的类型应相同或赋值兼容。实参和形参在数量上，类型上，顺序上应该严格一致，否则会发生类型不匹配的错误。

3. 函数的返回值

1. 函数的返回值是通过函数中的return语句获得的。

【return语句将被调用函数中的一个确定值带回到主函数中去。】

2. 函数值的类型。既然函数有返回值，这个值当然应属于某一个确定的类型，应当在定义函数时指定函数值的类型。

【注意：在定义函数时要指定函数的类型。】

3. 在定义函数时指定的函数类型一般应该和return语句中的表达式类型一致。

【如果函数的类型和return语句中表达式的值不一致，则以函数类型为准。对数值型数据，可以自动进行类型转换。即函数类型决定返回值的类型。】

4. 对于不带回值的函数，应当用定义函数为void类型（或称“空类型”）

4. 对被调用函数的声明和函数原型

在一个函数中调用另一个函数（即被调用函数）需要具备如下条件：

- 首先被调用的函数必须是已经定义的函数（函数库或用户自定义的函数）。
- 如果使用函数，应该在本文件头用 `#include` 指令将调用有关库函数时所需用的到的信息“包含”到文件中来。
- 如果使用用户自定义的函数，而该函数的位置在调用它的函数（即主函数）的后面（在同一个文件中），应该在主函数中对被调用的函数作**声明（delcaration）**。声明的作用是把函数名、函数参数的个数和参数类型等信息通知编译系统，以便在遇到函数调用时，编译系统能正确识别到函数并检查调用是否合法。

函数声明的一般形式有两种：

方式一：

```
1 | 函数类型 函数名(参数类型1 参数名1, 参数类型2 参数名2, ..., 参数类型n 参数名n);
```

方式二：

```
1 | 函数类型 函数名(参数类型1, 参数类型2, ..., 参数类型n);
```

注意：

函数的“定义”和“声明”不是同一回事。

- 函数的定义是指对函数功能的确立，包括指定函数名、函数值类型、形参及其类型以及函数体等，它是一个完整的、独立的函数单位。
- 函数声明的作用则是把函数的名字、函数类型以及形参的类型、个数和顺序通知编译系统，以便在调用该函数时系统按此进行对照检查，它不包含函数体。

5. 数组作为函数参数

数组元素作函数实参

数组元素可以用作函数实参，但是不能用作形参。因为形参是在函数被调用时临时分配的存储单元，不可能为一个数组元素单独分配存储单元（数组是一个整体，在内存中占连续的一段存储单元）。在用数组元素作函数参数实参时，把实参的值传给形参，是“值传递”方式。数据传递方向是从实参传到形参，单向传递。

一维数组名作函数参数

除了可以用数组元素作为函数参数外，还可以用数组名作函数参数（包括实参和形参）。

注意：用数组元素作实参时，向形参变量传递的是数组元素的值，而用数组名作函数函数参数时，向形参（数组名或指针变量）传递的是地址值。

多维数组名作函数参数

由于用法基本一致，其余不做详细介绍。

请参考：《C语言程序设计（第五版）》——谭浩强【第七章-用函数实现模块化程序设计 -167页】

6. 局部变量和全局变量

变量按存储区域分：全局变量、静态全局变量和静态局部变量都存放在内存的静态存储区域，局部变量存放在内存的栈区。

变量按作用域分：

- 全局变量：在整个工程文件内都有效；“在函数外定义的变量”，即从定义变量的位置到本源文件结束都有效。由于同一文件中的所有函数都能引用全局变量的值，因此如果在一个函数中改变了全局变量的值，就能影响到其他函数中全局变量的值。
- 静态全局变量：只在定义它的文件内有效，效果和全局变量一样，不过就在本文件内部；
- 静态局部变量：只在定义它的函数内有效，只是程序仅分配一次内存，函数返回后，该变量不会消失；静态局部变量的生存期虽然为整个工程，但是其作用仍与局部变量相同，即只能在定义该变量的函数内使用该变量。退出该函数后，尽管该变量还继续存在，但不能使用它。
- 局部变量：在定义它的函数内有效，但是函数返回后失效。“在函数内定义的变量”，即在一个函数内部定义的变量，只在本函数范围内有效。

注意：全局变量和静态变量如果没有手工初始化，则由编译器初始化为0。局部变量的值不可知

静态局部变量与全局变量最明显的区别就在于：全局变量在其定义后所有函数都能用，但是静态局部变量只能在一个函数里面用。

形参变量：只在被调用期间才分配内存单元，调用结束立即释放。

7. 变量的存储方式和生存期

变量的存储方式有两种：

- 静态存储方式：是指程序在运行期间由系统分配固定的存储空间的方式。
- 动态存储方式：是指在程序运行期间根据需要进行动态的分配存储空间的方式。

供用户使用的存储空间可分为3个部分：程序区，静态存储区，动态存储区。

全局变量全部存放在静态存储区中，在程序开始执行时给全局变量分配存储区，程序执行完毕就释放。

动态存储区中存放以下数据：

1. 函数形式参数。在调用函数时给形参分配存储空间。
2. 函数中定义的没有用static关键字声明的变量，即自动变量。
3. 函数调用时的现场保护和返回地址等。

每一个变量和函数都有两个属性：数据类型和数据的存储类别。【存储类别指的是数据在内存中存储的方式：静态存储和动态存储】

在定义和声明变量和函数时，一般应该同时指定其数据类型和存储类别，也可以采用默认方式指定（即如果用户不指定，系统会隐含地指定为某一种存储类别）。

存储类别包括4种：自动的（auto）、静态的（static）、寄存器的（register）、外部的（extern）。

- 自动变量（auto变量）
 - 函数中的局部变量，如果不专门声明为static（静态）存储类别，都是动态地分配存储空间的，数据存储在动态存储区中。函数的形参和在函数定义的局部变量（包括在复合语句中定义的局部变量），都属于此类。在调用该函数时，系统会给这些变量分配存储空间，在函数调用调用结束时就自动释放这些存储空间。因此这类局部变量称为自动变量。
 - 关键字auto可以省略不写，不写auto则隐含的指定为“自动存储类别”，它属于动态存储的方式。程序中大多数变量都属于自动变量。
- 静态局部变量（static局部变量）
 - 静态局部变量属于静态存储类别，在静态存储区域内分配存储单元。在整个程序运行期间都不释放。而自动变量（即动态局部变量）属于动态存储类别，分配在动态存储区空间而不再静态

存储区空间，函数调用结束后即释放。

- 对静态局部变量是在编译时赋初值的，即只赋一次初值，在程序运行时它已有初值。以后每次调用函数时不再重新赋初值而只是保留上次函数调用结束时的值。而对自动变量赋初值，不是在编译时进行的，而是在函数调用时进行的，每调用一次函数重新给一次初值，相当于执行一次赋值语句。
- 如果在定义局部变量时不赋值的话，则对静态局部变量来说，编译时自动赋初值0（对数值变量）或空字符 '\0'（对字符变量）。而对自动变量来说，它的值是一个不确定的值。这是由于每次函数调用结束后存储单元已释放，下次调用时又重新另分配存储单元，而所分配的单元中的内容是不可加的。
- 虽然静态局部变量在函数调用结束后仍然存在，但其他函数是不能引用他的。因为它是局部变量，只能被本函数引用，而不能被其他函数引用。
- 寄存器变量（register变量）
 - 寄存器变量的定义形式是：
`register 类型标识符 变量名`
 - 寄存器是与机器硬件密切相关的，不同类型的计算机，寄存器的数目是不一样的，通常为2到3个，对于在一个函数中说明的多于2到3个的寄存器变量，C编译程序会自动地将寄存器变量变为自动变量。
 - 由于受硬件寄存器长度的限制，所以寄存器变量只能是char、int或指针型。寄存器说明符只能用于说明函数中的变量和函数中的形参，因此不允许将外部变量或静态变量说明为"register"。
 - register型变量常用于作为循环控制变量，这是使用它的高速特点的最佳场合。比较下面两个程序的运算速度。
- 注意三种局部变量的存储位置是不同的
 - 自动变量存储在动态存储区
 - 静态局部变量存储在静态存储区
 - 寄存器存储在CPU中的寄存器中

全局变量的存储类别

- 在一个文件内扩展外部变量的作用域
 - 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结束。在定义点之前的函数不能引用该外部变量。如果由于某种考虑，在定义点之前的函数需要引用该外部变量,则应该在引用之前用关键字 `extern` 对该变量作“**外部变量声明**”，表示把该外部变量的作用域扩展到此位置。有了此声明，就可以从“声明”处起,合法地使用该外部变量。
 - 注意：提倡将外部变量的定义放在引用它的所有函数之前，这样可以避免在函数中多加一个 `extern` 声明。
 - 用 `extern` 声明外部变量时，类型名也可以省写。例如：`extern int A,B,C; ——> extern A,B,C`。
- 将外部变量的作用域扩展到其他文件
 - 第一种情况是在同一个源文件中使用外部变量的方法，如果有多个源文件，想在A文件中引用B文件中的已定义外部变量，该如何做？
 - 假设一个程序包含两个文件，两个文件都需要用到同一个外部变量Num,若在两个文件中各自定义一个外部变量Num，将会在进行程序的连接时出现“重复定义”的错误。
 - 因此，正确的做法是：在任一个文件中定义外部变量Num，然后在另一个文件中用关键字extern进行“外部变量声明”，即“extern Num”。
 - 在编译和链接时，系统就会知道Num有外部链接，可以从别处找到已定义的外部变量Num，并将另一个文件中定义的外部变量Num的作用域扩展到本文件，那么就可以在本文件中合法的使用变量Num了。

- 例子：分别编写两个源文件文件file1和file2,在file1中定义外部变量A,在file2中用extern来声明外部变量，把A的作用域扩展到file2中

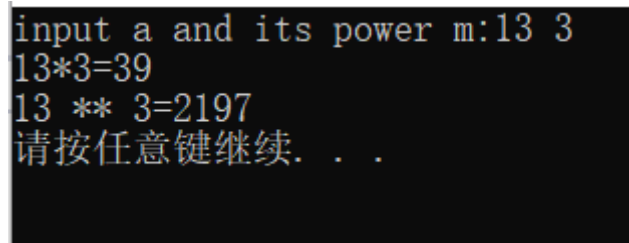
file1:

```
1 //file1
2 #include<stdio.h>
3 //给定b的值，输入a和m，求a*b和a**m(a的m次方)的值
4
5 int A; //定义外部变量
6 int power(int);
7 int main()
8 {
9     int b = 3, c, d, m;
10    printf("input a and its power m:");
11    scanf_s("%d %d", &A, &m);
12    c = A * b;
13    printf("%d*d=%d\n", A, b, c);
14    d = power(m);
15    printf("%d ** %d=%d\n", A, m, d);
16    system("pause");
17 }
```

file2:

```
1 //file2
2 extern A;
3 //把在file1文件中已定义的外部变量的作用域扩展到本文件
4 int power(int n)
5 {
6     int i, y = 1;
7     for ( i = 1; i <= n; i++)
8     {
9         y *= A;
10    }
11
12    return y;
13 }
```

运行结果:



```
input a and its power m:13 3
13*3=39
13 ** 3=2197
请按任意键继续. . .
```

- 解析：
 - 假设某一程序有5个源文件，那么只需要在其中一个源文件中定义外部变量A,然后在其余四个文件中使用关键字extern声明外部变量即可。各文件经过编译后会连接成一个可执行的目标文件。
 - 用这种方法扩展全局变量的作用域应十分慎重，因为在执行一个文件中的操作时可能会改变该全局变量的值，这样就会影响到另一个文件中全局变量的值，从而影响该文件中函数的执行结果。

- 将外部变量的作用域限制在本文件中
 - 若希望外部变量仅限于被本文件使用，而不被其它文件使用，那么可以在定义外部变量时加上一个static，例如：

```
1 static int A;  
2 int main()  
3 {  
4     .....  
5 }
```

