Pointers and Arrays

Every pointer has a type

Indicates what kind of thing the pointer points to

You can cast pointers to be pointers to other types.

This is powerful and also dangerous.

Common example: return value of malloc

Every pointer has a value

Pointer value

- **Address of an object of a particular type**
- **All pointers are 4 bytes (32-bits) for IA-32**

Created via the "&" operator

Can use "&" on all "lvalues"

Anything that can appear on the left-hand side of an assignment

Dereferenced via the "*" operator

Result is a value having type associated with pointer

Arrays and pointers closely related

Name of array can be referenced as if it were a pointer

Array referencing equivalent to pointer arithmetic and dereferencing (i.e. a[3] = *(a+3))

Pointers and arrays review

Assume array z[10]

- **z[i] returns i^{th} element of array z**
- **&z[i] returns the address of the i^{th} element of array z**
- **z alone returns address the array begins at or the address of the 0th element of array z (&z[0])**

```
int* ip;
int z[10];
ip = z;     /* equivalent to ip = &z[0]; */
z[5] = 3;   /* equivalent to ip=&z[5]; *ip=3 */
```

Pointers and arrays review

Recall: pointer arithmetic done based on type of pointer

```
char* cp1;
int* ip1;
double* dp1;
cp1++;  // Increments address by 1
ip1++;  // Increments address by 4
dp1++;  // Increments address by 8
```

