第7章　运行时环境

本章要点

- 程序执行时的存储器组织
- 完全静态运行时环境
- 基于栈的运行时环境
- 动态存储
- 参数传递机制
- TINY语言的运行时环境

在前几章中，我们已研究了实现源语言静态分析的编译程序各阶段。该内容包括了扫描、分析和静态语义分析。这个分析仅仅取决于源语言的特性，它与目标 (汇编或机器)语言及目标机器和它的操作系统的特性完全无关。

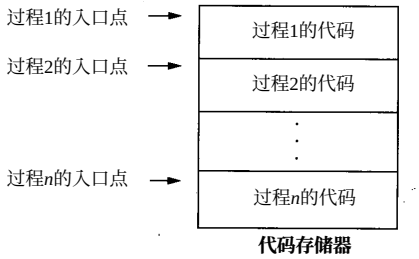
在本章及下一章中，我们将转向研究编译程序如何生成可执行代码的问题。这个研究包括了附加分析，例如由优化程序实现的分析，其中的一些可以与机器无关。但是代码生成的许多任务都依赖于具体的目标机器。然而同样地代码生成的一般特征在体系结构上仍保留了很大的变化。运行时环境(runtime environment)尤为如此，运行时环境指的是目标计算机的寄存器以及存储器的结构，用来管理存储器并保存指导执行过程所需的信息。实际上，几乎所有的程序设计语言都使用运行时环境的3个类型中的某一个，它的主要结构并不依赖于目标机器的特定细节。环境的这3个类型分别是：FORTRAN77的完全静态环境(fully static environment)特征、像C、C++、Pascal以及Ada这些语言的基于栈的环境(stack-based environment)，以及像LISP这样的函数语言的完全动态环境 (fully dynamic environment)。这3种类型的混合形式也是可能的。

本章将按顺序逐个讨论这3种环境，还指出哪些环境是可行的语言特征以及它们必须具有的特性。这包括了作用域及分配问题、过程调用的本质和不同的参数传递机制。这一章集中讨论的是环境的一般结构，而第8章着重于维护环境需要生成的真实代码。在这一点上，大家应记住编译程序只能间接地维护环境，在程序执行期间它必须生成代码进行必要的维护操作。相反地由于解释程序可以在其自己的数据结构中直接维护环境，因而它的任务就很简单。

本章的第一节包括了对所有运行时环境的一般特征及其与目标机器的体系结构之间的关系的论述；之后的两节探讨了静态环境和基于栈的环境，以及执行时的操作示例。由于基于栈的环境是最常见的，所以我们对于基于栈系统的不同变型和结构又要着重讲述。在这之后是一些动态存储问题，其中包括了完全动态环境和面向对象的环境。下面还会讲到有关环境操作的各种参数传递技术。本章最后简要描述了实现TINY语言所需的简单环境。

7.1　程序执行时的存储器组织

典型计算机的存储器可分为寄存器区域和较慢的直接编址的随机访问存储器　(RAM)。RAM区域还可再分为代码区和数据区。在绝大多数的语言中，执行时不可能改变代码区，且在概念上可将代码和数据区看作是独立的。另外由于代码区在执行之前是固定，所以在编译时所有代码的地址都是可计算的，代码区可如下所示：



特别地，在编译时还可以知道每个过程的入口点和函数⊖。对数据的分配不能这样说，它只有一小部分可在执行之前被分配到存储器中的固定位置。本章大部分内容都会谈论如何处理非固定的或动态的数据分配。

在执行之前，可以将一类数据固定在存储器中，它还包括了程序的全局和 /或静态数据(FORTRAN77与绝大多数的语言不同，它所有的数据都属于这一类)。这些数据通常都在一个固定区域内并以相似的风格单独分配给代码。在 Pascal 中，全局变量属于这一类， C的外部和静态变量也是如此。

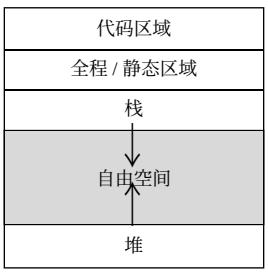
在组织全局/静态区域中出现的一个问题是它涉及到编译时所知的常量。这其中包括了 C和Pascal 的**const**声明以及代码本身所用的文字值，例如串“ **Hello %d\n**”和在C语句

```
printf (" Hello %d\n ", 12345 ) ;
```

中的整型值12345。诸如0和1这样较小的编译时常量通常由编译程序直接插入到代码中且不为其分配任何数据空间。同样地，由于编译程序已掌握了全局函数或过程的入口点且可直接将其插入到代码中，所以也不为它们分配全局数据区。然而我们却将大型的整型值、浮点值，特别是串文字分配到全局 /静态区域中的存储器，在启动时仅保存一次，之后再由执行代码从这些位置中得到(实际上，在C中串文字被看作是指针，因此它们必须按照这种方式来保存)。

用作动态数据分配的存储区可按多种方式组织。典型的组织是将这个存储器分为栈 (stack)区域和堆(heap)区域，栈区域用于其分配发生在后进先出 LIFO(last-in, first-out)风格中的数据，而堆区域则用于不符合LIFO协议(例如在C中的指针分配)的动态分配⊜。目标机器的体系结构通常包括处理器栈，利用了这个栈使得用处理器支持过程调用和返回 (使用基于存储器分配的主要机制)成为可能。有时，编译程序不得不将处理器栈的显式分配安排在存储器内的恰当位置中。

一种一般的运行时存储器组织如下所示，它具有上述所有的存储器分类：

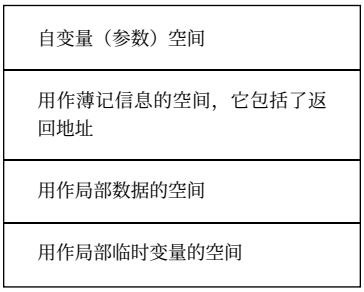


⊖ 更为可能的情况是，代码由装载程序装载到存储器的一个在执行开始时分配的区域中，因此这是完全不可预测的。但是之后的所有实际地址都由从固定的装载基地址的偏移自动计算得出，因此和固定地址的原理相同。有时，编译器的编写者必须留心生成可重定位代码 (relocatable code),其中都相对于某个基址 (base)(通常是寄存器)执行转移、调用以及引用。下一章将给出一些例子。

⊜ 读者应注意到堆通常是一个简单的线性存储器区域。将它称之为堆是一个历史原因，这与算法 (如堆类排序)中用到的堆数据结构无关。

上图中的箭头表示栈和堆的生长方向。传统上是将栈画作在存储器中向下生长，这样它的顶部实际就是在其所画区域的底部。堆也画得与栈相似，但它不是 LIFO结构且它的生长和缩短比箭头所表示的还要复杂(参见7.4节)。在某些组织中，栈和堆被分配在不同的存储器部分，而不是占据相同的区域。

存储器分配中的一个重要单元是过程活动记录(procedure activation record)，当调用或激活过程或函数时，它包含了为其局部数据分配的存储器。活动记录至少应包括以下几个部分：



在这里应强调(而且以后还要重复)这个图示仅仅表示的是活动记录的一般组织。包括其所含数据的顺序的特定细节则依赖于目标机器的体系结构、被编译的语言特性，甚至还有编译程序的编写者的喜好。

所有过程活动记录的某些部分，例如用于簿记信息的空间，具有相同的大小。而其他部分，诸如用于自变量和局部数据的空间会对每一个过程保持固定，但是每个过程都各不相同。某些活动记录还会由处理器自动分配到过程调用上(例如存储返回地址)。其他部分(如局部临时变量空间)可能需要由编译程序生成的指令显式地分配。根据语言的不同，可能将活动记录分配在静态区域(FORTRAN77)、栈区域(C、Pascal)、或堆区域(LISP)。当将活动记录保存在栈中时，它们有时指的是栈框架(stack frame)。

处理器寄存器也是运行时环境的结构部分。寄存器可用来保存临时变量、局部变量甚至是全局变量。当处理器具有多个寄存器时，正如在较新的 RISC处理器中一样，整个静态区域和整个活动记录都可完整地保存在寄存器中。处理器还具有特殊用途的寄存器以记录执行，如在大多数的体系结构中的程序计数器 (pc)、栈指针(sp)(stack pointer)。可能还会为跟踪过程活动而特别设计寄存器。这样的寄存器典型的有指向当前活动记录的框架指针 (fp)(frame pointer)，以及指向保存自变量(参数值)的活动记录区域的自变量指针(argument pointer)⊖。

运行时环境的一个特别重要的部分是当调用过程或函数时，对必须发生的操作序列的判定。这样的操作可能还包括活动记录的存储器分配、计算和保存自变量以及为了使调用有效而进行的必要的寄存器的保存和设置。这些操作通常指的是调用序列 (calling sequence)。过程或函数返回时需要的额外操作，如放置可由调用程序访问的返回值、寄存器的重新调整，以及活动记录存储器的释放，也通常被认为是调用序列的一个部分。如果需要，可将调用时执行的调用序列部分称作是调用序列(call sequence)，而返回时执行的部分称为返回序列(return sequence)。

调用序列设计的重要方面有： 1) 如何在调用程序和被调用程序之间分开调用序列操作 (也就是有多少调用序列的代码放在调用点上，多少放在每个过程的代码开头)；2)在多大程度上依赖处理器对调用支持而不是为调用序列的每一步生成显式代码。由于在调用点上比在被调用

⊖ 这些名称都是从VAX体系结构中得到的，但是类似的名称还可出现在其他体系结构中。

程序内更易生成调用序列代码，但是这样做会引起生成代码数量的生长，因为在每个调用点都要复制相同的代码所以第1点是一个很棘手的问题，后面还要再更详细地讲到这些问题。

调用程序至少要负责计算自变量并将它们放在可由被调用程序找到的位置上 (可能是直接放在被调用程序的活动记录中)。此外，调用程序或被调用程序或两者必须保存调用点上的机器状态，包括返回地址，可能还有正使用的寄存器。最后，在调用程序和被调用程序之间须用某个可能的合作方式建立所有附加的簿记信息。

7.2 完全静态运行时环境

最简单的运行时环境类型是所有数据都是静态的，且执行程序期间在存储器中保持固定。这样的环境可用来实现没有指针或动态分配，且过程不可递归调用的语言。此类语言的标准例子是FORTRAN77。

在完全静态环境中，不仅全局变量，所有的变量都是静态分配。因此，每个过程只有一个在执行之前被静态分配的活动记录。我们都可通过固定的地址直接访问所有的变量，而不论它们是局部的还是全局的，则整个程序存储器如下所示：

在这样的环境中，保留每个活动记录的簿记信息开销相对较小，而且也不需要在活动记录中保存有关环境的额外信息 (而不是返回地址)。用于这样环境的调用序列也十分简单。当调用一个过程时，就计算每个自变量，并将其保存到被调用过程的活动中恰当的参数位置。接着保存调用程序代码中的返回地址，并转移到被调用的过程的代码开头。返回时，转移到返回地址[⊖]。

例7.1 作为这种环境的具体示例，考虑程序清单 7-1中的FORTRAN77程序。这个程序有一个主过程和一个附加的过程**QUADMEAN**[⊜]。在主过程和**QUADMEAN**中有由**COMMON MAXSIZE**声明

⊖ 在大多数的体系结构中，子例程转移自动地保存返回地址；当执行返回指令时，也自动地再装载这个地址。

⊜ 我们忽略库函数SQRT，它由QUADMEAN调用而且在执行之前先被链接。

给出的全局变量⊖。

程序清单7-1 一个FORTRAN77示例程序

```
      PROGRAM TEST
      COMMON MAXSIZE
      INTEGER MAXSIZE
      REAL TABLE(10),TEMP
      MAXSIZE = 10
      READ *, TABLE(1),TABLE(2),TABLE(3)
      CALL QUADMEAN(TABLE,3,TEMP)
      PRINT *, TEMP
      END

      SUBROUTINE QUADMEAN(A,SIZE,QMEAN)
      COMMON MAXSIZE
      INTEGER MAXSIZE,SIZE
      REAL A(SIZE),QMEAN, TEMP
      INTEGER K
      TEMP = 0.0
      IF ((SIZE.GT.MAXSIZE).OR.(SIZE.LT.1)) GOTO 99
      DO 10 K = 1,SIZE
         TEMP = TEMP + A(K)*A(K)
10    CONTINUE
99    QMEAN = SQRT(TEMP/SIZE)
      RETURN
      END
```

忽略存储器中整型值和浮点值间可能有的大小区别，我们显示了图7-1中的这个程序的运行时环境⊜。在该图中，我们用箭头表示从主过程中调用时，过程**QUADMEAN**的参数**A**、**SIZE**和**QMEAN**的值。在FORTRAN77中，参数值是隐含的存储引用，所以调用（**TABLE**、**3**和**TEMP**）的参数地址就被复制到**QUADMEAN**的参数地址中。它有几个后果。首先，需要一个额外的复引用来访问参数值。其次，数组参数无需再重新设置和复制(因此，只给在**QUADMEAN**中的数组参数**A**分配一个空间，在调用时指出**TABLE**的基地址)。再次，像在调用中的值3的常量参数必须被放在一个存储器地址中而且在调用时要使用这个地址(7.5节将更完整地讨论参数传递机制)。

