（在IBM PC机上）

C语言和汇编语言的混合高级编程

尹彦芝

北京科海培训中心

一九八九年

（在IBM PC机上）

C语言和汇编语言的混合高级编程

尹彦芝

北京科海培训中心

一九八九年

编辑：科海培训中心教材部
发行：科海培训中心资料组
地址：北京2725信箱 科海培训中心
资料组
（北京海淀路332路黄庄站旁）
电话：2562954
印刷：河北省蔚县印刷厂

前　言

本书以大量的实例系统地阐述了用C语言和汇编语言混合编写程序的方法和技巧。C语言容易编写，且效率也比较高；汇编语言不易编写，但它效率最高，也最适合于直接对机器硬件进行控制。把这两种语言混合起来编程，能够得到最优的程序质量，也是目前最流行的程序设计方法。

本书的正文共17章，大体上可分为四个部分。第一部分包括第1、2、3、4、16章，这部分主要讲述从C语言调用汇编语言或从汇编语言调用C语言的原则和方法。还讲述了一些优化C语言程序的基本原理和方法。每一个阅读本书的读者都应该读一读前几章，尤其是其中的第1章。第16章牵涉到许多DOS的内部数据结构，比较难懂，不熟悉PC DOS（MS DOS）的读者可以不读。第二部分包括第5、6、7、8、9章，这部分主要讲述如何调用IBM PC机的ROM BIOS子程序。对于IBM PC机的汇编语言程序员来说，ROM BIOS子程序是一个非常重要的资源，必须加以掌握。这几章以具体的例子介绍如何从C语言中调用显示、键盘、串行口等的ROM BIOS子程序，从而达到对这些常用设备的控制。第三部分包括第10、11、12、13、14章，这部分讲述如何从C语言中直接存取IBM PC机的硬件，如显示缓冲区、打印机、声音、时间、串行口等。第二部分和第三部分的每一章各讲一个或几个硬件设备，读者可以挑选他所感兴趣的各章去阅读，不必连续。第四部分包括第15、17章。第15章是介绍一个帮助程序员进行优化工作的工具软件。第17章具有总结的性质，介绍编写优化C程序（包括速度和长度）的方法和调试程序的方法。

本书最后的六个附表提供了程序员所经常使用的一些系统重要资源表格。

本书共提供了13个完整的可以直接使用的具体实例，其内容简介如下。这些例子的C语言部分全都是用PC机上最流行的Microsoft C写成的，除第16章中的例子以外，其余都只要求 Microsoft C编译4.0版就可以了。这些例子中的汇编语言部分全都是用最流行的MASM写成的。

RAMsort：　第2章，一个对文本文件以行为单位按数字字母为序进行排序的程序。

Tic Tac：　第3章，一个和计算机下棋的游戏程序。

Encrypt：　第4章，一个对文件进行加密和解密的程序。

ShowFile：　第6章，一个在屏幕上显示文本文件的程序。

Lorder：　第7章，一个设置屏幕边界颜色的程序。

Fractal：　第8章，一个在屏幕上画裂线图的程序。

Term：　第9章，一个终端模拟程序。

Show File Ⅱ：　第11章，第二个在屏幕上显示文本文件的程序。

Pong：　第12章，一个和计算机玩乒乓球的动画游戏程序。

Term Ⅱ：　第13章，第二个终端模拟程序。

Noise Maker：　第14章，一个产生声音的程序。

IP histogram：　第15章，一个记录程序各部分执行时间的工具软件。

PATHSUB:　　第16章，一个修改DOS环境变量的程序

本书假设读者已经初步掌握了IBM PC机汇编语言程序设计（但不要求很熟练），已掌握了C语言程序设计。所以，对这两种语言本身，本书不再加以介绍。读者如有必要，可参考相应的说明手册或其它书籍。

中国科学院计算技术研究所　　尹彦芝

目　录

第一章　C语言和汇编语言程序的相互调用

这一章我们将解释怎样从C语言程序内调用汇编语言程序和怎样从汇编语言程序内调用C语言程序，我们还将讨论有关“段”和“连接程序”的一些问题，如：全局变量是存放在哪儿？怎样存取这些变量？段是怎样划分的？又是怎样组合的？等等。

