嵌入式培训专家

10年口碑积累，成功培养50000多名研发工程师，铸就专业品牌形象

华清远见的企业理念是不仅要做良心教育、做专业教育，更要做受人尊敬的职业教育。

《嵌入式Linux C语言开发》

作者：华清远见

专业始于专注 卓识源于远见

第2章 嵌入式Linux C语言基础

本章目标

在上一章中，读者了解了嵌入式的基本概念，学习了如何搭建嵌入式Linux开发环境，并且掌握了如何使用嵌入式Linux C语言相关开发工具。本章主要介绍嵌入式Linux C语言的基础知识。通过本章的学习，读者将会掌握如下内容：

- C语言的基本数据类型
- 变量的定义、作用域及存储方式
- 常量的定义方式
- 运算符和表达式
- 程序结构和控制语句
- 数组、结构体和指针
- 函数的定义和调用
- attribute机制简介
- 系统调用和API

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,同时提供了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,美国国家标准化协会(**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,并与 1989 年颁布，被称为 **ANSI C** 或是 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 项目始创于一九八四年，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，他是 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 以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、更安全的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 上基础上进行了扩展。在了解这些扩展之前，我们先简单回顾一下标准 C。

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。

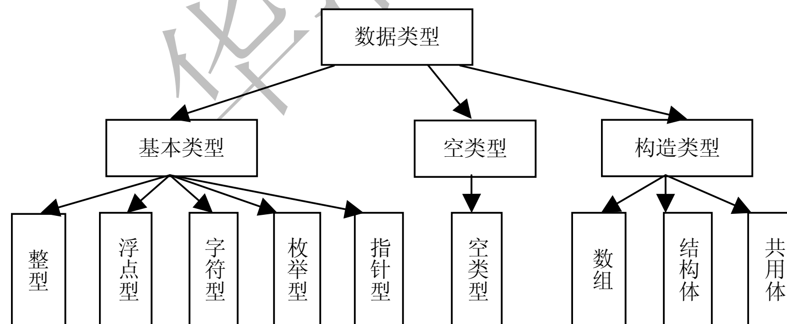


图 2.1 常见数据类型分类

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。
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。
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，它们有些有各自的类型标识符，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

华清远见

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

1．整型变量

变量是指在程序运行过程中其值可以发生变化的量。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　　不同类型整数所占的比特数

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”。从上表读者可以看到，短整型并不一定比整型短，它们 3 者之间只是遵循如下的简单规则：

短整型≤整型≤长整型

读者若想要查看适合当前机器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>
/* Number of bits in a 'char'.  */
#  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
```

可移植性提示 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就会带来两难的境地，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 signed 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，使用整型常量可以用八进制整数、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地，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，在第一句代码是使用常量“0x0000fff0L”对变量 s3c_irqwake_eintallow 进行赋初值，第二句是比较变量 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。

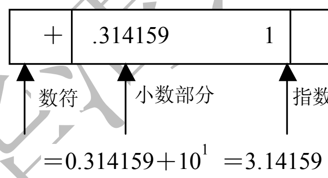


图 2.2 实型变量的存储方式

由上图可以看出，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>
void main(){
      float a;
      double b;
      /*单精度，有效数字为 7*/
      a=33333.33333;
      /*双精度，有效数字为 16*/
      b=33333.333333;
      printf("a = %f\, b = %f\n",a,b);
}
```

该程序的运行结果如下所示：

```
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，因此 b 的显示结果就是实际 b 的数值。

2．实型常量

在C语言中，实型常量只采用十进制。它有两种形式：十进制数形式和指数形式，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，.25，5.789，0.13，5.0，300.
指数形式	<尾数>E(e) <整型指数>	3.0E5，−6.8e18

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（8bits），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>
void 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);
}
```

该程序的运行结果如下所示：

```
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。

2．字符常量

字符常量是指用单括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。

- 字符常量只能用单引号括起来，不能用双引号或其他括号。
- 字符常量只能是单个字符，不能是字符串。

➢ 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 2.5 就是常见的转义字符以及它们的含义。

表 2.5 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	

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。

注意 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。

华清远见

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。

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”中实际是实现了一个 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 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。

注意 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。

下图 2.3 就表示了指针的含义：

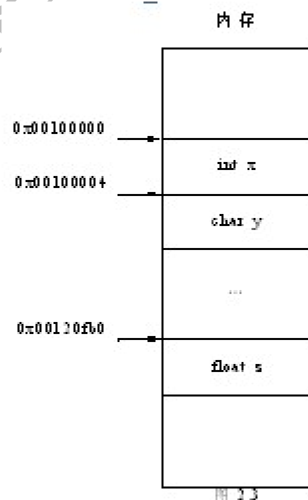


图 2.3

从该图中可以看出，如 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。

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。

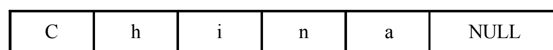
2．指针常量

华清远见

事实上，在 C 语言中，指针常量只有惟一的一个 NULL（空地址）。虽然指针是一个诸如 0x00100011 这样的字面值，但因为是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村村中的首地址。

2.3　变量与常量

2.3.1　变量的定义

1．定义形式

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是：

```
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 2.6 列举了一些常见基本数据类型变量的定义方式。

表 2.6　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;
```

易混淆点 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．变量的作用域

变量的作用域是变量可见的区域，这种变量有效性的范围称变量的作用域。变量的作用域是由变量的标识符作用域所决定的。一个变量根据其作用域的范围可以分为局部变量和全局变量。

（1）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。

切记 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：

```
func1()
{
        /*在 func1 中定义的局部变量 x*/
        int x;
        x = 10;
}
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，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知。

华清远见

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：

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

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。

（2）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*/
的作用域
int fz() /*函数 fz*/
{
……
}
main() /*主函数*/
{
……
}/
```

全局变量 x、y 的作用域

全局变量 a、b 的作用域

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。

➢ 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，并且可作初始赋值。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。

- 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。
- 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。
- 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，查看再此处的 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);
}
```

该程序的运行结果如下所示：

```
it is local
i is 134518324
```

可以看到，i 的值并不是全部变量所赋的初值，而是局部变量的值。

3. 变量的存储方式

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，还应有明确的存储类型说明。

在 C 语言中，对变量的存储类型说明有以下 4 种：

- auto 自动变量。
- static 静态变量。
- register 寄存器变量。

➢ extern 外部变量。

自动变量和寄存器变量属于动态存储方式，外部变量和静态变量属于静态存储方式。在介绍了变量的存储类型之后，可以知道对一个变量的说明不仅应说明其数据类型，还应说明其存储类型。因此变量说明的完整形式应为：

```
存储类型说明符 数据类型说明符 变量名，变量名…;
```

例如：

```
static int a,b;               /*说明 a,b 为静态整型变量*/
auto char c1,c2;              /*说明 c1,c2 为自动字符型变量*/
static int a[5]={1,2,3,4,5};  /*说明 a 为静态整型数组*/
extern int x,y;               /*说明 x,y 为外部整型变量*/
```

下面就分别就这 4 种存储类型进行说明。

（1）自动变量的类型说明符：auto

这种存储类型是 C 语言程序中使用最广泛的一种类型。C 语言规定，函数内凡未加存储类型说明的变量均视为自动变量，也就是说自动变量可省去说明符 auto。在前面的程序中所定义的变量只要未加存储类型说明符的都是自动变量，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。

➢ 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。

➢ 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。

➢ 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，但读者应尽量避免这种使用方式。

（2）静态变量的类型说明符：static

静态变量的类型说明符是 static。静态变量当然是属于静态存储方式，它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 加以定义后才能成为静态外部变量，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。

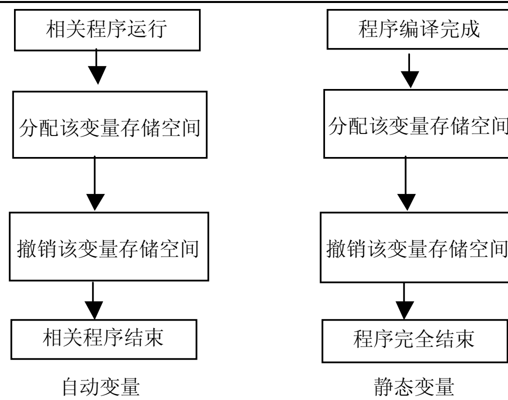


图 2.5 静态变量和动态变量

静态变量可分为静态局部变量和静态全局变量。

① 静态局部变量属于静态存储方式，它具有以下特点。

➢ 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 是静态局部变量的生存期及作用域示意图。

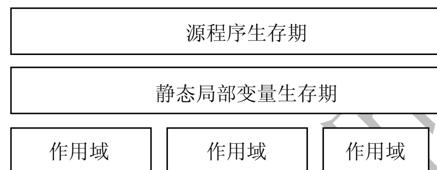


图 2.6 静态局部变量的生存期及作用域

➢ 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。

➢ 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 值。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间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，因此仍以采用局部静态变量为宜。

② 静态全局变量

全局变量（外部变量）在关键字之前再冠以 static 就构成了静态的全局变量。全局变量本身就是静态存储方式，静态全局变量当然也是静态存储方式。这两者在存储方式上并无不同。

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它。

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。

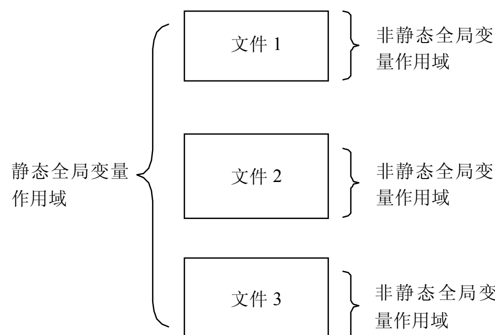


图 2.7　全局变量及非静态全局变量的区别

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。

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 的使用非常广泛。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*/
```

2.4 运算符与表达式

和其他程序设计语言一样，C 语言中表示运算的符号称为运算符。运算符是告诉编译程序执行特定算术或逻辑操作的符号，运算的对象称为操作数。

对一个操作数进行运算的运算符称为单目运算符，对两个操作数进行运算的运算符称为双目运算符，3 目运算符对 3 个操作数进行运算。用运算符和括号可以将操作数连接起来组成表达式。

C 语言提供了四十多个运算符，其中一部分跟其他高级语言相同（例如“+”、“−”、“*”等运算符），另外的与汇编语言类似，对计算机的底层硬件（如指定的物理地址）能进行访问。

C 语言的运算符功能强大，除了控制语句和输入输出以外的几乎所有的基本操作都可以用运算符来处理，例如，将“=”作为赋值运算符，方括号“[]”作为下标运算符等。

C 语言的运算符如表 2.7 所示。

表 2.7　　C 语言运算符类型

运算符类型	说　明
算术运算符	+ − * / %
关系运算符	> < == >= <= !=
逻辑运算符	! && ||
位运算符	<< >> ^ | & ~
赋值运算符	= 及其扩展赋值运算符
条件运算符	? :
逗号运算符	,
指针运算符	* &
求字节数运算符	Sizeof
强制类型转换运算符	(类型)
分量运算符	. →
下标运算符	[]
其他	如函数调用运算符（）

下面主要介绍基本运算符的使用。

华清远见

2.4.1　算术运算符和表达式

1．算术运算符

算术运算符包括双目的加减乘除四则运算符和求模运算符，以及单目的正负运算符，其列表如表 2.8 所示。

表 2.8　　算术运算符列表

运　算　符	描　　述	结　合　性
+	单目正	从右至左
−	单目负	从右至左
*	乘	从左至右
/	除和整除	从左至右
%	求模（求余）	从左至右
+	双目加	从左至右
−	双目减	从左至右

这里有几点需要说明。

➢ “+”、“–”、“*”、“/” 4 种运算符的操作数，可以是任意基本数据类型，其中 “+”、“–”、“*” 与一般算术运算规则相同。

➢ 除法运算符 “/” 包括了除和整除两种运算，当除数和被除数都是整型数时，结果只保留整数部分而自动舍弃小数部分；除数和被除数只要有一个浮点数，进行浮点数相除。

➢ 运算符 “–” 除了用作减法运算符之外，还有另一种用法，即用作负号运算符。用作负号运算符时只要一个操作数，其运算结果是取操作数的负值，如–（3+5）的结果是–8。

➢ 求模运算就是求余数，求模运算要求两个操作数只能是整数，如 5.8%2 或 5%2.0 都是不正确的。

除了上述常见的几种运算符之外，C 语言还提供了两个比较特殊的算术运算符：自增运算符 “++” 和自减运算符 “––”（关于这两个运算符在稍后的赋值运算符和表达式中会详细讲解）。

2．算术表达式

用算术运算符和括号可以将操作数连接起来组成算术表达式。

```
例如: a+2*b-5、18/3*(2.5+8)-'a'
```

在一个算术表达式中，允许不同的算术运算符以及不同类型的数据同时出现，在这样的混合运算中，要注意下面两个问题。

（1）运算符的优先级

C 语言对每一种运算符都规定了优先级，混合运算中应按次序从高优先级的运算执行到低优先级的运算。算术运算符的优先级从高到低排列如下（自左向右）：

```
( ) ++ -(负号运算符)-- * / % +- (加减法运算符)
```

（2）类型转换

不同类型的数值数据在进行混合运算时，要先转换成同一类型之后再运算，C 语言提供了两种方式的类型转换。 float

➢ 自动类型转换

这种转换是系统自动进行的，其转换规则如图 2.8 所示。

其中，float 型向 double 型的转换和 char 型向 int 型的转换是必定要进行的，即不管运算对象是否为不同的类型，这种转换都要进行。图中纵向箭头表示当运算对象为不同类型时的转换方向。如 int 型与 double 型数据进行运算时，是先将 int 型转换为 double 型，再对 double 型数据进行运算，最后的运算结果也为 double 型，例如：

```
100-"a"+40.5
```

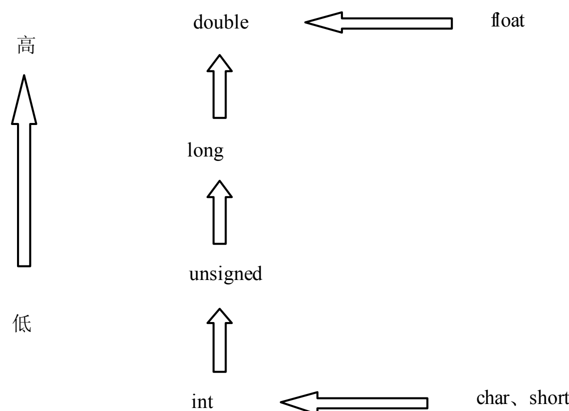


图 2.8 自动类型转换规则

这个表达式的运算过程是这样的。

第一步，计算“100−‘a’”，先将字符数据‘a’转换为 int 型数据 97（a 的 ASCII 码），运算结果为 3；
第二步，计算“3+40.5”，先将 float 型的 40.5 转换为 double 型，再将 int 型的 3 转换为 double 型，最后的运算结果为 double 型。

➢ 强制类型转换

利用强制类型转换运算符可以将一个表达式转换成所需的类型。强制类型转换的一般形式是：

```
(类型名)表达式
```

例如：(double)a 将 a 转换成 double 型，(int)(x+y)将 x+y 的值转换成 int 型(注意，不能写成(int)x+y 。强制类型转换一般用于自动类型转换不能达到目的的时候。例如，sum 和 n 是两个 int 型变量，则 sum/n 的结果是一个舍去了小数部分的整型数，这个整数很可能存在较大的误差，如果想得到较为精确的结果，则可将 sum/n 改写为 sum/（float）n 或（float）sum/n。

2.4.2 赋值运算符和表达式

1．赋值运算符

（1）单纯赋值运算符“=”

在前面的讲解中，读者已多次看到了符号“=”。在 C 语言中，“=”不是等号，而是赋值运算符，它是个双目运算符，结合性是从右向左，其作用是将赋值号“=”右边的操作数赋给左边的操作数。

示例：

```
x = y;    /*将变量 y 的值赋给变量 x(注意不是 x 等于 y)*/
a = 28;   /* 将 28 赋值给变量 a*/
j = j+2   /*把变量 j 的值加上 2，并把和赋值到 j 中*/
```

（2）复合赋值运算符“+=”、“-=”、“*=”、“/=”

在赋值符“=”之前加上其他运算符，即构成复合的运算符。C 语言规定有 10 种复合赋值运算符。除上面 4 种外，还有“%=”、“<<=”、“>>=”、“&=”、“^=”，“|=”，这些将在后面位运算中介绍。

示例：

