嵌入式 Linux C 语言程序设计基础教程

作者：华清远见

第 2 章 数据

本章目标

在上一章中，读者了解了嵌入式的基本概念，学习了嵌入式 Linux C 语言相关开发工具。本章主要介绍嵌入式 Linux C 语言的数据的相关知识。通过本章的学习，读者将会掌握如下内容：

- ANSI C 与 GNU C
- C 语言的基本数据类型
- 变量的定义、作用域、链接属性及存储方式
- 常量的定义方式
- 预处理
- 字长和数据类型
- 数据对齐
- 字节序

2.1 ANSI C与GNU C

2.1.1 ANSI C简介

C语言是国际上广泛流行的一种计算机高级编程语言，它具有丰富的数据类型以及运算符，并为结构程序设计提供了各种数据结构和控制结构，同时具有某些低级语言的特点，可以实现大部分汇编语言功能，非常适合编写系统程序，也可用来编写应用程序。而且，C语言程序具有很好的可移植性。

1983年，美国国家标准协会（ANSI）根据C语言问世以来各种版本对C的发展和扩充制定了新的标准，并于1989年颁布，被称为ANSI C或C89。目前流行的C编译系统都是以它为基础的。

2.1.2 GNU C简介

GNU项目始创于1984年，旨在开发一个类似UNIX，且为自由软件的完整的操作系统。GNU项目由很多独立的自由/开源软件项目组成，其官方站点为http://www.gnu.org 。如今，这些GNU中的软件项目已经和Linux内核一起成为GNU/Linux的组成部分。

GCC是GNU的一个项目，是一个用于编程开发的自由编译器。最初，GCC只是一个C语言编译器，它是GNU C Compiler的英文缩写。随着众多自由开发者的加入和GCC自身的发展，如今的GCC已经是一个支持众多语言的编译器了，其中包括C,C++,Ada,Object C和Java等。所以，GCC也由原来的GNU C Compiler变为GNU Compiler Collection，也就是GNU编译器家族的意思。

在Linux下编程最常用的C编译器就是GCC，除了支持ANSI C外，还对C语言进行了很多扩展，这些扩展对优化、目标代码布局、更安全地检查等方面提供了很强的支持。本文把支持GNU扩展的C语言称为GNU C。本章主要介绍GNU C的基本语法，最后会简单介绍一些常用的扩展。GNU C可以理解为在标准C的基础上进行了扩展。在了解这些扩展之前，我们先简单回顾一下标准C语言。

C语言的数据类型根据其不同的特点，可以分为基本类型、构造类型和空类型，其中每种类型都还包含了其他一系列数据类型，它们之间的关系如图2-1所示。

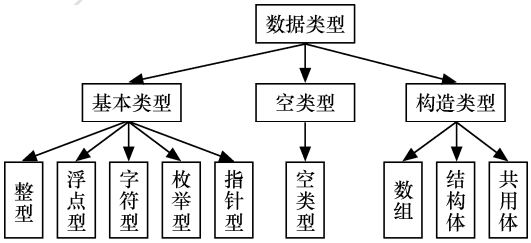

图2-1 常见数据类型分类

1．基本类型

基本类型是C语言程序设计中的最小数据单元，可以说是原子数据类型，而其他数据类型（如结构体、共用体等）都可以使用这些基本类型。

2．构造类型

构造类型正如其名字一样，是在基本数据类型的基础上构造而成的复合数据类型，它可以用于表示更为复杂的数据。

3．空类型

空类型是一种特殊的数据类型，它是所有数据类型的基础。要注意的是，空类型并非无类型，它本身也是一种数据结构，常用在数据类型的转换和参数的传递过程中。

在 C 语言中，所有的数据都必须指定它的数据类型，它们大多有自己的类型标识符，如表 2-1 所示。

表 2-1　　数据类型及其标识符

数 据 类 型	标　识　符	数 据 类 型	标　识　符
整型	int	结构体	struct
字符型	char	共用体	union
浮点型	float（单精度）	空类型	void
	double（双精度）	数组类型	无
枚举型	enum	指针类型	无

2.2 基本数据类型

2.2.1 整型家族

变量是指在程序运行过程中其值可以发生变化的量。

1. 整型变量

整型变量包括短整型（short int）、整型（int）和长整型（long int），它们都分为有符号（signed）和无符号（unsigned）两种，在内存中是以二进制的形式存放的。每种类型的整数占有一定大小的地址空间，因此它们所能表示的数值范围也有所限制。

要注意的是，不同的计算机体系结构中这些类型所占比特数有可能是不同的，表 2-2 列出的是常见的 32 位机中整型家族各数据类型所占的比特数。

表 2-2　　整型家族各类型所占的比特数

类　　型	比　特　数	取 值 范 围
[signed] int	32	-2147483648～2147483647
unsigned int	32	0～4294967295
[signed] short [int]	16	-32768～32767
unsigned short [int]	16	0～65535
long [int]	32	-2147483648～2147483647
unsigned long [int]	32	0～4294967295

上表中“[]”内的部分是可以省略的，如短整型可写作“short”。它们三者之间只是遵循如下的简单规则。

```
短整型≤整型≤长整型
```

若要查看适合当前计算机的各数据类型的取值范围，可查看文件“limits.h”（通常在编译器相关的目录下），如下是“limits.h”的部分示例。

```
#include <features.h>
#include <bits/wordsize.h>
/* 一个“char”的位数  */
#  define CHAR_BIT      8
```

```
/* 一个“signed char”的最大值和最小值 */
#  define SCHAR_MIN      (-128)
#  define SCHAR_MAX      127

/* 一个“signed short int”的最大值和最小值 */
#  define SHRT_MIN       (-32768)
#  define SHRT_MAX       32767

/* 一个“signed int”的最大值和最小值 */
#  define INT_MIN        (-INT_MAX - 1)
#  define INT_MAX        2147483647

/* 一个“unsigned int”的最大值和最小值 */
#  define UINT_MAX       4294967295U

/* 一个“signed long int”的最大值和最小值 */
/*若是 64 位机*/
#  if __WORDSIZE = = 64
#   define LONG_MAX      9223372036854775807L
#  else
#   define LONG_MAX      2147483647L
#  endif
#  define LONG_MIN       (-LONG_MAX - 1L)

/* 一个“unsigned long int”的最大值和最小值 */
/*若是 64 位机*/
#  if __WORDSIZE = = 64
#   define ULONG_MAX     18446744073709551615UL
#  else
#   define ULONG_MAX     4294967295UL
#  endif
```

在嵌入式开发中，经常需要考虑的一点就是可移植性的问题。通常，字符是否为有符号数会带来两难的境地，因此，最佳妥协方案就是把存储于 int 型变量的值限制在 signed int 和 unsigned int 的交集中，这可以获得最大程度上的可移植性，同时又不牺牲效率。

2．整型常量

常量就是在程序运行过程中其值不能被改变的量。在 C 语言中，使用整型常量可以有八进制整数、十进制整数和十六进制整数 3 种，其中十进制整数的表示最为简单，不需要有任何前缀，在此就不再赘述。

八进制整数需要以“0”作为前缀开头，如下所示。

```
010     0762     0537     -0107
```

十六进制的整数需要以“0x”作为前缀开头，由于在计算机中数据都是以二进制来进行存放的，数据类型的表示范围位数也一般都是 4 的倍数，因此，将二进制数据用十六进制表示是非常方便的，在 Linux 的内核代码中，到处都可见到采用十六进制表示的整数。

下面示例的几句代码就是从 Linux 内核源码中摘录出来的（/arch/arm/mach-s3c2410）。读者在这里不用知道这些代码的具体含义，而只需了解这些常量的表示方法。

```
unsigned long s3c_irqwake_eintallow = 0x0000fff0L; (irq.c)
if (pinstate = = 0x02) {……} (pm.c)
config &= 0xff; (gpio.c)
```

可以看到，第 1 句代码是使用常量“0x0000fff0L”对变量 s3c_irqwake_eintallow 进行赋初值，第 2 句是比较变量 pinstate 的值和 0x02 是否相等，而第 3 句则是对 config 进行特定的运算。

