# 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.

Example: The return value of malloc must be cast

# Every pointer has a value

The address of an object (of a particular type)

All pointers are 8 bytes for x86-64

8 bytes = 64 bits

# The “address-of” operator: &

Can use & on all “lvalues”

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

# The “dereference” 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  a[3] == *(a+3)

# Pointers and arrays review

**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**
  - **= the address of the 0$^{th}$ 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

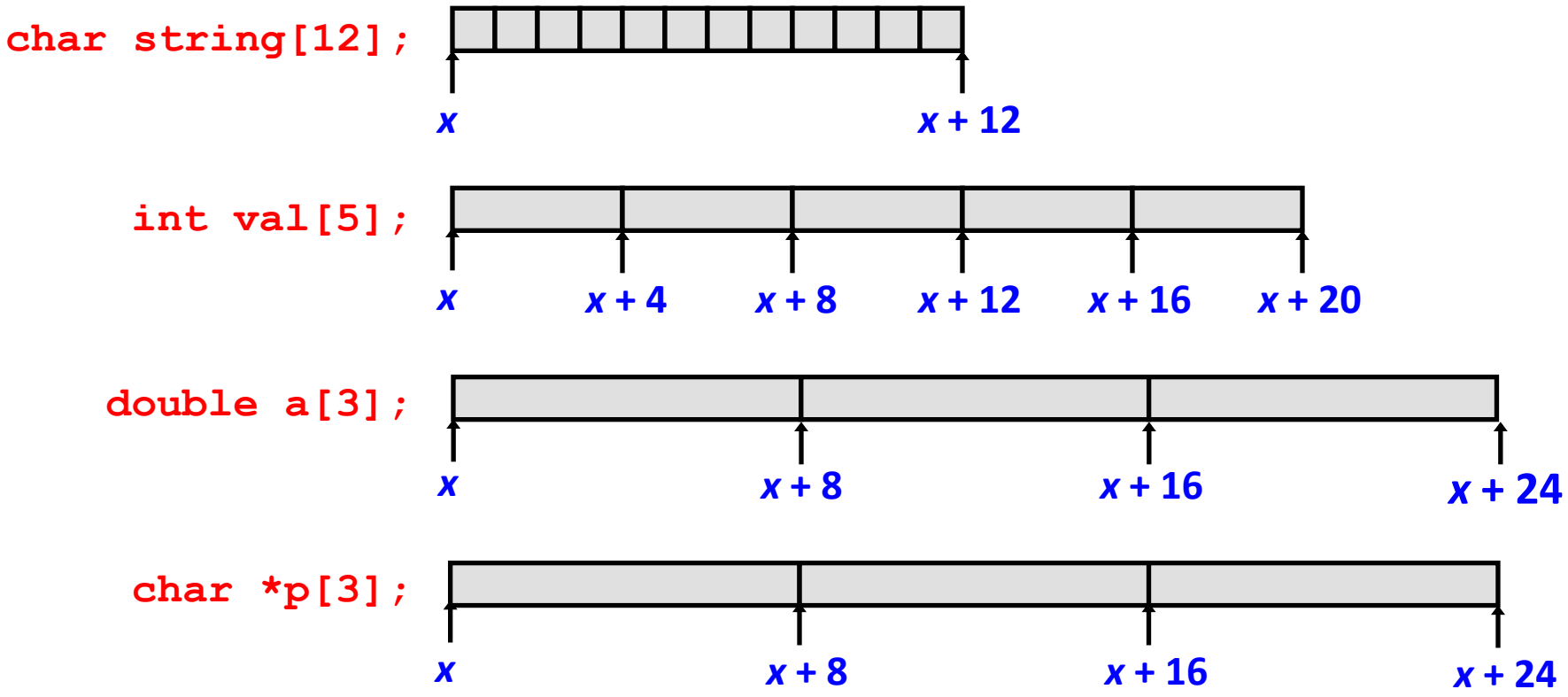
**Recall: pointer arithmetic is based on type of pointer**