读者应该已初步掌握汇编语言编程，对IBM PC机的硬件结构也应该有些一般的了解。虽然对于想用汇编语言编程的人来说（不管是编写由C语言调用的子程序还是编写可独立运行的程序），熟练地掌握汇编语言是必要的，但如果仅仅是为了看懂本书中的例子，并不要求读者如此熟悉汇编语言。因为汇编语言本身是一门需要专门阐述的问题，介绍它的书又很多，这里就不再讲解了。

从C语言中调用汇编语言子程序的具体办法应该随着C编译程序的不同而有所不同。本书理论上的阐述是适合于所有的C编译程序的，但所举的例子都是使用Microsoft C编译程序3.0版或4.0版和Microsoft Macro Assembler (MASM)，这是当今在PC机上最为流行的C编译和宏汇编。

一、汇编语言子程序调用方法的简单介绍

这一节以一个简单的例子说明如何从C语言中调用一个汇编语言子程序，介绍了一些最基本的概念。如果读者对汇编语言很熟悉，则可以跳过这一节。

程序1-1是C语言主程序，程序1-2是汇编语言子程序。主程序调用子程序是为了求1加2的和，并把加得的结果打印出来。子程序具体执行这个加法。这个简单的例子解释了编写一个可供C语言调用的汇编语言子程序的许多重要的概念。

这本书中以后提供的大部分汇编语言子程序的格式也都与程序1—2的格式是一样的：把段的段地址放在合适的段寄存器内，说明函数名是全局符号名，保留寄存器，取出通过堆栈传过来的参数，恢复寄存器，返回到调用程序。这一章我们将对这些概念分别加以详细的解释。

汇编语言子程序的一开头就是SEGMENT和ASSUME伪指令。这些伪指令告诉汇编程序，该程序应该装入哪一个段，数据应该装入哪一个段，各个段寄存器可能分别含有哪一段的段地址。SEGMENT和ENDS伪指令之间的内容就构成了一个段。这个例子比较简单，只有一个程序段。如果还有其它的段（如数据段），则应还有其它的SEGMENT和ENDS伪指令对。因为段名是_TEXT，所以就告诉了汇编程序把本程序放在_TEXT段内。以后我们会看到编译程序在对C语言源程序进行编译时也会产生一个_TEXT段，这两个_TEXT段将会组合在一起。然后，ASSUME伪指令告诉汇编程序，在程序执行过程中码段寄存器CS将含有_TEXT段的基地址。尽管ASSUME伪指令只是一种“假设”，但我们没有必要通过指令去设置CS，编译和连接程序会自动地作完这件事。

下一条语句声明函数名_add是一个公共符号名（public），使得它可供其它模块存取。如果没有这条语句，则这个函数只属于这个源码文件，连接程序将不知道这个名字，

其它模块也就无法调用它。应该注意，函数名前面加了一条下划线，这是遵守C编译程序的规则。C编译程序在所有的名字的前面都加一条下划线。因此，即使在C语言程序内原来写的被调用函数名是add，但编译程序告诉连接程序的却是_add。

接着，程序内定义了两个常数a和b，它们是取堆栈中参数时所需要的两个偏移量。当主程序调用子程序时，它是按相反的顺序（在本例中先b后a）把参数压入堆栈中。当子程序获得控制权以后，它保留BP寄存器原来的内容并设置BP为堆栈指针SP的当前值。一旦这样做了以后，就可以用［BP＋偏移量］的方式取出各个参数值。在这个例子里，通过［BP＋a］可以取出参数a，通过［BP＋b］可以取出参数b。下面我们还会更详细地讨论参数的传递问题。

接着是定义子程序本身。它是从PROC语句开始，到ENDP语句结束。伪指令PROC和ENDP就分别标志一个子程序（在汇编语言中也称之为“过程”）的开始和结束。PROC后面的关键字FAR（或NEAR）就说明了这个过程是远过程还是近过程，即别的程序调用这个程序时，应该用远的CALL指令（FAR CALL）还是近的CALL指令（NEAR CALL）。如果PROC后面省略了关键字，则意味着这个过程是近过程。这个关键字也决定了在该PROC和ENDP之间（即这个子程序体内）所有RET指令的属性，即是远返回还是近返回。

子程序本身首先保留BP寄存器原来的内容，然后置它为SP寄存器的当前值，以便利用BP作基址寄存器来读取参数。子程序主要功能是很简单的，参数a被装入AX寄存器，然后再加上参数b，再通过AX寄存器把结果返回给C程序。只要结果不超过16位，都可以借助AX寄存器把结果返回给C程序。后面还将要仔细讨论这个问题。