细心的读者可以看到，常量“0x0000fff0L”在最后有大写的“L”，这并不是十六进制的表示范围，那么这个“L”又是什么意思呢？

这就是整型常量的后缀表示。正如前文中所述，整型数据还可分为“长整型”、“短整型”、“无符号数”，整型常量可在结尾加上“L”或“l”代表长整型，“U”或“u”代表无符号整型。前面的第一句代码中由于指明了该常量 0x0000fff0 是长整型的，因此需要在其后加上“L”。

要注意变量 s3c_irqwake_eintallow 声明为“unsigned long”并不代表赋值的常量也一定是“unsigned long”数据类型。

2.2.2 实型家族

实型家族也就是通常所说的浮点数，在这里也分别就实型变量和实型常量进行讲解。

1. 实型变量

实型变量又可分为单精度（float）、双精度（double）和长双精度（long double）3 种。表 2-3 列出的是常见的 32 位机中实型家族各数据类型所占的比特数。

表 2-3　　实型家族各类型所占比特数

类　型	比 特 数	有 效 数 字	取 值 范 围
float	32	6～7	-3.4×10^{-38}～3.4×10^{38}
double	64	15～16	-1.7×10^{-308}～1.7×10^{308}
long double	64	18～19	-1.2×10^{-308}～1.2×10^{308}

要注意的是，这里的有效数字是指包括整数部分的全部数字总数。它在内存中的存储方式是以指数的形式表示的，如图 2-2 所示。

图 2-2　实型变量的存储方式

由图 2-2 可以看出，小数部分所占的位（bit）越多，数的精度就越高；指数部分所占的位数越多，则能表示的数值范围就越大。下面程序就显示了实型变量的有效数字位数。

```
#include <stdio.h>

int main()
{
    float a;
    double b;
```

```
    a = 33333.33333;
    b = 33333.333333;

    printf("a=%f,b=%lf\n", a, b);

    return 0;
}
```

程序执行结果如下：

```
linux@ubuntu:~/book/ch2$ cc float.c  -Wall
linux@ubuntu:~/book/ch2$./a.out
a=33333.332031,b=33333.333333
```

可以看出，由于 a 为单精度类型，有效数字长度为 7 位，因此 a 的小数点后 4 位并不是原先的数据，而由于 b 为双精度类型，有效数字为 16 位。因此 b 的显示结果就是实际 b 的数值。

2．实型常量

浮点常量又称为实数，一般含有小数部分。

在 C 语言中，实数只有十进制的实数，它又分为单精度实数和双精度实数，它们的表示方法基本相同。实数有两种表示方法， 即一般形式和指数形式，所有浮点常量都被默认为 double 类型。表 2-4 概括了实型常量的表示方法。

表 2-4　　实型常量的表示方法

形　　式	表 示 方 法	举　　例
十进制表示	由数码 0～9 和小数点组成	0.0，0.25，5.789，0.13，5.0，300.
指数形式	〈尾数〉E(e)〈整型指数〉	3.0E5，-6.8e18

从表 2-4 可以看出，一般形式的实数基本形式如下：

```
[+|-]M.N
```

例如：3.1，-23.1112，3.1415926

指数形式的实数一般是由尾数部分、字母 e 或 E 和指数部分组成。当一个实数的符号为正号时，可以省略不写，其表示的一般形式如下：

```
[+|-]M.N<e|E>[+|-]T
```

例如：

1.176e+10 表示 1.176×10^{10}

- 3.5789e-8 表示 -3.5789×10^{-8}

通常表示特别大或特别小的数。

举例：一个水分子的质量约为 3.0×10^{-23}g，1 夸脱水大约有 950g，编写一个程序，要求输入水的夸脱数，然后显示这么多水中包含说少水分子。

示例程序如下：

```
#include <stdio.h>

int main(int argc,char **argv)
{
    float mass_mol = 3.0e-23;
    float mass_qt = 950;
    float quarts;
    float molecules;
```

```
    printf(“mass_mol=%f %e\n”, mass_mol, mass_mol);
    printf("Enter the number of quarts of water: ");
    scanf("%f", &quarts);
    molecules = quarts * mass_qt / mass_mol;
    printf("%f quarts of water contain %e(%f) molecules.\n",
            quarts, molecules);

    return 0;
}
```

程序执行结果如下：

```
linux@ubuntu:~/book/ch2$ cc water.c -o water  -Wall
linux@ubuntu:~/book/ch2$./water
mass_mol=0.000000 3.000000e-23
Enter the number of quarts of water: 1
1.000000 quarts of water contain 3.166667e+25(31666665471894750551343104.000000) molecules.
```

可以看出，float 不是一个确定的数值，比如写一个很小的数，用科学计数法可以表示出来，但以%f 作输出时显示为 0。对于一些特别大的数据，使用实数的指数形式，更简洁，可读性更好。

2.2.3 字符型家族

1. 字符变量

字符变量可以看作是整型变量的一种，它的标识符为“char”，一般占用一个字节（8bit），它也分为有符号和无符号两种，读者完全可以把它当成一个整型变量。当它用于存储字符常量时（稍后会进行讲解），实际上是将该字符的 ASCII 码值（无符号整数）存储到内存单元中。

实际上，一个整型变量也可以存储一个字符常量，而且也是将该字符的 ASCII 码值（无符号整数）存储到内存单元中。但由于取名上的不同，字符变量则更多地用于存储字符常量。以下一段小程序显示了字符变量与整型变量实质上是相同的。

```
#include <stdio.h>
int main()
{
      char a,b;
      int c,d;
      /*赋给字符变量和整型变量相同的整数常量*/
      a = c = 65;
      /*赋给字符变量和整型变量相同的字符常量*/
      b = d = 'a';
      /*以字符的形式打印字符变量和整型变量*/
      printf("char a = %c, int c = %c\n", a, c);
      /*以整数的形式打印字符变量和整型变量*/
      printf("char b = %d, int d = %d\n", b, d);

      return 0;
}
```

程序执行结果如下：

```
linux@ubuntu:~/book/ch2$ cc char.c -o char -Wall
linux@ubuntu:~/book/ch2$./char
char a = A, int c = A
char b = 97, int d = 97
```

由此可见，字符变量和整型变量在内存中存储的内容实质是一样的。

表 2-5 显示了字符型数据占用的比特数。

表 2-5　　字符型所占的比特数

类　　型	比　特　数	取 值 范 围
[signed] char	8	-128～127
unsigned char	8	0～255

示例程序如下：

```
#include <stdio.h>

int main()
{
    char ch1 = 129;
    unsigned ch2 = -1;

    printf("ch1=%c-%d,ch2=%c-%d\n",
            ch1, ch1, ch2, ch2);

    return 0;
}
```

程序执行结果如下：

```
linux@ubuntu:~/book/ch2$ cc char.c -o char -Wall
linux@ubuntu:~/book/ch2$./char
ch1=-127,ch2=-1
```

从程序结果可以看出，给字符型变量赋值越界时，编译程序没有语法错误，但是结果不准确。但是，当越界后如何处理，这和编译器有关，在这里，就不再进一步解释。

2. 字符常量

字符常量是指用单引号括起来的一个字符，如’a’、’D’、’+’、’?’等都是字符常量。以下是使用字符常量时容易出错的地方，请读者仔细阅读。

① 字符常量只能用单引号括起来，不能用双引号或其他括号。

② 字符常量只能是单个字符，不能是字符串。

③ 字符可以是字符集中任意字符。但数字被定义为字符型之后就不能参与数值运算。如‘5’和 5 是不同的。‘5’是字符常量，不能直接参与运算，而只能以其 ASCII 码值（053）来参与运算。

除此之外，C 语言中还存在一种特殊的字符常量——转义字符。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。表 2-6 就是常见的转义字符以及它们的含义。

表 2-6　　转义字符及其含义

字 符 形 式	含　　义	ASCII 代码

\n	回车换行	10
\t	水平跳到下一制表位置	9
\b	向前退一格	8
\r	回车，将当前位置移到本行开头	13
\f	换页，将当前位置移到下页开头	12
\\	反斜线符“\”	92
\’	单引号符	39
\ddd	1～3 位八进制数所代表的字符	
\xhh	1～2 位十六进制数所代表的字符	

