

C 程序设计—函数的次序与变量的初值

侯济恭 王博文

[计算机科学(电脑)系]

摘要 本文从编译程序的角度阐述C语言程序中函数的次序、变量初值的设定及其作用域等问题。

关键词 编译程序, 程序系统, 程序设计

0 引言

C语言有许多优点,但也有些不尽人意之处:如函数的排列次序(即在源程序中出现 的次序),变量的初值设定及其作用域等。初学者稍不小心,往往因此导致错误。本文从编译程序角度阐述其中原因,以期帮助C爱好者了解C的一些内幕,更有效地利用C。

1 函数的次序

设有两个简单的C源程序

```
double example(x)
double x;
{ double y;
  y=x*x;
  return(y*3.14159);
}
```

```
main( )
{ double a ;
  scanf("%f", &a);
  a=example(a);
  printf("%f", a);
}
```

现将这两个函数组成一个源程序,可有两种方式,即

方式A:

```
#include <stdio.h>
example 函数
main 函数
```

方式B:

```
#include <stdio.h>
main 函数
example 函数
```

本文1989-09-07收到。

若据大多数C的教科书：*main*函数所在位置与程序运行无关。则A、B两种方式均可正确运行。但事实上方式B连编译都无法通过，可见前提非真。正确的断言是：当任何函数的返回值为整型或无返回值时，函数所在位置与运行结果无关。因此，要使方式B正确运行，则须将其变换，加入对函数*example*的说明，即变为

方式B'：

```
# include <stdio.h>

double example();

main 函数

example 函数
```

产生这种现象的原因是C编译器对函数调用处理。C编译器（下称CC，本CC是VAX-11机CC，所用汇编系VAX宏汇编）对源程序的翻译过程大体上是：读入并识别源程序的每一语句行*S*：

1) 若*S*是说明或定义语句，则将其符号名（变量名、函数名等）及其数据类型、存储类型、所属层次等有关信息填入符号表中，分配予一形式地址并填入符号表；若符号名说明在先，定义在后，则后者仅修改其存储属性。对每一个外部定义符号名，均产生一符号地址代码。

2) 若*S*是可执行语句则产生语法树，树结点的信息如存储类型，数据类型等则抄自符号表，若符号表中查无该符号名，则认定其为整型量，并据此填写符号表。

3) 生成语法树目标代码并写入目标文件（若为二遍扫描则先生成语法树，然后将语法树写入中间文件）。

以方式B编译过程为例。先编译*main*函数，将函数名*main*及其有关属性填入符号表中。当编译到 $\alpha = \text{example}(\alpha)$ 时，查符号表，发现尚未对*example*说明或定义。于是CC将其数据类型假定为整型量并填入符号表中。接着产生函数调用语法树如图1，树结点信息来自符号表。图中的“CALL”为函数调用内部算符，其左子结为函数名，右子结为函数实参。最后生成该树目标代码并写入目标文件中。去。*main*函数编译毕，编译*example*函数。处理函数名*example*时，先检查符号表，发现符号表信息有误，*example*的数据类型应是*double*，但函数调用目标码已写入目标文中，CC只好给出错误信息“redeclaration of example”。（如果*main*函数和*example*函数分开编译，则无法发现此类错误，这将导致不可预测之运行错误。）

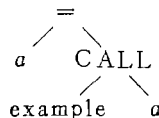


图1 函数调用语法树

方式A可正确运行的原因是：*example*函数定义的翻译先于*main*函数的翻译，当翻译 $\alpha = \text{example}(\alpha)$ 时，已可从符号表中查到*example*的正确信息了。方式B'可正确运行是因为：CC首先处理函数说明语句 `double example()`，将*example*的属性说明填入符号表，当编译至*main*函数 $\alpha = \text{example}(\alpha)$ 时，已可从符号表中获取与*example*有关的正确信息了。在翻译*example*函数定义时，仅修改符号表中*example*的存储类型，将外部说明（EXTRN）改为外部定义（EXTDEF）。

对上述结构模式，C的教科书[2]一般总告诫读者要在函数体内对所调用的非整型函数预先说明。这种方法不但累赘，事实上初学者经常忘记。一个值得推荐的方法是：把除主函数名以外的所有非整型函数作为一个函数名说明文件，再将其include到所需之源程序模块中去。对小型程序，最好将所有函数名在源程序开头予以说明。采用此法，一是可以不必关心函数的次序问题，二是可以查阅各函数，有利无弊。

