

Arrays

Young W. Lim

2021-10-18 Mon

Outline

- 1 Based on
- 2 Arrays
- 3 Pointers
- 4 Arrays and Loops
- 5 Multi-dimensional arrays
- 6 Fixed size arrays
- 7 Dynamically allocated arrays
- 8 Dot product examples

- 1 "Self-service Linux: Mastering the Art of Problem Determination",

Mark Wilding

- 1 "Computer Architecture: A Programmer's Perspective", Bryant & O'Hallaron

I, the copyright holder of this work, hereby publish it under the following licenses: GNU head Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled GNU Free Documentation License.

CC BY SA This file is licensed under the Creative Commons Attribution ShareAlike 3.0 Unported License. In short: you are free to share and make derivative works of the file under the conditions that you appropriately attribute it, and that you distribute it only under a license compatible with this one.

Compiling 32-bit program on 64-bit gcc

- `gcc -v`
- `gcc -m32 t.c`
- `sudo apt-get install gcc-multilib`
- `sudo apt-get install g++-multilib`
- `gcc-multilib`
- `g++-multilib`
- `gcc -m32`
- `objdump -m i386`

- Array declaration
- Array declaration examples
- Accessing an array element

Array declaration

```
T A[N];           // each element has the data type T
                  // N such elements
```

- 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 Declaration Examples

- allocation of contiguous region of LN bytes
 - L : the byte size of the data type T
 - N : the number of elements in an array
 - x_A, x_B, x_C, x_D : the starting address of each array

		$x + L \cdot i$	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

Array declaration example

```
int E[12];      // each element has the data type int
                // 12 such elements
```

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

- Pointer declaration
- Addressing modes
- Pointer examples

Pointer declaration

```
T *p;           // the type T of the pointed data *p
                // th pointer variable p
```

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

Addressing Modes

Imm		M[Imm]	Absolute
	(Ea)	M[R[Ea]]	Indirect
Imm	(Eb)	M[Imm + R[Eb]]	Base + displace
	(Eb, Ei)	M[R[Eb] + R[Ei]]	Indexed
Imm	(Eb, Ei)	M[Imm + R[Eb] + R[Ei]]	Indexed
	(, Ei, s)	M[R[Ei]*s]		Scaled Indexed
Imm	(, Ei, s)	M[Imm + R[Ei]*s]		Scaled Indexed
	(Eb, Ei, s)	M[R[Eb] + R[Ei]*s]		Scaled Indexed
Imm	(Eb, Ei, s)	M[Imm + R[Eb] + R[Ei]*s]		Scaled Indexed

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

<code>%edx</code>	x_E	E
<code>(%edx)</code>	$M[x_E]$	$*E$
<code>(%edx,%ecx,4)</code>	$M[x_E + i \cdot 4]$	$*(E+i)$
<code>8(%edx)</code>	$M[x_E + 2 \cdot 4]$	$E+2$
<code>-4(%edx,%ecx,4)</code>	$M[x_E + i \cdot 4 - 1 \cdot 4]$	$E+i-1$
<code>(%edx,%ecx,8)</code>	$M[x_E + i \cdot 2 \cdot 4]$	$*(E+ 2*i)$
<code>%ecx</code>	i	i

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

Pointer examples (3) references and dereferences

<code>movl %edx, %eax</code>	E	address \rightarrow	<code>%eax</code>
<code>movl (%edx), %eax</code>	$*E$	data \rightarrow	<code>%eax</code>
<code>movl (%edx,%ecx,4), %eax</code>	$*(E+i)$	data \rightarrow	<code>%eax</code>
<code>leal 8(%edx), %eax</code>	$E+2$	address \rightarrow	<code>%eax</code>
<code>leal -4(%edx,%ecx,4), %eax</code>	$E+i-1$	address \rightarrow	<code>%eax</code>
<code>movl (%edx,%ecx,8), %eax</code>	$*(E+ 2*i)$	data \rightarrow	<code>%eax</code>
<code>movl %ecx, %eax</code>	i	index \rightarrow	<code>%eax</code>

