

谭浩强 c 语言程序设计

1 C 语言概述

1.1 C 语言的发展过程

1.2 当代最优秀的程序设计语言

1.3 C 语言版本

1.4 C 语言的特点

1.5 面向对象的程序设计语言

1.6 C 和 C ++

1.7 简单的 C 程序介绍

1.8 输入和输出函数

1.9 C 源程序的结构特点

1.10 书写程序时应遵循的规则

1.11 C 语言的字符集

1.12 C 语言词汇

1.13 Turbo C 2.0 集成开发环境的使用

1.13.1 Turbo C 2.0 简介和启动

1.13.2 Turbo C 2.0 集成开发环境

1.13.3 File 菜单

1.13.4 Edit 菜单

1.13.5 Run 菜单

1.13.6 Compile 菜单

1.13.7 Project 菜单

1.13.8 Options 菜单

1.13.9 Debug 菜单

1.13.10 Break/watch 菜单

1.13.11 Turbo C 2.0 的配置文件

2 程序的灵魂—算法

2.1 算法的概念 21

2.2 简单算法举例 21

2.3 算法的特性 24

2.4 怎样表示一个算法 24

2.4.1 用自然语言表示算法 24

2.4.2 用流程图表示算法 24

2.4.3 三种基本结构和改进的流程图 28

2.4.4 用 N-S 流程图表示算法 29

2.4.5 用伪代码表示算法 30

2.4.6 用计算机语言表示算法 31

2.5 结构化程序设计方法 31

3 数据类型、运算符与表达式

3.1 C 语言的数据类型 32

3.2 常量与变量 33

3.2.1	常量和符号常量	33
3.2.2	变量	33
3.3	整型数据	34
3.3.1	整型常量的表示方法	34
3.3.2	整型变量	35
3.4	实型数据	37
3.4.1	实型常量的表示方法	37
3.4.2	实型变量	38
3.4.3	实型常数的类型	39
3.5	字符型数据	39
3.5.1	字符常量	39
3.5.2	转义字符	39
3.5.3	字符变量	40
3.5.4	字符数据在内存中的存储形式及使用方法	41
3.5.5	字符串常量	41
3.5.6	符号常量	42
3.6	变量赋初值	42
3.7	各类数值型数据之间的混合运算	43
3.8	算术运算符和算术表达式	44
3.8.1	C 运算符简介	44
3.8.2	算术运算符和算术表达式	45
3.9	赋值运算符和赋值表达式	47

3.10 逗号运算符和逗号表达式 48

3.11 小结 49

3.11.1 C 的数据类型 49

3.11.2 基本类型的分类及特点 49

3.11.3 常量后缀 49

3.11.4 常量类型 49

3.11.5 数据类型转换 49

3.11.6 运算符优先级和结合性 50

表达式 50

4 最简单的 C 程序设计—顺序程序设计

4.1 C 语句概述 51

4.2 赋值语句 53

4.3 数据输入输出的概念及在 C 语言中的实现 54

4.4 字符数据的输入输出 54

4.4.1 putchar 函数（字符输出函数） 54

4.4.2 getchar 函数（键盘输入函数） 55

4.5 格式输入与输出 55

4.5.1 printf 函数（格式输出函数） 56

4.5.2 scanf 函数(格式输入函数) 58

顺序结构程序设计举例 60

5 分支结构程序

5.1 关系运算符和表达式 61

5.1.1 关系运算符及其优先次序 61

5.1.2 关系表达式 61

5.2 逻辑运算符和表达式 62

5.2.1 逻辑运算符及其优先次序 62

5.2.2 逻辑运算的值 63

5.2.3 逻辑表达式 63

5.3 if 语句 64

5.3.1 if 语句的三种形式 64

5.3.2 if 语句的嵌套 67

5.3.3 条件运算符和条件表达式 69

5.4 switch 语句 70

5.5 程序举例 71

6 循环控制

6.1 概述 71

6.2 goto 语句以及用 goto 语句构成循环 71

6.3 while 语句 72

6.4 do-while 语句 74

6.5 for 语句 76

6.6 循环的嵌套 79

6.7 几种循环的比较	79
6.8 break 和 continue 语句	79
6.8.1 break 语句	79
6.8.2 continue 语句	80
6.9 程序举例	81
7 数组	
7.1 一维数组的定义和引用	82
7.1.1 一维数组的定义方式	82
7.1.2 一维数组元素的引用	83
7.1.3 一维数组的初始化	84
7.1.4 一维数组程序举例	84
7.2 二维数组的定义和引用	86
7.2.1 二维数组的定义	86
7.2.2 二维数组元素的引用	86
7.2.3 二维数组的初始化	87
7.2.4 二维数组程序举例	89
7.3 字符数组	89
7.3.1 字符数组的定义	89
7.3.2 字符数组的初始化	89
7.3.3 字符数组的引用	90
7.3.4 字符串和字符串结束标志	91

7.3.5	字符数组的输入输出	91
7.3.6	字符串处理函数	92
7.4	程序举例	94
	本章小结	97
8	函 数	
8.1	概述	98
8.2	函数定义的一般形式	99
8.3	函数的参数和函数的值	100
8.3.1	形式参数和实际参数	101
8.3.2	函数的返回值	102
8.4	函数的调用	106
8.4.1	函数调用的一般形式	106
8.4.2	函数调用的方式	106
8.4.3	被调用函数的声明和函数原型	107
8.5	函数的嵌套调用	108
8.6	函数的递归调用	109
8.7	数组作为函数参数	110
8.8	局部变量和全局变量	112
8.8.1	局部变量	113
8.8.2	全局变量	119
8.9	变量的存储类别	120

8.9.1 动态存储方式与静态动态存储方式 120

8.9.2 auto 变量 120

8.9.3 用 static 声明局部变量 121

8.9.4 register 变量 122

用 extern 声明外部变量 123

9 预处理命令

9.1 概述 124

9.2 宏定义 125

9.2.1 无参宏定义 126

9.2.2 带参宏定义 127

9.3 文件包含 128

9.4 条件编译 130

9.5 本章小结

10 指针

10.1 地址指针的基本概念 131

10.2 变量的指针和指向变量的指针变量 132

10.2.1 定义一个指针变量 133

10.2.2 指针变量的引用 133

10.2.3 指针变量作为函数参数 137

10.2.4 指针变量几个问题的进一步说明 140

10.3	数组指针和指向数组的指针变量	141
10.3.1	指向数组元素的指针	142
10.3.2	通过指针引用数组元素	143
10.3.3	数组名作函数参数	146
10.3.4	指向多维数组的指针和指针变量	148
10.4	字符串的指针指向字符串的指针变量	150
10.4.1	字符串的表示形式	152
10.4.2	使用字符串指针变量与字符数组的区别	158
10.5	函数指针变量	159
10.6	指针型函数	160
10.7	指针数组和指向指针的指针	161
10.7.1	指针数组的概念	161
10.7.2	指向指针的指针	164
10.7.3	main 函数的参数	166
10.8	有关指针的数据类型和指针运算的小结	167
10.8.1	有关指针的数据类型的小结	167
10.8.2	指针运算的小结	167
10.8.3	void 指针类型	168
11	结构体与共用体	
11.1	定义一个结构的一般形式	170
11.2	结构类型变量的说明	172

11.3	结构变量成员的表示方法	174
11.4	结构变量的赋值	174
11.5	结构变量的初始化	175
11.6	结构数组的定义	175
11.7	结构指针变量的说明和使用	177
11.7.1	指向结构变量的指针	177
11.7.2	指向结构数组的指针	179
11.7.3	结构指针变量作函数参数	180
11.8	动态存储分配	181
11.9	链表的概念	182
11.10	枚举类型	184
11.10.1	枚举类型的定义和枚举变量的说明	184
11.10.2	枚举类型变量的赋值和使用	185
11.11	类型定义符 typedef	
12	位运算	
12.1	位运算符 C 语言提供了六种位运算符：	189
12.1.1	按位与运算	191
12.1.2	按位或运算	192
12.1.3	按位异或运算	192
12.1.4	求反运算	193
12.1.5	左移运算	193

12.1.6	右移运算	193
12.2	位域（位段）	194
12.3	本章小结	
13	文件	
13.1	C 文件概述	197
13.2	文件指针	198
13.3	文件的打开与关闭	199
13.3.1	文件的打开(fopen 函数)	200
13.3.2	文件关闭函数（fclose 函数）	202
13.4	文件的读写	204
13.4.1	字符读写函数 fgetc 和 fputc	204
13.4.2	字符串读写函数 fgets 和 fputs	208
13.4.3	数据块读写函数 fread 和 fwrite	209
13.4.4	格式化读写函数 fscanf 和 fprintf	201
13.5	文件的随机读写	202
13.5.1	文件定位	202
13.5.2	文件的随机读写	203
13.6	文件检测函数	204
13.6.1	文件结束检测函数 feof 函数	204
13.6.2	读写文件出错检测函数	205

13.6.3 文件出错标志和文件结束标志置 0 函数 206

13.7 C 库文件 208

13.8 本章小结

C 语言教程

1 C 语言概述

1.1 C 语言的发展过程

C 语言是在 70 年代初问世的。一九七八年由美国电话电报公司(AT&T)贝尔实验室正式发表了 C 语言。同时由 B.W.Kernighan 和 D.M.Ritchie 合著了著名的“THE C PROGRAMMING LANGUAGE”一书。通常简称为《K&R》，也有人称之为《K&R》标准。但是，在《K&R》中并没有定义一个完整的标准 C 语言，后来由美国国家标准协会(American National Standards Institute)在此基础上制定了一个 C 语言标准，于一九八三年发表。通常称之为 ANSI C。

1.2 当代最优秀的程序设计语言

早期的 C 语言主要是用于 UNIX 系统。由于 C 语言的强大功能和各方面的优点逐渐为人们认识，到了八十年代，C 开始进入其它操作系统，并很快在各类大、中、小和微型计算机上得到了广泛的使用，成为当代最优秀的程序设计语言之一。

1.3 C 语言版本

目前最流行的 C 语言有以下几种：

- Microsoft C 或称 MS C
- Borland Turbo C 或称 Turbo C
- AT&T C

这些 C 语言版本不仅实现了 ANSI C 标准，而且在此基础上各自作了一些扩充，使之更加方便、完美。

1.4 C 语言的特点

1 • C 语言简洁、紧凑，使用方便、灵活。ANSI C 一共只有 32 个关键字：

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

9 种控制语句，程序书写自由，主要用小写字母表示，压缩了一切不必要的成分。

Turbo C 扩充了 11 个关键字：

asm _cs _ds _es _ss cdecl far

hugeinterrupt near pascal

注意：在 C 语言中，关键字都是小写的。

2• 运算符丰富。共有 34 种。C 把括号、赋值、逗号等都作为运算符处理。从而使 C 的运算类型极为丰富，可以实现其他高级语言难以实现的运算。

3• 数据结构类型丰富。

4• 具有结构化的控制语句。

5• 语法限制不太严格，程序设计自由度大。

6• C 语言允许直接访问物理地址，能进行位（bit）操作，能实现汇编语言的大部分功能，可以直接对硬件进行操作。因此有人把它称为中级语言。

7• 生成目标代码质量高，程序执行效率高。

8• 与汇编语言相比，用 C 语言写的程序可移植性好。

但是，C 语言对程序员要求也高，程序员用 C 写程序会感到限制少、灵活性大，功能强，但较其他高级语言在学习上要困难一些。

1.5 面向对象的程序设计语言

在 C 的基础上，一九八三年又由贝尔实验室的 Bjarne Stroustrup 推出了 C++。C++ 进一步扩充和完善了 C 语言，成为一种面向对象的程序设计语言。C++ 目前流行的最新版本是 Borland C++, Symantec C++ 和 Microsoft VisualC++。

C++ 提出了一些更为深入的概念，它所支持的这些面向对象的概念容易将问题空间直接地映射到程序空间，为程序员提供了一种与传统结构程序设计不同的思维方式和编程方法。因而也增加了整个语言的复杂性，掌握起来有一定难度。

1.6 C 和 C++

但是，C 是 C++ 的基础，C++ 语言和 C 语言在很多方面是兼容的。因此，掌握了 C 语言，再进一步学习 C++ 就能以一种熟悉的语法来学习面向对象的语言，从而达到事半功倍的目的。

1.7 简单的 C 程序介绍

为了说明 C 语言源程序结构的特点，先看以下几个程序。这几个程序由简到难，表现了 C 语言源程序在组成结构上的特点。虽然有关内容还未介绍，但可从这些例子中了解到组成一个 C 源程序的基本部分和书写格式。

【例 1.1】

```
main()
{
    printf("世界，您好！\n");
}
```

- main 是主函数的函数名，表示这是一个主函数。
- 每一个 C 源程序都必须有，且只能有一个主函数(main 函数)。
- 函数调用语句，printf 函数的功能是把要输出的内容送到显示器去显示。

- `printf` 函数是一个由系统定义的标准函数，可在程序中直接调用。

【例 1.2】

```
#include<math.h>
#include<stdio.h>
main()
{
    double x,s;
    printf("input number:\n");
    scanf("%lf",&x);
    s=sin(x);
    printf("sine of %lf is %lf\n",x,s);
}
```

- `include` 称为文件包含命令
- 扩展名为.h 的文件称为头文件
- 定义两个实数变量，以被后面程序使用
- 显示提示信息
- 从键盘获得一个实数 `x`
- 求 `x` 的正弦,并把它赋给变量 `s`
- 显示程序运算结果
- `main` 函数结束

程序的功能是从键盘输入一个数 `x`，求 `x` 的正弦值，然后输出结果。在 `main()` 之前的两行称为预处理命令(详见后面)。预处理命令还有其它几种，这里的 `include` 称为文件包含命令，其意义是把尖括号<>或引号""内指定的文件包含到本程序来，成为本程序的一部分。被包含的文件通常是由系统提供的，其扩展名为.h。因此也称为头文件或首部文件。C 语言的头文件中包括了各个标准库函数的函数原型。因此，凡是在程序中调用一个库函数时，都必须包含该函数原型所在的头文件。在本例中，使用了三个库函数：输入函数 `scanf`，正弦函数 `sin`,输出函数 `printf`。`sin` 函数是数学函数，其头文件为 `math.h` 文件，因此在程序的主函数前用 `include` 命令包含了 `math.h`。`scanf` 和 `printf` 是标准输入输出函数，其头文件为 `stdio.h`，在主函数前也用 `include` 命令包含了 `stdio.h` 文件。

需要说明的是，C 语言规定对 `scanf` 和 `printf` 这两个函数可以省去对其头文件的包含命令。所以在本例中也可以删去第二行的包含命令 `#include<stdio.h>`。

同样，在例 1.1 中使用了 `printf` 函数，也省略了包含命令。

在例题中的主函数体中又分为两部分，一部分为说明部分，另一部为分执行部分。说明是指变量的类型说明。例题 1.1 中未使用任何变量，因此无说明部分。C 语言规定，源程序中所有用到的变量都必须先说明，后使用，否则将会出错。这一点是编译型高级程序设计语言的一个特点，与解释型的 BASIC 语言是不同的。说明部分是 C 源程序结构中很重要的组成部分。本例中使用了两个变量 `x`，`s`，用来表示输入的自变量和 `sin` 函数值。由于 `sin` 函数要求这两个量必须是双精度浮点型，故用类型说明符 `double` 来说明这两个变量。说明部分后的四行为执行部分或称为执行语句部分，用以完成程序的功能。执行部分的第一行是输出语句，调用 `printf` 函数在显示器上输出提示字符串，请操作人员输入自变量 `x` 的值。第二行为输入语句，调用 `scanf` 函数，接受键盘上输入的数并存入变量 `x` 中。第三行是调用 `sin` 函数并把函数值送到变量 `s` 中。第四行是用 `printf` 函数输出变量 `s` 的值，即 `x` 的正弦值。程序结束。

运行本程序时，首先在显示器屏幕上给出提示串 `input number`，这是由执行部分的第一

行完成的。用户在提示下从键盘上键入某一数，如 5，按下回车键，接着在屏幕上给出计算结果。

1.8 输入和输出函数

在前两个例子中用到了输入和输出函数 `scanf` 和 `printf`，在以后要详细介绍。这里我们先简单介绍一下它们的格式，以便下面使用。

`scanf` 和 `printf` 这两个函数分别称为格式输入函数和格式输出函数。其意义是按指定的格式输入输出值。因此，这两个函数在括号中的参数表都由以下两部分组成：

“格式控制串”，参数表

格式控制串是一个字符串，必须用双引号括起来，它表示了输入输出量的数据类型。各种类型的格式表示法可参阅第三章。在 `printf` 函数中还可以在格式控制串内出现非格式控制字符，这时在显示屏幕上将原文照印。参数表中给出了输入或输出的量。当有多个量时，用逗号间隔。例如：

```
printf("sine of %lf is %lf\n",x,s);
```

其中 `%lf` 为格式字符，表示按双精度浮点数处理。它在格式串中两次出现，对应了 `x` 和 `s` 两个变量。其余字符为非格式字符则照原样输出在屏幕上。

【例 1.3】

```
int max(int a,int b);          /*函数说明*/
main()                        /*主函数*/
{
    int x,y,z;                /*变量说明*/
    int max(int a,int b);      /*函数说明*/
    printf("input two numbers:\n");
    scanf("%d%d",&x,&y);       /*输入 x,y 值*/
    z=max(x,y);               /*调用 max 函数*/
    printf("maxmum=%d",z);     /*输出*/
}
int max(int a,int b)          /*定义 max 函数*/
{
    if(a>b)return a;else return b; /*把结果返回主调函数*/
}
```

上面例中程序的功能是由用户输入两个整数，程序执行后输出其中较大的数。本程序由两个函数组成，主函数和 `max` 函数。函数之间是并列关系。可从主函数中调用其它函数。`max` 函数的功能是比较两个数，然后把较大的数返回给主函数。`max` 函数是一个用户自定义函数。因此主函数中要给出说明(程序第三行)。可见，在程序的说明部分中，不仅可以有变量说明，还可以有函数说明。关于函数的详细内容将在以后第五章介绍。在程序的每行后用 `/*` 和 `*/` 括起来的内容为注释部分，程序不执行注释部分。

上例中程序的执行过程是，首先在屏幕上显示提示串，请用户输入两个数，回车后由 `scanf` 函数语句接收这两个数送入变量 `x,y` 中，然后调用 `max` 函数，并把 `x,y` 的值传送给 `max` 函数的参数 `a,b`。在 `max` 函数中比较 `a,b` 的大小，把大者返回给主函数的变量 `z`，最后在屏幕上输出 `z` 的值。

1.9 C 源程序的结构特点

1. 一个 C 语言源程序可以由一个或多个源文件组成。
2. 每个源文件可由一个或多个函数组成。
3. 一个源程序不论由多少个文件组成，都有一个且只能有一个 `main` 函数，即主函数。
4. 源程序中可以有预处理命令(`include` 命令仅为其中的一种)，预处理命令通常应放在源文件或源程序的最前面。
5. 每一个说明，每一个语句都必须以分号结尾。但预处理命令，函数头和花括号“`{}`”之后不能加分号。
6. 标识符，关键字之间必须至少加一个空格以示间隔。若已有明显的间隔符，也可不再加空格来间隔。

1.10 书写程序时应遵循的规则

从书写清晰，便于阅读，理解，维护的角度出发，在书写程序时应遵循以下规则：

1. 一个说明或一个语句占一行。
 2. 用 `{}` 括起来的部分，通常表示了程序的某一层结构。`{}` 一般与该结构语句的第一个字母对齐，并单独占一行。
 3. 低一层次的语句或说明可比高一层次的语句或说明缩进若干格后书写。以便看起来更加清晰，增加程序的可读性。
- 在编程时应力求遵循这些规则，以养成良好的编程风格。

1.11 C 语言的字符集

字符是组成语言的最基本的元素。C 语言字符集由字母，数字，空格，标点和特殊字符组成。在字符常量，字符串常量和注释中还可以使用汉字或其它可表示的图形符号。

1. 字母

小写字母 `a~z` 共 26 个

大写字母 `A~Z` 共 26 个

2. 数字

`0~9` 共 10 个

3. 空白符

空格符、制表符、换行符等统称为空白符。空白符只在字符常量和字符串常量中起作用。在其它地方出现时，只起间隔作用，编译程序对它们忽略不计。因此在程序中使用空白符与否，对程序的编译不发生影响，但在程序中适当的地方使用空白符将增加程序的清晰性和可读性。

4. 标点和特殊字符

1.12 C 语言词汇

在 C 语言中使用的词汇分为六类：标识符，关键字，运算符，分隔符，常量，注释符等。

1. 标识符

在程序中使用的变量名、函数名、标号等统称为标识符。除库函数的函数名由系统定义外，其余都由用户自定义。C 规定，标识符只能是字母(A~Z, a~z)、数字(0~9)、下划线(_)组成的字符串，并且其第一个字符必须是字母或下划线。

以下标识符是合法的：

a, x, x3, BOOK_1, sum5

以下标识符是非法的：

3s 以数字开头

s*T 出现非法字符*

-3x 以减号开头

bowy-1 出现非法字符-(减号)

在使用标识符时还必须注意以下几点：

(1)标准 C 不限制标识符的长度，但它受各种版本的 C 语言编译系统限制，同时也受到具体机器的限制。例如在某版本 C 中规定标识符前八位有效，当两个标识符前八位相同时，则被认为是同一个标识符。

(2)在标识符中，大小写是有区别的。例如 BOOK 和 book 是两个不同的标识符。

(3)标识符虽然可由程序员随意定义，但标识符是用于标识某个量的符号。因此，命名应尽量有相应的意义，以便于阅读理解，作到“顾名思义”。

2. 关键字

关键字是由 C 语言规定的具有特定意义的字符串，通常也称为保留字。用户定义的标识符不应与关键字相同。C 语言的关键字分为以下几类：

(1) 类型说明符

用于定义、说明变量、函数或其它数据结构的类型。如前面例题中用到的 int, double 等

(2) 语句定义符

用于表示一个语句的功能。如例 1.3 中用到的 if else 就是条件语句的语句定义符。

(3) 预处理命令字

用于表示一个预处理命令。如前面各例中用到的 include。

3. 运算符

C 语言中含有相当丰富的运算符。运算符与变量，函数一起组成表达式，表示各种运算功能。运算符由一个或多个字符组成。

4. 分隔符

在 C 语言中采用的分隔符有逗号和空格两种。逗号主要用在类型说明和函数参数表中，分隔各个变量。空格多用于语句各单词之间，作间隔符。在关键字，标识符之间必须要有一个以上的空格符作间隔，否则将会出现语法错误，例如把 int a; 写成 inta; C 编译器会把 inta 当成一个标识符处理，其结果必然出错。

5. 常量

C 语言中使用的常量可分为数字常量、字符常量、字符串常量、符号常量、转义字符等多种。在后面章节中将专门给予介绍。

6. 注释符

C 语言的注释符是以 “/*” 开头并以 “*/” 结尾的串。在 “/*” 和 “*/” 之间的即为注释。程序编译时，不对注释作任何处理。注释可出现在程序中的任何位置。注释用来向用户提示或解释程序的意义。在调试程序中对暂不使用的语句也可用注释符括起来，使翻译跳过不作处理，待调试结束后再去掉注释符。

1.13 Turbo C 2.0 集成开发环境的使用

1.13.1 Turbo C 2.0 简介和启动

我们上机实习和将来考试都是使用 Borland Turbo C 2.0 这个版本。该系统是 DOS 操作系统支持下的软件，在 windows 98 环境下，可以在 DOS 窗口下运行。

我们机房是在 D 盘根目录下建立一个 TC 子目录下安装 Turbo C 2.0 系统的。TC 下还建立了两个子目录 LIB 和 INCLUDE, LIB 子目录中存放库文件, INCLUDE 子目录中存放所有头文件。

在 DOS 环境下或在 windows 98 的 DOS 窗口下运行 Turbo C 2.0 时，只要在 TC 子目录下键入 TC 并回车即可进入 Turbo C 2.0 集成开发环境。

在 windows 98 环境下，也可以选运行菜单，然后键入 d:\tc\tc 即可，也可以在 tc 文件夹找到 tc.exe 文件，然后用鼠标双击该文件名也可进入 Turbo C 2.0 集成开发环境。

Turbo C 是美国 Borland 公司的产品, Borland 公司是一家专门从事软件开发、研制的大公司。该公司相继推出了一套 Turbo 系列软件, 如 Turbo BASIC, TurboPascal, Turbo Prolog, 这些软件很受用户欢迎。该公司在 1987 年首次推出 TurboC 1.0 产品, 其中使用了全然一新的集成开发环境, 即使用了一系列下拉式菜单, 将文本编辑、程序编译、连接以及程序运行一体化, 大大方便了程序的开发。1988 年, Borland 公司又推出 Turbo C1.5 版本, 增加了图形库和文本窗口函数库等, 而 Turbo C 2.0 则是该公司 1989 年出版的。Turbo C2.0 在原来集成开发环境的基础上增加了查错功能, 并可以在 Tiny 模式下直接生成.COM (数据、代码、堆栈处在同一 64K 内存中) 文件。还可对数学协处理器 (支持 8087/80287/80387 等) 进行仿真。

Borland 公司后来又推出了面向对象的程序软件包 Turbo C++, 它继承发展 Turbo C 2.0 的集成开发环境, 并包含了面向对象的基本思想和设计方法。1991 年为了适用 Microsoft 公司的 Windows 3.0 版本, Borland 公司又将 Turbo C++ 作了更新, 即 Turbo C 的新一代产品 Borlandc C++也已经问世了。

1.13.2 Turbo C 2.0 集成开发环境

进入 Turbo C 2.0 集成开发环境中后, 屏幕上显示:



其中顶上一行为 Turbo C 2.0 主菜单，中间窗口为编辑区，接下来是信息窗口，最底下一行为参考行。这四个窗口构成了 Turbo C 2.0 的主屏幕，以后的编程、编译、调试以及运行都将在这个主屏幕中进行。

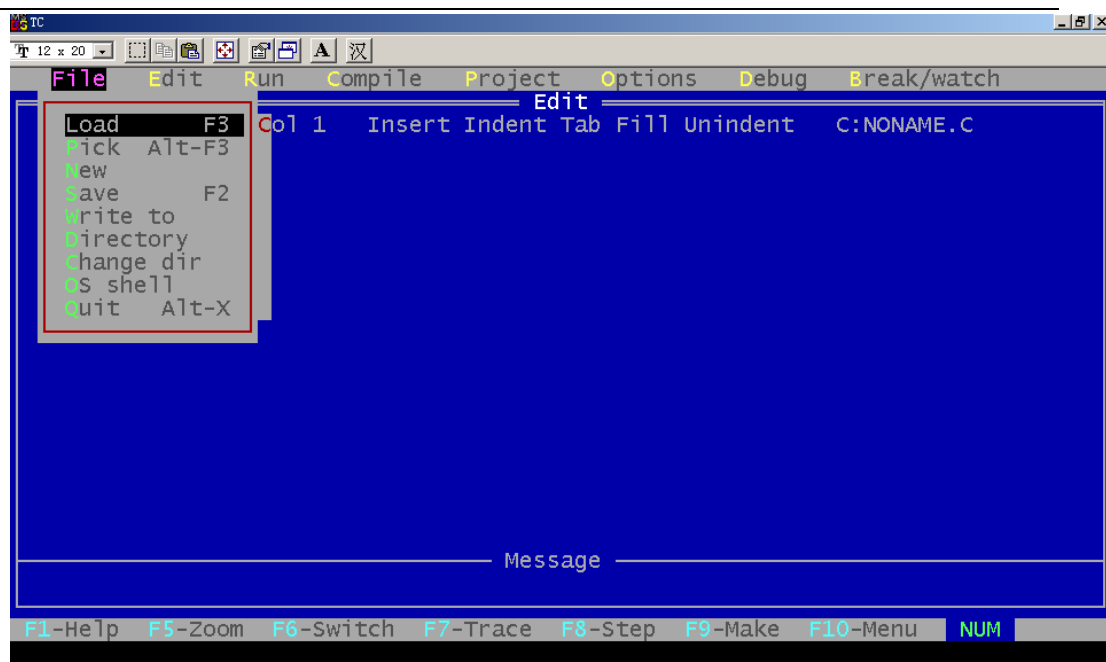
主菜单在 Turbo C 2.0 主屏幕顶上一行，显示下列内容：

File Edit Run Compile Project Options Debug Break/watch

除 Edit 外，其它各项均有子菜单，只要用 Alt 加上某项中第一个字母，就可进入该项的子菜单中。

1.13.3 File 菜单

按 Alt+F 可进入 File 菜单，如图：



File 菜单的子菜单共有 9 项，分别叙述如下：

1. Load: 装入一个文件，可用类似 DOS 的通配符(如*.C)来进行列表选择。也可装入其它

扩展名的文件, 只要给出文件名(或只给路径)即可。该项的热键为 F3, 即只要按 F3 即可进入该项, 而不需要先进入 File 菜单再选此项。

2. **Pick:** 将最近装入编辑窗口的 8 个文件列成一个表让用户选择, 选择后将该程序装入编辑区, 并将光标置在上次修改过的地方。其热键为 Alt-F3。
3. **New:** 新建文件, 缺省文件名为 NONAME.C, 存盘时可改名。
4. **Save:** 将编辑区中的文件存盘, 若文件名是 NONAME.C 时, 将询问是否更改文件名, 其热键为 F2。
5. **Write to:** 可由用户给出文件名将编辑区中的文件存盘, 若该文件已存在, 则询问要不要覆盖。
6. **Directory:** 显示目录及目录中的文件, 并可由用户选择。
7. **Change dir:** 显示当前默认目录, 用户可以改变默认目录。
8. **Os shell:** 暂时退出 Turbo C 2.0 到 DOS 提示符下, 此时可以运行 DOS 命令, 若想回到 Turbo C 2.0 中, 只要在 DOS 状态下键入 EXIT 即可。
9. **Quit:** 退出 Turbo C 2.0, 返回到 DOS 操作系统中, 其热键为 Alt+X。

说明:

以上各项可用光标键移动色棒进行选择, 回车则执行。也可用每一项的第一个大写字母直接选择。若要退到主菜单或从它的下一级菜单列表框退回均可用 Esc 键, Turbo C 2.0 所有菜单均采用这种方法进行操作, 以下不再说明。

1.13.4 Edit 菜单

按 Alt+E 可进入编辑菜单, 若再回车, 则光标出现在编辑窗口, 此时用户可以 进行文本编辑。编辑方法基本与 wordstar 相同, 可用 F1 键获得有关编辑方法的帮助信息。

1. 与编辑有关的功能键如下:

F1 获得 Turbo C 2.0 编辑命令的帮助信息;
F5 扩大编辑窗口到整个屏幕;
F6 在编辑窗口与信息窗口之间进行切换;
F10 从编辑窗口转到主菜单。

2. 编辑命令简介:

PageUp 向前翻页
PageDn 向后翻页
Home 将光标移到所在行的开始
End 将光标移到所在行的结尾
Ctrl+Y 删除光标所在的一行
Ctrl+T 删除光标所在处的一个词
Ctrl+KB 设置块开始
Ctrl+KK 设置块结尾
Ctrl+KV 块移动
Ctrl+KC 块拷贝
Ctrl+KY 块删除
Ctrl+KR 读文件
Ctrl+KW 存文件
Ctrl+KP 块文件打印
Ctrl+F1 如果光标所在处为 Turbo C 2.0 库函数, 则获得有关该函数的帮助信息

Ctrl+Q[查找 Turbo C 2.0 双界符的后匹配符

Ctrl+Q] 查找 Turbo C 2.0 双界符的前匹配符

说明:

- 1) Turbo C 2.0 的双界符包括以下几种符号:
 - a) 花括号 {和}
 - b) 尖括号 <和>
 - c) 圆括号 (和)
 - d) 方括号 [和]
 - e) 注释符 /*和*/
 - f) 双引号 "
 - g) 单引号 '
- 2) Turbo C 2.0 在编辑文件时还有一种功能, 就是能够自动缩进, 即光标定位和上一个非空字符对齐。在编辑窗口中, Ctrl+OL 为自动缩进开关的控制键。

1.13.5 Run 菜单

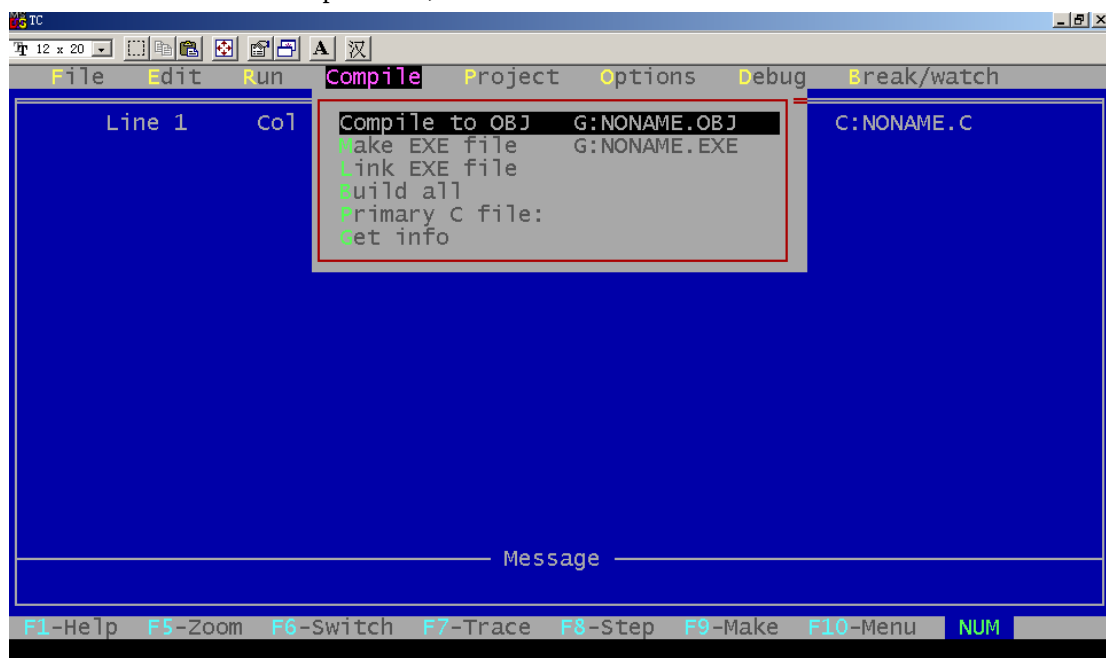
按 Alt+R 可进入 Run 菜单, 该菜单有以下各项, 如图所示:



