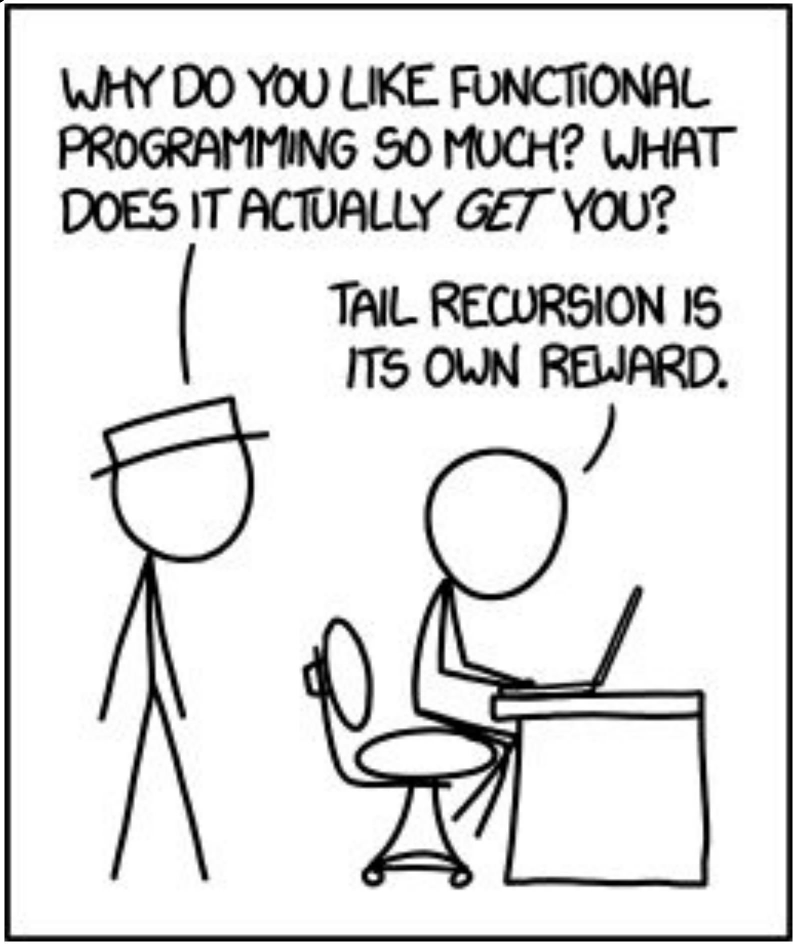
# Executables & Arrays

CSE 351 Summer 2021

**Instructor:**
Mara Kirdani-Ryan

**Teaching Assistants:**
Kashish Aggarwal
Nick Durand
Colton Jobes
Tim Mandzyuk


WHY DO YOU LIKE FUNCTIONAL PROGRAMMING SO MUCH? WHAT DOES IT ACTUALLY *GET* YOU?
TAIL RECURSION IS ITS OWN REWARD.

http://xkcd.com/1270/

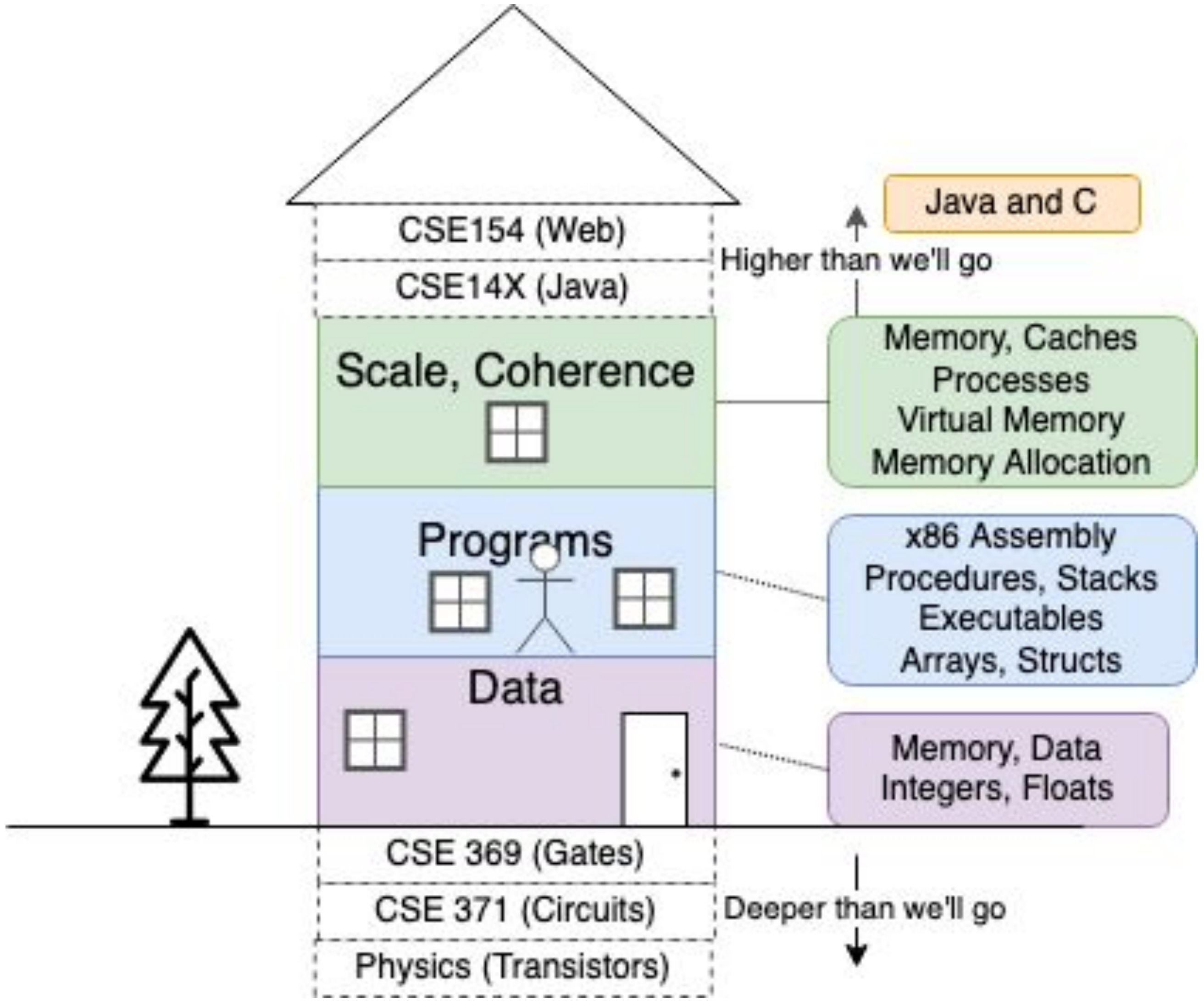
## Gentle, Loving Reminders

- hw11 due tonight! @8pm, unless we've talked
- hw12 due Friday
- Lab 2 due Wednesday (7/21)
  - GDB Tutorial on Gradescope walks through first phase
  - Extra Credit portion – make sure you also submit to the Lab 2 Extra Credit assignment on Gradescope
- Thanks for the feedback!
  - You can always submit more feedback; private ed posts, email us, or just me, anonymous feedback at feedback.cs.washington.edu

## Last week of unit 2!

- Start thinking about how things might fit together!
- We've covered:
  - x86 assembly
  - Neoliberalism and other values in processors
  - Critical Reading
- We'll cover:
  - Executables, Arrays, Structs, Buffer Overflow
  - Accessibility in computing, and computing spaces

## Second Floor



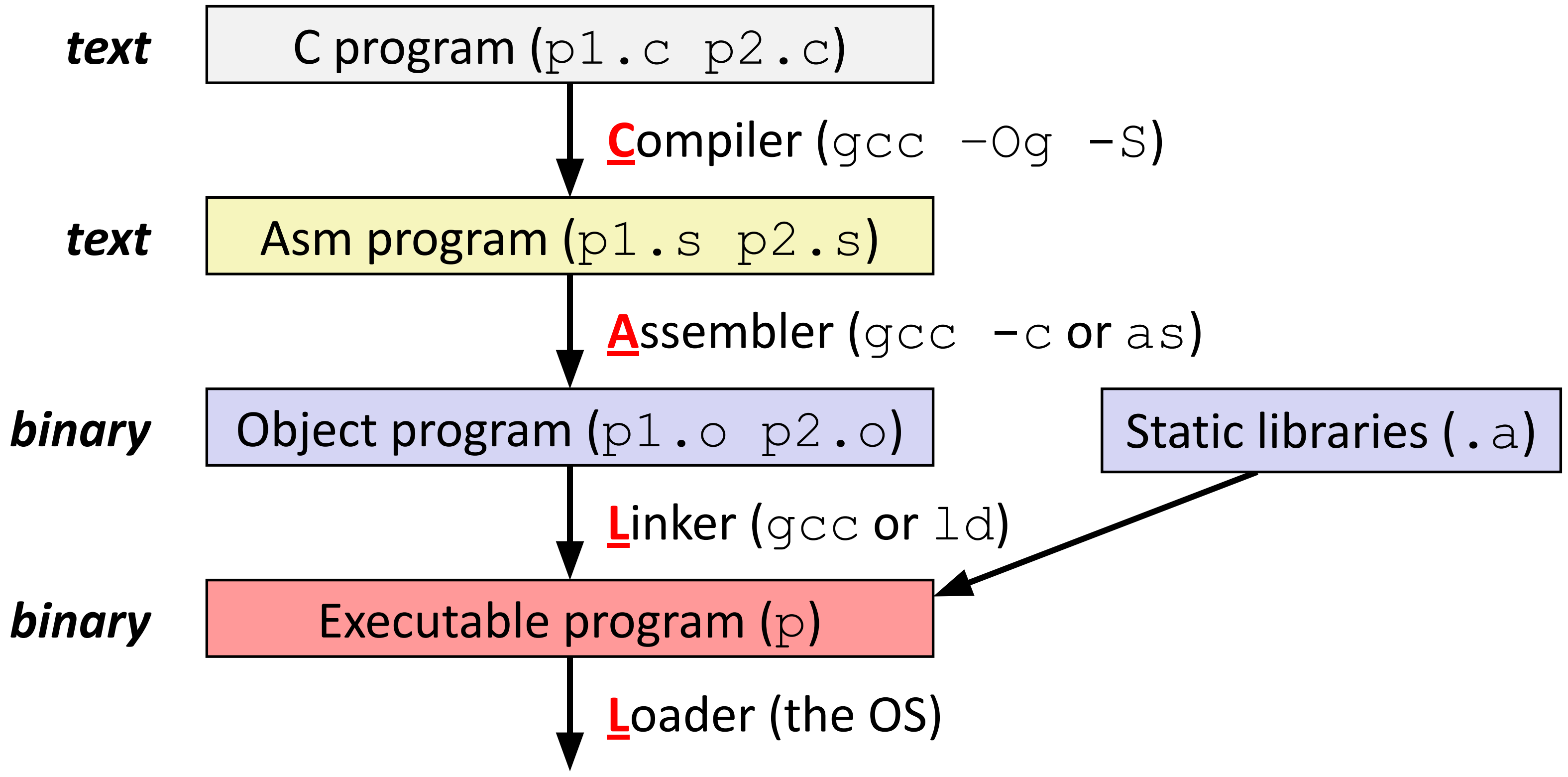
## Learning Objectives

Understanding this lecture means you can:

- Differentiate between executable sections
- Give a overview of how an executable is built from a C file, and explain the roles of the compiler, assembler, linker, and loader
- Show how arrays (1-dim, multi-dim, nested) are represented at the assembly level
- Explain how accessibility in computing relates ideology in computing

# Building an Executable from a C File

- Code in files `p1.c p2.c`
- Compile with command: `gcc -Og p1.c p2.c -o p`
  - Put resulting machine code in file `p`
- Run with command: `./p`



# Compiler

- **Input:** Higher-level language code (*e.g.* C, Java)
  - `foo.c`
- **Output:** Assembly language code (*e.g.* x86, ARM, MIPS)
  - `foo.s`

- First there's a preprocessor step to handle #directives
  - Macro substitution, plus other specialty directives
  - If curious/interested: http://tigcc.ticalc.org/doc/cpp.html
- Super complex, whole courses devoted to these!
- Compiler optimizations
  - "Level" of optimization specified by capital '`O`' flag (*e.g.* `-Og`, `-O3`)
  - Options: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

## Compiling Into Assembly

- C Code (`sum.c`)

```
void sumstore(long x, long y, long *dest) {
    long t = x + y;
    *dest = t;
}
```

- x86-64 assembly (`gcc -Og -S sum.c`)

```
sumstore(long, long, long*):
  addq    %rdi, %rsi
  movq    %rsi, (%rdx)
  ret
```

Warning: You may get different results with other versions of `gcc` and different compiler settings

## Assembler

- **Input:** Assembly language code (*e.g.* x86, ARM, MIPS)
  - `foo.s`
- **Output:** Object files (*e.g.* ELF, COFF)
  - `foo.o`
  - Contains *object code* and *information tables*

- Reads and uses *assembly directives*
  - *e.g.* `.text`, `.data`, `.quad`
  - x86: https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html
- Produces "machine language"
  - Does its best, but object file is *not* a completed binary
- Example: `gcc -c foo.s`

# Producing Machine Language

- **Simple cases:** arithmetic and logical operations, shifts, etc.
  - All necessary information is contained in the instruction itself

- What about the following?
  - Conditional jump
  - Accessing static data (*e.g.* global var or jump table)
  - `call`

- Addresses and labels are problematic because the final executable hasn't been constructed yet!
  - So how do we deal with these in the meantime?

# Object File Information Tables

- **Symbol Table** holds list of "items" that may be used by other files
  - *Non-local labels* – function names for `call`
  - *Static Data* – variables & literals that might be accessed across files

- **Relocation Table** holds list of "items" that this file needs the address of later (currently undetermined)
  - Any *label* or piece of *static data* referenced in an instruction in this file
    - Both internal and external

- Each file has its own symbol and relocation tables

## Object File Format

1) object file header: size and position of the other pieces of the object file
2) text segment: the machine code
3) data segment: data in the source file (binary)
4) relocation table: identifies lines of code that need to be "handled"
5) symbol table: list of this file's labels and data that can be referenced
6) debugging information

- More info: ELF format
  - http://www.skyfree.org/linux/references/ELF_Format.pdf

## Practice!

Where will the following symbols show up in the object file?

- A (non-static) user defined function
- A local variable
- A library function

🐶 text segment
🐈 data segment
🐑 symbol table
🦄 relocation table

🥶 Help!

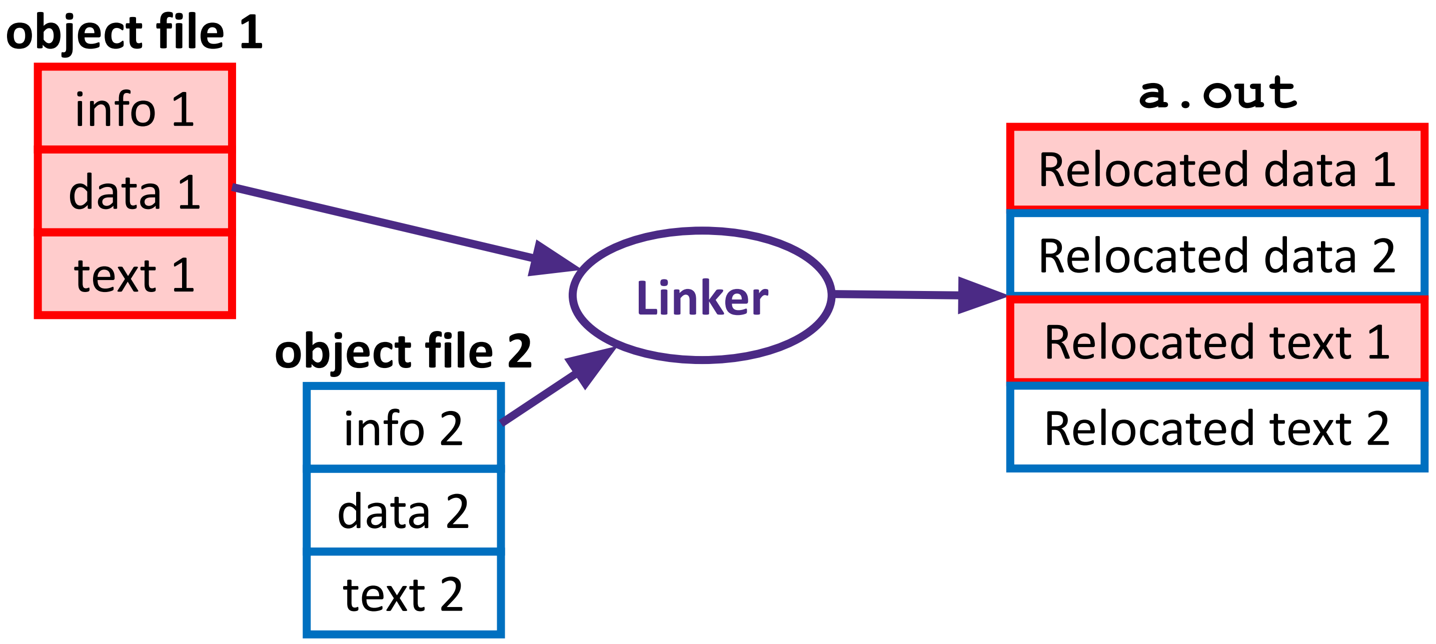
## Linker

- **Input:** Object files (e.g. ELF, COFF)
  - `foo.o`
- **Output:** executable binary program
  - `a.out`

- Combines several object files into a single executable (*linking*)
- Enables separate compilation/assembling of files
  - Changes to one file do not require recompiling of whole program

# Linking

1) Take `text` segment from each `.o` file and put them together
2) Take `data` segment from each `.o` file, put them together, and concatenate onto end of text segments
3) Resolve References
   - Go through Relocation Table; handle each entry



## Disassembling Object Code

- Disassembled:

```
0000000000400536 <sumstore>:
  400536:   48 01 fe        add    %rdi,%rsi
  400539:   48 89 32        mov    %rsi,(%rdx)
  40053c:   c3              retq
```

- **Disassembler** (`objdump -d sum`)
  - Useful tool for examining object code
    - (`man 1 objdump`)
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can run on `a.out` (complete executable) or `.o` file