Using pointer arithmetic with arrays requires knowledge of array allocation

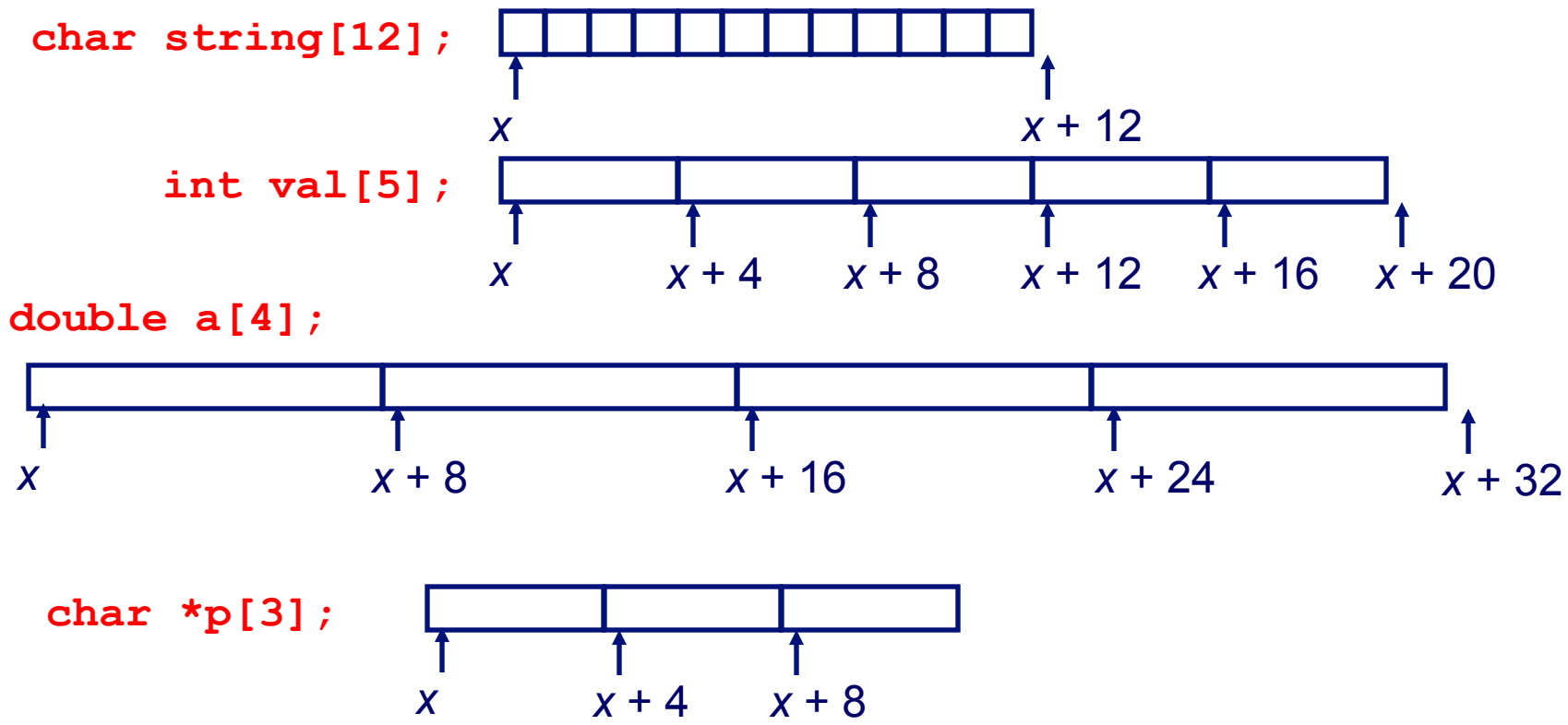
Array allocation

Basic Principle

T `A[`*L*`];`

Array of data type *T* and length *L*

***Contiguously allocated* region of *L* * `sizeof(`*T*`)` bytes**



Pointer arithmetic with arrays

As a result of contiguous allocation

- **Elements accessed by scaling the index by the size of the datum and adding to the start address**
- **Done via scaled index memory mode**

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

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	$x_D + 4i$

Practice Problem

```
short          S[7];
short         *T[3]
short        **U[6];
```

Array	Element Size	Total Size	Start Address	Element *i*
S			x_S	
T			x_T	
U			x_U	

Practice Problem

```
short    S[7];
short   *T[3]
short   **U[6];
```

Array	Element Size	Total Size	Start Address	Element *i*
S	`2`	`14`	x_S	`x+2i`
T	`4`	`12`	x_T	`x+4i`
U	`4`	`24`	x_U	`x+4i`

Arrays as function arguments

The basic data types in C are passed by value.

What about arrays?

Example:

```
int exp[32000000];

int x = foo(exp);
```

What is the declaration of the function foo?

```
int foo(int* f) { … }
```

The name of an array is equivalent to what?

Pointer to the first element of array!
Arrays are passed by reference

Arrays of pointers

Arrays of pointers are quite common in C

Example: print out name of month given its number

```
#include <stdlib.h>
#include <stdio.h>

char *monthName(int n) {
       static char *name[] = {
          "Illegal month", "January", "February", "March",
          "April", "May", "June", "July", "August",
          "September", "October", "November", "December"
       };
       return ( n < 1 || n > 12 ) ? name[0] : name[n];
}

int main(int argc, char *argv[]) {
        if (2 != argc) {
                fprintf(stderr, "Usage: %s <int>\n",
                                argv[0]);
                return 0;`
        }
        printf("%s\n", monthName(atoi(argv[1])));
        return 0;
}
```

argv

Command line arguments are passed in the `argv` array

Prototype for main routine

- `int main(int argc, char *argv[ ]);`
- `argv` is an array of what kind of things?

Can be declared like this

- `int main(int argc, char **argv);`
  - `argv` => `&argv[0]`
  - `argv[0]` => `char*`

Problem

Consider the following code

```
char *pLines[3];
char *a="abc";
char *d=”def";
char *g=”ghi";

pLines[0]=a;
pLines[1]=d;
pLines[2]=g;
```

What are the types and values of

- `pLines`
- `pLines[0]`
- `*pLines`
- `*pLines[0]`
- `**pLines`
- `pLines[0][0]`

Problem

Consider the following code

```
char *pLines[3];
char *a="abc";
char *d=”def";
char *g=”ghi";

pLines[0]=a;
pLines[1]=d;
pLines[2]=g;
```

