Arrays

Young W. Lim

2020-07-04 Sat

Outline

1. Array Background
   - Arrays
   - Pointers
   - Loops
   - Multi-dimensional arrays
   - Fixed size arrays
   - Dynamically allocated arrays

Array Declaration

- `T A[N]`
  - allocation of contiguous region of `NL` bytes
    - `L`: the byte size of the data type `T`
    - x_A: the starting address of the region
  - introduces an identifier `A`
    - `A` can be used as a pointer to the beginning of the array
      the value of this pointer is x_A
    - `A[i]` : the i-th element is at $x_A + L \cdot i$
    - i : index between 0 and $N - 1$

Array Delaration Examples

		$x + L \cdot i$	L	N	LN
`char`	`A[12];`	$x_A + 1 \cdot i$	1	12	12
`char *`	`B[8];`	$x_B + 4 \cdot i$	4	8	32
`double`	`C[6];`	$x_C + 8 \cdot i$	8	6	48
`double *`	`D[5];`	$x_D + 4 \cdot i$	4	5	20

Accessing an array element

- `int     E[12];`
- `E[i]` access
  - $x_E + 4i$
  - `%edx` : the starting address x_E of `E`
  - `%ecx` : the index value `i`
  - `E[i]` $\rightarrow$ `%eax`
  - `movl (%edx,%ecx,4), %eax`
  - move data at (`%edx + 4 * %ecx`) to `%eax`

Pointer declaration

- `T *p;`
  - `p` : a pointer to data of type `T`
  - x_p : the value of `p`
  - $x_p + L \cdot i$: the value `p+i`
  - L : the size of data type `T`

Pointer examples (1) assumptions

- Assumptions in accessing an integer array `int E[N]`
  - `%edx` holds the starting address of integer array `E`
  - `%ecx` holds the integer index `i`
  - `leal` to generate an address
    - `E, E[0], E[i]`
    - `*(&E[i] + i), &E[i]-E`
  - `movl` | to reference memory
    - `&E[2], E+i-1`

Pointer examples (2) references and dereferences

`movl %edx, %eax`	`E`	address →	`%eax`
`movl (%edx),%eax`	`*E`	data →	`%eax`
`movl (%edx,%ecx,4), %eax`	`*(E+i)`	data →	`%eax`
`leal 8(%edx),%eax`	`E+2`	address →	`%eax`
`leal -4(%edx,%ecx,4),%eax`	`E+i-1`	address →	`%eax`
`movl (%edx,%ecx,8),%eax`	`*(E+ 2*i)`	data →	`%eax`
`movl %ecx,%eax`	`i`	index →	`%eax`

- `%edx` holds x_E (`&E[0]`),
- `%ecx` holds i

Pointer examples (3) other interpretations

`movl %edx, %eax`	`(int *)`	`E`	x_E
`movl (%edx),%eax`	`(int)`	`E[0]`	$M[x_E]$
`movl (%edx,%ecx,4), %eax`	`(int)`	`E[i]`	$M[x_E + 4i]$
`leal 8(%edx),%eax`	`(int *)`	`&E[2]`	$x_E + 8$
`leal -4(%edx,%ecx,4),%eax`	`(int *)`	`E+i-1`	$x_E + 4i - 4$
`movl (%edx,%ecx,8),%eax`	`(int)`	`*(&E[i]+i)`	$x_E + 8i$
`movl %ecx,%eax`	`(int)`	`&E[i]-E`	i

- `%edx` holds x_E (`&E[0]`),
- `%ecx` holds i

Pointer examples (4) addresses and contents

- `leal` instruction is used to generate an address

```
leal 8(%edx),%eax          E+2
leal -4(%edx,%ecx,4),%eax  E+i-1
```

- `movl` instruction is used to reference memory

```
movl (%edx),%eax           *E
movl (%edx,%ecx,4), %eax   *(E+i)
movl (%edx,%ecx,8),%ea=    *(E+ 2*i)
```

except in some cases, where it copies an address

```
movl %edx, %eax             E
```

Pointer examples (5) element access

- to compute the offset of
  the desired element `E[i]` of
  a multi-dimensional array `E`
  a `movl` instruction is used
  with the start of the array x_E as the base address (`%edx`)
  and the possibly scaled offset as an index (`%ecx * 4`)
- `movl (%edx,%ecx,4), %eax` `*(E+i)` = `E[i]`

