C 语言的一些编程技巧及其深层理解

C 语言，是一门非常灵活而强大的编程语言。同样一个算法、一个功能，我们可以把它写得中规中矩，也可以把它写得晦涩难懂。而且很多自诩为编程高手的人，偏偏就喜欢把程序写成天书，认为让别人看不懂，却能实现正确的功能，此乃技术高超的表现。我不评价这样的作法是否可取，因为每个人都有各自的风格和个性。让他违背意愿去编程，那么编程可能就会变得索然无味，毫无乐趣。我只想说，要把程序写出格调，是需要资本的，是需要对 C 语言有较深入了理解的。很多时候不是我们想把程序写得难懂，而是我们要去看懂别人的程序。在这章中，振南列举一些我曾经见过和使用过的编程技巧，并进行深入的解析。

一、字符串的实质就是指针

字符串是 C 语言中最基础的概念，也是最常被用到的。在嵌入式开发中，我们经常要将一些字符串通过串口显示到串口助手或调试终端上，作为信息提示，以便让我们了解程序的运行情况；或者要将一些常量的值转为字符串，来显示到液晶等显示设备上。

那么 C 语言中的字符串到底是什么？其实字符串本身就是一个指针，它的值(即指针所指向的地址)就是字符串首字符的地址。

为了解释这个问题，我经常会举这样一个例子：如何将一个数值转化为相应的 16 进制字符串。比如，把 100 转为”0X64”。

我们可以写这样一个函数：

```
void Value2String(unsigned char value,char *str)
{
  unsigned char temp=0;

  str[0]='0';str[1]='X';str[4]=0;

  temp=value>>4;
  if(temp>=0 && temp<=9) str[2]='0'+temp;
  else if(temp>=10 && temp<=15) str[2]='A'+temp-10;

  temp=value&0X0F;
  if(temp>=0 && temp<=9) str[3]='0'+temp;
  else if(temp>=10 && temp<=15) str[3]='A'+temp-10;
}
```

没有问题，它的功能是正确的。在实现上，因为数值 0~9 和 A~F 在 ASCII 码值上并不连续（分别为 0X30~0X39 和 0X41~0X46），所以程序中以 9 为分界，进行了分情况处理。

但聪明一些的编程者，可能用这样的方法来实现：

```
void Value2String(unsigned char value,char *str)
{
  char Hex_Char_Table[16]={'0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F'};

  str[0]='0';str[1]='X';str[4]=0;
```

```
  str[2]=Hex_Char_Table[value>>4];
  str[3]=Hex_Char_Table[value&0X0F];
}
```

对，这是使用了查表的思想。虽然 0~9 和 A~F，在 ASCII 码值上不连续，但是我们可以把它们放到一个数组里，创造一种连续。然后用数值作为下标，直接获取对应的字符。

也许会有人觉得 Hex_Char_Table 定义起来太麻烦，要一个个去输入字符。其实可以这样作：

```
void Value2String(unsigned char value,char *str)
{
  char *Hex_Char_Table="0123456789ABCDEF";

  str[0]='0';str[1]='X';str[4]=0;

  str[2]=Hex_Char_Table[value>>4];
  str[3]=Hex_Char_Table[value&0X0F];
}
```

我们将字符数组换成了字符串常量。其实它们在内存中的表达是几乎一样的，其实质都是内存中的字节序列。如图 2.1 所示。

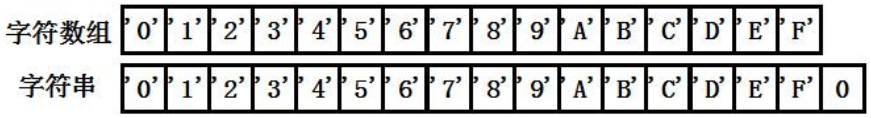


图 2.1 字符数组与字符串都是内存中的字节序列

不同点在于，字符数组在定义的时候要明确指定数组的大小，即它可以容纳多少个字符(字节)。而字符串的长度则以第一个等于 0 的字节为准。所以，字符串的字节序列中，一定有某一个字节的值为 0，它就是字符串的结束符。我们平时使用的 strlen 这个函数，计算字符串长度的原理，其实就是在检测这个 0。所以，如果我们拿一个没有 0 的字符数组(字节序列)传给 strlen，那么最终的结果很可能是错误的，甚至因为数组越界访问，而导致程序的崩溃。

上面，振南说“字符串本身就是指针”，那么见证这句话真正意义的时刻来了，我们将上面程序继续简化：

```
void Value2String(unsigned char value,char *str)
{
  str[0]='0';str[1]='X';str[4]=0;

  str[2]="0123456789ABCDEF"[value>>4];
  str[3]="0123456789ABCDEF"[value&0X0F];
}
```

Hex_Char_Table 这个指针变量其实是多余的，“字符串本身就是指针”，所以它后面可以直接用[]配合下标来取出其中的字符。凡是实质上为指针类型(即表达的是地址意义)的变量或常量，都可以直接用[]或*来访问它所指向的数据序列中的数据元素。

二、转义符

C 语言中要表达一个字节数据序列(内存中连续存储的若干个字节)，我们可以使用字节数组，如 unsigned char array[10]={0,1,2,3,4,5,6,7,8,9}。其实字符串，本质上也是一个字节序列，但是通常情况下它所存储的字节的值均为 ASCII 中可打印字符的码值，如'A'、' '、'|'等。那在字符串中是否也可以出现其它的值呢？这样，我们就可以用字符串的形式来表达一个字节序列了。很多时候，它可能比字节数组要方便一些。字符串中的转义符就是用来干这个的。请看如下程序：

```
const unsigned char array[10]={0,1,2,3,4,5,6,7,8,9};
char *array="\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09";
```

这两种写法，array 所指向的内存字节序列是基本一样的(后者最后还有一个 0)。当然，如果我们把 array 传到 strlen 去计算长度，返回的值为 0。因为它第一个字节的值为 0。但是我们仍然可以使用 array[n]的方式去访问序列中的数据。

```
char *str="ABCDEFG";
char *str="\x41\x42\x43\x44\x45\x46\x47";
```