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

# Array allocation

*Type* **ArrayName** [*Length*];

**Contiguously allocated region of *Length* × sizeof(*Type*) bytes**



# Pointer arithmetic with arrays

## As a result of contiguous allocation

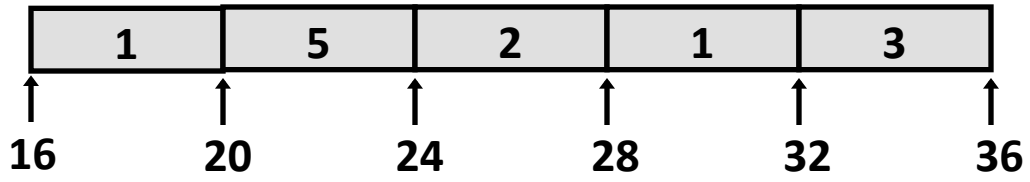
- Elements are accessed by scaling the index by the size of the element and adding to the starting address
- Assembly: scaled index memory mode

```
char    A[12];
int     B[8];
double  C[6];
int     *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** | **8** | **40** | $x_D$ | $x_D + 8i$ |

# Array Accessing Example

```
int myNums[5];
```




```
int get_num (int z[], int num {
  return z[num];
}
```

%rdi %rsi 4

Base + Index × size

**%rdi + %rsi × 4**

**(%rdi,%rsi,4)**

Assembly Code uses scaled, indexed mode

```
# On entry:
#     %rdi = z
#     %rsi = num
movl (%rdi,%rsi,4),%eax
```

# 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 | 8 | 24 | $x_T$ | x+8i |
| U | 8 | 48 | $x_U$ | x+8i |

# Array Loop Example

```
void zincr(int z[5]) {
  size_t i;
  for (i = 0; i < 5; i++)
    z[i]++;
}
```

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

# 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

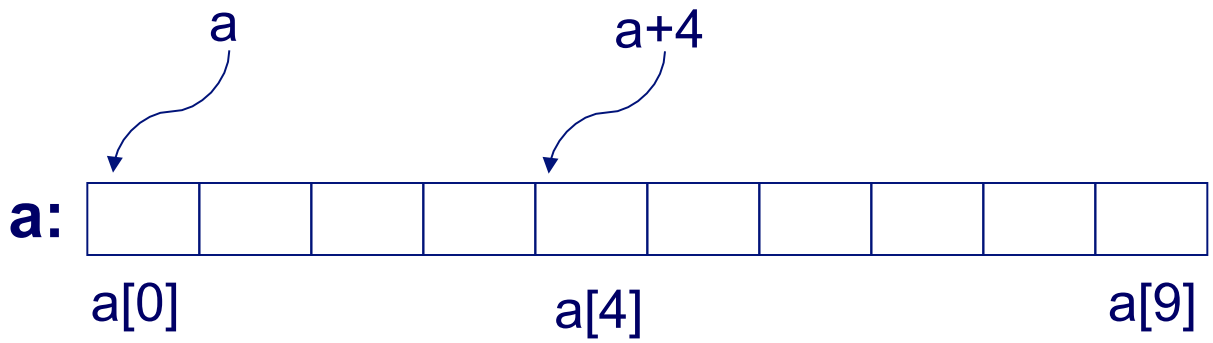
| | | |
|---|---|---|
| `pLines` | `char **` | `pLines` |
| `pLines[0]` | `char *` | `a` |
| `*pLines` | `char *` | `a` |
| `*pLines[0]` | `char` | `'a'` |
| `**pLines` | `char` | `'a'` |
| `pLines[0][0]` | `char` | `'a'` |

# Array access in C

## Arrays can be accessed two ways

- Via index a[4]
- Via pointer arithmetic *(a+4)

**int a[10];**

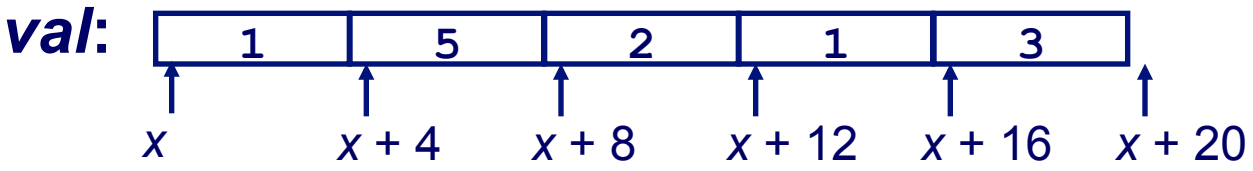



**(a+N) == &a[N]**
***(a+N) == a[N]**

# Array access examples

animation

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



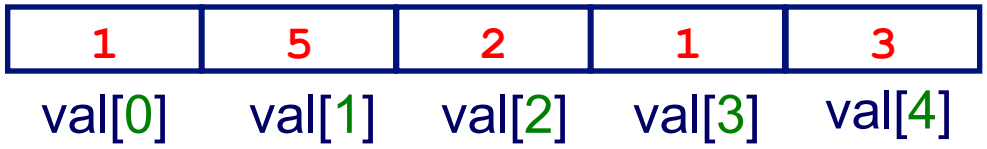

| Reference | Type | Value |
|---|---|---|
| `val[4]` | `int` | `3` |
| `val` | `int *` | `x` |
| `val+3` | `int *` | `x+12` |
| `&val[2]` | `int *` | `x+8` |
| `val[5]` | `int` | `?` |
| `*(val+1)` | `int` | `5` |
| `val + i` | `int *` | `x+4i` |

# Arrays in Assembly

## Arrays typically have very regular access patterns

- Optimizing compilers are *very good* at optimizing array indexing code
- Unfortunately, assembly code may not look at all like the source!

# 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;
}
```

