Lecture 6

<2016-04-13 Wed>

Contents

1 Arrays

1.1 1-Dimensional

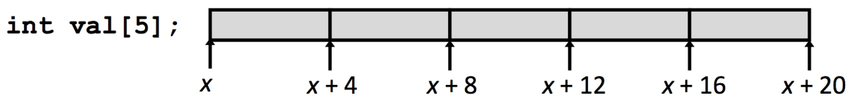
1.1.1 Array Allocation

`T A[L]`

- array of data type T and length L
- identifier A as a pointer to array element 0
- continguously allocated region of `L * sizeof(T)` bytes in memory

```
int val[5];
```

reference	type
val[4]	int
val	int*
&val[2]	int*
*(val+1)	int



1.1.2 example

```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig ucla = {1, 2, 3, 4, 5};

int get_digit(zip_dig z, int digit) {
  return z[digit];
}
```

register	variable
%rdi	z : first argument
%rsi	digit : second argument
%eax	lower-order 4 bytes of %rax

```
movl    (%rdi,%rsi,4), %eax     ;z[digit]
```

1.1.3 array loop example

```
void zincr(zip_dig z) {
  size_t i;
  for (int i = 0; i < ZLEN; i++)
    z[i]++;
}
```

register	variable
%rdi	z
%eax	i

```
zincr:
        movl      $0, %eax        ;i = 0
        jmp       .L3
.L4:
        addl      $1, (%rdi,%rax,4) ;z[i]++
        addq      $1, %rax          ;i++
.L3:
        cmpq      $4, %rax        ;compare i, ZLEN-1
        jbe       .L4             ;if <=, goto .L4
```

1.2 Multi-Dimensional (nested)

`T A[R][C]`

- 2D array of data type T
- R rows, C columns
- array size: `R * C * sizeof(T)`
- **row-major ordering**

1.2.1 Access

`T A[R][C]`

- each element of type T requires `K` bytes
- `A[i]` is an array of `C` element
  - starting address $A + i \times (C \times K)$
- `A[i][j]` is an element of type T
  - address $A + i \times (C \times K) + j \times K$

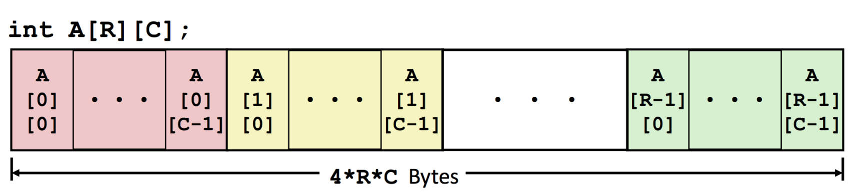


Figure 1: multidimensional array memory layout

Row-Major Ordering

```
zip_dig pgh[4];                 /* equivalent to int pgh[4][5] */

int *get_pgh_zip(int index) {
  return pgh[index];
}

int get_pgh_digit(int index, int dig) {
  return pgh[index][dig];
}
```

register	variable
%rdi	index
%rsi	dig
%rax	return value

```
get_pgh_zip:
      leaq      (%rdi,%rdi,4), %rax  ;%rax = 5 * index
      leaq      pgh(,%rax,4), %rax   ;%rax = pgh + (20 * index)

get_pgh_digit:
      leaq      (%rdi,%rdi,4), %rax  ;%rax = 5 * index
      addl      %rax, %rsi           ;%rsi += %rax
      movl      pgh(,%rsi,4), %eax   ;access memory MEM[ pgh + 4*(5*index+dig) ]
```

1.3 Multi-Level

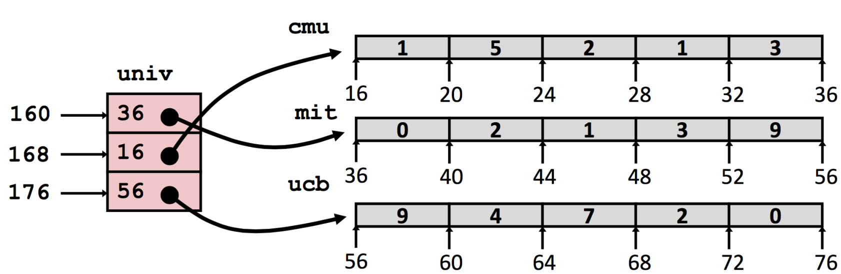


Figure 2: multi-level array

```
zip_dig ucla = {1, 2, 3, 4, 5};
```

```
#define UCOUNT 1
int *univ[UCOUNT] = {ucla};

int get_univ_digit(size_t index, size_t digit) {
  return univ[index][digit];
}

get_univ_digit:
      salq      $2, %rsi                ;%rsi *= 4
      addq      univ(,%rdi,8), %rsi     ;%rsi += univ[%rdi*8]
      movl      (%rsi), %eax            ;return memory located at address %rsi
      ret
```

- each element is a pointer
- each pointer points to an array of ints
- `univ[index][digit]` is equivalent to `MEM[ MEM[ univ+8*index ] + 4*digit ]`

1.4 Matrix

1.4.1 Fixed Dimension

```
#define N 16
typedef int fix_matrix[N][N];

int fix_ele(fix_matrix a, size_t i, size_t j) {
  return a[i][j];
}

fix_ele:
      salq      $6, %rsi                ;i *= 64  (64 = 16columns * sizeof(int))
      addq      %rsi, %rdi              ;a += i
      movq      (%rdi,%rdx,4), %eax     ;return memory at address
```

1.4.2 Variable Dimension, Implicit Indexing

```
int var_ele(size_t n, int a[n][n], size_t i, size_t j) {
  return a[i][j];
}
```

```
var_ele:
      imulq     %rdx, %rdi              ;n*i
      leaq      (%rsi,%rdi,4), %rax   ;a + 4*n*i
      movl      (%rax,%rcx,4), %eax   ;MEM[ a + 4*n*i + 4*j ]
      ret
```

1.4.3 Variable Dimension, Explicit Indexing

```
#define IDX(n, i, j) ((i)*(n)+(j))

int vec_ele(size_t n, int *a, size_t i, size_t j) {
  return a[ IDX(n, i, j) ];
}
```