Arrays

Recall arrays

`char foo[80];`

- An array of 80 characters

`int bar[40];`

- An array of 40 integers

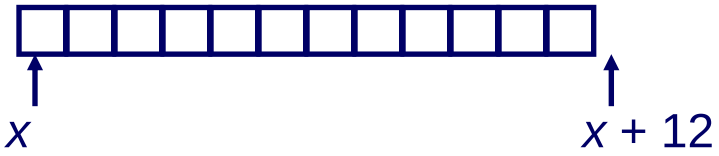
Array allocation

Basic Principle

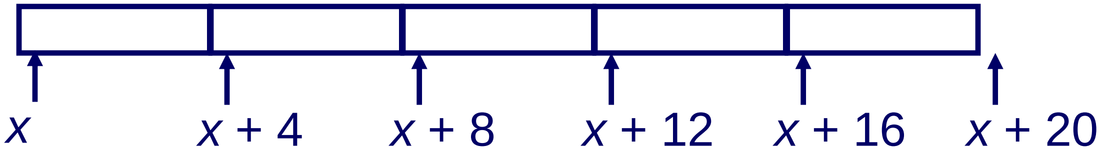
`T A[L];`

- A is an array of data type *T* and length *L*
- *Contiguously allocated* region of *L* * `sizeof(`*T*`)` bytes

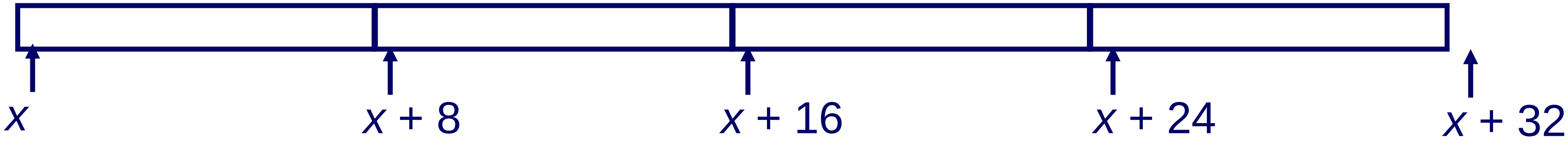
`char string[12];`



`int val[5];`



```
long a[4];
 char *p[4];
double a[4];
```



Pointers and arrays closely related

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

```
long a[5];    /* a is array of 5 long       */
long *lptr;   /* lptr is a pointer to long  */
lptr = a;     /* set lptr to point to a     */
```

Pointers and arrays closely related

Two ways to access array elements

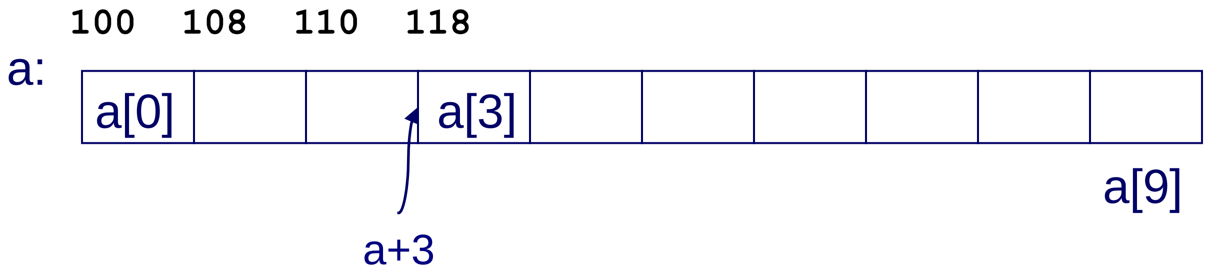
- Via array indexing

```
i = a[3];     /* set i to 3rd element of array */
```

- Via pointer arithmetic followed by dereferencing
  - Recall pointer arithmetic done based upon type of pointer!

```
i = *(a+3); /* set pointer to 3rd element of array */
            /* then, dereference pointer          */
```

  - If a is at `0x100`, what is the value of `a+3`?



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**

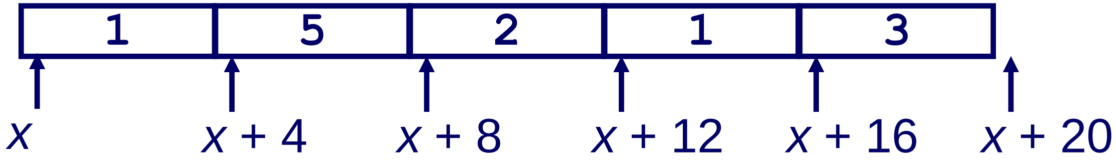
```
char      A[12];
char     *B[8];
long      C[6];
float     D[5]
```

Array	Element Size	Total Size	Start Address	Element i
A	**1**	**12**	x_A	$x_A + i$
B	**8**	**64**	x_B	$x_B + 8i$
C	**8**	**48**	x_C	$x_C + 8i$
D	**4**	**20**	x_D	$x_D + 4i$

Array access examples

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 3.35

```
short      S[7];
short     *T[3]
int       V[8];
```

Array	Element Size	Total Size	Start Address	Element *i*
S	`2`	`14`	x_S	`x+2i`
T	`8`	`24`	x_T	`x+8i`
U	`4`	`32`	x_U	`x+4i`

Arrays as function arguments

The basic data types in C are passed by value.