这样在其它文件中就算使用“extern A”，也不能使用本文件的外部变量A。
这种加上static声明，只能用于本文件的外部变量成为“静态外部变量”。
用static声明一个变量的作用：

- (1) 对局部变量用static声明，把它分配在静态存储区，该变量在整个程序执行期间所在的存储单元都不会释放。
- (2) 对全局变量用static声明，则该变量的作用域只限于本文件模块（即被声明的文件中）

8. 存储类别小结

对数据的定义，需要指定两种属性：**数据类型**和**存储类别**，分别使用两个关键字。

例如：

```
1 static int a;           //静态局部整型变量或静态外部整型变量  
2 auto char c;           //自动变量，在函数内定义使用  
3 register int d;        //寄存器变量，在函数内定义
```

此外，可以用 `extern` 声明已定义的外部变量，例如：

```
1 extern b;              //将已定义的外部变量b的作用域扩展至此
```

下面从不同角度做些归纳

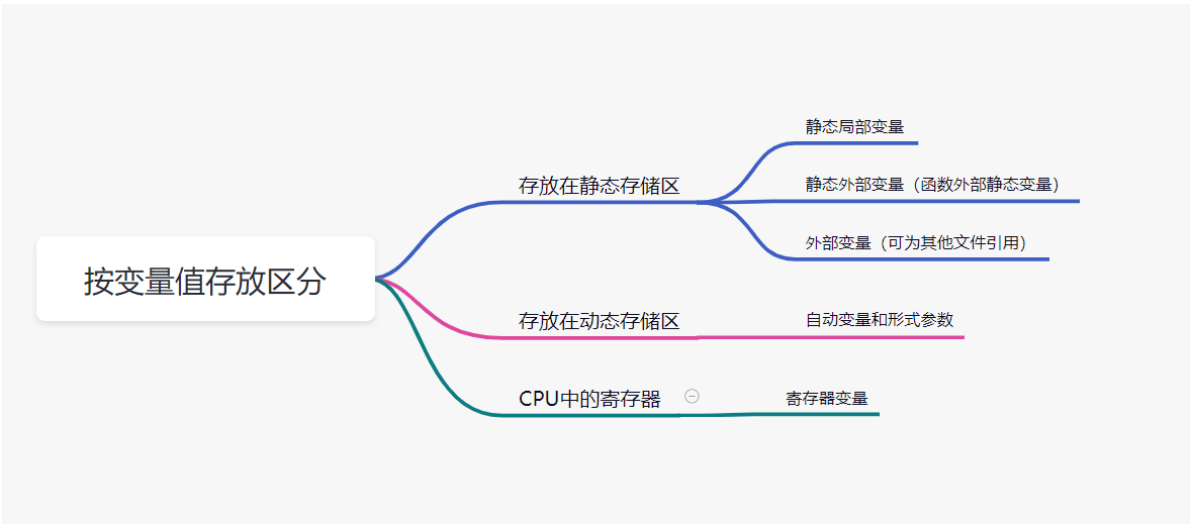
1. 从作用域角度分，有局部变量和全局变量。它们采用的存储类型如下：



2. 从变量存在的时间（生存期）来区分，有动态存储和静态存储两种类型。静态存储类型是整个程序运行时间都存在，而动态存储原则是在调用函数时临时分配分配单元。



3. 从变量值存放的位置来区分，可分为：



4. 关9.作用域和生存期的概念。

- 如果一个变量在某个文件或函数范围内是有效的，就称为该范围为该变量的**作用域**。在此作用域内可以引用该变量，在专业书中称变量在此作用域内“可见”，这种性质称为变量的可见性。
- 如果一个变量值在某一时刻是存在的，则认为这一时刻属于该变量的生存期，或称该变量在此时刻“存在”。

各种类型变量的作用域和存在性情况

变量存储类别	函数内		函数外	
	作用域	存在性	作用域	存在性
自动变量/局部变量	√	√	×	×
静态局部变量	√	√	×	√
静态外部变量	√	√	√ (只限本文件)	√
外部变量	√	√	√	√

5. `static` 对局部变量和全局变量的作用域不同。对于局部变量来说它使变量的由动态存储方式改变为静态存储方式。而对于全局变量来说，它使变量局部化（局部于本文件），但静态存储方式。从作

用域角度看，但凡有 `static` 声明的，其作用域都是局限的或者局限于本函数内（静态局部变量），或者局限于本文件内（静态外部变量）。

9. 内部函数和外部函数

根据函数能否被其他源文件调用，将函数区分为**内部函数**和**外部函数**。

内部函数

如果一个函数只能被本文件中其他函数所调用，它将称为**内部函数**。

在定义内部函数时，在函数名和函数类型前面加 `static`，即：

```
1 | static 类型名 函数名(形参表);
```

内部函数又称**静态函数**，因为它是 `static` 声明的。

外部函数

如果在定义函数时，在函数首部的最左端加关键字 `extern`，则此函数是外部函数，可供其他文件调用。

一般形式为：

```
1 | extern 类型名 函数名(形参表);
```

C语言规定，如果在定义函数时省略 `extern`，则默认为外部函数。

在需要调用此函数的其他文件中，需要对此函数作声明（不要忘记，即使在本文件中调用一个函数，也需要用函数原型进行声明）。在对此函数作声明时，要加关键字 `extern`，表示该函数“是在其他文件中定义的外部函数”。

第五章、指针【C语言的灵魂】

1. 指针及指针变量【概念、定义】

定义指针变量：`类型名称 *指针变量名;`

在定义指针变量时要注意：

1. 指针变量前面的“`*`”表示该变量为指针型变量。
2. 在定义指针变量时必须**指定基类型**。指针的基类型用来定义此指针变量可以指向的变量的类型。一个变量的指针的含义包括两个方面，一是存储单元编号表示的纯地址，一是它指向的存储单元的数据类型（如 `int`、`char`、`float` 等）。
3. 指向整型数据的指针类型表示为“`int*`”，读作“**指向 `int` 的指针**”或简称“**`int` 指针**”。
【`int*`、`float*`、`char*`，是三种不同的类型，不能混淆】
4. 指针变量中只能存放地址（指针），不要将一个整数赋给指针变量。

◦ 如：`*pointer_1=100;` // `pointer_1` 是指针变量，100 是整数，不合法

如果需要取出某个变量的地址，可以使用取址运算符 `&`：

例如：

```
1 | char *pa = &a;  
2 | int *pb = &b;
```


如果需要访问指针变量指向的数据类型，可以使用取值运算符 `*`：

例如：

```
1 printf("%c,%d\n", *pa, *pb);
```

访问地址里的值的两种方式：

直接访问：即按变量名进行的访问。

间接访问：即通过指针变量进行的访问。

注意：避免访问未初始化的指针。【因为未初始化的指针指向的地址是随机的，未初始化就使用是非常危险的!!!】

例如：【以下示例为**错误的**】

```
1 #include <stdio.h>
2 main(){
3     int *a;
4     *a = 123;
5 }
```

指针与指针变量

如果有一个变量专门来存放另一变量的地址（即指针），则称它为“指针变量”。

指针变量就是地址变量，用来存放地址，指针变量的值就是地址（即指针）。

注意：区分“指针”和“指针变量”这两个概念。指针就是一个地址，而指针变量是存放地址值的变量。

2. 引用指针变量

在引用指针变量时，可能有3种情况：

- 给指针变量赋值。
 - 如：`p = &a;` //把 `a` 的地址赋给指针变量 `p`。
 - 指针变量`p`的值是变量`a`的地址，`p`指向`a`。
- 引用指针变量指向的变量。
 - 如果已经执行 `p=&a;`，即指针变量`p`指向了整型变量`a`，则 `printf("%d", *p);`
 - 其作用是以整数形式输出变量`p`指向的变量的值，即变量`a`的值。
- 引用指针变量的值。
 - 如：`printf("%o", p);`
 - 其作用是以八进制整数输出指针变量的值，如果`p`指向变量`a`，就是输出了`a`的地址，即`&a`。

注意：要熟练掌握两个有关运算符。

- `&` 取地址运算符。`&a`是变量`a`的地址。
- `*` 指针运算符（或称“间接访问”运算符），`*p`代表指针变量`p`指向的对象。

指针变量作为函数参数

函数的参数不仅可以是整数型、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。

注意：不能企图通过改变指针形参的值而使指针实参的值改变。

3. 通过指针引用数组

指针变量既然可以指向变量，当然也可以指向数组元素（把某一元素的地址放到一个指针变量中）。所谓数组元素的指针就是数组元素的地址。

将数组元素地址赋值给指针变量，如：

```
1  int a[10]={1,3,5,7,9,11,13,15,17,19}; //定义a为包含10个整型数据的数组
2  int *p;                               //定义p为指向整型变量的指针变量
3  p = &a[0];                             //把a[0]元素的地址赋给指针变量p
```

下标法赋值：

指针变量 = &数组名[数值]; 将下表为 数组名[数值] 的元素地址，赋值给 指针变量。

不加标赋值：

指针变量 = 数组名; 将数组的首元素【即 数组名[0]】地址赋值给 指针变量。

下面两个语句等价：

```
1  int *p;
2  p = &a[0];
3  -----
4  int *p;
5  p = a;
```

引用数组元时指针的运算

在指针已指向一个数组元素时，可以对指针进行以下运算：

- 加一个整数（用 +或+=），如 p+1；
- 减一个整数（用 -或-=），如 p-1；
- 自加运算，如： p++; ++p;
- 自减运算，如： p--; --p;

两指针相减，如： p1-p2（只有p1和p2都指向同一数组中的元素时才有意义）。

分别说明如下：

- 如果指针变量p已指向数组中的一个元素，则p+1指向同一数组中的下一个元素，p-1指向同一数组元素中的上一个元素。
 - 注意：执行p+1时并不是将p的值（地址）简单的加1，而是加上一个数组元素所占的字节数。
 - 例如：数组元素是float型，每个元素是float型，每个元素占4个字节，则p+1意味着使p的值（地址）加4个字节，以使它指向下一元素。
- 如果p的初值为 &a[0],则表示 p+i 和 a+i 就是数组元素 a[i] 的地址。
- *(p+i) 或 *(a+i) 是 p+i 或 a+i 所指向的数组元素，即 a[i]。
 - 说明：[] 实际上是变址运算符，即将 a[i] 按 a+i 计算然后找出此地址单元中的地址。
- 如果指针变量 p1 和 p2 都指向同一组数组中的元素，如执行 p2-p1,结果是 p2-p1 的值（两个地址之差）除以数组元素的长度。
 - 注意：两个地址不能相加，如 p1+p2 是无实际意义的。

通过指针引用数组元素

引用一个数组元素，可以用下面两种方法：

- 下标法：如 `a[i]` 形式；
- 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 `p=a`。

指向数组元素的指针变量也可以带下标，如 `p[i]`。

`++` 和 `*` 同优先级，结合方向为自左向右。

`*(p++)` 与 `*(++p)`，作用不相同。

- `*(p++)`：是先取 `*p` 的值，然后使 `p+1`。
- `*(++p)`：是先 `p+1`，然后再取 `*p` 的值。

`++(*p)`：表示 `p` 所指向的元素值加1。

`--(*p)`：表示 `p` 所指向的元素值减1。

所以：

- `*(++p)` 相当与 `a[++i]`，先使 `p` 自加，再进行 `*` 运算。
- `*(--p)` 相当与 `a[--i]`，先使 `p` 自减，再进行 `*` 运算。

用数组名做函数参数

数组名做函数参数方法定义一般形式为：

```
1 返回值类型 方法名(参数类型 数组名[],参数列表.....){
2      方法体;
3      返回值;
4  }
```

