Assembly do IA-32 em ambiente Linux

Exercícios de Programação 4

(Adaptados do livro de Randal E. Bryant and David R. O'Hallaron)
Alberto J. Proença

Prazos

Entrega **impreterível**: segunda 10-Dez-01, no Lab. da disciplina (não serão aceites trabalhos entregues depois deste prazo).

Para mais informações, ler metodologia definida no enunciado dos Exercícios de Programação 1.

Introdução

A lista de exercícios que se apresenta estão directamente relacionados com a codificação de dados estruturados em *assembly* do IA-32, nomeadamente *arrays* e *structures*. O material complementar necessário a uma melhor compreensão dos exercícios é retirado do Cap. 3 do livro acima referido, e reproduzido nestas folhas (não traduzido; apenas os enunciados dos exercícios estão em Português).

Array Allocation and Access

Arrays in C are one means of aggregating scalar data into larger data types. C uses a particularly simple implementation of arrays, and hence the translation into machine code is fairly straightforward. One unusual feature of C is that one can generate pointers to elements within arrays and perform arithmetic with these pointers. These are translated into address computations in assembly code.

Optimizing compilers are particularly good at simplifying the address computations used by array indexing. This can make the correspondence between the C code and its translation into machine code somewhat difficult to decipher.

For data type ***T*** and integer constant ***N***, the declaration

```
T A[N];
```

has two effects. First, it allocates a contiguous region of ***L*N*** bytes in memory, where ***L*** is the size (in bytes) of data type ***T***. Let us denote the starting location as x_A. Second, it introduces an identifier `A` that can be used as a pointer to the beginning of the array. The value of this pointer will be x_A. The array elements can be accessed using an integer index ranging between ***0*** and ***N-1***. Array element ***i*** will be stored at address $x_A + L*i$.

As examples, consider the following declarations:

```
char        A[12];
char       *B[8];
double      C[6];
double     *D[5];
```

These declarations will generate arrays with the following parameters:

Array	Element Size	Total Size	Start Address	Element ***i***
A	1	12	x_A	$x_A + i$
B	4	32	x_B	$x_B + 4i$
C	8	48	x_C	$x_C + 8i$
D	4	20	x_D	$x_D + 4i$

Array A consists of 12 single-byte (char) elements. Array C consists of 6 double-precision floating-point values, each requiring 8 bytes. B and D are both arrays of pointers, and hence the array elements are 4 bytes each.

The memory referencing instructions of IA32 are designed to simplify array access. For example, suppose E is an array of int's, and we wish to compute E[i] where the address of E is stored in register %edx and i is stored in register %ecx. Then the instruction:

```
movl (%edx,%ecx,4),%eax
```

will perform the address computation $x_E + 4i$, read that memory location, and store the result in register %eax. The allowed scaling factors of 1, 2, 4, and 8 cover the sizes of the primitive data types.

Exercício 4.1 (*Localização de elementos de um array*):

Considere as seguintes declarações:

```
short          S[7];
short         *T[3];
short        **U[6];
long double    V[8];
long double   *W[4];
```

Preencha a tabela seguinte, com os valores do tamanho de cada elemento, o tamanho total, o endereço do elemento ***i***, para cada um destes *arrays*.

Array	Element Size	Total Size	Start Address	Element ***i***
S			x_S	
T			x_T	
U			x_U	
V			x_V	
W			x_W	

Pointer Arithmetic

C allows arithmetic on pointers, where the computed value is scaled according to the size of the data type referenced by the pointer. That is, if p is a pointer to data of type ***T*** and the value of p is x_p then the expression p+i has value x_p + ***L****i , where ***L*** is the size of data type ***T***.

The unary operators & and * allow the generation and dereferencing of pointers. That is, for an expression *Expr* denoting some object, *&Expr* is a pointer giving the address of the object. For an expression *Addr-Expr* denoting an address, **Addr-Expr* gives the value at that address. The expressions *Expr* and *&*Expr* are therefore equivalent. The array subscripting operation can be applied to both arrays and pointers. The array reference A[i] is identical to the expression *(A+i). It computes the address of the ith array element and then accesses this memory location.

Expanding on our earlier example, suppose the starting address of integer array `E` and integer index `i` are stored in registers `%edx` and `%ecx`, respectively. The following are some expressions involving E. We also show an assembly code implementation of each expression, with the result being stored in register `%eax`.