```
a += 30 等效于 a = a+30，相当于 a 先加 30，然后再赋给 a.。
t *= x+5 等效于 t=t*(x+5)
```

采用复合赋值运算符既能简化程序，也能提高编译效率。所以编写程序的时候，应尽可能地使用复合赋值运算符。在 Linux 内核中，也随处可见复合赋值运算符的使用，如下例中就是在/drivers/char/rtc.c 的 rtc_interrupt 函数中代码：

```
rtc_irq_data += 0x100;
```

上面语句中的变量 rtc_irq_data 在接收到 RTC 的中断后会更新数值。

2．赋值表达式

用赋值运算符将一个变量和一个表达式连接起来，就成了赋值表达式。一般形式如下：

```
<变量名><赋值运算符><表达式> 即: 变量 = 表达式
```

对赋值表达式求解的过程是：将赋值运算符右侧的“表达式”的值赋给左侧的变量。赋值表达式的值就是被赋值的变量的值，如“a = 5”这个赋值表达式的值是 5。

注意

➢ 赋值运算符的左边只能是一个变量名，而不能是一个常量或其他的表达式。例如：13=b、a+b=15、j*2=100 这些都是错误的赋值表达式。

➢ 赋值运算符右边的表达式也可以为一个赋值表达式，例如：a=(b=2)或 a=b=2 表示变量 a 和 b 的值均为 2，表达式的值为变量 a 的值 2。此方法适合于给几个变量同时赋一个值时使用。

3．特殊的赋值运算——自增自减

“++”是自增运算符，它的作用是使变量的值增加 1。“--”是自减运算符，其作用是使变量的值减少 1，例如：

```
i = i+1
```

这个赋值表达式是把变量 i 的值加上 1 后再赋给 i，即将变量 i 的值增加 1。那么在这里就可以利用自增运算符简化这个赋值表达式为：

```
i++ 或 ++i
```

又如：

```
i-- 或--i 等价 i = i-1
```

自增运算符和自减运算符是两个非常有用的运算符，由于通常一条 C 语言的语言在经过编译器的处理后会翻译为若干条汇编语句，如赋值语句等会涉及多次寄存器的赋值等操作，而自增或自减语句能直接被翻译为“inc”和“dec”，因此它的执行效率比“i = i+1”或“i = i–1”更高，而且前者的写法使程序更精练。这里有两点需要注意：

- 自增/自减运算符，仅用于变量，不能用于常量或表达式；
- ++和--的结合方向是自右至左。

自增和自减运算符可用在操作数之前，也可放在其后，但在表达式中这两种用法是有区别的。自增或自减运算符在操作数之前，C 语言在引用操作数之前就先执行加 1 或减 1 操作；运算符在操作数之后，C 语言就先引用操作数的值，而后再进行加 1 或减 1 操作，例如：

```
j=i++;
```

其执行过程是：先将变量 i 的值赋值给变量 j，再使变量 i 的值增 1。结果是，i 的值为 3，j 的值为 2。等价于下面两个语句：

```
j=i;
i=i+1;
```

再如以下示例：

```
j=++i;
```

其执行过程是：先将变量 i 的值增 1，再把新 i 的值赋给变量 j。结果是：i=3，j=3。
该语句等价于下面两个语句：

```
i = i+1;
j=i;
```

2.4.3 逗号运算符和表达式

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。
例如：

```
y =(x=a+b),(b+c);
```

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 3 点。

➢ 逗号表达式一般形式中的表达式 1 和表达式 2 也可以是逗号表达式。例如：表达式 1,(表达式 2,表达式 3)。这样就形成了嵌套情形。

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。

➢ 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。

2.4.4 位运算符和表达式

1．位运算符

位运算符是指进行二进制位的运算。C 语言中提供的位运算包括与（&）、或（|）、异或（^）、取反（～）、移位（“<<”或“>>”）这些逻辑操作。对汇编语言比较熟悉的读者对这些已经非常了解了，不过在此还是做一简单介绍。

（1）与（&）、或（|）和异或（^）

这 3 种位运算都是双目操作符。当两个位进行相与时，只有两者都为“1”时结果才为“1”；当两个位进行相或时，两者中只要有一方为“1”，结果就为“1”；而当两个位进行异或时，只要两者不同，结果就为“0”，否则结果为“1”。运算规则如图 2.9 所示。

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

华清远见

图 2.9 位运算操作示意图

这些位操作符在使用时按位来进行操作，比如有 3 和 5 进行与（&）、或（|）、异或（^）操作，用户应该将它们先写成二进制的形式，再进行运算，如下所示：

```
     0011            0011             0011
  &  0101          | 0101           ^ 0101
     0001            0111             0110
```

（2）移位操作符（“<<”或“>>”）

移位操作符只是简单地把一个值往左或往右移。在左移中，原值最左边的几位被丢弃，其右边多出的几位补 0，如图 2.10 所示。

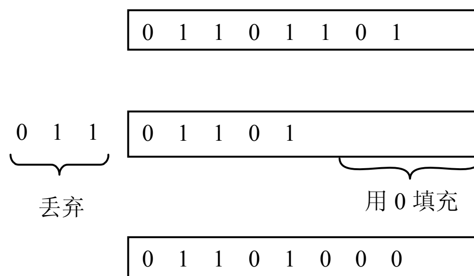


图 2.10 移位操作符操作过程

右移位虽然只是左移的相反操作，但却存在一个左移中不曾面临的问题——符号位的问题，也就是从左边移入新位时，可以选择两种方案：一是逻辑移位，左边移入的位简单地用 0 来填充；另一种是算术移位，左边移入的位由原先的符号位来决定，符号位为 1 则移入的位均为 1，符号位为 0 则移入的位均为 0，这样能够保证原数的正负形式不变。
比如，有数“1000101”，若将其右移两位，逻辑移位的结果是“0010001”，而算术移位的结果是“1110001”。由于在左移时不涉及逻辑位的取舍，因此，算术左移与逻辑左移的结果是一样的。

可移植性提示

C 语言标准说明无符号数执行的所有移位操作都是逻辑移位，但对于有符号数，到底是采用逻辑移位还是算术移位取决于编译器，不同的编译器所产生的结果有可能会不同。因此，一个程序若采用了有符号数的右移位操作，它是不可移植的。

关于位运算符有两点需要注意。

- 在这些移位运算符中，除了取反（~）是单目运算符，其余都是二元运算符，也就要求运算符两侧都有一个运算对象，位运算符的结合方向均为自左向右。
- 位运算符的对象只能为整型或字符型数据，不能是实型数据。

2．位表达式

将位运算符连接起来所构成的表达式称为位表达式。在位表达式中，依然要注意优先级的问题。在这些位运算符中，取反运算符（～）优先级最高，其次是移位运算符（<<和>>），再次是与（&）、或（|）和异或（^）。
在实际使用中，通常是将其进行赋值运算，因此，之前所提到的复合赋值操作符（“<<=”、“>>=”、“&=”、“^=”，“|=”）就相当常见了，比如：

```
a <<= 2;
```

就等价于：

```
a = a << 2;
```

读者应该注意到，在移位操作中，左移 n 位相当于将原数乘以 2^n，而右移 n 位则相当于将原数除以 2^n，因此，若读者希望操作有关乘除 2^n 的操作时，可以使用移位操作来代替乘除操作。由于移位操作在汇编语言中直接有与此相对应的指令，如“SHL”、“SAL”等，因此其执行效率是相当高的，表 2.9 列举了常见操作的执行时间（单位：机器周期）。

表 2.9　　基本运算执行时间

操　作	执 行 时 间
整数加法	1
整数乘法	4
整数除法	36
浮点加法	3
浮点乘法	5
浮点除法	38
移位	1

可以看到，乘除法（尤其是除法）操作都是相当慢地，因此若有以下两句语句：

```
a = (high + low)/2;
a = (high + low) >> 1;
```

这时，第二句语句会比第一句语句快很多。也正是由于位运算符的高效，在 Linux 内核代码中随处都可见到移位运算符的身影。如前面在赋值运算符中提到的有关 RTC 的例子中就有如下语句：

```
rtc_irq_data &= ~0xff;
rtc_irq_data |= (unsigned long)irq & 0xF0;
```

这两句语句看似比较复杂，但却是非常常见的程序写作方法，读者可以首先将复合赋值运算符展开，这样，第一句语句就成为以下形式：

```
rtc_irq_data = rtc_irq_data & ~ 0xff;
```

这时，由于取反运算符的优先级较高，因此，就先进行对 0xff 的取反操作，这就相当于为～0xff 加上了括号，如下所示：

```
rtc_irq_data = rtc_irq_data & (~ 0xff);
```

再接下来的步骤就比较明朗了，rtc_irq_data 先于 0xff 取反的结果 0xffffff00 相与，再将运算结果的值赋给 rtc_irq_data 变量本身。读者可以按照这种方法来分析第二条语句。

2.4.5 关系运算符和表达式

1．关系运算符

在程序中经常需要比较两个 值的大小关系，来决定程序下一步的工作。比较两个值的运算符称为关系运算符，在 C 语言中有以下关系运算符：

- < 小于。
- <= 小于或等于。
- > 大于。
- >= 大于或等于。
- == 等于。
- != 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。

在这 6 个关系运算符中，“<”、“<=”、“>”、“>=”的优先级相同，高于“==”和“!=”，“==”和“!=”的优先级相同。根据优先级的关系，以下式子具有等价的关系：

```
c>a+b      和      c>(a+b)
a>b==c     和      (a>b)==c
a=b>c      和      a=(b>c)
```

注意 “==”为关系运算符，判断两个数值是否相等，“=”为赋值运算符。

2．关系表达式

用关系表达式将两个式子（可以是各种类型的式子）连接起来的式子，称为关系表达式，关系表达式的一般形式为：

```
表达式 关系运算符 表达式
```

以下表达式都是合法的关系表达式。

华清远见

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

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)
```

关系表达式的值只有两种，即为“真”和“假”，用“1”和“0”表示。例如，关系表达式“5==3”的值为“假”（0），“5>3”的值为“真”（1）。
由于在 C 语言中，并不存在 bool（布尔）类型的值，因此，C 语言程序员以形成惯例，用“1”代表真，用“0”代表假。
另外，用户还可以通过 typede 来自定义 bool 类型，如下所示：

```
typedef unsigned char bool;
#define TRUE 1
#define FALSE 0
```

这样，在之后的使用时就可以用 bool 来定义变量，用 TRUE 和 FALSE 来判断表达式的值了。

2.4.6 逻辑运算符和表达式

1．逻辑运算符

C 语言中提供了 3 种逻辑运算符：与运算符（&&）、或运算符（||）和非运算符（!），其中与运算符（&&）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。
读者可以看到，逻辑运算符和位运算符（尤其是与或运算符）有很大的相似性。为了使读者更好地理解逻辑运算与位运算的区别，这里对逻辑运算的概念再做解释。
逻辑运算是用来判断一件事情是“对”的还是“错”的，或者说是“成立”还是“不成立”，判断的结果是二值的，即没有“可能是”或者“可能不是”，这个“可能”的用法是一个模糊概念。
在计算机里面进行的是二进制运算，逻辑判断的结果只有两个值，称这两个值为“逻辑值”，用数的符号表示就是“1”和“0”。其中“1”表示该逻辑运算的结果是“成立”的，如果一个逻辑运算式的结果为“0”，那么这个逻辑运算式表达的内容“不成立”。
【例】
通常一个教室有两个门，这两个门是并排的。要进教室从门 A 进可以，从门 B 进教室也行，用一句话来说是“要进教室去，可以从 A 门进‘或者’从 B 门进”。
这里，可以用逻辑符号来表示这一个过程：能否进教室用符号 C 表示，教室门分别为 A 和 B。C 的值为“1”表示可以进教室，为“0”表示进不了教室。A 和 B 的值为“1”时表示门是开的，为“0”表示门是关着的，那么它们之间的关系就可以用表 2.10 来表示。

表 2.10 示例逻辑关系

说 明	C	A	B
两个教室的门都关着，进不去教室	0	0	0
门 B 是开着的，可以进去	1	0	1
门 A 是开着的，可以进去	1	1	0
门 A 和 B 都是开着的，可以进去	1	1	1

把表中的过程写成逻辑运算就是：

华清远见

```
C = A || B
```

这就是一个逻辑表达式，它是一个“或”运算的逻辑表达式。这个表达式要表达的就是：如果要使得 C 为 1，只要 A“或”B 其中之一为“1”即可。所以“||”运算称为“或”运算。

【例】

假设一个房间外面有一个晒台，那么这个房间就纵向开着两个门，要到晒台去，必须要过这两个门，很明显这两个门必须都是开着的才行，否则只要其中一个门关着就去不了晒台。

这时，同样使用逻辑符号 C 来表示是否能去晒台，A 和 B 表示是否 A、B 门是否以开，那么它们之间的关系就可以用表 2.11 来表示。

表 2.11 示例逻辑关系

说　　明	C	A	B
两个门都关着，去不了晒台	0	0	0
门 A 关着，去不了晒台	0	0	1
门 B 关着，去不了晒台	0	1	0
门 A 与门 B 都开着，可以去晒台	1	1	1

把表中的过程写成逻辑运算式就是：

```
C = A && B
```

从上面的两例可以看出，在逻辑表达式里有参加逻辑运算的逻辑量和逻辑运算最后的结果（逻辑值），把这两个概念区分开来和记住它们是很重要的。

什么是逻辑量呢？凡是参加逻辑运算的变量、常量都是逻辑量，例如上例中的 A、B。而逻辑值则是逻辑量、逻辑表达式其最后的运算结果的值。下面两条规则在逻辑表达式中是非常重要的。

- 逻辑值只有“0”和“1”两个数，其中“1”表示逻辑真（成立），“0”表示逻辑假（不成立）。
- 一切非“0”的逻辑值都为真。例如：−1 的逻辑值为真（1），5 的逻辑值为真（1）。

表 2.12 列出了逻辑运算的真值表。

表 2.12 逻辑运算真值表

a	b	!a	!b	a&&b	a||b
真	真	假	假	真	真
真	假	假	真	假	真
假	真	真	假	假	真
假	假	真	真	假	假

2．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
```

其中的表达式也可以是逻辑表达式，从而组成了嵌套的情形。

在这里，首先要明确的还是优先级的问题，逻辑运算符和其他运算符优先级的关系如图 2.11 所示。

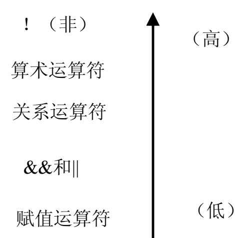


图 2.11　移位操作符操作过程

由以上优先级的顺序可以看出：

```
a>b && c>d 等价于(a>b) && (c>d)
!b= =c||d<a 等价于((!b)= =c)||(d<a)
a+b>c && x+y<b 等价于((a+b)>c) && ((x+y)<b)
```

逻辑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”。

2.4.7 sizeof 操作符

sizeof 是一个单目运算符，它的运算对象是变量或数据类型，其运算结果为一个整数。若运算对象为变量，则所求的结果是这个变量占用的内存空间字节数，若运算对象是数据类型，则所求结果是这种数据类型的变量占用的内存空间字节数。

它是一个很常用的工具，能够准确地测量这些变量或数据类型所占用的内存空间的大小，下面的程序显示了 sizeof 的用法：

```
#include <stdio.h>

enum week{
    SUN,MON,TUES,WED,THURS,FRI,SAT,
};
int main()
{
          int i;
          enum week myweek;
          printf("the size of int is %d bytes\n",sizeof(int));
          printf("the size of long is %d bytes \n",sizeof(long));
          printf("the size of short is %d bytes \n",sizeof(short));
          printf("the size of char is %d bytes \n",sizeof(char));
          printf("the size of week is %d bytes \n", sizeof (enum week));
          printf("the size of myweek is %d bytes \n", sizeof (myweek));
}
```

该程序的运行结果为：

```
the size of int is 4 bytes
the size of long is 4 bytes
the size of short is 2 bytes
the size of char is 1 bytes
the size of week is 4 bytes
the size of myweek is 4 bytes
```

从该结果中，可以清楚地看到不同数据类型及变量所占的字节数。

2.4.8 条件（？）运算符

条件运算符（？）是 C 语言中惟一一个三目运算符，它提供如 if-then-else 语句的简易操作，其一般形式为：