指针做函数参数定义一般形式为：

```
1 返回值类型 方法名(参数类型 *数组名,参数列表.....){ //这里的“*数组名”表示数组的首元素地址“数组名[0]”
2      方法体;
3      返回值;
4  }
```

两种定义方法等价。

`*数组名` 等价于 `数组名[0]`。

注意：数组名做方法参数时，传递的是数组首元素的地址，而非元素值。

常用这种方法通过调用一个函数来改变实参数组的值。

以表变量名和数组名作为函数参数的比较

参数类型	变量名	数组名
要求的形参类型	变量名	数组名或指针
传递参数	变量的值	实参数组首元素地址
通过函数调用能否改变实参的值	不能改变实参变量的值	能改变实参数组的值

注意：实参数组名代表一个固定的地址，或者说是指针常量，但形参数组名并不是一个固定的地址，而是按指针变量处理。

在函数调用进行虚实结合后，形参的值就是实参数组首元素的地址。在函数执行期间，它还可以再被赋值。

归纳分析：如果有一个实参数组，想要在函数中改变此数组中的元素的值，实参与形参的对应关系有以下4种情况。

- 形参和实参都用数组名。
 - 例如：

```
1  int main(){
2      int a[10];
3      ...
4      f(a,10);
5      ...
6  }
7  int f(int x[],int n){
8      ...
9  }
10 //由于形参数组名x接收了实参数组首元素a[0]的地址值，因此可以认为在函数调用期间，形参数组与
    实参数组共用一段内存单元。
```

- 实参用数组名，形参用指针变量。
 - 例如：

```
1  int main(){
2      int a[10];
3      ...
4      f(a,10);
5      ...
6  }
7  void f(int *x,int n){
8      ...
9  }
10 //实参a为数组名，形参数组x为int * 型的指针变量，调用函数开始后，形参x指向a[0]，即
    x=&a[0]。通过x的值改变，可以指向a数组的任一元素。
```

- 实参形参都用指针变量。
 - 例如：

```
1  int main(){
2      int a[10], *p = a;
3      ...
4      f(p,10);
5      ...
6  }
7  void f(int *x,int n){
8      ...
9  }
10 //实参p和形参x都是int * 型的指针变量。先使实参指针变量p指向数组a[0]，p的值是&a[0]。然后将p的值传给指针变量x，x的初始值也是&a[0]，通过x值的改变可以使x指向数组元素a的任一元素。
```

- 实参为指针变量，形参为数组名。

○ 例如：

```
1 int main(){
2     int a[10], *p = a;
3     ...
4     f(p,10);
5     ...
6 }
7 void f(int x[],int n){
8     ...
9 }
10 //实参p为指针变量，它指向a[0]。形参为数组名x，编译系统把x作为指针变量处理，今将a[0]的地址
    传给形参x，使x也指向a[0]。也可以理解为形参数组x和a数组共用同一段内存单元。在函数执行过程中
    可以使x[i]的值发生变化，而x[i]就是a[i]。
```

注意：如果使用指针变量作实参，必须先使指针变量有确定值，指向一个以定义的对象。

以上4种方法，实质上都是地址的传递。其中（3）、（4）两种只是形式上的不同，实际上形参都是使用指针变量。

通过指针引用多维数组

指针引用多维数组：除了表示取元素之外，还可以表示取哪一维

对于二维数组：

1、

```
int a[3][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
printf("%p, %p, %p", a, &a, *a);
```

a 是一个行指针。指向一个有四个元素的数组，占16个字节

&a 是一个指向二维数组的指针，二维数组有12个元素，占48个字节

*a 是一个指向int类型数据的指针。

2、

a[i][j] 等价于 *((a+i)+j)， &a[i][j] 等价于 (a+i)+j

a[i] 等价于 *(a+i)， &a[i]

```
printf("%d,%d", sizeof(* a), sizeof(* *a)); // *a =*(a+0)=a[0]
// a  &a[0]  a+1
printf("\n%p, %p, %p", a, a + 1, a + 2); //代表某一行的首地址
printf("\n%p, %p, %p", *a, *(a + 1), *(a + 2)); //代表某一行第一列的首地址
printf("\n%p, %p, %p", a[0], a[1], a[2]); //代表某一行第一列的首地址
```

3、二维数组名是指向行的，它不能对如下说明的指针变量p直接赋值：

```
1 int a[3][4]={10,11,12,13},{20,21,22,23},{30,31,32,33}},*p;
```

其原因就是p与a的对象性质不同，或者说二者不是同一级指针。C语言可以通过定义行数组指针的方法，使得一个指针变量与二维数组名具有相同的性质。

行数组指针的定义方法如下：

数据类型 (*指针变量名)[二维数组列数];

例如，对上述a数组，行数组指针定义如下：

int (p)[4]; 它表示，数组p有4个int型元素，分别为 (*p)[0]、(*p)[1]、(*p)[2]、(*p)[3]，亦即p指向的是有4个int型元素的一维数组，即p为行指针

此时，可用如下方式对指针p赋值：

```
p=a;
```

指针访问三维数组

数组与指针关系密切，数组元素除了可以使用下标来访问，还可用指针形式表示。数组元素可以很方便地用数组名常指针来表示，以3维int型数组A举例，其中的元素A[i][j][k]可用下述形式表示：

(1) `*(A[i][j]+k)`

A[i][j] 是int型指针，其值为 &A[i][j][0]，因此，A[i][j][k] 可表述为 `*(A[i][j]+k)`。

(2) `*(*(A[i]+j)+k)`

和第一种形式比较，不难发现 `A[i][j] = *(A[i]+j)`，A[i] 是二级指针，其值为 &A[i][0]。

(3) `*(*(A+i)+j)+k)`

将第2种形式的A[i]替换成了 `*(A+i)`，此处A是三级指针，其值为 &A[0]。

此处以3维数组举例，还可进一步推广到更高维的情况。

指针数组

指针也可作为数组中的元素，将一个个指针用数组形式组织起来，就构成了指针数组。

一个数组，若其元素均为指针类型数据，称为指针数组，也就是说，指针数组中的每一个元素都存放一个地址，相当于一个指针变量。

定义一维指针数组的一般形式为：

```
1 类型名 *数组名[数组长度];  
2  int *p[4];
```

用指向数组的指针作函数参数

一维数组名可以做函数参数，多维数组名可以做函数参数。用指针变量作形参，以接受实参数组名传递过来的地址。

可以有两种方法：

- 用指向变量的指针变量。
- 用指向一维数组的指针变量。

4. 通过指针引用字符串

字符串的应用方式：

- 用字符数组存放一个字符串，可以通过数组名和下标引用字符串中的一个字符，也可以通过数组名和格式声明 `%s` 输出该字符串。

```

1  #include <stdio.h>
2  main(){
3      char *string;
4      string="I love you";
5      printf("%s\n",string);
6
7      char string2[]="I love you";
8      printf("%s\n",string2);
9
10     char string3[]={ "I love you" };
11     printf("%s\n",string3);
12 }
13 //三种定义形式输出结果一样

```

- 用指针变量访问字符串。通过改变指针变量的值使它指向字符串中的不同字符。

使用字符串指针变量和字符数组的比较

1. 字符串由若干个元素组成，每个元素中放一个字符，而字符串指针变量中存放的是地址（字符串第一个字符的地址），绝不是将字符串放到字符串指针变量中。
2. 赋值方式。可以对字符串指针变量赋值，但不能对数组名赋值。
3. 初始化定义。对字符串指针变量赋初值：

```

1  char *a="I love china!";
2  //等价于
3  char *a;
4  a = "I Love china!";
5  //而对数组的初始化：
6  char str[14]="I love china!";
7  //不等价于
8  char str[14];
9  str[]="I love china!";

```

数组可以在定义时对各元素赋初值，但不能用赋值语句对字符串数组中全部元素整体赋值。

4. 存储单元的内容。编译时为字符数组分配若干存储单元，以存放各元素的值，而对字符串指针变量，只分配一个存储单元。
 - 如果定义了字符数组，但未能对它赋值，这时数组中的元素的值是不可预料的。可以引用（如输出）这些值，结果显然是无意义的，但不会造成严重的后果，容易发现和更正。
 - 如果定义了字符串指针变量，应当及时把字符串变量（或字符串元素）的地址赋给它，使它指向一个字符型数据，如果未对它赋予一个地址值，它并未具体指向一个确定的对象。此时如果向该指针变量指向的对象输入数据，可能会出现严重的后果。
5. 指针变量的值是可以改变的，而字符串数组名代表一个固定的值（数组首元素的地址），不能改变。
6. 字符串数组中各元素的值是可以取代的（可以对它们赋值），但字符串指针变量指向的字符串常量中的内容是不可以被取代的（不能对它们赋值）。
7. 引用数组元素。对字符串数组可以用下标法（用数组名和下标）引用一个数组元素，也可以用地址法引用数组元素。
8. 用指针变量指向一个格式字符串，可以用它代替printf函数中的格式字符串。

```

1  char *format;
2  format = "a=%d,b=%f\n"; //使format指向一个字符串
3  printf(format,a,b);
4  //这种printf函数称为可变格式输出函数。

```

字符指针做函数参数

实参和形参都可以选择字符数组名和字符指针变量，但存在区别：

- (1) 编译时为字符数组分配若干存储单元，以存放个元素的值，而对字符指针变量，只分配一个存储单元
- (2) 指针变量的值是可以改变的，而数组名代表一个固定的值（数组首元素的地址），不能改变

```
1 char *a="i am a student"
2 a=a+7;    //合法的
3
4 char str[]{"i am a student"};
5 str=str+7 //非法的
```

- (3) 字符数组中各元素的值是可以改变的，但字符指针变量指向的字符串常量中的内容是不能改变的

```
1 char a[]="house";
2 char *b="house";
3 a[2]='r';    //合法
4 *(b+2)='r';  //非法
```

接着，引入一个用字符数组名作为函数参数的例子，实现字符串的复制

```
1 #include<stdio.h>
2 int main(){
3     void copy_string(char from[] ,char to[]);
4     char a[]="i am a teacher";
5     char b[]="you are a student";
6
7     copy_string(a,b); //把a复制到b
8     printf("%s\n%s",a,b);
9 }
10 void copy_string(char from[], char to[]){
11     int i=0;
12     while(from[i]!='\0'){
13         to[i]=from[i]; i++;
14     }
15     to[i]='\0';
16 }
```

5. 指向函数的指针

函数名就是函数的指针，它代表函数的起始地址。

定义和使用指向函数的指针变量

定义指向函数的指针变量的一般形式为：

类型名 (*指针变量名)(函数参数列表)

这里的“类型名”是指函数的返回类型。

说明：

1. 定义指向函数的指针变量，并不意味着这个指针变量可以指向任何函数，它只能指向在定义时指定的类型的函数。
 - 在程序中把哪一个函数的地址赋给它，它就指向哪一个函数。在一个程序运行中，一个指针变量可以先后指向同类型的不同函数。

2. 如果要用指针调用函数，必须先使用指针变量指向该函数。
 - 如： `指针变量名 = 函数名`；这样就把“函数名”的入口地址赋给了指针变量“指针变量名”。
3. 在给函数指针变量赋值时，只须给出函数名而不必给出参数。
4. 用函数指针变量调用函数时，只需将（`*指针变量名`）代替函数名即可，在（`*指针变量名`）之后的括号中根据需要写上实参。
5. 对指向函数的指针变量不能进行算数运算，如 `p+n`, `p++`, `p--` 等运算是无意义的。
6. 用函数名调用函数，只能调用所指定的一个函数，而通过指针变量比较灵活，可以根据不同情况先后调用不同的函数。