(1) array references within loops

- very regular patterns
  that can be exploited by an optimizing compiler
- 5 decimal digit array example
- $abcd_{10} = a \cdot 10^4 + b \cdot 10^3 + c \cdot 10^2 + d \cdot 10^1 + e$
- $(((a \cdot 10 + b) \cdot 10 + c) \cdot 10 + d) \cdot 10 + e$

```
val = x[0];
val = 10 * val + x[1];   (x[0]*10 + x[1])
val = 10 * val + x[2];   ((x[0]*10 + x[1])*10 + x[2])
val = 10 * val + x[3];   (((x[0]*10 + x[1])*10 + x[2])*10 + x[3])
val = 10 * val + x[4];   ((((x[0]*10 + x[1])*10 + x[2])*10 + x[3])*10 + x[4])
```

(2) decimal5 and decimal5_opt

```
/* loop index              */
/* for loop                */
/* start, final condition */

int decimal5(int* x) {
  int i;
  int val=0;

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

  return(val);
}
```

```
/* pointer arithmetic */
/* do while loop      */
/* final condition    */

int decimal5_opt(int *x) {
  int val = 0;
  int *xend = x + 4;

  do {
    val = (10*val) + *x;
    x++;
  } while (x <= xend);

  return val;
}
```

(3) optimized c code

- rather than using a loop index *i* (`++i`)
  pointer arithmetic is used (`x++`)
  to step through successive array elements
- computes the address of the final array elements (`xend`)
  use a comparison to this address as the loop test
- a do-while loop is used since there will be at least
  one loop iteration

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

```
int *xend = x + 4;

  do {
    val = (10*val) + *x;
    x++;
  } while (x <= xend);
```

(4) decimal5_opt assembly

```
  movl 8(%ebp), %ecx
  xorl %eax, %eax
  leal 16(%ecx), %ebx
.L12:
  leal (%eax,%eax,4), %edx
  movl (%ecx), %eax
  leal (%eax,%edx,2), %eax
  addl $4, %ecx
  cmpl %ebx, %ecx
  jbe .L12
```

```
int decimal5_opt(int *x) {
  int val = 0;
  int *xend = x + 4;

  do {
    val = (10*val) + *x;
    x++;
  } while (x <= xend);

  return val;
}
```

(5) optimizations in assembly

- using `leal` to compute `5*val` as `val + 4*val`
  - `leal (%eax,%eax,4), %edx`
    `(%eax, %eax, 4) ; %eax + %eax*4 = %eax*5`
    `%eax * 5` → `%edx`
- using `leal` with a scaling factor to scale `10*val`
  - `movl (%ecx), %eax`
    now %eax has `*x` value
  - `leal (%eax,%edx,2), %eax`
    `(%eax, %edx, 2) ; %eax + %edx*2`
    `%eax` has `*x`
    `%edx` has old `%eax` * 5
    `%eax *10 + (%ecx)` → `%eax`

```
leal (%eax,%eax,4), %edx          do {
movl (%ecx), %eax                   val = (10*val) + *x;
leal (%eax,%edx,2), %eax            x++;
```

(6) assembly with low level comments

```
  movl 8(%ebp), %ecx                 ; M[%ebp+8]  -> %ecx
  xorl %eax, %eax                    ; %eax ^ %eax -> %eax
                                     ; 0 -> val
  leal 16(%ecx), %ebx                ; (%ecx+16) -> %ebx
                                     ; x + 4 -> xend
.L12:                                ; loop:
  leal (%eax,%eax,4), %edx           ; (%eax + %eax*4) -> %edx
                                     ; 5*val
  movl (%ecx), %eax                  ; M[%ecx] -> %eax
                                     ; *x
  leal (%eax,%edx,2), %eax           ; (%eax + %edx*2) -> %eax
                                     ; *x + 10*val -> val
  addl $4, %ecx                      ; 4 + %ecx -> %ecx
                                     ; x++
  cmpl %ebx, %ecx                    ; compare x :xend
  jbe  .L12                          ; if <=, goto loop
```

(7) assembly with high level comments

```
  movl 8(%ebp), %ecx             ; get base address of array x
  xorl %eax, %eax                ; val = 0
  leal 16(%ecx), %ebx            ; xend = x+4 (16 bytes = 4 dwords)
.L12:                            ; loop:
  leal (%eax,%eax,4), %edx       ; compute 5 *val
  movl (%ec), %eax               ; compute *x
  leal (%eax,%edx,2), %eax       ; compute *x + 2 * (5 * val)
  addl $4, %ecx                  ; x++
  cmpl %ebx, %ecx                ; compare x :xend
  jbe  .L12                      ; if <=, goto loop
```