What about arrays?

Example:

```
long exp[32000000];
long x = foo(exp);
```

What must the function declaration of foo be?

```
long foo(long* 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 (e.g. argv)

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

Practice problem

Consider the following code

```
char *pLines[3];
char *a="abc";
char *b="bcd";
char *c="cde";

pLines[0]=a;
pLines[1]=b;
pLines[2]=c;
```

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’`**

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

Array access examples

Pointer arithmetic

int E[20];

%rax == result

%rdx == start address of E

%rcx == index 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]+i)	int	M[$X_E + 4i + 4i$]	movl (%rdx, %rcx, 8), %eax

Practice problem 3.36

Suppose the address of short integer array S and integer index i are stored in %rdx and %rcx 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 %rax if it is a pointer and %ax if it is a short integer

Expression	Type	Value	Assembly
S+1	short *	$\text{Addr}_S + 2$	leaq 2(%rdx),%rax
S[3]	short	$M[\text{Addr}_S + 6]$	movw 6(%rdx),%ax
&S[i]	short *	$\text{Addr}_S + 2*i$	leaq(%rdx,%rcx,2),%rax
S[4*i+1]	short	$M[\text{Addr}_S + 8*i + 2]$	movw 2(%rdx,%rcx,8),%ax
S+i-5	short *	$\text{Addr}_S + 2*i - 10$	leaq -10(%rdx,%rcx,2),%rax

Multi-Dimensional Arrays

C allows for multi-dimensional arrays

`int x[N][P];`

- x is an N x 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: x[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 of 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?

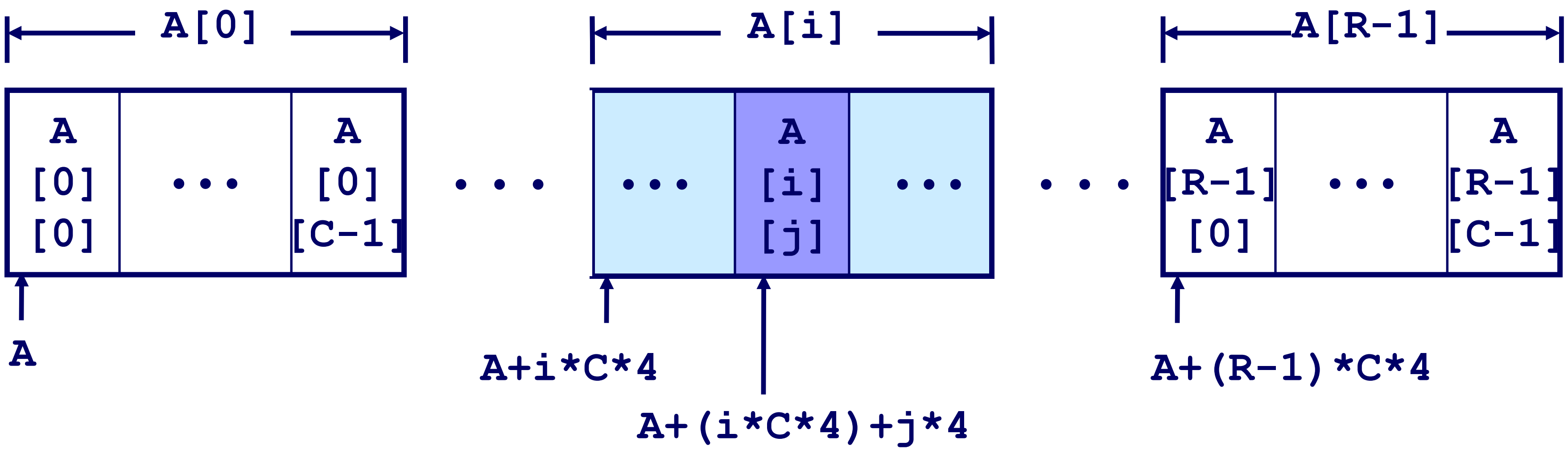
`A+`$(i*C*K)+j*K$

A[0][0]	A[1][0]	…	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

`int A[R][C];`



Example

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 %rax, i in %rdx, j in %rcx. If code below moves A[i][j] into %eax, how many columns are in A?