上面程序中的两种写法，是完成等价的。

字符串中的转义符的目的是为了在本应该只能看到 ASCII 可打印字符的序列中，可以表达其它数值或特殊字符。如经常使用的回车换行"\r\n"，其实质就是"\x0d\x0a"；通常我们所说的字符串结束符\0，其实就是 0 的八进制转义表达形式。

三、字符串常量的连接

在研读一些开源软件的源代码时，我见到了字符串常量的一个比较另类的用法，在这里介绍给大家。

有些时候，为了让字符串常量内容层次更加清晰，就可以把一个长字符串打散成若干个短字符串，它们顺序首尾相接，在意义上与长字符串是等价的。比如"0123456789ABCDEF"可以分解为"0123456789""ABCDEF"，即多个字符串常量可以直接连接，够成长字符串。这种写法，在 printf 打印调试信息的时候可能会更多用到。

```
printf("A:%d B:%d C:%d D:%d E:%d F:%d\r\n",1,2,3,4,5,6);

printf("A:%d " \
       "B:%d " \
       "C:%d " \
       "D:%d " \
       "E:%d " \
       "F:%d\r\n",1,2,3,4,5,6);
```

在 printf 的格式化串很长的时候，我们把它合理的打散，分为多行，程序就会显得更多工整。

四、长字符串的拆分技巧

很多时候我们需要进行长字符串的拆分。在振南的研发经历中，使用到这种操作的最典型的应用场合有三个。

1）NMEA 协议数据的解析

NMEA 可能很多人不太了解，但说到 GPS 肯定大家很熟悉。当我们从 GPS 模块中读取定位信息的时候，数据就是遵循 NMEA 协议格式的。图 2.2 为一个标准的 GPS 数据帧。

```
$GPGGA,121252.000,3937.3032,N,11611.6046,E,1,05,2.0,45.9,M,-5.7,M,,0000*77
$GPRMC,121252.000,A,3958.3032,N,11629.6046,E,15.15,359.95,070306,,,A*54
$GPVTG,359.95,T,,M,15.15,N,28.0,K,A*04
$GPGGA,121253.000,3937.3090,N,11611.6057,E,1,06,1.2,44.6,M,-5.7,M,,0000*72
$GPGSA,A,3,14,15,05,22,18,26,,,,,,,2.1,1.2,1.7*3D
$GPGSV,3,1,10,18,84,067,23,09,67,067,27,22,49,312,28,15,47,231,30*70
$GPGSV,3,2,10,21,32,199,23,14,25,272,24,05,21,140,32,26,14,070,20*7E
$GPGSV,3,3,10,29,07,074,,30,07,163,28*7D
```

图 2.2 一个符合 NMEA 协议标准的 GPS 数据帧

整个数据帧采用 ASCII 编码，它以$GP 作为开始，后面依次排列的是各项参数，参数之间使用,作为分隔。比如$GPRMC 为推荐定位信息，我当时就是使用这一条数据来获取经纬度信息的(当时是 Intel 杯嵌入式邀请赛需要作一个手持 GPS 跟踪器)。这条数据中 N 后面就是纬度，E 后面就是经度。我们要作的就是将它们从整个数据帧(一个长字符串)中提取出来。所以，这就涉及到了所谓的“长串拆分”。

2）后台 Shell 命令行的命令解析

在很多项目中，我都习惯于基于串口编写一个后台 Shell 系统，可以起到一个基本的调试作用。从而一定程度上减少修改代码和固件烧录的次数。比如，项目中如果涉及 DAC 电压经常的调整输出，我就会在后台中设计一个命令 SetV n，以便随时灵活的操控 DAC。随着项目功能的升级，后台命令也会变得开始复杂。比如 SetArg a b c d e f g h....，用于同时设置程序中多个关键参数的值；再比如 SetV channel n freq a，设置某通道第 n 个信号的输出幅值和频率。

这些命令通过 PC 上的串口助手或调试终端来发送，比如超级终端、SecureCRT 或 XShell 等。程序中从串口接收到命令之后，将其放入内存的缓冲区中，其形式就是一个字符串。命令字以及后面的若干参数之间使用空格来分隔。程序要匹配命令字，并提取参数，以便执行相应的操作。所以，这也涉及到长串的拆分。

3）DTU 模块的 AT 指令解析

AT 指令其实和 NMEA 是一个道理，它们都是一种通信协议格式，只不过 AT 指令更多使用在网络通信模块中，比如 SIM800、ESP8266、HC06 蓝牙串口等。举个例子，我们想知道网络信号强度，就可以向模块发送”AT+CSQ\r\n”，模块会返回”+CSQ: 29,0\r\n”。CSQ:后面的 29 就是信号强度。它们都是 ASCII 编码的，也就是一个字符串。我们需要将 29 从其中提取出来。当然，AT 指令也有比较复杂的，字符串会比较长，包含的参数也会比较多。所以，要想使用这些网络模块实现网络通信，就必须实现对 AT 指令的解析。

说了这么多，都是在说长串拆分很重要。根本问题是如何实现它？很多人可能都会想到使用那个分隔字符，比如空格、逗号。然后去一个个数要提取的参数前面有几个分隔字符，然后后将相应位置上的字符组成一个新的短字符串。如图 2.3 所示。

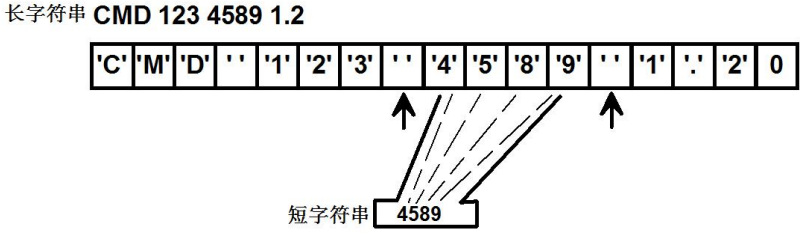


图 2.3 通过分隔字符定位要提取的部分

这种方法固然可行，但是略显笨拙。其实对于这种有明显分隔符的长字符串，我们可以采用“打散”或“爆炸”的思想，具体过程是这样的：将长字符串中的所有分隔符全部替换为'\0'，即字符串结束符。此时，长字符串就被分解成了在内存中顺序存放的若干个短字符串。如果要取出第 n 个短字符串，可以用这个函数：

```
char * substr(char *str,n)
{
  unsigned char len=strlen(str);

  for(;len>0;len--) {if(str[len-1]==' ') str[len-1]=0;}

  for(;n>0;n--)
  {
    str+=(strlen(str)+1);
  }

  return str;
}
```