Nested array view

- `int A[4][3]`
- `typedef int Row [3]`
  `Row A[4]`
  - data type `Row` is defined to be an array of three integers
    `int` ☐ `[3]`
  - array `A` contains four such arrays
    `Row` ☐ `[4]`
  - each `A[i]` requiring 12 bytes to store the three integers
  - the total array size is then 4*4*3 = 48 bytes

Multi-dimensional array view

- `int A[4][3]`
- a 2-dimensional array `A` with 4 rows and 3 columns referenced as `A[0][0]` trought `A[3][2]`
  - row major order
    all elements of row 0 followed by
    all elements of row 1, and so on
- viewing `A` as an array of 4 elments (`Row [4]`), each of which is an array of 3 `int`'s (`int [3]`)
  - first `A[0]` (row 0) followed by
    second `A[1]` (row 1), and so on

Row major order (1)

A[i][j]	$x_A + (i * 3 + j) * 4$	row i	col j
A[0][0]	$x_A + (0 * 3 + 0) * 4$	row 0	col 0
A[0][1]	$x_A + (0 * 3 + 1) * 4$		col 1
A[0][2]	$x_A + (0 * 3 + 2) * 4$		col 2
A[1][0]	$x_A + (1 * 3 + 0) * 4$	row 1	col 0
A[1][1]	$x_A + (1 * 3 + 1) * 4$		col 1
A[1][2]	$x_A + (1 * 3 + 2) * 4$		col 2
A[2][0]	$x_A + (2 * 3 + 0) * 4$	row 2	col 0
A[2][1]	$x_A + (2 * 3 + 1) * 4$		col 1
A[2][2]	$x_A + (2 * 3 + 2) * 4$		col 2
A[3][0]	$x_A + (3 * 3 + 0) * 4$	row 3	col 0
A[3][1]	$x_A + (3 * 3 + 1) * 4$		col 1
A[3][2]	$x_A + (3 * 3 + 2) * 4$		col 2

Row major order (2)

`A[i][j]`	$x_A + (i * 3) * 4 + j * 4$	$A[i] + j * 4$	row i	col j
`A[0][0]`	$x_A + (0 * 3) * 4 + 0 * 4$	$A[0] + 0 * 4$	row 0	col 0
`A[0][1]`	$x_A + (0 * 3) * 4 + 1 * 4$	$A[0] + 1 * 4$		col 1
`A[0][2]`	$x_A + (0 * 3) * 4 + 2 * 4$	$A[0] + 2 * 4$		col 2
`A[1][0]`	$x_A + (1 * 3) * 4 + 0 * 4$	$A[1] + 0 * 4$	row 1	col 0
`A[1][1]`	$x_A + (1 * 3) * 4 + 1 * 4$	$A[1] + 1 * 4$		col 1
`A[1][2]`	$x_A + (1 * 3) * 4 + 2 * 4$	$A[1] + 2 * 4$		col 2
`A[2][0]`	$x_A + (2 * 3) * 4 + 0 * 4$	$A[2] + 0 * 4$	row 2	col 0
`A[2][1]`	$x_A + (2 * 3) * 4 + 1 * 4$	$A[2] + 1 * 4$		col 1
`A[2][2]`	$x_A + (2 * 3) * 4 + 2 * 4$	$A[2] + 2 * 4$		col 2
`A[3][0]`	$x_A + (3 * 3) * 4 + 0 * 4$	$A[3] + 0 * 4$	row 3	col 0
`A[3][1]`	$x_A + (3 * 3) * 4 + 1 * 4$	$A[3] + 1 * 4$		col 1
`A[3][2]`	$x_A + (3 * 3) * 4 + 2 * 4$	$A[3] + 2 * 4$		col 2

Accessing multi-dimensional arrays

- the compiler generates code to compute the offset of the desired element
- then use a `movl` instruction
  - the start of the array as the base address
  - the (possibly scaled) offset as an index

Accessing 2-dimensional arrays

- computing the offset of the desired element
  - `T D[R][C];`
    array element `D[i][j]` is at memory address
    $x_D + (i \cdot C + j) \cdot L$
    where L is the size of the type T
- then use a `movl` instruction
  with a base address and a scaled index
  - `movl (%eax, %edx), %eax`