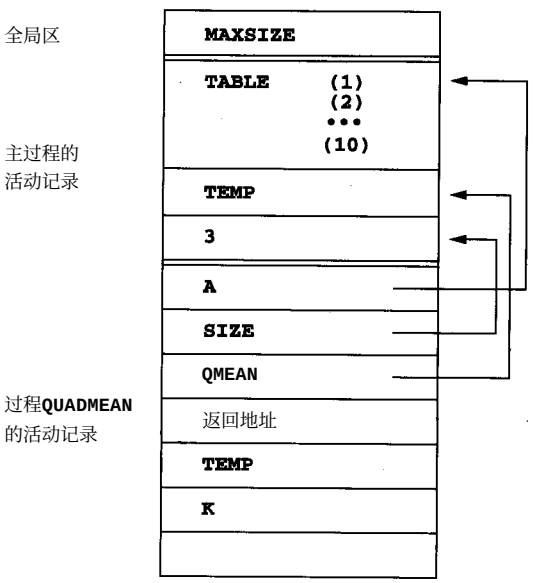


图7-1 程序清单7-1中程序的运行时环境

图7-1中还有一个特性需要解释一下，这

⊖ 实际上，FORTRAN77允许COMMON变量在不同的过程中具有不同的名称，而仍旧指的是相同存储器位置。从这个示例开始，我们将默认忽略这种复杂性。

⊜ 我们再次强调这个图示仅仅是示意性的。在操作中的实现实际与这里给出的有所差异。

就是QUADMEAN的活动记录结尾分配的未命名的地址。这个地址是一个在算术表达式计算中用来储存临时变量值的“凑合的”地址。在 QUADMEAN中可能需要两个运算。一个是循环中的TEMP+A(K)*A(K)的计算，另一个是当参数在调用SQRT时的TEMP/SIZE的计算。我们早已谈过需要为参数值分配空间(尽管在对库函数的调用中实际上是有差别的)。循环计算中也需要临时变量存储地址的原因在于每个算术运算必须在一个步骤中，所以就计算 A(K)*A(K)并在下一步中添加TEMP的值。如果没有足够的寄存器来放置这个临时变量值，或如果有一个调用要求保存这个值，那么在完成计算之前应先将值存储在活动记录中。编译程序可以总是先预测出在执行中它是否必要，然后再为分配临时变量的地址的恰当数量(以及大小)作出安排。

7.3 基于栈的运行时环境

在允许递归调用以及每一个调用中都重新分配局部变量的语言中，不能静态地分配活动记录。相反地，必须以一个基于栈的风格来分配活动记录，即当进行一个新的过程调用 (活动记录的压入(push))时，每个新的活动记录都分配在栈的顶部，而当调用退出时则再次解除分配(活动记录的弹出(pop))。活动记录的栈(stack of activation record)(也指运行时栈(runtime stack)或调用栈(call stack))就随着程序执行时发生的调用链生长或缩小。每个过程每次在调用栈上可以有若干个不同的活动记录，每个都代表了一个不同的调用。这样的环境要求的簿记和变量访问的技术比完全静态环境要复杂许多。特别地，活动记录中必须有额外的簿记信息，而且调用序列还包括设置和保存这个额外信息所需的步骤。基于栈的环境的正确性和所需簿记信息的数量在很大程度上依赖于被编译的语言的特性。在本节中为了提高难度复杂性，我们将考虑基于栈的环境的组织，它是由所涉及到的语言特性区分的。

7.3.1 没有局部过程的基于栈的环境

在一个所有过程都是全局的语言(例如C语言)中，基于栈的环境有两个要求：指向当前活动记录的指针的维护允许访问局部变量，以及位置记录或紧前面的活动记录 (调用程序的活动记录)允许在当前调用结束时恢复活动记录(且舍掉当前活动)。指向当前活动的指针通常称为框架指针(frame pointer) 或(fp)，且通常保存在寄存器中(通常也称作fp)。作为一个指向先前活动记录的指针，有关先前活动的信息一般是放在当前活动中，并被认为是控制链 (control link)或动态链(dynamic link) (之所以称之为动态的，是因为在执行时它指向调用程序的活动记录)。有时将这个指针称为旧**fp** (old fp)，这是因为它代表了fp的先前值。通常，这个指针被放在栈中参数区域和局部变量区域之间的某处，并且指向先前活动记录控制链。此外，还有一个栈指针(stack pointer)或sp，它通常指向调用栈上的最后位置(它有时称作栈顶部(top of stack)指针，或tos)。

现在考虑几个例子。

例7.2 利用Euclid算法的简单递归实现，计算两个非负整数的最大公约数，它的代码 (C语言)在程序清单7-2中。

程序清单7-2 例7-2的C代码

```
#include <stdio.h>

int x,y;
```

```
int gcd( int u, int v)
{ if (v == 0) return u;
  else return gcd(v,u % v);
}

main()
{ scanf("%d%d",&x,&y);
  printf("%d\n",gcd(x,y));
  return 0;
}
```

假设用户在该程序中输入了值15和10，那么main就初始化调用**gcd(15,10)**。这个调用导致了另一个递归调用 **gcd(10,5)**，(因为15 % 10 = 5)，而这又引起了第 3个调用**gcd(5,0)**(因为10%5=0)，它将返回值5。在第3个调用中，运行时环境如图7-2所示。请读者注意指向每个调用的**gcd**是如何向栈的顶部添加新的大小完全相同的活动记录，并且在每个新活动记录中，控制链指向先前活动记录的控制链。还请大家注意， fp指向当前活动记录的控制链，因此在下一个调用中当前的 fp就会变成下一个活动记录的控制链了。

调用最后一个**gcd**的之后，将按顺序从栈中删去每个活动，这样当在**main**中执行**printf**语句时，只在环境中保留了**main**和全局/静态区域的活动记录(我们已将**main**的记录显示为空。在实际中，它应包含将控制传回到操作系统的信息)。

最后，应指出在调用 **gcd** 时调用程序不需要为自变量值安排空间（与图 7-1中的FORTRAN77环境中的常量3不同），这是因为C语言使用值参数。7.5节将详细探讨这一点。

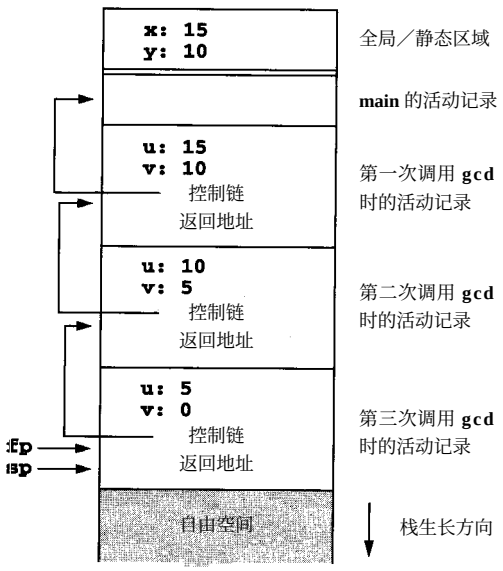


图7-2 例7.2的基于栈的环境

例7.3 考虑程序清单7-3中的C代码。这个代码包括将用来进一步描述本节相关内容的变量，但是它的基本操作如下所示。从**main**来的第1个调用是到**g(2)**的(这是因为**x**在这一点上有值2)。在这个调用中，**m**变成了2，而**y**则变成了1。接着**g**调用**f(1)**，而**f**也相应地调用**g(1)**。在这个到**g**的调用中，**m**变成了1，而**y**则变成了0，所以再也没有别的调用了。该点(在对**g**的第二次调用期间)上的运行时环境显示在图7-3a 中。

程序清单7-3 **例**7.3**的**C**程序**

```
int x = 2;

void g(int); /* prototype */

void f(int n)
{ static int x = 1;
```

```
    g(n);
    x--;
}

void g(int m)
{ int y = m--1;
  if (y > 0)
  { f(y);
    x--;
    g(y);
  }
}

main()
{ g(x);
  return 0;
}
```

现在对g和f的调用退出(f在返回之前它的静态局部变量x减1)，它们的活动记录从栈中弹出，且控制返回到紧随在第1次对g的调用的f的调用之后的点。现在g给外部变量x减1，并进行另一次调用g(1)，将m设为2将y设为1，这样就得到了图7-3b 中的运行时环境。在此之后再也没有调用了，从栈中就会弹出剩余的活动记录，程序退出。

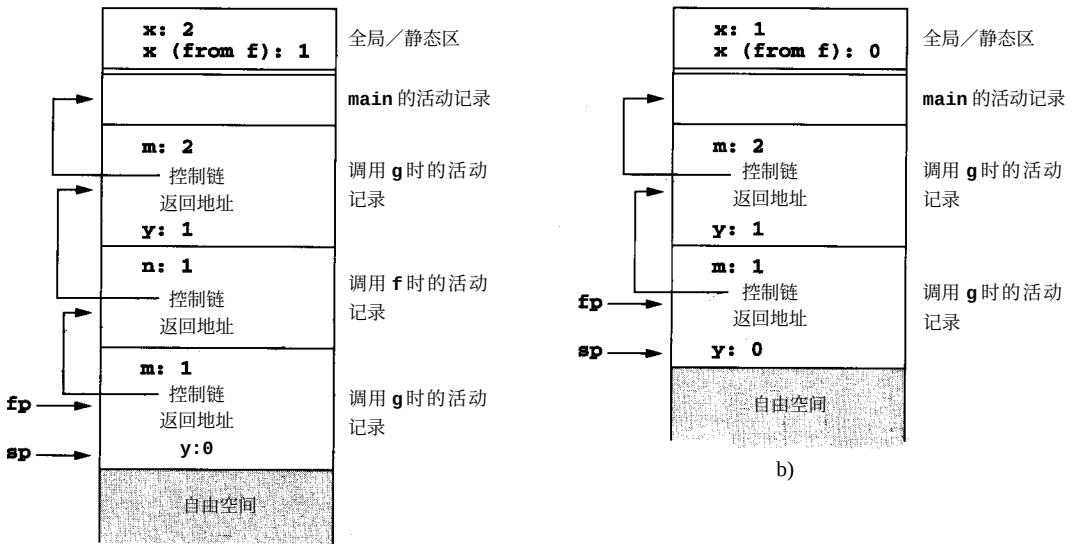


图7-3 程序清单7-3中程序的运行时环境

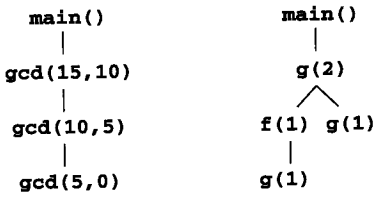
a) 在第2次对g 的调用时，程序清单7-3中程序的运行时环境

b) 在第3次对g 的调用时，程序清单7-3中程序的运行时环境

请注意，在图7-3b 中，对g的第3次调用的活动记录占据着(并覆盖)f的活动记录先前占着的存储器区域。大家还应注意到：由于f中的静态变量x必须坚持通过所有对f的调用，所以不可在f的活动记录中分配。因此，尽管不是全局变量，也必须在全局 /静态区域中与外部变量x

在一起分配。因为符号表会总能区分它与外部的x，并在程序每一个点上判定访问的正确变量，所以它们不会有什么混淆。

活动树(activation tree) 是分析程序中复杂调用结构的有用工具，每个活动记录 (或调用)都成为该树的一个节点，而每个节点的子孙则代表了与该节点相对应的调用时进行的调用。例如，程序清单 7-2中程序的活动树是线性的，在图 7-4a 中描述(对于输入15和10而言)，而程序清单7-3中程序的活动树在图7-4b 中描述。请注意，图 7-2和图7-3中显示的环境表示了在调用时由活动树的每个叶子代表的调用。一般而言，在特定调用开头的活动记录栈与活动树的相应节点到根节点的通路有一个结构等价。

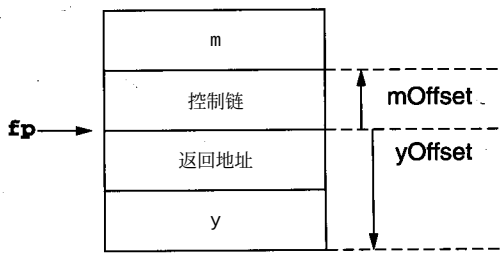


a) b)

图7-4 程序清单7-2和程序清单7-3中程序的活动树

1) **对名称的访问** 在基于栈的环境中，再也不能像在完全静态环境中那样用固定的地址访问参数和局部变量。而它们由当前框架指针的偏移量发现。在大多数的语言中，每个局部声明的偏移量仍是可由编译程序静态地计算出来，因为过程的声明在编译时是固定的，而且为每个声明分配的存储器大小也根据其数据类型而固定。

考虑程序清单7-3中C程序的过程g (参见图7-3中画出的运行时环境)。g的每个活动记录的格式完全相同，而参数 m和局部变量 y也总是位于活动记录中完全相同的相对位置。我们把这个距离称作 mOffset和yOffset。然后，在对 g的任何调用期间，都有下面的局部环境图：



