第 3 章 C51 编程基础

CHAPTER 3

【本章要点】

- C51 程序设计基础；
- 函数、数组和指针基础知识；
- 程序设计实例。

C 语言是一种通用的计算机程序设计语言，既可以用来编写计算机的系统程序，也可用来编写一般的应用程序。早期，计算机的系统软件主要用汇编语言编写，单片机应用系统更是如此。然而，由于汇编语言程序的可读性和可移植性都较差，采用汇编语言编写单片机应用程序不仅周期长，而且调试和排错都比较困难。为了提高单片机应用程序的开发效率，同时改善程序的可读性和可移植性，采用高级语言无疑是更优的选择。C 语言既具有一般高级语言的特点，又能直接操作计算机的硬件，表达和运算能力也较强，许多以前只能采用汇编语言来解决的问题现在都可以用 C 语言来解决。

本章重点介绍 C51 程序的基本语法知识，包括函数的定义调用，数组与指针的使用。同时，通过实例演示如何访问存储空间。

3.1 程序设计基础

3.1.1 标识符与关键字

在 C 语言中，标识符用于标识源程序中某个对象的名字，这些对象可以是函数、变量、常量、数组、数据类型、存储方式、语句等。标识符由字符串、数字和下画线等组成，第一个字符必须是字母或下画线。C 语言对大小写字母敏感，如 max 与 MAX 是两个完全不同的标识符。标识符的命名应当简洁明了，含义清晰，便于阅读理解。例如，用 max 表示最大值，用 TIMER0 表示定时器 0。

关键字(或保留字)是一类具有固定名称和特定含义的特殊标识符。在编写 C 语言源程序时，一般不允许将关键字另作别用，也就是对于标识符的命名不要与关键字相同。与其他计算机语言相比，C 语言的关键字是比较少的，ANSI C 标准共规定了 32 个关键字，表 3-1 按用途列出了 ANSI C 标准的关键字。

表 3-1 ANSI C 标准的关键字

关 键 字	用 途	说 明
auto	存储种类声明	用以声明局部变量,默认类型为 auto
break	程序语句	退出最内层循环体
case	程序语句	switch 语句中的选择项
char	数据类型声明	单字节整型数或字符型数据,通常占用 1 字节内存
const	数据类型声明	声明只读变量,表示变量值在程序执行过程中不可修改
continue	程序语句	跳过本次循环剩余部分,直接进入下一次循环
default	程序语句	switch 语句中的失败选择项,在所有 case 都不匹配时执行
do	程序语句	与 while 一起使用,构成 do-while 循环结构
double	数据类型声明	声明双精度符点型变量,通常占用 8 字节内存
else	程序语句	与 if 配合构成 if-else 选择结构定义条件不满足时的代码块
enum	数据类型声明	声明枚举类型
extern	存储种类声明	声明在其他程序模块中定义了的全局变量
float	数据类型声明	声明单精度浮点型变量,通常占用 4 字节内存
for	程序语句	构成 for 循环结构,用于执行固定次数的循环
goto	程序语句	构成 goto 转移结构
if	程序语句	构成 if-else 选择结构
int	数据类型声明	声明基本整型变量,通常占用 4 字节内存
long	数据类型声明	声明长整型变量,通常占用 8 字节内存
register	存储种类声明	使用 CPU 内部寄存器的变量
return	程序语句	从函数中返回,结束函数的执行,可选择性地返回一个值
short	数据类型声明	声明短整型变量,通常占用 2 字节内存
signed	数据类型声明	声明符号类型变量,二进制数据的最高位为符号位
sizeof	运算符	计算变量或数据类型所占的内存字节数
static	存储种类声明	声明静态变量
struct	数据类型声明	声明结构体类型
switch	程序语句	构成 switch 选择结构
typedef	数据类型声明	重新进行数据类型定义
union	数据类型声明	声明联合类型
unsigned	数据类型声明	声明无符号变量
void	数据类型声明	声明无类型变量
volatile	数据类型声明	声明该变量在程序执行中可被隐含地改变
while	程序语句	构成 while 和 do-while 循环结构

C51 编译器除了支持 ANSI C 标准的关键字以外,还根据 8051 单片机自身特点扩展出如表 3-2 所示的关键字。

表 3-2 C51 编译器的扩展关键字

关 键 字	用 途	说 明
_at_	地址定位	为变量进行存储器绝对空间地址定位
alien	函数特性声明	用以声明与 PL/M51 兼容的函数
bdata	存储器类型声明	可位寻址的 8051 内部数据存储器
bit	位变量声明	声明一个位变量或位类型的函数
code	存储器类型声明	8051 程序存储器空间

续表

关 键 字	用 途	说 明
compact	存储器模式	指定使用 8051 外部分页寻址数据存储器空间
data	存储器类型声明	直接寻址的 8051 内部数据存储器
idata	存储器类型声明	间接寻址的 8051 内部数据存储器
interrupt	中断函数声明	定义一个中断服务函数
large	存储器模式	指定使用 8051 外部数据存储器空间
pdata	存储器类型声明	分页寻址的 8051 外部数据存储器
priority	多任务优先声明	规定 RTX51 或 RTX51 Tiny 的任务优先级
reentrant	再入函数声明	定义一个再入函数
sbit	位变量声明	声明一个可位寻址变量
sfr	SFR 声明	声明一个 8 位的 SFR
sfr16	SFR 声明	声明一个 16 位的 SFR
small	存储器模式	指定使用 8051 内部数据存储器空间
task	任务声明	定义实时多任务函数
using	寄存器组定义	定义 8051 的工作寄存器组
xdata	存储器类型声明	8051 外部数据存储器

3.1.2 C51 程序的基本语法

1. 数据类型

C 语言的数据结构是以数据类型出现的，数据类型可分为基本数据类型和复杂数据类型，复杂数据类型由基本数据类型构造而成。C 语言中的基本数据类型有 char、int、short、long、float 和 double。对于 C51 编译器来说，short 类型等同于 int 类型，double 类型等同于 float 类型。Keil C51 编译器能够识别的数据类型如表 3-3 所示。

表 3-3 Keil C51 编译器能够识别的数据类型

数 据 类 型	长 度	值 域
unsigned char	单字节	0～255
signed char	单字节	－128～127
unsigned int	双字节	0～65 535
signed int	双字节	－32 768～ 32 767
unsigned long	4 字节	0～4 294 967 259
signed long	4 字节	－2 147 483 648～2 147 483 647
float	4 字节	±1.175 494E－38～±3.402 823E＋38
*	1～3 字节	对象的地址
bit	位	0 或 1
sfr	单字节	0～255
sfr16	双字节	0～65 535
sbit	位	0 或 1

1）char(字符型)

它有 signed char(有符号字符型)和 unsigned char(无符号字符型)之分，默认值为 signed char。它们的长度均为 1 字节，用于存放单字节的数据。

2) int(整型)

它有 signed int(有符号整型)和 unsigned int(无符号整型)之分,默认值为 signed int。它们的长度均为 2 字节,用于存放一个双字节的数据。

3) long(长整型)

它有 signed long(有符号长整型)和 unsigned long(无符号长整型)之分,默认值为 signed long。它们的长度均为 4 字节。

4) float(符点型)

它是符合 IEEE-754 标准的单精度浮点型数据,在十进制中具有 7 位有效数字。float 类型数据占用 4 字节(32 位二进制数),在内存中的存储格式如下:

字节地址	+0	+1	+2	+3
符点数内容	SEEEEEEE	EMMMMMMM	MMMMMMMM	MMMMMMMM

其中,S 为符号位,0 表示正,1 表示负。E 为阶码,占用 8 位二进制数,存储在 2 字节中。注意,阶码值是以 2 为底的指数再加上偏移量 127,这样处理的目的是避免出现负的阶码值,而指数是可正可负的。阶码的正常取值范围是 1～254,从而对应的实际指数的取值范围为 −126～127。M 为尾数的小数部分,用 23 位二进制数表示,存放在 3 字节中。尾数的整数部分永远为 1,因此不予保存,但它是隐含存在的。小数点位于隐含的整数位“1”的后面。一个浮点数的数值范围是:$(-1)^S \times 2^{E-127} \times (1.M)$。

浮点数 −12.5 在 IEEE-754 标准中的表示为 0xC1480000,其在内存中的存储格式如下:

