# 目录

# 第 7 章 循环程序设计

在实际问题的处理程序中，常常需要按照一定规律，多次重复执行一串语句，这类程序叫做循环程序。在前面章节的例子中也多次出现过循环程序。编写循环程序时，可以对照 C 语言程序中的循环结构，按照执行的流程写出对应的汇编语句。本章将进一步深入介绍循环程序的结构和控制方法、单重循环程序的设计和多重循环程序的设计。

## 7．1 循环程序的结构

### 7.1.1 循环程序的基本结构

循环程序一般由四部分组成。

（1） 置循环初值部分

为了保证程序能正常进行循环操作而必须做的初始化工作。循环初值分两类，一类是循环工作部分的初值，另一类是控制循环结束条件的初值。它们是在循环之外，只执行一次。

（2） 工作部分

需要重复执行的程序段。这是循环程序的核心，称之为循环体。

（3） 修改部分

按一定规律修改操作数地址及控制变量，以便每次执行循环体时得到新的数据。

（4） 控制部分

用来保证循环程序按规定的次数或特定条件正常循环。

循环程序的常见结构形式如图 7.1 所示。其中的工作部分与修改部分有时相互包含、相互交叉，不一定能明显分开。图 7.1（a）的结构形式是先工作后进行控制判断，因此，工作部分至少被执行一次。图 7.1（b）的结构形式是先进行控制判断后工作，因此，工作部分可能不被执行。

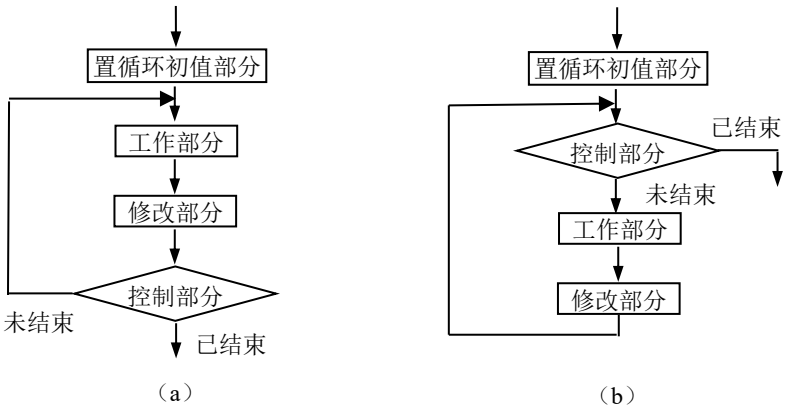


图 7.1 循环的两种结构

在 C 语言程序设计中，do…while 语句属于图 7.1（a）所示的结构，while 语句属于图 7.1（b）所示的结构， for 语句的本质也是图 7.1（b）所示的结构。从机器语言的角度来看，它

们的核心都是根据条件是否成立，进行转移，采用分支转移指令而在不引入新指令的情况下，是完全可以实现程序循环的。

### 7.1.2 循环控制方法

循环控制是循环程序中一个重要环节。最常见的控制方法有计数控制和条件控制，下面将分别介绍。

**1、计数控制**

当循环次数已知时，通常使用计数控制方法。假设循环次数为 n，常常用以下两种方法实现计数控制。

（1）先将循环次数 N 送入循环计数器中，然后，每循环一次，计数器减 1，直至循环计数器中的内容为 0 时结束循环。基本结构如下。

```
        mov   ecx，n          } 置循环初值部分
              ┆
pwork:        ……              工作部分
              ┆               修改部分
        dec   ecx             } 控制部分
        jnz   pwork
```

其中，工作部分、修改部分被重复执行 N 次，即当（ecx）=n，n-1，…，1 时重复执行，当（ecx）=0 时结束循环。循环次数 n 应被视为一个无符号数，应该大于 0。

（2）先将 0 送入循环计数器中，然后每循环一次，计数器加 1，直至循环计数器的内容与循环次数 N 相等时退出循环。基本结构如下。

```
        mov   ecx，0          } 置循环初值部分
              ┆
pwork:        ……              ；工作部分
              ┆               ；修改部分
        inc   ecx             }
        cmp   ecx，n          } 控制部分
        jne   pwork           }
```

其中，工作部分、修改部分被重复执行 N 次，即当（ecx）=0，1，…，n-1 时重复执行，当（ecx）=n 时结束循环。

上述两种计数方法的共同特点是每循环一次之后，在计数器中计数一次。它们的区别在于，第（1）种方法每计数一次之后，计数器的内容减 1；而第（2）种方法每计数一次之后，计数器的内容增 1。通常，称第（1）种方法为倒计数，称第（2）种方法为正计数。它们均选用了寄存器 ecx 作循环计数器。实用中，可根据寄存器的分配情况，选用任一通用寄存器或存储单元作为循环次数计数器。