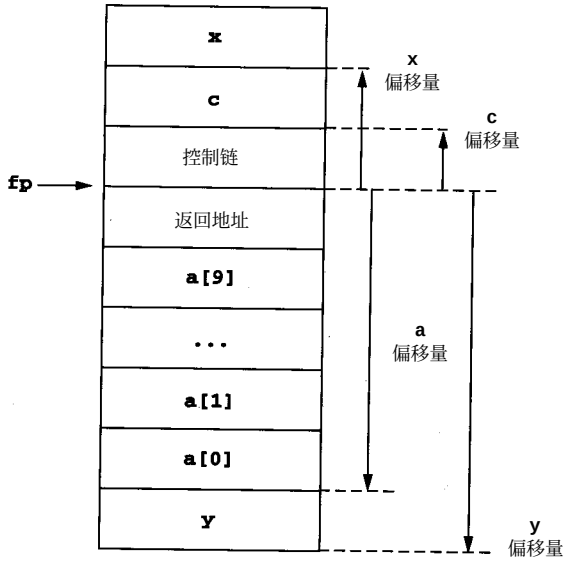
m和y都可根据它们从fp的固定偏移进行访问。例如，具体地假设运行时栈从存储器地址的高端向低端生长，整型变量的存储要求两个字节，地址要求 4个字节。若活动记录的组织如上所画，就有mOffset = +4和yOffset = -6，且对m和y的引用写成机器代码(假设是标准的汇编器约定)为4(fp)和-6(fp)。

局部数组和结构与简单变量相比分配和计算地址并不更加困难，如以下示例所示。

例7.4 考虑C过程

```
void f(int x, char c)
{ int a[10];
  double y;
  ...
}
```

对f调用的活动记录显示为：



而且假设整型是两个字节、地址4个字节、字符1个字节、双精度浮点数8个字节，那么就有以下的偏移值(再次假设栈的生成为负方向)，这些值在编译时都是可计算的：

名 称	偏 移 量
X	+5
C	+4
A	-24
Y	-32

现在对**a[i]**一个访问将要求计算地址

```
(-24+2*i)(fp)
```

(这里在产生式**2*i**中的因子是比例因子(scale factor)，它是从假设整型值占有两个字节得来的)。这样的存储器访问依赖于**i**的地址以及体系结构，可能只需要一条指令。

对于这个环境中的非局部的和静态名字而言，不能用同局部名字一样的方法访问它们。实际上，我们此处所考虑的情况——不带有过程的语言——所有的非局部的名字都是全局的，因此也就是静态的。所以在图7-3中，外部的(全局的)C变量**x**具有一个固定的静态地址，因此也就可以被直接访问(或是通过某个基指针的偏移而不是fp)。对来自**f**的静态局部变量**x**的访问也使用完全相同的风格。请注意，正如前一章所描述的，这个机制实现静态 (或词法的)作用域。如果需要动态作用域，那么就要使用一个更为复杂的访问机制(本节后面将要提到)。

2) **调用序列** 调用序列大致由以下的步骤组成⊖。当调用一个过程时，

① 计算自变量并将其存放在过程的新活动记录中的正确位置 (将其妥当地压入到运行时栈中就可做到这一点)。

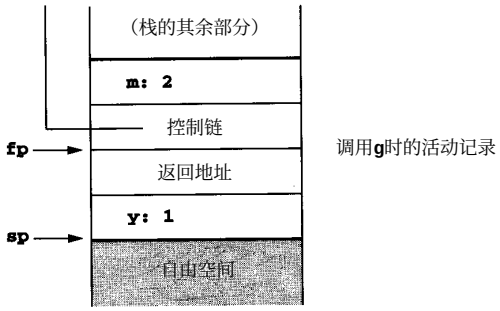
② 将fp作为控制链存放(压入)到新的活动记录中。

③ 改变fp以使其指向新的活动记录的开始(如已有了一个sp，则将该sp复制到该点上的fp中，也可以做到这一点)。

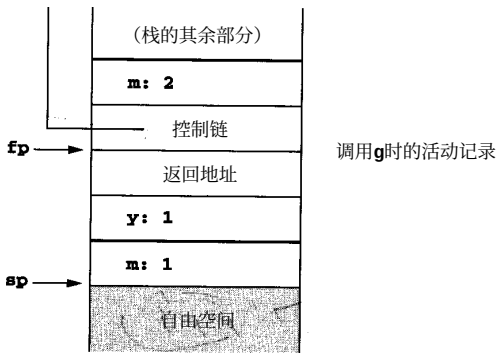
⊖ 这个描述忽略了必须发生的寄存器的任何保存。它还忽略了将返回值放在一个可用的地址的需要。

④ 将返回地址存放在新的活动记录中(如果需要)。
⑤ 完成到被调用的过程的代码一个转移。
当存在着一个过程时，则
① 将fp复制到sp中。
② 将控制链装载到fp中。
③ 完成到返回地址的一个转移。
④ 改变sp以弹出自变量。

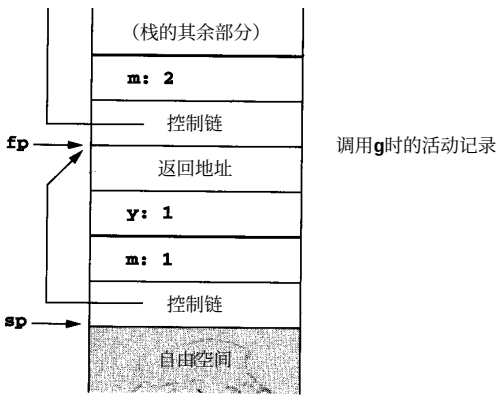
例7.5　考虑在前面图7-3b 中的对**g**的最后一个调用的情况：



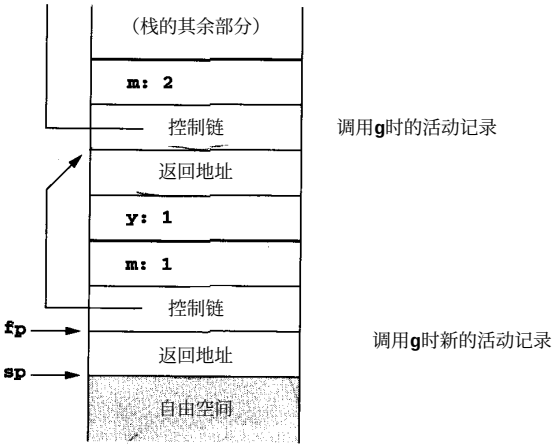
当进行对**g**新的调用时，首先将参数**m**的值压入到运行时栈中：



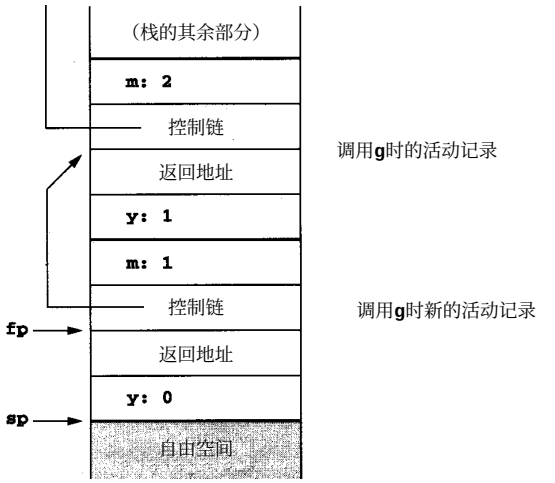
接着将fp压入到栈中：



现在将sp复制到fp中，并将返回的地址压入到栈中，就得出了到**g**的新的调用了：



最后，**g**在栈上分配和初始化新的**y**以完成对新的活动记录的构造：



3) **处理可变长度数据** 到这里我们已经描述了一种情况，所有的数据，无论是局部的还是全局的，都可在一个固定的地方，或由编译程序计算出的到 fp的固定偏移处找到。有时编译程序必须处理数据变化的可能性，表现在数据对象的数量和每个对象的大小上。发生在支持基于栈的环境的语言中的两个示例如下：① 调用中的自变量的数量可根据调用的不同而不同。② 数组参数或局部数组变量的大小可根据调用的不同而不同。

情况(1)的典型例子是C中的**printf**函数，其中的自变量的个数由作为第一个自变量传递的格式串决定。因此：

```
printf("%d%s%c", n, prompt, ch);
```

就有4个自变量(包括格式串**"%d%s%c"**)，但是

```
printf("Hello, world\n");
```

却只有一个自变量。通常， C编译程序一般通过把调用的自变量按相反顺序 (in reverse order)压入到运行时栈p来处理这一点。那么，在上面描述的实现中第 1个参数(它告知**printf**的代码

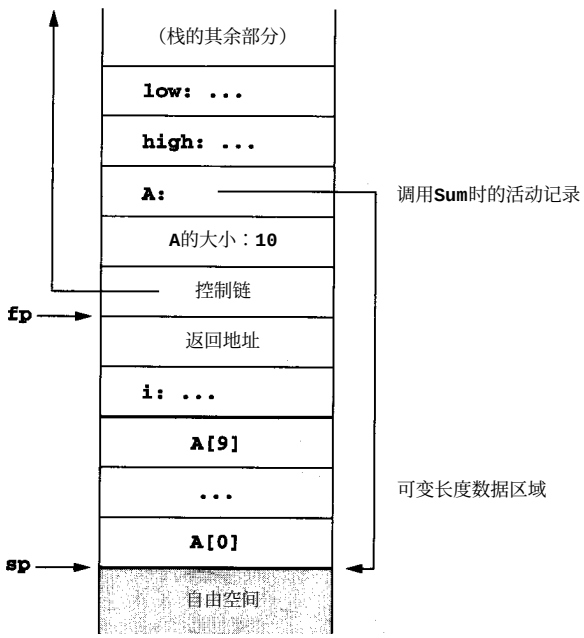
共有多少参数)通常是位于到fp的固定偏移的位置(使用上一个示例的假设得出实际上是+4)。另一个选择是使用一个在VAX体系结构中的诸如ap (自变量指针) 的处理器机制。还会对这以及其他的可能情况在练习中再进一步讨论。

Ada非约束数组 (unconstrained array)是情况(2)的一个示例：

```
type Int_Vector is
     array(INTEGER range <>) of INTEGER;

procedure Sum (low, high: INTEGER;
               A: Int_Vector) return INTEGER
is
   temp: Int_Array (low..high);
begin
  ...
end Sum;
```

(请注意局部变量**temp**，其大小不可预测)。处理这种情况的典型办法是：为变量长度数据使用间接的额外层，并将指针存放到一个在编译时可预测的地址中的实际数据里，同时执行期间用sp可管理的方法在运行栈的顶部进行真正的分配。

例7.6　给定前面定义的Ada **Sum**过程，假定环境的组织也同上面一样⊖，那么可以如下所示实现**Sum**的活动记录 (这个图示具体地显示了一个当数组大小为10时对**Sum**的调用)：



现在，对**A[i]**的访问就可由计算

```
@6 (fp) +2*i
```

得到。其中@意味着间接，且此时仍假设整型两个字节，地址4个字节。

注意，在上例所描述的实现中，调用程序必须知道**Sum**的任何活动记录的大小。而且编译

⊖ 这在Ada中实际是不够的，它将会导致嵌套过程。参见本节后面的讨论。

程序也了解在调用点上的参数部分和簿记部分的大小 (这是因为可以计算出自变量大小，而簿记部分与所有的过程都相同)，但是一般而言，却不知道调用点上的局部变量部分的大小。因此，这个实现就要求编译程序为每个过程预先计算出局部变量的大小并将其存放在符号表中以备后用。用类似的方法可处理可变长度局部变量。

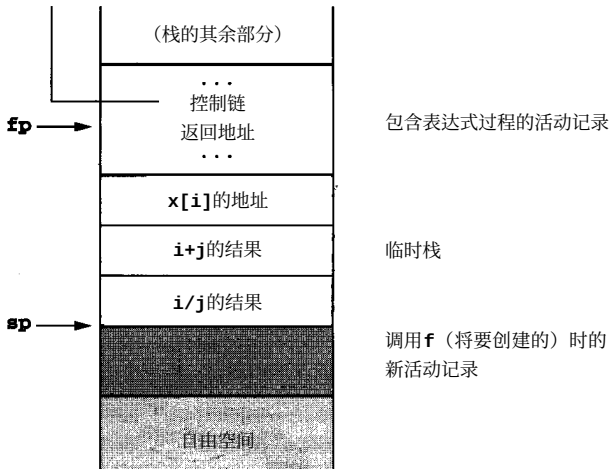
我们还需提醒大家C数组并不属于这类可变长度数据。实际上， C数组是一些指针，所以数组参数是由引用在C中传递且并不局部分配(而且它们不带有任何尺寸信息)。

4) **局部临时变量和嵌套声明** 基于栈的运行时环境还有两个需要提及的复杂问题：局部临时变量和嵌套声明。

过程调用时必须保存的计算是导致局部临时变量的部分原因。例如考虑C表达式：

```
x[i] = (i + j)*(i/k + f(j))
```

在这个表达式从左到右的求值计算中，在对 **f**的调用过程中需要保存中间结果： **x[i]**的地址(未决赋值)、计算**i+j**的和(加法的未决)以及**i/k**的商(和与**f(j)**的调用结果未决)。这些中间值可计算到寄存器中，并根据某个寄存器管理机制进行保存和恢复，或者可将它们作为临时变量存储在对**f**调用之前的运行时栈中。在这后一种情况下，运行时栈可能会出现在对 **f**的调用之前的点上，如下所示：