字节地址	+0	+1	+2	+3
符点数内容	11000001	01001000	00000000	00000000

需要指出的是,浮点型数据除了有正常数值之外,还可能出现非正常数值。根据 IEEE-754 标准,当浮点型数据取以下数值(十六进制数)时,即为非正常值:

- 0xFFFFFFFF 非数(NaN)
- 0x7F800000 正溢出(+INF)
- 0xFF800000 负溢出(−INF)

另外,由于 8051 单片机不包括捕获浮点运算错误的中断向量,因此用户必须自己根据可能出现的错误条件,用软件进行适当的处理。

除了以上 4 种基本数据类型之外,还有以下一些数据类型。

5) *(指针型)

指针型数据不同于以上 4 种基本数据类型。它本身是一个变量,但该变量存储的不是普通数据,而是指向另一个数据的地址。指针变量会占据一定的内存单元,在 C51 中,指针变量的长度一般为 1～3 字节。指针变量也具有类型,表示方法是在指针符号 * 的前面加上数据类型符号。例如,char * point1 表示 point1 是一个字符型指针变量; float * point2 表示 point2 是一个浮点型指针变量。指针变量的类型表示该指针所指向地址中数据的类型。使用指针型变量可以方便地对 8051 单片机的物理地址直接进行操作。

6) bit(位变量)

这是 C51 编译器的一种扩展数据类型,用于定义一个位变量,但不能定义位指针,也不

能定义位数组。

7）sfr(特殊功能寄存器)

这也是C51编译器的一种扩展数据类型,用它来定义8051单片机的所有内部8位特殊功能寄存器。

8）sfr16(16位特殊功能寄存器)

它占用两个连续的内存单元,用于定义8051单片机内部16位特殊功能寄存器。

9）sbit(可寻址位)

这也是C51编译器的一种扩展数据类型,用于定义8051单片机内部RAM中的可寻址位或SFR中的可寻址位。

例如:

```
sfr P0 = 0x80;
sbit FLAG1 = P0^1;
```

可以将8051单片机P0口地址定义为80H,将P0.1定义为FLAG1等。

在C语言程序的表达式或变量赋值运算中,有时会出现运算对象的数据不一致的情况,C语言允许任何标准数据类型之间的隐式转换。隐式转换按以下优先级别自动进行:

bit→char→int→long→float

signed→unsigned

其中,箭头方向仅表示数据类型的级别高低,转换时由低级别向高级别进行,而不是数据转换时的顺序。例如,将一个bit(位类型)变量赋给一个int(整型变量)时,不需要先将bit型变量转换成char型之后再转换成int型,而是将bit型变量直接转换成int型并完成赋值运算。一般来说,如果有几个不同类型的数据同时参加运算,先将低级别类型的数据转换成高级别类型,然后再运算处理,并且运算结果为高级别类型数据。C语言除了能对数据类型作自动隐式转换之外,还可以采用强制类型转换符()对数据类型作显式转换。

2. 常量

常量又称为标量,它的值在程序执行过程中不能改变。常量的数据类型有整型、浮点型、字符型和字符串型等,如表3-4所示。

表3-4 常量的数据类型

数据类型	实 例	说 明
整型常量	−123,123,0123,0xAF,123L	八进制由0引导,十六进制由0x引导,在数字后面加一个字母L构成长整数
浮点型常量	−1.2345,1.23E2	又称实型常量,字母E(e)后面必须是整数
字符型常量	'a','A','*'	由单引号'引导
字符串型常量	"student"	由双引号"引导 注意:字符串常量以转义字符'\0'作为结束符,因此'a'与"a"不同

3. 变量及其存储模式

变量是在程序执行过程中,其值能不断变化的量。使用变量之前,必须进行定义,指定一个标识符作为变量名,并明确其数据类型和存储模式,以便编译器为它分配相应的存储单元。在C51中对变量进行定义的格式如下:

[存储种类] 数据类型 [存储器类型] 变量名表;

其中,存储种类和存储器类型是可选项。变量的存储种类有四种:自动(auto)、外部(extern)、静态(static)和寄存器(register)。定义一个变量时,如果省略存储种类选项,则默认为自动变量(auto)。定义一个变量时,除了需要说明其数据类型之外,C51 编译器还允许说明变量的存储器类型。Keil C51 编译器完全支持 8051 系列单片机的硬件结构和存储器组织,对于每个变量都可以准确地赋予其存储器类型,使之能够在单片机系统内准确地定位。表 3-5 列出了 Keil C51 编译器所能识别的存储器类型。

表 3-5 Keil C51 编译器所能识别的存储器类型

存储器类型	说明
DATA	直接寻址的片内数据存储器(128B),访问速度最快
BDATA	可位寻址的片内数据存储器(16B),允许位与字节混合访问
IDATA	间接访问的片内数据存储器(256B),允许访问全部片内地址
PDATA	分页寻址的片外数据存储器(256B)
XDATA	片外数据存储器(64KB)
CODE	程序存储器(64KB)

定义变量时如果省略了存储器类型选项,则按编译时使用的存储器模式 SMALL、COMPACT 或 LARGE 来规定默认存储器类型,确定变量的存储器空间,函数中不能采用寄存器传递的参数变量和过程变量也保存在默认的存储器空间。C51 编译器的三种存储器模式(默认的存储器类型)对变量的影响如下。

(1) SMALL:变量被定义在 8051 单片机的片内数据存储器中,对这种变量的访问速度最快。另外,所有的对象,包括堆栈,都必须位于片内数据存储器中,而堆栈的长度是很重要的,实际栈长取决于不同函数的嵌套深度。

(2) COMPACT:变量被定义在分页寻址的片外数据存储器中,每页片外数据存储器的长度为 256 字节。这时,对变量的访问是通过寄存器间接寻址进行的。堆栈位于 8051 单片机片内数据存储器中。采用这种编译模式时,变量的高 8 位地址由 P2 口确定。采用这种模式的同时,必须适当改变启动配置文件 STARTUP.A51 中的参数:PDATASTARK 和 PDATALEN。用 BL51 进行连接时,还必须采用连接控制命令 PDATA 来对 P2 口地址进行定位,这样才能确保 P2 口为所需要的高 8 位地址。

(3) LARGE:变量被定义在片外数据存储器中(最大可达 64KB),使用数据指针(DPTR)来间接访问变量。这种访问数据的方法效率不高,尤其是对于两个以上字节的变量,可能会增加程序代码长度。

需要特别指出的是,变量的存储种类与存储器类型是完全无关的。例如:

```
static unsigned char data var1;          /* 在片内 RAM 区定义一个静态无符号字符型变量 var1 */
int var2;          /* 定义一个自动整型变量 var2,它的存储器类型由编译模式确定 */
#define CS573L XBYTE[0x2000];          /* 将符号 CS573L 定义成外部数据存储器地址 */
#define CS573H XBYTE[0x2000];          /* 将符号 CS573H 定义成外部数据存储器地址 */
```

注意:在用 XBYTE 时,要包含头文件 ABSACC.H。

8051 系列单片机具有许多种内部寄存器,其中一些是 SFR,如定时器方式控制寄存器(TMOD)、中断允许控制寄存器(IE)等。为了能够直接访问这些 SFR,C51 编译器扩充了

关键字 sfr 和 sfr16,利用这种扩充关键字可以在 C 语言源程序中直接对 8051 单片机的 SFR 进行定义,其定义方法如下:

```
sfr 特殊功能寄存器名 = 地址常量;
```

例如:

```
sfr P0 = 0x80;          //定义 I/O 口 P0,其地址为 0x80
```

这里需要注意的是,在关键字 sfr 后面必须跟一个标识符作为寄存器名,名字可任意选取,但建议遵循一般习惯。等号后面必须是常数,不允许有带运算符的表达式,而且该常量必须在 SFR 的地址范围之内(0x80～0xFF)。在新一代的 8051 单片机中,SFR 经常组合成 16 位来使用,采用关键字 sfr16 可以定义这种 16 位的特殊功能寄存器。例如,对于 8052 单片机的定时器 T2,可采用如下的方法来定义:

```
sfr16 T2 = 0xCC;      /* 定义 TIMER2,其地址为 T2L = 0xCC,T2H = 0xCD */
```