What are the types and values of

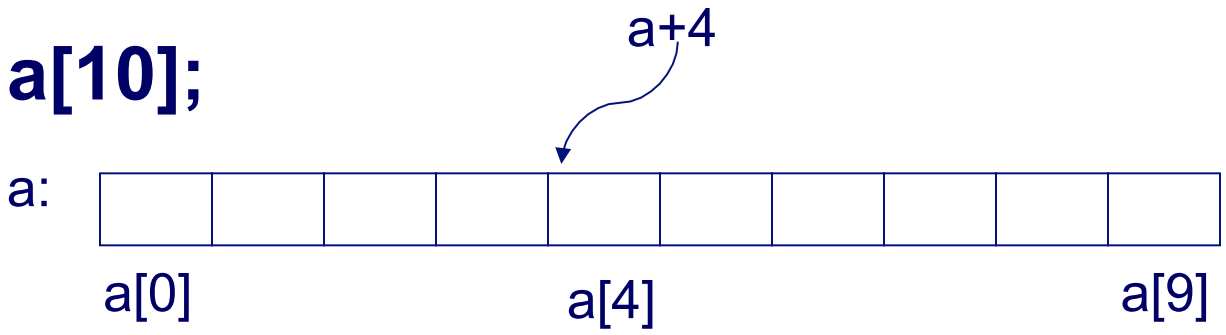
• pLines	char **	pLines
• pLines[0]	char *	a
• *pLines	char *	a
• *pLines[0]	char	‘a’
• **pLines	char	‘a’
• pLines[0][0]	char	‘a’

Array access in C

Arrays can be accessed two ways

- **Via index (i.e. a[9])**
- **Via pointer arithmetic (i.e. *(a+9))**

int a[10];




```
(a+N)   ==   &a[N]
*(a+N)  ==   a[N]
```

Array access examples

Pointer arithmetic

int E [20];

%eax == result

%edx == start address of E

%ecx == index i

Expression	Type	Value	Assembly Code
E	int *	X_E	
E[0]	int	M[X_E]	
E[i]	int	M[$X_E + 4i$]	
&E[2]	int *	$X_E + 8$	
E+i-1	int *	X_E + 4i - 4	
*(&E[i]+i)	int	M[$X_E + 4i + 4i$]	

Array access examples

Pointer arithmetic

int E [20];

%eax == result

%edx == start address of E

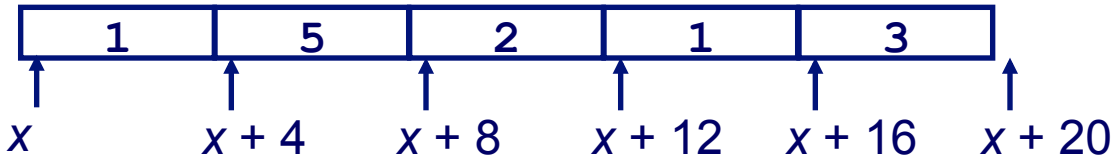
%ecx == index i

Expression	Type	Value	Assembly Code
E	int *	X_E	movl %edx, %eax
E[0]	int	M[X_E]	movl (%edx), %eax
E[i]	int	M[$X_E + 4i$]	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	M[$X_E + 4i + 4i$]	movl (%edx, %ecx, 8), %eax

Array access examples

val* is an array at address *x

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



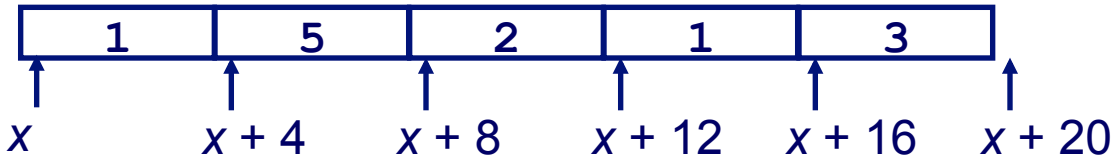
Reference	Type	Value
`val[4]`		
`val`		
`val+3`		
`&val[2]`		
`val[5]`		
`*(val+1)`		
`val + i`		

Array access examples

animation

val* is an array at address *x

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



Reference	Type	Value
`val[4]`	`int`	`3`
`val`	`int *`	`x`
`val+3`	`int *`	`x+12`
`&val[2]`	`int *`	`x+8`
`val[5]`	`int`	`?`
`*(val+1)`	`int`	`5`
`val + i`	`int *`	`x+4i`

Practice problem

Suppose the address of short integer array S and integer index i are stored in %edx and %ecx respectively. For each of the following expressions, give its type, a formula for its value, and an assembly code implementation. The result should be stored in %eax if it is a pointer and %ax if it is a short integer