很多时候我们需要一次性访问长字符串中的多个短字符串，此时振南经常会这样来作：通过一个循环，将长字符串中的所有分隔符替换为'\0'，在此过程中将每一个短字符串首字符的位置记录到一个数组中，代码如下：

```
unsigned char substr(unsigned char *pos,char *str)
{
  unsigned char len=strlen(str);
  unsigned char n=0,i=0;

  for(;i<len;i++) {if(str[i]==' ') {str[i]=0;pos[n++]=(i+1);}}

  return n;
}
```

好，举个例子：我们要提取”abc 1000 50 off 2500”中的”abc”、”50”和”off”，可以使用上面的函数来实现。

```
unsigned char pos[10];
char str[30];

strcpy(str,"abc 1000 50 off 2500");
substr(pos,str);

str+pos[0]; //"abc"
str+pos[2]; //"50"
str+pos[3]; //"off"
```

五、取出数值的各位数码

在实际项目中，我们经常需要提取一个数值的某些位的数码，比如用数码管来显示数值或将一个数值转成字符串，都会涉及到这一操作。

那如何实现这一操作呢？虽然这个问题看似很简单，但提出这一问题的人还不在少数。请看下面的函数。

```
void getdigi(unsigned char *digi,unsigned int num)
{
 digi[0]=(num/10000)%10;
 digi[1]=(num/1000)%10;
 digi[2]=(num/100)%10;
 digi[3]=(num/10)%10;
 digi[4]=num%10;
}
```

它的主要操作就是除法和取余。这个函数只是取出一个整型数各位的数码，那浮点呢？其实一样的道理，请看下面函数(我们默认整数与小数部分均取 4 位)。

```
void getdigi(unsigned char *digi1,unsigned char *digi2,unsigned float num)
{
 unsigned int temp1=num;
 unsigned int temp2=((num-temp1)*10000);

 digi1[0]=(temp1/1000)%10;
 digi1[1]=(temp1/100)%10;
 digi1[2]=(temp1/10)%10;
 digi1[3]=(temp1)%10;

 digi2[0]=(temp2/1000)%10;
 digi2[1]=(temp2/100)%10;
 digi2[2]=(temp2/10)%10;
 digi2[3]=(temp2)%10;
}
```

有人说，我更喜欢用 sprintf 函数，直接将数值格式化打印到字符串里，各位数码自然就得到了。

```
char digi[10];
sprintf(digi,"%d",num); //整型

char digi[10];
sprintf(digi,"%f",num); //浮点
```

没问题。但是在嵌入式平台上使用 sprintf 函数，通常代价是较大的。作为嵌入式工程师，一定要惜字如金，尤其是在硬件资源相对较为紧张的情况下。sprintf 非常强大，我们只是一个简单的提取数值数码或将数值转为相应的字符串的操作，使用它有些暴殄天物。这种时候，我通常选择写一个小函数或者宏来自己实现。

六、printf 的实质与使用技巧

上面说到 spintf，那我们顺便提一下 printf。printf 是我们非常熟悉的一个入门级的标准库函数，每当我们说出计算机金句”Hello World!”时，其实无意中就提到了它：printf(“hello world!”);

它可以某种特定的格式、进制或形式输出任何变量、常量和字符串，为我们提供了极大

的方便，甚至成为了很多人调试程序时重要的 Debug 手段。我们并不太了解 printf 函数的具体实现细节，并认为无需关心这些。但是在嵌入式中，我们就需要剖析一下它的实质了。

printf 函数的底层是基于一个 fputc 的函数，它用于实现单个字符的具体输出方式，比如是将字符显示到显示器上，或是存储到某个数组中(类似 sprintf)，或者是通过串口发送出去，甚至不是串口，而是以太网、CAN、I2C 等接口。

以下是一个 STM32 项目中 fputc 函数的实现：

```
int fputc(int ch, FILE *f)
{
   while((USART1->SR&0X40)==0);
   USART1->DR = (u8) ch;
   return ch;
}
```

fputc 中将 ch 通过 USART1 发出。这样，我们在调用 printf 的时候，相应的信息就会从 USART1 打印出来。

“上面你说的这些，我都知道，有什么新鲜的！”确实，通过串口打印信息是我们司空见惯的。那么下面的 fputc 你见过吗？

```
int fputc(int ch, FILE *f)
{
 LCD_DispChar(x,y,ch);
 x++;
 if(x>=X_MAX)
 {
  x=0;y++;
  if(y>=Y_MAX)
  {
   y=0;
  }
 }

 return ch;
}
```

这个 fputc 将字符显示在了液晶上(同时维护了字符的显示位置信息)，这样当我们调用 printf 的时候，信息会直接显示在液晶上。

说白了，fputc 就是对数据进行了定向输出。这样我们可以把 printf 变得更灵活，来应对更多样应用需求。

在振南经历的项目中，曾经有过这样的情况：单片机有多个串口，串口 1 用于打印调试信息，串口 2 与 ESP8266 WIFI 模块通信，串口 3 与 SIM800 GPRS 模块通信。3 个串口都需要格式化输出，但是 printf 只有一个，这该怎么办？我们解决方法是，修改 fputc 使得 printf 可以由 3 个串口分时复用。具体实现如下。

```
unsigned char us=0;

int fputc(int ch,FILE *f)
{
 switch(us)
```

```
 {
  case 0:
          while((USART1->SR&0X40)==0);USART1->DR=(u8)ch;    break;
  case 1:
          while((USART2->SR&0X40)==0);USART2->DR=(u8)ch;    break;
  case 2:
          while((USART3->SR&0X40)==0);USART3->DR=(u8)ch;    break;
 }

 return ch;
}
```