用指向函数的指针作函数参数

指向函数的指针变量的一个重要用途是把函数的入口地址作为参数传递到其他函数。

指向函数的指针可以作为函数参数，把函数的入口地址传递给形参，这样就能够在被调用的函数中使用实参函数。

它的原理简述如下：

有一个函数（假设函数名为fun），它有两个形参（x1和x2），定义x1和x2为指向函数的指针变量。再调用函数fun时，实参为两个函数名f1和f2，给形参传递的是f1和f2的入口地址。这样在函数fun中就可以调用f1和f2函数了。

例如：

```
1  实参函数名      f1          f2
2  void fun(int (*x1)(int),int (*x2)(int,int))//定义fun函数，形参是指向函数的指针变量
3  {
4      int a,b,i=3,j=5;
5      a=(*x1)(i);      //调用f1函数，i是实参
6      b=(*x2)(i,j);    //调用f2函数，i、j是实参
7  }
```

在fun函数中声明形参x1和x2为指向函数的指针变量，x1指向的函数有一个整型形参，x2指向的函数有两个整型实参。函数fun的形参x1和x2（指针变量）在函数fun未被调用时并不占内存单元，也不指向任何函数。在主函数调用fun函数时，把实参函数f1和f2的入口地址传给形参指针变量x1和x2，使x1和x2指向函数f1和f2。这时，在函数fun中，用 `*x1` 和 `*x2` 就可以调用函数f1和f2。`(*x1)(i)` 就相当于 `f1(i)`，`(*x2)(i,j)` 就相当于 `f2(i,j)`。

6. 返回指针值的函数

定义返回指针的函数的原型一般形式为：

```
1  类型名 *函数名(参数列表);
```

例如：

```
1  int *a(int x,int y);
```

a是函数名，调用它以后能得到一个 `int*` 型（指向整型数据）的指针，即整型数据的地址。x和y是函数a的形参，为整型。

请注意在 `*a` 两侧没有括号，在 `a` 的两侧分别是 `*` 运算符和 `()` 运算符。而 `()` 优先级高于 `*`，因此 `a` 先与 `()` 结合，显然这是函数形式。这个函数前面有一个 `*`，表示此函数是指针型函数（函数值是指针）。最前面的 `int` 表示返回的指针指向整型变量。

7. 指针数组和多重指针

定义一维指针数组的一般形式为：

```
1 | 类型名 *数组名[数组长度];
```

类型名中应包括符号 `*`，如 `int*` 表示指向整数数据的指针类型。

例如：

```
1 | int *p[4];
```

由于 `[]` 比 `*` 优先级高，因此 `p` 先与 `[4]` 结合，形成 `p[4]` 形式，表示 `p` 数组有 4 个元素。然后再与 `p` 前面的 `*` 结合，`*` 表示此数组是指针类型的，每个数组元素（相当于一个指针变量）都指向一个整型变量。

注意一定不要写成：

```
1 | int (*p)[4]; //这是指向一维数组的指针变量
```

指向指针数据的指针变量

定义一个指向指针数据类型的指针变量：

```
1 | char **p;
```

`p` 的前面有两个 `*` 号。`*` 运算符的结合性是从右到左，因此 `**p` 相当于 `*(*p)`，显然 `*p` 是指针变量的定义形式。如果没有最前面的 `*`，那就是定义了一个指向字符数据的指针变量。现在它前面又有一个 `*` 号，即 `char**p`。可以把它分成两部分看，即：`char*` 和 `(*p)`，后面的 `(*p)` 表示 `p` 是指针变量，前面的 `char*` 表示 `p` 指向的是 `char*` 型的数据。也就是说，`p` 指向一个字符指针变量（这个字符指针变量指向一个字符型数据）。

例如：使用指向指针数据的指针变量。

```
1 | #include<stdio.h>
2 | main(){
3 |     char *name={"Follow me","BASIC","Great wall","FORTRAN","Computer
   | design"};
4 |     char **p;
5 |     int i;
6 |     for(i=0;i<5;i++){
7 |         p=name+i;
8 |         printf("%s\n",**p);
9 |     }
10 | }
```

指针数组作main函数的形参

main函数的第1行一般写成以下形式：

```
int main() 或 int main(void)
```

括号中是空的或有“void”，表示main函数没有参数，调用main函数时不必给出实参。

这是一般程序常采用的形式。实际上，在某些情况下，main函数可以有参数，即：

```
1 | int main(int argc, char * argv[])
```

其中，`argc` 和 `argv` 就是main函数的形参，它们是程序的“命令行参数”。

`argc` (`argument count` 的缩写,意思是参数个数)，`argv` (`argument vector` 缩写,意思是参数向量`)，它是一个*char指针数组，数组中每一个元素（其值为指针）指向命令行中的一个字符串的首字符。

注意：如果用带参数的main函数，其第一个形参必须是int 型,用来接收形参个数，第二个形参必须是字符指针数组，用来接收从操作系统命令行传来的字符串中首字符的地址。通常main函数和其他函数组成一个文件模块，有一个文件名。对这个文件进行编译和连接，得到可执行文件(后缀为.exe)。用户执行这个可执行文件，操作系统就调用main函数，然后由main函数调用其他函数，从而完成程序的功能。

什么情况下main函数需要参数?main函数的形参是从哪里传递给它们的呢?

显然形参的值不可能在程序中得到。main函数是操作系统调用的，实参只能由操作系统给出。在操作命令状态下，实参是和执行文件的命令一起给出的。例如在DOS,UNIX或Linux等系统的操作命令状态下，在命令行中包括了命令名和需要传给main函数的参数。

命令行的一般形式为：

```
1 | 命令名 参数1 参数2 ... 参数n
```

命令名和各参数之间用空格分隔。

8. 动态内存分配与指向它的指针变量

对内存动态分配是通过系统提供的函数库来实现的，主要有 `malloc`，`calloc`，`free`，`realloc` 这4个函数。

- 用**malloc**函数开辟动态存储区：

其函数原型为：

```
1 | void * malloc(unsigned int size);
```

其作用是在内存的动态存储区中分配一个长度为size 的连续空间。形参size的类型定为无符号整型(不允许为负数)。

此函数的值(即“返回值”)是所分配区域的第一个字节的地址，或者说,此函数是一个指针型函数，返回的指针指向该分配域的第一个字节。如：

```
1 | malloc( 100);           //开辟100字节的临时分配域.函数值为其第1个字节的地址
```

注意指针的基类型为void，即不指向任何类型的数据，只提供一个纯地址。

如果此函数未能成功地执行(例如内存空间不足)，则返回空指针(NULL)。

- 用**calloc**函数开辟动态存储区：

其函数原型为：

```
1 | void * calloc(unsigned n,unsigned size);
```

其作用是在内存的动态存储区中分配n个长度为size 的连续空间,这个空间一般比较大, 足以保存一个数组。

用calloc函数可以为二维数组开辟动态存储空间, n为数组元素个数, 每个元素长度为size。这就是动态数组。函数返回指向所分配域的第一个字节的指针; 如果分配不成功, 返回NULL。

如:

```
1 | p= calloc(50,4);           //开辟50x4个字节的临时分配域,把首地址赋给指针变量p
```

- 用realloc函数重新分配动态存储区

其函数原型为:

```
1 | void * realloc(void * p,unsigned int size);
```

如果已经通过 malloc 函数或 calloc 函数获得了动态空间, 想改变其大小, 可以用 realloc 函数重新分配。

用realloc函数将p所指向的动态空间的大小改变为size。p的值不变。如果重分配不成功,返回NULL。

如:

```
1 | realloc(p,50);           //将p所指向的已分配的动态空间改为50字节
```

- 用free函数释放动态存储区

其函数原型为:

```
1 | void free(void * p);
```

其作用是释放指针变量p所指向的动态空间, 使这部分空间能重新被其他变量使用。p应是最近一次调用 calloc或malloc函数时得到的函数返回值。

如:

```
1 | free(p);                 //释放指针变量p所指向的已分配的动态空间
```

free函数无返回值。

注意: 以上4个函数的声明在 stdlib.h 头文件中,在用这些函数时应当用 “# include<stdlib.h>” 指令把 stdlib.h 头文件包含到程序文件中。

void指针类型

C99允许使用基类型为void的指针类型。可以定义一个基类型为void的指针变量 (即 void* 型变量) , 它不指向任何类型的数据。请注意: 不要把 “指向void类型” 理解为能指向 “任何的类型” 的数据, 而应理解为 “指向空类型” 或 “不指向确定的类型” 的数据。在将它的值赋给另一指针变量时由系统对它进行类型转换, 使之适合于被赋值的变量的类型。

例如:

```
1 | int a=3;                  //定义a为整型变量
2 |
3 | int *p1=&a;               //p1指向int型变量
4 |
```

```

5 char *p2; //p2指向char型变量
6
7 void *p3; //p3为无类型指针变量(基类型为void型)
8
9 p3=(void*)p1; //将p1的值转换为void*类型,然后赋值给p3
10
11 p2= (char*)p3; //将p3的值转换为char*类型,然后赋值给p2
12
13 printf("%d",* p1); //合法,输出整型变量a的值
14
15 p3= &a; printf("%d",* p3); //错误,p3是无指向的,不能指向a

```

这种空类型指针在形式上和其他指针一样,遵循C语言对指针的有关规定,它也有基类型,只是它的基类型是void。

可以这样定义:

```

1 void * p; //定义p是void*型的指针变量

```

void* 型指针代表“无指向的地址”,这种指针不指向任何类型的数据。不能企图通过它存取数据,在程序中它只是过渡性的,只有转换为有指向的地址,才能存取数据。

C 99这样处理,更加规范,更容易理解,概念也更清晰。

现在所用的一些编译系统在进行地址赋值时,会自动进行类型转换。

例如:

```

1 int * pt;
2 pt= (int*)mcaloc(100); //mcaloc(100)是void *型,把它转换为int*型
3 可以简化为
4 pt= mcaloc(100); //自动进行类型转换

```

赋值时,系统会先把 mcaloc(100) 转换为 pt 的类型,即 (int*) 型,然后赋给 pt,这样 pt 就指向存储区的首字节,在其指向的存储单元中可以存放整型数据。

9. 有关指针的小结

(1)首先要准确理解指针的含义。“指针”是C语言中一个形象化的名词,形象地表示“指向”的关系,其在物理上的实现是通过地址来完成的。正如高级语言中的“变量”,在物理上是“命名的存储单元”。Windows中的“文件夹”实际上是“目录”。离开地址就不可能弄清楚什么是指针。明确了“指针就是地址”,就比较容易理解了,许多问题也迎刃而解了。

例如:

- &a 是变量 a 的地址,也可称为变量 a 的指针。
- 指针变量是存放地址的变量,也可以说,指针变量是存放指针的变量。
- 指针变量的值是一个地址,也可以说,指针变量的值是一个指针。
- 指针变量也可称为地址变量,它的值是地址。
- &是取地址运算符, &a 是 a 的地址,也可以说, &是取指针运算符。&a 是变量 a 的指针(即指向变量 a 的指针)。
- 数组名是一个地址,是数组首元素的地址,也可以说,数组名是一个指针,是数组首元素的指针。
- 函数名是一个指针(指向函数代码区的首字节),也可以说函数名是一个地址(函数代码区首字节的地址)。
- 函数的实参如果是数组名,传递给形参的是一个地址,也可以说,传递给形参的是一个指针。

(2) 在C语言中，所有的数据都是有类型的，例如常量123并不是数学中的常数123，数学中的123是没有类型的，123和123.0是一样的，而在C语言中，所有数据都要存储在内存的存储单元中，若写成123，则认为是整数，按整型的存储形式存放，如果写成123.0，则认为是单精度实数，按单精度实型的存储形式存放。此外，不同类型数据有不同的运算规则。可以说，C语言中的数据都是“有类型的数据”，或称“带类型的数据”。

对地址而言，也是同样的，它也有类型，首先，它不是一个数值型数据，不是按整型或浮点型方式存储，它是按指针型数据的存储方式存储的（虽然在VisualC++中也为指针变量分配4个字节，但不同于整型数据的存储形式）。指针型存储单元是专门用来存放地址的，指针型数据的存储形式就是地址的存储形式。

