目录

第 8 章 子程序设计

子程序设计是程序设计中最重要的方法与技术之一。本节将重点介绍子程序的概念、主程序对子程序的调用指令、从子程序返回主程序的指令、主程序向子程序传递参数及子程序向主程序返回结果的方法、子程序中对寄存器的保护、子程序中局部变量的空间分配。在介绍基本概念和方法后，通过具体实例，讨论简单子程序、递归子程序的设计的方法。最后，根据所学原理，通过反汇编的手段，分析 C 语言程序中函数调用的实现方法。读者应熟练地掌握子程序设计的基本技术及子程序的调用方式，并能独立地编制各种功能的子程序。

8．1 子程序的概念

在 C 语言程序设计中会大量的使用函数。在汇编语言中将函数也称为子程序。为什么程序设计要将程序分成一个个函数呢？

函数或子程序是模块化程序设计或模块化设计发展的产物。在面对一个大型任务时首先要有一个全局观，将大任务划分成一系列子任务，并明确子任务之间的相互关系。之后才是解决各个子任务。对程序设计而言也是如此，先用主程序、子程序、子过程等框架把软件的主要结构和流程描述出来，并定义和调试好各个框架之间的输入、输出链接关系。然后再逐步求精，设计一系列以功能块为单位的算法。这种以功能块为单位进行程序设计，实现其求解算法的方法就是模块化设计。模块化的目的是为了降低程序复杂度，使程序设计、调试和维护等操作简单化。

1956 年 George A.Miller 在他的著名文章“奇妙的数字 7±2——人类信息处理能力的限度”（The Magic Number Seven, Plus or Minus Two: Some Limits on our capacity for Processing Information）中指出，普通人分辨和记忆同一类信息的不同品种和等级的数量一般不超过 5~9 项，这表明要使人的智力足以管理好程序，应坚持模块化设计。大型的单模块软件不仅可读性差、可靠性也常常难以保证。函数就是一段代码的封装，它为程序员开发和维护程序较大型的程序提供了有力的支撑。

函数设计的质量对于程序设计而言是非常重要的。表面上看，一个函数不能太大，实现的功能应简单，通常不超过 50 行，在一个屏幕上能看到一个完整的函数为好。另外，函数与函数之间的接口要简单，即模块内部耦合紧密，模块之间联系松散。此外，还有很多函数是让不同的程序共同使用，例如输入输出程序、数制转换程序、三角函数、指数函数、解线性方程组等都是采用子程序方式事先编好，组成函数库，提供给程序开发者使用。当程序开发者需要使用某个函数时，只要按照系统规定的调用方式对它调用即可。因而，可以非常有效的降低开发的劳动强度。当然，一个公司可以设计和开发一些这样的函数，供一个项目内部或者在项目之间共享使用。开发这些函数时就尽量不使用外部的全局变量，不依赖于外部的信息。

前面介绍了为什么要使用函数以及设计函数时应注意的问题。从计算机内部工作的原理来看，有很多疑问摆在我们的面前。为什么调用一个函数时能进入到函数中去执行？为什么函数执行完后又能够返回到调用处继续执行下面的语句？主函数和被调用函数之间是如何传递数

据的？函数可以被递归调用，在其运行过程中，不可能为其局部变量分配一个固定的存储空间，函数中的变量分配的空间又在什么位置呢？在学习了本章内容后，通过对高级语言编译生成的机器语言程序进行反汇编，将一一得到这些问题的答案。

在汇编语言程序设计中，通常将函数称为子程序。子程序的语法格式是什么？如何调用子程序，如何从子程序返回？如何在主程序与子程序之间传递参数？下面分别讨论这些问题。

8．2 子程序的基本用法

8.2.1 子程序的定义

子程序也是一段程序，它的基本格式如下：

```
子程序名    proc
            子程序体的语句.....
子程序名    endp
```

其中子程序名由用户定义，其命名应符合一般的标识符的命名规则。子程序要放在代码段中。从语法上看，子程序可以放在代码段的任何位置，但是从程序运行逻辑上看，还是有限制的。一般将其放在主程序结束之后，或者放在主程序开始之前。子程序放在主程序之后的结构如下：

```
.code
main   proc
      ……..
main   endp          ; 主程序运行到此处结束，返回操作系统
func1   proc         ; 子程序 func1
         ......
func1   endp
func2   proc         ; 子程序 func2
         ........
func2   endp
end
```

子程序放在主程序之前的结构如下：

```
.code
func1   proc         ; 子程序 func1
        ......
func1   endp
……….            ; 其他子程序
main    proc
      ……..
      ; 主程序运行到此处结束，返回操作系统
main    endp
```

此外，可以将主程序也写成一个子程序的形式，如同 C 语言的 main 函数。同样主程序可放在代码段的开头或者最后。形式如下：

```
.code
```

```
mainp   proc        ; 主子程序
   ..................
mainp   endp
func1    proc
   ..............
func1    endp
end   mainp
```

注意：子程序的名字是自己命名的，只要符合标识符的命名规范即可。

8.2.2 子程序的调用和返回

主程序与子程序存在调用与被调用的关系，如图 8.1 给出了两者之间的关系示意。调用子程序的程序称为主程序（或称调用程序），被调用的程序称为子程序。当然，主程序和子程序是一个相对的概念，一个子程序既被别的程序调用，也可以调用其他的子程序。一个子程序可调用其自身，形成递归调用关系。

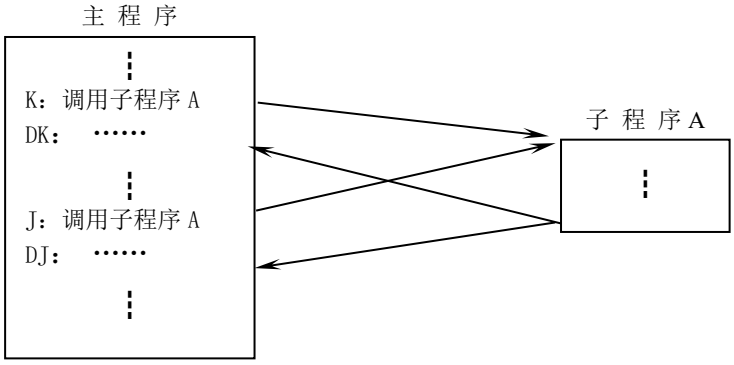


图 8.1 主程序和子程序关系示意图

在图 8.1 中，主程序在 K 处和 J 处均调用了子程序 A。当主程序调用子程序后，CPU 就转去执行子程序，执行完毕则返回到主程序的断点处继续往下执行。断点是指转子指令的直接后继指令的地址。如在 K 处调用子程序 A，断点就是 DK，子程序 A 执行完之后，返回到 DK 处继续往下执行。同理，在 J 处调用子程序 A，DJ 就是断点，子程序 A 执行完之后，返回到 DJ 处继续往下执行。

前面已学过无条件转移指令。在 K 处和 J 处调用子程序 A，就相当于 jmp　A，将 EIP 设置为希望转移的目的地址。但是两次调用后，从子程序返回的目的地不同，不可能在子程序中固定写死返回到一个位置。因而，在某个地方调用子程序时，要将从子程序返回的目的地址保存在某个位置，在返回时从该位置读取返回地址送给 EIP，从而实现程序的跳转。为了实现这些功能，x86 设计了两条机器指令 call 和 ret。

1、 子程序调用指令

1）直接调用

语句格式：call　过程名

功能：　①（EIP）→↓（esp）

②过程名的地址 EA→EIP

2）间接调用

语句格式：call　dword　ptr　opd

功能：　①（EIP）→↓（esp）保存断点地址在堆栈的栈顶

②（opd）→EIP

opd 可以是寄存器寻址、寄存器间接寻址、变址寻址、基址加变址寻址和直接寻址方式，这与 jmp 的间接转移指令中所使用的表达式是一样的。与 jmp 指令相比，call 执行多执行了一个步骤，先保存断点地址在堆栈的栈顶；第二个步骤与 jmp 相同，即通过直接调用的方式，得到目的地址，或者通过某种寻址方式间接从(opd)单元中取得转移地址。

例如：设数据段定义有变量 func_table dd myfunc；在代码段中有子程序 myfunc，则如下语句的功能是等价的。

① call myfunc

② call func_table

③ mov eax, func_table
call eax

④ lea eax, func_table
call dword ptr [eax]

⑤ mov eax, 0
call func_table[eax]