**2、条件控制**

在循环程序中，经常出现循环次数不固定，但与某些条件是否成立有关的情况。这些条件可以通过指令来测试，若测试比较的结果满足循环条件则继续循环，否则结束循环。

例如，将以某个地址开头的一个字符串拷贝到另一个区域，字符串以字节 0 结束。在重复的拷贝过程中，若读到的字节内容为 0，则循环结束。

在 C 语言程序中，条件可以简单的一个关系表达式，也可以是多个表达式的逻辑组合。在汇编语言程序设计中，从机器语言的角度来看，复杂的条件要拆解成多条机器指令。当然，循环结构程序的写法上可以套用分支转移程序中的条件控制方法。

编写程序可以是很灵活的，同一个问题可以用不同的方法解决。下面给出了一个例子，分别用固定循环次数和条件控制方法来完成同一任务。

**【例 7.1】** 统计(ax) 二进制编码中 1 的个数→cl。

下面介绍两种实现方法。

方法 1 的基本思想：依次判断(ax)中的各个二进制位上的数是 0 还是 1，若是 1 则将(cl)增 1。因此，可以固定循环 16 次，每次将(ax)左移 1 位，判断移到标志位 CF 中的值是否为 1，是 1 则将(cl)加 1，否则(cl)不变。用(bx)来控制循环次数。程序片段如下。

```
        mov   cl,  0
        mov   bx, 16
p:      sal   ax , 1
        jnc   next
        inc   cl
next:   dec   bx
        jnz   p
```

在上面的程序中，“sal ax，1”换成“shl ax，1”、“shr ax，1”、“rol ax，1”、“ror ax，1”都是正确的。当然，还可以设置(dx)=1,由“test ax，dx”的结果是否为 0，确定(ax)的最后一个二进制位是否为 1；之后，将(dx)左移 1 位，重复“test ax，dx”及之后的判断，进而统计(ax)中 1 出现的次数。

方法 2 的基本思想：在上面的方法中，当(ax)=0 时一定有(cl)=0，此时不需要用移位指令，就知道(ax)中 1 的个数为 0；在对(ax)不断左移 1 位的过程中，由于每次 (ax) 左移 1 位后，它的最右二进制位补 0，在经过一定次数的移动后，也许不需要移动 16 次，(ax)=0,从而也不再需要重复移位。程序片段如下。

```
        mov   cl，0
p:      and   ax，ax
        jz    exit          ；(ax）=0 时，结束循环转 exit
        sal   ax，1         ；将 ax 中的最高位移入 CF 中
        jnc   p             ；如果 CF=0，转 p
        inc   cl            ；如果 CF=1，则（cl）+1→cl
        jmp   p             ；转 p 处继续循环
exit:   ……
```

以上介绍的两种循环控制方法是最常用的方法。在解决实际问题时，究竟应该选用哪种方式，往往要根据问题给定的已知条件，在认真分析算法之后才能确定。

## 7.1.3 循环控制指令

除了使用分支转移指令外，X86 汇编语言为循环控制还提供了四条指令。虽然用前面的方法完全可以使用循环控制，但新指令的应用可提高程序执行效率，也可以简化程序的编写。

（1） 一般循环转移指令

使用格式：loop 标号

功能：(ecx/cx) -1→ecx/cx，若（ecx/cx）不为零，则转标号处执行。

基本等价于：　dec　ecx

　　jnz　标号

但是它们之间有细微的差别。直接采用指令“dec ecx”是会影响标志位的，在循环刚结束时，标志位是指令“dec ecx”所设置的。而 loop 指令不影响标志位。当然，在很多情况下，在循环结束后并不需要对标志位的值做判断处理，此时，上述两种写法可认为“等价”。

注意，在 32 位段程序中，使用的是 32 位的寄存器 ecx。在 16 位程序段中，使用的是 16 位的寄存器 cx。

（2）等于或为零循环转移指令

使用格式：loope/loopz　标号

功能：　（ecx/cx）-1→ecx/cx，如果（ecx/cx）不等于零并且 ZF 等于 1，则转标号处执行，否则顺序执行。

与 loop 指令一样，loope 也不影响标志位，ZF 在 loope 执行之前和执行之后保持不变。loope 也是用 ecx 来控制循环次数，当(ecx)=0 时，不再循环，但是，当(ecx)不为 0 时，可以因为某种条件满足而终止循环。

**【例 7.2】** 判断以 buf 为首址的 n 个字节中是否有非 0 字节。若有非 0 字节，则置 x 为 1，否则 x 置为 0。

基本思想：逐个元素判断是否为 0，最多循环 n 次，但在找到非 0 字节时，结束循环。用 ecx 来控制循环次数，用 ebx 来作为数组元素的下标。在循环结束后，一定有(ecx)=0 或者 ZF=0，当然这也就不排除(ecx)=0 且 ZF=0 的情况。换句话说，若循环结束时，ZF=0，并不能确定(ecx)是否为 0；若 ZF=1，则一定有(ecx)=0。

程序核心片段如下：

```
    buf   db   6 dup(0), 20, 10 dup(0)
    n     =    ($-buf)
    x     dd   0
    ……
    mov    ecx, n
    mov    ebx, -1
lopa:
    inc    ebx
    cmp    buf[ebx],0
    loope  lopa
    jz     all_zeros
    mov    x, 1
    jmp    exit
all_zeros:
    mov    x, 0
exit: ……
```