2 变量的初值

变量的初值涉及CC对变量的存贮分配问题。以下分别讨论。

1) 自动变量：C的存贮空间分配采用静态和栈式动态分配。栈式动态分配的原则是：每一个函数分配一个栈帧（图 2），栈帧大小根据该函数的所有自动变量（非静态的）、形式参数以及有关链接数据（如程序状态字，形参地址指针，返回地址等）的多寡而定。每一自动变量的地址是一个相对地址（栈帧内位移量）。当 α 所属的函数 f 活跃时， f 的栈帧存在，故 α 的地址存在，当 f 执行完毕， f 的栈帧撤消， α 的地址亦随之消亡。

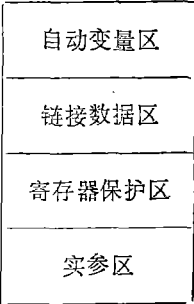


图 2 栈帧结构

例如example的自动变量 y 。当运行main函数时， y 的地址不存在（图 3 a），当main调用example时，example活跃，其栈帧出现（图 3 b）。example运行完毕，其栈帧撤消， y 的地址亦消亡（图 3 a）。

若有多层递归调用，例如example又调用自身，则栈又增长一帧（图 3 c）， y 的地址又出现在栈的另一处。可见，自动变量 y 可能出现在栈中任

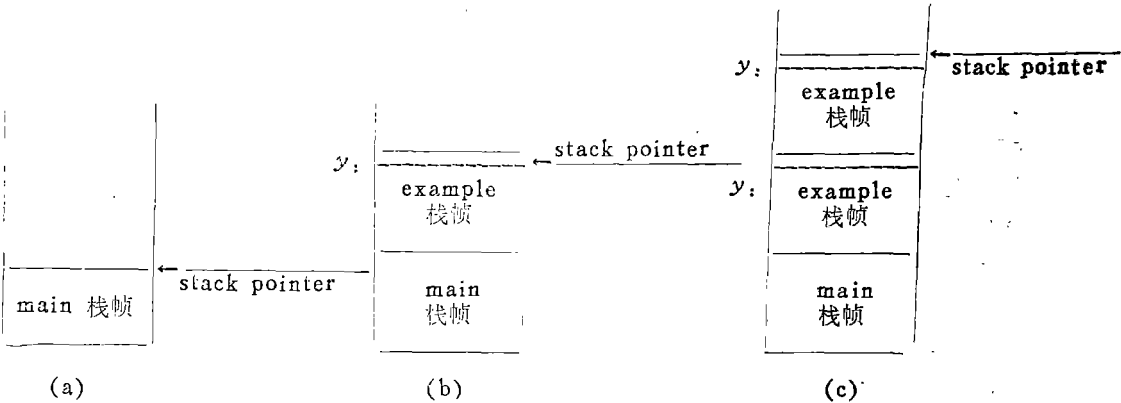


图 3 栈帧活动图

何位置。因此，当 y 的初值没有设定时，其初值未可预测，取决于所处栈单元之值。当自动变量显式置初值之时，编译程序将其翻译成将 i 送入所分配之栈单元指令。例如

```
try ( ) { int a=6; ...; }
```

的对应目标码是

函数头码	;	函数入口预处理代码
movl #6, -4(fp)	;	将6送入自动变量区相
		对地址-4, 即a的地址
:		

请注意置初值是函数体的第一条可执行指令。因此, 每当try函数运行时, a 所对应的栈单元总预置为6。故: 自动变量 a 设立初值后, 每当其所属的函数运行时, a 恒保持同一初值。

静态分配方法就不同了。所谓静态分配, 就是为静态变量分配一个符号地址, 这个地址在该目标程序调入主存时就一直存在, 直至该程序撤消为止。当静态自动变量 s 被显式置初值 i 时, CC 将 i 写入 s 所对应的符号地址; 当 s 无初值时, CC 将 s 所对应的符号地址清零。例如源程序 try_1 为

```
try_1() { static int a=14, b; ...; }
```

所对应的目标码是

```

      函数头码      ;
• align 4      ; 边界调整
• L15;          ; a的内部符号地址
• Long 14      ; 长字单元, 值14
• align 4      ;
• L16;          ; b的内部符号地址
• space 4      ; 保留4字节空间并清零
      :

```

可见静态自动变量的地址是固定的, 它的初值由编译确认和执行。静态变量仅置初值一次, 它能保持运算结果, 与其所属函数运行与否无关。

2) 外部变量: CC 对外部定义的变量采用静态分配法, 即分配一个同名符号地址。

(1) 外部变量定义: CC 对此类变量除分配一个同名符号地址外, 还产生一伪指令, 说明其为全局的, 可为外部模块所引用的变量名。例如外部变量 float d ; 所对应的目标码是

```

• align 4      ;
• globl -d      ; 全局量d, 外部量说明
• comm -d, 4    ; 符号名-d, 保留4字节空间