```
EXP1 ? EXE2: EXP3
```

这里 EXP1、EXP2 和 EXP3 都可以是表达式。

操作符“?”作用是这样的：先计算 EXP1 的逻辑值，如果其值为真，则计算 EXP2，并将数值结果作为整个表达式的数值；如果 EXP1 的逻辑值为假，则计算 EXP3，并以它的结果作为整个表达式的值，其执行过程如图 2.12 所示。

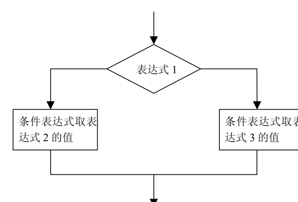


图 2.12　条件操作符的执行过程

条件运算符的优先级高于赋值运算符，读者可以自行分析一下以下语句的含义：

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

由于条件运算符的优先级高于赋值运算符，因此，先计算赋值语句的右边部分。

当 a 大于 b 为真（即 a 大于 b）时，条件表达式的值为 a；当 a 大于 b 为假（即 a 大于 b 不成立）时，条件表达式的值为 b。因此，max 变量的值就是 a 和 b 中较大的值（若 a 与 b 相等时取 b）。

2.4.9　运算符优先级总结

C 语言中的优先级一共分为 15 级，1 级最高，15 级最低。在有多个不同级别的运算符出现的表达式中，优先级较高的运算符将会先进行运算，优先级较低的运算符后运算。另外，如果在一个运算对象两侧的运算符的优先级相同时，则按运算符的结合性所规定的结合方向来进行处理。

C 语言的结合性有两种，即左结合性和右结合性。若为左结合性，则该操作数先与其左边的运算符相结合；若为右结合行，则该操作数先与其右边的运算符相结合。

因此，对于表达式“x−y+z”，读者可以看到 y 的左右两边的操作符“−”和“+”都为同一级别的优先级的，而它们也都具有左结合性，因此，y 就先与“−”相结合，故在该表达式中先计算“x−y”。

表 2.13 列举了 C 语言中的运算符的优先级和结合性。

表 2.13　运算符的优先级和结合性

优　先　级	运　算　符	含　　义	运算对象个数	结合方向
1	() [] -> .	圆括号 下标运算符 指向结构体成员运算符 结构体成员运算符		自左向右
2	! ~ ++ −− − （类型） * & sizeof	逻辑非运算 按位取反运算 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 地址与运算符 长度运算符	1（单目）	自右向左
3	* / %	乘法运算符 除法运算符 求余运算符	2（双目）	自左向右
4	+ −	加法运算符 减法运算符	2（双目）	自左向右
5	<< >>	左移运算符 右移运算符	2（双目）	自左向右
6	< <= > >=	关系运算符	2（双目）	自左向右
7	= = !=	等于运算符 不等于运算符	2（双目）	自左向右

华清远见

8	&	按位与运算符	2（双目）	自左向右
9	^	按位异或运算符	2（双目）	自左向右
10	|	按位或运算符	2（双目）	自左向右
11	&&	逻辑与运算符	2（双目）	自左向右
12	||	逻辑或运算符	2（双目）	自左向右
13	?:	条件运算符	3（三目）	自右向左
14	= += −= *= /= %= >>= <<= &= ^= |=	赋值运算符	2（双目）	自右向左
15	，	逗号运算符		自左向右

这些运算符的优先级看起来比较凌乱，表 2.14 所示为一个简单易记的口诀，可以帮助读者记忆。

表 2.14　　　　　　　　　　运算符的优先级口诀

口　　诀	含　　义
括号成员第一	括号运算符[]() 成员运算符. ->
全体单目第二	所有的单目运算符，比如++ −− +(正) −(负)等
乘除余三，加减四	这个“余”是指取余运算即%
移位五，关系六	移位运算符：<< >> ，关系：> < >= <= 等
等于(与)不等排第七	即= = !=
位与异或和位或“三分天下”八九十	这几个都是位运算：　位与(&)异或(^)位或(|)
逻辑或跟与 十二和十一	逻辑运算符：||和&& 注意顺序：优先级(||)低于优先级(&&)
条件高于赋值	三目运算符优先级排到 14 位只比赋值运算符和“,”高，需要注意的是赋值运算符很多
逗号运算级最低	逗号运算符优先级最低

对于结合性的记忆比较简单，读者可以注意到，大多数运算符的结合性都是自左向右的，惟独单目运算符、条件运算符和赋值运算符是自右向左。

华清远见

2.5　程序结构和控制语句

2.5.1　C 语言程序结构

从程序流程的角度来看，C 语言中的语句可以分为 3 种基本结构：顺序结构、分支结构和循环结构。

➢ 顺序结构的执行过程如图 2.13 所示，在这种结构中，程序会顺序执行各条语句。

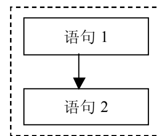


图 2.13 顺序结构

➢ 分支结构的执行过程如图 2.14 所示，在这种结构中，程序会根据某一条件的判断来决定程序的走向，比如当该条件成立时执行语句 1，当该条件不成立时执行语句 2。另外，也有可能会有多种条件的情况，比如，当条件 1 成立时执行语句 1，当条件 2 成立执行语句 2，在其他情况下执行语句 3、4 等。

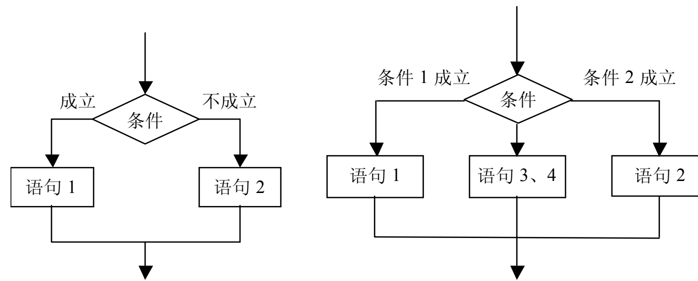


图 2.14 分支结构

➢ 循环结构的执行过程如图 2.15 所示，这种结构有两种形式：当型循环和直到型循环。当型循环首先判断条件是否成立，若条件成立则执行循环内的语句，若条件不成立则直接跳出循环；直到型循环是直接执行循环内的语句，直到条件不成立时退出循环体。

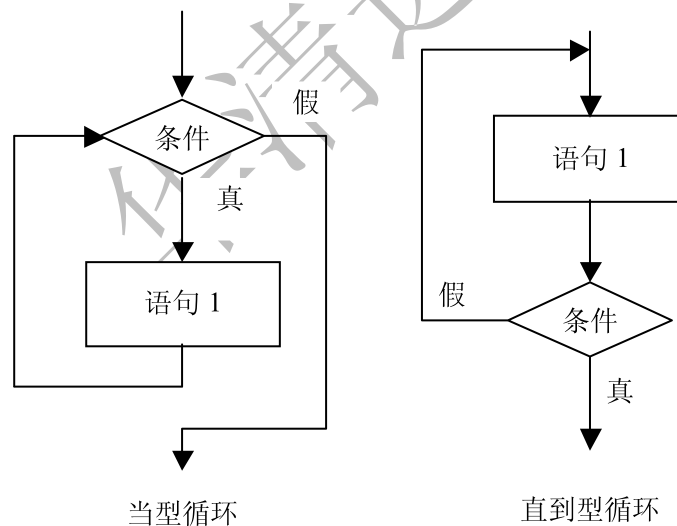


图 2.15 循环结构

2.5.2 C 语言控制语句

C 语言中的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，包括：条件判断语句、循环语句和转向语句。

➢ 条件判断语句

又称为选择语句，包括 if 语句和 switch 语句。

1. if 语句的形式

if 语句是用来判定所给定的条件是否满足，根据判定的结果（真或假）决定执行给出的操作，if 语句有 3 种形式。

- **if**（表达式） 语句 2
- **if**（表达式） 语句 1 **else** 语句 2
- **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

图 2.16 的（a）、（b）、（c）分别表示了这 3 种形式的 3 种执行情况。

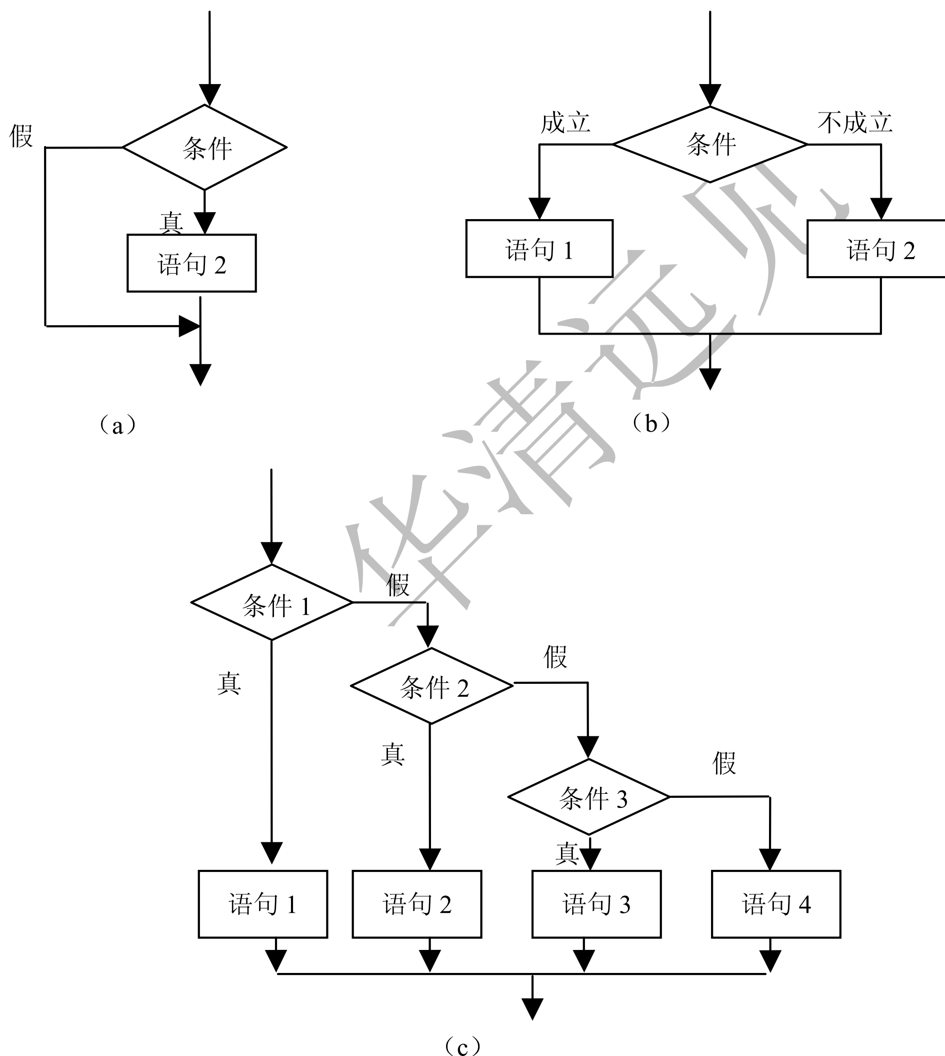


图 2.16 if 语句的 3 种形式

在 if 语句中，首先计算表达式中的逻辑值是真或是假。若是真，则进入相应的语句中执行，若是假，则跳出该语句直接执行下面的语句。

对于第一种单分支的情况，若判断表达式为真，则执行语句 2；若判断表达式为假，则跳出语句。

对于第二种双分支的情况，若判断表达式为真，则执行语句 1，否则就执行语句 2，可以看出，在这种情况中语句 1 和语句 2 有且仅有一条语句会被执行。

对于第三种多分支的情况，首先判断条件 1 是否为真，若为真则执行语句 1 并跳出，若为假则继续判断条件 2 是否执行，若条件 2 为真则执行语句 2 并跳出，否则继续判断条件 3，依此类推。
要注意的是，if 语句和 else 语句后中只能执行一条语句。

```
if(x > y)
        printf("x is bigger\n");
else
        printf("x is not bigger\n");
```

注意 语句必须以“;”结尾。

以下语句是不正确的：

```
if(x > y)
        y = x;
        printf("x is bigger\n");
else
        printf("x is not bigger\n");
```

可以看到，在此时，if 语句后有两条语句，这是不正确的。那么，如何表达在“x>y”的情况下将“x”的值赋给“y”并且打印出“x is bigger\n”呢？在 C 语言中复合语句被看作是单条语句，而不是多条语句。因此，只需在这两条语句外加上括号就可以了，如下所示：

```
if(x > y)
{
        y = x;
        printf("x is bigger\n");
}
else
        printf("x is not bigger\n");
```

2. if 语句的嵌套使用

在 if 语句中又包含一个或多个 if 语句称为 if 语句的嵌套，其形式一般如下：

```
if ( )
     if ( ) 语句 1    }
     else 语句 2      }  嵌套
else
     if ( )  语句 3   }
     else 语句 4      }  嵌套
```

注意这时，在外层的 if、else 后面不需要有“{}”。在这里，需要着重注意的是 if 和 else 的配对问题，请读者务必记住一个配对原则：在嵌套 if 语句中，else 总是与它上面最近的 if 配对。
例如：

```
if ( )
      if ( ) 语句 1
else
      if ( ) 语句 2
      else 语句 3
```

根据配对原则，第一个 else 会与第二个 if 配对，而不是与第一个 if 配对。如果希望 else 与第一个 if 配对的话，就要用大括号“{}”把 if 后的语句定义成一个复合语句。如下所示：

```
if ()
{
    if () 语句 1
}
else
    if () 语句 2
    else 语句 3
```

注意 复合语句的括号后不需要加“;”

3. switch 语句

if 语句只能从两者间选择之一，当要实现几种可能之一时，就要用 if...else if 甚至多重的嵌套 if 来实现，当分支较多时，程序变得复杂冗长，可读性降低。switch 开关语句专门处理多路分支的情形，使程序变得简洁。

switch 语句的一般格式为：

```
switch（表达式）
case 常量表达式 1：语句序列 1；
case 常量表达式 2：语句序列 2；
……
case 常量表达式 n：语句 n;
default：语句 n+1；
```

这里，switch 后表达式中的结果必须是整型值或字符型值。这里的常量表达式是指在编译期间进行求值的表达式，它不能是任何变量。“case”表达式后的各语句序列允许有多条语句，不需要按复合语句处理。

switch 语句比较特殊，这里的 case 标签并没有把语句列表划分为几个部分，它只是确定语句执行的入口点。switch 语句的执行过程是这样的：首先计算表达式的值，然后跳转到 case 标签值与表达式值相等的地方开始往下执行，如果没有跳转指令的话会一直执行到 switch 语句的最后。执行过程如图 2.17 所示。

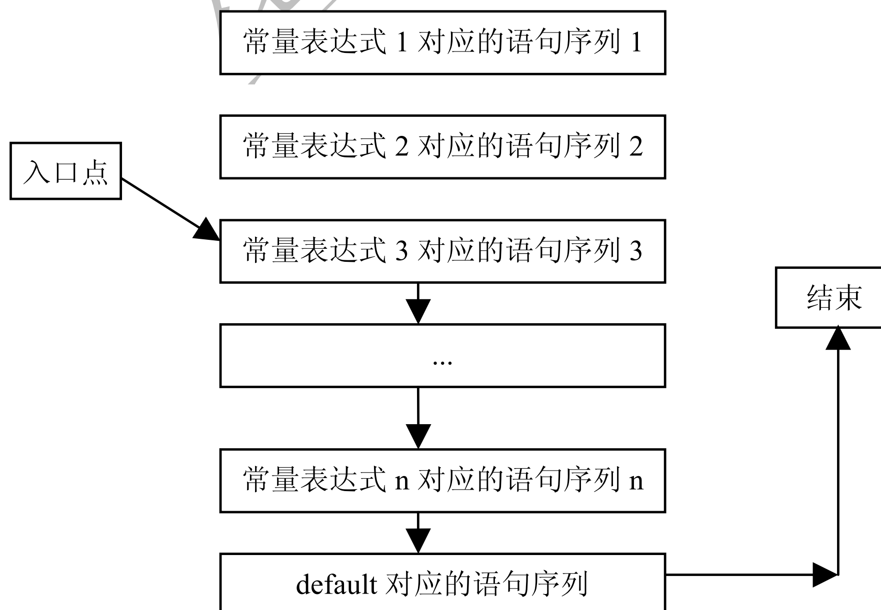


图 2.17 switch 语句的执行过程

如果希望执行完入口点的语句序列后直接跳出 switch 语句，就需要在语句序列后中加入 break 语句。这样，switch 的执行过程如图 2.18 所示。

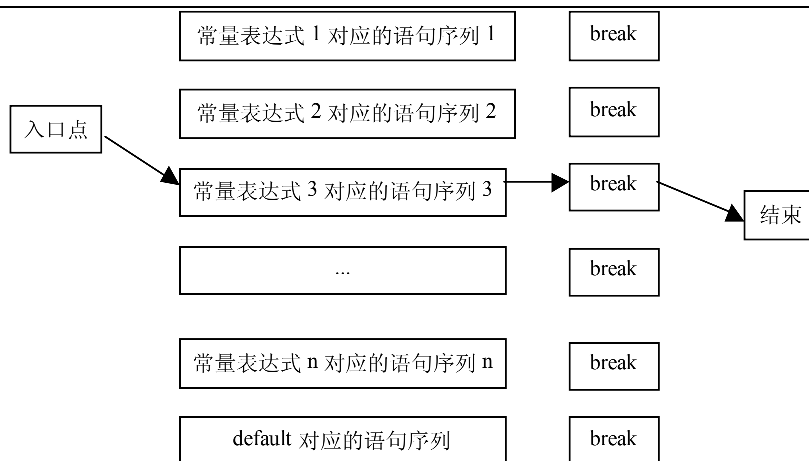


图 2.18 加 break 后的 switch 语句的执行过程

可以看出，在加入了 break 语句后，用户可以在执行完相应的语句序列后就跳出 switch 语句。

- 循环语句

1. while 和 do-while 语句

C 语言中有两种循环结构：当型和直到型，其中 while 语句是当型循环结构，它的格式如下：