示例程序如下：

```
#include <stdio.h>

int main()
{
    char c1 = 'a', c2 = 'b', c3 = 'c';
    char c4 = '\101', c5 = '\116';

    printf("a%c b%c\tabc%c\n", c1, c2, c3);
    printf("\t\b%c %c\n", c4, c5);

    return 0;
}
```

程序执行结果如下：

```
linux@ubuntu:~/book/ch2$ cc char2.c -Wall
linux@ubuntu:~/book/ch2$./a.out
aa bb   abcc
       A N
```

2.2.4 枚举家族

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 7 天，一年只有 12 个月，一个班每周有 6 门课程等。如果把这些量说明为整型、字符型或其他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，被定义为该枚举类型的变量取值不能超过定义的范围。

枚举类型是一种基本数据类型，而不是一种构造类型，因为它不能再分解为任何基本类型。

枚举类型定义的一般形式如下。

```
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。

下例中是嵌入式 Linux 的存储管理相关代码“/mm/sheme.c”中的实例，“sheme.c”中实际实现了一个 tmpfs 文件系统。

```
/* Flag allocation requirements to shmem_getpage and shmem_swp_alloc */
enum sgp_type {
        SGP_QUICK,          /*不要尝试更多的页表*/
        SGP_READ,           /*不要超过 i_size,不分配页表*/
        SGP_CACHE,          /*不要超过 i_size,可能会分配页表*/
        SGP_WRITE,          /*可能会超过 i_size,可能会分配页表*/
};
```

sgp_type 具体含义的说明比较冗长，在此读者主要学习 enum 的语法结构。这里的 sgp_type 是一个标识符，它所有可能的取值有 SGP_QUICK、SGP_READ、SGP_CACHE、SGP_WRITE，也就是枚举元素。这些枚举元素的变量实际上是以整型的方式存储的，这些符号名的实际值都是整型值。

比如，这里的 SGP_QUICK 是 0，SGP_READ 是 1，依此类推。在适当的时候，用户也可以为这些符号名指定特定的整型值，如下所示。

```
/* Flag allocation requirements to shmem_getpage and shmem_swp_alloc */
enum sgp_type {
        SGP_QUICK = 2,   /*不要尝试更多的页表*/
        SGP_READ  = 9,   /*不要超过 i_size,不分配页表*/
        SGP_CACHE = 19,  /*不要超过 i_size,可能会分配页表*/
        SGP_WRITE = 64,  /*可能会超过 i_size,可能会分配页表*/
};
```

2.2.5 指针家族

1. 指针的概念

C 语言之所以如此流行，其重要原因之一就在于指针，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，并能像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，是学习 C 语言中最重要的一环。能正确理解和使用指针是掌握 C 语言的一个标志。

在这里着重介绍指针的概念，指针的具体使用在后面的章节中会有详细的介绍。

何为指针呢？简单地说，指针就是地址。在计算机中，所有的数据都是存放在存储器中的。一般可以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 4 个内存单元（字节），字符量占 1 个内存单元（字节）等，这些在本章的 2.2.1 小节中已经进行了详细讲解。

为了正确地访问这些内存单元，必须为每个内存单元编号。根据一个内存单元的编号就可准确地找到该内存单元。内存单元的编号也叫做地址，通常也把这个地址称为指针。

内存单元的指针（地址）和内存单元的内容（具体存放的变量）是两个不同的概念。

图 2-3 就表示了指针的含义。

从图 2-3 中可以看出，0x00100000 等都是内存地址，也就是变量的指针，由于在 32 位机中地址长度都是 32bit，因此，无论哪种变量类型的指针都占 4 个字节。

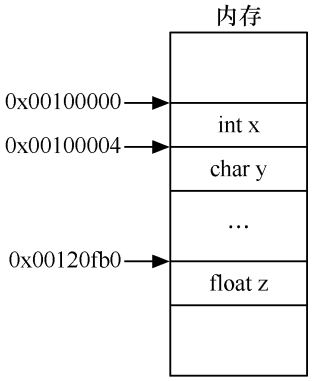

图 2-3　指针的含义

由于指针所指向的内存单元是用于存放数据的，而不同数据类型的变量占有不同的字节数，因此从图 2-3 中可以看出，一个整型变量占 4 个字节，故紧随其后的变量 *y* 的内存地址为变量 *x* 起始地址加上 4 个字节。

2．指针常量

事实上，在 C 语言中，指针常量只有唯一的一个 NULL（空地址）。虽然指针是一个诸如 0x00100011 这样的字面值，但是因为编译器负责把变量存放在计算机内存中的某个位置，所以程序员在程序运行前无法知道变量的地址。当一个函数每次被调用时，它的自动变量（局部变量）每次分配的内存位置都不同，因此，把非零的指针常量赋值给一个指针变量是没有意义的。

3．字符串常量

字符串常量看似是字符家族中的一员，但事实上，字符串常量与字符常量有着较大的区别。字符串常量是指用一对双引号括起来的一串字符，双引号只起定界作用，双引号括起的字符串中不能是双引号（“”）和反斜杠（\），它们特有的表示法在转义字符中已经介绍。例如“China”、“Cprogram”、“YES&NO”、“33312-2341”、“A”等都是合格的字符串常量。

在 C 语言中，字符串常量在内存中存储时系统自动在字符串的末尾加一个“串结束标志”，即 ASCII 码值为 0 的字符 NULL，通常用“\0”表示。因此在程序中，长度为 n 个字符的字符串常量，在内存中占有 n+1 个字节的存储空间。

例如，字符串“China”有 5 个字符，存储在内存中时占用 6 个字节。系统自动在字符串最后加上 NULL 字符，其存储形式为图 2-4 所示。

C	h	i	n	a	NULL

图 2-4　字符串常量的存储形式

要特别注意字符常量与字符串常量的区别，除了表示形式不同外，其存储方式也不相同。字符‘A’只占 1 个字节，而字符串常量“A”占 2 个字节。

本书之所以在指针家族处而不在字符家族中讲解字符串常量，是由于在程序中使用字符串常量会生成一个“指向字符串的常指针”。当一个字符串常量出现在一个表达式中时，表达式所引用的值是存储该字符串常量的内存首地址，而不是这些字符本身。

因此，用户可以把字符串常量赋值给一个字符类型的指针，用于指向该字符串在内存中的首地址。

2.3 变量与常量

2.3.1 变量的定义

在上一节中，读者学习了 C 语言中的基本数据类型。那么在程序中不同数据类型的变量如何使用呢？在 C 语言中使用变量采用先定义、后使用的规则，任何变量在使用前必须先进行定义。

变量定义的基本形式如下。

```
说明符（一个或多个） 变量或表达式列表
```

这里的说明符就是包含一些用于表明变量基本类型的关键字、存储类型和作用域。表 2-7 列举了一些常见基本数据类型变量的定义方式。

表 2-7 变量的定义方式

基 本 类 型	关 键 字	示 例
整型	int、unsigned、short、long	int a; unsigned long b;
浮点型	float、double	float a; double b;
字符型	char、unsigned	char a; unsigned char b;
枚举类型	enum	enum a;
指针类型	数据类型 *	int *a, *b; char *c;

通常，变量在定义时也可以将其初始化，如：

```
int i = 5;
```

这条语句实际上转化为两条语句：

```
int i;          /*定义*/
i = 5;          /*初始化*/
```

此外，指针的定义形式在这里需着重说明。

指针的定义形式为标识符加上“*”。有些读者习惯把“*”写在靠近数据类型的一侧，如：

```
int* a;
```

虽然编译器支持这种定义形式，但会在阅读代码时带来困扰，例如：

```
int* b, c, d;
```

读者会很自然地认为上面这条语句把 3 个变量都定义为指向整型的指针。事实上，只有变量 b 是整型指针，而 c、d 都是整型变量。因此，建议读者在定义指针变量时将“*”写在靠近变量名的一侧，如下所示。

```
int *b, *c, *d;
```

关于变量的定义和变量的声明是两个极易混淆的概念，在形式上也很接近。在对变量进行了定义后，存储器需要为其分配一定的存储空间，一个变量在其作用域范围内只能有一个定义。而变量的声明则不同，一个变量可以有多次声明，且存储器不会为其分配存储空间。在本书的稍后部分将会讲解它们使用上的区别。

2.3.2 typedef

typedef 是 C 语言的关键字，其作用是为一种数据类型定义一个新名字。这里的数据类型包括内部数据类型（如 int、char 等）和自定义的数据类型（如 struct 等）。

其基本用法如下所示。