注意，在分段管理模式中，调用可以分为段内直接调用、段内间接调用、段间直接调用、段间间接调用。所谓段内指主程序与子程序在同一个代码段内；段间指主程序与子程序不在同一个段。对于段间调用，就要先将段寄存器 CS 的值先压到堆栈中，之后再将断点的偏移地址压到堆栈中。在 32 位段扁平内存管理模式下（对应本书中的.model flat）变得简单了，只有断点的 32 位偏移地址压入堆栈。

2、返回指令

ret 指令用于子程序中，控制 CPU 返回到主程序的断点处继续往下执行。ret 有两种语法格式。

语句格式：**ret**

功能：从堆栈栈顶弹出一个双字送给 EIP，即↑（esp）→EIP

语句格式：ret　n

功能：

① 从堆栈栈顶弹出一个双字送给 EIP，即↑（esp）→EIP

②（esp）+ n→esp

说明:n 是正整数,且为偶数，用于废除栈顶无用的参数。n 为所有无用参数所占字节数的和。当主程序与子程序采用堆栈法传递参数时，可利用这种返回形式消除不再使用的入口参数对堆栈空间的占用。消除入口参数所占的空间是很有必要的。若不消除参数所占的空间，而该子程序又被反复的调用，则堆栈空间就会被消耗完，从而引发访问错误。

对于参数所占用的空间，可以在主程序通过执行 “add esp，n”来消除参数所占的空间。实际上 C 语言程序中函数调用都是在主程序中来清除参数所占的空间。因为，有像 printf 这类参数个数不定的函数。子程序中并不知道主程序传递过来了多少个参数，占据了多少个字节。而在主程序中有函数调用语句，其中就给定了参数的个数和大小，因而主程序是知道自己传递了多少字节的信息的。

8.2.3 主程序与子程序之间传递参数

主程序在调用子程序之前，必须把需要子程序处理的原始数据传递给子程序，即为子程序准备入口参数。子程序对其入口参数进行一系列处理之后得到处理结果，该结果必须送给调用它的主程序，即提供出口参数以便主程序使用。这种主程序为子程序准备入口参数、子程序为主程序提供出口结果的过程称为参数传递。

主程序与子程序之间传递参数的方式是事先约定好的。每一个子程序设计之前，必须确定其入口参数来自哪里，处理后的结果送往何处。一旦子程序按此约定设计出来，无论在何处对它进行调用都必须满足子程序的要求，否则，子程序将无法正常运行，或者得不到正确的结果。

常用的参数传递方式有寄存器法、约定单元法和堆栈法三种。

1、寄存器法

寄存器法就是将子程序的入口参数和出口参数都存放在约定的寄存器之中。该方法使用起来简单，但由于寄存器的个数有限，因而只适用于要传递的参数较少的情况。

2、约定单元法

约定单元法是把入口参数和出口参数都放到事先约定好的存储单元之中。此法的优点是每个子程序要处理的数据或送出的结果都有独立的存储单元，编写程序时不易出错；参数的数量可多可少。它致命的缺点是子程序中要用到全局变量，这样子程序就很难在多个程序间共享，模块的独立性受到很大的影响。

3、堆栈法

堆栈法通过堆栈来传递参数。该方法的优点是参数不占用寄存器，也无需另开辟存储单元，而是存放在公用的堆栈区，处理完之后堆栈恢复原状，仍可供其它程序段使用；参数的数量可多可少，具有较高的灵活性。

高级语言的编译器一般都将函数调用时的参数传递翻译成堆栈参数传递方式。如 C、pascal 等主要采用堆栈法传递参数，并规定了各参数压栈的次序以及清除栈顶参数的方法等。采用堆栈方法传递参数的最大优点是降低了子程序之间、子程序与主程序之间的关联程度，有利于提高程序的模块化程度，因而推荐使用该方法。

在使用堆栈法传递参数时，一定是先将参数压入堆栈，然后执行子程序调用 call 语句。因为一旦执行 call 语句，就要先将断点地址压入堆栈，然后将 EIP 改为子程序的入口地址，之后再取指令，就是子程序的开始位置的指令了。

下面通过一个 C 语言程序段的反汇编代码来观察参数传递的方法。C 程序片段如下：

```
int    x = 20;
int    y = 30;
printf("x=%d y=%d \n", x, y);
```

在程序调试时，在反汇编窗口，可看到如下代码：

```
int    x = 20;
00941653       mov      dword ptr [x],14h
int    y = 30;
0094165A       mov      dword ptr [y],1Eh
printf("x=%d y=%d \n", x, y);
00941661       mov      eax,dword ptr [y]
00941664       push     eax
00941665       mov      ecx,dword ptr [x]
```

```
00941668      push     ecx
00941669      push     offset string "x=%d y=%d \n" (0945B30h)
0094166E      call      _printf (0941037h)
00941673      add      esp,0Ch
```

如果跟踪进入函数 printf ，刚进入函数时，观察堆栈（也即观察内存，内存地址为(esp)），可以看到如下内容：

73 16 94 00 30 5b 94 00 14 00 00 00 1e 00 00 00

其中，栈顶 4 个字节为 00941673h，即断点地址；00945b30h 为各式串的首地址；之后是参数 x 的值 00000014h；最后是参数 y 的值 0000001eh。

在 call 指令之下，有指令“add esp, 0Ch”，其作用是消除三个参数所占的空间。

由上述分析可知，在刚刚进入子程序时，（esp）为栈顶元素的地址，其中存放的断点地址，([esp+4])中的内容就是最后压入的那个参数。参数被压在了断点地址的下方，如何在子程序中取到这些参数呢?

对于简单程序，我们可以使用[esp+4]去访问这个参数。但是假如多次访问这个参数，那就必须保证在子程序中(esp)不发生变化，否则取到的数据就不是同一个。一个很通常的做法是：进入子程序时，先保护(ebp)，即 push ebp，然后将(esp)送给 ebp，即执行 mov ebp, esp。如果跟踪进入 printf 函数，可以看到该函数的开头语句为：

```
push     ebp
mov      ebp, esp
```

执行这两条语句后，堆栈的示意图如图 8.2 所示。

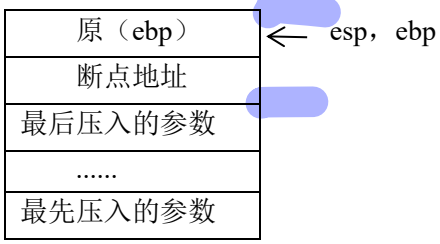


图 8.2 堆栈示意图

在子程序中不再改变 ebp，这样([ebp+8]) 中的内容为最后压入的那个参数。同样用([ebp+n]) 可以访问其它参数。在子程序中可以有 push 和 pop 指令，它们会改变 esp，但不会影响 ebp。

在 32 位平台中，常用堆栈来传递参数。当然，在写程序时可以混合使用寄存器、约定的存储单元和堆栈来传递参数。

8.2.4 子程序调用现场的保护

在用汇编语言编写程序时，往往在实现某功能的一段程序中，给一些寄存器指定了某种用途。若在该段程序中调用了子程序，子程序中又要使用寄存器，怎么办？例如，给定了一个学生信息表，每个学生都有一个姓名（字符串），现在给定一个名字（即一个字符串），要查询该名字是否出现在学生信息表中。显然，我们可以像 C 语言那样，写一个子程序完成两个串比较（strcmp）的功能。在主程序中，循环调用 strcmp，依次将给定的名字和学生信息表中的名字进行比较。假如在主程序中使用了 ecx 来控制循环次数，在子程序中使用 ecx 来记录两个字符串比较的字符个数，当从子程序返回主程序时，ecx 中的内容已被改变，主程序的循环次数控制就会出现错误。因此，在子程序设计中必须考虑现场的保护与恢复。

保护与恢复现场的工作可以在主程序中完成，也可以在子程序中完成，一般情况下是在子

程序中完成，即在子程序的开始处安排一串保护现场的语句，在返回指令之前再恢复现场。这样，主程序在调用子程序时均不必考虑保护、恢复现场的工作，其处理流程清晰，调用简单方便，整个代码简短紧凑。尤其是在进行模块程序设计时，公共子模块需要提供给大家调用，显然，这种方式对调用者来说是最简单的。

保护寄存器最简单的方式是使用堆栈。例如，子程序要改变寄存器 eax、ebx、ecx、edx 四个寄存器的内容，而主程序中不希望这些寄存器的内容被子程序破坏，则在子程序的开始处将这些寄存器的内容入栈保护，在子程序的返回指令之前用出栈指令恢复它们。实现方法如下：

```
push   eax
push   ebx
push   ecx
push   edx
.......
pop    edx
pop    ecx
pop    ebx
pop    eax
```