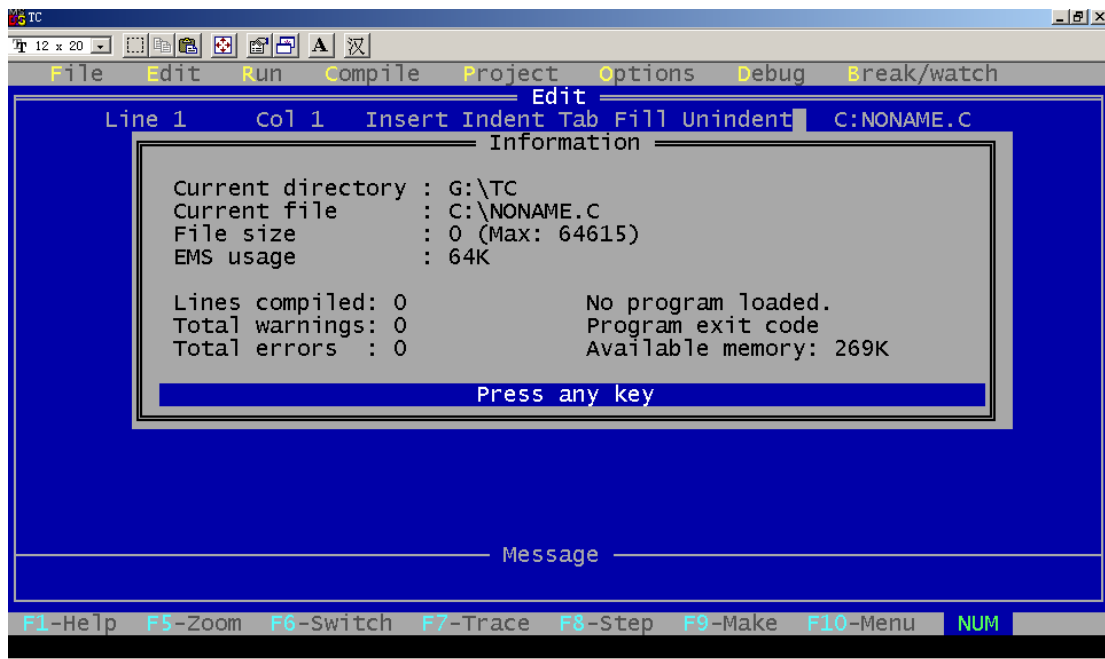
1. Run: 运行由 Project/Project name 项指定的文件名或当前编辑区的文件。如果对上次编译后的源代码未做过修改, 则直接运行到下一个断点(没有断点则运行到结束)。否则先进行编译、连接后才运行, 其热键为 Ctrl+F9。
2. Program reset: 中止当前的调试, 释放分给程序的空间, 其热键为 Ctrl+F2。
3. Go to cursor: :调试程序时使用, 选择该项可使程序运行到光标所在行。光标所在行必须为一条可执行语句, 否则提示错误。其热键为 F4。
4. Trace into: 在执行一条调用其它用户定义的子函数时, 若用 Trace into 项, 则执行长条将跟踪到该子函数内部去执行, 其热键为 F7。
5. Step over: 执行当前函数的下一条语句, 即使用户函数调用, 执行长条也不会跟踪进函数内部, 其热键为 F8。
6. User screen: 显示程序运行时在屏幕上显示的结果。其热键为 Alt+F5。

1.13.6 Compile 菜单

按 Alt+C 可进入 Compile 菜单, 该菜单有以下几个内容, 如图所示:

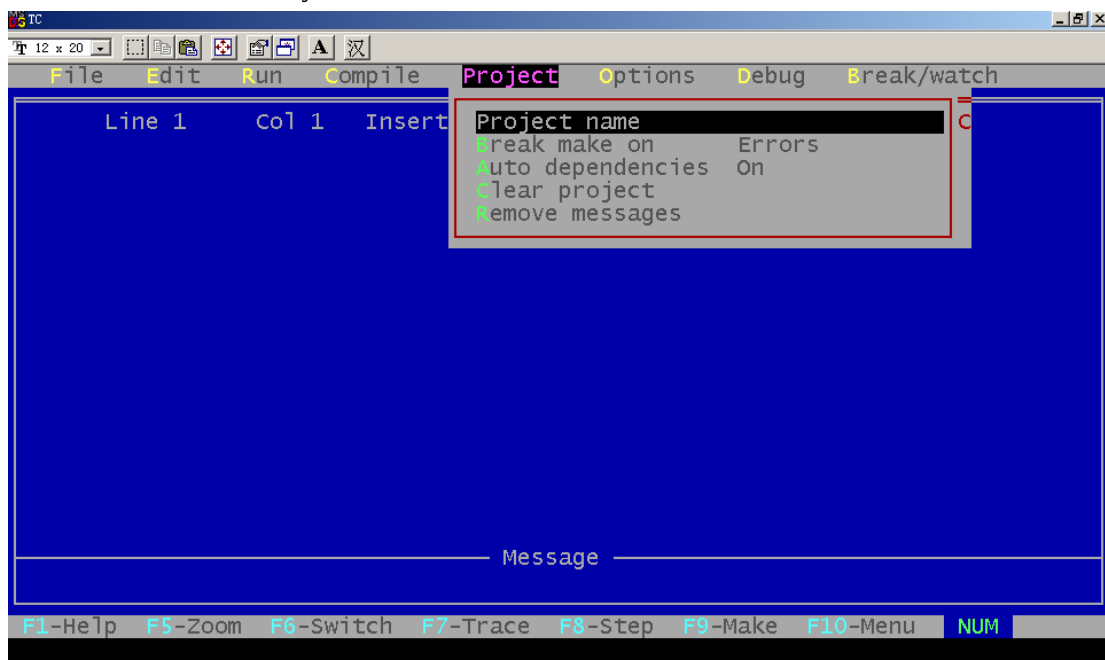


1. **Compile to OBJ:** 将一个 C 源文件编译生成.OBJ 目标文件, 同时显示生成的文件名。其热键为 Alt+F9。
2. **Make EXE file:** 此命令生成一个.EXE 的文件, 并显示生成的.EXE 文件名。其中.EXE 文件名是下面几项之一:
 - 1) 由 Project/Project name 说明的项目文件名。
 - 2) 若没有项目文件名, 则由 Primary C file 说明的源文件。
 - 3) 若以上两项都没有文件名, 则为当前窗口的文件名。
3. **Link EXE file:** 把当前.OBJ 文件及库文件连接在一起生成.EXE 文件。
4. **Build all:** 重新编译项目里的所有文件, 并进行装配生成.EXE 文件。该命令不作过时检查 (上面的几条命令要作过时检查, 即如果目前项目里源文件的日期和时间与目标文件相同或更早, 则拒绝对源文件进行编译)。
5. **Primary C file:** 当在该项中指定了主文件后, 在以后的编译中, 如没有项目文件名则编译此项中规定的主 C 文件, 如果编译中有错误, 则将此文件调入编辑窗口, 不管目前窗口中是不是主 C 文件。
6. **Get info:** 获得有关当前路径、源文件名、源文件字节大小、编译中的错误数目、可用空间等信息, 如图:



1.13.7 Project 菜单

按 Alt+P 可进入 Project 菜单, 该菜单包括以下内容, 如图所示:



1. Project name: 项目名具有 .PRJ 的扩展名, 其中包括将要编译、连接的文件名。例如有一个程序由 file1.c, file2.c, file3.c 组成, 要将这 3 个文件编译装配成一个 file.exe 的执行文件, 可以先建立一个 file.prj 的项目文件, 其内容如下:

```
file1.c
file2.c
file3.c
```

此时将 file.prj 放入 Project name 项中, 以后进行编译时将自动对项目文件中规定的三个源文件分别进行编译。然后连接成 file.exe 文件。如果其中有些文件已经编译成 .OBJ 文

件，而又没有修改过，可直接写上.OBJ 扩展名。此时将不再编译而只进行连接。

例如：

```
file1.obj  
file2.c  
file3.c
```

将不对 file1.c 进行编译，而直接连接。

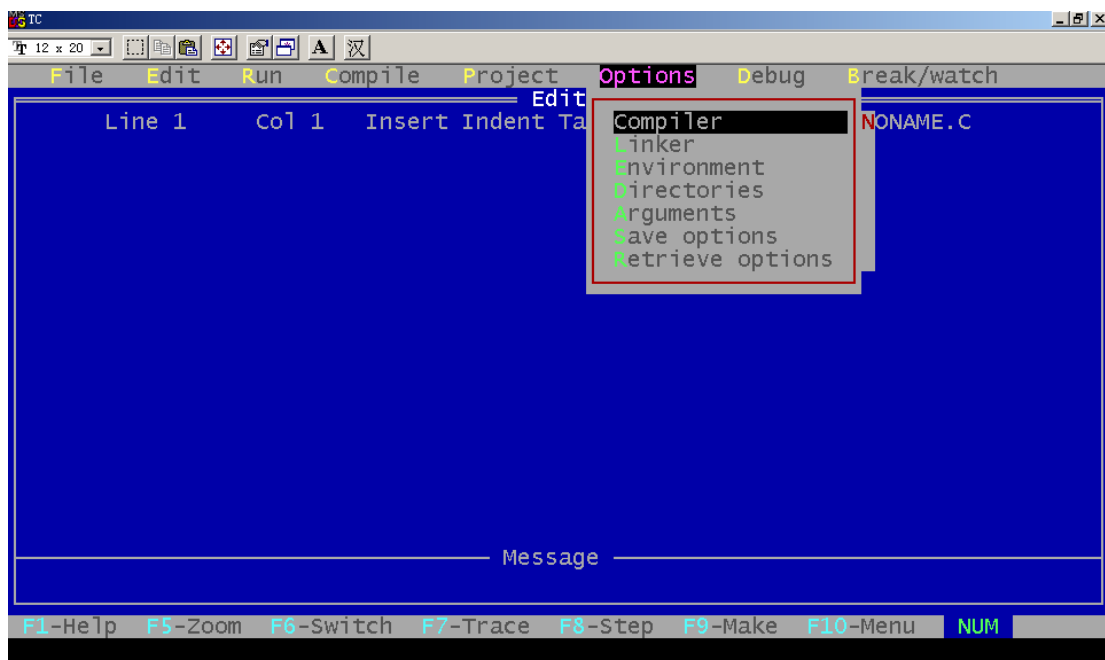
说明：

当项目文件中的每个文件无扩展名时，均按源文件对待，另外，其中的文件也可以是库文件，但必须写上扩展名.LIB。

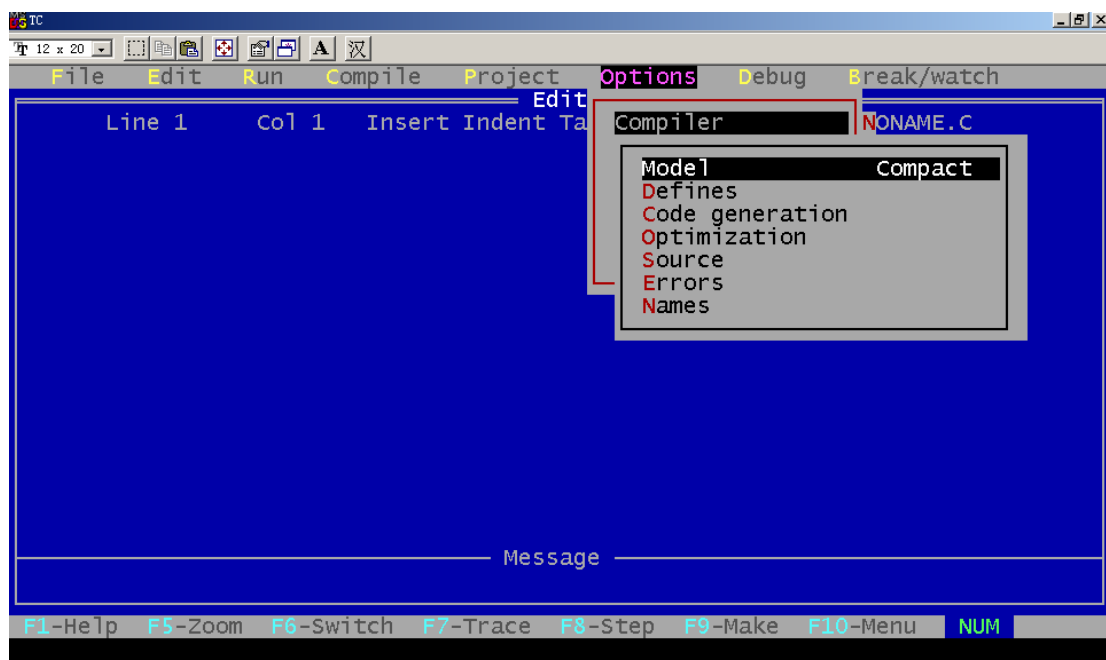
2. Break make on: 由用户选择是否有 Warning、Errors、Fatal Errors 时或 Link 之前退出 Make 编译。
3. Auto dependencies: 当开关置为 on，编译时将检查源文件与对应的.OBJ 文件日期和时间，否则不进行检查。
4. Clear project: 清除 Project/Project name 中的项目文件名。
5. Remove messages: 把错误信息从信息窗口中清除掉。

1.13.8 Options 菜单

按 Alt+O 可进入 Options 菜单，该菜单对初学者来说要谨慎使用，该菜单有以下几个内容，如图所示：



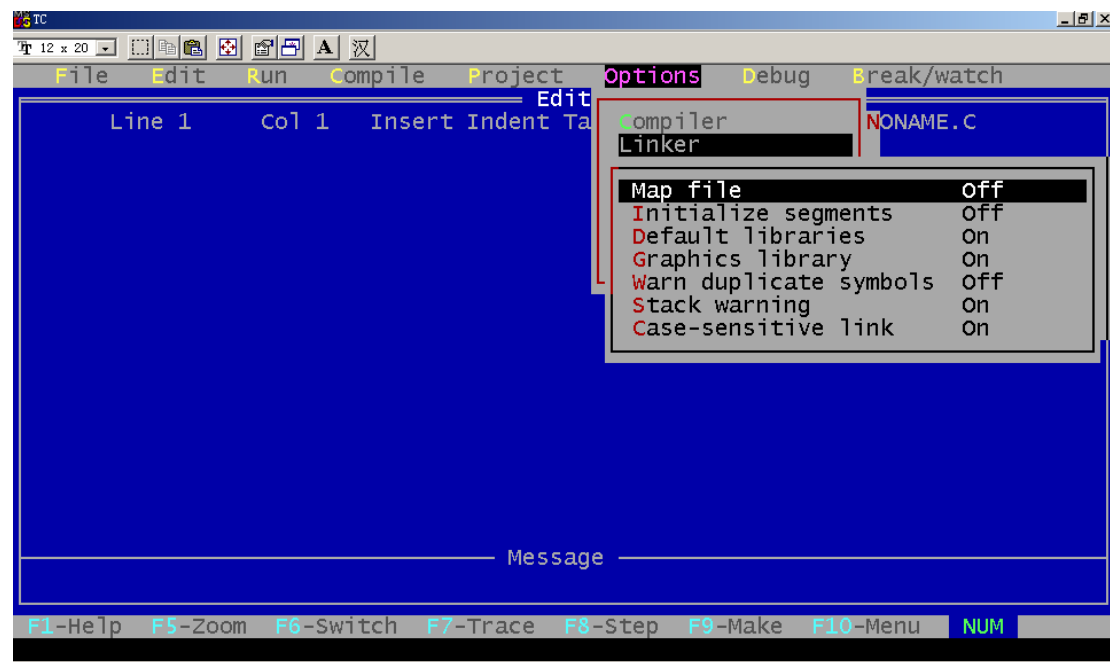
1. Compiler: 本项选择又有许多子菜单，可以让用户选择硬件配置、存储模型、调试技术、代码优化、对话信息控制和宏定义。这些子菜单如图所示：



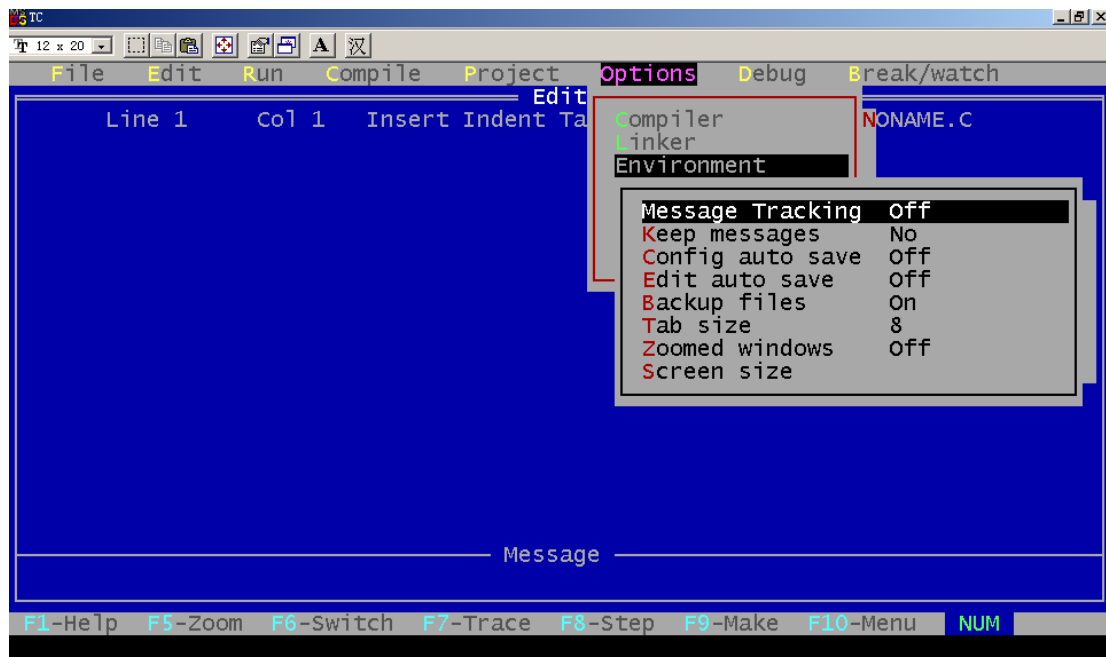
- 1) Model: 共有 Tiny, small, medium, compact, large, huge 六种不同模式可由用户选择。
- 2) Define: 打开一个宏定义框, 用户可输入宏定义。多重定义可用分号, 赋值可用等号。
- 3) Code generation: 它又有许多任选项, 这些任选项告诉编译器产生什么样的目标代码。
 - ✧ Calling convention 可选择 C 或 Pascal 方式传递参数。
 - ✧ Instruction set 可选择 8088/8086 或 80186/80286 指令系列。
 - ✧ Floating point 可选择仿真浮点、数学协处理器浮点或无浮点运算。
 - ✧ Default char type 规定 char 的类型。
 - ✧ Alignment 规定地址对准原则。
 - ✧ Merge duplicate strings 作优化用, 将重复的字符串合并在一起。
 - ✧ Standard stack frame 产生一个标准的栈结构。
 - ✧ Test stack overflow 产生一段程序运行时检测堆栈溢出的代码。
 - ✧ Line number 在 .OBJ 文件中放进行号以供调试时用。
 - ✧ OBJ debug information 在 .OBJ 文件中产生调试信息。
- 4) Optimization: 它又有许多任选项。
 - ✧ Optimize for 选择是对程序小型化还是对程序速度进行优化处理。
 - ✧ Use register variable 用来选择是否允许使用寄存器变量。
 - ✧ Register optimization 尽可能使用寄存器变量以减少过多的取数操作。
 - ✧ Jump optimization 通过去除多余的跳转和调整循环与开关语句的办法, 压缩代码。
- 5) Source: 它又有许多任选项。
 - ✧ Identifier length 说明标识符有效字符的个数, 默认为 32 个。
 - ✧ Nested comments 是否允许嵌套注释。
 - ✧ ANSI keywords only 是只允许 ANSI 关键字还是也允许 Turbo C2.0 关键字。
- 6) Error
 - ✧ Error stop after 多少个错误时停止编译, 默认为 25 个。
 - ✧ Warning stop after 多少个警告错误时停止编译, 默认为 100 个。

- ✧ Display warning
 - ✧ Portability warning 移植性警告错误。
 - ✧ ANSI Violations 侵犯了 ANSI 关键字的警告错误。
 - ✧ Common error 常见的警告错误。
 - ✧ Less common error 少见的警告错误。
- 7) Names : 用于改变段(segment)、组(group) 和类(class)的名字,默认值为 CODE, DATA, BSS。

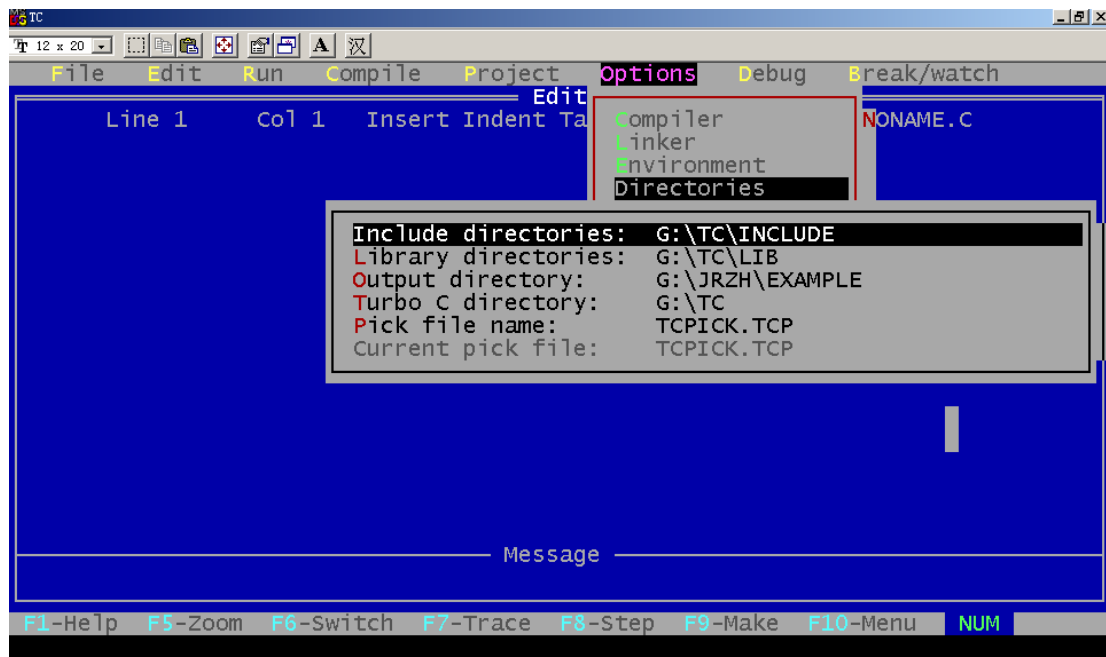
2. Linker: 本菜单设置有关连接的选择项, 它有以下内容, 如图所示:



- 1) Map file menu 选择是否产生.MAP 文件。
 - 2) Initialize segments 是否在连接时初始化没有初始化的段。
 - 3) Devault libraries 是否在连接其它编译程序产生的目标文件时去寻找其缺省库。
 - 4) Graphics library 是否连接 graphics 库中的函数。
 - 5) Warn duplicate symbols 当有重复符号时产生警告信息。
 - 6) Stack warinig 是否让连接程序产生 No stack 的警告信息。
 - 7) Case-sensitive link 是否区分大、小写字。
3. Environment: 菜单规定是否对某些文件自动存盘及制表键和屏幕大小的设置, 它有以下内容, 如图所示:



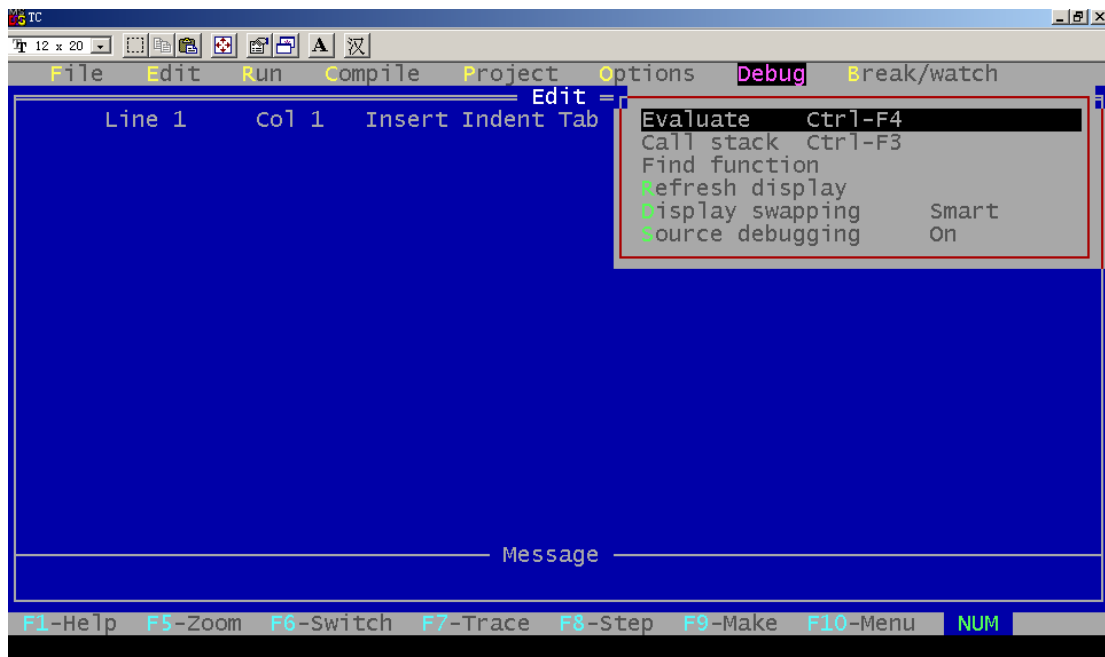
- 1) Message tracking:
 - ✧ Current file 跟踪在编辑窗口中的文件错误。
 - ✧ All files 跟踪所有文件错误。
 - ✧ Off 不跟踪。
 - 2) Keep message: 编译前是否清除 Message 窗口中的信息。
 - 3) Config auto save: 选 on 时, 在 Run, Shell 或退出集成开发环境之前, 如果 Turbo C 2.0 的配置被改过, 则所做的改动将存入配置文件中。选 off 时不存。
 - 4) Edit auto save: 是否在 Run 或 Shell 之前, 自动存储编辑的源文件。
 - 5) Backup file: 是否在源文件存盘时产生后备文件(.BAK 文件)。
 - 6) Tab size: 设置制表键大小, 默认为 8。
 - 7) Zoomed windows: 将现行活动窗口放大到整个屏幕, 其热键为 F5。
 - 8) Screen size 设置屏幕文本大小。
4. Directories: 规定编译、连接所需文件的路径, 有下列各项, 如图所示:



- (1) Include directories: 包含文件的路径, 多个子目录用";"分开。
 - (2) Library directories: 库文件路径, 多个子目录用";"分开。
 - (3) Output directoried: 输出文件(.OBJ, .EXE, .MAP 文件)的目录。
 - (4) Turbo C directoried: Turbo C 所在的目录。
 - (5) Pick file name: 定义加载的 pick 文件名, 如不定义则从 currentpick file 中取。
5. Arguments: 允许用户使用命令行参数。
 6. Save options: 保存所有选择的编译、连接、调试和项目到配置文件中, 缺省的配置文件为 TCCONFIG.TC。
 7. Retrive options 装入一个配置文件到 TC 中, TC 将使用该文件的选择项。

1.13.9 Debug 菜单

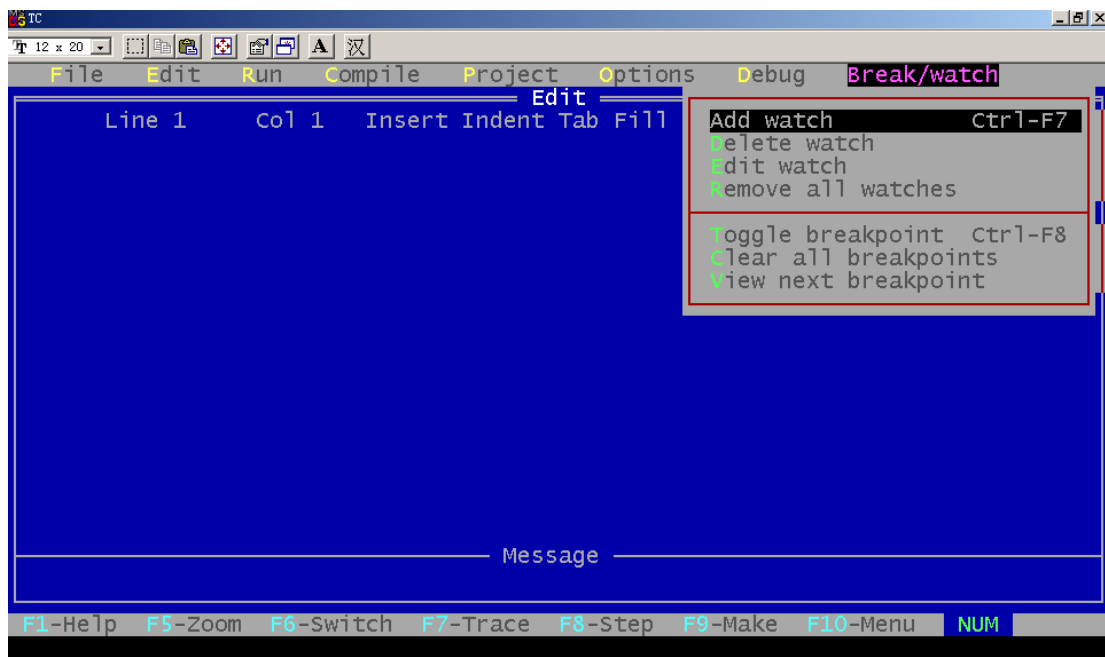
按 Alt+D 可选择 Debug 菜单, 该菜单主要用于查错, 它包括以下内容, 如图所示:



1. Evaluate
 - 1) Expression 要计算结果的表达式。
 - 2) Result 显示表达式的计算结果。
 - 3) New value 赋给新值。
2. Call stack: 该项不可接触。而在 Turbo C debugger 时用于检查堆栈情况。
3. Find function 在运行 Turbo C debugger 时用于显示规定的函数。
4. Refresh display 如果编辑窗口偶然被用户窗口重写了可用此恢复编辑窗口的内容。

1.13.10 Break/watch 菜单

按 Alt+B 可进入 Break/watch 菜单, 该菜单有以下内容, 如图所示:



1. Add watch: 向监视窗口插入一监视表达式。

2. Delete watch: 从监视窗口中删除当前的监视表达式。
3. Edit watch: 在监视窗口中编辑一个监视表达式。
4. Remove all : watches 从监视窗口中删除所有的监视表达式。
5. Toggle breakpoint: 对光标所在的行设置或清除断点。
6. Clear all breakpoints: 清除所有断点。
7. View next breakpoint: 将光标移动到下一个断点处。

1.13.11 Turbo C 2.0 的配置文件

所谓配置文件是包含 Turbo C 2.0 有关信息的文件, 其中存有编译、连接的选择和路径等信息。可以用下述方法建立 Turbo C 2.0 的配置:

1. 建立用户自命名的配置文件: 可以从 Options 菜单中选择 Options/Save options 命令, 将当前集成开发环境的所有配置存入一个由用户命名的配置文件中。下次启动 TC 时只要在 DOS 下键入:
`tc/c<用户命名的配置文件名>`
就会按这个配置文件中的内容作为 Turbo C 2.0 的选择。
2. 若设置 Options/Environment/Config auto save 为 on, 则退出集成开发环境时, 当前的设置会自动存放到 Turbo C 2.0 配置文件 TCCONFIG.TC 中。Turbo C 在启动时会自动寻找这个配置文件。
3. 用 TCINST 设置 Turbo C 的有关配置, 并将结果存入 TC.EXE 中。Turbo C 在启动时, 若没有找到配置文件, 则取 TC.EXE 中的缺省值。

2 程序的灵魂—算法

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（data structure）。
- 对操作的描述。即操作步骤，也就是算法（algorithm）。