```
typedef 数据类型 自定义数据类型
```

例如，用户可以使用以下语句。

```
typedef unsigned long uint32;
```

这样，就把标识符 uint32 声明为无符号长整型的类型了。之后，用户就可以用它来定义变量。

```
uint32 a;
```

此句等价于：

```
unsigned long a;
```

在大型程序开发中，typedef 的应用非常广泛。目的有两点，一是给变量一个易记且意义明确的新名字，二是简化一些比较复杂的类型声明。

在嵌入式的开发中，由于涉及可移植性的问题，typedef 的功能就更引人注目了。通过 typedef 可以为标识符取名为一个统一的名称，这样，在需要对这些标识符进行修改时，只需修改 typedef 的内容就可以了。

下面是“/include/asm-arm/type.h”里的内容。

```
#ifndef __ASSEMBLY__
/* 为有符号字符型取名为 s8 */
typedef signed char s8;
/* 为无符号字符型取名为 u8 */
typedef unsigned char u8;
/* 为有符号短整型取名为 s16 */
typedef signed short s16;
/* 为无符号短整型取名为 u16 */
typedef unsigned short u16;
/* 为有符号整型取名为 s32 */
typedef signed int s32;
/* 为无符号整型取名为 u32 */
typedef unsigned int u32;
/* 为有符号长长整型取名为 s64 */
typedef signed long long s64;
/* 为无符号长长整型取名为 u64 */
typedef unsigned long long u64;
```

2.3.3 常量的定义

1. const 定义常量

在 C 语言中，可以使用 const 来定义一个常量。常量的定义与变量的定义很相似，只需在变量名前加上 const 即可，如下所示。

```
int const a;
```

以上语句定义了 a 为一个整数常量。那么，既然 a 的值不能被修改，如何让 a 拥有一个值呢？

这里，一般有两种方法，其一是在定义时对它进行初始化，如下所示。

```
const int a = 10;
```

其二，在函数中声明为 const 的形参在函数被调用时会得到实参的值。

在这里需要着重讲解的是 const 涉及指针变量的情况，先看两个 const 定义。

```
const int *a;
int * const a;
```

在第一条语句中，const 用来修饰指针 a 所指向的对象，也就是说我们无法通过指针 a 来修改其指向的对象的值。但是 a 这个指针本身的值（地址）是可以改变的，即可以指向其他对象。

与此相反，在第二条语句中，const 修饰的是指针 a。因此，该指针本身（地址）的值是不可改变的，而该指针所指向的对象的值是可以改变的。

2. define 定义常量

define 实际是一个预处理指令，其实际的用途远大于定义常量这一功能。在这里，首先讲解 define 定义常量的基本用法，对于其他用途在本书的后续章节中会有详细介绍。

使用 define 定义常量实际是进行符号替换，其定义方法为

```
#define 符号名 替换列表
```

符号名必须符合标识符命名规则。替换列表可以是任意字符序列，如数字、字符、字符串、表达式等，例如：

```
#define MSG "I'm Antigloss!"      /*后面的所有 MSG 都会被替换为“I'm Antigloss!”*/
#define SUM 99                    /*后面的所有 SUM 都会被替换为 99*/
#define BEEP "\a"                     /*后面的所有 BEEP 都会被替换为“\a”*/
```

习惯上，人们用大写字母来命名符号名，而用小写字母来命名变量。

预处理指令#define 的最后面没有分号“;”，千万不要画蛇添足！

在 Linux 内核中，也广泛使用 define 来定义常量，如用于常见的出错处理的头文件中，“include/asm-generic/errno-base.h”就有如下定义：

```
#define EPERM           1       /*操作权限不足*/
#define ENOENT          2       /*没有该文件或目录*/
#define ESRCH           3       /*没有该进程*/
#define EINTR           4       /*被系统调用所中止*/
#define EIO             5       /*I/O 出错*/
#define ENXIO           6       /*没有这个设备或地址*/
#define E2BIG           7       /*命令列表太长*/
#define ENOEXEC         8       /*命令格式错误*/
```

2.3.4 作用域

变量的作用域定义：程序中可以访问一个指示符的一个或多个区域，即变量出现的有效区域，决定了程序的哪些部分通过变量名来访问变量。一个变量根据其作用域的范围可以分为函数原型作用域、局部变量和全局变量。

1. 函数原型作用域

函数原型中的参数，其作用域始于"("，结束于")"。

设有下列原型声明：

```
double Area(double radius);
```

radius 的作用域仅在于此，不能用于程序正文其他地方，因而可以省略。

2. 局部变量

在函数内部定义的变量称为局部变量。局部变量仅能被定义该变量的模块内部的语句所访问。换言之，局部变量在自己的代码模块之外是不可见的。

模块以左花括号开始，以右花括号结束。

对于局部变量，要了解的重要规则是，它们仅存在于定义该变量的执行代码块中，即局部变量在进入模块时生成（压入堆栈），在退出模块时消亡（弹出堆栈）。定义局部变量的最常见的代码块是函数，例如：

```
void func1 ()
{
        /*在 func1 中定义的局部变量 x*/
        int x;
        x = 10;
}
void func2 ()
{
        /*在 func2 中定义的局部变量 x*/
        int x;
        x = 2007;
}
```

整数变量 x 被定义了两次，一次在 func1 中，另一次在 func2 中。func1 和 func2 中的 x 互不相关，原因是每个 x 作为局部变量仅在被定义的模块内可见。

要注意的是，在一个函数内部可以在复合语句中定义变量，这些复合语句成为“分程序”或“程序块”，如下所示。

```
void func1()
{
        /*在 func1 中定义的局部变量 x*/
        int x;
        x = 10;
        ……
        {
             int c;  /*定义程序块内部的变量*/
             c = a + b;    /*变量 c 只在这两个括号内有效*/
        }
}
```

在上述的例子中，变量 c 只在最近的程序块中有效，离开该程序块就无效，并释放内存单元。

3．全局变量

与局部变量不同，全局变量贯穿整个程序，它的作用域为源文件，可被源文件中的任何一个函数使用。它们在整个程序执行期间保持有效。

全局变量定义在所有函数之外。

```
int a,b; /*全局变量*/

void f1()
{
……
}
float x,y; /*全局变量*/
int fz() /*函数 fz*/
{
……
}
```

全局变量 a、b 的作用域

全局变量 x、y 的作用域

```
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。x、y 定义在函数 f1 之后，在 f1 内没有对 x、y 的声明，所以它们在 f1 内无效。a、b 定义在源程序最前面，即所有函数之前，因此在 f1、f2 及 main 内不加声明就可使用。

可以看到，使用全局变量可以有效地建立起几个函数相互之间的联系。对于全局变量还有以下几点说明。

① 对于局部变量的定义和声明，可以不加区分，而对于全局变量则不然。全局变量的定义和全局变量的声明并不是一回事，全局变量定义必须在所有的函数之外，且只能定义一次，其一般形式为

```
[extern] 类型说明符 变量名，变量名……
```

其中方括号内的 extern 可以省去不写，例如：

```
int a,b;
```

等效于：

```
extern int a,b;
```

而全局变量声明出现在要使用该变量的各个函数内。在整个程序内，可能出现多次。全局变量声明的一般形式为

```
extern 类型说明符 变量名，变量名……
```

全局变量在定义时就已分配了内存单元，并且可做初始赋值。全局变量声明不能再赋初始值，只是表明在函数内要使用某外部变量。

② 外部变量可加强函数模块之间的数据联系，但是又使函数要依赖这些变量，因而使得函数的独立性降低。从模块化程序设计的观点来看这是不利的，因此在不必要时尽量不要使用全局变量。

③ 全局变量的内存分配是在编译过程中完成的，它在程序的全部执行过程中都要占用存储空间，而不是仅在需要时才开辟存储空间。

④ 在同一源文件中，允许全局变量和局部变量同名。在局部变量的作用域内，全局变量不起作用。因此，若在该函数中想要使用全局变量，则不能再定义一个同名的局部变量。

如有以下代码：

```
#include <stdio.h>
/*定义全局变量 i 并赋初值为 5*/
int i = 5;
int main()
{
/*定义局部变量 i，并未赋初值，i 的值不确定，由编译器自行给出*/
     int i;
/*打印出 i 的值，查看在此处的 i 是全局变量还是局部变量*/
     if(i != 5)
         printf ("it is local\n");
     printf ("i is %d\n",i);

     return 0;
}
```

