Assembly - Arrays

תרגול 7
מערכים

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.
- 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.

Basic:

- For data type T and integer constant N, the declaration: `T A[N];`

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

Example

- Declarations:

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

- Arrays created:

Array	Element Size	Total Size	Start Address	Element $\boldsymbol{i}$
A	1	12	x_A	$x_A + i$
B	8	64	x_B	$x_B + 8i$
C	4	24	x_C	$x_C + 4i$
D	8	40	x_D	$x_D + 8i$

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 `$P`, then the expression `P+i` has value `$P+Li` where `L` is the size of data type `T`.
- &Exp is a pointer giving the address of the object Exp.
- *Address gives the value at that address.
  - Therefore A[1] = *(A+1).

Examples:

- `E` = an integer array; %rdx = $`E`; %rcx = `i`

Expression	Type	Value	Assembly code
E	int *	x_E	movq %rdx,%rax
E[0]	int	$M[x_E]$	movl (%rdx),%eax
E[i]	int	$M[x_E + 4i]$	movl (%rdx,%rcx,4),%eax
&E[2]	int *	$x_E + 8$	leaq 8(%rdx),%rax
E + i – 1	int *	$x_E + 4i - 4$	leaq -4(%rdx,%rcx,4),%rax
*(E + i – 3)	int	$M[x_E + 4i - 12]$	movl -12(%rdx,%rcx,4),%eax
&E[i] – E	long	i	movq %rcx,%rax

Arrays and Loops

- Array references within loops often have very regular patterns that can be exploited by an optimizing compiler.
  - No need for a loop variable.
  - Using pointer arithmetic I: Instead of increasing the loop variable by one, it increases our pointer by the size of the data type.
  - Using pointer arithmetic II: it computes the address of the final array element, and uses a comparison to this address as the loop test (do-while loop).

Arrays and Loops Example

- Original code:

```
long decimal5(long* x)
{
    long i;
    long val = 0;

    for (i = 0; i < 5; i++)
        val = (10 * val) + x[i];

    return val;
}
```

- Assembly code:

```
# base address of array x in %rdi
  xorq    %rax,%rax            # val = 0
  leaq    32(%rdi),%rcx        # xend =x+4 (32 bytes = 4 quad words)
.L12:
  leaq    (%rax,%rax,4),%rdx   # compute 5*val
  movq    (%rdi),%rax          # compute *x
  leaq    (%rax,%rdx,2),%rax   # compute *x + 2*5*val
  addq    $8,%rdi              # x++
  cmpq    %rcx,%rdi            # compare x : xend
  jbe     .L12                 # if <=, goto loop:
```

Nested Arrays

- The general principles of array allocation and referencing hold even when we create arrays of arrays.
- Remember: 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.

Nested Arrays (2)

- 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].

Element	Address
`A[0][0]`	$x_{\mathtt{A}}$
`A[0][1]`	$x_{\mathtt{A}} + 4$
`A[0][2]`	$x_{\mathtt{A}} + 8$
`A[1][0]`	$x_{\mathtt{A}} + 12$
`A[1][1]`	$x_{\mathtt{A}} + 16$
`A[1][2]`	$x_{\mathtt{A}} + 20$
`A[2][0]`	$x_{\mathtt{A}} + 24$
`A[2][1]`	$x_{\mathtt{A}} + 28$
`A[2][2]`	$x_{\mathtt{A}} + 32$
`A[3][0]`	$x_{\mathtt{A}} + 36$
`A[3][1]`	$x_{\mathtt{A}} + 40$
`A[3][2]`	$x_{\mathtt{A}} + 44$

- 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.

Nested Arrays (3)

- 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.
- Given `T D[R][C]`

  array element `D[i][j]` is at memory address `$D+L(Ci+j)`.

(`L` is the size of data type `T` in bytes).

Code example:

`int A[4][3];` Copy `A[i][j]` into %eax:

```
# A %rdi, i in %rsi, j in %rdx
leaq    (%rsi,%rsi,2),%rax  # compute 3*i
leaq    (%rdi,%rax,4),%rax  # compute A + 4*3*i
movl    (%rax,%rdx,4),%eax  # read from M[A + 4*(3*i+j)]
```

An exercise:

- C code:

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

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

- Assembly code:

```
# i in %rdi, j in %rsi
sum_element:
  leaq    0(,%rdi,8),%rdx
  subq    %rdi,%rdx
  addq    %rsi,%rdx
  leaq    (%rsi,%rsi,4),%rax
  addq    %rax,%rdi
  movl    mat2(,%rdi,4),%eax
  addl    mat1(,%rdx,4),%eax
  ret
```

- What is the value of M?
- What is the value of N?

An exercise:

- C code:

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

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

- Assembly code:

```
# i in %rdi, j in %rsi
sum_element:
  leaq    0(,%rdi,8),%rdx
  subq    %rdi,%rdx
  addq    %rsi,%rdx
  leaq    (%rsi,%rsi,4),%rax
  addq    %rax,%rdi
  movl    mat2(,%rdi,4),%eax
  addl    mat1(,%rdx,4),%eax
  ret
```

- What is the value of M? 5
- What is the value of N? 7

Fixed size arrays

- Enable a number of clever optimizations Using pointers arithmetic.
  - While looping through rows / columns.
  - Walking on the matrix diagonal.
  - etc.

Dynamically Allocated Arrays

- Arbitrary size arrays require dynamically allocation.
- To allocate and initialize storage for an `N*M` array of integers, we use the Unix library function `calloc`:

```
int **matrix = calloc(sizeof(int), N*M);
```

- The `calloc` function takes two arguments:
  - The size of each array element.
  - The number of array elements required.
- It attempts to allocate space for the entire array (`S*N*M` bytes). If successful, it initializes the entire region of memory to 0s and returns a pointer to the first byte. If insufficient space is available, it returns null.

Dynamically Allocated Arrays (2)

- Now, we can use the indexing computation of row-major ordering to determine the position of an element in the matrix.
- In a n*n matrix: element (i,j) will be in position i*n+j.

```c
int var_ele(long n, int A[n][n], long i, long j)
{
    return A[i][j]; // return A[(i*n) + j]
}
```

```
# n in %rdi, A in %rsi, i in %rdx, j in %rcx
var_ele:
  imulq   %rdx,%rdi            # compute n*i
  leaq    (%rsi,%rdi,4),%rax   # compute A + 4*n*i
  movl    (%rax,%rcx,4),%eax   # read from M[A + 4(i*n+j)]
  ret
```

Dynamically Allocated Arrays (2)

- Now, w[illegible]n use the indexing computation of row-m[illegible] to determine the position of an el[illegible]
- In a n*n [illegible]position i*n+j.

Notice that we must use multiplication here instead of shifts and add!

```c
int var_ele(long n, [illegible]ng i, long j)
{
    return A[i][j]; // return [illegible]i*n) + j]
}
```

```
# n in %rdi, A in %rsi, i in %rdx, j in %rcx
var_ele:
   imulq    %rdx,%rdi              # compute n*i
   leaq     (%rsi,%rdi,4),%rax     # compute A + 4*n*i
   movl     (%rax,%rcx,4),%eax     # read from M[A + 4(i*n+j)]
   ret
```