其次，它不是一个简单的纯地址，还有一个指向的问题，也就是说它指向的是哪种类型的数据。如果没有这个信息，是无法通过地址存取存储单元中的数据的。所以，一个地址型的数据实际上包含3个信息：

- ①表示内存编号的纯地址。
- ②它本身的类型，即指针类型。
- ③以它为标识的存储单元中存放的是什么类型的数据，即基类型。

例如：已知变量为a为int型，&a为a的地址，它就包括以上3个信息，它代表的是一个整型数据的地址，int是&a的基类型（即它指向的是int型的存储单元）。可以把②和③两项合成一项，如“指向整型数据的指针类型”或“基类型为整型的指针类型”，其类型可以表示为“int*”型。这样，对地址数据来说，也可以说包含两个要素：内存编号（纯地址）和类型（指针类型和基类型）。这样的地址是“带类型的地址”而不是纯地址。

(3)要区别指针和指针变量。指针就是地址，而指针变量是用来存放地址的变量。有人认为指针是类型名，指针的值是地址。这是不对的。类型是没有值的，只有变量才有值，正确的说法是指针变量的值是一个地址。不要杜撰出“地址的值”这样莫须有的名词。地址本身就是一个值。

(4)什么叫“指向”？地址就意味着指向，因为通过地址能找到具有该地址的对象。对于指针变量来说，把谁的地址存放在指针变量中，就说此指针变量指向谁。但应注意：并不是任何类型数据的地址都可以存放在同一个指针变量中的，只有与指针变量的基类型相同的数据的地址才能存放在相应的指针变量中。

例如：

```
1  int a,*p;    //p是int类型的指针变量,基类型是int型
2
3  float b;
4
5  p= &a;      //a是int型,合法
6
7  p=&b;      //b是float型,类型不匹配
```

既然许多数据对象（如变量数组、字符串和函数等）都在内存中被分配存储空间，就有了地址，也就有了指针。可以定义一些指针变量，分别存放这些数据对象的地址，即指向这些对象。void* 指针是一种特殊的指针，不指向任何类型的数据。如果需要用此地址指向某类型的数据，应先对地址进行类型转换。可以在程序中进行显式的类型转换，也可以由编译系统自动进行隐式转换。无论用哪种转换，读者必须了解要进行类型转换。

(5)要深入掌握在对数组的操作中正确地使用指针，搞清楚指针的指向。一维数组名代表数组首元素的地址，如：

```
1  int *p,a[10];
2  p=a;
```

p是指向 int 型类型的指针变量，显然，p只能指向数组中的元素（int 型变量），而不是指向整个数组。在进行赋值时一定要先确定赋值号两侧的类型是否相同，是否允许赋值。

对"p=a;"，准确地说应该是：p指向a数组的首元素，在不引起误解的情况下，有时也简称为：p指向a数组，但读者对此应有准确的理解。同理，p 指向字符串，也应理解为p指向字符串中的首字符。

(6)有关指针变量的归纳比较

指针变量的类型及含义

变量定义	类型表示	含义
<code>int i;</code>	<code>int</code>	定义整型变量
<code>int *p;</code>	<code>int *</code>	定义p为指向整型数据的指针变量
<code>int a[5];</code>	<code>int [5]</code>	定义整型数组a，它有5个元素
<code>int *p[4];</code>	<code>int * [4]</code>	定义指针数组p，它由4个指向整型数据的指针元素组成
<code>int (*p) [4];</code>	<code>int(*) [4]</code>	p为指向包含4个元素的一维数组的指针变量
<code>int f();</code>	<code>int ()</code>	f为返回整型函数值的函数
<code>int *p();</code>	<code>int * ()</code>	p为返回一个指针的函数，该指针指向整型数据
<code>int(*p)();</code>	<code>int(*)()</code>	p为指向函数的指针，该函数返回一个整型值
<code>int **p;</code>	<code>int **</code>	p是一个指针变量，它指向一个指向整型数据的指针变量
<code>void *p;</code>	<code>void *</code>	p是一个指针变量，基类型为void（空类型），不指向具体的对象

为便于比较,在表中包括了其他一些类型的定义。

(7)指针运算。

①指针变量加（减）一个整数。

例如: `p++`, `p--`, `p+i`, `p-i`, `p+=i`, `p-=i` 等均是指针变量加(减)一个整数。

将该指针变量的原值（是一个地址）和它指向的变量所占用的存储单元的字节数相加（减）。

②指针变量赋值。

将一个变量地址赋给一个指针变量。

例如：


```

1  p= &a;                //(将变量a的地址赋给p)
2
3  P= array;             //(将数组array首元素地址赋给p)
4
5  p= &array[i];         //(将数组array 第i个元素的地址赋给p)
6
7  p= max;               //(max为已定义的函数.将max的人口地址赋给p)
8
9  p1= p2;               //(p1和p2是基类型相同指针变量,将p2的值赋给p1)
10

```

注意：不应把一个整数赋给指针变量。

③两个指针变量可以相减。

如果两个指针变量都指向同一个数组中的元素，则两个指针变量值之差是两个指针之间的元素个数。

④两个指针变量比较。

若两个指针指向同一个数组的元素，则可以进行比较。指向前面的元素的指针变量“小于”指向后面元素的指针变量。如果p1和p2不指向同一数组则比较无意义。

(8)指针变量可以有空值，即该指针变量不指向任何变量，可以这样表示：`p= NULL;`

其中，NULL是一个符号常量，代表整数0。在 `stdio.h` 头文件中对NULL进行了定义：`#define NULL 0`

它使p指向地址为0的单元。系统保证使该单元不作它用(不存放有效数据)。

应注意，p的值为NULL与未对p赋值是两个不同的概念。前者是有值的（值为0），不指向任何变量，后者虽未对p赋值但并不等于p无值,只是它的值是一个无法预料的值，也就是p可能指向一个事先未指定的单元。这种情况是很危险的。因此，在引用指针变量之前应对它赋值。

任何指针变量或地址都可以与NULL作相等或不相等的比较，例如：

```

1  if(p==NULL){
2      ...
3  }

```

指针是C语言中很重要的概念，是C的一个重要特色。

使用指针的优点：

①提高程序效率；

②在调用函数时当指针指向的变量的值改变时,这些值能够为主调函数使用，即可以从函数调用得到多个可改变的值；

③可以实现动态存储分配。

同时应该看到，指针使用实在太灵活，对熟练的程序人员来说，可以利用它编写出颇有特色、质量优良的程序，实现许多用其他高级语言难以实现的功能，但也十分容易出错，而且这种错误往往比较隐蔽。指针运用的错误可能会使整个程序遭受破坏，比如由于未对指针变量p赋值就向 `*p` 赋值，就可能破坏了有用的单元的内容。如果使用指针不当，会出现隐蔽的、难以发现和排除的故障。因此，使用指针要十分小心谨慎，要多上机调试程序，以弄清一些细节，并积累经验。

第六章 自定数据类型

1. 定义和使用结构体变量

由不同类型数据组成的组合型的数据结构，它称为结构体（`structre`）。

声明一个结构体类型的一般形式为：

```
1 struct 结构体名{
2     成员列表
3 };
```

`struct` 是声明结构体类型时必须使用的关键字，不能省略。

注意：结构体类型的名字是由一个关键字 `struct` 和结构体名组成的。结构体名是由用户指定的，又称“结构体标记”（structure tag），以区别于其他结构体类型。上面的结构体声明中 `Student` 就是结构体名（结构体标记）。

花括号内是该结构体所包括的子项，称为结构体的成员（member）。对各成员都应进行类型声明，即：

```
1 类型名 成员名；
```

“成员表列”（member list）也称为“域表”（field list），每一个成员是结构体中的一个域。成员名命名规则与变量名相同。

说明：

(1)结构体类型并非只有一种，而是可以设计出许多种结构体类型，还可以根据需要建立结构体类型，各自包含不同的成员。

(2)成员可以属于另一个结构体类型。

例如：

```
1 struct Date //声明一个结构体类型struct Date
2 {
3     int month; //月
4     int day; //日
5     int year; //年
6 };
7 struct Student //声明一个结构体类型struct Student
8 {
9     int num;
10    char name[20];
11    char sex;
12    int age;
13    struct Date birthday; //成员birthday属于struct Date类型
14    char addr[30];
15 };
```

定义结构体类型变量

1. 先声明结构体类型，再定义该类型的变量

上面已声明了一个结构体类型 `struct Student`，可以用它来定义变量。例如：

```

1 struct Student student1,student2;
2 -----
3 |           |           |
4 结构体类型名      结构体变量名

```

这种方式是声明类型和定义变量分离，在声明类型后可以随时定义变量，比较灵活。

2. 在声明类型的同时定义变量

例如：

```

1 struct Student
2 {
3     int num;
4     char name[20];
5     char sex;
6     int age;
7     float score;
8     char addr[30];
9 }student1,student2;

```

它的作用与第一种方法相同，但是在定义struct Student 类型的同时定义两个struct Student类型的变量 student1 和student2。

这种定义方法的一般形式为：

```

1 struct 结构体名{
2     成员列表
3 }变量名表列；

```

声明类型和定义变量放在一起进行，能直接看到结构体的结构，比较直观，在写小程序时用此方式比较方便,但写大程序时，往往要求对类型的声明和对变量的定义分别放在不同的地方，以使程序结构清晰，便于维护，所以一般不多用这种方式。

3. 不指定类型名而直接定义结构体类型变量

其一般形式为：

```

1 struct{
2     成员表列
3 }变量名表列；

```

指定了一个无名的结构体类型,它没有名字（不出现结构体名）。显然不能再以此结构体类型去定义其他变量。这种方式用得不多。

说明：

(1)结构体类型与结构体变量是不同的概念，不要混淆。只能对变量赋值、存取或运算，而不能对一个类型赋值、存取或运算。在编译时，对类型是不分配空间的,只对变量分配空间。

(2)结构体类型中的成员名可以与程序中的变量名相同，但二者不代表同一对象。例如，程序中可以另定义一个变量 num，它与 struct Student 中的 num 是两回事，互不干扰。

(3)对结构体变量中的成员（即“域”），可以单独使用，它的作用与地位相当于普通变量。关于对成员的引用方法见下节。

结构体变量的初始化和引用

在定义结构体变量时,可以对它初始化.即赋予初始值。然后可以引用这个变量,例如输出它的成员的值。

(1)在定义结构体变量时可以对它的成员初始化。初始化列表是用花括号括起来的一些常量,这些常量依次赋给结构体变量中的各成员。

注意:是对结构体变量初始化,而不是对结构体类型初始化。

C99 标准允许对某一成员初始化,如:

```
1 struct Student b= { .name= ' Zhang Fang ' }; //在成员名前有成员运算符"."
```

“ . name ” 隐含代表结构体变量 `b` 中的成员 `b.name`。其他未被指定初始化的数值型成员被系统初始化为 0, 字符型成员被系统初始化为 `'\0'`, 指针型成员被系统初始化为 `NULL`。

(2)可以引用结构体变量中成员的值, 引用方式为:

```
1 结构体变量名.成员名
```

“ . ” 是成员运算符,它在所有的运算符中优先级最高,因此可以把 `b.name` 作为一个整体来看待,相当于一个变量。

注意:不能企图通过输出结构体变量名来达到输出结构体变量所有成员的值。

下面用法不正确:

```
1 printf("%s\n",b); //企图用结构体变量名输出所有成员的值
```

只能对结构体变量中的各个成员分别进行输入和输出。

(3)如果成员本身又属一个结构体类型,则要用若干个成员运算符,一级一级地找到最低的一级的成员。只能对最低级的成员进行赋值或存取以及运算。如果在结构体 `struct Student` 类型的成员中包含另一个结构体 `struct date` 类型的成员 `birthday` (为一个结构体),则引用成员的方式为:

```
1 student1.num           //(结构体变量student1中的成员num)
2 student1.birthday.month //(结构体变量student1中的成员birthday中的成员month)
```