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

Practice problem

Suppose the address of short integer array S and integer index i are stored in %edx and %ecx respectively. For each of the following expressions, give its type, a formula for its value, and an assembly code implementation. The result should be stored in %eax if it is a pointer and %ax if it is a short integer

Expression	Type	Value	Assembly
S+1	short *	$Addr_S + 2$	leal 2(%edx),%eax
S[3]	short	$M[Addr_S + 6]$	movw 6(%edx),%ax
&S[i]	short *	$Addr_S + 2*i$	leal (%edx,%ecx,2),%eax
S[4*i+1]	short	$M[Addr_S + 8*i + 2]$	movw 2(%edx,%ecx,8),%ax
S+i-5	short *	$Addr_S + 2*i - 10$	leal -10(%edx,%ecx,2),%eax

Arrays in Assembly

Arrays typically have very regular access patterns

- Optimizing compilers are *very good* at optimizing array indexing code
- As a result, output may not look at all like the input

Example

Decimal conversion code

- Takes 5 integers between 0-9 and produces their base-10 value

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

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

    return val;
}
```

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

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

    return val;
}
```

Subscript version

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

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

   return val;
}
```

```
        .file   "decimal5.c"
        .text
        .p2align 2,,3
        .globl  decimal5
        .type   decimal5,@function
decimal5:
        pushl   %ebp
        movl    %esp, %ebp
        xorl    %eax, %eax          ; eax = val = 0
        pushl   %ebx                ; save ebx
        xorl    %ecx, %ecx          ; ecx = i = 0
        movl    8(%ebp), %ebx       ; ebx = x = array base ptr.

.L6:
        leal    (%eax,%eax,4), %edx     ; edx = 5*val
        movl    (%ebx,%ecx,4), %eax     ; eax = x[i]
        leal    (%eax,%edx,2), %eax     ; eax += 10*val
        incl    %ecx                ; i++
        cmpl    $4, %ecx            ; check loop cond. il<5
        jle     .L6
        popl    %ebx                ; restore ebx
        leave
        ret
.Lfe1:
        .size   decimal5,.Lfe1-decimal5
        .ident  "GCC: (GNU) 3.2.2 20030222 (Red Hat Linux 3.2.2-5)"
```

Pointer version

- Same functionality
- Similar complexity
- Preview to 2D loop

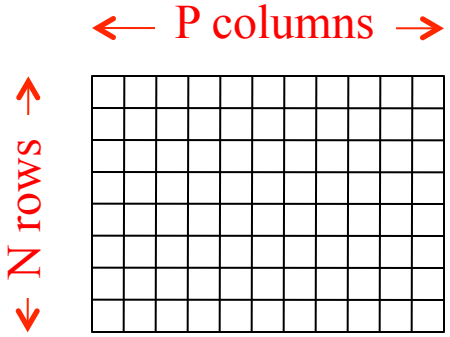
```
int decimal5_opt (int *x) {
    int val = 0;
    int *xend = x + 4;

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

    return val;
}
```

```
        .file   "decimal5_opt.c"
        .text
        .p2align 2,,3
        .globl  decimal5_opt
        .type   decimal5_opt,@function
decimal5_opt:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %ecx           ; ecx = x
        pushl   %ebx                    ; save ebx
        xorl    %eax, %eax              ; eax = val=0
        leal    16(%ecx), %ebx          ; ebx = xend=x+4
        .p2align 2,,3
.L2:
        leal    (%eax,%eax,4), %edx     ; edx = 5*val
        movl    (%ecx), %eax            ; eax = x[i]
        leal    (%eax,%edx,2), %eax     ; val += 10*val+x[i]
        addl    $4, %ecx                ; x++
        cmpl    %ebx, %ecx              ; (x <= xend)?
        jbe     .L2
        popl    %ebx                    ; restore ebx
        leave
        ret
.Lfe1:
        .size   decimal5_opt,.Lfe1-decimal5_opt
        .ident  "GCC: (GNU) 3.2.2 20030222 (Red Hat Linux 3.2.2-5)"
```

Multi-Dimensional Arrays



C allows for multi-dimensional arrays

```
int myArr[N][P];
```

- myArr is an N × P matrix
- N rows, P elements per row
- The dimensions of an array must be declared constants
  - i.e. N and P, must be #define constants
  - Compiler must be able to generate proper indexing code
- Can also have higher dimensions: my3dArr [N] [P] [Q]

Multidimensional arrays in C are stored in "row major" order

- Data grouped by rows
  - All elements of a given row are stored contiguously
  - A[0][*] = in contiguous memory followed by A[1][*]
  - The last dimension is the one that varies the fastest with linear access through memory
- Important to know for performance!

Multi-Dimensional Array Access

Consider array A

```
T A[R][C];
```

R = # of rows, C = # of columns, T = type (which has size K)

What is the size of a row in A?

C * K

What is the address of A[2][5]?

