Executables & Arrays

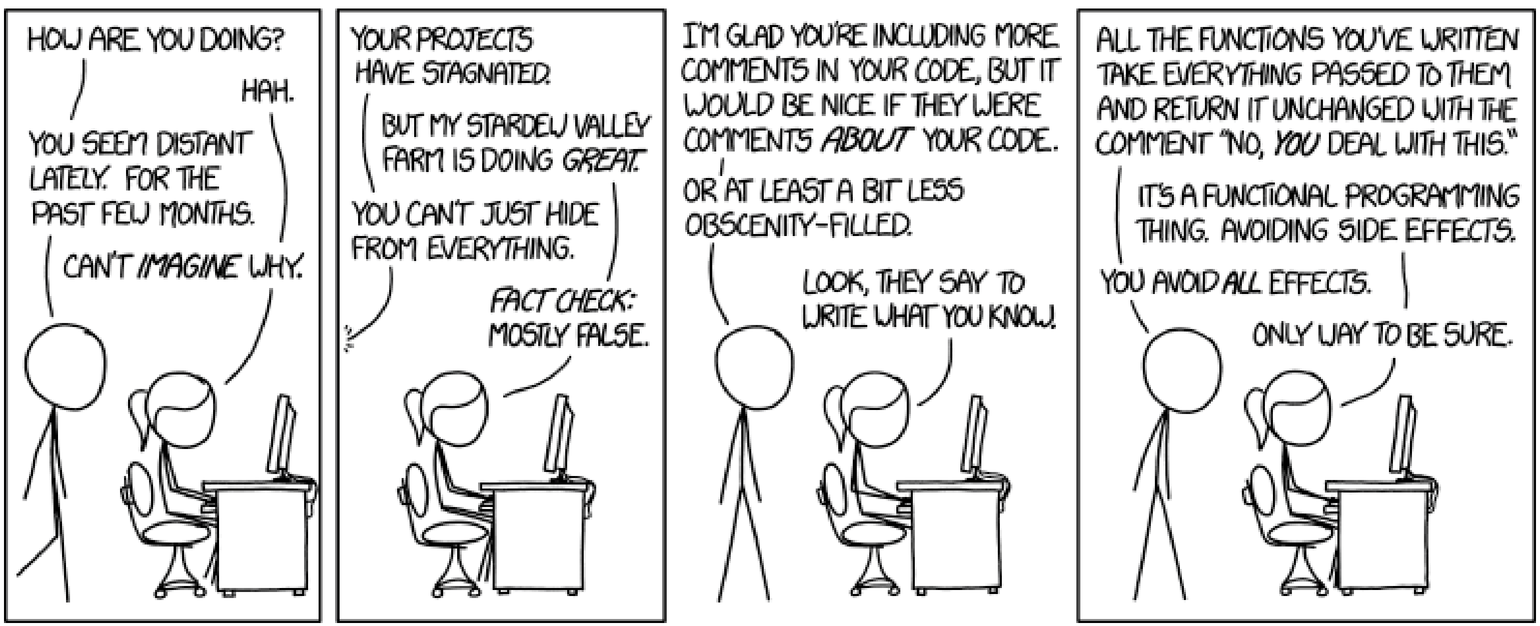
CSE 351 Autumn 2022

Instructor:

Justin Hsia

Teaching Assistants:

Angela Xu, Arjun Narendra, Armin Magness, Assaf Vayner, Carrie Hu, Clare Edmonds, David Dai, Dominick Ta, Effie Zheng, James Froelich, Jenny Peng, Kristina Lansang, Paul Stevans, Renee Ruan, Vincent Xiao



http://xkcd.com/1790/

Relevant Course Information

- Lab 2 & hw12 due Friday (10/28)
- hw13 due *next* Wednesday (11/2)
  - Based on the next two lectures, longer than normal

- Midterm (take home, 11/3-11/5)
  - Midterm review problems in section next week
  - Make notes and use the midterm reference sheet
  - Form study groups and look at past exams!

GDB Demo #2

- Let's examine the `pcount_r` stack frames on a real machine!
  - Using `pcount.c` from the course website

- You will need to use GDB to get through the Midterm
  - Useful debugger in this class and beyond!

- Pay attention to:
  - Checking the current stack frames (`backtrace`)
  - Getting stack frame information (`info frame <#>`)
  - Examining memory (`x`)

Instruction Set Philosophies, Revisited

- *Complex Instruction Set Computing* (CISC):
  Add more and more elaborate and specialized instructions as needed
  - **Design goals:** complete tasks in as few instructions as possible; minimize memory accesses for instructions
- *Reduced Instruction Set Computing* (RISC):
  Keep instruction set small and regular
  - **Design goals:** build fast hardware; instructions should complete in few clock cycles (ideally 1); minimize complexity and maximize performance
- How different are these two philosophies, really?