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

Pointer examples (4) other interpretations

<code>movl %edx, %eax</code>	<code>(int *)</code>	<code>E</code>	x_E
<code>movl (%edx), %eax</code>	<code>(int)</code>	<code>E[0]</code>	$M[x_E]$
<code>movl (%edx,%ecx,4), %eax</code>	<code>(int)</code>	<code>E[i]</code>	$M[x_E + i \cdot 4]$
<code>leal 8(%edx), %eax</code>	<code>(int *)</code>	<code>&E[2]</code>	$x_E + 2 \cdot 4$
<code>leal -4(%edx,%ecx,4), %eax</code>	<code>(int *)</code>	<code>E+i-1</code>	$x_E + (i - 1) \cdot 4$
<code>movl (%edx,%ecx,8), %eax</code>	<code>(int)</code>	<code>*(&E[i]+i)</code>	$x_E + 2i \cdot 4$
<code>movl %ecx, %eax</code>	<code>(int)</code>	<code>&E[i]-E</code>	i

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

Pointer examples (5) 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 **movl** copies an address

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


Pointer examples (6) 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` $*(E+i) = E[i]$
`M[%edx + %ecx * 4]`

TOC: Arrays and Loops

- (1) array references within loops
- (2) `decimal5` and `decimal5_opt`
- (3) optimized c code
- (4) `decimal5_opt` assembly
- (5) optimizations in assembly
- (6) optimizations in assembly
- (7) assembly with low level comments
- (8) assembly with high level comments

(1) 5 decimal array example

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

```
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

decimal5 source code

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

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

    return val;
}
```

- loop index
- for loop
- start, final condition

decimal5_opt source code

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

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

    return val;
}
```

- pointer arithmetic
- do while loop
- final condition

(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

decimal5 source code

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

decimal5_opt source code

```
int *xend = x + 4;  
  
do {  
    val = (10*val) + *x;  
    x++;  
} while (x <= xend);
```

(4) decimal5_opt assembly

decimal5_opt assembly code

```
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
```

decimal5_opt source code

```
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

optimized assembly code

```
leal (%eax,%eax,4), %edx
movl (%ecx), %eax
leal (%eax,%edx,2), %eax
```

source code

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

- using leal to compute $5*val$ as $val+4*val$

<code>(%eax, %eax, 4)</code>	<code>%eax + %eax*4 = %eax*5</code>
<code>leal (%eax,%eax,4), %edx</code>	<code>%eax * 5 → %edx</code>

(6) optimizations in assembly

optimized assembly code

```
leal (%eax,%eax,4), %edx
movl (%ecx), %eax
leal (%eax,%edx,2), %eax
```

source code

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

- using `leal` with a scaling factor to scale `10*val`

<code>movl (%ecx), %eax</code>	<code>*x → %eax</code>
<code>(%eax, %edx, 2)</code>	<code>%eax + %edx*2</code>
	<code>*x + (%eax *5) *2</code>
<code>leal (%eax,%edx,2), %eax</code>	<code>*x +10*%eax → %eax</code>

(7) assembly with low level comments

decimal5_opt assembly code (1)

```
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
```

(8) assembly with high level comments

decimal5_opt assembly code (2)

```
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 (%ecx), %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
```

TOC: Multi-dimensional arrays

- Nested array view
- Multi-dim array view example
- Nested array view example
- Row major order
- Accessing multi-dimensional arrays
- Accessing 2-dimensional arrays
- Accessing 2-dimensional arrays examples

Nested array view

2-d array declaration

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

nested array declaration

```
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

2-d array declaration

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

- a 2-dimensional array A with 4 rows and 3 columns referenced as A[0][0] through A[3][2]
 - **row major order**
all elements of row 0 followed by all elements of row 1, and so on

Nested array view example

2-d array declaration

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

- viewing A as an array of 4 elements (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 (1)

2-d array declaration

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

- %eax contains x_A
- %edx holds i
- %ecx holds j
- copy $A[i][j]$ to %eax

Accessing 2-dimensional array examples (2)

2-d array declaration

```
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 (shift logical left)
- shr (shift logical right)

- Fixed size arrays
- Dot product example of fixed size arrays
- Fixed size array (1) `fprod` and `fprod_opt`
- Fixed size array (2) assembly for `fprod`
- Fixed size array (3) assembly with comments

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

constant size

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

Dot product example of fixed size arrays (1)

Dot product example

```
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;
}
```

- i -th row of $A[i][j]$
- k -th column of $B[j][k]$

Dot product example of fixed size arrays (2)