注意，由于是在堆栈中保存各寄存器的内容，堆栈操作采用先进后出的原则，入栈保护的顺序与出栈恢复的顺序相反，这样才能保证子程序的运行不破坏主程序的工作现场。通过寄存器的保护和恢复，使得在子程序执行前后该寄存器的内容一样，就像子程序未执行过一样。

此外，如果一个寄存器用于存放返回结果，则该寄存器不应在子程序中保护和恢复。

8.2.5 子程序设计应注意的问题

1、程序摆放的位置

子程序不能随意摆在程序中，一般在代码段开头或者代码段结束部分。设有一个子程序 func1，直接摆放在调用语句之下，片段如下：

```
p1:    call   func1
func1  proc
       ..........
       ret
func1  endp
p2: ..........
```

当执行 p1 处的 call 指令时，会将断点地址即子程序 func1 的第 1 条指令的地址压栈（func1 的入口地址，断点），同时将子程序 func1 的地址送给 EIP，下面就会转到子程序开始执行。当执行到子程序中的 ret 指令时，从堆栈栈顶弹出一个双字送 EIP，即 func1 的入口地址再次送给 EIP，子程序会被再次执行。当然这之后执行到 ret 时，再弹出一个双字送给 EIP。程序的运行逻辑出现不可控的情况。

2、子程序中堆栈的使用问题

要想执行子程序后，能正确返回调用子程序的断点处，在子程序中就要注意堆栈的使用。计算机是非常严格的按照指令规定的功能执行程序的。例如在执行 ret 指令时，就一定是从当前的栈顶弹出一个双字内容送给 EIP，即完成“([esp])->EIP”和“(esp)+4->esp)”的操作。

若执行 ret 前的栈顶元素不是以前保存的断点地址，也就不能回到断点处继续执行。

一般在刚进入子程序时，可以写如下语句：

```
push    ebp
mov     ebp，esp
```

在子程序中保持 (ebp)不变，在 ret 之前，可以：

```
mov   esp，ebp
pop     ebp
```

这样可以保证执行 ret 时，(esp)指向的栈顶元素就是压入的断点地址。

8．3 子程序应用示例

8.3.1 字符串比较

【例 8.1】字符串比较，实现类似于 C 语言的 strcmp 函数。

函数 strcmp(char *str1, char *str2)中，str1 和 str2 分别是两个串的首地址，若两个字符串相等返回 0；若 str1 指向的串比 str2 指向的串小，则返回-1；若 str1 指向的串比 str2 指向的串大，则返回 1。

分析：采用堆栈传递参数，先将 str2 的首地址压入堆栈，再将 str1 的首地址压入堆栈。用 eax 来传递返回结果。(eax)=0 表示 str1==str2; (eax)=-1 表示 str1<str2; (eax)=1 表示 str1>str2。

两个字符串比较的算法不难，从首个对应字符开始比较，若对应字符不等，则根据这两个字符之间的大小关系，得到两个串的关系，串比较结束；若两个对应字符相等，则比较下一对字符，依此类推，用循环来实现该过程。当比较的字符对相等且为 0 时，说明串已比较结束，两个串相等，循环结束。

完整的程序如下。

```
.686P
.model flat, stdcall
   ExitProcess proto :dword
   includelib   kernel32.lib
   printf         proto c :ptr sbyte,    :vararg
   includelib   libcmt.lib
   includelib   legacy_stdio_definitions.lib
.data
   lpFmt     db   "%s > %s ?   %d (0,=;   -1, < ; 1, >) ", 0dh,0ah,0
   string1   db   'hello',0
   string2   db   'very good',0
   string3   db   'hello',0
.stack 200
.code
main   proc   c
     push   offset string2
     push   offset string1
     call     strcmp          ; 比较 string1 和 string2 开始的两个串
```

```
    add     esp, 8
    invoke printf,offset lpFmt, offset string1, offset string2, eax
    push    offset string3
    push    offset string1
    call    strcmp              ; 比较 string1 和 string3 开始的两个串
    add     esp, 8
    invoke printf,offset lpFmt, offset string1, offset string3, eax
    invoke ExitProcess, 0
main    endp

; 子程序  strcmp    str1    str2
; 功能：比较两个字符串 str1 和 str2 的大小关系
; 入口参数：两个串的首地址在堆栈中，str2 的首地址先入栈
; 出口参数：eax, 若前串小，则为(eax)=-1，若前串大为 1，相等为 0
; 算法思想：从串的最左端开始向右，逐一比较两个串对应的字符的关系。
; 若两个对应字符不相等则比较结束，由这两个字符的大小关系决定串的大小关系；
; 若两个对应字符相等并且不是串的结束，则继续向右比较；若是 0，则返回串相等。
; 寄存器分配：
;               edi, 指向 str1，即 (edi)为串 str1 中待比较字符的地址
;               esi, 指向 str2，即 (esi)为串 str2 中待比较字符的地址
;               dl, 用于缓存当前读取到的字符
strcmp    proc
      push    ebp
      mov     ebp, esp
      push    esi
      push    edi
      push    edx
      mov     edi, [ebp+8]    ; 第 1 个串的起始地址放入 edi 中
      mov     esi, [ebp+12]   ; 第 2 个串的起始地址放入 esi 中
strcmp_start:
      mov     dl,   [edi]
      cmp     dl,   [esi]
      ja      strcmp_large
      jb      strcmp_little
      cmp     dl, 0
      je      strcmp_equ
      inc     esi
      inc     edi
      jmp     strcmp_start
strcmp_large:
      mov     eax, 1
      jmp     strcmp_exit
strcmp_little:
      mov     eax, -1
      jmp     strcmp_exit
```

```
strcmp_equ:
        mov     eax, 0
strcmp_exit:
        pop     edx
        pop     edi
        pop     esi
        pop     ebp
        ret
strcmp  endp
end
```

8.3.2 数串转换

【例 8.2】 将一个给定的整数转换成指定基数的 ASCII 串，类似于实现一个 C 函数 itoa。

C 函数 itoa(int value, char *string, int radix)中， value 为一个待转换的数，string 是存放结果串的首地址，radix 是转化的基数，通常为 2、8、10、16 中的一个整数。注意，C 语言中还有一个无符号数转换为指定基数串的函数 ultoa(unsigned long value, char *string, int radix)。两者的核心内容是一样的。对于一个有符号数，先判断其是否小于 0，若小于 0，则先产生一个负号“-”，然后对该数进行求补，得到了其相反数的补码。将该数当成一个无符号数转换成指定进制的 ASCII 串即可。

一个无符号的二进制数转换为 P 进制数可采用“除 P 取余”法，其大致过程如下：

将待转换的二进制数除以 P，得到第一个商数和第一个余数，这第一个余数就是所求的 P 进制的个位数；将第一个商数除以 P，得到第二个商数和余数，这第二个余数就是所求 P 进制数的十位数；……；这一过程循环到商数为 0 时，所得到的余数就是所求 P 进制数的最高位数。

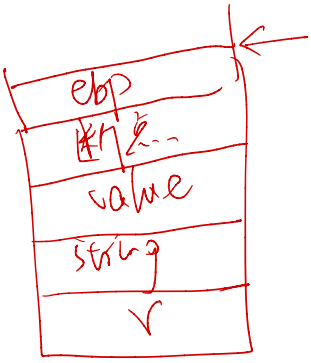
从上述“除 P 取余”过程可知，先得到的余数是 P 进制数的低位，后得到的余数是 P 进制数的高位，所以，可利用堆栈后进先出的原则，将每次除 P 所得余数入栈保存。当商数为 0 时，再将保存在栈中的余数逐一弹出，将其转换成 ASCII 码之后，送往 ASCII 码存储区。