假设 buf 缓冲区中的字节全为 0，则在循环结束时，一定有 (ecx)等于 0,且 ZF=1。

假设 buf 缓冲区中只有最后一个字节非 0,则在循环结束时,一定有 (ecx)等于 0,且 ZF=0。

因此，不能用 (ecx)是否为 0 来判断循环结束 buf 缓冲区中是否全为 0。

（3）不等于或不为零循环转移指令

使用格式：**loopne/loopnz　标号**

功能：(ecx/cx)-1→ecx/cx，如果(ecx/cx)不等于零并且 ZF 等于 0，则转标号处执行，否则顺序执行。

**【例 7.3】** 找出以 buf 为首址的 n 个字节中的第一个空格字符出现的位置，若无空格字符，则-1->x，否则将空格在串中出现的位置->x。

算法思想类似于例 7.2，程序核心片段如下：

```
    buf   db  'This is a test'
    n     =   ($-buf)
    x     dd  0
    ……
    mov   ecx, n
    mov   ebx, -1
lopa:
    inc     ebx
    cmp     buf[ebx],' '
    loopne  lopa
    jz      space_occur
    mov     x, -1
    jmp   exit
space_occur:
    mov   x, ebx
exit: ……
```

当然，不使用 loopne 也可以完成相同的功能，程序片段如下：

```
    mov   ecx, n
    mov   ebx, -1
lopa:
    inc   ebx
    cmp   buf[ebx],' '
    jz    space_occur
    loop  lopa
    mov   x, -1
    jmp   exit
space_occur:
    mov   x, ebx
exit: ……
```

（4）跳转指令

使用格式：**jecxz/jcxz　标号**

功能：当寄存器(ecx/cx)的值为 0 时转移到标号处执行，否则顺序执行。

该指令常放在循环开始前，用于检查循环次数否为 0，为 0 时跳过循环体；也常与比较指

令等组合使用，用于判断是由于计数值的原因还是由于满足比较条件而终止循环。

说明：① 所有的循环转移指令本身实施的对(ecx/cx)的值减 1 的操作不影响标志位。

② 在 16 位段的程序中，loop、loopz、loopnz 三条指令缺省使用 cx 寄存器；在 32 位段的程序中则缺省使用 ecx 寄存器。

③ 上述四条循环转移指令的位移量只能为 8 位，也即转移的范围在-128～+127 字节之内。

## 7．2 单重循环程序设计

所谓单重循环，即其循环体内不再包含循环结构。写循环程序并不复杂，只要构思好算法，分配好寄存器的用途，甚至写好 C 代码或者伪代码，然后翻译成汇编语句即可。写 C 伪代码的过程也是理清算法和变量空间分配的过程。

**【例 7.4】** 将以 buf1 为首址的字符串的内容拷贝到以 buf2 为首址的缓冲区中，字符串以 0 结束。

如果 用 C 语言描述，可以定义 char buf1[n]; char buf2[n]。数组元素的下标为 i；

```
i=0;
for (; ;) {
  if (buf1[i] ==0)
      break;
  buf2[i]=buf1[i];
  i++;
}
```

将明确算法思想后，在汇编语言编程前，先对寄存器用途进行分配。

用 ebx 来对应 i，存放要访问的数组元素的下标，这样 buf1[ebx]就对应第(ebx)个字符；用 al 来缓存当前读到的字符。程序核心片段可以写成如下的对应形式。

```
.data
   buf1 db  'Hello',0
   n = $-buf1
   buf2 db  n dup(0)
.code
start:
     mov  ebx, 0
p:   mov  al, buf1[ebx]
     cmp  al, 0
     jz   exit
     mov  buf2[ebx], al
     inc  ebx
     jmp  p
exit: ……
```

程序的其他部分可以参见以前的示例。

【例 7.5】已知以 buf1 为首址的字存储区中存放着 n 个有符号数，试编写程序，将其中大于等于 0 的数依次送入以 buf2 为首址的字存储区中，小于 0 的数依次送入以 buf3 为首址的字存储区中。

同样，先可以用 C 语言表达一下。定义有三个数组 short buf1[n], buf2[n],buf3[n]; 用 i、j、k 分别表示要访问的各个数组元素的下标。

```
for (i=0;i<n;i++)
   if (buf1[i]>0) {
       buf2[j]=buf1[i]; j++;
   }else {
       buf3[k]=buf1[i];  k++;
   }
```

用 ebx 来对应 i，用 esi 来对应 j；用 edi 来对应 k。它们的初值均为 0。程序的核心片段如下。

```
.data
  buf1  sword  10,20, -100,30, -5,70
  n = ($-buf1)/2
  buf2  sword  n dup(0)
  buf3  sword  n dup(0)
.code
start:
    mov  ebx, 0
    mov  esi, 0
    mov  edi, 0
p_loop:
    cmp  ebx, n
    jae  exit
    mov  ax, buf1[ebx*2]
    cmp  ax, 0
    jl   p_negative
    mov  buf2[esi*2], ax
    inc  esi
    jmp  p_modify
p_negative:
    mov  buf3[edi*2], ax
    inc  edi
p_modify:
    inc  ebx
    jmp  p_loop
exit:
```

