

N 诺考研系列

C 语言

考研复习攻略

N 诺

noobdream.com

2023 版

N 诺课程教研团队

2022. 04. 20 更新

写在前面的话

相信各位同学都知道，N 诺是一个大佬云集的平台。本书由 N 诺课程教研团队编写，针对计算机考研中的 **C 语言课程** 精心编写而成的书籍。

在阅读本书之前，首先要先说一下 N 诺对于 C 语言复习的**几点建议**：

1、不要过度依赖辅导书，一定要多练习，每章小结提供了大量的习题供同学们练习。可以在 N 诺的 **DreamJudge** 进行编程提交代码检验自己的学习成果，也可以在 N 诺的 **C 语言专业题库** 中进行大量的笔试练习，见识各种题型和考法。

2、本书的主要目的是帮助同学们在考研 C 语言课程中取得高分的成绩，也可作为数据结构课程中的前置阅读课程，帮助同学们打下扎实的编程语言基础。

3、学习 C 语言这门课程可以分为**两个流派**：笔试型和实战型。

笔试型：顾名思义，就是能在笔试考试中取得高分的成绩，但是有可能上机敲代码基本不会，因为他可能对各个知识点都了解，但是却没有上机实战过，即理论高手。

实战型：就是上机敲代码很厉害，你给他一个题他能很快把代码敲出来，但是笔试考试成绩却一般，因为笔试会考查很多平时写代码不常用到的知识。

由于计算机考研初试考查的是同学们对 C 语言各个知识点的掌握程度而非实际写代码的能力，所以本书针对笔试型进行讲解。即同学们不用担心自己不会写代码，不会写代码一样可以在考研初试中取得高分。

当然，如果你希望自己**实战水平**也变得很强，那么建议你去学习 N 诺的 **C 语言快速入门**。这门课程经过无数 C 语言萌新们验证过，可以让你在 3-5 天内入门。

传送门：<http://www.noobdream.com/Major/majorinfo/1/>

关于 N 诺

N 诺是全国最大的计算机考研在线学习平台。N 诺致力于为同学们提供一个良好的网络学习环境，一个与大佬们随时随地交流的舒适空间。如果你不知道 N 诺，那么你已经输在起跑线上了，因为 N 诺是 - 计算机学习考研必备神器。

N 诺收集并整理了**计算机考研的前世今生**，帮助你了解考研的点点滴滴。

<http://www.noobdream.com/post/585/>

在 N 诺，你可以查询各个院校的**考研信息**，包括分数线、录取人数、考试大纲、导师信息、经验交流等信息，应有尽有。

<http://www.noobdream.com/schoollist/>

在 N 诺，你可以尽情的刷各个科目的**题库**，还可以将题目加入**错题本**方便以后复习，也可以写下自己的**学习笔记**，记录考研过程中跌宕起伏。

<http://www.noobdream.com/Practice/index/>

在 N 诺，你可以在**讨论区**里发表你的问题或感想，与全国几百万考研 er 分享你的喜怒哀乐。

<http://www.noobdream.com/forum/0/>

在 N 诺，你可以将你不用**的书籍或资料**放到**二手交易市场**，既能帮助他人，还能为自己省下一笔当初买书买资料的费用。

<http://www.noobdream.com/Task/tasklist/>


如何使用本书?

N 诺 C 语言考研交流群 (976044819)



本书配套在线练习题库

传送门: <http://www.noobdream.com/Practice/clang/>



admin
本科学校: N诺大学
本科校友数量: 7

目标学校: 北京大学
目标学校用户数: 13

+ 学习笔记

公开笔记

我的主页

错题本

我的笔记

C语言

数据结构

操作系统

计算机组成原理

计算机网络

数据库

政治

题目搜索

输入题目编号或名称

查询

✓	题目编号: 1000	是构成C语言程序的基本单位	C语言	学习人数: 545
✓	题目编号: 1001	C语言程序从__开始执行	C语言	学习人数: 253
✓	题目编号: 1002	以下说法中正确的是	C语言	学习人数: 200
✓	题目编号: 1003	下列关于C语言的说法错误的是	C语言	学习人数: 172
✓	题目编号: 1004	下列正确的标识符是	C语言	学习人数: 188
✓	题目编号: 1005	下列C语言用户标识符中合法的是	C语言	学习人数: 206
✓	题目编号: 1006	下列四组选项中, 正确的C语言标识符	C语言	学习人数: 132
✓	题目编号: 1007	下列四组字符串中都可以用作C语言程	C语言	学习人数: 113
✓	题目编号: 1008	C语言中的简单数据类型包括	C语言	学习人数: 103
✓	题目编号: 1009	在C语言程序中, 表达式5%2的结果	C语言	学习人数: 113
✓	题目编号: 1010	如果int a=3,b=4; 则条件	C语言	学习人数: 131

温馨提醒: 微信小程序搜索 N 诺, 可以在小程序里刷题啦, 随时随地, 想刷就刷~

目录

写在前面的话	2
关于 N 诺	3
如何使用本书?	4
目录	5
第一章 程序设计基本知识	8
1.1 程序设计和 C 语言	9
1.2 算法 — 程序的灵魂	13
1.3 数据的类型	18
1.4 运算符和表达式	25
1.5 C 语句	32
1.6 数据的输入输出	36
1.7 程序的设计结构	43
1.8 本章小结	44
第二章 选择结构程序设计	45
2.1 选择结构和条件判断	46
2.2 用 if 语句实现选择结构	47
2.3 关系运算符和关系表达式	52
2.4 逻辑运算符和逻辑表达式	54
2.5 条件运算符和条件表达式	57
2.6 选择结构的嵌套	59
2.7 用 switch 语句实现多分支选择结构	61
2.8 本章小结	63
第三章 循环结构程序设计	64
3.1 goto 语句构成循环	65
3.2 用 while 语句实现循环	66
3.3 用 do...while 语句实现循环	69
3.4 用 for 语句实现循环	72
3.5 循环的嵌套	76
3.6 几种循环的比较	77
3.7 改变循环执行的状态	78
3.8 本章小结	81
第四章 数组	82
4.1 一维数组	83

4.2 二维数组	90
4.3 字符数组	94
4.4 本章小结	106
第五章 函数	107
5.1 为什么要用函数	108
5.2 如何定义函数	110
5.3 调用函数	112
5.4 对被调用函数的声明和函数原型	116
5.5 函数的嵌套调用	118
5.6 函数的递归调用	120
5.7 数组作为函数参数	124
5.8 局部变量和全局变量	130
*5.9 变量的存储方式和生存期	134
5.10 关于变量的声明和定义	137
*5.11 内部函数和外部函数	139
5.12 本章小结	141
第六章 指针	142
6.1 指针是什么	143
6.2 指针变量	145
6.3 指针变量的运用	154
6.4 通过指针引用数组	158
6.5 通过指针引用字符串	174
*6.6 指向函数的指针	180
*6.7 返回指针值的函数	182
*6.8 指针数组和多重指针	184
*6.9 动态内存分配与指向它的指针变量	192
6.10 本章小结	195
第七章 结构体、共用体和枚举类型	196
7.1 定义和使用结构体变量	197
7.2 使用结构体数组	202
7.3 结构体指针	205
*7.4 用指针处理链表	210
*7.5 共用体类型	214
7.6 使用枚举类型	217
*7.7 用 typedef 声明新类型名	220
7.8 本章小结	221
第八章 文件	222
8.1 C 文件的有关基本知识	223

8.2 打开与关闭文件	225
8.3 顺序读写数据文件	228
8.4 随机读写数据文件	238
8.5 文件读写的出错检测	240
8.6 本章小结	241
第九章 预处理和位运算	242
9.1 宏定义	243
9.2 文件包含	247
9.3 条件编译	248
9.4 位运算	251
9.5 位域	255
9.6 本章小结	258
完结撒花	259
N 诺考研系列图书	261
N 诺 Offer 训练营	262

N 诺
noobdream.com

第一章 程序设计基本知识

【本章知识点汇总】

C 语言特点

运行 C 的步骤

算法的特性

数据类型

32 个关键字

9 种控制语句

34 种运算符

输入与输出

3 种程序结构

本书配套视频精讲: <https://www.bilibili.com/video/BV1HJ41137fe>

1.1 程序设计和 C 语言

什么是计算机程序？

计算机程序（Computer program），也称为软件（software），简称程序（英语：Program），是指一组指示计算机或其他具有信息处理能力装置执行动作或做出判断的指令，通常用某种程序设计语言编写，运行于某种目标计算机体系结构上。

计算机程序是计算任务的处理对象和处理规则的描述。任何以计算机为处理工具的任务都是计算任务。处理对象是数据或信息，处理规则反映处理动作和步骤。

计算机程序通常是用高级语言编写源程序，程序包含数据结构，算法，存储方式 编译等，经过语言翻译程序（解释程序和编译程序）转换成机器接受的指令。程序可按其设计目的的不同，分为两类：一类是系统程序，它是为了方便和充分发挥计算机系统效能而设计的程序，通常由计算机制造厂商或专业软件公司设计，如操作系统、编译程序等；另一类是应用程序，它是为解决用户特定问题而设计的程序，通常由专业软件公司或用户自己设计，如账务处理程序、文字处理程序等。

什么是计算机语言？

计算机语言（Computer Language）指用于人与计算机之间通讯的语言。计算机语言是人与计算机之间传递信息的媒介。计算机系统最大特征是指令通过一种语言传达给机器。为了使电子计算机进行各种工作，就需要有一套用以编写计算机程序的数字、字符和语法规划，由这些字符和语法规则组成计算机各种指令（或各种语句）。这些就是计算机能接受的语言。

C 语言的发展

1972 年，美国贝尔实验室的 D.M.Ritchie 在 B 语言的基础上设计出了 C 语言。

1973 年，Ken Thompson 和 D.M.Ritchie 合作把 UNIX 的 90%以上用 C 语言改写，即 UNIX 第 5 版。

1978 年以后，C 语言先后移植到大、中、小和微型计算机上。

1978 年, Brian W.Kernighan 和 Dennis M.Ritchie 合著了影响深远的名著 The C Programming Language, 这本书中介绍的 C 语言成为后来广泛使用的 C 语言版本的基础, 它是实际上第一个 C 语言标准。

1983 年, 美国国家标准协会(ANSI), 根据 C 语言问世以来各种版本对 C 语言的发展和扩充, 制定了第一个 C 语言标准草案('83 ANSI C)。

1989 年, ANSI 公布了一个完整的 C 语言标准——ANSI X3. 159——1989(常称为 ANSI C 或 C 89)。

1990 年, 国际标准化组织 ISO(International Standard Organization)接受 C 89 作为国际标准 ISO/IEC 9899:1990, 它和 ANSI 的 C 89 基本上是相同的。

1999 年, ISO 又对 C 语言标准进行了修订, 在基本保留原来的 C 语言特征的基础上, 针对应用的需要, 增加了一些功能, 尤其是 C++中的一些功能, 并在 2001 年和 2004 年先后进行了两次技术修正, 它被称为 C 99, C99 是 C 89 的扩充。

C 语言特点

- 1、易于学习。
- 2、具有结构化的控制语句, 是完全模块化和结构化的语言。
- 3、语言简洁、紧凑, 使用方便、灵活。 32 个关键字、9 种控制语句, 程序形式自由, 目标代码质量高, 程序执行效率高。只比汇编程序生成的目标代码效率低 10%-20%。
- 4、可以处理底层的活动。
- 5、可在多种计算机平台上进行编译, 程序可移植性好(与汇编语言比)。
- 6、运算符丰富, 有 34 种运算符。
- 7、语法限制不太严格, 程序设计自由度大。
- 8、允许直接访问物理地址, 能进行位操作。

最简单的 C 语言程序

```
1. #include <stdio.h>
2.
3. int main() {
4.     printf("你好! N 诺!");
5.     return 0;
6. }
```

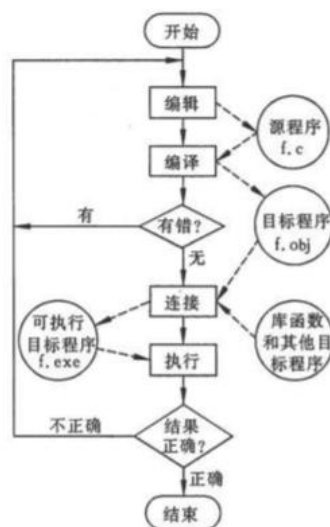
添加注释之后的程序

```
1. /*
2.  我的第一个 C 程序
3. */
4. #include <stdio.h> //这一行是头文件
5.
6. int main() { //这一行是主函数
7.     printf("你好! N 诺!"); //这一行是输出的内容
8.     return 0; //返回函数的值
9. }
```

其中, /* 这里填充注释内容 */, 这样可以填充多行注释内容

// 这里填充注释内容, 这样可以填充单行注释内容

运行 C 程序的步骤和方法



第一步 编辑：源程序 f.c

第二步 编译：先用 C 编译系统提供的“预处理器”（“预处理程序”或“预编辑器”）对程序中的预处理指令进行编译处理。由预处理得到的信息与程序其他部分一起，组成一个完整的、可以用来进行正式编译的源程序，然后由编译系统对源程序进行编译

例如，对于#include<stdio.h>将 stdio.h 头文件的内容读进来，取代#include<stdio.h>

第三步 连接：经过编译所得到的二进制目标文件（.obj）还不能供计算机直接执行。一个程序可以有好几个.c 文件，而编译时以单个.c 文件为对象的，一次编译只能得到与一个.c 文件相对应的目标文件（目标模块），它只是整个程序的一部分。必须把所有的编译后得到的目标模块连接装配起来，再与函数库相连成整体，生成一个可供计算机执行的目标程序，称为可执行程序（executive program）

第四步 运行可执行程序(.exe)，得到运行结果

解释

编辑：写 c 文件，可能多个

编译：将预处理指令进行处理，编译成一个.obj，一个.c 文件编译成一个.obj

连接：可能多个.c 文件，对应就有多个.obj 文件，将.obj 文件进行连接，得到.exe

运行：运行.exe

关于 include

include 进来的只是头文件.h，一般把具体的实现放到.c 文件中

例如：stdio.h 存放申明，stdio.c 存放具体实现

所以包含#include<stdio.h>的 test.c 文件编译的过程：

先在 test.c 中将#include<stdio.h>用 stdio.h 进行代替

编译 test.c 得到 test.obj

test.obj 与 stdio.obj（由 stdio.c 编译得到）进行连接，组成 test.exe

1.2 算法 — 程序的灵魂

一个程序应包括：

对数据的描述。在程序中要指定数据的类型和数据的组织形式，即数据结构。

对数据的描述。即操作步骤，也就是算法。

所以

程序 = 算法 + 数据结构

什么是算法

算法（Algorithm）是指解题方案的准确而完整的描述，是一系列解决问题的清晰指令，算法代表着用系统的方法描述解决问题的策略机制。也就是说，能够对一定规范的输入，在有限时间内获得所要求的输出。如果一个算法有缺陷，或不适合于某个问题，执行这个算法将不会解决这个问题。不同的算法可能用不同的时间、空间或效率来完成同样的任务。一个算法的优劣可以用空间复杂度与时间复杂度来衡量。

算法中的指令描述的是一个计算，当其运行时能从一个初始状态和（可能为空的）初始输入开始，经过一系列有限而清晰定义的状态，最终产生输出并停止于一个终态。一个状态到另一个状态的转移不一定是确定的。随机化算法在内的一些算法，包含了一些随机输入。

简单的算法举例

例 1：求 $1 + 2 + 3 + 4 + 5$ 的和

代码实现

```
1. #include <stdio.h>
2.
3. int main() {
4.     int sum = 0;
5.     for (int i = 1; i <= 5; i++) {
6.         //部分老版本编译器不支持 for 里面定义
7.         sum = sum + i;
8.     }
9.     printf("1 + 2 + 3 + 4 + 5 = %d\n", sum);
```

```
10.     return 0;
11. }
```

例 2: 求 5! 的值

代码实现

```
1.  #include <stdio.h>
2.
3.  int main() {
4.      int fac = 1;
5.      for (int i = 2; i <= 5; i++) {
6.          fac = fac * i;
7.      }
8.      printf("1 * 2 * 3 * 4 * 5 = %d\n", fac);
9.      return 0;
10. }
```

算法的特性

一个算法应该具有以下五个重要的特征:

有穷性

算法的有穷性是指算法必须能在执行有限个步骤之后终止;

确切性

算法的每一步骤必须有确切的定义,而不能是含糊的、模棱两可的;

输入项

一个算法有 0 个或多个输入,以刻画运算对象的初始情况,所谓 0 个输入是指算法本身定出了初始条件;

输出项

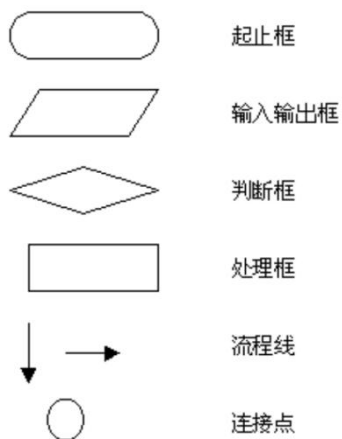
一个算法有一个或多个输出,以反映对输入数据加工后的结果。没有输出的算法是毫无意义的;

可行性

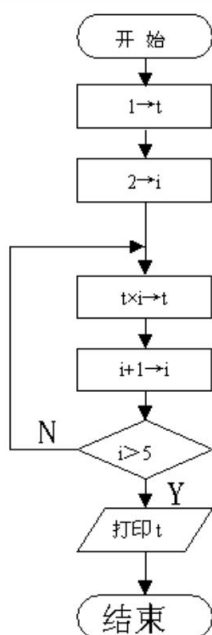
算法中执行的任何计算步骤都是可以被分解为基本的可执行的操作步骤,即每个计算步骤都可以在有限时间内完成(也称之为有效性)。

流程图

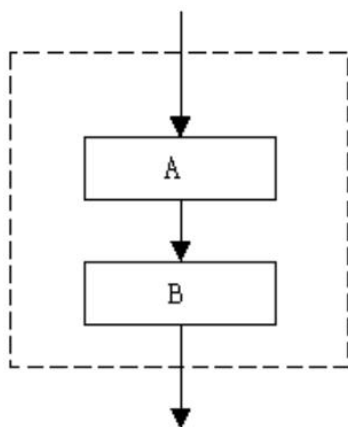
一般我们会使用流程图去表示一个算法。



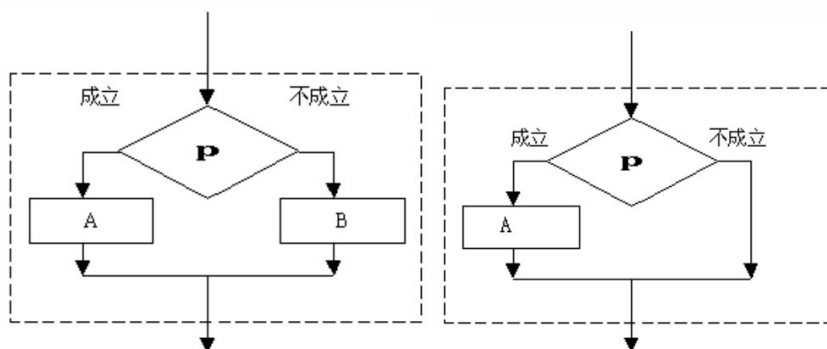
下面我们给出求 $5!$ 的流程图



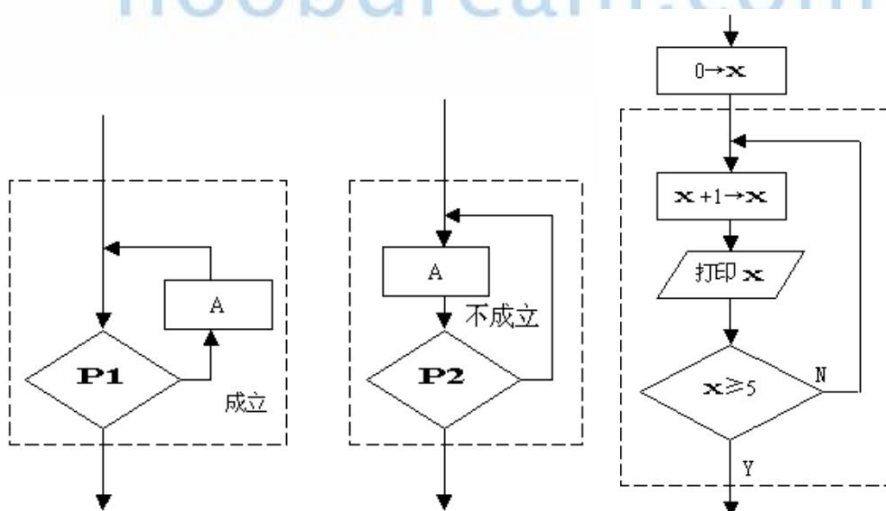
顺序结构流程图



选择结构流程图



循环结构流程图



三种基本结构的共同特点:

- 1、只有一个入口
- 2、只有一个出口
- 3、结构内的每一部分都有机会被执行到
- 4、结构内不存在“死循环”。

伪代码

伪代码使用介于自然语言和计算机语言之间的文字和符号来描述算法。

这里我们建议大家自行了解伪代码，最重要的是学会用 C 语言来实现算法。



1.3 数据的类型

我们已经看到程序中使用的各种变量都应预先加以定义，即先定义，后使用。

对变量的定义可以包括三个方面：

- 1、数据类型
- 2、存储类型
- 3、作用域

所谓数据类型是按被定义变量的性质，表示形式，占据存储空间的多少，构造特点来划分的。在 C 语言中，数据类型可分为：基本数据类型，构造数据类型，指针类型，空类型四大类。

这一节我们介绍基本数据类型中的整型、浮点型和字符型。其余类型在以后各章节中陆续介绍。

常量和变量

对于基本数据类型量，按其取值是否可改变又分为常量和变量两种。在程序执行过程中，其值不发生改变的量称为常量，其值可变的量称为变量。他们可与数据类型结合起来分类。例如，可分为整型常量、整型变量、浮点常量、浮点变量、字符常量、字符变量、枚举常量、枚举变量。在程序中，常量是可以不经过说明而直接引用的，而变量则必须先定义后使用。

在程序执行过程中，其值不发生改变的量称为常量。

- 1、直接常量（字面常量）

整型常量：12、0、-3

实型常量：4.6、-1.23

字符常量：‘a’、‘b’

- 2、标识符：用来标识变量名、符号常量名、函数名、数组名、类型名、文件名的有效字符序列。

- 3、符号常量：用标识符代表一个常量。在 C 语言中，可以用一个标识符来表示一个常量，称之为符号常量。

符号常量在使用之前必须先定义，其一般形式为：

#define 标识符 常量

其中#define 也是一条预处理命令（预处理命令都以'#'开头），成为宏定义命令（在后面预处理程序中将进一步介绍），其功能是把该标识符定义为其后的常量值。一经定义，以后在程序中所有出现该标识符的地方均代之以该常量值。

4、习惯上符号常量的标识符用大写字符，标量标识符用小写字符，以示区别。

例：符号常量的使用

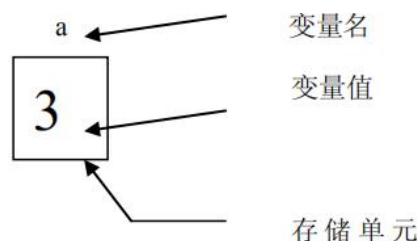
```
1. #include <stdio.h>
2. #define PRICE 30
3.
4. int main() {
5.     int num, total;
6.     num = 10;
7.     total = num * PRICE;
8.     printf("total = %d\n", total);
9.     return 0;
10. }
```

使用符号变量的好处是：

- 1、含义清楚
- 2、能做到“一改全改”

变量

其值可以改变的量称为变量。一个变量应该有一个名字，在内存中占据一定的存储单元。变量定义必须放在变量使用之前。一般放在函数体的开头部分。要区分变量名和变量值是两个不同的概念。



整型数据

整型常量

整型常量就是整常数。在 C 语言中，使用的整常数有八进制、十六进制和十进制三种。

1、十进制整常数：十进制整常数没有前缀。其数码为 0~9。

如：237、-568/65535/1627

提示：在程序中是根据前缀来区分各种进制数的。因此在书写常数时不要把前缀弄错造成结果不正确。

2、八进制整常数：八进制整常数必须以 0 开头，即以 0 作为八进制数的前缀。数码取值为 0~7。八进制数通常是无符号数。

如：015（十进制为 13）、0101（十进制为 65）

3、十六进制整常数：十六进制整常数的前缀为 0X 或 0x。其数码取值为 0~9，A~F 或 a~f。

如：0X2A（十进制为 42）、0XA0（十进制为 160）

4、整型常数的后缀：

十进制长整常数：158L（十进制为 158）、358000L（十进制为 358000）

八进制长整常数：012L（十进制为 10）、077L（十进制为 63）

十六进制长整常数：0X15L（十进制为 21）、0XA5L（十进制为 165）

长整型 158L 和基本整常数 158 在数值上并无区别。但对 158L，因为是长整型量，C 编译系统将为其分配 4 个字节存储空间。而对 158，因为是基本整型，只分配 2 个字节的存储空间。

无符号数也可用后缀表示，整型常数的无符号数的后缀为“U”或“u”。

如：358u，0x38Au，235Lu 均为无符号数。

整型变量

整型数据在内存中的存放形式

如：10

0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

数值是以补码表示的：

1、正数的补码和原码相同

2、负数的补码：将该数的绝对值的二进制形式按位取反再加 1

例如：求-10 的补码

10 的原码：

0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

取反：

1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

再加 1，得-10 的补码

1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

整型变量的分类

- 1、基本型：int，在内存中占 2 个字节
- 2、短整型：short int 或 short，同基本型
- 3、长整型：long int 或 long，在内存中占 4 个字节
- 4、无符号型：unsigned，所占内存空间字节数与相应的符号类型量相同。

由于省去了符号位，故不能表示负数，但正数的范围翻倍。

```
1. #include <stdio.h>
2.
3. int main() {
4.     int a, b;
5.     short c;
6.     long int d;
7.     unsigned int e;
8.     return 0;
9. }
```

浮点型数据

浮点型也称为实型。实型常量也称为实数或者浮点数。在 C 语言中，实数只采用十进制。它

有两种形式：十进制小数形式，指数形式。

1、十进制数形式

0.0、25.0、5.789、-267.8

注意：必须有小数点。

2、指数形式：由十进制数，加阶码标志“e”或“E”以及阶码（只能为整数，可以带符号）组成。

一般形式为： $a E n$ （ a 为十进制数， n 为十进制整数）

如：2.1E5（等于 2.1×10^5 ）、3.7E-2（等于 3.7×10^{-2} ）

实型变量

实型数据一般占 4 个字节（32 位）内存空间。

分类：

单精度型：float 4 个字节

双精度型：double 8 个字节

长双精度型：long double 16 个字节

区别在于范围和精度误差。

实型常数不分单、双精度，都按双精度 double 型处理。

字符型数据

字符常量

字符常量是用单引号括起来的一个字符

例如：

'a'、'b'、'='、'+'、'？'

字符常量有以下特点：

- 1、字符常量只能用单引号括起来，不能用双引号或其他括号。
- 2、字符常量只能是单个字符，不能是字符串。
- 3、字符可以是字符集中任意字符。但数字被定义为字符型之后就不能参与数值运算。

转义字符

转义字符是一种特殊的字符常量。转义字符以反斜线“\”开头，后跟一个或几个字符。

转义字符具有特定的含义，不同于字符原有的意义，故称“转义”字符。

常用的转义字符及其含义

转义字符	转义字符的意义	ASCII 代码
\n	回车换行	10
\t	横向跳到下一制表位置	9
\b	退格	8
\r	回车	13
\f	走纸换页	12

\\	反斜线符“\”	92
\'	单引号符	39
\”	双引号符	34
\a	鸣铃	7
\ddd	1~3 位八进制数所代表的字符	
\xhh	1~2 位十六进制数所代表的字符	

字符变量

字符变量用来存储字符常量，即单个字符。

字符变量的类型说明符是 `char`。字符变量类型定义的格式和书写规则都与整型变量相同。

例如：`char a, b;`

```
1. #include <stdio.h>
2.
3. int main() {
4.     char ch;
5.     ch = 'n';
6.     printf("%c\n", ch);
7.     return 0;
8. }
```

字符串常量

字符串常量是由一对双引号括起的字符序列。

例如: "NoobDream", "C program", "\$9.9"

字符串常量和字符常量是不同的量。

字符常量占一个字节的内存空间。字符串常量占的内存字节数等于字符串中字节数加 1. 增加的一个字节存放字符 "\0" (ASCII 码为 0)。这是字符串结束的标志。

例如: 字符串 "C program" 在内存中所占的字节为:

C		p	r	o	g	r	a	m	\0
---	--	---	---	---	---	---	---	---	----

‘a’ 在内存中占一个字节, 可表示为:

a

“a” 在内存中占两个字节, 可表示为:

a	\0
---	----

1.4 运算符和表达式

运算符是一种告诉编译器执行特定的数学或逻辑操作的符号。C 语言内置了丰富的运算符，并提供了以下类型的运算符：

- 1、算术运算符
- 2、关系运算符
- 3、逻辑运算符
- 4、位运算符
- 5、赋值运算符
- 6、杂项运算符

算术运算符

- + 把两个操作数相加
- 从第一个操作数中减去第二个操作数
- * 把两个操作数相乘
- / 分子除以分母
- % 取模运算符，整除后的余数
- ++ 自增运算符，整数值增加 1
- 自减运算符，整数值减少 1

关系运算符

- == 检查两个操作数的值是否相等，如果相等则条件为真。
- != 检查两个操作数的值是否相等，如果不相等则条件为真。
- > 检查左操作数的值是否大于右操作数的值，如果是则条件为真。
- < 检查左操作数的值是否小于右操作数的值，如果是则条件为真。
- >= 检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。
- <= 检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。

逻辑运算符

&& 称为逻辑与运算符。如果两个操作数都非零，则条件为真。

|| 称为逻辑或运算符。如果两个操作数中有任何一个非零，则条件为真。

! 称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。

位运算符

& 按位与操作，按二进制位进行"与"运算。运算规则：

$0 \& 0 = 0$;

$0 \& 1 = 0$;

$1 \& 0 = 0$;

$1 \& 1 = 1$;

| 按位或运算符，按二进制位进行"或"运算。运算规则：

$0 | 0 = 0$;

$0 | 1 = 1$;

$1 | 0 = 1$;

$1 | 1 = 1$;

^ 异或运算符，按二进制位进行"异或"运算。运算规则：

$0 \wedge 0 = 0$;

$0 \wedge 1 = 1$;

$1 \wedge 0 = 1$;

$1 \wedge 1 = 0$;

~ 取反运算符，按二进制位进行"取反"运算。运算规则：

$\sim 1 = 0$;

$\sim 0 = 1$;

<< 二进制左移运算符。

将一个运算对象的各二进制位全部左移若干位（左边的二进制位丢弃，右边补0）。

>> 二进制右移运算符。

将一个数的各二进制位全部右移若干位，正数左补0，负数左补1，右边丢弃。

变量赋初值

简单赋值运算符记为“=”。由“=”连接的式子称为赋值表达式。

变量 = 表达式

例如：x = a + b

赋值运算符具有右结合性。

a = b = c = 5 等价于 a = (b = (c = 5))

凡是表达式可以出现的地方均可出现赋值表达式。

例如：x = (a = 5) + (b = 8)

复合赋值运算符

在赋值符“=”之前加上其它二目运算符可构成复合赋值符。

= 简单的赋值运算符，把右边操作数的值赋给左边操作数

C = A + B 将把 A + B 的值赋给 C

+= 加且赋值运算符，把右边操作数加上左边操作数的结果赋值给左边操作数

C += A 相当于 C = C + A

-= 减且赋值运算符，把左边操作数减去右边操作数的结果赋值给左边操作数

C -= A 相当于 C = C - A

*= 乘且赋值运算符，把右边操作数乘以左边操作数的结果赋值给左边操作数

C *= A 相当于 C = C * A

/= 除且赋值运算符，把左边操作数除以右边操作数的结果赋值给左边操作数

C /= A 相当于 C = C / A

%= 求模且赋值运算符，求两个操作数的模赋值给左边操作数

C %= A 相当于 C = C % A

<<= 左移且赋值运算符

$C \ll= 2$ 等同于 $C = C \ll 2$

$\gg=$ 右移且赋值运算符

$C \gg= 2$ 等同于 $C = C \gg 2$

$\&=$ 按位与且赋值运算符

$C \&= 2$ 等同于 $C = C \& 2$

$\wedge=$ 按位异或且赋值运算符

$C \wedge= 2$ 等同于 $C = C \wedge 2$

$|=$ 按位或且赋值运算符

$C |= 2$ 等同于 $C = C | 2$

不同类型数据间的混合运算

如果赋值运算符两边的数据类型不同, 系统讲自动进行类型转换, 即把赋值好右边的类型换成左边的类型。

例如:

```
1. #include <stdio.h>
2.
3. int main() {
4.     double a = 9.7;
5.     int b = a;
6.     printf("%d\n", b);
7.     return 0;
8. }
```

b 的输出为: 9

强制类型转换运算符

```
1. #include <stdio.h>
2.
3. int main() {
4.     int a = 3, b = 2;
5.     float c;
6.     c = (float)a / (float)b;
7.     printf("%f\n", c);
8.     return 0;
9. }
```

如果不强制转换的话，输出的结果就会是整形，答案将不是 1.5 而是 1。

算术表达式和运算符的优先级与结合性

运算符的优先级确定表达式中项的组合。这会影响到一个表达式如何计算。某些运算符比其他运算符有更高的优先级，例如，乘除运算符具有比加减运算符更高的优先级。

例如 $x = 7 + 3 * 2$ ，在这里， x 被赋值为 13，而不是 20，因为运算符 $*$ 具有比 $+$ 更高的优先级，所以首先计算乘法 $3*2$ ，然后再加上 7。

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

赋值	= += -= *= /= %= >>= <<= &= ^= =	从右到左
逗号	,	从左到右

杂运算符

sizeof() 返回变量的大小。 sizeof(a) 将返回 4，其中 a 是整数。

& 返回变量的地址。 &a; 将给出变量的实际地址。

* 指向一个变量。 *a; 将指向一个变量。

?: 条件表达式 如果条件为真 ? 则值为 X: 否则值为 Y

逗号运算符和逗号表达式

在 C 语言中逗号 “,” 也是一种运算符，称为逗号运算符。其功能是把两个表达式连接起来组成一个表达式，称为逗号表达式。

其一般形式为：

表达式 1, 表达式 2

其求值过程是分别求两个表达式的值，并以表达式 2 的值作为整个逗号表达式的值。

```

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

本例中，y 等于整个逗号表达式的值，也就是表达式 2 的值，x 是第一个表达式的值。对于逗号表达式还要说明两点：

1、逗号表达式可以嵌套

表达式 1, (表达式 2, 表达式 3)

2、程序中使用逗号表达式, 通常是要分别求逗号表达式内各表达式的值, 并不一定要求整个逗号表达式的值。并不是在所有出现逗号的地方都组成逗号表达式, 如在变量说明中, 函数参数表中逗号只是用作各变量之间的间隔符。



1.5 C 语句

C 语句的作用和分类

一个函数包含声明部分和执行部分, 执行部分是由语句组成的, 语句的作用是向计算机系统发出操作指令, 要求执行相应的操作。一个 C 语句经过编译后产生若干条机器指令。声明部分不是语句, 它不产生机器指令, 只是对有关数据的声明。

C 语句分为以下 5 类

(1) 控制语句

控制语句用于完成一定的控制功能。C 只有 9 种控制语句, 它们的形式是:

序号 语句描述

1 if()...else... 条件语句

2 for()... 循环语句

3 while()... 循环语句

4 do...while() 循环语句

5 continue 结束本次循环语句

6 break 中止执行 switch 或循环语句

7 switch 多分支选择语句

8 return 从函数返回语句

9 goto 转向语句, 在结构化程序中基本不用 goto 语句

(2) 函数调用语句

函数调用语句由一个函数调用加一个分号构成, 例如:

```
printf("This is a c program");
```

其中 printf("This is a c program") 是一个函数调用, 加一个分号成为一个语句。

(3) 表达式语句

表达式语句由一个表达式加一个分号构成，最典型的是，由赋值表达式构成一个赋值语句。

例如：

`a=3` 是一个赋值表达式，而

`a=3;` 是一个赋值语句。

可以看到，一个表达式的最后加一个分号就成了一个语句。一个语句必须在最后有一个分号，分号是语句中不可缺少的组成部分，而不是两个语句间的分割符号。

例如：

`i=i+1`（是表达式，不是语句）

`i=i+1;`（是语句）

任何表达式都可以加上分号而成为语句，例如：

`i++;` 是一个语句，作用是使 `i` 值加 1。

表达式能构成语句是 C 语言的一个重要特色。其实“函数”调用语句也是属于表达式语句，因为函数调用（如 `sin(x)`）也属于表达式的一种。只是为了便于理解和使用，才把“函数调用语句”和“表达式语句”分开来说明。

(4) 空语句

下面就是一个空语句

;

此语句只有一个分号，它什么也不做。那么它的作用是什么呢？可以用来作为流程的转向点（流程从程序其他地方转到此语句处），也可用来作为循环语句中的循环体（循环体是空语句，表示循环体什么也不做）。

(5) 复合语句

可以用 `{}` 把一些语句和声明括起来成为复合语句（又称语句块）。

例如，下面是一个复合语句：

1. {

```
2.    float pi=3.14159,r=2.5,area;  
3.    area=pi*r*r;  
4.    printf("area=%f",area);  
5. }
```

可以在复合语句中包含声明部分，C99 允许将声明部分放在复合语句中的任何位置，但习惯上把它放在语句块开头位置。复合语句常用在 if 语句或循环体中，此时程序需要连续执行一组语句。

在复合语句中最后一个语句中最后的分号不能忽略不写。

赋值语句

赋值语句是由赋值表达式再加上分号构成的表达式语句。

其一般形式为： 变量 = 表达式;

赋值语句的功能和特点都与赋值表达式相同。它是程序中使用最多的语句之一。

在赋值语句的使用中需要注意以下几点：

1、由于在赋值符“=”右边的表达式也可以又是一个赋值表达式，因此，下述变形
变量 = (变量 = 表达式);

是成立的，从而形成嵌套的情形。

其展开之后的一般形式为：

变量 = 变量 = ... = 表达式;

例如：

a = b = c = d = e = 5;

按照赋值运算符的右结合性，因此实际上等效于：

e = 5; d = e; c = d; b = c; a = b;

2、注意在变量说明中给变量赋初值和赋值语句的区别。

给变量赋初值是变量说明的一部分，赋初值后的变量与其后的其他同类变量之间仍必须用逗号间隔，而赋值语句则必须用分号结尾。

例如：int a = 5, b, c;

3、在变量说明中，不允许连续给多个变量赋初值。

错误的写法: `int a = b = c = 5;`

正确的写法: `int a = 5, b = 5, c = 5;`

4、注意赋值表达式和赋值语句的区别。

赋值表达式是一种表达式, 它可以出现在任何允许表达式出现的地方, 而赋值语句则不能。

错误的写法: `if ((x = y + 5;) > 0) z = x;`

正确的写法: `if ((x = y + 5) > 0) z = x;`



1.6 数据的输入输出

所谓输入输出是以计算机为主体而言的。

我们介绍的数据输出是向标准输出设备显示器输出数据的语句。

在 C 语言中, 所有的数据输入/输出都是由库函数完成的, 因此都是函数语句。

在使用 C 语言库函数时, 要用预编译命令

```
#include
```

将有关“头文件”包括到源文件中。

使用标准输入输出库函数时要用到“stdio.h”文件, 因为源文件开头应有以下预编译命令:

```
#include<stdio.h> 或 #include "stdio.h"
```

stdio 是 standard input & output 的意思。

字符输入输出函数

字符输出 putchar

putchar 函数是字符输出函数, 其功能是在显示器上输出单个字符。

其一般形式为:

```
putchar(字符变量)
```

例如:

```
putchar('A') //输出大写字母 A
```

```
putchar(x) //输出字符变量 x 的值
```

```
putchar('\n')//换行
```

使用本函数前必须要用文件包含命令:

```
#include<stdio.h> 或 #include "stdio.h"
```

输出单个字符例子

```
1. #include <stdio.h>
2.
3. int main() {
4.     char a = 'N', b = 'o', c = 'b';
5.     putchar(a);putchar(b);putchar(b);putchar(c);putchar('\t');
```



```
6.    putchar(a);putchar(b);
7.    putchar('\n');
8.    putchar(b);putchar(c);
9.    return 0;
10. }
```

字符输入 getchar

getchar 函数的功能是从键盘上输入一个字符。

其一般形式为: getchar();

通常把输入的字符赋予一个字符变量, 构成赋值语句, 如:

char c;

c = getchar();

例:

```
1.  #include <stdio.h>
2.
3.  int main() {
4.      char c;
5.      printf("input a character\n");
6.      c = getchar();
7.      putchar(c);
8.      return 0;
9.  }
```

使用 getchar 函数还应注意几个问题:

- 1、getchar 函数只能接受单个字符, 输入数字也按字符处理。输入多于一个字符时, 只接收第一个字符。
- 2、使用本函数前必须包含头文件"stdio.h"。

格式输入输出函数

输入函数 scanf

scanf 函数称为格式输入函数, 即按用户指定的格式从键盘上把数据输入到指定的变量之中。

1. scanf 函数的一般形式

scanf 函数是一个标准库函数，它的函数原型在头文件“stdio.h”中，与 printf 函数相同，C 语言也允

许在使用 scanf 函数之前不必包含 stdio.h 文件。

scanf 函数的一般形式为：

scanf(“格式控制字符串”，地址表列);

其中，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：

&a, &b

分别表示变量 a 和变量 b 的地址。

这个地址就是编译系统在内存中给 a,b 变量分配的地址。在 C 语言中，使用了地址这个概念，这是与其

它语言不同的。应该把变量的值和变量的地址这两个不同的概念区别开来。变量的地址是 C 编译系统分配的，

用户不必关心具体的地址是多少。

变量的地址和变量值的关系如下：

在赋值表达式中给变量赋值，如：

a = 567

则，a 为变量名，567 是变量的值，&a 是变量 a 的地址。

但在赋值号左边是变量名，不能写地址，而 scanf 函数在本质上也是给变量赋值，但要求写变量的地址，如&a。这两者在形式上是不同的。&是一个取地址运算符，&a 是一个表达式，其功能是求变量的地址。

【例】

