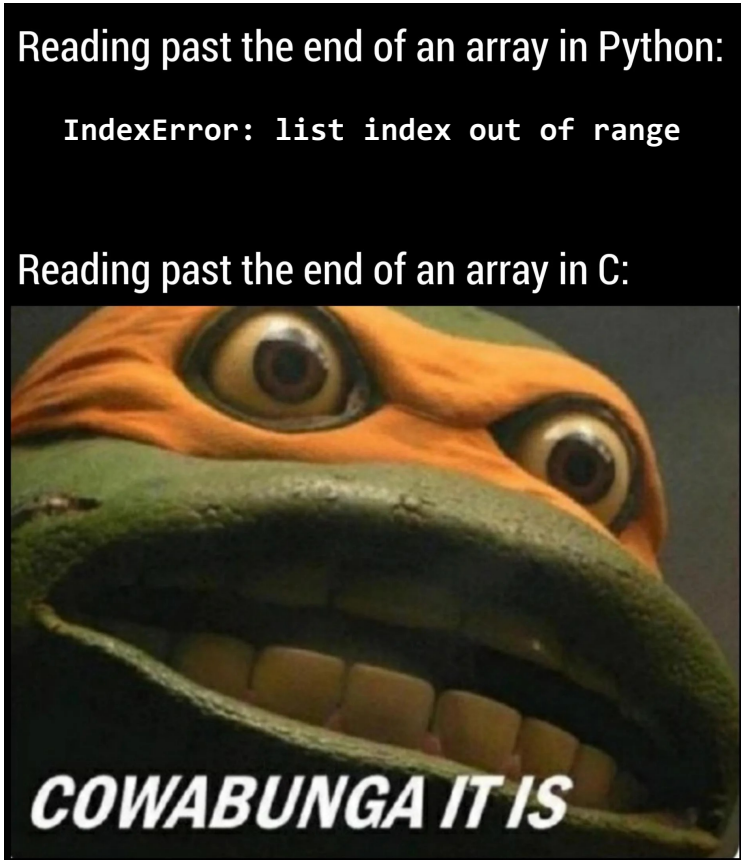
Executables & Arrays

CSE 351 Winter 2022

Instructor:
Sam Wolfson

Teaching Assistants:
Angela Xu
Anirudh Kumar
Catherine Guevara
Dara Stotland
Harrison Bay
Ian Hsiao
Kevin Wang
Mara Kirdani-Ryan
Nick Durand
Sanjana Sridhar



Relevant Course Information

- Lab 2 due Friday (2/4)
- hw13 due *next* Monday (2/7)
  - Based on the next two lectures, longer than normal

- Midterm (take home, 2/9—2/11)
  - Midterm review problems in section on Thursday
  - Make notes and use the midterm reference sheet
  - Form study groups and look at past exams!

Return to In-Person Instruction

- You should be prepared for the possibility of suddenly switching back to remote instruction (temporarily or indefinitely)
  - This class is designed to allow for asynchronous learning
- Face coverings required during all indoor, in-person interactions (lecture, section, in-person office hours)
  - Short breaks to sip water are okay
- Maintain physical distancing as much as possible
- You are allowed to attend any quiz section, provided there is enough seating/room
  - But please give priority to those officially enrolled ☺

Return to In-Person Instruction

- Some office hours will be **hybrid** and others virtual
  - Double-check calendar before coming
  - Hybrid office hours are a bit of an experiment, so please bear with us.
  - Zoom meeting links found in Zoom tab within Canvas
    - We encourage you to chat with other students in the lobby if TAs are in breakout rooms
  - *All* office hours will use a Google Sheets queuing system
  - Allen 3rd floor breakout limited to 19 people, please wait for "Enter" status:

Concept/Clarifications Question Queue (<5 mins)					Debugging Queue (>10 mins)				
Name	TA	Status	Question Description	Time Queued	Name	TA	Status	Question Description	Time Queued
Example 1		Done	Question about floating point encoding range.		Example 2		Done	Lab 5: running into a segfault in mm_malloc after reaching end of the heap.	
Leslie		Done	two's complement negation		Yutong		In Progress	Lab 1a segfault in selection sort	
Gabriela		Enter	bit shifting: logical vs arithmetic		Keysha		Enter	lab 1a withinSameBlock incorrect values	
Ishaan		Enter	endianness		Amadeus		Waiting	Lab 1a selectionSort edge case	