从 itoa(int value,char *string,int radix)的调用形式可知，子程序从堆栈中获取三个参数。完整的程序如下。

```
.686P
.model flat, stdcall
ExitProcess proto :dword
includelib  kernel32.lib
printf      proto c :ptr sbyte, :vararg
includelib  libcmt.lib
includelib  legacy_stdio_definitions.lib
.data
lpFmt     db    "%d > %s", 0dh,0ah,0
value     dd    123
string    db    20 dup(0)
radix     dd    10
```

```
.stack   200
.code
main   proc  c
    push   radix
    push   offset string
    push   value
    call   itoa
    add    esp, 12
    invoke printf,offset lpFmt, value, offset  string
    invoke ExitProcess, 0
main   endp

; 子程序 itoa   value, string, radix
; 功能：将一个有符号数转换成指定基数的串，
;       结果存放在以 string 为首址的缓冲区中
; 入口参数：从右到左用堆栈传递的三个参数（value, string, radix）
; 出口参数：无
; 算法思想：除基数取余法
; 寄存器分配：
;           (edx，eax) 作为被除数, 在除法运算后，eax 存放商，edx 存放余数
;           (ebx) 除数
;           (ecx) 转换出的字符个数
;           (esi) 存放转换结果的地址
itoa  proc
    push  ebp
    mov   ebp, esp
    push  eax
    push  ebx
    push  ecx
    push  edx
    push  esi
    mov   eax, [ebp+8]     ; 待转换的数
    mov   esi, [ebp+12]     ; 存放结果的缓冲区地址
    mov   ebx, [ebp+16]    ; 基数
    mov   ecx, 0           ; 转换出的字符个数
    cmp   eax, 0
    jge   itoa_unsigned
    neg   eax              ; 对负数，输出负号，并且转换成其相反数的补码
    mov   byte ptr [esi], '-'
    inc   esi
itoa_unsigned:       ; 下面是对一个无符号数(eax)转换
```



```
        mov     edx, 0
        div     ebx
        push    edx         ; 保存余数，最先计算出来的是最右端的数
        inc     ecx
        cmp     eax, 0
        jne     itoa_unsigned
    itoa_save:      ; 下面将堆栈中记录的各数取出，转换成 ASCII，并且送到缓冲区中
                    ;  ecx 的初值肯定是大于 0 的。
                    ;  即使对数值 0，也有一个数码 0 放入堆栈中了。
        pop     edx
        cmp     dl, 10
        jb      itoa_convert
        add     dl, 7     ; 对于 0-9 之间的数码直接加 30H，变成对应的 ASCII
                          ; 对于 A-F 之间的数码，要加 37H ，即  数码-10+'A'(41H)
    itoa_convert:
        add     dl,  30H
        mov     [esi],dl
        inc     esi
        dec     ecx
        jnz    itoa_save
        pop    esi
        pop    edx
        pop    ecx
        pop    ebx
        pop    eax
        pop    ebp
        ret
itoa    endp
end
```

8.3.3 串数转换

【例 8.3】 串数转换，将含有正负号的数字 ASCII 串转换为一个整型数。

类似于实现一个 C 函数 int atoi(const char *str); 其中 str 是待转换串的首地址，返回值是转换后的整型数。对于一个有符号数，先判断其是否小于 0，若小于 0，则先产生一个负号“-”，然后对该数进行求补，得到了其相反数的补码。将该数当成一个无符号数转换成指定进制的 ASCII 串即可。

对于一个无符号的数码 ASCII 串，例如,“123”，其对应的存储信息是“31H　32H　33H　00”。串以值为 0 的字节结束。

设转换结果在 eax 中，eax 的初值为 0。从左向右依次扫描各个字符，采用下面的方法进行处理：(eax)*10+当前读到的数 -> eax。

当读到第一个非 0 的字节时，计算 (eax)*10+第一个字节内容-30H ，此时(eax)=1；

当读到第二个非 0 的字节时，计算 (eax)*10+第二个字节内容-30H ，此时(eax)=1*10+2=0CH；

当读到第三个非 0 的字节时，计算 (eax)*10+第二个字节内容-30H ，此时(eax)=0CH*10+3=7BH；

当读到第四个字节时，其值为 0，循环结束。

换句话说，转换的数为： ((0*10+1)*10+2)*10+3。

最后，判断数字串前面的符号，若是加号或者无符号，则结果就是前面转换的无符号数；如果该数前面有负号，则将结果求补，得到其相反数的补码表示即是最后所求的结果。

```
.686P
.model flat, stdcall
 ExitProcess proto  :dword
 includelib  kernel32.lib
 printf      proto c :ptr sbyte, :vararg
 includelib  libcmt.lib
 includelib  legacy_stdio_definitions.lib
.data
 lpFmt    db  "%s > %d", 0dh,0ah,0
 value    dd   0
 string   db   '-123',0
.stack    200
.code
 main  proc  c
   push   offset string
   call   atoi
   add    esp, 4
   mov    value, eax
   invoke printf,offset lpFmt, offset  string, value
   invoke ExitProcess, 0
main  endp
; 子程序 atoi  string
; 功能：将一个含有正、负号的数字 ASCII 串转换为一个整型数
; 入口参数：待转换串的首地址
; 出口参数：eax
; 算法思想：循环对各数码处理，当读到一个新数码时，将前面转换的结果乘 10 然后再加当前数码
; 寄存器分配：
;           eax 存放转换结果
;           ebx 正、负号信息，ebx=1，表示有负号，为 0，则表示为正
;           edx 当前读到的数码
;           esi 当前数码的地址
```

```
atoi  proc
    push  ebp
    mov   ebp, esp
    push  ebx
    push  edx
    push  esi
    mov   esi, [ebp+8]    ; 待转换串的首地址
    mov   eax, 0
    mov   ebx, 0
    cmp   byte ptr [esi], '-'
    jnz   atoi_plus_judge
    mov   ebx,  1
    inc   esi
    jmp   atoi_convert
atoi_plus_judge:
    cmp   byte ptr [esi], '+'
    jnz   atoi_convert
    inc   esi
atoi_convert:
    mov   dl, [esi]
    cmp   dl, 0
    je    atoi_convert_over
    sub   dl, 30H
    movzx edx,dl
    imul  eax, 10
    add   eax, edx
    inc   esi
    jmp   atoi_convert
atoi_convert_over:
    cmp   ebx, 1
    jnz   atoi_over
    neg   eax
atoi_over:
    pop   esi
    pop   edx
    pop   ebx
    pop   ebp
    ret
atoi endp
end
```

在本例中假设串中都是 0-9 的数码，未考虑含有非数码的情况。

8.3.4 自我修改返回地址的子程序

下面给出一个有意不回到断点处执行的一个趣味程序。

【例 8.4】 直接显示在 call 语句之下的字符串

设调用形式如下：

```
call  display
msg1 db 'Very Good',0DH,0AH,0
```

在显示串’Very Good’之后，继续执行 msg1 之下的语句。

```
.686P
.model flat, stdcall
  ExitProcess proto :dword
  includelib  kernel32.lib
  putchar     proto c :byte   ;  显示给定 ASCII 对应的字符
  includelib  libcmt.lib
  includelib  legacy_stdio_definitions.lib
.stack 200
.code
 main  proc  c
   call  display
   msg1  db 'Very Good',0DH,0AH,0 → 断点 但返回时
   call  display
   msg2  db '12345',0DH,0AH,0
   invoke ExitProcess, 0
main  endp
; ____________________________________
; 子程序：显示一个字符串
display proc
   pop  ebx
p1:
   cmp  byte ptr [ebx],0
   je   exit
   invoke  putchar, byte ptr [ebx]
   inc  ebx
   jmp  p1
exit:
   inc  ebx
   push ebx
   ret
display endp
end
```

该程序运行后，在两行上显示了两个串 Very Good 和 12345。下面通过反汇编来分析该程

序的运行过程。反汇编窗口显示的信息如下。

```
008A2040 E8 20 00 00 00      call        display (08A2065h)
008A2045 56                  push        esi
008A2046 65 72 79            jb          _display@0+5Dh (08A20C2h)
008A2049 20 47 6F            and         byte ptr [edi+6Fh],al
008A204C 6F                  outs        dx,dword ptr [esi]
008A204D 64 0D 0A 00 E8 0F   or          eax,0FE8000Ah
008A2053 00 00               add         byte ptr [eax],al
008A2055 00 31               add         byte ptr [ecx],dh
008A2057 32 33               xor         dh,byte ptr [ebx]
008A2059 34 35               xor         al,35h
008A205B 0D 0A 00 6A 00      or          eax,6A000Ah
008A2060 E8 A0 EF FF FF      call        _ExitProcess@4 (08A1005h)
--- display_string.asm --------------------
008A2065 5B                  pop         ebx
p1:  cmp  byte ptr [ebx],0
008A2066 80 3B 00            cmp         byte ptr [ebx],0
     je   exit
......
```

在反汇编代码中首先看到的是“call display”对应的代码“call display (08A2065h)”。在反汇编程序中可以看到 “08A2065h 5B pop ebx”，08A2065h 正是子程序的入口地址，非常直观。不过，如果仔细的看一下这条指令的机器码，就会发现它的机器码是“E8 20 00 00 00”，这和 08A2065h 毫无关系，这是怎么回事？