程序1-1　add1 to 2.C

```
/* 举例：把1和2相加，打印结果3。
   main ( )
   功能：把1和2相加，打印结果3。
   算法：调用函数add把两个数相加，再调用函数printf去打印结果。
*/
   main ( )
   {
            printf ("1+2=%d.\n", add (1, 2) );
   }
```

程序1-2　add.asm

```
;   供C调用的汇编子程序，求两数的和。
   _TEXT    segment byte public 'CODE'
            assume CS:_TEXT
; _add (a, b)
;
;   功能：把两个输入参数相加并返回结果。
```

```
;       算法：保留BP，设置BP等于SP，以便利用BP来取堆栈中的输入参数。把两个输入
        参数相加，结果存在AX寄存器内。恢复BP寄存器，返回。
        public _add
        a = 4
        b = 6
_add    proc near
        push    bp
        mov     bp, sp
        mov     ax, [bp + a]
        add     ax, [bp + b]
        pop     bp
        ret
_add    endp
_TEXT   ends
        end
```

二、段和组

在讨论C语言和汇编语言混合编程以前，需要先讨论一下段和组的一些问题。当仅用C语言来编程序时，可以不关心这些问题；但当用汇编语言来编程序时，尤其是编写可以供C语言调用的汇编子程序时，却必须弄清楚这些问题。

可以以两种不同的观点来看待IBM PC的内存组织。一种观点是把整个内存看作是一个大块，在PC/XT机中这一大块就是1兆字节，地址空间是从00000到FFFFF（16进制）。另一种观点是把整个内存看作是由许多个小块组成的，每一小块是64K字节。当程序要寻址某一单元时，不但要说明这一小块的起始地址，而且要说明所寻址的单元在这一小块内的偏移量。PC的处理器（8086/8088/80286）与某些别的处理器不一样，它迫使我们一定要接受这后一种观点，即一定要把内存划分为许多小块。这些64K字节的小块就称为段。

中央处理器内部包含有4个段寄存器（8086/8088），段寄存器中存放的是当前所使用的各段的起始地址（或称为基地址）。在8080/8088中，地址位是20位，所以起始地址本来应是一个20位的数。但在8086/8088中同时也规定，起始地址不可以从绝对地址空间中的任一字节开始，而必须从16字节的边界开始（每16个字节称为1节）。既然起始地址一定是16的整数倍（即起始地址的最低4位一定是0），那么也就没有必要保存它的最低4位了。这就意味着段寄存器的内容和段内的偏移量都是16位的数。这使得地址计算变得方便多了，因为8086/8088/80286是16位处理器，处理16位数是它们的最基本的最有效的形式。

4个段寄存器的基本用法如下：

段名	段寄存器名	用法
程序段（Code）	CS	取指令
堆栈段（Stack）	SS	堆栈操作
数据段（Data）	DS	存取数据
附加数据段（Extra）	ES	在某些指令中用作第2个DS

在位置关系上，段与段之间有4种可能的情况。

1．相邻。一段接着一段。如果前一段不足64K字节，后一段则紧接其后（当然仍然以节为边界），以避免浪费。

2．重迭。程序中的几个段实际上在内存中是同一个地方，即实际上是同一个段。没有什么理由要求两个段寄存器一定要指向不同的地方。重迭的明显例子就是公用数据区。

3．部分重迭。即一个段还没有结束，另一个段又从这段内开始。或者说，一个段寄存器指向另一个段寄存器所管辖的段内。部分重迭的明显例子就是某些数据块的前后移动。

4．分离。各段之间没有什么关系，这是最常见的情况。

当数据或程序是来自几个分别编译出来的模块时（不管是来自C编译出来的模块，还是汇编出来的模块，还是两者的混合），常常需要把它们再合并一下，组合成一个或多个段。例如，两个模块可能都有一些全局变量，如果不把它们组合成一个段，那么在取不同模块内的全局变量时，需要使用不同的段寄存器。如果把它们组合成一个段，那么只需要一个段寄存器就可以了，在存取任何一个全局变量时都不用改换段寄存器。类似地，当一个模块内的函数调用另一个模块内的函数时，如果这两个模块没有组合成一个段，那么调用和返回都需要指明段地址和偏移量，即采用远调用和远返回。如果这两个模块组合成一个段，则调用和返回都不用说明段寄存器，都可采用近调用和近返回，既减短了目标码的长度，又加快了执行速度。