Instruction Set Philosophies, Revisited

- *Complex Instruction Set Computing* (CISC):
  Add more and more elaborate and specialized instructions as needed
  - **Design goals:** complete tasks in as few instructions as possible; minimize memory accesses for instructions
- *Reduced Instruction Set Computing* (RISC):
  Keep instruction set small and regular
  - **Design goals:** build fast hardware; instructions should complete in few clock cycles (ideally 1); minimize complexity and maximize performance
- How different are these two philosophies, really?
  - Both pursue **efficiency** (**minimalism** is a means to an end)

Mainstream ISAs, Revisited

Does anything feel “off” about this landscape?

x86

Designer	Intel, AMD
Bits	16-bit, 32-bit and 64-bit
Introduced	1978 (16-bit), 1985 (32-bit), 2003 (64-bit)
Design	CISC
Type	Register–memory
Encoding	Variable (1 to 15 bytes)
Branching	Condition code
Endianness	Little

Macbooks & PCs
(Core i3, i5, i7, M)
x86-64 Instruction Set

ARM

Designer	A[illegible]
Bits	32-bit, 64-bit
Introduced	1985
Design	RI[illegible]
Type	[illegible] Register
Encoding	AArch64/A64 and AArch32/A32 use 32-bit instructions, T32 (Thumb-2) [illegible]s mixed 16- and [illegible]
Branching	Condition code, compare and branch
Endianness	Bi (little as default)

Smartphone-like devices
(iPhone, iPad, Raspberry Pi)
ARM Instruction Set

RISC-V

Designer	University of California, Berkeley
Bits	32 · 64 · 128
Introduced	2010
Design	RISC
Type	Load–store
Encoding	Variable
Endianness	Little[1][3]

Mostly research
(some traction in embedded)
RISC-V Instruction Set

Tech Monopolization

- How many “dominant” ISAs are there?
  - 2: x86, ARM
- How many “dominant” phone brands are there?
  - 4: Samsung, Apple, Huawei, Xiaomi
- How many “dominant” operating systems are there?
  - 3/4: Android, iOS/macOS, Windows, Linux (?)
- How many “dominant” chip manufacturers are there?
  - 3: Intel, Samsung, TSMC

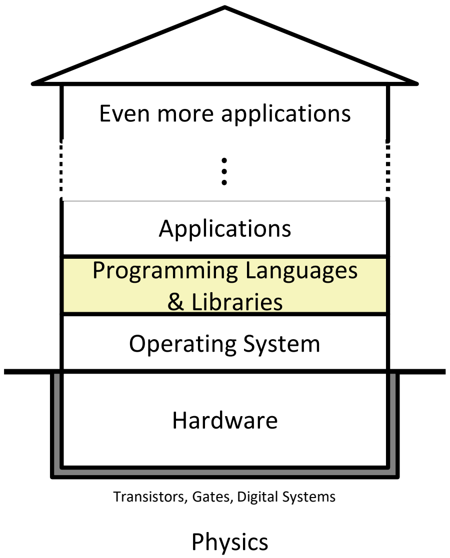
- It wasn’t always this way!
  - Combination of antitrust policies and (lack of) enforcement

Assembly Discussion Questions

- We taught you assembly using x86-64; you didn't have a choice
  - What are some of the advantages of this choice?
  - What are some of the drawbacks of this choice?
  - What are some possible assumptions we are making about our students or values we are forcing on our students with this choice?

The Hardware/Software Interface

- Topic Group 2: **Programs**
  - x86-64 Assembly, Procedures, Stacks, **Executables**



- How are programs created and executed on a CPU?
  - How does your source code become something that your computer understands?
  - How does the CPU organize and manipulate local data?

Reading Review

- Terminology:
  - CALL: compiler, assembler, linker, loader
  - Object file: symbol table, relocation table
  - Disassembly
  - Multidimensional arrays, row-major ordering
  - Multilevel arrays

- Questions from the Reading?

Building an Executable with C (Review)

can compile multiple source files into a single executable

- 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`

text — C program (`p1.c p2.c`)

↓ **C**ompiler (`gcc -Og -S`)

text — Asm program (`p1.s p2.s`)

↓ **A**ssembler (`gcc -c` or `as`)

binary — Object program (`p1.o p2.o`) — Static libraries (`.a`)

↓ **L**inker (`gcc` or `ld`)

binary — Executable program (`p`)

↓ **L**oader (the OS)

Compiler (Review)

- **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 (Review)

- 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 (Review)

- **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 (Review)

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

- Addresses and labels are problematic because the final executable hasn't been constructed yet!
  - Conditional and unconditional jumps
  - Accessing static data (*e.g.*, global variable or jump table)
  - `call`

- So how do we deal with these in the meantime?

Object File Information Tables (Review)

- Each object file has its own symbol and relocation tables

- **Symbol Table** holds list of “items” that may be used by other files "what I have"
  - *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) "what I need"
  - Any *label* or piece of *static data* referenced in an instruction in this file
    - Both internal and external

Object File Format

1) object file header: size and position of the other pieces of the object file "table of contents"
2) text segment: the machine code (Instructions)
3) data segment: data in the source file (binary) (Static Data & Literals)
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 (info for GDB)