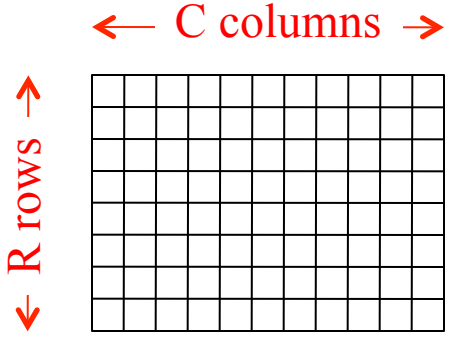
```
decimal5_opt:
        leaq    16(%rdi),%rdx
        movl    $0, %eax
.L45:
        leal    (%rax,%rax,4),%eax
        addl    %eax,%eax
        addl    (%rdi),%eax
        addq    $4,%rdi
        cmpq    %rdx,%rdi
        jbe     .L45
        rep ret
```

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

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

    return val;
}
```

# Multi-Dimensional Arrays




`int myArr[R][C];`

An R × C matrix

R rows, C elements per row

The dimensions of an array must be declared constants

- R and C must be #define constants

Compiler must be able to generate proper indexing code

Can also have higher dimensions: `int my3dArr[R][C][D];`

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

```
Type 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)
```

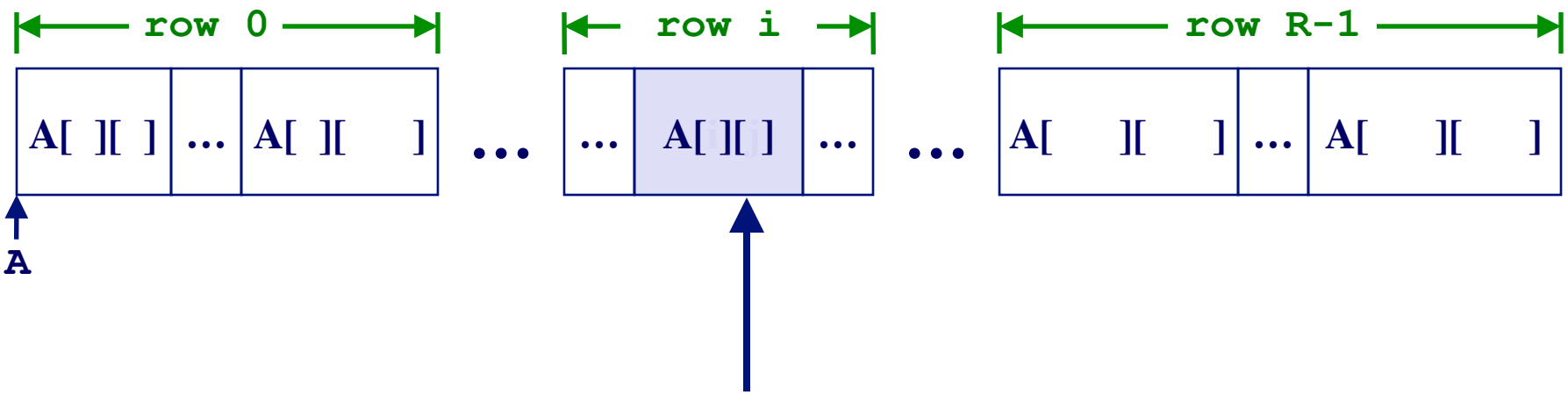
| | | | | |
|---|---|---|---|---|
| A[0][0] | A[0][1] | | | A[0][C-1] |
| A[1][0] | A[1][1] | | | A[1][C-1] |
| | | | | |
| | | | | |
| | | | | |
| A[R-1][0] | A[R-1][1] | | | A[R-1][C-1] |

# Multi-Dimensional Array Access

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

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

**Size of each row: C*4**

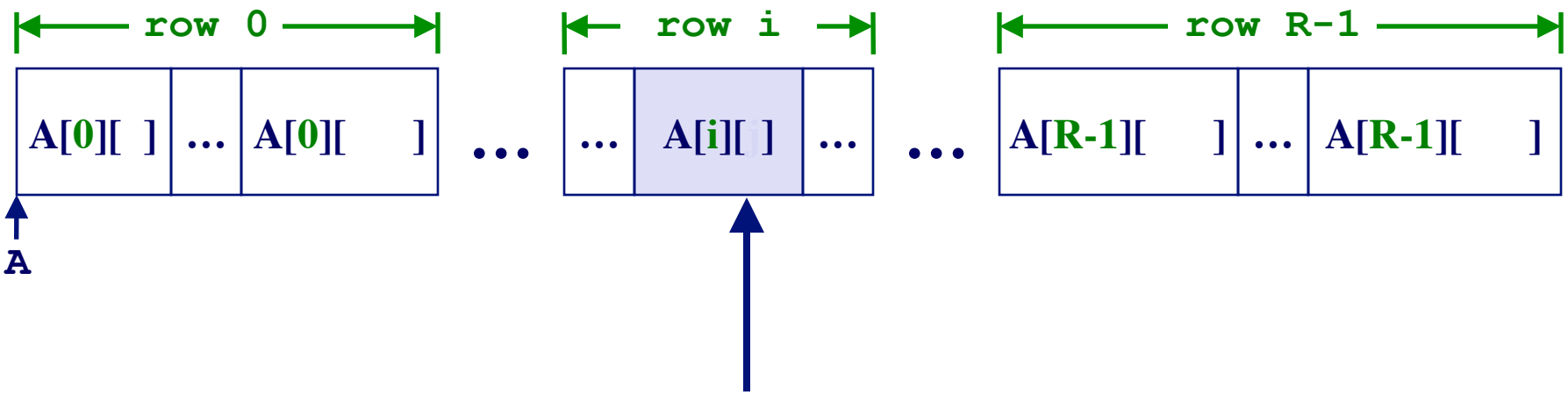