```
while（表达式）
{
        循环体语句
}
```

在执行 while 循环语句时，先判断表达式的值，再执行循环体中的内容。

与此相对应的 do-while 是直到型循环结构，它的格式为：

```
do
{
        循环体语句
} while（表达式);
```

在执行 do-while 循环语句时，先执行循环体里的内容，再执行 while 表达式里的值。

注意 勿忘 while 括号后的 “;”。

通常，对于同一个问题既可以用 while 语句也可以用 do-while 语句来解决。例如，想要求 1～100 的和，以下分别使用 while 和 do-while 语句来实现。

```
#include <stdio.h>
void main()
{
int sum = 0;
i = 0;
while(i <= 100)
{
    sum += i;
    i++;
}
printf("the sum of 100 is %d\n",
sum);
}
```

```
#include <stdio.h>
void main()
{
int sum = 0;
i = 0;
do
{
    sum += i;
    i++;
} while(i <= 100);
printf("the sum of 100 is %d\n",
sum);
}
```

这两个程序的运行结果是一样的。但如果把变量 i 的初值改为 101，运行结果就不一样。while 语句的运行结果是 0，而 do-while 语句的运行结果是 101。

2. for 循环语句

for 语句是 C 语言所提供的功能更强、使用更广泛的一种循环语句，其一般形式为：

```
for（表达式 1; 表达式 2; 表达 3）
 语句
```

括号中的 3 个表达式的含义如下

- 表达式 1：通常用来给循环变量赋初值，一般是赋值表达式。也允许在 for 语句外给循环变量赋初值，此时可以省略该表达式。
- 表达式 2：通常是循环条件，一般为关系表达式或逻辑表达式。
- 表达式 3：通常用来修改循环变量的值，一般是赋值语句。

注意 这 3 个表达式都可以是逗号表达式，即每个表达式都可由多个表达式组成。3 个表达式都是任选项，都可以省略。

for 后面的语句即为循环体。可以是一条语句，也可以是用括号“{}”包含起来的多条语句。

for 语句的语义是：首先计算表达式 1 的值，再计算表达式 2 的值，若值为真（非 0）则执行循环体一次，否则跳出循环。然后再计算表达式 3 的值，转回第 2 步重复执行。在整个 for 循环过程中，表达式 1 只计算一次，表达式 2 和表达式 3 则可能计算多次。循环体可能多次执行，也可能一次都不执行，它等价于下面的 while 语句。

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

例如，有如下 for 语句

```
for(i = 0; i < 10 ; i++)
       a[i] = i;
```

与它等价的 while 语句为

```
i = 0;
while(i < 10)
{
     a[i] = i;
     i++;
}
```

可以看出，使用 for 循环语句更加简洁明了。

在使用 for 语句中要注意以下几点。

- for 语句中的各表达式都可省略，但分号间隔符不能少。如：for（; 表达式; 表达式）省去了表达式 1; for（表达式;; 表达式）省去了表达式 2; for（表达式; 表达式; ）省去了表达式 3; for（; ; ）省去了全部表达式。
- 在循环变量已赋初值时，可省去表达式 1。如省去表达式 2 或表达式 3 则将造成无限循环，这时应在循环体内设法结束循环。
- 循环体可以是空语句。
- for 语句也可与 while、do-while 语句相互嵌套，构成多重循环。

➢ 转向语句

转向语句包括 break、continue 和 goto 语句。

1. break 语句

break 语句在前面的 switch 语句中已经出现过，只能用在 switch 语句或循环语句中，其作用是跳出 switch 语句或跳出本层循环，转去执行后面的语句。由于 break 语句的转移方向是明确的，所以不需要语句标号与之配合，break 语句的一般形式为：

```
break;
```

要注意的是 break 语句只能跳出一层循环。

2. continue 语句

continue 语句只能用在循环体中，其一般格式是：

```
continue;
```

其功能是结束本次循环，即不再执行循环体中 continue 语句之后的语句，转入下一次循环条件的判断与执行。continue 语句只结束本次的循环，并不跳出循环。

举个例子：输出 100 以内的所有能被 7 整除的正整数

```
#include <stdio.h>
void main()
{
      int n;
      for(n=1;n<=100;n++)
      {
            if (n%7!=0)
                   /*若不能被 7 整除，则跳出本次循环，继续下一次循环*/
                   continue;
            printf("%d ",n);
      }
}
```

3. goto 语句

goto 语句也称为无条件转移语句，其一般格式如下：

```
goto 语句标号;
```

其中语句标号是按标识符规定书写的符号，放在某一语句行的前面，标号后加冒号（:）。语句标号起标识语句的作用，与 goto 语句配合使用。

例如：

```
       i = 0;
loop: if(i < 7)
       {
          i++;
          goto loop;
       }
```

C 语言不限制程序中使用标号的次数，但各标号不得重名。goto 语句的语义是改变程序流向，转去执行语句标号所标识的语句。通常与条件判断语句配合使用，可用来实现条件转移，构成循环，跳出循环体等功能。

由于 goto 语句可以随意跳转，很容易造成程序结构的混乱和程序出错，因此在结构化程序设计中一般不主张使用 goto 语句，以免带来程序理解和调试上的困难。

2.6 数组、结构体和指针

➢ 数组

1. 数组的定义

在 C 语言中为了处理数据方便，把具有相同类型的若干变量按有序的形式顺序组织起来。这些按序排列的同类数据元素的集合称为数组。

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、结构体数组等等。

C 语言中使用数组必须先进行定义。数组定义的一般形式为：

```
类型说明符 数组名[常量表达式];
```

类型说明符可以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。

对于数组的定义，这里有几点是需要特别注意的。

- 数组的类型实际上是指数组元素的取值类型。对于同一个数组，其所有元素的数据类型都是相同的。
- 数组名应当符合标识符的命名规定。
- 数组名不能与其他变量名相同，例如，以下的书写是错误的。

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

- 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，它需要在数组定义时就确定下来，不能随着程序的运行动态更改。它的下标从 0 开始计算，因此 5 个元素分别为 a[0]、a[1]、a[2]、a[3]、a[4]。
- 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，如 a[3+2]、b[5+9]。
- 允许在同一个类型说明中说明多个数组和多个变量。

在内存中，数组中的元素占用连续的存储空间，并且根据单个元素所占存储空间来进行内存分配。例如有 6 个元素的两个数组，分别为整型和字符型，它们在内存中的存放形式如图 2.19 所示。

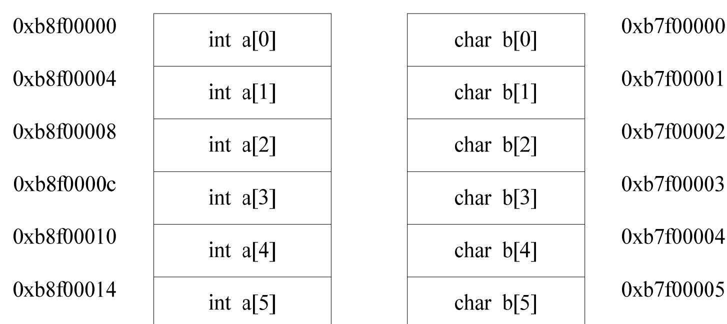


图 2.19 数组在内存中的存储形式

2. 数组的引用

C 语言中规定了数组必须逐个元素引用，而不能整体引用。因此，数组的引用实际上就是数组元素的引用。数组元素的一般表示方法为：

```
数组名[下标]
```

其中的下标只能为整型常量或整型表达式。这里的方括号“[]”读者在之前介绍的运算符中已经见到过。它实际上就是下标引用符，优先级是最高的，并且具有右结合性。例如有以下小程序：

```
#include <stdio.h>
void main()
{
        /*数组定义，有 10 个元素*/
        int i,a[10];

        for(i=0;i<10;)
                /*下标为整型表达式，注意标号范围为 0 到 9*/
                a[i++]=2*i+1;
        printf("display all these numbers...\n");
        for(i=9;i>=0;i--)
                /*下表为为整型表达式，标号是范围为 0 到 9*/
                printf("a[%d] is %d\n",i,a[i]);
}
```

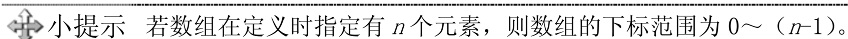
其运行结果如下所示：

```
display all these numbers...
a[9] is 19
a[8] is 17
a[7] is 15
a[6] is 13
a[5] is 11
a[4] is 9
a[3] is 7
a[2] is 5
a[1] is 3
a[0] is 1
```

虽然通过下标可以很方便的访问数组中的元素，但一不小心就会出现下标超出数组范围的情况。ANSI C 没有对使用越界下标的行为做出定义。因此，一个越界下标有可能导致以下几种结果之一：

- 程序仍能正确运行；
- 程序会异常终止或崩溃；
- 程序能继续运行，但无法得出正确的结果；
- 其他情况。

因此，在引用数组元素时，一定要仔细地处理下标，以防止出现数组越界问题。


小提示　若数组在定义时指定有 n 个元素，则数组的下标范围为 0～（n-1）。

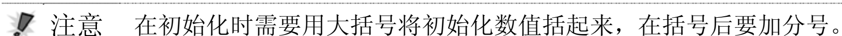
3. 数组的初始化

数组的初始化有 3 种方式。

（1）定义时整体初始化

与变量在定义时初始化一样，数组也可以在定义时进行初始化，如对字符数组进行初始化：

```
char a[10]={'a','b','c','d','e','f','g','h','j','k'};
```


注意　在初始化时需要用大括号将初始化数值括起来，在括号后要加分号。

（2）定义时部分初始化

数组在定义时可以对其中的部分数据进行初始化。当“{}”中值的个数少于元素个数时，只给前面部分元素赋值。例如如下定义就是对数组的前 5 个数据初始化，而后 5 个数据自动赋 0(在字符数组中自动赋‘\0’)。

```
char a[10]={'a','b','c','d','e'};
```

（3）数组全部赋值

若想要对数组中的元素全部赋值，则可以省略数组下标中的常量。编译器会根据初始化列表自动计算数组元素的个数，如下所示：

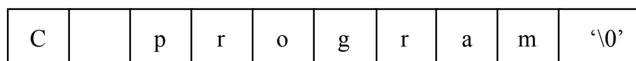
```
char a[]={'a','b','c','d','e','f','g','h','j','k'};
```

注意此时“[]”不能省略。

注意　数组的元素不能整体赋值，只能单个赋值，比如，若定义“char a[10] = { ‘a’ };”则只为该数组中的第一个元素赋值。

4. 字符串

在 C 语言中，没有单独的字符串数据类型，而是通过对字符数组的操作来实现的。字符串的内容保存在字符数组里，规定以‘\0’作为字符串结束标志。比如“C program”实际上占有 10 个字节，在内存中的存储方式如图 2.20 所示



C		p	r	o	g	r	a	m	‘\0’

图 2.20　字符串在内存中的存储方式

字符串的初始化可以按照字符数组的初始化方式进行，也可以按照字符串双引号的初始化方式进行，如下所示：

```
char a[6]={'C','h','i','n','a','\0'};
char a[6]={"China"};
```

注意　在使用字符数组的方式进行初始化时，要加上“‘\0’”结束符。

由于使用双引号的方式简单明了，因此，字符串的初始化一般都用双引号的方式来处理。

5. 字符串的处理

学习过 C 语言的读者都知道可以用“%s”来“输入输出一个字符串”。例如以下语句就可将“China”字符串输出。

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

使用“%s”格式符输出字符串时，遇到‘\0’就自动停止，并且在输出字符中不包含‘\0’。即使数组长度大于字符串的实际长度，也只输出到‘\0’就停止。

输出字符串时，printf 的参数是字符数组名，而不是字符元素名。写成如下所示是不对的。

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

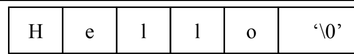
同样，用户也可以在 scanf 中使用“%s”直接输入字符串，以下语句可以接受用户输入的字符串。

```
char str[20];
scanf("%s",str);
```

这时，如果用户从键盘输入“Hello”并按回车，如下所示：

```
Hello↙
```

则系统就会自动在其后面加上一个‘\0’结束符，内存中的存储方式如图 2.21 所示。



H	e	l	l	o	'\0'

图 2.21　Hello 字符串的存储方式

实际上，数组名所代表的含义就是数组在内存中存放的首地址（关于这一点在指针的讲解中会有详细阐述），因此在 scanf 的参数中只需指定存放字符串的数组名即可，而不用再加上取地址符(&)。

在 C 语言的库函数中，提供了一些用来处理字符串的函数，使用起来非常方便，用户引入头文件 string.h 即可。表 2.15 列举了常见的字符串处理函数及使用示例。

表 2.15　常见字符串处理函数及使用示例

函　数　名	函数说明及定义	使 用 示 例
puts	puts(char *str) 将一个字符串（以 '\0' 结束的字符序列）输出到终端	char str = {"Hello world"}; puts(str);
gets	从终端输入一个字符串到字符数组，并得到一个返回值	gets(str); 等待用户输入
strcat	char *strcat (char *dest,const char *src); strcat()会将参数 src 字符串复制到参数 dest 所指的字符串尾。第一个参数 dest 要有足够的空间来容纳要复制的字符串	char a[30]="string(1)"; char b[]="string(2)"; printf("%s\n",strcat(a,b));
strcpy	char*strcpy(char *dest,const char *src); strcpy()会将参数 src 字符串复制至参数 dest 所指的字符串中	char a[30]="string(1)"; char b[]="string(2)"; printf("%s\n",strcpy(a,b));
strncpy	char * strncpy(char *dest,const char *src,size_t n) strncpy()会将参数 src 字符串复制前 n 个字符至参数 dest 所指的字符串	char a[30]="string(1)"; char b[]="string(2)"; printf("%s\n",strncpy(a,b,4));
strcmp	int strcmp(const char *s1,const char *s2); strcmp()用来比较参数 s1 和 s2 字符串。字符串大小的比较是以 ASCII 码表上的顺序来决定，此顺序亦为字符的值。strcmp()首先将 s1 第一个字符值减去 s2 第一个字符值，若差值为 0 则再继续比较下个字符，若差值不为 0 则将差值返回 若参数 s1 和 s2 所有字符相同则返回 0。s1 若大于 s2 则返回大于 0 的值。s1 若小于 s2 则返回小于 0 的值	char a[30]="aBcdEf"; char b[]="AbCdEF"; printf("%s\n",strcmy(a,b));
strlen	size_t strlen (const char *s); strlen()用来计算指定的字符串 s 的长度，不包括结束字符"\0"	char b[]="AbCdEF"; printf("%d\n",strlen(b));

6. 二维数组

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。在实际使用中有很多数组是二维的或多维的。多维数组中元素有多个下标，以标识它在数组中的位置。

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。二维数组类型定义的一般形式是：

```
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 3 行 4 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，其下标在两个方向上变化。元素实际的存储位置是连续的，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呢？
通常有两种方式：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。
图 2.22 中，按行顺次存放。先存放 a[0]行，再存放 a[1]行，最后存放 a[2]行。每行中的 4 个元素也是依次存放的。由于数组 a 说明为 int 类型，每一行包含 4 个元素，所以每行占有 16 个字节（图中每一格为 4 个字节）。