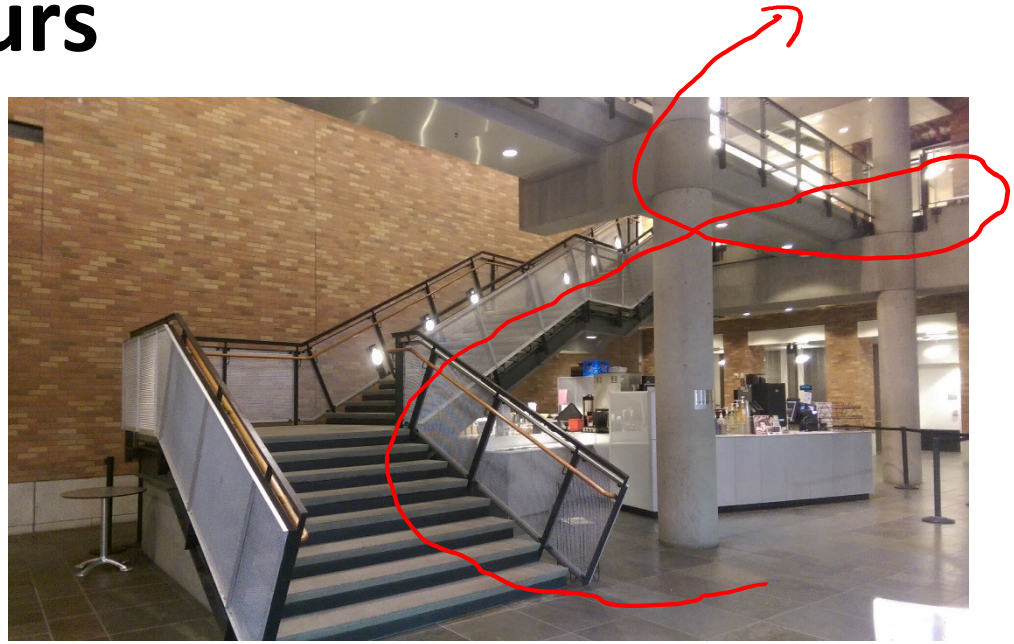
Return to In-Person Instruction

- Extenuating circumstances
  - Students (and staff) still face an extremely varied set of environments and circumstances
  - For formal accommodations, go through Disability Resources for Students (DRS)
  - We will try to be accommodating otherwise, but the earlier you reach out, the better

- Don't suffer in silence – talk to a staff member!
  - We have a 1-on-1 meeting request form

In-Person Office Hours

- Allen 3rd floor breakout
  - Up the stairs in the CSE Atrium (Allen Center, not Gates)



  - At the top of two flights, the open area with the whiteboard wall is the 3rd floor breakout!

x86-64 Linux Register Usage (Review)

%rax	Return value - Caller saved
%rbx	Callee saved
%rcx	Argument #4 - Caller saved
%rdx	Argument #3 - Caller saved
%rsi	Argument #2 - Caller saved
%rdi	Argument #1 - Caller saved
%rsp	Stack pointer
%rbp	Callee saved

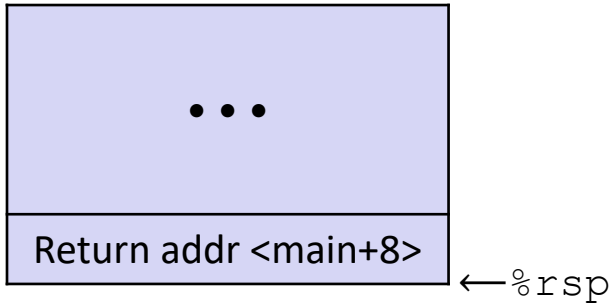
%r8	Argument #5 - Caller saved
%r9	Argument #6 - Caller saved
%r10	Caller saved
%r11	Caller Saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