# What Can be Disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:    file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:
30001001:
30001003:
30001005:
3000100a:
```

**Reverse engineering forbidden by Microsoft End User License Agreement**

- Anything that can be interpreted as executable code
- Disassembler examines bytes and attempts to reconstruct assembly source

## Loader

- **Input:** executable binary program, command-line arguments
  - `./a.out arg1 arg2`
- **Output:** <program is run>

- Loader duties primarily handled by OS/kernel
  - More about this when we learn about processes
- Memory sections (Instructions, Static Data, Stack) set up
- Registers are initialized

## Feelings check: Executables!

## Data Structures in Assembly

- **Arrays**
  - **One-dimensional**
  - Multidimensional (nested)
  - Multilevel
- Structs
  - Alignment
- ~~Unions~~

## Review: Array Allocation

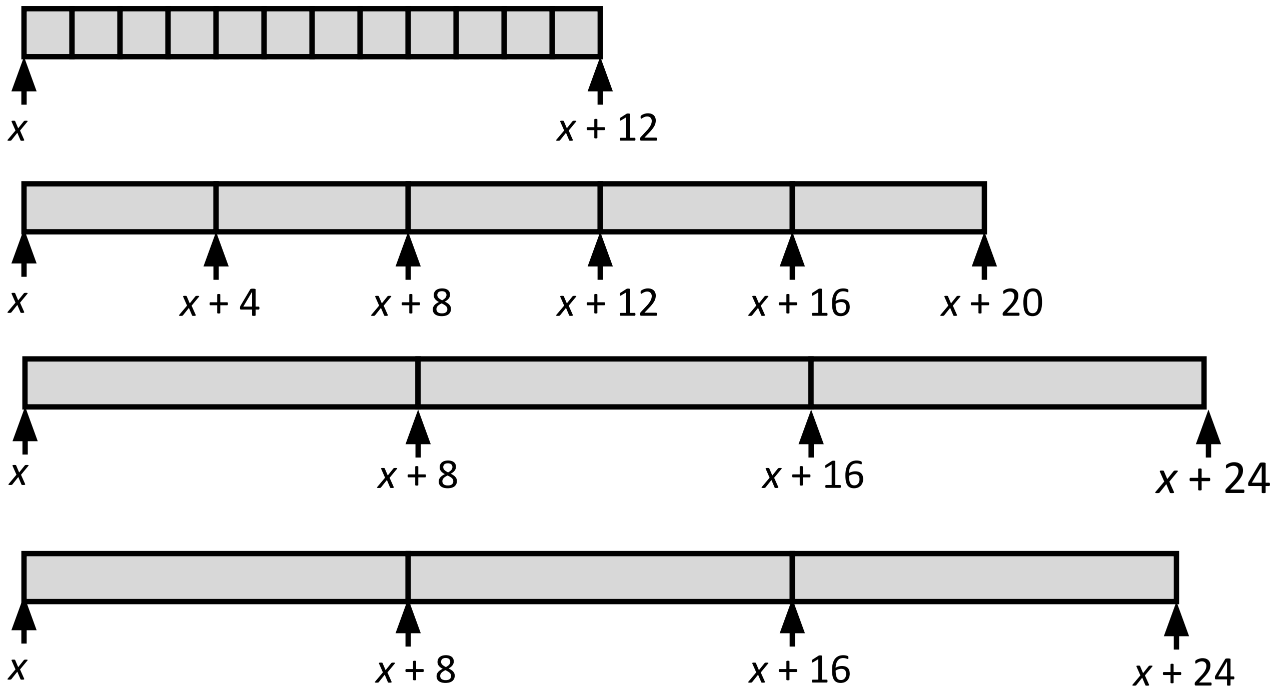
- Basic Principle
  - **`T`** `A[N];` → array of data type **`T`** and length `N`
  - *Contiguously* allocated region of `N*sizeof(`**`T`**`)` bytes
  - Identifier `A` returns address of array (type **`T*`**)

**`char`** `msg[12];`

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

**`double`** `a[3];`

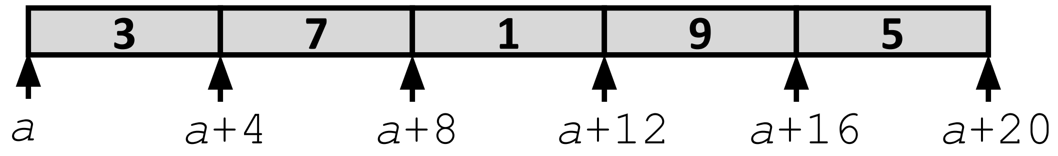
**`char*`** `p[3];`
(or **`char`** **`*`**`p[3];`)



# Review: Array Access

- Basic Principle
  - `T A[N];` → array of data type `T` and length `N`
  - Identifier `A` returns address of array (type `T*`)

`int x[5];`



- Reference / Type / Value

| Reference | Type | Value |
|---|---|---|
| `x[4]` | `int` | `5` |
| `x` | `int*` | `a` |
| `x+1` | `int*` | `a + 4` |
| `&x[2]` | `int*` | `a + 8` |
| `x[5]` | `int` | `??` (whatever's in memory at addr `x+20`) |
| `*(x+1)` | `int` | `7` |
| `x+i` | `int*` | `a + 4*i` |

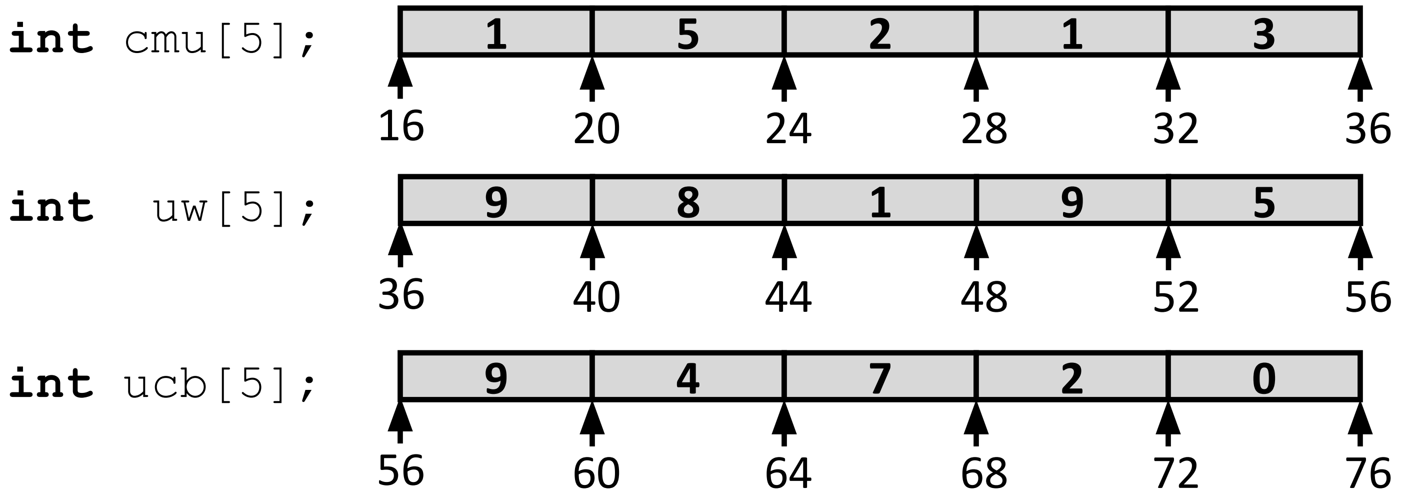
## Array Example

```
// arrays of ZIP code digits
int cmu[5] = { 1, 5, 2, 1, 3 };
int  uw[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };
```

brace-enclosed list initialization

## Array Example

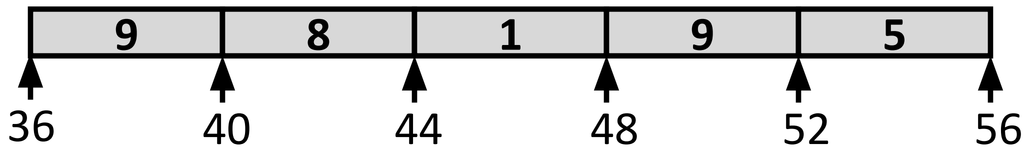
```
// arrays of ZIP code digits
int cmu[5] = { 1, 5, 2, 1, 3 };
int  uw[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };
```



- Example arrays happened to be allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

## Array Accessing Example

```
int uw[5];
```



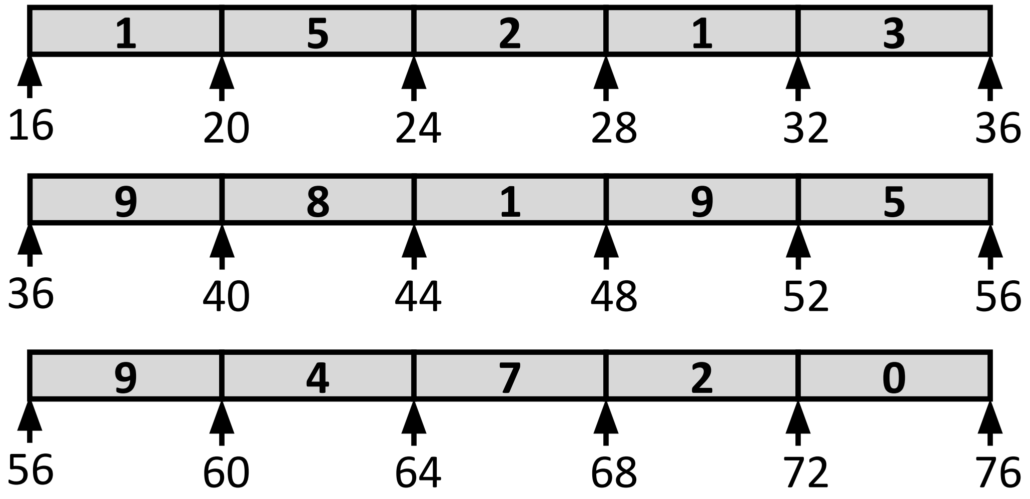
```
// return specified digit of ZIP code
int get_digit(int z[5], int digit) {
  return z[digit];
}
```

```
get_digit:
  movl  (%rdi,%rsi,4), %eax   # z[digit]
```

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired digit at `%rdi+4*%rsi`, so use `(%rdi,%rsi,4)`

# Referencing Examples

```
int cmu[5];
int  uw[5];
int ucb[5];
```



| Reference | Address | Value | Guaranteed? |
|---|---|---|---|
| `uw[3]` | `48` | `9` | `Yes!` |
| `uw[6]` | `60` | `4` | `No!` |
| `uw[-1]` | `32` | `3` | `No!` |
| `cmu[15]` | **`80`** | **`??`** | **`???`** |

- No bounds checking
- Example arrays happened to be allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

## C Details:  Arrays and Pointers

- Arrays are (almost) identical to pointers
  - `char *string` and `char string[]` are nearly identical declarations
  - Differ in subtle ways:  initialization, `sizeof()`, etc.

- An array name is an expression (not a variable) that returns the address of the array
  - It *looks* like a pointer to the first (0th) element
    - `*ar` same as `ar[0]`, `*(ar+2)` same as `ar[2]`
  - An array name is read-only (no assignment) because it is a *label*
    - Cannot use "`ar = <anything>`"

## C Details: Arrays and Functions

- Declared arrays only allocated while in scope:

```
char* foo() {
    char string[32]; ...;
    return string;
}
```

BAD!

- An array is passed to a function as a pointer:
  - Array size gets lost!

*Really* `int *ar`

```
int foo(int ar[], unsigned int size) {
    ... ar[size-1] ...
}
```

Must explicitly pass the size!

## Feelings check:
## 1D Arrays!

# Data Structures in Assembly

- **Arrays**
  - One-dimensional
  - **Multidimensional (nested)**
  - Multilevel
- Structs
  - Alignment
- ~~Unions~~

## Nested Array Example

```
int sea[4][5] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

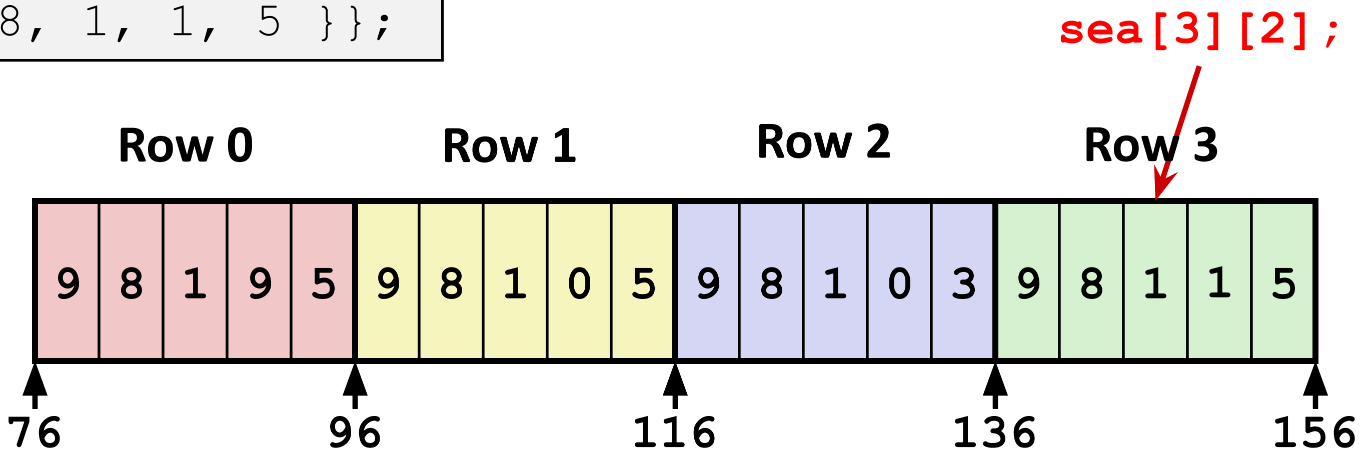
Remember, `T A[N]` is an array with elements of type `T`, with length `N`

- What is the layout in memory?

