Borland C++ 中，由于 int 类型的长度是 4 字节，因此汇编器就把 int a1 = 1; 变换成了 _a1 label dword 和 dd 1。同样，这里也定义了相当于全局变量 *a*2～*a*5 的标签 _a2～_a5，它们各自的初始值 2～5 也都被存储在了 4 字节的领域中。

接下来，让我们来看一下 _BSS 段定义的内容。这里定义了相当于全局变量 *b*1～*b*5 的标签 _b1～_b5。（6）的 db 4 dup(?) 表示的是申请分配了 4 字节的领域，但值尚未确定（这里用？来表示）的意思。db（define byte）表示有 1 个长度是 1 字节的内存空间。因而，db 4 dup(?) 的情况下，就是 4 字节的内存空间。这里大家要注意不要和 dd 4 混淆了。db 4 dup(?) 表示的是 4 个长度是 1 字节的内存空间。而 db 4 表示的则是双字节（＝4 字节）的内存空间中存储的值是 4。

在 _DATA 和 _BSS 的段定义中，全局变量的内存空间都得到了确保，这一点大家想必都清楚了吧。因而，从程序的开始到结束，所有部分都可以参阅全局变量。而这里之所以根据是否进行了初始化把全局变量的段定义划分为了两部分，是因为在 Borland C++ 中，程序运行时没有初始化的全局变量的领域（_BSS 段定义）都会被设定为 0 进行初始化。可见，通过汇总，初始化很容易实现，只要把内存的特定范围全部设定为 0 就可以了。

10.10　临时确保局部变量用的内存空间

为什么局部变量只能在定义该变量的函数内进行参阅呢？这是因为，局部变量是临时保存在寄存器和栈中的。正如本章前半部分讲的那样，函数内部利用的栈，在函数处理完毕后会恢复到初始状态，因此局部变量的值也就被销毁了，而寄存器也可能会被用于其他目的。因此，局部变量只是在函数处理运行期间临时存储在寄存器和栈上。

在代码清单 10-6 中定义了 10 个局部变量。这是为了表示存储局部变量的不仅仅是栈，还有寄存器。为确保 *c*1～*c*10 所需的领域，寄存器空闲时就使用寄存器，寄存器空间不足的话就使用栈。

下面让我们来看一下代码清单 10-7 中 _TEXT 段定义的内容。(7) 表示的是 MyFunc 函数的范围。在 MyFunc 函数中定义的局部变量所需要的内存领域，会被尽可能地分配在寄存器中。大家可能会认为用高性能的寄存器来代替普通的内存是很奢侈的事情，不过编译器不会这么认为，只要寄存器有空间，编译器就会使用它。因为与内存相比，使用寄存器时访问速度会高很多，这样就可以更快速地进行处理。局部变量利用寄存器，是 Borland C++ 编译器最优化的运行结果。旧的编译器没有类似的最优化功能，局部变量就可能会仅仅使用栈。

代码清单中的 (8) 表示的是往寄存器中分配局部变量的部分。仅仅对局部变量进行定义是不够的，只有在给局部变量赋值时，才会被分配到寄存器的内存区域。(8) 就相当于给 5 个局部变量 *c*1～*c*5 分别赋予数值 1～5 这一处理。eax、edx、ecx、ebx、esi 是 Pentium 等 x86 系列 32 位 CPU 寄存器的名称 (参考表 10-2)。至于使用哪一个寄存器，则要由编译器来决定。这种情况下，寄存器只是被单纯地用于存储变量的值，和其本身的角色没有任何关系。

x86 系列 CPU 拥有的寄存器中，程序可以操作的有十几个。其中空闲的，最多也只有几个。因而，局部变量数目很多的时候，可分配的寄存器就不够了。这种情况下，局部变量就会申请分配栈的内存空间。虽然栈的内存空间也是作为一种存储数据的段定义来处理的，但在程序各部分都可以共享并临时使用这一点上，它和 _DATA 段定义及 _BSS 段定义在性质上还是有些差异的。例如，在函数入口处为变量申请分配栈的内存空间的话，就必须在函数出口处进行释放。否则，经

过多次调用函数后，栈的内存空间就会被用光了。

在（8）这一部分中，给局部变量 *c*1～*c*5 分配完寄存器后，可用的寄存器数量就不足了。于是，剩下的 5 个局部变量 *c*6～*c*10 就被分配了栈的内存空间，如（9）所示。函数入口（10）处的 add esp,-20 指的是，对栈数据存储位置的 esp 寄存器（栈指针）的值做减 20 的处理。为了确保内部变量 *c*6～*c*10 在栈中，就需要保留 5 个 int 类型的局部变量（4 字节 ×5 = 20 字节）所需的空间。（11）中的 mov ebp,esp 这一处理，指的是把当前 esp 寄存器的值复制到 ebp 寄存器中。之所以需要（11）这一处理，是为了通过在函数出口处的（12）这一 move esp,ebp 的处理，把 esp 寄存器的值还原到原始状态，从而对申请分配的栈空间进行释放，这时栈中用到的局部变量就消失了。这也是栈的清理处理。在使用寄存器的情况下，局部变量则会在寄存器被用于其他用途时自动消失（图 10-6）。