这里 T2 为特殊功能寄存器名,等号后面是它的低字节地址,其高字节地址必须在物理上直接位于低字节之后。这种定义方法适用于新一代的 8051 单片机中新增加的 SFR,但不能用于定时器/计数器 TIMER0 和 TIMER1 的定义。

在 8051 单片机应用系统中,经常需要访问 SFR 中的某些位,C51 编译器为此提供了一种扩充关键字 bit,利用它可以定义可位寻址对象。定义方法有如下三种。

(1) sbit 位变量名=位地址。

这种方法将位的绝对地址赋给位变量,位地址必须位于 0x80～0xFF。例如:

```
sbit OV = 0xD2;
sbit CY = 0xD7;
```

(2) sbit 位变量名=特殊功能寄存器名^位位置。

当可寻址位位于 SFR 中时,可采用这种方法。位位置是一个 0～7 的常数。例如:

```
sfr PSW = 0xD0;
sbit OV = PSW^2;
sbit CY = PSW^7;
```

(3) sbit 位变量名=字节地址^位位置。

这种方法以一个常数(字节地址)作为基地址,该常数必须在 0x80H～0xFF。位位置是一个 0～7 的常数。例如:

```
int bdata ibase;              /* 在位寻址定义一个整型变量 ibase */
char bdata bary[4];           /* 在位寻址定义一个数组 array[4] */
```

使用关键字 sbit 可以独立访问可位寻址对象的某一位。例如:

```
sbit mybit0 = ibase^0;
sbit mybit15 = ibase^15;
sbit Ary07 = bary[0]^7;
sbit Ary37 = bary[3]^7;
```

采用这种方法定义可位寻址变量时,要求基址对象的存储器类型为 bdata,操作符^后面的位位置的最大值取决于指定的基地址类型:char 类型是 0～7;int 类型是 0～15;long 类型是 0～31。

需要注意的是,sbit 是一个独立的关键字,不要将它与关键字 bit 相混淆。关键字 bit 是 C51 编译器的一种扩充数据类型,用来定义一个普通位变量,它的值是二进制数的 0 或 1。一个函数中可以包含 bit 类型的参数,函数的返回值也可以为 bit 类型,例如:

```
static bit direction_bit                    // 定义一个静态位变量 direction_bit
extern bit lock_prt_port                    // 定义一个外部位变量 lock_prt_port
bit bfunc(bit b0,bit b1) // 定义一个返回位类型值的函数 bfunc,函数中包含有两个位类型参数
b0,b1
{…
    return(b1);                             // 返回一个位类型值 b1
}
```

如果在函数中禁止使用中断(# pragma disable)或函数中包含有明确的寄存器组切换(using n),则该函数不能返回位类型值,否则在编译时会产生编译错误。另外,不能定义位指针,也不能定义位数组。

上面介绍了变量及其定义方法,这在编写 C 语言程序时是十分重要的。从变量的作用范围来看,还有全局变量和局部变量之分。全局变量是指在程序开始处或各个功能函数的外部定义的变量。在程序开始处定义的全局变量在整个程序中有效,供程序中所有函数共同使用;而在各功能函数外部定义的全局变量只对从该定义之后的函数有效,定义之前的函数则不能使用它。

局部变量是指在函数内部或以花括号{}围起来的功能块内部所定义的变量,局部变量只在定义它的函数或功能块内有效,在该函数或功能块以外则不能使用它。局部变量可以与全局变量同名,但在这种情况下,局部变量的优先级较高,而同名的全局变量在该功能块被暂时屏蔽。

从变量的存在时间来看,又可分为静态存储变量和动态存储变量。静态存储变量是指该变量在程序运行期间其存储空间固定不变;动态存储变量是指该变量的存储空间不确定,在程序运行期间根据需要动态地为该变量分配存储空间。一般来说,全局变量为静态存储变量,局部变量为动态存储变量。

4. 用 typedef 重新定义数据类型

在 C 语言程序中,除了可以采用上面所介绍的数据类型之外,用户还可以根据自己的需要对数据类型重新定义,重新定义时需用到关键字 typedef,定义格式如下:

typedef 已有的数据类型 新的数据类型名;

其中,“已有的数据类型”是指 C 语言中支持的所有数据类型,包括结构、指针和数组等。“新的数据类型名”可按用户自己的习惯或任务需求自定义。关键字 typedef 的作用只是将 C 语言中已有的数据类型作了置换。因此可用置换后的新数据类型名来进行变量定义。例如:

```
int a;                          //传统变量声明表达式
int myint_t;                    //使用新的类型名 myint_t 替换变量名 a
typedef int myint_t;            //在语句开头加上 typedef 关键字,myint_t 就是定义的新类型
```

3.1.3 运算符与表达式

C 语言对数据有很强的表达能力,具有十分丰富的运算符。运算符就是完成某种特定

运算的符号。表达式则是由运算符及运算对象所组成的具有特定含义的一个语法结构。C语言是一种表达式语言，在任意一个表达式的后面加上分号(;)就构成了一个表达式语句。通过运算符和表达式，可以组成C语言程序的各种语句。

运算符按其在表达式中所起的作用，可分为12个常见的运算符，如表3-6所示。运算符按其在表达式中与运算对象的关系，可分为单目运算符、双目运算符和三目运算符。单目运算符只需要有一个运算对象，双目运算符要求有两个运算对象，三目运算符要求有三个运算对象。运算符的优先级如表3-7所示。掌握各种运算符的意义和使用规则，对于编写正确的C语言程序是十分重要的。

表3-6 运算符与表达式

分 类	运 算 符	示 例	说 明
赋值运算符	=:赋值运算符	x=9	将常数9赋给变量x

算术运算符	+：加或取正值运算符
	-：减或取负值运算符
	*：乘运算符
	/：除运算符
	%：取余运算符
		i=9;
		i=i+1;
		i=i*5;
		i=i/10;
		i=i%10;
			i=9
			i=10
			i=50
			i=5
			i=5

增值和减量运算符	++：增量运算符 --：减量运算符	i++;	i自增1
关系运算符	>：大于 <：小于 >=：大于或等于 <=：小于或等于 ==：等于 !=：不等于	x>y、x+y>z、(x=3)>(y>4)	运算结果只能为0或1
逻辑运算符	‖：逻辑或 &&：逻辑与 !：逻辑非	x&&y、a‖b、!z	运算结果只能为0或1

位运算符	~：按位取反
	<<：左移
	>>：右移
	&：按位与
	^：按位异或
	|：按位或
		i=15<<2;
		i=0x80|0x08;
		i=~0x03;
			i=60
			i=0x88
			i=0xFC

复合赋值运算	+=：加法赋值
	-=：减法赋值
	*=：乘法赋值
	/=：除法赋值
	%=：取模赋值
	<<=：左移位赋值
	>>=：右移位赋值
	&=：逻辑与赋值
	|=：逻辑或赋值
	^=：逻辑异或赋值
	~=：逻辑非赋值
		i=9;
		i+=1;
		i*=5;
		i/=10;
		i%=10;
		i<<=2;
		i|=0x08;
		i~=i;
			i=9
			i=10
			i=50
			i=5
			i=5
			i=20
			i=28
			i=0xE3

续表

分　类	运　算　符	示　例	说　明
逗号运算符	表达式 1,表达式 2,...,表达式 n	(a=3*5,a*4),a+5;	从左到右依次计算出各个表达式的值,而整个逗号表达式的值是最右边表达式(即表达式 n)的值 a=20
条件运算符	逻辑表达式? 表达式 1: 表达式 2	Max=a>b?a:b;	若 a>b,则 Max=a; 若 a<b,则 Max=b
指针和地址运算符	*: 取内容 &: 取地址	int i=3, *ip; ip=&i;	i 的地址赋给了指针变量 ip,指针变量 ip 指向 i,则 *ip=3

强制类型转换运算符	(类型名)	int c;
		int a=5;
		float b=1;
		c=(int)b+5;
			c=6

sizeof 运算符	sizeof(表达式)或 sizeof(数据类型)	int a,b;
		a=sizeof(b);
		b=sizeof(char);
			整型变量 b 占 4B(Visual C++系统中),所以 a=4; b=1