```
1. #include <stdio.h>
2.
3. int main(){
4.     int a,b,c;
5.     printf("input a,b,c\n");
```

```
6.    scanf("%d%d%d",&a,&b,&c);
7.    printf("a=%d,b=%d,c=%d",a,b,c);
8.    return 0;
9. }
```

在本例中, 由于 `scanf` 函数本身不能显示提示串, 故先用 `printf` 语句在屏幕上输出提示, 请用户输入 `a`、`b`、`c` 的值。执行 `scanf` 语句, 则退出屏幕进入用户屏幕等待用户输入。用户输入 `7 8 9` 后按下回车键,

此时, 系统又将返回屏幕。在 `scanf` 语句的格式串中由于没有非格式字符在“`%d%d%d`”之间作输入时的间隔, 因此在输入时要用一个以上的空格或回车键作为每两个输入数之间的间隔。如:

7 8 9

或 7

8

9

2. 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:

1) 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

2) “*”符:用以表示该输入项,读入后不赋予相应的变量,即跳过该输入值。

如:

```
scanf("%d %*d %d",&a,&b);
```

当输入为: 1 2 3 时, 把 1 赋予 a, 2 被跳过, 3 赋予 b。

3) 宽度:用十进制整数指定输入的宽度(即字符数)。

例如:

```
scanf("%5d",&a);
```

输入: 12345678

只把 12345 赋予变量 a, 其余部分被截去。

又如:

```
scanf("%4d%4d",&a,&b);
```

输入: 12345678

将把 1234 赋予 a, 而把 5678 赋予 b。

4) 长度:长度格式符为 l 和 h, l 表示输入长整型数据(如%ld) 和双精度浮点数(如%lf)。h 表示输入短整型

数据。

使用 scanf 函数还必须注意以下几点:

1) 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。

2) 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。

3) 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。

4) 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。

例如:

```
scanf("%c%c%c",&a,&b,&c);
```

输入为:

```
d e f
```

则把'd'赋予 a, ' ' 赋予 b, 'e'赋予 c。

只有当输入为:

```
def
```

时, 才能把'd'赋予 a, 'e'赋予 b, 'f'赋予 c。

如果在格式控制中加入空格作为间隔,

如:

```
scanf("%c %c %c",&a,&b,&c);
```

则输入时各数据之间可加空格。

输出函数 printf

printf 函数称为格式输出函数, 其关键字最末一个字母 f 即为“格式”(format)之意。其功能是按用户指定的格式, 把指定的数据显示到显示器屏幕上。在前面的例题中我们已多次使用过这个函数。

printf 函数调用的一般形式

printf 函数是一个标准库函数, 它的函数原型在头文件“stdio.h”中。

printf 函数调用的一般形式为:

printf(“格式控制字符串”, 输出表列)

其中格式控制字符串用于指定输出格式。格式控制串可由格式字符串和非格式字符串两种组成。格式字符串是以%开头的字符串, 在%后面跟有各种格式字符, 以说明输出数据的类型、形式、长度、小数位数等。

如:

“%d”表示按是十进制整型输出

“%ld”表示按十进制长整型输出

“%c”表示按字符型输出等

输出表列中给出了各个输出项, 要求格式字符串和各输出项在数量和类型上应该一一对应。

例:

```
1. #include <stdio.h>
2.
3. int main() {
4.     int a = 88, b = 89;
5.     printf("%d %d\n", a, b);
6.     printf("%d,%d\n", a, b);
7.     printf("%c,%c\n", a, b);
8.     printf("a = %d, b = %d\n", a, b);
9.     return 0;
10. }
```

1.7 程序的设计结构

顺序结构

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

选择结构

```
1. #include <stdio.h>
2.
3. int main() {
4.     int a, b;
5.     scanf("%d%d", &a, &b);
6.     if (a > b) {
7.         printf("max num is %d\n", a);
8.     }
9.     else {
10.        printf("max num is %d\n", b);
11.    }
12.    return 0;
13. }
```

循环结构

```
1. #include <stdio.h>
2.
3. int main() {
4.     int n, sum = 0;
5.     scanf("%d", &n);
6.     for (int i = 1; i <= n; i++) {
7.         sum = sum + i;
8.     }
9.     printf("1+2+3+...+%d = %d\n", n, sum);
10.    return 0;
11. }
```

1.8 本章小结

易错点汇总

1、`#include <stdio.h>`和`#include "stdio.h"`的区别

`<>`搜索顺序为：系统目录 -> 环境变量目录 -> 用户自定义目录

`""`搜索顺序为：用户自定义目录 -> 系统目录 -> 环境变量目录

这个区别带来的主要影响是效率问题，如果一个你自己定义的头文件，你用`<>`来包含，那么搜索这个头文件时，将会先从系统目录查找。其实这个头文件可能就在的工程目录下，但是还是要先把系统目录搜索一遍，这样自然就降低效率了。

所以，一般除非引用自己定义的头文件或源文件使用`""`速度会更快之外，引用其他文件用`<>`速度会更快。

对于 C 语言初学的同学来说，记住统一使用`<>`即可。

2、中文符号和英文符号不能混淆使用。

很多初学 C 语言的同学特别容易混淆中文符号和英文符号。

比如

英文逗号， 中文逗号，

英文分号； 中文分号；

请同学一定要记得，C 语言代码统一都是使用英文符号。

一般情况下只有在可以出现中文的地方才能出现中文的标点符号。

实战练习

DreamJudge 1000 A+B 问题

DreamJudge 1186 你好，N 诺

DreamJudge 1122 欢迎来到 NoobDream

DreamJudge 1137 中文测试题目

DreamJudge 1132 平均值

DreamJudge 1160 球的半径和体积

DreamJudge 1125 求三角形的面积

第二章 选择结构程序设计

【本章知识点汇总】

选择结构

条件判断

if 语句

if else 语句

关系运算符与表达式

逻辑运算符与表达式

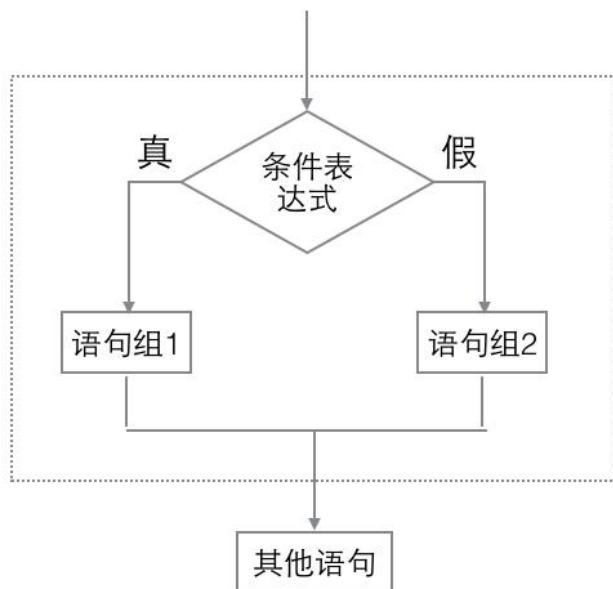
if 语句的嵌套

switch 语句

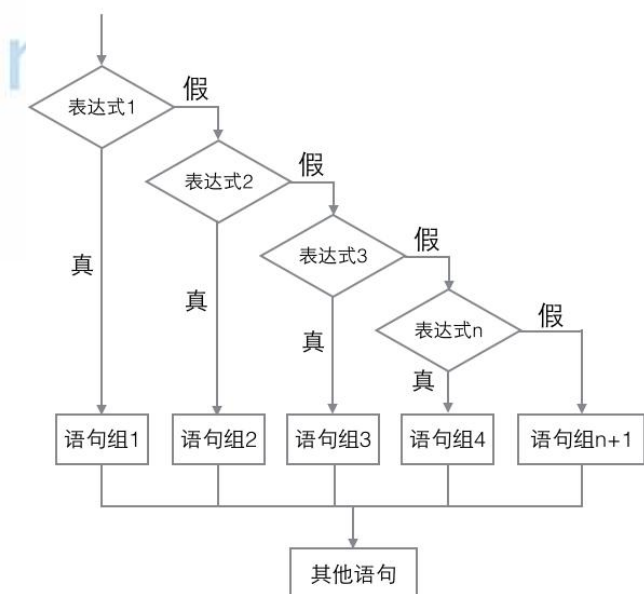
本书配套视频精讲: <https://www.bilibili.com/video/BV1HJ41137fe>

2.1 选择结构和条件判断

双分支选择结构的执行过程



多分支选择结构的执行过程



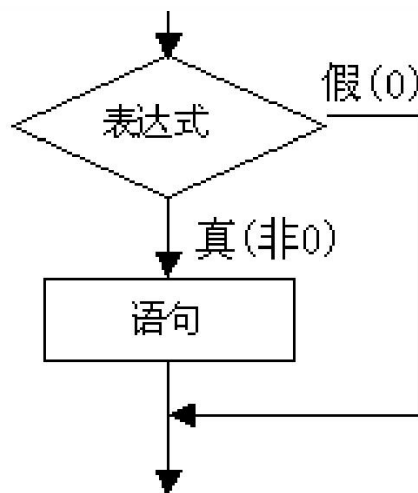
2.2 用 if 语句实现选择结构

用 if 语句可以构成分支结构。它根据给定的条件进行判断, 以决定执行某个分支程序段。C 语言的 if 语句有三种基本形式。

1. 第一种形式为基本形式: if

1. if(表达式) 语句

其语义是: 如果表达式的值为真, 则执行其后的语句, 否则不执行该语句。其过程可表示为下图。



【例】

```
1. #include<stdio.h>
2.
3. int main(){
4.     int a, b, max;
5.     printf("input two numbers:");
6.     scanf("%d%d", &a, &b);
7.     max = a;
8.     if (max < b) max = b;
9.     printf("max=%d", max);
10.    return 0;
11. }
```

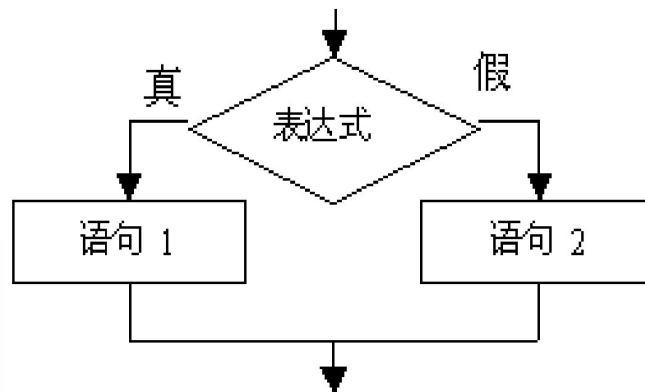
本例程序中, 输入两个数 a,b。把 a 先赋予变量 max, 再用 if 语句判别 max 和 b 的大小, 如 max 小于 b, 则把 b 赋予 max。因此 max 中总是大数, 最后输出 max 的值。

2. 第二种形式为: if-else

1. **if**(表达式)
2. 语句 1;
3. **else**
4. 语句 2;

其语义是：如果表达式的值为真，则执行语句 1，否则执行语句 2。

其执行过程可表示为下图。



【例】

```
1. #include<stdio.h>
2.
3. int main(){
4.     int a, b;
5.     printf("input two numbers: ");
6.     scanf("%d%d", &a, &b);
7.     if (a > b)
8.         printf("max=%d\n", a);
9.     else
10.        printf("max=%d\n", b);
11.     return 0;
12. }
```

输入两个整数，输出其中的大数。

改用 if-else 语句判别 a,b 的大小，若 a 大，则输出 a，否则输出 b。

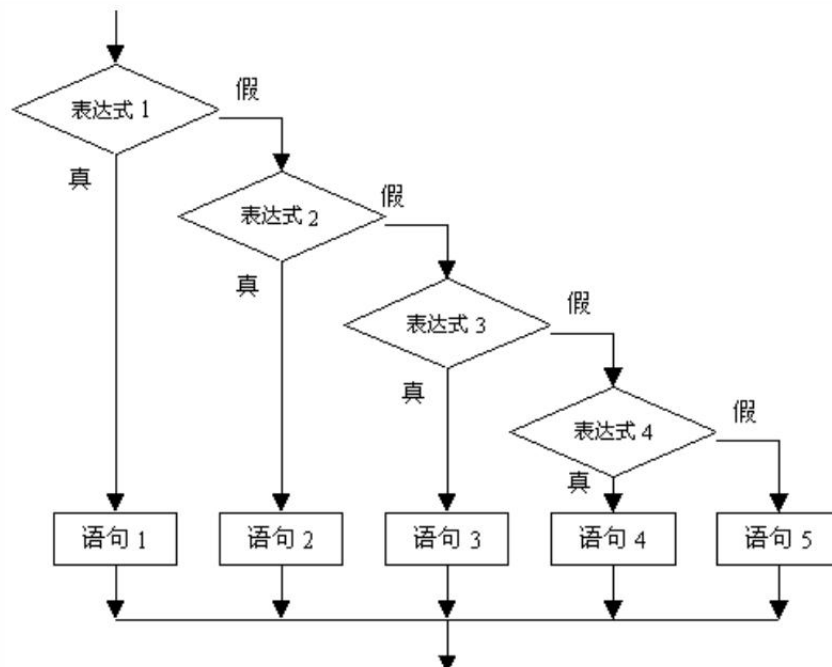
3. 第三种形式为 if-else-if 形式

前二种形式的 if 语句一般都用于两个分支的情况。当有多个分支选择时，可采用 if-else-if 语句，其一般形式为：

1. **if**(表达式 1)
2. 语句 1;
3. **else if**(表达式 2)

4. 语句 2;
5. **else if**(表达式 3)
6. 语句 3;
7. ...
8. **else if**(表达式 m)
9. 语句 m;
10. **else**
11. 语句 n;

其语义是：依次判断表达式的值，当出现某个值为真时，则执行其对应的语句。然后跳到整个 if 语句之外继续执行程序。如果所有的表达式均为假，则执行语句 n。然后继续执行后续程序。if-else-if 语句的执行过程如图所示。



【例】

```

1. #include<stdio.h>
2.
3. int main(){
4.     char c;
5.     printf("input a character: ");
6.     c = getchar();
7.     if (c < 32)
8.         printf("This is a control character\n");
9.     else if (c >= '0' && c <= '9')
10.        printf("This is a digit\n");
11.    else if (c >= 'A' && c <= 'Z')
12.        printf("This is a capital letter\n");

```

```

13.     else if (c >= 'a' && c <= 'z')
14.         printf("This is a small letter\n");
15.     else
16.         printf("This is an other character\n");
17.     return 0;
18. }

```

本例要求判别键盘输入字符的类别。可以根据输入字符的 ASCII 码来判别类型。由 ASCII 码表可知 ASCII 值小于 32 的为控制字符。在“0”和“9”之间的为数字，在“A”和“Z”之间为大写字母，在“a”和“z”之间为小写字母，其余则为其它字符。这是一个多分支选择的问题，用 if-else-if 语句编程，判断输入字符 ASCII 码所在的范围，分别给出不同的输出。例如输入为“g”，输出显示它为小写字符。

4. 在使用 if 语句中还应注意以下问题：

1) 在三种形式的 if 语句中，在 if 关键字之后均为表达式。该表达式通常是逻辑表达式或关系表达式，但也可以是其它表达式，如赋值表达式等，甚至也可以是一个变量。

例如：

1. if(a=5) 语句;
2. if(b) 语句;

都是允许的。只要表达式的值为非 0，即为“真”。

如在：

1. if(a=5)...

中表达式的值永远为非 0，所以其后的语句总是要执行的，当然这种情况在程序中不一定会出现，但在语法上是合法的。

又如，有程序段：

1. if(a=b)
2. printf("%d",a);
3. else
4. printf("a=0");

本语句的语义是，把 b 值赋予 a，如为非 0 则输出该值，否则输出“a=0”字符串。这种用法在程

序中是经常出现的。

2) 在 if 语句中，条件判断表达式必须用括号括起来，在语句之后必须加分号。

3) 在 if 语句的三种形式中, 所有的语句应为单个语句, 如果要想在满足条件时执行一组(多个)语句, 则必

须把这一组语句用 {} 括起来组成一个复合语句。但要注意的是在 } 之后不能再加分号。

例如:

```
1.  if (a > b)
2.  {
3.      a++;
4.      b++;
5.  }
6.  else
7.  {
8.      a = 0;
9.      b = 10;
10. }
```

N 诺

noobdream.com

2.3 关系运算符和关系表达式

关系运算符

在 C 语言中有以下关系运算符：

1. < 小于
2. <= 小于或等于
3. > 大于
4. >= 大于或等于
5. == 等于
6. != 不等于

关系运算符都是双目运算符，其结合性均为左结合。关系运算符的优先级低于算术运算符，高于赋值运算符。在六个关系运算符中，<,<=,>,>=的优先级相同，高于==和!=，==和!=的优先级相同。

关系表达式

关系表达式的一般形式为：

1. 表达式 关系运算符 表达式

例如：

1. $a+b > c-d$
2. $x > 3/2$
3. $'a'+1 < c$
4. $-i-5*j == k+1$

都是合法的关系表达式。由于表达式也可以又是关系表达式。因此也允许出现嵌套的情况。

例如：

1. $a > (b > c)$
2. $a != (c == d)$

关系表达式的值是“真”和“假”，用“1”和“0”表示。

如：

$5 > 0$ 的值为“真”，即为 1。

$(a=3)>(b=5)$ 由于 $3>5$ 不成立, 故其值为假, 即为 0。

【例】

```
1. #include<stdio.h>
2.
3. int main() {
4.     char c = 'k';
5.     int i = 1, j = 2, k = 3;
6.     float x = 3e+5, y = 0.85;
7.     printf("%d,%d\n", 'a'+5<c, -i-2*j>=k+1);
8.     printf("%d,%d\n", 1<j<5, x-5.25<=x+y);
9.     printf("%d,%d\n", i+j+k== -2*j, k==j==i+5);
10.    return 0;
11. }
```

在本例中求出了各种关系运算符的值。字符变量是以它对应的 ASCII 码参与运算的。对于含多个关系运算符的表达式, 如 $k==j==i+5$, 根据运算符的左结合性, 先计算 $k==j$, 该式不成立, 其值为 0, 再计算 $0==i+5$, 也不成立, 故表达式值为 0。

noobdream.com

2.4 逻辑运算符和逻辑表达式

逻辑运算符

C语言中提供了三种逻辑运算符：

- 1) && 与运算
- 2) || 或运算
- 3) ! 非运算

与运算符&&和或运算符||均为双目运算符。具有左结合性。非运算符!为单目运算符，具有右结合性。逻辑运算符和其它运算符优先级的关系可表示如下：

！（非）

算术运算符

关系运算符

&& 和 ||

赋值运算符

！（非） \rightarrow &&（与） \rightarrow ||（或）

“&&”和“||”低于关系运算符，“！”高于算术运算符。

按照运算符的优先顺序可以得出：

$a > b \ \&\& \ c > d$ 等价于 $(a > b) \&\& (c > d)$

$!b == c \ || \ d < a$ 等价于 $((!b) == c) \ || \ (d < a)$

$a + b > c \ \&\& \ x + y < b$ 等价于 $((a + b) > c) \ \&\& \ ((x + y) < b)$

逻辑运算的值

逻辑运算的值也为“真”和“假”两种，用“1”和“0”来表示。其求值规则如下：

1. 与运算 &&：参与运算的两个量都为真时，结果才为真，否则为假。

例如：

$5 > 0 \ \&\& \ 4 > 2$

由于 $5 > 0$ 为真， $4 > 2$ 也为真，相与的结果也为真。

2. 或运算 ||：参与运算的两个量只要有一个为真，结果就为真。两个量都为假时，结果为假。

例如:

$5 > 0 \parallel 5 > 8$

由于 $5 > 0$ 为真, 相或的结果也就为真。

3. 非运算!: 参与运算量为真时, 结果为假; 参与运算量为假时, 结果为真。

例如:

$!(5 > 0)$

的结果为假。

虽然 C 编译在给出逻辑运算值时, 以 “1”代表“真”, “0”代表“假”。但反过来在判断一个量是为“真”

还是为“假”时, 以“0”代表“假”, 以非“0”的数值作为“真”。例如:

由于 5 和 3 均为非“0”因此 $5 \&\&3$ 的值为“真”, 即为 1。

又如:

$5 \parallel 0$ 的值为“真”, 即为 1。

逻辑表达式

逻辑表达式的一般形式为:

1. 表达式 逻辑运算符 表达式

其中的表达式可以又是逻辑表达式, 从而组成了嵌套的情形。

例如:

$(a \&\&b) \&\&c$

根据逻辑运算符的左结合性, 上式也可写为:

$a \&\&b \&\&c$

逻辑表达式的值是式中各种逻辑运算的最后值, 以 “1”和“0”分别代表“真”和“假”。

【例】

```
1. #include<stdio.h>
2.
3. int main(){
4.     char c = 'k';
5.     int i = 1, j = 2, k = 3;
```

```
6.     float x = 3e+5,y = 0.85;
7.     printf("%d,%d\n", !x*!y, !!!x);
8.     printf("%d,%d\n", x||i&&j-3, i<j&&x<y);
9.     printf("%d,%d\n", i==5&&c&&(j=8), x+y||i+j+k);
10.    return 0;
11. }
```

本例中!x 和!y 分别为 0, !x*!y 也为 0, 故其输出值为 0。由于 x 为非 0, 故!!!x 的逻辑值为 0。对 x||i&&j-3 式, 先计算 j-3 的值为非 0, 再求 i&&j-3 的逻辑值为 1, 故 x||i&&j-3 的逻辑值为 1。对 i<j&&x<y 式, 由于 i<j 的值为 1, 而 x<y 为 0 故表达式的值为 1, 0 相与, 最后为 0, 对 i==5&&c&&(j=8)式, 由于 i==5 为假, 即值为 0, 该表达式由两个与运算组成, 所以整个表达式的值为 0。对于式 x+y||i+j+k 由于 x+y 的值为非 0, 故整个或表达式的值为 1。



2.5 条件运算符和条件表达式

如果在条件语句中，只执行单个的赋值语句时，常可使用条件表达式来实现。不但使程序简洁，也提高了运行效率。

条件运算符为?和:，它是一个三目运算符，即有三个参与运算的量。

由条件运算符组成条件表达式的一般形式为：

1. 表达式 1 ? 表达式 2 : 表达式 3

其求值规则为：如果表达式 1 的值为真，则以表达式 2 的值作为条件表达式的值，否则以表达式 3 的值作为整个条件表达式的值。

条件表达式通常用于赋值语句之中。

例如条件语句：

```
1. if(a > b) max = a;  
2. else max = b;
```

可用条件表达式写为

1. max = (a > b) ? a : b;

执行该语句的语义是：如 a>b 为真，则把 a 赋予 max，否则把 b 赋予 max。

使用条件表达式时，还应注意以下几点：

1) 条件运算符的运算优先级低于关系运算符和算术运算符，但高于赋值符。

因此

1. max = (a > b) ? a : b;

可以去掉括号而写为

1. max = a > b ? a : b;

2) 条件运算符?和: 是一对运算符，不能分开单独使用。

3) 条件运算符的结合方向是自右至左。

例如：

1. a > b ? a : c > d ? c : d

应理解为

1. a > b ? a : (c > d ? c : d)

这也就是条件表达式嵌套的情形，即其中的表达式 3 又是一个条件表达式。

【例】

```
1. #include<stdio.h>
2.
3. int main() {
4.     int a, b, max;
5.     printf("\n input two numbers: ");
6.     scanf("%d%d", &a, &b);
7.     printf("max=%d", a>b?a:b);
8.     return 0;
9. }
```

用条件表达式对上例重新编程，输出两个数中的大数。



2.6 选择结构的嵌套

当 if 语句中的执行语句又是 if 语句时, 则构成了 if 语句嵌套的情形。

其一般形式可表示如下:

1. **if**(表达式)
2. **if** 语句;
3. 或者为
4. **if**(表达式)
5. **if** 语句;
6. **else**
7. **if** 语句

在嵌套内的 if 语句可能又是 if-else 型的, 这将会出现多个 if 和多个 else 重叠的情况, 这时要特别注

意 if 和 else 的配对问题。

例如:

1. **if**(表达式 1)
2. **if**(表达式 2)
3. 语句 1;
4. **else**
5. 语句 2;

其中的 else 究竟是与哪一个 if 配对呢?

应该理解为:

1. **if**(表达式 1)
2. **if**(表达式 2)
3. 语句 1;
4. **else**
5. 语句 2;

还是应理解为:

1. **if**(表达式 1)
2. **if**(表达式 2)
3. 语句 1;
4. **else**
5. 语句 2;

为了避免这种二义性, C 语言规定, else 总是与它前面最近的 if 配对, 因此对上述例子应按前一种情况理解。

【例】

```
1. #include<stdio.h>
2.
3. int main() {
4.     int a, b;
5.     printf("please input A,B: ");
6.     scanf("%d%d", &a, &b);
7.     if (a != b)
8.         if (a > b) printf("A>B\n");
9.         else printf("A<B\n");
10.    else printf("A=B\n");
11.    return 0;
12. }
```

比较两个数的大小关系。

本例中用了 if 语句的嵌套结构。采用嵌套结构实质上是为了进行多分支选择，实际上有三种选择即 $A>B$ 、 $A<B$ 或 $A=B$ 。这种问题用 if-else-if 语句也可以完成。而且程序更加清晰。因此，在一般情况下较少使用 if 语句的嵌套结构。以使程序更便于阅读理解。

【例】

```
1. #include<stdio.h>
2.
3. int main() {
4.     int a, b;
5.     printf("please input A,B: ");
6.     scanf("%d%d", &a, &b);
7.     if (a == b) printf("A=B\n");
8.     else if (a > b) printf("A>B\n");
9.     else printf("A<B\n");
10.    return 0;
11. }
```

2.7 用 switch 语句实现多分支选择结构

C 语言还提供了另一种用于多分支选择的 switch 语句，其一般形式为：

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

其语义是：计算表达式的值。并逐个与其后的常量表达式值相比较，当表达式的值与某个常量表达式的值相等时，即执行其后的语句，然后不再进行判断，继续执行后面所有 case 后的语句。如表达式的值与所有 case 后的常量表达式均不相同，则执行 default 后的语句。

【例】

```
1. #include<stdio.h>  
2.  
3. int main() {  
4.     int a;  
5.     printf("input integer number: ");  
6.     scanf("%d", &a);  
7.     switch (a){  
8.         case 1: printf("Monday\n");  
9.         case 2: printf("Tuesday\n");  
10.        case 3: printf("Wednesday\n");  
11.        case 4: printf("Thursday\n");  
12.        case 5: printf("Friday\n");  
13.        case 6: printf("Saturday\n");  
14.        case 7:printf("Sunday\n");  
15.        default:printf("error\n");  
16.    }  
17.    return 0;  
18. }
```

本程序是要求输入一个数字，输出一个英文单词。但是当输入 3 之后，却执行了 case3 以及以后的所有语句，输出了 Wednesday 及以后的所有单词。这当然是不希望的。为什么会出现这种情况呢？这恰恰反应 switch 语句的一个特点。在 switch 语句中，“case 常量表达式”只相

当于一个语句标号，表达式的值和某标号相等则转向该标号执行，但不能在执行完该标号的语句后自动跳出整个 switch 语句，所以出现了继续执行所有后面 case 语句的情况。这是与前面介绍的 if 语句完全不同的，应特别注意。为了避免上述情况，C 语言还提供了一种 break 语句，专用于跳出 switch 语句，break 语句只有关键字 break，没有参数。在后面还将详细介绍。修改例题的程序，在每一 case 语句之后增加 break 语句，使每一次执行之后均可跳出 switch 语句，从而避免输出不应有的结果。

【例】

```
1. #include<stdio.h>
2.
3. int main() {
4.     int a;
5.     printf("input integer number: ");
6.     scanf("%d", &a);
7.     switch (a){
8.         case 1: printf("Monday\n");break;
9.         case 2: printf("Tuesday\n");break;
10.        case 3: printf("Wednesday\n");break;
11.        case 4: printf("Thursday\n");break;
12.        case 5: printf("Friday\n");break;
13.        case 6: printf("Saturday\n");break;
14.        case 7:printf("Sunday\n");break;
15.        default:printf("error\n");
16.    }
17.    return 0;
18. }
```

在使用 switch 语句时还应注意以下几点：

- 1) 在 case 后的各常量表达式的值不能相同，否则会出现错误。
- 2) 在 case 后，允许有多个语句，可以不用 {} 括起来。
- 3) 各 case 和 default 子句的先后顺序可以变动，而不会影响程序执行结果。
- 4) default 子句可以省略不用。

2.8 本章小结

易错点汇总

如果没有 {} 明确语句的作用范围，那么 if 等语句的作用范围有多大？

答：if 之后最近的那条语句属于 if 的作用范围。

```
1.  if (a > b)
2.      if (c > d)
3.          x = 1
4.      else
5.          x = 2
```

等价于

```
1.  if (a > b) {
2.      if (c > d) {
3.          x = 1
4.      }
5.      else {
6.          x = 2
7.      }
8. }
```

noobdream.com

实战练习

DreamJudge 1187 A - B

DreamJudge 1031 判断是否是整数

DreamJudge 1036 三个数的最大值

DreamJudge 1038 成绩的等级

DreamJudge 1040 利润提成

DreamJudge 1091 促销计算

DreamJudge 1037 计算函数

第三章 循环结构程序设计

【本章知识点汇总】

while 语句

do...while 语句

for 语句

循环的嵌套

几种循环的比较

break 语句

continue 语句

goto 语句

本书配套视频精讲: <https://www.bilibili.com/video/BV1HJ41137fe>

3.1 goto 语句构成循环

循环结构是程序中一种很重要的结构。其特点是, 在给定条件成立时, 反复执行某程序段, 直到条件不成立为止。给定的条件称为循环条件, 反复执行的程序段称为循环体。C 语言提供了多种循环语句, 可以组成各种不同形式的循环结构。

- 1) 用 goto 语句和 if 语句构成循环;
- 2) 用 while 语句;
- 3) 用 do-while 语句;
- 4) 用 for 语句;

goto 语句是一种无条件转移语句, 与 BASIC 中的 goto 语句相似。goto 语句的使用格式为:

1. goto 语句标号;

其中标号是一个有效的标识符, 这个标识符加上一个“:”一起出现在函数内某处, 执行 goto 语句后, 程序将跳转到该标号处并执行其后的语句。另外标号必须与 goto 语句同处于一个函数中, 但可以不在一个循环层中。通常 goto 语句与 if 条件语句连用, 当满足某一条件时, 程序跳到标号处运行。goto 语句通常不用, 主要因为它将使程序层次不清, 且不易读, 但在多层嵌套退出时, 用 goto 语句则比较合理。

【例】用 goto 语句和 if 语句构成循环求 $1+2+3+\dots+100$ 。

```
1. #include <stdio.h>
2.
3. int main() {
4.     int i, sum = 0;
5.     i = 1;
6.     loop:
7.     if(i <= 100) {
8.         sum = sum + i;
9.         i++;
10.        goto loop;
11.    }
12.    printf("%d\n", sum);
13.    return 0;
14. }
```

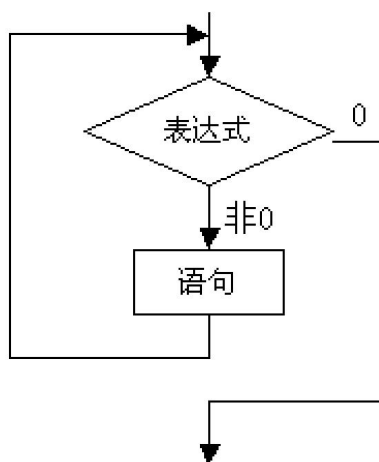

3.2 用 while 语句实现循环

while 语句的一般形式为:

1. **while**(表达式) 语句

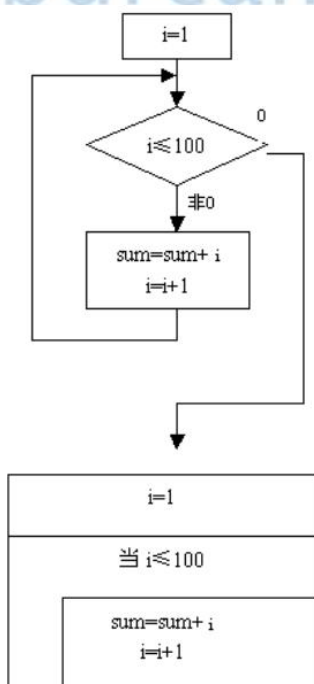
其中表达式是循环条件, 语句为循环体。

while 语句的语义是: 计算表达式的值, 当值为真(非 0)时, 执行循环体语句。其执行过程可用下图表示。



【例】用 while 语句求 $1+2+3+\dots+100$ 。

用传统流程图和 N-S 结构流程图表示算法, 见图:



```

1. #include <stdio.h>
2.
3. int main() {
4.     int i, sum = 0;
5.     i = 1;
6.     while(i <= 100) {
7.         sum = sum + i;
8.         i++;
9.     }
10.    printf("%d\n", sum);
11.    return 0;
12. }

```

【例】统计从键盘输入一行字符的个数。

```

1. #include <stdio.h>
2.
3. int main() {
4.     int n = 0;
5.     printf("input a string:\n");
6.     while(getchar() != '\n') n++;
7.     printf("%d", n);
8.     return 0;
9. }

```

本例程序中的循环条件为 `getchar()!='\n'`,其意义是, 只要从键盘输入的字符不是回车就继续循环。循环体 `n++`完成对输入字符个数计数。从而程序实现了对输入一行字符的字符个数计数。

使用 `while` 语句应注意以下几点:

1) `while` 语句中的表达式一般是关系表达或逻辑表达式, 只要表达式的值为真(非 0)即可继续循环。

【例】

```

1. #include <stdio.h>
2.

```

```
3. int main() {  
4.     int a = 0,n;  
5.     printf("input n: ");  
6.     scanf("%d", &n);  
7.     while (n--)  
8.         printf("%d ", a++*2);  
9.     return 0;  
10. }
```

本例程序将执行 n 次循环, 每执行一次, n 值减 1。循环体输出表达式 $a++*2$ 的值。该表达式等效于 $(a*2; a++)$ 。

2) 循环体如包括有一个以上的语句, 则必须用 $\{\}$ 括起来, 组成复合语句。

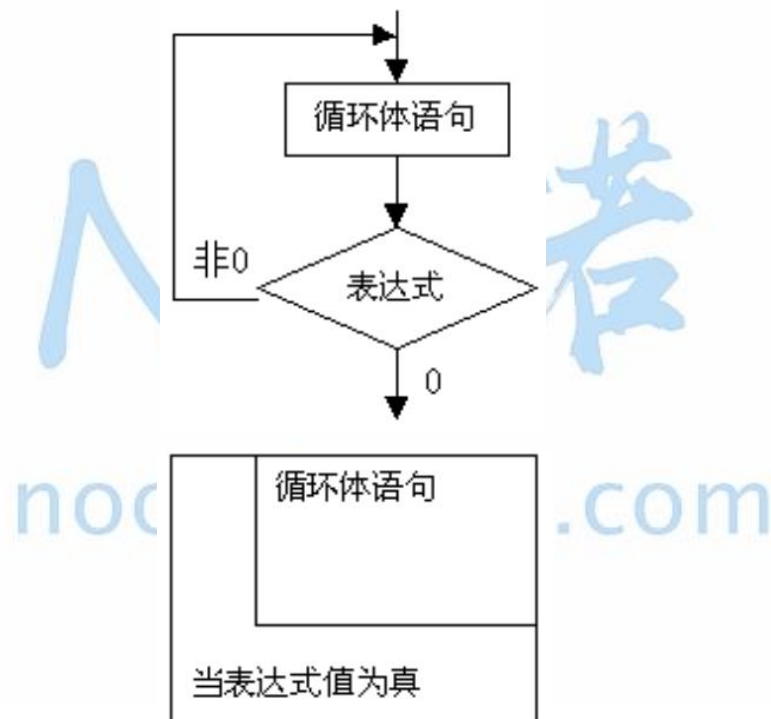


3.3 用 do...while 语句实现循环

do-while 语句的一般形式为:

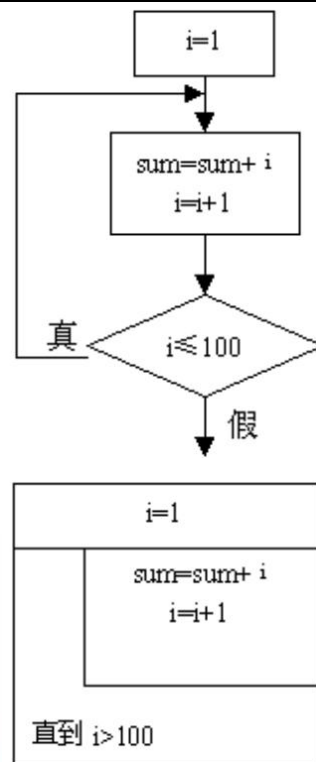
1. **do**
2. 语句
3. **while**(表达式);

这个循环与 while 循环的不同在于:它先执行循环中的语句,然后再判断表达式是否为真,如果为真则继续循环;如果为假,则终止循环。因此,do-while 循环至少要执行一次循环语句。其执行过程可用下图表示。



【例】用 do-while 语句求 $1+2+3+\dots+100$ 。

用传统流程图和 N-S 结构流程图表示算法, 见图:



```
1. #include <stdio.h>
2.
3. int main() {
4.     int i, sum = 0;
5.     i = 1;
6.     do {
7.         sum = sum + i;
8.         i++;
9.     }
10.    while(i <= 100);
11.    printf("%d\n", sum);
12.    return 0;
13. }
```

同样当有许多语句参加循环时, 要用 "{" 和 "}" 把它们括起来。

【例】while 和 do-while 循环比较。

(1)

```
1. #include <stdio.h>
2.
3. int main() {
4.     int sum = 0, i;
5.     scanf("%d", &i);
```

```
6.     while(i <= 10) {  
7.         sum = sum + i;  
8.         i++;  
9.     }  
10.    printf("sum=%d", sum);  
11.    return 0;  
12. }
```

(2)

```
1.  #include <stdio.h>  
2.  
3.  int main() {  
4.      int sum = 0, i;  
5.      scanf("%d", &i);  
6.      do {  
7.          sum = sum + i;  
8.          i++;  
9.      }  
10.     while(i <= 10);  
11.     printf("sum=%d", sum);  
12.     return 0;  
13. }
```

noobdream.com

3.4 用 for 语句实现循环

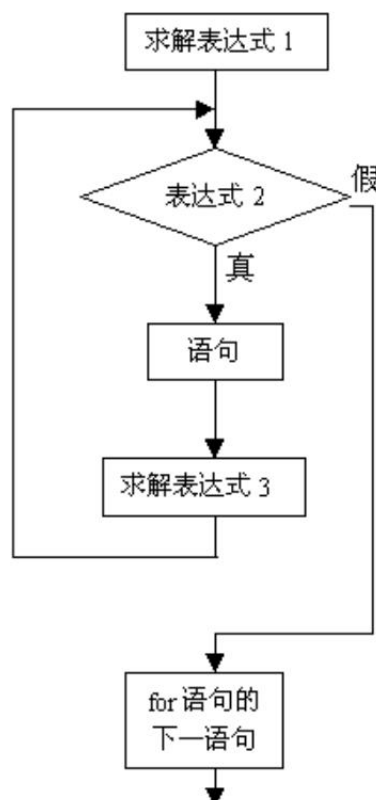
在 C 语言中, for 语句使用最为灵活, 它完全可以取代 while 语句。它的一般形式为:

1. **for**(表达式 1; 表达式 2; 表达式 3)
2. 语句

它的执行过程如下:

- 1) 先求解表达式 1。
- 2) 求解表达式 2, 若其值为真 (非 0), 则执行 for 语句中指定的内嵌语句, 然后执行下面第 3) 步; 若其值为假 (0), 则结束循环, 转到第 5) 步。
- 3) 求解表达式 3。
- 4) 转回上面第 2) 步继续执行。
- 5) 循环结束, 执行 for 语句下面的一个语句。

其执行过程可用下图表示。



for 语句最简单的应用形式也是最容易理解的形式如下:

1. **for**(循环变量赋初值; 循环条件; 循环变量增量)
2. 语句

循环变量赋初值总是一个赋值语句, 它用来给循环控制变量赋初值; 循环条件是一个关系表达式, 它决定什么时候退出循环; 循环变量增量, 定义循环控制变量每循环一次后 按什么方式变化。这三个部分之间用 “; ” 分开。

例如:

```
1. for(i = 1; i <= 100; i++)
2.     sum = sum + i;
```

先给 i 赋初值 1, 判断 i 是否小于等于 100, 若是则执行语句, 之后值增加 1。再重新判断, 直到条件为假, 即 $i > 100$ 时, 结束循环。

相当于:

```
1. i = 1;
2. while(i <= 100) {
3.     sum = sum + i;
4.     i++;
5. }
```

对于 for 循环中语句的一般形式, 就是如下的 while 循环形式:

```
1. 表达式 1;
2. while(表达式 2) {
3.     语句
4.     表达式 3;
5. }
```

注意:

1) for 循环中的“表达式 1 (循环变量赋初值)”、“表达式 2 (循环条件)”和“表达式 3 (循环变量增量)”都是选择项, 即可以缺省, 但 “; ” 不能缺省。

2) 省略了“表达式 1 (循环变量赋初值)”, 表示不对循环控制变量赋初值。

3) 省略了“表达式 2 (循环条件)”, 则不做其它处理时便成为死循环。

例如:

```
1. for(i = 1; ; i++)
2.     sum = sum + i;
```

相当于:

```
1. i = 1;
2. while(1) {
```



```
3.     sum = sum + i;  
4.     i++;  
5. }
```

4) 省略了“表达式 3(循环变量增量)”，则不对循环控制变量进行操作,这时可在语句体中加入修改循环控制变量的语句。

例如：

```
1.  for(i = 1; i <= 100; ) {  
2.     sum = sum + i;  
3.     i++;  
4. }
```

5) 省略了“表达式 1（循环变量赋初值）”和“表达式 3(循环变量增量)”。

例如：