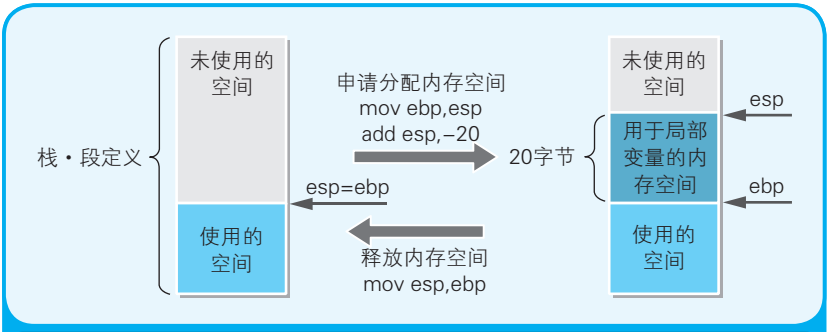

图 10-6　用于局部变量的栈空间的申请分配和释放

（9）中的 5 行代码是往栈空间中代入数值的部分。由于在向栈申请内存空间前，借助 mov ebp,esp 这个处理，esp 寄存器的值被保存到了 ebp 寄存器中，因此，通过使用 [ebp - 4]、[ebp - 8]、[ebp - 12]、

[ebp - 16]、[ebp - 20] 这样的形式，就可以将申请分配的 20 字节的栈内存空间切分成 5 个长度分别是 4 字节的空间来使用（图 10-7）。例如，（9）中的 mov dword ptr [ebp - 4], 6 表示的就是，从申请分配的内存空间的下端（ebp 寄存器指示的位置）开始往前 4 字节的地址（[ebp - 4]）中，存储着 6 这一 4 字节的数据。

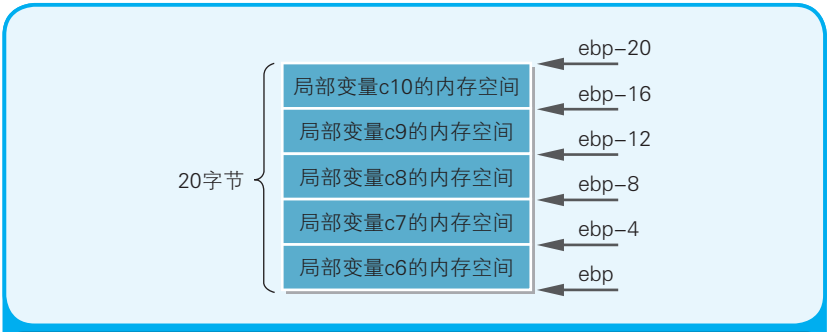

图 10-7　将栈的内存空间进行分割

关于往全局变量中代入局部变量的数值这一内容，这里不再进行说明。这时可能有读者会产生疑问，既然不进行说明，那为什么代码清单 10-6 中没有省略掉该部分呢？这是为了避免编译器的最优化功能。如果仅进行定义局部变量并代入数值这一处理的话，编译器的最优化功能就会启动，届时编译器就会认为某些代码没有意义，从而导致汇编语言的源代码无法生成。这样看来，编译器还是很聪明的吧！

10.11　循环处理的实现方法

接下来，让我们继续解析汇编语言的源代码，看一下 for 循环及 if 条件分支等 C 语言程序的**流程控制**是如何实现的[①]。代码清单 10-8 是将

① 通过利用 for 语句及 if 语句来改变程序流程的机制称为“流程控制”。

局部变量 *i* 作为循环计数器[①]连续进行 10 次循环的 C 语言源代码。在 for 语句中，调用了不做任何处理的 MySub 函数。这里我们把代码清单 10-8 转换成汇编语言，然后仅把相当于 for 处理的部分摘出来，如代码清单 10-9 所示。

代码清单 10-8　执行循环处理的 C 语言源代码

```
// 定义 MySub 函数
void MySub()
{
      // 不做任何处理
}

// 定义 MyFunc 函数
Void MyFunc()
{
    int i;
    for (i = 0; i < 10; i++ )
    {
         // 重复调用 MySub 函数 10 次
         MySub();
    }
 }
```