表 3-7　运算符优先级

优先级	运　算　符	含　义	要求运算对象的个数	结合方向
1	() [ ] -> .	圆括号 下标运算符 指向结构体成员运算符 引用结构体成员运算符	无	自左至右
2	! ~ ++ -- - (类型) * & sizeof	逻辑非运算符 按位取反运算符 自增运算符 自减运算符 负号运算符 类型转换运算符 指针运算符 取地址运算符 长度运算符	1(单目运算符)	自右至左
3	* / %	乘法运算符 除法运算符 求余运算符	2(双目运算符)	自左至右
4	+ -	加法运算符 减法运算符	2(双目运算符)	自左至右
5	<< >>	左移运算符 右移运算符	2(双目运算符)	自左至右
6	< <= > >=	关系运算符	2(双目运算符)	自左至右
7	== !=	等于运算符 不等运算符	2(双目运算符)	自左至右
8	&	按位与运算符	2(双目运算符)	自左至右

续表

优先级	运 算 符	含 义	要求运算对象的个数	结 合 方 向
9	^	按位异或运算符	2(双目运算符)	自左至右
10	|	按位或运算符	2(双目运算符)	自左至右
11	&&	逻辑与运算符	2(双目运算符)	自左至右
12	||	逻辑或运算符	2(双目运算符)	自左至右
13	?:	条件运算符	3(三目运算符)	自右至左
14	= += -= *= /= %= >>= <<= &= ^= |=	赋值运算符	2(双目运算符)	自右至左
15	,	逗号运算符(顺序求值运算符)	无	自左至右

3.1.4 C51 程序的基本语句

C51 程序中有 6 类基本语句,如表 3-8 所示。

表 3-8 基本语句

名 称	格 式	示 例	说 明

表达式语句	表达式;	i=9;
		i=i+1;
		i=i*5;
		i=i/10;
		i=i%10;
			—

复合语句	{
	表达式;
	表达式;
	…
	}
		{
		int a=1 ;
		int b=a ;
		int c=a+b ;
		}
			复合语句是由若干条语句组合而成的一种语句,用一个大括号“{}”将若干语句组合在一起而形成的一种功能块

条件语句	if(条件表达式)语句	if(i==1)
		{
		i++;
		}
			如果 i 等于 1,i 自增 1

	if(条件表达式)语句 1
	else 语句 2
		if(i==1)
		{
		i=0;
		}
		else i++;
			如果 i 等于 1,i 重新赋值为 0,否则自增 1

	if(条件表达式 1) 语句 1
	else if(条件表达式 2) 语句 2
	else if(条件表达式 3) 语句 3
	else if(条件表达式 4) 语句 4
	…
	else if(条件表达式 m) 语句 m
	else 语句 n
		if(i==1)
		{
		i=0;
		}
		else i++;
			—

续表

名　称	格　式	示　例	说　明

分支语句	switch(表达式)
	{
	case 常量表达式 1：语句 1
	break;
	case 常量表达式 2：语句 2
	break;
	…
	case 常量表达式 n：语句 n
	break;
	default：语句 d
	}
		switch(i)
		{
		case 0：
		P1＝0X01;
		break;
		case 1：
		P1＝0X02;
		break;
		case 2：
		P1＝0X04;
		break;
		case 3：
		P1＝0X08;
		break;
		}
			根据不同的 i 值，给 P1 口赋相应的值

循环语句	while(条件表达式) 语句	while(1);	条件表达式的结果为真(非 0 值)时，程序就重复执行后面的语句，因此本语句意为程序陷入死循环

	do 语句 while(条件表达式);	do
		{
		i++;
		}
		while(i＜10);
			i 一直自增到 10

	for([初始设定表达式];[循环条件表达式];[更新表达式])语句	int sum＝1;
		int i＝1;
		for
		(i＝1;i＜＝n;i++)
		{
		sum ＝sum＊i;
		}
			计算 n 的阶乘

返回语句	return(表达式); return;	return a;	如果这个函数有返回值，那么 return 语句就返回一个值，如果 return 语句后边带有表达式。则要计算表达式的值，并将表达式的值作为该函数的返回值； 如果这个函数没有返回值，那么 return 语句就退出相应的函数

【本节小结】 本节主要讲述了 C51 编程的基础知识，主要介绍了标识符关键字、数据类型、变量及存储模式、运算符与表达式。

3.2 函数

函数是 C 语言中的一种基本模块，一个 C 语言程序就是由若干个模块化的函数所构成

的。C语言程序总是由主函数 main()开始，main()函数是一个控制程序流程的特殊函数，是程序的起点。在进行程序设计的过程中，如果所设计的程序较大，一般应将其分成若干个子程序模块，每个模块完成一种特定的功能。在C语言中，子程序是用函数来实现的。对于一些需要经常使用的子程序，可以将它们设计成一个专门的函数库，以供反复调用。此外，C51编译器还提供了丰富的运行库函数，用户可以根据需要随时调用。这种模块化的程序设计方法可以大大提高编程效率和速度。

3.2.1 函数的定义

从用户的角度来看，有两种函数：标准库函数和用户自定义函数。标准库函数由C51编译器提供，不需要用户进行定义即可直接调用。用户自定义函数是用户根据需要编写的能实现特定功能的函数，必须先定义之后才能调用。函数定义的一般形式为：

```
函数类型 函数名(形式参数表)
形式参数说明
{
    局部变量定义
    函数体语句
}
```

其中，“函数类型”说明了自定义函数返回值的类型。“函数名”是用标识符表示的自定义函数名字。“形式参数表”中列出的是在主调用函数与被调用函数之间传递数据的形式参数，形式参数的类型必须加以说明。ANSI C标准允许在形式参数表中对形式参数的类型进行说明。如果定义的是无参函数，可以没有形式参数表，但圆括号不能省略。“局部变量定义”是对在函数内部使用的局部变量进行定义。“函数体语句”是为完成该函数的特定功能而设置的各种语句。如果定义函数时只给出一对花括号{}，而不给出其局部变量和函数体语句，则该函数为空函数，这种空函数也是合法的。在进行C语言模块化程序设计时，各模块的功能可通过函数来实现。开始时只设计最基本的模块，其他作为扩充功能在以后需要时再加上。编写程序时，可在将来准备扩充的地方添写一个空函数，这样可使程序的结构清晰，可读性好，而且易于扩展。

3.2.2 函数的调用

1. 函数的调用形式

C语言程序中，函数是可以互相调用的。函数调用就是在一个函数体中引用另外一个已经定义了的函数，前者称为主调用函数，后者称为被调用函数。函数调用的一般形式为：

```
函数名(实际参数表)
```

其中，“函数名”指出被调用的函数。“实际参数表”中可以包含多个实际参数，各个参数之间用逗号隔开。实际参数的作用是将它的值传递给被调用函数中的形式参数。需要注意的是，函数调用中的实际参数与函数定义中的形式参数必须在个数、类型及顺序上严格保持一致，以便将实际参数的值正确地传递给形式参数，否则就会在函数调用时产生意想不到的结果。如果调用的是无参函数，则可以没有实际参数表，但圆括号不能省略。

在C语言中，通常可以采用以下方式完成函数的调用。

1）函数语句

在主调函数中，将函数调用作为一条语句，例如：

```
funl();
```

这是无参调用，它不要求被调函数返回一个确定的值，只要求它完成一定的操作。

2）函数表达式

在主调函数中，将函数调用作为一个运算对象直接出现在表达式中，这种表达式称为函数表达式，例如：

```
c = power(x,n) + power(y,m);
```

这其实是一个赋值语句，包括两个函数调用，每个函数调用都有一个返回值，将两个返回值相加的结果赋值给变量 c。因此，这种函数调用方式要求被调用函数返回一个确定的值。

3）函数参数

在主调函数中，将函数调用作为另一个函数调用的实际参数，例如：

```
y = power(power(i,j),k);
```

其中，函数调用 power(i,j)放在另一个函数调用 power(power(i,j),k)的实际参数表中，以其返回值作为另一个函数调用的实际参数。这种在调用一个函数的过程中又调用了另外一个函数的方式，称为嵌套调用。在输出一个函数的值时，经常采用这种方法。

2. 对被调用函数的说明

与使用变量一样，在调用一个函数之前(包括标准库函数)，必须对该函数的类型进行说明，即“先说明，后调用”。如果调用的是库函数，一般应该在程序的开始处用预处理命令 #include 将有关函数说明的头文件包含进来。