不能用 `student1.birthday` 来访问 `student1` 变量中的成员 `birthday`, 因为 `birthday` 本身是一个结构体成员。

(4)对结构体变量的成员可以像普通变量一样进行各种运算(根据其类型决定可以进行的运算)。

例如:

```
1 student2.score = student1.score;           //(赋值运算)
2 sum = student1.score+student2.score;       //(加法运算)
3 student1.age++;                             //(自加运算)
```

由于“ . ” 运算符的优先级最高,因此 `student1.age++` 是对(`student1.age`)进行自加运算,而不是先对 `age` 进行自加运算。

(5)同类的结构体变量可以互相赋值,如:

```
1 student1 = student2; //假设student1和student2已定义为同类型的结构体变量
```

(6)可以引用结构体变量成员的地址,也可以引用结构体变量的地址。

例如:

```
1 scanf("%d",&student1.num); //(输入student1. num的值)
2 printf("%o",&student1);    //(输出结构体变量student1的起始地址)
```

但不能用以下语句整体读入结构体变量, 例如:

```
1 scanf("%d,%s,%c,%d,%f,%s\n",&.student1);
```

说明: 结构体变量的地址主要用作函数参数, 传递结构体变量的地址。

2. 使用结构体数组

(1)定义结构体数组一般形式是:

```
1 struct 结构体名{
2     成员表列
3 }数组名[数组长度];
```

先声明一个结构体类型(如 `struct Person`), 然后再用此类型定义结构体数组:

```
1 结构体类型 数组名[数组长度];
2
3 struct Person{
4     char name[20];
5     int age;
6 };
7
8 struct Person leader[3]; //leader是结构体数组名
```

(2)对结构体数组初始化的形式是在定义数组的后面加上:

```
1 结构体类型 数组名[数组长度] = {初值表列};
```

如:

```
1 struct Person leader[3] = {"Li",0,"Zhang",0,"Sun",0};
```

3. 结构体指针

指向结构体变量的指针

指向结构体对象的指针变量既可指向结构体变量, 也可指向结构体数组中的元素。指针变量的基类型必须与结构体变量的类型相同。

例如:

```
1 struct Student* pt;    //pt可以指向structStudent类型的变量或数组元素
```

说明: 为了使用方便和直观, C语言允许把 `(*p).num` 用 `p->num` 代替, “->”代表一个箭头, `p->num` 表示 `p` 所指向的结构体变量中的 `num` 成员。同样, `(*p).name` 等价于 `p->name`。“->”称为指向运算符。

如果 `p` 指向一个结构体变量 `stu`, 以下3种用法等价:

- ① `stu.成员名` (如`stu.num`);
- ② `(*p).成员名` (如`(*p).num`);
- ③ `p->成员名` (如`p->num`)。

指向结构体数组的指针

可以用指针变量指向结构体数组的元素。

例如：有3个学生的信息，放在结构体变量中，要求输出全部学生的信息。


- (1)声明结构体类型 `struct Student`，并定义结构体数组，同时初始化;
- (2)定义一个指向 `struct Student` 类型数据的指针变量p;
- (3)使P指向结构体数组的首元素，输出它指向的元素中的有关信息;
- (4)使p指向结构体数组的下一个元素，输出它指向的元素中的有关信息;
- (5)再使p指向结构体数组的下一个元素，输出它指向的元素中的有关信息。

编写程序:

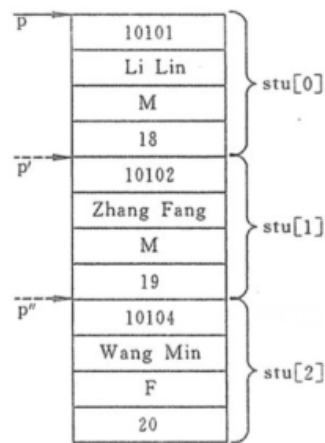
```
1  #include<stdio.h>
2  struct Student{           //声明结构体类型structStudent
3      int num;
4      char name[20];
5      char sex;
6      int age;
7  };
8  struct Student stu[3]={10101,"Li Lin",'M',18},{10102,"Zhang Fang",'M',19},
9  {10104,"Wang Min",'F',20}}; //定 义结构体数组并初始化
10 int main(){
11     struct Student* p;//定义指向structStudent结构体变量的指针变量
12     printf(" No. Name          sex age\n");
13     for (p= stu;p<stu+3;p++){
14         printf(" %5d %-20s %2c %4d\n",p->num,p->name,p->sex,p->age); //输出结
15     }
16     return 0;
17 }
```

运行结果：

No.	Name	sex	age
10101	Li Lin	M	18
10102	Zhang Fang	M	19
10104	Wang Min	F	20

 程序分析：p 是指向 struct Student 结构体类型数据的指针变量。在 for 语句中先

使 p 的初值为 stu，也就是数组 stu 中序号为 0 的元素（即 stu[0]）的起始地址，见图 9.6 中 p 的指向。在第 1 次循环中输出 stu[0] 的各个成员值。然后执行 p++，使 p 自加 1。p 加 1 意味着 p 所增加的值为结构体数组 stu 的一个元素所占的字节数（在 Visual C++ 环境下，本例中一个元素所占的字节数理论上为 4+20+1+4=29 字节，实际分配 32 字节）。执行 p++ 后 p 的值等于 stu+1，p 指向 stu[1]，见图 9.6 中 p' 的指向。在第 2 次循环中输出 stu[1] 的各成员值。在执行 p++ 后，p 的值等于 stu+2，它的指向见图 9.6 中的 p''，再输出 stu[2] 的各成员值。在执行 p++ 后，p 的值变为 stu+3，已不再小于 stu+3 了，不再执行循环。



注意：

(1)如果p的初值为 stu，即指向 stu 的序号为0的元素，p加1后，就指向下一个元素。

例如：

```
1  (++p)->num    //先使p自加1，然后得到p指向的元素中的num成员值(10102)
2  (p++)->num    //先求得p->num的值(即10101)，然后再使p自加1，指向stu[1]
```

请注意以上二者的不同。

(2)程序定义了p是一个指向 struct Student 类型对象的指针变量，它用来指向一个 struct Student 类型的对象(p的值是 stu 数组的一个元素(如 stu[0] 或 stu[1])的起始地址)，不应用来指向 stu 数组元素中的某一成员。

例如，下面的用法是不对的：

```
1  p= stu[1].name; //stu[1].name是stu[1]元素中的成员name 的首字符的地址
```

编译时将给出“警告”信息，表示地址的类型不匹配。不要认为反正p是存放地址的，可以将任何地址赋给它。如果一定要将某一成员的地址赋给p，可以用强制类型转换，先将成员的地址转换成p的类型。例如：

```
1  p= (struct Student *)stu[0].name;
```

此时，p的值是 stu[0] 元素的name成员的起始地址。可以用“printf(“%s”,p);”输出 stu[0] 中成员name的值。但是，p仍保持原来的类型。如果执行“printf(“%s”,p+1);”，则会输出 stu[1] 中name的值。执行 p++ 时，p的值的增量是结构体 struct Student 的长度。

用结构体变量和结构体变量的指针作函数参数

将一个结构体变量的值传递给另一个函数，有3个方法：

(1)用结构体变量的成员作参数。例如，用 stu[1]. num 或 stu[2].name 作函数实参，将实参值传给形参。用法和用普通变量作实参是一样的，属于“值传递”方式。应当注意实参与形参的类型保持一致。

(2)用结构体变量作实参。用结构体变量作实参时，采取的也是“值传递”的方式，将结构体变量所占的内存单元的内容全部按顺序传递给形参，形参也必须是同类型的结构体变量。在函数调用期间形参也要占用内存单元。这种传递方式在空间和时间上开销较大，如果结构体的规模很大时，开销是很可观的。此外，由于采用值传递方式，如果在执行被调用函数期间改变了形参(也是结构体变量)的值，该值不能返回

主调函数，这往往造成使用上的不便。因此一般较少用这种方法。

(3)用指向结构体变量(或数组元素)的指针作实参，将结构体变量(或数组元素)的地址传给形参。

4. 用指针处理链表

什么是链表？

链表是动态地进行存储分配的一种结构。

作用是为了避免内存的浪费，它是根据需要开辟内存单元设定的。

单向链表

由 head 的 next 指向下个节点

头指针：head（整个链表都必须包含head）

结点：必须包含两部分（1）用户需要用的实际数据（2）下一个节点的地址

空指针（表尾）：NULL

建立链表（利用结构体）

```
1 struct Student
2 {   int num;
3     float score;
4     struct Student *next; //next是指针变量，指向下一个结构体的地址
5 };
```

输出链表

```
1 void output(struct student *head) // 定义一个链表输出的函数
2 {
3     struct student *p; // 定义结构体指针变量p1，用于结点的后移，以实现输出操作
4     p = head; // 将head赋给p1，以实现对该链表的操作
5     if (p != NULL) // 建立一个while循环，结束条件是到达尾结点
6     do
7     {
8         printf("%d\n%f\n", p1->num, p1->score); // 输出结点中的数值部分
9         p1 = p1->next; // 将下一个结点的位置赋给p1
10    }while(p != NULL); //当p不是空地址时循环
11
12 }
```

注意：

malloc() 分配内存后最后记得 free() 释放内存。

可以参看以下博客学习：

<https://blog.csdn.net/linwh8/article/details/49648601>

5. 共同体类型

使几个不同的变量共享同一段内存的结构，称为"共同体"类型结构。

定义共同体类型变量的的一般形式为：

```
1 union 共用体名{
2     成员列表
3 }变量列表;
```

例如：

```
1 union Data{           //表示不同类型的变量i,ch,f可以存放到同一段存储单元中
2     int t;
3     char ch;
4     float f;
5 }a,b,c;               //在声明类型同时定义变量
```

也可以将类型声明与变量定义分开：

```
1 union Data{           //声明共用体类型
2     int I;
3     char ch;
4     float f;
5 };
6 union Data a,b,c;     //用共用体类型定义变量
```

即先声明一个 `union Data` 类型，再将 `a,b,c` 定义为 `union Data` 类型的变量。

当然也可以直接定义共用体变量，例如：

```
1 union{                //没有定义共用体类型名
2     int i;
3     char ch;
4     float f;
5 }a,b,c;
```

可以看到，“共用体”与“结构体”的定义形式相似。但它们的含义是不同的。

结构体变量所占内存长度是各成员占的内存长度之和。每个成员分别占有其自己的内存单元。而共用体变量所占的内存长度等于最长的成员的长度。例如，上面定义的“共用体”变量 `a`, `b`, `c` 各占4个字节（因为一个 `float` 型变量占4个字节），而不是各占 $4+1+4=9$ 个字节。

引用共用体变量的方式

只有先定义了共同体变量才能引用它，但应注意，不能引用共同变量，而只能引用共同体变量中的成员。

例如：【上面定义的 `a,b,c` 共用体】

```
1 a.i           //引用共同体变量中的整型变量i
2 a.ch          //引用共同体变量中的整型变量i
3 a.f           //引用共同体变量中的整型变量i
4
5 //不能只引用共同体变量,下面的引用就是错误的
6 printf("%d",a);
7
8 //正确的写法为
9 printf("%d",a.i);
10 printf("%c",a.ch);
11 printf("%f",a.f);
```

共用体类型数据的特点

在使用共用体型数据时要主要以下特点：

`workday` 和 `weekend` 被定义为枚举变量,花括号中的 `sun,mon,...,sat` 称为枚举元素或枚举常量。它们是由用户指定的名字。枚举变量和其他数值型量不同,它们的值只限于花括号中指定的值之一。例如枚举变量 `workday` 和 `weekend` 的值只能是 `sun` 到 `sat` 之一。

```
1 | workday= mon; //正确,mon是指定的枚举常量之一
2 |
3 | weekend= sun; //正确,sun是指定的枚举常量之一
4 |
5 | weekday = monday; //不正确,monday不是指定的枚举常量之一
```