Nikiklaus Wirth 提出的公式：

数据结构+算法=程序

教材认为：

程序=算法+数据结构+程序设计方法+语言工具和环境

这 4 个方面是一个程序涉及人员所应具备的知识。

本课程的目的是使同学知道怎样编写一个 C 程序，进行编写程序的初步训练，因此，只介绍算法的初步知识。

2.1 算法的概念

做任何事情都有一定的步骤。为解决一个问题而采取的方法和步骤，就称为算法。

- 计算机算法：计算机能够执行的算法。
- 计算机算法可分为两大类：
 - 数值运算算法：求解数值；
 - 非数值运算算法：事务管理领域。

2.2 简单算法举例

【例 2.1】求 $1 \times 2 \times 3 \times 4 \times 5$ 。

最原始方法：

步骤 1：先求 1×2 ，得到结果 2。

步骤 2：将步骤 1 得到的乘积 2 乘以 3，得到结果 6。

步骤 3：将 6 再乘以 4，得 24。

步骤 4：将 24 再乘以 5，得 120。

这样的算法虽然正确，但太繁。

改进的算法：

S1: 使 $t=1$

S2: 使 $i=2$

S3: 使 $t \times i$ ，乘积仍然放在在变量 t 中，可表示为 $t \times i \rightarrow t$

S4: 使 i 的值+1，即 $i+1 \rightarrow i$

S5: 如果 $i \leq 5$, 返回重新执行步骤 S3 以及其后的 S4 和 S5; 否则, 算法结束。

如果计算 $100!$ 只需将 S5: 若 $i \leq 5$ 改成 $i \leq 100$ 即可。

如果该求 $1 \times 3 \times 5 \times 7 \times 9 \times 11$, 算法也只需做很少的改动:

S1: $1 \rightarrow t$

S2: $3 \rightarrow i$

S3: $t \times i \rightarrow t$

S4: $i+2 \rightarrow i$

S5: 若 $i \leq 11$, 返回 S3, 否则, 结束。

该算法不仅正确, 而且是计算机较好的算法, 因为计算机是高速运算的自动机器, 实现循环轻而易举。

思考: 若将 S5 写成: S5: 若 $i < 11$, 返回 S3; 否则, 结束。

【例 2.2】有 50 个学生, 要求将他们之中成绩在 80 分以上者打印出来。

如果, n 表示学生学号, n_i 表示第 i 个学生学号; g 表示学生成绩, g_i 表示第 i 个学生成绩; 则算法可表示如下:

S1: $1 \rightarrow i$

S2: 如果 $g_i \geq 80$, 则打印 n_i 和 g_i , 否则不打印

S3: $i+1 \rightarrow i$

S4: 若 $i \leq 50$, 返回 S2, 否则, 结束。

【例 2.3】判定 2000 — 2500 年中的每一年是否闰年, 将结果输出。

闰年的条件:

- 1) 能被 4 整除, 但不能被 100 整除的年份;
- 2) 能被 100 整除, 又能被 400 整除的年份;

设 y 为被检测的年份, 则算法可表示如下:

S1: $2000 \rightarrow y$

S2: 若 y 不能被 4 整除, 则输出 y “不是闰年”, 然后转到 S6

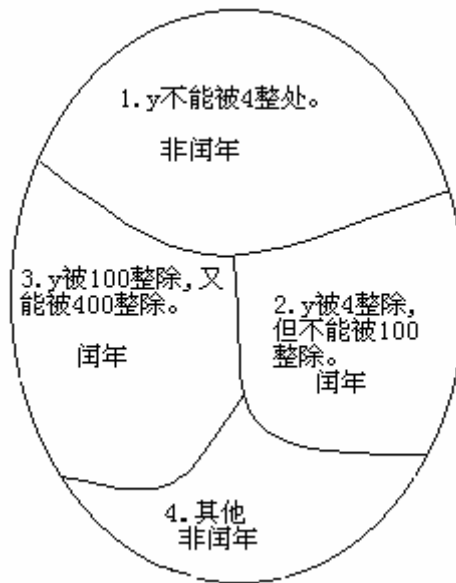
S3: 若 y 能被 4 整除, 不能被 100 整除, 则输出 y “是闰年”, 然后转到 S6

S4: 若 y 能被 100 整除, 又能被 400 整除, 输出 y “是闰年” 否则输出 y “不是闰年”, 然后转到 S6

S5: 输出 y “不是闰年”。

S6: $y+1 \rightarrow y$

S7: 当 $y \leq 2500$ 时, 返回 S2 继续执行, 否则, 结束。



【例 2.4】求 $1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots + \frac{1}{99} - \frac{1}{100}$ 。

算法可表示如下：

S1: sigh=1

S2: sum=1

S3: deno=2

S4: sigh=(-1)×sigh

S5: term= sigh×(1/deno)

S6: term=sum+term

S7: deno= deno +1

S8:若 deno≤100, 返回 S4; 否则, 结束。

【例 2.5】对一个大于或等于 3 的正整数, 判断它是不是一个素数。

算法可表示如下：

S1: 输入 n 的值

S2: i=2

S3: n 被 i 除, 得余数 r

S4:如果 r=0, 表示 n 能被 i 整除, 则打印 n “不是素数”, 算法结束; 否则执行 S5

S5: i+1→i

S6:如果 i≤n-1, 返回 S3; 否则打印 n “是素数”; 然后算法结束。

改进：

S6:如果 $i \leq \sqrt{n}$, 返回 S3; 否则打印 n “是素数”; 然后算法结束。

2.3 算法的特性

- 有穷性：一个算法应包含有限的操作步骤而不能是无限的。
- 确定性：算法中每一个步骤应当是确定的，而不能应当是含糊的、模棱两可的。
- 有零个或多个输入。
- 有一个或多个输出。
- 有效性：算法中每一个步骤应当能有效地执行，并得到确定的结果。

对于程序设计人员，必须会设计算法，并根据算法写出程序。

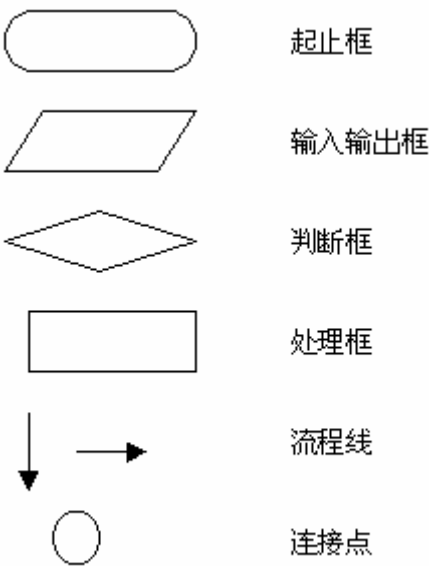
2.4 怎样表示一个算法

2.4.1 用自然语言表示算法

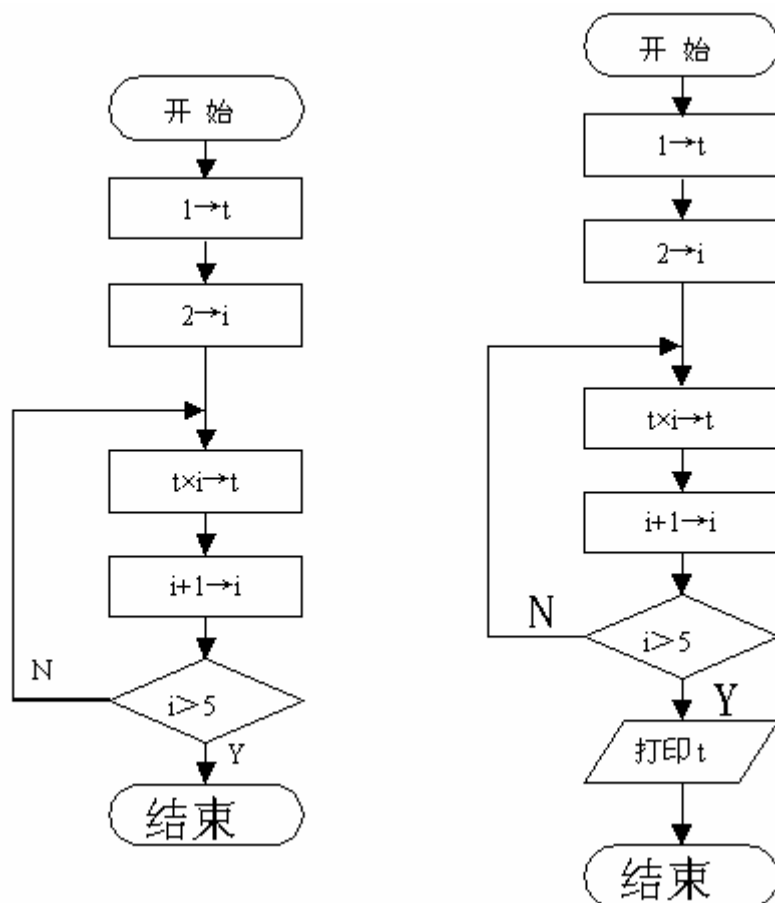
除了很简单的问题，一般不用自然语言表示算法。

2.4.2 用流程图表示算法

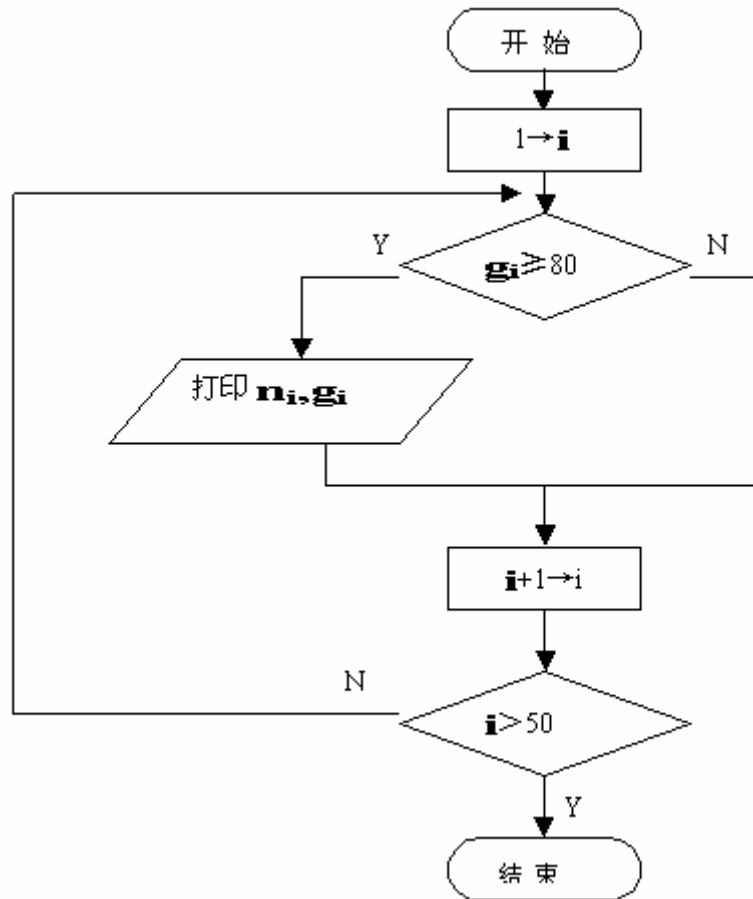
流程图表示算法，直观形象，易于理解。



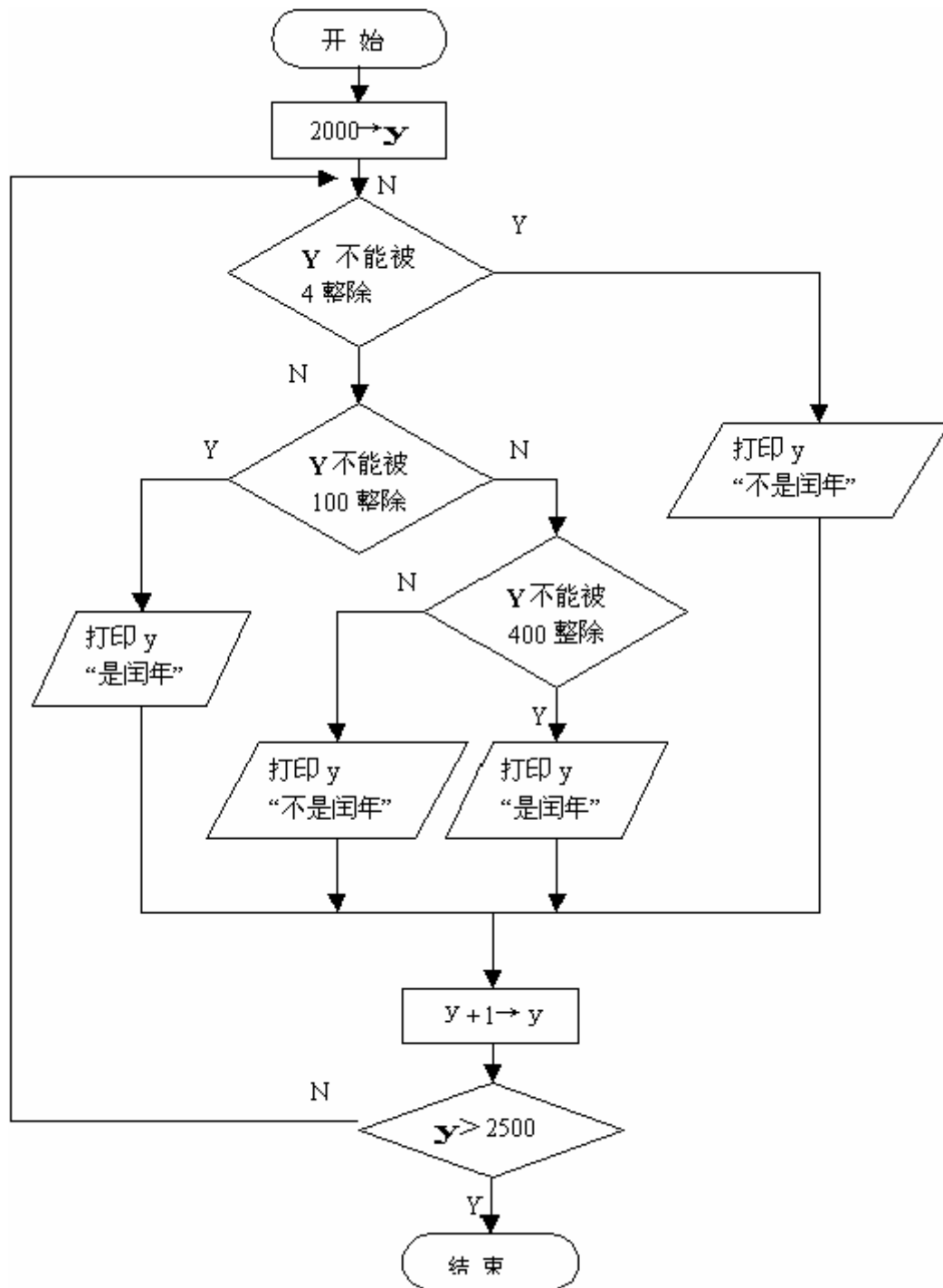
【例 2.6】将例 2.1 求 5!的算用流程图表示。



【例 2.7】将例 2.2 的算用流程图表示。



【例 2.8】将例 2.3 判定闰年的算用流程图表示。



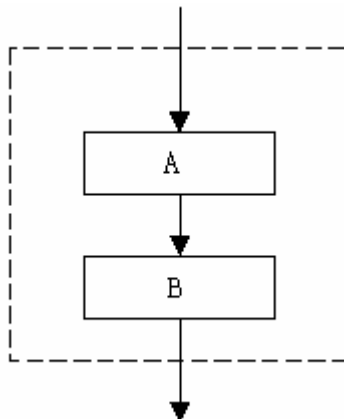
【例 2.9】将例 2.4 求 $1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots + \frac{1}{99} - \frac{1}{100}$ 的算用流程图表示。

一个流程图包括：

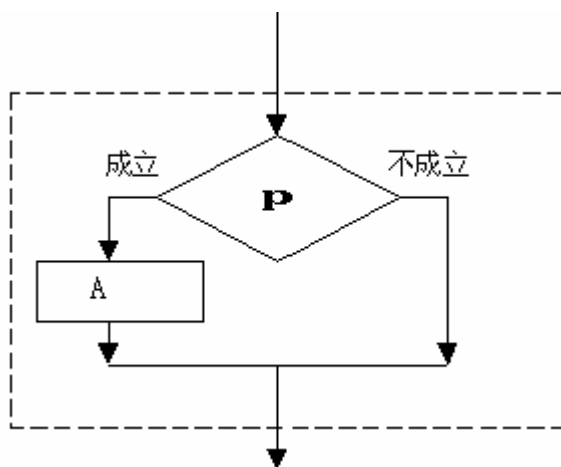
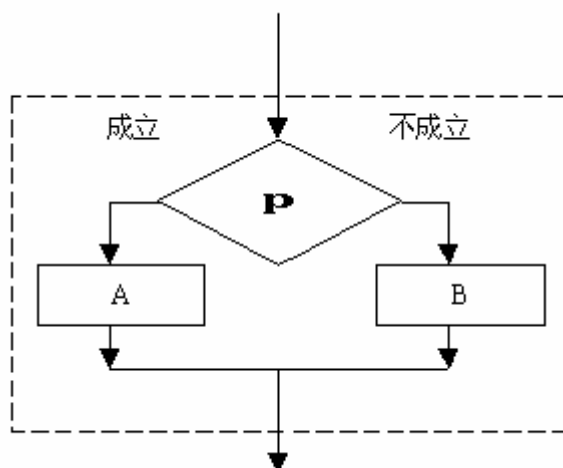
1. 表示相应操作的框；
2. 带箭头的流程线；
3. 框内外必要的文字说明。

2.4.3 三种基本结构和改进的流程图

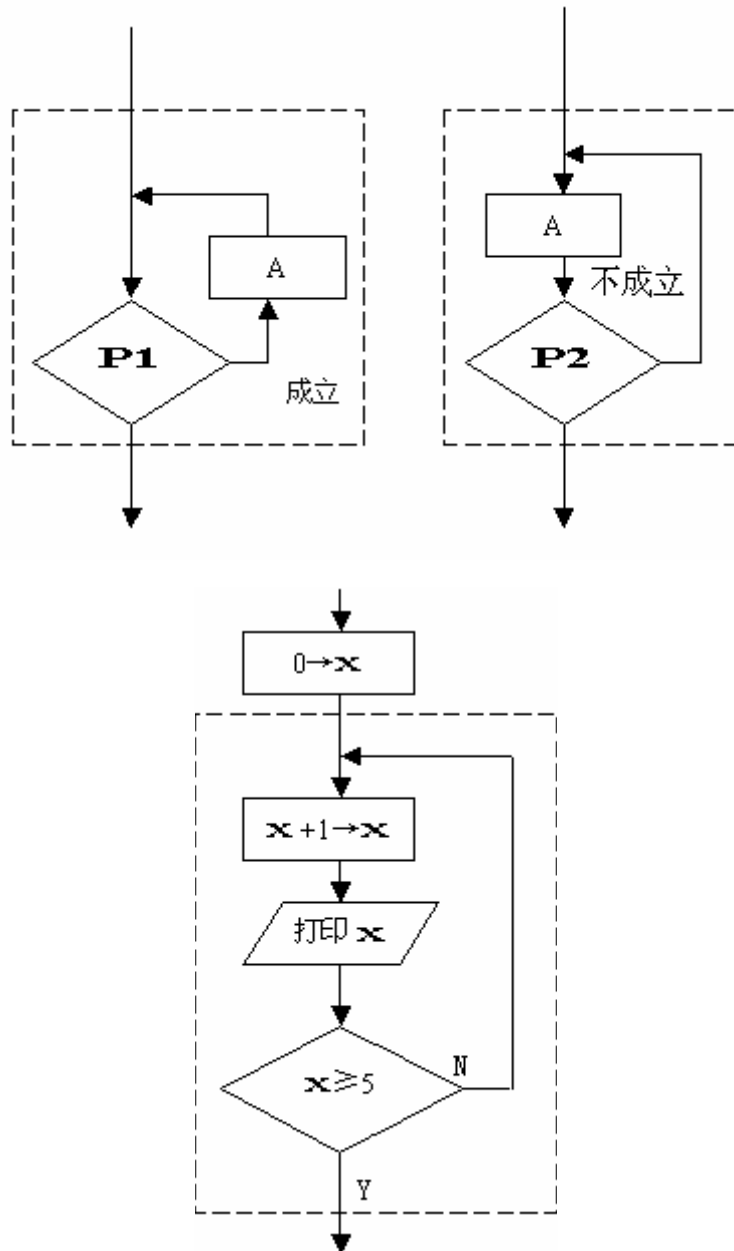
1. 顺序结构：



2. 选择结构：



3. 循环结构



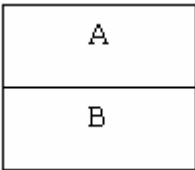
三种基本结构的共同特点：

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

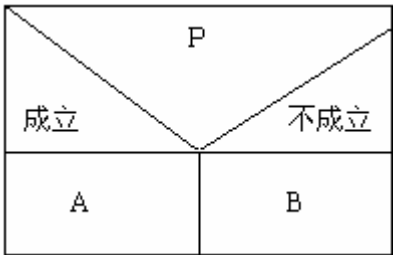
2.4.4 用 N-S 流程图表示算法

1973 年美国学者提出了一种新型流程图：N-S 流程图。

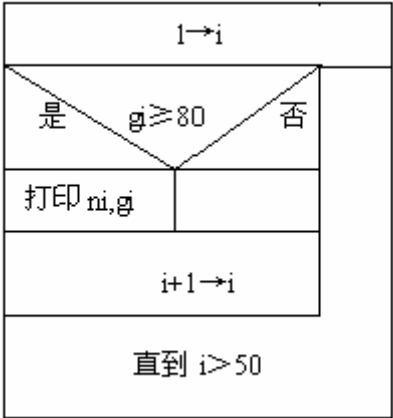
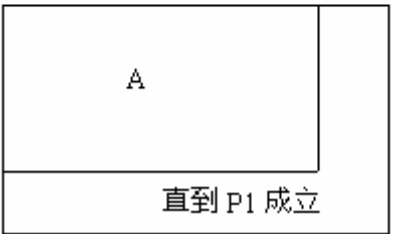
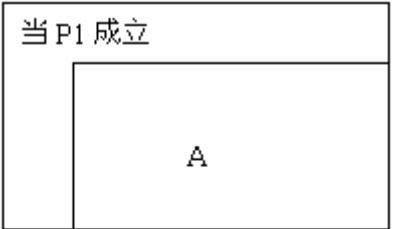
顺序结构：



选择结构：



循环结构：



2.4.5 用伪代码表示算法

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。

2.4.6 用计算机语言表示算法

- 我们的任务是用计算机解题，就是用计算机实现算法；
- 用计算机语言表示算法必须严格遵循所用语言的语法规则。

【例 2.20】求 $1 \times 2 \times 3 \times 4 \times 5$ 用 C 语言表示。

```
main()
{int i,t;
  t=1;
  i=2;
  while(i<=5)
  {t=t*i;
    i=i+1;
  }
  printf("%d",t);
}
```



【例 2.21】求级数的值。

```
main()
{
  int sigh=1;
  float deno=2.0,sum=1.0,term;
  while(deno<=100)
  { sigh= -sigh;
    term= sigh/ deno;
    sum=sum+term;
    deno=deno+1;
  }
  printf("%f",sum);
}
```



2.5 结构化程序设计方法

- 自顶向下；
- 逐步细化；
- 模块化设计；
- 结构化编码。

3 数据类型、运算符与表达式

3.1 C 语言的数据类型

在第一章中，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：

- 数据类型
- 存储类型
- 作用域

在本章中，我们只介绍数据类型的说明。其它说明在以后各章中陆续介绍。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。

数据类型

基本类型

整型

字符型

实型(浮点型)

单精度型

双精度型

枚举类型

构造类型

数组类型

结构体类型

共用体类型

指针类型

空类型

1. 基本数据类型: 基本数据类型最主要的特点是, 其值不可以再分解为其它类型。也就是说, 基本数据类型是自我说明的。
2. 构造数据类型: 构造数据类型是根据已定义的一个或多个数据类型用构造的方法来定义的。也就是说, 一个构造类型的值可以分解成若干个“成员”或“元素”。每个“成员”都是一个基本数据类型或又是一个构造类型。在 C 语言中, 构造类型有以下几种:
 - 数组类型
 - 结构体类型
 - 共用体(联合)类型
3. 指针类型: 指针是一种特殊的, 同时又是具有重要作用的数据类型。其值用来表示某个变量在内存存储器中的地址。虽然指针变量的取值类似于整型量, 但这是两个类型完全不

同的量，因此不能混为一谈。

4. 空类型：在调用函数值时，通常应向调用者返回一个函数值。这个返回的函数值是具有一定的数据类型的，应在函数定义及函数说明中给以说明，例如在例题中给出的 `max` 函数定义中，函数头为：`int max(int a,int b);`其中“`int`”类型说明符即表示该函数的返回值为整型量。又如在例题中，使用了库函数 `sin`，由于系统规定其函数返回值为双精度浮点型，因此在赋值语句 `s=sin(x);`中，`s` 也必须是双精度浮点型，以便与 `sin` 函数的返回值一致。所以在说明部分，把 `s` 说明为双精度浮点型。但是，也有一类函数，调用后并不需要向调用者返回函数值，这种函数可以定义为“空类型”。其类型说明符为 `void`。在后面函数中还要详细介绍。

在本章中，我们先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中陆续介绍。

3.2 常量与变量

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。它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，常量是可以不经说明而直接引用的，而变量则必须先定义后使用。整型量包括整型常量、整型变量。

3.2.1 常量和符号常量

在程序执行过程中，其值不发生改变的量称为常量。

- 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’。
- 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。
- 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），称为宏定义命令（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。

- 习惯上符号常量的标识符用大写字母，变量标识符用小写字母，以示区别。

【例 3.1】符号常量的使用。

```
#define PRICE 30
main()
{
    int num,total;
    num=10;
```

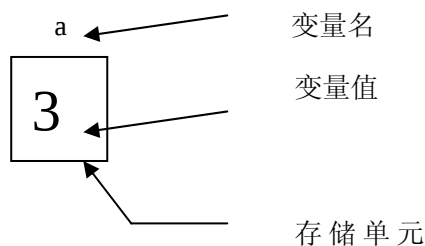
```
total=num* PRICE;  
printf("total=%d",total);  
}
```



- 用标识符代表一个常量，称为符号常量。
- 符号常量与变量不同，它的值在其作用域内不能改变，也不能再被赋值。
- 使用符号常量的好处是：
 - 含义清楚；
 - 能做到“一改全改”。

3.2.2 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。



3.3 整型数据

3.3.1 整型常量的表示方法

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。

以下各数是合法的十进制整常数：

237、-568、65535、1627；

以下各数不是合法的十进制整常数：

023 (不能有前导 0)、23D (含有非十进制数码)。

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。

以下各数是合法的八进制数：

015(十进制为 13)、0101(十进制为 65)、0177777(十进制为 65535)；

以下各数不是合法的八进制数：

256(无前缀 0)、03A2(包含了非八进制数码)、-0127(出现了负号)。

- 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。

以下各数是合法的十六进制整常数：

0X2A(十进制为 42)、0XA0(十进制为 160)、0XFFFF(十进制为 65535)；

以下各数不是合法的十六进制整常数：

5A(无前缀 0X)、0X3H(含有非十六进制数码)。

- 4) 整型常数的后缀：在 16 位字长的机器上，基本整型的长度也为 16 位，因此表示的数的范围也是有限定的。十进制无符号整常数的范围为 0~65535，有符号数为 -32768~+32767。八进制无符号数的表示范围为 0~0177777。十六进制无符号数的表示范围为 0X0~0XFFFF 或 0x0~0xFFFF。如果使用的数超过了上述范围，就必须用长整型数来表示。长整型数是用后缀“L”或“l”来表示的。

例如：

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)、0200000L(十进制为 65536)；

十六进制长整常数：

0X15L(十进制为 21)、0XA5L(十进制为 165)、0X10000L(十进制为 65536)。

长整数 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 编译系统将为它分配 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。因此在运算和输出格式上要予以注意，避免出错。

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”。

例如：

358u,0x38Au,235Lu 均为无符号数。

前缀，后缀可同时使用以表示各种类型的数。如 0XA5Lu 表示十六进制无符号长整数 A5，其十进制为 165。

3.3.2 整型变量

1. 整型数据在内存中的存放形式

如果定义了一个整型变量 i：

```
int i;  
i=10;
```

i 10

0	0	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：

- 正数的补码和原码相同；
- 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
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

由此可知，左面的第一位是表示符号的。

2. 整型变量的分类

- 1) 基本型：类型说明符为 `int`，在内存中占 2 个字节。
- 2) 短整型：类型说明符为 `short int` 或 `short`。所占字节和取值范围均与基本型相同。
- 3) 长整型：类型说明符为 `long int` 或 `long`，在内存中占 4 个字节。
- 4) 无符号型：类型说明符为 `unsigned`。

无符号型又可与上述三种类型匹配而构成：

- 无符号基本型：类型说明符为 `unsigned int` 或 `unsigned`。
- 无符号短整型：类型说明符为 `unsigned short`。
- 无符号长整型：类型说明符为 `unsigned long`。

各种无符号类型量所占的内存空间字节数与相应的有符号类型量相同。但由于省去了符号位，故不能表示负数。

有符号整型变量：最大表示 32767

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

无符号整型变量：最大表示 65535

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

下表列出了 Turbo C 中各类整型量所分配的内存字节数及数的表示范围。

类型说明符	数的范围		字节数
<code>int</code>	-32768~32767	即 $-2^{15} \sim (2^{15}-1)$	2
<code>unsigned int</code>	0~65535	即 $0 \sim (2^{16}-1)$	2
<code>short int</code>	-32768~32767	即 $-2^{15} \sim (2^{15}-1)$	2
<code>unsigned short int</code>	0~65535	即 $0 \sim (2^{16}-1)$	2
<code>long int</code>	-2147483648~2147483647	即 $-2^{31} \sim (2^{31}-1)$	4
<code>unsigned long</code>	0~4294967295	即 $0 \sim (2^{32}-1)$	4

以 13 为例：

`int` 型：

00	00	00	00	00	00	11	01
----	----	----	----	----	----	----	----

`short int` 型：

00	00	00	00	00	00	11	01
----	----	----	----	----	----	----	----

`long int` 型：

00	00	00	00	00	00	00	00	00	00	00	00	00	00	11	01
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

`unsigned int` 型：

00	00	00	00	00	00	11	01
----	----	----	----	----	----	----	----

unsigned short int 型:

00	00	00	00	00	00	11	01
----	----	----	----	----	----	----	----

unsigned long int 型:

00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	11	01
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

3. 整型变量的定义

变量定义的一般形式为:

类型说明符 变量名标识符, 变量名标识符, ...;

例如:

int a,b,c; (a,b,c 为整型变量)

long x,y; (x,y 为长整型变量)

unsigned p,q; (p,q 为无符号整型变量)

在书写变量定义时, 应注意以下几点:

- 允许在一个类型说明符后, 定义多个相同类型的变量。各变量名之间用逗号间隔。类型说明符与变量名之间至少用一个空格间隔。
- 最后一个变量名之后必须以“;”号结尾。
- 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。

【例 3.2】整型变量的定义与使用。

```
main()
{
    int a,b,c,d;
    unsigned u;
    a=12;b=-24;u=10;
    c=a+u;d=b+u;
    printf("a+u=%d,b+u=%d\n",c,d);
}
```



4. 整型数据的溢出

【例 3.3】整型数据的溢出。

```
main()
{
    int a,b;
    a=32767;
    b=a+1;
    printf("%d,%d\n",a,b);
}
```



32767:

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

-32768

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

【例 3.4】

```
main(){
    long x,y;
    int a,b,c,d;
    x=5;
    y=6;
    a=7;
    b=8;
    c=x+a;
    d=y+b;
    printf("c=x+a=%d,d=y+b=%d\n",c,d);
}
```



从程序中可以看到：x, y 是长整型变量，a, b 是基本整型变量。它们之间允许进行运算，运算结果为长整型。但 c, d 被定义为基本整型，因此最后结果为基本整型。本例说明，不同类型的量可以参与运算并相互赋值。其中的类型转换是由编译系统自动完成的。有关类型转换的规则将在以后介绍。

3.4 实型数据

3.4.1 实型常量的表示方法

实型也称为浮点型。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。它有二种形式：十进制小数形式，指数形式。

- 1) 十进制数形式：由数码 0~9 和小数点组成。

例如：

0.0、25.0、5.789、0.13、5.0、300.、-267.8230

均为合法的实数。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。

其一般形式为：

$a E n$ （a 为十进制数，n 为十进制整数）

其值为 $a \times 10^n$ 。

如：

2.1E5 (等于 2.1×10^5)

3.7E-2 (等于 3.7×10^{-2})

0.5E7 (等于 0.5×10^7)

-2.8E-2 (等于 -2.8×10^{-2})

以下不是合法的实数：

345 (无小数点)

E7 (阶码标志 E 之前无数字)

-5 (无阶码标志)

53.-E3 (负号位置不对)

2.7E (无阶码)

标准 C 允许浮点数使用后缀。后缀为 “f” 或 “F” 即表示该数为浮点数。如 356f 和 356. 是等价的。

【例 3.5】说明了这种情况。

```
main(){
    printf("%f\n",356.);
    printf("%f\n",356);
    printf("%f\n",356f);
}
```