```
1.  for( ; i <= 100; ) {  
2.     sum = sum + i;  
3.     i++;  
4. }
```

相当于：

```
1.  while(i <= 100) {  
2.     sum = sum + i;  
3.     i++;  
4. }
```

6) 3 个表达式都可以省略。

例如：

```
1.  for(;;) 语句
```

相当于：

```
1.  while(1) 语句
```

7) 表达式 1 可以是设置循环变量的初值的赋值表达式，也可以是其他表达式。

例如：

```
1.  for(sum = 0; i <= 100; i++)  
2.     sum = sum + i;
```

8) 表达式 1 和表达式 3 可以是一个简单表达式也可以是逗号表达式。

```
1. for(sum = 0, i = 1; i <= 100; i++)  
2.     sum = sum + i;
```

或:

```
1. for(i = 0, j = 100; i <= 100; i++, j--)  
2.     k = i + j;
```

9) 表达式 2 一般是关系表达式或逻辑表达式, 但也可以是数值表达式或字符表达式, 只要其值非零, 就执行循环体。

例如:

```
1. for(i = 0; (c = getchar()) != '\n'; i += c);
```

又如:

```
1. for( ; (c = getchar()) != '\n'; )  
2.     printf("%c", c);
```

N 诺
noobdream.com

3.5 循环的嵌套

【例】

```
1. #include <stdio.h>
2.
3. int main() {
4.     int i, j, k;
5.     printf("i j k\n");
6.     for (i = 0; i < 2; i++)
7.         for(j = 0; j < 2; j++)
8.             for(k = 0; k < 2; k++)
9.                 printf("%d %d %d\n", i, j, k);
10.    return 0;
11. }
```

【例】用符号@画一个 $n * n$ 的正方形。

```
1. #include <stdio.h>
2.
3. int main() {
4.     int n;
5.     scanf("%d", &n);
6.     for (int i = 1; i <= n; i++) {
7.         for (int j = 1; j <= n; j++) {
8.             printf("@");
9.         }
10.        printf("\n");
11.    }
12.    return 0;
13. }
```

3.6 几种循环的比较

- 1) 四种循环都可以用来处理同一个问题，一般可以互相代替。但一般不提倡用 `goto` 型循环。
- 2) `while` 和 `do-while` 循环，循环体中应包括使循环趋于结束的语句。`for` 语句功能最强。
- 3) 用 `while` 和 `do-while` 循环时，循环变量初始化的操作应在 `while` 和 `do-while` 语句之前完成，而 `for` 语句可以在表达式 1 中实现循环变量的初始化。



3.7 改变循环执行的状态

用 break 语句提前终止循环

break 语句通常用在循环语句和开关语句中。当 break 用于开关语句 switch 中时,可使程序跳出 switch 而执行 switch 以后的语句;如果没有 break 语句,则将成为一个死循环而无法退出。break 在 switch 中的用法已在前面介绍开关语句时的例子中碰到,这里不再举例。

当 break 语句用于 do-while、for、while 循环语句中时,可使程序终止循环而执行循环后面的语句,通常 break 语句总是与 if 语句联在一起。即满足条件时便跳出循环。

【例】

```
1. #include <stdio.h>
2. #include <conio.h>
3.
4. int main() {
5.     int i = 0;
6.     char c;
7.     while(1) /*设置循环*/
8.     {
9.         c = '\0'; /*变量赋初值*/
10.        while(c != 13 && c != 27) /*键盘接收字符直到按回车或 Esc 键*/
11.        {
12.            c = getch();
13.            printf("%c\n", c);
14.        }
15.        if(c == 27) break; /*判断若按 Esc 键则退出循环*/
16.        i++;
17.        printf("The No. is %d\n", i);
18.    }
19.    printf("The end");
20.    return 0;
21. }
```

注意:

- 1) break 语句对 if-else 的条件语句不起作用。
- 2) 在多层循环中,一个 break 语句只向外跳一层。

用 continue 语句提前结束本次循环

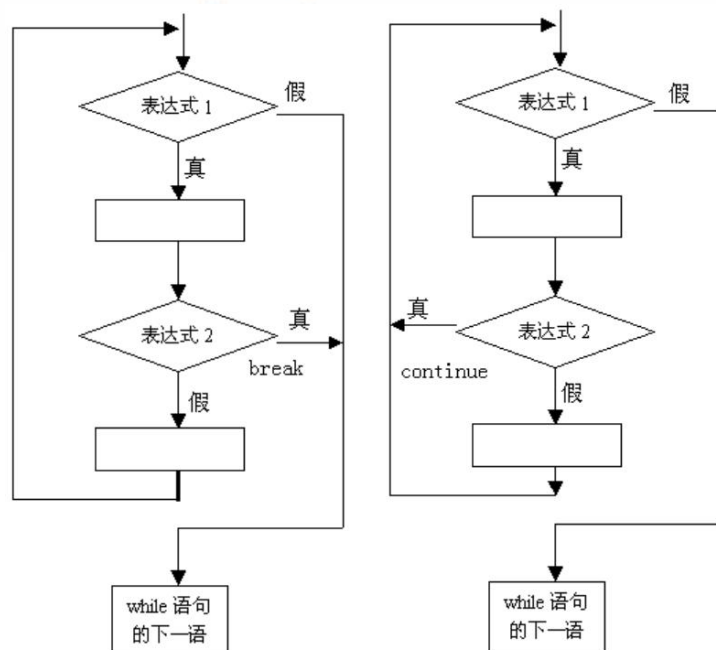
continue 语句的作用是跳过循环本中剩余的语句而强行执行下一次循环。continue 语句只用在 for、while、do-while 等循环体中,常与 if 条件语句一起使用,用来加速循环。其执行过程可用下图表示。

1)

```
1. while(表达式 1)
2. {
3.     .....
4.     if(表达式 2) break;
5.     .....
6. }
```

2)

```
1. while(表达式 1)
2. {
3.     .....
4.     if(表达式 2) continue;
5.     .....
6. }
```



【例】

```
1. #include <stdio.h>
2. #include <conio.h>
3.
4. int main() {
5.     char c;
6.     while(c!=13) /*不是回车符则循环*/
7.     {
8.         c=getch();
9.         if(c==0X1B)
10.            continue; /*若按 Esc 键不输出便进行下次循环*/
11.         printf("%c\n", c);
12.     }
13.     return 0;
14. }
```

break 语句和 continue 语句的区别

break 语句或 continue 语句之后语句不会执行

break 语句是跳出本层循环，而 continue 语句是跳出本次循环

1、break: while 循环 break 是用于永久终止循环。即不执行本次循环中 break 后面的语句，直接跳出循环。

2、continue: while 循环 continue 是用于终止本次循环。即本次循环中 continue 后面的代码不执行，进行下一次循环的入口判断。

3.8 本章小结

C 语言专业题库 – 手机在线刷题



实战练习

DreamJudge 1133 求 1 到 n 的和

DreamJudge 1047 分数求和

DreamJudge 1045 平方和与倒数和

DreamJudge 1043 计算 S_n

DreamJudge 1007 整除

第四章 数组

【本章知识点汇总】

一维数组

二维数组

字符数组



本书配套视频精讲: <https://www.bilibili.com/video/BV1HJ41137fe>

4.1 一维数组

定义

在C语言中使用数组必须先进行定义。

一维数组的定义方式为：

1. 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 main()
2. {
3.     int a;
4.     float a[10];
5.     .....
6. }
```

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
2. int main()
3. {
4.     int a[3+2],b[7+FD];
5.     .....
6. }
```

是合法的。

但是下述说明方式是错误的。

```
1. int main()
2. {
3.     int n=5;
4.     int a[n];
5.     .....
6. }
```

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];
```

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顺序号。

数组元素的一般形式为:

```
1. 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:

1. `a[5]`
2. `a[i+j]`
3. `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。

例如，输出有 10 个元素的数组必须使用循环语句逐个输出各下标变量：

1. `for(i = 0; i < 10; i++)`
2. `printf("%d", a[i]);`

而不能用一个语句输出整个数组。

下面的写法是错误的：

1. `printf("%d", a);`

【例】

```
1. #include <stdio.h>
2.
3. int main() {
4.     int i, a[10];
5.     for(i = 0; i <= 9; i++)
6.         a[i] = i;
7.     for(i = 9; i >= 0; i--)
8.         printf("%d ", a[i]);
9.     return 0;
10. }
```

【例】

```
1. #include <stdio.h>
2.
3. int main() {
4.     int i, a[10];
5.     for(i = 0; i < 10; i++)
6.         a[i++] = i;
7.     for(i = 9; i >= 0; i--)
8.         printf("%d", a[i]);
9.     return 0;
10. }
```

【例】

```
1. #include <stdio.h>
2.
3. int main() {
4.     int i, a[10];
5.     for(i = 0; i < 10; )
6.         a[i++] = 2 * i + 1;
7.     for(i = 0; i <= 9; i++)
8.         printf("%d ", a[i]);
9.     printf("\n%d %d\n", a[11/2], a[10/2]);
10.    return 0;
11. }
```

本例中用一个循环语句给 a 数组各元素送入奇数值，然后用第二个循环语句输出各个奇数。在第一个 for 语句中，表达式 3 省略了。在下标变量中使用了表达式 i++，用以修改循环变量。当然第二个 for 语句也可以这样作，C 语言允许用表达式表示下标。程序中最后一个 printf 语句输出了两次 a[5] 的值，可以看出当下标不为整数时将自动取整。

初始化

给数组赋值的方法除了用赋值语句对数组元素逐个赋值外，还可采用初始化赋值和动态赋值的方法。

数组初始化赋值是指在数组定义时给数组元素赋予初值。数组初始化是在编译阶段进行的。这样将减少运行时间，提高效率。

初始化赋值的一般形式为：

1. 类型说明符 数组名[常量表达式]={值, 值.....值};

其中在 { } 中的各数据值即为各元素的初值，各值之间用逗号间隔。

例如：

1. int a[10]={ 0,1,2,3,4,5,6,7,8,9 };
2. 相当于
3. a[0]=0;a[1]=1...a[9]=9;

C 语言对数组的初始化赋值还有以下几点规定：

1) 可以只给部分元素赋初值。

当{ }中值的个数少于元素个数时, 只 给前面部分元素赋值。

例如:

```
1. int a[10]={0,1,2,3,4};
```

表示只给 a[0]~a[4] 5 个元素赋值, 而后 5 个元素自动赋 0 值。

2) 只能给元素逐个赋值, 不能给数组整体赋值。

例如给十个元素全部赋 1 值, 只能写为:

```
1. int a[10]={1,1,1,1,1,1,1,1,1,1};
```

而不能写为:

```
1. int a[10]=1;
```

3) 如给全部元素赋值, 则在数组说明中, 可以不给出数组元素的个数。

例如:

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

可写为:

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

noobdream.com

应用

可以在程序执行过程中, 对数组作动态赋值。这时可用循环语句配合 scanf 函数逐个对数组元素赋值。

【例】

```
1. #include <stdio.h>
2.
3. int main() {
4.     int i, max, a[10];
5.     printf("input 10 numbers:\n");
6.     for(i = 0; i < 10; i++)
7.         scanf("%d", &a[i]);
8.     max = a[0];
9.     for(i = 1; i < 10; i++)
10.        if(a[i] > max)
```

```

11.         max = a[i];
12.     printf("maxum=%d\n", max);
13.     return 0;
14. }

```

本例程序中第一个 for 语句逐个输入 10 个数到数组 a 中。然后把 a[0]送入 max 中。在第二个 for 语句中, 从 a[1]到 a[9]逐个与 max 中的内容比较, 若比 max 的值大, 则把该下标变量送入 max 中, 因此 max 总是在已比较过的下标变量中为最大者。比较结束, 输出 max 的值。

【例】输入 10 个数, 从大到小输出。

```

1.  #include <stdio.h>
2.
3.  int main() {
4.      int i, j, p, q, s, a[10];
5.      printf("input 10 numbers:\n");
6.      for(i = 0; i < 10; i++)
7.          scanf("%d", &a[i]);
8.      for(i = 0; i < 10; i++) {
9.          p = i;
10.         q = a[i];
11.         for(j = i + 1; j < 10; j++)
12.             if(q < a[j]) {
13.                 p = j;
14.                 q = a[j];
15.             }
16.         if(i != p) {
17.             s = a[i];
18.             a[i] = a[p];
19.             a[p] = s;
20.         }
21.         printf("%d ", a[i]);
22.     }
23.     return 0;
24. }

```

本例程序中用了两个并列的 for 循环语句, 在第二个 for 语句中又嵌套了一个循环语句。第一个 for 语句用于输入 10 个元素的初值。第二个 for 语句用于排序。本程序的排序采用

逐个比较的方法进行。在 i 次循环时, 把第一个元素的下标 i 赋于 p , 而把该下标变量值 $a[i]$ 赋于 q 。然后进入小循环, 从 $a[i+1]$ 起到最后一个元素止逐个与 $a[i]$ 作比较, 有比 $a[i]$ 大者则将其下标送 p , 元素值送 q 。一次循环结束后, p 即为最大元素的下标, q 则为该元素值。若此时 $i \neq p$, 说明 p, q 值均已不是进入小循环之前所赋之值, 则交换 $a[i]$ 和 $a[p]$ 之值。此时 $a[i]$ 为已排序完毕的元素。输出该值之后转入下一次循环。对 $i+1$ 以后各个元素排序。



4.2 二维数组

定义

前面介绍的数组只有一个下标, 称为一维数组, 其数组元素也称为单下标变量。在实际问题中有很多量是二维的或多维的, 因此C语言允许构造多维数组。多维数组元素有多个下标, 以标识它在数组中的位置, 所以也称为多下标变量。本小节只介绍二维数组, 多维数组可由二维数组类推而得到。

二维数组定义的一般形式是:

1. 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)。

引用

二维数组的元素也称为双下标变量，其表示的形式为：

1. 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：

1. 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。

【例】一个学习小组有 5 个人，每个人有三门课的考试成绩。求全组分科的平均成绩和各科总平均成绩。

	张	王	李	赵	周
Math	80	61	59	85	76
C	75	65	63	87	77
Foxpro	92	71	70	90	85

可设一个二维数组 a[5][3]存放五个人三门课的成绩。再设一个一维数组 v[3]存放所求得各分科平均成绩，设变量 average 为全组各科总平均成绩。编程如下：

```
1. #include <stdio.h>
2.
3. int main() {
4.     int i, j, s = 0, average, v[3], a[5][3];
5.     printf("input score\n");
6.     for(i = 0; i < 3; i++) {
7.         for(j = 0; j < 5; j++) {
8.             scanf("%d", &a[j][i]);
9.             s = s + a[j][i];
10.        }
11.        v[i] = s / 5;
12.        s = 0;
13.    }
14.    average = (v[0] + v[1] + v[2]) / 3;
15.    printf("math:%d\nc language:%d\nbase:%d\n", v[0], v[1], v[2]);
16.    printf("total:%d\n", average );
```

```
17.     return 0;
18. }
```

程序中首先用了一个双重循环。在内循环中依次读入某一门课程各个学生的成绩，并把这些成绩累加起来，退出内循环后再把该累加成绩除以 5 送入 $v[i]$ 之中，这就是该门课程的平均成绩。外循环共循环三次，分别求出三门课各自的平均成绩并存放在 v 数组之中。退出外循环之后，把 $v[0], v[1], v[2]$ 相加除以 3 即得到各科总平均成绩。最后按题意输出各个成绩。

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。

例如对数组 $a[5][3]$:

1) 按行分段赋值可写为:

```
1. int a[5][3]={ {80,75,92},{61,65,71},{59,63,70},{85,87,90},{76,77,85} };
```

2) 按行连续赋值可写为:

```
1. int a[5][3]={ 80,75,92,61,65,71,59,63,70,85,87,90,76,77,85};
```

这两种赋初值的结果是完全相同的。

【例】

```
1. #include <stdio.h>
2.
3. int main() {
4.     int i, j, s = 0, average, v[3];
5.     int a[5][3] = {{80,75,92},{61,65,71},{59,63,70},{85,87,90},{76,77,85}};
6.     for(i = 0; i < 3; i++) {
7.         for(j = 0; j < 5; j++)
8.             s = s + a[j][i];
9.         v[i] = s / 5;
10.        s = 0;
11.    }
12.    average = (v[0] + v[1] + v[2]) / 3;
13.    printf("math:%d\nc language:%d\ndFoxpro:%d\n", v[0], v[1], v[2]);
```

```
14.     printf("total:%d\n", average);
15.     return 0;
16. }
```

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

赋值后的元素值为:

```
1.  0 1 0
2.  0 0 2
3.  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:

```
1.  a[0]
2.  a[1]
3.  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。

4.3 字符数组

定义

形式与前面介绍的数值数组相同。

例如：

```
char c[10];
```

由于字符型和整型通用，也可以定义为 `int c[10]`但这时每个数组元素占 2 个字节的内存单元。

字符数组也可以是二维或多维数组。

例如：

```
char c[5][10];
```

即为二维字符数组。

初始化

字符数组也允许在定义时作初始化赋值。

例如：

```
1. char c[10]={'c', ' ', 'p', 'r', 'o', 'g', 'r', 'a', 'm'};
```

赋值后各元素的值为：

数组 c

c[0]的值为 'c'

c[1]的值为 ' '

c[2]的值为 'p'

c[3]的值为 'r'

c[4]的值为 'o'

c[5]的值为 'g'

c[6]的值为 'r'

c[7]的值为 'a'

c[8]的值为 'm'

其中 c[9]未赋值，由的值为 'p' 系统自动赋予 0 值。

当对全体元素赋初值时也可以省去长度说明。

例如：

```
1. char c[]={ 'c', ' ', 'p', 'r', 'o', 'g', 'r', 'a', 'm' };
```

这时 c 数组的长度自动定为 9

引用

【例】

```
1. #include <stdio.h>
2.
3. int main() {
4.     int i, j;
5.     char a[][5] = {{ 'B', 'A', 'S', 'I', 'C' }, { 'd', 'B', 'A', 'S', 'E' }};
6.     for(i = 0; i <= 1; i++) {
7.         for(j = 0; j <= 4; j++)
8.             printf("%c", a[i][j]);
9.         printf("\n");
10.    }
11.    return 0;
12. }
```

本例的二维字符数组由于在初始化时全部元素都赋以初值，因此一维下标的长度可以不加说明。

字符串和字符串结束标志

在C语言中没有专门的字符串变量，通常用一个字符数组来存放一个字符串。前面介绍字符串常量时，已说明字符串总是以'\0'作为串的结束符。因此当把一个字符串存入一个数组时，也把结束符'\0'存入数组，并以此作为该字符串是否结束的标志。有了'\0'标志后，就不必再用字符数组的长度来判断字符串的长度了。

C语言允许用字符串的方式对数组作初始化赋值。

例如：

```
1. char c[]={'c', ' ', 'p', 'r', 'o', 'g', 'r', 'a', 'm'};
```

可写为:

```
1. char c[]={"C program"};
```

或去掉{}写为:

```
1. char c[]="C program";
```

用字符串方式赋值比用字符逐个赋值要多占一个字节, 用于存放字符串结束标志'\0'。上面的数组 c 在内存中的实际存放情况为:

C		p	r	o	g	r	a	m	\0
---	--	---	---	---	---	---	---	---	----

'\0'是由 C 编译系统自动加上的。由于采用了'\0'标志, 所以在用字符串赋初值时一般无须指定数组的长度, 而由系统自行处理。

字符数组的输入输出

在采用字符串方式后, 字符数组的输入输出将变得简单方便。

除了上述用字符串赋初值的办法外, 还可用 printf 函数和 scanf 函数一次性输出输入一个字符数组中的字符串, 而不必使用循环语句逐个地输入输出每个字符。

【例】

```
1. #include <stdio.h>
2.
3. int main() {
4.     char c[] = "BASIC\ndBASE";
5.     printf("%s\n", c);
6.     return 0;
7. }
```

注意在本例的 printf 函数中, 使用的格式字符串为“%s”, 表示输出的是一个字符串。而在输出表列中给出数组名则可。不能写为:

```
1. printf("%s", c[]);
```

【例】

```
1. #include <stdio.h>
2.
3. int main() {
4.     char st[15];
```



```
5.    printf("input string:\n");
6.    scanf("%s", st);
7.    printf("%s\n", st);
8.    return 0;
9. }
```

本例中由于定义数组长度为 15，因此输入的字符串长度必须小于 15，以留出一个字节用于存放字符串结束标志`\0`。应该说明的是，对一个字符数组，如果不作初始化赋值，则必须说明数组长度。还应该特别注意的是，当用 `scanf` 函数输入字符串时，字符串中不能含有空格，否则将以空格作为串的结束符。

例如当输入的字符串中含有空格时，运行情况为：

```
1. input string:
2. this is a book
```

输出为：

```
1. this
```

从输出结果可以看出空格以后的字符都未能输出。为了避免这种情况，可多设几个字符数组分段存放含空格的串。

程序可改写如下：

【例】

```
1. #include <stdio.h>
2.
3. int main() {
4.     char st1[6], st2[6], st3[6], st4[6];
5.     printf("input string:\n");
6.     scanf("%s%s%s%s", st1, st2, st3, st4);
7.     printf("%s %s %s %s\n", st1, st2, st3, st4);
8.     return 0;
9. }
```

本程序分别设了四个数组，输入的一行字符的空格分段分别装入四个数组。然后分别输出这四个数组中的字符串。

在前面介绍过，`scanf` 的各输入项必须以地址方式出现，如 `&a,&b` 等。但在前例中却是以数组名方式出现的，这是为什么呢？

这是由于在 C 语言中规定，数组名就代表了该数组的首地址。整个数组是以首地址开头的一块连续的内存单元。

如有字符数组 `char c[10]`，在内存可表示如图。

c[0]	c[1]	c[2]	c[3]	c[4]	c[5]	c[6]	c[7]	c[8]	c[9]
------	------	------	------	------	------	------	------	------	------

设数组 `c` 的首地址为 2000，也就是说 `c[0]` 单元地址为 2000。则数组名 `c` 就代表这个首地址。因此在 `c` 前面不能再加地址运算符 `&`。如写作 `scanf("%s",&c);` 则是错误的。在执行函数 `printf("%s",c)` 时，按数组名 `c` 找到首地址，然后逐个输出数组中各个字符直到遇到字符串终止标志 `'\0'` 为止。

使用字符串处理函数

C 语言提供了丰富的字符串处理函数，大致可分为字符串的输入、输出、合并、修改、比较、转换、复制、搜索几类。使用这些函数可大大减轻编程的负担。用于输入输出的字符串函数，在使用前应包含头文件 `"stdio.h"`，使用其它字符串函数则应包含头文件 `"string.h"`。

下面介绍几个最常用的字符串函数。

1. 字符串输出函数 `puts`

格式： `puts (字符数组名)`

功能：把字符数组中的字符串输出到显示器。即在屏幕上显示该字符串。

【例】

```
1. #include <stdio.h>
2.
3. int main() {
4.     char c[] = "BASIC\ndBASE";
5.     puts(c);
6.     return 0;
7. }
```

从程序中可以看出 `puts` 函数中可以使用转义字符，因此输出结果成为两行。`puts` 函数完全可以由 `printf` 函数取代。当需要按一定格式输出时，通常使用 `printf` 函数。

2. 字符串输入函数 `gets`

格式： `gets (字符数组名)`

功能：从标准输入设备键盘上输入一个字符串。

本函数得到一个函数值，即为该字符数组的首地址。

【例】

```
1. #include <stdio.h>
2.
3. int main() {
4.     char st[15];
5.     printf("input string:\n");
6.     gets(st);
7.     puts(st);
8.     return 0;
9. }
```

可以看出当输入的字符串中含有空格时，输出仍为全部字符串。说明 `gets` 函数并不以空格作为字符串输入结束的标志，而只以回车作为输入结束。这是与 `scanf` 函数不同的。

3. 字符串连接函数 `strcat`

格式：`strcat` (字符数组名 1, 字符数组名 2)

功能：把字符数组 2 中的字符串连接到字符数组 1 中字符串的后面，并删去字符串 1 后的串标志“\0”。

本函数返回值是字符数组 1 的首地址。

【例】

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     static char st1[30] = "My name is ";
6.     char st2[10];
7.     printf("input your name:\n");
8.     gets(st2);
9.     strcat(st1, st2);
10.    puts(st1);
11.    return 0;
12. }
```

本程序把初始化赋值的字符数组与动态赋值的字符串连接起来。要注意的是，字符数组 1

应定义足够的长度，否则不能全部装入被连接的字符串。

4. 字符串拷贝函数 strcpy

格式：strcpy(字符数组名 1, 字符数组名 2)

功能：把字符数组 2 中的字符串拷贝到字符数组 1 中。串结束标志“\0”也一同拷贝。字符数组名 2，也可以是一个字符串常量。这时相当于把一个字符串赋予一个字符数组。

【例】

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     char st1[15], st2[] = "C Language";
6.     strcpy(st1, st2);
7.     puts(st1); printf("\n");
8.     return 0;
9. }
```

本函数要求字符数组 1 应有足够的长度，否则不能全部装入所拷贝的字符串。

5. 字符串比较函数 strcmp

格式：strcmp(字符数组名 1, 字符数组名 2)

功能：按照 ASCII 码顺序比较两个数组中的字符串，并由函数返回值返回比较结果。

字符串 1 = 字符串 2，返回值 = 0；

字符串 1 > 字符串 2，返回值 > 0；

字符串 1 < 字符串 2，返回值 < 0。

本函数也可用于比较两个字符串常量，或比较数组和字符串常量。

【例】

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     int k;
```

```

6.     static char st1[15], st2[] = "C Language";
7.     printf("input a string:\n");
8.     gets(st1);
9.     k = strcmp(st1,st2);
10.    if(k == 0) printf("st1=st2\n");
11.    if(k > 0) printf("st1>st2\n");
12.    if(k < 0) printf("st1<st2\n");
13.    return 0;
14. }

```

本程序中把输入的字符串和数组 st2 中的串比较，比较结果返回到 k 中，根据 k 值再输出结果提示串。当输入为 dbase 时，由 ASCII 码可知“dBASE”大于“C Language”故 $k > 0$ ，输出结果“st1>st2”。

6. 测字符串长度函数 strlen

格式：strlen(字符数组名)

功能：测字符串的实际长度(不含字符串结束标志‘\0’) 并作为函数返回值。

【例】

```

1.  #include <stdio.h>
2.  #include <string.h>
3.
4.  int main() {
5.      int k;
6.      static char st[] = "C language";
7.      k = strlen(st);
8.      printf("The length of the string is %d\n", k);
9.      return 0;
10. }

```

应用

【例】把一个整数按大小顺序插入已排好序的数组中。

为了把一个数按大小插入已排好序的数组中，应首先确定排序是从大到小还是从小到大进行的。设排序是从大到小进序的，则可把欲插入的数与数组中各数逐个比较，当找到第一个比插入数小的元素 i 时，该元素之前即为插入位置。然后从数组最后一个元素开始到该元素为止，

逐个后移一个单元。最后把插入数赋予元素 i 即可。如果被插入数比所有的元素值都小则插入最后位置。

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     int i, j, p, q, s, n;
6.     int a[11] = {127,3,6,28,54,68,87,105,162,18};
7.     for(i = 0; i < 10; i++) {
8.         p = i; q = a[i];
9.         for(j = i + 1; j < 10; j++)
10.            if(q < a[j]) {
11.                p = j; q = a[j];
12.            }
13.         if(p != i) {
14.             s = a[i];
15.             a[i] = a[p];
16.             a[p] = s;
17.         }
18.         printf("%d ",a[i]);
19.     }
20.     printf("\ninput number:\n");
21.     scanf("%d", &n);
22.     for(i = 0; i < 10; i++)
23.         if(n > a[i]) {
24.             for(s = 9; s >= i; s--)
25.                 a[s+1] = a[s];
26.             break;
27.         }
28.     a[i] = n;
29.     for(i = 0; i <= 10; i++)
30.         printf("%d ", a[i]);
31.     printf("\n");
32.     return 0;
33. }
```

本程序首先对数组 a 中的 10 个数从大到小排序并输出排序结果。然后输入要插入的整数 n 。再用一个 `for` 语句把 n 和数组元素逐个比较, 如果发现有 $n > a[i]$ 时, 则由一个内循环把 i 以下各元素值顺次后移一个单元。后移应从后向前进行(从 $a[9]$ 开始到 $a[i]$ 为止)。后移结束跳出外循环。插入点为 i , 把 n 赋予 $a[i]$ 即可。如所有的元素均大于被插入数, 则并未进行过后

移工作。此时 $i=10$ ，结果是把 n 赋予 $a[10]$ 。最后一个循环输出插入数后的数组各元素值。程序运行时，输入数 47。从结果中可以看出 47 已插入到 54 和 28 之间。

【例】在二维数组 a 中选出各行最大的元素组成一个一维数组 b 。

```
1. a=( 3 16 87 65
2.    4 32 11 108
3.    10 25 12 37)
4. b=(87 108 37)
```

本题的编程思路是，在数组 A 的每一行中寻找最大的元素，找到之后把该值赋予数组 B 相应的元素即可。

程序如下：

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     int a[][4] = {3,16,87,65,4,32,11,108,10,25,12,27};
6.     int b[3], i, j, l;
7.     for(i = 0; i <= 2; i++) {
8.         l = a[i][0];
9.         for(j = 1; j <= 3; j++)
10.            if(a[i][j] > l) l = a[i][j];
11.        b[i] = l;
12.    }
13.    printf("array a:\n");
14.    for(i = 0; i <= 2; i++) {
15.        for(j = 0; j <= 3; j++)
16.            printf("%5d", a[i][j]);
17.        printf("\n");
18.        printf("\narray b:\n");
19.        for(i = 0; i <= 2; i++)
20.            printf("%5d", b[i]);
21.        printf("\n");
22.    return 0;
23. }
```

程序中第一个 `for` 语句中又嵌套了一个 `for` 语句组成了双重循环。外循环控制逐行处理，并把每行的第 0 列元素赋予 l 。进入内循环后，把 l 与后面各列元素比较，并把比 l 大者赋予 l 。内循环结束时 l 即为该行最大的元素，然后把 l 值赋予 $b[i]$ 。等外循环全部完成时，数组 b 中已装入了 a 各行中的最大值。后面的两个 `for` 语句分别输出数组 a 和数组 b 。

【例】输入五个国家的名称按字母顺序排列输出。

本题编程思路如下：五个国家名应由一个二维字符数组来处理。然而C语言规定可以把一个二维数组当成多个一维数组处理。因此本题又可以按五个一维数组处理，而每一个一维数组就是一个国家名字符串。用字符串比较函数比较各一维数组的大小，并排序，输出结果即可。

编程如下：

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     char st[20], cs[5][20];
6.     int i, j, p;
7.     printf("input country's name:\n");
8.     for(i = 0; i < 5; i++)
9.         gets(cs[i]);
10.    printf("\n");
11.    for(i = 0; i < 5; i++) {
12.        p = i; strcpy(st, cs[i]);
13.        for(j = i+1; j < 5; j++)
14.            if(strcmp(cs[j], st) < 0) {
15.                p = j;
16.                strcpy(st, cs[j]);
17.            }
18.        if(p != i) {
19.            strcpy(st, cs[i]);
20.            strcpy(cs[i], cs[p]);
21.            strcpy(cs[p], st);
22.        }
23.        puts(cs[i]);
24.    }
25.    printf("\n");
26.    return 0;
27. }
```

本程序的第一个 for 语句中，用 gets 函数输入五个国家名字符串。上面说过C语言允许把一个二维数组按多个一维数组处理，本程序说明 cs[5][20]为二维字符数组，可分为五个一维数组 cs[0], cs[1], cs[2], cs[3], cs[4]。因此在 gets 函数中使用 cs[i]是合法的。在第二个 for 语句中又嵌套了一个 for 语句组成双重循环。这个双重循环完成按字母顺序排序的工作。在外层循环中把字符数组 cs[i]中的国名字符串拷贝到数组 st 中，并把下标 i 赋予 P。进入

内层循环后, 把 `st` 与 `cs[i]` 以后的各字符串作比较, 若有比 `st` 小者则把该字符串拷贝到 `st` 中, 并把其下标赋予 `p`。内循环完成后如 `p` 不等于 `i` 说明有比 `cs[i]` 更小的字符串出现, 因此交换 `cs[i]` 和 `st` 的内容。至此已确定了数组 `cs` 的第 `i` 号元素的排序值。然后输出该字符串。在外循环全部完成之后即完成全部排序和输出。



4.4 本章小结

易错点汇总

- 1.数组是程序设计中最常用的数据结构。数组可分为数值数组(整数组, 实数组), 字符数组以及后面将要介绍的指针数组, 结构数组等。
- 2.数组可以是一维的, 二维的或多维的。
- 3.数组类型说明由类型说明符、数组名、数组长度(数组元素个数)三部分组成。数组元素又称为下标变量。数组的类型是指下标变量取值的类型。
- 4.对数组的赋值可以用数组初始化赋值, 输入函数动态赋值和赋值语句赋值三种方法实现。对数值数组不能用赋值语句整体赋值、输入或输出, 而必须用循环语句逐个对数组元素进行操作。

实战练习

DreamJudge 1006 字符串翻转
DreamJudge 1033 细菌繁殖
DreamJudge 1083 简单的求和
DreamJudge 1062 杨辉三角形
DreamJudge 1473 字符菱形
DreamJudge 1577 画长方形 1
DreamJudge 1578 画长方形 2
DreamJudge 1377 旋转矩阵 - 北航
DreamJudge 1216 旋转方阵

第五章 函数

【本章知识点汇总】

函数的定义

函数的调用

函数的声明

函数的嵌套调用

函数的递归调用

数组作为函数参数

局部变量和全局变量

变量的存储方式

关于变量的声明和定义

内部函数和外部函数

本书配套视频精讲: <https://www.bilibili.com/video/BV1HJ41137fe>

5.1 为什么要用函数

在前面已经介绍过，C 源程序是由函数组成的。虽然在前面各章的程序中大都只有一个主函数 `main()`，但实用程序往往由多个函数组成。函数是 C 源程序的基本模块，通过对函数模块的调用实现特定的功能。C 语言中的函数相当于其它高级语言的子程序。C 语言不仅提供了极为丰富的库函数(如 Turbo C，MS C 都提供了三百多个库函数)，还允许用户建立自己定义的函数。用户可把自己的算法编成一个个相对独立的函数模块，然后用调用的方法来使用函数。可以说 C 程序的全部工作都是由各式各样的函数完成的，所以也把 C 语言称为**函数式语言**。

由于采用了函数模块式的结构，C 语言易于实现结构化程序设计。使程序的层次结构清晰，便于程序的编写、阅读、调试。

在 C 语言中可从不同的角度对函数分类。

1. 从函数定义的角度看，函数可分为库函数和用户定义函数两种。

1) 库函数：由 C 系统提供，用户无须定义，也不必在程序中作类型说明，只需在程序前包含有该函数原型的头文件即可在程序中直接调用。在前面各章的例题中反复用到 `printf`、`scanf`、`getchar`、`putchar`、`gets`、`puts`、`strcat` 等函数均属此类。

2) 用户定义函数：由用户按需要写的函数。对于用户自定义函数，不仅要在程序中定义函数本身，而且在主调函数模块中还必须对该被调函数进行类型说明，然后才能使用。

2. C 语言的函数兼有其它语言中的函数和过程两种功能，从这个角度看，又可将函数分为有返回值函数和无返回值函数两种。

1) 有返回值函数：此类函数被调用执行完后将向调用者返回一个执行结果，称为函数返回值。如数学函数即属于此类函数。由用户定义的这种要返回函数值的函数，必须在函数定义和函数说明中明确返回值的类型。

2) 无返回值函数：此类函数用于完成某项特定的处理任务，执行完成后不向调用者返回函数值。这类函数类似于其它语言的过程。由于函数无须返回值，用户在定义此类函数时可指定它的返回为“空类型”，空类型的说明符为“`void`”。

3. 从主调函数和被调函数之间数据传送的角度看又可分为无参函数和有参函数两种。

1) 无参函数：函数定义、函数说明及函数调用中均不带参数。主调函数和被调函数之间不进行参数传送。此类函数通常用来完成一组指定的功能，可以返回或不返回函数值。

2) 有参函数: 也称为带参函数。在函数定义及函数说明时都有参数, 称为形式参数(简称为形参)。在函数调用时也必须给出参数, 称为实际参数(简称为实参)。进行函数调用时, 主调函数将把实参的值传送给形参, 供被调函数使用。

4. C 语言提供了极为丰富的库函数, 这些库函数又可从功能角度作以下分类。

1) 字符类型分类函数: 用于对字符按 ASCII 码分类: 字母, 数字, 控制字符, 分隔符, 大小写字母等。

2) 转换函数: 用于字符或字符串的转换; 在字符量和各类数字量(整型, 实型等)之间进行转换; 在大、小写之间进行转换。

3) 目录路径函数: 用于文件目录和路径操作。

4) 诊断函数: 用于内部错误检测。

5) 图形函数: 用于屏幕管理和各种图形功能。

6) 输入输出函数: 用于完成输入输出功能。

7) 接口函数: 用于与 DOS, BIOS 和硬件的接口。

8) 字符串函数: 用于字符串操作和处理。

9) 内存管理函数: 用于内存管理。

10) 数学函数: 用于数学函数计算。

11) 日期和时间函数: 用于日期, 时间转换操作。

12) 进程控制函数: 用于进程管理和控制。

13) 其它函数: 用于其它各种功能。

以上各类函数不仅数量多, 而且有的还需要硬件知识才会使用, 因此要想全部掌握则需要一个较长的学习过程。应首先掌握一些最基本、最常用的函数, 再逐步深入。由于课时关系, 我们只介绍了很少一部分库函数, 其余部分读者可根据需要查阅有关手册。

还应该指出的是, 在 C 语言中, 所有的函数定义, 包括主函数 `main` 在内, 都是平行的。也就是说, 在一个函数的函数体内, 不能再定义另一个函数, 即不能嵌套定义。但是函数之间允许相互调用, 也允许嵌套调用。习惯上把调用者称为主调函数。函数还可以自己调用自己, 称为递归调用。`main` 函数是主函数, 它可以调用其它函数, 而不允许被其它函数调用。因此, C 程序的执行总是从 `main` 函数开始, 完成对其它函数的调用后再返回到 `main` 函数, 最后由 `main` 函数结束整个程序。一个 C 源程序必须有, 也只能有一个主函数 `main`。

5.2 如何定义函数

1、无参函数的定义形式

1. 类型标识符 函数名()
2. {
3. 声明部分
4. 语句
5. }

其中类型标识符和函数名称为函数头。类型标识符指明了本函数的类型，函数的类型实际上是函数返回值的类型。 该类型标识符与前面介绍的各种说明符相同。函数名是由用户定义的标识符，函数名后有一个空括号，其中无参数，但括号不可少。

{ } 中的内容称为函数体。在函数体中声明部分，是对函数体内部所用到的变量的类型说明。在很多情况下都不要求无参函数有返回值，此时函数类型符可以写为 `void`。

我们可以改写一个函数定义：

1. `void Hello()`
2. {
3. `printf ("Hello,world \n");`
4. }

这里，只把 `main` 改为 `Hello` 作为函数名，其余不变。`Hello` 函数是一个无参函数，当被其它函数调用时，输出 `Hello world` 字符串。

2、有参函数定义的一般形式

1. 类型标识符 函数名(形式参数表列)
2. {
3. 声明部分
4. 语句
5. }

有参函数比无参函数多了一个内容，即形式参数表列。在形参表中给出的参数称为形式参数，它们可以是各种类型的变量，各参数之间用逗号间隔。在进行函数调用时，主调函数将赋予这些形式参数实际的值。形参既然是变量，必须在形参表中给出形参的类型说明。

例如，定义一个函数，用于求两个数中的大数，可写为：

1. `int max(int a, int b)`

```
2. {
3.     if (a > b) return a;
4.     else return b;
5. }
```

第一行说明 `max` 函数是一个整型函数，其返回的函数值是一个整数。形参为 `a,b`,均为整型量。`a,b` 的具体值是由主调函数在调用时传送过来的。在 `{}` 中的函数体内，除形参外没有使用其它变量，因此只有语句而没有声明部分。在 `max` 函数体中的 `return` 语句是把 `a`(或 `b`) 的值作为函数的值返回给主调函数。有返回值函数中至少应有一个 `return` 语句。在 C 程序中，一个函数的定义可以放在任意位置，既可放在主函数 `main` 之前，也可放在 `main` 之后。

例如：可把 `max` 函数置在 `main` 之后，也可以把它放在 `main` 之前。修改后的程序如下所示。

```
1. #include <stdio.h>
2.
3. int max(int a, int b) {
4.     if(a > b) return a;
5.     else return b;
6. }
7.
8. int main() {
9.     int max(int a, int b);
10.    int x, y, z;
11.    printf("input two numbers:\n");
12.    scanf("%d%d", &x, &y);
13.    z = max(x, y);
14.    printf("maxum=%d", z);
15.    return 0;
16. }
```

现在我们可以从函数定义、函数说明及函数调用的角度来分析整个程序，从中进一步了解函数的各种特点。程序的第 3 行至第 6 行为 `max` 函数定义。进入主函数后，因为准备调用 `max` 函数，故先对 `max` 函数进行说明(程序第 9 行)。函数定义和函数说明并不是一回事，在后面还要专门讨论。可以看出函数说明与函数定义中的函数头部分相同，但是末尾要加分号。程序第 13 行为调用 `max` 函数，并把 `x,y` 中的值传送给 `max` 的形参 `a,b`。`max` 函数执行的结果(`a` 或 `b`)将返回给变量 `z`。最后由主函数输出 `z` 的值。

5.3 调用函数

前面已经说过，在程序中是通过函数的调用来执行函数体的，其过程与其它语言的子程序调用相似。C语言中，函数调用的一般形式为：

1. 函数名(实际参数表)

对无参函数调用时则无实际参数表。实际参数表中的参数可以是常数，变量或其它构造类型数据及表达式。各实参之间用逗号分隔。

函数调用的形式

在C语言中，可以用以下几种方式调用函数：

1. 函数表达式：函数作为表达式中的一项出现在表达式中，以函数返回值参与表达式的运算。这种方式要求函数是有返回值的。例如： $z=\max(x,y)$ 是一个赋值表达式，把 \max 的返回值赋予变量 z 。

2. 函数语句：函数调用的一般形式加上分号即构成函数语句。例如： $\text{printf}("%d",a);\text{scanf}("%d",&b);$ 都是以函数语句的方式调用函数。

3. 函数实参：函数作为另一个函数调用的实际参数出现。这种情况是把该函数的返回值作为实参进行传送，因此要求该函数必须是有返回值的。例如： $\text{printf}("%d",\max(x,y));$ 即是把 \max 调用的返回值又作为 printf 函数的实参来使用的。在函数调用中还应该注意的一个问题是求值顺序的问题。所谓求值顺序是指对实参表中各量是自左至右使用呢，还是自右至左使用。对此，各系统的规定不一定相同。介绍 printf 函数时已提到过，这里从函数调用的角度再强调一下。

【例】