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

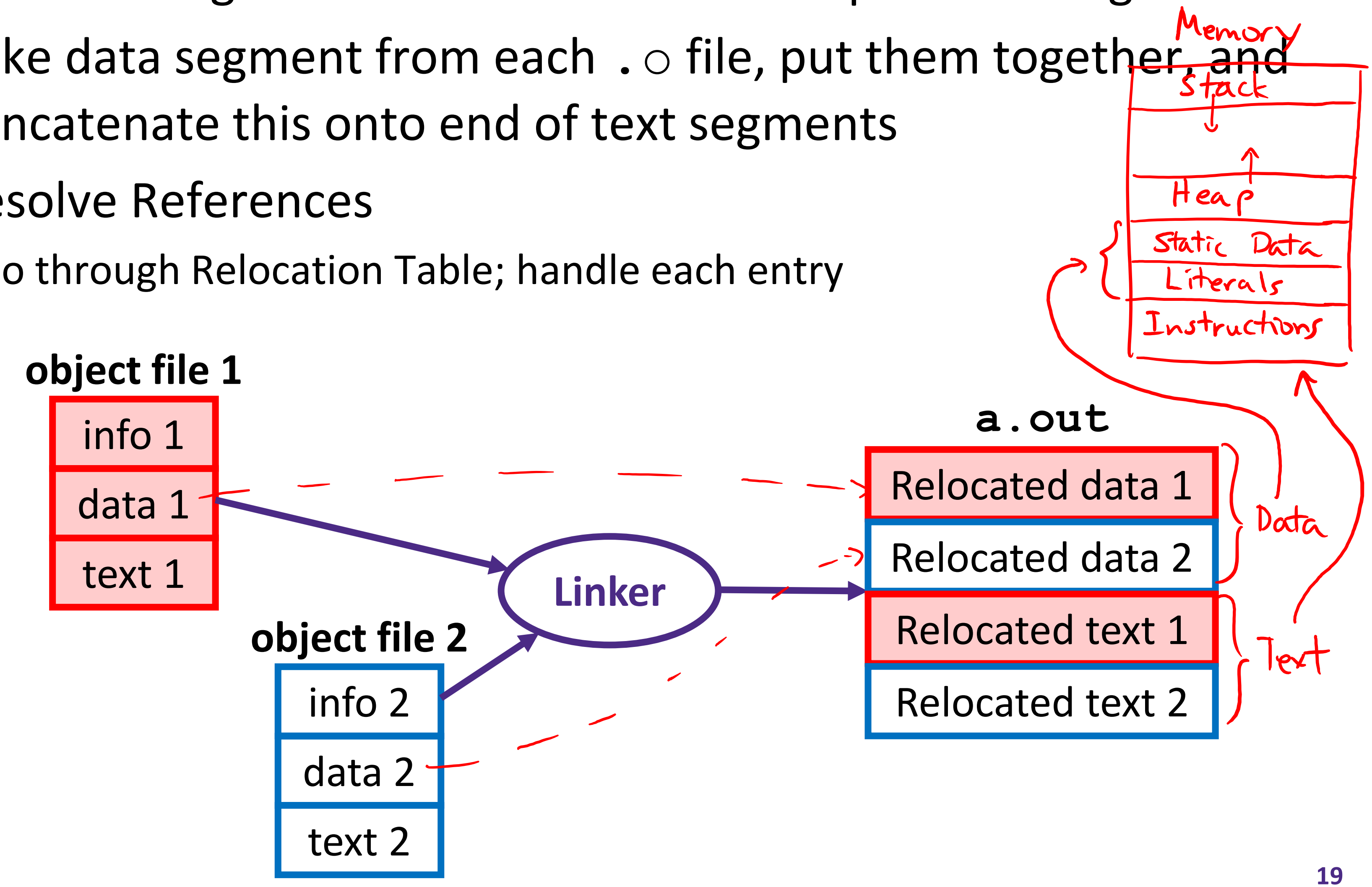
Linker (Review)

- **Input:** Object files (*e.g.*, ELF, COFF)
  - `foo.o`
- **Output:** executable binary program
  - `a.out` ← gcc's default executable name

- 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 (Review)

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 this onto end of text segments
3) Resolve References
   - Go through Relocation Table; handle each entry



Disassembling Object Code (Review)

- Disassembled:

```
0000000000400536 <sumstore>:
                 36 37 38
  400536:        48 01 fe        add    %rdi,%rsi
                 39 3a 3b
  400539:        48 89 32        mov    %rsi,(%rdx)
                 3c
  40053c:        c3              retq
```

address of instruction   object code bytes (hex)   interpreted assembly instructions

- **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 either executable or object 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

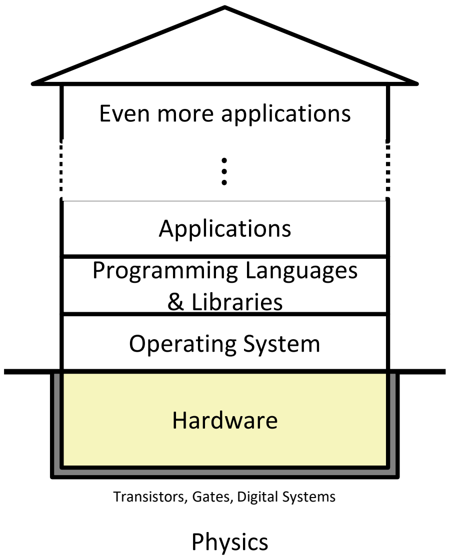
Loader (Review)

- **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) are set up
- Registers are initialized

The Hardware/Software Interface

- Topic Group 1: **Data**
  - Memory, Data, Integers, Floating Point, **Arrays**, Structs