# Nested Array Example

```
int sea[4][5] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

Remember, **`T`** `A[N]` is an array with elements of type **`T`**, with length `N`



- "Row-major" ordering of all elements
- Elements in the same row are contiguous
- Guaranteed (in C)

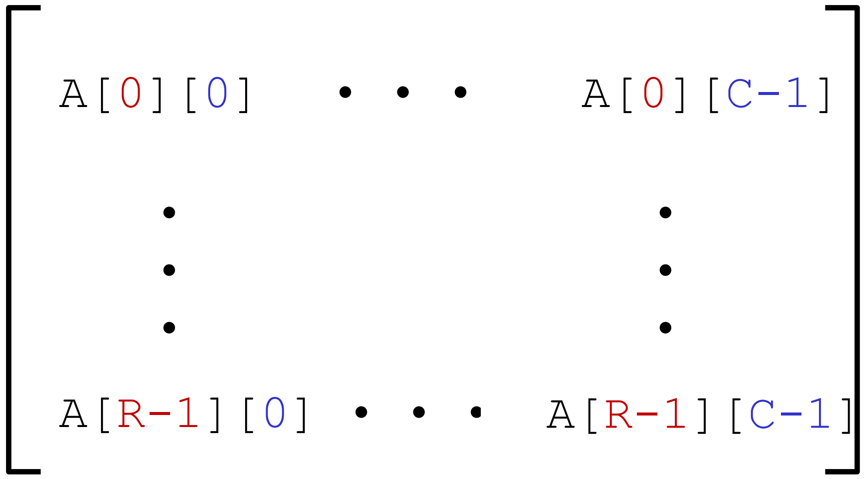
## Two-Dimensional (Nested) Arrays

- Declaration: **`T`** `A[R][C];`
  - 2D array of data type `T`
  - `R` rows, `C` columns
  - Each element requires **`sizeof`**`(`**`T`**`)` bytes
- Array size?

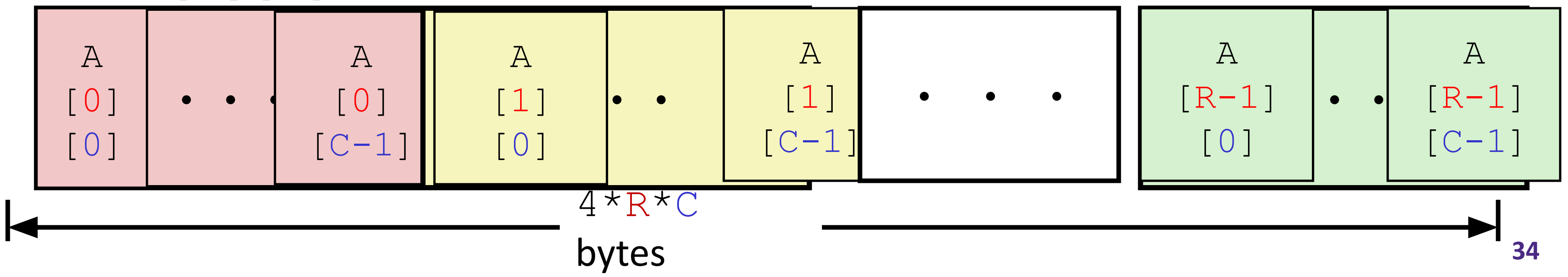
$$\begin{bmatrix} \texttt{A[0][0]} & \cdots & \texttt{A[0][C-1]} \\ \vdots & & \vdots \\ \texttt{A[R-1][0]} & \cdots & \texttt{A[R-1][C-1]} \end{bmatrix}$$

# Two-Dimensional (Nested) Arrays

- Declaration: `T A[R][C];`
  - 2D array of data type `T`
  - `R` rows, `C` columns
  - Each element requires `sizeof(T)` bytes
- Array size:
  - `R*C*sizeof(T)` bytes
- Arrangement: **row-major** ordering



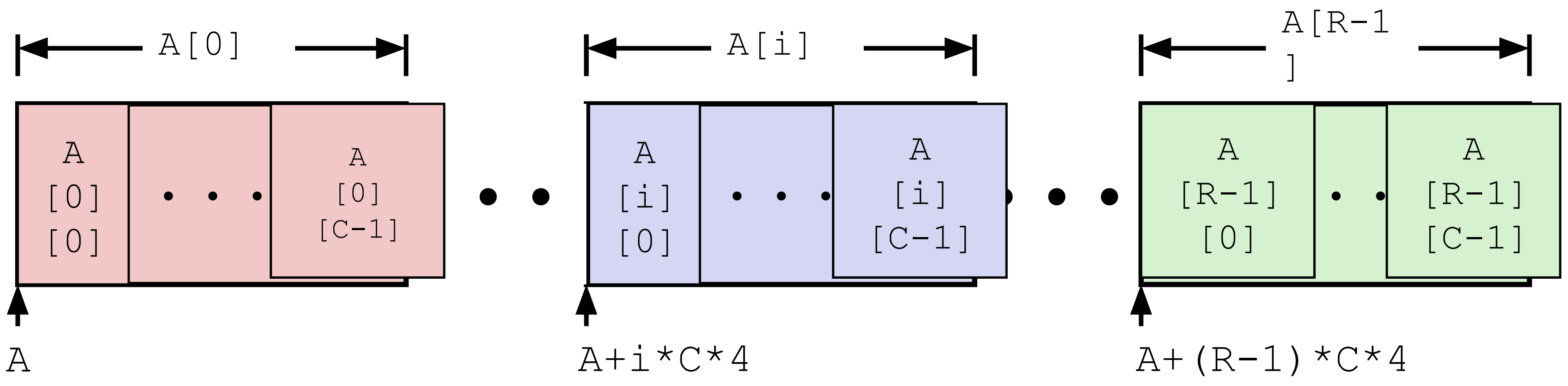
`int A[R][C];`



# Nested Array Row Access

- Row vectors
  - Given **`T`** `A[R][C]`,
    - `A[i]` is an array of `C` elements ("row `i`")
    - `A` is address of array
    - Starting address of row `i` = `A + i*(C * sizeof(`**`T`**`))`

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



# Nested Array Row Access Code

```
int* get_sea_zip(int index)
{
  return sea[index];
}
```

```
int sea[4][5] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

```
get_sea_zip(int):
        movslq  %edi, %rdi
        leaq    (%rdi,%rdi,4), %rax
        leaq    sea(,%rax,4), %rax
        ret

sea:
        .long   9
        .long   8
        .long   1
        .long   9
        .long   5
        .long   9
        .long   8
...
```

# Nested Array Row Access Code

```
int* get_sea_zip(int index)
{
  return sea[index];
}
```

```
int sea[4][5] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

- What data type is `sea[index]`?
- What is its value?

```
  # %rdi = index
  leaq (%rdi,%rdi,4),%rax
  leaq sea(,%rax,4),%rax
```

**Translation?**

# Nested Array Row Access Code

```
int* get_sea_zip(int index)
{
  return sea[index];
}
```

```
int sea[4][5] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

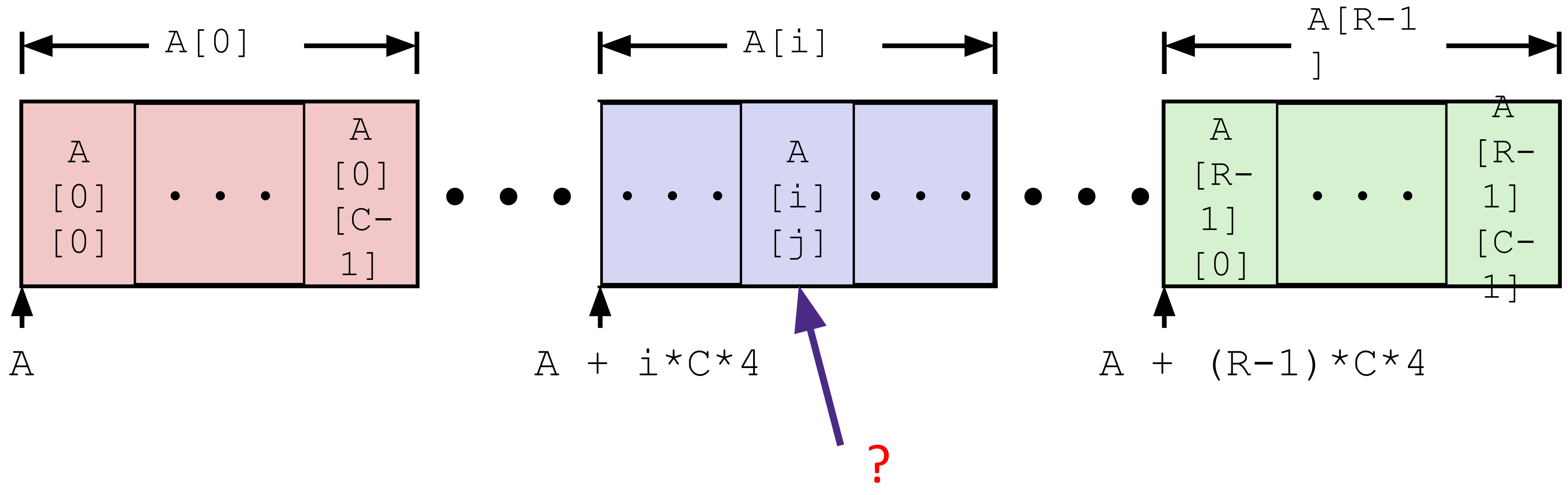
```
  # %rdi = index
  leaq (%rdi,%rdi,4),%rax  # 5 * index
  leaq sea(,%rax,4),%rax   # sea + (20 * index)
```

- Row Vector
  - `sea[index]` is array of 5 `int`s
  - Starting address = `sea+20*index`
- Assembly Code
  - Computes and returns address
  - Compute as: `sea+4*(index+4*index)= sea+20*index`

## Nested Array Element Access

- Array Elements
  - `A[i][j]` is element of type **`T`**, which requires *K* bytes
  - Address of `A[i][j]` is