程序执行结果如下：

```
linux@ubuntu:~/book/ch2$ cc test.c -Wall
linux@ubuntu:~/book/ch2$./a.out
it is local
```

```
i is 134518324
```

可以看到，i 的值并不是全局变量所赋的初值，而是局部变量的值。

⑤ 全局变量的作用域可以通过关键字 extern 扩展到整个文件或其它文件。

2.3.5 链接属性

组成一个程序的各个源文件分别被编译之后，所有的目标文件以及那些从一个或多个函数库中引用的函数链接在一起，形成可执行程序。如果相同的标识符出现在几个不同的源文件中时，它们是表示同一个变量还是不同的变量呢？这个就取决于标识符的链接属性。

1. 空链接

作用域为代码块作用域或者函数原型作用域的变量，为空链接。当具有空链接的相同的变量名出现在几个不同的源文件中时，他们是由其定义所在的代码块或函数原型所私有的，被当做独立不同的实体。

2. 内部链接

使用 static 声明的全局变量，属于文件作用域，在文件的任何地方都可以使用，这就是我们所说的内部链接。属于内部链接的变量在同一个源文件内的所有声明中指同一个变量，但位于不同源文件的多个声明则属于不同的变量。注意这里的 static 只对全局变量的声明，有改变链接属性的效果。若声明的是局部变量，是使这个变量成为静态变量，延长了变量的生命周期，使得变量在程序的整个运行过程中都存在，直到程序结束。

3. 外部链接

全局变量，可以在文件的任何地方使用，默认是外部链接。属于外部链接属性的变量，无论声明多少次、位于几个源文件都表示同一个变量。

```
int  g = 5;                          //  外部全局变量
static int do = 3;                 //  静态全局变量

int main(int argc , char * argv[])
{
……
}
```

全局变量 g，从定义处到文件结束，都可以被引用，也可以被其他文件通过关键字 extern 引用到。全局变量 do，由于有 static 修饰，限制了使用范围，只能在当前文件使用。

2.3.6 存储模型

变量是程序中数据的存储空间的抽象。变量的存储方式可分为静态存储和动态存储两种。

静态存储变量通常是在程序编译时就分配一定的存储空间并一直保持不变，直至整个程序结束。在上一部分中介绍的全局变量的存储方式即属于此类存储方式。

动态存储变量是在程序执行过程中使用它时才分配存储单元，使用完毕立即释放。典型的例子是函数的形参。在函数定义时并不给形参分配存储单元，只是在函数被调用时，才予以分配，调用函数完毕立即释放。如果一个函数被多次调用，则反复地分配、释放形参变量的存储单元。

从以上分析可知，静态存储变量是一直存在的，而动态存储变量则时而存在时而消失。因此，这种由于变量存储方式不同而产生的特性称为变量的生存期，生存期表示了变量存在的时间。

生存期和作用域是从时间和空间这两个不同的角度来描述变量的特性。这两者既有联系，又有区别。一个变量的存储方式究竟属于哪一种存储方式，并不能仅仅从作用域来判断，还应有明确的存储模型说明。

变量的存储模型由作用域、链接点及存储期三大属性来描述。其中，存储期描述的是变量在内存中的生存时间。存储模型也经常被表达为存储类，共有以下 5 中存储模型。

1．自动

变量的声明语法为

```
〈存储类型〉数据类型 变量名
```

auto 为存储类说明符，可以说明一个变量为自动变量。该类具有动态存储期、代码块的作用域和空链接。如果变量没有初始化，它的值是不确定的。

代码块或者函数头部定义的变量，可使用存储类修饰符 auto 来明确标识属于自动存储类型。若没有使用 auto 修饰，也属于自动存储类型。

例如：

```
{
    int i,j,k;
    char c;
    ……
}
```

等价于：

```
{
    auto int i,j,k;
    auto char c;
    ……
}
```

自动变量具有以下特点。

① 自动变量的作用域仅限于定义该变量的模块内。在函数中定义的自动变量，只在该函数内有效。在复合语句中定义的自动变量，只在该复合语句中有效。

② 自动变量属于动态存储方式，只有在定义该变量的函数被调用时才给它分配存储单元，开始它的生存期。函数调用结束，释放存储单元，结束生存期。因此函数调用结束之后，自动变量的值不能保留。在复合语句中定义的自动变量，在退出复合语句后也不能再使用，否则将引起错误。

③ 由于自动变量的作用域和生存期都局限于定义它的模块内（函数或复合语句内），因此不同的模块中允许使用同名的变量而不会混淆。即使在函数内定义的自动变量也可与该函数内部的复合语句中定义的自动变量同名，但读者应尽量避免使用这种方式。

例如：

```
int loop(int n)
{
  int m;
   m = 2;
      {
       int i, m ;                    //m,i 的作用域
       m = 20;
       for (i = m; i < n ; i ++)
           printf(……);
      }
      return m;                      //m 作用域，i 消失
```

```
}
```

2．寄存器

在一个代码块内（或在一个函数头部作为参量）使用修饰符 register 声明的变量属于寄存器存储类。register 修饰符暗示编译程序相应的变量将被频繁使用，如果可能的话，应将其保存在 CPU 的寄存器中，从而加快其存取速度。该类与自动存储类相似，具有自动存储期、代码块作用域和空连接。如果没有被初始化，它的值也是不确定的。

使用 register 修饰符有几点限制。

① register 变量必须是能被 CPU 寄存器所接受的类型，这通常意味着 register 变量必须是一个单个的值，并且其长度应小于或等于整型的长度。这与处理器的类型有关。

② 声明为 register 仅仅是一个请求，而非命令，因此变量仍然可能是普通的自动变量，没有放在寄存器中。

③ 由于变量有可能存储在寄存器中，因此不能用取地址运算符“&”获取 register 变量的地址。如果有这样的写法，编译器会报错。

④ 只有局部变量和形参可以作为 register 变量，全局变量不行。

⑤ 实际上有些系统并不把 register 变量存放在寄存器中，而优化的编译系统则可以自动识别使用频繁的变量而把他们放在寄存器中。

3．静态、空链接

静态变量的类型说明符：static。在一个代码块内使用存储类修饰符 static 声明的局部变量属于静态空连接存储类。该类具有静态存储时期、代码块作用域和空链接。

静态变量的存储空间是在编译完成后就分配了，并且在程序运行的全部过程中都不会撤销。这里要区别的是，属于静态存储方式的变量不一定就是静态变量。

例如外部变量虽属于静态存储方式，但不一定是静态变量，必须由 static 加以定义后才能称为静态外部变量，或称静态全局变量。

图 2-5 显示了静态变量和动态变量的区别。

静态变量可分为静态局部变量和静态全局变量。

静态局部变量属于静态存储方式，它具有以下特点。

① 静态局部变量在函数内定义，它的生存期为整个程序执行期间，但是其作用域仍与自动变量相同，只能在定义该变量的函数内使用该变量。退出该函数后，尽管该变量还继续存在，但不能使用它。图 2-6 所示是静态局部变量的生存期及作用域示意图。

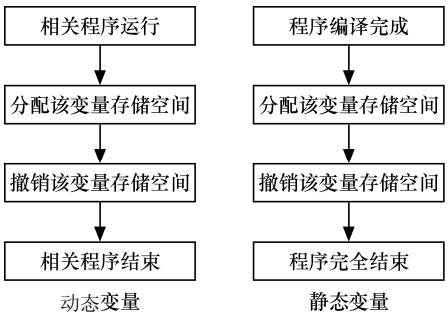

图 2-5　静态变量和动态变量

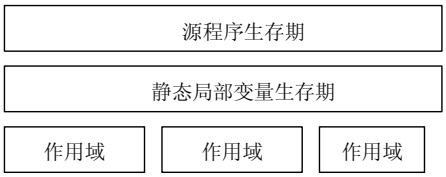

图 2-6　静态局部变量的生存期及作用域

② 可以对构造类静态局部量赋初值，例如数组。若未赋初值，则由系统自动初始化为 0。

③ 基本数据类型的静态局部变量若在说明时未赋初值，则系统自动赋予 0。而对自动变量不赋初值，其值是不确定的。根据静态局部变量的特点，可以看出它是一种生存期为整个程序运行期的变量。虽然离开定义它的函数后不能使用，但如再次调用定义它的函数时，它又可以继续使用，并且保留了上次被调用后的值。