```
1. #include <stdio.h>
2.
3. int main() {
4.     int i=8;
5.     printf("%d\n%d\n%d\n%d\n",++i,--i,i++,i--);
6.     return 0;
7. }
```

如按照从右至左的顺序求值。运行结果应为：

8
7
7
8

如对 `printf` 语句中的 `++i`, `--i`, `i++`, `i--` 从左至右求值，结果应为：

9
8
8
9

应特别注意的是，无论是从左至右求值，还是自右至左求值，其输出顺序都是不变的，即输出顺序总是和实参表中实参的顺序相同。由于编译器规定是自右至左求值，所以结果为 8, 7, 7, 8。上述问题如还不理解，上机一试就明白了。

函数调用时的数据传递

形式参数和实际参数

前面已经介绍过，函数的参数分为形参和实参两种。在本小节中，进一步介绍形参、实参的特点和两者的关系。形参出现在函数定义中，在整个函数体内都可以使用，离开该函数则不能使用。实参出现在主调函数中，进入被调函数后，实参变量也不能使用。形参和实参的功能是作数据传送。发生函数调用时，主调函数把实参的值传送给被调函数的形参从而实现主调函数向被调函数的数据传送。

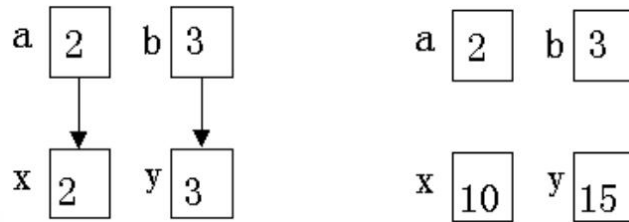
函数的形参和实参具有以下特点：

1. 形参变量只有在被调用时才分配内存单元，在调用结束时，即刻释放所分配的内存单元。因此，形参只有在函数内部有效。函数调用结束返回主调函数后则不能再使用该形参变量。
2. 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，以便把这些值传送给形参。因此应预先用赋值，输入等办法使实参获得确定值。

3. 实参和形参在数量上，类型上，顺序上应严格一致，否则会发生类型不匹配”的错误。

4. 函数调用中发生的数据传送是单向的。即只能把实参的值传送给形参，而不能把形参的值反向地传送给实参。因此在函数调用过程中，形参的值发生改变，而实参中的值不会变化。



下面的例子可以说明这个问题。

```
1. #include <stdio.h>
2.
3. int s(int n) {
4.     int i;
5.     for(i = n - 1; i >= 1; i--)
6.         n = n + i;
7.     printf("n=%d\n", n);
8. }
9.
10. int main() {
11.     int n;
12.     printf("input number\n");
13.     scanf("%d", &n);
14.     s(n);
15.     printf("n=%d\n", n);
16.     return 0;
17. }
```

本程序中定义了一个函数 `s`，该函数的功能是求 1 到 `n` 的和 的值。在主函数中输入 `n` 值，并作为实参，在调用时传送给 `s` 函数的形参量 `n` (注意，本例的形参变量和实参变量的标识符都为 `n`，但这是两个不同的量，各自的作用域不同)。在主函数中用 `printf` 语句输出一次 `n` 值，这个 `n` 值是实参 `n` 的值。在函数 `s` 中也用 `printf` 语句输出了一次 `n` 值，这个 `n` 值是形参最后取得的 `n` 值 0。从运行情况看，输入 `n` 值为 100。即实参 `n` 的值为 100。把此值传给函数 `s` 时，形参 `n` 的初值也为 100，在执行函数过程中，形参 `n` 的值变为 5050。返回主函数之后，输出实参 `n` 的值仍为 100。可见实参的值不随形参的变化而变化。

函数的返回值

函数的值是指函数被调用之后, 执行函数体中的程序段所取得的并返回给主调函数的值。如调用正弦函数取得正弦值, 调用例子的 `max` 函数取得的最大数等。对函数的值(或称函数返回值)有以下一些说明:

1) 函数的值只能通过 `return` 语句返回主调函数。

`return` 语句的一般形式为:

1. `return` 表达式;

或者为:

1. `return` (表达式);

该语句的功能是计算表达式的值, 并返回给主调函数。在函数中允许有多个 `return` 语句, 但每次调用只能有一个 `return` 语句被执行, 因此只能返回一个函数值。

2) 函数值的类型和函数定义中函数的类型应保持一致。如果两者不一致, 则以函数类型为准, 自动进行类型转换。

3) 如函数值为整型, 在函数定义时可以省去类型说明。

4) 不返回函数值的函数, 可以明确定义为“空类型”, 类型说明符为“`void`”。如例子中函数 `s` 并不向主函数返回函数值, 因此可定义为:

```
1. void s(int n)
2. {
3.     .....
4. }
```

一旦函数被定义为空类型后, 就不能在主调函数中使用被调函数的函数值了。例如, 在定义 `s` 为空类型后, 在主函数中写下述语句

1. `sum=s(n);`

就是错误的。

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空类型。

5.4 对被调用函数的声明和函数原型

在主调函数中调用某函数之前应对该被调函数进行说明（声明），这与使用变量之前要先进行变量说明是一样的。在主调函数中对被调函数作说明的目的是使编译系统知道被调函数返回值的类型，以便在主调函数中按此种类型对返回值作相应的处理。

其一般形式为：

1. 类型说明符 被调函数名(类型 形参, 类型 形参...);

或为：

1. 类型说明符 被调函数名(类型, 类型...);

括号内给出了形参的类型和形参名，或只给出形参类型。这便于编译系统进行检错，以防止可能出现的错误。

例 main 函数中对 max 函数的说明为：

1. `int max(int a, int b);`

或写为：

1. `int max(int, int);`

C语言中又规定在以下几种情况时可以省去主调函数中对被调函数的函数说明。

1) 如果被调函数的返回值是整型或字符型时，可以不对被调函数作说明，而直接调用。这时系统将自动对被调函数返回值按整型处理。样例中的主函数中未对函数 s 作说明而直接调用即属此种情形。

2) 当被调函数的函数定义出现在主调函数之前时，在主调函数中也可以不对被调函数再作说明而直接调用。例如前面的样例中，函数 max 的定义放在 main 函数之前，因此可在 main 函数中省去对 max 函数的函数说明 `int max(int a, int b)`。

3) 如在所有函数定义之前，在函数外预先说明了各个函数的类型，则在以后的各主调函数中，可不再对被调函数作说明。例如：

1. `char str(int a);`
2. `float f(float b);`

```
3.  main()
4.  {
5.      .....
6.  }
7.  char str(int a)
8.  {
9.      .....
10. }
11. float f(float b)
12. {
13.     .....
14. }
```

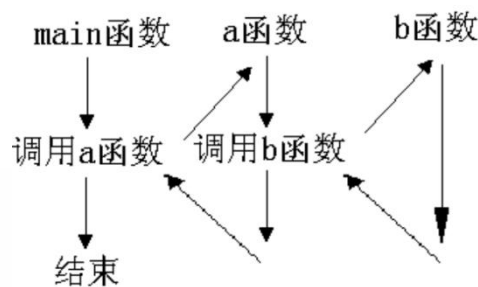
其中第一，二行对 `str` 函数和 `f` 函数预先作了说明。因此在以后各函数中无须对 `str` 和 `f` 函数再作说明就可直接调用。

4) 对库函数的调用不需要再作说明，但必须把该函数的头文件用 `include` 命令包含在源文件前部。

noobdream.com

5.5 函数的嵌套调用

C语言中不允许作嵌套的函数定义。因此各函数之间是平行的，不存在上一级函数和下一级函数的问题。但是C语言允许在一个函数的定义中出现对另一个函数的调用。这样就出现了函数的嵌套调用。即在被调函数中又调用其它函数。这与其它语言的子程序嵌套的情形是类似的。其关系可表示如图。



图表示了两层嵌套的情形。其执行过程是：执行 main 函数中调用 a 函数的语句时，即转去执行 a 函数，在 a 函数中调用 b 函数时，又转去执行 b 函数，b 函数执行完毕返回 a 函数的断点继续执行，a 函数执行完毕返回 main 函数的断点继续执行。

【例 8.4】计算 $s = 2^2! + 3^2!$

本题可编写两个函数，一个是用来计算平方值的函数 f1，另一个是用来计算阶乘值的函数 f2。主函数先调 f1 计算出平方值，再在 f1 中以平方值为实参，调用 f2 计算其阶乘值，然后返回 f1，再返回主函数，在循环程序中计算累加和。

```

1. #include <stdio.h>
2.
3. long f1(int p) {
4.     int k;
5.     long r;
6.     long f2(int);
7.     k = p * p;
8.     r = f2(k);
9.     return r;
10. }
11.
12. long f2(int q) {
13.     long c = 1;
14.     int i;
15.     for(i = 1; i <= q; i++)
  
```

```
16.         c = c * i;
17.     return c;
18. }
19.
20. int main() {
21.     int i;
22.     long s = 0;
23.     for (i = 2; i <= 3; i++)
24.         s = s + f1(i);
25.     printf("s=%ld\n", s);
26.     return 0;
27. }
```

在程序中，函数 f1 和 f2 均为长整型，都在主函数之前定义，故不必再在主函数中对 f1 和 f2 加以说明。在主程序中，执行循环程序依次把 i 值作为实参调用函数 f1 求 i^2 值。在 f1 中又发生对函数 f2 的调用，这时是把 i^2 的值作为实参去调 f2，在 f2 中完成求 $i^2!$ 的计算。f2 执行完毕把 C 值(即 $i^2!$)返回给 f1，再由 f1 返回主函数实现累加。至此，由函数的嵌套调用实现了题目的要求。由于数值很大，所以函数和一些变量的类型都说明为长整型，否则会造成计算错误。

noobdream.com

5.6 函数的递归调用

一个函数在它的函数体内调用它自身称为递归调用。这种函数称为递归函数。C语言允许函数的递归调用。在递归调用中，主调函数又是被调函数。执行递归函数将反复调用其自身，每调用一次就进入新的一层。

例如有函数 f 如下：

```
1. int f(int x)
2. {
3.     int y;
4.     z = f(y);
5.     return z;
6. }
```

这个函数是一个递归函数。但是运行该函数将无休止地调用其自身，这当然是不正确的。为了防止递归调用无终止地进行，必须在函数内有终止递归调用的手段。常用的办法是加条件判断，满足某种条件后就不再作递归调用，然后逐层返回。下面举例说明递归调用的执行过程。

【例】用递归法计算 $n!$

用递归法计算 $n!$ 可用下述公式表示：

$$n! = 1 \quad (n = 0, 1)$$

$$n \times (n - 1)! \quad (n > 1)$$

按公式可编程如下：

```
1. #include <stdio.h>
2.
3. long ff(int n) {
4.     long f;
5.     if(n < 0) printf("n<0,input error");
6.     else if(n == 0 || n == 1) f = 1;
7.     else f = ff(n - 1) * n;
8.     return(f);
9. }
10.
11. int main() {
12.     int n;
13.     long y;
14.     printf("input a inteager number:\n");
15.     scanf("%d", &n);
```



```
16.    y = ff(n);
17.    printf("%d!=%ld", n, y);
18.    return 0;
19. }
```

程序中给出的函数 `ff` 是一个递归函数。主函数调用 `ff` 后即进入函数 `ff` 执行, 如果 $n < 0, n == 0$ 或 $n = 1$ 时都将结束函数的执行, 否则就递归调用 `ff` 函数自身。由于每次递归调用的实参为 $n-1$, 即把 $n-1$ 的值赋予形参 n , 最后当 $n-1$ 的值为 1 时再作递归调用, 形参 n 的值也为 1, 将使递归终止。然后可逐层退回。

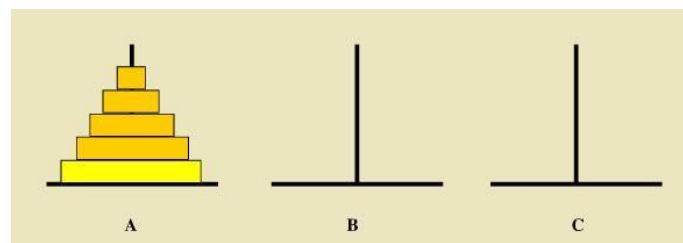
下面我们再举例说明该过程。设执行本程序时输入为 5, 即求 $5!$ 。在主函数中的调用语句即为 `y=ff(5)`, 进入 `ff` 函数后, 由于 $n=5$, 不等于 0 或 1, 故应执行 `f=ff(n-1)*n`, 即 `f=ff(5-1)*5`。该语句对 `ff` 作递归调用即 `ff(4)`。

进行四次递归调用后, `ff` 函数形参取得的值变为 1, 故不再继续递归调用而开始逐层返回主调函数。`ff(1)`的函数返回值为 1, `ff(2)`的返回值为 $1*2=2$, `ff(3)`的返回值为 $2*3=6$, `ff(4)`的返回值为 $6*4=24$, 最后返回值 `ff(5)`为 $24*5=120$ 。

上例也可以不用递归的方法来完成。如可以用递推法, 即从 1 开始乘以 2, 再乘以 3... 直到 n 。递推法比递归法更容易理解和实现。但是有些问题则只能用递归算法才能实现。典型的问题是 Hanoi 塔问题。

【例】Hanoi 塔问题

一块板上有三根针, A, B, C。A 针上套有 64 个大小不等的圆盘, 大的在下, 小的在上。如图所示。要把这 64 个圆盘从 A 针移动 C 针上, 每次只能移动一个圆盘, 移动可以借助 B 针进行。但在任何时候, 任何针上的圆盘都必须保持大盘在下, 小盘在上。求移动的步骤。



本题算法分析如下, 设 A 上有 n 个盘子。

如果 $n=1$, 则将圆盘从 A 直接移动到 C。

如果 $n=2$, 则:

1. 将 A 上的 $n-1$ (等于 1)个圆盘移到 B 上;
2. 再将 A 上的一个圆盘移到 C 上;
3. 最后将 B 上的 $n-1$ (等于 1)个圆盘移到 C 上。

如果 $n=3$, 则:

A. 将 A 上的 $n-1$ (等于 2, 令其为 n')个圆盘移到 B(借助于 C), 步骤如下:

- (1) 将 A 上的 $n'-1$ (等于 1)个圆盘移到 C 上。
- (2) 将 A 上的一个圆盘移到 B。
- (3) 将 C 上的 $n'-1$ (等于 1)个圆盘移到 B。

B. 将 A 上的一个圆盘移到 C。

C. 将 B 上的 $n-1$ (等于 2, 令其为 n')个圆盘移到 C(借助 A), 步骤如下:

- (1) 将 B 上的 $n'-1$ (等于 1)个圆盘移到 A。
- (2) 将 B 上的一个盘子移到 C。
- (3) 将 A 上的 $n'-1$ (等于 1)个圆盘移到 C。

到此, 完成了三个圆盘的移动过程。

从上面分析可以看出, 当 n 大于等于 2 时, 移动的过程可分解为三个步骤:

第一步 把 A 上的 $n-1$ 个圆盘移到 B 上;

第二步 把 A 上的一个圆盘移到 C 上;

第三步 把 B 上的 $n-1$ 个圆盘移到 C 上; 其中第一步和第三步是类同的。

当 $n=3$ 时, 第一步和第三步又分解为类同的三步, 即把 $n'-1$ 个圆盘从一个针移到另一个针上, 这里的 $n'=n-1$ 。显然这是一个递归过程, 据此算法可编程如下:

```
1. #include <stdio.h>
2.
3. move(int n, int x, int y, int z) {
4.     if(n == 1)
5.         printf("%c-->%c\n", x, z);
6.     else {
7.         move(n - 1, x, z, y);
8.         printf("%c-->%c\n", x, z);
```

```

9.         move(n - 1, y, x, z);
10.    }
11. }
12.
13. int main() {
14.     int h;
15.     printf("input number:\n");
16.     scanf("%d", &h);
17.     printf("the step to moving %2d disks:\n", h);
18.     move(h, 'a', 'b', 'c');
19.     return 0;
20. }

```

从程序中可以看出,move 函数是一个递归函数,它有四个形参 n,x,y,z。n 表示圆盘数,x,y,z 分别表示三根针。move 函数的功能是把 x 上的 n 个圆盘移动到 z 上。当 n==1 时,直接把 x 上的圆盘移至 z 上,输出 x→z。如 n!=1 则分为三步:递归调用 move 函数,把 n-1 个圆盘从 x 移到 y;输出 x→z;递归调用 move 函数,把 n-1 个圆盘从 y 移到 z。在递归调用过程中 n=n-1,故 n 的值逐次递减,最后 n=1 时,终止递归,逐层返回。

当 n=4 时程序运行的结果为:

```

1. input number:
2. 4
3. the step to moving 4 disks:
4. a-->b
5. a-->c
6. b-->c
7. a-->b
8. c-->a
9. c-->b
10. a-->b
11. a-->c
12. b-->c
13. b-->a
14. c-->a
15. b-->c
16. a-->b
17. a-->c
18. b-->c

```

5.7 数组作为函数参数

数组可以作为函数的参数使用, 进行数据传送。数组用作函数参数有两种形式, 一种是把数组元素(下标变量)作为实参使用; 另一种是把数组名作为函数的形参和实参使用。

1. 数组元素作函数实参

数组元素就是下标变量, 它与普通变量并无区别。因此它作为函数实参使用与普通变量是完全相同的, 在发生函数调用时, 把作为实参的数组元素的值传送给形参, 实现单向的值传送。下例说明了这种情况。

【例】判别一个整数数组中各元素的值, 若大于 0 则输出该值, 若小于等于 0 则输出 0 值。编程如下:

```
1. #include <stdio.h>
2.
3. void nzp(int v) {
4.     if(v > 0)
5.         printf("%d ", v);
6.     else
7.         printf("%d ", 0);
8. }
9.
10. int main() {
11.     int a[5], i;
12.     printf("input 5 numbers\n");
13.     for(i = 0; i < 5; i++) {
14.         scanf("%d", &a[i]);
15.         nzp(a[i]);
16.     }
17.     return 0;
18. }
```

本程序中首先定义一个无返回值函数 `nzp`, 并说明其形参 `v` 为整型变量。在函数体中根据 `v` 值输出相应的结果。在 `main` 函数中用一个 `for` 语句输入数组各元素, 每输入一个就以该元素作实参调用一次 `nzp` 函数, 即把 `a[i]` 的值传送给形参 `v`, 供 `nzp` 函数使用。

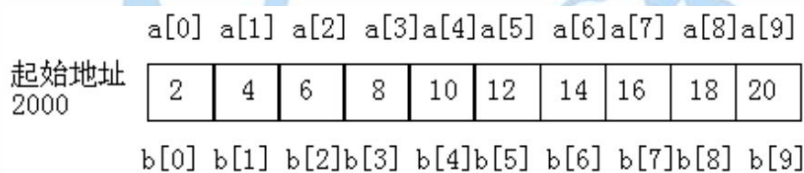
2. 数组名作为函数参数

用数组名作函数参数与用数组元素作实参有几点不同:

1) 用数组元素作实参时, 只要数组类型和函数的形参变量的类型一致, 那么作为下标变量的

数组元素的类型也和函数形参变量的类型是一致的。因此, 并不要求函数的形参也是下标变量。换句话说, 对数组元素的处理是按普通变量对待的。用数组名作函数参数时, 则要求形参和相对应的实参都必须是类型相同的数组, 都必须有明确的数组说明。当形参和实参二者不一致时, 即会发生错误。

2) 在普通变量或下标变量作函数参数时, 形参变量和实参变量是由编译系统分配的两个不同的内存单元。在函数调用时发生的值传送是把实参变量的值赋予形参变量。在用数组名作函数参数时, 不是进行值的传送, 即不是把实参数组的每一个元素的值都赋予形参数组的各个元素。因为实际上形参数组并不存在, 编译系统不为形参数组分配内存。那么, 数据的传送是如何实现的呢? 在我们曾介绍过, 数组名就是数组的首地址。因此在数组名作函数参数时所进行的传送只是地址的传送, 也就是说把实参数组的首地址赋予形参数组名。形参数组名取得该首地址之后, 也就等于有了实在的数组。实际上是形参数组和实参数组为同一数组, 共同拥有一段内存空间。



上图说明了这种情形。图中设 a 为实参数组, 类型为整型。 a 占有以 2000 为首地址的一块内存区。 b 为形参数组名。当发生函数调用时, 进行地址传送, 把实参数组 a 的首地址传送给形参数组名 b , 于是 b 也取得该地址 2000。于是 a, b 两数组共同占有以 2000 为首地址的一段连续内存单元。从图中还可以看出 a 和 b 下标相同的元素实际上也占相同的两个内存单元(整型数组每个元素占二字节)。例如 $a[0]$ 和 $b[0]$ 都占用 2000 和 2001 单元, 当然 $a[0]$ 等于 $b[0]$ 。类推则有 $a[i]$ 等于 $b[i]$ 。

【例】数组 a 中存放了一个学生 5 门课程的成绩, 求平均成绩。

```
1. #include <stdio.h>
2.
3. float aver(float a[5]) {
4.     int i;
5.     float av, s = a[0];
6.     for(i = 1; i < 5; i++)
7.         s = s + a[i];
8.     av = s / 5;
```

```

9.     return av;
10. }
11.
12. int main() {
13.     float sco[5], av;
14.     int i;
15.     printf("input 5 scores:\n");
16.     for(i = 0; i < 5; i++)
17.         scanf("%f", &sco[i]);
18.     av = aver(sco);
19.     printf("average score is %5.2f", av);
20.     return 0;
21. }

```

本程序首先定义了一个实型函数 `aver`，有一个形参为实型数组 `a`，长度为 5。在函数 `aver` 中，把各元素值相加求出平均值，返回给主函数。主函数 `main` 中首先完成数组 `sco` 的输入，然后以 `sco` 作为实参调用 `aver` 函数，函数返回值送 `av`，最后输出 `av` 值。从运行情况可以看出，程序实现了所要求的功能。

3) 前面已经讨论过，在变量作函数参数时，所进行的值传送是单向的。即只能从实参传向形参，不能从形参传回实参。形参的初值和实参相同，而形参的值发生改变后，实参并不变化，两者的终值是不同的。而当用数组名作函数参数时，情况则不同。由于实际上形参和实参为同一数组，因此当形参数组发生变化时，实参数组也随之变化。当然这种情况不能理解为发生了“双向”的值传递。但从实际情况来看，调用函数之后实参数组的值将由于形参数组值的变化而变化。

【例】题目同上例相同。改用数组名作函数参数。

```

1.  #include <stdio.h>
2.
3.  void nzp(int a[5]) {
4.      int i;
5.      printf("\nvalues of array a are:\n");
6.      for(i = 0; i < 5; i++) {
7.          if(a[i] < 0) a[i] = 0;
8.          printf("%d ", a[i]);
9.      }

```

```

10. }
11.
12. int main() {
13.     int b[5], i;
14.     printf("input 5 numbers:\n");
15.     for(i = 0; i < 5; i++)
16.         scanf("%d", &b[i]);
17.     printf("initial values of array b are:\n");
18.     for(i = 0; i < 5; i++)
19.         printf("%d ", b[i]);
20.     nzp(b);
21.     printf("\nlast values of array b are:\n");
22.     for(i = 0; i < 5; i++)
23.         printf("%d ", b[i]);
24.     return 0;
25. }

```

本程序中函数 `nzp` 的形参为整数组 `a`，长度为 5。主函数中实参数组 `b` 也为整型，长度也为 5。在主函数中首先输入数组 `b` 的值，然后输出数组 `b` 的初始值。然后以数组名 `b` 为实参调用 `nzp` 函数。在 `nzp` 中，按要求把负值单元清 0，并输出形参数组 `a` 的值。返回主函数之后，再次输出数组 `b` 的值。从运行结果可以看出，数组 `b` 的初值和终值是不同的，数组 `b` 的终值和数组 `a` 是相同的。这说明实参形参为同一数组，它们的值同时得以改变。

用数组名作为函数参数时还应注意以下几点：

- a. 形参数组和实参数组的类型必须一致，否则将引起错误。
- b. 形参数组和实参数组的长度可以不相同，因为在调用时，只传送首地址而不检查形参数组的长度。当形参数组的长度与实参数组不一致时，虽不至于出现语法错误(编译能通过)，但程序执行结果将与实际不符，这是应予以注意的。

【例】如把上例修改如下：

```

1. #include <stdio.h>
2.
3. void nzp(int a[8]) {
4.     int i;
5.     printf("\nvalues of array are:\n");
6.     for(i = 0; i < 8; i++) {

```

```

7.         if(a[i] < 0) a[i] = 0;
8.         printf("%d ", a[i]);
9.     }
10. }
11.
12. int main() {
13.     int b[5], i;
14.     printf("input 5 numbers:\n");
15.     for(i = 0; i < 5; i++)
16.         scanf("%d", &b[i]);
17.     printf("initial values of array b are:\n");
18.     for(i = 0; i < 5; i++)
19.         printf("%d ", b[i]);
20.     nzp(b);
21.     printf("\nlast values of array b are:\n");
22.     for(i = 0; i < 5; i++)
23.         printf("%d ", b[i]);
24.     return 0;
25. }

```

本程序与上例程序比, `nzp` 函数的形参数组长度改为 8, 函数体中, `for` 语句的循环条件也改为 `i<8`。因此, 形参数组 `a` 和实参数组 `b` 的长度不一致。编译能够通过, 但从结果看, 数组 `a` 的元素 `a[5]`, `a[6]`, `a[7]`显然是无意义的。

c. 在函数形参表中, 允许不给出形参数组的长度, 或用一个变量来表示数组元素的个数。

例如, 可以写为:

```
1. void nzp(int a[])
```

或写为

```
1. void nzp(int a[], int n)
```

其中形参数组 `a` 没有给出长度, 而由 `n` 值动态地表示数组的长度。`n` 的值由主调函数的实参进行传送。

由此, 上例又可改为下例的形式。

【例】

```

1. #include <stdio.h>
2.
3. void nzp(int a[], int n) {
4.     int i;

```



```

5.     printf("\nvalues of array a are:\n");
6.     for(i = 0; i < n; i++) {
7.         if(a[i] < 0) a[i] = 0;
8.         printf("%d ", a[i]);
9.     }
10. }
11.
12. int main() {
13.     int b[5], i;
14.     printf("input 5 numbers:\n");
15.     for(i = 0; i < 5; i++)
16.         scanf("%d", &b[i]);
17.     printf("initial values of array b are:\n");
18.     for(i = 0; i < 5; i++)
19.         printf("%d ", b[i]);
20.     nzp(b, 5);
21.     printf("\nlast values of array b are:\n");
22.     for(i = 0; i < 5; i++)
23.         printf("%d ", b[i]);
24.     return 0;
25. }

```

本程序 nzp 函数形参数组 a 没有给出长度, 由 n 动态确定该长度。在 main 函数中, 函数调用语句为 nzp(b, 5), 其中实参 5 将赋予形参 n 作为形参数组的长度。

d. 多维数组也可以作为函数的参数。在函数定义时对形参数组可以指定每一维的长度, 也可省去第一维的长度。因此, 以下写法都是合法的。

```
1. int MA(int a[3][10])
```

或

```
1. int MA(int a[][10])
```


5.8 局部变量和全局变量

在讨论函数的形参变量时曾经提到, 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。这一点表明形参变量只有在函数内才是有效的, 离开该函数就不能再使用了。这种变量有效性的范围称变量的作用域。不仅对于形参变量, C 语言中所有的量都有自己的作用域。变量说明的方式不同, 其作用域也不同。C 语言中的变量, 按作用域范围可分为两种, 即局部变量和全局变量。

局部变量

局部变量也称为内部变量。局部变量是在函数内作定义说明的。其作用域仅限于函数内, 离开该函数后再使用这种变量是非法的。

例如:

```
1.  int f1(int a) /*函数 f1*/
2.  {
3.      int b,c;
4.      .....
5.  }
6.  a,b,c 有效
7.
8.  int f2(int x) /*函数 f2*/
9.  {
10.     int y,z;
11.     .....
12. }
13. x,y,z 有效
14.
15. main()
16. {
17.     int m,n;
18.     .....
19. }
20. m,n 有效
```

在函数 f1 内定义了三个变量, a 为形参, b,c 为一般变量。在 f1 的范围内 a,b,c 有效, 或者说 a,b,c 变量的作用域限于 f1 内。同理, x,y,z 的作用域限于 f2 内。m,n 的作用域限于

main 函数内。关于局部变量的作用域还要说明以下几点：

- 1) 主函数中定义的变量也只能在主函数中使用，不能在其它函数中使用。同时，主函数中也不能使用其它函数中定义的变量。因为主函数也是一个函数，它与其它函数是平行关系。这一点是与其它语言不同的，应予以注意。
- 2) 形参变量是属于被调函数的局部变量，实参变量是属于主调函数的局部变量。
- 3) 允许在不同的函数中使用相同的变量名，它们代表不同的对象，分配不同的单元，互不干扰，也不会发生混淆。如在前例中，形参和实参的变量名都为 `n`，是完全允许的。
- 4) 在复合语句中也可定义变量，其作用域只在复合语句范围内。

例如：

```
1.  main()
2.  {
3.      int s,a;
4.      .....
5.      {
6.          int b;
7.          s=a+b;
8.          ..... /*b 作用域*/
9.      }
10.     ..... /*s,a 作用域*/
11. }
```

noobdream.com

【例】

```
1.  #include <stdio.h>
2.
3.  int main() {
4.      int i = 2, j = 3, k;
5.      k = i + j;
6.      {
7.          int k = 8;
8.          printf("%d\n", k);
9.      }
10.     printf("%d\n", k);
11.     return 0;
12. }
```

本程序在 `main` 中定义了 `i,j,k` 三个变量，其中 `k` 未赋初值。而在复合语句内又定义了一

个变量 `k`，并赋初值为 8。应该注意这两个 `k` 不是同一个变量。在复合语句外由 `main` 定义的 `k` 起作用，而在复合语句内则由在复合语句内定义的 `k` 起作用。因此程序第 4 行的 `k` 为 `main` 所定义，其值应为 5。第 7 行输出 `k` 值，该行在复合语句内，由复合语句内定义的 `k` 起作用，其初值为 8，故输出值为 8，第 9 行输出 `i`, `k` 值。`i` 是在整个程序中有效的，第 7 行对 `i` 赋值为 3，故以输出也为 3。而第 9 行已在复合语句之外，输出的 `k` 应为 `main` 所定义的 `k`，此 `k` 值由第 4 行已获得为 5，故输出也为 5。

全局变量

全局变量也称为外部变量，它是在函数外部定义的变量。它不属于哪一个函数，它属于一个源程序文件。其作用域是整个源程序。在函数中使用全局变量，一般应作全局变量说明。只有在函数内经过说明的全局变量才能使用。全局变量的说明符为 `extern`。但在一个函数之前定义的全局变量，在该函数内使用可不再加以说明。

例如：

```
1. int a, b; /*外部变量*/
2. void f1() /*函数 f1*/
3. {
4.     .....
5. }
6. float x, y; /*外部变量*/
7. int fz() /*函数 fz*/
8. {
9.     .....
10. }
11. main() /*主函数*/
12. {
13.     .....
14. }
```

从上例可以看出 `a`、`b`、`x`、`y` 都是在函数外部定义的外部变量，都是全局变量。但 `x`、`y` 定义在函数 `f1` 之后，而在 `f1` 内又无对 `x`、`y` 的说明，所以它们在 `f1` 内无效。`a`、`b` 定义在源程序最前面，因此在 `f1`、`f2` 及 `main` 内不加说明也可使用。

【例】输入正方体的长宽高 l,w,h。求体积及三个面 $x*y$, $x*z$, $y*z$ 的面积。

```
1. #include <stdio.h>
2.
3. int s1, s2, s3;
4. int vs( int a, int b, int c) {
5.     int v;
6.     v = a * b * c;
7.     s1 = a * b;
8.     s2 = b * c;
9.     s3 = a * c;
10.    return v;
11. }
12.
13. int main() {
14.     int v, l, w, h;
15.     printf("input length,width and height\n");
16.     scanf("%d%d%d", &l, &w, &h);
17.     v = vs(l, w, h);
18.     printf("\nv=%d,s1=%d,s2=%d,s3=%d\n", v, s1, s2, s3);
19.     return 0;
20. }
```

【例】外部变量与局部变量同名。

```
1. #include <stdio.h>
2.
3. int a = 3, b = 5; /*a,b 为外部变量*/
4. int max(int a, int b) /*a,b 为外部变量*/
5. {
6.     int c;
7.     c = a > b ? a : b;
8.     return(c);
9. }
10.
11. int main() {
12.     int a = 8;
13.     printf("%d\n", max(a, b));
14.     return 0;
15. }
```

如果同一个源文件中，外部变量与局部变量同名，则在局部变量的作用范围内，外部变量被“屏蔽”，即它不起作用。

*5.9 变量的存储方式和生存期

动态存储方式和静态存储方式

前面已经介绍了，从变量的作用域（即从空间）角度来分，可以分为全局变量和局部变量。

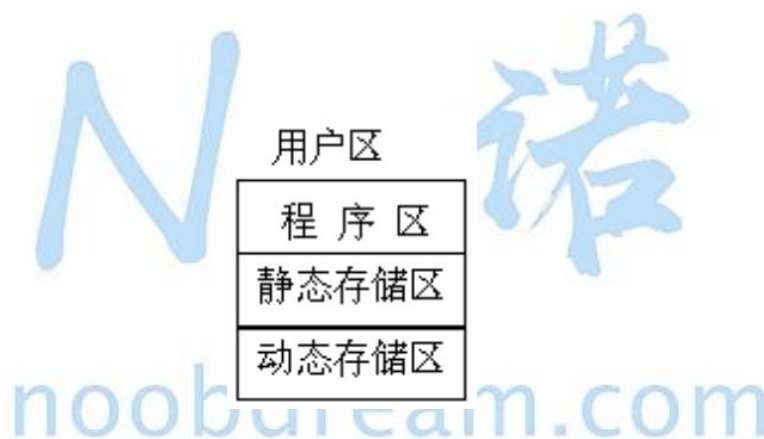
从另一个角度，从变量值存在的时间（即生存期）角度来分，可以分为静态存储方式和动态存储方式。

静态存储方式：是指在程序运行期间分配固定的存储空间的方式。

动态存储方式：是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。

用户存储空间可以分为三个部分：

- 1) 程序区；
- 2) 静态存储区；
- 3) 动态存储区；



全局变量全部存放在静态存储区，在程序开始执行时给全局变量分配存储区，程序行完毕就释放。在程序执行过程中它们占据固定的存储单元，而不动态地进行分配和释放；

动态存储区存放以下数据：

- 1) 函数形式参数；
- 2) 自动变量（未加 `static` 声明的局部变量）；
- 3) 函数调用时的现场保护和返回地址；

对以上这些数据，在函数开始调用时分配动态存储空间，函数结束时释放这些空间。

在 `c` 语言中，每个变量和函数有两个属性：数据类型和数据的存储类别。

局部变量的存储类别

函数中的局部变量，如不专门声明为 `static` 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。这类局部变量称为自动变量。自动变量用关键字 `auto` 作存储类别的声明。

例如：

```
1. int f(int a) /*定义 f 函数, a 为参数*/
2. {
3.     auto int b, c = 3; /*定义 b, c 自动变量*/
4.     .....
5. }
```

`a` 是形参，`b`，`c` 是自动变量，对 `c` 赋初值 3。执行完 `f` 函数后，自动释放 `a`，`b`，`c` 所占的存储单元。关键字 `auto` 可以省略，`auto` 不写则隐含定为“自动存储类别”，属于动态存储方式。

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。

【例】考察静态局部变量的值。

```
1. #include <stdio.h>
2.
3. int f(int a) {
4.     auto int b = 0;
5.     static int c = 3;
6.     b = b + 1;
7.     c = c + 1;
8.     return(a + b + c);
9. }
10.
11. int main() {
12.     int a = 2, i;
13.     for(i = 0; i < 3; i++)
14.         printf("%d", f(a));
15.     return 0;
16. }
```

对静态局部变量的说明:

- 1) 静态局部变量属于静态存储类别，在静态存储区内分配存储单元。在程序整个运行期间都不释放。而自动变量（即动态局部变量）属于动态存储类别，占动态存储空间，函数调用结束后即释放。
- 2) 静态局部变量在编译时赋初值，即只赋初值一次；而对自动变量赋初值是在函数调用时进行，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。
- 3) 如果在定义局部变量时不赋初值的话，则对静态局部变量来说，编译时自动赋初值 0（对数值型变量）或空字符（对字符变量）。而对自动变量来说，如果不赋初值则它的值是一个不确定的值。

【例】打印 1 到 5 的阶乘值。

```
1. #include <stdio.h>
2.
3. int fac(int n) {
4.     static int f = 1;
5.     f = f * n;
6.     return(f);
7. }
8.
9. int main() {
10.    int i;
11.    for(i = 1; i <= 5; i++)
12.        printf("%d!=%d\n", i, fac(i));
13.    return 0;
14. }
```

5.10 关于变量的声明和定义

register 变量

为了提高效率, C 语言允许将局部变量得值放在 CPU 中的寄存器中, 这种变量叫“寄存器变量”, 用关键字 `register` 作声明。

【例】使用寄存器变量。

```
1. #include <stdio.h>
2.
3. int fac(int n) {
4.     register int i, f = 1;
5.     for(i = 1; i <= n; i++)
6.         f = f * i;
7.     return(f);
8. }
9.
10. int main() {
11.     int i;
12.     for(i = 0; i <= 5; i++)
13.         printf("%d!=%d\n", i, fac(i));
14.     return 0;
15. }
```

说明:

- 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) 局部静态变量不能定义为寄存器变量。

用 extern 声明外部变量

外部变量 (即全局变量) 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。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终了。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` 对该变量作“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。

【例】用 `extern` 声明外部变量，扩展程序文件中的作用域。

```
1. #include <stdio.h>
2.
3. int max(int x, int y) {
4.     int z;
5.     z = x > y ? x : y;
6.     return(z);
7. }
8.
9. int main() {
10.    extern int A,B;
11.    printf("%d\n", max(A, B));
12.    return 0;
13. }
14. int A = 13, B = -8;
```

说明：在本程序文件的最后 1 行定义了外部变量 A，B，但由于外部变量定义的位置在函数 main 之后，因此本来在 main 函数中不能引用外部变量 A，B。现在我们在 main 函数中用 extern 对 A 和 B 进行“外部变量声明”，就可以从“声明”处起，合法地使用该外部变量 A 和 B。

noobdream.com

*5.11 内部函数和外部函数

函数本质上是全局的, 因为一个函数要被另外的函数调用, 但是, 也可以指定函数只能被本文件调用, 而不能被其他文件调用。根据此特点, 将函数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, 它称为内部函数。在定义内部函数时, 在函数名和函数类型的前面加 **static**。函数首部的一般格式为

1. **static** 类型标识符 函数名 (形参表)

如

1. **static int** fun(**int** a, **int** b)

内部函数又称**静态 (static) 函数**。使用内部函数, 可以使函数只局限于所在文件。如果在不同的文件中有同名的内部函数, 互不干扰。通常把只能由同一文件使用的函数和外部变量放在一个文件中, 在它们前面都冠以 **static** 使之局部化, 其他文件不能引用。

外部函数

(1) 在定义函数时, 如果在函数首部的最左端冠以关键字 **extern**, 则表示此函数是外部函数, 可供其他文件调用。

如函数首部可以写为

1. **extern int** fun (**int** a, **int** b)

这样, 函数 fun 就可以为其他文件调用。如果在定义函数时省略 **extern**, 则默认为外部函数。本书前面所用的函数都是外部函数。

(2) 在需要调用此函数的文件中, 用 **extern** 声明所用的函数是外部函数。

例: 输入两个整数, 要求输出其中的大者。用外部函数实现。

```
1. //file1.cpp (文件1)
2. #include <stdio.h>
3. int main() {
4.     extern int max(int,int); //声明在本函数中将要调用在其他文件中定义的max函数
5.     int a, b;
6.     scanf("%d%d", &a, &b);
7.     printf("%d\n", max(a, b));
8.     return 0;
```

```
9. }
10.
11. //file2.cpp (文件2)
12. int max(int x, int y) {
13.     int z;
14.     z = x > y ? x : y;
15.     return z;
16. }
```

运行情况如下：

```
1. 7 -34
2. 7
```

通过此例可知：使用 `extern` 声明就能够在文件中调用其他文件中定义的函数，或者说把该函数的作用域扩展到本文件。`extern` 声明的形式就是在函数原型基础上加关键字 `extern`。由于函数在本质上是外部的，在程序中经常要调用其他文件中的外部函数，为方便编程，C++允许在声明函数时省写 `extern`。上例 程序 `main` 函数中的函数声明可写成

```
1. int max(int,int);
```

这就是我们多次用过的函数原型。由此可以进一步理解函数原型的作用。用函数原型能够把函数的作用域扩展到定义该函数的文件之外（不必使用 `extern`）。只要在使用该函数的每一个文件中包含该函数的函数原型即可。函数原型通知编译系统：该函数在本文件中稍后定义，或在另一文件中定义。

利用函数原型扩展函数作用域最常见的例子是 `#include` 命令的应用。在 `#include` 命令所指定的头文件中包含有调用库函数时所需的信息。例如，在程序中需要调用 `sin` 函数，但三角函数并不是由用户在本文件中定义的，而是存放在数学函数库中的。按以上的介绍，必须在本文件中写出 `sin` 函数的原型，否则无法调用 `sin` 函数。`sin` 函数的原型是 `double sin(double x);`

本来应该由程序设计者在调用库函数时先从手册中查出所用的库函数的原型，并在程序中一一写出来，但这显然是麻烦而困难的。为减少程序设计者的困难，在头文件 `cmath` 中包括了所有数学函数的原型和其他有关信息，用户只需用以下 `#include` 命令：