把多个模块组合成段的工作是由连接程序（LINK）完成的。连接程序做这件工作的依据是来自编译程序或汇编程序所产生的目标码文件，它只把段名相同，类别名也相同，且组合类型都是“PUBLIC”的各个模块组合成一个段。在用汇编语言程序时，目标文件中的这些供连接程序使用的信息又是直接来自汇编语言源程序。汇编语言程序员对这些信息有着完全的控制。在用C语言编程序时，这些信息是由编译程序自动产生的，C语言程序员几乎无需关心这一点，反过来也就意味着不可能有过多的控制。

在某些情况下，可能需要进一步实现组合，把几个段组合成一个更大的单位——组。不要求在同一组内的各个段的段名和类别名都是相同的，也不要求它们是某一特定的组合类型。同一组的各个段只是在内存中连续存放在一起，共用一个段寄存器，程序员可以通过相对于组的起点的偏移量，而不是相对于段的起点的偏移量来说明一个地址。既然一个组共用一个段寄存器，所以一个组的大小也不能超过64K字节。与模块组合成段相类似，段组合成组也是由连接程序完成的，C编译程序会自动地在目标码文件中产生有关组的信息，汇编程序则是根据汇编语言源程序中的GROUP伪指令在目标文件中产生有关组的信息的。

三、编译程序的内存模式

上一节中我们谈到的只是问题的一个方面，即模块比较小，可以把它们组合成一个段；段还比较小，可以把它们组合成一个组，这样可以简化调用子程序和取数的操作。但是问题还有另一面，有时仅仅一个模块就已经大到超过64K字节，或者程序所处理的数据超过64K字节。许多编译程序，包括Micrsooft C编译，允许程序大于64K字节，也允许使用多于64K字节的数据，代价就是牺牲效率。当程序超过64K字节以后，调用和返回都必须用远调用和远返回。当数据超过64K字节以后，指向数据的指针就必须是32位的而不

是16位的。这些不同的处理段的方式就称作编译程序的内存模式。

我们之所以需要编译程序内存模式，是因为IBM PC机的分段寻址方案迫使程序员或编译程序做出决定：或者是使用16位指针和近（NEAR）的函数调用，从而产生的程序目标码短，执行速度快，但却只有4个64K段可用。或者是使用32位指针和远（FAR）的函数调用，从而产生的程序目标码长，执行速度慢，但却可以使用整个内存空间。当用汇编语言编程序时，这种决定是由程序员来做的，可以逐个变量不同，逐个函数不同，即可以根据需要进行任意的控制。当用C语言编程序时，内存模式是在编译时由用户选定的。一旦选定，在整个程序的编译过程中就都用这个模式，除非在源程序中显式地指定另一种模式。

以Microsoft C编译为例，它提供了如下5种内存模式。

1．小模式

小模式产生一个只有两个缺省段（一个代码段和一个数据段）的目标文件。代码段和数据段都限制不超过64K字节。尽管这种模式比较小，但绝大部分程序用这种模式也就可以了。小模式程序中所有的调用都是近（NEAR）调用，所有的指针都是16位的指针。对于个别不在代码段内的函数，可以用far关键字来调用。对于个别不在数据段内的数据，可以用far或huge关键字来修正指针。

2．中模式

中模式产生的目标文件有一个数据和多个程序，它适用于程序量比较大（多于64K字节）而数据量比较小（少于64K字节）的情况。所有的缺省指针都是16位指针。在同一个模块内的缺省调用是近调用，在模块之间的缺省调用是远调用。每一个模块就是一个独立的段。对于数据可使用far和huge关键字，对于代码可使用near关键字来推翻缺省规定。

3．紧凑模式

紧凑模式产生的目标文件有一个程序段和多个数据段，它适用于程序量比较小（少于64K字节）而数据量比较大（多于64K字节）的情况。所有的缺省指针都是32位指针，数据是按需要分成多个段的。所有的缺省调用都是近调用。对于数据可以使用near和huge关键字，对于代码可以使用far关键字来推翻缺省规定。这种模式只在Microsoft C4.0及其以后版本中才提供。

4．大模式

大模式产生的目标文件有多个程序段和多个数据段，它适用于程序量和数据量都比较大的情况。所有的缺省指针都是32位指针。在同一个模块内的缺省调用是近调用，在模块之间的缺省调用是远调用。对于数据可以使用near和huge关键字，对于代码可以使用near关键字来推翻缺省规定。