因此，当多次调用一个函数且要求在调用之前保留某些变量的值时，可以考虑采用静态局部变量。虽然用全局变量也可以达到上述目的，但全局变量有时会造成意外的副作用，因此仍以采用静态局部变量为宜。

4．静态、外部链接

未使用 static 修饰的全局变量属于静态、外部链接存储类。具有静态存储时期、文件作用域和外部链接。仅在编译时初始化一次。如未明确初始化，它的字节也被设定为 0。在使用外部变量的函数中使用 extern 关键字来再次声明。如果是在其他文件中定义的，则必须使用 extern。

5．静态、内部链接

全局变量在关键字之前再冠以 static 就构成了静态的全局变量，属于静态、内部链接存储类。与静态、外部链接存储类不同的是，具有内部链接，使得仅能被与它在同一个文件的函数使用。这样的变量也是仅在编译时初始化一次。如未明确初始化，它的字节被设定为 0。

这两者的区别在于非静态全局变量的作用域是整个源程序，但当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的；而静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其他源文件中不能使用。

由于静态全局变量的作用域局限于一个源文件内，只能被该源文件内的函数使用，因此可以避免在其他源文件中引起错误。

图 2-7 是静态全局变量及非静态全局变量的区别示意图。

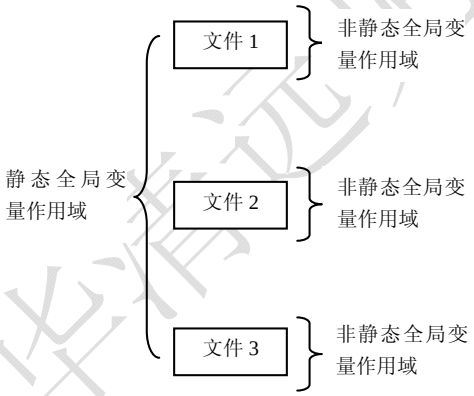

图 2-7　静态全局变量及非静态全局变量的区别

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

例如：源文件 a.c

```
int a = 10;                          //全局变量，静态外部链接
static int b = 20;                       //静态全局变量，静态内部链接

int f()
{
    int m = 30;
    return m;
}
```

源文件 b.c

```
#include <stdio.h>
extern int a;                    //使用 extern 关键字，声明外部变量
extern int b;                  //使用 extern 关键字，声明外部变量
```

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

    return 0;
}
```

编译程序，出现以下错误：

```
linux@ubuntu:~/book$ cc *.c
/tmp/cc25tJCv.o: In function `main':
static_b.c:(.text+0x22): undefined reference to `b'
collect2: ld returned 1 exit status
```

分析：本例说明了，普通全局变量，是外部链接，可以被其它文件引用。有 static 关键字修饰的静态全局变量，是内部链接，限制了变量只在当前文件使用。

2.4 预处理

本书的第 1 章已介绍过编译过程中的预处理阶段。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）之前所做的工作。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。当编译一个程序时，系统将自动调用预处理程序对程序中的“#”号开头的预处理部分进行处理，处理完毕之后可以进入源程序的编译阶段。

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。本节介绍最常用的几种预处理功能。

2.4.1 预定义

在 C 语言源程序中允许用一个标识符来表示一串符号，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，都用宏定义中的符号串去替换，这称为宏替换或宏展开。

1. 预定义符号

在 C 语言中，有一些预处理定义的符号串，它们的值是字符串常量，或者是十进制数字常量，通常在调试程序时用于输出源程序的各项信息，表 2-8 归纳了这些预定义符号。

表 2-8 预定义符号表

符　　号	示　　例	含　　义
__FILE__	/home/david/hello.c	正在预编译的源文件名
__LINE__	5	文件当前行的行号
__FUNCTION__	main	当前所在的函数名
__DATE__	Mar 13 2009	预编译文件的日期
__TIME__	23:04:12	预编译文件的时间
__STDC__	1	如果编译器遵循 ANSI C，则值为 1

这些预定义符号通常可以在程序出错处理时使用。下面的程序显示了这些预定义符号的基本用法。

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

```
int main()
{
    printf("The file is %s\n", __FILE__);
    printf("The line is %d\n", __LINE__);
    printf("The function is %s\n", __FUNCTION__);
    printf("The date is %s\n", __DATE__);
    printf("The time is %s\n", __TIME__);

    return 0;
}
```

要注意的是，这些预定义符号中__LINE__和__STDC__是整数常量，其他都是字符串常量，该程序的输出结果如下所示。

```
The file is /home/david/hello.c
The line is 5
The function is main
The date is Mar 13 2009
The time is 23:08:42
```

2. 宏定义

以上是 C 语言中自带的预定义符号，除此之外，用户自己也可以编写宏定义。宏定义是由源程序中的宏定义#define 语句完成的，而宏替换是由预处理程序自动完成的。在 C 语言中，宏分为带参数和不带参数两种，下面分别讲解这两种宏的定义和使用。

（1）无参宏定义

无参宏定义的宏名（也就是标识符）后不带参数，其定义的一般形式为

```
#define 标识符 字符串
```

其中，

#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。

前面介绍过的符号常量的定义就是一种无参宏定义。此外，用户还可对程序中反复使用的表达式进行宏定义，例如：

```
#define M (y + 3)
```

这样就定义了 M 表达式为“(y + 3)”，在此后编写程序时，所有的“(y + 3)”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替换，即用“(y + 3)”表达式去替换所有的宏名 M，然后再进行编译。

```
#include <stdio.h>

#define M (y + 3)
int main()
{
      int s, y;
      printf("input a number: ");
      scanf("%d", &y);
      s = 5 × M;
      printf("s = %d\n", s);
```

```
        return 0;
    }
```

在上例程序中首先进行宏定义，定义 M 为表达式“(y + 3)”，在“s=5×M”中作了宏调用，在预处理时经宏展开后该语句变为

```
    s = 5 ×(y + 3)
```

这里要注意的是，在宏定义中表达式“(y + 3)”两边的括号不能少，否则该语句展开后就成为如下所示：

```
    s = 5 × y + 3
```

这样显然是错误的，通常把这种现象叫做宏的副作用。

对于宏定义还要说明以下几点。

① 宏定义用宏名来表示一串符号，在宏展开时又以该符号串取代宏名，这只是一种简单的替换，符号串中可以包含任何字符，可以是常数，也可以是表达式，预处理程序对它不做任何检查。如有错误，只能在编译已被宏展开后的源程序时发现。

② 宏定义不是声明或语句，在行末不必加分号，如加上分号则连分号也一起替换。

③ 宏定义的作用域包括从宏定义命名起到源程序结束，如要终止其作用域可使用#undef 命令来取消宏作用域，例如：

```
    #define PI 3.14159
    func1()
    {
        ……
    }
    #undef PI
    func2()
    /*表示 PI 只在 func1 函数中有效，在 func2 函数中无效*/
```

④ 宏名在源程序中若用引号括起来，则预处理程序不对其进行宏替换。

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

上例中定义宏名 OK 表示 100，但在 printf 语句中 OK 被引号括起来，因此不做宏替换。

⑤ 宏定义允许嵌套，在宏定义的符号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替换。

⑥ 习惯上宏名用大写字母表示，以便与变量区别，但也允许用小写字母表示。

⑦ 对输出格式做宏定义，可以减少编写麻烦，例如：

```
    #include <stdio.h>

    #define P printf
    #define D "%d\n"
    #define F "%f\n"
    int 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);

        return 0;
    }
```

（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，而且要用实参去替换形参。

带参宏定义的一般形式为