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

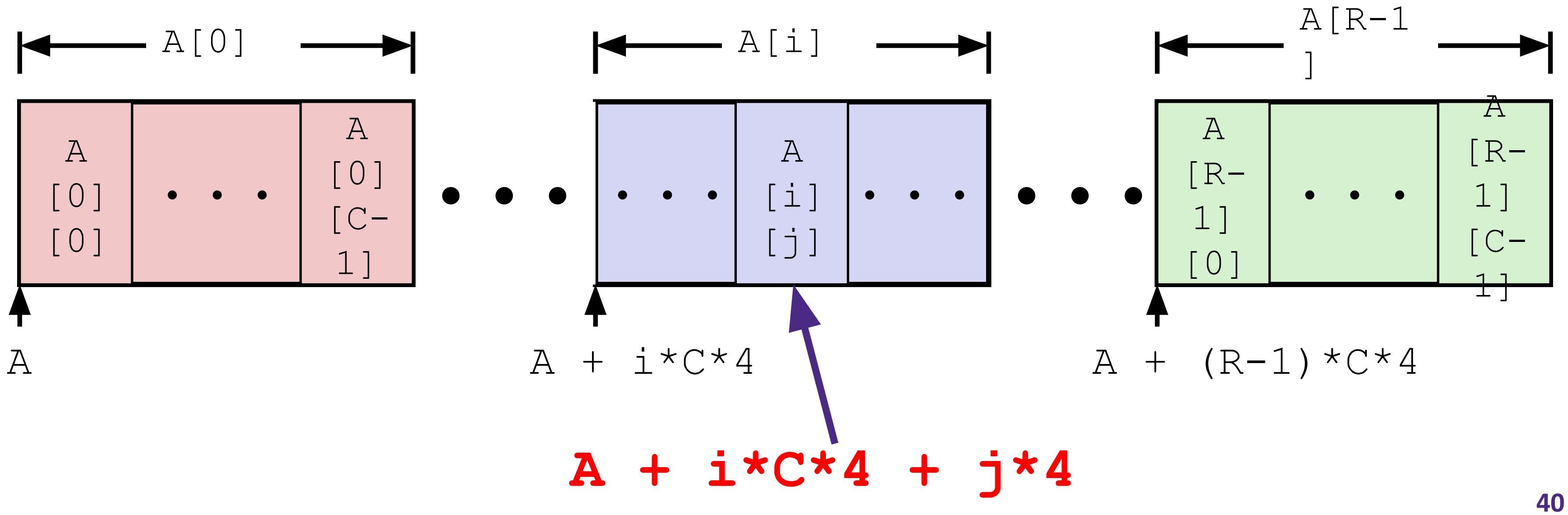


# Nested Array Element Access

- Array Elements
  - `A[i][j]` is element of type **`T`**, which requires $K$ bytes
  - Address of `A[i][j]` is

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

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



# Nested Array Element Access Code

```
int get_sea_digit
   (int index, int digit)
{
  return sea[index][digit];
}
```

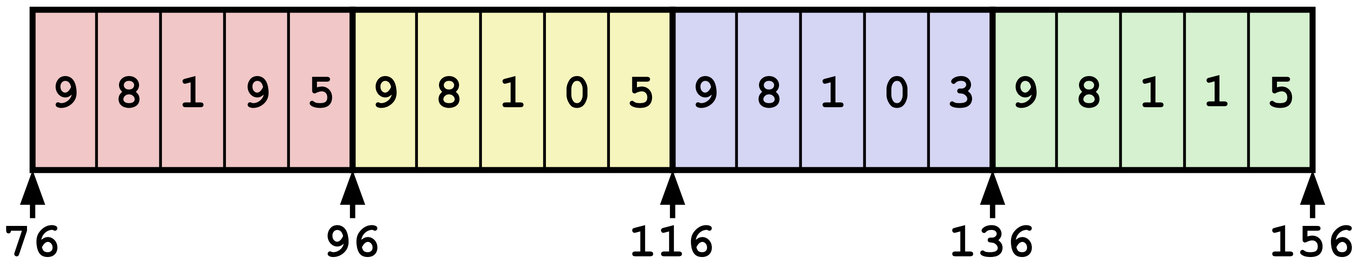
```
int sea[4][5] =
  {{ 9, 8, 1, 9, 5 },
   { 9, 8, 1, 0, 5 },
   { 9, 8, 1, 0, 3 },
   { 9, 8, 1, 1, 5 }};
```

```
    leaq   (%rdi,%rdi,4), %rax   # 5*index
    addl   %rax, %rsi            # 5*index+digit
    movl   sea(,%rsi,4),  %eax   # *(sea + 4*(5*index+digit))
```

- Array Elements
  - `sea[index][digit]` is an **`int`** (**`sizeof`**`(`**`int`**`)=4`)
  - Address = `sea + 5*4*index + 4*digit`
- Assembly Code
  - Address as: `sea + ((index+4*index) + digit)*4`
  - `movl` performs memory reference

# Multidimensional Referencing Examples

`int sea[4][5];`



| Reference | Address | Value | Guaranteed? |
|---|---|---|---|
| `sea[3][3]` | 148 | 1 | Yes! |
| `sea[2][5]` | 136 | 9 | Yes! |
| `sea[2][-1]` | 112 | 5 | Yes! |
| `sea[4][-1]` | 152 | 5 | Yes! |
| `sea[0][19]` | 152 | 5 | Yes! |
| `sea[0][-1]` | **72** | **??** | **??** |

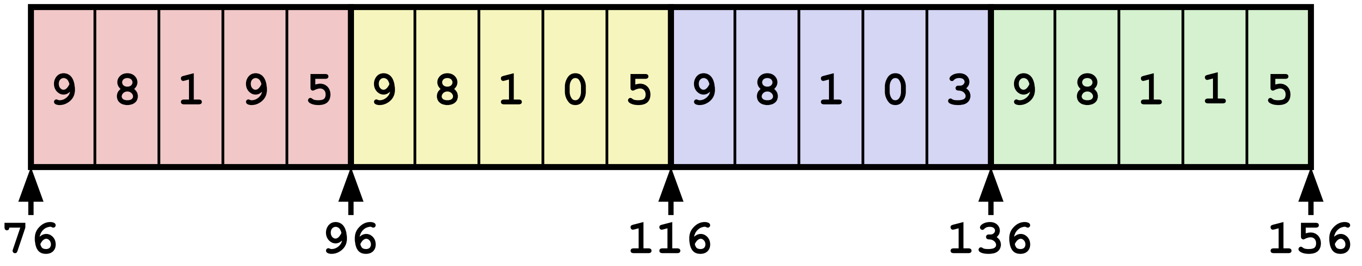
- Code does not do any bounds checking
- Ordering of elements within array guaranteed

## Feelings check: 1-D, multi-D Arrays

# Checking in...

- Which of the following statements is FALSE?

`int sea[4][5];`



🐶 `sea[4][-2]` is a *valid* array reference

🐈 `sea[1][1]` makes *two* memory accesses

🐑 `sea[2][1]` will *always* be a higher address than `sea[1][2]`

🦄 `sea[2]` is calculated using *only* `lea`

🥶 We're lost…

# Data Structures in Assembly

- **Arrays**
  - One-dimensional
  - Multidimensional (nested)
  - **Multilevel**
- Structs
  - Alignment
- ~~Unions~~

# Multilevel Array Example

**Multilevel Array Declaration(s):**

```
int cmu[5] = { 1, 5, 2, 1, 3 };
int  uw[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };
```

```
int* univ[3] = {uw, cmu, ucb};
```

Is a multilevel array the same thing as a 2D array?

NO

**2D Array Declaration:**

```
int univ2D[3][5] = {
  { 9, 8, 1, 9, 5 },
  { 1, 5, 2, 1, 3 },
  { 9, 4, 7, 2, 0 }
};
```

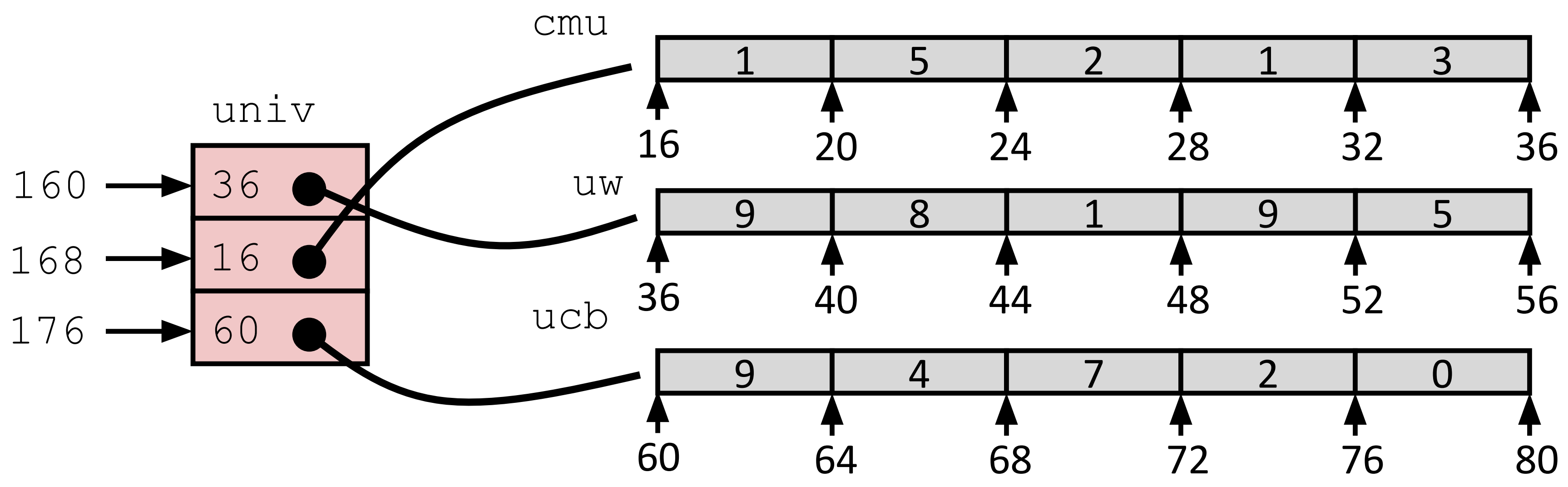
One array declaration = one contiguous block of memory

# Multilevel Array Example

```
int cmu[5] = { 1, 5, 2, 1, 3 };
int  uw[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };
```

```
int* univ[3] = {uw, cmu, ucb};
```