- 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 $A[i][0], A[i][1], \dots, A[i][15]$ in sequence
- these elements occupy adjacent locations in memory
- use a pointer A_p to access the successive locations
 $A_p += 1$

Dot product example of fixed size arrays (3)

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

Fixed size array (1) fprod and fprod_opt

fprod source code

```
#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;
}
```

fprod_opt source code

```
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;
        cnt--;
    } while (cnt >= 0);

    return result;
}
```

Fixed size array (2) assembly for fprod

fprod assembly code

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

fprod_opt source code

```
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; // 64 bytes stride  
        cnt--;  
    }  
    return result;  
}
```

Fixed size array (3) assembly with comments

fprod assembly code 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

Dynamically allocated arbitrary size Arrays

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

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 requires the program to explicitly free allocated space using the library function **free**
 - `void free(void *ptr)`

Dynamically allocated Arrays (1)

- define a `vmatrix` type as simply as `int *`

vmatrix definition

```
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);`

new_vmatrix definition

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


Dynamically allocated Arrays (2)

- `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

Accessing a dynamically allocated array example (1)

dynamically allocated array type

```
typedef int *vmatrix;  
  
vmatrix A // int *A;
```

dynamic allocation

```
vmatrix new_vmatrix(int n) {  
    return (vmatrix) calloc( sizeof(int), n*n );  
}  
  
. . .
```

accessing

```
int var_elem (vmatrix A, int i, int j, int n)  
{  
    return A[i*n +j];  
}
```

Accessing a dynamically allocated array example (2)

accessing function parameters

```
int var_elem (vmatrix A, int i, int j, int n)
{
    return A[(i*n)+j];
}
```

%ebp + 4	Return Address
%ebp + 8	A (array name)
%ebp +12	i (row index)
%ebp +16	j (column index)
%ebp +20	n (column size)

Accessing a dynamically allocated array example (3)

var_elem assembly with comment 1

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

var_elem assembly with comment 2

```
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)

fixed size array $A[4][4]$

```
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
```

variable size array $A[] [n]$

```
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 (2)

- dynamic version is somewhat more complex
 - unknown row size (the 1st dimension is not known)
 - must use a **multiply** instruction to scale i by n rather than a series of shifts (sals) and adds
- this multiplication is not significant performance penalty for modern processors

Index computation for dynamically allocated arrays

- in many cases, the compiler can simplify the index computations for variable sized arrays using the same principles as the fixed array case
- the compiler is able to eliminate the integer **multiplication** by exploiting the **sequential access pattern** resulting from the loop structures

Register Spilling

- Spilling is a common problem for IA32, since the processor has so few registers
- not enough registers to hold all the needed temporary data
- must keep some local variables in memory
- **register spilling**

Dynamically Allocated Array (1)

vprod ver 1

```
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;
}
```

vprod ver 2

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

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

Dynamically Allocated Array (2)

vprod ver 2 assembly

```
.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
```

```
%ebp + 4  Return Address  
%ebp + 8  A (array name)  
%ebp +12  B (array name)  
%ebp +16  i (row index)  
%ebp +20  j (column index)  
%ebp +24  n (column size)
```

vprod ver 2

```
int vprod(vmatrix A,  
          vmatrix B,  
          int i, int k,  
          int n)  
{  
    int *Ap = &A[i*n];  
    int njk = k, cnt = n, result = 0;  
  
    if (n <= 0) return result;  
    do {  
        result += (*Ap) * B[njk];  
        Ap++;  
        njk += n;  
        cnt--;  
    } while (cnt);  
    return result;  
}
```

Dynamically Allocated Array (3)

vprod ver 2 assembly with comment

```
.L37:
    movl 12(%ebp), %eax      ; M[%ebp + 12] -> %eax
    movl (%ebx), %edi       ; M[%ebx] -> %edi
    addl $4, %ebx          ; %ebx + 4 -> %ebx
    imull (%eax,%ecx,4), %edi ; M[%eax + %ecx*4] * %edi -> %edi
    addl %edi, %esi        ; %edi + %esi -> %esi
    addl 24(%ebp), %ecx     ; M[%ebp + 24] + %ecx -> %ecx
    decl %edx              ; -1 + %edx -> %edx
    jnz .L37               ; if cnt != 0, goto loop
```

vprod ver 2 assembly legends

%ebp + 4	Return Address	%edx holds cnt
%ebp + 8	A (array name)	%ebx holds Ap
%ebp +12	B (array name)	%ecx holds njk
%ebp +16	i (row index)	%esi holds results
%ebp +20	j (column index)	
%ebp +24	n (column size)	

Dynamically Allocated Array (4)

vprod ver 2 assembly with comment2