Reminder: Procedure Call Example

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Initial Stack Structure



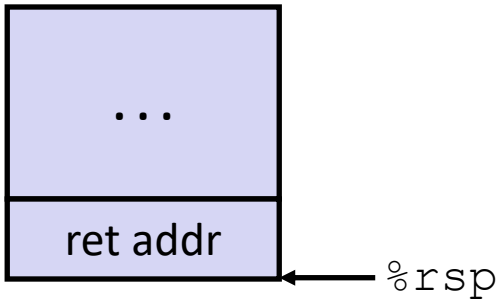
Callee-Saved Example (step 1)

```
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x + v2;
}
```

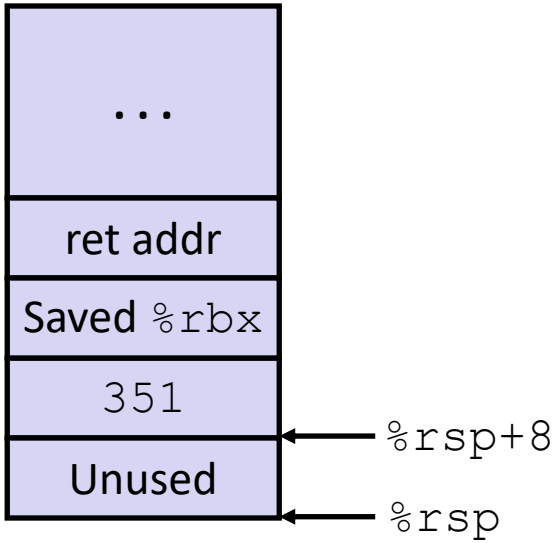
need to save x so we can use after call

```
call_incr2:
  pushq   %rbx      # store old value of rbx
  subq    $16, %rsp
  movq    %rdi, %rbx    # save x into rbx (callee-saved)
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    %rbx, %rax    # use rbx after call
  addq    $16, %rsp
  popq    %rbx
  ret
```

Initial Stack Structure



Resulting Stack Structure

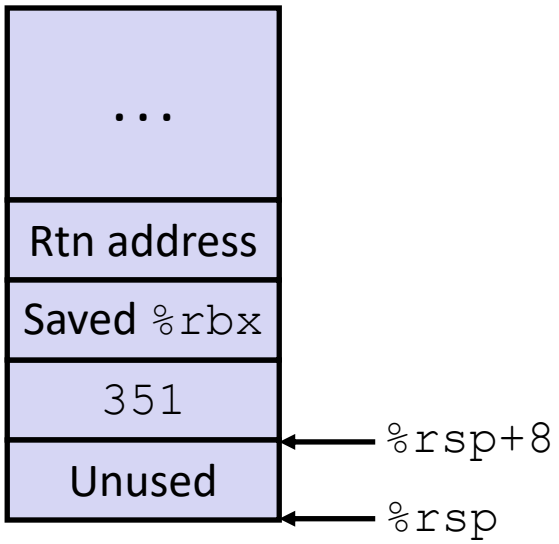


Callee-Saved Example (step 2)

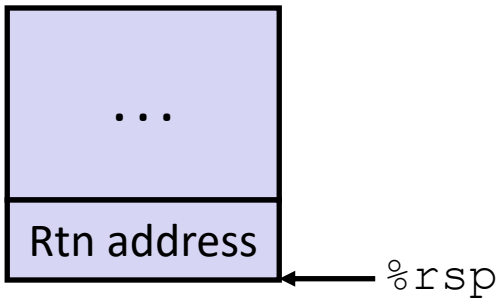
```
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x + v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx   # restore old value of rbx
  ret
```

Stack Structure



Pre-return Stack Structure



Why Caller *and* Callee Saved?

- We want *one* calling convention to simply separate implementation details between caller and callee

- In general, neither caller-save nor callee-save is "best":
  - If caller isn't using a register, caller-save is better
  - If callee doesn't need a register, callee-save is better
  - If "do need to save", callee-save generally makes smaller programs
    - Functions are called from multiple places

- So… "some of each" and compiler tries to "pick registers" that minimize amount of saving/restoring

Register Conventions Summary

- **Caller**-saved register values need to be pushed onto the stack before making a procedure call *only if the Caller needs that value later*
  - **Callee** may change those register values
- **Callee**-saved register values need to be pushed onto the stack *only if the Callee intends to use those registers*
  - **Caller** expects unchanged values in those registers

- Don't forget to restore/pop the values later!

Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- **Illustration of Recursion**

Recursive Function

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}
```

logical right shift

stop once all 1s shifted off

Value of LSB

shift off LSB and recurse

Counts the number of 1's in the binary representation of x.

Compiler Explorer:

https://godbolt.org/z/naP4ax

- Compiled with `-O1` instead of `-Og` for more natural instruction ordering

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  jne     .L8
  ret
.L8:
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
  ret
```

Recursive Function: Base Case

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

```
pcount_r:
  movl    $0, %eax        # prepare return value of 0
  testq   %rdi, %rdi      # jump to L8 if x & x != 0
  jne     .L8
  ret
.L8:
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
  ret
```

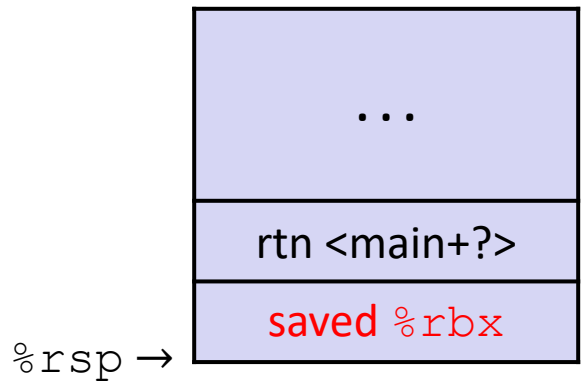
Recursive Function: Callee Register Save

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}
```

need x across procedure call

Register	Use(s)	Type
%rdi	x	Argument

The Stack



Need original value of x *after* recursive call to pcount_r.

"Save" by putting in %rbx (**callee** saved), but need to save old value of %rbx before you change it.

```
pcount_r:
   movl    $0, %eax
   testq   %rdi, %rdi
   jne     .L8
   ret
.L8:
   pushq   %rbx
   movq    %rdi, %rbx
   shrq    %rdi
   call    pcount_r
   andl    $1, %ebx
   addq    %rbx, %rax
   popq    %rbx
   ret
```

push before changing

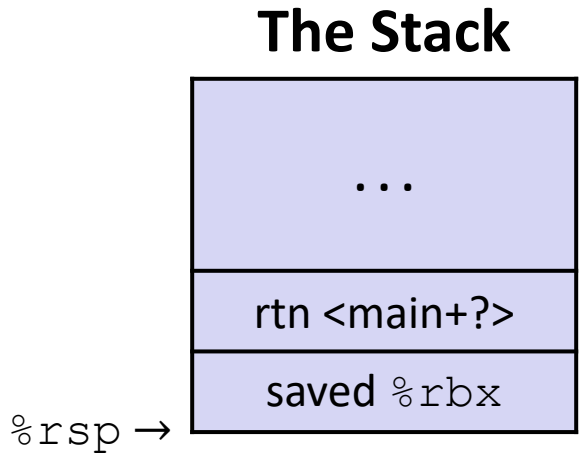
store x for this stack frame

pop/ restore before returning

Recursive Function: Call Setup

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rdi	x (new)	Argument
%rbx	x (old)	Callee saved

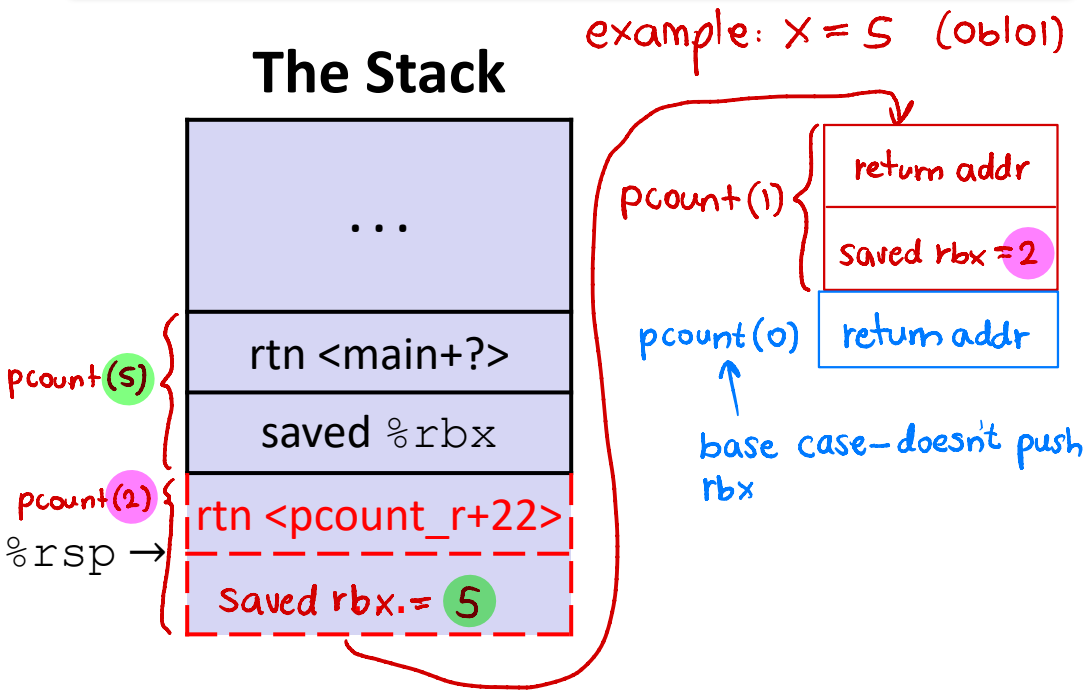