5．巨模式

巨模式与大模式相似，只是它不仅允许总的数据量超过64K字节，而且允许每一个数据项超过64K字节。它也还有些具体限制，可参考编译程序手册。这种模式只在Microsoft C 4.0及其以后版本中才提供。

对于大部分应用程序，小模式就已足够了。在这种模式下，所有的数据都在同一个段内，数据段寄存器DS和堆栈段寄存器SS都指向这个段。所有的程序也都在同一段内，所有的子程序调用都是近调用。这使编写可供C语言调用的汇编语言子程序的工作变得比较

简单。本书的所有例子都是建立在小模式的基础上。

四、Microsoft C中段和组的使用

在C语言源程序中，除了程序以外，还可以有几种类型的数据，列表如下：

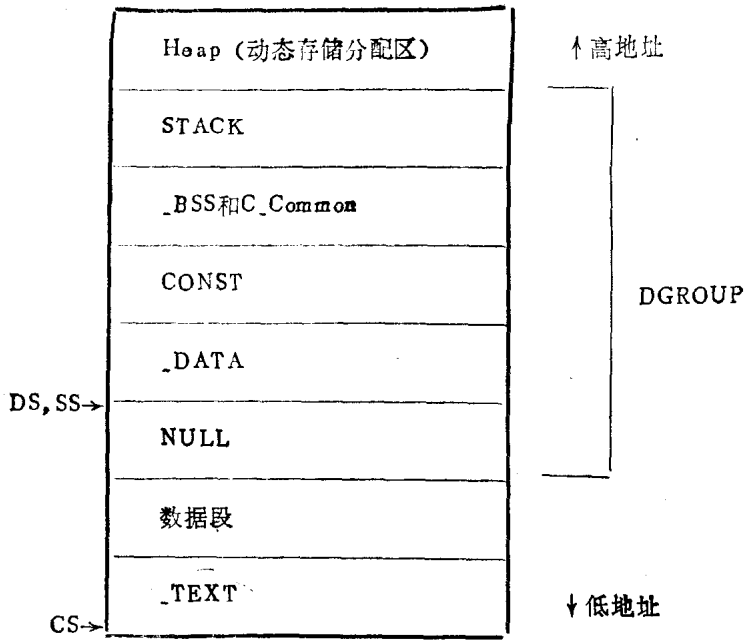
类　型	用　途
码 (Code)	由编译程序所产生的所有可执行码。
全局数据 (Global)	供所有模块存取。可以被初始化，也可以未被初始化。
静态数据 (Static)	只在一特定模块或函数内可以存取。可以被初始化，也可以未被初始化。
参数 (Parameter)	传给被调用函数的参数。
自动数据 (Automatic)	只在函数的某一段时间内存在的变量，也称为局部变量

程序1-3举例说明了所有这些类型的码和数据的定义及其使用办法。对于一个C语言程序员来说，这些数据类型应该是很熟悉的。为了理解怎样存取供各个模块共同使用的全局数据，怎样建立属于汇编语言模块的局部变量，首先就应该弄清楚C编译程序是怎样去为这些变量分配存储段的。这种分配办法是随编译程序的不同而不同，这里讲的是Microsoft C 编译3.0版和4.0版。

Microsoft C编译产生如下几种段：

段名	存放的内容
_BSS	未初始化的静态数据（除了那些用far关键字加以强制说明的以外）。
C_common	包含小模式和中模式的所有未初始化的全局数据。在紧凑模式或大模式中，这种类型的数据是放在类别名为FAR_BSS的数据段中。
_DATA	已初始化的全局数据和静态数据（除了那些用far关键字加以强制说明的以外）。这是缺省的数据段。
数据段	用关键字far强制说明的全局和静态数据。已初始化的数据项其类别名为FAR_DATA，未初始化的数据项其类别名为FAR_BSS。
STACK	自动变量，局部数据项。
CONST	只读常数（浮点常数，远数据的各个段值）。
_TEXT	码。

_DATA，CONST，_BSS和STACK几个段被组合成一个名为DGROUP的组。在正常操作时，DS和SS都指向这个组的起点。此外，在这个组的开头还有一个名为NULL的段，这个段用于存放编译程序拷贝权等告示信息和检测无效指针。在程序运行之前和运行之后都会检查这个NULL段，如果发现内容已经改变了，则可能有什么东西出了严重的错误。最常见的错误就是程序间接地往0指针（它指向这个段的起点）写了什么东西。如果是这种错误，则显示出“Null pointer assignment”信息。当然，这个组的总字节数不能超过64K。