在调用的时候，根据需要将 us 赋以不同的值，printf 就归谁所用了。

```
#define U_TO_DEBUG        us=0;
#define U_TO_ESP8266      us=1;
#define U_TO_SIM800       us=2;

U_TO_DEBUG
printf("hello world!");

U_TO_ESP8266
printf("AT\r\n");

U_TO_SIM800
printf("AT\r\n");
```

七、关于浮点数的传输

很多人不能很好的使用和处理浮点，其主要根源在于对它的表达与存储方式不是很理解。最典型的例子就是经常有人问我：“如何使用串口来发送一个浮点数？”

我们知道 C 语言中有很多数据类型，其中 unsigned char、unsigned short、unsigned int、unsigned long 我们称其为整型，顾名思义它们可以表达整型数。而能够表达的数值范围与数据类型所占用的字节数有关。数值的表达方法比如简单，如下图所示。

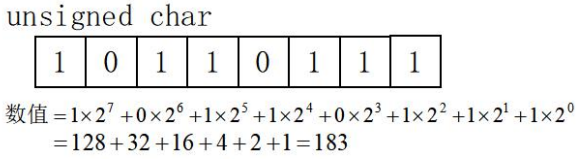


图 2.4 整型变量数值的计算方法

一个字节可以表达 0~255，两个字节(unsigned short)自然就可以表达 0~65535，依次类推。

当需要把一个整型数值发送出去的时候，我们可以这样作：

```
unsigned short a=0X1234;

UART_Send_Byte(((unsigned char *)&a)[0]);
UART_Send_Byte(((unsigned char *)&a)[1]);
```

也就是将构成整型的若干字节顺序发送即可。当然接收方一定要知道如何还原数据，也就是说它要知道自己接收到的若干字节拼在一起是什么类型，这是由具体通信协议来保障的。

```
unsigned char buf[2];
usnigned short a;

UART_Receive_Byte(buf+0);
UART_Receive_Byte(buf+1);

a=(*(usnigned short *)buf);
```

OK，关于整型比较容易理解。但是换成 float，很多人就有些迷糊了。因为 float 的数值表达方式有些复杂。有些人使用下面的方法来进行浮点的发送。

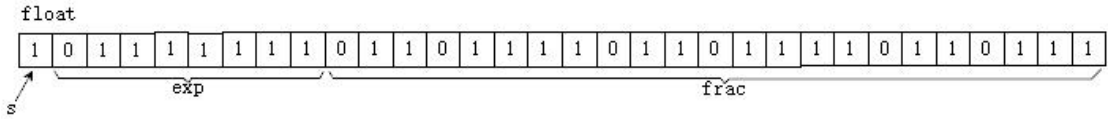
```
float a=3.14;
char str[10]={0};

ftoa(str,a); //浮点转为字符串 即 3.14 转为"3.14"

UART_Send_Str(str); //通过串口将字符串发出
```

很显然这种方法非常的“业余”。还有人问我：“浮点小数字前后的数字可以发送，但是小数点怎么发？”这赤裸裸的体现了他对浮点类型的误解。

不要被 float 数值的表象迷惑，它实质上只不过是 4 个字节而已，如下图所示。



$$数值 = (-1)^s \times 1.frac \times 2^{exp-127}$$
$$= -1 \times 1.01101111011011110110111 \times 2^0$$
$$= -1.4352940839879613$$

图 2.5 浮点变量数值的计算方法

所以，正确的发送浮点数的方法是这样的：

```
float a=3.14;

UART_Send_Byte(((unsigned char *)&a)[0]);
UART_Send_Byte(((unsigned char *)&a)[1]);
UART_Send_Byte(((unsigned char *)&a)[2]);
UART_Send_Byte(((unsigned char *)&a)[3]);
```

接收者将数据还原为浮点：

```
unsigned char buf[4];
float a;

UART_Receive_Byte(buf+0);
UART_Receive_Byte(buf+1);
```

```
UART_Receive_Byte(buf+2);
UART_Receive_Byte(buf+3);

a=*((float *)buf);
```

其实我们应该发现数据类型的实质：不论是什么数据类型，它的基本组成无非就是内存中存储的若干个字节。只是我们人为的赋予了这些字节特定的编码方式或数值表达。看穿了这些，我们认识到了数据的本质了，我们甚至可以直接操作数据。

八、关于数据的直接操作

直接操作数据？我们来举个例子：取一个整型数的相反数。一般的实现方法是这样的：

```
int a=10;
int b=-a; //-1*a;
```

这样的操作可能会涉及到一次乘法运算，花费更多的时间。当我们了解了整型数的实质，就可以这样来作：

```
int a=10;
int b=(~a)+1;
```

这也许还不足以说明问题，那我们再来看一个例子：取一个浮点数的相反数。似乎只能这样来作：

```
float a=3.14;
float b=a*-1.0;
```

其实我们可以这样来作：

```
float a=3.14;
float b;

((unsigned char *)&a)[3]^=0X80;
b=a;
```

没错，我们可以直接修改浮点在内存中的高字节的符号位。这比乘以-1.0 的方法要高效的多。

当然，这些操作都需要你对 C 语言中的指针有炉火纯青的掌握。

九、浮点的四舍五入与比较

我们先说第一个问题：如何实现浮点的四舍五入？很多人遇到过这个问题，其实很简单，只需要把浮点+0.5 然后取整即可。

OK，第二个问题：浮点的比较。这个问题还有必要好好说一下。首先我们要知道，C 语言中的判等，即==，是一中强匹配的行为。也就是，比较双方必须每一个位都完全一样，才认定它们相等。这对于整型来说，是可以的。但是 float 类型则不适用，因为两个看似相等的浮点数，其实它们的内存表达不能保证每一个位都完全一样。

这个时候，我们作一个约定：两个浮点只要它们之差 m 足够小，则认为它们相等，m 一般取 10e-6。也就是说，只要两个浮点小数点后 6 位相同，则认为它们相等。也正是因为

这个约定，很多 C 编译器把 float 的精度设定为小数点后 7 位，比如 Keil。

```
float a,b;

if(a==b) ...   //错误

if(fabs(a-b)<=0.000001) ...//正确
```