3.4.2 实型变量

1. 实型数据在内存中的存放形式

实型数据一般占 4 个字节 (32 位) 内存空间。按指数形式存储。实数 3.14159 在内存中的存放形式如下：

+	.314159	1
数符	小数部分	指数

- 小数部分占的位 (bit) 数愈多, 数的有效数字愈多, 精度愈高。
- 指数部分占的位数愈多, 则能表示的数值范围愈大。

2. 实型变量的分类

实型变量分为：单精度 (float 型)、双精度 (double 型) 和长双精度 (long double 型) 三类。

在 Turbo C 中单精度型占 4 个字节 (32 位) 内存空间, 其数值范围为 $3.4\text{E}-38 \sim 3.4\text{E}+38$, 只能提供七位有效数字。双精度型占 8 个字节 (64 位) 内存空间, 其数值范围为 $1.7\text{E}-308 \sim 1.7\text{E}+308$, 可提供 16 位有效数字。

类型说明符	比特数 (字节数)	有效数字	数的范围
float	32 (4)	6~7	$10^{-37} \sim 10^{38}$
double	64(8)	15~16	$10^{-307} \sim 10^{308}$
long double	128(16)	18~19	$10^{-4931} \sim 10^{4932}$

实型变量定义的格式和书写规则与整型相同。

例如：

float x,y; (x,y 为单精度实型量)

double a,b,c; (a,b,c 为双精度实型量)

3. 实型数据的舍入误差

由于实型变量是由有限的存储单元组成的, 因此能提供的有效数字总是有限的。如下例。

【例 3.6】实型数据的舍入误差。

```
main()
{float a,b;
  a=123456.789e5;
  b=a+20
  printf("%f\n",a);
}
```

```
printf("%f\n",b);  
}
```



注意: $1.0/3*3$ 的结果并不等于 1。

【例 3.7】

```
main()  
{  
    float a;  
    double b;  
    a=33333.33333;  
    b=33333.33333333333333;  
    printf("%f\n%f\n",a,b);  
}
```



- 从本例可以看出, 由于 **a** 是单精度浮点型, 有效位数只有七位。而整数已占五位, 故小数二位后之后均为无效数字。
- **b** 是双精度型, 有效位为十六位。但 Turbo C 规定小数后最多保留六位, 其余部分四舍五入。

3.4.3 实型常数的类型

实型常数不分单、双精度, 都按双精度 **double** 型处理。

3.5 字符型数据

字符型数据包括字符常量和字符变量。

3.5.1 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'、'='、'+'、'？'`

都是合法字符常量。

在 C 语言中, 字符常量有以下特点:

- 1) 字符常量只能用单引号括起来, 不能用双引号或其它括号。
- 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。如 `'5'` 和 `5` 是不同的。`'5'` 是字符常量, 不能参与运算。

3.5.2 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。例如，在前面各例题 `printf` 函数的格式串中用到的“`\n`”就是一个转义字符，其意义是“回车换行”。转义字符主要用来表示那些用一般字符不便于表示的控制代码。

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
<code>\n</code>	回车换行	10
<code>\t</code>	横向跳到下一制表位置	9
<code>\b</code>	退格	8
<code>\r</code>	回车	13
<code>\f</code>	走纸换页	12
<code>\\</code>	反斜线符“\”	92
<code>\'</code>	单引号符	39
<code>\"</code>	双引号符	34
<code>\a</code>	鸣铃	7
<code>\ddd</code>	1~3 位八进制数所代表的字符	
<code>\xhh</code>	1~2 位十六进制数所代表的字符	

广义地讲，C 语言字符集中的任何一个字符均可用转义字符来表示。表中的 `\ddd` 和 `\xhh` 正是为此而提出的。`ddd` 和 `hh` 分别为八进制和十六进制的 ASCII 代码。如 `\101` 表示字母“A”，`\102` 表示字母“B”，`\134` 表示反斜线，`\XOA` 表示换行等。

【例 3.8】转义字符的使用。

```
main()
{
    int a,b,c;
    a=5; b=6; c=7;
    printf("  ab  c\tde\r\n");
    printf("hijk\tL\bM\n");
}
```



3.5.3 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;
```

3.5.4 字符数据在内存中的存储形式及使用方法

每个字符变量被分配一个字节的内存空间, 因此只能存放一个字符。字符值是以 ASCII 码的形式存放在变量的内存单元之中的。

如 x 的十进制 ASCII 码是 120, y 的十进制 ASCII 码是 121。对字符变量 a,b 赋予 'x' 和 'y' 值:

a='x';

b='y';

实际上是在 a,b 两个单元内存放 120 和 121 的二进制代码:

a:

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

b:

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

所以也可以把它们看成是整型量。C 语言允许对整型变量赋以字符值, 也允许对字符变量赋以整型值。在输出时, 允许把字符变量按整型量输出, 也允许把整型量按字符量输出。

整型量为二字节量, 字符量为单字节量, 当整型量按字符型量处理时, 只有低八位字节参与处理。

【例 3.9】向字符变量赋以整数。

```
main()
{
    char a,b;
    a=120;
    b=121;
    printf("%c,%c\n",a,b);
    printf("%d,%d\n",a,b);
}
```



本程序中定义 a, b 为字符型, 但在赋值语句中赋以整型值。从结果看, a, b 值的输出形式取决于 printf 函数格式串中的格式符, 当格式符为 "c" 时, 对应输出的变量值为字符, 当格式符为 "d" 时, 对应输出的变量值为整数。

【例 3.10】

```
main()
{
    char a,b;
    a='a';
    b='b';
    a=a-32;
    b=b-32;
    printf("%c,%c\n%d,%d\n",a,b,a,b);
}
```



本例中, `a`, `b` 被说明为字符变量并赋予字符值, C 语言允许字符变量参与数值运算, 即用字符的 ASCII 码参与运算。由于大小写字母的 ASCII 码相差 32, 因此运算后把小写字母换成大写字母。然后分别以整型和字符型输出。

3.5.5 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: `"CHINA"`, `"C program"`, `"$12.5"` 等都是合法的字符串常量。

字符串常量和字符常量是不同的量。它们之间主要有以下区别:

- 1) 字符常量由单引号括起来, 字符串常量由双引号括起来。
- 2) 字符常量只能是单个字符, 字符串常量则可以含一个或多个字符。
- 3) 可以把一个字符常量赋予一个字符变量, 但不能把一个字符串常量赋予一个字符变量。在 C 语言中没有相应的字符串变量。这是与 BASIC 语言不同的。但是可以用一个字符数组来存放一个字符串常量。在数组一章内予以介绍。
- 4) 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。增加的一个字节中存放字符 `"\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
---	--	---	---	---	---	---	---	---	----

字符常量 `'a'` 和字符串常量 `"a"` 虽然都只有一个字符, 但在内存中的情况是不同的。

`'a'` 在内存中占一个字节, 可表示为:

a

`"a"` 在内存中占二个字节, 可表示为:

a	\0
---	----

3.6 变量赋初值

在程序中常常需要对变量赋初值, 以便使用变量。语言程序中可有多种方法为变量提供初值。本小节先介绍在作变量定义的同时给变量赋以初值的方法。这种方法称为初始化。在变量定义中赋初值的一般形式为:

类型说明符 变量 1= 值 1, 变量 2= 值 2, ……;

例如:

```
int a=3;
```

```
int b,c=5;
```

```
float x=3.2,y=3f,z=0.75;
```

```
char ch1='K',ch2='P';
```

应注意, 在定义中不允许连续赋值, 如 `a=b=c=5` 是不合法的。

【例 3.11】

```
main()
{
```

```
int a=3,b,c=5;
b=a+c;
printf("a=%d,b=%d,c=%d\n",a,b,c);
}
```

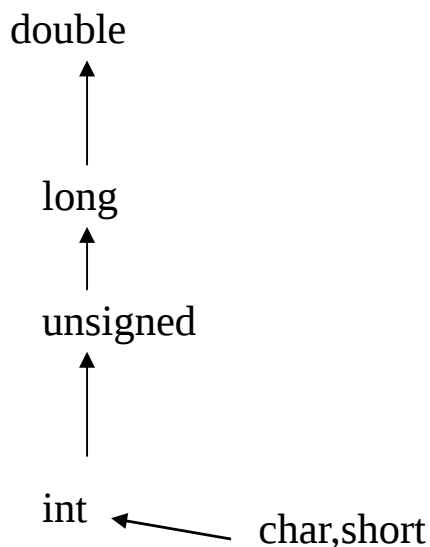


3.7 各类数值型数据之间的混合运算

变量的数据类型是可以转换的。转换的方法有两种，一种是自动转换，一种是强制转换。自动转换发生在不同数据类型的量混合运算时，由编译系统自动完成。自动转换遵循以下规则：

- 1) 若参与运算量的类型不同，则先转换成同一类型，然后进行运算。
- 2) 转换按数据长度增加的方向进行，以保证精度不降低。如 `int` 型和 `long` 型运算时，先把 `int` 量转成 `long` 型后再进行运算。
- 3) 所有的浮点运算都是以双精度进行的，即使仅含 `float` 单精度量运算的表达式，也要先转换成 `double` 型，再作运算。
- 4) `char` 型和 `short` 型参与运算时，必须先转换成 `int` 型。
- 5) 在赋值运算中，赋值号两边量的数据类型不同时，赋值号右边量的类型将转换为左边量的类型。如果右边量的数据类型长度左边长时，将丢失一部分数据，这样会降低精度，丢失的部分按四舍五入向前舍入。

下图表示了类型自动转换的规则。



【例 3.12】

```
main(){
float PI=3.14159;
int s,r=5;
s=r*r*PI;
printf("s=%d\n",s);
}
```

}



本例程序中, PI 为实型; s, r 为整型。在执行 `s=r*r*PI` 语句时, r 和 PI 都转换成 double 型计算, 结果也为 double 型。但由于 s 为整型, 故赋值结果仍为整型, 舍去了小数部分。

强制类型转换

强制类型转换是通过类型转换运算来实现的。

其一般形式为:

(类型说明符) (表达式)

其功能是把表达式的运算结果强制转换成类型说明符所表示的类型。

例如:

(float) a 把 a 转换为实型

(int)(x+y) 把 x+y 的结果转换为整型

在使用强制转换时应注意以下问题:

- 1) 类型说明符和表达式都必须加括号(单个变量可以不加括号), 如把 `(int)(x+y)` 写成 `(int)x+y` 则成了把 x 转换成 int 型之后再与 y 相加了。
- 2) 无论是强制转换或是自动转换, 都只是为了本次运算的需要而对变量的数据长度进行的临时性转换, 而不改变数据说明时对该变量定义的类型。

【例 3.13】

```
main(){
    float f=5.75;
    printf("(int)f=%d,f=%f\n",(int)f,f);
}
```



本例表明, f 虽强制转为 int 型, 但只在运算中起作用, 是临时的, 而 f 本身的类型并不改变。因此, `(int)f` 的值为 5(删去了小数)而 f 的值仍为 5.75。

3.8 算术运算符和算术表达式

C 语言中运算符和表达式数量之多, 在高级语言中是少见的。正是丰富的运算符和表达式使 C 语言功能十分完善。这也是 C 语言的主要特点之一。

C 语言的运算符不仅具有不同的优先级, 而且还有一个特点, 就是它的结合性。在表达式中, 各运算量参与运算的先后顺序不仅要遵守运算符优先级别的规定, 还要受运算符结合性的制约, 以便确定是自左向右进行运算还是自右向左进行运算。这种结合性是其它高级语言的运算符所没有的, 因此也增加了 C 语言的复杂性。

3.8.1 C 运算符简介

C 语言的运算符可分为以下几类:

1. 算术运算符: 用于各类数值运算。包括加(+)、减(-)、乘(*)、除(/)、求余(或称模运算, %)、自增(++)、自减(--)共七种。
2. 关系运算符: 用于比较运算。包括大于(>)、小于(<)、等于(==)、大于等于(>=)、小于等

于($\leq$)和不等于($\neq$)六种。

3. 逻辑运算符:用于逻辑运算。包括与($\&\&$)、或($\|\|$)、非($\!$)三种。
4. 位操作运算符:参与运算的量,按二进制位进行运算。包括位与($\&$)、位或($\|\|$)、位非($\sim$)、位异或($\wedge$)、左移($\ll$)、右移($\gg$)六种。
5. 赋值运算符:用于赋值运算,分为简单赋值($=$)、复合算术赋值($+=$, $-=$, $*=$, $/=$, $\%=$)和复合位运算赋值($\&=$, $\|=$, $\wedge=$, $\gg=$, $\ll=$)三类共十一种。
6. 条件运算符:这是一个三目运算符,用于条件求值($?:$)。
7. 逗号运算符:用于把若干表达式组合成一个表达式($,$)。
8. 指针运算符:用于取内容($*$)和取地址($\&$)二种运算。
9. 求字节数运算符:用于计算数据类型所占的字节数(sizeof)。
10. 特殊运算符:有括号 $()$,下标 $[]$,成员 $(\rightarrow, .)$ 等几种。

3.8.2 算术运算符和算术表达式

1. 基本的算术运算符

- 加法运算符“ $+$ ”:加法运算符为双目运算符,即应有两个量参与加法运算。如 $a+b$, $4+8$ 等。具有右结合性。
- 减法运算符“ $-$ ”:减法运算符为双目运算符。但“ $-$ ”也可作负值运算符,此时为单目运算,如 $-x$, -5 等具有左结合性。
- 乘法运算符“ $*$ ”:双目运算,具有左结合性。
- 除法运算符“ $/$ ”:双目运算具有左结合性。参与运算量均为整型时,结果也为整型,舍去小数。如果运算量中有一个是实型,则结果为双精度实型。

【例 3.14】

```
main(){
printf("\n\n%d,%d\n",20/7,-20/7);
printf("%f,%f\n",20.0/7,-20.0/7);
}
```



本例中, $20/7$, $-20/7$ 的结果均为整型,小数全部舍去。而 $20.0/7$ 和 $-20.0/7$ 由于有实数参与运算,因此结果也为实型。

- 求余运算符(模运算符)“ $\%$ ”:双目运算,具有左结合性。要求参与运算的量均为整型。求余运算的结果等于两数相除后的余数。

【例 3.15】

```
main(){
printf("%d\n",100%3);
}
```



本例输出 100 除以 3 所得的余数 1。

2. 算术表达式和运算符的优先级和结合性

表达式是由常量、变量、函数和运算符组合起来的式子。一个表达式有一个值及其类型,它们等于计算表达式所得结果的值和类型。表达式求值按运算符的优先级和结合性规定的顺序进行。单个的常量、变量、函数可以看作是表达式的特例。

算术表达式是由算术运算符和括号连接起来的式子。

- **算术表达式：**用算术运算符和括号将运算对象（也称操作数）连接起来的、符合 C 语法规则的式子。

以下是算术表达式的例子：

```
a+b
(a*2) / c
(x+r)*8-(a+b) / 7
++I
sin(x)+sin(y)
(++i)-(j++)+(k--)
```

- **运算符的优先级：**C 语言中，运算符的运算优先级共分为 15 级。1 级最高，15 级最低。在表达式中，优先级较高的先于优先级较低的进行运算。而在一个运算量两侧的运算符优先级相同时，则按运算符的结合性所规定的结合方向处理。
- **运算符的结合性：**C 语言中各运算符的结合性分为两种，即左结合性(自左至右)和右结合性(自右至左)。例如算术运算符的结合性是自左至右，即先左后右。如有表达式 $x-y+z$ 则 y 应先与“-”号结合，执行 $x-y$ 运算，然后再执行 $+z$ 的运算。这种自左至右的结合方向就称为“左结合性”。而自右至左的结合方向称为“右结合性”。最典型的右结合性运算符是赋值运算符。如 $x=y=z$ ，由于“=”的右结合性，应先执行 $y=z$ 再执行 $x=(y=z)$ 运算。C 语言运算符中有不少为右结合性，应注意区别，以避免理解错误。

3. 强制类型转换运算符

其一般形式为：

(类型说明符) (表达式)

其功能是把表达式的运算结果强制转换成类型说明符所表示的类型。

例如：

```
(float) a      把 a 转换为实型
(int)(x+y)     把 x+y 的结果转换为整型
```

4. 自增、自减运算符

自增 1，自减 1 运算符：自增 1 运算符记为“++”，其功能是使变量的值自增 1。

自减 1 运算符记为“--”，其功能是使变量值自减 1。

自增 1，自减 1 运算符均为单目运算，都具有右结合性。可有以下几种形式：

$++i$ i 自增 1 后再参与其它运算。

$--i$ i 自减 1 后再参与其它运算。

$i++$ i 参与运算后， i 的值再自增 1。

$i--$ i 参与运算后， i 的值再自减 1。

在理解和使用上容易出错的是 $i++$ 和 $i--$ 。特别是当它们出在较复杂的表达式或语句中时，常常难于弄清，因此应仔细分析。

【例 3.16】

```
main(){
int i=8;
printf("%d\n",++i);
printf("%d\n",--i);
printf("%d\n",i++);
printf("%d\n",i--);
printf("%d\n",-i++);
```

```
printf("%d\n",-i--);
```

```
}
```



i 的初值为 8，第 2 行 i 加 1 后输出故为 9；第 3 行减 1 后输出故为 8；第 4 行输出 i 为 8 之后再加 1(为 9)；第 5 行输出 i 为 9 之后再减 1(为 8)；第 6 行输出 -8 之后再加 1(为 9)，第 7 行输出 -9 之后再减 1(为 8)。

【例 3.17】

```
main(){
int i=5,j=5,p,q;
p=(i++)+(i++)+(i++);
q=(++j)+(++j)+(++j);
printf("%d,%d,%d,%d",p,q,i,j);
```

```
}
```



这个程序中，对 $P=(i++)+(i++)+(i++)$ 应理解为三个 i 相加，故 P 值为 15。然后 i 再自增 1 三次相当于加 3 故 i 的最后值为 8。而对于 q 的值则不然， $q=(++j)+(++j)+(++j)$ 应理解为 q 先自增 1，再参与运算，由于 q 自增 1 三次后值为 8，三个 8 相加的和为 24，j 的最后值仍为 8。

3.9 赋值运算符和赋值表达式

1. 赋值运算符

简单赋值运算符和表达式:简单赋值运算符记为“=”。由“=”连接的式子称为赋值表达式。其一般形式为:

变量=表达式

例如:

$x=a+b$

$w=\sin(a)+\sin(b)$

$y=i+++--j$

赋值表达式的功能是计算表达式的值再赋予左边的变量。赋值运算符具有右结合性。

因此

$a=b=c=5$

可理解为

$a=(b=(c=5))$

在其它高级语言中，赋值构成了一个语句，称为赋值语句。而在 C 中，把“=”定义为运算符，从而组成赋值表达式。凡是表达式可以出现的地方均可出现赋值表达式。

例如，式子:

$x=(a=5)+(b=8)$

是合法的。它的意义是把 5 赋予 a，8 赋予 b，再把 a,b 相加，和赋予 x，故 x 应等于 13。

在 C 语言中也可以组成赋值语句，按照 C 语言规定，任何表达式在其末尾加上分号就构成成为语句。因此如

$x=8;a=b=c=5;$

都是赋值语句，在前面各例中我们已大量使用过了。

2. 类型转换

如果赋值运算符两边的数据类型不相同，系统将自动进行类型转换，即把赋值号右边的类型换成左边的类型。具体规定如下：

- 1) 实型赋予整型，舍去小数部分。前面的例子已经说明了这种情况。
- 2) 整型赋予实型，数值不变，但将以浮点形式存放，即增加小数部分(小数部分的值为 0)。
- 3) 字符型赋予整型，由于字符型为一个字节，而整型为二个字节，故将字符的 ASCII 码值放到整型量的低八位中，高八位为 0。整型赋予字符型，只把低八位赋予字符量。

【例 3.18】

```
main(){
    int a,b=322;
    float x,y=8.88;
    char c1='k',c2;
    a=y;
    x=b;
    a=c1;
    c2=b;
    printf("%d,%f,%d,%c",a,x,a,c2);
}
```



本例表明了上述赋值运算中类型转换的规则。a 为整型，赋予实型量 y 值 8.88 后只取整数 8。x 为实型，赋予整型量 b 值 322，后增加了小数部分。字符型量 c1 赋予 a 变为整型，整型量 b 赋予 c2 后取其低八位成为字符型(b 的低八位为 01000010，即十进制 66，按 ASCII 码对应于字符 B)。

3. 复合的赋值运算符

在赋值符“=”之前加上其它二目运算符可构成复合赋值符。如 +=, -=, *=, /=, %=, <<=, >>=, &=, ^=, |=。

构成复合赋值表达式的一般形式为：

变量 双目运算符=表达式

它等效于

变量=变量 运算符 表达式

例如：

a+=5	等价于 a=a+5
x*=y+7	等价于 x=x*(y+7)
r%=p	等价于 r=r%p

复合赋值符这种写法，对初学者可能不习惯，但十分有利于编译处理，能提高编译效率并产生质量较高的目标代码。

3.10 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。

【例 3.19】

```
main(){
int a=2,b=4,c=6,x,y;
y=(x=a+b),(b+c);
printf("y=%d,x=%d",y,x);
}
```



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) 逗号表达式一般形式中的表达式 1 和表达式 2 也可以又是逗号表达式。
例如：

表达式 1, (表达式 2, 表达式 3)
形成了嵌套情形。因此可以把逗号表达式扩展为以下形式：
表达式 1, 表达式 2, …表达式 n

整个逗号表达式的值等于表达式 n 的值。

- 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。

3.11 小结

3.11.1 C 的数据类型

基本类型， 构造类型， 指针类型， 空类型

3.11.2 基本类型的分类及特点

	类型说明符	字节	数值范围
字符型	char	1	C 字符集
基本整型	int	2	-32768~32767
短整型	short int	2	-32768~32767
长整型	long int	4	-214783648~214783647
无符号型	unsigned	2	0~65535
无符号长整型	unsigned long	4	0~4294967295
单精度实型	float	4	3/4E-38~3/4E+38
双精度实型	double	8	1/7E-308~1/7E+308

3.11.3 常量后缀

L 或 l	长整型
U 或 u	无符号数
F 或 f	浮点数

3.11.4 常量类型

整数，长整数，无符号数，浮点数，字符，字符串，符号常数，转义字符。

3.11.5 数据类型转换

- 自动转换:在不同类型数据的混合运算中，由系统自动实现转换，由少字节类型向多字节类型转换。不同类型的量相互赋值时也由系统自动进行转换，把赋值号右边的类型转换为左边的类型。
- 强制转换:由强制转换运算符完成转换。

3.11.6 运算符优先级和结合性

一般而言，单目运算符优先级较高，赋值运算符优先级低。算术运算符优先级较高，关系和逻辑运算符优先级较低。多数运算符具有左结合性，单目运算符、三目运算符、赋值运算符具有右结合性。

3.11.7 表达式

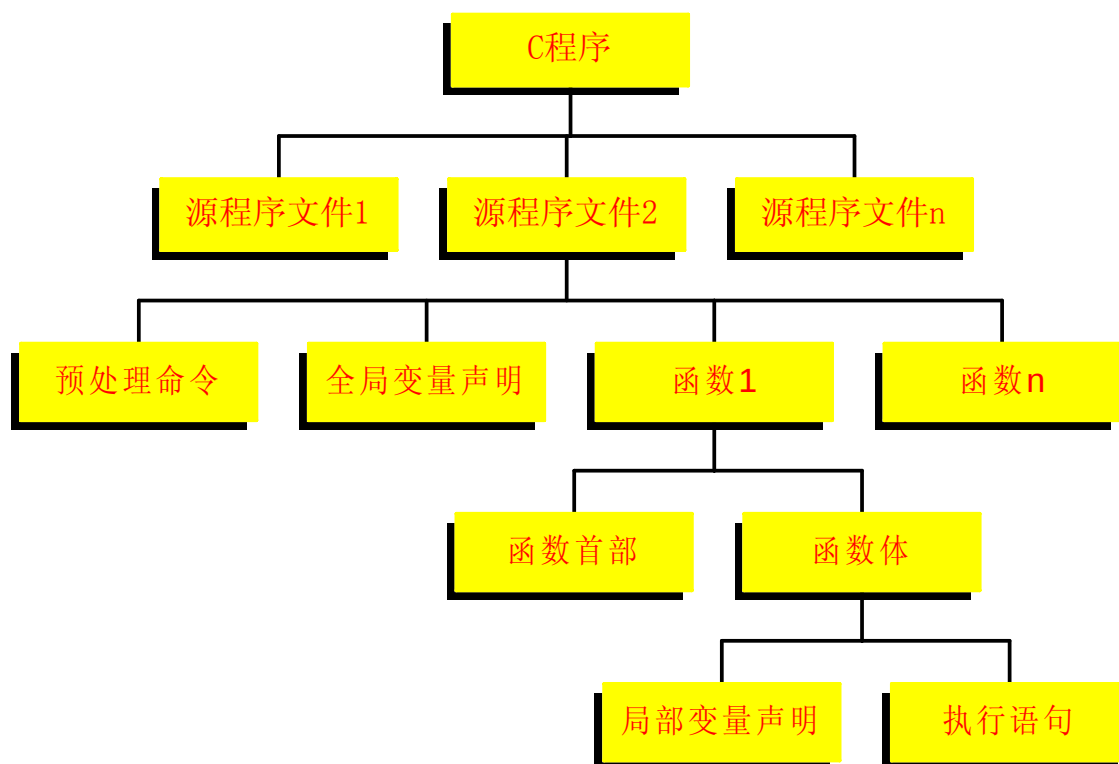
表达式是由运算符连接常量、变量、函数所组成的式子。每个表达式都有一个值和类型。表达式求值按运算符的优先级和结合性所规定的顺序进行。

4 最简单的 C 程序设计—顺序程序设计

从程序流程的角度来看，程序可以分为三种基本结构，即顺序结构、分支结构、循环结构。这三种基本结构可以组成所有的各种复杂程序。C 语言提供了多种语句来实现这些程序结构。本章介绍这些基本语句及其在顺序结构中的应用，使读者对 C 程序有一个初步的认识，为后面各章的学习打下基础。

4.1 C 语句概述

C 程序的结构：



C 程序的执行部分是由语句组成的。程序的功能也是由执行语句实现的。

C 语句可分为以下五类：

- 1) 表达式语句
- 2) 函数调用语句
- 3) 控制语句
- 4) 复合语句
- 5) 空语句

1. 表达式语句：表达式语句由表达式加上分号“;”组成。

其一般形式为：

表达式；

执行表达式语句就是计算表达式的值。

例如：

`x=y+z;` 赋值语句;

`y+z;` 加法运算语句, 但计算结果不能保留, 无实际意义;

`i++;` 自增 1 语句, `i` 值增 1。

2. 函数调用语句: 由函数名、实际参数加上分号 “;” 组成。

其一般形式为:

函数名(实际参数表);

执行函数语句就是调用函数体并把实际参数赋予函数定义中的形式参数, 然后执行被调函数体中的语句, 求取函数值 (在后面函数中再详细介绍)。

例如:

`printf("C Program");`调用库函数, 输出字符串。

3. 控制语句: 控制语句用于控制程序的流程, 以实现程序的各种结构方式。它们由特定的语句定义符组成。C 语言有九种控制语句。可分成以下三类:

1) 条件判断语句: `if` 语句、`switch` 语句;

2) 循环执行语句: `do while` 语句、`while` 语句、`for` 语句;

3) 转向语句: `break` 语句、`goto` 语句、`continue` 语句、`return` 语句。

4. 复合语句: 把多个语句用括号 `{}` 括起来组成的一个语句称复合语句。

在程序中应把复合语句看成是单条语句, 而不是多条语句。

例如:

```
{ x=y+z;
  a=b+c;
  printf("%d%d", x, a);
}
```

是一条复合语句。

复合语句内的各条语句都必须以分号 “;” 结尾, 在括号 “`{}`” 外不能加分号。

5. 空语句: 只有分号 “;” 组成的语句称为空语句。空语句是什么也不执行的语句。在程序中空语句可用来作空循环体。

例如

```
while(getchar() != '\n')
;
```

本语句的功能是, 只要从键盘输入的字符不是回车则重新输入。

这里的循环体为空语句。

4.2 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, 下述形式

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接合性, 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它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。

如下述说明是错误的:

```
int a=b=c=5
```

必须写为

```
int a=5, b=5, c=5;
```

而赋值语句允许连续赋值。

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。

下述语句是合法的:

```
if((x=y+5)>0) z=x;
```

语句的功能是, 若表达式 $x=y+5$ 大于 0 则 $z=x$ 。

下述语句是非法的:

```
if((x=y+5;)>0) z=x;
```

因为 $x=y+5$; 是语句, 不能出现在表达式中。

4.3 数据输入输出的概念及在 C 语言中的实现

- 1) 所谓输入输出是以计算机为主体而言的。
- 2) 本章介绍的是向标准输出设备显示器输出数据的语句。
- 3) 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。
- 4) 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, 因此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。

5) 考虑到 printf 和 scanf 函数使用频繁, 系统允许在使用这两个函数时可不加

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

或

```
#include "stdio.h"
```

4.4 字符数据的输入输出

4.4.1 putchar 函数（字符输出函数）

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('\101'); (也是输出字符 A)
```

```
putchar('\n');   (换行)
```

对控制字符则执行控制功能, 不在屏幕上显示。

使用本函数前必须要用文件包含命令:

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

或

```
#include "stdio.h"
```

【例 4.1】输出单个字符。

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

```
main() {
```

```
    char a='B', b='o', c='k';
```

```
    putchar(a);putchar(b);putchar(b);putchar(c);putchar('\t');
```

```
    putchar(a);putchar(b);
```

```
    putchar('\n');
```

```
    putchar(b);putchar(c);
```

```
}
```



4.4.2 getchar 函数（键盘输入函数）

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 4.2】输入单个字符。

```
#include<stdio.h>
void main() {
    char c;
    printf("input a character\n");
    c=getchar();
    putchar(c);
}
```



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文件 “`stdio.h`”。
- 3) 在 TC 屏幕下运行含本函数程序时，将退出 TC 屏幕进入用户屏幕等待用户输入。输入完毕再返回 TC 屏幕。
- 4) 程序最后两行可用下面两行的任意一行代替：

```
putchar(getchar());
printf("%c", getchar());
```

4.5 格式输入与输出

4.5.1 printf 函数（格式输出函数）

`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。

1. 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。但作为一个特例，不要求在使用 `printf` 函数之前必须包含 `stdio.h` 文件。

`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。

非格式字符串在输出时原样照印，在显示中起提示作用。

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 4.3】

```
main()
```

```
{  
    int a=88,b=89;  
    printf("%d %d\n",a,b);  
    printf("%d,%d\n",a,b);  
    printf("%c,%c\n",a,b);  
    printf("a=%d,b=%d",a,b);  
}
```



本例中四次输出了 a,b 的值,但由于格式控制串不同,输出的结果也不相同。第四行的输出语句格式控制串中,两格式串%d 之间加了一个空格(非格式字符),所以输出的 a,b 值之间有一个空格。第五行的 printf 语句格式控制串中加入的是非格式字符逗号,因此输出的 a,b 值之间加了一个逗号。第六行的格式串要求按字符型输出 a,b 值。第七行中为了提示输出结果又增加了非格式字符串。

2. 格式字符串

在 Turbo C 中格式字符串的一般形式为:

[标志][输出最小宽度][.精度][长度]类型

其中方括号[]中的项为可选项。

各项的意义介绍如下:

1) 类型:类型字符用以表示输出数据的类型,其格式符和意义如下表所示:

格式字符	意 义
d	以十进制形式输出带符号整数(正数不输出符号)
o	以八进制形式输出无符号整数(不输出前缀 0)
x, X	以十六进制形式输出无符号整数(不输出前缀 0x)
u	以十进制形式输出无符号整数
f	以小数形式输出单、双精度实数
e, E	以指数形式输出单、双精度实数
g, G	以%f 或%e 中较短的输出宽度输出单、双精度实数
c	输出单个字符
s	输出字符串

2) 标志:标志字符为-、+、#、空格四种,其意义下表所示:

标 志	意 义
-	结果左对齐,右边填充格
+	输出符号(正号或负号)
空格	输出值为正时冠以空格,为负时冠以负号
#	对 c, s, d, u 类无影响;对 o 类,在输出时加前缀 o;对 x 类,在输出时加前缀 0x;对 e, g, f 类当结果有小数时才给出小数点

3) 输出最小宽度:用十进制整数来表示输出的最少位数。若实际位数多于定义的宽度,则按实际位数输出,若实际位数少于定义的宽度则补以空格或 0。

4) 精度:精度格式符以“.”开头,后跟十进制整数。本项的意义是:如果输出数字,则表示小数的位数;如果输出的是字符,则表示输出字符的个数;若实际位数大于所定义的精度数,则截去超过的部分。

5. 长度:长度格式符为 h, l 两种, h 表示按短整型量输出, l 表示按长整型量输出。

【例 4.4】

```
main()
{
    int a=15;
    float b=123.1234567;
    double c=12345678.1234567;
    char d='p';
    printf("a=%d,%5d,%o,%x\n", a, a, a, a);
    printf("b=%f,%lf,%5.4lf,%e\n", b, b, b, b);
    printf("c=%lf,%f,%8.4lf\n", c, c, c);
    printf("d=%c,%8c\n", d, d);
}
```



本例第七行中以四种格式输出整型变量 a 的值, 其中“%5d”要求输出宽度为 5, 而 a 值为 15 只有两位故补三个空格。第八行中以四种格式输出实型量 b 的值。其中“%f”和“%lf”格式的输出生同, 说明“l”符对“f”类型无影响。“%5.4lf”指定输出宽度为 5, 精度为 4, 由于实际长度超过 5 故应该按实际位数输出, 小数位数超过 4 位部分被截去。第九行输出双精度实数, “%8.4lf”由于指定精度为 4 位故截去了超过 4 位的部分。第十行输出字符变量 d , 其中“%8c”指定输出宽度为 8 故在输出字符 p 之前补加 7 个空格。