Accessing 2-dimensional array examples

- `int A[4][3]`
  - %eax contains x_A
  - %edx holds i
  - %ecx holds j
  - copy `A[i][j]` to %eax

```
sall $2, %ecx                ; %ecx*4           ; j*4 -> %ecx
leal (%edx, %edx, 2), %edx   ; %edx + %edx*2    ; i*3 -> %edx
leal (%ecx, %edx, 4), %edx   ; %ecx + %edx*4    ; j*4 + i*3*4 -> %edx
movl (%eax, %edx), %eax      ; %eax + %edx      ; M[xA + 4(i*3 + j)] -> %eax
```

```
sal (shift arithmetic left)        sar (shift arithmetic right)
shl (hsift logical left)           shr (shift logical right)
```

Fixed size arrays (1)

- an array with a known constant size
  an array of constant known size

- ```
  #define N 16
  typedef int fmatrix[N][N];
  ```
  ```

Fixed size arrays (2)

- dot product example
  - *i*-th row of `A[i][j]`
  - *k*-th column of `B[j][k]`

```
int fprod(fmatrix A, fmatrix B, int i, int k)
{
   int j;  int result = 0;

   for (j=0; j<N; j++)
      result += A[i][j] * B[j][k];

   return result;
}
```

Fixed size arrays (3)

- the c compiler is able to make many optimizations
  for code operating on multi-dimensional arrays of fixed size
- the loop will access the *i*-th row elements of array `A`
  `A[i][0]`, `A[i][1]`, ... , `A[i][15]` in sequence
- these elements occupy adjacent locations in memory
- use a pointer `Ap` to access the successive locations
  `Ap + 1=`

Fixed size arrays (4)

- the loop will access the k-th column elements of array `B`
  `B[0][k], B[1][k], ... , B[15][k]` in sequence
- these elements occupy 64-byte bytes apart locations in memory
- use a pointer `Bp` to access these successive locations
  `Bp += N`
- in c, this pointer is shown as being incremented by $N = 16$,
  althogh in fact the actual pointer is incremented by 4*16=64 bytes

Fixed Size Array (1) fprod and fprod_opt

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

int fprod(fmatrix A,
fmatrix B, int i, int k)
{
  int j;
  int result = 0;

  for (j=0; j<N; j++)
    result +=
      A[i][j] * B[j][k];

  return result;
}
```

```
int fprod_opt(fmatrix A,
fmatrix B, int i, int k) {
  int *Ap = &A[i][0];
  int *Bp = &B[0][k];
  int cnt = N - 1;
  int result = 0;

  do {
    result += (*Ap) * (*Bp);
    Ap += 1;
    Bp += N;
    cnt--;
  } while (cnt >= 0);
  return result;
}
```

Fixed Size Array (2) assembly for `fprod`

```
.L23:
  movl (%edx), %eax
  imull (%ecx), %eax
  addl %eax, %esi
  addl $64, %ecx
  addl $4, %edx
  decl %ebx
  jns .L23
```

```
int fprod_opt(fmatrix A,
fmatrix B, int i, int k) {
  int *Ap = &A[i][0];
  int *Bp = &B[0][k];
  int result = 0;

  do {
    result += (*Ap) * (*Bp);
    Ap += 1;  // 4 bytes stride
    Bp += N;  // 4*16 = 64 bytes stride
    cnt--;
  } while (cnt >= 0);
  return result;
}
```

Fixed Size Array (3) assembly with comments

```
.L23:
  movl (%edx), %eax       ; M[%edx] -> %eax          ; compute t = *Ap
  imull (%ecx), %eax      ; M[%ecx] * %eax -> %eax   ; compute v = *Bp + t
  addl %eax, %esi         ; %eax + %esi -> %esi      ; add v result
  addl $64, %ecx          ; 64 + %ecx -> %ecx        ; add 64 to Bp
  addl $4, %edx           ; 4 + %edx -> %edx         ; add 4 to Ap
  decl %ebx               ; %ebx -1 -> %ebx          ; decrement cnt
  jns .L23                                           ; if >=, goto loop
```

Arbitrary Size Array

- one-dimensional arrays of variable size
  no known constant size
  `int []` in a function argument

- multi-dimensional arrays of variable size
  when all the sizes are known at the compile time
  except the size of the first dimension
  `int [][L][M][N]` in a function argument

Dynamic Memory Allocation

- the heap is a pool of memory available for storing data structures
- storage on the heap is allocated
  using the library functions
    - malloc allocates uninitialized size bytes
      `void * malloc(size_t size)`
    - calloc allocates initialized nmemb elements of size bytes
      `void * calloc(size_t nmemb, size_t size)`
- C requiresthe program to explicitly free allocated sapce
  using the library function free
    - `void free(void *ptr)`