注意，在程序中选择正确的转移指令，如判断(ax)<0 转移，要用 jl 而不能用 jb。而对于“cmp  ebx，n”之后的转移指令采用“jae  exit”，因为数组元素的个数应看成一个无符号

的数，应该用无符号数比较转移指令。当然，当 n 不是一个很大的数（n<=7FFFFFFFH）时，将它看成一个有符号数也是正数，因此用“jge exit”也可以实现相同的功能。另外，也注意为每个分支安排出口。

## 7．3 多重循环程序设计

多重循环即循环体内再套有循环。设计多重循环时，可以从外层循环到内层循环一层一层地进行。通常在设计外层循环时，仅把内层循环看成一个处理粗框，然后再将该粗框细化，分成置初值、工作、修改和控制四个组成部分。当内层循环设计完之后，用其替代外层循环体中被视为一个处理粗框的对应部分，这样就构成了一个多重循环。对于程序，这种替换是必要的，对于流程图，如果关系复杂，可以不替换，只要把细化的流程图与其对应的处理框联系起来即可。下面以两重循环为例说明多重循环程序的设计。

**【例 7.6】**设以 buf 为首址的双字存储区中存放着 n 个有符号数，试编写程序，将其中的数按从小到大的顺序排列，并输出排序结果。

同样，我们可以先用高级语言（或伪代码）来表达算法思想。数组定义为 int buf[n];

```
for  (i=0;i<n-1;i++) {
    将数组中第 i 小的数，排在 buf[i]的位置。
}
```

由于在排第 i 小的数时，buf[0],……,buf[i-1]是已经排好的，因此第 i 小的数只从 buf[i]到 buf[n-1]中找。此时算法思想有了进一步细化，可表示如下：

```
for  (i=0;i<n-1;i++) {
    从 buf[i]到 buf[n-1]中找最小的数，将其排在 buf[i]的位置。
}
```

再进一步细化，可以将 buf[i]和数组后面的数逐个比较，若发现后面的数比 buf[i]小，则交换两者的顺序。这又是一个循环。算法细化结果如下：

```
for  (i=0;i<n-1;i++) {
    // 从 buf[i]到 buf[N-1]中找最小的数，将其排在 buf[i]的位置。
    for (j=i+1;j<n;j++)
       if (buf[i]>buf[j])  交换 buf[i] 和 buf[j]
}
```

至此，通过由粗到细、逐步细化的方法，将算法细化每个步骤可以直接用一条或几条语句来描述即可。在编写汇编语言源程序之前，还要分配寄存器的用途。例如，用 esi 来对应 i，用 edi 来对应 j，用 eax 来表示中间读到的数据。

在写程序时，同样按照模块化的思想，可以先写外层循环的语句，内循环处暂时以“……”或者注释等代替。写好外循环，阅读感觉无误后，再补充内循环的语句。本例中外循环程序段如下。

```
        mov  esi, 0
Out_Loop:    ; 外循环
        cmp  esi, n-1
        jae  exit
         ……          ; 此处是内循环
```

```
Inner_Loop_Over:
    inc  esi
    jmp  Out_Loop
exit:
```

对于输出排序结果，可以用一个循环次数固定的单循环来实现，算法思想和寄存器用途的分析从略。完整的程序如下。

```
.686P
.model flat, c
  ExitProcess proto stdcall :dword
  includelib  kernel32.lib
  printf      proto :ptr sbyte, :vararg
  includelib  libcmt.lib
  includelib  legacy_stdio_definitions.lib
.data
  lpFmt   db  "%d ", 0
  buf     sdword  -10, 20, 30,-100, 25, 60
  n     =  ($-buf)/4
.stack 200
.code
 main  proc
   mov  esi, 0
Out_Loop:     ; 外循环
   cmp  esi, n-1
   jae  exit
        ; 下面是内循环
   lea  edi, [esi+1]     ; 等价于 mov edi，esi 和 inc edi 两条语句的功能
   Inner_Loop:
      cmp  edi, n
       jae  Inner_Loop_Over
       mov  eax, buf[esi*4]
       cmp  eax, buf[edi*4]
       jle  Inner_Modify
       xchg eax, buf[edi*4]
       mov  buf[esi*4], eax
  Inner_Modify:    ; 修改内循环的控制变量
       inc  edi
       jmp  Inner_Loop
  Inner_Loop_Over:
  inc  esi
  jmp  Out_Loop
exit:  ; 用循环输出结果
```

```
    mov  esi, 0
display:
    cmp  esi, n
    jae  Program_Over
     invoke printf,offset lpFmt, buf[esi*4]
    inc  esi
    jmp  display
Program_Over:
   invoke ExitProcess, 0
main  endp
end
```