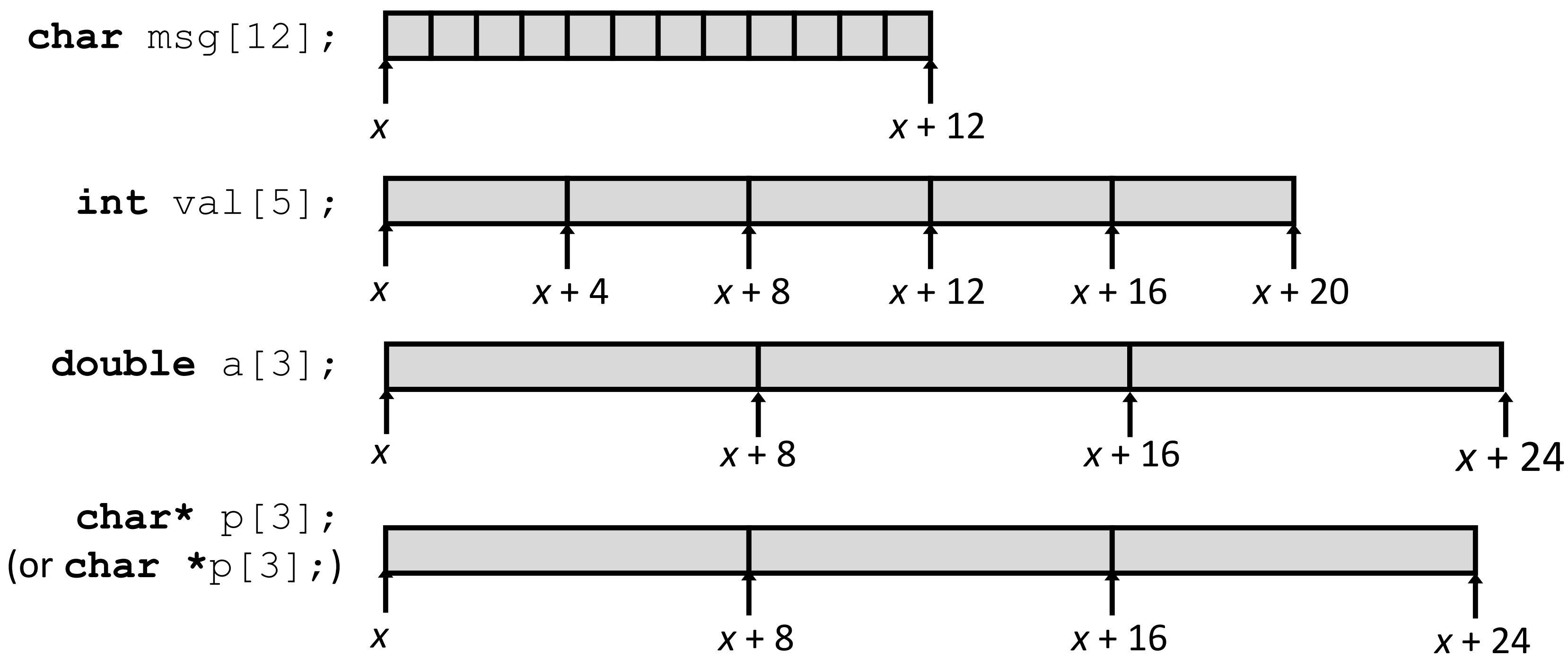
- How do we store information for other parts of the house of computing to access?
  - How do we represent data and what limitations exist?
  - What design decisions and priorities went into these encodings?

Data Structures in C

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

Array Allocation (Review)

- 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*`**)



Array Access (Review)

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

`int x[5];`

	3	7	1	9	5
index	0	1	2	3	4	"5"
a	a+4	a+8	a+12	a+16	a+20

- Reference Type Value

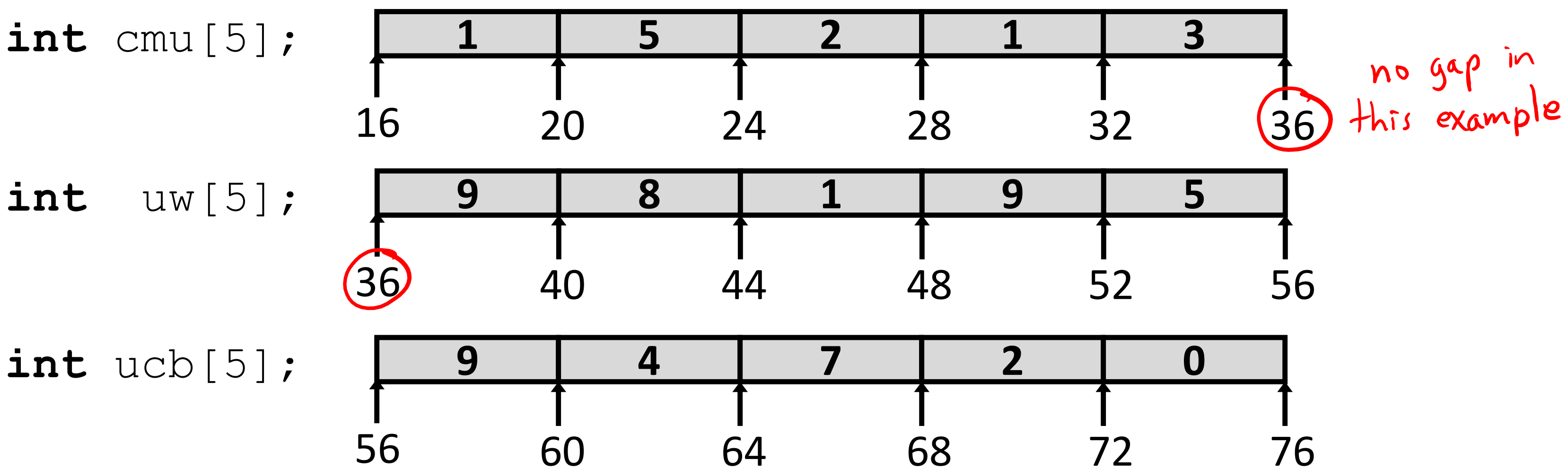
Reference	Type	Value
`x[4]`	**int**	5
`x`	**int***	a
`x+1` ← ptr arithmetic	**int***	a + 4
`&x[2]`	**int***	a + 8
`x[5]`	**int**	?? (whatever's in memory at addr `x+20`)
`*(x+1)`	**int**	7
`x+i`	**int***	a + 4*i