枚举常量是由程序设计者命名的,用什么名字代表什么含义,完全由程序员根据自己的需要而定,并在程序中作相应处理。

也可以不声明有名字的枚举类型,而直接定义枚举变量,例如:

```
1 | enum{sun,mon,tue,wed,thu,fri,sat} workday,weekend;
```

声明枚举类型用 `enum` 开头。

声明枚举类型的一般形式为:

```
1 | enum [枚举名]{枚举元素.....};
```

说明:

(1) C编译对枚举类型的枚举元素按常量处理,故称枚举常量。不要因为它们是标识符(有名字)而把它们看作变量,不能对它们赋值。

(2) 每一个枚举元素都代表一个整数,C语言编译按定义时的顺序默认它们的值为 `0,1,2,3,4,5...`。在上面的定义中,`sun` 的值自动设为0,`mon` 的值为1,..`sat` 的值为6。如果有赋值语句:

```
1 | workday= mon; //相当于
2 | workday= 1; //枚举常量是可以引用和输出的。例如:
3 | printf("%d",workday); //将输出整数1。
```

也可以人为地指定枚举元素的数值,在定义枚举类型时显式地指定,例如:

```
1 | enum weekday{sun=7,mon=1,tue,wed,thu,fri,sat} workday,week_end;
```

指定枚举常量 `sun` 的值为7,`mon` 为1,以后顺序加1,`sat` 为6。

由于枚举型变量的值是整数,因此 `c99` 把枚举类型也作为整型数据中的一种,即用户自行定义的整数类型。

(3)枚举元素可以用来作判断比较。例如:

```
1 | if( workday== mon)..
2 | if( workday> sun)...
```

枚举元素的比较规则是按其在初始化时指定的整数来进行比较的。如果定义时未人为指定,则按上面的默认规则处理,即第1个枚举元素的值为0,故 `mon > sun`, `sat > fri`。

7. 用typedef声明新类型名

用typedef指定新的类型名来代替已有的类型名。

有以下两种情况：

1. 简单地用一个新的类型名代替原有的类型名

例如：

```
1 typedef int Integer;    //指定用Integer为类型名,作用与int相同
2 typedef float Real;     //指定用Real为类型名,作用与float相同
```

指定用 Integer 代表int类型,Real代表float。这样，以下两行等价：

```
1 int i,j;
2 float a,b;
3
4 Integer i,j;
5 Real a,b;
```

2. 命名一个简单的类型名代替复杂的类型表示方法

从前面已知，除了简单的类型（如int，float等）、C程序中还会用到许多看起来比较复杂的类型，包括结构体类型、共用体类型枚举类型、指针类型、数组类型等，如：

```
1 float*[] (指针数组)
2 float( * ) [5] (指向5个元素的一维数组的指针)
3 double * (double * ) (定义函数，函数的参数是double*型数据，即指向double数据的指针，函数返回值也是指向double数据的指针)
4 double( * )() (指向函数的指针，函数返回值类型为double)
5 int * ( * ( * ) [10]) (void) (指向包含10个元素的一维数组的指针，数组元素的类型为函数指针(函数的地址)，函数没有参数，函数返回值是int 指针)
```

有些类型形式复杂，难以理解，容易写错。C允许程序设计者用一个简单的名字代替复杂的类型形式。

例如：

(1)命名一个新的类型名代表结构体类型：

```
1 typedef struct int month;
2 int year;
3 } Date;
```

以上声明了一个新类型名Date，代表上面的一个结构体类型。然后可以用新的类型名Date去定义变量，如：

```
1 Date birthday; //定义结构体类型变量birthday ,不要写成struct Date birthday;
2 Date* p;       //定义结构体指针变量p.指向此结构体类型数据
```

(2)命名一个新的类型名代表数组类型：

```
1 typedef int Num[ 100]; //声明Num为整型数组类型名
2 Num a;                //定义a为整型数组名,它有100个元素
```

(3)命名一个新的类型名代表指针类型：

```
1 typedef char * String; //声明String为字符指针类型
2 String p,s[10];        //定义p为字符指针变量,s为字符指针数组
```

(4)命名一个新的类型名代表指向函数的指针类型：

```
1 typedef int(*Pointer)(); //声明Pointer为指向函数的指针类型,该函数返回整型值
2 Pointer p1,p2;          //p1,p2为Pointer类型的指针变量
```

归纳起来，声明一个新的类型名的方法是：

```
1 ①先按定义变量的方法写出定义体(如:int i;)。
2 ②将变量名换成新类型名(例如:将i换成Count)。
3 ③在最前面加typedef(例如:typedef int Count)。
4 ④然后可以用新类型名去定义变量。
```

简单地说，就是按定义变量的方式，把变量名换上新类型名，并且在最前面加typedef，就声明了新类型名代表原来的类型。

以定义上述的数组类型为例来说明：

```
1 ①先按定义数组变量形式书写: int a[100]。
2 ②将变量名a换成自己命名的类型名:int Num[100]。
3 ③在前面加上typedef,得到typedef int Num[100]。
4 ④用来定义变量:Num a;
5 相当于定义了:int a[100];
6 同样,对字符指针类型,也是:
7 ①char * p; //定义变量p的方式
8 ②char * String; //用新类型名String 取代变量名p
9 ③typedef char * String; //加typedef
10 ④String p; //用新类型名String定义变量,相当char*p;
```

习惯上，常把用typedef声明的类型名的第1个字母用大写表示，以便与系统提供的标准类型标识符相区别。

第七章 对文件的输入输出

1. C文件的有关基本知识

什么是文件

文件有不同的类型,在程序设计中,主要用到两种文件:

(1)程序文件。包括源程序文件(后缀为.c)、目标文件(后缀为.obj)、可执行文件(后缀为.exe)等。这种文件的内容是程序代码。

(2)数据文件。文件的内容不是程序，而是供程序运行时读写的数据，如在程序运行过程中输出到磁盘（或其他外部设备）的数据，或在程序运行过程中供读人的数据。如一批学生的成绩数据、货物交易的数据等。

- 文件(file)是程序设计中一个重要的概念。所谓“文件”一般指存储在外部介质上数据的集合。一批数据是以文件的形式存放在外部介质(如磁盘)上的。
- 输入输出是数据传送的过程，数据如流水一样从一处流向另一处，因此常将输入输出形象地称为流(stream)，即**数据流**。

- C语言把文件看作一个字符（或字节）的序列，即由一个一个字符（或字节）的数据顺序组成。一个输入输出流就是一个字符流或字节（内容为二进制数据）流。
- C的**数据文件**由一连串的字符（或字节）组成，而不考虑行的界限，两行数据间不会自动加分隔符，对文件的存取是以字符（字节）为单位的。**输入输出数据流的开始和结束仅受程序控制而不受物理符号（如回车换行符）控制**，这就增加了处理的灵活性。这种文件称为**流式文件**。

文件名

一个文件要有一个唯一的文件标识，以使用户识别和引用。

文件标识包括3部分: (1)文件路径; (2)文件名主干; (3)文件后缀。

文件路径表示文件在外部存储设备中的位置。如:

1	D:\CC\temp\file1.dat
2	-----
3	↑ ↑ ↑
4	文件路径 文件名主干 文件后缀

表示 `file1.dat` 文件存放在 `D` 盘中的 `CC` 目录下的 `temp` 子目录下。

文件名主干的命名规则遵循标识符的命名规则。

后缀 用来表示文件的性质，如:

1	doc	(word生成的文件)
2	txt	(文本文件)
3	dat	(数据文件)
4	c	(C语 言源程序文件)
5	cpp	(C++源程序文件)
6	for	(FORTRAN语言源程序文件)
7	pas	(Pascal语 言源程序文件)
8	obj	(目标文件)
9	exe	(可执行文件)
10	ppt	(电子幻灯文件)
11	bmp	(图形文件)
12	...	

文件的分类

根据数据的组织形式，数据文件可分为**ASCII文件**和**二进制文件**。数据在内存中是以二进制形式存储的，如果不加转换地输出到外存，就是二进制文件，可以认为它就是存储在内存的数据的映像，所以也称之为映像文件(`imagefile`)。如果要求在外存上以ASCII代码形式存储,则需要存储前进行转换。ASCII文件又称文本文件(`text file`),每一个字节存放一个字符的ASCII代码。

一个数据在磁盘上怎样存储呢?

字符一律以ASCII形式存储，数值型数据既可以用ASCII形式存储，也可以用二进制形式存储。

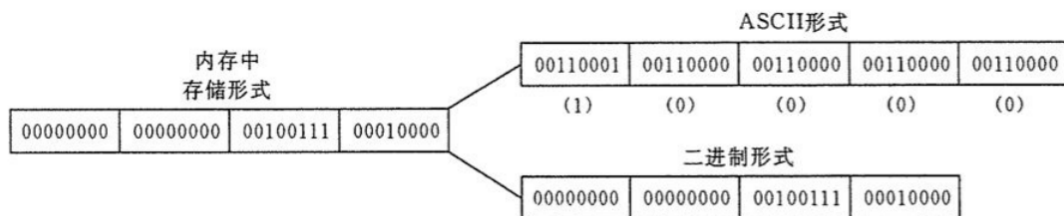


图 10.1

文件缓冲区

ANSI C标准采用“缓冲文件系统”处理数据文件，所谓缓冲文件系统是指系统自动地在内存区为程序中每一个正在使用的文件开辟一个文件缓冲区。从内存向磁盘输出数据必须先送到内存中的缓冲区，装满缓冲区后才一起送到磁盘去。如果从磁盘向计算机读入数据，则一次从磁盘文件将一批数据输入到内存缓冲区(充满缓冲区)，然后再从缓冲区逐个地将数据送到程序数据区(给程序变量)，见图10.2。这样做是为了节省存取时间，提高效率，缓冲区的大小由各个具体的C编译系统确定。

说明：每一个文件在内存中只有一个缓冲区，在向文件输出数据时，它就作为输出缓冲区,在从文件输入数据时，它就作为输入缓冲区。

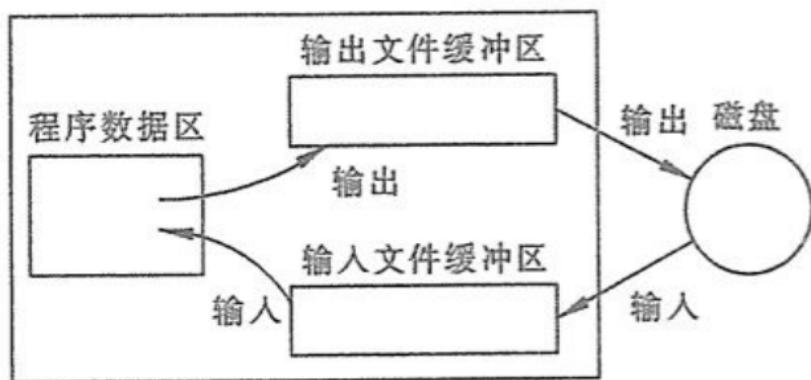


图 10.2

文件类型指针

缓冲文件系统中，关键的概念是“文件类型指针”，简称“文件指针”。每个被使用的文件都在内存中开辟一个相应的文件信息区，用来存放文件的有关信息(如文件的名称、文件状态及文件当前位置等)。这些信息是保存在一个结构体变量中的。该结构体类型是由系统声明的，取名为FILE。例如有一种C编译环境提供的 `stdio.h` 头文件中有以下的文件类型声明：

```
1  typedef struct{
2      short level;           //缓冲区“满”或“空”的程度
3      unsigned flags;       //文件状态标志
4      char fd;              //文件描述符
5      unsigned char hold;   //如缓冲区无内容不读取字符
6      short bsize;         //缓冲区的大小
7      unsigned char * buffer; //数据缓冲区的位置
8      unsigned char * curp;  //文件位置标记指针当前的指向
9      unsigned istemp;      //临时文件指示器
10     short token;          //用于有效性检查
11 }FILE;
```