在这种情况下，先前描述的利用 sp的调用序列并未改变。此外，编译程序还可以很便利地从fp计算出栈顶的位置(在缺少变量长度数据时)，这是因为临时变量所要求的数量是编译时决定的量。

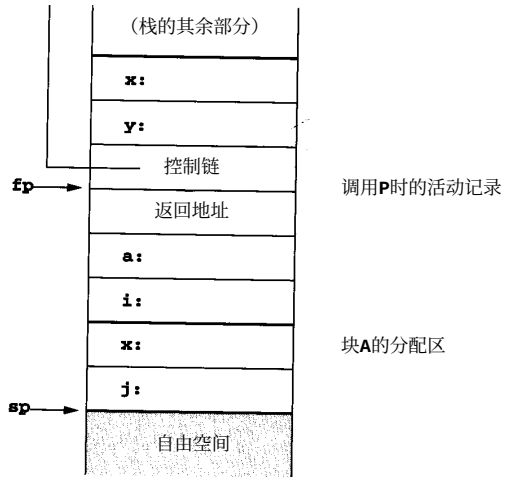
嵌套声明也出现了类似的问题。考虑以下的C代码

```
void p( int x, double y)
{ char a;
  int i;
  ...
A:{ double x;
    int j;
    ...
  }
  ...
B:{ char * a;
```

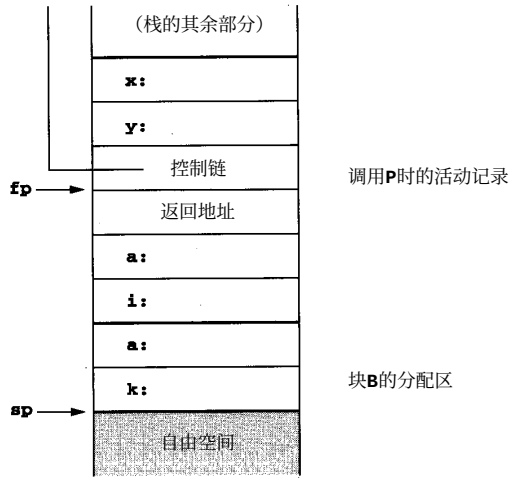
```
        int k;
        ...
      }
      ...
    }
```

在这个代码中，在过程p的主体中嵌套着两个分别标作A和B的块(也称作复合语句)，它们每个都有两个作用域仅仅覆盖着其所在块 (也就是说向上直到下一个闭合的括号)的局部声明。在块进入之前无需对这些块的局部声明进行分配，而且块 A和块B也无需同时进行分配。编译程序能够像对待过程一样处理块，并且在每次进入块时创建新的活动记录，并在退出时抛弃它。然而，由于这样的块比过程简单得多，所以它的效率并不高：这样的块没有参数且无返回地址，而且总是立即被执行而不是从其他地方调用。一个更简单的方法是按照与临时表达式相类似的办法在嵌套的块中处理声明，并在进入块时在栈中分配它们而在退出时重新分配。

例如，在上面所给出的简单C代码中进入块A之后，运行时栈应如下所示：



当进入块B后，则如下所示：



这样的实现必须这样小心地分配嵌套声明，周围过程块的 fp的偏移在编译时计算。特别是，这

样的数据必须要在任何变量长度数据之前进行分配。例如在上面才给出的代码中，位于块 **A**上的变量**j**从**p**的fp的偏移是 - 17 (再次假设整型2个字节，地址4个字节，浮点实数有个字节，而字符是1个字节)，块**B**中的**k**的偏移是 - 13。

7.3.2 带有局部过程的基于栈的环境

如果在语言编译时允许有局部过程声明，那么前面所讲到的运行时环境就无效了，因为没有提供非局部的和非全局的引用。

例如，考虑程序清单7-4中的Pascal代码 (在Ada中也可以写出类似的程序)：在对**q**的调用中，运行时环境如图 7-5所示。当使用标准的静态作用域规则时，在 **q**中每次提及**n**必须指的是**p**的局部整型变量**n**。正如我们在图7-5中所看到的一样，使用至今为止保存在运行时环境中的任何簿记信息都无法找到这个**n**。

程序清单7-4 Pascal**程序显示非局部的非全局的引用**

```
program nonLocalRef;

procedure p;
var n: integer;

      procedure q;
      begin
        (* a reference to n is now
           non-local non-global *)
      end; (* q *)

      procedure r(n: integer);
      begin
        q;
      end; (* r *)

begin (* p *)
  n := 1;
  r(2);
end; (* p *)

begin (* main *)
  p;
end.
```

若我们愿意接受动态作用域，那么利用控制链有可能找到 **n**。观察图7-5，看到通过跟随控制链就可以找到在 **r**的活动记录中的这个 **n**，而且若 **r**没有**n**的说明，则可以通过跟随另一个控制链来找到**p**的**n** (这个处理称为链接 (chaining)，我们很快还会看到这个方法)。不幸的是，不仅仅是这个实现是动态作用域，可以找到 **n**的偏移也会随着调用的不同而不同 (请注意，在**r**中的**n**与在**p**中的**n**具有不同的偏移)。因此，在这样的实现中，必须在执行时保存用于每个过程的局部符号表，这样才能允许在每个活动记录中查询标识符，以及若它退出的话也可以看到，并且可以判定出它的偏移。这是运行时环境的最主要的额外复杂性。

解决这个问题的方法也实现静态作用域，是将一个称作访问链 (access link)的额外簿记信息添加到每个活动记录中。除了可以指向代表过程的定义环境而不是调用环境之外，访

问链与控制链相似。正是由于这个原因，即使它不是编译时决定的量，访问链有时也被称作静态链 (static link)⊖。

图7-6显示了将图7-5中的运行时栈修改之后包括了访问链的情况。在这个新的环境之下，r和q的活动记录的访问链都指向p的活动记录，这是r和q都在p中声明的缘故。因为这总是p的一个活动记录，现在位于q中的对n的引用会引起后接访问链，这里n可在固定偏移处找到。通常，通过将访问链装载到寄存器中，然后根据到这个寄存器(它此时作为fp)的偏移访问n，而在代码中完成。例如，使用前面描述的大小约定，若寄存器r用作访问链，则在用值4(fp)装载r之后可将p中的n作为-6(r)来访问(访问链从图7-6中的fp得到偏移+4)。

注意，正如由它将要到达的位置上的括号中的注解所指出的，过程p的活动记录本身并未包含有访问链。这是因为p是一个全局过程，所以p中的任何非局部的引用必须都是全局引用且通过全局引用机制来访问。因此，访问链就是多余的了(实际上为了与其他的过程保持连贯性，可以很便利地插入空的或是任意的访问链)。

上面讨论的情况实际是最简单的，其中非局部引用是指向下一个最外面的作用域中的声明。而指向最远的作用域中的声明的非局部引用也是可能的。例如在程序清单7-5中的代码。

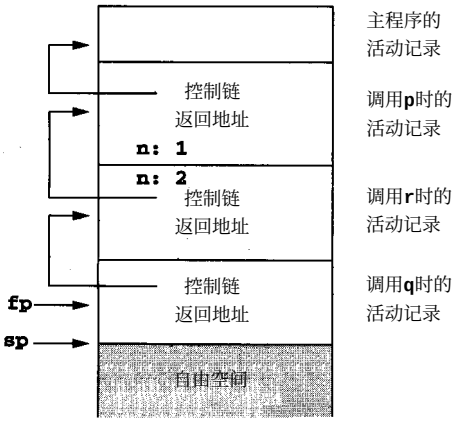


图7-5 程序清单7-4中程序的运行时栈

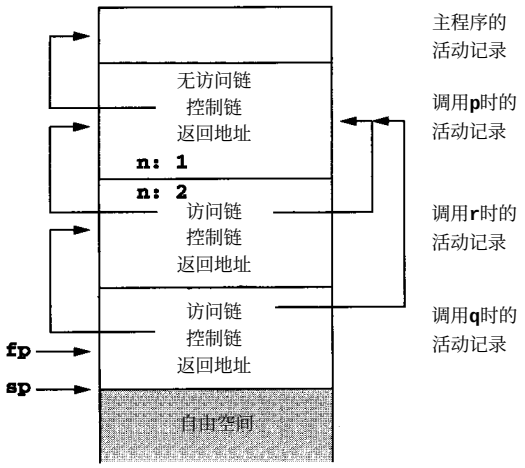


图7-6 添加了访问链后的程序清单7-4中程序的运行时栈

程序清单7-5 示范访问链的Pascal代码

```
program chain;

procedure p;
var x: integer;

   procedure q;
      procedure r;
      begin
        x := 2;
        ...
        if ... then p;
      end; (* r *)
```

⊖ 我们当然了解定义过程，但却不知道它的活动记录的确切位置。

```
    begin
      r;
    end; (* q *)

begin
  q;
end; (* p *)

begin (* main *)
  p;
end.
```

在这段代码中，在过程**q**中说明了过程**r**，而过程**q**又是在过程**p**中说明的。因此，对于**r**中的**x**的赋值 (即**p**的**x**) 必须越过两个作用域层去寻找**x**。图7-7显示了在对**r**的 (第1个) 调用之后的运行时栈 (由于**r**可能递归地调用**p**，所以到**r**的调用可能不止一个)。在这种环境中，必须跟随两个访问链才能到达**x**，这个过程称作访问链接 (access chaining)。访问链接是通过重复地取出访问链实现的，利用前面取出的链好像它就是fp。图7-7中的**x**可如下进行访问 (使用前面的大小约定)：

Load 4(fp) into register r.

Load 4(r) into register r.

Now access **x** as -6(r).

对于用于访问链工作的方法，编译程序必须能够在局部访问名字之前判定出要链接多少个嵌套层。这就要求编译程序预先为每个声明计算出嵌套层 (nesting level) 属性。通常，将最远的作用域 (Pascal中的主程序层或是C中的外部作用域) 给定为嵌套层0，每次进入一个函数或过程 (在编译时)，嵌套层就增加1，退出时则减去1。例如，在程序清单7-5的代码中，由于过程 **p**是全局的，所以它的嵌套层为0；由于在进入**p**时嵌套层增加了，所以变量**x**的嵌套层为1；由于过程**q**对于**p**是局部的，所以它的嵌套层也为 1；而过程**r**的嵌套层为2的原因是当进入**q**时嵌套层再次增加了。最后，在**r**内的嵌套再一次增加到3。

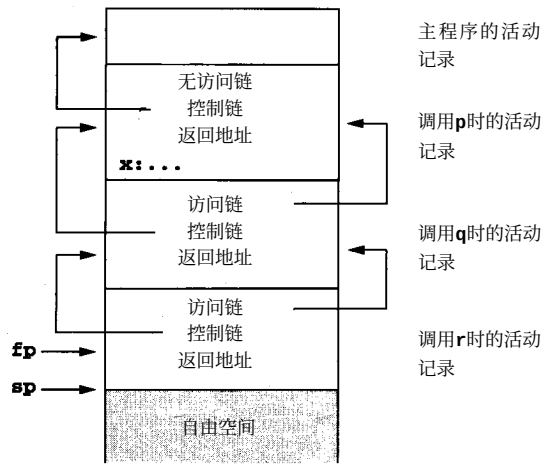


图7-7 程序清单7-5代码中第1次对**r**的调用之后的运行时栈

现在通过比较在访问点上的嵌套层与名字声明的嵌套层，可判断出访问非局部名字所必须的链接的数量；后面跟随的访问链接数是这两个嵌套层的差。例如，在前面的情况中，对 **x**的赋值发生在嵌套层3，而**x**具有嵌套层1，所以必须跟在两个访问链接之后。一般而言，若在嵌套层中的差是 ***m***，那么为访问链接而必须生成的代码须将***m*** 装入到一个寄存器r上，且使第1个使用fp，其他的使用r。

由于必须为每个带有大的嵌套差的非局部引用执行一个很长的指令序列，所以对于变量访问访问链接看起来效率不高。但在实际运用中，嵌套层极少有超过两个到 3个深度的，而且大多数的非局部引用都是对全局变量的 (嵌套层为0)，这样就可利用前面所讲到的直接办法对它们继续访问了。在嵌套层索引的查询表中有一个实现访问链接方法，链接不会带来执行开销。该方法中所用到的数据结构称作显示 (display)。它的结构与使用在练习中论述。

1) **调用序列** 实现访问链接的调用序列的改变相对比较简单。在实现中，调用时必须将访问链压入到fp之前的运行时栈中，退出之后必须用一个额外的量来修改 sp以便像对自变量一样删掉访问链接。

唯一的问题是在调用时寻找过程的访问链接。通过使用附在正调用的过程声明之上的 (编译时)嵌套层信息可以解决这个问题。实际上，我们所要做的仅是生成一个访问链接，就像在过程调用的同一嵌套层上访问一个变量。这样计算所得出的地址就是相应的访问链。当然若过程是局部的 (在嵌套层中的差是 0)，那么访问链与控制链就是相同的(而且也与在调用点上的fp相同)。