使用 printf 函数时还要注意一个问题, 那就是输出表列中的求值顺序。不同的编译系统不一定相同, 可以从左到右, 也可从右到左。Turbo C 是按从右到左进行的。请看下面两个例子:

【例 4.5】

```
main() {
    int i=8;
    printf("%d\n%d\n%d\n%d\n%d\n%d\n", ++i, --i, i++, i--, -i++, -i--);
}
```



【例 4.6】

```
main() {
    int i=8;
    printf("%d\n", ++i);
    printf("%d\n", --i);
    printf("%d\n", i++);
    printf("%d\n", i--);
    printf("%d\n", -i++);
    printf("%d\n", -i--);
}
```



这两个程序的区别是用一个 printf 语句和多个 printf 语句输出。但从结果可以看出是不同的。为什么结果会不同呢? 就是因为 printf 函数对输出表中各量求值的顺序是自右至左进行的。在第一例中, 先对最后一项“-i--”求值, 结果为-8, 然后 i 自减 1 后为 7。再

对“ $-i++$ ”项求值得-7，然后 i 自增 1 后为 8。再对“ $i--$ ”项求值得 8，然后 i 再自减 1 后为 7。再求“ $i++$ ”项得 7，然后 i 再自增 1 后为 8。再求“ $--i$ ”项， i 先自减 1 后输出，输出值为 7。最后才求输出表列中的第一项“ $++i$ ”，此时 i 自增 1 后输出 8。

但是必须注意，求值顺序虽是自右至左，但是输出顺序还是从左至右，因此得到的结果是上述输出结果。

4.5.2 scanf 函数(格式输入函数)

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 4.7】

```
main() {
    int a, b, c;
    printf("input a, b, c\n");
    scanf("%d%d%d", &a, &b, &c);
    printf("a=%d, b=%d, c=%d", a, b, c);
}
```



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 TC 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 TC 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。

【例 4.8】

```
main() {  
    char a, b;  
    printf("input character a,b\n");  
    scanf ("%c%c", &a, &b);  
    printf ("%c%c\n", a, b);  
}
```



由于 scanf 函数"%c%c"中没有空格, 输入 M N, 结果输出只有 M。而输入改为 MN 时则可输出 MN 两字符。

【例 4.9】

```
main() {  
    char a, b;  
    printf("input character a,b\n");  
    scanf ("%c %c", &a, &b);  
    printf ("\n%c%c\n", a, b);  
}
```



本例表示 scanf 格式控制串"%c %c"之间有空格时, 输入的数据之间可以有空格间隔。

5) 如果格式控制串中有非格式字符则输入时也要输入该非格式字符。

例如:

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

其中用非格式符", "作间隔符, 故输入时应为:

5, 6, 7

又如:

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

则输入应为:

a=5, b=6, c=7

6) 如输入的数据与输出的类型不一致时, 虽然编译能够通过, 但结果将不正确。

【例 4.10】

```
main() {  
    int a;  
    printf("input a number\n");  
    scanf("%d",&a);  
    printf("%ld",a);  
}
```



由于输入数据类型为整型，而输出语句的格式串中说明为长整型，因此输出结果和输入数据不符。如改动程序如下：

【例 4.11】

```
main() {  
    long a;  
    printf("input a long integer\n");  
    scanf("%ld",&a);  
    printf("%ld",a);  
}
```



运行结果为：

```
input a long integer  
1234567890  
1234567890
```

当输入数据改为长整型后，输入输出数据相等。

【例 4.12】

```
main() {  
    char a,b,c;  
    printf("input character a,b,c\n");  
    scanf("%c %c %c",&a,&b,&c);  
    printf("%d,%d,%d\n%c,%c,%c\n",a,b,c,a-32,b-32,c-32);  
}
```



输入三个小写字母，输出其 ASCII 码和对应的大写字母。

【例 4.13】

```
main() {  
    int a;  
    long b;  
    float f;  
    double d;  
    char c;  
    printf("\nint:%d\nlong:%d\nfloat:%d\ndouble:%d\nchar:%d\n",sizeof(a),sizeof(b),  
    sizeof(f),sizeof(d),sizeof(c));  
}
```



输出各种数据类型的字节长度。

4.6 顺序结构程序设计举例

【例 4.14】输入三角形的三边长，求三角形面积。

已知三角形的三边长 a, b, c ，则该三角形的面积公式为：

$$area = \sqrt{s(s-a)(s-b)(s-c)},$$

其中 $s = (a+b+c)/2$

源程序如下：

```
#include<math.h>
main()
{
    float a,b,c,s,area;
    scanf("%f,%f,%f",&a,&b,&c);
    s=1.0/2*(a+b+c);
    area=sqrt(s*(s-a)*(s-b)*(s-c));
    printf("a=%7.2f,b=%7.2f,c=%7.2f,s=%7.2f\n",a,b,c,s);
    printf("area=%7.2f\n",area);
}
```



【例 4.15】求 $ax^2+bx+c=0$ 方程的根， a, b, c 由键盘输入，设 $b^2-4ac>0$ 。

求根公式为：

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad p = \frac{-b}{2a}$$

$$\text{令 } q = \frac{\sqrt{b^2 - 4ac}}{2a}, \quad q = \frac{\sqrt{b^2 - 4ac}}{2a}$$

则 $x_1 = p + q$

$x_2 = p - q$

源程序如下：

```
#include<math.h>
main()
{
    float a,b,c,disc,x1,x2,p,q;
    scanf("a=%f,b=%f,c=%f",&a,&b,&c);
    disc=b*b-4*a*c;
    p=-b/(2*a);
    q=sqrt(disc)/(2*a);
    x1=p+q;x2=p-q;
```

```
printf("\nx1=%5.2f\nx2=%5.2f\n",x1,x2);  
}
```



5 分支结构程序

5.1 关系运算符和表达式

在程序中经常需要比较两个量的大小关系，以决定程序下一步的工作。比较两个量的运算符称为关系运算符。

5.1.1 关系运算符及其优先次序

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。

5.1.2 关系表达式

关系表达式的一般形式为：

表达式 关系运算符 表达式

例如：

```
a+b>c-d
x>3/2
'a'+1<c
-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：

```
a>(b>c)
a!=(c==d)
```

等。

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 5.1】

```
main() {
```

```
char c='k';
int i=1, j=2, k=3;
float x=3e+5, y=0.85;
printf("%d, %d\n", 'a'+5<c, -i-2*j>=k+1);
printf("%d, %d\n", 1<j<5, x-5.25<=x+y);
printf("%d, %d\n", i+j+k==2*j, k==j==i+5);
}
```



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。

5.2 逻辑运算符和表达式

5.2.1 逻辑运算符极其优先次序

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) $\rightarrow$ `&&` (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	$((a + b) > c) \ \&\& \ ((x + y) < b)$

5.2.2 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 || 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||0` 的值为“真”，即为 1。

5.2.3 逻辑表达式

逻辑表达式的一般形式为：

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 5.2】

```
main() {  
    char c='k';  
    int i=1, j=2, k=3;  
    float x=3e+5, y=0.85;  
    printf("%d,%d\n", !x*y, !!x);  
    printf("%d,%d\n", x||i&&j-3, i<j&&x<y);  
    printf("%d,%d\n", i==5&&c&&(j=8), x+y||i+j+k);  
}
```



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 1。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 0，0 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。

5.3 if 语句

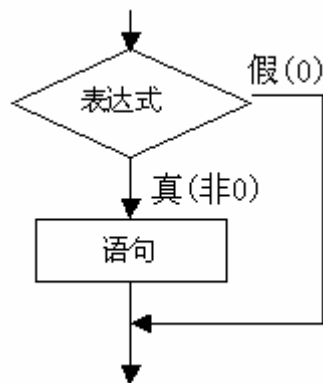
用 if 语句可以构成分支结构。它根据给定的条件进行判断，以决定执行某个分支程序段。C 语言的 if 语句有三种基本形式。

5.3.1 if 语句的三种形式

1. 第一种形式为基本形式：if

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 5.3】

```
main() {  
    int a, b, max;  
    printf("\n input two numbers:  ");  
    scanf("%d%d", &a, &b);  
    max=a;  
    if (max<b) max=b;  
    printf("max=%d", max);  
}
```

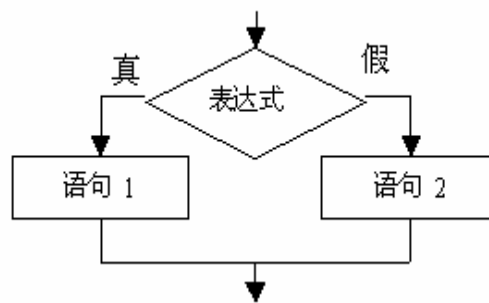


本例程序中，输入两个数 a, b。把 a 先赋予变量 max，再用 if 语句判别 max 和 b 的大小，如 max 小于 b，则把 b 赋予 max。因此 max 中总是大数，最后输出 max 的值。

2. 第二种形式为：if-else

```
if(表达式)  
    语句 1;  
else  
    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 5.4】**

```
main() {  
    int a, b;  
    printf("input two numbers:  ");  
    scanf("%d%d", &a, &b);  
    if(a>b)  
        printf("max=%d\n", a);  
    else  
        printf("max=%d\n", b);  
}
```



输入两个整数，输出其中的大数。

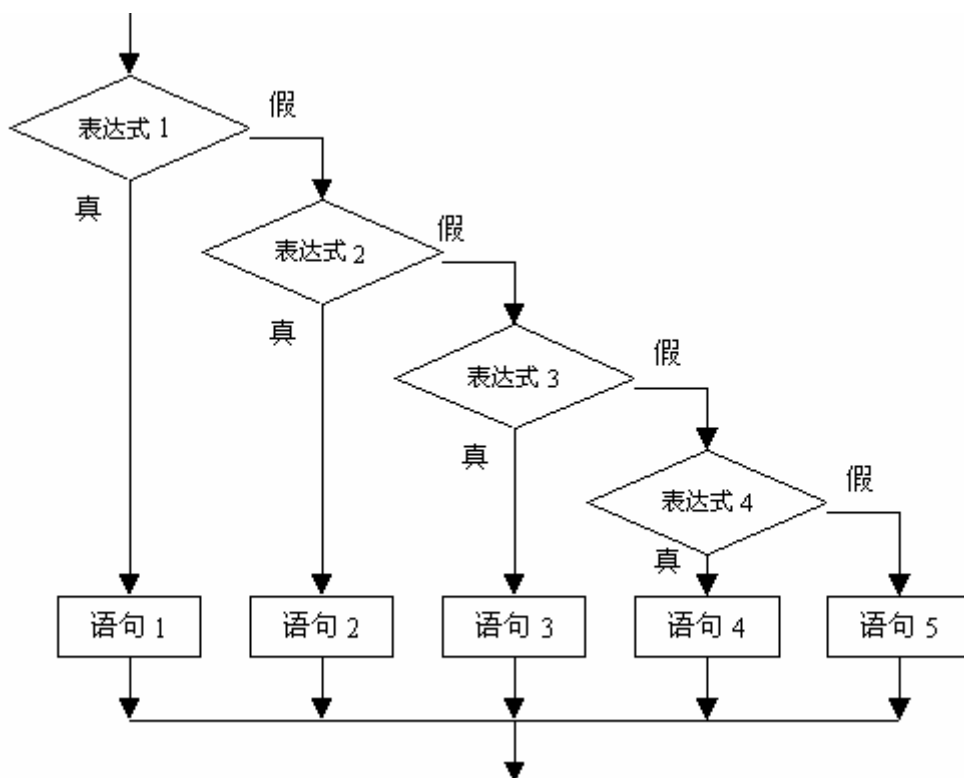
改用 if-else 语句判别 a, b 的大小，若 a 大，则输出 a，否则输出 b。

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：

```
if(表达式 1)  
    语句 1;  
else if(表达式 2)  
    语句 2;  
else if(表达式 3)  
    语句 3;  
...  
else if(表达式 m)  
    语句 m;  
else  
    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 3-3 所示。



【例 5.5】

```

#include "stdio.h"
main() {
    char c;
    printf("input a character:  ");
    c=getchar();
    if(c<32)
        printf("This is a control character\n");
    else if(c>='0' && c<='9')
        printf("This is a digit\n");
    else if(c>='A' && c<='Z')
        printf("This is a capital letter\n");
    else if(c>='a' && c<='z')
        printf("This is a small letter\n");
    else
        printf("This is an other character\n");
}
  
```



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母。

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：

```
if(a=5) 语句;
```

```
if(b) 语句;
```

都是允许的。只要表达式的值为非 0，即为“真”。

如在：

```
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：

```
if(a=b)
    printf("%d",a);
else
    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：

```
if(a>b)
{a++;
 b++;}
else
{a=0;
 b=10;}
```

5.3.2 if 语句的嵌套

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：

```
if(表达式)
    if 语句;
```

或者为

```
if(表达式)
    if 语句;
else
    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：

```
if(表达式 1)
```

```
    if(表达式 2)
        语句 1;
    else
        语句 2;
```

其中的 else 究竟是与哪一个 if 配对呢?

应该理解为:

```
    if(表达式 1)
        if(表达式 2)
            语句 1;
        else
            语句 2;
```

还是应理解为:

```
    if(表达式 1)
        if(表达式 2)
            语句 1;

    else
        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 5.6】

```
main() {
    int a,b;
    printf("please input A,B:   ");
    scanf("%d%d",&a,&b);
    if(a!=b)
        if(a>b) printf("A>B\n");
        else    printf("A<B\n");
        else    printf("A=B\n");
}
```



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 5.7】

```
main() {
    int a,b;
    printf("please input A,B:   ");
    scanf("%d%d",&a,&b);
    if(a==b) printf("A=B\n");
    else if(a>b) printf("A>B\n");
    else printf("A<B\n");
}
```



5.3.3 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：

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：

```
if(a>b) max=a;
else max=b;
```

可用条件表达式写为

```
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

```
max=(a>b)?a:b
```

可以去掉括号而写为

```
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：

```
a>b?a:c>d?c:d
```

应理解为

```
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 5.8】

```
main() {
    int a, b, max;
    printf("\n input two numbers:  ");
    scanf("%d%d", &a, &b);
    printf("max=%d", a>b?a:b);
}
```



用条件表达式对上例重新编程，输出两个数中的大数。

5.4 switch 语句

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：

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

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时，则执行 default 后的语句。

【例 4.9】

```
main() {  
    int a;  
    printf("input integer number:      ");  
    scanf("%d",&a);  
    switch (a) {  
        case 1:printf("Monday\n");  
        case 2:printf("Tuesday\n");  
        case 3:printf("Wednesday\n");  
        case 4:printf("Thursday\n");  
        case 5:printf("Friday\n");  
        case 6:printf("Saturday\n");  
        case 7:printf("Sunday\n");  
        default:printf("error\n");  
    }  
}
```



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？这恰恰反应了 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 4.10】

```
main() {
```

```
int a;
printf("input integer number:  ");
scanf("%d",&a);
switch (a) {
    case 1:printf("Monday\n");break;
    case 2:printf("Tuesday\n"); break;
    case 3:printf("Wednesday\n");break;
    case 4:printf("Thursday\n");break;
    case 5:printf("Friday\n");break;
    case 6:printf("Saturday\n");break;
    case 7:printf("Sunday\n");break;
    default:printf("error\n");
}
```



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。

5.5 程序举例

【例 4.11】输入三个整数, 输出最大数和最小数。

```
main() {
    int a, b, c, max, min;
    printf("input three numbers:  ");
    scanf("%d%d%d",&a, &b, &c);
    if(a>b)
        {max=a;min=b;}
    else
        {max=b;min=a;}
    if(max<c)
        max=c;
    else
        if(min>c)
            min=c;
    printf("max=%d\nmin=%d", max, min);
}
```



本程序中, 首先比较输入的 a, b 的大小, 并把大数装入 max, 小数装入 min 中, 然后再与 c 比较, 若 max 小于 c, 则把 c 赋予 max; 如果 c 小于 min, 则把 c 赋予 min。因此 max

内总是最大数，而 min 内总是最小数。最后输出 max 和 min 的值即可。

【例 4.12】计算器程序。用户输入运算数和四则运算符，输出计算结果。

```
main() {  
    float a,b;  
    char c;  
    printf("input expression: a+(-,*,/)b \n");  
    scanf("%f%c%f",&a,&c,&b);  
    switch(c) {  
        case '+': printf("%f\n",a+b);break;  
        case '-': printf("%f\n",a-b);break;  
        case '*': printf("%f\n",a*b);break;  
        case '/': printf("%f\n",a/b);break;  
        default: printf("input error\n");  
    }  
}
```



本例可用于四则运算求值。switch 语句用于判断运算符，然后输出运算值。当输入运算符不是+, -, *, /时给出错误提示。

6 循环控制

6.1 概述

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；

6.2 goto 语句以及用 goto 语句构成循环

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：

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 6.1】用 goto 语句和 if 语句构成循环， $\sum_{n=1}^{100} n$ 。

```
main()
{
    int i, sum=0;
    i=1;
loop:  if(i<=100)
        {sum=sum+i;
          i++;
          goto loop;}
    printf("%d\n", sum);
}
```



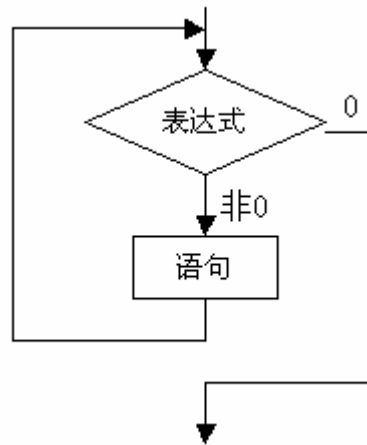
6.3 while 语句

while 语句的一般形式为：

while(表达式) 语句

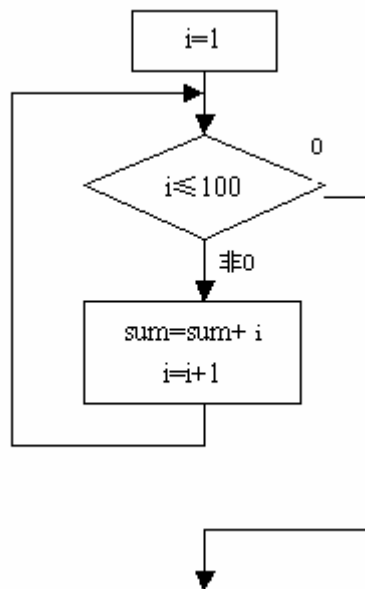
其中表达式是循环条件，语句为循环体。

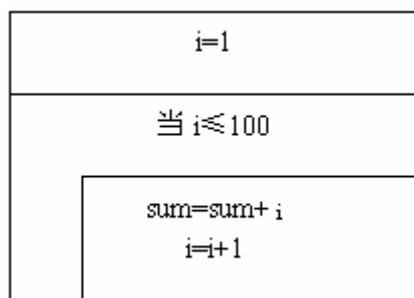
while 语句的语义是：计算表达式的值，当值为真(非 0)时，执行循环体语句。其执行过程可用下图表示。



【例 6.2】用while语句求 $\sum_{n=1}^{100} n$ 。

用传统流程图和 N-S 结构流程图表示算法，见图：





```
main()
{
    int i, sum=0;
    i=1;
    while(i<=100)
    {
        sum=sum+i;
        i++;
    }
    printf("%d\n", sum);
}
```



【例 6.3】统计从键盘输入一行字符的个数。

```
#include <stdio.h>
main() {
    int n=0;
    printf("input a string:\n");
    while(getchar()!='\n') n++;
    printf("%d", n);
}
```



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 6.4】

```
main() {
    int a=0, n;
    printf("\n input n: ");
    scanf("%d", &n);
    while (n--)
        printf("%d ", a++*2);
}
```

}



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++)。

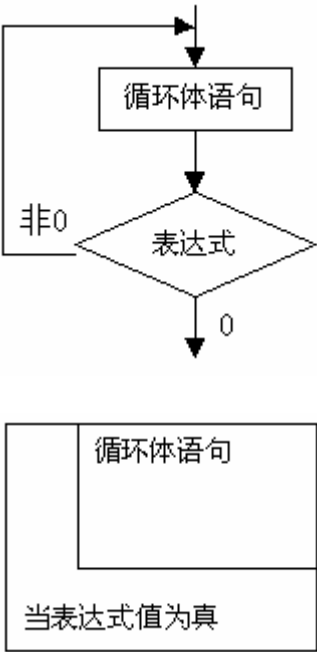
2) 循环体如包括有一个以上的语句，则必须用 {} 括起来，组成复合语句。

6.4 do-while 语句

do-while 语句的一般形式为：

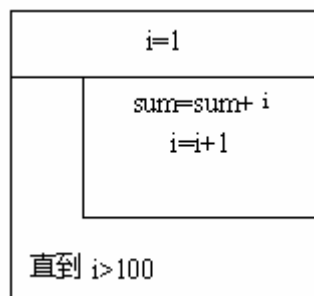
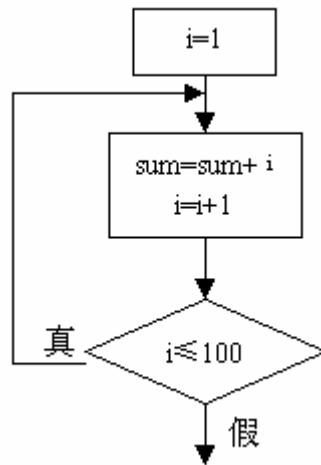
```
do
    语句
while(表达式);
```

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 6.5】用do-while语句求 $\sum_{n=1}^{100} n$ 。

用传统流程图和 N-S 结构流程图表示算法，见图：



```

main()
{
    int i, sum=0;
    i=1;
    do
    {
        sum=sum+i;
        i++;
    }
    while(i<=100)
    printf("%d\n", sum);
}

```



同样当有许多语句参加循环时，要用“{”和“}”把它们括起来。

【例 6.6】while 和 do-while 循环比较。

```

(1) main()
{
    int sum=0, i;
    scanf("%d", &i);
    while(i<=10)
    {
        sum=sum+i;
        i++;
    }
}

```

```
printf("sum=%d", sum);  
}
```



```
(2) main()  
{int sum=0, i;  
  scanf("%d", &i);  
  do  
  {sum=sum+i;  
   i++;  
  }  
  while(i<=10);  
  printf("sum=%d", sum);  
}
```



6.5 for 语句

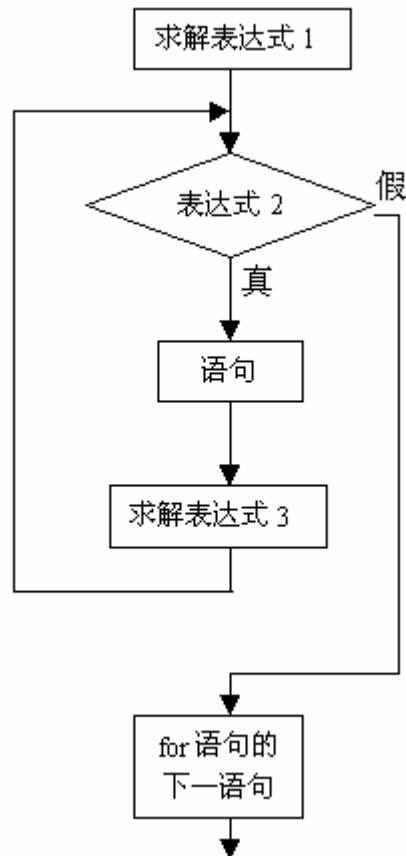
在 C 语言中, for 语句使用最为灵活, 它完全可以取代 while 语句。它的一般形式为:

for(表达式 1; 表达式 2; 表达式 3) 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：

for(循环变量赋初值; 循环条件; 循环变量增量) 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：

```
for(i=1; i<=100; i++) 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：

```
i=1;
while (i<=100)
{ sum=sum+i;
  i++;
}
```

对于 for 循环中语句的一般形式，就是如下的 while 循环形式：

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

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：

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

相当于：

```
i=1;
while(1)
{sum=sum+i;
i++;}
```

- 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：

```
for(i=1;i<=100;)
{sum=sum+i;
i++;}
```

- 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：

```
for(;i<=100;)
{sum=sum+i;
i++;}
```

相当于：

```
while(i<=100)
{sum=sum+i;
i++;}
```

- 6) 3 个表达式都可以省略。

例如：

```
for(;; )语句
```

相当于：

```
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：

```
for(sum=0;i<=100;i++)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。

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

或：

```
for(i=0, j=100;i<=100;i++, j--)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：

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

又如：

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

6.6 循环的嵌套

【例 6.7】

```
main()
{
    int i, j, k;
    printf("i j k\n");
    for (i=0; i<2; i++)
        for(j=0; j<2; j++)
            for(k=0; k<2; k++)
                printf("%d %d %d\n", i, j, k);
}
```



6.7 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。

6.8 break 和 continue 语句

6.8.1 break 语句

`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 6.8】

```
main()
{
    int i=0;
    char c;
    while(1)                                /*设置循环*/
    {
        c='0';                             /*变量赋初值*/
    }
}
```

```
while(c!=13&& c!=27) /*键盘接收字符直到按回车或 Esc 键*/
{
    c=getch();
    printf("%c\n", c);
}
if(c==27)
    break;          /*判断若按 Esc 键则退出循环*/
i++;
printf("The No. is %d\n", i);
}
printf("The end");
}
```



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。

6.8.2 continue 语句

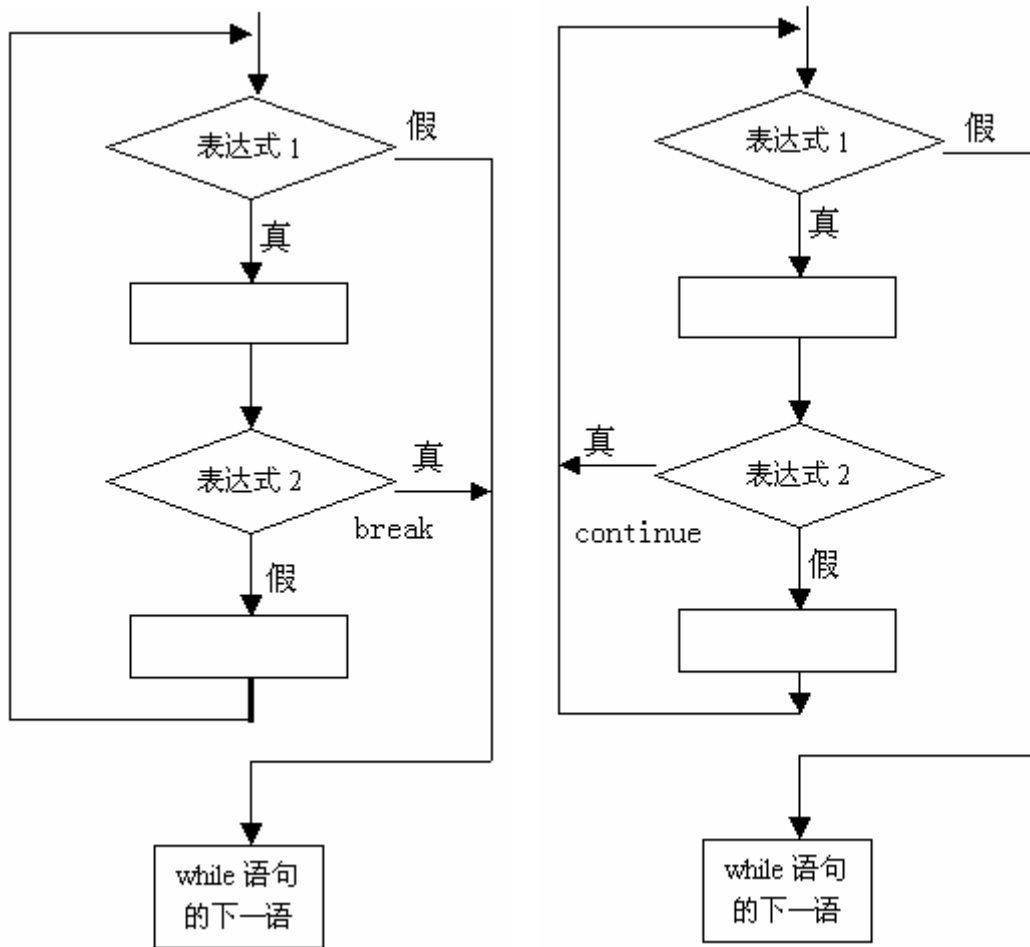
continue 语句的作用是跳过循环本中剩余的语句而强行执行下一次循环。continue 语句只用在 for、while、do-while 等循环体中, 常与 if 条件语句一起使用, 用来加速循环。其执行过程可用下图表示。

1) while(表达式 1)

```
{ .....
    if(表达式 2)break;
    .....
}
```

2) while(表达式 1)

```
{ .....
    if(表达式 2)continue;
    .....
}
```



【例 6.9】

```

main()
{
    char c;
    while(c!=13)    /*不是回车符则循环*/
    {
        c=getch();
        if(c==0X1B)
            continue; /*若按 Esc 键不输出便进行下次循环*/
        printf("%c\n", c);
    }
}

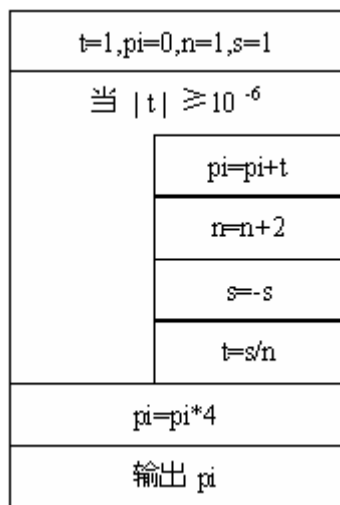
```



6.9 程序举例

【例 6.10】用 $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots$ 公式求 π 。

N-S 流程图:



```
#include<math.h>
main()
{
    int s;
    float n, t, pi;
    t=1, pi=0; n=1.0; s=1;
    while(fabs(t)>1e-6)
    {
        pi=pi+t;
        n=n+2;
        s=-s;
        t=s/n;
    }
    pi=pi*4;
```

```
printf("pi=%10.6f\n", pi);
```

}

【例 6.11】判断 m 是否素数。

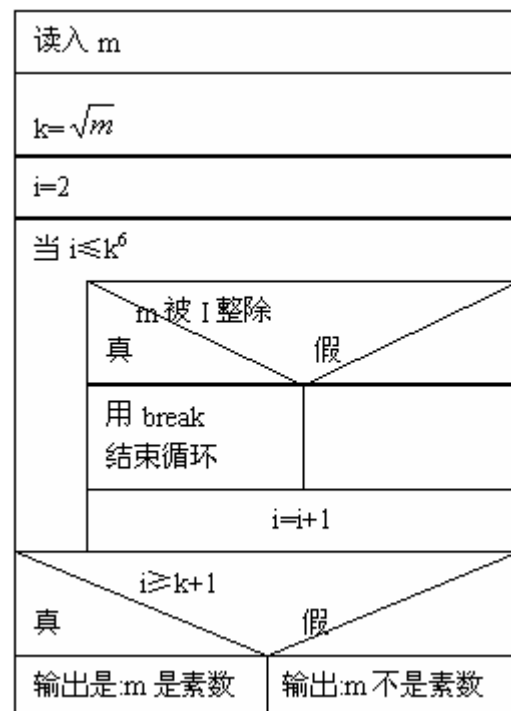
N-S 流程图:

```
#include<math.h>
main()
{
    int m, i, k;
    scanf("%d", &m);
    k=sqrt(m);
    for(i=2; i<=k; i++)
        if(m%i==0) break;
    if(i>=k+1)
        printf("%d is a prime number\n", m);
    else
        printf("%d is not a prime number\n", m);
}
```



【例 6.12】求 100 至 200 间的全部素数。

```
#include<math.h>
main()
{
    int m, i, k, n=0;
    for(m=101; m<=200; m=m+2)
    {
        k=sqrt(m);
        for(i=2; i<=k; i++)
```



```
        if(m%i==0)break;
        if(i>=k+1)
            {printf("%d",m);
              n=n+1;}
        if(n%n==0)printf("\n");
    }
printf("\n");
}
```



7 数组

在程序设计中，为了处理方便，把具有相同类型的若干变量按有序的形式组织起来。这些按序排列的同类数据元素的集合称为数组。在 C 语言中，数组属于构造数据类型。一个数组可以分解为多个数组元素，这些数组元素可以是基本数据类型或是构造类型。因此按数组元素的类型不同，数组又可分为数值数组、字符数组、指针数组、结构数组等各种类别。本章介绍数值数组和字符数组，其余的在以后各章陆续介绍。

7.1 一维数组的定义和引用

7.1.1 一维数组的定义方式

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：

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：

```
int a[10];           说明整型数组 a，有 10 个元素。
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。
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：

```
main()
{
    int a;
    float a[10];
    .....
}
```

是错误的。

- 4) 方括号中常量表达式表示数组元素的个数，如 `a[5]` 表示数组 `a` 有 5 个元素。但是其下标从 0 开始计算。因此 5 个元素分别为 `a[0]`, `a[1]`, `a[2]`, `a[3]`, `a[4]`。
- 5) 不能在方括号中用变量来表示元素的个数，但是可以是符号常数或常量表达式。

例如：

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

是合法的。

但是下述说明方式是错误的。

```
main()
{
    int n=5;
    int a[n];
    .....
}
```

- 6) 允许在同一个类型说明中，说明多个数组和多个变量。

例如：

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

7.1.2 一维数组元素的引用

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

数组元素的一般形式为：

数组名[下标]

其中下标只能为整型常量或整型表达式。如为小数时，C 编译将自动取整。

例如：

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

都是合法的数组元素。

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

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：