Array Example

brace-enclosed list initialization

```
// 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 };
```

20 B each



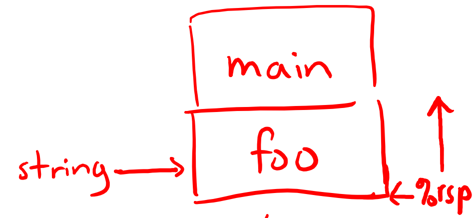
- Example arrays happened to be allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

(could have allocated variables in-between)

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 (0^{th}) 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 the scope is valid:

data will get overwritten by future stack frames

array is allocated on stack

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

BAD!

returns stack addr that is < %rsp

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

Really `int* ar` (%rdi can only fit 8 bytes)

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

Must explicitly pass the size!

Data Structures in C

- **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 }};
```

2D array

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

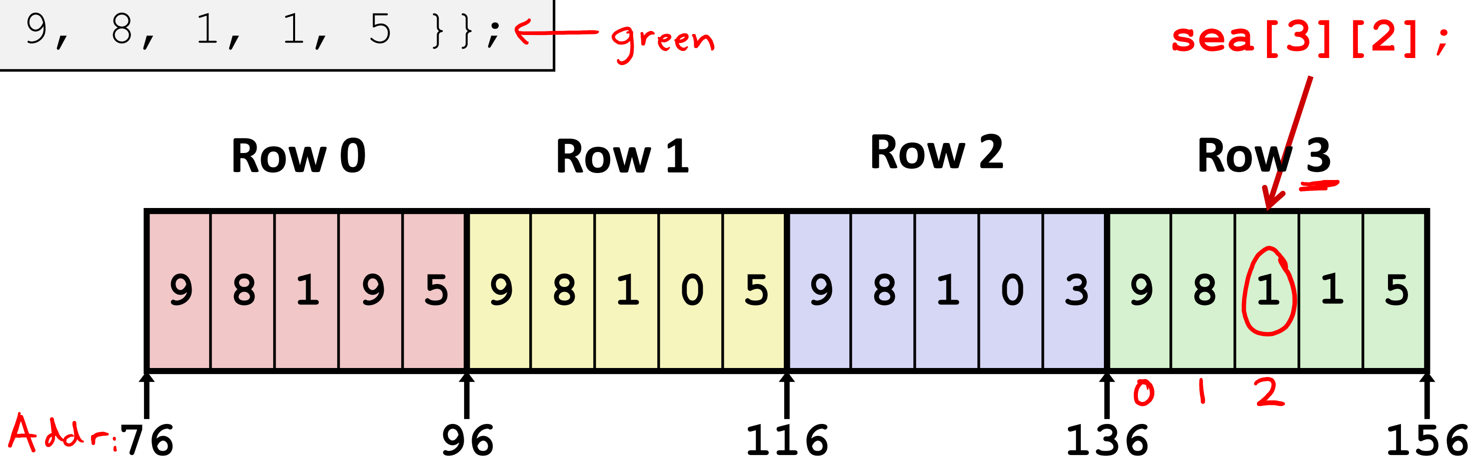
- What is the layout in memory?

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 }};
```

← red
← yellow
← blue
← green

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)