例如可考虑程序清单 7-4上的**r**中对**q**的调用。在**r**内，位于嵌套层 2，而**q**的声明在嵌套层1中 (这是因为**q**对于**p**而言是局部的，而在**p**内的嵌套层是 1)。因此，一个访问步骤就要求计算**q**的访问链，而在图 7-6中，**q**的访问链指向**p**的活动记录 (且与**r**的访问链相同)。

请注意，即使是位于定义环境的多重活动中，这个过程也将计算出正确的访问链，因为计算是在运行时而不是在编译时进行的 (利用编译时嵌套层)。例如，假设有程序清单7-5的代码，在对**r**的第2个调用后 (假定是对**p**的递归调用)，运行时栈如图7-8所示。在该图中，**r**有两个不同的活动记录，带有两个不同的访问链，指向**q**的不同的活动记录，代表**r**不同的定义环境。

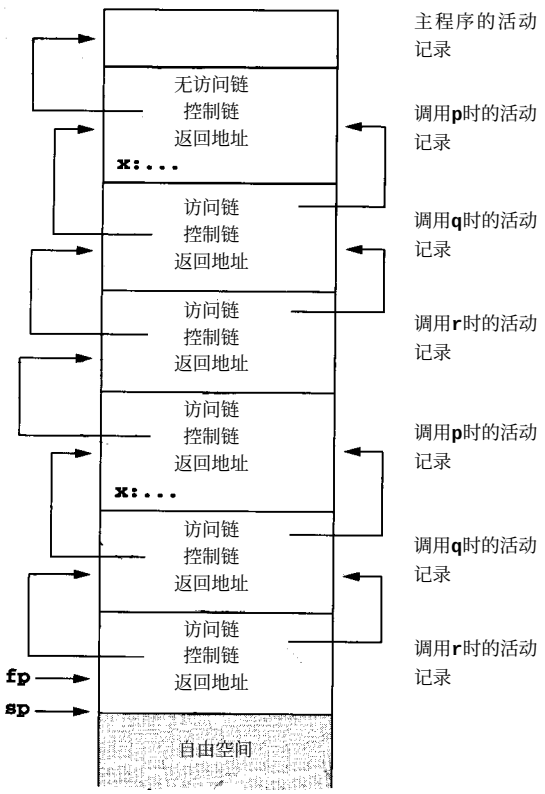


图7-8 在程序清单7-5的代码中对**r**的第2次调用之后的运行时栈

7.3.3 带有过程参数的基于栈的环境

在某些语言中，不仅允许有局部过程，而且还可将过程作为参数传递。在这样的语言中，当调用一个作为参数传递的过程时，编译程序不可能像前一节所讲的那样生成代码以计算调用点上的访问链。当将过程作为参数传递时，必须预先计算出过程的访问链并与过程代码的指针一同传递。因此，再也不能将过程参数值看作是一个简单的代码指针了，它应包含一个访问指针，定义解决非局部引用的环境。这个指针对，一个代码指针和一个访问链，或一个指令指针 (instruction pointer) 和环境指针 (environment pointer)，一同表示了过程或函数参数的值，它们通称为闭包 (closure)(这是因为访问链“ 闭合了 ”由非局部引用引起的“ 洞 ”)⊖。我们将闭包表示为<ip, ep>，其中ip表示过程的指令指针 (代码指针或入口点)，而ep表示过程的环境指针 (访问链)。

⊖ 这个术语在λ微积分中有它自己的来源，且它不会与正则表达式或 NFA状态中的ε 闭包的 (Kleene) 闭包运算相混淆。

例7.7 考虑程序清单7-6中的标准Pascal程序，它有一个过程**p**，带有一个也是过程的参数**a**。在**q**中对**p**调用之后，**q**的过程**r**传递到**p**，**p**中的对**a**的调用实际上调用的是**r**，而且这个调用仍必须在**q**的活动中寻找非局部变量**x**。当调用**p**时，将**a**构造为闭包<ip, ep>，其中ip是指向**r**的代码的指针，而ep是在调用点fp的拷贝（也就是它指向调用**q**的环境，其中定义了**r**）。**a**的ep的值由图7-9的虚线指明，表示在**q**中的调用**p**之后的环境。接着当在**p**内调用**a**时，就将**a**的ep用作其活动记录的静态链，如图7-10所示。

程序清单7-6 **带有作为参数的过程的标准**Pascal**代码**

```
program closureEx(output);

procedure p(procedure a);
begin
  a;
end;

procedure q;
var x:integer;

      procedure r;
      begin
        writeln(x);
      end;

begin
  x := 2;
  p(r);
end; (* q *)

begin (* main *)
  q;
end.
```

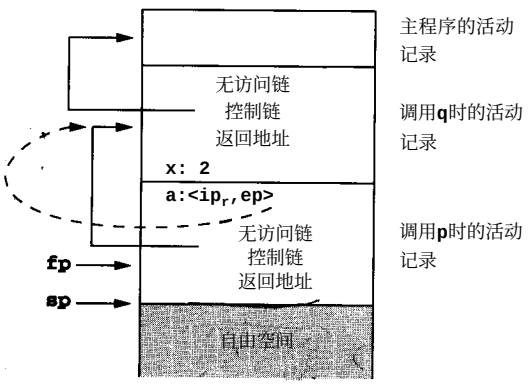


图7-9 程序清单7-6的代码中调用p之后的运行时栈

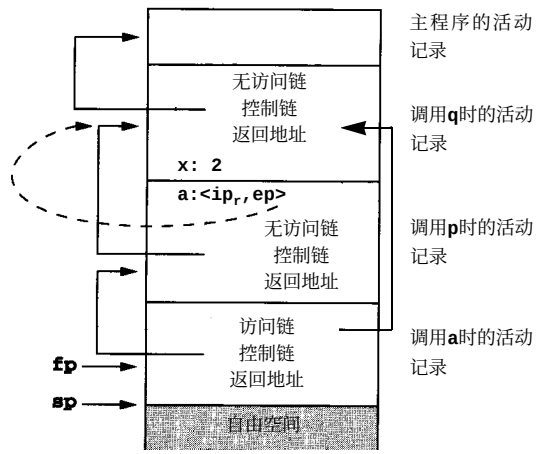


图7-10 程序清单7-6的代码中调用a之后的运行时栈

现在，刚刚描述的环境中的调用序列能很清楚地区分常规过程和过程参数。同前面一样，

常规过程调用是使用过程的嵌套层取出访问链，并直接跳到过程的代码 (在编译时已知)。而另一方面，过程参数早已得到了它的访问链，并已将其存放在局部活动记录中，而这一记录又必须取出并插入到新的活动记录中。然而，编译程序却无法直接得到过程代码的地址；而必须对存放在当前活动记录中的ip进行间接调用。

为了保持简洁和一致性，编译程序的编写者可能会希望要避免常规过程与过程参数之间的这种区别，并将所有的过程都作为环境中的闭包。实际上，若语言对过程的处理越普通，则这个方法就越合理。例如，若允许有过程变量，或可以动态地计算过程变量，则过程的<ip, ep>表示就变成了对这种情形的要求了。图7-11显示了当所有的过程值都存放在作为闭包的环境中时图7-10的环境。

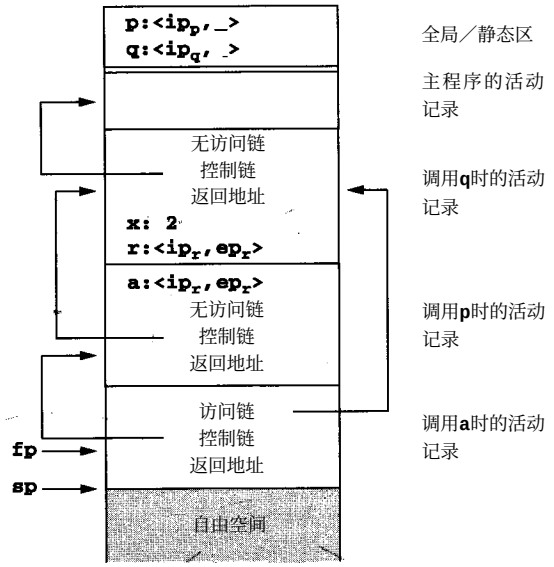


图7-11 程序清单7-6的代码调用a之后的运行时栈，所有的过程都作为环境中的闭包

最后，我们注意到C、Modula-2和Ada都避免本节所提到的复杂情况：对于C而言，它没有局部过程(即使它有参数和变量)；对于Modula-2而言，是一个特殊的规则限制了过程参数和过程变量值都应是全局过程；而对于Ada而言，则是由于它没有过程参数和变量。

7.4 动态存储器

7.4.1 完全动态运行时环境

上一节讨论的基于运行时环境的栈在C、Pascal以及Ada这样的标准命令式语言中是最普通的环境格式。但这样的环境也有限制。尤其是如果一种语言在过程中对局部变量的引用可返回到调用程序，无论是显式的还是隐含的，在过程退出时的基于栈的环境都会导致摇摆引用(dangling reference)，这是因为过程的活动记录将从栈中释放分配。最简单的示例是返回局部变量的地址，如在C代码中：

```
int * dangle(void)
{ int x ;
  return &x;}
```

现在赋值**addr = dangle()**使**addr**指向活动栈中的不安全的地址，它的值可由后面对任何过程的调用随机改变。C对此类问题的处理是，只说明这样的程序是错误的(尽管没有哪个编译程序会给出错误信息)。换而言之，C的语义被建立在基于栈的环境之下。

若调用可返回局部函数，则会发生更为复杂的摇摆引用情况。例如，如 C允许有局部函数定义，则程序清单7-7的代码就会出现一个对x和g的参数的间接摇摆引用，在**g**退出后调用**f**就可访问到它了。当然C是通过禁止局部过程的存在来避免这个问题的。诸如 Modula-2的其他语言既有局部过程也有过程变量、参数和返回值，就必须定出一个特殊规则使程序报错 (在Modula-2中，该规则是仅有全局过程才可以是自变量或返回值——这甚至是从 Pascal风格过程

变量的主要退步)。

程序清单7-7 显示返回局部函数引起的摇摆引用的伪C代码

```
typedef int (* proc)(void);

proc g(int x)
{ int f(void) /* illegal local function */
  { return x;}
  return f; }

main()
{ proc c;
  c = g(2);
  printf("%d\n",c()); /* should print 2 */
  return 0;
}
```

但是在很多语言中，这样的规则并不适用，即那些像LISP和ML的函数程序设计语言。设计函数语言的一个主要原则是函数应尽可能的通用，而这就意味着函数应是能够局部定义的，并像参数一样传递，作为结果返回。因此，对于此类的语言而言，基于栈的运行时环境并不合适，而且需要一个更一般的环境格式。因为活动记录仅在对它们所有的引用都消失了才再重新分配，而且这又要求活动记录在执行时可动态地释放任意次，所以称这个环境为完全动态的(fully dynamic)。因为完全动态运行时环境包含了要在执行时跟踪引用，以及在执行时任意次地找寻和重新分配存储器的不可访问区域(这种处理称作废弃单元收集 (garbage collection))，所以这种环境比基于栈的环境要复杂许多。

尽管在这个环境中增加的了复杂性，其活动记录的基本结构仍保持不变：必须为参数和局部变量分配空间，而且仍然需要控制链和访问链。当然，现在当控制返回到调用程序时 (且使用控制链来恢复先前的环境)，退出的活动记录仍留在存储器中，而且在以后的某个时刻被重新分配。因此这个环境的整个额外的复杂性可被压缩到存储器管理程序中，这个管理程序将取代带有更普通的分配和重新分配例程的运行时栈操作。本节后面还会谈到一些有关这样的存储器管理程序的设计问题。

7.4.2 面向对象的语言中的动态存储器

面向对象的语言在运行时环境中要求特殊的机制以完成其增添的特征：对象、方法、继承以及动态装订。这一小节将给出有关这些特征的各种实现技术。我们假设读者对于基本的面向对象的技术和概念比较熟悉[⊖]。

面向对象语言在对运行时环境方面的要求差异很大。Smalltalk和C++是这种差异的极好的代表者：Smalltalk要求与LISP相似的完全动态环境；而C++则在设计上花了很大的功夫以保持C的基于栈的环境，它并不需要自动动态存储器管理。在这两种语言中，存储器中的对象可被看作是传统记录结构和活动记录之间的交叉，且带有作为记录域的实例变量 (数据成员)。这个结构与传统记录在对方法和继承特征的访问上有着一定的差别。