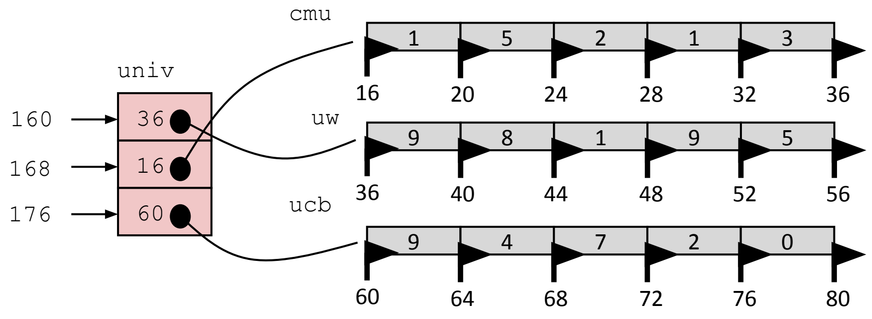
- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 8 bytes each
- Each pointer points to array of `int`s



Note: this is how Java represents multidimensional arrays

# Element Access in Multilevel Array

```
int get_univ_digit
  (int index, int digit)
{
  return univ[index][digit];
}
```



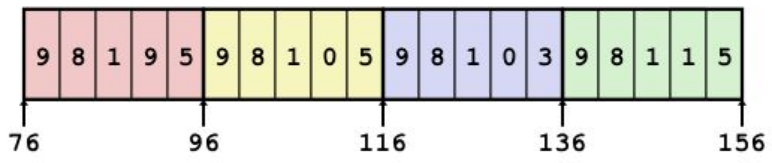
```
salq    $2, %rsi            # rsi = 4*digit
addq    univ(,%rdi,8), %rsi # p = univ[index] + 4*digit
movl    (%rsi), %eax        # return *p
ret
```

- Computation
  - Elem access Mem[Mem[`univ+8*index`]+`4*digit`]
  - Must do **two memory reads**
    - get pointer to row array, access element within array
  - But allows inner arrays to be different lengths

# Array Element Accesses

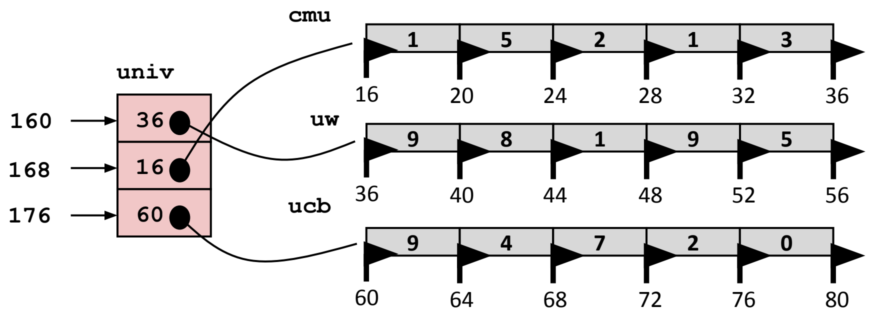
## Multidimensional array

```
int get_sea_digit
  (int index, int digit)
{
  return sea[index][digit];
}
```



## Multilevel array

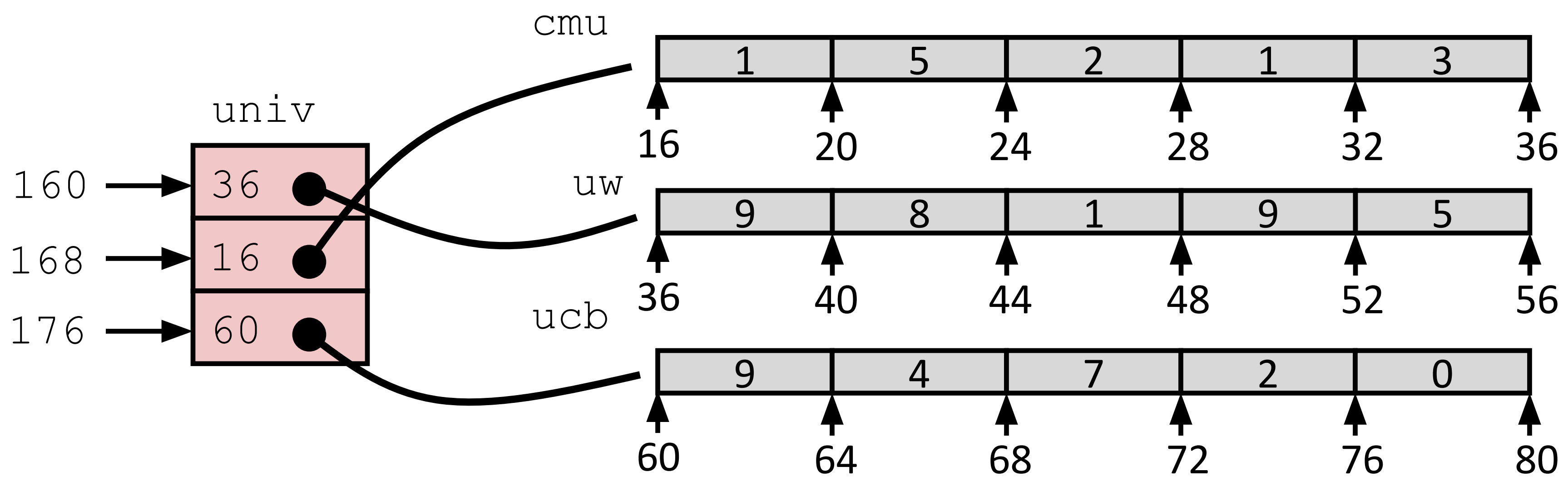
```
int get_univ_digit
  (int index, int digit)
{
  return univ[index][digit];
}
```



Access *looks* the same, but it isn't:

Mem[sea+20*index+4*digit]    Mem[Mem[univ+8*index]+4*digit]

# Multilevel Referencing Examples



| Reference | Address | Value | Guaranteed? |
|---|---|---|---|
| `univ[2][3]` | 72 | 2 | Yes! |
| `univ[1][5]` | 52 | 5 | Yes! |
| `univ[2][-2]` | 52 | 5 | No! |
| `univ[3][-1]` | ?? | ?? | No! |
| `univ[1][12]` | **84** | **??** | **No!** |

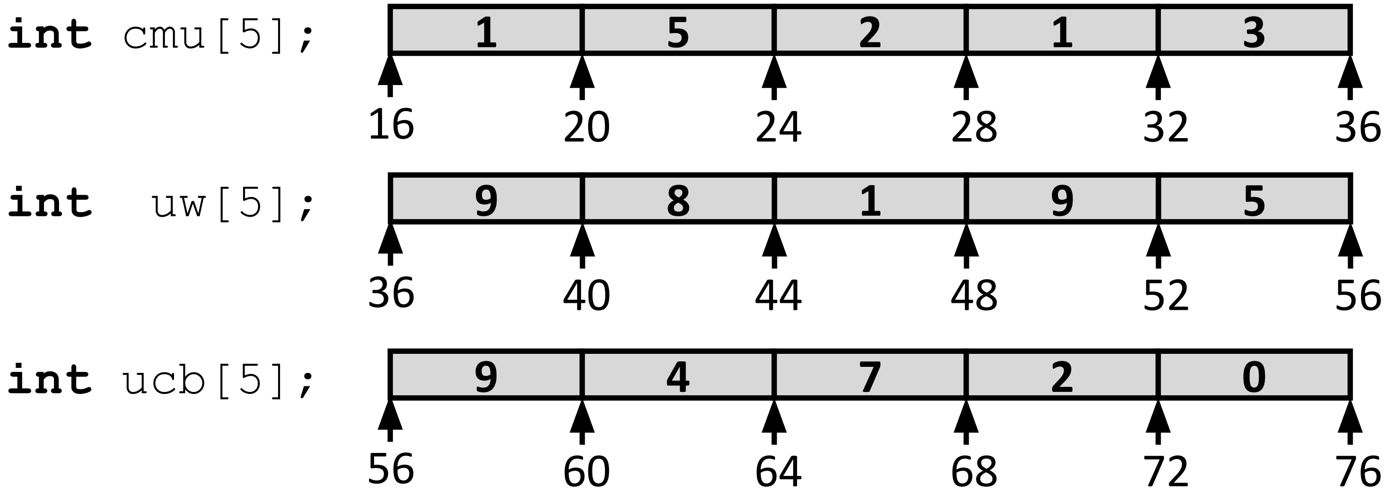
- C code does not do any bounds checking
- Location of each lower-level array in memory is *not* guaranteed

## Feelings check: Multi-level arrays

## Summary

- Contiguous allocations of memory
- No bounds checking (and no default initialization)
- Can usually be treated like a pointer to first element
- `int a[4][5];` → array of arrays
  - all levels in one contiguous block of memory
- `int* b[4];` → array of pointers to arrays
  - First level in one contiguous block of memory
  - Each element in the first level points to another “sub” array
  - Parts anywhere in memory