```
1. #include <cmath>
```

即可。这时，在该文件中就能合法地调用各数学库函数了。

5.12 本章小结

实战练习

DreamJudge 1135 矩阵求和

DreamJudge 1128 疯狂的小面包

DreamJudge 1095 Y/N

DreamJudge 1032 变位词

DreamJudge 1026 删除字符串

DreamJudge 1014 加密算法



C 语言专业题库在线练习

<http://www.noobdream.com/Practice/clang/>

第六章 指针

【本章知识点汇总】

指针变量

通过指针引用数组

通过指针引用字符串

指向函数的指针

返回指针值的函数

指针数组和多重指针

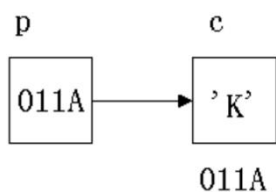
动态内存分配

本书配套视频精讲: <https://www.bilibili.com/video/BV1HJ41137fe>

6.1 指针是什么

指针是C语言中广泛使用的一种数据类型。运用指针编程是C语言最主要的风格之一。利用指针变量可以表示各种数据结构；能很方便地使用数组和字符串；并能象汇编语言一样处理内存地址，从而编出精练而高效的程序。指针极大地丰富了C语言的功能。学习指针是学习C语言中最重要的一环，能否正确理解和使用指针是我们是否掌握C语言的一个标志。同时，指针也是C语言中最为困难的一部分，在学习中除了要正确理解基本概念，还必须要多编程，上机调试。只要作到这些，指针也是不难掌握的。

在计算机中，所有的数据都是存放在存储器中的。一般把存储器中的一个字节称为一个内存单元，不同的数据类型所占用的内存单元数不等，如整型量占 2 个单元，字符型占 1 个单元等，在前面已有详细的介绍。为了正确地访问这些内存单元，必须为每个内存单元编上号。根据一个内存单元的编号即可准确地找到该内存单元。内存单元的编号也叫做地址。既然根据内存单元的编号或地址就可以找到所需的内存单元，所以通常也把这个地址称为指针。内存单元的指针和内存单元的内容是两个不同的概念。可以用一个通俗的例子来说明它们之间的关系。我们到银行去存取款时，银行工作人员将根据我们的帐号去找我们的存款单，找到之后在存单上写入存款、取款的金额。在这里，帐号就是存单的指针，存款数是存单的内容。对于一个内存单元来说，单元的地址即为指针，其中存放的数据才是该单元的内容。在C语言中，允许用一个变量来存放指针，这种变量称为指针变量。因此，一个指针变量的值就是某个内存单元的地址或称为某内存单元的指针。



图中，设有字符变量 C，其内容为“K”（ASCII 码为十进制数 75），C 占用了 011A 号单元(地址用十六进数表示)。设有指针变量 P，内容为 011A，这种情况我们称为 P 指向变量 C，或说 P 是指向变量 C 的指针。

严格地说，一个指针是一个地址，是一个常量。而一个指针变量却可以被赋予不同的指针

值，是变量。但常把指针变量简称为指针。为了避免混淆，我们中约定：“指针”是指地址，是常量，“指针变量”是指取值为地址的变量。定义指针的目的是为了通过指针去访问内存单元。

既然指针变量的值是一个地址，那么这个地址不仅可以是变量的地址，也可以是其它数据结构的地址。在一个指针变量中存放一个数组或一个函数的首地址有何意义呢？因为数组或函数都是连续存放的。通过访问指针变量取得了数组或函数的首地址，也就找到了该数组或函数。这样一来，凡是出现数组，函数的地方都可以用一个指针变量来表示，只要该指针变量中赋予数组或函数的首地址即可。这样做，将会使程序的概念十分清楚，程序本身也精练，高效。在C语言中，一种数据类型或数据结构往往都占有一组连续的内存单元。用“地址”这个概念并不能很好地描述一种数据类型或数据结构，而“指针”虽然实际上也是一个地址，但它却是一个数据结构的首地址，它是“指向”一个数据结构的，因而概念更为清楚，表示更为明确。这也是引入“指针”概念的一个重要原因。

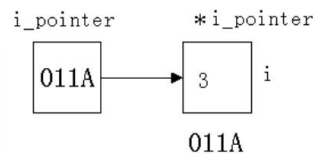
noobdream.com

6.2 指针变量

使用指针变量的例子

变量的指针就是变量的地址。存放变量地址的变量是指针变量。即在C语言中, 允许用一个变量来存放指针, 这种变量称为指针变量。因此, 一个指针变量的值就是某个变量的地址或称为某变量的指针。

为了表示指针变量和它所指向的变量之间的关系, 在程序中用“*”符号表示“指向”, 例如, `i_pointer` 代表指针变量, 而 `*i_pointer` 是 `i_pointer` 所指向的变量。



因此, 下面两个语句作用相同:

1. `i=3;`
2. `*i_pointer=3;`

第二个语句的含义是将 3 赋给指针变量 `i_pointer` 所指向的变量。

怎样定义指针变量

对指针变量的定义包括三个内容:

- (1) 指针类型说明, 即定义变量为一个指针变量;
- (2) 指针变量名;
- (3) 变量值(指针)所指向的变量的数据类型。

其一般形式为:

1. 类型说明符 *变量名;

其中, *表示这是一个指针变量, 变量名即为定义的指针变量名, 类型说明符表示本指针变量所指向的变量的数据类型。

例如:

1. `int *p1;`

表示 `p1` 是一个指针变量, 它的值是某个整型变量的地址。或者说 `p1` 指向一个整型变量。至于 `p1` 究竟指向哪一个整型变量, 应由向 `p1` 赋予的地址来决定。

再如:

1. `int *p2;` /*p2 是指向整型变量的指针变量*/
2. `float *p3;` /*p3 是指向浮点变量的指针变量*/
3. `char *p4;` /*p4 是指向字符变量的指针变量*/

应该注意的是, 一个指针变量只能指向同类型的变量, 如 P3 只能指向浮点变量, 不能时而指向一个浮点变量, 时而又指向一个字符变量。

怎样引用指针变量

指针变量同普通变量一样, 使用之前不仅要定义说明, 而且必须赋予具体的值。未经赋值的指针变量不能使用, 否则将造成系统混乱, 甚至死机。指针变量的赋值只能赋予地址, 决不能赋予任何其它数据, 否则将引起错误。在 C 语言中, 变量的地址是由编译系统分配的, 对用户完全透明, 用户不知道变量的具体地址。

两个有关的运算符:

- 1) `&` 取地址运算符。
- 2) `*` 指针运算符 (或称“间接访问”运算符)。

C 语言中提供了地址运算符`&`来表示变量的地址。

其一般形式为:

1. `&` 变量名;

如`&a` 表示变量 `a` 的地址, `&b` 表示变量 `b` 的地址。变量本身必须预先说明。

设有指向整型变量的指针变量 `p`, 如要把整型变量 `a` 的地址赋予 `p` 可以有以下两种方式:

(1) 指针变量初始化的方法

1. `int a;`
2. `int *p = &a;`

(2) 赋值语句的方法

1. `int a;`
2. `int *p;`
3. `p = &a;`

不允许把一个数赋予指针变量, 故下面的赋值是错误的:

1. `int *p;`
2. `p = 1000;`

被赋值的指针变量前不能再加“*”说明符, 如写为*p=&a 也是错误的。

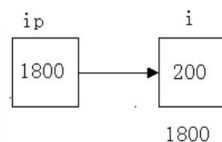
假设:

```
1. int i = 200, x;
2. int *ip;
```

我们定义了两个整型变量 i,x,还定义了一个指向整型数的指针变量 ip。i,x 中可存放整数,而 ip 中只能存放整型变量的地址。我们可以把 i 的地址赋给 ip:

```
1. ip = &i;
```

此时指针变量 ip 指向整型变量 i,假设变量 i 的地址为 1800,这个赋值可形象理解为下图所示的联系。



以后我们便可以通过指针变量 ip 间接访问变量 i,例如:

```
1. x = *ip;
```

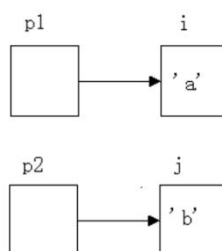
运算符*访问以 ip 为地址的存储区域,而 ip 中存放的是变量 i 的地址,因此,*ip 访问的是地址为 1800 的存储区域(因为是整数,实际上是从 1800 开始的两个字节),它就是 i 所占用的存储区域,所以上面的赋值表达式等价于

```
1. x = i;
```

另外,指针变量和一般变量一样,存放在它们之中的值是可以改变的,也就是说可以改变它们的指向,假设

```
1. int i, j, *p1, *p2;
2. i = 'a';
3. j = 'b';
4. p1 = &i;
5. p2 = &j;
```

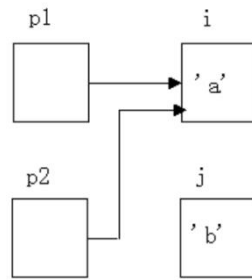
则建立如下图所示的联系:



这时赋值表达式:

```
1. p2 = p1
```

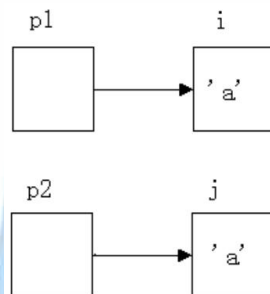
就使 p2 与 p1 指向同一对象 i, 此时 *p2 就等价于 i, 而不是 j, 图所示:



如果执行如下表达式:

1. *p2 = *p1;

则表示把 p1 指向的内容赋给 p2 所指的区域, 此时就变成图所示



通过指针访问它所指向的一个变量是以间接访问的形式进行的, 所以比直接访问一个变量要费时间, 而且不直观, 因为通过指针要访问哪一个变量, 取决于指针的值(即指向), 例如 "*p2=*p1;" 实际上就是 "j=i;", 前者不仅速度慢而且目的不明。但由于指针是变量, 我们可以通过改变它们的指向, 以间接访问不同的变量, 这给程序员带来灵活性, 也使程序代码编写得更为简洁和有效。

指针变量可出现在表达式中, 设

1. int x, y, *px = &x;

指针变量 px 指向整数 x, 则 *px 可出现在 x 能出现的任何地方。例如:

1. y = *px+5; /*表示把 x 的内容加 5 并赋给 y*/
2. y = ++*px; /**px 的内容加上 1 之后赋给 y, ++*px 相当于++(*px)*/
3. y = *px++; /*相当于 y=*px; px++*/

【例】

```

1. #include <stdio.h>
2.
3. int main() {
4.     int a,b;
5.     int *pointer_1, *pointer_2;
  
```

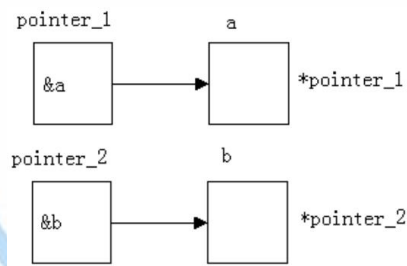
```

6.    a = 100; b = 10;
7.    pointer_1 = &a;
8.    pointer_2 = &b;
9.    printf("%d,%d\n", a, b);
10.   printf("%d,%d\n", *pointer_1, *pointer_2);
11.   return 0;
12. }

```

对程序的说明:

1) 在开头处虽然定义了两个指针变量 `pointer_1` 和 `pointer_2`, 但它们并未指向任何一个整型变量。只是提供两个指针变量, 规定它们可以指向整型变量。程序第 5、6 行的作用就是使 `pointer_1` 指向 `a`, `pointer_2` 指向 `b`。



2) 最后一行的 `*pointer_1` 和 `*pointer_2` 就是变量 `a` 和 `b`。最后两个 `printf` 函数作用是相同的。

3) 程序中有两处出现 `*pointer_1` 和 `*pointer_2`, 请区分它们的不同含义。

4) 程序第 5、6 行的 “`pointer_1=&a`” 和 “`pointer_2=&b`” 不能写成 “`*pointer_1=&a`” 和 “`*pointer_2=&b`”。

请对下面再的关于 “&” 和 “*” 的问题进行考虑:

1) 如果已经执行了 “`pointer_1=&a;`” 语句, 则 `&*pointer_1` 是什么含义?

2) `*&a` 含义是什么?

3) `(pointer_1)++` 和 `pointer_1++` 的区别?

【例】输入 `a` 和 `b` 两个整数, 按先大后小的顺序输出 `a` 和 `b`。

```

1.  #include <stdio.h>
2.
3.  int main() {
4.      int *p1, *p2, *p, a, b;
5.      scanf("%d,%d", &a, &b);

```

```
6.     p1 = &a; p2 = &b;
7.     if(a < b) {
8.         p = p1;
9.         p1 = p2;
10.        p2 = p;
11.    }
12.    printf("a=%d,b=%d\n", a, b);
13.    printf("max=%d,min=%d\n", *p1, *p2);
14.    return 0;
15. }
```

指针变量作为函数参数

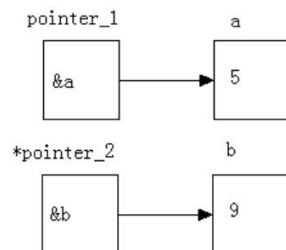
函数的参数不仅可以是整型、实型、字符型等数据，还可以是指针类型。它的作用是将一个变量的地址传送到另一个函数中。

【例】输入的两个整数按大小顺序输出。今用函数处理，而且用指针类型的数据作函数参数。

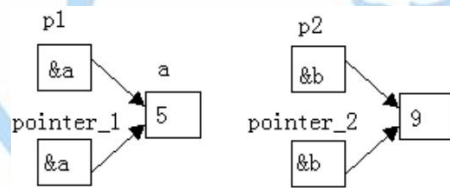
```
1.  #include <stdio.h>
2.
3.  swap(int *p1, int *p2) {
4.      int temp;
5.      temp = *p1;
6.      *p1 = *p2;
7.      *p2 = temp;
8.  }
9.
10. int main() {
11.     int a, b;
12.     int *pointer_1, *pointer_2;
13.     scanf("%d,%d", &a, &b);
14.     pointer_1 = &a;
15.     pointer_2 = &b;
16.     if(a < b) swap(pointer_1, pointer_2);
17.     printf("%d,%d\n", a, b);
18.     return 0;
19. }
```

对程序的说明：

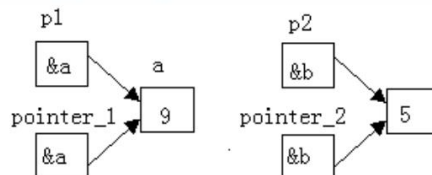
swap 是用户定义的函数，它的作用是交换两个变量(a 和 b)的值。swap 函数的形参 p1、p2 是指针变量。程序运行时，先执行 main 函数，输入 a 和 b 的值。然后将 a 和 b 的地址分别赋给指针变量 pointer_1 和 pointer_2，使 pointer_1 指向 a，pointer_2 指向 b。



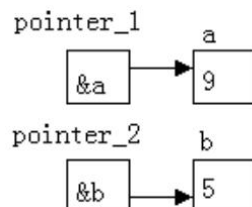
接着执行 if 语句，由于 $a < b$ ，因此执行 swap 函数。注意实参 pointer_1 和 pointer_2 是指针变量，在函数调用时，将实参变量的值传递给形参变量。采取的依然是“值传递”方式。因此虚实结合后形参 p1 的值为&a，p2 的值为&b。这时 p1 和 pointer_1 指向变量 a，p2 和 pointer_2 指向变量 b。



接着执行 swap 函数的函数体使 *p1 和 *p2 的值互换，也就是使 a 和 b 的值互换。



函数调用结束后，p1 和 p2 不复存在（已释放）如图。



最后在 main 函数中输出的 a 和 b 的值是已经过交换的值。

请注意交换 *p1 和 *p2 的值是如何实现的。请找出下列程序段的错误：

```

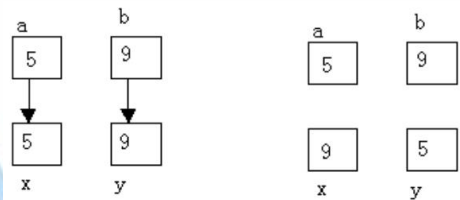
1. swap(int *p1, int *p2) {
2.     int *temp;
3.     *temp = *p1; /*此语句有问题*/
  
```

```
4.    *p1 = *p2;
5.    *p2 = temp;
6. }
```

请考虑下面的函数能否实现实现 a 和 b 互换。

```
1. swap(int x, int y) {
2.     int temp;
3.     temp = x;
4.     x = y;
5.     y = temp;
6. }
```

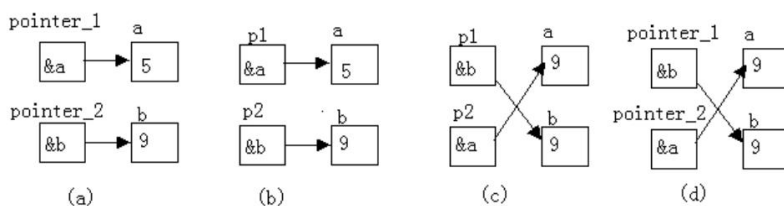
如果在 main 函数中用 “swap(a,b);” 调用 swap 函数, 会有什么结果呢? 请看下图所示。



【例】请注意, 不能企图通过改变指针形参的值而使指针实参的值改变。

```
1. #include <stdio.h>
2.
3. swap(int *p1, int *p2) {
4.     int *p;
5.     p = p1;
6.     p1 = p2;
7.     p2 = p;
8. }
9.
10. int main() {
11.     int a, b;
12.     int *pointer_1, *pointer_2;
13.     scanf("%d,%d", &a, &b);
14.     pointer_1 = &a;
15.     pointer_2 = &b;
16.     if(a < b) swap(pointer_1, pointer_2);
17.     printf("%d,%d\n", *pointer_1, *pointer_2);
18.     return 0;
19. }
```

其中的问题在于不能实现如图所示的第四步（d）。



【例】输入 `a`、`b`、`c` 3 个整数，按大小顺序输出。

```
1. #include <stdio.h>
2.
3. swap(int *pt1, int *pt2) {
4.     int temp;
5.     temp = *pt1;
6.     *pt1 = *pt2;
7.     *pt2 = temp;
8. }
9.
10. exchange(int *q1, int *q2, int *q3) {
11.     if(*q1 < *q2) swap(q1, q2);
12.     if(*q1 < *q3) swap(q1, q3);
13.     if(*q2 < *q3) swap(q2, q3);
14. }
15.
16. int main() {
17.     int a, b, c, *p1, *p2, *p3;
18.     scanf("%d,%d,%d", &a, &b, &c);
19.     p1 = &a; p2 = &b; p3 = &c;
20.     exchange(p1, p2, p3);
21.     printf("%d,%d,%d \n", a, b, c);
22.     return 0;
23. }
```


6.3 指针变量的运用

指针变量可以进行某些运算,但其运算的种类是有限的。它只能进行赋值运算和部分算术运算及关系运算。

1. 指针运算符

1) 取地址运算符&:取地址运算符&是单目运算符,其结合性为自右至左,其功能是取变量的地址。在 scanf 函数及前面介绍指针变量赋值中,我们已经了解并使用了&运算符。

2) 取内容运算符*:取内容运算符*是单目运算符,其结合性为自右至左,用来表示指针变量所指的变量。在*运算符之后跟的变量必须是指针变量。

需要注意的是指针运算符*和指针变量说明中的指针说明符*不是一回事。在指针变量说明中,“*”是类型说明符,表示其后的变量是指针类型。而表达式中出现的“*”则是一个运算符用以表示指针变量所指的变量。

【例】

```
1. #include <stdio.h>
2.
3. int main() {
4.     int a = 5, *p = &a;
5.     printf ("%d", *p);
6.     return 0;
7. }
```

表示指针变量 p 取得了整型变量 a 的地址。printf("%d",*p)语句表示输出变量 a 的值。

2. 指针变量的运算

1) 赋值运算:指针变量的赋值运算有以下几种形式。

- ① 指针变量初始化赋值,前面已作介绍。
- ② 把一个变量的地址赋予指向相同数据类型的指针变量。

例如:

```
1. int a, *pa;
2. pa = &a; /*把整型变量 a 的地址赋予整型指针变量 pa*/
```

- ③ 把一个指针变量的值赋予指向相同类型变量的另一个指针变量。

如:

```
1. int a, *pa = &a, *pb;
2. pb = pa; /*把 a 的地址赋予指针变量 pb*/
```

由于 `pa, pb` 均为指向整型变量的指针变量, 因此可以相互赋值。

④ 把数组的首地址赋予指向数组的指针变量。

例如:

```
1. int a[5], *pa;
2. pa = a;
```

(数组名表示数组的首地址, 故可赋予指向数组的指针变量 `pa`)

也可写为:

```
1. pa = &a[0]; /*数组第一个元素的地址也是整个数组的首地址, 也可赋予 pa*/
```

当然也可采取初始化赋值的方法:

```
1. int a[5], *pa = a;
```

⑤ 把字符串的首地址赋予指向字符类型的指针变量。

例如:

```
1. char *pc;
2. pc = "C Language";
```

或用初始化赋值的方法写为:

```
1. char *pc = "C Language";
```

这里应说明的是并不是把整个字符串装入指针变量, 而是把存放该字符串的字符数组的首地址装入指针变量。在后面还将详细介绍。

⑥ 把函数的入口地址赋予指向函数的指针变量。

例如:

```
1. int (*pf)();
2. pf = f; /*f 为函数名*/
```

2) 加减算术运算

对于指向数组的指针变量, 可以加上或减去一个整数 `n`。设 `pa` 是指向数组 `a` 的指针变量, 则 `pa+n, pa-n, pa++, ++pa, pa--, --pa` 运算都是合法的。指针变量加或减一个整数 `n` 的意义是把指针指向的当前位置(指向某数组元素)向前或向后移动 `n` 个位置。应该注意, 数组指针变量向前或向后移动一个位置和地址加 1 或减 1 在概念上是不同的。因为数组可以有不同的类型, 各种类型的数组元素所占的字节长度是不同的。如指针变量加 1, 即向后移动 1 个位置表示指针变量指向下一个数据元素的首地址。而不是在原地址基础上加 1。例如:

1. `int a[5], *pa;`
2. `pa = a; /*pa 指向数组 a, 也是指向 a[0]*/`
3. `pa = pa + 2; /*pa 指向 a[2], 即 pa 的值为&a[2]*/`

指针变量的加减运算只能对数组指针变量进行, 对指向其它类型变量的指针变量作加减运算是毫无意义的。

3) 两个指针变量之间的运算: 只有指向同一数组的两个指针变量之间才能进行运算, 否则运算毫无意义。

① 两指针变量相减: 两指针变量相减所得之差是两个指针所指数组元素之间相差的元素个数。实际上是两个指针值(地址)相减之差再除以该数组元素的长度(字节数)。例如 `pf1` 和 `pf2` 是指向同一浮点数组的两个指针变量, 设 `pf1` 的值为 `2010H`, `pf2` 的值为 `2000H`, 而浮点数组每个元素占 4 个字节, 所以 `pf1-pf2` 的结果为 $(2000H-2010H)/4=4$, 表示 `pf1` 和 `pf2` 之间相差 4 个元素。两个指针变量不能进行加法运算。例如, `pf1+pf2` 是什么意思呢? 毫无实际意义。

② 两指针变量进行关系运算: 指向同一数组的两指针变量进行关系运算可表示它们所指数组元素之间的关系。

例如:

1. `pf1 == pf2` 表示 `pf1` 和 `pf2` 指向同一数组元素;
2. `pf1 > pf2` 表示 `pf1` 处于高地址位置;
3. `pf1 < pf2` 表示 `pf2` 处于低地址位置。

指针变量还可以与 0 比较。

设 `p` 为指针变量, 则 `p==0` 表明 `p` 是空指针, 它不指向任何变量;

`p!=0` 表示 `p` 不是空指针。

空指针是由对指针变量赋予 0 值而得到的。

例如:

1. `#define NULL 0`
2. `int *p = NULL;`

对指针变量赋 0 值和不赋值是不同的。指针变量未赋值时, 可以是任意值, 是不能使用的。否则将造成意外错误。而指针变量赋 0 值后, 则可以使用, 只是它不指向具体的变量而已。

【例】

1. `#include <stdio.h>`
- 2.
3. `int main() {`

```
4.   int a = 10, b = 20, s, t, *pa, *pb; /*说明 pa,pb 为整型指针变量*/
5.   pa = &a; /*给指针变量 pa 赋值, pa 指向变量 a*/
6.   pb = &b; /*给指针变量 pb 赋值, pb 指向变量 b*/
7.   s = *pa + *pb; /*求 a+b 之和, (*pa 就是 a, *pb 就是 b)*/
8.   t = *pa * *pb; /*本行是求 a*b 之积*/
9.   printf("a=%d\nb=%d\na+b=%d\na*b=%d\n", a, b, a+b, a*b);
10.  printf("s=%d\nt=%d\n", s, t);
11.  return 0;
12. }
```

【例】

```
1.  #include <stdio.h>
2.
3.  int main() {
4.      int a, b, c, *pmax, *pmin; /*pmax,pmin 为整型指针变量*/
5.      printf("input three numbers:\n"); /*输入提示*/
6.      scanf("%d%d%d", &a, &b, &c); /*输入三个数字*/
7.      if(a > b) { /*如果第一个数字大于第二个数字. . . */
8.          pmax = &a; /*指针变量赋值*/
9.          pmin = &b; /*指针变量赋值*/
10.     }
11.     else {
12.         pmax = &b; /*指针变量赋值*/
13.         pmin = &a; /*指针变量赋值*/
14.     }
15.     if(c > *pmax) pmax = &c; /*判断并赋值*/
16.     if(c < *pmin) pmin = &c; /*判断并赋值*/
17.     printf("max=%d\nmin=%d\n", *pmax, *pmin); /*输出结果*/
18.     return 0;
19. }
```

6.4 通过指针引用数组

一个变量有一个地址，一个数组包含若干元素，每个数组元素都在内存中占用存储单元，它们都有相应的地址。所谓数组的指针是指数组的起始地址，数组元素的指针是数组元素的地址。

数组元素的指针

一个数组是由连续的一块内存单元组成的。数组名就是这块连续内存单元的首地址。一个数组也是由各个数组元素(下标变量)组成的。每个数组元素按其类型不同占有几个连续的内存单元。一个数组元素的首地址也是指它所占有的几个内存单元的首地址。

定义一个指向数组元素的指针变量的方法，与以前介绍的指针变量相同。

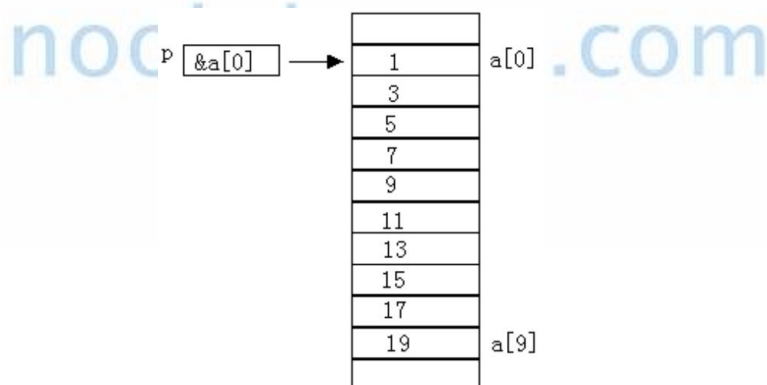
例如：

1. `int a[10];` /*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*/

应当注意，因为数组为 `int` 型，所以指针变量也应为指向 `int` 型的指针变量。下面是对指针变量赋值：

1. `p = &a[0];`

把 `a[0]` 元素的地址赋给指针变量 `p`。也就是说，`p` 指向 `a` 数组的第 0 号元素。



C 语言规定，数组名代表数组的首地址，也就是第 0 号元素的地址。因此，下面两个语句等价：

1. `p = &a[0];`
2. `p = a;`

在定义指针变量时可以赋给初值：

1. `int *p = &a[0];`

它等效于:

1. `int *p;`
2. `p = &a[0];`

当然定义时也可以写成:

1. `int *p = a;`

从图中我们可以看出有以下关系:

`p`, `a`, `&a[0]`均指向同一单元, 它们是数组 `a` 的首地址, 也是 0 号元素 `a[0]`的首地址。应该说明的是 `p` 是变量, 而 `a`, `&a[0]`都是常量。在编程时应予以注意。

数组指针变量说明的一般形式为:

1. 类型说明符 *指针变量名;

其中类型说明符表示所指数组的类型。从一般形式可以看出指向数组的指针变量和指向普通变量的指针变量的说明是相同的。

通过指针引用数组元素

C 语言规定: 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。引入指针变量后, 就可以用两种方法来访问数组元素了。

如果 `p` 的初值为`&a[0]`,则:

- 1) `p+i` 和 `a+i` 就是 `a[i]`的地址, 或者说它们指向 `a` 数组的第 `i` 个元素。
- 2) `*(p+i)`或`*(a+i)`就是 `p+i` 或 `a+i` 所指向的数组元素, 即 `a[i]`。例如, `*(p+5)`或`*(a+5)`就是 `a[5]`。
- 3) 指向数组的指针变量也可以带下标, 如 `p[i]`与`*(p+i)`等价。

根据以上叙述, 引用一个数组元素可以用:

- 1) 下标法, 即用 `a[i]`形式访问数组元素。在前面介绍数组时都是采用这种方法。
- 2) 指针法, 即采用`*(a+i)`或`*(p+i)`形式, 用间接访问的方法来访问数组元素, 其中 `a` 是数组名, `p` 是指向数组的指针变量, 其初值 `p=a`。

【例】输出数组中的全部元素。（下标法）

```
1. #include <stdio.h>
2.
3. int main() {
4.     int a[10], i;
5.     for(i = 0; i < 10; i++)
6.         a[i] = i;
7.     for(i = 0; i < 5; i++)
8.         printf("a[%d]=%d\n", i, a[i]);
9.     return 0;
10. }
```

【例】输出数组中的全部元素。（通过数组名计算元素的地址，找出元素的值）

```
1. #include <stdio.h>
2.
3. int main() {
4.     int a[10], i;
5.     for(i = 0; i < 10; i++)
6.         *(a + i) = i;
7.     for(i = 0; i < 10; i++)
8.         printf("a[%d]=%d\n", i, *(a + i));
9.     return 0;
10. }
```

【例】输出数组中的全部元素。（用指针变量指向元素）

```
1. #include <stdio.h>
2.
3. int main() {
4.     int a[10], i, *p;
5.     p = a;
6.     for(i = 0; i < 10; i++)
7.         *(p + i) = i;
8.     for(i = 0; i < 10; i++)
9.         printf("a[%d]=%d\n", i, *(p + i));
10.    return 0;
11. }
```

【例】

```
1. #include <stdio.h>
2.
3. int main() {
4.     int a[10], i, *p = a;
5.     for(i = 0; i < 10;){
6.         *p = i;
7.         printf("a[%d]=%d\n", i++, *p++);
8.     }
9.     return 0;
10. }
```

几个注意的问题：

- 1) 指针变量可以实现本身的值的改变。如 `p++` 是合法的；而 `a++` 是错误的。因为 `a` 是数组名，它是数组的首地址，是常量。
- 2) 要注意指针变量的当前值。请看下面的程序。

【例】找出错误。

```
1. #include <stdio.h>
2.
3. int main() {
4.     int *p, i, a[10];
5.     p = a;
6.     for(i = 0; i < 10; i++)
7.         *p++ = i;
8.     for(i = 0; i < 10; i++)
9.         printf("a[%d]=%d\n", i, *p++);
10.    return 0;
11. }
```

【例】改正。

```
1. #include <stdio.h>
2.
3. int main() {
4.     int *p, i, a[10];
5.     p = a;
6.     for(i = 0; i < 10; i++)
7.         *p++ = i;
8.     p = a;
```



```
9.     for(i = 0; i < 10; i++)
10.         printf("a[%d]=%d\n", i, *p++);
11.     return 0;
12. }
```

3) 从上例可以看出, 虽然定义数组时指定它包含 10 个元素, 但指针变量可以指到数组以后的内存单元, 系统并不认为非法。

4) $*p++$, 由于++和*同优先级, 结合方向自右而左, 等价于 $*(p++)$ 。

5) $*(p++)$ 与 $*(++p)$ 作用不同。若 p 的初值为 a, 则 $*(p++)$ 等价 a[0], $*(++p)$ 等价 a[1]。

6) $(*p)++$ 表示 p 所指向的元素值加 1。

7) 如果 p 当前指向 a 数组中的第 i 个元素, 则

1. $*(p--)$ 相当于 a[i--];
2. $*(++p)$ 相当于 a[++i];
3. $*(--p)$ 相当于 a[--i]。

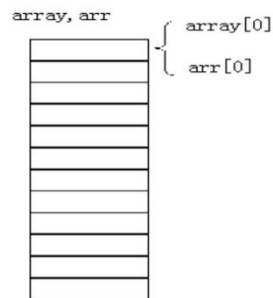
用数组名作为函数参数

数组名可以作函数的实参和形参。如:

```
1.  int main()
2.  {
3.      int array[10];
4.      .....
5.      .....
6.      f(array, 10);
7.      .....
8.      .....
9.  }
10.
11. f(int arr[], int n);
12. {
13.     .....
14.     .....
15. }
```

array 为实参数组名, arr 为形参数组名。在学习指针变量之后就更容易理解这个问题了。数组名就是数组的首地址, 实参向形参传送数组名实际上就是传送数组的地址, 形参得

到该地址后也指向同一数组。这就好象同一件物品有两个彼此不同的名称一样。



同样，指针变量的值也是地址，数组指针变量的值即为数组的首地址，当然也可作为函数的参数使用。

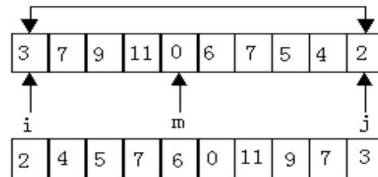
【例】

```

1. #include <stdio.h>
2.
3. float aver(float *pa);
4.
5. int main() {
6.     float sco[5], av, *sp;
7.     int i;
8.     sp = sco;
9.     printf("input 5 scores:\n");
10.    for(i = 0; i < 5; i++)
11.        scanf("%f", &sco[i]);
12.    av = aver(sp);
13.    printf("average score is %5.2f", av);
14.    return 0;
15. }
16.
17. float aver(float *pa) {
18.     int i;
19.     float av, s = 0;
20.     for(i = 0; i < 5; i++)
21.         s = s + *pa++;
22.     av = s / 5;
23.     return av;
24. }
```

【例】将数组 a 中的 n 个整数按相反顺序存放。

算法为：将 $a[0]$ 与 $a[n-1]$ 对换，再 $a[1]$ 与 $a[n-2]$ 对换……,直到将 $a[(n-1)/2]$ 与 $a[n-\text{int}((n-1)/2)]$ 对换。今用循环处理此问题，设两个“位置指示变量” i 和 j ， i 的初值为 0， j 的初值为 $n-1$ 。将 $a[i]$ 与 $a[j]$ 交换，然后使 i 的值加 1， j 的值减 1，再将 $a[i]$ 与 $a[j]$ 交换，直到 $i=(n-1)/2$ 为止，如图所示。



程序如下：

```

1. #include <stdio.h>
2.
3. void inv(int x[], int n) /*形参 x 是数组名*/
4. {
5.     int temp, i, j, m = (n - 1) / 2;
6.     for(i = 0; i <= m; i++) {
7.         j = n - 1 - i;
8.         temp = x[i];
9.         x[i] = x[j];
10.        x[j] = temp;
11.    }
12.    return;
13. }
14.
15. int main() {
16.     int i, a[10] = {3,7,9,11,0,6,7,5,4,2};
17.     printf("The original array:\n");
18.     for(i = 0; i < 10; i++)
19.         printf("%d,", a[i]);
20.     printf("\n");
21.     inv(a, 10);
22.     printf("The array has benn inverted:\n");
23.     for(i = 0; i < 10; i++)
24.         printf("%d,", a[i]);
25.     printf("\n");
26.     return 0;
27. }
```

对此程序可以作一些改动。将函数 `inv` 中的形参 `x` 改成指针变量。

【例】对上例可以作一些改动。将函数 `inv` 中的形参 `x` 改成指针变量。

程序如下：

```
1. #include <stdio.h>
2.
3. void inv(int *x, int n) /*形参 x 为指针变量*/
4. {
5.     int *p, temp, *i, *j, m = (n - 1) / 2;
6.     i = x;
7.     j = x + n - 1;
8.     p = x + m;
9.     for( ; i <= p; i++, j--) {
10.         temp = *i;
11.         *i = *j;
12.         *j = temp;
13.     }
14.     return;
15. }
16.
17. int main() {
18.     int i, a[10] = {3,7,9,11,0,6,7,5,4,2};
19.     printf("The original array:\n");
20.     for(i = 0; i < 10; i++)
21.         printf("%d,", a[i]);
22.     printf("\n");
23.     inv(a, 10);
24.     printf("The array has benn inverted:\n");
25.     for(i = 0; i < 10; i++)
26.         printf("%d,", a[i]);
27.     printf("\n");
28.     return 0;
29. }
```

运行情况与前一程序相同。

【例】从 0 个数中找出其中最大值和最小值。

调用一个函数只能得到一个返回值，今用全局变量在函数之间“传递”数据。程序如下：

```
1. #include <stdio.h>
2.
```

```

3.  int max, min; /*全局变量*/
4.  void max_min_value(int array[], int n) {
5.      int *p, *array_end;
6.      array_end = array+n;
7.      max = min = *array;
8.      for(p = array + 1; p < array_end; p++)
9.          if(*p > max) max = *p;
10.         else if (*p < min) min = *p;
11.     return;
12. }
13. int main() {
14.     int i, number[10];
15.     printf("enter 10 integer umbers:\n");
16.     for(i = 0; i < 10; i++)
17.         scanf("%d", &number[i]);
18.     max_min_value(number, 10);
19.     printf("max=%d,min=%d\n", max, min);
20.     return 0;
21. }

```

说明:

1) 在函数 max_min_value 中求出的最大值和最小值放在 max 和 min 中。由于它们是全局, 因此主函数中可以直接使用。

2) 函数 max_min_value 中的语句:

```
1. max = min = *array;
```

array 是数组名, 它接收从实参传来的数组 number 的首地址。

array 相当于(&array[0])。上述语句与 max=min=array[0];等价。

3) 在执行 for 循环时, p 的初值为 array+1, 也就是使 p 指向 array[1]。以后每次执行 p++, 使 p 指向下一个元素。每次将*p 和 max 与 min 比较。将大者放入 max, 小者放 min。

4) 函数 max_min_value 的形参 array 可以改为指针变量类型。实参也可以不用数组名, 而用指针变量传递地址。

上例程序可改为:

```

1.  #include <stdio.h>
2.
3.  int max, min; /*全局变量*/
4.  void max_min_value(int *array, int n) {
5.      int *p, *array_end;

```

```

6.     array_end = array + n;
7.     max = min = *array;
8.     for(p = array + 1; p < array_end; p++)
9.         if(*p > max) max = *p;
10.        else if (*p < min) min = *p;
11.     return;
12. }
13.
14. int main() {
15.     int i, number[10], *p;
16.     p = number; /*使 p 指向 number 数组*/
17.     printf("enter 10 integer umbers:\n");
18.     for(i = 0; i < 10; i++, p++)
19.         scanf("%d", p);
20.     p = number;
21.     max_min_value(p, 10);
22.     printf("max=%d,min=%d\n", max, min);
23.     return 0;
24. }

```

归纳起来，如果有一个实参数组，想在函数中改变此数组的元素的值，实参与形参的对应关系有以下4种：

1) 形参和实参都是数组名。

```

1.  int main()
2.  {
3.      int a[10];
4.      .....
5.      f(a, 10)
6.      .....
7.      f(int x[], int n)
8.      {
9.          .....
10.     }
11.     return 0;
12. }

```

a 和 x 指的是同一组数组。

2) 实用数组，形参用指针变量。

```

1.  int main()

```

```
2. {  
3.     int a[10];  
4.     .....  
5.     f(a,10)  
6.     .....  
7.     f(int *x,int n)  
8.     {  
9.         .....  
10.    }  
11.    return 0;  
12. }
```

3) 实参、型参都用指针变量。

4) 实参为指针变量，型参为数组名。

【例】用实参指针变量改写将 n 个整数按相反顺序存放。

```
1. #include <stdio.h>  
2.  
3. void inv(int *x, int n) {  
4.     int *p, m, temp, *i, *j;  
5.     m = (n - 1) / 2;  
6.     i = x;  
7.     j = x + n - 1;  
8.     p = x + m;  
9.     for( ; i <= p; i++, j--) {  
10.        temp = *i;  
11.        *i = *j;  
12.        *j = temp;  
13.    }  
14.    return;  
15. }  
16.  
17. int main() {  
18.     int i, arr[10] = {3,7,9,11,0,6,7,5,4,2},*p;  
19.     p = arr;  
20.     printf("The original array:\n");  
21.     for(i = 0; i < 10; i++, p++)  
22.         printf("%d,", *p);  
23.     printf("\n");  
24.     p = arr;
```

```
25.     inv(p, 10);
26.     printf("The array has benn inverted:\n");
27.     for(p = arr; p < arr + 10; p++)
28.         printf("%d,", *p);
29.     printf("\n");
30.     return 0;
31. }
```

注意：`main` 函数中的指针变量 `p` 是有确定值的。即如果用指针变作实参，必须现使指针变量有确定值，指向一个已定义的数组。

【例】用选择法对 10 个整数排序。