# Multi-Dimensional Array Access

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

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

**Size of each row: C*4**

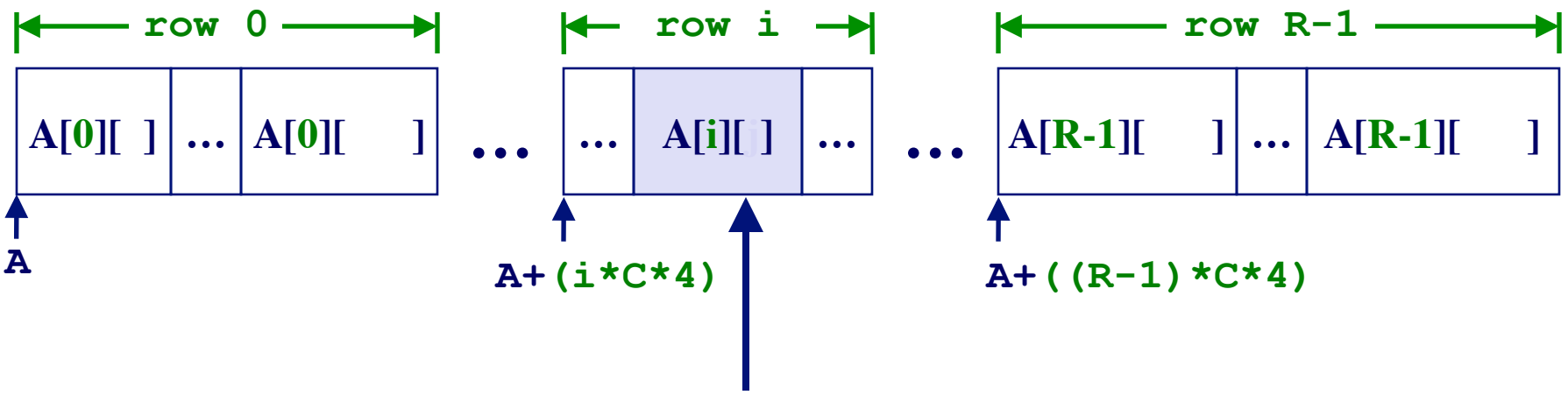


# Multi-Dimensional Array Access

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

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

**Size of each row: C*4**

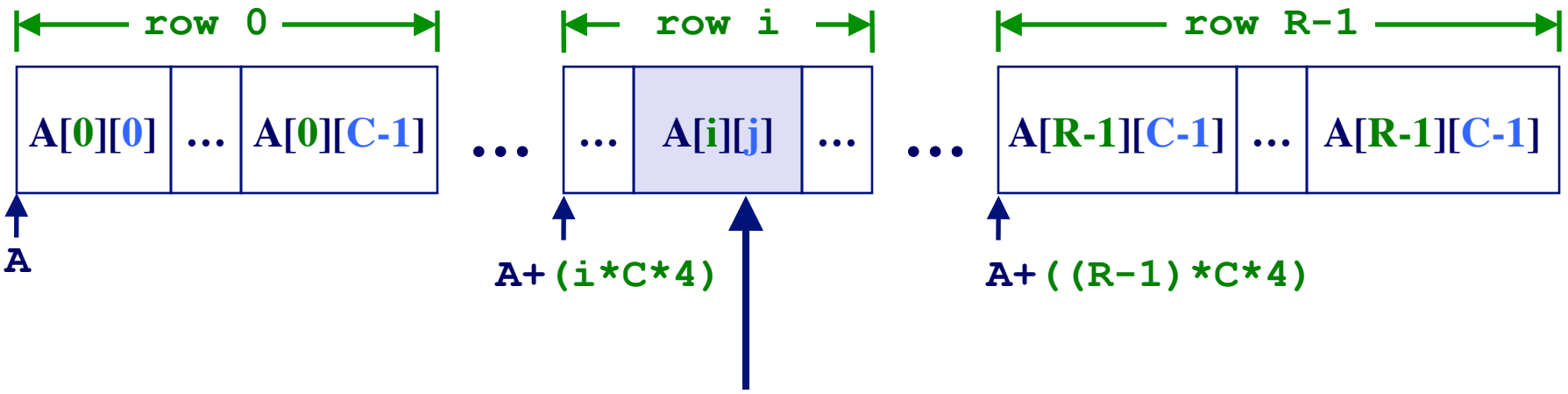


# Multi-Dimensional Array Access

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

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

**Size of each row: C*4**

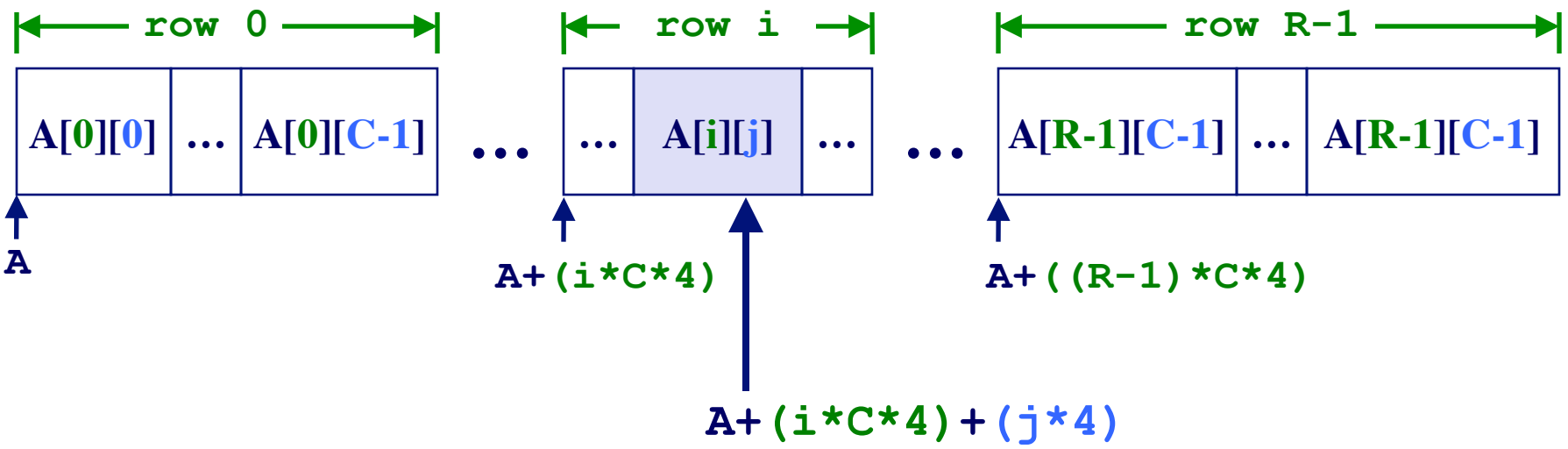