为了便于读者理解和查阅，下面这张表格列出了所有5种内存模式下的各个段名，这些段的对齐类型，组合类型，类别名以及它们所属的组名。

内存模式	段名	对齐类型	组合类型	类别名	组
小模式	_TEXT	byte	public	CODE	—
	数据段（1）	para	private	FAR_DATA	—
	数据段（2）	para	public	FAR_BSS	—
	NULL	para	public	BEGDATA	DGROUP
	_DATA	word	public	DATA	DGROUP
	CONST	word	public	CONST	DGROUP
	_BSS	word	public	BSS	DGROUP
	STACK	para	stack	STACK	DGROUP
中模式	模块名_TEXT	byte	public	CODE	—
	⋮				
	⋮				
	数据段（1）	para	private	FAR_DATA	—
	数据段（2）	para	public	FAR_BSS	—
	NULL	para	public	BEGDATA	DGROUP
	_DATA	word	public	DATA	DGROUP
	CONST	word	public	CONST	DGROUP
	_BSS	word	public	BSS	DGROUP
	STACK	para	stack	STACK	DGROUP
紧凑模式	_TEXT	byte	public	CODE	—
	数据段（3）	para	private	FAR_DATA	—
	数据段（4）	para	public	FAR_BSS	—
	NULL	para	public	BEGDATA	DGROUP

	_DATA	word	public	DATA	DGROUP
	CONST	word	public	CONST	DGROUP
	_BSS	word	public	BSS	DGROUP
	STACK	para	stack	STACK	DGROUP
大模式	模块名_TEXT	byte	public	CODE	—
	⋮				
	⋮				
	数据段（3）	para	private	FAR_DATA	—
	数据段（4）	para	public	FAR_BSS	—
	NULL	para	public	BEGDATA	DGROUP
	_DATA	word	public	DATA	DGROUP
	CONST	word	public	CONST	DGROUP
	_BSS	word	public	BSS	DGROUP
	STACK	para	stack	STACK	DGROUP
巨模式	模块名_TEXT	byte	public	CODE	—
	⋮				
	⋮				
	数据段（3）	para	private	FAR_DATA	—
	数据段（4）	para	public	FAR_BSS	—
	NULL	para	public	BEGDATA	DGROUP
	_DATA	word	public	DATA	DGROUP
	CONST	word	public	CONST	DGROUP
	_BSS	word	public	BSS	DGROUP
	STACK	para	stack	STACK	DGROUP

注解：

数据段（1）：已初始化的far或huge数据段。

数据段（2）：未初始化的far或huge数据段。

数据段（3）：已初始化的全局和静态数据段。

数据段（4）：未初始化的全局和静态数据段。

为了在汇编语言内使用C的数据项或在C语言内使用汇编语言的数据项，这些数据项必须同时为两种语言所感知。它们在定义语言内和使用语言内不但应该名字相同，类型相同（所占的字节数一样），而且应该位于正确的段和组内。外部函数名是一种例外。在汇编语言中调用C语言函数时，可以简单地说明其类型为NEAR就可以了，而不用说明是属于_TEXT段，连接程序会正确地发现被调用函数驻留在哪一段内。通过声明这个外部函数为NEAR，就等于告诉汇编程序和连接程序，被调用函数是在与调用者相同的段内。对于C模块，段的说明是由编译程序自动处理的；对于汇编语言模块，段的说明必须由程序员去处理。为了与C语言互相连接，汇编语言程序员的处理原则与编译程序的处理原则必须相一致。程序1-3是一个C语言程序，其中定义了几个变量。程序1-4是一个汇编语言程序（部分），它定义了几个同样的变量。从这两个程序中可以看出C语言和汇编语言在变

量定义上的对应关系。

应该特别注意一下变量名的命名规则。C编译程序总是自动地在所定义的函数和变量名前再加一个下线符。例如，若C语言程序中定义了一个变量abc，则编译完后这个变量的名字就成了_abc。在汇编语言中如果想引用这个变量，则也使用名字_abc。反过来也如此，如果在汇编语言中定义了一个变量_xy，则在C语言程序中引用这个变量时应该用变量xy。