要解释机器码的内容，还得从计算机执行指令的过程说起。当程序开始执行时，(EIP)= 008A2040H， 即指向第 1 条指令。在根据 EIP 取出指令并译码后，EIP 就会加上本条指令的长度。本条指令占 5 个字节，故取出第 1 条指令后(EIP)= 008A2045H。执行 call 指令，先将(EIP)压栈，这样压入堆栈的内容正好是断点地址，之后(EIP)+本指令中的偏移量（即 00000020H)->EIP，因此新的 EIP 为 08A2065h。在程序编译生成机器指令的时候，不会在指令中固定一个单元的物理地址，因为只有调度执行程序时才会由操作系统决定空间的分配。当前编译的指令在本程序空间中有一个相对地址，而转移的目的地也有一个相对地址，在转移指令中就存放两个地址之间的相对位移量（减去本指令的长度）即可。

在汇编窗口的第一条指令之后看到的汇编语句并不是源程序中的语句。看一下这些指令的机器码，它们是“56 65 72 79 20 47 6F 6F 64 0D 0A 00 ”，这实际上是“’Very Good’,0DH,0AH,0”对应的内容，前面几个字节是字符串的 ASCII，后面三个字节是“0DH, 0AH,0”。这些二进制数也被当成机器码解析了，在反汇编窗口显示了它们对应的语句。这就说明 0-1 串的解析是依赖上下文的，本来存储的是一个字符串，但在调试窗口，从当前的 EIP 开始，将代码段的内容当成指令解析，就得到窗口中看到的结果。

跟踪进入子程序，此时观察堆栈，可看到栈顶元素是 008A2045h，这是执行 call 时，CPU 自动压入堆栈的断点地址，也就是字符串 msg1 的首地址。子程序中首先执行的是 pop ebx，(ebx)为串的首地址，之后是从该地址开始逐个取出字符显示，直到遇到字节 0。在循环结束后，(ebx)指向的是字节 0 的地址。之后，将(ebx)加 1 并压入堆栈，此时栈顶的内容正好是第二条

call 指令的起始地址。执行 ret 后，可以看到如下的反汇编代码。

```
008A2051 E8 0F 00 00 00       call        display (08A2065h)
008A2056 31 32                xor         dword ptr [edx],esi
008A2058 33 34 35 0D 0A 00 6A xor         esi,dword ptr [esi+6A000A0Dh]
008A205F 00 E8                add         al,ch
```

对第 2 条 call display 指令的分析与对第 1 条 call display 指令的分析是相似的，请读者自己分析其机器码、执行过程。

对上面的例子稍加改造，可以得到一个用机器语言编写的程序。主程序的片段如下。

```
main  proc  c
   db   0E8H,20H,0,0,0
   db  'Very Good',0DH,0AH,0
   db   0E8H,0FH,0,0,0
   db  '12345',0DH,0AH,0
   invoke ExitProcess, 0
```

注意，在本例中，我们没有将“invoke ExitProcess,0”换成机器语言编码。对于 8.4 中的例子的反汇编，可以看到“invoke ExitProcess, 0”对应的是：

```
008A205E 6A 00                push        0
008A2060 E8 A0 EF FF FF       call        _ExitProcess@4 (08A1005h)
```

从 call 指令的机器码中看到的位移量为 FFFFEFA0H，它对应的是 -1060H 的补码表示。call 指令之下的地址是 008A2065h，由 008A2065h－1060h = 008A1005h，08A1005h 正是在反汇编窗口中看到的信息。

如果直接用机器码写上段程序，运行并不成功。原因是：ExitProcess 是一个外部函数，如在程序中不出现它的名字，则编译器不会将相应的函数实现段链接到程序中。换句话说，调转到 08A1005h，找不到 ExitProcess 的程序。此外，外部库函数的存放位置也是不能由用户编写程序来控制的。

当然，介绍上述内容的目的并不是要读者去用机器码去写程序，而是让读者更好的理解计算机内部是 0-1 串世界的本质、计算机根据 EIP 取相应单元的内容并解释执行、数据和指令是一种动态变化的等理念。

8.3.5 自我修改的子程序

在 8.3.4 节给出了一个直接使用机器码编写程序的例子。更进一步在程序的运行过程中，程序的代码也可以自我修改。

【例 8.5】 在程序中，将数据段的一段数据拷贝到代码段，让程序运行这段数据对应的代码。

```
.686P
.model flat, stdcall
  ExitProcess proto :dword
  VirtualProtect proto :dword, :dword,:dword,:dword
  includelib  kernel32.lib
  putchar     proto c :byte   ;  显示给定 ASCII 对应的字符
```

```
  includelib  libcmt.lib
  includelib  legacy_stdio_definitions.lib
.stack 200
.data
 machine_code db 0E8H,20H,0,0,0
              db 'Very Good',0DH,0AH,0
               db   0E8H,0FH,0,0,0
               db '12345',0DH,0AH,0
 len = $-machine_code
 oldprotect   dd  ?
.code
 main   proc  c
    mov  eax, len
    mov  ebx, 40H
    lea  ecx, CopyHere
     invoke VirtualProtect,ecx,eax,ebx,offset oldprotect
     mov  ecx,len
      mov  edi,offset CopyHere
     mov  esi,offset machine_code
CopyCode:
     mov  al, [esi]
     mov  [edi],al
      inc  esi
     inc  edi
     loop CopyCode
CopyHere:
   db  len dup(0)
   invoke ExitProcess, 0
main  endp
; ---------------------------------------
; 子程序：显示一个字符串
display proc
   pop  ebx
lp:
   cmp  byte ptr [ebx],0
   je   exit
   invoke  putchar, byte ptr [ebx]
   inc  ebx
   jmp  lp
exit:
   inc  ebx
```

```
        push ebx
        ret
    display endp
end
```

该程序先将数据段的一些内容拷贝到了代码段中，使得修改后的代码段与程序执行前的代码段不同。在程序运行时显示 Very Good 和 12345 两个串。由于 Windows 系统中，代码段是受保护的，不能随意更改，因而使用了 Windows 系统提供的 API 函数 VirtualProtect 来改变调用进程的一段页的保护属性。有兴趣的读者可以通过其他资料更多的了解函数 VirtualProtect。

8．4　C 程序中函数的运行机理

C 语言中的函数通常定义有形式参数，用于接收从调用函数传递过来的信息。在函数中，也会定义局部变量，其作用域在函数内部。本节从机器语言的视角探讨参数是如何传递的，局部变量的空间、参数的空间是如何分配和释放的，函数运行结果如何返回。理解了这些内容，对于编写正确的 C 语言程序也是很有帮助的，同时也有助于熟练地用汇编语言编写子程序。

【例 8.6】 从机器语言角度分析 C 语言程序的实现细节。

对于如下 C 语言程序，通过观察反汇编程序、堆栈段、寄存器、变量地址和存储单元内容等手段，可以更深入了理解它的运行机理。

```
#include <stdio.h>
int fadd(int x, int y)
{
    int u, v, w;
    u = x + 10;
    v = y + 25;
    w = u + v;
    return w;
}
int main(int argc, char* argv[])
{
    int   a = 100;      // 0x 64
    int   b = 200;      // 0x C8
    int   sum = 0;
    sum = fadd(a, b);
    printf("%d\n", sum);
    return 0;
}
```

① 参数的传递

在主程序中，有 sum=fadd(a,b);该语句反汇编的结果如下。

```
00E11668   mov        eax,dword ptr [b]
00E1166B   push       eax
00E1166C   mov        ecx,dword ptr [a]
00E1166F   push       ecx
00E11670   call       _fadd (0E11217h)
```

```
00E11675    add             esp,8
00E11678    mov              dword ptr [sum],eax
```

从此段反汇编语句可以看到：函数调用时，先将参数 b 的值(即 200)压入了堆栈，再将参数 a 的值(即 100)压入了堆栈，压到堆栈中的内容是变量 a 和 b 的值，与它们的地址无关。在调用子程序指令之下，有“add esp,8”用于清除这两个参数所占的空间。

在执行 call 指令时，跟踪进入子程序时，观察以(esp)为起始的内存段的信息，可看到：

75 16 e1 00 64 00 00 00 c8 00 00 00

3 个双字数据分别对应断点地址 00E11675h、第 1 个参数值 100、第 2 个参数值 200。

② 函数体语句之前的汇编代码

进入函数后，在第 1 条变量定义语句之前，有一段如下的反汇编代码：

```
00E115F0    push            ebp
00E115F1    mov             ebp,esp
00E115F3    sub             esp,4Ch
00E115F6    push            ebx
00E115F7    push            esi
00E115F8    push            edi
00E115F9    mov             ecx,offset _DC9CF1C2_c_function@c (0E19003h)
00E115FE    call            @__CheckForDebuggerJustMyCode@4 (0E1119Fh)
```