Expression	Type	Value	Assembly Code
`E`	`Int *`	x_E	`movl %edx,%eax`
`E[0]`	`int`	$Mem[x_E]$	`movl (%edx),%eax`
`E[i]`	`int`	$Mem[x_E + 4i]$	`movl (%edx,%ecx,4),%eax`
`&E[2]`	`Int *`	$x_E + 8$	`leal 8(%edx),%eax`
`E+i-1`	`int *`	$x_E + 4i - 4$	`leal -4(%edx,%ecx,4),%eax`
`*(&E[i]+i)`	`int`	$Mem[x_E + 4i + 4i]$	`movl (%edx,%ecx,8),%eax`
`&E[i]-E`	`int`	`i`	`movl %ecx,%eax`

In these examples, the `leal` instruction is used to generate an address, while `movl` is used to reference memory (except in the first case, where it copies an address). The final example shows that one can compute the difference of two pointers within the same data structure, with the result divided by the size of the data type.

Exercício 4.2 (*Apontadores*):

Considere que o apontador para o início do *array* `S` (de inteiros `short`)e que o índice inteiro `i` estão armazenados nos registos `%edx` e `%ecx`, respectivamente. Para cada uma das expressões seguintes, indique o seu tipo, a fórmula de cálculo do seu valor, e uma implementação em código *assembly*. O resultado deverá ser colocado no registo `%eax` se for um apontador, e em `%ax` se for um inteiro `short.`

Expression	Type	Value	Assembly Code
`S+1`			
`S[3]`			
`&S[i]`			
`S[4*i+1]`			
`S+i-5`			

Nested Arrays

The general principles of array allocation and referencing hold even when we create arrays of arrays. For example, the declaration:

```
int A[4][3];
```

is equivalent to the declaration:

```
typedef int row3_t[3];
row3_t A[4];
```

Data type `row3_t` is defined to be an array of three integers. Array `A` contains four such elements, each requiring 12 bytes to store the three integers. The total array size is then `4*4*3 =48` bytes.

Array `A` can also be viewed as a two-dimensional array with four rows and three columns, referenced as `A[0][0]` through `A[3][2]`. The array elements are ordered in memory in “row major” order, meaning all elements of row 0, followed by all elements of row 1, and so on.

Element	Address
`A[0][0]`	x_A
`A[0][1]`	$x_A + 4$
`A[0][2]`	$x_A + 8$
`A[1][0]`	$x_A + 12$
`A[1][1]`	$x_A + 16$
`A[1][2]`	$x_A + 20$
`A[2][0]`	$x_A + 24$
`A[2][1]`	$x_A + 28$
`A[2][2]`	$x_A + 32$
`A[3][0]`	$x_A + 36$
`A[3][1]`	$x_A + 40$
`A[3][2]`	$x_A + 44$

This ordering is a consequence of our nested declaration. Viewing `A` as an array of four elements, each of which is an array of three `int`'s, we first have `A[0]` (i.e., row 0), followed by `A[1]`, and so on.

To access elements of multidimensional arrays, the compiler generates code to compute the offset of the desired element and then uses a `movl` instruction using the start of the array as the base address and the (possibly scaled) offset as an index. In general, for an array declared as:

T `D[`***R***`][`***C***`];`

array element `D[i][j]` is at memory address $x_D + L(C$*`i + j`) where ***L*** `is` the size of data type ***T*** in bytes.

As an example, consider the `4*3` integer array `A` defined earlier. Suppose register `%eax` contains x_A, that `%edx` holds `i`, and `%ecx` holds `j`. Then array element `A[i][j]` can be copied to register `%eax` by the following code:

```
  A in %eax, i in %edx, j in %ecx
1   sall   $2,%ecx                  j * 4
2   leal   (%edx,%edx,2),%edx       i * 3
3   leal   (%ecx,%edx,4),%edx       j * 4 + i * 12
4   movl   (%eax,%edx),%eax         Read Mem[xA + 4(3 * i + j)]
```

Exercício 4.3 (*Matriz de 2 dimensões*):

Considere o seguinte código fonte, onde `M` e `N` são constantes declarados com `#define`.

```
int mat1[M][N];
int mat2[N][M];

int sum_element(int i, int j)
{
   return mat1[i][j] + mat2[j][i];
}
```

Na compilação deste programa, GCC gera o seguinte código *assembly*:

```
movl   8(%ebp),%ecx
movl   12(%ebp),%eax
leal   0(,%eax,4),%ebx
leal   0(,%ecx,8),%edx
subl   %ecx,%edx
addl   %ebx,%eax
sall   $2,%eax
movl   mat2(%eax,%ecx,4),%eax
addl   mat1(%ebx,%edx,4),%eax
```

Use técnicas de *reverse engineering* para determinar os valores de `M` e `N` baseados neste código *assembly*.