十、出神入化的 for 循环

for 循环我们再熟悉不过了，通常我们使用它都是中规中矩的，如下例：

```
int i;
for(i=0;i<100;i++)
{...}
```

但是如果我们对 for 循环的本质有更深刻的理解的话，就可以把它用得出神入化。

for 后面的括号中的东西我称之为“循环控制体”，分为三个部分，如下图所示。

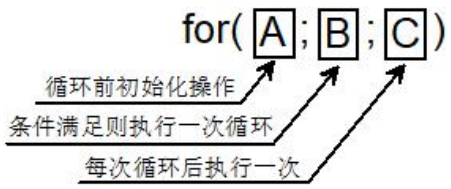


图 2.6 for 循环的“循环控制体”

A、B、C 三个部分，其实随意性很大，可以是任意一个表达式。所以，我们可以这样写一个死循环：

```
for(1;1;1) //1 本身就是一个表达式：常量表达式
{
  ...
}
```

当然，我们经常会把它简化成：

```
for(;;) //
{
  ...
}
```

既然循环控制体中的 A 只是在循环开始前作一个初始化的操作，那我这样写应该也没毛病：

```
int i=0;

for(printf("Number:\r\n");i<10;i++)
{
  printf("    %d\r\n",i);
}
```

B 是循环执行的条件，而 C 是循环执行后的操作，那我们就可以把一个标准的 if 语句写成 for 的形式，而实现同样的功能：

```
if(strstr("hello world!","abc"))
{
  printf("Find Sub-string");
```

```
}

 char *p;

 for(p=strstr("hello world!","abc");p;p=NULL)
 {
  printf("Find Sub-string");
 }
```

以上的例子可能有些鸡肋，“一个 if 能搞定的事情，我为什么要用 for？”，没错。我们这里主要是为了解释 for 循环的灵活用法。深入理解了它的本质，有助于我们在实际开发中让工作事半功倍，以及看懂别人的代码。

以下我再列举几个 for 循环灵活应用的例子，供大家回味。

例 1：

```
 char *p;

 for(p="abcdefghijklmnopqrstuvwxyz";printf(p);p++) printf("\r\n");
```

提示：printf 我们太熟悉了，但是有几个人知道 printf 是有返回值的？输出应该是怎样的？

例 2：

```
 char *p;
 unsigned char n;

 for(p="ablmnl45ln",n=0;((*p=='l')?(n++):0),*p;p++);
```

提示：还记得 C 语言中的三目运算和逗号表达式吗？n 应该等于几？

例 3：

```
    unsigned char *index="C[XMZA[C[NK[RDEX@";
    char *alphabet="EHUIRZWXABYPOMQCTGSJDFKLNV ";
    int i=0;

    for(;(('@'!=index[i])?1:(printf("!!Onz\r\n"),0));i++)
      {printf("%c",alphabet[index[i]-'A']);}
```

提示：天书模式已开启。如果看不懂，你可能会错过什么哦！

十一、隐藏的死循环

有些时候我们会发现 for 循环变成了一个死循环：

```
 unsigned char i;

 for(i=4;i>=0;i--) ....
```

我们本希望循环 5 次，然后结束，但是实际情况是陷入了死循环。这种错误在实际开发中，还比较难发现。其原因在于 i 的类型，无符号整型是永远不小于 0 的。我们需要将 i 的类型改为有符号型。

```
signed char i;

for(i=4;i>=0;i--) ....
```

OK，这样就对了。细节虽小，但是对实际开发的影响还是蛮大的，请大家引以为戒。

下面的两个例子中 for 循环也是死循环，请自行分析：

例 1：

```
unsigned char i;

for(i=0;i<256;i++) ...
```

提示：i 的数据类型。

例 2：

```
char str[20];
char *p;
unsigned char n=0;

for(p=strcpy(str,"                abcd");((*p)=' ');p++,n++);
```

提示：这个例子，不光会死循环，而且还可能会让程序直接崩溃。判等的==你会不会经常直接写错成=（赋值表达式）。

十三、看似多余的空循环

有时我们会看到这样的代码：

```
do
{
 ..... //do something
}while(0);
```

代码本身实际只运行了一次，为什么要在它外面加一层 do while 呢？这看似是多余的。其实不然，我们来看下面例子：

```
#define DO_SOMETHING fun1();fun2();

void main(void)
{
 while(1) DO_SOMETHING;
}
```

while(1) DO_SOMETHING;本意应该是不断调用 fun1 和 fun2，但实际上只有 fun1 得到运行。其中原因大家应该明白。所以，我们可以这样来写：

```
#define DO_SOMETHING do{ fun1();fun2();}while(0);
```

do while 就如同一个框架把要运行的代码框起来，成为一个整体。

十四、独立执行体

我在 C 语言编程的过程中，经常乐于使用一种“局部独立化”的方式，我称之为“独立执行体”，如下例：

```
void fun(int a,int b,int c)
{
 Int tmp=0;
```

```
 //主体计算

 {                    //独立执行体，解决临时性问题
  int c=0;
  c=(a>b)?a:b;
  printf("max:%d\r\n",c);
 }

 {                    //独立执行体
  int c=0,d=0,.....,res=0.;
  //数据处理算法
  printf("result:%d\r\n",res);
 }

 //进一步计算
}
```

编程时，我们经常需要解决一些小问题，比如想对一些数据进行临时性的处理，查看中间结果；或是临时性的突发奇想，试探性的作一些小算法。这过程中可能需要独立的变量，以及独立于主体程序的执行逻辑，但又觉得不至于去专门定义一个函数，只是想一带而过。比如上例，函数 fun 主要对 a、b、c 这 3 个参数进行计算(使用某种算法)，过程中想临时看一下 a 和 b 谁比较大，由第一个“独立执行体”来完成，其中的代码由自己的{}扩起来。

其实我们可以更深层的去理解 C 语言中的{}，它为我们开辟了一个可自由编程的独立空间。在{}里，可以定义变量，可以调用函数以及访问外层代码中的变量，可以作宏定义等等。平时我们使用的函数，它的{}部分其实就是一个“独立执行体”。

“独立执行体”的思想，也许可以让我们编程更加灵活方便，可以随时让我们直接得到一块自由编程的静土。

上一节中的 do while(0)，其实完全可以把 do while(0)去掉，只用{}即可：

```
#define DO_SOMETHING {fun1();fun2();}
```