这一段代码是编译器自动增加的代码，并不是函数中的哪条语句的对应产物。注意，不同的编译开关生成的代码是不同的。

push ebp　和　mov ebp, esp　的作用是在堆栈在保存了原(ebp)，同时让 ebp 指向了当前栈顶。之后，“sub　esp, 4Ch”直接移动了栈顶指针，留出了一部分堆栈空间，这一片空间是分配给程序的局部变量的。

③ 函数参数和局部变量的空间分配

函数体语句对应的反汇编代码如下。

```
int u, v, w;
u = x + 10;
00E11603    mov             eax,dword ptr [ebp+8]
00E11606    add              eax,0Ah
00E11609    mov             dword ptr [ebp-4],eax
v = y + 25;
00E1160C    mov              eax,dword ptr [ebp+0Ch]
00E1160F    add             eax,19h
00E11612    mov              dword ptr [ebp-8],eax
w = u + v;
00E11615    mov              eax,dword ptr [ebp-4]
00E11618    add             eax,dword ptr [ebp-8]
00E1161B    mov              dword ptr [ebp-0Ch],eax
```

从这段代码可以看到，参数 x 对应的地址是 [ebp+8]，参数 y 对应的地址是 [ebp+0Ch]，这正是函数调用时，传递的两个值在堆栈中的存放位置。从代码中也可以看到局部变量 u, v, w 的地址分别是[ebp-4]、[ebp-8]、[ebp-0Ch]，它们是在堆栈中分配的空间。两个相邻的局部变量之间距离是 4 字节。由于变量 u 定义在最开头，其地址为[ebp-4]，之后的变量继续在堆栈的上方分配空间。

注意：在生成上述代码时，需要将配置属性“C/C++ -> 代码生成->基本运行时检查”改

为“默认值”。用不同的配置项生成的代码是不同的，变量也不一定“紧凑”的放在一起。

④ 结果的返回

对于 return 语句，翻译的结果如下：

```
return w;
00E1161E  mov         eax,dword ptr [ebp-0Ch]
00E11621  pop         edi
00E11622  pop         esi
00E11623  pop         ebx
00E11624  mov         esp,ebp
00E11626  pop         ebp
00E11627  ret
```

最后返回变量的值放在了 eax 中。在主程序中，函数调用生成的语句也是将 eax 中的值传送给接受返回值的变量。

当 ret 之前，恢复了函数开头自动保存的寄存器，又将 esp 还原成了进入函数时的状态，这样，局部变量所在的空间又会分配给新的函数使用，其生命周期终止，相当于它们所占的空间自动释放。

通过上述例子的分析，可以得到如下结论和写程序时应注意的问题。

① 函数参数与函数的局部变量一样，它们的空间分配都在堆栈上；

② 以刚进入子程序时的堆栈状态为参考，函数参数所在单元的地址是在堆栈栈顶之下（栈顶是执行 call 指令时压入的断点地址）；该函数的局部变量的空间是在当前堆栈栈顶之上（地址更小一些）；

③ 函数参数和变量的地址都是 [ebp+n]的形式，是一种变址寻址；对于参数，其地址表达式中的 n 为正数，对于局部变量，n 为负数；

④ ebp 在函数中保持不变，因而[ebp+n]才会固定的对应某个参数和变量的地址；在刚进入函数时，要保护 ebp,然后将(esp)赋值给 ebp，由此状态可以算出变量和参数的地址；

⑤ 调用函数时参数压栈，参数的地址与调用函数时的实参的地址无关；对于传值参数，函数参数所在的单元中的内容就是传递进来的值，此时改变函数参数所在单元的值，与外部的实参变量无关；对于指针类型的参数，压入堆栈的是变量的地址；在函数中通过参数可以间接访问到实参变量中的内容，可实现对实参变量中内容的修改；

⑥ 函数返回值放在 eax 中；注意，在函数调用中，函数返回一个结构或者面向对象的程序设计中返回一个对象时，编译器将会生成更复杂的代码，来实现数据的传递；

⑦ 不要返回函数中局部变量和参数的地址，局部变量和参数的空间在堆栈中，函数返回后，这些空间在调用下一个函数时又会被使用，原单元中的内容会被覆盖，根据返回的地址去访问对应的单元时，内容就并非原来的值；

⑧ 破坏堆栈中存放的断点地址，会导致程序出现异常。例如，在函数 fadd 返回前增加一条语句：*(int *)(&x - 1) = 1; 就会破坏断点地址导致执行 ret 指令时，EIP 被赋给了一个超出本程序中间范围的地址（00000001），程序运行崩溃。

8．5 汇编语言中子程序的高级用法

在汇编语言的子程序设计中，汇编语言的编译器也支持实现了一些伪指令，包括带参数的

函数说明、带参数的子程序调用和定义局部变量。使用这些伪指令，可以简化程序的编写工作，提高程序的可读性，也可以避免一些参数传递时易犯的一些错误。使用这些高级用法，写汇编语言程序就有点类似于用 C 语言写程序了。

再次强调伪指令不是机器指令，编译器会对其进行处理，生成相应的机器语言程序。这种方法减轻了程序员的劳动强度，但是也掩盖了计算机内部工作的一些机制。读者可以通过源程序与汇编语言程序的对比，就编译和计算机内部的工作机制有更好的理解。

8.5.1 局部变量的定义和使用

紧跟在 proc 语句之后，可以用 local 伪指令说明仅在本函数内使用的局部变量。

格式：**local　变量名[[数量]][:类型]**

括号中的“数量”用于说明重复单元的数量，类似 dup 的效果。类型可以是基本类型名字，如 byte、word、dword、qword、sbyte、sword、sdword、sqword 等等。

在使用局部变量时，若只出现单个单量，用法还是很简单的。下面给出一个局部变量定义和用法的示例。

【例 8.7】 子程序中局部变量和参数的用法示例。

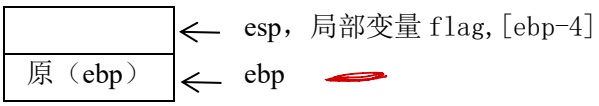
子程序 func 有一个参数，子程序的功能是获取该参数，并送到局部变量 flag 中。汇编源程序片段如下：

```
    push   100
    call   func
    add    esp, 4
    ……
func proc
    local flag:dword
    push   ebx
    mov    ebx, [ebp+8]
    mov    flag,ebx
    pop    ebx
    ret
func endp
```

通过反汇编观察含有局部变量的子程序编译后的结果，可以看到编译后在子程序开始自动加上以下语句：

```
    push   ebp
    mov    ebp,esp
    add    esp, 0FFFFFFFCH ;  等价于 add esp, -4 或者 sub esp, 4
```

前 2 条语句“push ebp 和 mov ebp，esp”是一个固定模式。第三条语句中 esp 的减少的值与局部变量所占空间的大小有关，即 esp 向栈的上方移动，留出了局部变量所占的空间。执行前三条语句后，堆栈状态如图 8.3 所示。



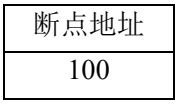


图 8.3 堆栈示意图

从图 8.3 中可以看到，参数所在的单元的地址是 [ebp+8]，局部变量 flag 所在的单元的地址是 [ebp-4]。

注意：当子程序中定义有局部变量，或者子程序有参数（参见 8.5.2 节）时，编译器才会增加语句“push ebp”和“mov ebp,esp”,并用[ebp±n]表示参数和局部变量的地址。

局部变量所指的单元在堆栈中。虽然在后续的指令语句中可以直接引用局部变量的名字，就像使用在 data 段中定义的全局变量那样，但是两者还是有些差异需要注意。

(1) 单个局部变量作为源操作数或目的操作数，对应的寻址方式是变址寻址方式，其地址是[ebp-n]；其中 n 为正数。在机器指令中，-n 表示为 n 的补码。一般先定义的局部变量的地址大一些（减去的 n 小），后定义的变量地址小（减去的 n 大）；单个全局变量对应的寻址是直接寻址；

(2) 可以将 offset 放在全局变量前，获取它的地址，即得到一个数值，但是不能将 offset 放在局部变量前；可以用 lea 指令来获取局部变量的地址；

(3) 对于定义数组型的局部变量，例如 local x[20]:dword,不能使用 x[BR+IR*F] 的形式访问，因为 x 本身就是对于[ebp+n]；而对于全局变量 x，可以用 x[BR+IR*F]的形式访问。对于局部变量 x，x[IR*F]在机器指令中对应的是 [IR*F+ebp+n],是一个基址加变址的寻址方式。

特别提示：

(1) 有些编译器对 x[BR+IR*F]编译时不报错，但生成的机器指令有一些变化，出现非预期的结果。

(2) 在定义局部变量或者参数时，编译器会自动在原代码前加上几条机器指令,ebp 已赋予了特定的作用，不要写改变 ebp 值的指令。

注意，proto 和 proc 伪指令语句中都没有说明返回参数。若某函数具有返回值，则按照一般的约定，将四个字节的返回值存放在 eax 寄存器中；两个字节的返回值存放在 ax 中；一个字节的返回值存放在 al 中。 若某函数具有多个返回参数，则应在 proto 和 proc 语句中增加指针类型的参数，使函数处理的结果能通过指针存放到调用者的程序空间中，达到将多个参数返回的目的。

8.5.2 子程序的原型说明、定义和调用

① 原型说明伪指令 proto

格式：函数名 proto [函数类型][语言类型][[参数名]:参数类型],[[参数名]:参数类型]...