```
salq $2, %rcx                 ; 4j
leaq (%rdx, %rdx, 2), %rdx    ; 3i
leaq (%rcx, %rdx, 4), %rdx    ; 12i + 4j
movl (%rax, %rdx), %eax       ; A[i][j]
```

$\mathbf{A} + (i * C * K) + j * K$

$\mathbf{A} + (i * C * 4) + j * 4$

$\mathbf{A} + (i * 3 * 4) + j * 4$

Practice problem 3.37

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

```
long P[M][N];
long Q[N][M];
long sum_element(long i, long j){
    return (P[i][j] + Q[j][i]);
}
```

```
    /* i in %rdi, j in %rsi */
sum_element:
    leaq    0(,%rdi,8), %rdx        ; rdx = 8i
    subq    %rdi, %rdx              ; rdx = 7i
    addq    %rsi, %rdx              ; rdx = 7i + j
    leaq    (%rsi, %rsi, 4), %rax   ; rax = 5j
    addq    %rax, %rdi              ; rdi = 5j + i
    movq    Q(,%rdi,8), %rax        ; rax = M[Q+8*(5j+i)]
    addq    P(,%rdx,8), %rax        ; rax += M[P+8*(7i+j)]
```

```
Columns in Q => M=5
Columns in P => N=7
```

Something to watch out for

```
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 done assuming a 12x13 array

```
A + (i*C + j) * K    = A + (13*i + 26) * 4
                     = A + (13*(i+2) + 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

Array optimizations

Fixed sized arrays are easy for the compiler to optimize

- Results can be complex to understand

Example

- Dot-product of matrices

```
#define N 16
typedef long fix_matrix[N][N]
fix_matrix A;
```

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

```
long fix_prod_ele
(fix_matrix A, fix_matrix B,  long i, long k)
{
    long j;
    long result = 0;
    for (j = 0; j < N; j++)
        result += A[i][j] * B[j][k];
    return result;
}
```

```
/* rdi=A    rsi=B    rdx=i    rcx=k                */
/* r8=> j    rax=>result                           */
    mov     $0x0,%eax          ; rax = result = 0
    mov     $0x0,%r8d          ; r8 = j = 0
    shl     $0x7,%rdx          ; rdx = 128*i
    add     %rdx,%rdi          ; rdi = A+128*i
    jmp     .L2
.L1:
    mov     %r8,%r9            ; tmp = j
    shl     $0x7,%r9           ; tmp = 128*j
    add     %rsi,%r9           ; tmp = B+128*j
    mov     (%r9,%rcx,8),%r9   ; tmp = M[8*k+B+128*j]
    imul    (%rdi,%r8,8),%r9   ; tmp *= M[8*j+A+128*i]
    add     %r9,%rax           ; result += tmp
    add     $0x1,%r8           ; j++
.L2:
    cmp     $0xf,%r8           ; j == 15?
    jle     .L1
    retq
```

```
long fix_prod_ele_opt
(fix_matrix A, fix_matrix B, long i, long k)
{
    long *Aptr = &A[i][0];
    long *Bptr = &B[0][k];
    long cnt = N - 1;
    long result = 0;
    do {
        result += (*Aptr) * (*Bptr);
        Aptr += 1;
        Bptr += N;
        cnt--;
    } while (cnt >= 0);
    return result;
}
```

```
/* rdi=A    rsi=B    rdx=i    rcx=k                */
/* rcx=> cnt    rax=>result                        */
   shl     $0x7,%rdx              ; rdx = 128*i
   add     %rdx,%rdi              ; rdi = Aptr = A+128*i
   lea     (%rsi,%rcx,8),%rsi     ; rsi = Bptr = B+8*k
   mov     $0x0,%eax              ; rax = result = 0
   mov     $0xf,%ecx              ; rcx = cnt = 15
.L1:
   mov     (%rsi),%rdx            ; tmp = M[Bptr]
   imul    (%rdi),%rdx            ; tmp *= M[Aptr]
   add     %rdx,%rax              ; result += tmp
   add     $0x8,%rdi              ; Add 8 to Aptr
   add     $0x128,%rsi            ; Add 128 to Bptr
   sub     $0x1,%rcx              ; cnt--
   jns     .L1
   retq
```

Practice problem 3.38

```
#define N 16
typedef long fix_matrix[N][N];
void fix_set_diag(fix_matrix A, long val) {
    long i;
    for (i=0; i<N; i++)
        A[i][i] = val;
}
```

```
    mov     $0x0,%eax
.L1:
    mov     %rsi,(%rdi,%rax,8)
    add     $17,%rax
    cmp     $110,%rax
    jne     .L1
    retq
```

Note: Book uses int matrix, we use long

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, long val) {
    long *Aptr = &A[0][0];     /* Use Aptr to index into matrix */
    long i = 0;                /* Offset into Aptr for next element to set */
    long iend = N*(N+1);       /* Stopping condition */
    do {
        Aptr[i] = val;         /* Index into memory i long ints */
        i += (N+1);            /* Go down a row and forward one column */
    } while (i != iend);       /* Repeat until at top of matrix */
}
```

Dynamically Allocated Arrays

What if we don’t know *any* of the dimensions for our array?

- C array logic doesn’t handle this really well
- Cannot generate multi-dimensional indexing code unless dimensions are known at compile-time
- Must handle pointer/addresses/indices 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 Amatrix = newVarMatrix(n);
```

Accessing Dynamic Arrays

Must do the indexing explicitly

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

```
int varEle(varMatrix A, int i, int j, int 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].

- M[A+4*(n*i + j)]