```
.L37:
    movl 12(%ebp), %eax      ; B -> %eax           %eax: B
    movl (%ebx), %edi       ; *Ap -> %edi        %edi: *Ap
    addl $4, %ebx           ; Ap + 1 -> Ap       %ebx: Ap
    imull (%eax,%ecx,4), %edi ; B[njk] * *Ap -> %edi %edi: B[njk] * *Ap
    addl %edi, %esi         ; %edi + result -> %esi %esi: result
    addl 24(%ebp), %ecx     ; n + njk -> %ecx    %ecx: njk
    decl %edx               ; cnt -1 -> %edx     %edx: cnt
    jnz .L37                ; if cnt != 0, goto loop
```

vprod ver 2 assembly legends

%ebp + 4	Return Address	%edx holds cnt
%ebp + 8	A (array name)	%ebx holds Ap
%ebp +12	B (array name)	%ecx holds njk
%ebp +16	i (row index)	%esi holds results
%ebp +20	j (column index)	
%ebp +24	n (column size)	

Dot product examples of variable size arrays (1)

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

vprod ver 2 source

```
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;
}
```

Dot product examples of variable size arrays (2)

- $A[i*n+j] * B[j*n+k]$
- eliminate the integer multiplication $i*n$ and $j*n$ by exploiting the **sequential access pattern**
- rather than generating a pointer variable B_p the compiler creates an integer variable njk for n Times j Plus k since its value $n*j+k$ relative to the original code
- initially, njk equals k , and it is incremented by n by each iteration

Dot product examples of variable size arrays (3)

- i -th row of A and k -th column of B
 $A[i*n+j] * B[j*n+k]$
- accessing i -th row of A
 - a pointer type `int *Ap`
 - assign the starting address of i -th row `Ap = &A[i*n+0]`
`Ap` points to the first element of the i -th row
 - sequentially access n elements by incrementing the pointer `Ap++`
another loop variable `cnt = n, n-1, ..., 1`
 - after `Ap = &A[i*n]`, `*Ap` accesses `A[i*n+0]`
 - after `Ap++`, `*Ap` accesses `A[i*n+1]`
 - after `Ap++`, `*Ap` accesses `A[i*n+2]`

Dot product examples of variable size arrays (4)

- i -th row of A and k -th column of B
 $A[i*n+j] * B[j*n+k]$
- accessing k -th column of B
 - each column element has a stride of n
 - increments njk by n : $njk + n;=$
 - $njk = k, k+n, k+2n, k+3n, \dots$
 - $B[0*n+k]$ the first element of the k -th column
 - $B[1*n+k]$ the second element of the k -th column
 - $B[2*n+k]$ the third element of the k -th column

Dot product examples of variable size arrays (5)

Assumptions

- %edx holds cnt
- %ebx holds Ap
- %ecx holds njk
- %esi holds results

partial assembly listing

```
.L37:  
    movl 12(%ebp), %eax      ; Get B  
    movl (%ebx), %edi       ; Get Ap  
    addl $4, %ebx           ; Inc Ap  
    imull (%eax,%ecx,4), %edi ; multiply by B[njk]  
    addl %edi, %esi         ; add to result  
    addl 24(%ebp), %ecx     ; add n to njk  
    decl %edx               ; dec cnt  
    jnz .L37                ; if cnt != 0, goto loop
```

Dot product examples of variable size arrays (6)

- variables B and n must be retrieved 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

register spilling

```
movl 12(%ebp), %eax      ; B -> %eax      %eax: B
addl 24(%ebp), %ecx      ; n + njk -> %ecx  %ecx: njk
```

```
%ebp + 4  Return Address      %edx holds cnt
%ebp + 8  A (array name)      %ebx holds Ap
%ebp +12  B (array name)      %ecx holds njk
%ebp +16  i (row index)       %esi holds results
%ebp +20  j (column index)
%ebp +24  n (column size)
```