注意，使用变址寻址方式访问数组元素是非常方便的。用 32 位寄存器作为元素索引，它乘以一个比例因子，比例因子对应每个元素的长度。

当然，在程序中添加适当的注释，给标号取一个好记忆易理解的名字，都可以提高程序的可读性。

## 7．4 循环程序中的细节分析

在编写循环程序时，也有许多注意的细节，若不仔细就可能出现各种各样的问题。下面通过示例来分析，修改一个循环程序后导致的后果。

**【例 7.7】** 已知有 n 个元素存放在以 buf 为首址的双字存储区中，试统计其中负元素的个数存放到变量 r 中。

显然，每个元素为一个 32 位有符号二进制数。统计其中负元素个数的工作可用循环程序实现。

存储单元及寄存器分配如下：

ebx：buf 存储区的地址指针，初值为 buf 的偏移地址，每循环一次之后，其值增 1。

ecx：循环计数器，初值为 buf 区中元素的个数 n，每循环一次之后，其值减 1。

eax：用来记录负元素的个数，初值为零。

双字变量 r：用来存放负元素的个数。

程序的流程图如图 7.2 所示。统计负元素个数的程序流程图的循环结构类同于图 7.1(a)。在循环结束之后，用（eax）→r 将负元素的个数送入了字变量 r 之中。

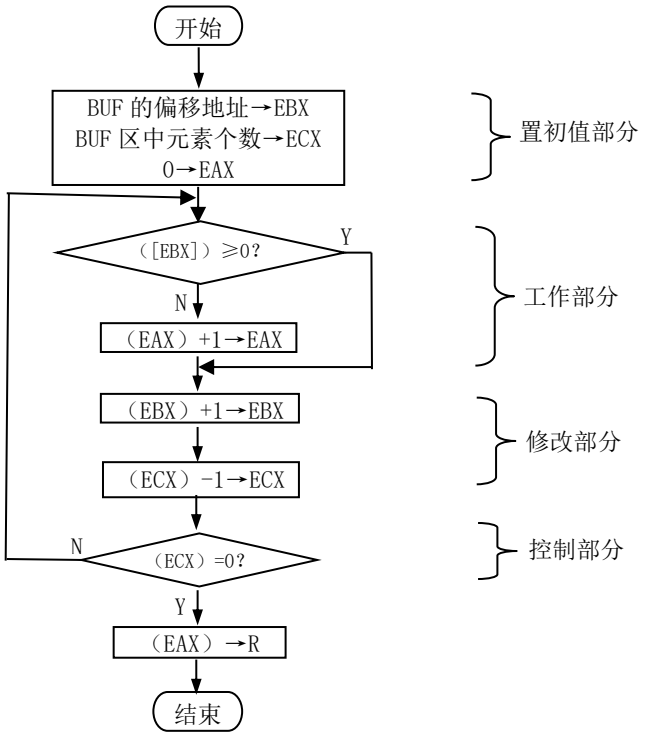


图 7.2　统计负元素个数程序流程图

程序如下：

```
.686P
.model flat, c
  ExitProcess proto stdcall :dword
  includelib  kernel32.lib
  includelib  msvcrt.lib
  printf      proto :ptr sbyte, :vararg
  includelib  libcmt.lib
  includelib  legacy_stdio_definitions.lib
.data
  lpFmt  db "%d",0ah, 0dh, 0
  buf    dd  -20,50,-30,6,100,-200,70
  n      =  ($-buf)/4          ; buf 区中元素个数
  r      dd  0
.stack 200
.code
 main  proc
   lea  ebx, buf
   mov  ecx, n              ;置循环初值部分
   xor  eax, eax
lopa:
   cmp  dword ptr [ebx],0   ;工作部分（循环体）
   jge  next
```

```
    inc  eax
next:
    add  ebx, 4              ;修改部分
    dec  ecx
    jnz  lopa                ;控制部分
    mov  r, eax
    invoke printf,offset lpFmt, r    ; 显示负数个数
    invoke ExitProcess, 0
main  endp
end
```

该程序的循环体被重复执行了 n 次，即当(ecx)=n，n-1，…，1 时循环执行，当(ecx)等于 0 时时结束循环，将负元素个数送入字变量 r 中之后，返回操作系统状态。程序执行后，显示 3，表示有 3 个负数。

程序虽然简单，但写程序时要注意一些细节问题，如语句的摆放顺序、标号的位置等，稍有不慎，就可能导致各种各样的问题。下面将对例 7.7 中程序进行细小的修改，分析修改后程序运行情况的变化。

（1）交换置循环初值部分中的语句的位置

```
    xor  eax，eax
    lea  ebx，buf
    mov  ecx，  n
```

这三条语句之间是没有先后关系的，交换前后的功能完全等价。

（2）交换修改部分的语句

设调整后的程序的核心段如下：

```
lopa:  cmp  dword ptr [ebx],0
       jge  next
       inc  eax      ; eax 用来记录负元素的个数
next:  dec  ecx      ; ecx 用来记录待判断的元素个数
       add  ebx, 4   ; ebx 待访问的元素的地址
       jnz  lopa
```