```
1.  #include <stdio.h>
2.
3.  void sort(int x[], int n) {
4.      int i, j, k, t;
5.      for(i = 0; i < n-1; i++) {
6.          k = i;
7.          for(j = i + 1; j < n; j++)
8.              if(x[j] > x[k]) k = j;
9.          if(k != i) {
10.             t = x[i];
11.             x[i] = x[k];
12.             x[k] = t;
13.          }
14.      }
15. }
16.
17. int main() {
18.     int *p, i, a[10] = {3,7,9,11,0,6,7,5,4,2};
19.     printf("The original array:\n");
20.     for(i = 0; i < 10; i++)
21.         printf("%d,", a[i]);
22.     printf("\n");
23.     p = a;
24.     sort(p, 10);
25.     for(p = a, i = 0; i < 10; i++) {
26.         printf("%d ", *p);
27.         p++;
28.     }
```



```
29.    printf("\n");
30.    return 0;
31. }
```

说明：函数 `sort` 用数组名作为形参，也可改为用指针变量，这时函数的首部可以改为：
`sort(int *x,int n)` 其他可一律不改。

通过指针引用多维数组

本小节以二维数组为例介绍多维数组的指针变量。

1. 多维数组的地址

设有整型二维数组 `a[3][4]`如下：

0 1 2 3

4 5 6 7

8 9 10 11

它的定义为：

```
1. int a[3][4] = {{0,1,2,3},{4,5,6,7},{8,9,10,11}}
```

设数组 `a` 的首地址为 1000，各下标变量的首地址及其值如图所示。

1000 0	1002 1	1004 2	1006 3
1008 4	1010 5	1012 6	1014 7
1016 8	1018 9	1020 11	1022 12

前面介绍过，C语言允许把一个二维数组分解为多个一维数组来处理。因此数组 `a` 可分解为三个一维数组，即 `a[0]`，`a[1]`，`a[2]`。每一个一维数组又含有四个元素。

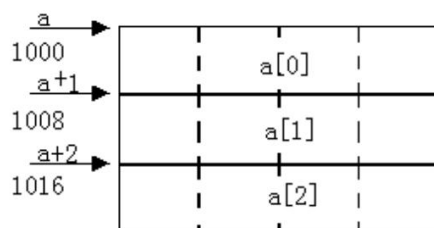
\xrightarrow{a}	<code>a[0]</code>	=	1000 0	1002 1	1004 2	1006 3
	<code>a[1]</code>	=	1008 4	1010 5	1012 6	1014 7
	<code>a[2]</code>	=	1016 8	1018 9	1020 11	1022 12

例如 `a[0]`数组，含有 `a[0][0]`，`a[0][1]`，`a[0][2]`，`a[0][3]`四个元素。

数组及数组元素的地址表示如下：

从二维数组的角度来看，`a` 是二维数组名，`a` 代表整个二维数组的首地址，也是二维数

组 0 行的首地址, 等于 1000。a+1 代表第一行的首地址, 等于 1008。如图:



a[0]是第一个一维数组的数组名和首地址, 因此也为 1000。*(a+0)或*a 是与 a[0]等效的, 它表示一维数组 a[0]0 号元素的首地址, 也为 1000。&a[0][0]是二维数组 a 的 0 行 0 列元素首地址, 同样是 1000。因此, a, a[0], *(a+0), *a, &a[0][0]是相等的。

同理, a+1 是二维数组 1 行的首地址, 等于 1008。a[1]是第二个一维数组的数组名和首地址, 因此也为 1008。&a[1][0]是二维数组 a 的 1 行 0 列元素地址, 也是 1008。因此 a+1, a[1], *(a+1), &a[1][0]是等同的。

由此可得出: a+i, a[i], *(a+i), &a[i][0]是等同的。

此外, &a[i]和 a[i]也是等同的。因为在二维数组中不能把&a[i]理解为元素 a[i]的地址, 不存在元素 a[i]。C 语言规定, 它是一种地址计算方法, 表示数组 a 第 i 行首地址。

由此, 我们得出: a[i], &a[i], *(a+i)和 a+i 也都是等同的。

另外, a[0]也可以看成是 a[0]+0, 是一维数组 a[0]的 0 号元素的首地址, 而 a[0]+1 则是 a[0]的 1 号元素首地址, 由此可得出 a[i]+j 则是一维数组 a[i]的 j 号元素首地址, 它等于 &a[i][j]。

	a[0]	a[0]+1	a[0]+2	a[0]+3
a	1000 0	1002 1	1004 2	1006 3
a+1	1008 4	1010 5	1012 6	1014 7
a+2	1016 8	1018 9	1020 11	1022 12

由 a[i]=*(a+i)得 a[i]+j=*(a+i)+j。由于*(a+i)+j 是二维数组 a 的 i 行 j 列元素的首地址, 所以, 该元素的值等于*(*(a+i)+j)。

【例】

```
1. #include <stdio.h>
2.
3. int main() {
```

```

4.     int a[3][4]={0,1,2,3,4,5,6,7,8,9,10,11};
5.     printf("%d", a);
6.     printf("%d", *a);
7.     printf("%d", a[0]);
8.     printf("%d", &a[0]);
9.     printf("%d\n", &a[0][0]);
10.    printf("%d", a+1);
11.    printf("%d", *(a+1));
12.    printf("%d", a[1]);
13.    printf("%d", &a[1]);
14.    printf("%d\n", &a[1][0]);
15.    printf("%d", a+2);
16.    printf("%d", *(a+2));
17.    printf("%d", a[2]);
18.    printf("%d", &a[2]);
19.    printf("%d\n", &a[2][0]);
20.    printf("%d", a[1]+1);
21.    printf("%d\n", *(a+1)+1);
22.    printf("%d,%d\n", *(a[1]+1), (*(a+1)+1));
23.    return 0;
24. }
```

2. 指向多维数组的指针变量

把二维数组 `a` 分解为一维数组 `a[0],a[1],a[2]`之后, 设 `p` 为指向二维数组的指针变量。可定义为:

```
1. int (*p)[4]
```

它表示 `p` 是一个指针变量, 它指向包含 4 个元素的一维数组。若指向第一个一维数组 `a[0]`, 其值等于 `a,a[0]`, 或`&a[0][0]`等。而 `p+i` 则指向一维数组 `a[i]`。从前面的分析可得出`*(p+i)+j` 是二维数组 `i` 行 `j` 列的元素的地址, 而`*(*(p+i)+j)`则是 `i` 行 `j` 列元素的值。

二维数组指针变量说明的一般形式为:

```
1. 类型说明符 (*指针变量名)[长度]
```

其中“类型说明符”为所指数组的数据类型。“*”表示其后的变量是指针类型。“长度”表示二维数组分解为多个一维数组时, 一维数组的长度, 也就是二维数组的列数。应注意“(*指针变量名)”两边的括号不可少, 如缺少括号则表示是指针数组(本章后面介绍), 意义就完全不同了。

【例】

```
1. #include <stdio.h>
2.
3. int main() {
4.     int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
5.     int(*p)[4];
6.     int i, j;
7.     p = a;
8.     for(i = 0; i < 3; i++) {
9.         for(j = 0; j < 4; j++)
10.            printf("%2d ", (*(p + i) + j));
11.        printf("\n");
12.    }
13.    return 0;
14. }
```



6.5 通过指针引用字符串

字符串的引用方式

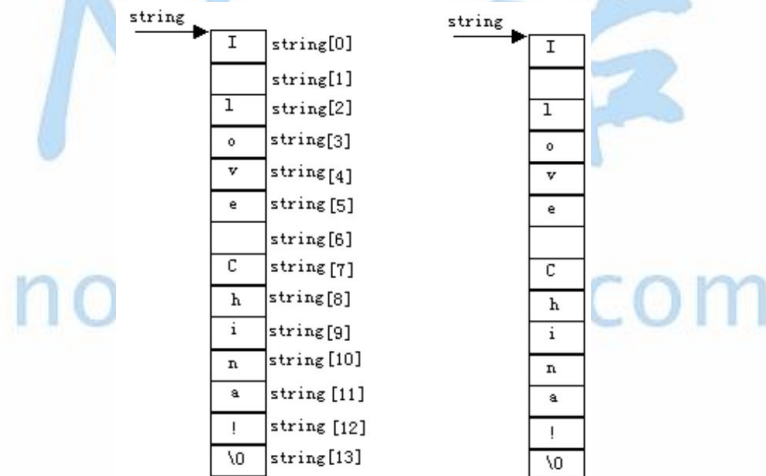
在 C 语言中, 可以用两种方法访问一个字符串。

1) 用字符数组存放一个字符串, 然后输出该字符串。

【例】

```
1. #include <stdio.h>
2.
3. int main() {
4.     char string[] = "Hello NoobDream!";
5.     printf("%s\n", string);
6.     return 0;
7. }
```

说明: 和前面介绍的数组属性一样, `string` 是数组名, 它代表字符数组的首地址。



2) 用字符串指针指向一个字符串。

【例】

```
1. #include <stdio.h>
2.
3. int main() {
4.     char *string = "Hello NoobDream!";
5.     printf("%s\n", string);
6.     return 0;
7. }
```

字符串指针变量的定义说明与指向字符变量的指针变量说明是相同的。只能按对指针变量的赋

值不同来区别。对指向字符变量的指针变量应赋予该字符变量的地址。

如:

```
1. char c, *p = &c;
```

表示 `p` 是一个指向字符变量 `c` 的指针变量。

而:

```
1. char *s = "C Language";
```

则表示 `s` 是一个指向字符串的指针变量。把字符串的首地址赋予 `s`。

上例中, 首先定义 `string` 是一个字符指针变量, 然后把字符串的首地址赋予 `string`(应写出整个字符串, 以便编译系统把该串装入连续的一块内存单元), 并把首地址送入 `string`。

程序中的:

```
1. char *ps = "C Language";
```

等效于:

```
1. char *ps;  
2. ps = "C Language";
```

【例】输出字符串中 `n` 个字符后的所有字符。

```
1. #include <stdio.h>  
2.  
3. int main() {  
4.     char *ps = "this is a book";  
5.     int n = 10;  
6.     ps = ps + n;  
7.     printf("%s\n", ps);  
8.     return 0;  
9. }
```

运行结果为:

```
1. book
```

在程序中对 `ps` 初始化时, 即把字符串首地址赋予 `ps`, 当 `ps=ps+10` 之后, `ps` 指向字符“b”, 因此输出为“book”。

【例】在输入的字符串中查找有无‘k’字符。

```
1. #include <stdio.h>  
2.  
3. int main() {  
4.     char st[20], *ps;
```

```

5.     int i;
6.     printf("input a string:\n");
7.     ps = st;
8.     scanf("%s", ps);
9.     for(i = 0; ps[i] != '\0'; i++)
10.        if(ps[i] == 'k'){
11.            printf("there is a 'k' in the string\n");
12.            break;
13.        }
14.     if(ps[i] == '\0') printf("There is no 'k' in the string\n");
15.     return 0;
16. }

```

【例】本例是将指针变量指向一个格式字符串，用在 `printf` 函数中，用于输出二维数组的各种地址表示的值。但在 `printf` 语句中用指针变量 `PF` 代替了格式串。这也是程序中常用的方法。

```

1.  #include <stdio.h>
2.
3.  int main() {
4.      static int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
5.      char *PF;
6.      PF = "%d,%d,%d,%d,%d\n";
7.      printf(PF, a, *a, a[0], &a[0], &a[0][0]);
8.      printf(PF, a+1, *(a+1), a[1], &a[1], &a[1][0]);
9.      printf(PF, a+2, *(a+2), a[2], &a[2], &a[2][0]);
10.     printf("%d,%d\n", a[1]+1, *(a+1)+1);
11.     printf("%d,%d\n", *(a[1]+1), (*(a+1)+1));
12.     return 0;
13. }

```

【例】本例是把字符串指针作为函数参数的使用。要求把一个字符串的内容复制到另一个字符串中，并且不能使用 `strcpy` 函数。函数 `cprstr` 的形参为两个字符指针变量。`pss` 指向源字符串，`pds` 指向目标字符串。注意表达式：`(*pds=*pss)!='\0'` 的用法。

```

1.  #include <stdio.h>
2.
3.  void cprstr(char *pss, char *pds) {
4.      while((*pds = *pss) != '\0'){

```

```

5.         pds++;
6.         pss++;
7.     }
8. }
9.
10. int main() {
11.     char *pa = "CHINA", b[10], *pb;
12.     pb = b;
13.     cpustr(pa, pb);
14.     printf("string a=%s\nstring b=%s\n", pa, pb);
15.     return 0;
16. }

```

在本例中，程序完成了两项工作：一是把 pss 指向的源字符串复制到 pds 所指向的目标字符串中，二是判断所复制的字符是否为'\0'，若是则表明源字符串结束，不再循环。否则，pds 和 pss 都加 1，指向下一字符。在主函数中，以指针变量 pa, pb 为实参，分别取得确定值后调用 cpustr 函数。由于采用的指针变量 pa 和 pss, pb 和 pds 均指向同一字符串，因此主函数和 cpustr 函数中均可使用这些字符串。也可以把 cpustr 函数简化为以下形式：

```

1. void cpustr(char *pss, char *pds) {
2.     while ((*pds++ = *pss++) != '\0');
3. }

```

即把指针的移动和赋值合并在一个语句中。进一步分析还可发现'\0'的 ASCII 码为 0，对于 while 语句只看表达式的值为非 0 就循环，为 0 则结束循环，因此也可省去“!= '\0'”这一判断部分，而写为以下形式：

```

1. void cpustr (char *pss, char *pds) {
2.     while (*pds++ = *pss++);
3. }

```

表达式的意义可解释为，源字符向目标字符赋值，移动指针，若所赋值为非 0 则循环，否则结束循环。这样使程序更加简洁。

【例】简化后的程序如下所示。

```

1. #include <stdio.h>
2.
3. void cpustr (char *pss, char *pds) {
4.     while (*pds++ = *pss++);
5. }

```



```
6.  
7. int main() {  
8.     char *pa = "CHINA", b[10], *pb;  
9.     pb = b;  
10.    strcpy(pa, pb);  
11.    printf("string a=%s\nstring b=%s\n", pa, pb);  
12.    return 0;  
13. }
```

使用字符指针变量和字符数组的比较

用字符数组和字符指针变量都可实现字符串的存储和运算。但是两者是有区别的。在使用时应注意以下几个问题:

1. 字符串指针变量本身是一个变量, 用于存放字符串的首地址。而字符串本身是存放在以该首地址为首的一块连续的内存空间中并以 ‘\0’ 作为串的结束。字符数组是由于若干个数组元素组成的, 它可用来存放整个字符串。

2. 对字符串指针方式

```
1. char *ps = "C Language";
```

可以写为:

```
1. char *ps;  
2. ps = "C Language";
```

而对数组方式:

```
1. static char st[] = {"C Language"};
```

不能写为:

```
1. char st[20];  
2. st = {"C Language"};
```

而只能对字符数组的各元素逐个赋值。

从以上几点可以看出字符串指针变量与字符数组在使用时的区别, 同时也可看出使用指针变量更加方便。前面说过, 当一个指针变量在未取得确定地址前使用是危险的, 容易引起错误。但是对指针变量直接赋值是可以的。因为 C 系统对指针变量赋值时要给以确定的地址。

因此,

```
1. char *ps = "C Language";
```

或者

```
1. char *ps;  
2. ps = "C Language";
```

都是合法的。

N 诺

noobdream.com

*6.6 指向函数的指针

什么是函数的指针

用函数指针变量调用函数

怎样定义和使用指向函数的指针变量

在C语言中, 一个函数总是占用一段连续的内存区, 而函数名就是该函数所占内存区的首地址。我们可以把函数的这个首地址(或称入口地址)赋予一个指针变量, 使该指针变量指向该函数。然后通过指针变量就可以找到并调用这个函数。我们把这种指向函数的指针变量称为“函数指针变量”。

函数指针变量定义的一般形式为:

1. 类型说明符 (*指针变量名)();

其中“类型说明符”表示被指函数的返回值的类型。“(* 指针变量名)”表示“*”后面的变量是定义的指针变量。最后的空括号表示指针变量所指的是一个函数。

例如:

1. `int (*pf)();`

表示 pf 是一个指向函数入口的指针变量, 该函数的返回值(函数值)是整型。

【例】本例用来说明用指针形式实现对函数调用的方法。

```
1. #include <stdio.h>
2.
3. int max(int a, int b){
4.     if(a > b) return a;
5.     else return b;
6. }
7.
8. int main() {
9.     int max(int a, int b);
10.    int(*pmax)(int a, int b);
11.    int x, y, z;
12.    pmax = max;
13.    printf("input two numbers:\n");
14.    scanf("%d%d", &x, &y);
15.    z = (*pmax)(x, y);
16.    printf("maximum=%d", z);
17.    return 0;
```

18. }

从上述程序可以看出用，函数指针变量形式调用函数的步骤如下：

- 1) 先定义函数指针变量，如后一程序中第 9 行 `int (*pmax)();` 定义 `pmax` 为函数指针变量。
- 2) 把被调函数的入口地址(函数名)赋予该函数指针变量，如程序中第 11 行 `pmax=max;`
- 3) 用函数指针变量形式调用函数，如程序第 14 行 `z=(*pmax)(x,y);`
- 4) 调用函数的一般形式为：

1. (*指针变量名)(实参表)

使用函数指针变量还应注意以下两点：

- a) 函数指针变量不能进行算术运算，这是与数组指针变量不同的。数组指针变量加减一个整数可使指针移动指向后面或前面的数组元素，而函数指针的移动是毫无意义的。
- b) 函数调用中“(*指针变量名)”的两边的括号不可少，其中的*不应该理解为求值运算，在此处它只是一种表示符号。

noobdream.com

*6.7 返回指针值的函数

前面我们介绍过, 所谓函数类型是指函数返回值的类型。在 C 语言中允许一个函数的返回值是一个指针(即地址), 这种返回指针值的函数称为指针型函数。

定义指针型函数的一般形式为:

1. 类型说明符 *函数名(形参表)
2. {
3. /*函数体*/
4. }

其中函数名之前加了“*”号表明这是一个指针型函数, 即返回值是一个指针。类型说明符表示了返回的指针值所指向的数据类型。

如:

1. `int *ap(int x, int y)`
2. {
3. /*函数体*/
4. }

表示 `ap` 是一个返回指针值的指针型函数, 它返回的指针指向一个整型变量。

【例】本程序是通过指针函数, 输入一个 1~7 之间的整数, 输出对应的星期名。

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. int main(){
5.     int i;
6.     char *day_name(int n);
7.     printf("input Day No:\n");
8.     scanf("%d", &i);
9.     if(i < 0) exit(0);
10.    printf("Day No:%2d-->%s\n", i, day_name(i));
11.    return 0;
12. }
13.
14. char *day_name(int n) {
15.     static char *name[] = {
16.         "Illegal day",
17.         "Monday",
18.         "Tuesday",
19.         "Wednesday",
```

```
20.     "Thursday",
21.     "Friday",
22.     "Saturday",
23.     "Sunday"
24. };
25.     return((n < 1 || n > 7) ? name[0] : name[n]);
26. }
```

本例中定义了一个指针型函数 `day_name`，它的返回值指向一个字符串。该函数中定义了一个静态指针数组 `name`。`name` 数组初始化赋值为八个字符串，分别表示各个星期名及出错提示。形参 `n` 表示与星期名所对应的整数。在主函数中，把输入的整数 `i` 作为实参，在 `printf` 语句中调用 `day_name` 函数并把 `i` 值传送给形参 `n`。`day_name` 函数中的 `return` 语句包含一个条件表达式，`n` 值若大于 7 或小于 1 则把 `name[0]` 指针返回主函数输出出错提示字符串 “Illegal day”。否则返回主函数输出对应的星期名。主函数中的第 7 行是个条件语句，其语义是，如输入为负数($i < 0$)则中止程序运行退出程序。`exit` 是一个库函数，`exit(1)` 表示发生错误后退出程序，`exit(0)` 表示正常退出。

应该特别注意的是函数指针变量和指针型函数这两者在写法和意义上的区别。如 `int(*p)()` 和 `int *p()` 是两个完全不同的量。

`int(*p)()` 是一个变量说明，说明 `p` 是一个指向函数入口的指针变量，该函数的返回值是整型量，`(*p)` 的两边的括号不能少。

`int *p()` 则不是变量说明而是函数说明，说明 `p` 是一个指针型函数，其返回值是一个指向整型量的指针，`*p` 两边没有括号。作为函数说明，在括号内最好写入形式参数，这样便于与变量说明区别。

对于指针型函数定义，`int *p()` 只是函数头部分，一般还应该有函数体部分。

*6.8 指针数组和多重指针

什么是指针数组

一个数组的元素值为指针则是指针数组。指针数组是一组有序的指针的集合。指针数组的所有元素都必须是具有相同存储类型和指向相同数据类型的指针变量。

指针数组说明的一般形式为:

1. 类型说明符 *数组名[数组长度]

其中类型说明符为指针值所指向的变量的类型。

例如:

1. `int *pa[3]`

表示 `pa` 是一个指针数组, 它有三个数组元素, 每个元素值都是一个指针, 指向整型变量。

【例】通常可用一个指针数组来指向一个二维数组。指针数组中的每个元素被赋予二维数组每一行的首地址, 因此也可理解为指向一个一维数组。

```
1. #include <stdio.h>
2.
3. int main(){
4.     int a[3][3] = {1,2,3,4,5,6,7,8,9};
5.     int *pa[3] = {a[0],a[1],a[2]};
6.     int *p = a[0];
7.     int i;
8.     for(i = 0; i < 3; i++)
9.         printf("%d,%d,%d\n", a[i][2-i], *a[i], (*(a+i)+i));
10.    for(i = 0; i < 3; i++)
11.        printf("%d,%d,%d\n", *pa[i], p[i], *(p+i));
12.    return 0;
13. }
```

本例程序中, `pa` 是一个指针数组, 三个元素分别指向二维数组 `a` 的各行。然后用循环语句输出指定的数组元素。其中 `*a[i]` 表示 `i` 行 0 列元素值; `*(a+i)+i` 表示 `i` 行 `i` 列的元素值; `*pa[i]` 表示 `i` 行 0 列元素值; 由于 `p` 与 `a[0]` 相同, 故 `p[i]` 表示 0 行 `i` 列的值; `*(p+i)` 表示 0 行 `i` 列的值。读者可仔细领会元素值的各种不同的表示方法。

应该注意指针数组和二维数组指针变量的区别。这两者虽然都可用来表示二维数组, 但是其表示方法和意义是不同的。

二维数组指针变量是单个的变量, 其一般形式中"(*指针变量名)"两边的括号不可少。而指

针数组类型表示的是多个指针(一组有序指针)在一般形式中"*指针数组名"两边不能有括号。

例如:

```
1. int (*p)[3];
```

表示一个指向二维数组的指针变量。该二维数组的列数为 3 或分解为一维数组的长度为 3。

```
1. int *p[3]
```

表示 p 是一个指针数组, 有三个下标变量 p[0], p[1], p[2]均为指针变量。

指针数组也常用来表示一组字符串, 这时指针数组的每个元素被赋予一个字符串的首地址。指向字符串的指针数组的初始化更为简单。例如在例 10.32 中即采用指针数组来表示一组字符串。其初始化赋值为:

```
1. char *name[] = {
2.     "Illegal day",
3.     "Monday",
4.     "Tuesday",
5.     "Wednesday",
6.     "Thursday",
7.     "Friday",
8.     "Saturday",
9.     "Sunday"
10.    };
```

完成这个初始化赋值之后, name[0]即指向字符串"Illegal day", name[1]指向"Monday".....。

noobdream.com

指针数组也可以用作函数参数

【例】指针数组作指针型函数的参数。在本例主函数中, 定义了一个指针数组 name, 并对 name 作了初始化赋值。其每个元素都指向一个字符串。然后又以 name 作为实参调用指针型函数 day_name, 在调用时把数组名 name 赋予形参变量 name, 输入的整数 i 作为第二个实参赋予形参 n。在 day_name 函数中定义了两个指针变量 pp1 和 pp2, pp1 被赋予 name[0]的值(即*name), pp2 被赋予 name[n]的值即*(name+ n)。由条件表达式决定返回 pp1 或 pp2 指针给主函数中的指针变量 ps。最后输出 i 和 ps 的值。

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. int main() {
5.     static char *name[] = {
```



```

6.     "Illegal day",
7.     "Monday",
8.     "Tuesday",
9.     "Wednesday",
10.    "Thursday",
11.    "Friday",
12.    "Saturday",
13.    "Sunday"
14. };
15. char *ps;
16. int i;
17. char *day_name(char *name[], int n);
18. printf("input Day No:\n");
19. scanf("%d", &i);
20. if(i < 0) exit(1);
21. ps = day_name(name,i);
22. printf("Day No:%2d-->%s\n", i, ps);
23. return 0;
24. }
25.
26. char *day_name(char *name[], int n) {
27.     char *pp1, *pp2;
28.     pp1 = *name;
29.     pp2 = *(name+n);
30.     return((n < 1 || n > 7) ? pp1 : pp2);
31. }

```

【例】输入 5 个国名并按字母顺序排列后输出。现编程如下：

```

1. #include <stdio.h>
2. #include <string.h>
3.
4. int main() {
5.     void sort(char *name[], int n);
6.     void print(char *name[], int n);
7.     static char *name[] = {
8.         "CHINA", "AMERICA", "AUSTRALIA",
9.         "FRANCE", "GERMAN"};
10.    int n = 5;
11.    sort(name, n);
12.    print(name, n);
13. }
14.

```

```

15. void sort(char *name[], int n) {
16.     char *pt;
17.     int i, j, k;
18.     for(i = 0; i < n-1; i++){
19.         k = i;
20.         for(j = i + 1; j < n; j++)
21.             if(strcmp(name[k], name[j]) > 0) k = j;
22.         if(k != i){
23.             pt = name[i];
24.             name[i] = name[k];
25.             name[k] = pt;
26.         }
27.     }
28. }
29.
30. void print(char *name[], int n){
31.     int i;
32.     for (i = 0; i < n; i++)
33.         printf("%s\n", name[i]);
34. }

```

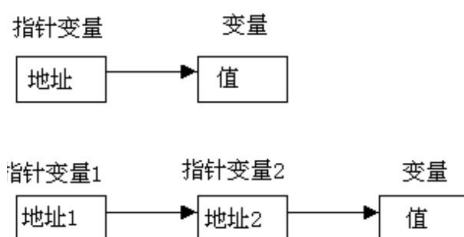
说明:

在以前的例子中采用了普通的排序方法, 逐个比较之后交换字符串的位置。交换字符串的物理位置是通过字符串复制函数完成的。反复的交换将使程序执行的速度很慢, 同时由于各字符串(国名)的长度不同, 又增加了存储管理的负担。用指针数组能很好地解决这些问题。把所有的字符串存放在一个数组中, 把这些字符数组的首地址放在一个指针数组中, 当需要交换两个字符串时, 只须交换指针数组相应两元素的内容(地址)即可, 而不必交换字符串本身。

本程序定义了两个函数, 一个名为 `sort` 完成排序, 其形参为指针数组 `name`, 即为待排序的各字符串数组的指针。形参 `n` 为字符串的个数。另一个函数名为 `print`, 用于排序后字符串的输出, 其形参与 `sort` 的形参相同。主函数 `main` 中, 定义了指针数组 `name` 并作了初始化赋值。然后分别调用 `sort` 函数和 `print` 函数完成排序和输出。值得说明的是在 `sort` 函数中, 对两个字符串比较, 采用了 `strcmp` 函数, `strcmp` 函数允许参与比较的字符串以指针方式出现。`name[k]`和 `name[j]`均为指针, 因此是合法的。字符串比较后需要交换时, 只交换指针数组元素的值, 而不交换具体的字符串, 这样将大大减少时间的开销, 提高了运行效率。

指向指针数据的指针变量

如果一个指针变量存放的又是另一个指针变量的地址, 则称这个指针变量为指向指针的指针变量。在前面已经介绍过, 通过指针访问变量称为间接访问。由于指针变量直接指向变量, 所以称为“单级间址”。而如果通过指向指针的指针变量来访问变量则构成“二级间址”。

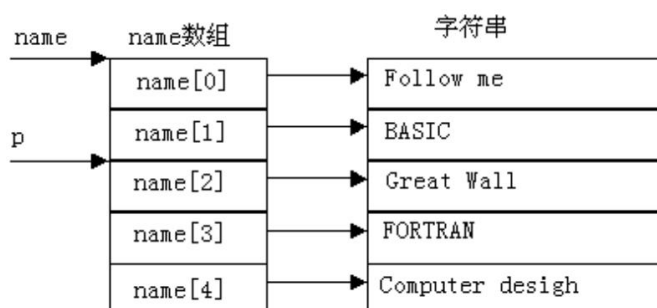


从下图可以看到, `name` 是一个指针数组, 它的每一个元素是一个指针型数据, 其值为地址。`Name` 是一个数据, 它的每一个元素都有相应的地址。数组名 `name` 代表该指针数组的首地址。`name+1` 是 `mane[i]` 的地址。`name+1` 就是指向指针型数据的指针 (地址)。还可以设置一个指针变量 `p`, 使它指向指针数组元素。`P` 就是指向指针型数据的指针变量。怎样定义一个指向指针型数据的指针变量呢? 如下:

1. `char **p;`

`p` 前面有两个*号, 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, 表示指针变量 `p` 是指向一个字符指针型变量的。`*p` 就是 `p` 所指向的另一个指针变量。

从下图可以看到, `name` 是一个指针数组, 它的每一个元素是一个指针型数据, 其值为地址。`name` 是一个数组, 它的每一个元素都有相应的地址。数组名 `name` 代表该指针数组的首地址。`name+1` 是 `mane[i]` 的地址。`name+1` 就是指向指针型数据的指针 (地址)。还可以设置一个指针变量 `p`, 使它指向指针数组元素。`P` 就是指向指针型数据的指针变量。



如果有:

```
1. p = name + 2;
2. printf("%o\n", *p);
```

```
3. printf("%s\n", *p);
```

则, 第一个 `printf` 函数语句输出 `name[2]` 的值 (它是一个地址), 第二个 `printf` 函数语句以字符串形式 (`%s`) 输出字符串 “Great Wall”。

【例】使用指向指针的指针。

```
1. #include <stdio.h>
2.
3. int main() {
4.     char *name[] = {
5.         "Follow me", "BASIC", "Great Wall",
6.         "FORTRAN", "Computer design"
7.     };
8.     char **p;
9.     int i;
10.    for(i = 0; i < 5; i++) {
11.        p = name + i;
12.        printf("%s\n", *p);
13.    }
14.    return 0;
15. }
```

说明: `p` 是指向指针的指针变量。

【例】一个指针数组的元素指向数据的简单例子。

```
1. #include <stdio.h>
2.
3. int main() {
4.     static int a[5] = {1,3,5,7,9};
5.     int *num[5] = {&a[0], &a[1], &a[2], &a[3], &a[4]};
6.     int **p, i;
7.     p = num;
8.     for(i = 0; i < 5; i++) {
9.         printf("%d\t", **p);
10.        p++;
11.    }
12.    return 0;
13. }
```

说明: 指针数组的元素只能存放地址。

指针数组做 main 函数的形参

前面介绍的 main 函数都是不带参数的。因此 main 后的括号都是空括号。实际上, main 函数可以带参数, 这个参数可以认为是 main 函数的形式参数。C 语言规定 main 函数的参数只能有两个, 习惯上这两个参数写为 argc 和 argv。因此, main 函数的函数头可写为:

```
1. int main(argc, argv)
```

C 语言还规定 argc(第一个形参)必须是整型变量, argv(第二个形参)必须是指向字符串的指针数组。加上形参说明后, main 函数的函数头应写为:

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

由于 main 函数不能被其它函数调用, 因此不可能在程序内部取得实际值。那么, 在何处把实参值赋予 main 函数的形参呢? 实际上, main 函数的参数值是从操作系统命令行上获得的。当我们要运行一个可执行文件时, 在 DOS 提示符下键入文件名, 再输入实际参数即可把这些实参传送到 main 的形参中去。

DOS 提示符下命令行的一般形式为:

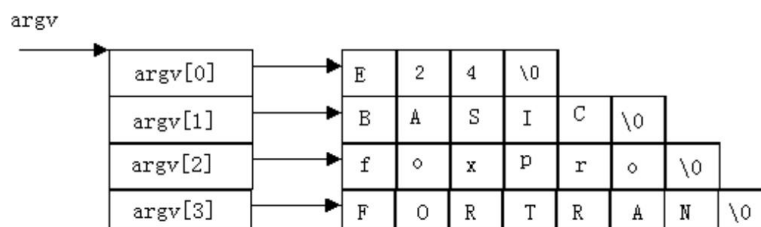
```
1. C:\>可执行文件名 参数 参数.....;
```

但是应该特别注意的是, main 的两个形参和命令行中的参数在位置上不是一一对应的。因为, main 的形参只有二个, 而命令行中的参数个数原则上未加限制。argc 参数表示了命令行中参数的个数(注意: 文件名本身也算一个参数), argc 的值是在输入命令行时由系统按实际参数的个数自动赋予的。

例如有命令行为:

```
1. C:\>E24 BASIC foxpro FORTRAN
```

由于文件名 E24 本身也算一个参数, 所以共有 4 个参数, 因此 argc 取得的值为 4。argv 参数是字符串指针数组, 其各元素值为命令行中各字符串(参数均按字符串处理)的首地址。指针数组的长度即为参数个数。数组元素初值由系统自动赋予。其表示如图所示:



【例】

```
1. #include <stdio.h>
2.
```

```
3. int main(int argc, char *argv) {  
4.     while(argc-->1)  
5.         printf("%s\n", *++argv);  
6.     return 0;  
7. }
```

本例是显示命令行中输入的参数。如果上例的可执行文件名为 e24.exe, 存放在 A 驱动器的盘内。因此输入的命令行为:

```
1. C:\>a:e24 BASIC foxpro FORTRAN
```

则运行结果为:

```
1. BASIC  
2. foxpro  
3. FORTRAN
```

该行共有 4 个参数, 执行 main 时, argc 的初值即为 4。argv 的 4 个元素分为 4 个字符串的首地址。执行 while 语句, 每循环一次 argv 值减 1, 当 argv 等于 1 时停止循环, 共循环三次, 因此共可输出三个参数。在 printf 函数中, 由于打印项*++argv 是先加 1 再打印, 故第一次打印的是 argv[1]所指的字符串 BASIC。第二、三次循环分别打印后二个字符串。而参数 e24 是文件名, 不必输出。

*6.9 动态内存分配与指向它的指针变量

动态内存分配

在数组一章中，曾介绍过数组的长度是预先定义好的，在整个程序中固定不变。C语言中不允许动态数组类型。

例如：

```
1. int n;  
2. scanf("%d", &n);  
3. int a[n];
```

用变量表示长度，想对数组的大小作动态说明，这是错误的。但是在实际的编程中，往往会发生这种情况，即所需的内存空间取决于实际输入的数据，而无法预先确定。对于这种问题，用数组的办法很难解决。为了解决上述问题，C语言提供了一些内存管理函数，这些内存管理函数可以按需要动态地分配内存空间，也可把不再使用的空间回收待用，为有效地利用内存资源提供了手段。

常用的内存管理函数有以下三个：

1. 分配内存空间函数 malloc

调用形式：

```
1. (类型说明符*)malloc(size)
```

功能：在内存的动态存储区中分配一块长度为"size"字节的连续区域。函数的返回值为该区域的首地址。

“类型说明符”表示把该区域用于何种数据类型。

(类型说明符*) 表示把返回值强制转换为该类型指针。

“size”是一个无符号数。

例如：

```
1. pc = (char *)malloc(100);
```

表示分配 100 个字节的内存空间，并强制转换为字符数组类型，函数的返回值为指向该字符数组的指针，把该指针赋予指针变量 pc。

2. 分配内存空间函数 calloc

calloc 也用于分配内存空间。

调用形式:

1. (类型说明符*)calloc(n, size)

功能: 在内存动态存储区中分配 n 块长度为“size”字节的连续区域。函数的返回值为该区域的首地址。

(类型说明符*)用于强制类型转换。

calloc 函数与 malloc 函数的区别仅在于一次可以分配 n 块区域。

例如:

1. ps = (struct stu*)calloc(2, sizeof(struct stu));

其中的 sizeof(struct stu)是求 stu 的结构长度。因此该语句的意思是: 按 stu 的长度分配 2 块连续区域, 强制转换为 stu 类型, 并把其首地址赋予指针变量 ps。

3. 释放内存空间函数 free

调用形式:

1. free(void*ptr);

功能: 释放 ptr 所指向的一块内存空间, ptr 是一个任意类型的指针变量, 它指向被释放区域的首地址。被释放区应是由 malloc 或 calloc 函数所分配的区域。

【例】分配一块区域, 输入一个学生数据。

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. int main() {
5.     struct stu {
6.         int num;
7.         char *name;
8.         char sex;
9.         float score;
10.    }*ps;
11.    ps = (struct stu*)malloc(sizeof(struct stu));
12.    ps->num = 102;
13.    ps->name = "Zhang ping";
14.    ps->sex = 'M';
15.    ps->score = 62.5;
16.    printf("Number=%d\nName=%s\n", ps->num, ps->name);
17.    printf("Sex=%c\nScore=%f\n", ps->sex, ps->score);
```



```
18.    free(ps);  
19.    return 0;  
20. }
```

本例中，定义了结构 `stu`，定义了 `stu` 类型指针变量 `ps`。然后分配一块 `stu` 大内存区，并把首地址赋予 `ps`，使 `ps` 指向该区域。再以 `ps` 为指向结构的指针变量对各成员赋值，并用 `printf` 输出各成员值。最后用 `free` 函数释放 `ps` 指向的内存空间。整个程序包含了申请内存空间、使用内存空间、释放内存空间三个步骤，实现存储空间的动态分配。

void 指针类型

ANSI 新标准增加了一种“void”指针类型，即可以定义一个指针变量，但不指定它是指向哪一种类型数据。



6.10 本章小结

定义	含 义
<code>int i;</code>	定义整型变量 <code>i</code>
<code>int *p</code>	<code>p</code> 为指向整型数据的指针变量
<code>int a[n];</code>	定义整型数组 <code>a</code> , 它有 <code>n</code> 个元素
<code>int *p[n];</code>	定义指针数组 <code>p</code> , 它由 <code>n</code> 个指向整型数据的指针元素组成
<code>int (*p)[n];</code>	<code>p</code> 为指向含 <code>n</code> 个元素的一维数组的指针变量
<code>int f();</code>	<code>f</code> 为带回整型函数值的函数
<code>int *p();</code>	<code>p</code> 为带回一个指针的函数, 该指针指向整型数据
<code>int (*p)();</code>	<code>p</code> 为指向函数的指针, 该函数返回一个整型值
<code>int **p;</code>	<code>P</code> 是一个指针变量, 它指向一个指向整型数据的指针变量

指针运算小结

现把全部指针运算列出如下:

1) 指针变量加(减)一个整数:

例如: `p++`、`p--`、`p+i`、`p-i`、`p+=i`、`p-=i`

一个指针变量加(减)一个整数并不是简单地将原值加(减)一个整数, 而是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内存单元字节数加(减)。

2) 指针变量赋值: 将一个变量的地址赋给一个指针变量。

1. `p = &a;` (将变量 `a` 的地址赋给 `p`)
2. `p = array;` (将数组 `array` 的首地址赋给 `p`)
3. `p = &array[i];` (将数组 `array` 第 `i` 个元素的地址赋给 `p`)
4. `p = max;` (`max` 为已定义的函数, 将 `max` 的入口地址赋给 `p`)
5. `p1 = p2;` (`p1` 和 `p2` 都是指针变量, 将 `p2` 的值赋给 `p1`)

注意: 不能如下:

1. `p = 1000;`

3) 指针变量可以有空值, 即该指针变量不指向任何变量:

1. `p = NULL;`

4) 两个指针变量可以相减: 如果两个指针变量指向同一个数组的元素, 则两个指针变量值之差是两个指针之间的元素个数。

5) 两个指针变量比较: 如果两个指针变量指向同一个数组的元素, 则两个指针变量可以进行比较。指向前面的元素的指针变量“小于”指向后面的元素的指针变量。

第七章 结构体、共用体和枚举类型

【本章知识点汇总】

结构体变量

结构体数组

结构体指针

链表

共用体类型

枚举类型

typedef 声明新类型



本书配套视频精讲: <https://www.bilibili.com/video/BV1HJ41137fe>

7.1 定义和使用结构体变量

在实际问题中, 一组数据往往具有不同的数据类型。例如, 在学生登记表中, 姓名应为字符型; 学号可为整型或字符型; 年龄应为整型; 性别应为字符型; 成绩可为整型或实型。显然不能用一个数组来存放这一组数据。因为数组中各元素的类型和长度都必须一致, 以便于编译系统处理。为了解决这个问题, C 语言中给出了另一种构造数据类型——“结构 (structure)”或叫“结构体”。它相当于其它高级语言中的记录。“结构”是一种构造类型, 它是由若干“成员”组成的。每一个成员可以是一个基本数据类型或者又是一个构造类型。结构既是一种“构造”而成的数据类型, 那么在说明和使用之前必须先定义它, 也就是构造它。如同在说明和调用函数之前要先定义函数一样。

定义一个结构的一般形式为:

1. **struct** 结构名 {
2. 成员表列
3. };

成员表列由若干个成员组成, 每个成员都是该结构的一个组成部分。对每个成员也必须作类型说明, 其形式为:

1. 类型说明符 成员名;

成员名的命名应符合标识符的书写规定。例如:

1. **struct** stu {
2. **int** num;
3. **char** name[20];
4. **char** sex;
5. **float** score;
6. };

在这个结构定义中, 结构名为 `stu`, 该结构由 4 个成员组成。第一个成员为 `num`, 整型变量; 第二个成员为 `name`, 字符数组; 第三个成员为 `sex`, 字符变量; 第四个成员为 `score`, 实型变量。应注意在括号后的分号是不可少的。结构定义之后, 即可进行变量说明。凡说明为结构 `stu` 的变量都由上述 4 个成员组成。由此可见, 结构是一种复杂的数据类型, 是数目固定, 类型不同的若干有序变量的集合。

说明结构变量有以下三种方法。以上面定义的 `stu` 为例来加以说明。

1. 先定义结构, 再说明结构变量。

如:

```
1. struct stu {  
2.     int num;  
3.     char name[20];  
4.     char sex;  
5.     float score;  
6. };  
7. struct stu boy1, boy2;
```

说明了两个变量 boy1 和 boy2 为 stu 结构类型。也可以用宏定义使一个符号常量来表示一个结构类型。

例如:

```
1. #define STU struct stu  
2. STU {  
3.     int num;  
4.     char name[20];  
5.     char sex;  
6.     float score;  
7. };  
8. STU boy1, boy2;
```