代码清单 10-9　将代码清单 10-8 中的 for 语句转换成汇编语言的结果

```
      xor       ebx, ebx          ; 将 eax 寄存器清 0
@4    call      _MySub            ; // 调用 MySub 函数
      inc       ebx               ; //ebx 寄存器的值加 1
      cmp       ebx,10            ; // 将 ebx 寄存器的值和 10 进行比较
      jl        short @4          ; // 如果小于 10 就跳转到 @4
```

C 语言的 for 语句是通过在括号中指定循环计数器的初始值（*i* = 0）、循环的继续条件（*i* < 10）、循环计数器的更新（*i*++）这 3 种形式来进行循环处理的。与此相对，在汇编语言的源代码中，循环是通过比较指令（cmp）和跳转指令（jl）来实现的。

① 用来计算循环次数的变量称为“循环计数器”。

下面就让我们按照代码清单 10-9 的内容的顺序来进行说明。MyFunc 函数中用到的局部变量只有 *i*，变量 *i* 申请分配了 ebx 寄存器的内存空间。for 语句的括号中的 *i*=0; 被转换成了 xor ebx,ebx 这一处理。xor 指令会对左起第一个操作数和右起第二个操作数进行 XOR 运算，然后把结果存储在第一个操作数中。由于这里把第一个操作数和第二个操作数都指定为了 ebx，因此就变成了对相同数值进行 XOR 运算。也就是说，不管当前 ebx 寄存器的值是什么，结果肯定都是 0[①]。虽然用 mov 指令的 mov ebx,0 也会得到同样的结果，但与 mov 指令相比，xor 指令的处理速度更快。这里，编译器的最优化功能也会启动。

ebx 寄存器的值初始化后，会通过 call 指令调用 MySub 函数（_MySub）。从 MySub 函数返回后，则会通过 inc 指令对 ebx 寄存器的值做加 1 处理。该处理就相当于 for 语句的 *i*++，++ 是当前数值加 1 的意思。

下一行的 cmp 指令是用来对第一个操作数和第二个操作数的数值进行比较的指令。cmp ebx,10 就相当于 C 语言的 *i*<10 这一处理，意思是把 ebx 寄存器的数值同 10 进行比较。汇编语言中比较指令的结果，会存储在 CPU 的标志寄存器中。不过，标志寄存器的值，程序是无法直接参考的。那么，程序是怎么来判断比较结果的呢？

实际上，汇编语言中有多个跳转指令，这些跳转指令会根据标志寄存器的值来判定是否需要跳转。例如，最后一行的 jl，是 jump on less than（小于的话就跳转）的意思。也就是说，jl short @4 的意思就是，前面运行的比较指令的结果若“小”的话就跳转到 @4 这个标签。

① 相同数值进行 XOR 运算，运算结果为 0。XOR 运算的规则是，值不同时结果为 1，值相同时结果为 0。例如，01010101 和 01010101 进行 XOR 运算的话，就会分别对该数字的各数字位进行 XOR 运算。因为这两个数的每个位都相同，因此，运算结果就是 00000000。

代码清单 10-10 是按照代码清单 10-9 中汇编语言源代码的处理顺序重写的 C 语言源代码（由于 C 语言中无法使用 @ 字符开头的标签，因此这里用了 L4 这个标签名），也是对程序实际运行过程的一个直接描述。不过看来看去还是觉得使用 for 语句的代码清单 10-8 的源代码更智能些。人们经常说“汇编语言是对 CPU 的实际运行进行直接描述的低级编程语言，C 语言是用与人类的感觉相近的表现来描述的高级编程语言”，此时，想必大家都能深切体会这句话的意思了吧。此外，代码清单 10-10 的第一行中的 *i*^=*i*，意思是对 *i* 和 *i* 进行 XOR 运算，并把结果代入 *i*。为了和汇编语言的源代码进行同样的处理，这里把将变量 *i* 的值清 0 这一处理，通过对变量 *i* 和变量 *i* 进行 XOR 运算来实现了。借助 *i*^=*i*，*i* 的值就变成了 0。

代码清单 10-10　用 C 语言来表示代码清单 10-9 的处理顺序

```
    i ^= i;
L4: MySub();
    i++;
    if (i < 10) goto L4;
```

10.12　条件分支的实现方法

下面让我们来看一下条件分支的实现方法。条件分支的实现方法同循环处理的实现方法类似，使用的也是 cmp 指令和跳转指令，这一点估计大家也预料到了。

没错，条件分支就是利用这些指令来实现的。不过，为了以防万一，我们来确认一下。代码清单 10-11 是，根据变量 *a* 的值来调用不同函数（MySub1 函数、MySub2 函数、MySub3 函数）的 C 语言源代码。为了实现条件分支，这里使用了 if 语句。示例中被调用的各个函数，都不进行任何处理。将代码清单 10-11 的 MyFunc 函数处理转换成汇编