其中它还有一个好处，就是当你不需要这段代码的时候，你可以直接在{}前面加上 if(0)即可。一个“独立执行体”的外层是可以受 if、do while、while、for 等这些条件控制的。

十五、多用()无坏处

!0+1，它的值等于多少？其实连我这样的老手也不能马上给出答案，2 还是 0？按 C 语言规定的运算符优先级来说，应该是!大于+，所以结果应该是 2。

但是如果把它放在宏里，有时候就开始坑人了：

```
#define VALUE !0+1

int a;
a=VALUE&0;
```

踩过此类坑的人无需多说，自能领会。(a=2&0 呢，还是 a=!0+1&0 呢？它们的值截然不同)。

这里出现了一些运算优先级和结合律的差错。为了让我们的语义和意图正确的得以表

达，所以建议多用一些()。

```
#define VALUE ((!0)+1)

int a;
a=VALUE&0;
```

这样，a 的值就一定是 0 了。

另外，有时候优先级还与 C 语言编译器有关，同一个表达式在不同的平台上，可能表达的意义是不同的。所以，为了代码的可植移性、正确性以及可读性，振南强烈建议多用一些()。

十六、==的反向测试

C 语言中的=与==，有时候是一个大坑。主要体现在条件判断时的值比较，如下例：

```
int a=0;

If(a=1)
{
 //代码
}
```

也许我们的原意是判断 a 若为 1，则执行。但实际上 if 根本不起作用，因为错把==写成了=。

C 语言中的赋值操作也是一种表达式，称为赋值表达式，它的值即为赋值后变量的值。而 C 语言中条件判断又是一种宽泛的判断，即非 0 为真，为 0 则假。所以 if(a=1)这样的代码编译是不会报错的。

这种错误通常是很难排查出来的，尤其是在复杂的算法中，只能一行行代码的跟踪。所以对于变量值的比较判断，振南建议使用“==的反向测试”，并养成习惯。

```
int a=0;

if(1==a)
{
 //代码
}
```

如果把==错写成了=，因为常量无法被赋值，所以编译时会报错。

十七、赋值操作的实质

原来一位哈工程理学院教授(搞数学的)讲述的自己的一个困惑，一直以来都被我们当成一个笑谈在说。他学 C 语言的时候，首先 a=1，然后后面又来一个 a=2，这让他非常不解，a 怎么可能同样等于 1 又等于 2 呢？

其实这是因为他对计算机运行机制不了解，这个 a 不是他数学稿纸上的代数变量，而是计算机中实实在在的“电”，或者说“信号”。

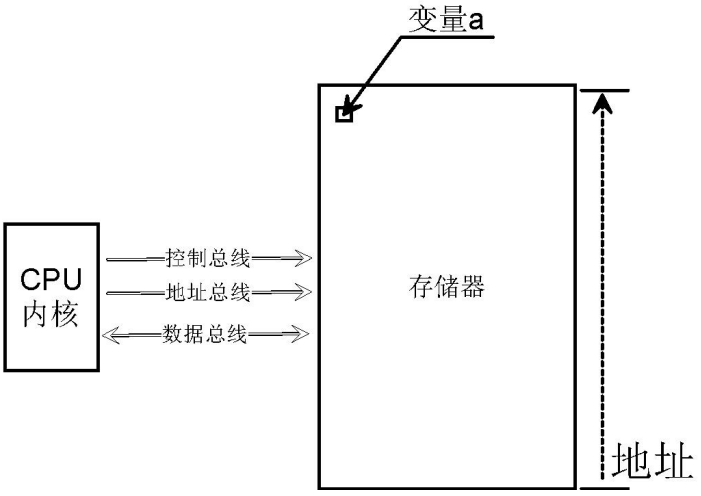


图 2.7 计算机中 CPU 与存储器的寻址与数据传输

其实不限于 C 语言，所有计算机语言的目的都是控制计算机硬件，实现电信号的传输、存储等操作，最终达成某一功能。变量是存储器中的一小块空间，它源自于形如 int a 这样的代码编译后由编译器所作的存储器分配。对变量的赋值就是 CPU 内核通过三总线将数据传输到存储器特定地址单元上的过程。所以，a=1;a=2;只是两次数据传输过程而已。

这个教授当时算是个外行，其实对于我们也是一样的，想要真正掌握编程语言，只流于代码表面的意思是不行的，必须对它在硬件上产生的操作有明确的认识，对计算机内部的运行机理有深入理解才可以。

十八、关于补码

补码是一个很基础的概念，但是对于很多人来说，其实有些迷糊，这里对补码进行一些通俗而深刻的讲解。

C 语言中的整型类型有两种，无符号与有符号。无符号比较好理解：

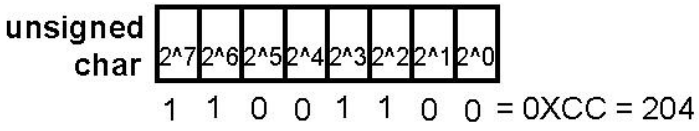


图 2.8 无符号整型的数值表达

只需要将每一个位乘以它的权值，再求和即是其所表达的数值。它所有的位都用来表达数值，因此上图中类型能表达的范围为 0~255(8 个位)。但是如何表达负数，比如-10，这个时候就涉及到负数的补码了。

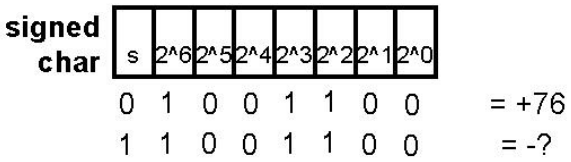


图 2.9 有符号整型的数值表达

有符号整型的最高位被定义为符号位，0 为正数，1 为负数。上图中前一行等于+76，后一行等于多少？-76？那就错了。对于负数的数值要按其补码来计算：

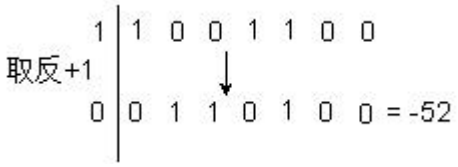


图 2.10 有符号整型负数数值计算方法

为什么要引入补码的概念，符号位表示符号，其它位直接表示其绝对值，不是更好吗？这其实是一个数字游戏。我们要知道一个前提：CPU 中只有加法器，而没有减法器。OK，我们看下面的例子。