如果调用的是用户自定义函数，而且该函数与调用它的主调用函数在同一个文件中，一般应该在主调用函数中对被调用函数的类型进行说明。函数说明的一般形式为

类型标识符 被调用的函数名(形式参数表)

其中，“类型标识符”说明了函数返回值的类型。“形式参数表”说明了各个形式参数的类型。需要注意的是，函数的说明与函数的定义完全不同。函数的定义是对函数功能的确立，是一个完整的函数单位；而函数的说明只是说明了函数返回值的类型。二者在书写形式上也不一样，函数说明结束时，在圆括号的后面需要用一个;作为结束标志，而在函数定义时，被定义函数名的圆括号后面没有分号(;)，即函数定义还未结束，后面应该接着书写形式参数说明和被定义函数体的部分。

如果被调用函数是在主调用函数前面定义的，或者已经在程序文件的开始处说明了所有被调用函数的类型，可以不必在主调函数中对被调用函数进行说明，也可以将所有用户自定义函数的说明另存为一个专门的头文件，需要时用 #include 将其包含到主程序中去。

C 语言程序中不允许在一个函数定义的内部包括另一个函数的定义，即不允许嵌套函数定义，但允许在调用一个函数的过程中包含另一个函数调用，即允许嵌套函数调用。

3. 函数的参数和函数的返回值

通常，在进行函数调用时，主调用函数与被调用函数之间具有数据传递关系。这种传递是通过函数的参数实现的。在定义一个函数时，位于函数名后面圆括号中的变量名称为“形

式参数”，而在调用函数时，函数名后面括号中的表达式称为“实际参数”。形式参数在未发生函数调用之前，不占用内存单元，因而没有值。只有在发生函数调用时，它才被分配内存单元，同时获得从主调用函数中实际参数传递过来的值。函数调用结束后，它所占用的内存单元也被释放。

实际参数可以是常数，也可以是变量或表达式，但要求它们具有确定的值。进行函数调用时，主调用函数将实际参数的值传递给被调用函数中的形式参数。为了完成正确的参数传递，实际参数的类型必须与形式参数的类型一致，如果两者不一致，则会发生“类型不匹配”错误。

4. 实际参数的传递方式

在进行函数调用时，必须用主调用函数中的实际参数来替换被调用函数中的形式参数，这就是所谓的参数传递。在C语言中，对于不同类型的实际参数，有以下三种不同的传递方式。

1）基本类型的实际参数传递

当函数的参数是基本类型的变量时，主调用函数将实际参数的值传递给被调用函数中的形式参数，这种方式称为值传递。函数中的形式参数在未发生函数调用之前是不占用内存的，只有在进行函数调用时才为其分配临时存储单元，而函数的实际参数则是要占用实际的存储单元。值传递方式是将实际参数的值传递到为被调用函数中形式参数分配的临时存储单元中，函数调用结束后，临时存储单元被释放，形式参数的值也就不复存在，但实际参数所占用的存储单元仍然保持原来的值不变。这种参数传递方式在执行被调用函数时，如果形式参数的值发生变化，主调用函数中实际参数的值不会受到影响，因此值传递是一种单向传递。

2）数组类型的实际参数传递

当函数的参数是数组类型的变量时，主调用函数将实际参数数组的起始地址传递到被调用函数中形式参数的临时存储单元，这种方式称为地址传递。地址传递方式在执行被调用函数时，形式参数通过实际参数传来的地址，直接到主调用函数中去存取相应的数组元素，故形式参数的变化会改变实际参数的值，因此地址传递是一种双向传递。

3）指针类型的实际参数传递

当函数的参数是指针类型的变量时，主调用函数将实际参数的地址传递给被调用函数中形式参数的临时存储单元，因此也属于地址传递。在执行被调用函数时，也是直接到主调用函数中去访问实际参数变量。在这种情况下，形式参数的变化会改变实际参数的值。

3.2.3 中断服务程序

C51编译器支持在C语言源程序中直接编写8051单片机的中断服务函数程序，从而减轻了采用汇编语言编写中断服务程序的烦琐程度。为了满足在C语言中直接编写中断服务函数的需要，C51编译器对函数的定义进行了扩展，增加了一个扩展关键字interrupt。它是函数定义时的一个选项，加上这个选项即可以将一个函数定义成中断服务函数。定义中断服务函数的一般形式为

```
函数类型 函数名(形式参数表) [interrupt n] [using n]
```

关键字interrupt后面的n是中断号，n的取值范围为0～31。编译器从8n＋3处产生

中断向量,具体的中断号 n 和中断向量取决于 8052 系列单片机芯片型号,常用中断源和中断向量如表 3-9 所示。

表 3-9 常用中断号与中断向量

中断号 n	中断源	中断向量 8n+3	中断号 n	中断源	中断向量 8n+3
0	外部中断 0	0003H	3	定时器 1	001BH
1	定时器 0	000BH	4	串口	0023H
2	外部中断 1	0013H	5	定时器 2	002BH

8051 系列单片机可以在片内 RAM 中使用 4 个不同的工作寄存器组,每个寄存器组中包含 8 个工作寄存器(R0～R7)。C51 编译器扩展了一个关键字 using,专门用来选择 8051 单片机中不同的工作寄存器组。using 后面的 n 是一个 0～3 的常整数,分别选中 4 个不同的工作寄存器组,如表 1-4 所示。在定义一个函数时,using 是一个选项,如果不用于该选项,则由编译器自动选择一个寄存器组作为寄存器组访问。需要注意的是,关键字 using 和 interrupt 的后面都不能有带运算符的表达式。

关键字 using 对函数目标代码的影响如下:

- 在函数的入口处将当前工作寄存器组保护到堆栈中;
- 指定的工作寄存器内容不会改变;
- 函数退出之前将被保护的工作寄存器从堆栈中恢复。

使用关键字 using 在函数中确定一个工作寄存器组时,必须十分小心,一定要保证任何寄存器组的切换都只在控制的区域内发生。如果做不到这一点,将产生错误的结果。另外还要注意,带 using 属性的函数原则上不能返回 bit 类型的值,并且关键字 using 不允许用于外部函数。

关键字 interrupt 也不允许用于外部函数,它对中断函数目标代码的影响如下:

- 在进入中断函数时,特殊功能寄存器 ACC、B、DPH、DPL、PSW 将被保存入栈;
- 如果不使用关键字 using 进行工作寄存器组切换,则将中断函数中用到的全部工作寄存器都入栈保存;
- 函数退出之前所有的寄存器出栈恢复;
- 中断函数由 8051 单片机指令 RETI 结束。

编写 8051 单片机中断函数时应遵循以下规则。

(1) 中断函数不能进行参数传递。如果中断函数中包含任何参数声明,都将导致编译出错。

(2) 中断函数没有返回值。如果企图定义一个返回值,将得到不正确的结果,因此建议在定义中断函数时将其定义为 void 类型,以明确说明没有返回值。

(3) 在任何情况下,都不能直接调用中断函数,否则会产生编译错误。因为中断函数退出是由 8051 单片机指令 RETI 完成的,RETI 指令影响 8051 单片机的硬件中断系统。如果在没有实际中断请求下直接调用中断函数,则 RETI 指令的操作结果会产生一个致命的错误。

(4) 如果在中断函数中调用了其他函数,则被调用函数所使用的寄存器必须与中断函数相同。用户必须保证按要求使用相同的寄存器组,否则会产生不正确的结果。如果定义中断函数时没有使用 using 选项,则由编译器自动选择一个寄存器组当作绝对寄存器访问。

另外，由于中断的产生不可预测，中断函数对其他函数的调用可能形成递归调用，需要时可被中断函数所调用的其他函数定义成再入函数。

(5) C51 编译器从绝对地址 8n+3 处产生一个中断向量，其中 n 为中断号。该向量包含一个到中断函数入口地址的绝对跳转。在对源程序编译时，可用编译控制命令 NOINTVECTOR 抑制中断向量的产生，从而使用户有能力从独立的汇编程序模块中提供中断向量。