功能：用于说明本模块中要调用的过程或函数。

“函数类型”指明了该子程序的类型，可以是 NEAR、FAR 中的一种。在 32 位段扁平内存管理模式下,即存储模型说明为“.model flat”，应该选择 NEAR，默认值也是 NEAR。

“语言类型”指明参数传递的方式采用哪种语言的规定。在 32 位段扁平内存管理模式下，支持 c 和 stdcall，在 16 位内存管理模型下，支持 c、pascal、stdcall、syscall 等。如果在存储模型说明中，指明了语言类型，例如“.model flat, stdcall”，且本函数使用的语言类

型于模型说明一致，则可用缺省值，如果本函数的语言类型不一致，则可显示给出本函数的语言类型。c 和 stdcall 都是将函数原型说明中的最右边的参数最先入栈，最左边的参数最后入栈，区别在于 c 是在调用程序中清除参数所占的空间，而 stdcall 是在函数中清除堆栈所占的空间。

在原型说明中，可以省掉参数的名称。这与 C 语言在函数说明时的规定是一致的。

proto 伪指令的功能类似于 C 语言中的函数说明。对于在本程序中调用的其他程序定义的函数，应该使用 proto 进行原型说明。只有这样，编译的时候才能确定传递参数的大小，例如一个参数值是 0，若没有该参数的原型说明，就不知道是压一个字还是一个双字的 0 进栈。对于在本程序中定义的函数，若是先定义后调用，则可以不使用 proto 进行原型说明。如果用 invoke 伪指令调用该函数之前既没有用 proto 语句进行说明，又没有用完整的 proc 语句预先对该函数进行定义，那么，汇编程序进行语法检查时就会报错。

② 子程序的完整定义

格式：**函数名 proc [函数类型][语言类型][uses 寄存器表][,参数名[:类型]]...**

功能：定义一个新的函数（函数体应紧跟其后）。

利用 proc 伪指令对函数进行完整定义的格式和 proto 的格式非常相似。其中，“函数类型”、“语言类型”和各参数应与对应的 proto 说明一致。此处的参数名是用来传递数据的，可以在程序中直接引用，故不能省略。类型有：byte、sbyte、word、sword、dword、sdword、等等。对 32 位段，类型省略表示是“:dword”，对 16 位段，当参数类型为字时，“:word”可以省略。uses 后面所列的寄存器是需要在子程序中入栈保护的（由汇编程序自动生成入栈保护和出栈恢复的指令语句）。此外，proc 语句中还可以使用表明本函数是否能被外模块调用的可见性（Visibility）选项以及规定函数开始和结束处的隐含处理方法（如堆栈处理等）的选项，有兴趣的读者可以参考宏汇编手册，这里不再介绍。各部分的分隔符分别是空格或逗号。

③ 函数调用伪指令 invoke

格式：**invoke 函数名 [,参数]...**

功能：调用由完整的函数定义伪指令 proc 定义的函数。其中，参数可以是各种表达式。

在前面的程序示例中，已多次出现了 invoke 伪指令，如在调用 printf 函数显示信息时就用的 invoke。

invoke 与 call 都完成了子程序的调用，但 invoke 可直接在函数名后加上传递的参数，因而使用起来比 call 简单。但是，站在机器语言的角度，invoke 是伪指令，在生成的执行程序中是看不到 invoke 的。编译器在堆汇编源程序进行编译时，将 invoke 伪指令转换成相关的基本汇编指令序列，包括参数中表达式的计算、参数的入栈操作、call 语句、堆栈栈顶指针复原等。当然，不使用 invoke 伪指令，程序员可自己编写语句完成参数入栈与使用、call 子程序、堆栈空间恢复等操作。

8.5.3 子程序的高级用法举例

【例 8.8】 对于例 8.6 所示的 C 语言程序，按照子程序的高级用法写出汇编源程序。

```
.686P
.model flat, c
  ExitProcess proto stdcall :dword
  includelib  kernel32.lib
```

```
    includelib  libcmt.lib
    includelib  legacy_stdio_definitions.lib
    printf      proto :ptr sbyte, :vararg
.data
    lpFmt    db   "%d ", 0dh,0ah,0
.stack 200
.code
 myfadd  proc   x:dword, y:dword
        local u:dword,v:dword,w:dword
        mov   eax, x
        add   eax, 10
        mov   u,   eax     ; u=x+10;
        mov   eax, y
        add   eax, 25
        mov   v,   eax     ; v=y+25;
        add   eax, u
        mov   w,   eax     ; w=u+v;
        ret
myfadd   endp
main proc
     local  a:dword,b:dword,sum:dword
     mov   a, 100
     mov   b, 200
    invoke myfadd, a, b
     mov   sum, eax
     invoke printf,offset lpFmt, sum
     invoke ExitProcess, 0
    ret
main  endp
end
```

在上面的例子中，为了在主程序中也能定义局部整型变量 a、b 和 sum，需要将主程序段写在子程序段中。我们将该主子程序命名为 main。在程序结束处有语句“end main”，指明程序是 main 的起始处开始执行。

在定义局部变量时，可以将几个变量的定义写在一行，也可以写成多行。例如“local u:dword,v:dword,w:dword”等价于：

```
local  u:dword
local  v:dword
local  w:dword
```

注意，子程序的命名不能够起名为 fadd。在 x86 中，fadd 是一条机器指令，实现浮点数的加法运算。在汇编语言程序中，变量、标号、子程序等的名字不应与机器指令的名字相同。

由于是先定义函数 myfadd 而后用 invoke 调用，故省掉了函数的原型说明。可以在 data 段

之前加上函数原型说明，如 “myfadd proto :dword, :dword”。

如果对上述程序进行反汇编，可以看到各条伪指令对应的机器指令，以及参数变量、局部变量的空间分配和访问的方法。细心的读者会对于 ret 指令的编译结果为：

```
leave
ret
```

这里出现了一条陌生的指令“leave”。指令 leave 与如下两条指令的作用是等效的。

```
mov esp, ebp
pop ebp
```

本节给出了一个简单而完整的使用局部变量和参数的应用示例。采用 invoke 伪指令编写汇编语言程序有些类似于用 C 语言编写程序。希望读者在会使用这些高级用法的同时，能够站在机器语言程序的视角理解程序的运行机理。

8．6 递归子程序的设计

在数学中很多问题的求解可以用递归的方式表达。例如，求一个正整数 n 的阶乘 f(n)，可以使用如下表达式：

$$f(n) = \begin{cases} 1 & n = 1 \\ n * f(n-1) & n > 1 \end{cases}$$

用 C 语言实现该功能的程序如下：

```
#include <stdio.h>
int   jiecheng(int x)
{       int t;
        if (x ==1)   return 1;
        t = jiecheng(x - 1);
        t = t*x;
        return t;
}
int   main(int argc, char* argv[])
{
        int   a = 3;
        int   b;
        b = jiecheng(a);
        printf("%d\n",b);
        return 0;
}
```

【例 8.9】 求一个正整数 n 的阶乘

用汇编语言编写用递归方法实现求阶乘功能的程序如下。

```
.686
.model flat, stdcall
   ExitProcess proto :dword
   includelib   kernel32.lib
   printf        proto c :ptr sbyte, :vararg
   includelib   libcmt.lib
```

```
        includelib  legacy_stdio_definitions.lib
.data
lpFmt   db      " %d ", 0dh, 0ah,0
.stack 200
.code
jiecheng    proc
        local   t: dword
        cmp     dword ptr [ebp+8], 1
        jne     lp
        mov     eax, 1
        ret     4
lp:
        mov     eax, [ebp+8]
        dec     eax
        push    eax
        call    jiecheng
DK:     mov     t,   eax
        imul    eax, [ebp+8]
        ret     4
jiecheng endp
main    proc    c   ; 主程序
        push    3
        call    jiecheng
        invoke printf,offset lpFmt, eax
        invoke ExitProcess, 0
main    endp
end
```