由于 dec、add 指令都会影响标志位，交换 add 和 dec 指令的顺序后，jnz lopa 中所用的 ZF 是“add  ebx, 4”设置的。此时的循环次数已不受 ecx 控制，相当于语句“ dec ecx”成了“废语句”。

下面更深入的分析修改后程序运行的结果。程序会死循环吗？

表面上看，只要(ebx)的初值是 4 的倍数，通过不断的加 4，最后会变成 0。例如(ebx)=7FFFFFFC,加 4 后变成 80000000H,再加 4，变成 80000004H，依此类推，最后加到(ebx)=0FFFFFFFCH, 此时再加 4，(ebx)=0; ZF =1。 jnz 的条件不成立，不转移到 lopa 处，循环结束。也许有读者会问，若(ebx)的初值不是 4 的倍数，那不论怎样加 4 就永远得不到结果为 0 的情况，程序死循环。其实这两种情况都不会发生，程序运行后，出现一个异常界面，提示在程序指令地址为 *** 处出现了“访问冲突”，表示程序要访问一个不被允许访问的内存单元。编写过 C 语言程序的人对此异常窗口应该不陌生，在指针、数组等使用不正确时可能出现该问题。

回顾一下 2.6 节所学内容，是不难解释为什么会出现这种情况的。因为在保护模式下，各个程序都有自己的空间，通过分段来限制自己只访问它内部的空间，不能越界去访问别人的空间。在循环过程中，(ebx)在不断的增加，到一定的时候就会冲出本程序限定的空间，从而触发异常。

提示：如果单击异常窗口中的“中断”按钮，会出现程序的调试窗口。在该窗口的左边，可以看到，程序是执行到 “cmp dword ptr [ebx],0”时出现了异常。再打开寄存器窗口，可以看到 ebx、ecx。 (ecx)的初值是 n(n=7)，每循环 1 次，(ecx)减 1，减到 0 后，再减 1，就是 0FFFFFFFFH，继续不断的减 1，直到出现异常。异常发生时(ecx)=0FFFFF009H，表明循环次数为 0FFEH 次（0FFFFF009+0FFEH=7）。

（3）置循环初值部分语句写到了循环中

设调整后的程序的核心段如下：

```
lopa:
        lea  ebx,  buf
        cmp  dword ptr [ebx],0
        jge  next
        inc  eax
next:   add  ebx, 4
        dec  ecx
        jnz  lopa
```

程序运行后，将显示 7，即统计出缓冲区中有 7 个负数，这显然是错误的。原因就在于每次循环访问的数据都是缓冲区的开头大那一个数据，数据元素指针在循环中被错误的复原到数组的开始位置。

（4）将 ecx 赋初值语句写到了循环中

```
lopa:  mov ecx，n
```

表面上看，是死循环，但是实际运行结果与 (2) 相同，因为 ebx 的不断增加，使得[ebx]访问的单元超出了本程序的保护范围，程序运行崩溃。

（5）程序段的优化

前面的程序中，用 ecx 来控制循环次数，用 ebx 来指明访问单元的地址。下面有变址寻址方式来访问存储单元。用 ebx 来指示元素下标，因而也可用于循环次数控制。减少使用一个寄存器，核心程序段如下。

```
        xor  eax, eax
        xor  ebx, ebx
lopa:
        cmp  buf[ebx*4],0
        jge  next
        inc  eax
next:
        inc  ebx
        cmp  ebx，n    ; 每循环一次，ebx 加 1，当 ebx 等于 n 时，循环结束
        jnz  lopa
```

本例中如果将标号 lopa 上移一行，变成 “lopa:  xor ebx, ebx”就会导致死循环。

从上面的这些例子可以看到，写程序时要仔细。稍有不慎，就会导致程序不能完成预定的功能。在阅读和分析程序时，要仔细思考每个语句带来的变化，前后的语句能否按算法有机的结合在一起，对执行结果也不能想当然。

## 7．5 与 C 循环程序反汇编的比较

通过阅读 C 语言循环程序的反汇编代码，可以熟悉用汇编语言写代码的方法，在某种程度上实现了“人工的编译”。当然，通过对照 C 源程序和反汇编代码，也可以了解编译器所做的一些工作。此外，也可以思考编译器可以对生成的代码做哪些优化。

C 语言程序功能，对一个数组数据按从小到大的顺序排序，输出排序结果。在例 7.6 中，给出了汇编语言实现的代码。C 语言程序如下。

```
#include <stdio.h>
#define n 6
int main(int argc, char* argv[])
{
    int buf[n] = { -10, 20, 30, -100, 25, 60 };
    int x;
    int  i;
    int  j;
    for (i = 0; i < n - 1; i++) {
       for (j = i + 1; j < n;j++)
          if (buf[i] > buf[j]) {
             x = buf[i];
             buf[i] = buf[j];
             buf[j] = x;
          }
    }
    for (i = 0; i < N;i++)
       printf("%d ", buf[i]);
    return 0;
}
```