```

无显式初值的外部变量, 其初值随机器类型而异, 有的确定为0, 有的不予确定, 如本CC就是如此。对有显式设立初值的外部变量, CC 将初值写入对应的地址中去。例如

```
int b=6; ...
```

的对应目标码是

```

    • globl  __y      ; 全局量 y
__y:                      ; 符号地址 __y
    • long  6        ; 初值为长字 6
    :

```

(2) 静态外部定义: 此静态的含义是局部量, 与静态分配的含义不同。对静态外部变量, CC 除分配予一个符号名地址外, 若该变量有显式置初值, 则将初值写入该地址, 否则将该地址清零。例如

```
static int x, y=7; ...
```

对应的目标码是

```

    • align  4
__x:                      ; x 的符号地址
    • space  4
    • align  4
__y:                      ; y 的符号地址
    • long   7
    :

```

(3) 外部说明: 关键字 `extern` 说明其后的变量名为外部的, 对此 CC 仅将符号名有关属性填入符号表中, 以备以后的变量正确引用。有些 CC 则产生外部伪指令 `• EXTRN<变量名>` 以告知汇编和链接程序: 此变量名在外部模块中。

(4) 引用规则: 外部变量的引用规则与函数的引用规则相同, 引起出错的原因也相同。

3) 寄存器变量: CC 根据用户的要求分配寄存器给指定的变量。当寄存器不够分配时 (VAX 中可供分配的寄存器是 5 个), 则相应的变量被转变成自动变量, 再按自动变量存贮分配原则分配地址。当寄存器变量 t 无显式置初值时, CC 认定所分配的寄存器 r_i 即为 t (登记在符号表中), 故其初值未可知。当 t 有显式置初值时, CC 将其翻译成将初值送入寄存器 r_i 指令。对函数参数是寄存器的情况, 则翻译成将对应参数值送入寄存器指令。例如

```

try3(x)
register x ;
{
    register t1, t2=6 ;
    x=t1+t2 ; ... ;
}

```

对应的目标码是

```

    函数头码 ;
movl  4(ap), r11      ; r11 对应参数 x, 取实参 x 值送 r11
movl  # 6, r9         ; r9 对应变量 t2, t2=6
addl3 r9, r10, r0     ; r0=r9+r10, 即 r0=t1+t2
movl  r0, r11         ; x=r0, 即 x=t1+t2
    :

```

其中 ap 为实参栈指针, 4为形参 x 的栈偏移量。

可见寄存器变量的初值设定与自动变量的初值设定是一致的: 有显式初值, 则当其所属函数运行时, 其初值恒定, 否则未可预测。

CC对数组, 结构的初值处理方式与上述讨论相同, 所不同是代码长度较长, 对指针型变量的初值, 如 $char *p = "abc"$, 则将 abc 存入一数据区, 不再另述。

3 变量的作用域

由CC的编译过程可知, 外部变量的存在域从其定义/说明点始, 直至程序的物理终点, 而其作用域等于存在域扣去内层同名变量存在域。以下主要讨论自动变量的作用域。

CC为每一个函数 f 建立一个符号子表, 更精确地说, 为每一个分程序建立一个子符号表, 其处理过程如下: (1) 遇左大括号时, 层次计数器 $blevel$ 增1, 此后所定义的符号名, 其层次属性均为 $blevel$; (2) 当处理变量名 x 的定义时, 若外层有同名变量, 则将其掩盖, 即在符号表 x 的属性中加注“hidden”, 使其不起作用但存在, 同时自身加注掩盖“hide”标志; 此后任何对同名变量的引用, 均使用无“hidden”标志的变量; (3) 遇右大括号, 即本分程序结束, $blevel$ 减1; (4) 从符号表中注销本层所属符号名。若符号名中有“hide”标志, 则解除被其掩盖的同名变量的“hidden”标志。

由此程序中外层任何点均无法引用内层变量, 内层变量一出所属层次便消亡, 故无法作用到随后的外层程序任何点。当一个函数被翻译完毕, 其符号表亦随之消亡(函数体实际上也是一层分程序)。因此自动变量的作用域仅限于所属的分程序。

```

float e      ;           /* 0层    */
try _4(x, y)
int x, y     ;           /* 1层    */
{
    int a     ;           /* 2层    */
    {
        float a;         x=a++;           /* 3层    */
        {
            int a=6 ;    x=x+a;           /* 4层    */
        }
    }
}