```
for(i=0; i<10; i++)
    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：

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

【例 7.1】

```
main()
{
    int i, a[10];
    for(i=0; i<=9; i++)
```

```
    a[i]=i;
    for(i=9;i>=0;i--)
        printf("%d ",a[i]);
}
```

**【例 7.2】**

```
main()
{
    int i,a[10];
    for(i=0;i<10;)
        a[i++]=i;
    for(i=9;i>=0;i--)
        printf("%d",a[i]);
}
```

**【例 7.3】**

```
main()
{
    int i,a[10];
    for(i=0;i<10;)
        a[i++]=2*i+1;
    for(i=0;i<=9;i++)
        printf("%d ",a[i]);
    printf("\n%d %d\n",a[5.2],a[5.8]);
}
```



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。

7.1.3 一维数组的初始化

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：

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：

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

相当于 `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:

```
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:

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

而不能写为:

```
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:

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

可写为:

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

7.1.4 一维数组程序举例

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 7.4】

```
main()
{
    int i, max, a[10];
    printf("input 10 numbers:\n");
    for(i=0; i<10; i++)
        scanf("%d", &a[i]);
    max=a[0];
    for(i=1; i<10; i++)
        if(a[i]>max) max=a[i];
    printf("maxmum=%d\n", max);
}
```



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 7.5】

```
main()
{
```

```
int i, j, p, q, s, a[10];
printf("\n input 10 numbers:\n");
for(i=0; i<10; i++)
    scanf("%d", &a[i]);
for(i=0; i<10; i++) {
    p=i; q=a[i];
    for(j=i+1; j<10; j++)
        if(q<a[j]) { p=j; q=a[j]; }
    if(i!=p)
        {s=a[i];
         a[i]=a[p];
         a[p]=s; }
    printf("%d", a[i]);
}
```



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≠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。

7.2 二维数组的定义和引用

7.2.1 二维数组的定义

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：

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：

```
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：

```
a[0][0], a[0][1], a[0][2], a[0][3]
a[1][0], a[1][1], a[1][2], a[1][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）。

7.2.2 二维数组元素的引用

二维数组的元素也称为双下标变量，其表示的形式为：

数组名[下标][下标]

其中下标应为整型常量或整型表达式。

例如：

`a[3][4]`

表示 `a` 数组三行四列的元素。

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

【例 7.6】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：

```
main()
{
    int i, j, s=0, average, v[3], a[5][3];
    printf("input score\n");
    for(i=0; i<3; i++)
    {
        for(j=0; j<5; j++)
        { scanf("%d", &a[j][i]);
          s=s+a[j][i]; }
        v[i]=s/5;
        s=0;
    }
    average =(v[0]+v[1]+v[2])/3;
    printf("math:%d\nc language:%d\ndbase:%d\n", v[0], v[1], v[2]);
```

```
printf("total:%d\n", average );
}
```



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，退出内循环后再将该累加成绩除以 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。

7.2.3 二维数组的初始化

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

例如对数组 $a[5][3]$ ：

1) 按行分段赋值可写为：

```
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：

```
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 7.7】

```
main()
{
    int i, j, s=0, average, v[3];
    int a[5][3]={ {80, 75, 92}, {61, 65, 71}, {59, 63, 70}, {85, 87, 90}, {76, 77, 85} };
    for(i=0; i<3; i++)
    { for(j=0; j<5; j++)
        s=s+a[j][i];
        v[i]=s/5;
        s=0;
    }
    average=(v[0]+v[1]+v[2])/3;
    printf("math:%d\n language:%d\n Foxpro:%d\n", v[0], v[1], v[2]);
    printf("total:%d\n", average);
}
```



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：

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

是对每一行的第一列元素赋值，未赋值的元素取 0 值。赋值后各元素的值为：

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

```
3 0 0
```

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

赋值后的元素值为:

```
0 1 0
```

```
0 0 2
```

```
3 0 0
```

- 2) 如对全部元素赋初值, 则第一维的长度可以不给出。

例如:

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

可以写为:

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

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

如二维数组 `a[3][4]`, 可分解为三个一维数组, 其数组名分别为:

```
a[0]
```

```
a[1]
```

```
a[2]
```

对这三个一维数组不需另作说明即可使用。这三个一维数组都有 4 个元素, 例如:

一维数组 `a[0]` 的元素为 `a[0][0]`, `a[0][1]`, `a[0][2]`, `a[0][3]`。

必须强调的是, `a[0]`, `a[1]`, `a[2]` 不能当作下标变量使用, 它们是数组名, 不是一个单纯的下标变量。

7.2.4 二维数组程序举例

7.3 字符数组

用来存放字符量的数组称为字符数组。

7.3.1 字符数组的定义

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。

7.3.2 字符数组的初始化

字符数组也允许在定义时作初始化赋值。

例如：

```
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：

```
char c[]={‘c’, ‘ ’, ‘p’, ‘r’, ‘o’, ‘g’, ‘r’, ‘a’, ‘m’};
```

这时 C 数组的长度自动定为 9。

7.3.3 字符数组的引用

【例 7.8】

```
main()
{
    int i, j;
    char a[][5]={{'B', 'A', 'S', 'I', 'C'}, {'d', 'B', 'A', 'S', 'E'}};
    for(i=0; i<=1; i++)
    {
        for(j=0; j<=4; j++)
            printf("%c", a[i][j]);
        printf("\n");
    }
}
```



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以说明。

7.3.4 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：

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

可写为：

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

或去掉 {} 写为：

```
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。

7.3.5 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 7.9】

```
main()
{
    char c[]="BASIC\nBASE";
    printf("%s\n", c);
}
```



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：

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

【例 7.10】

```
main()
{
    char st[15];
    printf("input string:\n");
    scanf("%s", st);
    printf("%s\n", st);
}
```



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：

```
input string:
this is a book
输出为:
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 7.11】

```
main()
{
    char st1[6], st2[6], st3[6], st4[6];
    printf("input string:\n");
    scanf("%s%s%s%s", st1, st2, st3, st4);
    printf("%s %s %s %s\n", st1, st2, st3, st4);
}
```



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。

7.3.6 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 7.12】

```
#include "stdio.h"
main()
{
    char c[]="BASIC\ndBASE";
    puts(c);
}
```



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 7.13】

```
#include "stdio.h"
main()
{
    char st[15];
    printf("input string:\n");
    gets(st);
    puts(st);
}
```



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 7.14】

```
#include "string.h"
main()
{
    static char st1[30]="My name is ";
    int st2[10];
    printf("input your name:\n");
    gets(st2);
}
```

```
    strcat(st1,st2);  
    puts(st1);  
}
```



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 7.15】

```
#include "string.h"  
main()  
{  
    char st1[15],st2[]="C Language";  
    strcpy(st1,st2);  
    puts(st1);printf("\n");  
}
```



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 7.16】

```
#include "string.h"  
main()  
{ int k;  
    static char st1[15],st2[]="C Language";  
    printf("input a string:\n");  
    gets(st1);  
    k=strcmp(st1,st2);  
    if(k==0) printf("st1=st2\n");  
    if(k>0) printf("st1>st2\n");  
    if(k<0) printf("st1<st2\n");  
}
```



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 7.17】

```
#include "string.h"
main()
{ int k;
  static char st[]="C language";
  k=strlen(st);
  printf("The lenth of the string is %d\n",k);
}
```



7.4 程序举例

【例 7.18】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, 则可将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。

```
main()
{
  int i, j, p, q, s, n, a[11]={127, 3, 6, 28, 54, 68, 87, 105, 162, 18};
  for(i=0; i<10; i++)
  { p=i; q=a[i];
    for(j=i+1; j<10; j++)
      if(q<a[j]) {p=j; q=a[j];}
    if(p!=i)
    {
      s=a[i];
      a[i]=a[p];
      a[p]=s;
    }
    printf("%d ", a[i]);
  }
  printf("\ninput number:\n");
  scanf("%d", &n);
  for(i=0; i<10; i++)
    if(n>a[i])
```

```
        {for(s=9;s>=i;s--) a[s+1]=a[s];
        break;}
        a[i]=n;
    for(i=0;i<=10;i++)
        printf("%d ",a[i]);
    printf("\n");
}
```



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 7.19】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。

```
a=( 3  16 87  65
    4  32 11 108
    10 25 12  37)
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:

```
main()
{
    int a[][4]={3,16,87,65,4,32,11,108,10,25,12,27};
    int b[3],i,j,l;
    for(i=0;i<=2;i++)
        { l=a[i][0];
        for(j=1;j<=3;j++)
            if(a[i][j]>l) l=a[i][j];
        b[i]=l;}
    printf("\narray a:\n");
    for(i=0;i<=2;i++)
        { for(j=0;j<=3;j++)
        printf("%5d",a[i][j]);
        printf("\n");}
    printf("\narray b:\n");
    for(i=0;i<=2;i++)
        printf("%5d",b[i]);
    printf("\n");
}
```



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 1。进入内循环后，把 1 与后面各列元素比较，并把比 1 大者赋予 1。内循环结束时 1 即为该行最大的元素，然后把 1 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 7.20】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：

```
main()
{
    char st[20], cs[5][20];
    int i, j, p;
    printf("input country's name:\n");
    for(i=0; i<5; i++)
        gets(cs[i]);
    printf("\n");
    for(i=0; i<5; i++)
    { p=i; strcpy(st, cs[i]);
    for(j=i+1; j<5; j++)
        if(strcmp(cs[j], st)<0) {p=j; strcpy(st, cs[j]);}
    if(p!=i)
    {
        strcpy(st, cs[i]);
        strcpy(cs[i], cs[p]);
        strcpy(cs[p], st);
    }
    puts(cs[i]);}printf("\n");
}
```



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。

7.5 本章小结

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。

8 函 数

8.1 概述

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，又可把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。

8.2 函数定义的一般形式

1. 无参函数的定义形式

```
类型标识符 函数名()  
{声明部分  
  语句  
}
```

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:

```
void Hello()  
{  
    printf ("Hello, world \n");  
}
```

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

类型标识符 函数名(形式参数表列)

{声明部分

语句

}

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:

```
int max(int a, int b)
{
    if (a>b) return a;
    else return b;
}
```

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。

【例 8.1】

```
int max(int a, int b)
{
    if(a>b) return a;
    else return b;
}
main()
{
    int max(int a, int b);
    int x, y, z;
    printf("input two numbers:\n");
    scanf("%d%d", &x, &y);
    z=max(x, y);
    printf("maxmum=%d", z);
}
```



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 1 行至第 5 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 8 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 12 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。

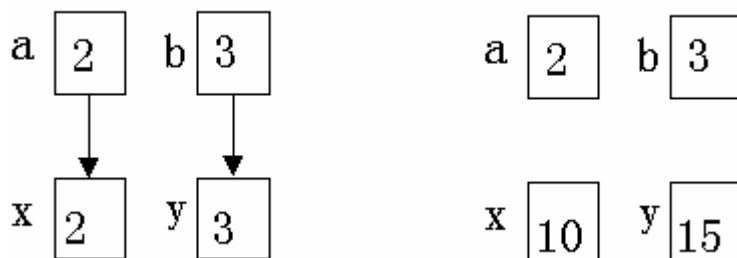
8.3 函数的参数和函数的值

8.3.1 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。



【例 8.2】可以说明这个问题。

```
main()
{
    int n;
    printf("input number\n");
    scanf("%d", &n);
    s(n);
    printf("n=%d\n", n);
}

int s(int n)
{
    int i;
    for(i=n-1; i>=1; i--)
```

```
n=n+i;
printf("n=%d\n",n);
}
```



本程序中定义了一个函数s，该函数的功能是求 $\sum n_i$ 的值。在主函数中输入n值，并作为实参，在调用时传送给s 函数的形参变量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。

8.3.2 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 8.1 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：

return 表达式;

或者为：

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 8.2 中函数 s 并不向主函数返函数值，因此可定义为：

```
void s(int n)
{ .....
}
```

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

```
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。

8.4 函数的调用

8.4.1 函数调用的一般形式

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：

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。

8.4.2 函数调用的方式

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：`printf("%d", a); 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：`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 8.3】

```
main()
{
    int i=8;
    printf("%d\n%d\n%d\n%d\n", ++i, --i, i++, i--);
}
```



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

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 Turbo C 现定是自右至左求值，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。

8.4.3 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：

类型说明符 被调函数名(类型 形参, 类型 形参...);

或为：

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 8.1 main 函数中对 max 函数的说明为：

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

或写为：

```
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。例 8.2 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。例如例 8.1 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：

```
char str(int a);
float f(float b);
main()
{
    .....
}
char str(int a)
{
    .....
}
float f(float b)
{
    .....
}
```

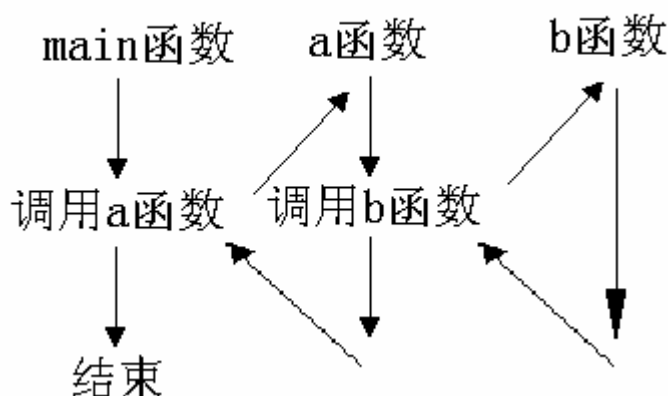
其中第一，二行对 str 函数和 f 函数预先作了说明。因此在以后各函数中无须对

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。

8.5 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。

```
long f1(int p)
{
    int k;
    long r;
    long f2(int);
    k=p*p;
    r=f2(k);
    return r;
}
long f2(int q)
{
    long c=1;
    int i;
    for(i=1;i<=q;i++)
```

```
        c=c*i;
    return c;
}
main()
{
    int i;
    long s=0;
    for (i=2;i<=3;i++)
        s=s+f1(i);
    printf("\ns=%ld\n",s);
}
```



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。

8.6 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：

```
int f(int x)
{
    int y;
    z=f(y);
    return z;
}
```

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 8.5】用递归法计算 n!

用递归法计算 n! 可用下述公式表示：

$$\begin{aligned} n! &= 1 & (n=0, 1) \\ n \times (n-1)! & & (n>1) \end{aligned}$$

按公式可编程如下：

```
long ff(int n)
{
    long f;
    if(n<0) printf("n<0, input error");
    else if(n==0||n==1) f=1;
    else f=ff(n-1)*n;
    return(f);
}

main()
{
    int n;
    long y;
    printf("\ninput a inteager number:\n");
    scanf("%d",&n);
    y=ff(n);
    printf("%d!=%ld",n,y);
}
```



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 $24*5=120$ 。

例 8.5 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 8.6】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 5.4 所示。要把这 64 个圆盘从 A 针移动到 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:

```
move(int n, int x, int y, int z)
{
    if(n==1)
        printf("%c-->%c\n", x, z);
    else
    {
        move(n-1, x, z, y);
        printf("%c-->%c\n", x, z);
        move(n-1, y, x, z);
    }
}

main()
{
    int h;
    printf("\ninput number:\n");
    scanf("%d", &h);
    printf("the step to moving %2d disk(s):\n", h);
    move(h, 'a', 'b', 'c');
}
```



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 $x \rightarrow z$ 。如 $n \neq 1$ 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 $x \rightarrow z$; 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:

```
input number:
4
the step to moving 4 disk:
a→b
a→c
b→c
a→b
c→a
c→b
a→b
a→c
b→c
b→a
c→a
b→c
a→b
a→c
b→c
```

8.7 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。例 5.4 说明了这种情况。

【例 8.7】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:

```
void nzp(int v)
{
    if(v>0)
        printf("%d ",v);
    else
        printf("%d ",0);
}

main()
{
    int a[5],i;
    printf("input 5 numbers\n");
    for(i=0;i<5;i++)
        {scanf("%d",&a[i]);
         nzp(a[i]);}
}
```

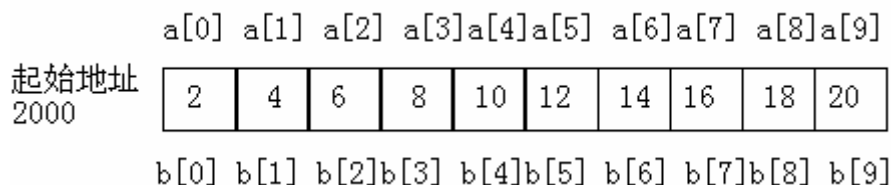


本程序中首先定义一个无返回值函数 `nzp`，并说明其形参 `v` 为整型变量。在函数体中根据 `v` 值输出相应的结果。在 `main` 函数中用一个 `for` 语句输入数组各元素，每输入一个就以该元素作实参调用一次 `nzp` 函数，即把 `a[i]` 的值传送给形参 `v`，供 `nzp` 函数使用。

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 8.8】数组 `a` 中存放了一个学生 5 门课程的成绩，求平均成绩。

```
float aver(float a[5])
{
    int i;
    float av,s=a[0];
    for(i=1;i<5;i++)
        s=s+a[i];
    av=s/5;
    return av;
}

void main()
```

```
{
    float sco[5], av;
    int i;
    printf("\ninput 5 scores:\n");
    for(i=0; i<5; i++)
        scanf("%f", &sco[i]);
    av=aver(sco);
    printf("average score is %5.2f", av);
}
```



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。为了说明这种情况，把例 5.4 改为例 5.6 的形式。

【例 8.9】题目同 8.7 例。改用数组名作函数参数。

```
void nzp(int a[5])
{
    int i;
    printf("\nvalues of array a are:\n");
    for(i=0; i<5; i++)
    {
        if(a[i]<0) a[i]=0;
        printf("%d ", a[i]);
    }
}

main()
{
    int b[5], i;
    printf("\ninput 5 numbers:\n");
    for(i=0; i<5; i++)
        scanf("%d", &b[i]);
    printf("initial values of array b are:\n");
    for(i=0; i<5; i++)
        printf("%d ", b[i]);
    nzp(b);
    printf("\nlast values of array b are:\n");
}
```

```
    for(i=0;i<5;i++)
        printf("%d ",b[i]);
}
```



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：

- 形参数组和实参数组的类型必须一致，否则将引起错误。
- 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 8.10】如把例 8.9 修改如下：

```
void nzp(int a[8])
{
    int i;
    printf("\nvalues of array aare:\n");
    for(i=0;i<8;i++)
    {
        if(a[i]<0)a[i]=0;
        printf("%d ",a[i]);
    }
}

main()
{
    int b[5],i;
    printf("\ninput 5 numbers:\n");
    for(i=0;i<5;i++)
        scanf("%d",&b[i]);
    printf("initial values of array b are:\n");
    for(i=0;i<5;i++)
        printf("%d ",b[i]);
    nzp(b);
    printf("\nlast values of array b are:\n");
    for(i=0;i<5;i++)
        printf("%d ",b[i]);
}
```



本程序与例 8.9 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:

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

或写为

```
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, 例 8.10 又可改为例 8.11 的形式。

【例 8.11】

```
void nzp(int a[], int n)
{
    int i;
    printf("\nvalues of array a are:\n");
    for(i=0; i<n; i++)
    {
        if(a[i]<0) a[i]=0;
        printf("%d ", a[i]);
    }
}

main()
{
    int b[5], i;
    printf("\ninput 5 numbers:\n");
    for(i=0; i<5; i++)
        scanf("%d", &b[i]);
    printf("initial values of array b are:\n");
    for(i=0; i<5; i++)
        printf("%d ", b[i]);
    nzp(b, 5);
    printf("\nlast values of array b are:\n");
    for(i=0; i<5; i++)
        printf("%d ", b[i]);
}
```



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。

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

或

```
int MA(int a[][10])。
```

8.8 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。

8.8.1 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：

```
int f1(int a)          /*函数 f1*/
{
    int b, c;
    .....
}
a, b, c 有效

int f2(int x)          /*函数 f2*/
{
    int y, z;
    .....
}
x, y, z 有效

main()
{
    int m, n;
    .....
}
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：

```
main()
{
    int s,a;
    .....
    {
        int b;
        s=a+b;
        .....          /*b 作用域*/
    }
    .....          /*s,a 作用域*/
}
```

【例 8.12】

```
main()
{
    int i=2, j=3, k;
    k=i+j;
    {
        int k=8;
        printf("%d\n", k);
    }
    printf("%d\n", k);
}
```



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。

8.8.2 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：

```
int a,b;          /*外部变量*/
void f1()          /*函数 f1*/
{
    .....
}
```

```
float x,y;      /*外部变量*/
int fz()        /*函数 fz*/
{
    .....
}
main()          /*主函数*/
{
    .....
}
```

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 8.13】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。

```
int s1, s2, s3;
int vs( int a, int b, int c)
{
    int v;
    v=a*b*c;
    s1=a*b;
    s2=b*c;
    s3=a*c;
    return v;
}
main()
{
    int v, l, w, h;
    printf("\ninput length,width and height\n");
    scanf("%d%d%d",&l,&w,&h);
    v=vs(l, w, h);
    printf("\nv=%d, s1=%d, s2=%d, s3=%d\n", v, s1, s2, s3);
}
```



【例 8.14】外部变量与局部变量同名。

```
int a=3,b=5;    /*a,b 为外部变量*/
max(int a, int b) /*a,b 为外部变量*/
{int c;
    c=a>b?a:b;
    return(c);
}
main()
{int a=8;
    printf("%d\n",max(a,b));
}
```



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。

8.9 变量的存储类别

8.9.1 动态存储方式与静态动态存储方式

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。

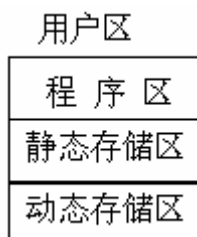
从另一个角度，从变量值存在的时间（即生存期）角度来分，可以分为静态存储方式和动态存储方式。

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。

8.9.2 auto 变量

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：

```
int f(int a)          /*定义 f 函数，a 为参数*/
```

```
{auto int b,c=3;    /*定义 b, c 自动变量*/  
.....  
}
```

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。

8.9.3 用 static 声明局部变量

有时希望函数中的局部变量的值在函数调用结束后不消失而保留原值, 这时就应该指定局部变量为“静态局部变量”, 用关键字 static 进行声明。

【例 8.15】考察静态局部变量的值。

```
f(int a)  
{auto b=0;  
  static c=3;  
  b=b+1;  
  c=c+1;  
  return(a+b+c);  
}  
main()  
{int a=2, i;  
  for(i=0; i<3; i++)  
    printf("%d", f(a));  
}
```



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 8.16】打印 1 到 5 的阶乘值。

```
int fac(int n)  
{static int f=1;  
  f=f*n;  
  return(f);  
}  
main()  
{int i;  
  for(i=1; i<=5; i++)
```

```
printf("%d!=%d\n", i, fac(i));  
}
```



8.9.4 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 8.17】使用寄存器变量。

```
int fac(int n)  
{register int i, f=1;  
 for(i=1; i<=n; i++)  
     f=f*i;  
 return(f);  
}  
  
main()  
{int i;  
 for(i=0; i<=5; i++)  
     printf("%d!=%d\n", i, fac(i));  
}
```



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。

8.9.5 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 8.18】用 `extern` 声明外部变量, 扩展程序文件中的作用域。

```
int max(int x, int y)  
{int z;  
 z=x>y?x:y;  
 return(z);  
}  
  
main()  
{extern A, B;
```

```
printf("%d\n", max(A, B));  
}  
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。

9 预处理命令

9.1 概述

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。

9.2 宏定义

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。

9.2.1 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：

```
#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：

```
#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 9.1】

```
#define M (y*y+3*y)
main() {
```

```

int s,y;
printf("input a number: ");
scanf("%d",&y);
s=3*M+4*M+5*M;
printf("s=%d\n",s);
}

```



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:

$$s=3*(y*y+3*y)+4*(y*y+3*y)+5*(y*y+3*y);$$

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:

```
#define M y*y+3*y
```

在宏展开时将得到下述语句:

$$s=3*y*y+3*y+4*y*y+3*y+5*y*y+3*y;$$

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:

```

#define PI 3.14159
main()
{
    .....
}
#undef PI
f1()
{
    .....
}

```

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 9.2】

```
#define OK 100
main()
{
    printf("OK");
    printf("\n");
}
```



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：

```
#define PI 3.1415926
#define S PI*y*y          /* PI 是已定义的宏名*/
```

对语句：

```
printf("%f", S);
```

在宏代换后变为：

```
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：

```
#define STU struct stu
```

在程序中可用 STU 作变量说明：

```
STU body[5], *p;
```

```
#define INTEGER int
```

在程序中即可用 INTEGER 作整型变量说明：

```
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：

```
#define PIN1 int *
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：

PIN1 a, b; 在宏代换后变成：

```
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：

```
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 9.3】中就采用了这种方法。

```
#define P printf
#define D "%d\n"
#define F "%f\n"
main() {
    int a=5, c=8, e=11;
    float b=3.8, d=9.7, f=21.08;
    P(D F, a, b);
    P(D F, c, d);
    P(D F, e, f);
}
```



9.2.2 带参宏定义

C 语言允许宏带有参数。在宏定义中的参数称为形式参数，在宏调用中的参数称为实际参数。

对带参数的宏，在调用中，不仅要宏展开，而且要用实参去代换形参。

带参宏定义的一般形式为：

#define 宏名(形参表) 字符串

在字符串中含有各个形参。

带参宏调用的一般形式为：

宏名(实参表)；

例如：

```
#define M(y) y*y+3*y      /*宏定义*/
.....
k=M(5);                  /*宏调用*/
.....
```

在宏调用时，用实参 5 去代替形参 y，经预处理宏展开后的语句为：

k=5*5+3*5

【例 9.4】

```
#define MAX(a, b) (a>b)?a:b
main() {
    int x, y, max;
    printf("input two numbers:  ");
    scanf("%d%d", &x, &y);
    max=MAX(x, y);
    printf("max=%d\n", max);
}
```



上例程序的第一行进行带参宏定义，用宏名 MAX 表示条件表达式 (a>b)?a:b，形参 a, b 均出现在条件表达式中。程序第七行 max=MAX(x, y) 为宏调用，实参 x, y，将代换形参 a, b。

宏展开后该语句为：

```
max=(x>y)?x:y;
```

用于计算 x, y 中的大数。

对于带参的宏定义有以下问题需要说明：

1. 带参宏定义中，宏名和形参表之间不能有空格出现。

例如把：

```
#define MAX(a,b) (a>b)?a:b
```

写为：

```
#define MAX (a,b) (a>b)?a:b
```

将被认为是无参宏定义，宏名 MAX 代表字符串 (a,b) (a>b)?a:b。宏展开时，宏调用语句：

```
max=MAX(x,y);
```

将变为：

```
max=(a,b) (a>b)?a:b(x,y);
```

这显然是错误的。

2. 在带参宏定义中，形式参数不分配内存单元，因此不必作类型定义。而宏调用中的实参有具体的值。要用它们去代换形参，因此必须作类型说明。这是与函数中的情况不同的。在函数中，形参和实参是两个不同的量，各有自己的作用域，调用时要把实参值赋予形参，进行“值传递”。而在带参宏中，只是符号代换，不存在值传递的问题。
3. 在宏定义中的形参是标识符，而宏调用中的实参可以是表达式。

【例 9.5】

```
#define SQ(y) (y)*(y)
```

```
main() {
```

```
    int a,sq;
```

```
    printf("input a number:  ");
```

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

```
    sq=SQ(a+1);
```

```
    printf("sq=%d\n",sq);
```

```
}
```



上例中第一行为宏定义，形参为 y。程序第七行宏调用中实参为 a+1，是一个表达式，在宏展开时，用 a+1 代换 y，再用 (y)*(y) 代换 SQ，得到如下语句：

```
sq=(a+1)*(a+1);
```

这与函数的调用是不同的，函数调用时要把实参表达式的值求出来再赋予形参。而宏代换中对实参表达式不作计算直接地照原样代换。

4. 在宏定义中，字符串内的形参通常要用括号括起来以避免出错。在上例中的宏定义中 (y)*(y) 表达式的 y 都用括号括起来，因此结果是正确的。如果去掉括号，把程序改为以下形式：

【例 9.6】

```
#define SQ(y) y*y
```

```
main() {
```

```
    int a,sq;
```

```
    printf("input a number:  ");
```

```
scanf("%d",&a);
sq=SQ(a+1);
printf("sq=%d\n",sq);
}
```



运行结果为：

```
input a number:3
sq=7
```

同样输入 3，但结果却是不一样的。问题在哪里呢？这是由于代换只作符号代换而不作其它处理而造成的。宏代换后将得到以下语句：

```
sq=a+1*a+1;
```

由于 a 为 3 故 sq 的值为 7。这显然与题意相违，因此参数两边的括号是不能少的。即使在参数两边加括号还是不够的，请看下面程序：

【例 9.7】

```
#define SQ(y) (y)*(y)
main() {
    int a,sq;
    printf("input a number:  ");
    scanf("%d",&a);
    sq=160/SQ(a+1);
    printf("sq=%d\n",sq);
}
```



本程序与前例相比，只把宏调用语句改为：

```
sq=160/SQ(a+1);
```

运行本程序如输入值仍为 3 时，希望结果为 10。但实际运行的结果如下：

```
input a number:3
sq=160
```

为什么会得这样的结果呢？分析宏调用语句，在宏代换之后变为：

```
sq=160/(a+1)*(a+1);
```

a 为 3 时，由于“/”和“*”运算符优先级和结合性相同，则先作 $160/(3+1)$ 得 40，再作 $40*(3+1)$ 最后得 160。为了得到正确答案应在宏定义中的整个字符串外加括号，程序修改如下：

【例 9.8】

```
#define SQ(y) ((y)*(y))
main() {
    int a,sq;
    printf("input a number:  ");
    scanf("%d",&a);
    sq=160/SQ(a+1);
    printf("sq=%d\n",sq);
}
```



以上讨论说明，对于宏定义不仅应在参数两侧加括号，也应在整个字符串外加括号。

5. 带参的宏和带参函数很相似，但有本质上的不同，除上面已谈到的各点外，把同一表达式用函数处理与用宏处理两者的结果有可能是不同的。

【例 9.9】

```
main() {
    int i=1;
    while(i<=5)
        printf("%d\n", SQ(i++));
}

SQ(int y)
{
    return((y)*(y));
}
```



【例 9.10】

```
#define SQ(y) ((y)*(y))

main() {
    int i=1;
    while(i<=5)
        printf("%d\n", SQ(i++));
}
```



在例 9.9 中函数名为 SQ，形参为 Y，函数体表达式为 ((y)*(y))。在例 9.10 中宏名为 SQ，形参也为 y，字符串表达式为 (y)*(y))。例 9.9 的函数调用为 SQ(i++)，例 9.10 的宏调用为 SQ(i++)，实参也是相同的。从输出结果来看，却大不相同。

分析如下：在例 9.9 中，函数调用是把实参 i 值传给形参 y 后自增 1。然后输出函数值。因而要循环 5 次。输出 1~5 的平方值。而在例 9.10 中宏调用时，只作代换。SQ(i++) 被代换为 ((i++)*(i++))。在第一次循环时，由于 i 等于 1，其计算过程为：表达式中前一个 i 初值为 1，然后 i 自增 1 变为 2，因此表达式中第 2 个 i 初值为 2，两相乘的结果也为 2，然后 i 值再自增 1，得 3。在第二次循环时，i 值已有初值为 3，因此表达式中前一个 i 为 3，后一个 i 为 4，乘积为 12，然后 i 再自增 1 变为 5。进入第三次循环，由于 i 值已为 5，所以这将是最后一次循环。计算表达式的值为 5*6 等于 30。i 值再自增 1 变为 6，不再满足循环条件，停止循环。

从以上分析可以看出函数调用和宏调用二者在形式上相似，在本质上是完全不同的。

6. 宏定义也可用来定义多个语句，在宏调用时，把这些语句又代换到源程序内。看下面的例子。

【例 9.11】

```
#define SSSV(s1, s2, s3, v) s1=1*w; s2=1*h; s3=w*h; v=w*1*h;

main() {
    int l=3, w=4, h=5, sa, sb, sc, vv;
```

```

SSSV(sa, sb, sc, vv);
printf("sa=%d\nsb=%d\nsc=%d\nvv=%d\n", sa, sb, sc, vv);
}

```



程序第一行为宏定义，用宏名 SSSV 表示 4 个赋值语句，4 个形参分别为 4 个赋值符左部的变量。在宏调用时，把 4 个语句展开并用实参代替形参。使计算结果送入实参之中。

9.3 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：

```
#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：

```
#include "stdio.h"
```

```
#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：

```
#include "stdio.h"
```

```
#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。

9.4 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：

```
#ifdef 标识符
```

```
程序段 1
```

```
#else
    程序段 2
#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)，本格式中的 `#else` 可以没有，即可以写为：

```
#ifdef 标识符
    程序段
#endif
```

【例 9.12】

```
#define NUM ok
main() {
    struct stu
    {
        int num;
        char *name;
        char sex;
        float score;
    } *ps;
    ps=(struct stu*)malloc(sizeof(struct stu));
    ps->num=102;
    ps->name="Zhang ping";
    ps->sex='M';
    ps->score=62.5;
    #ifdef NUM
    printf("Number=%d\nScore=%f\n", ps->num, ps->score);
    #else
    printf("Name=%s\nSex=%c\n", ps->name, ps->sex);
    #endif
    free(ps);
}
```