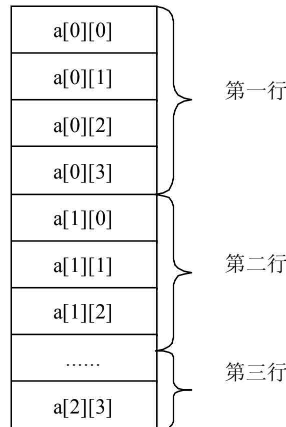


图 2.22　二维数组的存储方式

多维数组的引用和一维数组的引用很相似，也是通过下标引用的方式进行的。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 1][下标 2]
```

其中下标应为整型常量或整型表达式，例如：

```
a[3][4] 表示 a 数组第三行第四列的元素。
```

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；而数组元素中的下标是该元素在数组中的位置标识。前者只能是常量，后者可以是常量、变量或表达式。

二维数组初始化也是在定义时给各元素赋以初值。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]可以有以下两种赋值方式。
（1）分段赋值

```
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

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

可以看到，多维数组初始化时每个括号“{}”都代表一行，这里的每一行都可以如一维数组一样进行部分赋值，并且如果对全部元素赋初值，则第一维的长度可以不给出。当然，如果采用按行连续赋值的方式，只有最后一维可以部分赋值，其他必须全部赋值。在实际使用时，通常采用分段赋值的方式为二维数组初始化。

- 结构体

1. 结构体的定义

结构体和数组一样，也是一种构造型数据类型。与数组不同的是，在结构体中可以包含不同数据类型的成员。结构体的使用非常灵活，可以方便地构建复杂的数据类型。
定义结构体变量的一般形式为：

```
struct 结构体名
{
类型 成员名;
```

华清远见

```
类型 成员名;
...
}结构体变量名;
```

这里的结构体名是结构体的标识符，不是变量名。类型可以是基本的数据类型也可以是其他构造型数据类型。

结构体中每个成员都是变量。和数组不同的是，数组中的元素是通过下标来访问的，而结构体变量中的成员是通过成员名来访问的。

下面举一个例子来说明怎样定义结构体变量。

```
struct person
{
char name[8];
int age;
char sex;
char address[20];
float salary;
} p1;
```

以上定义了一个结构体名称为 person 的结构体变量 p1。如果省略变量名 p1，则变成对结构体的说明。也可以用下面的定义方式：

```
struct person
{
char name[8];
int age;
char sex;
char address[20];
float address;
};
struct person p1;
```

如果要定义多个具有相同类型的结构体变量，用第二种方法比较方便。它定义结构体类型，再用结构体名来定义变量。

如果省略结构体名，则称之为无名结构体，这种情况常常出现在函数内部，例如：

```
struct
{
char name[8];
int age;
char sex;
char address[20];
float salary;
} p1, p2;
```

2. 结构体变量的使用

结构体变量是不同数据类型的若干数据的集合体。在程序中使用结构体变量时，一般情况下不能把它作为一个整体参加数据处理，参加各种运算和操作的是结构体变量的各个成员项。

结构体变量的成员用以下一般形式表示：

结构体变量名.成员名

在定义了结构体变量后，就可以用不同的赋值方法对结构体变量的每个成员赋值。例如：

```
p1.age = 20;
p1.salary = 4000.50;
```

结构体变量在使用中应注意以下几点：

- 不能将一个结构体类型变量作为一个整体加以引用，而只能对结构体类型变量中的各个成员分别引用。
- 如果成员本身又属一个结构体类型，则要用若干个成员运算符，一级一级地找到最低

集团官网：www.hqyj.com 嵌入式学院：www.embedu.org 企业学院：www.farsight.com.cn 北京|上海|深圳|成都|南京|武汉|西安|广州
移动互联网学院：www.3g-edu.org 物联网学院：www.topsight.cn 咨询电话：400-706-1880，15010390966

的一级成员。只能对最低级的成员进行赋值或存取以及运算。

- 对成员变量可以像普通变量一样进行各种运算（根据其类型决定可以进行的运算）
- 在数组中，数组是不能彼此赋值的，而结构体类型变量可以相互赋值。只有同一结构体类型的结构体变量之间允许相互赋值，而不同结构体类型的结构体变量之间不允许相互赋值，即使两者包含有同样的成员。

➢ 指针

1. 指针的概念

本书在前面的内容里已经提到过指针的概念，简单地说，指针就是地址。在这里，读者可以把计算机的内存看做是一条街道上的一排房屋，每个房屋都可以容纳数据，每个房屋都有一个门牌号用来标识自身的位置。

由于现在大多数的计算机是 32 位的，也就是说地址的字宽是 32 位的，因此，指针也就是 32 位的。可以看到，由于计算机内存的地址都是统一的宽度，而以内存地址作为变量地址的指针也就都是 32 位宽度。

> 注意　请读者务必注意所有数据类型的指针（整型、字符型、数组、结构等）在 32 位机上都是 32 位（4 个字节）。

由于变量的地址是该变量独一无二的标识，因此，只要知道这些地址就一定能找到该变量，就像人们日常生活中写信的地址一样。那么，这些 32 位的变量的指针如何来记录呢？在 C 语言中，可以将这些地址（指针）赋值给专门用于存储地址的变量，这些变量就称为指针变量。

2. 指针变量的定义

指针变量和其他变量一样，在使用之前要先定义，其一般形式为：

```
类型说明符 *变量名;
```

其中，“*”表示一个指针变量，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;
```

以上代码表示 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：

```
staic int *p2;  /*p2 是指向静态整型变量的指针变量*/
float *p3;      /*p3 是指向浮点型变量的指针变量*/
char *p4;       /*p4 是指向字符型变量的指针变量*/
```

对于指针变量的定义，需要注意以下两点。

- 指针变量的变量名是“*”后面的内容，而不是“*p2”、“*p3”。“*”只是说明定义的是一个指针变量。
- 虽然所有的指针变量都是等长的，但仍然需要定义指针的类型说明符。因为对指针变量的其他操作（如加、减等）都涉及指针所指向变量的数据宽度。要注意的是，一个指针变量只能指向同类型的变量。上例中的 p3 只能指向浮点型变量，不能时而指向一个浮点变量，时而又指向一个字符型变量。

3. 指针变量的赋值

指针变量在使用前不仅要定义说明，而且要赋予具体的值。未经赋值的指针变量不能随便使用，否则将造成程序运行错误。指针变量的值只能是变量的地址，不能是其他数据，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，用户不知道变量的具体地址。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。

集团官网：www.hqyj.com 嵌入式学院：www.embedu.org 企业学院：www.farsight.com.cn 北京 | 上海 | 深圳 | 成都 | 南京 | 武汉 | 西安 | 广州
移动互联网学院：www.3g-edu.org 物联网学院：www.topsight.cn 咨询电话：400-706-1880，15010390966

```
int i, *p;
p = &i;
```

同样，我们也可以在定义指针的时候对其初始化，如下所示

```
int  i, *p=&i;
```

上面两种赋值方式是等价的。在第二种方式中的“*”并不是赋值的部分，完整的赋值语句应该是“p=&i;”，“*”只是表明变量 p 是指针类型。这里需要明确的一点是：指针变量只能存放地址（指针），而不能将一个整型数据赋给指针(NULL 除外)。

4. 指针变量的引用

先来看与指针相关的两个运算符

- & 取地址运算符。
- * 指针运算符（间接存取运算符）。

例如“&a”就是取变量 a 的地址，而*b 就是取指针变量 b 所指向的存储单元即对象。通过一个指针访问它所指向的对象的值称为变量的间接访问（通过操作符“*”）。

一个指定类型的指针变量通过引用后可以表示相应类型的变量。比如，一个整型的指针变量引用后可以表示一个整型变量，一个浮点型的指针变量引用后可以表示一个浮点型的数据。这时的操作符“*”就像是打开大门的钥匙，将该大门打开后就能取到对象的内容。

对于指针的引用经常会出现以下的错误：

```
int *a;
*a = 52;
```

虽然已经定义了指针变量 a，但并没有对它进行初始化，也就是说没有让 a 指向一个对象。这时，变量 a 的值是不确定的，即随机指向一个内存单元。这样的代码在执行时通常会出现“segmentation fault”的错误，原因是访问了一个非法地址。因此，在对指针变量进行间接引用之前一定要确保它们已经被指向一个合法的对象。

表 2.16 列举了一些常见的指针表达式，请读者仔细研读其中的内容，务必弄清每个表达式的含义。图中以方框表示地址，以椭圆表示该地址所指向的内容。

表 2.16　　　　指针表达式归纳说明

表达式语句	表达式说明	表达式图示
char ch= ‘a’ ; char *cp = &ch;	初始化指针 cp，并赋初值为变量 ch 的地址	cp → ch
*cp =’ a’ ;	将字符 ‘a’ 赋给 cp 所指向的对象，此时，*cp 的值为 ‘a’（要确保 cp 已经被初始化），cp 的值并没有改变	*cp → ‘a’
*cp + 1	由于“*”操作符的优先级要高于“+”操作符，因此，cp 首先执行取内容操作，即将*cp 的值加 1 为 ‘b’	*cp → ‘b’
*(cp + 1)	让 cp 指向下一个内存单元，再取出其中的内容。因为下一个内存单元有可能是非法地址，所以此操作一定要格外小心，图示中的“？”表示其内容不确定	cp ?
cp++	由于“++”位于 cp 之后，因此在该表达式中，先取出 cp 的值作为表达式的值，再将 cp 的值加 1	原 cp cp 表达式的值
++cp	由于“++”位于 cp 之前，因此在该表达式中，先将 cp 的值加 1，再将 cp 的值作为表达式的值	原 cp cp 表达式的值
*cp++	这时出现了两个运算符，这两个运算符位于同一个优先级，且结合性自右向左，因此，它相当于*(cp++)。由于“++”操作符位于 cp 的右边，因此，这里涉及 3 个步骤。 （1）产生 cp 的一份拷贝。	cp cp 表达式的值

	(2) ++操作符增加 cp 的值。 (3) 在原 cp 拷贝的部分执行间接访问操作，因此，表达式的值是提取原 cp 的内容	
*++cp	这里与上例相似，也是出现了两个运算符，它相当于*(++cp)。由于“++”运算符位于 cp 的左边，因此，该表达式先将 cp 的值加 1，再取其中的内存单元的内容	原 cp / cp / 表达式的值

这里要注意的是，若把一个变量的地址赋给指针意味着指针所指向的内存单元实际上就是存储该变量的内存单元。因此，无论改变指针所指向的内存单元的内容还是直接改变变量的内容，都会有相同的效果。例如，下面的程序就说明了这个问题：

```
#include <stdio.h>
int main()
{
	int *p1, *p2, a, b;
	a = 1; b = 20;
	/*将 a 和 b 的地址赋给 p1 和 p2*/
	p1 = &a;
	p2 = &b;
	/*从打印的结果可以看出，这时 a 和 b 的地址与 p1、p2 相同，
	a 和 b 的内容也与 p1、p2 所指向的内容相同，它们实际是同一块内存单元*/
	printf("a = %d, b = %d\n",a ,b);
	printf("*p1 = %d, *p2 = %d\n", *p1, *p2);
	printf("&a = 0x%x, &b = 0x%x\n",&a ,&b);
	printf("p1 = 0x%x, p2 = 0x%x\n", p1, p2);
	/*此时改变 p1 所指向的内存单元的内容*/
	*p1 = 20;
	/*这时变量 a 的值由原来的 1 变成了 20*/
	printf("after changing *p1, a also changed correspondingly......\n");
	/*可以看出，a 的值和 p1 所指向的内容改变了*/
	printf("a = %d, b = %d\n",a ,b);
	printf("*p1 = %d, *p2 = %d\n", *p1, *p2);

}
```

该程序显示了指针和变量之间的关系：若将变量的地址赋给指针，就相当于让指针指向了该变量。内存中的变化情况如图 2.23 所示。

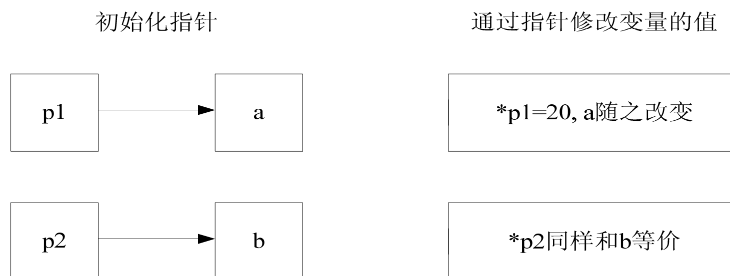


图2.23 指针和变量关系示意图

该程序的运行结果如下所示：

```
a = 1, b = 20
*p1 = 1, *p2 = 20
&a = 0x12ff70, &b = 0x12ff6c
```

```
p1 = 0x12ff70, p2 = 0x12ff6c
after changing *p1, a also changed correspondingly......
a = 20, b = 20
*p1 = 20, *p2 = 20
```

若指针在初始化时未将变量 a 的地址赋给指针变量 p1 而是动态分配内存，那么此后变量 a 的值并不会被指针改变，修改后的程序如下所示：

```
#include <stdio.h>

int main()
{
	int *p1, *p2, a, b;
	a = 1; b = 20;
	/*给 p1、p2 动态分配内存*/
	if((p1=(int *)malloc(sizeof(int))) == NULL)
	{
		perror(malloc);
		return;
	}
	if((p2=(int *)malloc(sizeof(int))) == NULL)
	{
		perror(malloc);
		return;
	};
	printf("a = %d, b = %d\n",a ,b);
	/*此时*p1、*p2 的值还未初始化*/
	printf("*p1 = %d, *p2 = %d\n", *p1, *p2);
	printf("&a = 0x%x, &b = 0x%x\n",&a ,&b);
	/*注意此时，a、b 的地址和 p1、p2 的值是不同的*/
	printf("p1 = 0x%x, p2 = 0x%x\n", p1, p2);
	*p1 = b;
	*p2 = a;
	printf("after changing *p1, a also changed correspondingly......\n");
	/*此时 a、b 的值没有发生改变*/
	printf("a = %d, b = %d\n",a ,b);
	printf("*p1 = %d, *p2 = %d\n", *p1, *p2);
	free(p1);
	free(p2);
}
```

该程序的运行结果如下所示：

```
a = 1, b = 20
*p1 = -842150451, *p2 = -842150451
&a = 0x12ff70, &b = 0x12ff6c
p1 = 0x370fe0, p2 = 0x371018
after changing *p1, a also changed correspondingly......
a = 1, b = 20
*p1 = 20, *p2 = 1
```

由于在该程序中，变量 a、b 和指针 p1、p2 所指向的是不同的存储单元，因此它们之间的赋值互不干扰。

小知识

malloc 函数是用于动态分配内存的，它可以分配指定大小的内存区域，通常用于指针的初始化，其函数原型为：void *malloc(size_t size)；该函数返回已分配的内存区域首地址。若该函数的返回值为 NULL（在下一节中会有讲解），则说明内存分配出错。由于用 malloc 分配的内存无法由程序自动回收，因此在使用完后必须调用函数 free 将所分配的内存释放掉，否则将会出现内存泄漏的问题。

5. NULL 指针

在上一节的实例程序中，读者已经看到了有关 NULL 指针的使用。在 C 语言中指针常量只有 NULL 一个。那么，NULL 究竟代表的是什么含义呢？

C 语言标准中定义了一个 NULL 指针，其值为 0。在实际编程中，NULL 指针的使用是非常普遍的，因为它可以用来表明一个指针目前并未指向任何对象。

例如，在数组中查找特定值的函数在查找成功时应该返回一个指向查找到的数组元素的指针，如果该数组不包含指定的值时，函数就要返回一个 NULL 指针。很多用户都习惯在初始化时将指针设置为 NULL，这是一个好的习惯。但在这里需要注意的是，对 NULL 指针进行间接引用操作是非法的，因为它还没有指向任何对象。因此，在对指针进行间接引用前，应该先判断该指针是否为 NULL，这样才不会出现错误。

6. 指针和数组

◆ 数组的指针

每个变量都有自己的地址(存储该变量的内存单元的编号)，一个数组包含了多个元素，每个数组元素在内存中也都占用存储单元，并且这些存储单元都是连续的。它们在内存中的分布如图 2.24 所示。

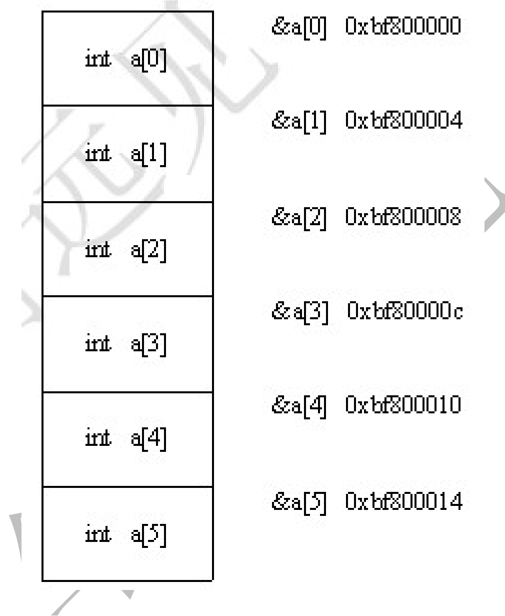


图 2.24 数组在内存中的分布

由图中可以看出，数组中每一个元素都有自己的地址。这些地址可以由各个元素加上取地址符“&”构成。&a[0]就表示数组中第一个元素的地址，&a[1]就表示数组中第二个元素的地址，以此类推。

数组的指针指的是数组的首地址。显然，数组中第一个元素的地址&a[0]就是整个数组的首地址。在 C 语言中，规定使用数组名来代表该数组的起始地址。因此，以下这两个表达式的值是相等的：