实现对象的一个简单机制是，初始化代码将所有当前的继承特征 (和方法)直接地复制到记

⊖ 以后的讨论也假设只有继承性是可用的。“注意与参考”部分中的某些著作还谈到了多重继承性。

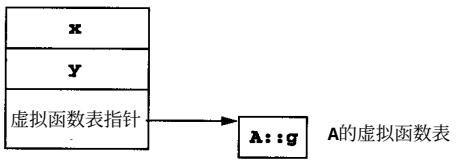
录结构中(将方法当作代码指针)。但这样做极浪费空间。另外一种方法是在执行时将类结构的一个完整的描述保存在每个点的存储器中，并由超类指针维护继承性 (有时这也称作继承图(inheritance graph))。接着同用于它的实例变量的域一起，每个对象保持一个指向其定义类的指针，通过这个类就可找到所有 (局部和继承的)的方法。此时，只记录一次方法指针 (在类结构中)，而且对于每个对象并不将其复制到存储器中。由于是通过类继承的搜索来找到这个机制的，所以该机制还实现继承性与动态联编。其缺点在于：虽然实例变量具有可预测的偏移量(正如在标准环境中的局部变量一样)，方法却没有，而且它们必须由带有查询功能的符号表结构中的名字维护。然而，它是对于诸如Smalltalk的高度动态语言的合理的结构，其中对于类结构的改变可以发生在执行中。

将整个类结构保存在环境中的另一种方法是，计算出每个类的可用方法的代码指针列表，并将其作为一个虚拟函数表 (virtual function table)(C++术语) 而存放在(静态)存储器。它的优点在于：可做出安排以使每个方法都有一个可预测的偏移量，而且也就不再需要用一系列表查询遍历类的层次结构。现在每个对象都包括了一个指向相应的虚拟函数表而不是类结构的指针(当然，这个指针的位置必须也有可预测的偏移量)。这种简化仅在类结构本身是固定在执行之前的情况下才成立。它是C++中选择的方法。

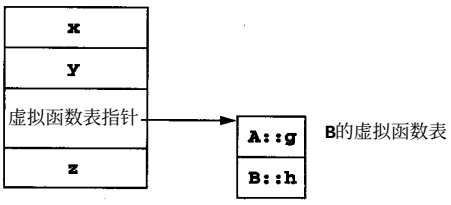
例7.8 考虑以下的C++类声明：

```
class A
{ public:
  double x,y;
  void f();
  virtual void g();
};
class B:public A
{ public:
  double z;
  void f();
  virtual void h();
};
```

类**A**的一个对象应出现在存储器中(带有它的虚拟函数表)，如下所示：



而类**B**的一个对象则应如下所示：



每次增添对象结构时，注意虚拟函数指针如何保留固定的地址，这样就可在执行之前知道

它的偏移量。还应注意(由于函数**f**没有声明“虚拟”)，它并不遵守C++动态联编，因此也就不出现在虚拟函数表 (或环境中的任何其他地方)：在编译时决定对f的调用。

7.4.3 堆管理

在7.4.1节中，我们讨论了如果完全支持一般的函数，那么与在大多数编译语言使用的基于栈的运行时环境比，更需要具有动态性。但在绝大多数语言中，即使是基于栈的环境也需要一些动态功能以处理指针分配和重新分配。处理这样的分配的数据结构称作堆，堆通常作为存储器中的一个线性块分配，这样如果需要它还可以生长，而且对栈的干扰尽可能小 (7.1节已显示了堆位于栈区域的相反一端的存储器块中)。

本章一直到这里，我们的注意力都是放在活动记录和运行时栈的组织上。而在本节中，我们希望描述一下如何管理堆，以及如何将堆操作扩展，提供带一般函数功能的语言要求的动态分配。

堆提供两个操作：分配操作和释放操作。分配操作通常是按字节数得到一个大小参数 (或显式或隐含)，并返回一个指向正确大小的存储器块的指针，或若不存在则返回一个空指针。释放操作得到一个指向被分配的存储器块的指针并再次将它标为空的 (释放操作必须还能通过或显式或隐含的参数来发现将空的块的大小。)这两个操作存在于许多语言的不同名称之下：在Pascal分别称作**new**和**dispose**，在C++中称作**new**和**delete**。C语言中有这些操作的若干个版本，但最基本的是**malloc**和**free**，它们都是标准库 (**stdlib.h**) 的一部分，此时它们基本都有以下的声明：

```
void * malloc (unsigned nbytes);
void free (void * ap);
```

我们将用这些声明作为堆管理的基本描述。

维护堆和实现这些函数的标准方法是使用空块的环形链接列表， **malloc**从中得到存储器而**free**返回存储器。它具有简单的优点，但也有缺点：其一， **free**操作不能辨认出它的指针自变量是否是它真正指向的由 **malloc**先前分配的合法块。若用户传递了一个无效的指针时，则堆就会很容易和很快坏掉。其二 (问题并不很严重)是必须注意，返回处附近有空的块列表时要合并 (coalesce)块，因为这样会导致最大的空块。若不合并，堆就会很快变成碎片 (fragmented)，也就是被分割成大量的较小的块，这样尽管有足够多的全部可用空间可用于分配，但分配大块时却失败了(在合并中当然也可能有碎片)。

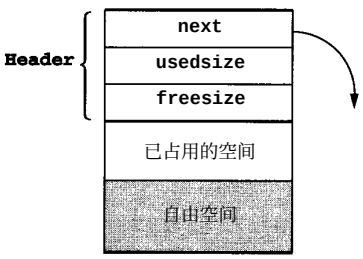
在这里我们指出使用环形链接的列表数据结构的 **malloc**和**free**在实现上的一点差异，这个数据结构保存分配的和空的块 (因此也就不易坏掉)，也提供在自合并块方面的优点。程序清单7-8给出了代码。

程序清单7-8 维护邻近的存储器的C代码，使用指向使用块和空块的指针

```
#define NULL 0
#define MEMSIZE 8096 /* change for different sizes */

typedef double Align;
typedef union header
  { struct { union header *next;
             unsigned usedsize;
             unsigned freesize;
           } s;
    Align a;
  } Header;
```

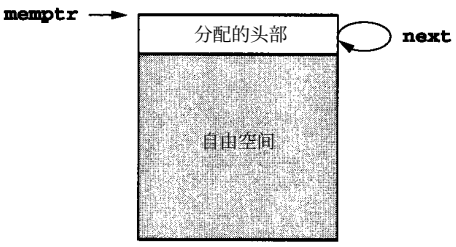
```
static Header mem[MEMSIZE];
static Header *memptr = NULL;

void *malloc(unsigned nbytes)
{ Header *p, *newp;
  unsigned nunits;
  nunits = (nbytes+sizeof(Header)-1)/sizeof(Header) + 1;
  if (memptr == NULL)
  { memptr->s.next = memptr = mem;
    memptr->s.usedsize = 1;
    memptr->s.freesize = MEMSIZE-1;
  }
  for(p=memptr;
      (p->s.next!=memptr) && (p->s.freesize<nunits);
      p=p->s.next);
  if (p->s.freesize < nunits) return NULL;
  /* no block big enough */
  newp = p+p->s.usedsize;
  newp->s.usedsize = nunits;
  newp->s.freesize = p->s.freesize - nunits;
  newp->s.next = p->s.next;
  p->s.freesize = 0;
  p->s.next = newp;
  memptr = newp;
  return (void *) (newp+1);
}

void free(void *ap)
{ Header *bp, *p, *prev;
  bp = (Header *) ap - 1;
  for (prev=memptr,p=memptr->s.next;
       (p!=bp) && (p!=memptr); prev=p,p=p->s.next);
  if (p!=bp) return;
  /* corrupted list, do nothing */
  prev->s.freesize += p->s.usedsize + p->s.freesize;
  prev->s.next = p->s.next;
  memptr = prev;
}
```

这个代码使用容量MEMSIZE的静态分配数组作为堆，但也可使用操作系统调用分配堆。我们定义了一个数据类型Header保存每个存储器块的簿记信息，定义了具有Header类型元素的堆数组，这样就可很容易地将簿记信息保存在存储器块中。类型Header包含了3块信息：指向列表的下一个块的指针，当前分配空间的长度 (位于存储器之后)，以及任何后面的自由间的长度(若有的话)。因此，列表中的每个块都有格式

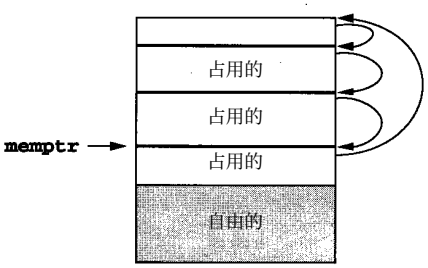


程序清单7-8中类型**Header**的定义还使用了一个**union**声明和**Align**数据类型(在代码中将其设为**double**)。这是将存储器元素排在合理的字节边界上，根据系统的不同，这有时是需要的，有时是不需要的。后面的描述中可安全地把这种复杂性忽略掉。

堆操作还需要的另一片数据是指向环形链接的列表中的一个块的指针。这个指针称作**memptr**，它总是指向具有一些自由空间的块(通常是被分配或释放的最后一个空间)。它被初始化为NULL，但是在**malloc**的第一次调用上，对初始化代码的执行是通过将**memptr**设置为堆数组的开头并初始化数组的头部，如下所示：



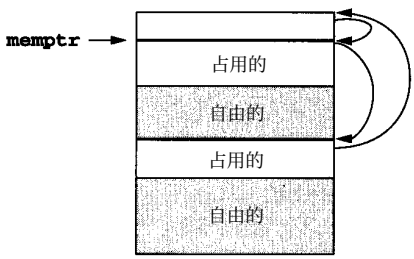
这个在第1次调用**malloc**时分配的初始化头部永远也不会被释放。这时在列表中有一个块，而其余的**malloc**代码搜索该列表并从具有足够自由空间的第1个块中返回一个新块(这是首次适用(**first fit**)算法)。因此在对**malloc**的3次调用之后，该列表看起来应是这样的：



注意，当连续分配块时，每次都会生成一个新块，并且还有前面块所剩下的自由空间(因此从块的自由空间生成的分配总是将**freesize**设置为0)。**memptr**跟随在新块的构造之后，所以它总是指向某个自由空间的块。大家还要注意，**malloc**总是增加指向新创建块的指针，所以也将头部保护起来而不会被客户程序覆盖(只要在返回存储器中使用正向偏移)。

现在来考虑**free**过程的代码。它首先把用户传递的指针减1，以找到块的头部。接着它再搜索列表以寻找与之相同的指针，以保护该列表防止坏掉，而且还能计算指向先前块的指针。一旦找到就将该块从列表中删除，且将其使用过的和自由空间都添加到先前块的自由空间中，所以也就自动地合并了自由空间。请读者注意，还将**memptr**设置为指向包含了刚才释放的存储器的块。

例如，假设将上图中3个使用过的块的中间一个释放了，则堆和与之相关的块列表应如下所示：



7.4.4 堆的自动管理

由于程序员必须编写出到分配和释放存储器的明确的调用，所以用 **malloc**和**free**完成指针的动态分配和重新分配是管理堆的手工 (manual)方法。相反地，运行时栈则是由调用序列自动地(automatically)管理。在一种需要完全动态的运行时环境的语言中，堆也必须类似地自动管理。然而尽管在每个过程调用中可以很方便地调度对 **malloc**的调用，但是由于活动记录必须要持续到其所有的引用都消失为止，所以退出时却很难调度对 **free**的调用。因此，自动存储器管理涉及到了前面分配的但不再使用的存储器的回收，可能是在它被分配的很久以后，而没有明确的对**free**的调用。这个过程称作垃圾回收(garbage collection)。

在存储块不再引用时，无论是直接还是通过指针间接的引用，识别是一项比维护堆存储块的列表复杂得多的任务。标准的技术是执行标识和打扫 (mark and sweep)垃圾回收⊖。在这种方法中，直到一个对 **malloc**的调用失败之前都不会释放存储器，在这时将垃圾回收程序激活，寻找可被引用的所有存储器并释放所有未引用的存储器。这是通过两遍来完成的。第 1遍递归地顺着所有的指针前进，从所有当前的可访问指针值开始，并标出到达的每个存储器块。这个过程要求额外的位存储标识。另一个遍则线性地打扫存储器，并将未标出的块返回到自由存储器中。虽然这个过程通常要寻找足够的相邻自由存储器以满足一系列的新要求，但存储器仍有可能是非常破碎，故尽管是在垃圾回收之后，大的存储请求仍旧会失败。因此，垃圾回收经常也会通过将所有的分配的空间移到堆的末尾，以及在另一端留下相邻的自由空间的唯一一个大型块而执行存储器压缩(memory compaction)。这个过程还必须在存储器中更新对那些在执行程序时被移掉的区域的所有引用。

标识和打扫垃圾回收有若干个缺点：它要求额外的存储 (用于标识)，在存储器中的两个遍导致了过程中很大的延迟，有时需要几秒钟，而每一次调用垃圾回收程序又都需要几分钟时间。这对于那些许多涉及到了交互和即时响应的应用程序显然是不合适的。

可以通过将可用的存储器分为两个部分并每次只从一个部分中分配存储来对这个过程进行改进。在标识遍时，将所有到达了的块都复制到未被使用的另一半存储器中。这就意味着在存储时不再要求额外的标识位而且一个遍就够了。它还自动地进行压缩。一但位于使用的区域中的所有可到达的块都复制好时，就将使用的和未使用的存储器部分相互交换，而过程依然继续进行。这种方法称作停机和复制 (stop-and-copy)或二部空间 (two space)垃圾回收。然而它对存储回收中的过程延迟改进不大。