Structures

The C `struct` declaration creates a data type that groups objects of possibly different types into a single object. The different components of a structure are referenced by names. The implementation of structures is similar to that of arrays in that all of the components of a structure are stored in a contiguous region of memory, and a pointer to a structure is the address of its first byte. The compiler maintains information about each structure type indicating the byte offset of each field. It generates references to structure elements using these offsets as displacements in memory referencing instructions.

As an example, consider the following structure declaration:

```
struct rec {
    int i;
    int j;
    int a[3];
    int *p;
};
```

This structure contains four fields: two 4-byte `int`'s, an array consisting of three 4-byte `int`'s, and a 4-byte integer pointer, giving a total of 24 bytes:

Offset	0	4	8			20
Contents	i	j	a[0]	a[1]	a[2]	p

Observe that array `a` is embedded within the structure. The numbers along the top of the diagram give the byte offsets of the fields from the beginning of the structure.

To access the fields of a structure, the compiler generates code that adds the appropriate offset to the address of the structure. For example, suppose variable `r` of type `struct rec *` is in register `%edx`. Then the following code copies element `r->i` to element `r->`j:

```
movl    (%edx),%eax             Get r->i
movl    %eax,4(%edx)            Store in r->j
```

Since the offset of field `i` is `0`, the address of this field is simply the value of `r`. To store into field `j`, the code adds offset 4 to the address of `r`.

To generate a pointer to an object within a structure, we can simply add the field's offset to the structure address. For example, we can generate the pointer `&(r->a[1])` by adding offset `8 + 4*1 =12.` For pointer `r` in register `%edx` and integer variable `i` in register `%eax`, we can generate the pointer value `&(r->a[i])` with the single instruction:

```
 r in %eax, i in %edx
leal    8(%eax,%edx,4),%ecx     %ecx = &r->a[i]
```

As a final example, the following code implements the statement:

```
r->p = &r->a[r->i + r->j];
```

starting with `r` in register `%edx`:

```
movl    4(%edx),%eax            Get r->j
addl    (%edx),%eax             Add r->i
leal    8(%edx,%eax,4),%eax     Compute &r->[r->i + r->j]
movl    %eax,20(%edx)           Store in r->p
```

As these examples show, the selection of the different fields of a structure is handled completely at compile time. The machine code contains no information about the field declarations or the names of the fields.

Exercício 4.4 (*Structures*):

Considere a seguinte declaração de uma estrutura (*structure*).

```
struct prob {
    int *p;
    struct {
        int x;
        int y;
    } s;
    struct prob *next;
};
```

Esta declaração mostra que uma estrutura pode estar incluida numa outra, tal como *arrays* podem também estar incluidas em estruturas, estruturas em *arrays*, e *arrays* em *arrays*.

O procedimento seguinte (com algumas expressões omitidas) trabalha com esta estrutura:

```
void sp_init(struct prob *sp)
{
    sp->s.x = ________;
    sp->p = ________;
    sp->next = ________;
}
```

a) Qual o valor dos deslocamentos (em *bytes*) dos seguintes campos:

```
    p:
  s.x:
  s.y:
 next:
```

b) Quantos *bytes* ao todo necessita esta estrutura?

c) O compilador gera o seguinte código *assembly* para o corpo de `sp_init`:

```
    movl   8(%ebp),%eax
    movl   8(%eax),%edx
    movl   %edx,4(%eax)
    leal   4(%eax),%edx
    movl   %edx,(%eax)
    movl   %eax,12(%eax)
```

Com base neste código, preencha as expressões em falta no código C de `sp_init`.

Nº ____________ Nome ____

Exercício 4.1 (*Localização de elementos de um array*):

Array	Element Size	Total Size	Start Address	Element ***i***
S			x_S	
T			x_T	
U			x_U	
V			x_V	
W			x_W	

Exercício 4.2 (*Apontadores*):

Expression	Type	Value	Assembly Code
`S+1`			
`S[3]`			
`&S[i]`			
`S[4*i+1]`			
`S+i-5`			

Exercício 4.3 (*Matriz de 2 dimensões*):

M = ________________ **N = __________________**

Exercício 4.4 (*Structures*):

a)

```
   p: _________
 s.x: _________
 s.y: _________
next: _________
```

b) .

c)

```
movl   8(%ebp),%eax
movl   8(%eax),%edx
movl   %edx,4(%eax)
leal   4(%eax),%edx
movl   %edx,(%eax)
movl   %eax,12(%eax)
```

```
void sp_init(struct prob *sp)
{
    sp->s.x = ________;
    sp->p =   ________;
    sp->next =________;
}
```