```
a 和 &a[0]
```

需要特别说明一点的是，虽然数组名是一个指针，代表该数组的起始地址，但不能将一个指针赋值给一个数组，这是为什么呢？

实际上，数组和指针还是有很大的区别的。数组是一个具有固定数量的数据元素的集合。数组在内存中的位置是程序的运行过程中是无法动态改变的。因此，数组名可以理解成一种指针常量，它可以在运算中作为指针参与，但不允许被赋值。例如下面的程序：

```
#include <stdio.h>

int main()
{
        int a[10], *b;
        int i;
        /*给 b 分配内存空间*/
```

```
        if((b=(int *)malloc(10 * sizeof(int))) == NULL)
        {
            perror("malloc");
            return -1;
        }
        memset(b, 0, 10);
        /*直接将指针 b 赋给数组名 a 是错误的*/
        a=b;
        for(i=0; i < 10; i++)
            printf("a[%d] is %d\n", i, a[i]);
    }
```

在该程序中，试图将指针 b 赋给数组名 a，这时编译器报错指出这是一个不可赋值的左值（即无法给常量赋值）。

小提示 &a[0]包含两个运算符，取地址符和下标运算符。由于它们都位于同一个优先级，并且具有右结合性，所以先取得数组 a 的第一个元素 a[0]，再对这个元素作取地址运算。

◆ 下标的引用

在前面关于指针变量的引用中已经讲述了关于指针表达式的内容。读者可以看到，指针也可以进行一定的运算（比如加、减等），例如，*（a+2）就是取指针 a 后面第二个对象的内容。

注意，指针的一次相加是以其所指向对象的数据宽度为单位的（不是以字节为单位）。也就是说，对于指向整型变量的指针，加 1 操作就相当于向后移动 4 个字节；对于指向字符型变量的的指针，加 1 操作就相当于向后移动 1 个字节。

因此，对于数组而言，指针的相加就相当于依次指向数组中的下一个元素，如图 2.25 所示，这里的 a 指向的是数组的第一个元素。

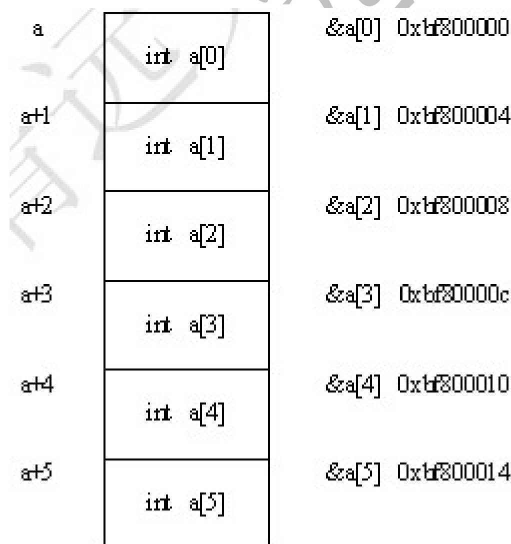


图 2.25 指针运算和数组的关系

由上图可以看出，指针的加法运算和数组的下标运算有如下的对应关系：

```
    数组名 + i  --->   数组名[i]
```

箭头的左边是一个指针常量，它指向箭头右边的变量。事实上，在 C 语言中指针的效率往往高于数组下标。因此，编译器对程序中数组下标的操作全部转换为对指针的偏移量的操作。

表 2.17 总结了对指针和数组的常见等价操作。

表 2.17 指针和数组的常见等价操作

指针 操 作	数组 操 作	说 明
array	&array[0]	数组首地址

*array	array[0]	数组的第一个元素
array + i	&array[i]	数组第 i 个元素的地址
*(array + i)	array[i]	数组的第 i 个元素
*array + b	array[0] + b	数组第 1 个元素的值加 b
*(array+i)+b	array[i] + b	数组第 i 个元素的值加 b
*array++ (当前指向第 i 个元素)	array[i++]	先取得第 i 个元素的值，i 再加 1
*++array (当前指向第 i 个元素)	array[++i]	先将 i 加 1，再取得第 i 个元素的值
*array-- (当前指向第 i 个元素)	array[i--]	先取得第 i 个元素的值，i 再减 1
*--array (当前指向第 i 个元素)	array[--i]	先将 i 减 1，再取得第 i 个元素的值

◆ 数组和指针异同点

（1）相同点

在 C 语言中，指针和数组有很大通用性。

规则 1：表达式中的数组名被编译器当作指向该数组第一个元素的指针。

规则 2：下标总是与指针的偏移量相同。

规则 3：在函数参数的声明中，数组名被编译器当作指向该数组第一个元素的指针。

其中的规则 1 和规则 2 实际上阐述的就是本节前面部分有关数组的指针以及下标引用相关内容，这里对规则 3 做详细阐述。

规则 3 所表明的是若数组在函数的参数中出现，则编译器将数组按照指针的方式来处理。为什么 C 语言要把数组参数作为指针呢？这里是出于效率的考虑。

所有非数组形式的数据实参均以值传递的方式，即对实参做一份拷贝并传递给被调用的函数，函数不能修改实参的值，而只能修改形参的值。

在处理数组的过程中，如果也拷贝整个数组，那么时间和空间上的开销都可能是非常大的。因此，在 C 语言中，采用的是将数组的首地址传递给被调函数的形参，在被调函数中再对数组进行操作。图 2.26 解释了这一过程。

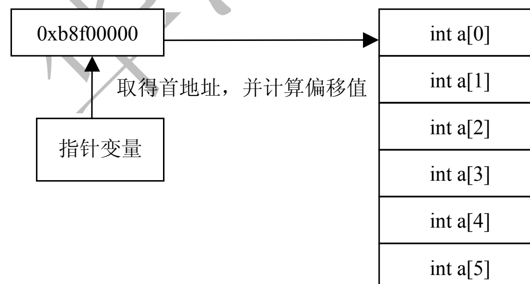


图 2.26 被调函数采用指针的方式

正是因为传递的是数组首地址，所以在函数的定义或声明中可以不用给出数组的维数。编译器在处理时并不会分配指定大小的内存空间，以下的程序是可以正常运行的：

```
#include <stdio.h>
#include <errno.h>
/*定义该函数时不需要定义其中数组的维数*/
void copy(int a[], int b[])
{
        int i;
        for(i=0; i < 10; i++)
                /*在编译器中是采用*(b+i) = *(a+i)的方式来处理的*/
                b[i] = a[i];
}
```

```
int main()
{
        int a[10], b[10];
        int i;
        for(i=0;i< 10; i++)
                a[i] = i;
        /*使用数组名作为实参*/
        copy(a,b);
        for(i = 0; i < 10; i++)
                printf("b[%d] is %d\n", i, b[i]);
}
```

甚至当函数中定义的数组维数小于实际数组的维数时，程序也能正常运行，如将 copy 程序改成如下所示的情况：

```
/*函数中定义的数组维数小于实际数组的维数*/
void copy(int a[5], int b[5])
{
         int i;
         for(i=0; i < 10; i++)
                 /*在编译器中是采用*(b+i) = *(a+i)的方式来处理的*/
                 b[i] = a[i];
}
```

当然，采用这种方法定义会使得程序的可读性变差，不利于程序的后期维护和修改。因此，建议读者在函数的数组声明时采用指针的形式。指针形式的 copy 函数如下所示：

```
void copy(int *a, int *b)
{
        int i;
        for(i=0;i<10; i++)
                *(b+i) = *(a+i);
}
```

（2）不同点

指针和数组实际上是两种截然不同的数据类型。由于在 C 语言中，指针和数组在处理上有很多相似的情况，因此，初学者经常会认为指针等于数组。表 2.18 指出数组和指针的区别，请读者切实掌握。

表 2.18　　指针和数组的不同点

不　同　点	指　　针	数　　组
含义	用于保存数据的地址	用于保存数据
访问数据的方式	采用间接访问，首先取得指针的内容，把它作为地址，然后从这个地址提取数据	直接访问数据
用途	通常用于动态数据结构	通常用于存储固定数目且数据类型相同的元素
内存的分配	定义指针时，编译器只分配存储指针自身的空间	对象空间由编译器自动分配和删除

指针使用中有一个特殊情况，当指针指向一个字符串常量。例如：

```
char *p = "hello world";
```

这时，指针所指向的对象被定义为只读的。如果用户试图通过指针修改这个字符串的值，程序就会出现错误。

注意　不可以对除字符串常量以外类型的指针按以上方法初始化，如“int *a = 1;”，但“int *a = &b;”是正确的。

◆ 多维数组

在 C 语言中实际上并没有多维数组的概念，多维数组实际上是低维数组的组合。例如，二维数组 a[4][3]实际上可以看做 4 个一维数组的组合，它们之间的关系如图 2.27 所示。

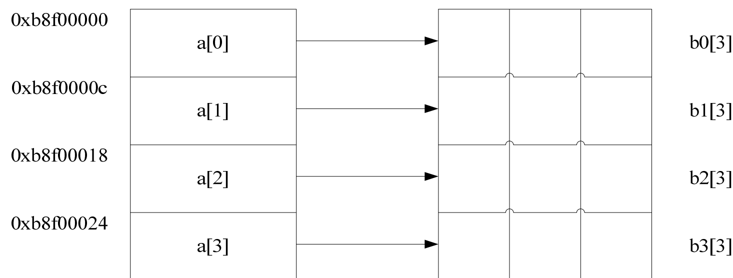


图2.27 多维数组的内存分布

a[i]所代表的是一维数组名（图中的 b0[3]、b1[3]、b2[3]、b3[3]），而不再是原先的一个元素。在多维数组中，数组名 a 依然代表整个数组的首地址，而这时，a+1（a[1]）代表第二行的首地址（b1[3]的首地址），同理，a+2（a[2]）代表的是第三行的首地址。要记住的是，这时的 a[0]代表的是第 0 行，a[1]代表的是第 1 行，依此类推。

注意　确定指针偏移量“1”所代表的单位是通过“1”之前的元素单位来定的，在二维数组中，当偏移量前的元素单位为整个数组时，偏移值单位为行；当偏移量前的元素单位为行时，偏移值单位为行中的元素。

那么，这时如何来表示第 0 行的第一列元素的地址呢？

这时可以用 a[0]+1 来表示，注意此时的 1 代表的是每列元素的字节数而不是每行元素的字节数。因此，a[0]+0、a[0]+1、a[0]+2 分别表示的是 a[0][0]、a[0][1]、a[0][2]的地址（即&a[0][0]、&a[0][1]、&a[0][2]），如图 2.28 所示。

华清远见

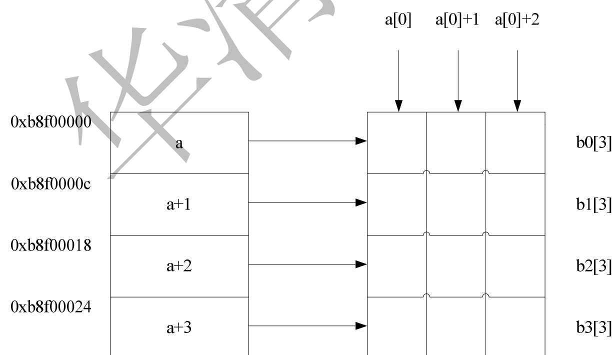


图2.28 用指针访问多维数组

在数组的下标引用中已经提到过，a[0]和*（a+0）是等价的，a[1]和*（a+1）是等价的，因此，a[0]+1 和*(a+0)+1 的值也是等价的，它们都是&a[0][1]。同样，a[1]+2 和*(a+1)+2 也是等价的，它们都是&a[1][2]。

这里，需要对 a[i]的性质作进一步的说明。当 a 是一维数组名时，a[i]代表 a 中的第 i 个元素；当 a 是二维数组名时，a[i]代表多维数组的第 i 行，a+i 是指向第 i 行的一个常指针。读者也可以把 a[i]理解成一个一维数组。由此得到以下两个等价的公式：

```
    a[i] <==>  *(a+i)
 a[i][j] <==>  *( *(a+i) + j)
```

小技巧　在多维数组中，读者可以依次对这些维数进行降维处理，例如，有三维数组 a[5][4][4]，那么 a[i][j]和 a[i]表示的都是地址值，其中 a[i][j]的指针偏移量以一维数组为单位，a[i]的指针偏移量以后面两维数组为单位。

表 2.19 总结了二维数组的指针表示法。

表 2.19 二维数组的指针表示

表 示 形 式	含 义
a	二维数组名，指向一维数组 a[0]，即第 0 行首地址
a[0]、*(a+0)、*a	第 0 行第 0 列元素的首地址
a+1、&a[1]	第 1 行首地址
a[1]、*(a+1)	第 0 行第 0 列元素地址
a[1]+2、*(a+1)+2、&a[1][2]	第 1 行第 2 列元素地址
(a[1]+2)、(*(a+1)+2)、a[1][2]	第 1 行第 2 列元素的值

在实际使用时，通常使用到二维数组就足够了。更多维的处理会导致程序的可读性及维护难度等加大。因此，建议尽量不要使用二维以上的数组。

◆ 用指针来处理字符串

前面已经提到过，在 C 语言中并没有字符串这个数据类型。实际上，C 语言中的字符串是通过字符数组来存储的。而指向字符串的指针的含义是:指针变量的值是字符串的首地址，即指针指向字符串第一个字符。下例中就是读者熟知的字符串的数组表示形式：

```
#include <stdio.h>
void main()
{
        char string[] = "I love Embedded world!";
        printf("%s\n",string);
}
```

字符数组和其他类型的数组一样，可以在定义时赋初值，这里，使用字符串的复制方式——双括号。那么，若将其改为指针形式是怎么样的呢？

```
#include <stdio.h>
void main()
{
        char *string = "I love Embedded world!";
        printf("%s\n",string);
}
```

这就是指向字符串的指针。请读者一定要注意，第一种方式是把字符串的内容保存在数组里，即数组的每一个元素保存一个字符。但指针变量只有 4 个字节，只能用来保存地址。因此，指针变量里的值是字符串“I love Embedded world!”的首地址，而字符串本身存储在内存的其他地方。

在使用指向字符串的指针时，有以下两点需要注意。

①虽然数组名也表示数组的首地址，但由于数组名为指针常量，其值是不能改变的（不能进行自加、自减操作等）。如果把数组的首地址赋给一个指针变量，就可以移动这个指针来访问数组中的元素。

②在使用 scanf 时，其参数前要加上取地址符 &，表明是一个地址。如果指针变量的值已经是一个字符数组的首地址，那么可以直接把指针变量作为参数而不需要再加上取地址符&。

◆ 指针数组

数组是一些同类数据的集合，它们顺序地放在内存中。当数组中的每个元素都是指针时，就引出了指针数组的概念。

所谓指针数组就是每个元素都是同一类型的指针。一维指针数组的定义如下所示：

```
类型名 *数组名[数组长度];
```

例如有以下定义：

```
int *p[6];
char *n[9];
```

这样就定义了一个指向 int 类型的指针数组和一个指向 char 类型的指针数组。要注意，这里由于“[]”的优先级高于“*”，因此，数组名 p 先与“[]”结合，这就构成了一个数组的形式。

那么，指针数组何时使用呢？指针数组最常见的用途是用于指向多个字符串。数组中的每个指针元素都指向一个字符串，这样就可以实现对字符串的灵活操作。下面就以 main 函数的形参为例进行介绍。
本书前面例子里的 main 函数，后面跟着的是空括号，不带任何参数。实际上，main 函数可以带上参数。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。第一个形参（argc）必须是整型变量，第二个形参（argv）必须是指向字符串的指针数组。因此，加上形参说明后，main 函数变成下面的形式：

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

由于 main 函数不能被其他函数调用，所以不可能在程序内部得到实参。那么，是在何处把实参值赋予 main 函数的形参呢？实际上，main 函数的实参是从操作系统的命令行上获得的。当我们要运行一个可执行文件时，在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。
提示符下命令行的一般形式为：