2. 在定义结构类型的同时说明结构变量。

例如:

```
1. struct stu {  
2.     int num;  
3.     char name[20];  
4.     char sex;  
5.     float score;  
6. }boy1, boy2;
```

这种形式的说明的一般形式为:

```
1. struct 结构名 {  
2.     成员表列  
3. }变量名表列;
```

3. 直接说明结构变量

例如:

```
1. struct {  
2.     int num;
```

```
3.     char name[20];
4.     char sex;
5.     float score;
6. }boy1, boy2;
```

这种形式的说明的一般形式为:

```
1. struct {
2.     成员表列
3. }变量名表列;
```

第三种方法与第二种方法的区别在于第三种方法中省去了结构名, 而直接给出结构变量。三种方法中说明的 boy1,boy2 变量都具有下图所示的结构。

num	name	sex	score

说明了 boy1,boy2 变量为 stu 类型后, 即可向这两个变量中的各个成员赋值。在上述 stu 结构定义中, 所有的成员都是基本数据类型或数组类型。成员也可以又是一个结构, 即构成了嵌套的结构。例如, 下图给出了另一个数据结构。

num	name	sex	birthday			score
			month	day	year	

按图可给出以下结构定义:

```
1. struct date {
2.     int month;
3.     int day;
4.     int year;
5. };
6.
7. struct {
8.     int num;
9.     char name[20];
10.    char sex;
11.    struct date birthday;
12.    float score;
13. }boy1, boy2;
```

首先定义一个结构 date, 由 month(月)、day(日)、year(年) 三个成员组成。在定义并说明变

量 boy1 和 boy2 时, 其中的成员 birthday 被说明为 data 结构类型。成员名可与程序中其它变量同名, 互不干扰。

结构变量成员表示方法

在程序中使用结构变量时, 往往不把它作为一个整体来使用。在 ANSI C 中除了允许具有相同类型的结构变量相互赋值以外, 一般对结构变量的使用, 包括赋值、输入、输出、运算等都是通过结构变量的成员来实现的。

表示结构变量成员的一般形式是:

1. 结构变量名.成员名

例如:

1. boy1.num 即第一个人的学号
2. boy2.sex 即第二个人的性别

如果成员本身又是一个结构则必须逐级找到最低级的成员才能使用。

例如:

1. boy1.birthday.month

即第一个人出生的月份成员可以在程序中单独使用, 与普通变量完全相同。

结构变量的赋值

结构变量的赋值就是给各成员赋值。可用输入语句或赋值语句来完成。

【例】给结构变量赋值并输出其值。

```
1. #include <stdio.h>
2.
3. int main() {
4.     struct stu {
5.         int num;
6.         char *name;
7.         char sex;
8.         float score;
9.     }boy1, boy2;
10.    boy1.num = 102;
11.    boy1.name = "Zhang ping";
12.    printf("input sex and score\n");
```



```

13.     scanf("%c %f", &boy1.sex, &boy1.score);
14.     boy2 = boy1;
15.     printf("Number=%d\nName=%s\n", boy2.num, boy2.name);
16.     printf("Sex=%c\nScore=%f\n", boy2.sex, boy2.score);
17.     return 0;
18. }

```

本程序中用赋值语句给 num 和 name 两个成员赋值, name 是一个字符串指针变量。用 scanf 函数动态地输入 sex 和 score 成员值, 然后把 boy1 的所有成员的值整体赋予 boy2。最后分别输出 boy2 的各个成员值。本例表示了结构变量的赋值、输入和输出的方法。

结构变量的初始化

和其他类型变量一样, 对结构变量可以在定义时进行初始化赋值。

【例】对结构变量初始化。

```

1.  #include <stdio.h>
2.
3.  int main() {
4.      struct stu /*定义结构*/
5.      {
6.          int num;
7.          char *name;
8.          char sex;
9.          float score;
10.     }boy2, boy1 = {102, "Zhang ping", 'M', 78.5};
11.     boy2 = boy1;
12.     printf("Number=%d\nName=%s\n", boy2.num, boy2.name);
13.     printf("Sex=%c\nScore=%f\n", boy2.sex, boy2.score);
14.     return 0;
15. }

```

本例中, boy2, boy1 均被定义为外部结构变量, 并对 boy1 作了初始化赋值。在 main 函数中, 把 boy1 的值整体赋予 boy2, 然后用两个 printf 语句输出 boy2 各成员的值。

7.2 使用结构体数组

定义结构体数组

数组的元素也可以是结构类型的。因此可以构成结构型数组。结构数组的每一个元素都是具有相同结构类型的下标结构变量。在实际应用中, 经常用结构数组来表示具有相同数据结构的一个群体。如一个班的学生档案, 一个车间职工的工资表等。方法和结构变量相似, 只需说明它为数组类型即可。

例如:

```
1. struct stu {  
2.     int num;  
3.     char *name;  
4.     char sex;  
5.     float score;  
6. }boy[5];
```

定义了一个结构数组 boy, 共有 5 个元素, boy[0]~boy[4]。每个数组元素都具有 struct stu 的结构形式。对结构数组可以作初始化赋值。

例如:

```
1. struct stu {  
2.     int num;  
3.     char *name;  
4.     char sex;  
5.     float score;  
6. }boy[5] = {  
7.     {101, "Li ping", "M", 45},  
8.     {102, "Zhang ping", "M", 62.5},  
9.     {103, "He fang", "F", 92.5},  
10.    {104, "Cheng ling", "F", 87},  
11.    {105, "Wang ming", "M", 58};  
12. }
```

当对全部元素作初始化赋值时, 也可不给出数组长度。

【例】计算学生的平均成绩和不及格的人数。

```
1. #include <stdio.h>  
2.  
3. struct stu {
```

```

4.     int num;
5.     char *name;
6.     char sex;
7.     float score;
8. }boy[5] = {
9.     {101,"Li ping",'M',45},
10.    {102,"Zhang ping",'M',62.5},
11.    {103,"He fang",'F',92.5},
12.    {104,"Cheng ling",'F',87},
13.    {105,"Wang ming",'M',58},
14. };
15.
16. int main() {
17.     int i, c = 0;
18.     float ave, s = 0;
19.     for(i = 0; i < 5; i++) {
20.         s += boy[i].score;
21.         if(boy[i].score < 60) c += 1;
22.     }
23.     printf("s=%f\n", s);
24.     ave = s / 5;
25.     printf("average=%f\ncount=%d\n", ave, c);
26.     return 0;
27. }

```

本例程序中定义了一个外部结构数组 boy，共 5 个元素，并作了初始化赋值。在 main 函数中用 for 语句逐个累加各元素的 score 成员值存于 s 之中，如 score 的值小于 60(不及格)即计数器 C 加 1，循环完毕后计算平均成绩，并输出全班总分，平均分及不及格人数。

【例】建立同学通讯录

```

1. #include <stdio.h>
2.
3. #define NUM 3
4. struct mem {
5.     char name[20];
6.     char phone[10];
7. };
8.
9. int main() {
10.     struct mem man[NUM];

```

```
11.     int i;
12.     for(i = 0; i < NUM; i++) {
13.         printf("input name:\n");
14.         gets(man[i].name);
15.         printf("input phone:\n");
16.         gets(man[i].phone);
17.     }
18.     printf("name\t\t\tphone\n\n");
19.     for(i = 0; i < NUM; i++)
20.         printf("%s\t\t\t%s\n",man[i].name, man[i].phone);
21.     return 0;
22. }
```

本程序中定义了一个结构 `mem`，它有两个成员 `name` 和 `phone` 用来表示姓名和电话号码。在主函数中定义 `man` 为具有 `mem` 类型的结构数组。在 `for` 语句中，用 `gets` 函数分别输入各个元素中两个成员的值。然后又在 `for` 语句中用 `printf` 语句输出各元素中两个成员值。

N 诺
noobdream.com

7.3 结构体指针

指向结构体变量的指针

一个指针变量当用来指向一个结构变量时，称之为结构指针变量。结构指针变量中的值是指向的结构变量的首地址。通过结构指针即可访问该结构变量，这与数组指针和函数指针的情况是相同的。

结构指针变量说明的一般形式为：

- 1. `struct` 结构名 *结构指针变量名

例如，在前面的例题中定义了 `stu` 这个结构，如要说明一个指向 `stu` 的指针变量 `pstu`，可写为：

- 1. `struct stu *pstu;`

当然也可在定义 `stu` 结构时同时说明 `pstu`。与前面讨论的各类指针变量相同，结构指针变量也必须要先赋值后才能使用。赋值是把结构变量的首地址赋予该指针变量，不能把结构名赋予该指针变量。如果 `boy` 是被说明为 `stu` 类型的结构变量，则：

- 1. `struct stu *pstu;`

是正确的，而：

- 1. `pstu = &stu`

是错误的。

结构名和结构变量是两个不同的概念，不能混淆。结构名只能表示一个结构形式，编译系统并不对它分配内存空间。只有当某变量被说明为这种类型的结构时，才对该变量分配存储空间。因此上面 `&stu` 这种写法是错误的，不可能去取一个结构名的首地址。有了结构指针变量，就能更方便地访问结构变量的各个成员。

其访问的一般形式为：

- 1. `(*结构指针变量).成员名`

或为：

- 1. 结构指针变量->成员名

例如：

- 1. `(*pstu).num`

或者：

- 1. `pstu->num`

应该注意 `(*pstu)` 两侧的括号不可少，因为成员符 “.” 的优先级高于 “*”。如去掉括号写作 `*pstu.num` 则等效于 `*(pstu.num)`，这样，意义就完全不对了。

下面通过例子来说明结构指针变量的具体说明和使用方法。

【例】

```
1. #include <stdio.h>
2.
3. struct stu {
4.     int num;
5.     char *name;
6.     char sex;
7.     float score;
8. } boy1 = {102, "Zhang ping", 'M', 78.5}, *pstu;
9.
10. int main() {
11.     pstu = &boy1;
12.     printf("Number=%d\nName=%s\n", boy1.num, boy1.name);
13.     printf("Sex=%c\nScore=%f\n\n", boy1.sex, boy1.score);
14.     printf("Number=%d\nName=%s\n", (*pstu).num, (*pstu).name);
15.     printf("Sex=%c\nScore=%f\n\n", (*pstu).sex, (*pstu).score);
16.     printf("Number=%d\nName=%s\n", pstu->num, pstu->name);
17.     printf("Sex=%c\nScore=%f\n\n", pstu->sex, pstu->score);
18.     return 0;
19. }
```

本例程序定义了一个结构 `stu`，定义了 `stu` 类型结构变量 `boy1` 并作了初始化赋值，还定义了一个指向 `stu` 类型结构的指针变量 `pstu`。在 `main` 函数中，`pstu` 被赋予 `boy1` 的地址，因此 `pstu` 指向 `boy1`。然后在 `printf` 语句内用三种形式输出 `boy1` 的各个成员值。从运行结果可以看出：

1. 结构变量.成员名
2. (*结构指针变量).成员名
3. 结构指针变量->成员名

这三种用于表示结构成员的形式是完全等效的。

指向结构体数组的指针

指针变量可以指向一个结构数组，这时结构指针变量的值是整个结构数组的首地址。结构指针

变量也可指向结构数组的一个元素, 这时结构指针变量的值是该结构数组元素的首地址。设 `ps` 为指向结构数组的指针变量, 则 `ps` 也指向该结构数组的 0 号元素, `ps+1` 指向 1 号元素, `ps+i` 则指向 `i` 号元素。这与普通数组的情况是一致的。

【例】用指针变量输出结构数组。

```
1. #include <stdio.h>
2.
3. struct stu {
4.     int num;
5.     char *name;
6.     char sex;
7.     float score;
8. }boy[5] = {
9.     {101,"Zhou ping",'M',45},
10.    {102,"Zhang ping",'M',62.5},
11.    {103,"Liou fang",'F',92.5},
12.    {104,"Cheng ling",'F',87},
13.    {105,"Wang ming",'M',58},
14. };
15.
16. int main() {
17.     struct stu *ps;
18.     printf("No\tName\t\t\tSex\tScore\t\n");
19.     for(ps = boy; ps < boy+5; ps++)
20.         printf("%d\t%s\t\t%c\t%f\t\n", ps->num, ps->name, ps->sex, ps->score);
21.     return 0;
22. }
```

在程序中, 定义了 `stu` 结构类型的外部数组 `boy` 并作了初始化赋值。在 `main` 函数内定义 `ps` 为指向 `stu` 类型的指针。在循环语句 `for` 的表达式 1 中, `ps` 被赋予 `boy` 的首地址, 然后循环 5 次, 输出 `boy` 数组中各成员值。应该注意的是, 一个结构指针变量虽然可以用来访问结构变量或结构数组元素的成员, 但是, 不能使它指向一个成员。也就是说不允许取一个成员的地址来赋予它。因此, 下面的赋值是错误的。

```
1. ps = &boy[1].sex;
```

而只能是:

```
1. ps = boy; (赋予数组首地址)
```

或者是:

```
1. ps = &boy[0];(赋予 0 号元素首地址)
```

用结构体变量和结构体变量的指针作函数参数

在 ANSI C 标准中允许用结构变量作函数参数进行整体传送。但是这种传送要将全部成员逐个传送, 特别是成员为数组时将会使传送的时间和空间开销很大, 严重地降低了程序的效率。因此最好的办法就是使用指针, 即用指针变量作函数参数进行传送。这时由实参传向形参的只是地址, 从而减少了时间和空间的开销。

【例】计算一组学生的平均成绩和不及格人数。用结构指针变量作函数参数编程。

```
1. #include <stdio.h>
2.
3. struct stu {
4.     int num;
5.     char *name;
6.     char sex;
7.     float score;
8. }boy[5] = {
9.     {101,"Li ping",'M',45},
10.    {102,"Zhang ping",'M',62.5},
11.    {103,"He fang",'F',92.5},
12.    {104,"Cheng ling",'F',87},
13.    {105,"Wang ming",'M',58},
14. };
15.
16. int main() {
17.     struct stu *ps;
18.     void ave(struct stu *ps);
19.     ps = boy;
20.     ave(ps);
21. }
22.
23. void ave(struct stu *ps) {
24.     int c = 0,i;
25.     float ave, s = 0;
26.     for(i = 0; i < 5; i++, ps++) {
27.         s += ps->score;
```

```
28.     if(ps->score < 60) c += 1;
29.     }
30.     printf("s=%f\n", s);
31.     ave = s / 5;
32.     printf("average=%f\ncount=%d\n", ave, c);
33. }
```

本程序中定义了函数 `ave`，其形参为结构指针变量 `ps`。`boy` 被定义为外部结构数组，因此在整个源程序中有效。在 `main` 函数中定义说明了结构指针变量 `ps`，并把 `boy` 的首地址赋予它，使 `ps` 指向 `boy` 数组。然后以 `ps` 作实参调用函数 `ave`。在函数 `ave` 中完成计算平均成绩和统计不及格人数的工作并输出结果。由于本程序全部采用指针变量作运算和处理，故速度更快，程序效率更高。



*7.4 用指针处理链表

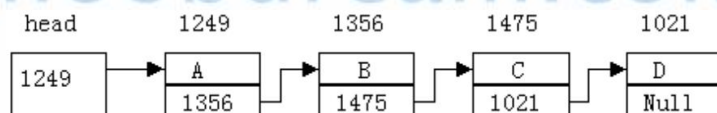
什么是链表

在前面的例子中采用了动态分配的办法为一个结构分配内存空间。每一次分配一块空间可用来存放一个学生的数据，我们可称之为一个结点。有多少个学生就应该申请分配多少块内存空间，也就是说要建立多少个结点。当然用结构数组也可以完成上述工作，但如果预先不能准确把握学生人数，也就无法确定数组大小。而且当学生留级、退学之后也不能把该元素占用的空间从数组中释放出来。

用动态存储的方法可以很好地解决这些问题。有一个学生就分配一个结点，无须预先确定学生的准确人数，某学生退学，可删去该结点，并释放该结点占用的存储空间。从而节约了宝贵的内存资源。另一方面，用数组的方法必须占用一块连续的内存区域。而使用动态分配时，每个结点之间可以是不连续的(结点内是连续的)。结点之间的联系可以用指针实现。即在结点结构中定义一个成员项用来存放下一结点的首地址，这个用于存放地址的成员，常把它称为指针域。

可在第一个结点的指针域内存入第二个结点的首地址，在第二个结点的指针域内又存放第三个结点的首地址，如此串连下去直到最后一个结点。最后一个结点因无后续结点连接，其指针域可赋为 0。这样一种连接方式，在数据结构中称为“链表”。

下图为最简单链表的示意图。



图中，第 0 个结点称为头结点，它存放有第一个结点的首地址，它没有数据，只是一个指针变量。以下的每个结点都分为两个域，一个是数据域，存放各种实际的数据，如学号 num，姓名 name，性别 sex 和成绩 score 等。另一个域为指针域，存放下一结点的首地址。链表中的每一个结点都是同一种结构类型。

例如，一个存放学生学号和成绩的结点应为以下结构：

```

1. struct stu {
2.     int num;
3.     int score;
4.     struct stu *next;
5. }
  
```

前两个成员项组成数据域，后一个成员项 `next` 构成指针域，它是一个指向 `stu` 类型结构的指针变量。

链表的基本操作对链表的主要操作有以下几种：

1. 建立链表；
2. 结构的查找与输出；
3. 插入一个结点；
4. 删除一个结点；

下面通过例题来说明这些操作。

【例】建立一个三个结点的链表，存放学生数据。为简单起见，我们假定学生数据结构中只有学号和年龄两项。可编写一个建立链表的函数 `creat`。程序如下：

```
1. #define NULL 0
2. #define TYPE struct stu
3. #define LEN sizeof (struct stu)
4. struct stu {
5.     int num;
6.     int age;
7.     struct stu *next;
8. };
9.
10. TYPE *creat(int n) {
11.     struct stu *head, *pf, *pb;
12.     int i;
13.     for(i = 0; i < n; i++) {
14.         pb = (TYPE*) malloc(LEN);
15.         printf("input Number and Age\n");
16.         scanf("%d%d", &pb->num, &pb->age);
17.         if(i == 0) pf = head = pb;
18.         else pf->next = pb;
19.         pb->next = NULL;
20.         pf = pb;
21.     }
22.     return(head);
23. }
```

在函数外首先用宏定义对三个符号常量作了定义。这里用 `TYPE` 表示 `struct stu`，用 `LEN`

表示 `sizeof(struct stu)` 主要的目的是为了在以下程序内减少书写并使阅读更加方便。结构 `stu` 定义为外部类型，程序中的各个函数均可使用该定义。`creat` 函数用于建立一个有 `n` 个结点的链表，它是一个指针函数，它返回的指针指向 `stu` 结构。在 `creat` 函数内定义了三个 `stu` 结构的指针变量。`head` 为头指针，`pf` 为指向两相邻结点的前一结点的指针变量。`pb` 为后一结点的指针变量。

【例】建立一个学生链表，存放学号和分数两项，并输出链表的数据。

```
1. #include <stdio.h>
2. #include <malloc.h>
3.
4. struct Student {
5.     int num;
6.     float score;
7.     struct Student *next;
8. };
9.
10. int n;
11. struct Student* create() {
12.     n = 0;
13.     struct Student *head;
14.     struct Student *p1, *p2;
15.     p1 = p2 = (struct Student *)malloc(sizeof(Student));
16.     scanf("%d,%f", &p1->num, &p1->score);
17.     head = NULL;
18.     while (p1->num != 0) {
19.         n = n + 1;
20.         if (n == 1) head = p1;
21.         else p2->next = p1;
22.         p2 = p1;
23.         p1 = (struct Student *)malloc(sizeof(Student));
24.         scanf("%d,%f", &p1->num, &p1->score);
25.     }
26.     return head;
27. };
28.
29. void print(struct Student *head) {
30.     struct Student *p;
31.     printf("Total %d Students.\n", n);
32.     p = head;
```

```
33.     while (p != NULL) {
34.         printf("%d %.2f\n", p->num, p->score);
35.         p = p->next;
36.     }
37. }
38.
39. int main(){
40.     struct Student *head;
41.     head = create();
42.     print(head);
43.     return 0;
44. }
```

总结

线性表的链式存储相比于顺序存储，有两大优势：

- 1、链式存储的数据元素在物理结构没有限制，当内存空间中没有足够大的连续的内存空间供顺序表使用时，可能使用链表能解决问题。（链表每次申请的都是单个数据元素的存储空间，可以利用上一些内存碎片）
- 2、链表中结点之间采用指针进行链接，当对链表中的数据元素实行插入或者删除操作时，只需要改变指针的指向，无需像顺序表那样移动插入或删除位置的后续元素，简单快捷。

链表和顺序表相比，不足之处在于，当遍历操作时，由于链表中结点的物理位置不相邻，使得计算机查找起来相比较顺序表，速度要慢。

*7.5 共用体类型

什么是共用体类型

共用体是一种特殊的数据类型，允许您在相同的内存位置存储不同的数据类型。您可以定义一个带有多成员的共用体，但是任何时候只能有一个成员带有值。共用体提供了一种使用相同的内存位置的有效方式。

定义共用体

为了定义共用体，您必须使用 `union` 语句，方式与定义结构类似。`union` 语句定义了一个新的数据类型，带有多个成员。`union` 语句的格式如下：

```
1. union [union tag]
2. {
3.     member definition;
4.     member definition;
5.     ...
6.     member definition;
7. } [one or more union variables];
```

`union tag` 是可选的，每个 `member definition` 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。在共用体定义的末尾，最后一个分号之前，您可以指定一个或多个共用体变量，这是可选的。下面定义一个名为 `Data` 的共用体类型，有三个成员 `i`、`f` 和 `str`：

```
1. union Data
2. {
3.     int i;
4.     float f;
5.     char str[20];
6. } data;
```

现在，`Data` 类型的变量可以存储一个整数、一个浮点数，或者一个字符串。这意味着一个变量（相同的内存位置）可以存储多个多种类型的数据。您可以根据需要在一个共用体内使用任何内置的或者用户自定义的数据类型。

共用体占用的内存应足够存储共用体中最大的成员。例如，在上面的实例中，`Data` 将占用 20 个字节的内存空间，因为在各个成员中，字符串所占用的空间是最大的。下面的实例将显示上面的共用体占用的总内存大小：

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. union Data
5. {
6.     int i;
7.     float f;
8.     char str[20];
9. };
10. int main( )
11. {
12.     union Data data;
13.     printf( "Memory size occupied by data : %d\n", sizeof(data));
14.     return 0;
15. }
```

当上面的代码被编译和执行时，它会产生下列结果：

```
1. Memory size occupied by data : 20
```

访问共用体成员

为了访问共用体的成员，我们使用成员访问运算符（.）。成员访问运算符是共用体变量名称和我们要访问的共用体成员之间的一个句号。您可以使用 **union** 关键字来定义共用体类型的变量。下面的实例演示了共用体的用法：

```
1. #include <stdio.h>
2. #include <string.h>
3.
4. union Data
5. {
6.     int i;
7.     float f;
8.     char str[20];
9. };
10. int main( )
11. {
12.     union Data data;
13.     data.i = 10;
14.     data.f = 220.5;
```

```
15. strcpy( data.str, "C Programming");  
16. printf( "data.i : %d\n", data.i);  
17. printf( "data.f : %f\n", data.f);  
18. printf( "data.str : %s\n", data.str);  
19. return 0;  
20. }
```



7.6 使用枚举类型

在实际问题中, 有些变量的取值被限定在一个有限的范围内。例如, 一个星期内只有七天, 一年只有十二个月, 一个班每周有六门课程等等。如果把这些量说明为整型, 字符型或其它类型显然是不妥当的。为此, C 语言提供了一种称为“枚举”的类型。在“枚举”类型的定义中列举出所有可能的取值, 被说明为该“枚举”类型的变量取值不能超过定义的范围。应该说明的是, 枚举类型是一种基本数据类型, 而不是一种构造类型, 因为它不能再分解为任何基本类型。

枚举类型的定义

1. 枚举的定义枚举类型定义的一般形式为:

1. `enum` 枚举名{ 枚举值表 };

在枚举值表中应罗列出所有可用值。这些值也称为枚举元素。

例如:

该枚举名为 `weekday`, 枚举值共有 7 个, 即一周中的七天。凡被说明为 `weekday` 类型变量的取值只能是七天中的某一天。

2. 枚举变量的说明

如同结构和联合一样, 枚举变量也可用不同的方式说明, 即先定义后说明, 同时定义说明或直接说明。设有变量 `a,b,c` 被说明为上述的 `weekday`, 可采用下述任一种方式:

1. `enum weekday{ sun,mou,tue,wed,thu,fri,sat };`
2. `enum weekday a,b,c;`

或者为:

1. `enum weekday{ sun,mou,tue,wed,thu,fri,sat }a,b,c;`

或者为:

1. `enum { sun,mou,tue,wed,thu,fri,sat }a,b,c;`

枚举类型变量的赋值和使用

枚举类型在使用中有以下规定:

1. 枚举值是常量，不是变量。不能在程序中用赋值语句再对它赋值。

例如对枚举 weekday 的元素再作以下赋值：

```
1. sun=5;
2. mon=2;
3. sun=mon;
```

都是错误的。

2. 枚举元素本身由系统定义了一个表示序号的数值，从 0 开始顺序定义为 0, 1, 2…。如在 weekday 中，sun 值为 0，mon 值为 1，…，sat 值为 6。

【例】

```
1. #include <stdio.h>
2.
3. int main(){
4.     enum weekday {
5.         sun, mon, tue, wed, thu, fri, sat
6.     } a, b, c;
7.     a = sun;
8.     b = mon;
9.     c = tue;
10.    printf("%d,%d,%d", a, b, c);
11.    return 0;
12. }
```

说明：只能把枚举值赋予枚举变量，不能把元素的数值直接赋予枚举变量。如：

```
1. a=sum;
2. b=mon;
```

是正确的。而：

```
1. a = 0;
2. b = 1;
```

是错误的。如一定要把数值赋予枚举变量，则必须用强制类型转换。

如：

```
1. a = (enum weekday)2;
```

其意义是将序号为 2 的枚举元素赋予枚举变量 a，相当于：

```
1. a = tue;
```

还应该说明的是枚举元素不是字符常量也不是字符串常量，使用时不要加单、双引号。

【例】

```
1. #include <stdio.h>
2.
3. int main() {
4.     enum color { red = 1, green, blue };
5.     enum color favorite_color;
6.     /* ask user to choose color */
7.     printf("请输入你喜欢的颜色: (1. red, 2. green, 3. blue): ");
8.     scanf("%d", &favorite_color);
9.     /* 输出结果 */
10.    switch (favorite_color) {
11.        case red:
12.            printf("你喜欢的颜色是红色");
13.            break;
14.        case green:
15.            printf("你喜欢的颜色是绿色");
16.            break;
17.        case blue:
18.            printf("你喜欢的颜色是蓝色");
19.            break;
20.        default:
21.            printf("你没有选择你喜欢的颜色");
22.    }
23.    return 0;
24. }
```

noobdream.com

*7.7 用 typedef 声明新类型名

C 语言不仅提供了丰富的数据类型, 而且还允许由用户自己定义类型说明符, 也就是说允许由用户为数据类型取“别名”。类型定义符 `typedef` 即可用来完成此功能。例如, 有整型量 `a, b`, 其说明如下:

```
1. int a, b;
```

其中 `int` 是整型变量的类型说明符。`int` 的完整写法为 `integer`, 为了增加程序的可读性, 可把整型说明符用 `typedef` 定义为:

```
1. typedef int INTEGER
```

这以后就可用 `INTEGER` 来代替 `int` 作整型变量的类型说明了。例如:

```
1. INTEGER a, b;
```

它等效于:

```
1. int a, b;
```

用 `typedef` 定义数组、指针、结构等类型将带来很大的方便, 不仅使程序书写简单而且使意义更为明确, 因而增强了可读性。

例如:

```
1. typedef char NAME[20];
```

 表示 `NAME` 是字符数组类型, 数组长度为 20。

然后可用 `NAME` 说明变量, 如:

```
1. NAME a1, a2, s1, s2;
```

完全等效于:

```
1. char a1[20], a2[20], s1[20], s2[20]
```

又如:

```
1. typedef struct stu {
2.     char name[20];
3.     int age;
4.     char sex;
5. } STU;
```

定义 `STU` 表示 `stu` 的结构类型, 然后可用 `STU` 来说明结构变量:

```
1. STU body1, body2;
2. typedef 定义的一般形式为:
3. typedef 原类型名 新类型名
```

其中原类型名中含有定义部分, 新类型名一般用大写表示, 以便于区别。有时也可用宏定义来代替 `typedef` 的功能, 但是宏定义是由预处理完成的, 而 `typedef` 则是在编译时完成的, 后者更为灵活方便。

7.8 本章小结

实战练习

DreamJudge 1052 学生成绩管理

DreamJudge 1053 偷菜时间表

DreamJudge 1054 塑身菜单

DreamJudge 1151 成绩排序

DreamJudge 1177 查找学生信息

DreamJudge 1476 查找学生信息 2

C 语言专业题库在线练习

<http://www.noobdream.com/Practice/clang/>

noobdream.com

第八章 文件

【本章知识点汇总】

打开文件

关闭文件

顺序读写数据文件

随机读写数据文件

文件读写的出错检测

本书配套视频精讲: <https://www.bilibili.com/video/BV1HJ41137fe>

8.1 C 文件的有关基本知识

什么是文件

所谓“文件”是指一组相关数据的有序集合。这个数据集有一个名称，叫做文件名。实际上在前面的各章中我们已经多次使用了文件，例如源程序文件、目标文件、可执行文件、库文件 (头文件)等。

文件通常是驻留在外部介质(如磁盘等)上的，在使用时才调入内存中来。从不同的角度可对文件作不同的分类。从用户的角度看，文件可分为普通文件和设备文件两种。普通文件是指驻留在磁盘或其它外部介质上的一个有序数据集，可以是源文件、目标文件、可执行程序；也可以是一组待输入处理的原始数据，或者是一组输出的结果。对于源文件、目标文件、可执行程序可以称作程序文件，对输入输出数据可称作数据文件。

设备文件是指与主机相联的各种外部设备，如显示器、打印机、键盘等。在操作系统中，把外部设备也看作是一个文件来进行管理，把它们的输入、输出等同于对磁盘文件的读和写。通常把显示器定义为标准输出文件，一般情况下在屏幕上显示有关信息就是向标准输出文件输出。如前面经常使用的 `printf, putchar` 函数就是这类输出。键盘通常被指定标准的输入文件，从键盘上输入就意味着从标准输入文件上输入数据。`scanf, getchar` 函数就属于这类输入。

从文件编码的方式来看，文件可分为 ASCII 码文件和二进制码文件两种。ASCII 文件也称为文本文件，这种文件在磁盘存放时每个字符对应一个字节，用于存放对应的 ASCII 码。例如，数 5678 的存储形式为：

ASCII 码:	00110101	00110110	00110111	00111000
	↓	↓	↓	↓
十进制码:	5	6	7	8

共占用 4 个字节。

ASCII 码文件可在屏幕上按字符显示，例如源程序文件就是 ASCII 文件，用 DOS 命令 TYPE 可显示文件的内容。由于是按字符显示，因此能读懂文件内容。

二进制文件是按二进制的编码方式来存放文件的。

例如，数 5678 的存储形式为：

00010110 00101110

只占二个字节。二进制文件虽然也可在屏幕上显示，但其内容无法读懂。C 系统在处理这些文件时，并不区分类型，都看成是字符流，按字节进行处理。输入输出字符流的开始和结

束只由程序控制而不受物理符号(如回车符)的控制。因此也把这种文件称作“流式文件”。本章讨论流式文件的打开、关闭、读、写、定位等各种操作。

文件指针

在C语言中用一个指针变量指向一个文件,这个指针称为文件指针。通过文件指针就可对它所指的文件进行各种操作。

定义说明文件指针的一般形式为:

1. **FILE** *指针变量标识符;

其中 **FILE** 应为大写,它实际上是由系统定义的一个结构,该结构中含有文件名、文件状态和文件当前位置等信息。在编写源程序时不必关心 **FILE** 结构的细节。

例如:

1. **FILE** *fp;

表示 **fp** 是指向 **FILE** 结构的指针变量,通过 **fp** 即可找存放某个文件信息的结构变量,然后按结构变量提供的信息找到该文件,实施对文件的操作。习惯上也笼统地把 **fp** 称为指向一个文件的指针。

noobdream.com

8.2 打开与关闭文件

文件在进行读写操作之前要先打开, 使用完毕要关闭。所谓打开文件, 实际上是建立文件的各种有关信息, 并使文件指针指向该文件, 以便进行其它操作。关闭文件则断开指针与文件之间的联系, 也就禁止再对该文件进行操作。在 C 语言中, 文件操作都是由库函数来完成的。在本章内将介绍主要的文件操作函数。

用 fopen 函数打开数据文件

fopen 函数用来打开一个文件, 其调用的一般形式为:

1. 文件指针名 = fopen(文件名, 使用文件方式);

其中,

“文件指针名”必须是被说明为 FILE 类型的指针变量;

“文件名”是被打开文件的文件名;

“使用文件方式”是指文件的类型和操作要求。

“文件名”是字符串常量或字符串数组。

例如:

1. FILE *fp;
2. fp = ("file a", "r");

其意义是在当前目录下打开文件 file a, 只允许进行“读”操作, 并使 fp 指向该文件。

又如:

1. FILE *noob
2. noob = ("c:\\noobdream", "rb")

其意义是打开 C 驱动器磁盘的根目录下的文件 noobdream, 这是一个二进制文件, 只允许按二进制方式进行读操作。两个反斜线“\\”中的第一个表示转义字符, 第二个表示根目录。使用文件的方式共有 12 种, 下面给出了它们的符号和意义。

文件使用方式 意义

1. “rt” 只读打开一个文本文件, 只允许读数据
2. “wt” 只写打开或建立一个文本文件, 只允许写数据
3. “at” 追加打开一个文本文件, 并在文件末尾写数据
4. “rb” 只读打开一个二进制文件, 只允许读数据
5. “wb” 只写打开或建立一个二进制文件, 只允许写数据
6. “ab” 追加打开一个二进制文件, 并在文件末尾写数据
7. “rt+” 读写打开一个文本文件, 允许读和写

8. “wt+” 读写打开或建立一个文本文件, 允许读写
9. “at+” 读写打开一个文本文件, 允许读, 或在文件末追加数据
10. “rb+” 读写打开一个二进制文件, 允许读和写
11. “wb+” 读写打开或建立一个二进制文件, 允许读和写
12. “ab+” 读写打开一个二进制文件, 允许读, 或在文件末追加数据

对于文件使用方式有以下几点说明:

1) 文件使用方式由 r,w,a,t,b, +六个字符拼成, 各字符的含义是:

1. r(read): 读
2. w(write): 写
3. a(append): 追加
4. t(text): 文本文件, 可省略不写
5. b(banary): 二进制文件
6. +: 读和写

2) 凡用“r”打开一个文件时, 该文件必须已经存在, 且只能从该文件读出。

3) 用“w”打开的文件只能向该文件写入。若打开的文件不存在, 则以指定的文件名建立该文件, 若打开的文件已经存在, 则将该文件删去, 重建一个新文件。

4) 若要向一个已存在的文件追加新的信息, 只能用“a”方式打开文件。但此时该文件必须是存在的, 否则将会出错。

5) 在打开一个文件时, 如果出错, fopen 将返回一个空指针值 NULL。在程序中可以用这一信息来判别是否完成打开文件的工作, 并作相应的处理。因此常用以下程序段打开文件:

6)

```
1. if((fp = fopen("c:\\noobdream", "rb") == NULL) {  
2.     printf("error on open c:\\noobdream file!");  
3.     getch();  
4.     exit(1);  
5. }
```

这段程序的意义是, 如果返回的指针为空, 表示不能打开 C 盘根目录下的 noobdream 文件, 则给出提示信息 “error on open c:\\noobdream file!”, 下一行 getch()的功能是从键盘输入

一个字符，但不在屏幕上显示。在这里，该行的作用是等待，只有当用户从键盘敲任一键时，程序才继续执行，因此用户可利用这个等待时间阅读出错提示。敲键后执行 `exit(1)` 退出程序。

7) 把一个文本文件读入内存时，要将 ASCII 码转换成二进制码，而把文件以文本方式写入磁盘时，也要把二进制码转换成 ASCII 码，因此文本文件的读写要花费较多的转换时间。对二进制文件的读写不存在这种转换。

8) 标准输入文件(键盘)，标准输出文件(显示器)，标准出错输出(出错信息)是由系统打开的，可直接使用。

用 `fclose` 函数关闭数据文件

文件一旦使用完毕，应用关闭文件函数把文件关闭，以避免文件的数据丢失等错误。

`fclose` 函数调用的一般形式是：

1. `fclose(文件指针);`

例如：

1. `fclose(fp);`

正常完成关闭文件操作时，`fclose` 函数返回值为 0。如返回非零值则表示有错误发生。

8.3 顺序读写数据文件

怎样向文件读写字符

对文件的读和写是最常用的文件操作。在C语言中提供了多种文件读写的函数：

- 字符读写函数：fgetc 和 fputc
- 字符串读写函数：fgets 和 fputs
- 数据块读写函数：fread 和 fwrite
- 格式化读写函数：fscanf 和 fprintf

下面分别予以介绍。使用以上函数都要求包含头文件 `stdio.h`。

字符读写函数 fgetc 和 fputc

字符读写函数是以字符(字节)为单位的读写函数。每次可从文件读出或向文件写入一个字符。

1. 读字符函数 fgetc

fgetc 函数的功能是从指定的文件中读一个字符，函数调用的形式为：

1. 字符变量 = fgetc(文件指针);

例如：

1. `ch = fgetc(fp);`

其意义是从打开的文件 `fp` 中读取一个字符并送入 `ch` 中。

对于 fgetc 函数的使用有以下几点说明：

- 1) 在 fgetc 函数调用中，读取的文件必须是以读或读写方式打开的。
- 2) 读取字符的结果也可以不向字符变量赋值，

例如：

1. `fgetc(fp);`

但是读出的字符不能保存。

- 3) 在文件内部有一个位置指针。用来指向文件的当前读写字节。在文件打开时，该指针总是指向文件的第一个字节。使用 fgetc 函数后，该位置指针将向后移动一个字节。因此可连续多次使用 fgetc 函数，读取多个字符。应注意文件指针和文件内部的位置指针不是一回事。文件指针是指向整个文件的，须在程序中定义说明，只要不重新赋值，文件指针的值是不变的。文件内部的位置指针用以指示文件内部的当前读写位置，每读写一次，该指针均向后移动，它

不需在程序中定义说明, 而是由系统自动设置的。

【例】读入文件 c1.doc, 在屏幕上输出。

```
1. #include<stdio.h>
2. #include<stdlib.h>//exit
3. #include<conio.h>//getch
4.
5. int main() {
6.     FILE *fp;
7.     char ch;
8.     if((fp = fopen("d:\\noob\\example\\c1.txt", "rt")) == NULL) {
9.         printf("\nCannot open file strike any key exit!");
10.        getch();
11.        exit(1);
12.    }
13.    ch = fgetc(fp);
14.    while(ch != EOF) {
15.        putchar(ch);
16.        ch = fgetc(fp);
17.    }
18.    fclose(fp);
19.    return 0;
20. }
```

本例程序的功能是从文件中逐个读取字符, 在屏幕上显示。程序定义了文件指针 `fp`, 以文本文件方式打开文件“d:\\noob\\example\\ex1_1.c”, 并使 `fp` 指向该文件。如打开文件出错, 给出提示并退出程序。程序第 12 行先读出一个字符, 然后进入循环, 只要读出的字符不是文件结束标志(每个文件末有一结束标志 EOF)就把该字符显示在屏幕上, 再读入下一字符。每读一次, 文件内部的位置指针向后移动一个字符, 文件结束时, 该指针指向 EOF。执行本程序将显示整个文件。

2. 写字符函数 fputc

`fputc` 函数的功能是把一个字符写入指定的文件中, 函数调用的形式为:

```
1. fputc(字符量, 文件指针);
```

其中, 待写入的字符量可以是字符常量或变量, 例如:

```
1. fputc('a', fp);
```

其意义是把字符 `a` 写入 `fp` 所指向的文件中。

对于 `fputc` 函数的使用也要说明几点:

- 1) 被写入的文件可以用写、读写、追加方式打开, 用写或读写方式打开一个已存在的文件时将清除原有的文件内容, 写入字符从文件首开始。如需保留原有文件内容, 希望写入的字符以文件末开始存放, 必须以追加方式打开文件。被写入的文件若不存在, 则创建该文件。
- 2) 每写入一个字符, 文件内部位置指针向后移动一个字节。
- 3) `fputc` 函数有一个返回值, 如写入成功则返回写入的字符, 否则返回一个 `EOF`。可用此来判断写入是否成功。

【例】从键盘输入一行字符, 写入一个文件, 再把该文件内容读出显示在屏幕上。

```
1. #include<stdio.h>
2. #include<stdlib.h>//exit
3. #include<conio.h>//getch
4.
5. int main() {
6.     FILE *fp;
7.     char ch;
8.     if((fp = fopen("d:\\noob\\example\\string", "wt+")) == NULL) {
9.         printf("Cannot open file strike any key exit!");
10.        getch();
11.        exit(1);
12.    }
13.    printf("input a string:\n");
14.    ch = getchar();
15.    while (ch != '\n') {
16.        fputc(ch, fp);
17.        ch = getchar();
18.    }
19.    rewind(fp);
20.    ch = fgetc(fp);
21.    while(ch != EOF) {
22.        putchar(ch);
23.        ch = fgetc(fp);
24.    }
25.    printf("\n");
26.    fclose(fp);
27.    return 0;
28. }
```

程序中第 8 行以读写文本文件方式打开文件 `string`。程序第 14 行从键盘读入一个字符

后进入循环, 当读入字符不为回车符时, 则把该字符写入文件之中, 然后继续从键盘读入下一字符。每输入一个字符, 文件内部位置指针向后移动一个字节。写入完毕, 该指针已指向文件末。如要把文件从头读出, 须把指针移向文件头, 程序第 19 行 `rewind` 函数用于把 `fp` 所指文件的内部位置指针移到文件头。第 21 至 24 行用于读出文件中的一行内容。

【例】把命令行参数中的前一个文件名标识的文件, 复制到后一个文件名标识的文件中, 如命令行中只有一个文件名则把该文件写到标准输出文件(显示器)中。