最近又提出了一个大大减少延迟的方法, 称为生育的垃圾回收(generational garbage collection), 它将一个永久的存储区域添加到前一段描述的回收方案中。将存在时间足够长的被分配的对象只复制到永久空间中，并在随后的存储回收时不再重新分配。这就意味着垃圾回收程序在更新的存储分配时只需要搜索存储器中的很小的一个部分。当然永久存储器也有可能由于不可达到的存储而用尽，但这相对于前面的问题就不那么严重了，这是因为临时存储会很快消失，而可被分配的存储则总会有的。人们已证明了这个处理很好，在虚拟存储系统中尤为如此。

读者可在“注意与参考”一节中查阅此种方法的具体细节以及其他的垃圾回收方法。

7.5 参数传递机制

我们已经看到了在过程调用中，参数是如何通过调用程序在跳到被调用过程的代码之前与

⊖ 另一种称为引用计数(reference counting)的更为简单的方法也经常用到。参见“注意与参考”一节。

活动记录中的位置相对应的，该活动记录则由自变量或参数值组成。因此对于被调用的过程而言，参数代表了没有附加代码的完全正式的值，但该值仅在代码可以发现其最终值的活动记录中建立一个位置，这一最终值只在发生调用时才退出。建立这些值的过程有时是自变量的参数的绑定(binding)。自变量的值是如何由过程代码解释依赖于源语言的采用的特定参数传递机制(parameter passing mechanism(s))。正如早已提到过的，FORTRAN77采用的就是将参数传递到位置而不是值的机制，而C则将所有的自变量都当作是值。其他的语言，诸如C++、Pascal和Ada则是提供参数传递机制的选择。

在本节中，我们将讨论两个最常用的参数传递机制，值传递 (pass by value)和引用传递(pass by reference)(有时也称作由值调用和由引用调用)，此外还有两个重要方法，由值的结果传递(pass by value-result)和由名字传递(pass by name)(也称作延迟赋值(delayed evaluation))。它们的一些变形则放到了练习中。

未由参数传递机制本身说明的一个问题是自变量计算的顺序。在大多数情况下，这个顺序对于程序的执行并不重要，而且任何的计算顺序都是产生相同的结果。此时为了有较高的效率或其他原因，编译程序可能会选择改变自变量计算的顺序。但是许多语言却允许会导致副作用的自变量调用(改变存储器)。例如C的函数调用

```
f(++x,x);
```

会使**x**的值改变，所以不同的计算顺序会引起不同的结果。在这样的语言中，可能会要指定诸如从左到右的标准顺序，或者由编译程序的编写者来决定，而此时调用的结果在各种实现中都各不相同。特别地，C编译程序是从右到左计算它们的自变量。这就允许有不同数量的自变量(例如在**printf**函数中)，如在7.3.1节中的讨论。

7.5.1 值传递

在这个机制中，自变量是在调用时计算的表达式，而且在执行过程时，它们的值就成为了参数的值。这是在C中唯一可用的参数传递机制，且在Pascal和Ada中是缺省的(Ada还允许将这样的参数显式地指定为传入(**in**)参数)。

在最简单的格式中，这就意味着值参数在执行过程中是作为常量值，而且可将值传递解释为用自变量的值取代过程体中的所有参数。Ada使用这个值传递的格式，此时不能给这样的参数赋值或作为局部变量来使用。C和Pascal采用更宽松的观点，在其中的值参数在本质上被看作是被初始化了的局部变量，该变量可被用作常规变量，但是它们的改变不会引起任何非局部的改变。

在诸如C这样仅提供值传递的语言中，通过改变它的参数直接写一个过程来达到目的是不可能的。例如，以下用C中写出的**inc2**函数并没有达到其预想的效果：

```
void inc2( int x)
/* incorrect! */
{ ++x;++x; }
```

但在理论上，可能用函数恰当的一般性，通过返回相应的值而不是改变参数值来完成所有的计算，像C这样的语言通常提供使用值传递的方法，进行非局部改变。在C中，它使用了传递地址而不是值的格式(而且因此改变了参数的数据类型)：

```
void inc2( int* x)
/* now ok */
{ ++(*x);++(*x); }
```

当然由于要求**y**的地址而不是它的值，所以增加变量**y**，这个函数必须被称作**inc2(&y)**。

由于数组是隐含指针，这种方法在C中用于数组时特别好，而且值传递允许改变单个的数组元素：

```
void init(int x[],int size)
/* this works fine when called
   as init(a), where a is an array */
{ int i;
   for(i=0;i<size;++i) x[i]=0;
}
```

值传递在编译程序上没有特殊的要求。通过对自变量进行最直接的计算和活动记录的构造就可很便利地完成它了。

7.5.2 引用传递

在引用传递中，自变量必须与分配的地址一起变化（至少是原则上）。并非传递变量的值，引用传递的是变量的地址，因此参数就变成了自变量的别名（alias），而且在参数上发生的任何变化都会出现在自变量上。在FORTRAN77中，引用传递是唯一的参数传递机制的。在Pascal中，通过使用var关键字来得到引用传递，而在C++中，则是通过在参数说明中使用特殊字符&：

```
void inc2( int & x)
/* C++ reference parameter */
{ ++x;++x; }
```

现在就可调用这个函数而无需特别使用地址操作符：**inc2(y)**工作地很好。

引用传递要求编译程序计算自变量的地址（它也必须具有这样的地址），这个自变量将存放在局部活动记录中。由于局部“值”实际上是环境中的别处的地址，所以编译程序还要将对引用参数的局部访问转为间接访问。

在诸如FORTRAN77这样的只可用引用传递的语言中，通常要为是不带地址的值的自变量提供一个位置。像

```
p(2+3)
```

的调用在FORTRAN77中是非法的，编译程序必须为表达式**2+3**“创造”出一个地址，并将值计算到这个地址中，接着再将地址传递到调用中。一般地，这通过在调用程序的活动记录中创建一个临时地址进行（在FORTRAN77中它是静态的）。在练习7.1中有这方面的一个例子，通过为它在主过程的活动记录中创建一个存储器地址将值3作为自变量传递。

引用传递的一个方面是它不要求复制被传递的值，这与值传递不同。当要复制的值是一个较大的结构（或是在除了C或C++之外的语言中的一个数组）时，这有时会很重要。在这种情况下，能够由引用来传递自变量就是很重要的了，但是它禁止自变量的值有任何变化，因此就能够做到值传递而无需覆盖值的拷贝。这个选项是由C++所提供的，在其中可将调用写作：

```
void f( const MuchData & x )
```

其中的**MuchData**是带有大型结构的数据类型。这仍是引用传递，但是编译程序还必须执行一个静态检查：**x**从不出现在一个赋值的左边或是被改变⊖。

⊖ 在完全安全的方法中并不能总是这样做。

7.5.3 值结果传递

除了未建立真正的别名之外，这个机制得到的结果与引用传递类似：在过程中复制和使用自变量的值，然后当过程退出时，再将参数的最终值复制回自变量的地址。因此，这个方法有时也被称为复制进，复制出，或复制存储。这是 Ada的传入(in)传出(out)参数机制(Ada还有一个简单的传出(out)参数，该参数没有传递进的初始值：这可称作结果传递)。

值结果传递与引用传递的唯一区别在于别名的表现不同。例如，在以下的代码中 (C语法)：

```
void p(int x, int y)
{ ++x;
  ++y;
}

main()
{ int a=1;
  p (a,a);
  return 0;
}
```

在调用p之后，若使用了引用传递，则**a**的值为3；若使用了值结果传递，则**a**的值为2。

这个机制尚未指定的问题，可能随语言或实现的不同而不同，包括复制到自变量的结果的顺序以及自变量的地址是否仅在入口处计算和存储，或是否在退出时重新计算。

Ada还有一个问题：它的定义说明传入 (in)传出(out)参数会通过引用作为遍来真正地实现，而且任何在两个机制之下不同的计算(由此就涉及到了别名)都是错误的。

从编译程序的编写者的观点来看，值结果传递要求对运行时栈以及调用序列的基本结构的进行若干个修改。首先，被调用的程序不能释放活动记录，这是因为复制出的 (局部)值还必须对调用程序适用。其次，调用程序必须或是在建立新的活动记录开始之前就将自变量的地址压入到栈中，或是必须从被调用的程序中重新计算出这些返回的地址。

7.5.4 名字传递

这是传递机制中最复杂的参数了。由于名字传递的思想是直到在被调用的程序真正使用了自变量(作为一个参数)之后才对这个自变量赋值，所以它还称作延迟赋值 (delayed evaluation)。因此，自变量的名称或是它在调用点上的结构表示取代了它对应的参数的名字。例如在代码

```
void p(int x)
{ ++x; }
```

中，若做出了一个如**p(a[i])**的调用时，其结果是计算**++(a[i])**。因此，若在**p**中使用**x**之前改变**i**，那么它的结果就与在引用传递或在值结果传递中的不同。例如，在代码 (C语法)

```
int i;
int a[10];

void p(int x)
{ ++i;
  ++x;
}

main()
( i = 1;
  a[1]=1;
```

```
    a[2]=2;
    p(a[i]);
    return 0;
}
```

对p的调用的结果是将a[2]设置为3并保持a[1]不变。

名字传递的解释如下：在调用点上的自变量的文本被看作是它自己右边的函数，每当在被调用的过程的代码中到达相应的参数名时，就要计算它。我们总是在调用程序的环境中计算自变量，而总是在过程的定义环境中执行过程。

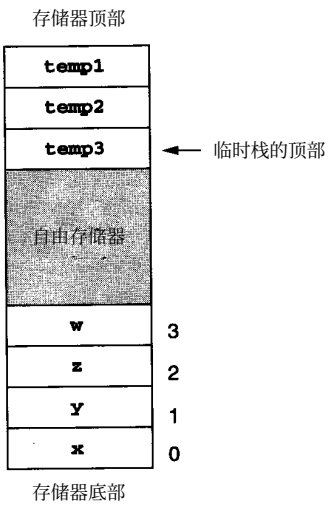
在语言Algo160中，名字传递是作为一种参数传递机制 (连同值传递)提供的，但是由于几个原因却变得并不流行。首先在副作用方面，它的结果会令人吃惊 (正如前一个例子所示)。其次，由于每个自变量本质上必须被变成过程（有时叫做挂起或形实转换程序），每当计算自变量时都须调用该过程，所以在实现上是有一定困难的。再者，由于它不仅将简单的自变量计算转变为过程调用而且还引起多个赋值，所以它的效率并不高。这种机制的一个变形称作懒惰赋值(lazy evaluation)，它最近在纯的函数语言中已较为流行了，在其中通过记忆第一次调用的计算值来 (memoizing)阻止重新赋值。因为从未用过的自变量也不会被赋值，所以懒惰赋值可以真正地使实现更为有效。提供懒惰赋值作为传递机制的参数的语言是 Miranda和Haskell。读者可查看“引用与参考”一节获得更多的信息。

7.6 TINY语言的运行时环境

在本章的最后一节，我们描述 TINY语言运行时环境的结构，所用的示例是一个用于编译的较小的简单语言。此时所用的方法与机器无关，读者还可查看下一章中特定机器上实现的示例。

TINY所需的环境比本章所提到过的任何环境都要简单得多。实际上， TINY没有过程，而且它的所有变量都是全局的，因此也就不需要一个活动记录的栈了，而唯一所需的动态存储是在表达式赋值期间临时所用的(在FORTRAN77中，甚至也可将其变成静态的，参见练习)。

TINY环境的一个简单示例是将变量放在程序存储器底部的绝对地址中，并将临时栈分配到顶部。这样，假设有4个变量x、y、z和w，这些变量在存储器底部得到绝对地址0到3，运行时环境在执行中的一个存放了3个临时变量的点上看起来如下所示：



根据体系结构，可能会需要设置一些簿记寄存器以指向存储器的底部和/或顶部，接着再使用“绝对的”变量地址作为底部指针的偏移，或是使用存储器的顶部指针作为“临时栈的顶部”指针，或是计算固定顶部指针的临时变量的偏移。当然如果处理器栈是可用的，还有可能就是将其作为临时栈。

为了实现这个运行时环境，TINY编译程序中的符号表必须如最后一章中所描述的那样在存储器中保留变量的地址。这是通过**st_insert**函数中提供地址参数以及包括重新得到变量地址的**st_lookup**函数完成的(附录B的第1166行到第1171行)：

```
void st_insert( char * name, int lineno, int loc ) ;
int st_lookup ( char * name ) ;
```

此时的语义分析程序必须在第一次遇到变量时就为其赋值，这是通过保留一个初始化到第1个地址中的静态存储器地址计数器来实现的(附录B第1413行)：

```
static int location = 0;
```

然后无论何时遇到变量(在读语句、赋值语句或标识符表达式中)，语义分析程序执行代码(附录B第1454行)：

```
if (st_lookup(t->attr.name) == -1)
    st_insert(t->attr.name,t->lineno,location++);
else
    st_insert(t->attr.name,t->lineno,0);
```

当**st_lookup**返回-1时，变量并不在表中。此时就记录下一个新的地址并添加了位置计数器。另一种情况是变量早已在表中，此时符号表忽略地址参数(并写下0作为一个虚构的地址)。