若对该程序进行反汇编，可以看到如下的结果。

```
    int buf[n] = { -10, 20, 30, -100, 25, 60 };
00A813DE  mov         dword ptr [buf],0FFFFFFF6h
00A813E5  mov         dword ptr [ebp-18h],14h
00A813EC  mov         dword ptr [ebp-14h],1Eh
00A813F3  mov         dword ptr [ebp-10h],0FFFFFF9Ch
00A813FA  mov         dword ptr [ebp-0Ch],19h
00A81401  mov         dword ptr [ebp-8],3Ch
    int x;
    int  i;
```

```
    int  j;
    for (i = 0; i < n - 1; i++) {
00A81408  mov         dword ptr [i],0
00A8140F  jmp         main+5Ah (0A8141Ah)
00A81411  mov         eax,dword ptr [i]
00A81414  add         eax,1
00A81417  mov         dword ptr [i],eax
00A8141A  cmp         dword ptr [i],5
00A8141E  jge         main+0B0h (0A81470h)
        for (j = i + 1; j < n;j++)
00A81420  mov         eax,dword ptr [i]
00A81423  add         eax,1
00A81426  mov         dword ptr [j],eax
00A81429  jmp         main+74h (0A81434h)
00A8142B  mov         eax,dword ptr [j]
00A8142E  add         eax,1
00A81431  mov         dword ptr [j],eax
00A81434  cmp         dword ptr [j],6
00A81438  jge         main+0AEh (0A8146Eh)
          if (buf[i] > buf[j]) {
00A8143A  mov         eax,dword ptr [i]
          if (buf[i] > buf[j]) {
00A8143D  mov         ecx,dword ptr [j]
00A81440  mov         edx,dword ptr buf[eax*4]
00A81444  cmp         edx,dword ptr buf[ecx*4]
00A81448  jle         main+0ACh (0A8146Ch)
             x = buf[i];
00A8144A  mov         eax,dword ptr [i]
00A8144D  mov         ecx,dword ptr buf[eax*4]
00A81451  mov         dword ptr [x],ecx
             buf[i] = buf[j];
00A81454  mov         eax,dword ptr [i]
00A81457  mov         ecx,dword ptr [j]
00A8145A  mov         edx,dword ptr buf[ecx*4]
00A8145E  mov         dword ptr buf[eax*4],edx
             buf[j] = x;
00A81462  mov         eax,dword ptr [j]
00A81465  mov         ecx,dword ptr [x]
00A81468  mov         dword ptr buf[eax*4],ecx
          }
    }
```

```
00A8146C  jmp         main+6Bh (0A8142Bh)
00A8146E  jmp         main+51h (0A81411h)
    for (i = 0; i < n;i++)
00A81470  mov         dword ptr [i],0
00A81477  jmp         main+0C2h (0A81482h)
00A81479  mov         eax,dword ptr [i]
00A8147C  add         eax,1
00A8147F  mov         dword ptr [i],eax
00A81482  cmp         dword ptr [i],6
00A81486  jge         main+0E9h (0A814A9h)
        printf("%d ", buf[i]);
00A81488  mov         esi,esp
00A8148A  mov         eax,dword ptr [i]
00A8148D  mov         ecx,dword ptr buf[eax*4]
00A81491  push        ecx
00A81492  push        0A85858h
00A81497  call        dword ptr ds:[0A89114h]
        printf("%d ", buf[i]);
00A8149D  add         esp,8
00A814A0  cmp         esi,esp
00A814A2  call        __RTC_CheckEsp (0A81136h)
00A814A7  jmp         main+0B9h (0A81479h)
    return 0;
00A814A9  xor         eax,eax
}
```

当然，用不同的优化级别，生成的目标文件会有差异。

从本例中，可以看到C语言编译后的代码长度要长得多，执行的效率也没有直接用汇编语言写的程序运行快。不可否认的是，用 C 语言编写的程序的易读性要强。两者都有各自的优点。

在多模块程序设计中，我们将看到C语言和汇编语言的混合编程，这样对一些关键核心代码用汇编语言实现，使得其有较高的执行效率；同时也保持了用高级语言编写程序的优点。C语言中提供的一些库函数是也正是这样做的，用汇编语言编写使其具有更高的执行效率。

## 7．6 循环控制伪指令

与分支程序设计中的条件判断伪指令类似，Microsoft 的汇编语言程序编译器也支持循环控制伪指令。使用这些伪指令可以简化程序的编写，但对初学者不推荐使用。从理论知识的学习角度来看，还是应该使用对应的机器指令，理解机器工作的基本原理。

1、循环执行伪指令

指令格式如下：

.while　条件表达式1

语句序列 1
[.break　[.if　条件表达式 2]]
[.continue　[.if　条件表达式 3]]
语句序列 2

.endw

当条件表达式 1 为真时，条件成立，执行.while 和.endw 之间的语句序列，然后再回到.while 处进行条件判断，重复此过程，直到条件不成立，转移到.endw 之后的位置。在循环体中，可以含有中断循环伪指令（.break）或者继续循环伪指令（.continue）。这些语句的作用与 C 语言程序设计中的相应语句的作用是相同的。