```
#define 宏名(形参表) 字符串
```

在字符串中含有多个形参。带参宏调用的一般形式为

```
宏名(实参表);
```

例如：

```
#define M(y) y + 3 /*宏定义*/
```

若想调用以上宏，可以采用如下方法：

```
K = M(5); /*宏调用*/
```

在宏调用时，用实参 5 代替宏定义中的形参 y，经预处理宏展开后的语句为

```
K = 5 + 3
```

以下这段程序就是常见的比较两个数大小的宏表示，如下所示。

```
#include <stdio.h>

#define MAX(a,b) (a > b)?a:b /*宏定义*/
int main()
{
    int x = 10, y = 20, max;
    max = MAX(x, y); /* 宏调用 */
    printf("max = %d\n", max);
    return 0;
}
```

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。在程序中“max = MAX(x, y);”为宏调用，实参 x、y 将替换形参 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。

由于宏定义非常容易出错，因此，对于带参的宏定义有以下问题需要特别说明。

① 带参宏定义中，宏名和形参表之间不能有空格出现。例如：

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

写为

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

这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，因此不必做类型定义。这与函数中的情况是不同的。在函数中，形参和实参是两个不同的量，各有各的作用域，调用时要把实参值赋予形参，进行值传递。而在带参宏定义中，只是符号替换，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，例如：

```
#define SQ(y) (y)*(y)           /*宏定义*/
sq = SQ(a+1);                   /*宏调用*/
```

上例中第一行为宏定义，形参为 y；而在宏调用中实参为“a+1”，是一个表达式，在宏展开时，用“a + 1”替换 y，再用“(y) * (y)”替换 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，而宏替换中对实参表达式不做计算直接地照原样替换。

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)”表达式中的 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：

```
#define SQ(y) y * y          /*宏定义无括号*/
sq = SQ(a + 1);              /*宏调用*/
```

由于替换只做简单的符号替换而不做其他处理，因此在宏替换之后将得到以下语句。

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

这显然与题意相违背，因此参数两边的括号是不能少的。

其实，宏定义即使在参数两边加括号也还是不够的，例如：

```
#define SQ(y) (y) * (y)      /*宏定义有括号*/
sq = 160 / SQ(a + 1);            /*宏调用依然出错*/
```

读者可以分析一下宏调用语句，在宏替换之后变为

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

由于“/”和“*”运算符优先级和结合性相同，所以先做 160 / (a + 1)，再将结果与(a + 1)相乘，所以程序运行的结果依然是错误的。那么，究竟怎样进行宏定义才能正确呢？

下面是正确的宏定义。

```
#define SQ(y) ((y) * (y))        /*正确的宏定义*/
sq = 160 / SQ(a + 1);            /*宏调用结果正确*/
```

以上讨论说明，对于宏定义不仅应在参数两侧加括号，还应在整个符号串外加括号。

带参宏定义和带参函数很相似，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。

例如有以下两段程序，第一个程序是采用调用函数的方式来实现的。

```
#include <stdio.h>

/*程序 1，函数调用*/
int SQ(int y)                        /*函数定义*/
{
    Return (y × y);
}

int main()
{
    int i = 1;
    while (i <= 5)
    {
        printf("%d ", SQ(i++));          /*函数调用*/
    }

    return 0;
}
```

下面的第二个程序是采用宏定义的方式来实现的。

```
/*程序 2，宏定义*/
#include <stdio.h>
```

```
#define SQ(y) ((y)×(y))                    /*宏定义*/

int main()
{
    int i = 1;
    while (i <= 5)
    {
        printf("%d ", SQ(i++));            /*宏调用*/
    }

    return 0;
}
```

可以看到，不管是形参、实参还是具体的表达式都是一样的，但运行的结果却截然不同，函数调用的运行结果为

```
linux@ubuntu:~/book/ch2$ cc test.c -Wall
linux@ubuntu:~/book/ch2$./a.out
1 4 9 16 25
```

而宏调用的运行结果却是：

```
linux@ubuntu:~/book/ch2$ cc test.c -Wall
linux@ubuntu:~/book/ch2$./a.out
1 9 25
```

这是为什么呢？请读者先自己思考再看下面的分析。

在第一个程序中，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。

在第二个中宏调用时，实参和形参只作替换，因此 SQ(i++)被替换为((i++) × (i++))。在第一次循环时，由于在其计算过程中 i 值一直为 1，两相乘的结果为 1，然后 i 值两次自增 1，变为 3。

在第二次循环时，i 已有初值为 3，同理相乘的结果为 9，然后 i 再两次自增 1 变为 5。进入第 3 次循环，由于 i 值已为 5，所以这将是最后一次循环。相乘的结果为 25。i 值再两次自增 1 变为 7，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，在本质上却是完全不同的，表 2-9 总结了宏与函数的不同之处。

表 2-9　　宏与函数的不同之处

属　　性	宏	函　　数
处理阶段	预处理阶段，只是符号串的简单的替换	编译阶段
代码长度	每次使用宏时，宏代码都被插入到程序中。因此，除了非常小的宏之外，程序的长度都将被大幅增长	(除了 inline 函数之外)函数代码只出现在一个地方，每次使用这个函数，都只调用那个地方的同一份代码
执行速度	更快	存在函数调用/返回的额外开销(inline 函数除外)
操作符优先级	宏参数的求值是在所有周围表达式的上下文环境中，除非它们加上括号，否则邻近操作符的优先级可能会产生不可预料的结果	函数参数只在函数调用时求值一次，它的结果值传递给函数，因此，表达式的求值结果更容易预测
参数求值	参数每次用于宏定义时，它们都将重新求值。由于多次求值，具有副作用	参数在函数被调用前只求值一次，在函数中多次使用参数并不会导致多

	的参数可能会产生不可预料的结果	种求值问题，参数的副作用不会造成任何特殊的问题
参数类型	宏与类型无关，只要对参数的操作是合法的，它可以使用任何参数类型	函数的参数与类型有关，如果参数的类型不同，就需要使用不同的函数，即使它们执行的任务是相同的

2.4.2 文件包含

文件包含是 C 语言预处理程序的另一个重要功能，文件包含命令行的一般形式为

```
#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>
```

文件包含语句的功能是把指定的文件插入该语句行位置，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写。有些公用的符号常量、宏、结构、函数等的声明或定义可单独组成一个文件，在其他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写那些公用量，从而节省时间，并减少错误。

这里，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：使用尖括号表示在系统头文件目录中去查找（头文件目录可以由用户来指定）；使用双引号则表示首先在当前的源文件目录中查找，若未找到才到系统头文件目录中去查找。用户编程时可根据自己文件所在的位置来选择其中一种形式。

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别的文件。

2.4.3 条件编译

预处理程序提供了条件编译的功能，可以按不同的条件去编译不同的程序代码，从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 3 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 语句定义过，则编译程序段 1，否则编译程序段 2。如果没有程序段 2（为空），本格式中的#else 可以没有。例如有以下程序。

```
#include <stdio.h>
#define NUM OK /*宏定义*/
```

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

    ps = (struct stu*)malloc(sizeof(struct stu));
    ps->num = 102;
    ps->name="David";
    ps->score=92.5;
#ifdef NUM /*条件编译，若定义了 NUM，则打印以下内容*/
    printf("Number = %d\nScore = %f\n", ps->num, ps->score);
#else /*若没有定义 NUM，则打印以下内容*/
    printf("Name=%s\n",ps->name);
#endif
    free(ps);

    return 0;
}
```

该程序的运行结果如下：

```
linux@ubuntu:~/book/ch2$ cc test.c -Wall
linux@ubuntu:~/book/ch2$./a.out
Number = 102
Score = 92.500000
```

在程序中根据 NUM 是否被定义来决定编译哪一个 printf 语句。因为在程序的第二行中定义了宏 NUM，因此应编译第一个 printf 语句，运行结果则为输出学号和成绩。

在此程序中，宏 NUM 是符号串“OK”的别名，其实也可以为任何符号串，甚至不给出任何符号串，如下所示。

```
#define NUM
```

这样也具有同样的意义。读者可以试着将本程序中的宏定义去掉，看一下程序的运行结果，这种形式的条件编译通常用在调试程序中。在调试时，可以将要打印的信息用“#ifdef __DEBUG__”语句包含起来，在调试完成之后，可以直接去掉宏定义“#define __DEBUG__”，这样就可以做成产品的发布版本了。条件编译语句和宏定义语句一样，在#ifdef 语句后不能加分号（;）。