```
可执行文件名 参数 1  参数 2…
```

argc 参数表示了命令行中参数的个数（注意：文件名本身也算一个参数），argc 的值是在输入命令行时由系统按实际参数的个数自动计算的。举个例子：

```
linux@farsight~$ E624  BASIC  dbase  FORTRAN
```

由于文件名 E624 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 2.29 所示。

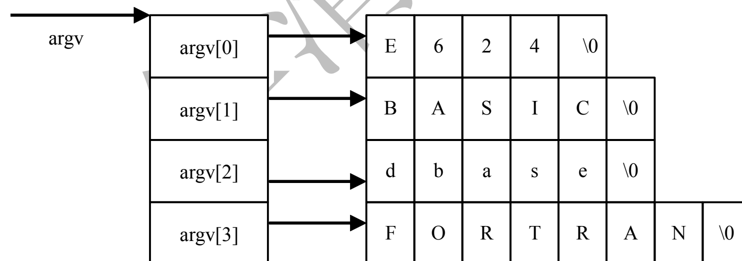


图 2.29　argv 指针数组示意图

要注意的是，在指针数组中，数组名仍然表示数组的首地址，但这里数组的首地址是指针数组的首地址，数组的偏移量仍然表示相应数组的元素。但要注意的是，这时数组内的元素都是地址，因此，若要读取数组内指针所指向的内容要使用取内容符“*”。
下面的程序显示了指针数组的使用方法。

```
#include <stdio.h>
void main(int argc, char *argv[])
{
      int i = 0;
      while(argc > 1)
      {
      /*指针数组的下标引用*/
            argv[++i];
      /*输出指针数组的第 i 个元素所指向的内容*/
            printf("%s\n", argv[i]);
```

```
        /*计数器减 1*/
            --argc;
        }
    }
```

该程序的运行结果如下所示：

```
C:\>E624 BASIC dbase FORTRAN
BASIC
dbase
FORTRAN
```

从该程序中可以看到，使用“argv[i]”可以以下标引用的方式取得数组中的元素，即 argv[i]指向某一字符串。

◆ 指向指针的指针

当指针指向的对象也是一个指针时，称为指向指针的指针。它的定义方式如下所示：

```
数据类型 **变量名;
```

要注意的是，这里“指针的指针”还是变量，它的值是其他指针变量的地址，例如有如下定义：

```
char **p;
```

这样就定义了一个指向指针的指针。可以用这个指针来指向一个指针数组。上面的程序改为用指向指针的指针：

```
void main(int argc, char **argv)
{
        /*argc 在程序开始时自动赋值*/
        while(argc > 1)
        {
                /*数组名代表数组首地址*/
                ++argv;
                /*打印出数组所指向的内容，注意“*argv”为地址*/
                printf("%s\n", *argv);
                /*计数器减 1*/
                --argc;
        }
}
```

从该程序中可以看到，使用“++argv”和使用“argv[++i]”是等价的，可以取得数组中的其他元素，“*argv”指向的是某个字符串。

7. 结构体数组和结构体指针

（1）结构体数组

结构体数组就是具有相同结构体类型的变量的集合。假如要定义一个班级 40 个同学的姓名、性别、年龄和住址，可以定义一个结构体数组，如下所示：

```
struct{
char name[8];
char sex;
int age;
char addr[20];
}student[40];
```

也可定义为：

```
struct stu{
char name[8];
char sex;
int age;
char addr[40];
};
struct stu student[40];
```

数组中每个元素都是结构体变量，要访问其成员的方法为：

```
结构体数组元素.成员名
```

例如：

```
student[0].name
student[30].age
```

实际上结构体数组相当于一个二维构造，第一维是结构体数组元素，每个元素是一个结构体变量，第二维是结构体成员。

（2）结构体指针

它由一个加在结构体变量名前的“*”操作符来定义，例如用前面已说明的结构体定义一个结构体指针：

```
struct stu{
char name[8];
char sex;
int age;
char addr[40];
}*student, s1;
```

当然也可省略结构体指针名，只作结构体说明，然后再用下面的语句定义结构体指针。

```
struct stu *student, s1;
```

使用结构体指针对结构体成员的访问，与结构体变量对结构体成员的访问在表达方式上有所不同。结构体指针对结构体成员的访问表示为：

```
结构体指针名->结构体成员
```

例如：

```
student->name
```

实际上，student->name 就是(*student).name 的缩写形式。

当用结构体指针来访问结构体成员时必须先让这个指针指向一个结构体变量：

```
student = &s1;
```

注意　结构体变量的首地址就是其第一个成员的首地址。

2.7 函数

2.7.1 概述

函数是能完成一定功能的执行代码段。函数就像是一个“黑盒子”，如图 2.30 所示，用户只要将数据送进去就能得到结果，而函数内部究竟是如何工作的调用程序并不关心。

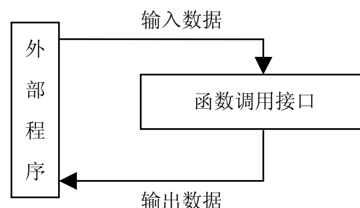


图 2.30　函数与外部程序的关系

外部程序所关心的仅限于输入给函数的数据以及函数输出的数据。函数提供了编制程序的手段，使之易于理解、编写、修改和维护。可以说，函数是实现模块化编程的重要工具。

C 语言程序中的函数在数目并没有上限。但一个 C 程序中有且仅有一个以 main 为名的函数，这个函数被称为主函数。整个程序就从这个主函数开始执行，在主函数中还可以调用其他函数来完成所需要的工作。解决问题时人们习惯把一个规模较大的问题划分成若干个子问题。对每个子问题编写一个函数来解决。所以，C 语言程序一般是由大量的小函数而不是由少量的大函数构成，即所谓“小函数构成大程序”。这样做的好处是让各个模块相互独立，并且任务单一。

表 2.20 列举了常见的函数分类

表 2.20　　　　常见的函数分类说明

分类角度	分　类	说　明
函数定义的角度	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，只需在程序中包含有该函数原型的头文件即可在程序中直接调用，如 printf 等
	用户定义函数	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
有无返回值	有返回值函数	被调用执行完后将向调用者返回一个值
	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
主调函数和被调函数之间数据传送的角度	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
	有参函数	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
库函数功能	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
	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
	目录路径函数	用于文件目录和路径操作
	诊断函数	用于内部错误检测
	图形函数	用于屏幕管理和各种图形功能
	输入输出函数	用于完成输入输出功能
	接口函数	用于与 DOS、BIOS 和硬件的接口
	字符串函数	用于字符串操作和处理
	内存管理函数	用于内存管理
	数学函数	用于数学函数计算
	日期和时间函数	用于日期、时间转换操作
	进程控制函数	用于进程管理和控制

2.7.2　函数定义和声明

➢　函数定义

函数定义就是函数体的实现。无参函数的一般形式为：

```
类型说明符 函数名()
{
```

华清远见

```
        语句
    }
```

其中类型说明符和函数名称为函数头。
类型说明符指明了本函数的类型，即函数返回值的类型。这里的类型说明符可以包括数据类型说明符、存储类型说明符以及时间域说明符。
函数名是用户定义的标识符，函数名后有一个空括号。“{}”中的内容称为函数体。函数体中有若干条语句，实现特定的功能。如果函数不需要有返回值时，函数类型说明符可以写为 void。
例如，下面是一个很简单的函数定义：

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

这里的 Hello 是一个无参函数。当被其他函数调用时，输出 Hello world！字符串。
有参函数的一般形式为：

```
类型说明符 函数名(形式参数列表)
{
     类型说明
     语句
}
```

可以看到，有参函数比无参函数多了形式参数列表。列表中的参数可以是任意类型的变量，各参数之间用逗号分隔。在进行函数调用时，调用函数将赋予这些形式参数实际的值。如将上述的 Hello 函数增加一个整型参数，就形成了一个有参函数，如下所示：

```
void Hello(int i)
{
     printf ("Hello,world ! The num is %d\n", i);
}
```

这里的变量“i”就是形式参数，在运行时由主函数传值进来。

➢ 函数声明

当编译器遇到一个函数调用时，它产生代码来传递参数，同时接收该函数返回的值（如果有的话）。但编译器是如何知道被调用函数期望接受的是什么类型和多少数量的参数，如何知道该函数的返回值（如果有的话）的类型呢？
在 C 语言中，用户可以通过两种方法向编译器提供一些关于函数的特定信息。

①如果同一源文件的前面已经出现了该函数的定义，那么编译器就会记住它的参数数量和类型，以及函数的返回值类型。

②如果在同一源文件的前面没有该函数的定义，则需要提供该函数的函数原型。用户自定义的函数原型通常可以一起写在头文件中，通过头文件引用的方式来声明。

函数原型的一般形式为。
（1）函数类型 函数名（参数类型 1，参数类型 2...）;
（2）函数类型 函数名（参数类型 1 参数名 1，参数类型 2 参数名 2...）;
第一种形式是基本的形式，同时为了便于阅读程序，也允许在函数原型中加上参数名，这就成了第二种形式。但编译器实际上并不检查参数名，参数名可以任意改变。
函数原型与函数头要求保持一致，即函数类型、函数名、参数个数、参数类型和参数顺序完全一样。一般为了方便声明函数类型，读者可以直接将函数头复制过来再加上“；”即可。

小知识 实际上，如果在调用函数之前没有对函数进行声明，则编译系统会把第一次遇到的该函数形式（函数定义或函数调用）作为函数的声明，并将函数类型默认为 int 型。

细心的读者可能还记得，在前面的内容中介绍过变量的声明。对于全局变量的声明可以加上 extern 标识，同样对于函数的声明，也可以使用 extern。如果函数的声明中带有关键字 extern，是告诉编译器这个函数是在别的源文件里定义的。

2.7.3 函数的参数、返回值和调用方法

➢ 函数的参数

函数的参数分为形参和实参两种。

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。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。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，如图 2.31 所示。

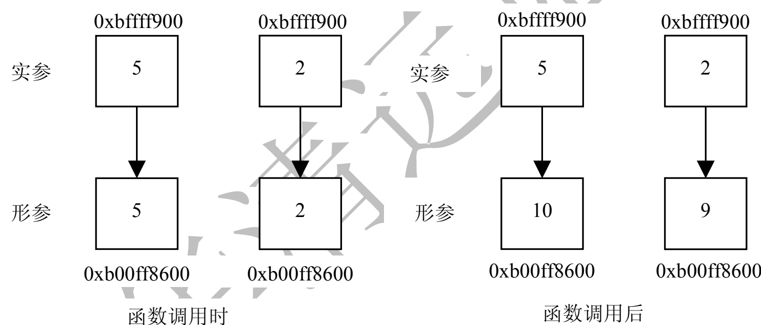


图 2.31 实参和形参在函数调用时的变化情况

从图中可以看出，实参和形参所占用的存储单元完全是独立的。在函数调用时，实参把存储单元中的数据赋值给形参的存储单元；而在函数调用后，若形参的值发生了改变，它也无法传递给实参（由于参数的传递是单向的，从实参传递给形参）。因此，若希望从被调用函数将值传递给调用函数只能通过返回语句（return）或以指针的形式。

➢ 函数的返回值

函数的返回值是值被调用函数返回给调用函数的值。

（1）函数的返回值只能通过 return 语句返回主调函数，return 语句的一般形式为：

```
return 表达式;
```

或者

```
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，因此只能返回一个值。

（2）函数返回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，则以函数定义中的类型为准，自动进行类型转换。

（3）如函数返回值为整型，在函数定义时可以省去类型说明。

（4）没有返回值的函数，可以明确定义为空类型，类型说明符为 void。

➢ 函数的调用

在前面的两小节中，读者已经学习了函数的参数传递以及函数的返回值。这样，函数的调用就变得很简单，其一般形式为：

```
函数名（实参列表);
```

如果是调用无参函数，则实参表列可以没有，但括弧不能省略。如果实参表列包含多个实参则各参数间用逗号隔开。实参与形参的个数应相等，类型应一致。实参与形参按顺序对应，一一传递数据。这里，对实参表取值的顺序并不是确定的，有的系统按自左至右顺序求实参的值，有的系统则按自右至左顺序。函数的参数传递采用的是值传递的方式。

按照函数在程序中出现的不同位置，有以下 3 种函数调用方式。

①函数语句：把函数调用作为一个语句。这时不要求函数带回值，只要求函数完成一定的操作，如：

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

②函数表达式：函数出现在一个表达式中，这种表达式称为函数表达式。这时要求函数带回一个确定的值以参加表达式的运算，如：

```
i = sum(a,b);
```

③函数参数：函数调用作为一个函数的实参。函数调用作为函数的参数，实质上也是函数表达式形式调用的一种，因为函数的参数本来就要求是表达式形式，如：

```
printf("the sum of a and b is %d\n",sum(a, b));
```

注意　被调函数必须是已经声明了的函数，或者被调函数的位置位于调用函数之前。

2.8　attribute 机制介绍

GNU C 的一大特色就是__attribute__机制。__attribute__可以设置函数属性（Function Attribute）、变量属性（Variable Attribute）和类型属性（Type Attribute）。

__attribute__书写特征是：__attribute__前后都有两个下划线，并切后面会紧跟一对原括弧，括弧里面是相应的__attribute__参数。

__attribute__语法格式为：　__attribute__ ((attribute-list))

➢ 函数属性（Function Attribute）

函数属性可以帮助开发者把一些特性添加到函数声明中，从而可以使编译器在错误检查方面的功能更强大。 __attribute__机制也很容易同非 GNU 应用程序做到兼容。GNU CC 需要使用 –Wall 编译选项来击活该功能，这是控制警告信息的一个很好的方法。下面介绍几个常见的属性参数。

__attribute__ format

该__attribute__属性可以给被声明的函数加上类似 printf 或者 scanf 的特征。它可以使编译器检查函数声明和函数实际调用参数之间的格式化字符串是否匹配。该功能十分有用，尤其是处理一些很难发现的 bug。

format 的语法格式为：　format (archetype, string-index, first-to-check)

format 属性告诉编译器，按照 printf, scanf, strftime 或 strfmon 的参数表格式规则对该函数的参数进行检查。"archetype"指定是哪种风格；"string-index"指定传入函数的第几个参数是格式化字符串；"first-to-check"指定从函数的第几个参数开始按上述规则进行检查。

具体使用格式如下：

__attribute__((format(printf,m,n)))

__attribute__((format(scanf,m,n)))

其中参数 m 与 n 的含义为：

m：第几个参数为格式化字符串（format string）；

华清远见

n：参数集合中的第一个，即参数“…”里的第一个参数在函数参数总数排在第几。
在使用上，__attribute__((format(printf,m,n)))是最常用的，而另一种很少见到。下面举例说明，其中 myprint 为自己定义的一个带有可变参数的函数，其功能类似于 printf：

```
//m=1；n=2
extern void myprint(const char *format,...) __attribute__((format(printf,1,2)));
//m=2；n=3
extern void myprint(int l，const char *format,...) __attribute__((format(printf,2,3)));
```

需要特别注意的是，如果 myprint 是一个函数的成员函数，那么 m 和 n 的值可有点“悬乎”了，例如：

```
//m=3；n=4
extern void myprint(int l，const char *format,...) __attribute__((format(printf,3,4)));
```

其原因是，类成员函数的第一个参数实际上一个“隐身”的“this”指针。（有点 C++基础的都知道点 this 指针，不知道你在这里还知道吗？）

这里给出测试用例：attribute.c，代码如下：

```
1：
2：extern void myprint(const char *format,...) __attribute__((format(printf,1,2)));
3：
4：void test()
5：{
6：  myprint("i=%d\n",6);
7：  myprint("i=%s\n",6);
8：  myprint("i=%s\n","abc");
9：  myprint("%s,%d,%d\n",1,2);
10：}
```

运行$gcc –Wall –c attribute.c attribute 后，输出结果为：

```
attribute.c: In function `test':
attribute.c:7: warning: format argument is not a pointer (arg 2)
attribute.c:9: warning: format argument is not a pointer (arg 2)
attribute.c:9: warning: too few arguments for format
```

如果在 attribute.c 中的函数声明去掉__attribute__((format(printf,1,2)))，再重新编译，既运行$gcc –Wall –c attribute.c attribute 后，则并不会输出任何警告信息。