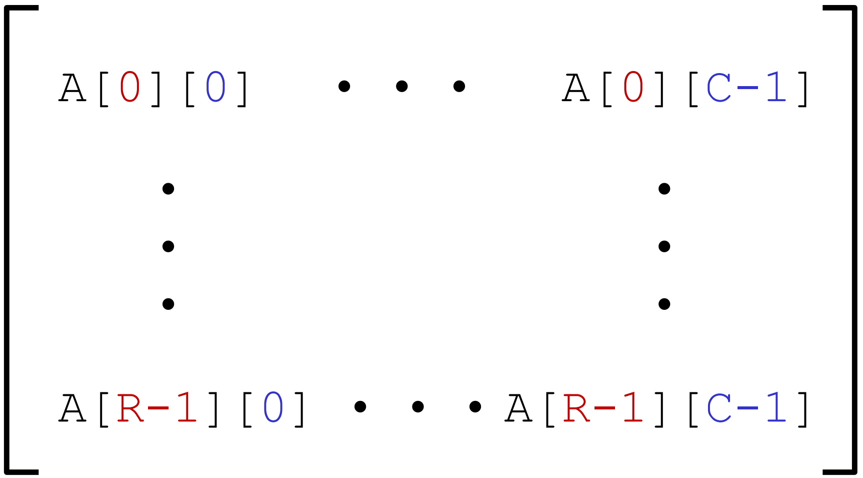
Two-Dimensional (Nested) Arrays

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

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

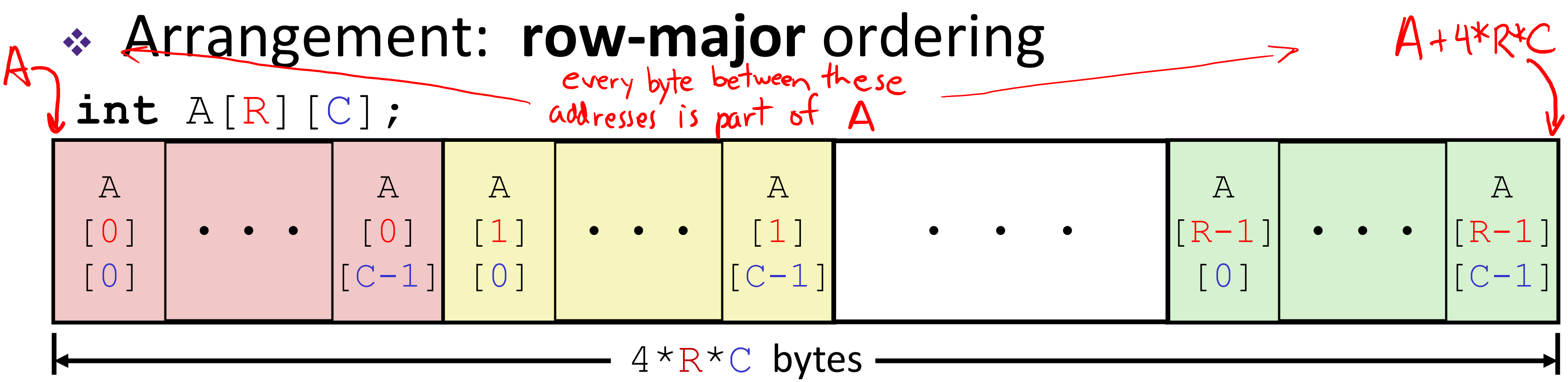
Two-Dimensional (Nested) Arrays

- 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

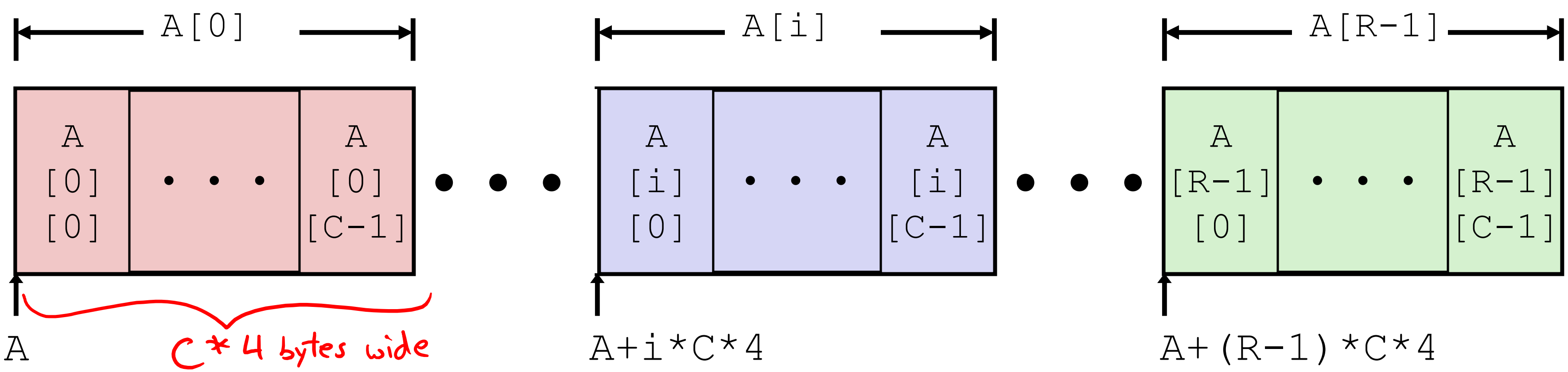
`int A[R][C];`



Nested Array Row Access

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

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



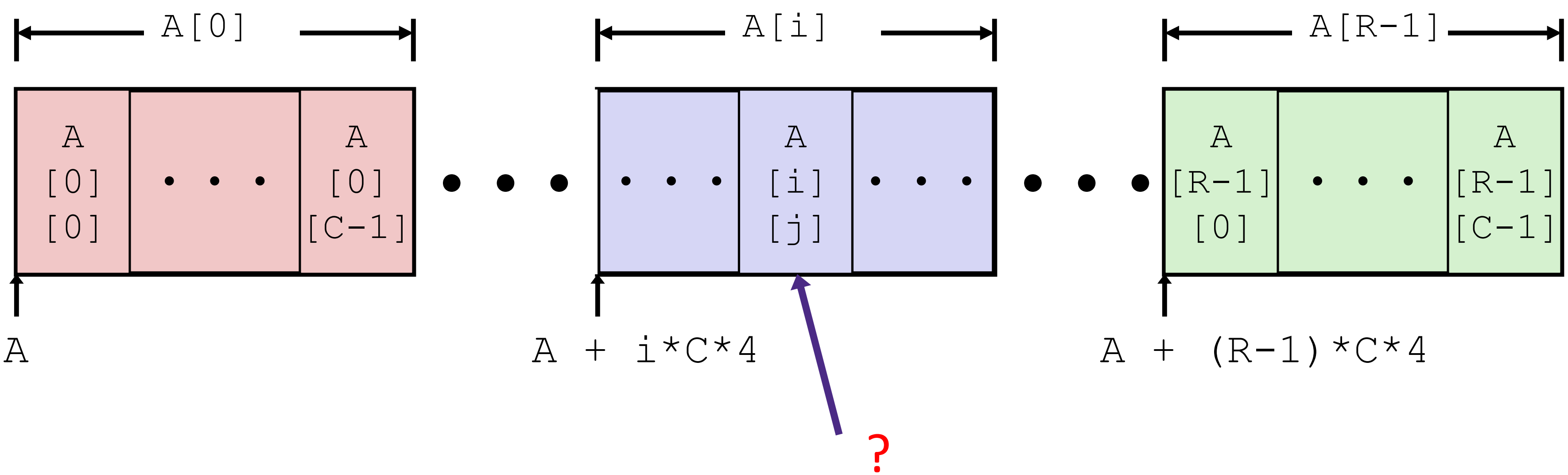
Nested Array Element Access

reminder: ar[j] = *(ar+j)