```
A + 2*C*K + 5*K
```

What is the address of `A[i][j]` given in A, C, K, i, and j?

```
A + (i*C*K) + (j*K)
```

A[0][0]	A[0][1]	…	A[0][C-2]	A[0][C-1]
A[1][0]	A[1][1]	…	A[1][C-2]	A[1][C-1]
|	|		|	|
A[R-2][0]	A[R-2][1]	…	A[R-2][C-2]	A[R-2][C-1]
A[R-1][0]	A[R-1][1]	…	A[R-1][C-2]	A[R-1][C-1]

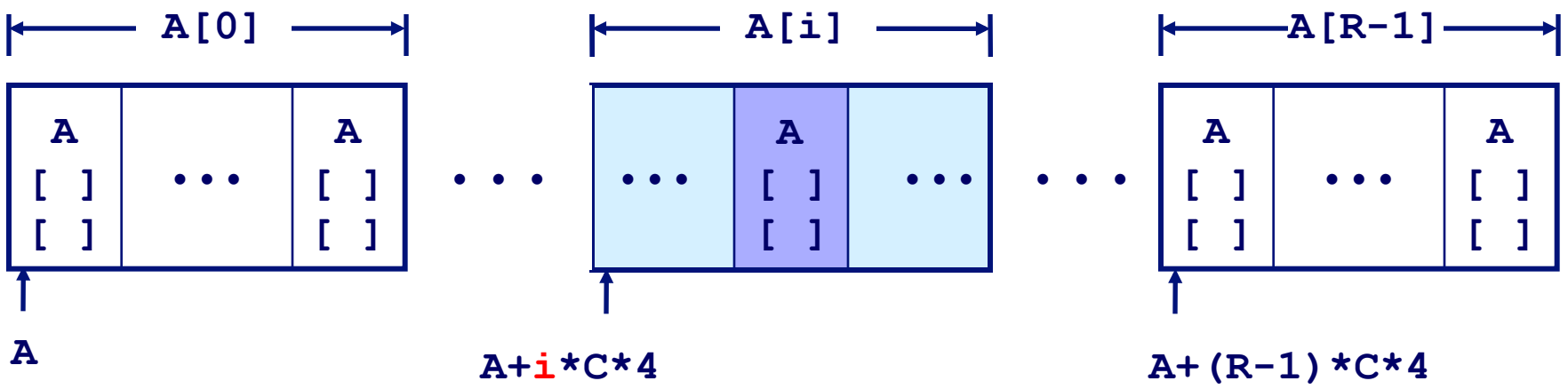
Multi-Dimensional Array Access

Example: Where is A [i] [j] stored?