递归调用的子程序的局部变量以及参数每次运行都在不同的空间上。

第一次调用子程序	第二次进入子程序	第三次进入子程序
		主程序断点地址 DK ← esp
		1
		局部变量 t
		(ebp)2
	主程序断点地址 DK ← esp	主程序断点地址 DK
	2	2
	局部变量 t	局部变量 t
	(ebp)1	(ebp)1
主程序断点地址 DK ← esp	主程序断点地址 DK	主程序断点地址 DK
3	3	3

图 8.5 堆栈变迁的示意图

整个程序的运行大致过程如图 8.4 所示，即依次执行①、②、…、⑩、A、B、C。

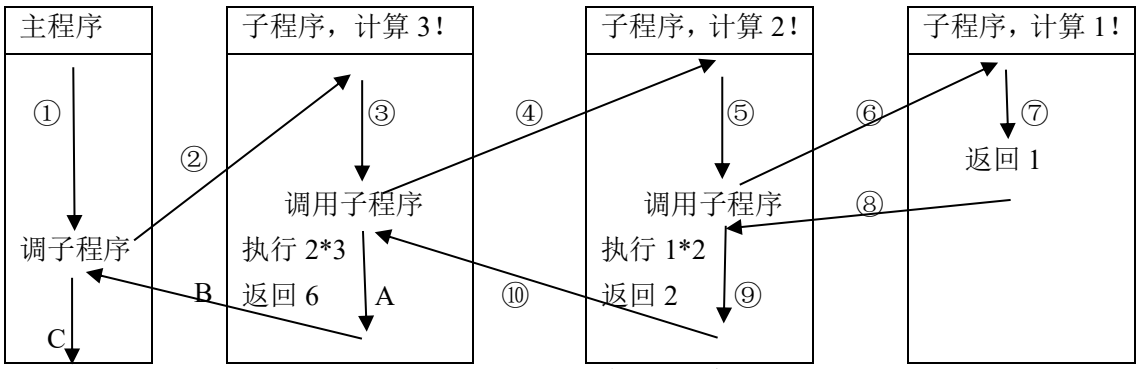


图 8.4 程序运行过程的示意图

注意，每次调用子程序时，子程序的代码所在的内存空间并没有变化，但为了方便理解，我们将子程序每次调用都“重新抄写”了一遍。在子程序的最上方，给出了进入子程序的入口参数，即计算哪一个数的阶乘。如果单纯从编写和理解程序的角度来看，在计算 3!时，要递归调用子程序计算 2!，可简单的认为被调用的子程序直接返回了希望的结果，即得到了 2!在 eax 中，而不用考虑它是如何实现的；也即将图 8.18 中的第④到第⑩视为一个整体。

下面使用调试的方法，观察程序执行的细节，特别是堆栈的变化情况。在图 8.5 中，给出了三次调用子程序时的堆栈变迁的示意图。

从程序中可以看到，每次进入子程序时，都先保存了当前的(ebp)，然后将(esp)赋值给ebp。从反汇编代码中可以看到局部变量 t 的地址是[ebp-4]。每当调用一次子程序时，都将局部变量 t 的空间分配在[ebp-4]的位置上，但(ebp)在每一次调用中都会是发生变化的，因而，递归调用的子程序的局部变量以及参数每次运行都在不同的空间上。

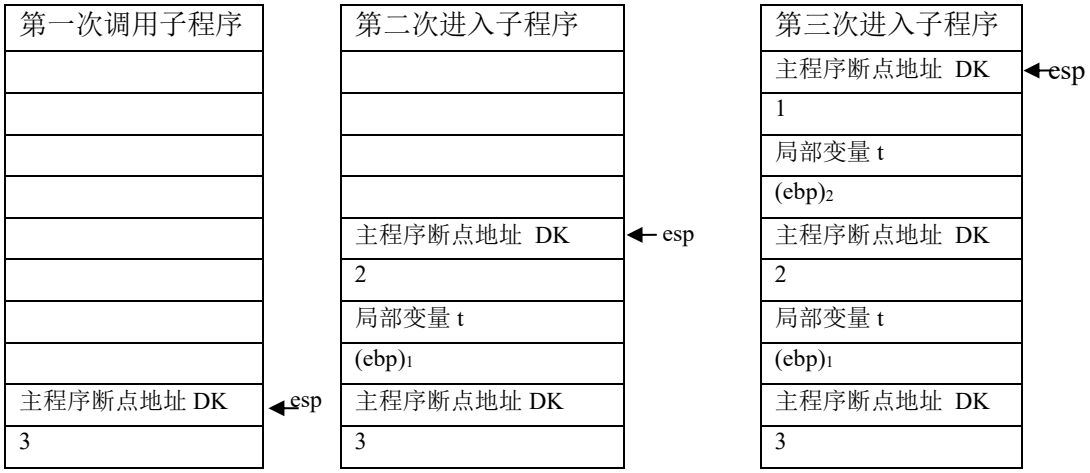


图 8.5 堆栈变迁的示意图

使用完整的子程序定义和调用伪指令 invoke，程序代码段如下，其他部分保持不变。

```
.code
jiecheng  proc  x:dword
   local  t: dword
   cmp    x, 1
   jne    lp
   mov    eax, 1
   ret
lp:
   mov    eax, x
   dec    eax
   invoke jiecheng, eax
   mov    t, eax
   imul   eax, x
   ret
jiecheng endp
main  proc  c
   invoke  jiecheng, 3
   invoke  printf,offset lpFmt, eax
   invoke  ExitProcess, 0
main  endp
```

由于在模型说明中使用了“.model flat, stdcall”，而在 jiecheng 的函数定义中未指明语言类型，故缺省使用 stdcall，表明要在子程序中清除参数所占的堆栈空间。由于子程序中有一个双字类型的参数，因此编译器对 ret 的编译结果是“leave”和“ret 4”。如果不清除子程序参数所占的空间，在递归程序调用时就不能正确的获得程序的断点地址，造成程序运行异常。如果将 jiecheng 定义改为“jiecheng proc c x:dword”，即指明语言类型为 C 语言类型，生成的代码就会不同。有兴趣的读者，可以使用反汇编的方法，观察高级写法对应的机器语言程序。

习题 8

8.1 子程序调用的形式有哪两种？

8.2 CPU 在执行 call 指令和 ret 指令时分别会完成哪些操作？

8.3 主程序和子程序之间有哪几种传递参数的方法？

8.4 在子程序调用时，如何实现寄存器中内容的保护和恢复？

8.5 编写一个子程序，实现将一个字符串拷贝的功能（类似 C 语言中的 strcpy），要求在子程序通过参数传递信息源串的首地址、目的缓冲区的首地址，不得使用子程序之外的变量。源字符串以字节 0 结束。

8.6 编写一个子程序，实现将一个缓冲区拷贝的功能（类似 C 语言中的 memcpy），要求在子程序通过参数传递信息源缓冲区的首地址、目的缓冲区的首地址、拷贝的字节数。子程序中不得全局变量。

8.7 编写一个子程序，实现对两个参数按指定的运算符参数进行运算的函数，返回运算结果。类似 C 语言函数：int myop(int x,int y,char op); 当 op=’+’时，返回值为 x+y。

8.8 在 8.3.4 节例 8.4 的子程序 display 中，在 ret 之前若漏写了 inc ebx，即最后一部分如下：

```
exit:
  ; inc  ebx
    push ebx
    ret
```

运行结果如何？

8.9 在 8.4 节例 8.4 的程序，若将子程序放到了最后一个语句“invoke ExitProcess, 0”之前，程序片段如下，运行结果会如何？

```
.....
call display
msg2 db '12345',0DH,0AH,0
display proc
    ......
display endp
invoke ExitProcess, 0
```

上机实践 8

8.1 设有如下 C 语言程序，运行该程序时，显示了 x=100，但是 *p 的值不是 100。试分析产生这一现象的原因。

```
#include <stdio.h>
int* f()
{
    int t;
    t = 100;
    return &t;
}
int main(int argc, char* argv[])
{
    int  *p;
   int  x;
    p = f();
     x = *p;
    printf("x=%d  *p=%d\n", x, *p);
    return 0;
}
```

8.2 设有如下 C 语言程序，运行该程序时程序崩溃 。分析为什么数组越界会导致程序崩溃？

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    int  a[10];
    int  i;
    for (i = 0; i < 20; i++)
        a[i] = i;
    return 0;
}
```