- Array Elements
  - `A[i][j]` is element of type **T**; let `sizeof(T)` = *t* bytes
  - Address of (`A[i]`)`[j]` is (A + i * C * sizeof(T)) + j * sizeof(T)

    address

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

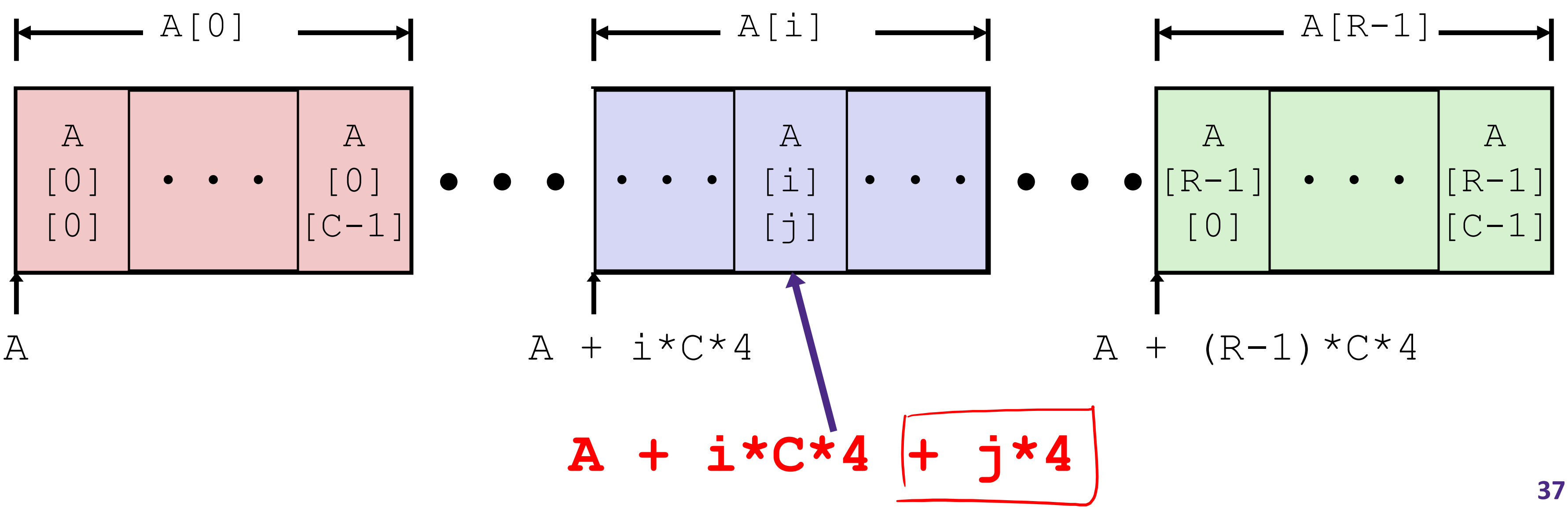


Nested Array Element Access

- Array Elements
  - `A[i][j]` is element of type **T**; let `sizeof(T)` = t bytes
  - Address of `A[i][j]` is

    `A + i*(C*t) + j*t = A + (i*C + j)*t`

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



Data Structures in C

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

Multilevel Array Example

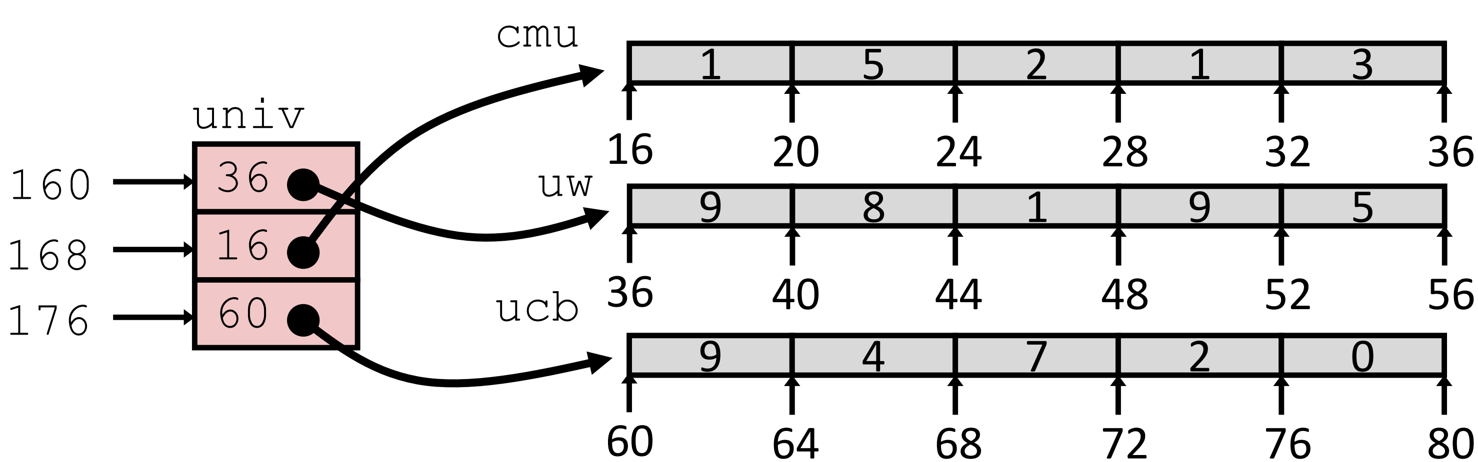
Note: this is how Java represents multidimensional arrays!