3.2.4 程序预处理

C 语言与其他高级程序设计语言的一个主要区别就是对程序的编译预处理功能，编译预处理器是 C 语言编译器的一个组成部分。在 C 语言中，通过一些预处理命令可以在很大程度上为 C 语言本身提供许多关于功能和符号等方面的扩展，增强了 C 语言的灵活性和方便性。预处理命令可以在编写程序时加在需要的地方，但它只在程序编译时起作用，且通常是按行进行处理的，因此又称为编译控制行。C 语言的预处理命令类似于汇编语言中的伪指令。编译器在对整个程序进行编译之前，先对程序中的编译控制行进行预处理，然后再将预处理的结果与整个 C 语言源程序一起进行编译，从而产生目标代码。C51 编译器的预处理器支持所有满足 ANSI C 标准规则的项处理命令。常用的预处理命令有宏定义、文件包含和条件编译命令。为了与一般 C 语言语句相区别，预处理命令由符号 # 开头。

1. 宏定义

宏定义命令为 #define，它的作用是用一个字符串来进行替换，而这个字符串既可以是常数，也可以是其他任何字符串，甚至还可以是带参数的宏。宏定义的简单形式是符号常量定义，复杂形式是带参数的宏定义。

1) 不带参数的宏定义

不带参数的宏定义又称符号常量定义，一般格式为

```
#define 标识符 常量表达式
```

其中，“标识符”是所定义的宏符号(也称宏名)。它的作用是在程序中使用所指定的标识符来代替所指定的常量表达式。例如，#define NaN 0xFFFFFFFF 就是用 NaN 这个符号来代替常数 0xFFFFFFFF。使用了这个宏定义之后，程序中就不必每次都写出常数 0xFFFFFFFF，而可以用符号 NaN 来代替。在编译时，编译器会自动将程序中所有的符号名代替成常数 0xFFFFFFFF。这种方法不仅使得可以在 C 语言源程序中用一个简单的符号名来替换一个很长的字符串，还可以定义一些有意义的标识符来提高程序的可读性。

通常，程序的所有符号定义都集中放在程序的开始处，以便检查和修改，提高程序的可靠性。另外，如果需要修改程序中的某个常量，可以不必修改整个程序，只要修改一下相应的符号常量定义行即可。

在实际使用宏定义时，按一般习惯，通常将宏符号名用大写字母表示，以区别于其他的变量名。宏定义不是 C 语言的语句，因此在宏定义行的末尾不要加分号，否则在编译时将连同分号一起进行替换而导致出现语法错误。在进行宏定义时，可以引用已经定义过的宏符号名，即可以进行层层代换，但最多不能超过 8 级嵌套。需要注意的是，预处理命令对于程序中用双引号括起来的字符串内的字符，即使该字符与宏符号名相同，也不作替换。

宏符号名的有效范围是从宏定义命令 #define 开始，直到本源文件结束。通常将宏定

义命令＃define 写在源程序的开头，函数的外面，作为源文件的一部分，从而在整个文件范围内有效。另外，需要时可以用命令＃undef 来终止宏定义的作用域。

2）带参数的宏定义

带参数的宏定义与符号常量定义的不同之处在于对源程序中出现的宏符号名不仅进行字符串替换，而且还进行参数替换。带参数宏定义的一般格式为

```
＃define 宏符号名(参数表) 表达式
```

其中，表达式内包含了在括号中所指定的参数，这些参数称为形式参数，在以后的程序中它们将被实际参数所替换。带参数的宏定义将一个带形式参数的表达式定义为一个带形式参数表的宏符号名，对程序中所有带实际参数表的该宏符号名用指定的表达式来替换，同时用参数表中的实际参数替换表达式中对应的形式参数。

带参数的宏定义常用来代表一些简短的表达式，将直接插入的代码代替函数调用，从而提高程序的执行效率，例如：

```
＃define MIN(x,y) (((x)<(y))?(x):(y))
```

该段代码定义了一个带参数的宏 MIN(x,y)，以后在程序中就可以用这个宏而不用函数 MIN()。

m＝MIN(u,v)；经宏展开后，成为 m＝(((u)<(v))?(u)：(v))；

带参数的宏定义可以引用已定义过的宏定义，即宏定义的嵌套(最多不超过 8 级)。

2. 文件包含

文件包含是指一个程序文件将另一个指定的文件的全部内容包含进来。在前面的例子中，已经多次使用过文件包含命令＃include < stdio. h>，就是将 C51 编译器提供的输入输出库函数的说明文件 stdio. h 包含到自己的程序中去。文件包含命令的一般格式为

```
＃include <文件名>
```

文件包含命令＃include 的功能是用指定文件的全部内容替换该预处理行。在进行较大规模程序设计时，文件包含命令十分有用。为了适应模块化编程的需要，可以将组成 C 语言的各个功能函数分散到多个程序文件中，分别由若干人员完成编程，最后再用＃include 命令将它们嵌入到一个总的程序文件中去。需要注意的是，一个＃include 命令只能指定一个被包含文件，如果程序中需要包含多个文件，则需要使用多个包含命令。此外，还可以将一些常用的符号常量、带参数的宏以及构造类型的变量定义在一个独立的文件中，当某个程序需要时再将其包含进来。这样做可以减少重复劳动，提高编程效率。

文件包含命令＃include 通常放在 C 语言程序的开头，被包含的文件一般是一些公用的宏定义和外部变量说明，当它们出错或是由于某种原因需要修改其内容时，只需对相应的包含文件进行修改，而不必对使用它们的各个程序文件都修改，这样有利于程序的维护和更新。当程序中需要调用 C51 编译器提供的各种库函数的时候，必须在程序的开头使用＃include 命令将相应函数的说明文件包含进来。

3. 条件编译

一般情况下，对 C 语言程序进行编译时，所有的程序行都参加编译，但有时希望对其中一部分内容在满足一定条件时才进行编译，这就是所谓的条件编译。条件编译可以选择不同的编译范围，从而产生不同的代码。C51 编译器的预处理器提供以下条件编译命令：

#if、#else、#endif、#ifdef、#ifndef。这些命令有三种使用格式，分述如下。

1）条件编译命令格式一

```
#ifdef 标识符
    程序段 1
#else
    程序段 2
#endif
```

该命令格式的功能是如果指定的标识符已被定义，则程序段 1 参加编译并产生有效代码，而忽略程序段 2，否则程序段 2 参加编译并产生有效代码而忽略掉程序段 1。其中，#else 和程序段 2 可以没有。这里的程序段既可以是 C 语言的语句组，也可以是命令行。这种条件编译对于提高 C 语言源程序的通用性很有好处。

2）条件编译命令格式二

```
#ifndef 标识符
    程序段 1
#else
    程序段 2
#endif
```

该命令格式与第一种命令格式只在第一行上不同，它的作用与第一种刚好相反，即如果指定的标识符未被定义，则程序段 1 参加编译并产生有效代码，而忽略掉程序段 2，否则程序段 2 参加编译并产生有效代码，而忽略掉程序段 1。

3）条件编译命令格式三

```
#if 常量表达式 1
    程序段 1
#elif 常量表达式 2
    程序段 2
…
#elif 常量表达式 n-1
    程序段 n-1
#else
    程序段 n
#endif
```

这种格式条件编译的功能是如果常量表达式 1 的值为真(非 0)，则程序段 1 参加编译，然后控制传递个匹配的 #endif 命令，结束本次条件编译，继续下面的编译处理。否则，如果常量表达式 1 的值为假(0)，则忽略掉程序段 1(不参加编译)，而将控制传递给下面的一个 #elif 命令，对常量表达式 2 的值进行判断。如果常量表达式 2 的值为假(0)，则将控制再传递给下一个 #elif 命令。如此进行，直到遇到 #else 或 #endif 命令为止。使用这种条件编译格式可以事先给定某一条件，使程序在不同的条件下完成不同的功能。

4. 其他预处理命令

除了上面介绍的宏定义、文件包含和条件编译预处理命令之外，C51 编译器还支持 #error、#pragma 和 #line 预处理命令。#line 命令一般很少使用，下面介绍 #error 和 #pragma 命令的功能和使用方法。

#error 命令通常嵌入在条件编译之中，以便捕捉一些不可预料的编译条件。正常情况下，该条件的值为假；若条件的值为真，则输出一条由 #error 命令后面的字符串所给出的

错误信息并停止编译。例如,如果有 # define MYVAL,它的值必须为 0 或 1。为了测试 MYVAL 的值是否正确,可在程序中安排如下一段条件编译代码:

```
#if (MYVAL! = 0&&MYVAL! = 1)
#error MYVAL must be defined to either 0 or 1
#endif
```