# Referencing Examples

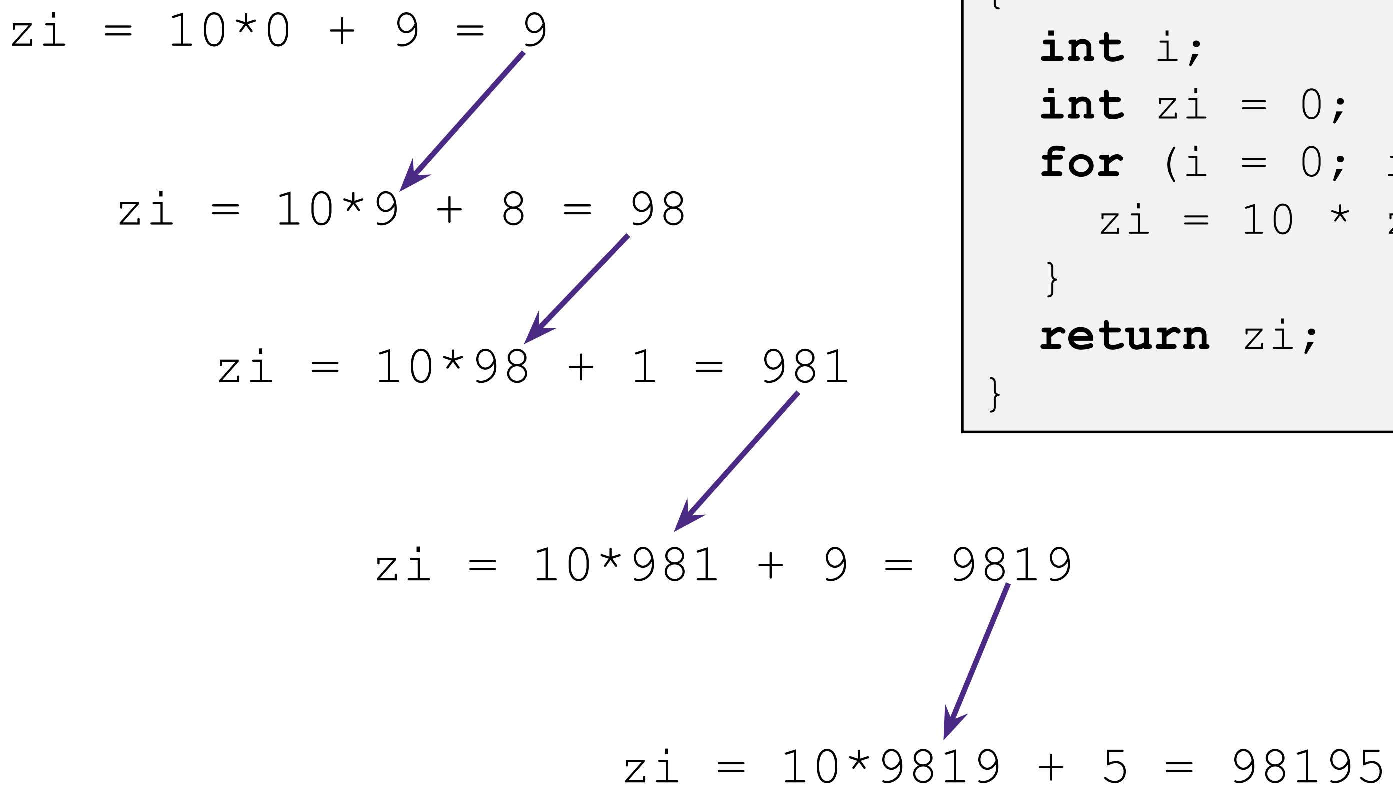


| Reference | Address | Value | Guaranteed? |
|---|---|---|---|
| `uw[3]` | `36 + 4* 3 = 48` | `9` | Yes |
| `uw[6]` | `36 + 4* 6 = 60` | `4` | No |
| `uw[-1]` | `36 + 4*-1 = 32` | `3` | No |
| `cmu[15]` | `16 + 4*15 = 76` | `??` | No |

- No bounds checking
- Example arrays happened to be allocated in successive 20 byte blocks
    - Not guaranteed to happen in general

# Array Loop Example

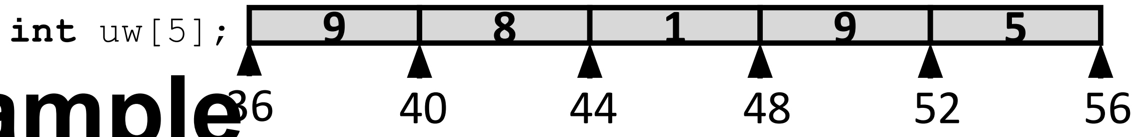
```
int zd2int(int z[5])
{
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```



```
zi = 10*0 + 9 = 9
zi = 10*9 + 8 = 98
zi = 10*98 + 1 = 981
zi = 10*981 + 9 = 9819
zi = 10*9819 + 5 = 98195
```

| | | | | |
|---|---|---|---|---|
| 9 | 8 | 1 | 9 | 5 |

# Array Loop Example



- Original:

```
int zd2int(int z[5])
{
  int i;
  int zi = 0;
  for (i = 0; i < 5; i++) {
    zi = 10 * zi + z[i];
  }
  return zi;
}
```

- Transformed:
  - Eliminate loop variable `i`, use pointer `zend` instead
  - Convert array code to pointer code
    - Pointer arithmetic on `z`
  - Express in do-while form (no test at entrance)

```
int zd2int(int z[5])
{
  int zi = 0;
  int *zend = z + 5;
  do {
    zi = 10 * zi + *z;
    z++;
  } while (z < zend);
  return zi;
}
```

address just past 5th digit

Increments by 4 (size of int)

# Array Loop Implementation

gcc with -O1

- Registers:

  ```
  %rdi    z
  %rax    zi
  %rcx    zend
  ```

- Computations
  -
  -

```
int zd2int(int z[5])
{
  int zi = 0;
  int *zend = z + 5;
  do {
    zi = 10 * zi + *z;
    z++;
  } while (z < zend);
  return zi;
}
```

```
    # %rdi = z
    leaq 20(%rdi),%rcx   #
    movl $0,%eax  #
.L17:
    leal  (%rax,%rax,4),%edx #
    movl  (%rdi),%eax #
    leal  (%rax,%rdx,2),%eax #
    addq $4,%rdi  #
    cmpq %rdi,%rcx    #
    jne .L17   #
```

# Array Loop Implementation `gcc with -O1`

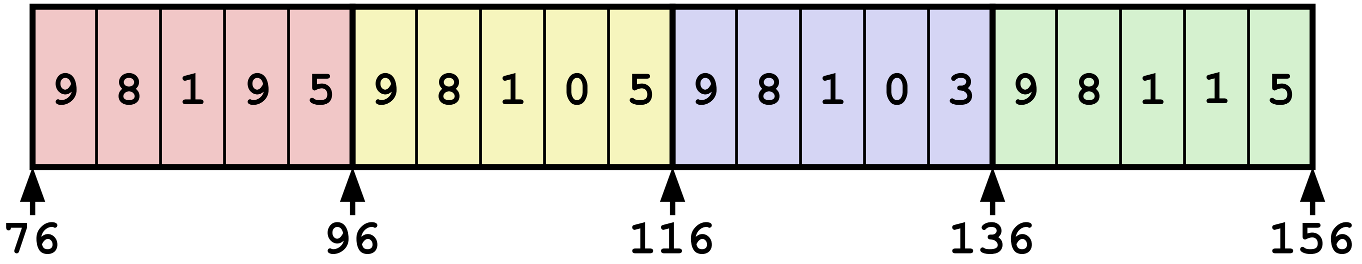
- Registers:
  ```
  %rdi    z
  %rax    zi
  %rcx    zend
  ```
- Computations
  - `10*zi + *z` implemented as: `*z + 2*(5*zi)`
  - `z++` increments by 4 (size of `int`)

```
int zd2int(int z[5])
{
  int zi = 0;
  int *zend = z + 5;
  do {
    zi = 10 * zi + *z;
    z++;
  } while (z < zend);
  return zi;
}
```