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：

```
#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：

```
#ifndef 标识符
    程序段 1
#else
```

程序段 2

#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：

#if 常量表达式

程序段 1

#else

程序段 2

#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 9.13】

```
#define R 1
main() {
    float c,r,s;
    printf("input a number: ");
    scanf("%f",&c);
    #if R
        r=3.14159*c*c;
        printf("area of round is: %f\n",r);
    #else
        s=c*c;
        printf("area of square is: %f\n",s);
    #endif
}
```



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。

9.5 本章小结

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。

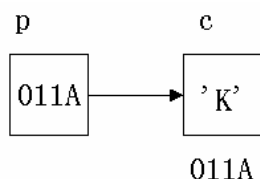
5. 文件包含是预处理的一个重要功能，它可用来把多个源文件连接成一个源文件进行编译，结果将生成一个目标文件。
6. 条件编译允许只编译源程序中满足条件的程序段，使生成的目标程序较短，从而减少了内存的开销并提高了程序的效率。
7. 使用预处理功能便于程序的修改、阅读、移植和调试，也便于实现模块化程序设计。

10 指针

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。

10.1 地址指针的基本概念

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，字符型量占 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（地址用十六进制数表示）。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。

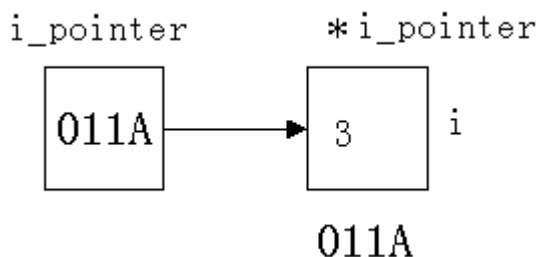
既然指针变量的值是一个地址，那么这个地址不仅可以是变量的地址，也可以是其它数据结构的地址。在一个指针变量中存放一个数组或一个函数的首地址有何意义呢？因为数组或函数都是连续存放的。通过访问指针变量取得了数组或函数的首地址，也就找到了该数组或函数。这样一来，凡是出现数组，函数的地方都可以用一个指针变量来表示，只要该指针变量中赋予数组或函数的首地址即可。这样做，将会使程序的概念十分清楚，程序本身也

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。

10.2 变量的指针和指向变量的指针变量

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：

```
i=3;
```

```
*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。

10.2.1 定义一个指针变量

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：

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：`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：

```
int *p2;          /*p2 是指向整型变量的指针变量*/
```

```
float *p3;        /*p3 是指向浮点变量的指针变量*/
```

```
char *p4;         /*p4 是指向字符变量的指针变量*/
```

需要注意的是，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。

10.2.2 指针变量的引用

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：

&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

```
int a;
int *p=&a;
```

- (2) 赋值语句的方法

```
int a;
int *p;
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：

```
int *p;
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。

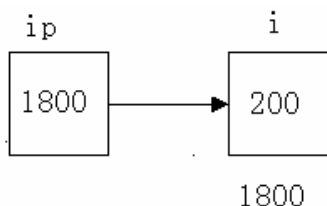
假设：

```
int i=200, x;
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`：

```
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：

```
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

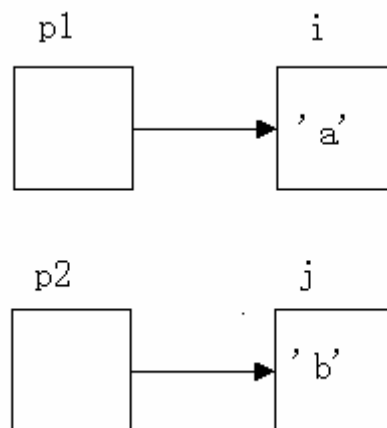
是地址为 1800 的存储区域(因为是整数,实际上是从 1800 开始的两个字节),它就是 i 所占用的存储区域,所以上面的赋值表达式等价于

```
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

```
int i, j, *p1, *p2;  
i='a';  
j='b';  
p1=&i;  
p2=&j;
```

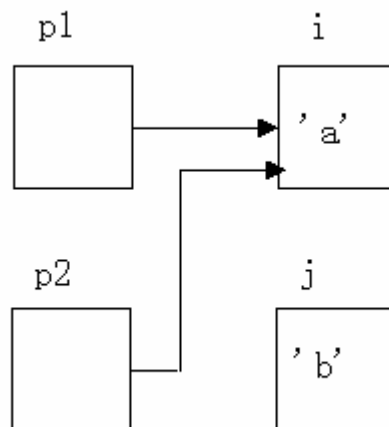
则建立如下图所示的联系:



这时赋值表达式:

```
p2=p1
```

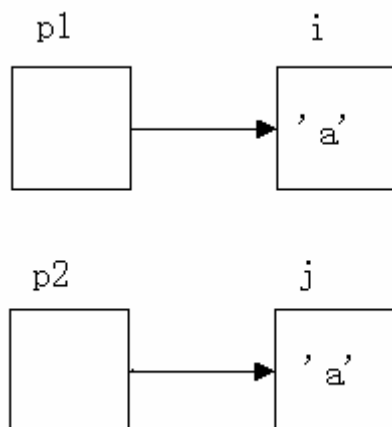
就使 p2 与 p1 指向同一对象 i, 此时 *p2 就等价于 i, 而不是 j, 图所示:



如果执行如下表达式:

```
*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

```
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:

```
y=*px+5; /*表示把 x 的内容加 5 并赋给 y*/
```

```
y=++*px; /**px 的内容加上 1 之后赋给 y, ++*px 相当于++(*px)*/
```

```
y=*px++; /*相当于 y=*px; px++*/
```

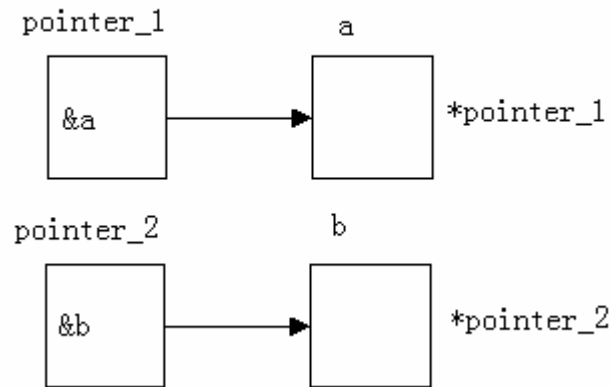
【例 10.1】

```
main()
{ int a,b;
  int *pointer_1, *pointer_2;
  a=100;b=10;
  pointer_1=&a;
  pointer_2=&b;
  printf("%d,%d\n", a,b);
  printf("%d,%d\n", *pointer_1, *pointer_2);
}
```



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

```
main()
{ int *p1,*p2,*p, a, b;
  scanf("%d,%d",&a,&b);
  p1=&a;p2=&b;
  if(a<b)
    {p=p1;p1=p2;p2=p;}
  printf("\na=%d, b=%d\n", a, b);
  printf("max=%d,min=%d\n", *p1, *p2);
}
```



10.2.3 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 10.3】题目同例 10.2，即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。

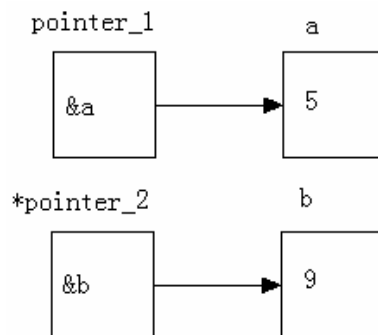
```
swap(int *p1,int *p2)
{int temp;
  temp=*p1;
  *p1=*p2;
```

```
*p2=temp;
}
main()
{
    int a,b;
    int *pointer_1,*pointer_2;
    scanf("%d,%d",&a,&b);
    pointer_1=&a;pointer_2=&b;
    if(a<b) swap(pointer_1,pointer_2);
    printf("\n%d,%d\n",a,b);
}
```

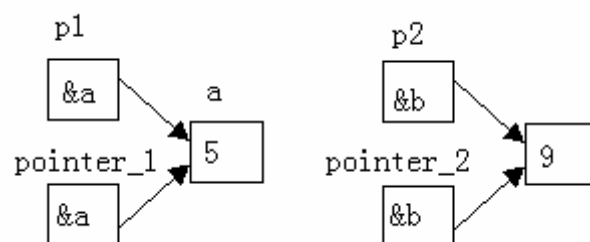


对程序的说明：

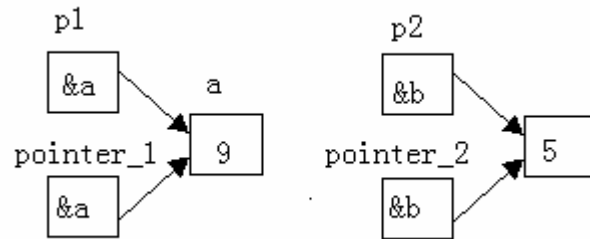
swap 是用户定义的函数，它的作用是交换两个变量（a 和 b）的值。swap 函数的形参 p1、p2 是指针变量。程序运行时，先执行 main 函数，输入 a 和 b 的值。然后将 a 和 b 的地址分别赋给指针变量 pointer_1 和 pointer_2，使 pointer_1 指向 a，pointer_2 指向 b。



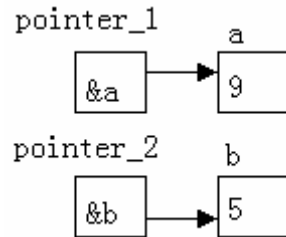
接着执行 if 语句，由于 $a < b$ ，因此执行 swap 函数。注意实参 pointer_1 和 pointer_2 是指针变量，在函数调用时，将实参变量的值传递给形参变量。采取的依然是“值传递”方式。因此虚实结合后形参 p1 的值为 &a，p2 的值为 &b。这时 p1 和 pointer_1 指向变量 a，p2 和 pointer_2 指向变量 b。



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：

```

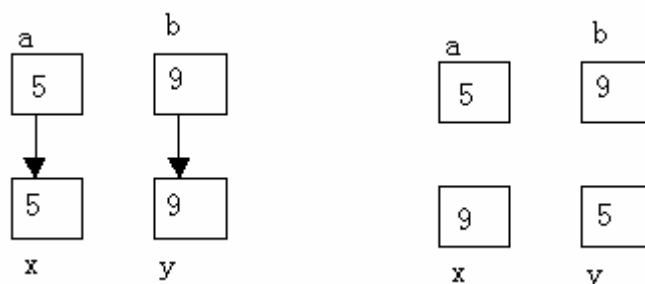
swap(int *p1, int *p2)
{
    int *temp;
    *temp = *p1;    /*此语句有问题*/
    *p1 = *p2;
    *p2 = *temp;
}
  
```

请考虑下面的函数能否实现实现 a 和 b 互换。

```

swap(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
  
```

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 10.4】 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。

```

swap(int *p1, int *p2)
{
    int *p;
    p = p1;
  
```

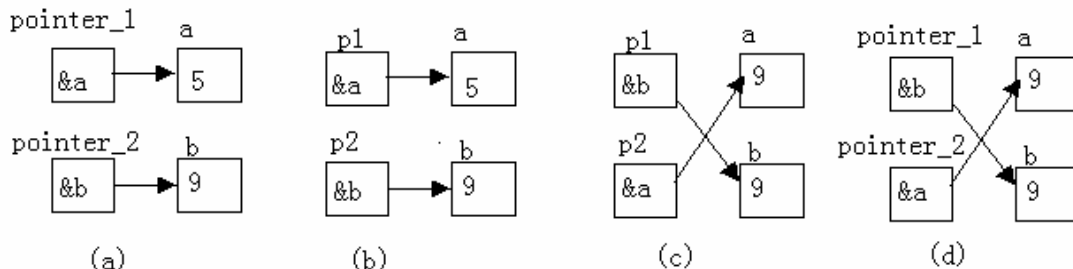
```

p1=p2;
p2=p;
}
main()
{
    int a,b;
    int *pointer_1,*pointer_2;
    scanf("%d,%d",&a,&b);
    pointer_1=&a;pointer_2=&b;
    if(a<b) swap(pointer_1,pointer_2);
    printf("\n%d,%d\n",*pointer_1,*pointer_2);
}

```



其中的问题在于不能实现如图所示的第四步 (d)。



【例 10.5】输入 a、b、c 3 个整数，按大小顺序输出。

```

swap(int *pt1,int *pt2)
{int temp;
 temp=*pt1;
 *pt1=*pt2;
 *pt2=temp;
}
exchange(int *q1,int *q2,int *q3)
{ if(*q1<*q2) swap(q1,q2);
  if(*q1<*q3) swap(q1,q3);
  if(*q2<*q3) swap(q2,q3);
}
main()
{
    int a,b,c,*p1,*p2,*p3;
    scanf("%d,%d,%d",&a,&b,&c);
    p1=&a;p2=&b; p3=&c;
    exchange(p1,p2,p3);
    printf("\n%d,%d,%d\n",a,b,c);
}

```



10.2.4 指针变量几个问题的进一步说明

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 10.6】

```
main() {  
    int a=5,*p=&a;  
    printf ("%d",*p);  
}
```



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:

```
int a,*pa;  
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:

```
int a,*pa=&a,*pb;  
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:

```
int a[5],*pa;  
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:

```
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:

```
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:

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

或用初始化赋值的方法写为:

```
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:

```
int (*pf)();  
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:

```
int a[5], *pa;  
pa=a; /*pa 指向数组 a, 也是指向 a[0]*/  
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:

```
pf1==pf2 表示 pf1 和 pf2 指向同一数组元素;  
pf1>pf2 表示 pf1 处于高地址位置;  
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:

```
#define NULL 0  
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 10.7】

```
main() {
    int a=10, b=20, s, t, *pa, *pb; /*说明 pa, pb 为整型指针变量*/
    pa=&a;                          /*给指针变量 pa 赋值, pa 指向变量 a*/
    pb=&b;                          /*给指针变量 pb 赋值, pb 指向变量 b*/
    s=*pa+*pb;                     /*求 a+b 之和, (*pa 就是 a, *pb 就是 b)*/
    t=*pa**pb;                     /*本行是求 a*b 之积*/
    printf("a=%d\nb=%d\na+b=%d\na*b=%d\n", a, b, a+b, a*b);
    printf("s=%d\nt=%d\n", s, t);
}
```

**【例 10.8】**

```
main() {
    int a, b, c, *pmax, *pmin;      /*pmax, pmin 为整型指针变量*/
    printf("input three numbers:\n"); /*输入提示*/
    scanf("%d%d%d", &a, &b, &c);    /*输入三个数字*/
    if(a>b) {                       /*如果第一个数字大于第二个数字... */
        pmax=&a;                    /*指针变量赋值*/
        pmin=&b;                    /*指针变量赋值*/
    } else {
        pmax=&b;                    /*指针变量赋值*/
        pmin=&a;                    /*指针变量赋值*/
    }
    if(c>*pmax) pmax=&c;             /*判断并赋值*/
    if(c<*pmin) pmin=&c;            /*判断并赋值*/
    printf("max=%d\nmin=%d\n", *pmax, *pmin); /*输出结果*/
}
```



10.3 数组指针和指向数组的指针变量

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。

10.3.1 指向数组元素的指针

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：

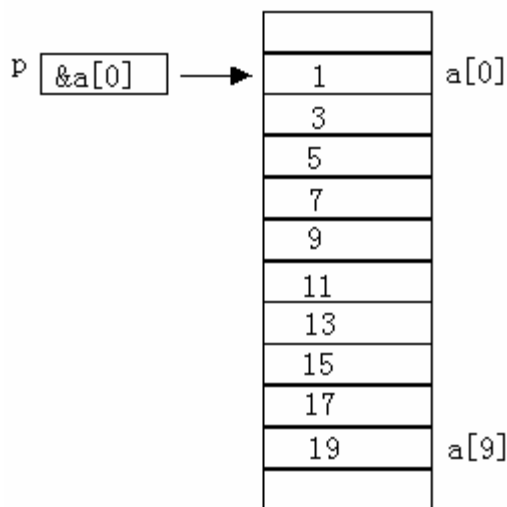
```
int a[10];    /*定义 a 为包含 10 个整型数据的数组*/
```

```
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：

```
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：

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

```
p=a;
```

在定义指针变量时可以赋给初值：

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

它等效于：

```
int *p;
```

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

当然定义时也可以写成：

```
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：

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。

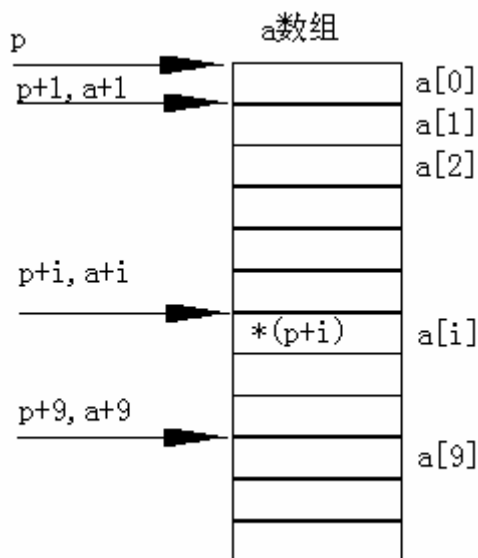
10.3.2 通过指针引用数组元素

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，其处值 $p=a$ 。

【例 10.9】输出数组中的全部元素。（下标法）

```
main() {
    int a[10], i;
    for(i=0; i<10; i++)
        a[i]=i;
    for(i=0; i<5; i++)
        printf("a[%d]=%d\n", i, a[i]);
}
```



【例 10.10】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）

```
main() {
    int a[10], i;
    for(i=0; i<10; i++)
        *(a+i)=i;
```

```
for(i=0;i<10;i++)
    printf("a[%d]=%d\n", i, *(a+i));
}
```



【例 10.11】输出数组中的全部元素。(用指针变量指向元素)

```
main() {
    int a[10], i, *p;
    p=a;
    for(i=0;i<10;i++)
        *(p+i)=i;
    for(i=0;i<10;i++)
        printf("a[%d]=%d\n", i, *(p+i));
}
```



【例 10.12】

```
main() {
    int a[10], i, *p=a;
    for(i=0;i<10;) {
        *p=i;
        printf("a[%d]=%d\n", i++, *p++);
    }
}
```



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 10.13】找出错误。

```
main() {
    int *p, i, a[10];
    p=a;
    for(i=0;i<10;i++)
        *p++=i;
    for(i=0;i<10;i++)
        printf("a[%d]=%d\n", i, *p++);
}
```



【例 10.14】改正。

```
main() {
    int *p, i, a[10];
```

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



- 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
 - `*(p--)` 相当于 `a[i--]`;
 - `*(++p)` 相当于 `a[++i]`;
 - `*(--p)` 相当于 `a[--i]`。

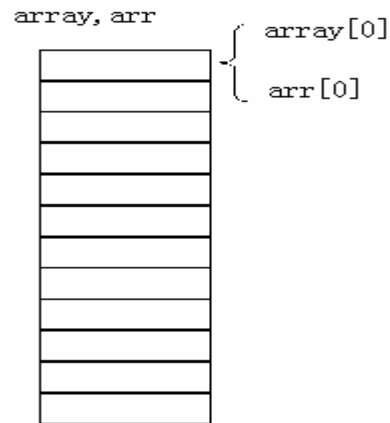
10.3.3 数组名作函数参数

数组名可以作函数的实参和形参。如:

```
main()
{int array[10];
    .....
    .....
    f(array, 10);
    .....
    .....
}

f(int arr[], int n);
{
    .....
    .....
}
```

`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。

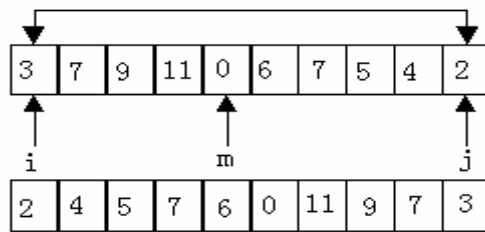
【例 10.15】

```
float aver(float *pa);
main() {
    float sco[5], av, *sp;
    int i;
    sp=sco;
    printf("\ninput 5 scores:\n");
    for(i=0;i<5;i++) scanf("%f",&sco[i]);
    av=aver(sp);
    printf("average score is %5.2f",av);
}
float aver(float *pa)
{
    int i;
    float av, s=0;
    for(i=0;i<5;i++) s=s+*pa++;
    av=s/5;
    return av;
}
```



【例 10.16】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：

```
void inv(int x[], int n)    /*形参x是数组名*/
{
    int temp, i, j, m=(n-1)/2;
    for(i=0; i<=m; i++)
    {
        j=n-1-i;
        temp=x[i]; x[i]=x[j]; x[j]=temp;
    }
    return;
}

main()
{
    int i, a[10]={3, 7, 9, 11, 0, 6, 7, 5, 4, 2};
    printf("The original array:\n");
    for(i=0; i<10; i++)
        printf("%d, ", a[i]);
    printf("\n");
    inv(a, 10);
    printf("The array has been inverted:\n");
    for(i=0; i<10; i++)
        printf("%d, ", a[i]);
    printf("\n");
}
```



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 10.17】对例 10.16 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：

```
void inv(int *x, int n)    /*形参x为指针变量*/
{
    int *p, temp, *i, *j, m=(n-1)/2;
    i=x; j=x+n-1; p=x+m;
    for(; i<=p; i++, j--)
        {temp=*i; *i=*j; *j=temp;}
    return;
}

main()
{
    int i, a[10]={3, 7, 9, 11, 0, 6, 7, 5, 4, 2};
    printf("The original array:\n");
```

```
for(i=0;i<10;i++)
    printf("%d,",a[i]);
printf("\n");
inv(a,10);
printf("The array has been inverted:\n");
for(i=0;i<10;i++)
    printf("%d,",a[i]);
printf("\n");
}
```



运行情况与前一程序相同。

【例 10.18】从 0 个数中找出其中最大值和最小值。

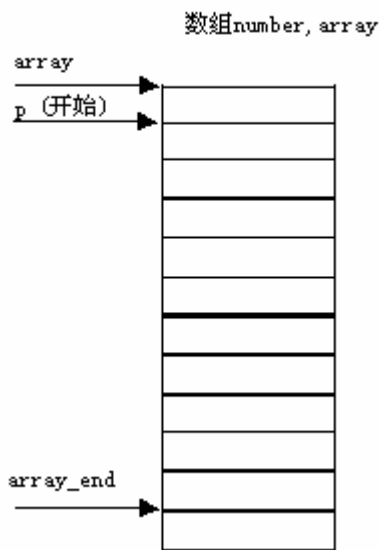
调用一个函数只能得到一个返回值，今用全局变量在函数之间“传递”数据。程序如下：

```
int max,min;        /*全局变量*/
void max_min_value(int array[],int n)
{int *p,*array_end;
 array_end=array+n;
 max=min=*array;
 for(p=array+1;p<array_end;p++)
     if(*p>max)max=*p;
     else if (*p<min)min=*p;
 return;
}
main()
{int i,number[10];
 printf("enter 10 integer umbers:\n");
 for(i=0;i<10;i++)
     scanf("%d",&number[i]);
 max_min_value(number,10);
 printf("\nmax=%d,min=%d\n",max,min);
}
```



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，因此在主函数中可以直接使用。
- 2) 函数 max_min_value 中的语句：
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。

【例 10.19】程序可改为：

```
int max,min;      /*全局变量*/
void max_min_value(int *array,int n)
{int *p,*array_end;
 array_end=array+n;
 max=min=*array;
 for(p=array+1;p<array_end;p++)
     if(*p>max)max=*p;
     else if (*p<min)min=*p;
 return;
}
main()
{int i,number[10],*p;
 p=number;        /*使p指向number数组*/

 printf("enter 10 integer umbers:\n");

 for(i=0;i<10;i++,p++)
     scanf("%d",p);
 p=number;
 max_min_value(p,10);
 printf("\nmax=%d,min=%d\n",max,min);
}
```



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。

```

main()                                f(int x[], int n)
{int a[10];                          {
    .....                            .....
    f(a, 10)                         }
    .....
}

```

a 和 x 指的是同一组数组。

2) 实用数组，形参用指针变量。

```

main()                                f(int *x, int n)
{int a[10];                          {
    .....                            .....
    f(a, 10)                         }
    .....
}

```

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 10.20】用实参指针变量改写将 n 个整数按相反顺序存放。

```

void inv(int *x, int n)
{int *p, m, temp, *i, *j;
  m=(n-1)/2;
  i=x; j=x+n-1; p=x+m;
  for(; i<=p; i++, j--)
    {temp=*i; *i=*j; *j=temp;}
  return;
}

main()
{int i, arr[10]={3, 7, 9, 11, 0, 6, 7, 5, 4, 2}, *p;
  p=arr;
  printf("The original array:\n");
  for(i=0; i<10; i++, p++)
    printf("%d, ", *p);
  printf("\n");
  p=arr;
  inv(p, 10);
  printf("The array has benn inverted:\n");
  for(p=arr; p<arr+10; p++)
    printf("%d, ", *p);
  printf("\n");
}

```



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 10.21】用选择法对 10 个整数排序。

```
main()
{int *p, i, a[10]={3, 7, 9, 11, 0, 6, 7, 5, 4, 2};
 printf("The original array:\n");
 for(i=0;i<10;i++)
    printf("%d,", a[i]);
 printf("\n");
 p=a;
 sort(p, 10);
 for(p=a, i=0;i<10;i++)
    {printf("%d  ", *p);p++;}
 printf("\n");
}

sort(int x[], int n)
{int i, j, k, t;
 for(i=0;i<n-1;i++)
    {k=i;
     for(j=i+1;j<n;j++)
        if(x[j]>x[k])k=j;
     if(k!=i)
        {t=x[i];x[i]=x[k];x[k]=t;}
    }
}
```



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。

10.3.4 指向多维数组的指针和指针变量

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

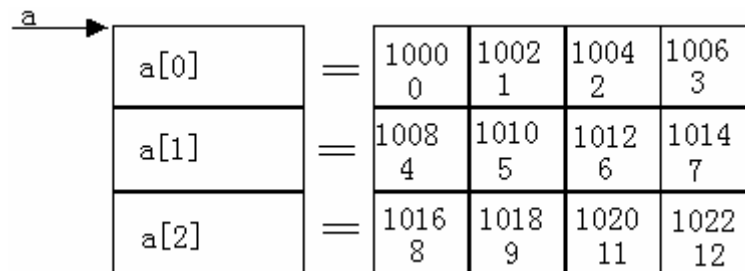
它的定义为：

```
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

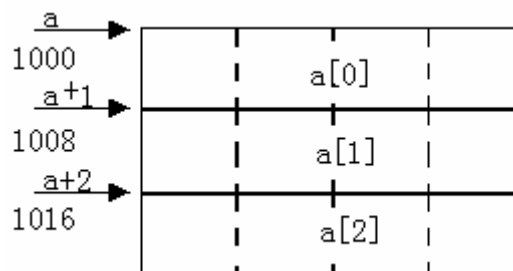
前面介绍过，C 语言允许把一个二维数组分解为多个一维数组来处理。因此数组 a 可分解为三个一维数组，即 $a[0]$ ， $a[1]$ ， $a[2]$ 。每一个一维数组又含有四个元素。



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	1002	1004	1006
a+1	0	1	2	3
a+2	1008	1010	1012	1014
	4	5	6	7
	1016	1018	1020	1022
	8	9	11	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 10.22】

```
main() {
    int a[3][4]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
    printf("%d, ", a);
    printf("%d, ", *a);
    printf("%d, ", a[0]);
    printf("%d, ", &a[0]);
    printf("%d\n", &a[0][0]);
    printf("%d, ", a+1);
    printf("%d, ", *(a+1));
    printf("%d, ", a[1]);
    printf("%d, ", &a[1]);
    printf("%d\n", &a[1][0]);
    printf("%d, ", a+2);
    printf("%d, ", *(a+2));
    printf("%d, ", a[2]);
    printf("%d, ", &a[2]);
    printf("%d\n", &a[2][0]);
    printf("%d, ", a[1]+1);
    printf("%d\n", *(a+1)+1);
    printf("%d, %d\n", *(a[1]+1), *((a+1)+1));
}
```



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：

```
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：

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 10.23】

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



10.4 字符串的指针指向字符串的指针变量

10.4.1 字符串的表示形式

在 C 语言中，可以用两种方法访问一个字符串。

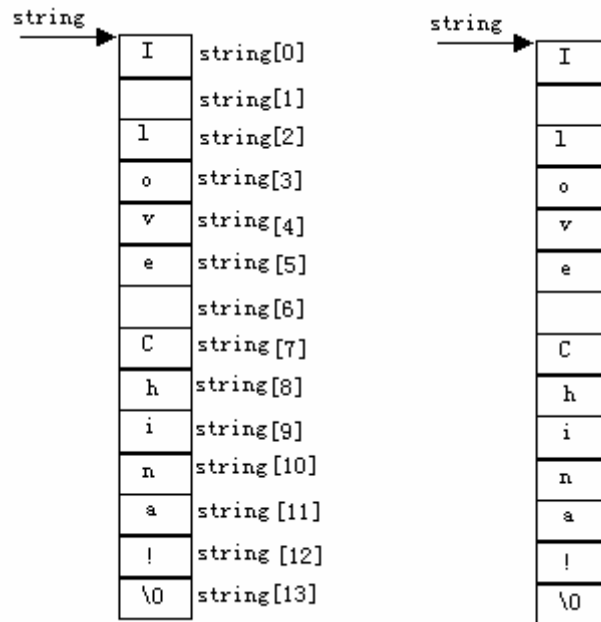
- 1) 用字符数组存放一个字符串，然后输出该字符串。

【例 10.24】

```
main() {  
    char string[]="I love China!";  
    printf("%s\n", string);  
}
```



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 10.25】

```
main() {
    char *string="I love China!";
    printf("%s\n", string);
}
```



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：

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

表示 p 是一个指向字符变量 c 的指针变量。

而：

```
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：

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

等效于：

```
char *ps;
```

```
ps="C Language";
```

【例 10.26】输出字符串中 n 个字符后的所有字符。

```
main() {
    char *ps="this is a book";
    int n=10;
    ps=ps+n;
```

```
printf("%s\n", ps);
}
```



运行结果为：

```
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 10.27】 在输入的字符串中查找有无‘k’字符。

```
main() {
    char st[20], *ps;
    int i;
    printf("input a string:\n");
    ps=st;
    scanf("%s", ps);
    for(i=0; ps[i]!='\0'; i++)
        if(ps[i]=='k') {
            printf("there is a 'k' in the string\n");
            break;
        }
    if(ps[i]=='\0') printf("There is no 'k' in the string\n");
}
```



【例 10.28】 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。

```
main() {
    static int a[3][4]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
    char *PF;
    PF="%d, %d, %d, %d, %d\n";
    printf(PF, a, *a, a[0], &a[0], &a[0][0]);
    printf(PF, a+1, *(a+1), a[1], &a[1], &a[1][0]);
    printf(PF, a+2, *(a+2), a[2], &a[2], &a[2][0]);
    printf("%d, %d\n", a[1]+1, *(a+1)+1);
    printf("%d, %d\n", *(a[1]+1), (*(a+1)+1));
}
```



【例 10.29】 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 cpystr 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。

```
cpystr(char *pss, char *pds) {
    while((*pds=*pss)!='\0') {
```

```
        pds++;
        pss++; }
}

main() {
    char *pa="CHINA", b[10], *pb;
    pb=b;
    cpystr(pa, pb);
    printf("string a=%s\nstring b=%s\n", pa, pb);
}
```



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 cprstr 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 cprstr 函数中均可使用这些字符串。也可以把 cprstr 函数简化为以下形式：

```
cprstr(char *pss, char*pds)
{while ((*pds++=*pss++)!='\0');}
```

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：

```
cprstr (char *pss, char *pds)
{while (*pds++=*pss++);}
```

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 10.30】简化后的程序如下所示。

```
cpystr(char *pss, char *pds) {
    while(*pds++=*pss++);
}

main() {
    char *pa="CHINA", b[10], *pb;
    pb=b;
    cpystr(pa, pb);
    printf("string a=%s\nstring b=%s\n", pa, pb);
}
```



10.4.2 使用字符串指针变量与字符数组的区别

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

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

可以写为：

```
char *ps;
```

```
ps="C Language";
```

而对数组方式：

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

不能写为：

```
char st[20];
```

```
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，

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

或者

```
char *ps;
```

```
ps="C Language";
```

都是合法的。

10.5 函数指针变量

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：

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：

```
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 10.31】本例用来说明用指针形式实现对函数调用的方法。

```
int max(int a, int b) {  
    if(a>b) return a;  
    else return b;  
}
```

```
main() {  
    int max(int a, int b);  
    int (*pmax)();
```

```
int x, y, z;
pmax=max;
printf("input two numbers:\n");
scanf("%d%d", &x, &y);
z=(*pmax)(x, y);
printf("maxmum=%d", z);
}
```



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：

(*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。

10.6 指针型函数

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：

```
类型说明符 *函数名(形参表)
{
    ..... /*函数体*/
}
```

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：

```
int *ap(int x, int y)
{
    ..... /*函数体*/
}
```

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 10.32】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。

```
main() {
    int i;
    char *day_name(int n);
    printf("input Day No:\n");
```

```
scanf("%d",&i);
if(i<0) exit(1);
printf("Day No:%2d-->%s\n", i, day_name(i));
}
char *day_name(int n){
    static char *name[]={ "Illegal day",
                           "Monday",
                           "Tuesday",
                           "Wednesday",
                           "Thursday",
                           "Friday",
                           "Saturday",
                           "Sunday"};
    return((n<1||n>7) ? name[0] : name[n]);
}
```



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。

10.7 指针数组和指向指针的指针

10.7.1 指针数组的概念

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：

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:

```
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 10.33】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。

```
main() {
    int a[3][3]={1, 2, 3, 4, 5, 6, 7, 8, 9};
    int *pa[3]={a[0], a[1], a[2]};
    int *p=a[0];
    int i;
    for(i=0; i<3; i++)
        printf("%d, %d, %d\n", a[i][2-i], *a[i], *((a+i)+i));
    for(i=0; i<3; i++)
        printf("%d, %d, %d\n", *pa[i], p[i], *(p+i));
}
```



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:

```
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。

```
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:

```
char *name[]={ "Illegal day",
                "Monday",
                "Tuesday",
                "Wednesday",
                "Thursday",
                "Friday",
                "Saturday",
                "Sunday"};
```

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".....。

指针数组也可以用作函数参数。

【例 10.34】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。

```
main() {
    static char *name[]={ "Illegal day",
                           "Monday",
                           "Tuesday",
                           "Wednesday",
                           "Thursday",
                           "Friday",
                           "Saturday",
                           "Sunday"};

    char *ps;
    int i;
    char *day_name(char *name[], int n);
    printf("input Day No:\n");
    scanf("%d",&i);
    if(i<0) exit(1);
    ps=day_name(name, i);
    printf("Day No:%2d-->%s\n", i, ps);
}

char *day_name(char *name[], int n)
{
    char *pp1,*pp2;
    pp1=*name;
    pp2=(name+n);
    return((n<1 || n>7)? pp1:pp2);
}
```



【例 10.35】输入 5 个国名并按字母顺序排列后输出。现编程如下：

```
#include "string.h"

main() {
    void sort(char *name[], int n);
    void print(char *name[], int n);
    static char *name[]={ "CHINA", "AMERICA", "AUSTRALIA",
                           "FRANCE", "GERMAN"};

    int n=5;
    sort(name, n);
```

```
    print(name, n);
}

void sort(char *name[], int n) {
    char *pt;
    int i, j, k;
    for(i=0; i<n-1; i++) {
        k=i;
        for(j=i+1; j<n; j++)
            if(strcmp(name[k], name[j])>0) k=j;
        if(k!=i) {
            pt=name[i];
            name[i]=name[k];
            name[k]=pt;
        }
    }
}

void print(char *name[], int n) {
    int i;
    for (i=0; i<n; i++) printf("%s\n", name[i]);
}
```



说明:

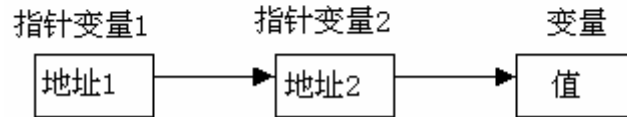
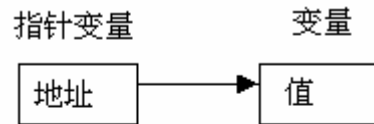
在以前的例子中采用了普通的排序方法, 逐个比较之后交换字符串的位置。交换字符串的物理位置是通过字符串复制函数完成的。反复的交换将使程序执行的速度很慢, 同时由于各字符串(国名)的长度不同, 又增加了存储管理的负担。用指针数组能很好地解决这些问题。把所有的字符串存放在一个数组中, 把这些字符串的首地址放在一个指针数组中, 当需要交换两个字符串时, 只须交换指针数组相应两元素的内容(地址)即可, 而不必交换字符串本身。

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。

10.7.2 指向指针的指针

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”。



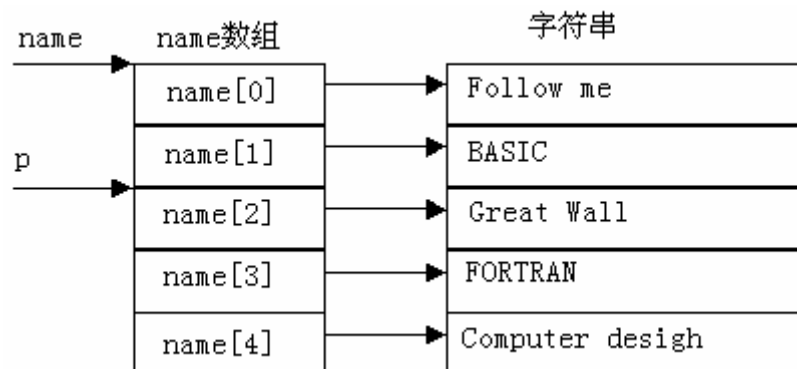
从下图可以看到，`name` 是一个指针数组，它的每一个元素是一个指针型数据，其值为地址。`Name` 是一个数据，它的每一个元素都有相应的地址。数组名 `name` 代表该指针数组的首地址。`name+1` 是 `mane[i]` 的地址。`name+1` 就是指向指针型数据的指针（地址）。还可以设置一个指针变量 `p`，使它指向指针数组元素。`P` 就是指向指针型数据的指针变量。

怎样定义一个指向指针型数据的指针变量呢？如下：

```
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：

```
p=name+2;
printf("%o\n", *p);
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 10.36】使用指向指针的指针。

```
main()
{char *name[]={"Follow me", "BASIC", "Great Wall", "FORTRAN", "Computer desighn"};
  char **p;
  int i;
  for(i=0; i<5; i++)
  {p=name+i;
```

```
    printf("%s\n", *p);  
}  
}
```



说明:

p 是指向指针的指针变量。

【例 10.37】一个指针数组的元素指向数据的简单例子。

```
main()  
{static int a[5]={1,3,5,7,9};  
  int *num[5]={&a[0],&a[1],&a[2],&a[3],&a[4]};  
  int **p,i;  
  p=num;  
  for(i=0;i<5;i++)  
    {printf("%d\t",**p);p++;}  
}
```



说明:

指针数组的元素只能存放地址。

10.7.3 main 函数的参数

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:

```
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:

```
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:

```
C:\>可执行文件名 参数 参数.....;
```

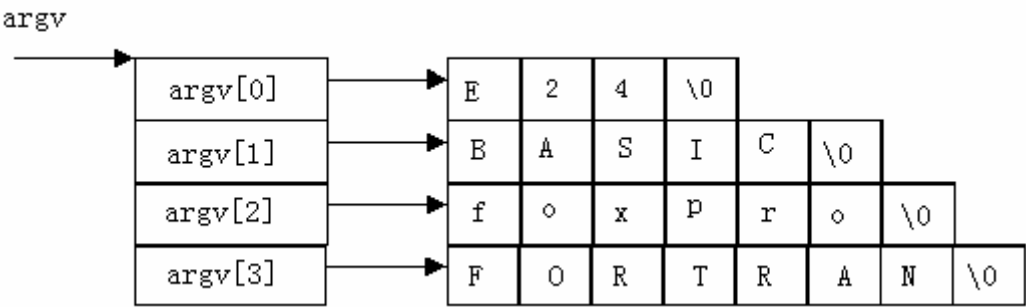
但是应该特别注意的是, main 的两个形参和命令行中的参数在位置上不是一一对应的。因为, main 的形参只有二个, 而命令行中的参数个数原则上未加限制。argc 参数表示了命令行中参数的个数(注意: 文件名本身也算一个参数), argv 的值是在输入命令行时由系统按实际参数的个数自动赋予的。

例如有命令行为:

```
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 10.38】

```
main(int argc, char *argv) {  
    while(argc-->1)  
        printf("%s\n", *++argv);  
}
```



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：

```
C:\>a:e24 BASIC foxpro FORTRAN
```

则运行结果为：

```
BASIC  
foxpro  
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。

10.8 有关指针的数据类型和指针运算的小结

10.8.1 有关指针的数据类型的小结

定义	含 义
int i;	定义整型变量 i
int *p	p 为指向整型数据的指针变量
int a[n];	定义整型数组 a，它有 n 个元素
int *p[n];	定义指针数组 p，它由 n 个指向整型数据的指针元素组成
int (*p)[n];	p 为指向含 n 个元素的一维数组的指针变量
int f();	f 为带回整型函数值的函数
int *p();	p 为带回一个指针的函数，该指针指向整型数据
int (*p)();	p 为指向函数的指针，该函数返回一个整型值
int **p;	p 是一个指针变量，它指向一个指向整型数据的指针变量

10.8.2 指针运算的小结

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。

p=&a; (将变量 a 的地址赋给 p)

p=array; (将数组 array 的首地址赋给 p)

p=&array[i]; (将数组 array 第 i 个元素的地址赋给 p)

p=max; (max 为已定义的函数，将 max 的入口地址赋给 p)

p1=p2; (p1 和 p2 都是指针变量，将 p2 的值赋给 p1)

注意：不能如下：

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：

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。

10.8.3 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。

11 结构体与共用体

11.1 定义一个结构的一般形式

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：

```
struct 结构名  
{成员表列};
```

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：

```
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：

```
struct stu  
{  
    int num;  
    char name[20];  
    char sex;  
    float score;  
};
```

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。

11.2 结构类型变量的说明

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：

```
struct stu  
{  
    int num;
```

```

    char name[20];
    char sex;
    float score;
};
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：

```

#define STU struct stu
STU
{
    int num;
    char name[20];
    char sex;
    float score;
};
STU boy1, boy2;

```

2. 在定义结构类型的同时说明结构变量。

例如：

```

struct stu
{
    int num;
    char name[20];
    char sex;
    float score;
}boy1, boy2;

```

这种形式的说明的一般形式为：

```

struct 结构名
{
    成员表列
} 变量名表列;

```

3. 直接说明结构变量。

例如：

```

struct
{
    int num;
    char name[20];
    char sex;
    float score;
}boy1, boy2;

```

这种形式的说明的一般形式为：

```

struct
{
    成员表列
}

```

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:

```
struct date
{
    int month;
    int day;
    int year;
};
struct{
    int num;
    char name[20];
    char sex;
    struct date birthday;
    float score;
}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。

11.3 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:

结构变量名. 成员名

例如:

boy1.num 即第一个人的学号
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:

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。

11.4 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 11.1】给结构变量赋值并输出其值。

```
main()
{
    struct stu
    {
        int num;
        char *name;
        char sex;
        float score;
    } boy1, boy2;
    boy1.num=102;
    boy1.name="Zhang ping";
    printf("input sex and score\n");
    scanf("%c %f", &boy1.sex, &boy1.score);
    boy2=boy1;
    printf("Number=%d\nName=%s\n", boy2.num, boy2.name);
    printf("Sex=%c\nScore=%f\n", boy2.sex, boy2.score);
}
```



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。

11.5 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 11.2】对结构变量初始化。

```
main()
{
    struct stu    /*定义结构*/
    {
        int num;
        char *name;
        char sex;
        float score;
    } boy2, boy1={102, "Zhang ping", 'M', 78.5};
    boy2=boy1;
```

```
printf("Number=%d\nName=%s\n", boy2. num, boy2. name);
printf("Sex=%c\nScore=%f\n", boy2. sex, boy2. score);
}
```



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。

11.6 结构数组的定义

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：

```
struct stu
{
    int num;
    char *name;
    char sex;
    float score;
}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：

```
struct stu
{
    int num;
    char *name;
    char sex;
    float score;
}boy[5]={
    {101, "Li ping", "M", 45},
    {102, "Zhang ping", "M", 62.5},
    {103, "He fang", "F", 92.5},
    {104, "Cheng ling", "F", 87},
    {105, "Wang ming", "M", 58};
}
```

当对全部元素作初始化赋值时，也可不给出数组长度。

【例 11.3】计算学生的平均成绩和不及格的人数。

```
struct stu
{
    int num;
    char *name;
```

```

    char sex;
    float score;
} boy[5]={
    {101, "Li ping", 'M', 45},
    {102, "Zhang ping", 'M', 62.5},
    {103, "He fang", 'F', 92.5},
    {104, "Cheng ling", 'F', 87},
    {105, "Wang ming", 'M', 58},
};

main()
{
    int i, c=0;
    float ave, s=0;
    for(i=0; i<5; i++)
    {
        s+=boy[i].score;
        if(boy[i].score<60) c+=1;
    }
    printf("s=%f\n", s);
    ave=s/5;
    printf("average=%f\ncount=%d\n", ave, c);
}

```



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 11.4】建立同学通讯录

```

#include "stdio.h"
#define NUM 3
struct mem
{
    char name[20];
    char phone[10];
};

main()
{
    struct mem man[NUM];
    int i;
    for(i=0; i<NUM; i++)
    {
        printf("input name:\n");
        gets(man[i].name);
        printf("input phone:\n");
    }
}

```



11.7

11.7.

—

倒去

struct 结构名 *结构指针变量名

当然

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。

是正

是错误的。

其以
式立

结构指针变量->成员名

或者：

```
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 11.5】

```
struct stu
{
    int num;
    char *name;
    char sex;
    float score;
} boy1={102, "Zhang ping", 'M', 78.5}, *pstu;

main()
{
    pstu=&boy1;
    printf("Number=%d\nName=%s\n", boy1.num, boy1.name);
    printf("Sex=%c\nScore=%f\n\n", boy1.sex, boy1.score);
    printf("Number=%d\nName=%s\n", (*pstu).num, (*pstu).name);
    printf("Sex=%c\nScore=%f\n\n", (*pstu).sex, (*pstu).score);
    printf("Number=%d\nName=%s\n", pstu->num, pstu->name);
    printf("Sex=%c\nScore=%f\n\n", pstu->sex, pstu->score);
}
```



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:

结构变量. 成员名

(*结构指针变量). 成员名

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。

11.7.2 指向结构数组的指针

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 11.6】用指针变量输出结构数组。

```
struct stu
{
    int num;
```

```

    char *name;
    char sex;
    float score;
}boy[5]={
    {101,"Zhou ping",'M',45},
    {102,"Zhang ping",'M',62.5},
    {103,"Liou fang",'F',92.5},
    {104,"Cheng ling",'F',87},
    {105,"Wang ming",'M',58},
};

main()
{
    struct stu *ps;
    printf("No\tName\t\t\tSex\tScore\t\n");
    for(ps=boy;ps<boy+5;ps++)
        printf("%d\t%s\t\t%c\t%f\t\n",ps->num,ps->name,ps->sex,ps->score);
}

```



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。

```
ps=&boy[1].sex;
```

而只能是：

```
ps=boy; (赋予数组首地址)
```

或者是：

```
ps=&boy[0]; (赋予 0 号元素首地址)
```

11.7.3 结构指针变量作函数参数

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 11.7】 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。

```

struct stu
{
    int num;
    char *name;
    char sex;
    float score;}boy[5]={

```

```

        {101, "Li ping", 'M', 45},
        {102, "Zhang ping", 'M', 62.5},
        {103, "He fang", 'F', 92.5},
        {104, "Cheng ling", 'F', 87},
        {105, "Wang ming", 'M', 58},
    };

main()
{
    struct stu *ps;
    void ave(struct stu *ps);
    ps=boy;
    ave(ps);
}

void ave(struct stu *ps)
{
    int c=0, i;
    float ave, s=0;
    for(i=0; i<5; i++, ps++)
    {
        s+=ps->score;
        if(ps->score<60) c+=1;
    }
    printf("s=%f\n", s);
    ave=s/5;
    printf("average=%f\ncount=%d\n", ave, c);
}

```



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。

11.8 动态存储分配

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：

```

int n;
scanf("%d", &n);
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：

(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：

```
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：

(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：

```
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。

2. 释放内存空间函数 free

调用形式：

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 11.8】分配一块区域，输入一个学生数据。

```
main()
{
    struct stu
    {
        int num;
        char *name;
        char sex;
        float score;
    } *ps;
    ps=(struct stu*)malloc(sizeof(struct stu));
```

```

ps->num=102;
ps->name="Zhang ping";
ps->sex='M';
ps->score=62.5;
printf("Number=%d\nName=%s\n", ps->num, ps->name);
printf("Sex=%c\nScore=%f\n", ps->sex, ps->score);
free(ps);
}

```



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。

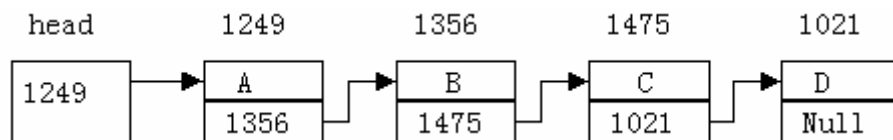
11.9 链表的概念

在例 7.8 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：

```

struct stu
{ int num;
  int score;

```

```

    struct stu *next;
}

```

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 11.9】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：

```

#define NULL 0
#define TYPE struct stu
#define LEN sizeof (struct stu)

struct stu
{
    int num;
    int age;
    struct stu *next;
};

TYPE *creat(int n)
{
    struct stu *head, *pf, *pb;
    int i;
    for(i=0; i<n; i++)
    {
        pb=(TYPE*) malloc (LEN);
        printf("input Number and Age\n");
        scanf("%d%d", &pb->num, &pb->age);
        if(i==0)
            pf=head=pb;
        else pf->next=pb;
        pb->next=NULL;
        pf=pb;
    }
    return (head);
}

```

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。

11.10 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。

11.10.1 枚举类型的定义和枚举变量的说明

1. 枚举的定义枚举类型定义的一般形式为：

```
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：

```
enum weekday { sun, mon, tue, wed, thu, fri, sat };
```

```
enum weekday a, b, c;
```

或者为：

```
enum weekday { sun, mon, tue, wed, thu, fri, sat } a, b, c;
```

或者为：

```
enum { sun, mon, tue, wed, thu, fri, sat } a, b, c;
```

11.10.2 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：

```
sun=5;
```

```
mon=2;
```

```
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 11.10】

```
main() {
```

```
enum weekday
{ sun, mon, tue, wed, thu, fri, sat } a, b, c;
a=sun;
b=mon;
c=tue;
printf("%d, %d, %d", a, b, c);
}
```



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:

```
a=sum;
b=mon;
```

是正确的。而:

```
a=0;
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:

```
a=(enum weekday)2;
```

其意义是将顺序号为 2 的枚举元素赋予枚举变量 a，相当于:

```
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 11.11】

```
main() {
    enum body
    { a, b, c, d } month[31], j;
    int i;
    j=a;
    for(i=1; i<=30; i++) {
        month[i]=j;
        j++;
        if (j>d) j=a;
    }
    for(i=1; i<=30; i++) {
        switch(month[i])
        {
            case a:printf(" %2d %c\t", i, 'a'); break;
            case b:printf(" %2d %c\t", i, 'b'); break;
            case c:printf(" %2d %c\t", i, 'c'); break;
            case d:printf(" %2d %c\t", i, 'd'); break;
            default:break;
        }
    }
    printf("\n");
}
```



11.11 类型定义符 typedef

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：

```
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：

```
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：

```
INTEGER a, b;
```

它等效于：

```
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：

```
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：

```
NAME a1, a2, s1, s2;
```

完全等效于：

```
char a1[20], a2[20], s1[20], s2[20]
```

又如：

```
typedef struct stu
{ char name[20];
  int age;
  char sex;
} STU;
```

定义 STU 表示 stu 的结构类型，然后可用 STU 来说明结构变量：

```
STU body1, body2;
```

typedef 定义的一般形式为：

```
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。

12 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。

12.1 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

12.1.1 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:

00001001	(9 的二进制补码)
&00000101	(5 的二进制补码)
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 12.1】

```
main() {  
    int a=9, b=5, c;  
    c=a&b;  
    printf("a=%d\nb=%d\nc=%d\n", a, b, c);  
}
```



12.1.2 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：

```
00001001
|00000101
00001101      (十进制为 13) 可见 9|5=13
```

【例 12.2】

```
main() {
    int a=9, b=5, c;
    c=a|b;
    printf("a=%d\nb=%d\nc=%d\n", a, b, c);
}
```



12.1.3 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：

```
00001001
^00000101
00001100      (十进制为 12)
```

【例 12.3】

```
main() {
    int a=9;
    a=a^5;
    printf("a=%d\n", a);
}
```



12.1.4 求反运算

求反运算符“~”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位按位求反。

例如~9 的运算为：

~(0000000000001001) 结果为：111111111110110

12.1.5 左移运算

左移运算符“<<”是双目运算符。其功能把“<<”左边的运算数的各二进位全部左移若干位，由“<<”右边的数指定移动的位数，高位丢弃，低位补 0。

例如：

$a \ll 4$

指把 a 的各二进位向左移动 4 位。如 $a=00000011$ (十进制 3)，左移 4 位后为 00110000 (十进制 48)。

12.1.6 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位全部右移若干位，“ $\gg$ ”右边的数指定移动的位数。

例如：

设 $a=15$,

$a \gg 2$

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 12.4】

```
main() {
    unsigned a, b;
    printf("input a number:  ");
    scanf("%d", &a);
    b = a >> 5;
    b = b & 15;
    printf("a=%d\tb=%d\n", a, b);
}
```



请再看一例！

【例 12.5】

```
main() {
    char a = 'a', b = 'b';
    int p, c, d;
    p = a;
    p = (p << 8) | b;
    d = p & 0xff;
    c = (p & 0xff00) >> 8;
    printf("a=%d\nb=%d\nc=%d\nd=%d\n", a, b, c, d);
}
```



12.2 位域（位段）

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：

```
struct 位域结构名  
{ 位域列表 };
```

其中位域列表的形式为：

```
类型说明符 位域名: 位域长度
```

例如：

```
struct bs  
{  
    int a:8;  
    int b:2;  
    int c:6;  
};
```

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：

```
struct bs  
{  
    int a:8;  
    int b:2;  
    int c:6;  
}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：

```
struct bs  
{  
    unsigned a:4  
    unsigned :0      /*空域*/  
    unsigned b:4      /*从下一单元开始存放*/  
    unsigned c:4  
}
```

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:

```
struct k
{
    int a:1
    int :2          /*该 2 位不能使用*/
    int b:3
    int c:2
};
```

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:

位域变量名·位域名

位域允许用各种格式输出。

【例 12.6】

```
main() {
    struct bs
    {
        unsigned a:1;
        unsigned b:3;
        unsigned c:4;
    } bit,*pbit;
    bit.a=1;
    bit.b=7;
    bit.c=15;
    printf("%d,%d,%d\n",bit.a,bit.b,bit.c);
    pbit=&bit;
    pbit->a=0;
    pbit->b&=3;
    pbit->c|=1;
    printf("%d,%d,%d\n",pbit->a,pbit->b,pbit->c);
}
```



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:

`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:

`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。

12.3 本章小结

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。

13 文件

13.1 C 文件概述

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，这种文件在磁盘中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。

13.2 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：

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：

```
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。

13.3 文件的打开与关闭

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。

13.3.1 文件的打开(fopen 函数)

fopen 函数用来打开一个文件，其调用的一般形式为：

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：

```
FILE *fp;
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：

```
FILE *fphzk
fphzk=("c:\\hzk16","rb")
```

其意义是打开 C 驱动器磁盘的根目录下的文件 hzk16，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
"rt"	只读打开一个文本文件，只允许读数据
"wt"	只写打开或建立一个文本文件，只允许写数据
"at"	追加打开一个文本文件，并在文件末尾写数据
"rb"	只读打开一个二进制文件，只允许读数据
"wb"	只写打开或建立一个二进制文件，只允许写数据
"ab"	追加打开一个二进制文件，并在文件末尾写数据

"rt+"	读写打开一个文本文件，允许读和写
"wt+"	读写打开或建立一个文本文件，允许读写
"at+"	读写打开一个文本文件，允许读，或在文件末追加数据
"rb+"	读写打开一个二进制文件，允许读和写
"wb+"	读写打开或建立一个二进制文件，允许读和写
"ab+"	读写打开一个二进制文件，允许读，或在文件末追加数据

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：
 - r(read): 读
 - w(write): 写
 - a(append): 追加
 - t(text): 文本文件，可省略不写
 - b(binary): 二进制文件
 - +: 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)


```
if((fp=fopen("c:\\hzk16", "rb"))==NULL)
{
    printf("\nerror on open c:\\hzk16 file!");
    getch();
    exit(1);
}
```

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 hzk16 文件，则给出提示信息 "error on open c:\\ hzk16 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。

13.3.2 文件关闭函数（fclose 函数）

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：

fclose(文件指针);

例如：

```
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。

13.4 文件的读写

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。

13.4.1 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：

```
字符变量=fgetc(文件指针);
```

例如：

```
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：

```
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 13.1】读入文件 c1.doc，在屏幕上输出。

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

```
main()
```

```
{
```

```
FILE *fp;
```

```
char ch;
```

```
if((fp=fopen("d:\\jrzh\\example\\c1.txt","rt"))==NULL)
```

```

    {
        printf("\nCannot open file strike any key exit!");
        getch();
        exit(1);
    }
    ch=fgetc(fp);
    while(ch!=EOF)
    {
        putchar(ch);
        ch=fgetc(fp);
    }
    fclose(fp);
}

```



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`，以读文本文件方式打开文件“d:\\jrzh\\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：

`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：

```
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 13.2】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。

```

#include<stdio.h>
main()
{
    FILE *fp;
    char ch;
    if((fp=fopen("d:\\jrzh\\example\\string", "wt+"))==NULL)
    {
        printf("Cannot open file strike any key exit!");
    }
}

```

```

    getch();
    exit(1);
}
printf("input a string:\n");
ch=getchar();
while (ch!='\n')
{
    fputc(ch, fp);
    ch=getchar();
}
rewind(fp);
ch=fgetc(fp);
while(ch!=EOF)
{
    putchar(ch);
    ch=fgetc(fp);
}
printf("\n");
fclose(fp);
}

```



程序中第 6 行以读写文本文件方式打开文件 string。程序第 13 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 20 至 25 行用于读出文件中的一行内容。

【例 13.3】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。

```

#include<stdio.h>
main(int argc, char *argv[])
{
    FILE *fp1, *fp2;
    char ch;
    if(argc==1)
    {
        printf("have not enter file name strike any key exit");
        getch();
        exit(0);
    }
    if((fp1=fopen(argv[1], "rt"))==NULL)
    {
        printf("Cannot open %s\n", argv[1]);
        getch();
    }
}

```

```

        exit(1);
    }
    if(argc==2) fp2=stdout;
    else if((fp2=fopen(argv[2], "wt+"))==NULL)
    {
        printf("Cannot open %s\n", argv[1]);
        getch();
        exit(1);
    }
    while((ch=fgetc(fp1))!=EOF)
        fputc(ch, fp2);
    fclose(fp1);
    fclose(fp2);
}

```



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 25 行至 28 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。

13.4.2 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：

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：

```
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 13.4】从 string 文件中读入一个含 10 个字符的字符串。

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

```
main()
```

```

{
    FILE *fp;
    char str[11];
    if((fp=fopen("d:\jrzh\example\string", "rt"))==NULL)
    {
        printf("\nCannot open file strike any key exit!");
        getch();
        exit(1);
    }
}

```

```

}
fgets(str, 11, fp);
printf("\n%s\n", str);
fclose(fp);
}

```



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：

`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：

```
fputs("abcd", fp);
```

其意义是把字符串 “abcd” 写入 `fp` 所指的文件之中。

【例 13.5】在例 13.2 中建立的文件 `string` 中追加一个字符串。

```

#include<stdio.h>
main()
{
    FILE *fp;
    char ch, st[20];
    if((fp=fopen("string", "at+"))==NULL)
    {
        printf("Cannot open file strike any key exit!");
        getch();
        exit(1);
    }

    printf("input a string:\n");
    scanf("%s", st);
    fputs(st, fp);
    rewind(fp);
    ch=fgetc(fp);
    while(ch!=EOF)
    {
        putchar(ch);
        ch=fgetc(fp);
    }
    printf("\n");
    fclose(fp);
}

```



本例要求在 string 文件末加写字符串, 因此, 在程序第 6 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 15 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。

13.4.3 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:

```
fread(buffer, size, count, fp);
```

写数据块函数调用的一般形式为:

```
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。

size 表示数据块的字节数。

count 表示要读写的数据块块数。

fp 表示文件指针。

例如:

```
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 13.6】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。

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

```
struct stu
```

```
{
```

```
    char name[10];
```

```
    int num;
```

```
    int age;
```

```
    char addr[15];
```

```
} boya[2], boyb[2], *pp, *qq;
```

```
main()
```

```
{
```

```
    FILE *fp;
```

```
    char ch;
```

```
    int i;
```

```
    pp=boya;
```

```
    qq=boyb;
```

```
    if((fp=fopen("d:\\jrzh\\example\\stu_list", "wb+"))==NULL)
```

```
{
```



```

main()
{
    FILE *fp;
    char ch;
    int i;
    pp=boya;
    qq=boyb;
    if((fp=fopen("stu_list", "wb+"))==NULL)
    {
        printf("Cannot open file strike any key exit!");
        getch();
        exit(1);
    }
    printf("\ninput data\n");
    for(i=0;i<2;i++, pp++)
        scanf("%s%d%d%s", pp->name, &pp->num, &pp->age, pp->addr);
    pp=boya;
    for(i=0;i<2;i++, pp++)
        fprintf(fp, "%s %d %d %s\n", pp->name, pp->num, pp->age, pp->
            addr);
    rewind(fp);
    for(i=0;i<2;i++, qq++)
        fscanf(fp, "%s %d %d %s\n", qq->name, &qq->num, &qq->age, qq->addr);
    printf("\n\nname\tnumber\t\tage\t\taddr\n");
    qq=boyb;
    for(i=0;i<2;i++, qq++)
        printf("%s\t%5d\t\t%7d\t\t%s\n", qq->name, qq->num, qq->age,
            qq->addr);
    fclose(fp);
}

```



与例 10.6 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 25 和 32 行分别对它们重新赋予了数组的首地址。

13.5 文件的随机读写

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。

13.5.1 文件定位

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：

`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：

`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：

`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。

13.5.2 文件的随机读写

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 13.8】 在学生文件 `stu_list` 中读出第二个学生的数据。

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

```
struct stu
```

```
{
```

```
    char name[10];
```

```
    int num;
```

```
    int age;
```

```
    char addr[15];
```

```
}boy,*qq;
```

```
main()
```

```
{
```

```
    FILE *fp;
```

```
    char ch;
```


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。

13.7 C 库文件

C 系统提供了丰富的系统文件，称为库文件，C 的库文件分为两类，一类是扩展名为“.h”的文件，称为头文件，在前面的包含命令中我们已多次使用过。在“.h”文件中包含了常量定义、类型定义、宏定义、函数原型以及各种编译选择设置等信息。另一类是函数库，包括了各种函数的目标代码，供用户在程序中调用。通常在程序中调用一个库函数时，要在调用之前包含该函数原型所在的“.h”文件。

下面给出 Turbo C 的全部“.h”文件。

Turbo C 头文件

- ALLOC.H 说明内存管理函数(分配、释放等)。
- ASSERT.H 定义 assert 调试宏。
- BIOS.H 说明调用 IBM-PC ROM BIOS 子程序的各个函数。
- CONIO.H 说明调用 DOS 控制台 I/O 子程序的各个函数。
- CTYPE.H 包含有关字符分类及转换的名类信息(如 isalpha 和 toascii 等)。
- DIR.H 包含有关目录和路径的结构、宏定义和函数。
- DOS.H 定义和说明 MSDOS 和 8086 调用的一些常量和函数。
- ERRON.H 定义错误代码的助记符。
- FCNTL.H 定义在与 open 库子程序连接时的符号常量。
- FLOAT.H 包含有关浮点运算的一些参数和函数。
- GRAPHICS.H 说明有关图形功能的各个函数，图形错误代码的常量定义，正对不同驱动程序的各种颜色值，及函数用到的一些特殊结构。
- IO.H 包含低级 I/O 子程序的结构和说明。
- LIMIT.H 包含各环境参数、编译时间限制、数的范围等信息。
- MATH.H 说明数学运算函数，还定了 HUGE_VAL 宏，说明了 matherr 和 matherr 子程序用到的特殊结构。
- MEM.H 说明一些内存操作函数(其中大多数也在 STRING.H 中说明)。
- PROCESS.H 说明进程管理的各个函数，spawn...和 EXEC ...函数的结构说明。
- SETJMP.H 定义 longjmp 和 setjmp 函数用到的 jmp buf 类型，说明这两个函数。
- SHARE.H 定义文件共享函数的参数。
- SIGNAL.H 定义 SIG[ZZ(Z) [ZZ)]IGN 和 SIG[ZZ(Z) [ZZ)]DFL 常量，说明 raise 和 signal 两个函数。
- STDARG.H 定义读函数参数表的宏。(如 vprintf, vsprintf 函数)。
- STDDEF.H 定义一些公共数据类型和宏。
- STDIO.H 定义 Kernighan 和 Ritchie 在 Unix System V 中定义的标准和扩展的类型和宏。还定义标准 I/O 预定义流：stdin, stdout 和 stderr，说明 I/O 流子程序。
- STDLIB.H 说明一些常用的子程序：转换子程序、搜索/ 排序子程序等。
- STRING.H 说明一些串操作和内存操作函数。
- SYS\STAT.H 定义在打开和创建文件时用到的一些符号常量。
- SYS\TYPES.H 说明 ftime 函数和 timeb 结构。
- SYS\TIME.H 定义时间的类型 time[ZZ(Z) [ZZ)]t。

- TIME.H 定义时间转换子程序 asctime、localtime 和 gmtime 的结构，ctime、difftime、gmtime、localtime 和 stime 用到的类型，并提供这些函数的原型。
- VALUE.H 定义一些重要常量，包括依赖于机器硬件的和为与 Unix System V 相兼容而说明的一些常量，包括浮点和双精度值的范围。

13.8 本章小结

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。