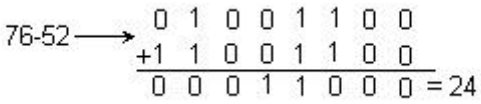


图 2.11 使用补码通过加法实现减法操作

可以看到，补码将符号位也统一到了计算过程中，并且巧妙的使用加法实现了减法操作。这对于简化 CPU 中的算术逻辑电路(ALU)具有重要意义。

十九、关于-1

为了说明关于-1 的问题，我们先来看一个例子：

```
signed short a=-1;

if(-1==a)
{
  //....
}
```

这个 if 条件成立吗？似乎这是一句废话。其实不然，它不一定成立。

我们要知道 C 语言中的判等==运算是一种强匹配，也就是比较的双方必须每一个位都匹配才被认为相等。上例中，a 在内存中的表示是 0XFFFF(补码)，但是-1 这个常量在内存中的表示在不同的硬件平台上却不尽相同，在 16 位 CPU 平台上是 0XFFFF，它们是相等的。而在 32 位 CPU 平台上则是 0XFFFFFFFF，它们就不相等。所以稳妥的办法是：

```
signed short a=-1;

if(((signed short)-1)==a)
{
  //....
}
```

我们看到-1 的补码是全 F，位数而且与 CPU 平台相关。所以-1 经常还有另一个妙用，即可以用于判断硬件平台的 CPU 位数，便于提高代码的可移植性(32 位平台的 int(-1)为 0XFFFFFFFF，而 16 位平台则是 0XFFFF)。

二十、字节位逆序

我给大家出一道有意思的题目：如何快速得到一个字节的位逆序字节。比如 0X33 的位逆序字节是 0XCC。

有人给了我这样一段代码：

```
unsigned char reverse_byte(unsigned char byte)
{
 unsigned char i=0;
 unsigned char temp=0;

 for(i=0;i<8;i++)
 {
  if(byte&(0x01<<i)) temp|=(0x80>>i);
 }

 return temp;
}
```

这段代码很简洁，也很巧妙。但是它却不是最快的。后来作了改进：

```
unsigned char reverse_byte(unsigned char byte)
{
 unsigned char temp=0;

 if(byte&0x01) temp|=0x80;
 if(byte&0x02) temp|=0x40;
 if(byte&0x04) temp|=0x20;
 if(byte&0x08) temp|=0x10;
 if(byte&0x10) temp|=0x08;
 if(byte&0x20) temp|=0x04;
 if(byte&0x40) temp|=0x02;
 if(byte&0x80) temp|=0x01;

 return temp;
}
```

这样把循环打开，确实会提速不少。但它仍不是最快的实现方案。请看如下代码：

```
unsigned char rbyte[256]={0x00,0x80,0x40,0xc0,0x20,.....};

#define REVERSE_BYTE(x) rbyte[x]
```

恍然大悟了没有？使用字节数组事先准备好位逆序字节，然后直接以字节的值为下标索引，直接取数据即可。这种方法的名字叫“空间换时间”。

这个问题我问过很多人，多数人并不能直接给出最佳方案。倒是有不少人问我这个问题有什么实际意义，为什么要去计算位逆序字节？请大家想想，如果我们把电路上的数据总线焊反或插反了该怎么解决。

二十一、关于 volatile

现在的编译器越来越智能，它们会对我们的代码进行不同程度的优化。请看下例：

```
unsigned char a;

a=1;
a=2;
a=3;
```

这样一段代码，有些编译器会认为 a=1 与 a=2 根本就是毫无意义，会把它们优化掉，只剩下 a=3。但是，有些时候这段代码是有特殊用途的：

```
unsigned char xdata a _at_ 0X1111;

a=1;
a=2;
a=3;
```

a 不单单是一个变量，而是一个外部总线的端口(51 平台)。向它赋值会产生相应的外部总线上的时序输出，从而对外部器件实现控制。这种时候，a=1 和 a=2 不能被优化掉。举个例子：a 所指向的外部总线端口，是一个电机控制器的接口，向它写入 1 是加速，写入 2 是减速，写入 3 是反向。那么上面的代码就是加速->减速->反向，这样一个控制过程。如果被优化的话，那最后就只有反向了。

为了防止这种被“意外”伦的情况发生，我们可以在变量的定义上加一个修饰词 volatile。

```
volatile unsigned char xdata a _at_ 0X1111;

a=1;
a=2;
a=3;
```

这样，编译器就会对它单独对待，不再优化了。

volatile 最常出现的地方，就是对芯片中寄存器的定义，比如 STM32 固件库中有这样的代码：

```
#define __IO volatile
typedef struct
{
  __IO uint32_t CRL;
  __IO uint32_t CRH;
  __IO uint32_t IDR;
  __IO uint32_t ODR;
  __IO uint32_t BSRR;
  __IO uint32_t BRR;
  __IO uint32_t LCKR;
} GPIO_TypeDef;
```

这是对 STM32 的 GPIO 寄存器组的定义，每一项都是一个__IO 类型，其实就是 volatile。

这样是为了对片内外设的物理寄存器的访问一定是真正落实的，而不是经过编译器优化，而变成去访问缓存之类的东西。

二十二、关于变量互换

初学 C 语言的时候，有一个小编程题我们应该都记得，就是变量互换。

```
int a,b;
int temp;

temp=a;
a=b;
b=temp;
```

变量 a 与 b 的值互换，在这过程中一定需要一个中间变量 temp 作为中转。不用这个中间变量能不能实现？请看下面的代码：

```
int a,b;

a=a+b;
b=a-b;
a=a-b;
```

可以说上面代码有点小巧妙，那么下面的代码就真正是巧妙了：

```
int a,b;

a=a^b;
b=a^b;
a=a^b;
```

异或运算有一个性质叫自反性，这个可以实现很多巧妙的操作，大家可以深入研究一下。(异或位运算比上面的加减法更严谨，因为加减法是可能会溢出的)