语言源代码后，结果就如代码清单 10-12 所示。

代码清单 10-11　进行条件分支的 C 语言源代码

```
// 定义 MySub1 函数
void MySub1()
{
      // 不做任何处理
}

// 定义 MySub2 函数
void MySub2()
{
    // 不做任何处理
}

// 定义 MySub3 函数
void MySub3()
{
    // 不做任何处理
}

// 定义 MyFunc 函数
void MyFunc()
{
    int a = 123;
    // 根据条件调用不同的函数
    if (a > 100)
    {
        MySub1();
    }
    else if (a < 50)
    {
        MySub2();
    }
    else
    {
        MySub3();
    }
}
```

代码清单 10-12　将代码清单 10-11 的 MyFunc 函数转换成汇编语言后的结果

```
_MyFunc     proc    near
     push   ebp;
     mov    ebp,esp;
```

```
      mov     eax,123          ; 把 123 存入 eax 寄存器中
      cmp     eax,100          ; 把 eax 寄存器的值同 100 进行比较
      jle     short @8         ; 比 100 小时，跳转到 @8 标签
      call    _MySub1          ; 调用 MySub1 函数
      jmp     short @11        ; 跳转到 @11 标签
@8:   cmp     eax,50           ; 把 eax 寄存器的值同 50 进行比较
      jge     short @10        ; 大于 50 时，跳转到 @10 标签
      call    _MySub2          ; 调用 MySub2 函数
      jmp     short @11        ; 跳转到 @11 标签
@10:  call    _MySub3          ; 调用 MySub4 函数
@11:  pop     ebp
      ret
_MyFunc       endp
```

代码清单 10-12 中用到了三种跳转指令，分别是比较结果小时跳转的 jle（jump on less or equal）、大时跳转的 jge（jump on greater or equal）、不管结果怎样都无条件跳转的 jmp。在这些跳转指令之前还有用来比较的 cmp 指令，比较结果被保存在了标志寄存器中。这里我们添加了注释，大家不妨顺着程序的流程看一下。虽然同 C 语言源代码的处理流程不完全相同，不过大家应该知道处理结果是相同的。此外，还有一点需要注意的是，eax 寄存器表示的是变量 *a*。

虽然大部分的 C 语言参考书中都写着“为了便于理解程序的结构，应尽量避免使用无条件分支的 goto 语句”，不过，在汇编语言这一领域中，如果不使用相当于 C 语言 goto 语句的 jmp 指令，就无法实现循环和条件分支。由此看来，关于应不应该在 C 语言中使用 goto 语句，大家没有必要这么紧张。

10.13　了解程序运行方式的必要性

通过对 C 语言源代码和汇编语言源代码进行比较，想必大家对“程序是怎样跑起来的”又有了更深的理解。而且，从汇编语言源代码中获得的知识，在某些情况下对查找 bug 的原因也是有帮助的。

让我们来看个示例。代码清单 10-13 是更新全局变量 *counter* 的值的 C 语言程序。MyFunc1 函数和 MyFunc2 函数的处理内容，都是把全局变量 *counter* 的值放大到 2 倍。counter *= 2 ; 指的是把 *counter* 的数值乘以 2，然后再把所得结果赋值到 *counter* 的意思。这里，假设我们利用**多线程处理**[①]，同时调用了一次 MyFunc1 函数和 MyFunc2 函数。这时，全局变量 *counter* 的数值，理应变成 100×2×2 = 400。然而，某些时候结果也可能会是 200。至于为什么会出现该 bug，如果没有调查过汇编语言的源代码，也就是说如果对程序的实际运行方式不了解的话，是很难找到其原因的。

代码清单 10-13　两个函数更新同一个全局变量数值的 C 语言程序

```
// 定义全局变量
int counter = 100;

// 定义 MyFunc1 函数
void MyFunc1()
{
   counter *= 2;
}

// 定义 MyFunc2 函数
void MyFunc2()
{
   counter *=2;
}
```

将代码清单 10-13 的 counter *= 2; 部分转换成汇编语言源代码后，结果就如代码清单 10-14 所示。这里希望大家注意的是，C 语言源代码中 counter *= 2; 这一个指令的部分，在汇编语言源代码，也就是实际运行的程序中，分成了 3 个指令。如果只是看 counter *= 2; 的话，就会以为 *counter* 的数值被直接扩大为了原来的 2 倍。然而，实际上执行的却

① “线程”是操作系统分配给 CPU 的最小运行单位。源代码的一个函数就相当于一个线程。多线程处理指的是在一个程序中同时运行多个函数的意思。