```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  jne     .L8
  ret
.L8:                    (implicit)
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    $1, %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
  ret
```

Recursive Function: Call

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rax	Recursive call return value	Return value
%rbx	x (old)	Callee saved



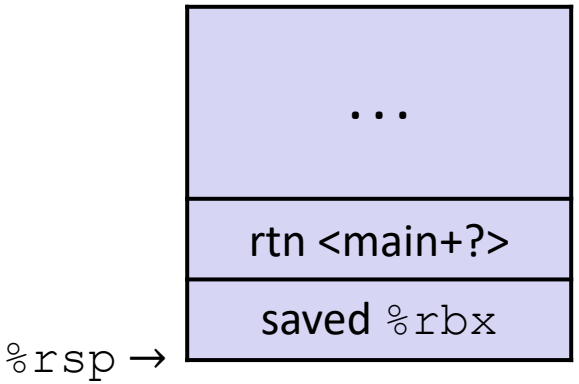

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  jne     .L8
  ret
.L8:
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
  ret
```

Recursive Function: Result

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rax	Return value	Return value
%rbx	x&1	Callee saved

The Stack




```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  jne     .L8
  ret
.L8:
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
  ret
```

x

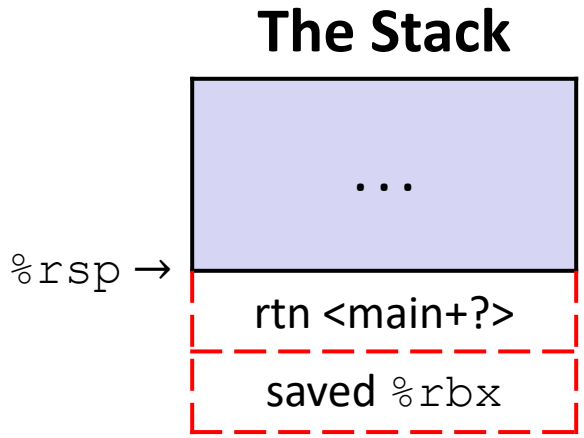
assumed the same across call

x

Recursive Function: Completion

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rax	Return value	Return value
%rbx	Previous %rbx value	Callee restored




```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  jne     .L8
  ret
.L8:
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
  ret
```

restore before return

Observations About Recursion

- "It Just Works!!"
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return address
  - Register saving conventions prevent one function call from corrupting another's data
    - Unless the code explicitly does so (*e.g.,* buffer overflow)
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out (LIFO)
- Also works for mutual recursion (P calls Q; Q calls P)