程序1-3 typex.c

```
/*  C语言中码和数据类型举例。 */
    int a, b;                          /* Uninitialized global data. */
    int x = { 5 };                     /* Initialized global data. */
    static int i;                      /* Uninitialized static data local to this module. */
    static int y = { 9 };              /* Initialized static data local to this module. */
/*  retNext (inc)：返回由这个程序维护的变量count的当前值，然后使变量count增
    加值inc。
*/
    retNext (inc)
    int inc;              /* Parameter data. */
    {
          static int lastinc;          /*Uninitialized static data local to this function.*/
          static int count = { 0 };    /* Initialized static data local to this function.*/
          int retValue;                /* Automatic data. */
          lastinc = inc;               /* Code  */
          retValue = count;            /* Code. */
          count += inc;                /* Code. */
          return (retValue) ;          /* Code. */
```

程序1-4 decl.asm

```
;   等效的汇编语言程序
; First we declare the segments and groups:
_TEXT       SEGMENT    BYTE PUBLIC 'CODE'
_TEXT       ENDS
_BSS        SEGMENT    WORD PUBLIC 'BSS'
_BSS        ENDS
_DATA       SEGMENT    WORD PUBLIC 'DATA'
_DATA       ENDS
```

```
CONST       SEGMENT    WORD PUBLIC 'CONST'
CONST       ENDS
DGROUP      GROUP      CONST, _BSS, _DATA
; Next we declare the variables:
_DATA       SEGMENT
            EXTRN      _a:WORD          ; int a, b
            EXTRN      _b:WORD
            PUBLIC     _x               ; int x={5}
_x          DW         5
_y          DW         9                ; static int y={9}
_count      DW         0                ; static int count={0}
_DATA       ENDS
_BSS        SEGMENT
_i          DW         ?                ; static int i
_lastinc    DW         ?                ; static int lastinc
_BSS        ENDS
```

; 注意，变量_y，_i，_lastinc和_count只与这个模块有关，与外部的模块无关。参数inc和自动变量retValue在这儿没有加以声明，它们是在运行时分配在堆栈中的，见程序1-6。

程序1-5 caller.asm

```
; 与一条C语言语句“retNext (10)”等效的汇编语言程序。
ASSUME  CS:_TEXT
EXTRN   _retNext:NEAR
        .
        .
        .
; 下面是从汇编语言内调用一个C函数。
MOV     AX, 10          ; Push the parameter
PUSH    AX
CALL    _retNext        ; Call the function
POP     BX              ; Pop the parameter without destroying AX
; 执行到这一点，AX已经含有了由retNext (10) 返回的值。
;   :
```

程序1-6 called.asm

```
; 与程序1-3中的C语言函数retNext等效的汇编语言函数。
; 程序1-4中的说明必须加在这个程序的前面，但其中对变量_a和_b的说明可以不要。
```

```
                ASSUME  CS:_TEXT, DS:DGROUP, SS:DGROUP
_TEXT           SEGMENT
                PUBLIC  _retNext
_retNext        PROC NEAR
                inc = 4
                PUSH    BP
                MOV     BP, SP
                PUSH    DI
                PUSH    SI
                MOV     BX, [BP+inc]
                MOV     _lastinc, BX
                MOV     AX, _count
                ADD     _count, BX
                POP     SI
                POP     DI
                POP     BP
                RET
_retNext        ENDP
_TEXT           ENDS
```

五、调用规则

所有的C编译程序在编译函数调用语句时基本上遵循下列同样的过程：

1. 保留所有可能被被调用函数破坏而调用者又还需要使用的寄存器。大部分编译程序都假设某些寄存器是不会被被调用函数破坏的，对这些寄存器则不需要保留。

2. 把要传给被调用函数的参数压入堆栈。大部分C编译程序都按与参数表中出现的顺序相反的顺序把参数压入堆栈。这样，参数表中的第1个参数总是最后1个被压入，因而也就存在堆栈的顶部。

3. 通过CALL指令发出函数调用。可能是近调用，也可能是远调用。然后就执行函数，执行完以后再返回到调用者。

4. 取得返回值。大部分编译程序都是利用AX（或AX/DX）返回简单值，利用静态存储区返回较大的值。

5. 从堆栈中把参数弹出不要。

6. 恢复所保留的寄存器。

除了这些操作以外，还假设某些条件总是满足的。特别是段寄存器CS，DS，SS的值将假设总是指向特定的一组存储段的。

下面就来详细讨论Microsoft C编译程序3.0版和4.0版的调用规则。这两个版本的调