```

为方便讨论, 特作约定: a_k , k 代表同名变量 a 所属层次。对同名变量 a 掩盖过程如下: (1) 处理 a_2 , 填写符号表, 掩盖标志清空(图4a); (2) 处理 a_3 , 发现同名变量 a_2 , 将 a_2 掩盖标志域置成hidden。从 a_2 的表入口 p_0 始往下扫描, 找到一空表项 p , 填入 a_3 , 其掩盖标志域置成hide。(图4b); (3) 处理 a_4 , 发现 a_3 未被掩盖, 掩盖之。从 a_3 之表入口 p 始往下扫描, 找至一空项 q , 填入 a_4 及其有关属性(图4c)。

查表方向如图 4 虚线所示：从 p 至 p 环形查找， q 为动态指针。表溢出条件是 $p = q$ 。

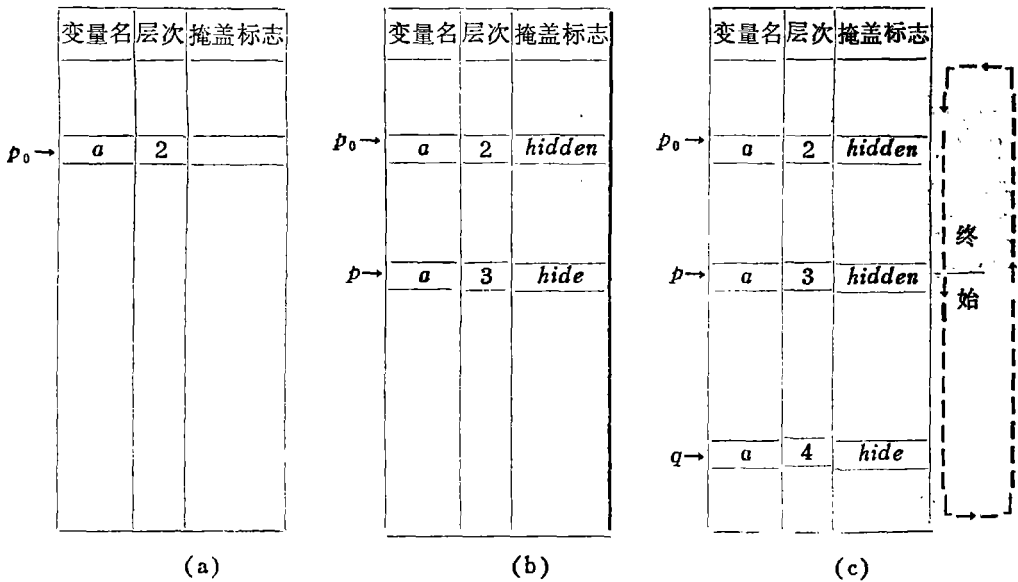


图 4 同名变量掩盖操作示意

撤消掩盖过程是以上过程之逆：（1）退出第 4 层，从符号表中消除 α_4 （图 4 c），由于 α_4 有 *hide* 标志，故从 q 始向上扫描，遇到第一个同名有被掩盖标志的变量 α_3 ，再继续向上扫描，知表中尚有 α_2 存在，故修改 α_3 的掩盖标志减为 *hide*（图 4 b）；（2）退出第 3 层，消除 α_3 。扫描方式同上。由于再没有被掩盖的同名变量，故修改 α_2 的掩盖标志域为空（图 4 a）；（3）退出第 2 层，消除 α_2 。本函数所有自动变量和形式参数全部从符号表中注销。

4 结论

本文所有结论仍分析 VAX CC 源程序的结果，所有汇编码均由该 CC 产生。作为本文的结束，兹将以上讨论之结论列于表 1，以供参考。

表 1 变量属性表

	外 部 变 量			自 动 变 量		
	说明	定义	静态定义	静态自动量	自动	寄存器
地址分配	无	静态	静态	静态	动态	
系统建立初值	无	不可测	0	0	不可测	不可测
初始化次数	无	1	1	1	随函数执行次数	
初始化执行者	无	编译器	编译器	编译器	程序指令	
生存期	无	与所属程序同寿		所属程序	与所属层次同寿	
共享性		多个模块	所属模块	所 属	层 次	

参 考 文 献

- [1] Johnson, S. C. , *A Tour Through The Portable CCompiler*, Bell laboratories, (1978).
- [2] 李德华, C语言BNF解译及其程序设计, 陕西科学技术出版社, (1986).
- [3] Aho, A.V. and Ullman, J.C., *Principles of Compiler Design*, Addison-Wesley, (1977).
- [4] Kernighan, B. W. , Ritchie, D. M. , *The C Programming Language*, Englewood Cliffs, NJ, Prentice-Hall, (1978).

Programming in C Language: Order of Functions and Initialization of Variables

Hou Jigong Wang Bowen

Abstract In C language programming, the order of functions, the initialization of variables, and the scope of variables are being set forth from the angle of compiler.

Keywords Compiler, Programming systems, programming