上面所述内容处理了在TINY程序中分配命名了的变量：位于存储器顶部的临时变量的分配以及保留这个分配所需的操作都由代码生成器负责，这将在下一章讨论到。

练习

7.1 为以下的FORTRAN77程序的运行时环境画出一个可能的组织结构，它应与图7-1相类似。还要保证包括了与对AVE的调用时存在的一样的存储器指针。

```
      REAL A(SIZE),AVE
      INTEGER N,I
   10 READ *, N
      IF (N.LE.0.OR.N.GT.SIZE) GOTO 99
      READ *,(A(I),I=1,N)
      PRINT *, 'AVE = ',AVE(A,N)
      GOTO 10
   99 CONTINUE
      END
      REAL FUNCTION AVE(B,N)
      INTEGER I,N
      REAL B(N),SUM
      SUM = 0.0
      DO 20 I=1,N
   20 SUM=SUM+B(I)
      AVE = SUM/N
      END
```

7.2 为以下的C程序的运行时环境画出一个可能的组织结构，它应与图7-2相类似。

a. 在进入函数**f**中的块**A**之后。
b. 在进入函数**g**中的块**B**之后。

```
int a[10];
char * s = "hello";

int f(int i, int b[])
{ int j=i;
  A:{ int i=j;
      char c = b[i];
      ...
    }
  return 0;
}

void g(char * s)
{ char c = s[0];
  B:{ int a[5];
    ...
  }
}

main()
{ int x=1;
  x = f(x,a);
  g(s);
  return 0;
}
```

7.3 在对**factor**的第2次调用之后为程序清单 4-1中的C程序的运行时环境画出一个可能的组织结构，假设输入串为**(2)**。

7.4 为以下的Pascal程序画出活动记录的栈，并在对过程 **c**的第2次调用之后表示出控制和访问链。描述如何在**c**中访问变量**x**。

```
program env;

procedure a;
var x:integer ;

  procedure b;
    procedure c;
    begin
      x := 2;
      b;
    end;
  begin (* b *)
      c;
    end;

begin (* a *)
  b;
end;

begin (* main *)
  a;
end.
```

7.5 为以下的Pascal程序画出活动记录的栈

a. 在对**p**的第1次调用中对**a**的调用之后。

b. 在对**p**的第2次调用中对**a**的调用之后。

c. 程序打印出什么？为什么？

```
program closureEx(output);
var x:integer;

procedure one;
begin
  writeln(x);
end;

procedure p(procedure a);
begin
  a;
end;

procedure q;
var x:integer;
      procedure two;
      begin
        writeln(x);
      end;
begin
  x := 2;
  p(one);
  p(two);
end; (* q *)

begin (*main *)
  x := 1;
  q;
end.
```

7.6 考虑以下的Pascal程序。假设一个用户输入包括了3个数1、2和0，则当第1次打印数1画出活动记录的栈。包括所有的控制和访问链以及所有的参数和全局变量，并假设将所有的过程都存储在作为闭包的环境中。

```
program procenv(input,output);

procedure dolist (procedure print);
var x:integer;
  procedure newprint;
  begin
    print;
    writeln(x);
  end;
begin (* dolist *)
  readln(x);
  if x = 0 then begin
    print;
    print;
```

```
  end
  else dolist(newprint);
end; (* dolist *)

procedure null;
begin
end;

begin (* main *)
  dolist ( null ) ;
end.
```

7.7 为了完成完整的静态分配，FORTRAN77编译程序需要构造对于程序中的任何表达式计算所需的临时变量的最大数的估计。设计一个方法来估计计算一个表达式所需临时变量的数目，计算通过表达式树的遍历进行。假设表达式是从左到右赋值且必须将每个左子表达式保存在临时变量中。

7.8 在允许过程调用中包含可变数量自变量的语言中，找到第 1个自变量的一种方法是根据7.3.1节所述按相反的顺序计算出自变量。

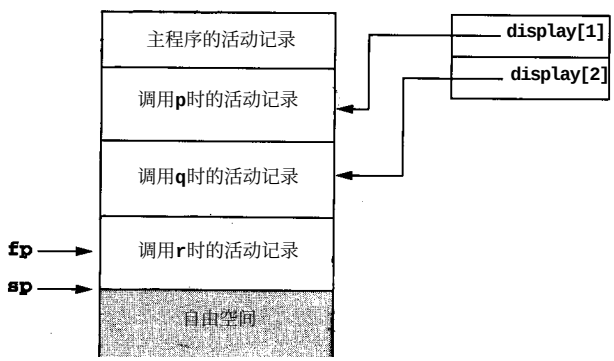
a. 按相反顺序计算自变量的另一种方法是识别活动记录，以使用第1个自变量即使是在可变数量自变量时也是适用的。描述这样的活动记录组织以及它所需的调用序列。

b. 另一个办法是使用除sp和fp之外的另一个称作ap(自变量指针)的指针。描述使用ap寻找第1个自变量和其所需的调用序列的活动记录结构。

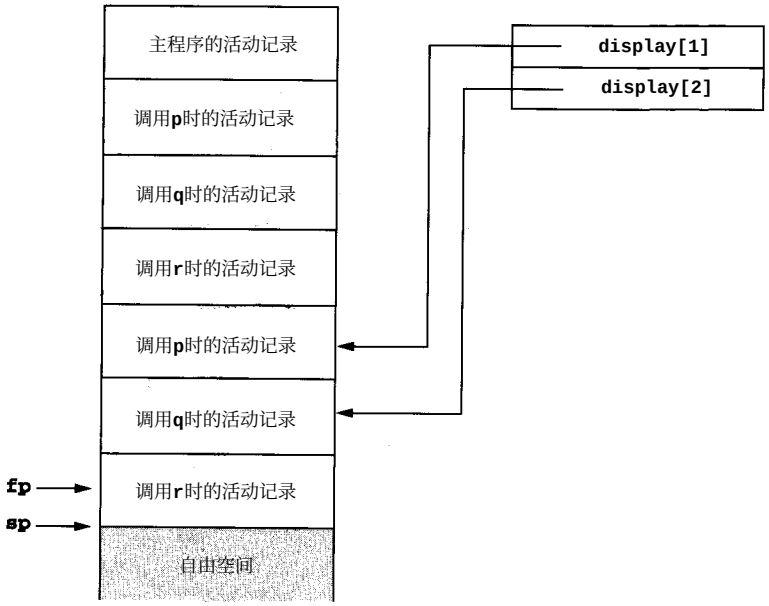
7.9 本书讲解了如何处理通过值传递可变长参数 (如开放式数组)(参见例7.6)，并提出了一个与变长局部变量作用相似的方法。但是当两个变长参数和局部参数都出现时就会有问题了。请利用以下的Ada过程作为示例来描述这个问题并提出解决办法：

```
type IntAr is Array(Integer range <>) of Integer;
...
procedure f(x:IntAr; n:Integer) is
y: Array(1..n) of Integer;
i: Integer;
begin
...
end f;
```

7.10 利用局部过程在语言中访问链的另一种方法是由嵌套层在栈外的一个数组中保存访问链。这个数组称作显示 (display)。例如当图7-7中的运行时栈带有显示时，应如下所示：



而图7-8中的运行时栈如下所示：



a. 描述显示如何可以从很深嵌套的过程中提高非局部引用的效率。

b. 使用显示重做练习**7.4**。

c. 描述实现显示必需的调用序列。

d. 利用过程参数在一个语言中使用显示出现了一个问题。利用练习 7.5描述问题。

7.11 考虑C语法中的以下过程：

```
void f( char c, char s[10], double r )
{ int * x;
  int y[5];
  ...
}
```

a. 使用标准C参数传递约定，并假设数据大小为：整型 = 2个字节，字符 = 1个字节，双精度 = 8个字节，地址 = 4个字节，利用本章所描述的活动记录结构判断以下的fp的偏移：(1) **c**. (2) **s[7]**. (3) **y[2]**.

b. 假设所有的参数都由值(包括数组)传递，重复**a**。

c. 假设所有的参数都引用传递，重复**a**。

7.12 执行以下的C程序并用运行时环境解释其输出：

```
#include <stdio.h>

void g(void)
{ {int x;
   printf("%d\n",x);
   x = 3;}
   {int y;
   printf("%d\n",y);}
}
```

```
int* f(void )
{ int x;
  printf("%d\n",x);
  return &x;
}

void main()
{ int *p;
  p = f();
  *p =1;
  f(;)
  g();
}
```

7.13 为以下的C++类画出对象的存储器框架以及如7.4.2节所述的虚拟函数表：

```
class A
{ public:
  int a;
  virtual void f();
  virtual void g();
};
class B : public A
{ public :
  int b;
  virtual void f();
  void h();
};
class C : public B
{ public:
  int c;
  virtual void g();
}
```

7.14 在面向对象的语言中的虚拟函数表为一个方法而保存层次图搜索遍历，但这是有代价的。请解释这个代价是什么。

7.15 利用7.5节中所谈到的4个参数传递办法给出以下程序的输出(用C语法编写)：

```
#include <stdio.h>
int i=0;

void p(int x, int y)
{ x += 1;
  i += 1;
  y += 1;
}

main()
{ int a[2]={1,1};
  p(a[i],a[i]);
  printf("%d %d\n",a[0],a[1]);
  return 0;
}
```

7.16 利用7.5节中所谈到的参数传递办法给出以下程序的输出(在C语法中)：

```
#include <stdio.h>
int i=0;

void swap(int x, int y)
{ x = x + y;
  y = x - y;
  x = x - y;
}

main()
{ int a[3] = {1,2,0};
  swap(i,a[i]);
  printf("%d %d %d %d\n",i,a[0],a[1],a[2]);
  return 0;
}
```

7.17 假设将FORTRAN77的子例程**p**说明如下：

```
SUBROUTINE P(A)
INTEGER A
PRINT *, A
A = A + 1
RETURN
END
```

且从主程序中调用如下：

```
CALL P(1)
```

在某些FORTRAN77系统中，这将导致一个运行时的错误。而在其他系统中却不会发生运行时错误，但若用1作为其自变量而再调用一次子例程，它将会打印出值 2。请根据运行时环境解释这两个行为是如何发生的。

7.18 名字传递的一个变形称作由文本传递 (pass by text) ，其中的自变量的赋值是用的延迟风格，这与名字传递相同，但每个自变量都是在被调用的过程的环境中而不是在调用的环境中赋值。

a. 说明由文本传递的结果可与名字传递不同。

b. 描述一个运行时环境组织以及可被用作实现由文本传递的调用序列。

编程练习

7.19 正如在7.5节中所描述的一样，名字传递或延迟赋值都可被看作是将自变量包在一个函数体中(或中止)，它在参数每次出现在代码中时被调用。重写练习 **7.16**的C代码以在这个风格中实现`swap`函数的参数，并证实该结果确实是与由名字传递相等。

7.20 **a.** 正如在7.5.4节中所述，名字传递中的一个有效的实现可以通过记忆它第 1次赋值的自变量的值。重写前一个练习中的代码以实现这样的记忆，并比较两个练习的结果。

b. 记忆会导致与名字传递所不同的结果。请解释它是如何发生的。

7.21 可将压缩 (7.4.4节) 垃圾回收分成两个不同的步骤，并当存储器要求由于缺少足够的大型块而失败时也可由`malloc`完成。

a. 重写7.4.3节中的`malloc`过程以包括压缩步骤。

b. 压缩需要先前分配空间改变的位置信息，这意味着程序必须发现这些改变。描述如何使用指向内存块的指针表来解决这个问题，并重写 **a**部分的代码以包括这个功能。

注意与参考

FORTRAN77(以及更早的FORTRAN版本)的完全静态环境给出了一个类似汇编环境的原始而直接的环境设计方法。基于栈的环境随着诸如 Algol60的包含递归的语言的出现而流行起来(Naur[1963])。Randell和Russell [1964]详细描述了早期的Alogol60基于栈的环境。用于一些C编译器的活动记录组织和调用序列的描述在 Johnson和Ritchie [1981]中。用显示代替访问链(练习**7.10**)在Fisher和LeBlanc[1991]中有详细描述，其中还包括在有过程形式参数的语言中使用时将出现的问题。

动态内存管理在许多数据结构的书中有所讨论，比如 Aho,Hopcrott和Ullman [1983]。一个实用的近期浏览在Drozdek和Simon[1995]中给出。`malloc`和`free`的代码实现也是类似的，但Kernighan 和Ritchie[1988]中比7.4.3节的代码稍微简单一些。编译中使用的堆结构设计在 Fraser和Hanson [1995]中讨论。

一个垃圾回收的浏览可以在Wilson[1992]或Cohen[1981]中找到。生成的垃圾回收和用于函数语言ML运行时环境在Appel [1992]中描述。Gofer函数语言编译器(Jones[1984])包括标记和清除以及一个两阶段垃圾回收。

Budd[1987]描述一个用于小型Smalltalk系统的完全动态环境，包括继承图的应用和带有引用计数的垃圾回收器。C++中使用的虚拟函数表在Ellis和Stroustrup[1990]中描述，还包括处理多继承的扩展。

更多的参数传递技术可以在Louden[1993]中找到，其中还有懒惰赋值的描述。懒惰赋值的实现技术可以在Peyton Jones[1987]中找到。