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 语句定义过，则编译程序段 1，否则编译程序段 2，这与第一种形式的功能正好相反。

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，如果常量表达式的值为真（非 0），则编译程序段 1，否则编译程序段 2。因此可以使程序在不同条件下，完成不同的功能。

```
#include <stdio.h>

#define IS_CURCLE 1
int main()
{
    float c = 2, r, s;
#if IS_CURCLE
    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
    return 0;
}
```

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，将 IS_CURCLE 定义为 1，因此在条件编译时，只编译计算和输出圆面积的代码部分。

上面介绍的程序的功能可以使用条件语句来实现。但是使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，也比较麻烦。采用条件编译，则可以根据编译条件选择性地进行编译，生成的目标程序较短，尤其在调试和发布不同版本时非常有用。

2.5 需要注意的问题

嵌入式开发很重要的一个问题就是可移植性的问题。Linux 是一个可移植性非常好的系统，这也是嵌入式 Linux 能够迅速发展起来的一个主要原因。所以，嵌入式 Linux 在可移植性方面所做的工作是非常值得学习的。本节结合嵌入式 Linux 实例来讲解嵌入式开发在可移植性方面需要考虑的问题。

2.5.1 字长和数据类型

能够由计算机一次完成处理的数据称为字，不同体系结构的字长通常会有所区别，例如，现在通用的处理器字长为 32 位。

为了解决不同的体系结构有不同字长的问题，嵌入式 Linux 中给出两种数据类型，其一是不透明数据类型，其二是长度明确的数据类型。

不透明数据类型隐藏了它们的内部格式或结构。在 C 语言中，它们就像黑盒一样，开发者们利用 typedef 声明一个类型，把它叫做不透明数据类型，并希望其他开发者不要重新将其转化为对应的那个标准 C 语言类型。

例如用来保存进程标识符的 pid_t 类型的实际长度就被隐藏起来了，尽管任何人都可以揭开它的面纱，因为其实它就是一个 int 型数据。

长度明确的数据类型也很常见。作为一个程序员，通常在程序中需要操作硬件设备，这时就必须明确知道数据的长度。

嵌入式 Linux 内核在“asm/types.h”中定义了这些长度明确的类型，表 2-10 是这些类型的完整说明。

表 2-10　　类型说明

类　型	描　述	类　型	描　述
s8	带符号字节	s32	带符号 32 位整数
u8	无符号字节	u32	无符号 32 位整数
s16	带符号 16 位整数	s64	带符号 64 位整数
u16	无符号 16 位整数	u64	无符号 64 位整数

这些长度明确的数据类型大部分是通过 typedef 对标准的 C 语言类型进行映射得到的。在嵌入式 Linux 中的“/asm-arm/types.h”就有如下定义。

```
typedef __signed__ char __s8;
typedef unsigned char __u8;
typedef __signed__ short __s16;
typedef unsigned short __u16;
typedef __signed__ int __s32;
typedef unsigned int __u32;
typedef __signed__ long long __s64;
typedef unsigned long long __u64;
```

2.5.2　数据对齐

对齐是内存数据与内存中的相对位置相关的问题。如果一个变量的内存地址正好是它长度的整数倍，它就被称作是自然对齐的。例如，对于一个 32 位（4 个字节）类型的数据，如果它在内存中的地址刚好可以被 4 整除（最低两位是 0），那它就是自然对齐的。

一些体系结构对对齐的要求非常严格。通常基于 RISC 的系统载入未对齐的数据会导致处理器陷入一种可处理的错误，还有一些系统可以访问没有对齐的数据，但性能会下降。编写可移植性高的代码要避免对齐问题，保证所有的类型都能够自然对齐。

2.5.3　字节序

字节顺序是指一个字中各个字节的顺序，有大端模式（big-endian）和小端模式（little-endian）。大端模式是指在这种格式中，字数据的高字节存储在低地址中，而字数据的低字节则存放在高地址中。小端模式与大端存储格式相反，在小端存储格式中，低地址中存放的是字数据的低字节，高地址存放的是字数据的高字节。

ARM 体系结构支持大端模式和小端模式两种内存模式。

请看下面的一段代码，通过这段代码我们可以查看一个字（通常为 4 个字节）数据的每个字节在内存中的分布情况，即可以分辨出当前系统采用哪种字节顺序模式。

```
typedef unsigned char byte;
typedef unsigned int  word;
word val32 = 0x87654321;
byte val8 = *((byte*)&val32);
```

这段代码在小端模式和大端模式下其运行结果分别为 val8 = 0x21 和 val8 = 0x87。其实，变量 val8 所在的地方是 val32 的低地址，因此若 val8 值为 val32 的低字节（0x21），则本系统是小端模式；若 val8 值为 val32 的高字节（0x87），则本系统是大端模式，如图 2-8 所示。

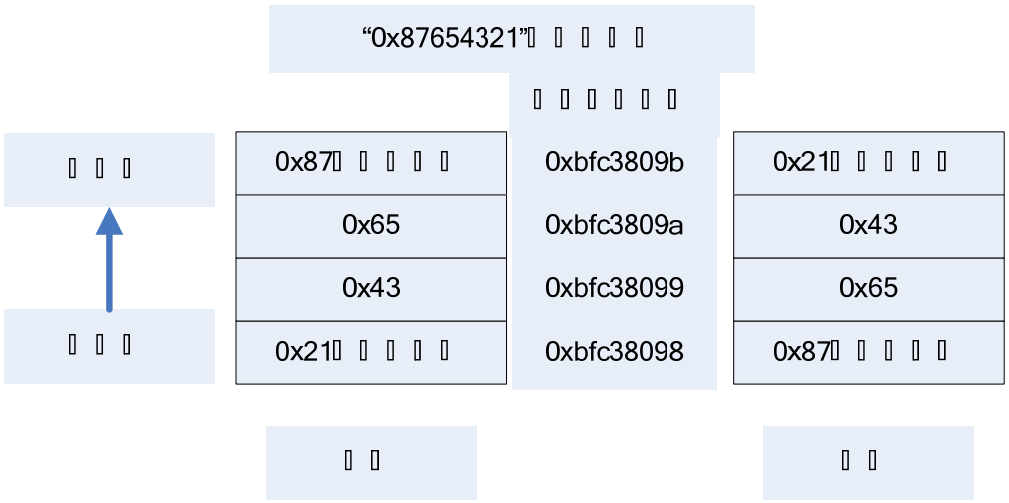

图 2-8　小端模式和大端模式

这种功能也可以用 union 联合体来实现，建议读者动手编程尝试一下。

小结

本章是嵌入式 Linux C 语言中最为基础的一章。

首先，本章中讲解了 C 语言的基本数据类型，在这里读者要着重掌握的是各种数据类型的区别和联系以及它们的内存占用情况。

其次，本章讲解了基本的常量和变量，typedef 用法，预处理。这里需要着重掌握的是变量的作用域和存储方式，要理解 static 限制符的作用。

最后，本章讲解了在在嵌入式开发中，考虑到程序的可移植性，关于数据的一些特别需要注意的问题，包括字长和数据类型、数据对齐、字节序。

本章每一部分都以 ARM Linux 内核实例进行讲解，读者可以看到在 Linux 内核中是如何组织和使用这些基本元素的。

思考与练习

1．下面这个表达式的类型和值是什么？

```
(float)(25/15)
```

2．思考：假如有一个程序，它把一个 long 整型变量复制给一个 short 整型变量。当编译这种程序时会发生什么情况？当运行程序时会发生什么情况？你认为其他编译器的结果也是这样吗？

3．判断下面的语句是否正确。

假定一个函数 a 声明的一个自动变量 x，你可以在其他函数内访问变量 x，只要你使用了下面的声明。

```
extern int x;
```

联系方式

集团官网：www.hqyj.com　　嵌入式学院：www.embedu.org　　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　　物联网学院：www.topsight.cn　　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

全国免费咨询电话：400-706-1880

双休日及节假日请致电值班手机：15010390966

在线咨询：张老师 QQ（619366077），王老师 QQ（2814652411），杨老师 QQ（1462495461）

企业培训洽谈专线：010-82600901

院校合作洽谈专线：010-82600350，在线咨询：QQ（248856300）