```
A
[i]
[j]
```

```
int A[R][C];
```

Size of each row: C*4



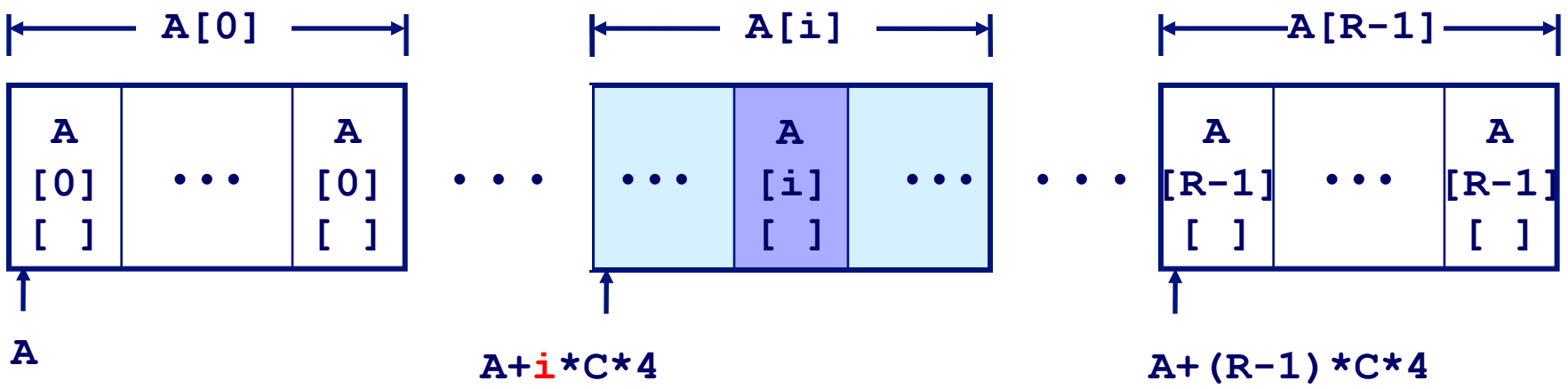
Multi-Dimensional Array Access

Example: Where is A [i] [j] stored?

```
A
[i]
[j]
```

```
int A[R][C];
```

Size of each row: C*4



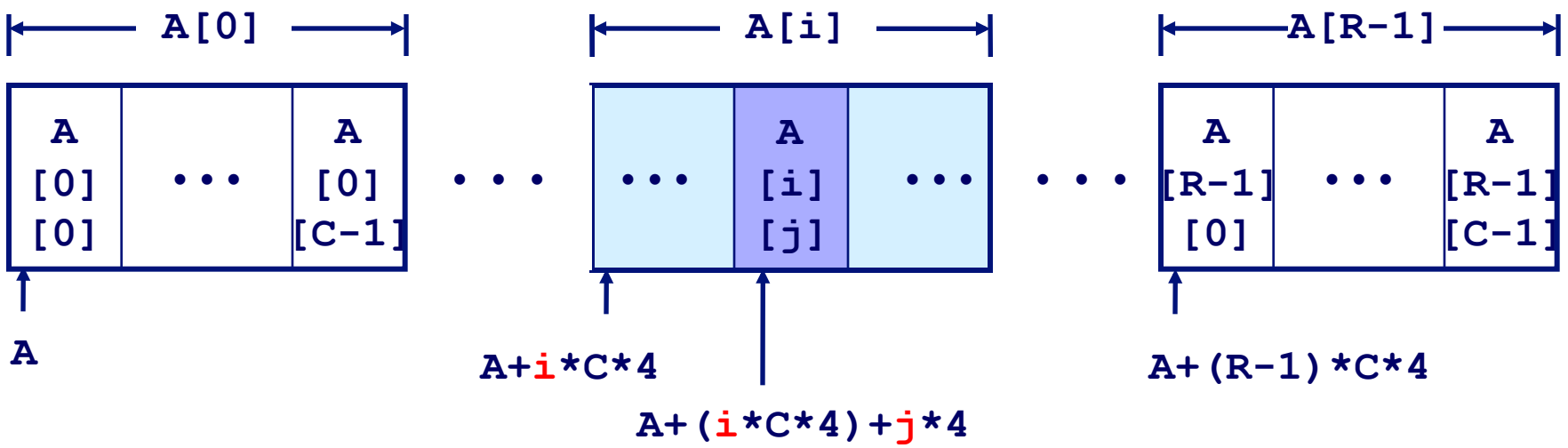
Multi-Dimensional Array Access

Example: Where is A [i] [j] stored?

```
A
[i]
[j]
```

```
int A[R][C];
```

Size of each row: C*4



Assume: integer array A with address in %eax, i in %edx, j in %ecx. If code below moves A[i][j] into %eax, how many columns are in A?

```
1    sall $2, %ecx
2    leal (%edx, %edx, 2), %edx
3    leal (%ecx, %edx, 4), %edx
4    movl (%eax, %edx), %eax
```

int A[R][3]:

A [0] [0]	x_A + 0
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

.
.
.

Assume: integer array A with address in %eax, i in %edx, j in %ecx. If code below moves A[i][j] into %eax, how many columns are in A?