# Multi-Dimensional Array Access

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

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

**Size of each row: C*4**



# 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

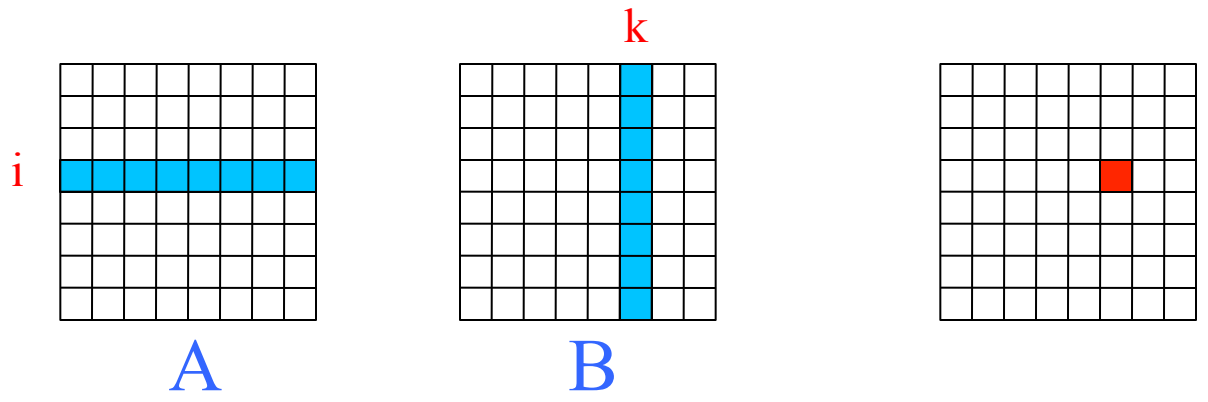
**Fixed sized arrays are easy for the compiler to optimize**

- But resulting assembly code can be difficult to understand.

**Can we write code that compiles more efficiently?**

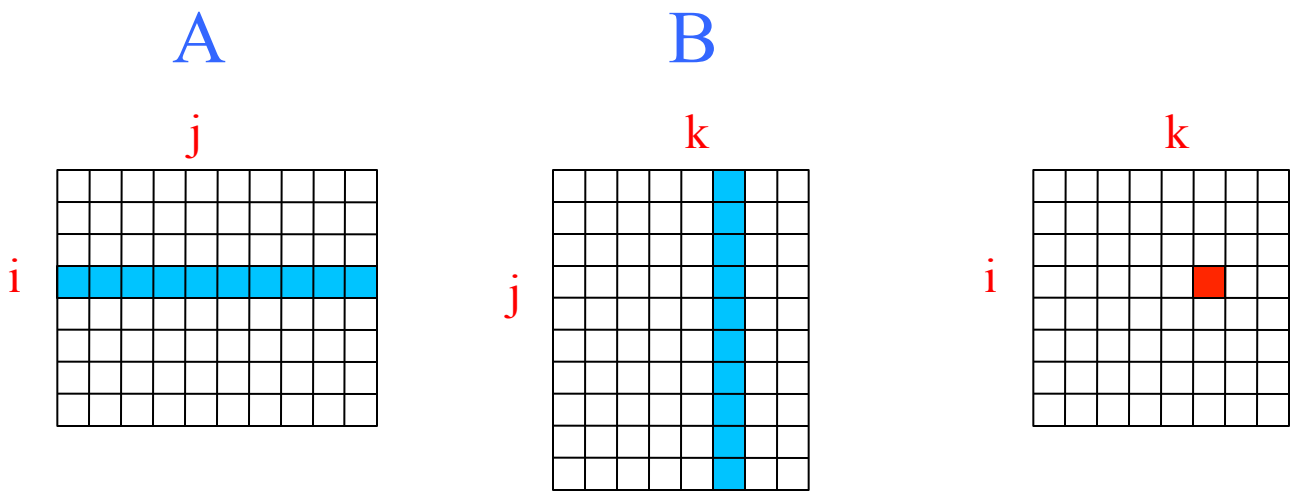
**Example: Compute the dot-product**

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



# Dot Product

```
for i,k…
  C[i,k] = 0;
  for all j…
    C[i,k] += A[i,j] × B[j,k]
```



```
#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;
}
```

```
f.L50:
     movslq   %r8d, %rdx
     movq     %rdx, %r9
     salq     $6, %r9
     addq     %rsi, %r9
     movl     (%r9,%rcx,4), %r9d
     imull    (%rdi,%rdx,4), %r9d
     addl     %r9d, %eax
     addl     $1, %r8d
     cmpl     $15, %r8d
     jle      .L50
```

```
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;
}
```

```
.L52:
     movl     (%rcx), %esi
     imull    (%rdi), %esi
     addl     %esi, %eax
     addq     $4, %rdi
     addq     $64, %rcx
     subl     $1, %edx
     jns      .L52
```

# 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) ]**