```
    # %rdi = z
    leaq 20(%rdi),%rcx   # rcx = zend = z+5
    movl $0,%eax  # rax = zi = 0
.L17:
    leal  (%rax,%rax,4),%edx # zi + 4*zi = 5*zi
    movl  (%rdi),%eax # eax = *z
    leal  (%rax,%rdx,2),%eax # zi = *z + 2*(5*zi)
    addq $4,%rdi  # z++
    cmpq %rdi,%rcx    # zend - z
    jne .L17   # if != goto loop
```

# Strange Referencing Examples

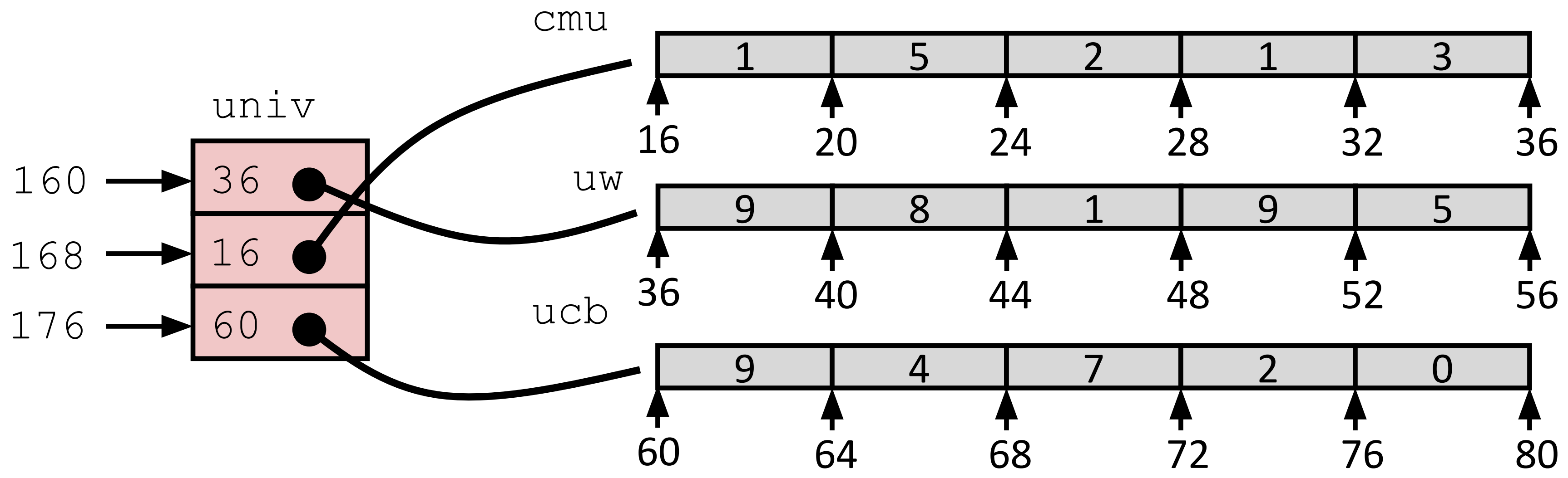
`int sea[4][5];`



| Reference | Address | Value | Guaranteed? |
|---|---|---|---|
| `sea[3][3]` | `76+20*3+4*3 = 148` | `1` | Yes |
| `sea[2][5]` | `76+20*2+4*5 = 136` | `9` | Yes |
| `sea[2][-1]` | `76+20*2+4*-1 = 112` | `5` | Yes |
| `sea[4][-1]` | `76+20*4+4*-1 = 152` | `5` | Yes |
| `sea[0][19]` | `76+20*0+4*19 = 152` | `5` | Yes |
| `sea[0][-1]` | `76+20*0+4*-1 = 72` | `??` | No |

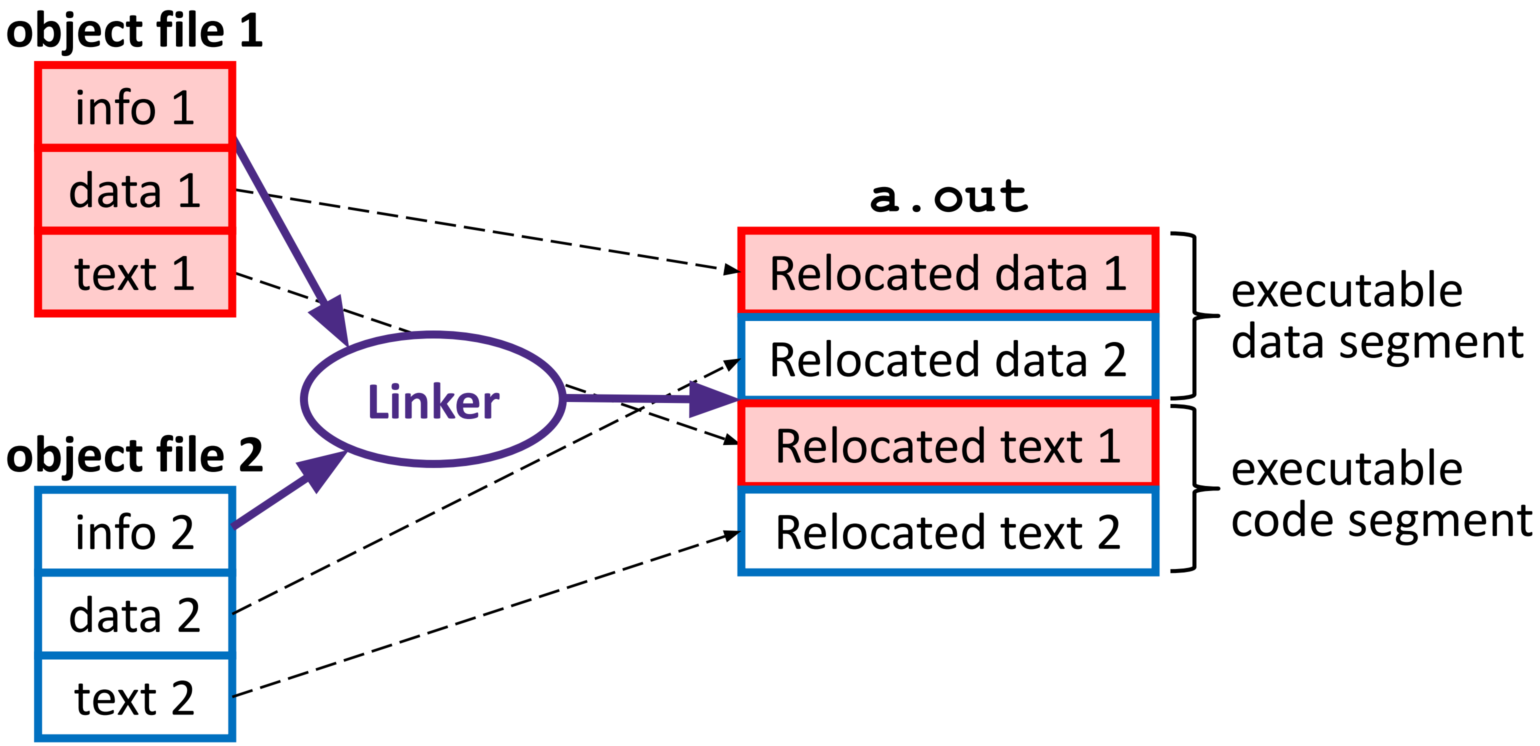
- Code does not do any bounds checking
- Ordering of elements within array guaranteed

# Strange Referencing Examples



| Reference | Address | Value | Guaranteed? |
|---|---|---|---|
| `univ[2][3]` | `60+4*3 = 72` | `2` | Yes |
| `univ[1][5]` | `16+4*5 = 36` | `9` | No |
| `univ[2][-2]` | `60+4*-2 = 52` | `5` | No |
| `univ[3][-1]` | `#@%!^??` | `??` | No |
| `univ[1][12]` | `16+4*12 = 64` | `4` | No |

- C code does not do any bounds checking
- Location of each lower-level array in memory is *not* guaranteed


object file 1
info 1
data 1
text 1
object file 2
info 2
data 2
text 2
Linker
a.out
Relocated data 1
Relocated data 2
Relocated text 1
Relocated text 2
executable data segment
executable code segment

## Individuals & Structures

## Breakouts!
## Define Accessibility!

## Arrays in C

- No bounds checking (considered inefficient)
  - Prioritization of values, efficiency > safety
  - C - Camping; bounds checking is like hot water

## Two narratives: C

"I think programmers should know enough to not access array elements out of bounds. It's a relatively simple check to insert at the language level, and if you can't remember to add it, you shouldn't write C"

- *Emphasis on the individual*

"C is an absolutely awful language; why on earth doesn't it implement bounds checking? It's an expense, but a relatively nominal one, and the language would be so much easier to use"

- *Emphasis on the structure*

## We’ve seen this before!

But, let’s look elsewhere

## Two narratives: Privacy

"I think people should know enough to change the privacy settings on their phones. It's a relatively simple setting change and if you can't figure that out, you don't deserve privacy"

- *Emphasis on the individual*

"Phones are awful; why on earth aren't they private by default? Or, why aren't the options presented to users? It's not a complicated check, and folks would have better relationships with phones because of it"

- *Emphasis on the structure*

## Two narratives: Accessibility

"I think people should know enough to change font sizes on their phone. It's a relatively simple settings change, and if you can't figure that out, you shouldn't use a phone"

- *Emphasis on the individual*

"It's bananas that phones do anything before checking font size. So many people are vision impaired, how do manufacturers expect anything from people before they can read what's on the screen?"

- *Emphasis on the structure*

## Individual vs. Structure

- There's lots of examples, especially in tech
  - Privacy is a commodity
  - People should know better than to click on ads
  - "You're a bad person if you don't recycle"
  - Everyone should aim for zero-waste
  - Don't compare floats for equality
  - Remember to check array bounds in C
  - "If you can't access *x*, you shouldn't use *x*"
  - …
  - …
  - …

## This comes up everywhere! Remember Neoliberalism?

“Personal Responsibility”
Individualism, etc.

## Neoliberalism

- I know it’s not CS, but there’s a strong influence

## Neoliberalism

- **Neoliberalism**: Everything that happens to you is because of your actions. You’re free to make your own decisions. Your access *anything* (housing, medical care) is your responsibility.
  - *Tends to ignore systemic/structural bias & inequity*

## Neoliberalism & Masculinity

- **Neoliberalism**: Everything that happens to you is because of your actions. You're free to make your own decisions. Your access *anything* (housing, medical care) is your responsibility.
    - *Tends to ignore systemic/structural bias & inequity*
- **Masculinity:** Keeping your man card means not asking for help, not showing your emotions, not caring about personal expression, and perpetuating cultures of violence and dominance.
    - *This, at least, was my experience*

# Rugged, Individualistic, Minimalism!

## Neoliberalism & Masculinity & CS

- **C: *Rugged*, *Individualistic*, Minimalism**
  - You're on your own, there's no one to ask for help, and if you mess up, it's your own fault
- **Sound familiar?**
  - Neoliberalism's "personal responsibility"
  - Masculinity's pressure to refuse help?
  - **Individual emphasized over the structure that the individual works within**

## We’re switching perspectives to accessibility!

## Accessibility, definition

- *Usable by people experiencing disabilities*
  - Usually, around vision/mobility deficits
- *Usable by anyone, independent of their physical or cognitive capabilities*
  - A bit stronger, a bit more verbose

## What do we mean by “usable”?

## Defining Usability

- If I write C and forget to bounds-check arrays, whose fault is it?
  - Mine? “I should have known better”
  - K&R’s? “They should have known better”
- Blame tends to be an individualistic focus
  - Sometimes helpful, i.e. malicious criminal cases
  - Sometimes less helpful, i.e. racism
- **Use, without causing harm, independent of physical or cognitive capabilities**
  - Inaccessibility is a structural issue, not a personal one

## Is C accessible?

## Is programming accessible?

## Accessibility & CS

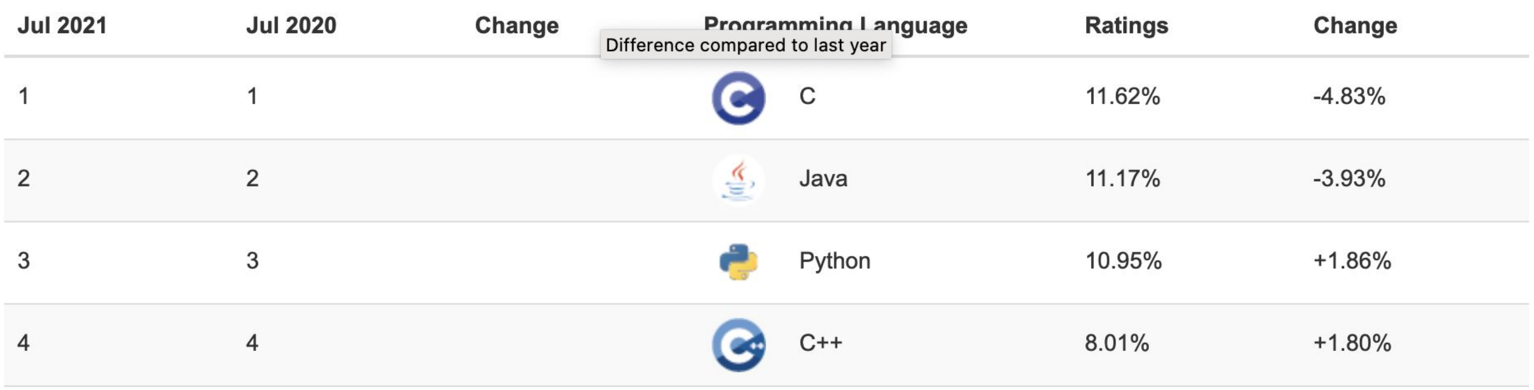
- CS, programming, in general, is inaccessible
  - Lots of cognitive requirements
- Many consumer technologies aren’t accessible
  - Few designed with accessibility in mine
- Programming tech isn’t accessible either!
  - Tendency to over-emphasize individual → ideological foundations of CS
  - Many structures aren’t usable, “just don’t use them” isn’t always an option
  - Also, switches emphasis back to individual, away from structural inequity

## Not everyone can go camping!

### Even among computer scientists.

## Maybe just don't use C?

- You don't have a choice!
  - You might work on legacy code (lots of C)
  - You might work in software systems (lots of C)
  - You might want to hack on Arduinos (C by default)
  - You might just be programming (C's #1, 01/2021)
- They really didn't think this through...



Difference compared to last year

| Jul 2021 | Jul 2020 | Change | Programming Language | Ratings | Change |
|---|---|---|---|---|---|
| 1 | 1 | | C | 11.62% | -4.83% |
| 2 | 2 | | Java | 11.17% | -3.93% |
| 3 | 3 | | Python | 10.95% | +1.86% |
| 4 | 4 | | C++ | 8.01% | +1.80% |

# Was C intentionally inaccessible?

*Probably not, honestly*

**Schmitz Hall**
NE 41st St & 15th Ave NE, Seattle
1970
*Designed by Waldron & Pomeroy*

**Condon Hall**
NE 41st St & 12th Ave NE, Seattle
1973,
*Designed by Mitchell/Giurgola Associates and Joyce, Copeland, Vaughan & Nordfors*

"It was the trend at the time, no one knew any better, and no one questioned their ideology"