- Multilevel Array Declaration(s):

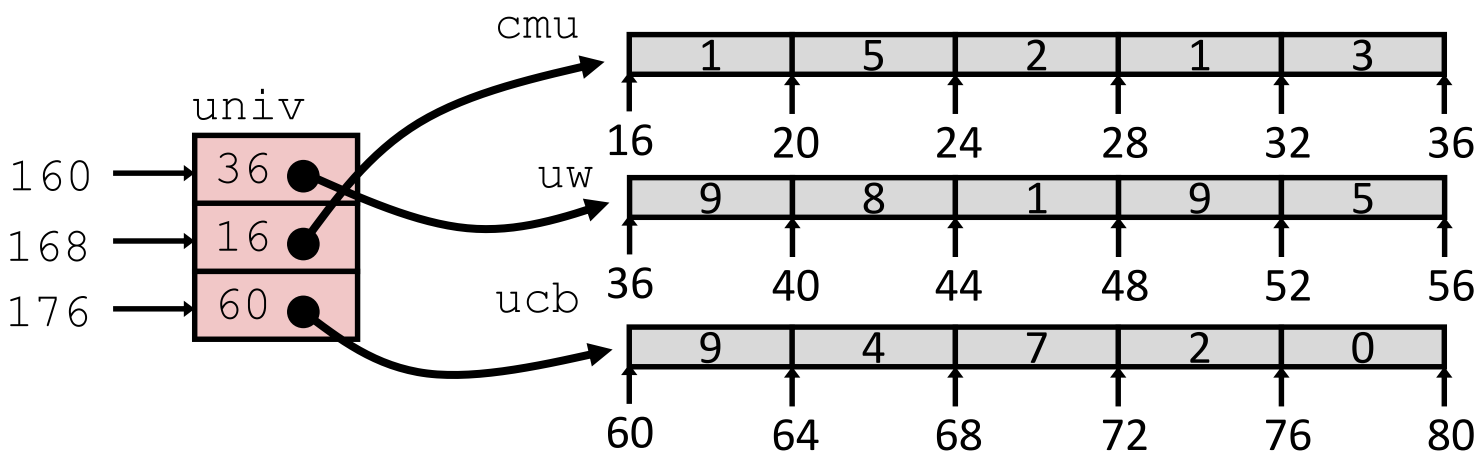
```
int cmu[5] = { 1, 5, 2, 1, 3 };
int  uw[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };
```

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

  - Variable `univ` denotes array of 3 pointer elements
  - Each pointer points to a separate array of `int`s
    - *Could* have inner arrays of different lengths!



Multilevel Array Element Access



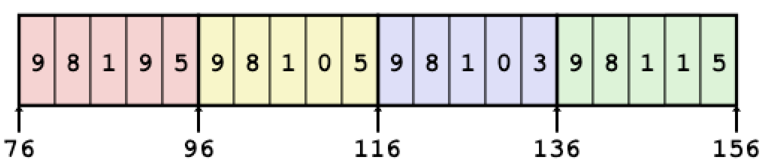

```
int get_univ_digit (int index, int digit) {
  return univ[index][digit];
}
```

- Mem[Mem[`univ+8*index`]+`4*digit`]
  - Must do **two memory reads**: (1) get pointer to row array, (2) access element within array

Array Element Accesses

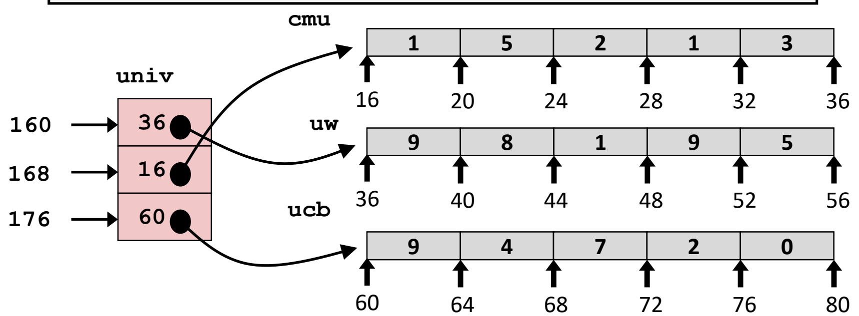
Multidimensional array:

```
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];
}
```



- Accesses *look* the same, but aren't:

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

- Memory layout is different:
  - One array declaration = one contiguous block of memory

Summary

- Building an executable
  - Multistep process: compiling, assembling, linking
  - Object code finished by linker using symbol and relocation tables to produce machine code (with finalized addresses)
  - Loader sets up initial memory from executable
- Arrays
  - Contiguous allocations of memory
  - No bounds checking (and no default initialization)
  - Can usually be treated like a pointer to first element
  - Multidimensional → array of arrays in one contiguous block
  - Multilevel → array of pointers to arrays
    - Each array/part separate in memory