```
1. #include<stdio.h>
2. #include<stdlib.h>//exit
3. #include<conio.h>//getch
4.
5. int main(int argc, char *argv[]) {
6.     FILE *fp1, *fp2;
7.     char ch;
8.     if(argc == 1) {
9.         printf("have not enter file name strike any key exit");
10.        getch();
11.        exit(0);
12.    }
13.    if((fp1 = fopen(argv[1], "rt")) == NULL) {
14.        printf("Cannot open %s\n", argv[1]);
15.        getch();
16.        exit(1);
17.    }
18.    if(argc == 2) fp2 = stdout;
19.    else if((fp2 = fopen(argv[2], "wt+")) == NULL) {
20.        printf("Cannot open %s\n", argv[1]);
21.        getch();
22.        exit(1);
23.    }
24.    while((ch = fgetc(fp1)) != EOF)
25.        fputc(ch, fp2);
26.    fclose(fp1);
27.    fclose(fp2);
28.    return 0;
29. }
```

本程序为带参的 `main` 函数。程序中定义了两个文件指针 `fp1` 和 `fp2`, 分别指向命令行参数中给出的文件。如命令行参数中没有给出文件名, 则给出提示信息。程序第 18 行表示如

果只给出一个文件名, 则使 `fp2` 指向标准输出文件(即显示器)。程序第 24 行至 25 行用循环语句逐个读出文件 1 中的字符再送到文件 2 中。再次运行时, 给出了一个文件名, 故输出给标准输出文件 `stdout`, 即在显示器上显示文件内容。第三次运行, 给出了二个文件名, 因此把 `string` 中的内容读出, 写入到 `OK` 之中。可用 DOS 命令 `type` 显示 `OK` 的内容。

字符串读写函数 `fgets` 和 `fputs`

1. 读字符串函数 `fgets`

函数的功能是从指定的文件中读一个字符串到字符数组中, 函数调用的形式为:

1. `fgets(字符数组名, n, 文件指针);`

其中的 `n` 是一个正整数。表示从文件中读出的字符串不超过 `n-1` 个字符。在读入的最后一个字符后加上串结束标志 `'\0'`。

例如:

1. `fgets(str, n, fp);`

的意义是从 `fp` 所指的文件中读出 `n-1` 个字符送入字符数组 `str` 中。

【例】从 `string` 文件中读入一个含 10 个字符的字符串。

```
1. #include<stdio.h>
2. #include<stdlib.h>//exit
3. #include<conio.h>//getch
4.
5. int main() {
6.     FILE *fp;
7.     char str[11];
8.     if((fp=fopen("d:\\noob\\example\\string", "rt")) == NULL) {
9.         printf("\nCannot open file strike any key exit!");
10.        getch();
11.        exit(1);
12.    }
13.    fgets(str, 11, fp);
14.    printf("\n%s\n", str);
15.    fclose(fp);
16.    return 0;
17. }
```

本例定义了一个字符数组 `str` 共 11 个字节, 在以读文本文件方式打开文件 `string` 后, 从中读出 10 个字符送入 `str` 数组, 在数组最后一个单元内将加上 `'\0'`, 然后在屏幕上显示输

出 `str` 数组。输出的十个字符正是例 13.1 程序的前十个字符。

对 `fgets` 函数有两点说明：

- 1) 在读出 `n-1` 个字符之前，如遇到了换行符或 EOF，则读出结束。
- 2) `fgets` 函数也有返回值，其返回值是字符数组的首地址。

2. 写字符串函数 `fputs`

`fputs` 函数的功能是向指定的文件写入一个字符串，其调用形式为：

1. `fputs(字符串, 文件指针);`

其中字符串可以是字符串常量，也可以是字符数组名，或指针变量，例如：

1. `fputs("abcd", fp);`

其意义是把字符串“abcd”写入 `fp` 所指的文件之中。

【例】在上例中建立的文件 `string` 中追加一个字符串。

```
1. #include<stdio.h>
2. #include<stdlib.h>//exit
3. #include<conio.h>//getch
4.
5. int main() {
6.     FILE *fp;
7.     char ch, st[20];
8.     if((fp = fopen("string", "at+")) == NULL) {
9.         printf("Cannot open file strike any key exit!");
10.        getch();
11.        exit(1);
12.    }
13.    printf("input a string:\n");
14.    scanf("%s", st);
15.    fputs(st, fp);
16.    rewind(fp);
17.    ch = fgetc(fp);
18.    while(ch != EOF) {
19.        putchar(ch);
20.        ch = fgetc(fp);
21.    }
22.    printf("\n");
23.    fclose(fp);
24.    return 0;
25. }
```


本例要求在 `string` 文件末加写字符串, 因此, 在程序第 8 行以追加读写文本文件的方式打开文件 `string`。然后输入字符串, 并用 `fputs` 函数把该串写入文件 `string`。在程序 16 行用 `rewind` 函数把文件内部位置指针移到文件首。再进入循环逐个显示当前文件中的全部内容。

数据块读写函数 `fread` 和 `fwrite`

C 语言还提供了用于整块数据的读写函数。可用来读写一组数据, 如一个数组元素, 一个结构变量的值等。

读数据块函数调用的一般形式为:

1. `fread(buffer, size, count, fp);`

写数据块函数调用的一般形式为:

1. `fwrite(buffer, size, count, fp);`

其中:`buffer` 是一个指针, 在 `fread` 函数中, 它表示存放输入数据的首地址。在 `fwrite` 函数中, 它表示存放输出数据的首地址。

1. `size` 表示数据块的字节数。
2. `count` 表示要读写的数据块块数。
3. `fp` 表示文件指针。

例如:

1. `fread(fa, 4, 5, fp);`

其意义是从 `fp` 所指的文件中, 每次读 4 个字节(一个实数)送入实数组 `fa` 中, 连续读 5 次, 即读 5 个实数到 `fa` 中。

【例】从键盘输入两个学生数据, 写入一个文件中, 再读出这两个学生的数据显示在屏幕上。

```
1. #include<stdio.h>
2. #include<stdlib.h>//exit
3. #include<conio.h>//getch
4.
5. struct stu {
6.     char name[10];
7.     int num;
8.     int age;
9.     char addr[15];
10. }boya[2], boyb[2], *pp, *qq;
11.
12. int main() {
```

```

13. FILE *fp;
14. char ch;
15. int i;
16. pp = boya;
17. qq = boyb;
18. if((fp=fopen("d:\\noob\\example\\stu_list", "wb+")) == NULL) {
19.     printf("Cannot open file strike any key exit!");
20.     getch();
21.     exit(1);
22. }
23. printf("\ninput data\n");
24. for(i = 0; i < 2; i++, pp++)
25.     scanf("%s%d%d%s", pp->name, &pp->num, &pp->age, pp->addr);
26. pp = boya;
27. fwrite(pp, sizeof(struct stu), 2, fp);
28. rewind(fp);
29. fread(qq, sizeof(struct stu), 2, fp);
30. printf("\n\\name\\t\\number age addr\\n");
31. for(i = 0; i < 2; i++, qq++)
32.     printf("%s\\t%5d%7d %s\\n", qq->name, qq->num, qq->age, qq->addr);
33. fclose(fp);
34. return 0;
35. }

```

本例程序定义了一个结构 `stu`, 说明了两个结构数组 `boya` 和 `boyb` 以及两个结构指针变量 `pp` 和 `qq`。 `pp` 指向 `boya`, `qq` 指向 `boyb`。 程序第 16 行以读写方式打开二进制文件 “`stu_list`”，输入二个学生数据之后，写入该文件中，然后把文件内部位置指针移到文件首，读出两块学生数据后，在屏幕上显示。

格式化读写函数 `fscanf` 和 `fprintf`

`fscanf` 函数, `fprintf` 函数与前面使用的 `scanf` 和 `printf` 函数的功能相似, 都是格式化读写函数。两者的区别在于 `fscanf` 函数和 `fprintf` 函数的读写对象不是键盘和显示器, 而是磁盘文件。

这两个函数的调用格式为:

1. `fscanf`(文件指针, 格式字符串, 输入表列);

```
2. fprintf(文件指针, 格式字符串, 输出表列);
```

例如:

```
1. fscanf(fp, "%d%s", &i, s);  
2. fprintf(fp, "%d%c", j, ch);
```

用 `fscanf` 和 `fprintf` 函数也可以完成前面例子中的问题。修改后的程序如下例所示。

【例】用 `fscanf` 和 `fprintf` 函数成前面例子中的问题。

```
1. #include<stdio.h>  
2. #include<stdlib.h>//exit  
3. #include<conio.h>//getch  
4.  
5. struct stu {  
6.     char name[10];  
7.     int num;  
8.     int age;  
9.     char addr[15];  
10. }boya[2], boyb[2], *pp, *qq;  
11.  
12. int main() {  
13.     FILE *fp;  
14.     char ch;  
15.     int i;  
16.     pp = boya;  
17.     qq = boyb;  
18.     if((fp=fopen("stu_list","wb+")) == NULL) {  
19.         printf("Cannot open file strike any key exit!");  
20.         getch();  
21.         exit(1);  
22.     }  
23.     printf("\ninput data\n");  
24.     for(i = 0; i < 2; i++, pp++)  
25.         scanf("%s%d%d%s", pp->name, &pp->num, &pp->age, pp->addr);  
26.     pp = boya;  
27.     for(i = 0; i < 2; i++, pp++)  
28.         fprintf(fp,"%s %d %d %s\n", pp->name, pp->num, pp->age, pp->addr);  
29.     rewind(fp);  
30.     for(i = 0; i < 2; i++, qq++)  
31.         fscanf(fp,"%s %d %d %s\n", qq->name, &qq->num, &qq->age, qq->addr);  
32.     printf("\n\nname\tnumber age addr\n");  
33.     qq = boyb;  
34.     for(i = 0; i < 2; i++, qq++)
```

```
35.         printf("%s\t%5d%7d %s\n", qq->name, qq->num, qq->age, qq->addr);  
36.     fclose(fp);  
37.     return 0;  
38. }
```

与前面的例子相比, 本程序中 `fscanf` 和 `fprintf` 函数每次只能读写一个结构数组元素, 因此采用了循环语句来读写全部数组元素。还要注意指针变量 `pp,qq` 由于循环改变了它们的值, 因此在程序的 26 和 33 行分别对它们重新赋予了数组的首地址。



8.4 随机读写数据文件

前面介绍的对文件的读写方式都是顺序读写, 即读写文件只能从头开始, 顺序读写各个数据。但在实际问题中常要求只读写文件中某一指定的部分。为了解决这个问题可移动文件内部的位置指针到需要读写的位置, 再进行读写, 这种读写称为随机读写。实现随机读写的关键是要按要求移动位置指针, 这称为文件的定位。

文件位置标记及其定位

移动文件内部位置指针的函数主要有两个, 即 `rewind` 函数和 `fseek` 函数。

`rewind` 函数前面已多次使用过, 其调用形式为:

1. `rewind(文件指针);`

它的功能是把文件内部的位置指针移到文件首。

下面主要介绍 `fseek` 函数。

`fseek` 函数用来移动文件内部位置指针, 其调用形式为:

1. `fseek(文件指针, 位移量, 起始点);`

其中:

“文件指针”指向被移动的文件。

“位移量”表示移动的字节数, 要求位移量是 `long` 型数据, 以便在文件长度大于 64KB 时不会出错。当用常量表示位移量时, 要求加后缀“L”。“起始点”表示从何处开始计算位移量, 规定的起始点有三种: 文件首, 当前位置和文件尾。

其表示方法如下表。

起始点	表示符号	数字表示
文件首	SEEK_SET	0
当前位置	SEEK_CUR	1
文件末尾	SEEK_END	2

例如:

1. `fseek(fp, 100L, 0);`

其意义是把位置指针移到离文件首 100 个字节处。

还要说明的是 `fseek` 函数一般用于二进制文件。在文本文件中由于要进行转换, 故往往计算的位置会出现错误。

随机读写

在移动位置指针之后, 即可用前面介绍的任一种读写函数进行读写。由于一般是读写一个数据块, 因此常用 `fread` 和 `fwrite` 函数。

下面用例题来说明文件的随机读写。

【例】在学生文件 `stu_list` 中读出第二个学生的数据。

```
1. #include<stdio.h>
2. #include<stdlib.h>//exit
3. #include<conio.h>//getch
4.
5. struct stu {
6.     char name[10];
7.     int num;
8.     int age;
9.     char addr[15];
10. }boy, *qq;
11.
12. int main() {
13.     FILE *fp;
14.     char ch;
15.     int i = 1;
16.     qq = &boy;
17.     if((fp = fopen("stu_list","rb")) == NULL) {
18.         printf("Cannot open file strike any key exit!");
19.         getch();
20.         exit(1);
21.     }
22.     rewind(fp);
23.     fseek(fp, i*sizeof(struct stu), 0);
24.     fread(qq, sizeof(struct stu), 1, fp);
25.     printf("\n\nname\tnumber age addr\n");
26.     printf("%s\t%d %d %s\n", qq->name, qq->num, qq->age, qq->addr);
27.     return 0;
28. }
```

文件 `stu_list` 已由前面的例子的程序建立, 本程序用随机读出的方法读出第二个学生的数据。程序中定义 `boy` 为 `stu` 类型变量, `qq` 为指向 `boy` 的指针。以读二进制文件方式打开文件, 程序第 22 行移动文件位置指针。其中的 `i` 值为 1, 表示从文件头开始, 移动一个 `stu` 类型的长度, 然后再读出的数据即为第二个学生的数据。

8.5 文件读写的出错检测

C 语言中常用的文件检测函数有以下几个。

1、文件结束检测函数 feof 函数

调用格式：

1. feof(文件指针);

功能：判断文件是否处于文件结束位置，如文件结束，则返回值为 1，否则为 0。

2、读写文件出错检测函数

ferror 函数调用格式：

1. ferror(文件指针);

功能：检查文件在用各种输入输出函数进行读写时是否出错。如 ferror 返回值为 0 表示未出错，否则表示有错。

3、文件出错标志和文件结束标志置 0 函数

clearerr 函数调用格式：

1. clearerr(文件指针);

功能：本函数用于清除出错标志和文件结束标志，使它们为 0 值。

8.6 本章小结

经验总结

1. C 系统把文件当作一个“流”，按字节进行处理。
2. C 文件按编码方式分为二进制文件和 ASCII 文件。
3. C 语言中，用文件指针标识文件，当一个文件被 打开时，可取得该文件指针。
4. 文件在读写之前必须打开，读写结束必须关闭。
5. 文件可按只读、只写、读写、追加四种操作方式打开，同时还必须指定文件的类型是二进制文件还是文本文件。
6. 文件可按字节，字符串，数据块为单位读写，文件也可按指定的格式进行读写。
7. 文件内部的位置指针可指示当前的读写位置，移动该指针可以对文件实现随机读写。

C 语言专业题库在线练习

<http://www.noobdream.com/Practice/clang/>

第九章 预处理和位运算

【本章知识点汇总】

宏定义

文件包含

条件编译

位运算

位域



本书配套视频精讲: <https://www.bilibili.com/video/BV1HJ41137fe>

9.1 宏定义

在前面各章中, 已多次使用过以“#”号开头的预处理命令。如包含命令`#include`, 宏定义命令`#define`等。在源程序中这些命令都放在函数之外, 而且一般都放在源文件的前面, 它们称为预处理部分。所谓预处理是指在进行编译的第一遍扫描(词法扫描和语法分析)之前所作的工作。预处理是C语言的一个重要功能, 它由预处理程序负责完成。当对一个源文件进行编译时, 系统将自动引用预处理程序对源程序中的预处理部分作处理, 处理完毕自动进入对源程序的编译。C语言提供了多种预处理功能, 如宏定义、文件包含、条件编译等。合理地使用预处理功能编写的程序便于阅读、修改、移植和调试, 也有利于模块化程序设计。本章介绍常用的几种预处理功能。

在C语言源程序中允许用一个标识符来表示一个字符串, 称为“宏”。被定义为“宏”的标识符称为“宏名”。在编译预处理时, 对程序中所有出现的“宏名”, 都用宏定义中的字符串去代换, 这称为“宏代换”或“宏展开”。宏定义是由源程序中的宏定义命令完成的。宏代换是由预处理程序自动完成的。在C语言中, “宏”分为有参数和无参数两种。下面分别讨论这两种“宏”的定义和调用。

无参宏定义

无参宏的宏名后不带参数。其定义的一般形式为:

1. `#define 标识符 字符串`

其中的“#”表示这是一条预处理命令。凡是以“#”开头的均为预处理命令。“`define`”为宏定义命令。“标识符”为所定义的宏名。“字符串”可以是常数、表达式、格式串等。在前面介绍过的符号常量的定义就是一种无参宏定义。此外, 常对程序中反复使用的表达式进行宏定义。

例如:

1. `#define M (y*y+3*y)`

它的作用是指定标识符 `M` 来代替表达式`(y*y+3*y)`。在编写源程序时, 所有的`(y*y+3*y)`都可由 `M` 代替, 而对源程序作编译时, 将先由预处理程序进行宏代换, 即用`(y*y+3*y)`表达式去置换所有的宏名 `M`, 然后再进行编译。

【例】

```
1. #include<stdio.h>
2. #define M (y*y+3*y)
```

```

3.
4. int main() {
5.     int s, y;
6.     printf("input a number: ");
7.     scanf("%d", &y);
8.     s = 3 * M + 4 * M + 5 * M;
9.     printf("s=%d\n", s);
10.    return 0;
11. }

```

上例程序中首先进行宏定义, 定义 M 来替代表达式 $(y*y+3*y)$, 在 $s=3*M+4*M+5*M$ 中作了宏调用。在预处理时经宏展开后该语句变为:

```
1. s = 3 * (y*y+3*y) + 4 * (y*y+3*y) + 5 * (y*y+3*y);
```

但要注意的是, 在宏定义中表达式 $(y*y+3*y)$ 两边的括号不能少。否则会发生错误。如当作以下定义后:

```
1. #define M y*y+3*y
```

在宏展开时将得到下述语句:

```
1. s = 3*y*y+3*y + 4*y*y+3*y + 5*y*y+3*y;
```

这相当于:

$$3y^2+3y+4y^2+3y+5y^2+3y;$$

显然与原题意要求不符。计算结果当然是错误的。因此在作宏定义时必须十分注意。应保证在宏代换之后不发生错误。

对于宏定义还要说明以下几点:

- 1) 宏定义是用宏名来表示一个字符串, 在宏展开时又以该字符串取代宏名, 这只是一种简单的代换, 字符串中可以含任何字符, 可以是常数, 也可以是表达式, 预处理程序对它不作任何检查。如有错误, 只能在编译已被宏展开后的源程序时发现。
- 2) 宏定义不是说明或语句, 在行末不必加分号, 如加上分号则连分号也一起置换。
- 3) 宏定义必须写在函数之外, 其作用域为宏定义命令起到源程序结束。如要终止其作用域可使用 `#undef` 命令。

例如:

```

1. #define PI 3.14159
2. int main()
3. {
4.     .....

```

```
5. }
6.
7. #undef PI
8. f1()
9. {
10.     .....
11. }
```

表示 PI 只在 main 函数中有效, 在 f1 中无效。

4) 宏名在源程序中若用引号括起来, 则预处理程序不对其作宏代换。

【例】

```
1. #include<stdio.h>
2. #define OK 100
3.
4. int main() {
5.     printf("OK");
6.     printf("\n");
7.     return 0;
8. }
```

上例中定义宏名 OK 表示 100, 但在 printf 语句中 OK 被引号括起来, 因此不作宏代换。程序的运行结果为: OK 这表示把“OK”当字符串处理。

5) 宏定义允许嵌套, 在宏定义的字符串中可以使用已经定义的宏名。在宏展开时由预处理程序层层代换。

例如:

```
1. #define PI 3.1415926
2. #define S PI*y*y /* PI 是已定义的宏名*/
```

对语句:

```
1. printf("%f", S);
```

在宏代换后变为:

```
1. printf("%f", 3.1415926*y*y);
```

6) 习惯上宏名用大写字母表示, 以便于与变量区别。但也允许用小写字母。

7) 可用宏定义表示数据类型, 使书写方便。

例如:

```
1. #define STU struct stu
```

在程序中可用 STU 作变量说明:

```
1. STU body[5], *p;
2. #define INTEGER int
```

在程序中即可用 `INTEGER` 作整型变量说明:

```
1. INTEGER a, b;
```

应注意用宏定义表示数据类型和用 `typedef` 定义数据说明符的区别。

宏定义只是简单的字符串代换, 是在预处理完成的, 而 `typedef` 是在编译时处理的, 它不是作简单的代换, 而是对类型说明符重新命名。被命名的标识符具有类型定义说明的功能。

请看下面的例子:

```
1. #define PIN1 int *  
2. typedef (int *) PIN2;
```

从形式上看这两者相似, 但在实际使用中却不相同。

下面用 `PIN1`, `PIN2` 说明变量时就可以看出它们的区别:

```
1. PIN1 a, b; 在宏代换后变成:  
2. int *a, b;
```

表示 `a` 是指向整型的指针变量, 而 `b` 是整型变量。

然而:

```
1. PIN2 a, b;
```

表示 `a, b` 都是指向整型的指针变量。因为 `PIN2` 是一个类型说明符。由这个例子可见, 宏定义虽然也可表示数据类型, 但毕竟是作字符代换。在使用时要分外小心, 以避出错。

8) 对“输出格式”作宏定义, 可以减少书写麻烦。

【例】中就采用了这种方法。

```
1. #include<stdio.h>  
2. #define P printf  
3. #define D "%d\n"  
4. #define F "%f\n"  
5.  
6. int main(){  
7.     int a=5, c=8, e=11;  
8.     float b=3.8, d=9.7, f=21.08;  
9.     P(D F,a,b);  
10.    P(D F,c,d);  
11.    P(D F,e,f);  
12.    return 0;  
13. }
```

9.2 文件包含

文件包含是 C 预处理程序的另一个重要功能。

文件包含命令的一般形式为：

```
1. #include "文件名"
```

在前面我们已多次用此命令包含过库函数的头文件。例如：

```
1. #include "stdio.h"
```

```
2. #include "math.h"
```

文件包含命令的功能是把指定的文件插入该命令行位置取代该命令行，从而把指定的文件和当前的源程序文件连成一个源文件。在程序设计中，文件包含是很有用的。一个大的程序可以分为多个模块，由多个程序员分别编程。有些公用的符号常量或宏定义等可单独组成一个文件，在其它文件的开头用包含命令包含该文件即可使用。这样，可避免在每个文件开头都去书写那些公用量，从而节省时间，并减少出错。

对文件包含命令还要说明以下几点：

1. 包含命令中的文件名可以用双引号括起来，也可以用尖括号括起来。例如以下写法都是允许的：

```
1. #include "stdio.h"
```

```
2. #include <math.h>
```

但是这两种形式是有区别的：

使用尖括号表示在包含文件目录中去查找(包含目录是由用户在设置环境时设置的)，而不在源文件目录去查找；使用双引号则表示首先在当前的源文件目录中查找，若未找到才到包含目录中去查找。用户编程时可根据自己文件所在的目录来选择某一种命令形式。

2. 一个 `include` 命令只能指定一个被包含文件，若有多个文件要包含，则需用多个 `include` 命令。

3. 文件包含允许嵌套，即在一个被包含的文件中又可以包含另一个文件。

9.3 条件编译

预处理程序提供了条件编译的功能。可以按不同的条件去编译不同的程序部分，因而产生不同的目标代码文件。这对于程序的移植和调试是很有用的。

条件编译有三种形式，下面分别介绍：

1. 第一种形式：

1. `#ifdef` 标识符
2. 程序段 1
3. `#else`
4. 程序段 2
5. `#endif`

它的功能是，如果标识符已被 `#define` 命令定义过则对程序段 1 进行编译；否则对程序段 2 进行编译。如果没有程序段 2(它为空白)，本格式中的`#else` 可以没有，即可以写为：

1. `#ifdef` 标识符
2. 程序段
3. `#endif`

【例】

```
1. #include<stdio.h>
2. #include<stdlib.h>
3. #define NUM ok
4.
5. int main() {
6.     struct stu {
7.         int num;
8.         char *name;
9.         char sex;
10.        float score;
11.    } *ps;
12.    ps = (struct stu*)malloc(sizeof(struct stu));
13.    ps->num = 102;
14.    ps->name = "Zhang ping";
15.    ps->sex = 'M';
16.    ps->score = 62.5;
17.    #ifdef NUM
18.        printf("Number=%d\nScore=%f\n",ps->num,ps->score);
19.    #else
20.        printf("Name=%s\nSex=%c\n",ps->name,ps->sex);
```



```
21.     #endif
22.     free(ps);
23.     return 0;
24. }
```

由于在程序的第 16 行插入了条件编译预处理命令, 因此要根据 NUM 是否被定义过来决定编译那一个 printf 语句。而在程序的第一行已对 NUM 作过宏定义, 因此应对第一个 printf 语句作编译故运行结果是输出了学号和成绩。

在程序的第一行宏定义中, 定义 NUM 表示字符串 OK, 其实也可以为任何字符串, 甚至不给出任何字符串, 写为:

```
1. #define NUM
```

也具有同样的意义。只有取消程序的第一行才会去编译第二个 printf 语句。读者可上机试作。

2. 第二种形式:

```
1. #ifndef 标识符
2.     程序段 1
3. #else
4.     程序段 2
5. #endif
```

与第一种形式的区别是将“ifdef”改为“ifndef”。它的功能是, 如果标识符未被#define 命令定义过则对程序段 1 进行编译, 否则对程序段 2 进行编译。这与第一种形式的功能正相反。

3. 第三种形式:

```
1. #if 常量表达式
2.     程序段 1
3. #else
4.     程序段 2
5. #endif
```

它的功能是, 如常量表达式的值为真(非 0), 则对程序段 1 进行编译, 否则对程序段 2 进行编译。因此可以使程序在不同条件下, 完成不同的功能。

【例】

```
1. #include<stdio.h>
2. #define R 1
3.
4. int main(){
```



```
5.     float c, r, s;  
6.     printf("input a number: ");  
7.     scanf("%f",&c);  
8.     #if R  
9.         r=3.14159*c*c;  
10.        printf("area of round is: %f\n",r);  
11.    #else  
12.        s=c*c;  
13.        printf("area of square is: %f\n",s);  
14.    #endif  
15.    return 0;  
16. }
```

本例中采用了第三种形式的条件编译。在程序第一行宏定义中，定义 R 为 1，因此在条件编译时，常量表达式的值为真，故计算并输出圆面积。

上面介绍的条件编译当然也可以用条件语句来实现。但是用条件语句将会对整个源程序进行编译，生成的目标代码程序很长，而采用条件编译，则根据条件只编译其中的程序段 1 或程序段 2，生成的目标程序较短。如果条件选择的程序段很长，采用条件编译的方法是十分必要的。

noobdream.com

9.4 位运算

前面介绍的各种运算都是以字节作为最基本位进行的。但在很多系统程序中常要求在位(bit)一级进行运算或处理。C语言提供了位运算的功能,这使得C语言也能像汇编语言一样用来编写系统程序。

位运算符C语言提供了六种位运算符:

- & 按位与
- | 按位或
- ^ 按位异或
- ~ 取反
- << 左移
- >> 右移

按位与运算

按位与运算符"&"是双目运算符。其功能是参与运算的两数各对应的二进位相与。只有对应的两个二进位均为1时,结果位才为1,否则为0。参与运算的数以补码方式出现。

例如: $9 \& 5$ 可写算式如下:

1. 00001001 (9 的二进制补码)
2. &00000101 (5 的二进制补码)
3. 00000001 (1 的二进制补码)

可见 $9 \& 5 = 1$ 。

按位与运算通常用来对某些位清0或保留某些位。例如把a的高八位清0,保留低八位,可作 $a \& 255$ 运算(255的二进制数为0000000011111111)。

【例】

```
1. #include<stdio.h>
2.
3. int main(){
4.     int a = 9, b = 5, c;
5.     c = a & b;
6.     printf("a=%d\nb=%d\nc=%d\n", a, b, c);
7.     return 0;
```

```
8. }
```

按位或运算

按位或运算符“|”是双目运算符。其功能是参与运算的两数各对应的二进位相或。只要对应的二个二进位有一个为 1 时，结果位就为 1。参与运算的两个数均以补码出现。

例如：9|5 可写算式如下：

```
1. 00001001
2. |00000101
3. 00001101 (十进制为 13)可见 9|5=13
```

【例】

```
1. #include<stdio.h>
2.
3. int main(){
4.     int a = 9, b = 5, c;
5.     c = a | b;
6.     printf("a=%d\nb=%d\nc=%d\n", a, b, c);
7.     return 0;
8. }
```

noobdream.com

按位异或运算

按位异或运算符“^”是双目运算符。其功能是参与运算的两数各对应的二进位相异或，当两对应的二进位相异时，结果为 1。参与运算数仍以补码出现，例如 9^5 可写成算式如下：

```
1. 00001001
2. ^00000101
3. 00001100 (十进制为 12)
```

【例】

```
1. #include<stdio.h>
2.
3. int main(){
4.     int a = 9;
5.     a = a ^ 5;
```

```
6.     printf("a=%d\n", a);
7.     return 0;
8. }
```

求反运算

求反运算符 \sim 为单目运算符，具有右结合性。其功能是对参与运算的数的各二进制位按位求反。

例如 ~ 9 的运算为：

1. $\sim(0000000000001001)$ 结果为：111111111110110

左移运算

左移运算符“ \ll ”是双目运算符。其功能把“ \ll ”左边的运算数的各二进制位全部左移若干位，由“ \ll ”右边的数指定移动的位数，高位丢弃，低位补 0。

例如：

1. $a \ll 4$

指把 a 的各二进制位向左移动 4 位。如 $a=00000011$ (十进制 3)，左移 4 位后为 00110000(十进制 48)。

noobdream.com

右移运算

右移运算符“ \gg ”是双目运算符。其功能是把“ \gg ”左边的运算数的各二进制位全部右移若干位，“ \gg ”右边的数指定移动的位数。

例如：

```
1.  a = 15;
2.  a >> 2;
```

表示把 000001111 右移为 00000011(十进制 3)。应该说明的是，对于有符号数，在右移时，符号位将随同移动。当为正数时，最高位补 0，而为负数时，符号位为 1，最高位是补 0 或是补 1 取决于编译系统的规定。Turbo C 和很多系统规定为补 1。

【例】

```
1. #include<stdio.h>
2.
3. int main(){
4.     unsigned a, b;
5.     printf("input a number: ");
6.     scanf("%d", &a);
7.     b = a >> 5;
8.     b = b & 15;
9.     printf("a=%d\tb=%d\n",a,b);
10.    return 0;
11. }
```

【例】

```
1. #include<stdio.h>
2.
3. int main(){
4.     char a='a', b='b';
5.     int p, c, d;
6.     p = a;
7.     p = (p <<8 ) | b;
8.     d = p & 0xff;
9.     c = (p & 0xff00) >> 8;
10.    printf("a=%d\nb=%d\nc=%d\nd=%d\n", a, b, c, d);
11.    return 0;
12. }
```

9.5 位域

有些信息在存储时，并不需要占用一个完整的字节，而只需占几个或一个二进制位。例如在存放一个开关量时，只有 0 和 1 两种状态，用一位二进制位即可。为了节省存储空间，并使处理简便，C 语言又提供了一种数据结构，称为“位域”或“位段”。所谓“位域”是把一个字节中的二进制位划分为几个不同的区域，并说明每个区域的位数。每个域有一个域名，允许在程序中按域名进行操作。这样就可以把几个不同的对象用一个字节的二进制位域来表示。

1. 位域的定义和位域变量的说明

位域定义与结构定义相仿，其形式为：

1. **struct** 位域结构名 {
2. 位域列表
3. };

其中位域列表的形式为：

1. 类型说明符 位域名: 位域长度

例如：

1. **struct** bs {
2. **int** a:8;
3. **int** b:2;
4. **int** c:6;
5. };

位域变量的说明与结构变量说明的方式相同。可采用先定义后说明，同时定义说明或者直接说明这三种方式。

例如：

1. **struct** bs {
2. **int** a:8;
3. **int** b:2;
4. **int** c:6;
5. }data;

说明 data 为 bs 变量，共占两个字节。其中位域 a 占 8 位，位域 b 占 2 位，位域 c 占 6 位。

对于位域的定义尚有以下几点说明:

1) 一个位域必须存储在同一个字节中, 不能跨两个字节。如一个字节所剩空间不够存放另一位域时, 应从下一单元起存放该位域。也可以有意使某位域从下一单元开始。

例如:

```
1. struct bs {  
2.     unsigned a:4  
3.     unsigned :0 /*空域*/  
4.     unsigned b:4 /*从下一单元开始存放*/  
5.     unsigned c:4  
6. }
```

在这个位域定义中, a 占第一字节的 4 位, 后 4 位填 0 表示不使用, b 从第二字节开始, 占用 4 位, c 占用 4 位。

2) 由于位域不允许跨两个字节, 因此位域的长度不能大于一个字节的长度, 也就是说不能超过 8 位二进制。

3) 位域可以无位域名, 这时它只用来作填充或调整位置。无名的位域是不能使用的。

例如:

```
1. struct k {  
2.     int a:1  
3.     int :2 /*该 2 位不能使用*/  
4.     int b:3  
5.     int c:2  
6. };
```

从以上分析可以看出, 位域在本质上就是一种结构类型, 不过其成员是按二进制分配的。

2. 位域的使用

位域的使用和结构成员的使用相同, 其一般形式为:

```
1. 位域变量名·位域名
```

位域允许用各种格式输出。

【例】

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

```

2.
3. int main(){
4.     struct bs {
5.         unsigned a:1;
6.         unsigned b:3;
7.         unsigned c:4;
8.     } bit, *pbit;
9.     bit.a = 1;
10.    bit.b = 7;
11.    bit.c = 15;
12.    printf("%d,%d,%d\n", bit.a, bit.b, bit.c);
13.    pbit = &bit;
14.    pbit->a = 0;
15.    pbit->b &= 3;
16.    pbit->c |= 1;
17.    printf("%d,%d,%d\n", pbit->a, pbit->b, pbit->c);
18.    return 0;
19. }

```

上例程序中定义了位域结构 `bs`，三个位域为 `a,b,c`。说明了 `bs` 类型的变量 `bit` 和指向 `bs` 类型的指针变量 `pbit`。这表示位域也是可以使用指针的。程序的 9、10、11 三行分别给三个位域赋值(应注意赋值不能超过该位域的允许范围)。程序第 12 行以整型量格式输出三个域的内容。第 13 行把位域变量 `bit` 的地址送给指针变量 `pbit`。第 14 行用指针方式给位域 `a` 重新赋值，赋为 0。第 15 行使用了复合的位运算符"`&=`"，该行相当于：

```
1. pbit->b = pbit->b & 3
```

位域 `b` 中原有值为 7，与 3 作按位与运算的结果为 3($111 \& 011 = 011$, 十进制值为 3)。同样，程序第 16 行中使用了复合位运算符"`|=`"，相当于：

```
1. pbit->c = pbit->c | 1
```

其结果为 15。程序第 17 行用指针方式输出了这三个域的值。

9.6 本章小结

经验总结

1. 预处理功能是C语言特有的功能，它是在对源程序正式编译前由预处理程序完成的。程序员在程序中用预处理命令来调用这些功能。
2. 宏定义是用一个标识符来表示一个字符串，这个字符串可以是常量、变量或表达式。在宏调用中将用该字符串代换宏名。
3. 宏定义可以带有参数，宏调用时是以实参代换形参。而不是“值传送”。
4. 为了避免宏代换时发生错误，宏定义中的字符串应加括号，字符串中出现的形式参数两边也应加括号。
5. 文件包含是预处理的一个重要功能，它可用来把多个源文件连接成一个源文件进行编译，结果将生成一个目标文件。
6. 条件编译允许只编译源程序中满足条件的程序段，使生成的目标程序较短，从而减少了内存的开销并提高了程序的效率。
7. 使用预处理功能便于程序的修改、阅读、移植和调试，也便于实现模块化程序设计。

1. 位运算是C语言的一种特殊运算功能，它是以二进制位为单位进行运算的。位运算符只有逻辑运算和移位运算两类。位运算符可以与赋值符一起组成复合赋值符。如`&=`, `|=`, `^=`, `>>=`, `<<=`等。
2. 利用位运算可以完成汇编语言的某些功能，如置位，位清零，移位等。还可进行数据的压缩存储和并行运算。
3. 位域在本质上也是结构类型，不过它的成员按二进制位分配内存。其定义、说明及使用的方式都与结构相同。
4. 位域提供了一种手段，使得可在高级语言中实现数据的压缩，节省了存储空间，同时也提高了程序的效率。

C语言专业题库在线练习

<http://www.noobdream.com/Practice/clang/>

完结撒花

当你看到这里, 说明你已经读完了本书所有的内容。恭喜你, 学完 C 语言考研复习攻略的全部内容, 我们相信你一定会在考试中取得非常不错的成绩。

如果本书对你有所帮助, 希望你能在考试之后将还能记得的考试题目发表在 N 诺的交流区里与同学们分享讨论, 也可以帮助下一届学弟学妹, 让他们不用再费力地去收集真题, 少走一些弯路, 并且我们会将你的名字或 N 诺 ID 放在题目的后面进行特别鸣谢。

最后, 我们不仅希望你能在 C 语言考试中取得满分的成绩, 也希望你能如愿以偿的考上心目中理想的院校, 加油! Go!Go!Go!

一定要做的说明: N 诺出版的考研系列书籍都将以**电子版**的形式进行发布更新, 需要纸质版的同学自行打印学习即可。

N 诺考研系列书籍为什么只发布电子版不发布纸质版？

原因一：发布纸质版需要提前很久将书籍整理成册，时间太赶容易敷衍了事，我们相信慢工出细活。

原因二：纸质版一经印刷，便无法修改，就算发现问题或者想对某些内容进行优化也没办法。而电子版可以随时勘误进行修改，灵光一现的时候还能对前面写的不够好的地方进行优化。

原因三：电子版少了中间商，可以给同学们节约更多的费用。纸质版的话出版社、印刷商都要从中获取利润，最终羊毛出在羊身上。



如何获取 N 诺考研系列书籍？

noobdream.com

访问 N 诺平台（www.noobdream.com）的兑换中心即可兑换或购买各种你想要的书籍或资料。

另外，**本书会不断的进行更新，所以需要最新版的同学，请去官网兑换中心进行兑换**，只要一次兑换，后续版本更新都可看到，不用重复兑换。

本书每个月都会进行一次版本迭代，如果书中有错别字或者对本书有其他建议可以向官方群管理员反馈，在下一次版本迭代中就会进行修正更新。

N 诺考研系列图书

《计算机考研报考指南》

《C 语言考研复习攻略》

《数据结构考研复习攻略》

《操作系统考研复习攻略》

《计算机网络考研复习攻略》

《计算机组成原理考研复习攻略》

《数据库考研复习攻略》

《计算机考研机试攻略 - 高分篇》

《计算机考研机试攻略 - 满分篇》

考研路上，N 诺与你携手同行。

N 诺 Offer 训练营

感谢同学们一直以来对 N 诺不遗余力的支持，N 诺能快速发展起来离不开每一个 N 诺 er 的帮助。

计算机专业是一个靠技术实力说话的专业，很多同学的编程功底和项目水平都不是很好，在找工作的时候很容易处处碰壁。

N 诺里有非常多的大佬，N 诺技术团队有来自腾讯、阿里、字节、百度等各个大厂的同学，我们希望能帮助更多的同学提升自己的编程水平，最终拿到一个满意的 Offer。

同学们可以把参加 N 诺 Offer 训练营视为一次对自己的投资，投资未来，请相信自己的潜力也请相信 N 诺的能力。多年以后，回想起来，相信参加 N 诺 Offer 训练营会是你人生中一次重要的转折点。

N 诺 Offer 训练营的面向人群

1、面向就业

大学不是人生的终点，很多同学大学毕业之后就会面临着找工作的问题，提升自己的技术能力可以找到一份更好的工作，不用担心毕业即失业的烦恼。

N 诺 Offer 训练营致力于给有梦想、肯拼搏、敢奋斗的同学提高最好的平台！

2、面向硕士

研究生也不是人生的终点，很多同学读研之后虽然在学历上进行了一次升级，但是如果技术能力不行，依然很难找到一份满意的工作。

所以趁着读研留出来的缓冲时间，通过 Offer 训练营提升自己的能力，让自己的未来可以自由选择！

3、课程形式

由于有很多同学不方便参与线下的培训，所以 Offer 训练营有线上班和线下班两种供同学

们灵活选择。

4、报名要求

具备本科学历，愿意通过奋斗去实现人生的价值。

参与线下班的同学需要完成开课前作业，用作业考察态度，筛选出有决心、有毅力改变自己的同学。

5、你将获得

编程能力的迅速提升，结合项目实战，逐步打下坚实的编程基础，培养积极、主动的学习能力。同学们在训练营里从小白逐步成长为大佬，训练营中不少本科同学就拿到了阿里、腾讯、头条、百度、美团等一线互联网大厂的 Offer，研究生同学更是手握多个 Offer 可以挑选。

N 诺 Offer 训练营的优势

Offer 训练营的技术学习氛围浓厚，同学们乐于分享与交流，能更加专注于提升自身能力。

N 诺的技术团队都是顶级的大佬，具有多年的教学经验，能帮助零基础的同学找到快速提高编程能力的学习方式。

N 诺 Offer 训练营的课程信息

Offer 训练营分为线上和线下两种模式，开设 4 种班型：

- 实习一对一（针对大一、大二、研一的同学）
- 校招一对一（针对大三、大四、研二、研三的同学）
- 跨专业一对一（针对非计算机专业的在校学生）
- 社招一对一（针对已毕业的同学）

要想了解 N 诺 Offer 训练营的方方面面，可以登录 N 诺官网（noobdream.com）查看。

最后，感谢缘分让我们在 N 诺相遇，愿每一个 N 诺 er，所有的坚持都不被辜负，所有的梦想都能绽放出耀眼的光芒~