注意，默认情况下，编译器是能识别类似 printf 的“标准”库函数。

__attribute__ noreturn

该属性通知编译器函数从不返回值，当遇到类似函数需要返回值而却不可能运行到返回值处就已经退出来的情况，该属性可以避免出现错误信息。C 库函数中的 abort（）和 exit（）的声明格式就采用了这种格式，如下所示：

```
extern void exit(int) __attribute__((noreturn));
extern void abort(void) __attribute__((noreturn));
```

为了方便理解，大家可以参考如下的例子：

```
//name: noreturn.c ；测试__attribute__((noreturn))
extern void myexit();
```

华清远见

```
int test(int n)
{
 if ( n > 0 )
 {
 myexit();
 /* 程序不可能到达这里*/
 }
 else
 return 0;
}
```

编译显示的输出信息为：

```
$gcc –Wall –c noreturn.c
noreturn.c: In function `test':
noreturn.c:12: warning: control reaches end of non-void function
```

警告信息也很好理解，因为你定义了一个有返回值的函数 test 却有可能没有返回值，程序当然不知道怎么办了！

加上__attribute__((noreturn))则可以很好的处理类似这种问题。把

extern void myexit();

修改为：

extern void myexit() __attribute__((noreturn));

之后，编译不会再出现警告信息。

__attribute__ const

该属性只能用于带有数值类型参数的函数上。当重复调用带有数值参数的函数时，由于返回值是相同的，所以此时编译器可以进行优化处理，除第一次需要运算外，其它只需要返回第一次的结果就可以了，进而可以提高效率。该属性主要适用于没有静态状态（static state）和副作用的一些函数，并且返回值仅仅依赖输入的参数。

为了说明问题，下面举个非常“糟糕”的例子，该例子将重复调用一个带有相同参数值的函数，具体如下：

```
extern int square(int n) __attribute__((const));
...
 for (i = 0; i < 100; i++ )
 {
 total += square(5) + i;
 }
```

通过添加__attribute__((const))声明，编译器只调用了函数一次，以后只是直接得到了相同的一个返回值。

事实上，const 参数不能用在带有指针类型参数的函数中，因为该属性不但影响函数的参数值，同样也影响到了参数指向的数据，它可能会对代码本身产生严重甚至是不可恢复的严重后果。

并且，带有该属性的函数不能有任何副作用或者是静态的状态，所以，类似 getchar（）或 time（）的函数是不适合使用该属性的。

华清远见

-finstrument-functions

该参数可以使程序在编译时，在函数的入口和出口处生成 instrumentation 调用。恰好在函数入口之后并恰好在函数出口之前，将使用当前函数的地址和调用地址来调用下面的 profiling 函数。（在一些平台上，__builtin_return_address 不能在超过当前函数范围之外正常工作，所以调用地址信息可能对 profiling 函数是无效的。）

```
void __cyg_profile_func_enter(void *this_fn, void *call_site);
void __cyg_profile_func_exit(void *this_fn, void *call_site);
```

其中，第一个参数 this_fn 是当前函数的起始地址，可在符号表中找到；第二个参数 call_site 是指调用处地址。

instrumentation 也可用于在其它函数中展开的内联函数。从概念上来说，profiling 调用将指出在哪里进入和退出内联函数。这就意味着这种函数必须具有可寻址形式。如果函数包含内联，而所有使用到该函数的程序都要把该内联展开，这会额外地增加代码长度。如果要在 C 代码中使用 extern inline 声明，必须提供这种函数的可寻址形式。

可对函数指定 no_instrument_function 属性，在这种情况下不会进行 instrumentation 操作。例如，可以在以下情况下使用 no_instrument_function 属性：上面列出的 profiling 函数、高优先级的中断例程以及任何不能保证 profiling 正常调用的函数。

no_instrument_function

如果使用了-finstrument-functions ，将在绝大多数用户编译的函数的入口和出口点调用 profiling 函数。使用该属性，将不进行 instrument 操作。

constructor/destructor

若函数被设定为 constructor 属性，则该函数会在 main（）函数执行之前被自动的执行。类似的，若函数被设定为 destructor 属性，则该函数会在 main（）函数执行之后或者 exit（）被调用后被自动的执行。拥有此类属性的函数经常隐式的用在程序的初始化数据方面。

这两个属性还没有在面向对象 C 中实现。

同时使用多个属性

可以在同一个函数声明里使用多个__attribute__，并且实际应用中这种情况是十分常见的。使用方式上，你可以选择两个单独的__attribute__，或者把它们写在一起，可以参考下面的例子：

```
/* 把类似 printf 的消息传递给 stderr 并退出 */
extern void die(const char *format, ...)
  __attribute__((noreturn))
  __attribute__((format(printf, 1, 2)));
```

或者写成

```
extern void die(const char *format, ...)
  __attribute__((noreturn, format(printf, 1, 2)));
```

如果带有该属性的自定义函数追加到库的头文件里，那么所以调用该函数的程序都要做相应的检查。

和非 GNU 编译器的兼容性

庆幸的是，__attribute__设计的非常巧妙，很容易作到和其它编译器保持兼容，也就是说，如果工作在其它的非 GNU 编译器上，可以很容易的忽略该属性。即使__attribute__使用了多个参数，也可以很容易的使用一对圆括弧进行处理，例如：

华清远见

```
/* 如果使用的是非 GNU C, 那么就忽略__attribute__ */
#ifndef __GNUC__
# define __attribute__(x) /*NOTHING*/
#endif
```

需要说明的是，__attribute__适用于函数的声明而不是函数的定义。所以，当需要使用该属性的函数时，必须在同一个文件里进行声明，例如：

```
/* 函数声明 */
void die(const char *format, ...) __attribute__((noreturn))
 __attribute__((format(printf,1,2)));

void die(const char *format, ...)
{
 /* 函数定义 */
}
```

更多的属性含义参考：
http://gcc.gnu.org/onlinedocs/gcc-4.0.0/gcc/Function-Attributes.html

变量属性（Variable Attributes）

关键字__attribute__也可以对变量（variable）或结构体成员（structure field）进行属性设置。这里给出几个常用的参数的解释，更多的参数可参考本文给出的连接。

在使用__attribute__参数时，你也可以在参数的前后都加上“__”（两个下划线），例如，使用__aligned__而不是 aligned，这样，你就可以在相应的头文件里使用它而不用关心头文件里是否有重名的宏定义。

aligned (alignment)

该属性规定变量或结构体成员的最小的对齐格式，以字节为单位。例如：

```
int x __attribute__ ((aligned (16))) = 0;
```

编译器将以 16 字节（注意是字节 byte 不是位 bit）对齐的方式分配一个变量。也可以对结构体成员变量设置该属性，例如，创建一个双字对齐的 int 对，可以这么写：

```
struct foo { int x[2] __attribute__ ((aligned (8))); };
```

如上所述，你可以手动指定对齐的格式，同样，你也可以使用默认的对齐方式。如果 aligned 后面不紧跟一个指定的数字值，那么编译器将依据你的目标机器情况使用最大最有益的对齐方式。例如：

```
short array[3] __attribute__ ((aligned));
```

选择针对目标机器最大的对齐方式，可以提高拷贝操作的效率。

aligned 属性使被设置的对象占用更多的空间，相反的，使用 packed 可以减小对象占用的空间。

需要注意的是，attribute 属性的效力与你的连接器也有关，如果你的连接器最大只支持 16 字节对齐，那么你此时定义 32 字节对齐也是无济于事的。

packed

华清远见

使用该属性可以使得变量或者结构体成员使用最小的对齐方式，即对变量是一字节对齐，对域（field）是位对齐。

下面的例子中，x 成员变量使用了该属性，则其值将紧放置在 a 的后面：

```
struct test
{
char a;
int x[2] __attribute__ ((packed));
};
```

其它可选的属性值还可以是：cleanup，common，nocommon，deprecated，mode，section，shared，tls_model，transparent_union，unused，vector_size，weak，dllimport，dlexport 等，

详细信息可参考：

http://gcc.gnu.org/onlinedocs/gcc-4.0.0/gcc/Variable-Attributes.html#Variable-Attributes

类型属性（Type Attribute）

关键字__attribute__也可以对结构体（struct）或共用体（union）进行属性设置。大致有六个参数值可以被设定，即：aligned, packed, transparent_union, unused, deprecated 和 may_alias。

在使用__attribute__参数时，你也可以在参数的前后都加上“__”（两个下划线），例如，使用__aligned__而不是 aligned，这样，你就可以在相应的头文件里使用它而不用关心头文件里是否有重名的宏定义。

aligned (alignment)

该属性设定一个指定大小的对齐格式（以字节为单位），例如：

```
struct S { short f[3]; } __attribute__ ((aligned (8)));
typedef int more_aligned_int __attribute__ ((aligned (8)));
```

该声明将强制编译器确保（尽它所能）变量类型为 struct S 或者 more-aligned-int 的变量在分配空间时采用 8 字节对齐方式。

如上所述，你可以手动指定对齐的格式，同样，你也可以使用默认的对齐方式。如果 aligned 后面不紧跟一个指定的数字值，那么编译器将依据你的目标机器情况使用最大最有益的对齐方式。例如：

```
struct S { short f[3]; } __attribute__ ((aligned));
```

这里，如果 sizeof（short）的大小为 2（byte），那么，S 的大小就为 6。取一个 2 的次方值，使得该值大于等于 6，则该值为 8，所以编译器将设置 S 类型的对齐方式为 8 字节。

aligned 属性使被设置的对象占用更多的空间，相反的，使用 packed 可以减小对象占用的空间。

需要注意的是，attribute 属性的效力与你的连接器也有关，如果你的连接器最大只支持 16 字节对齐，那么你此时定义 32 字节对齐也是无济于事的。

packed

使用该属性对 struct 或者 union 类型进行定义，设定其类型的每一个变量的内存约束。当用在 enum 类型定义时，暗示了应该使用最小完整的类型（it indicates that the smallest integral type should be used）。

下面的例子中，my-packed-struct 类型的变量数组中的值将会紧紧的靠在一起，但内部的成员变量 s 不会被“pack”，如果希望内部的成员变量也被 packed 的话，my-unpacked-struct 也需要使用 packed 进行相应的约束。

```
struct my_unpacked_struct
{
```

华清远见

```
  char c;
  int i;
};

struct my_packed_struct
{
  char c;
  int i;
  struct my_unpacked_struct s;
}__attribute__ ((__packed__));
```

其它属性的含义见：
http://gcc.gnu.org/onlinedocs/gcc-4.0.0/gcc/Type-Attributes.html#Type-Attributes
变量属性与类型属性举例
下面的例子中使用__attribute__属性定义了一些结构体及其变量，并给出了输出结果和对结果的分析。
程序代码为：

```
struct p
{
int a;
char b;
char c;
}__attribute__((aligned(4))) pp;

struct q
{
int a;
char b;
struct n qn;
char c;
}__attribute__((aligned(8))) qq;

int main()
{
printf("sizeof(int)=%d,sizeof(short)=%d.sizeof(char)=%d\n",sizeof(int),sizeof(short),sizeof(char));
printf("pp=%d,qq=%d \n", sizeof(pp),sizeof(qq));
return 0;
}
```

输出结果：

```
sizeof(int)=4,sizeof(short)=2.sizeof(char)=1
pp=8,qq=24
```

分析：
sizeof(pp):
sizeof(a)+ sizeof(b)+ sizeof(c)=4+1+1=6<23=8= sizeof(pp)

华清远见

sizeof(qq):

sizeof(a)+ sizeof(b)=4+1=5

sizeof(qn)=8;即 qn 是采用 8 字节对齐的，所以要在 a，b 后面添 3 个空余字节，然后才能存储 qn，4+1+（3）+8+1=17

因为 qq 采用的对齐是 8 字节对齐，所以 qq 的大小必定是 8 的整数倍，即 qq 的大小是一个比 17 大又是 8 的倍数的一个最小值，由此得到

17<24+8=24= sizeof(qq)

2.3 系统调用和应用程序编程接口

2.3.1 系统调用

操作系统是从硬件抽象出来的平台，在该平台上用户可以运行应用程序。它负责直接与硬件交互，向用户程序提供公共服务，并使它们同硬件特性隔离。一种计算机硬件要运行 Linux 系统，至少需要提供两种运行模式：高优先级的核心模式和低优先级的用户模式。

实际上许多计算机都有两种以上的执行模式。如：intel 80x86 体系结构就有四层执行特权，内层特权最高。Linux 只需要两层即可以了：核心运行在高优先级，称之为核心态；其它应用软件包括 shell，文本编辑器，Xwindow 等等都是在低优先级运行，称之为用户态。之所以采取不同的执行模式主要原因是为了保护系统。由于用户进程在较低的特权级别上运行，它们将不会意外或故意的破坏其它进程或内核。程序造成的破坏会被局部化而不影响系统中其它活动或者进程。当用户进程需要完成特权模式下才能完成的某些功能时，必须严格按照系统调用提供接口才能进入特权模式，然后执行调用所提供的有限功能。

在 Linux 中，为了更好地保护内核，将程序的运行空间分为内核空间和用户空间（也就是常称的内核态和用户态），它们分别运行在不同的级别上，逻辑上是相互隔离的。因此，用户进程在通常情况下不允许访问内核数据，也无法使用内核函数，它们只能在用户空间操作用户数据，调用用户空间的函数。而系统调用正是操作系统向用户程序提供支持的接口，通过这些接口应用程序向操作系统请求服务，控制转向操作系统，而操作系统在完成服务后，将控制和结果返回给用户程序。

Linux 系统调用部分是非常精简的系统调用（只有 280 个左右），它继承了 Unix 系统调用中最基本和最有用的部分。这些系统调用按照功能逻辑大致可分为进程控制、进程间通信、文件系统控制、系统控制、存储管理、网络管理、socket 控制、用户管理等几类。

2.3.2 应用程序编程接口（API）

系统调用并不直接与程序员进行交互，它仅仅是一个通过软中断机制向内核提交请求以获取内核服务的接口。系统调用接口看起来和 C 程序中的普通函数调用很相似，它们通常是通过库把这些函数调用映射成进入操作系统所需要的操作。 这些操作只是提供一个基本功能集，而通过库对这些操作的引用和封装，可以形成丰富而且强大的系统调用库。这里体现了机制与策略相分离的编程思想——系统调用只是提供访问核心的基本机制，而策略是通过系统调用库来体现。所以，实际使用中程序员调用的通常是用户编程接口——API(Application Programming Interface)。

例如获取进程号的 API 函数 getpid（）对应 getpid 系统调用。但并不是所有的函数都对应一个系统调用，有时，一个 API 函数会需要几个系统调用来共同完成函数的功能，甚至还有一些 API 函数不需要调用相应的系统调用（因此它所完成的不是内核提供的服务）。

集团官网：www.hqyj.com 嵌入式学院：www.embedu.org 企业学院：www.farsight.com.cn 北京 | 上海 | 深圳 | 成都 | 南京 | 武汉 | 西安 | 广州
移动互联网学院：www.3g-edu.org 物联网学院：www.topsight.cn 咨询电话：400-706-1880，15010390966

在 Linux 中，应用程序编程接口（API）遵循了在 Unix 中最流行的应用编程界面标准——POSIX 标准。POSIX 标准是由 IEEE 和 ISO/IEC 共同开发的标准系统。该标准基于当时现有的 Unix 实践和经验，描述了操作系统的系统调用编程接口（实际上就是 API），用于保证应用程序可以在源代码一级上在多种操作系统上移植运行。这些系统调用编程接口主要是通过 C 库（libc）实现的。

2.3.3 系统命令

系统命令相对 API 更高了一层，它实际上一个可执行程序，它的内部调用了应用编程接口（API）来实现相应的功能，它们之间的关系如图 2.32 所示。

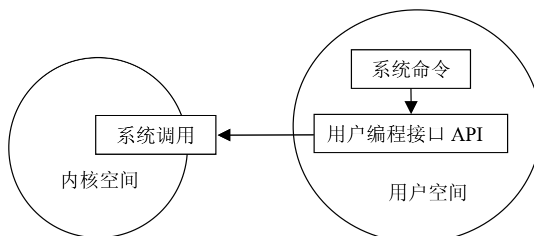


图 2.32 系统调用、API 及系统命令之间的关系

华清远见

本 章 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内存的占用情况。
然后本章讲解了基本的常量和变量，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。
接下来本章分别介绍了算术、赋值、逗号、位、关系、逻辑运算和表达式，以及 sizeof 操作符和条件运算符。这里需要读者着重掌握的是各种运算符的优先级关系。
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。

动 手 练 练

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，当你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

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路 250 号银海大厦 11 层 B 区，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-25590506

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218

广州地址：广州市天河区中山大道 268 号天河广场 3 层，电话：020-28916067

华清远见