当 MYVAL 的值出错时,输出出错信息并停止编译。

pragma 命令通常用在源程序向编译器传送控制命令,使用格式为

```
#pragma 编译命令名序列
```

pragma 命令可以出现在 C 语言源程序中的任何一行,从而使编译器能重复执行某些编译控制命令以达到某种特殊的目的。如果 # pragma 命令后面的参数不是 C51 编译器的合法控制命令,编译器将忽略其作用。需要指出的是,并非所有的 C51 编译器控制命令都可以在 C 语言源程序中采用 # pragma 预处理命令多次,C51 编译器的首要控制命令只能使用一次,如果多次使用,将导致致命的编译错误。

【本节小结】 本节讲述了函数的相关知识,主要包括函数的定义与调用、中断服务程序和程序预处理。

3.3 数组与指针

数组是一组具有固定数目和相同类型数组元素的有序集合,其数组元素的类型为该数组的基本类型,构成一个数组的各元素必须是同一类型的变量,不允许在同一数组中出现不同类型的变量。

数组数据是用同一个名字的不同下标访问的。例如,对于数组 a[i],当 i=0,1,2,…,n 时,a[0],a[1],…,a[n]分别是数组 a[i]的元素(或成员)。数组可分为一维、二维、三维、多维数组,常用的有一维、二维数组和字符数组。

指针是 C 语言中的一个重要概念,指针类型数据在 C 语言程序中的使用十分普遍。正确使用指针类型数据可以有效地表示复杂的数据结构,不但可以直接处理内存地址,而且可以更为有效地使用数组。

3.3.1 一维数组

定义一维数组的格式如下:

数据类型 数组名[整型表达式];

例如,char ch[10]; 定义了一个一维字符型数组,它有 10 个元素,每个元素由不同的下标表示,分别为 ch[0],ch[1],ch[2],…,ch[9]。注意:数组的第一个元素的下标为 0 而不是 1,即数组的第一个元素是 ch[0]而不是 ch[1],而数组的第 10 个元素为 ch[9]。

3.3.2 二维数组

定义二维数组的格式如下:

数据类型 数组名[常量表达式][常量表达式];

例如，int a[3][5]；定义了一个 3 行 5 列共计 15 个元素的二维数组。二维数组的存取顺序是按行存取的：先存取第一行元素的第 0 列、第 1 列、第 2 列……直到第一行的最后一列，然后返回到第二行开始，再取第二行的第 0 列、第 1 列……直到第二行的最后一列，以此类推，直到最后一行的最后一列。

C 语言允许使用多维数组，有了二维数组的基础，理解掌握多维数组并不困难。例如，float a[2][3][4]；定义了一个类型为浮点数的三维数组。

数组中的值可以在程序运行期间用循环和键盘输入语句进行赋值，但这样做将耗费许多机器运行时间，对大型数组而言尤其如此，对此可以用数组初始化的方法加以解决。数组初始化就是在定义说明数组的同时给数组赋新值，这项工作是在程序的编译中完成的。对数组的初始化可用以下方法实现。

(1) 在定义数组时对数组的全部元素赋予初值，例如：

```
int idata a[6] = {0,1,2,3,4,5};
int a[3][4] = {(1,2,3,4),(5,6,7,8),(19,10,11,12)};
int a[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

(2) 只对数组的部分元素初始化，例如：

```
int idata a[10] = (0,1,2,3,4,5);
```

(3) 若定义数组时对数组的全部元素均不赋予初值，则数组的全部元素默认赋值为 0，例如：

```
int idata a[10];
```

a[0]～a[9]全部被赋予初值 0。

3.3.3 字符数组

字符数组用以存储字符数据，每个数组元素存放一个字符，因此可以用字符数组存储长度不同的字符串。

1) 字符数组的定义

字符数组的定义与其他的数组定义类似，例如，“char a[10];”定义了一个有 10 个元素、每个元素存储一个字符的一维字符数组 a。

2) 字符数组初始化

字符数组初始化最直接的方法是将各字符逐个赋予数组中的各个元素。例如，“char a[10]= {'C','H','E','N','G',' ','D','U'};”定义了一个字符型数组 a[]，它有 10 个数组元素，并且将 9 个字符(其中包括一个字符串结束标志'\0')分别赋给了 a[0]～a[8]，剩余的 a[9]被自动赋予空格字符。初始化后的数组内容为：

```
{'C','H','E','N','G',' ','D','U',' ','/0',''}
```

C 语言还允许用字符串常量直接给字符数组赋予初值，其方法有以下两种形式：

```
char a[10] = {"CHENGDU"};
char a[10] = "CHENGDU";
```

用双引号(" ")括起来的一串字符称为字符串常量(如"Happy")，C 编译器会自动地在字符末尾加上结束符'\0'(NULL)。用单引号('')括起来的字符为字符的 ASCII 码值，而不

是字符串。例如'a'表示 a 的 ASCII 码值为 97；而"a"表示一个字符串，它由一个字符('a')和结束符('\0')组成。

一个字符串可以用一维数组来装入，但数组的元素数目一定要比字符多一个，以便 C 编译器自动在其后面加入结束符'\0'。

若干字符串可以装入一个二维字符数组中，称为字符数组。数组的第一个下标是字符串的数量，第二个下标定义每个字符串的最大长度，该长度应当比这批字符串中最长的字符串多一个字符，用于装入结束符'\0'。例如，"char a[60][81];"定义了一个二维字符数组 a，它可容纳 60 个字符串，每串最长可达 80 个字符。

例如：

```
uchar code msg[ ][17] = {("This is a test",\n),("message 1",\n),("message 2",\n)};
```

这是一个二维数组，第二个下标必须给定，因为它不能从数据表中得到，第一个下标可默认由数据常量表决定(本例中实际为 3)。

3.3.4 指针与地址

为了了解地址和指针的基本概念，必须了解在单片机中数据是如何存储和读取的。

在单片机中，所有的数据都存储在存储器中。一般把存储器中的一个字节称为一个内存单元。一旦程序中定义了一个变量，C51 编译器在编译时就给这个变量在存储器中分配相应的存储单元。不同的数据类型所占用的内存单元数不等，如整型变量占两个单元，字符型变量占一个单元等。为了正确地访问这些内存单元，必须为每个内存单元分配一个编号，根据内存单元的编号就可以准确地找到该内存单元，这个编号被称为地址。

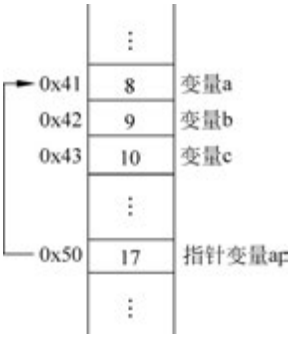

图 3-1 变量与地址对应关系图

根据内存单元编号或地址就可以找到所需的内存单元，通常也把这个地址称为指针，因此，地址和指针是同一事物的不同名称。内存单元的指针和内存单元的内容是两个不同的概念。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。假设程序中定义了三个字符型变量 a、b、c，其值分别为 6、8、10。在编译时，编译器将地址 0x41 的内存单元分配给变量 a，将地址 0x42 的内存单元分配给变量 b，将地址为 0x43 的内存单元分配给变量 c，则变量 a、b、c 与地址为 0x41～0x43 的内存单元之间对应的关系如图 3-1 所示。

对于变量，要弄清两个概念：变量名和变量值。前者是一个数据的标号，后者是一个数据的内容。在存储器中，变量 a、b、c 是不存在的，编译的时候就会将这些变量名用地址代替。对变量值的存取是通过地址进行的，存取的方式有两种。

1. 直接访问方式

例如：

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

其执行过程是：首先，根据变量名与内存单元地址的对应关系，找到变量 a 在内存中的位置，即地址 0x41；然后，从此地址对应的字节中取出 a 变量的值 8，并通过 printf()语句按一

定的格式输出。这种访问方式就是直接访问方式。

2. 间接访问方式

例如，要访问变量 a 的值时，可以将变量 a 的地址放在另外的内存单元（如放在 0x50、0x51、0x52 这三个字节组成的一组内存单元中）中。访问时，首先找到存放变量 a 的地址的内存单元地址（0x50、0x51、0x52），从中取出变量 a 的地址（0x41），然后从此地址对应的字节中取出 a 变量的值 8。这种访问方式就是间接访问方式，并使用了指针。

在进行单片机指针声明之前，首先应该了解单片机的存储类型与存储区关系。从物理结构上，存储区可以分为 4 个部分：片内数据存储区、片外数据存储区、片内程序存储区和片外程序存储区。由于程序在运行时，内外部程序存储区对用户是透明的，所以在逻辑上存储区可以分为片内数据存储区、片外数据存储区和程序存储区。不同型号的单片机，其片内 RAM 的大小是不一样的，如 8051 有 128 字节的片内 RAM，8052 有 256 字节的片内 RAM。

单片机 C 语言存储类型与存储区的关系如下。

- data：可寻址片内 RAM，地址空间为 00H～7FH。
- bdata：可位寻址的片内 RAM，地址空间为 20H～2FH，包含 128 个位地址。
- idata：可寻址片内 RAM，允许访问全部内部 RAM。要访问单片机高 128 字节存储。空间时，必须使用该存储类型。

指针变量的定义与一般变量的定义类似，其一般形式如下：

数据类型 [存储器类型 1] * [存储器类型 2] 标识符;

其中，“标识符”是所定义的指针变量名。“数据类型”说明了该指针变量所指向的变量的类型。“存储器类型 1”和“存储器类型 2”是可选项，它是 C51 编译器的一种扩展。如果带有“存储器类型 1”选项，则指针被定义为基于存储器的指针；无此选项时，被定义为一般指针。这两种指针的区别在于它们的存储字节不同。一般指针在内存中占用 3 字节，第 2 和第 3 字节分别存放该指针的高位和低位地址偏移量。存储器类型的编码值如下：

存储器类型 1	idata/data/bdata	xdata	pdata	code
编码值	0x00	0x01	0xFE	0xFF

“存储器类型 2”选项用于指定指针本身的存储器空间。

一般指针可用于存取任何变量而不必考虑变量在 8051 单片机存储器空间的位置，许多 C51 库函数采用了一般指针，函数可以利用一般指针来存取位于任何存储器空间的数据。请注意汇编语言代码中，指针的第 1 字节是存储器类型编码值，第 2 和第 3 字节是地址偏移量。

如果在定义一般指针时带有“存储器类型 2”选项，则可指定一般指针本身的存储器空间位置，例如：