2、重复执行伪指令

指令格式如下：

.repeat
语句序列
.until 条件表达式

该语句的执行流程与 C 语言中的“do ……while” 语句是一样的，即先执行语句序列，然后判断条件表达式是否成立。若条件成立，则继续执行 repeat 后的语句序列结构；若不成立，则循环结束。在语句序列中与 while 语句一样，也可以含有中断循环伪指令（.break）或者继续循环伪指令（.continue）。

repeat 伪指令还有一种格式如下：

.repeat
语句序列
.untilcxz [条件表达式]

在该语句中，条件表达式是可选项。当无条件表达式只有 untilcxz 时，它等价于 loop 指令，即先执行(ecx)-1->ecx，然后判断(ecx)是否为 0，若不为 0，则继续执行.repeat 后的语句序列；若(ecx)为 0，则循环结束。在.untilcxz 后，有条件表达式是，类似于 loopne 指令，当条件表达式不成立且(ecx)≠0 时循环，直到条件表达式成立或者(ecx)=0。

3、中断循环伪指令

中断循环伪指令有两种形式，一是简单的无条件的 break，语句格式为：

.break

该语句的等价于 jmp，无条件的跳转到循环语句的下方。另一种是带条件的 break，语句格式为：

.break .if 条件表达式

当条件表达式成立时它跳出循环，否则继续执行 .break 之下的语句。

4、继续循环伪指令

继续循环伪指令也有两种形式，一是简单的无条件的继续循环，语句格式为：

.continue

语句的执行流程与 C 语言中的 continue 是相同的，它结束本次循环，重头开始下一次循环。

另一种是带条件的 continue，语句格式为：

.continue .if 条件表达式

当条件表达式成立时，它结束本次循环，重头开始下一次循环；否则继续执行本次循环，

即要执行.continue 之下的语句。

下面给出一个简单的使用条件流控制伪指令编写的程序，实现的功能与例 7.7 相同，即统计以 buf 为首址的双字存储区中负元素的个数。

```
.686P
.model flat, c
  ExitProcess proto stacall :dword
  printf      proto  :vararg
  includelib  libcmt.lib
  includelib  legacy_stdio_definitions.lib
.data
  lpFmt db  "%d",0ah, 0dh, 0
  buf   sdword  -20,50,-30,6,100,-200,70
  n     =  ($-buf)/4
  r     dd  0
.stack 200
.code
 main  proc
   xor  eax, eax
   xor  ebx, ebx
   mov  ecx, n
 .repeat
   .if buf[ebx*4]<0
       inc  eax
   .endif
   inc  ebx
.untilcxz
   mov  r, eax
   invoke printf,offset lpFmt, r
   invoke ExitProcess, 0
main  endp
end
```

注意，在定义 buf 时应使用 sdword 而不能是 dd 或者 dword。只有这样编译器才会选择有符号数的比较转移指令。在最开始的存储模型说明、函数原型说明上也有一点变化，但编译后的最终结果是一样的。

# 习题 7

7.1　设以 buf 为首址的双字存储区中存放着 n 个有符号数，试编写程序，找出其中的最大数并显示。

7.2　设分别以 str1 和 str2 为首址的字节存储区中存放以 0 为结束字节的字符串，试编写程序比较两个串是否相等，若相等，则输出 equal，否则输出 not equal。

7.3　对习题 7.2 进行修改，两个字符串在程序的运行过程中由用户输入。

输入串的方法用 scanf 函数。

数据段中的定义的变量有:

```
lpFmt db “%19s”, 0
str1  db  20 dup(0) ……
invoke scanf, offset lpFmt, offset str1
```

7.4　设以 buf 为首址的双字存储区中存放着 n 个有符号数,试编写程序,用冒泡排序的方法，对其按从小到大的顺序排列，之后输出排序结果。

7.5　对习题 7.4 进行修改，　n 个有符号数在程序的运行过程中由用户输入。

# 上机实践 7

7.1　编写一个程序，实现将一个数字 ASCII 串转换为整数的功能（类似 C 语言中的 atoi）。

7.2　编写一个程序，实现将一个整数转换成 ASCII 串的功能（类似 C 语言中的 itoa）。

7.3　编写一个程序,实现对一个二维数组的求和。要求不能使用二重循环,只能使用单循环。

7.4　设数据段定义有如下字符串表，其中每个字符串都是 10 个字节，以 0 结束。

```
stringstab  db 'good',0,(10+ stringstab -$) dup(0)
            db 'hello',0,(20+ stringstab -$) dup(0)
            db 'asm',0,(30+ stringstab -$) dup(0)
            db 'language',0,(40+ stringstab -$) dup(0)
```

编写一个程序，输入一个字符串，在 stringstab 中查找该串是否出现（两个串完全相同才算出现）。