Dynamically allocated Arrays (1)

- define a `vmatrix` type as simply as `int *`
  `typedef int *vmatrix;`
- to allocate and initialize storage
  for an $n \times n$ array of integers,
  `calloc` library function can be used
  `calloc(sizeof(int), n*n);`

```
var_matrix new_var_matrix(int n)
{
   return (vmatrix) calloc(sizeof(int), n*n);
}
```

Dynamically allocated Arrays (2)

- in many applications, a code is required to work for arbitrary size arrays that have been dynamically allocated
- for these we must explicitly enocde the mapping of multi-dimensional arrays into one-dimensional ones

Dynamically allocated Arrays (3)

- `calloc` library function has two arguments
  - the size of each array element
  - the number of array elements required
- attempts to allocate space for the entire array
  - if successful,
    it initializes the entire region of memory to 0s
  - if sufficient space is not available,
    it returns null

row-major order index computation examples

- ```
  int var_ele (var_matrix A, int i, int j, int n)
  {
     return A[(i*n)+j];
  }
  ```

- ```
  movl 8(%ebp), %edx            ; Get A
  movl 12(%ebp), %eax           ; Get i
  imull 20(%ebp), %eax          ; Compute n*i
  addl 16(%ebp), %eax           ; Compute n*i + j
  movl (%edx, %eax, 4), %eax    ; Get A[i*n + j]
  ```

Comparing index computations (1)

- dynamic version is somewhat more complex
  must use a multiply instruction to scale i by n
  rather than a series of shifts and adds
- this multiplication is not significant performance penalty
- in many cases, the compiler can simplify
  the index computations for variable sized arrays
  using the same principles as the fixed array case

Comparing index computations (2)

- rather than generating a pointer variable Bptr
  the compiler creates an integer variable nTjPk
  for n Times j Plus k
  since its value n*j+k relative to the original code
- initially, nTjPk equals k, and
  it is incremented by n by each iteration

Index computations for dynamically allocated arrays

- in many cases, the compiler can simplify
  the index computation for variable sized arrays
  using the same principles as for fixed-size arrays
- the compile is able to eliminate the integer multiplication
  $i * n$ and $j * n$ by exploiting the sequential access patter
  resulting from the loop structures

Register Spilling (1)

- variables B and n must be retrievd from memory on each iteration
- register spilling
- the compiler chose to spill variables B and n because they are read only they do not change value within the loop
- Spilling is a common problem for IA32, since the processor has so few registers

Register Spilling (2)

- not enough registers to hold all the needed temporary data
- must keep some local variables in memory
- `register spilling`

Dynamically Allocated Array (1)

```
typedef int *vmatrix;
. . .
vmatrix A  // int *A;
. . .
vmatrix func(int n) {
  return (vmatrix)
    calloc(
      sizeof(int), n*n);
}
. . .
int foo(vmatrix A, int i,
int j, int n) {
  return A[(i*n) + j);
}
```

```
movl 8(%ebp), %edx
movl 12(%ebp), %eax
imull 20(%ebp), %eax
addl 16(%ebp), %eax
movl (%eax, %eax, 4), %eax
```

Dynamically Allocated Array (2)

```
typedef int *vmatrix;

int vprod(vmatrix A,
vmatrix B, int i,
int k, int n) {
  int j;
  int result = 0;

  for (j=0; j<n; j++) {
    result +=
      A[i*n+j] * B[j*n+k];

  return result;
}
```

```
int vprod(vmatrix A,
vmatrix B, int i,
int k, int n) {
  int *Ap = &A[i*n];
  int nT = n, result = 0;

  if (n <= 0) return result;
  do {
    result += (*Ap) * B[nT];
    Ap++;
    nT+= n;
    cnt--;
  } while (cnt);
  return result;
}
```

Dynamically Allocated Array (3)

```
.L37:
  movl 12(%ebp), %eax
  movl (%ebx), %edi
  addl $4, %ebx
  imull (%eax,%ecx,4), %edi
  addl %edi, %esi
  addl 24(%ebp), %ecx
  decl %edx
  jnz .L37
```

```
int vprod(vmatrix A,
vmatrix B, int i,
int k, int n) {
  int *Ap = &A[i*n];
  int nT = n, result = 0;

  if (n <= 0) return result;
  do {
    result += (*Ap) * B[nT];
    Ap++;
    nT+= n;
    cnt--;
  } while (cnt);
  return result;
}
```