```
1   sall $2, %ecx                    ; 4j
2   leal (%edx, %edx, 2), %edx       ; 3i
3   leal (%ecx, %edx, 4), %edx       ; 12i + 4j
4   movl (%eax, %edx), %eax          ; A[i][j]
```

A+(*i* **C** *K*)+ *j* **K*

A+(*i* **C** 4)+ *j* *4

A+(*i* *3* 4)+ *j* *4

Practice problem

Assume M and N are #define constants. Given the following, what are their values?

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

```
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
```

Practice problem

Assume M and N are #define constants. Given the following, what are their values?

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

```
movl    8(%ebp),%ecx                 ecx=i
movl    12(%ebp),%eax                eax=j
leal    0(,%eax,4),%ebx              ebx=4j
leal    0(,%ecx,8),%edx              edx=8i
subl    %ecx,%edx                    edx=7i
addl    %ebx,%eax                    eax=5j
sall    $2, %eax                     eax=20j
movl    mat2(%eax,%ecx,4),%eax       eax=M[mat2+4i+20j] … 4*(i+5j)
addl    mat1(%ebx,%edx,4),%eax       eax+=M[mat1+28i+4j] … 4*(7i+j)
```

mat2 has 5 columns : M=5

mat1 has 7 columns : N=7

Watch Out For Indexing Errors

```
int A[12][13];  // A has 12 rows of 13 ints each
```

Will the C compiler permit us to do this?

```
int x = A[3][26];
```

What will happen?

Indexing calculation is done assuming a 12x13 array

```
    A + (C*i  + j)  * K
= A + (13*3 + 26) * 4
= A + (13*5 + 0)  * 4
```

Same as A[5][0]

What about this?

```
int x = A[14][2];
```

C does not check array bounds!

(Contrast this to managed languages)

Improving Code Efficiency

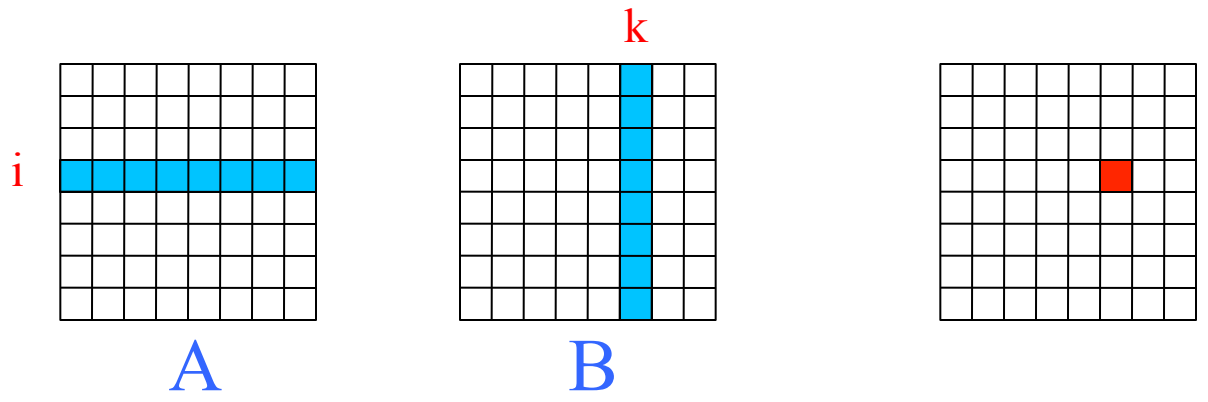
Fixed sized arrays are easy for the compiler to optimize

- But resulting assembly code can be difficult to understand.

Can we write code that compiles more efficiently?

Example: Compute the dot-product

```
#define N 16
typedef int fix_matrix[N][N]
fix_matrix A, B;
```



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