```
char * xdata strptr;                    /* 位于 xdata 空间的一般指针 */
int * data numptr;                      /* 位于 data 空间的一般指针 */
long * idata varptr;                    /* 位于 idata 空间的一般指针 */
```

基于存储器指针所指对象具有明确的存储器空间，长度可为 1 字节（存储器类型为 idata、data、pdata）或 2 字节（存储器类型为 code、xdata），例如：

```
char data * str;                        /* 指向 data 空间 char 类型数据的指针 */
```

```
int xdata * numtab;                    /* 指向 xdata 空间 int 型数据的指针 */
long code * powtab                     /* 指向 code 空间 long 型数据的指针 */
```

3.3.5 指针的寻址

C51 语言中的数组寻址和普通 C 语言基本一致,只是当数组存储在片内时,由于片内 RAM 资源十分有限,所以很难有比较复杂的数据结构,而且在编程过程中也尽量避免在片内 RAM 中使用较大数组。对于片外数据存储和程序存储空间的指针访问,单片机有其自身的特点。

1. 指向 data 区的指针寻址实现

例如,在一个检测系统中,通过 AD 转换把外部数据输入单片机,单片机对 6 次采样的数据求和,这些数据都存储在 data 区中。

```
unsigned char data input - data[6];
unsigned int data sum, i;
void collect - data( );
unsigned char data * data yc;
void main( )
{
sum = 0;
collect - data( );                     /* 采集数据函数,输入数组 Input - Data 中 */
yc = input - data;
for(i = 0;i < = 6;i++;yc++)
  sum += * yc;
  }
```

可以看出,这里指针的应用和普通 C 语言是差不多的。

2. 指向片外数据存储区的指针寻址实现

由于单片机的片内数据存储区有限,因此在单片机应用中常常要扩展片外数据存储区,对于片外数据存储区和扩展端口,采用指针寻址可以方便地进行数据的读写操作。

```
#include"reg52.h"
void main( )
{
  unsigned char xdata * p;
  unsigned char i;
  p = 0x0050;
  for( i = 0x00 ;i < 0x10 ; i++)
  {
     * p = i;
    p++;
  }
  while( 1 );
}
```

本例对片外数据存储区 xdata 的地址 50H～5FH 依次赋值 0x00～0x0F。

3. 指向程序存储区的指针寻址实现

指针指向数据存储区的实质就是 C 语言中指向函数的指针这一概念,可以利用函数指针来实现对程序存储区的访问和函数调用。函数指针的定义格式如下:

```
类型标识符 ( * 指针变量名)([参数],[参数 2],…);
```

在定义好指针以后，就可以给指针变量赋值，使其指向目标函数的起始地址，然后用(*指针变量名)([参数1],[参数2],…)；即可调用这个函数。例如，主程序中需要调用一个键盘扫描函数scan()，代码如下：

```
void scan( );
void main( )
{
  void( *yc )( );                    //定义指向函数指针
  yc = scan;                         //给指针yc赋值,yc指向程序存储区
  for( ; ; )
  {
  ( * yc )( );                       //调用函数
  ...
  }
}
```

如果函数scan()本身已经固化在程序存储区某个固定地址(如0x6000)，可以直接给yc指定数值(yc=0x6000;)，调用格式相同。

3.3.6 使用_at_关键字进行绝对地址定位

在C51编译器中，可以使用_at_关键字为变量指定存储器中的绝对地址，以便直接访问特定存储器空间，其一般格式如下：

[存储器类型] 数据类型说明符 变量名 _at_ 地址常数;

其中，存储器类型为data、bdata、idata、pdata等C51能识别的数据类型。如果省略，则按存储模式规定的默认存储器类型确定变量的存储器区域；数据类型为C51支持的数据类型。地址常数用于指定变量的绝对地址，必须位于有效的存储器空间之内；使用_at_定义的变量必须为全局变量；绝对变量不能被初始化；bit类型的变量及函数不能使用_at_指定。

示例1：

```
#include<reg51.h>
char xdata LED_Data[50] _at_ 0x8000;
main( )
{
  LED_Data[0] = 0x23;
}
```

在Keil中运行以上程序，可以在存储器窗口中输入x：0x8000，按Enter键后可以看到0x8000地址中的值为0x23。

示例2：

```
#define uchar unsigned char              //定义符号uchar为数据类型符unsigned char
#define uint unsigned int                //定义符号uint为数据类型符unsigned int
data uchar x1 _at_ 0x40;                 //在data区中定义字节变量x1,它的地址为40H
xdata uint x2 _at_ 0x2000;               //在xdata区中定义字变量x2,它的地址为2000H
void main(void)
{
    x1 = 0xff;
    x2 = 0x1234;
    ……
```

```
    while(1);
}}
```

3.3.7 应用举例

【例 3-1】 在外部 RAM 中以 0x100 为首地址的空间定义一个一维数组 a,并将 a 中的数据放入 idata 中。

参考程序代码见本书配套的源码资源。

【例 3-2】 通过指针对 data、idata、xdata 和 ROM 存储空间的访问,并将数组赋值到 idata 指定位置。

参考程序代码见本书配套的源码资源。

【本节小结】 本节讲述了数组与指针的相关知识,主要包括一维、二维以及字符数组的定义、指针与地址的关系、指针的运用、几种存储空间访问方式。

3.4 习题

3-1 C51 有哪几种存储类型?

3-2 bit 和 sbit 定义的位变量有什么区别?

3-3 sizeof 是函数吗? 它的使用方式是什么?

3-4 C51 程序语言中,* 和 & 运算符分别用来实现对指针的什么操作?

3-5 函数分为几种? 分别是什么?

3-6 简述函数的形参与实参的区别和用法。

3-7 中断号 3 的含义是什么?

3-8 中断函数能进行参数传递吗? 它有返回值吗?

3-9 数组在存储空间中是按什么顺序排列的? 若数组的首地址为 0200H,那么数组中第 5 个元素的存储地址是多少?

3-10 字符串的结束符'\0'是否会占用 1 字节?

3-11 如何在外部 ROM 中指定位置建立数组?

3-12 如何使用指针将 idata 区域地址 0x50 上的值赋予 xdata 区域地址 0x0050?