二十三、关于 sizeof

C 语言中的 sizeof 我们应该是非常熟悉的，它的作用就是用来计算一个变量或类型所占用的字节数。

```
sizeof(int) //如果是 32 位 CPU 平台，值为 4，即 4 个字节
int a; sizeof(a) //同上
sizeof(struct ...) //计算某结构体的大小
```

这个很简单，我们再来看下面的代码：

```
char *pc="abc";
sizeof(pc) //指针的 sizeof 等于多少？
sizeof(*pc) //指针指向的单元的 sizeof 等于多少？
```

pc 用来指向 char 类型的变量。pc 本身是一个指针类型，在 32 位平台上 sizeof(pc)的值为 4，即指针类型占用 4 个字节(与 CPU 平台有关)。*pc 是 pc 所指向的变量，所以 sizeof(*pc)

的值为 1。

好，还能理解吧，那我们再来看：

```
char a1[]="abcd";
sizeof(a1) //数组的 sizeof 等于多少？

void fun(char a1[]) //形参 a1 的 sizeof 等于多少？
{
  //....
}
```

第一个 sizeof(a1)等于 5，因为它是一个数组（最后还有一个字符串结束符'\0'）。第二个 sizeof(a1)等于 4，形参中的 a1 不再是一个数组，而是一个指针。

好，下面的实例估计很多人没见到过：

```
struct {} a,b,c;
sizeof(a) //空结构体的 sizeof 等于多少?
```

空结构体类型变量的大小是多少？这个问题似乎有些奇葩，没什么实用性。空结构体有什么用？

这个问题可以揭示一些比较深层的问题，我们平时注意不到。空结构体的大小是 1，即占用 1 个字节。当我们的程序还仅仅是一个框架的时候，一些结构体还只是一个空壳，只是拿一个 struct 的定义在那占位置而已，此时就涉及到空结构体问题了。通常编译器会给空结构体分配 1 个字节的内存空间。为什么？如果不分配空间，那程序中的多个同类型结构体变量如何区分呢？比如 a、b、c 这三个变量，它们必须要被分配到不同的地址上去，各占 1 个字节的空间。

另外，因为 sizeof 有一个()，所以很多人想当然的把它当成一个函数。但其实它表达的是一个常数(运算符)，它的值在程序编译期间就确定了。比如 sizeof(i++)，其中 i 为 int 类型，那么它的值就是 4(32 位平台)。

二十四、memcpy 的效率

memcpy 函数的功能是用来作内存搬运，就是将数据从一个数组赋值到另一个数组。它的实现很简单：

```
void memcpy(unsigned char *pd,const unsigned char *ps,unsigned int len)
{
  unsigned int i=0;

  for(i=0;i<len;i++) pd[i]=ps[i];
}
```

但是这种实现方式，其实是比较肤浅而低效的。作为嵌入式或硬件工程师，如果对上面的代码看不出什么问题的话，那可能要好好找找自身的原因。

上面的代码，对 CPU 数据总线带宽的利用率不高，我们把它改成这样：

```
void memcpy(unsigned char *pd,const unsigned char *ps,unsigned int len)
{
  unsigned int i=0;
```

```
    unsigned int temp=len/sizeof(unsigned int);

    for(i=0;i<temp;i++) ((unsigned int *)pd)[i]=((unsigned int *)ps)[i];
    i*=sizeof(unsigned int);
    for(;i<len;i++) pd[i]=ps[i];
}
```

改进后的代码最大限度的利用了 CPU 数据总线带宽，每次传输多个字节(如 32 位平台为 4 字节)。这一实例告诉我们：C 语言程序很多时候需要考虑硬件层面的因素，如 CPU 总线、内存结构等。

二十五、[]的本质

当我们想取出一个数组中的某个元素时，我们会用到[]，采用下标的方式。如下例：

```
int a[3]={1,2,3};
a[1]; //数组 a 的第 2 个元素
```

其实我们可以用其它方式取出这个元素，即*(a+1)。可以看到[]与*，在功能上有相似之处。其实[]并不限于与数组搭配访问数组元素，它的实质是：访问以指针所指向的地址为开始地址，以其下标为偏移量的存储单元中的数据，如下图。

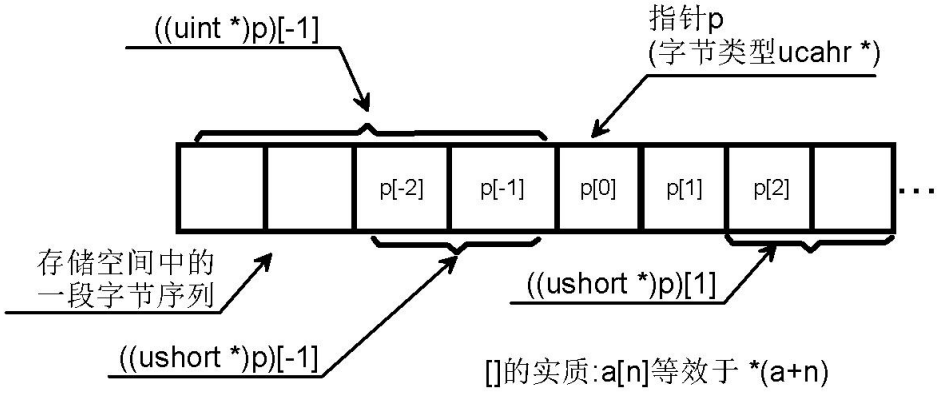


图 2.12 []的实质其实就是所谓的“基址偏移量取值”

上图可能颠覆了一些人对[]的认识，下标还能是负数？[]可以在一个开始地址后面去取数据，为什么不能在它前面取数据呢？我们可以理解[]是对指针加减和取值操作的综合。

认清了[]的实质，再加上对 C 语言的精髓--指针深刻的理解，我们编程将会非常灵活，肆意挥洒。

二十六、#与##(串化与连接)

C 语言中的#与##可能很多人都不了解，更没有用过，因为在一般的教材上都没有对它们的介绍。但是把它们用好了，也能使我们的代码别有一番格调。