```
int fix_prod_ele (fix_matrix A, fix_matrix 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 fix_prod_ele_opt(fix_matrix A,
      fix_matrix B, int i, int k)
{
   int *Aptr = &A[i][0];
   int *Bptr = &B[0][k];
   int cnt = N - 1;
   int result = 0;

   do {
      result += (*Aptr) * (*Bptr);
      Aptr += 1;
      Bptr += N;
      cnt--;
   } while (cnt >= 0);

   return result;
}
```

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

```
int fix_prod_ele (fix_matrix A, fix_matrix 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 fix_prod_ele_opt(fix_matrix A, fix_matrix B, int i, int k)
{
    int *Aptr = &A[i][0];
    int *Bptr = &B[0][k];
    int cnt = N - 1;
    int result = 0;

    do {
        result += (*Aptr) * (*Bptr);
        Aptr += 1;
        Bptr += N;
        cnt--;
    } while (cnt >= 0);

    return result;
}
```

```
.L2:
        cmpl    $15, -12(%ebp)          ; j <= 15?
        jle     .L5
        jmp     .L3
.L5:
        movl    16(%ebp), %eax          ; eax = i
        sall    $4, %eax                ; eax = 16i
        addl    -12(%ebp), %eax         ; eax = 16i+j
        leal    0(,%eax,4), %ebx        ; ebx = 4*(16i+j)
        movl    8(%ebp), %esi           ; esi = A
        movl    -12(%ebp), %eax         ; eax = j
        sall    $4, %eax                ; eax = 16j
        addl    20(%ebp), %eax          ; eax = 16j+k
        leal    0(,%eax,4), %ecx        ; ecx = 4*(16j+k)
        movl    12(%ebp), %edx          ; edx = B
        movl    (%esi,%ebx), %eax       ; eax = A[i][j]
        imull   (%edx,%ecx), %eax       ; eax *= B[j][k]
        movl    %eax, %edx          ; edx = A[i][j]*B[j][k]
        leal    -16(%ebp), %eax         ; eax = result
        addl    %edx, (%eax)            ; result += edx
        leal    -12(%ebp), %eax         ; eax = j
        incl    (%eax)                  ; j++
        jmp     .L2
.L3:
```

```
.L23:
        movl    (%edx),%eax         ; Compute t = *Aptr
        imull   (%ecx),%eax         ; Compute v = *Bptr * t
        addl    %eax,%esi           ; Add v result
        addl    $64,%ecx            ; Add 64 to Bptr
        addl    $4,%edx             ; Add 4 to Aptr
        decl    %ebx                ; Decrement cnt
        jns     .L23                ; if >=, goto .L23
```

Practice problem

```
void fix_set_diag(fix_matrix A, int val) {
    int i;
    for (i=0; i<N; i++)
        A[i][i] = val;
}
```

```
    movl    12(%ebp),%edx
    movl    8(%ebp), %eax
    movl    $15,%ecx
    addl    $1020,%eax
    .p2align 4,,7    /* Optimize cache */
.L50:
    movl    %edx,(%eax)
    addl    $-68,%eax
    decl    %ecx
    jns     .L50
```

Create a C code program fix_set_diag_opt that uses optimizations similar to those in the assembly code, in the same style as the previous slide

```
void fix_set_diag_opt(fix_matrix A, int val) {
    int *Aptr = &A[0][0]+255;      /* End of matrix 255*4 = 1020 (N=16) */
    int cnt = N-1;
    do {
        *Aptr = val;           /* Set current location to val */
        Aptr -= (N+1);         /* Go up a row and back one entry */
        cnt--;            /* Decrement counter */
    } while (cnt >= 0);        /* Repeat until at top of matrix */
}
```

Dynamically Allocated Arrays

What if we don't know the dimensions for our array at compile-time?

The compiler cannot generate multi-dimensional indexing code unless dimensions are known at compile-time

Solution: Perform indexing / address calculations in C code

```
typedef int *VarMatrix;
```

VarMatrix is a pointer to an int

Can also be a pointer to a matrix of ints!

How to allocate an one of these, of dimension n x n:

```
VarMatrix newVarMatrix(int n) {
  return (VarMatrix) calloc(sizeof(int), n*n);
}
VarMatrix m = newVarMatrix(n);
```

Accessing Dynamic Arrays

Must do the indexing explicitly

Write the C code for a function that returns A[i][j]

```
int getElt(VarMatrix A, int i, int j, int n)
  { ... }

x = getElt (myMat, 4, 33, n);
```

A points to an n x n matrix of integers.

The dimension n is an argument.

We want the value of A[i][j].

Memory [A+4*(n*i + j)]