不同的C编译系统的FILE类型包含的内容不完全相同，但大同小异。对以上结构体中的成员及其含义可不深究，只须知道其中存放文件的有关信息即可。

定义一个指向文件型数据的指针变量：

```
1 FILE * fp;
```

定义 `fp` 是一个指向 `FILE` 类型数据的指针变量。可以使 `fp` 指向某一个文件的文件信息区(是一个结构体变量)，通过该文件信息区中的信息就能够访问该文件。也就是说，通过文件指针变量能够找到与它关联的文件。如果有 n 个文件，应设 n 个指针变量，分别指向 n 个 `FILE` 类型变量，以实现对 n 个文件的访问，见图 10.3。

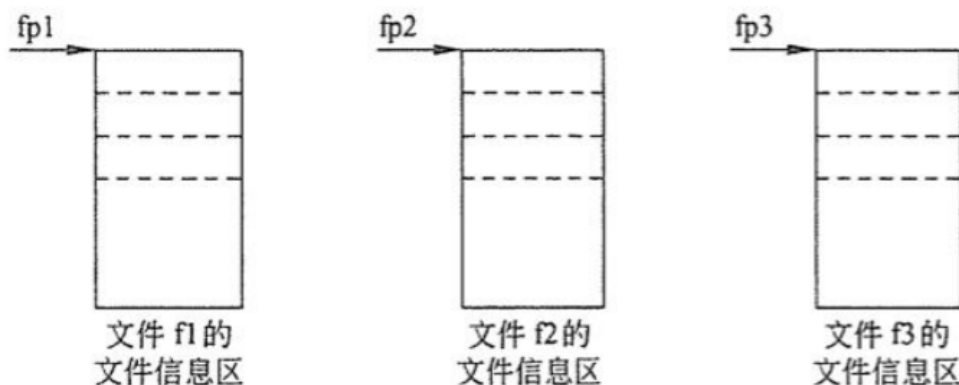


图 10.3

为方便起见，通常将这种指向文件信息区的指针变量简称为**指向文件的指针变量**。

注意：指向文件的指针变量并不是指向外部介质上的数据文件的开头，而是指向内存中的文件信息区的开头。

2. 打开与关闭文件

实际上，所谓“打开”是指为文件建立相应的信息区（用来存放有关文件的信息）和文件缓冲区（用来暂时存放输入输出的数据）。

在编写程序时，在打开文件的同时，一般都指定一个指针变量指向该文件，也就是建立起指针变量与文件之间的联系，这样，就可以通过该指针变量对文件进行读写了。所谓“关闭”是指撤销文件信息区和文件缓冲区，使文件指针变量不再指向该文件，显然就无法进行对文件的读写了。

用 `fopen` 函数打开数据文件

ANSI C 规定了用标准输入输出函数 `fopen` 来实现打开文件。

`fopen` 函数的调用方式为：

```
1 fopen(文件名, 使用文件方式);
```

例如：

```
1 fopen("a1", "r");
```

表示要打开名字为 `a1` 的文件，使用文件方式为“读入”(r代表read，即读入)。

`fopen` 函数的返回值是指向 `a1` 文件的指针(即 `a1` 文件信息区的起始地址)。

通常将 `fopen` 函数的返回值赋给一个指向文件的指针变量。如：


```
1 FILE* fp;           //定义一个指向文件的指针变量fp
2 fp= fopen("a1", "r"); //将fopen函数的返回值赋给指针变量fp
```

这样 fp 就和文件 a1 相联系了,或者说, fp 指向了 a1 文件。可以看出, 在打开一个文件时, 通知编译系统以下3个信息:

- ①需要打开文件的名字, 也就是准备访问的文件的名字;
- ②使用文件的方式(“读”还是“写”等);
- ③让哪一个指针变量指向被打开的文件。

表10. 1 使用文件方式

文件使用方式	含义	如果指定的文件不存在
r (只读)	为了输入数据, 打开一个已存在的文本文件	出错
w (只写)	为了输出数据, 打开一个文本文件	建立新文件
a (追加)	向文本文件尾添加数据	出错
rb (只读)	为了输入数据, 打开一个二进制文件	出错
wb (只写)	为了输出数据, 打开一个二进制文件	建立新文件
ab (追加)	向二进制文件尾添加数据	出错
"r+" (读写)	为了读和写, 打开一个文本文件	出错
"w+" (读写)	为了读和写, 建立一个新的文本文件	建立新文件
"a+" (读写)	为了读和写, 打开一个文本文件	出错
"rb+" (读写)	为了读和写, 打开一个二进制文件	出错
"wb+" (读写)	为了读和写, 建立一个新的二进制文件	建立新文件
"ab+" (读写)	为读写打开一个二进制文件	出错

- (1)用 r 方式打开的文件只能用于向计算机输入而不能用作向该文件输出数据, 而且该文件应该已经存在, 并存有数据, 这样程序才能从文件中读数据。不能用 r 方式打开一个并不存在的文件, 否则出错。
- (2)用 w 方式打开的文件只能用于向该文件写数据(即输出文件), 而不能用来向计算机输入。如果原来不存在该文件, 则在打开文件前新建一个以指定的名字命名的文件。如果原来已存在一个以该文件名命名的文件, 则在打开文件前先将该文件删去, 然后重新建立一个新文件。
- (3)如果希望向文件末尾添加新的数据(不希望删除原有数据), 则应该用a方式打开。但此时应保证该文件已存在; 否则将得到出错信息。打开文件时, 文件读写位置标记移到文件末尾。
- (4)用“r+”、“w+”、“a+”方式打开的文件既可用于输入数据, 也可用于输出数据。用“r+”方式时该文件应该已经存在, 以便计算机从中读数据。用“w+”方式则新建一个文件, 先向此文件写数据, 然后可以读此文件中的数据。用“a+”方式打开的文件, 原来的文件不被删去, 文件读写位置标记移到文件末尾, 可以添加, 也可以读。
- (5)如果不能实现“打开”的任务, fopen函数将会带回一个出错信息。出错的原因可能是: 用 r 方式打开一个并不存在的文件; 磁盘出故障; 磁盘已满无法建立新文件等。此时fopen函数将带回一个空指针值 NULL (在 stdio.h 头文件中, NULL 已被定义为0)。

常用下面的方法打开一个文件:


```

1  if ((fp= fopen("file1", "r"))== NULL){
2      printf("cannot open this file\n");
3      exit(0);
4  }

```

即先检查打开文件的操作有否出错，如果有错就在终端上输出cannot open this file。

exit函数的作用是关闭所有文件，终止正在执行的程序，待用户检查出错误，修改后重新运行。

(7)在表10.1中，有12种文件使用方式，其中有6种是在第一个字母后面加了字母b的(如rb,wb,ab,rb+,wb+,ab+)，b表示二进制方式。其实，带b和不带b只有一个区别，即对换行的处理。由于在C语言用一个'\n'即可实现换行，而在Windows系统中为实现换行必须要用“回车”和“换行”两个字符，即'\r'和'\n'。因此,如果使用的是文本文件并且用w方式打开，在向文件输出时，遇到换行符'\n'时，系统就把它转换为'\r'和'\n'两个字符，否则在Windows系统中查看文件时，各行连成一片，无法阅读。同样，如果有文本文件且用r方式打开，从文件读入时，遇到'\r'和'\n'两个连续的字符，就把它们转换为'\n'一个字符。如果使用的是二进制文件，在向文件读写时，不需要这种转换。加b表示使用的是二进制文件，系统就不进行转换。

(8)如果用wb的文件使用方式，并不意味着在文件输出时把内存中按ASCII形式保存的数据自动转换成二进制形式存储。输出的数据形式是由程序中采用什么读写语句决定的。例如，用fscanf和fprintf函数是按ASCII方式进行输入输出，而fread和fwrite函数是按二进制进行输入输出。各种对文件的输入输出语句，详见下一节（3.顺序读写数据文件）。

在打开一个输出文件时，是选w还是wb方式，完全根据需要，如果需要对回车符进行转换的，就用w，如果不需要转换的，就用wb。带b只是通知编译系统：不必进行回车符的转换。如果是文本文件（例如一篇文章），显然需要转换，应该用w方式。如果是用二进制形式保存的一批数据，并不准备供人阅读，只是为了保存数据，就不必进行上述转换。可以用wb方式。一般情况下，带b的用于二进制文件，常称为二进制方式，不带b的用于文本文件，常称为文本方式，从理论上说，文本文件也可以wb方式打开，但无必要。

(9)程序中可以使用3个标准的流文件——标准输入流、标准输出流和标准出错输出流。系统已对这3个文件指定了与终端的对应关系。标准输入流是从终端的输入，标准输出流是向终端的输出，标准出错输出流是当程序出错时将出错信息发送到终端。

程序开始运行时系统自动打开这3个标准流文件。因此，程序编写者不需要在程序中用fopen函数打开它们。所以以前我们用到的从终端输入或输出到终端都不需要打开终端文件。系统定义了3个文件指针变量stdin, stdout和stderr,分别指向标准输入流、标准输出流和标准出错输出流,可以通过这3个指针变量对以上3种流进行操作，它们都以终端作为输入输出对象。例如程序中指定要从stdin所指的文件输入数据，就是指从终端键盘输入数据。

用fclose函数关闭数据文件

在使用完一个文件后应该关闭它，以防止它再被误用。“关闭”就是撤销文件信息区和文件缓冲区，使文件指针变量不再指向该文件，也就是文件指针变量与文件“脱钩”，此后不能再通过该指针对原来与其相联系的文件进行读写操作，除非再次打开，使该指针变量重新指向该文件。

关闭文件用fclose函数。fclose函数调用的一般形式为：

```

1  fclose(文件指针);

```

如果不关闭文件就结束程序运行将会丢失数据。因为，在向文件写数据时，是先将数据输出到缓冲区，待缓冲区充满后才正式输出给文件。如果当数据未充满缓冲区时程序结束运行，就有可能使缓冲区中的数据丢失。用fclose函数关闭文件时，先把缓冲区中的数据输出到磁盘文件，然后才撤销文件信息区。有的编译系统在程序结束前会自动先将缓冲区中的数据写到文件，从而避免了这个问题，但还是应当养

成在程序终止之前关闭所有文件的习惯。

fclose函数也带回一个值，当成功地执行了关闭操作，则返回值为0；否则返回EOF(-1)。

3. 顺序读写数据文件

在顺序写时，先写入的数据存放在文件的中前面的位置，后写入的数据存放在文件中后面的数据。

在顺序读时，先读文件中最前面的数据，后读文件中后面的数据。

顺序读写需要用函数库来实现。使用前需要导入 `#include<stdio.h>`

怎么向文件读写字符

读写一个字符的函数

函数名	调用形式	功能	返回值
<code>fgetc</code>	<code>fgetc(fp)</code>	从 <code>fp</code> 指向的文件中读入一个字符	读成功，带回所读的字符，失败则返回文件结束标志EOF（即-1）
<code>fputc</code>	<code>fputc(ch, fp)</code>	把字符 <code>ch</code> 写文件指针变量 <code>fp</code> 所指向的文件中	输入成功，返回值就是输出的字符；输出失败，则返回EOF（即-1）

说明：fgetc 的第1个字母f代表文件（file），中间的get表示“获取”，最后一个字母c表示字符（character），fgetc的含义很清楚：从文件读取一个字符。fputc也类似。

此节未完，将于2022年6月后继续更新.....