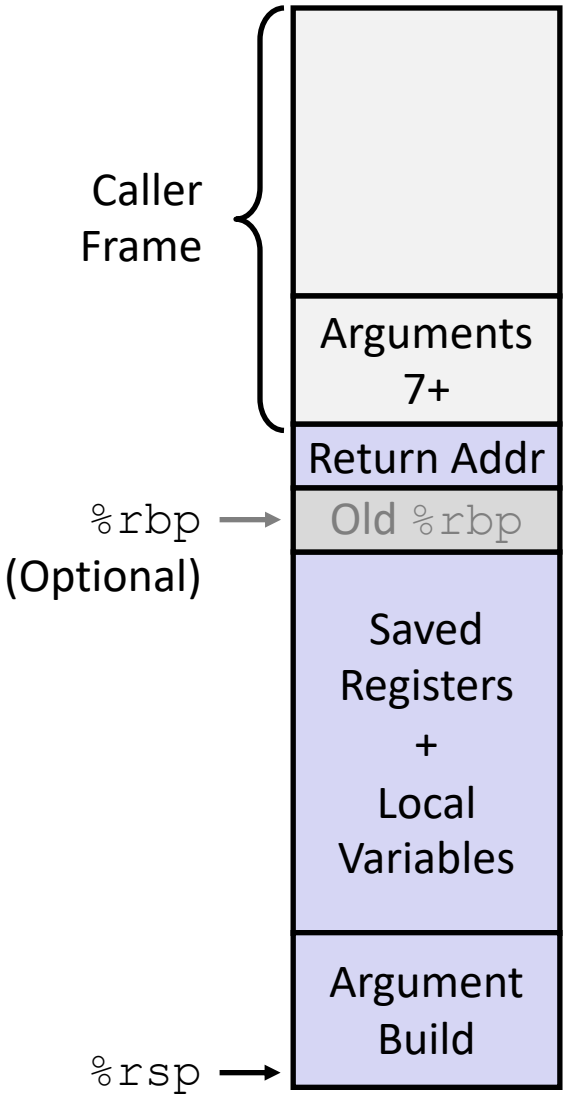
x86-64 Stack Frames

- Many x86-64 procedures have a minimal stack frame
  - Only return address is pushed onto the stack when procedure is called

- A procedure *needs* to grow its stack frame when it:
  - Has too many local variables to hold in **caller**-saved registers
  - Has local variables that are arrays or structs
  - Uses `&` to compute the address of a local variable
  - Calls another function that takes more than six arguments
  - Is using **caller**-saved registers and then calls a procedure
  - Modifies/uses **callee**-saved registers

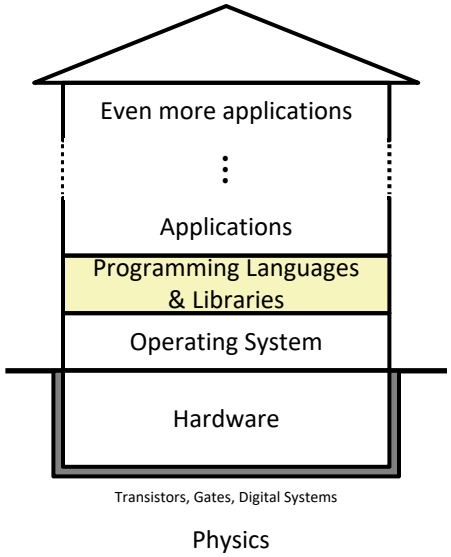
x86-64 Procedure Summary

- Important Points
  - Procedures are a combination of *instructions* and *conventions*
    - Conventions prevent functions from disrupting each other
  - Stack is the right data structure for procedure call/return
    - If P calls Q, then Q returns before P
  - Recursion handled by normal calling conventions
- Heavy use of registers
  - Faster than using memory
  - Use limited by data size and conventions
- Minimize use of the Stack



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

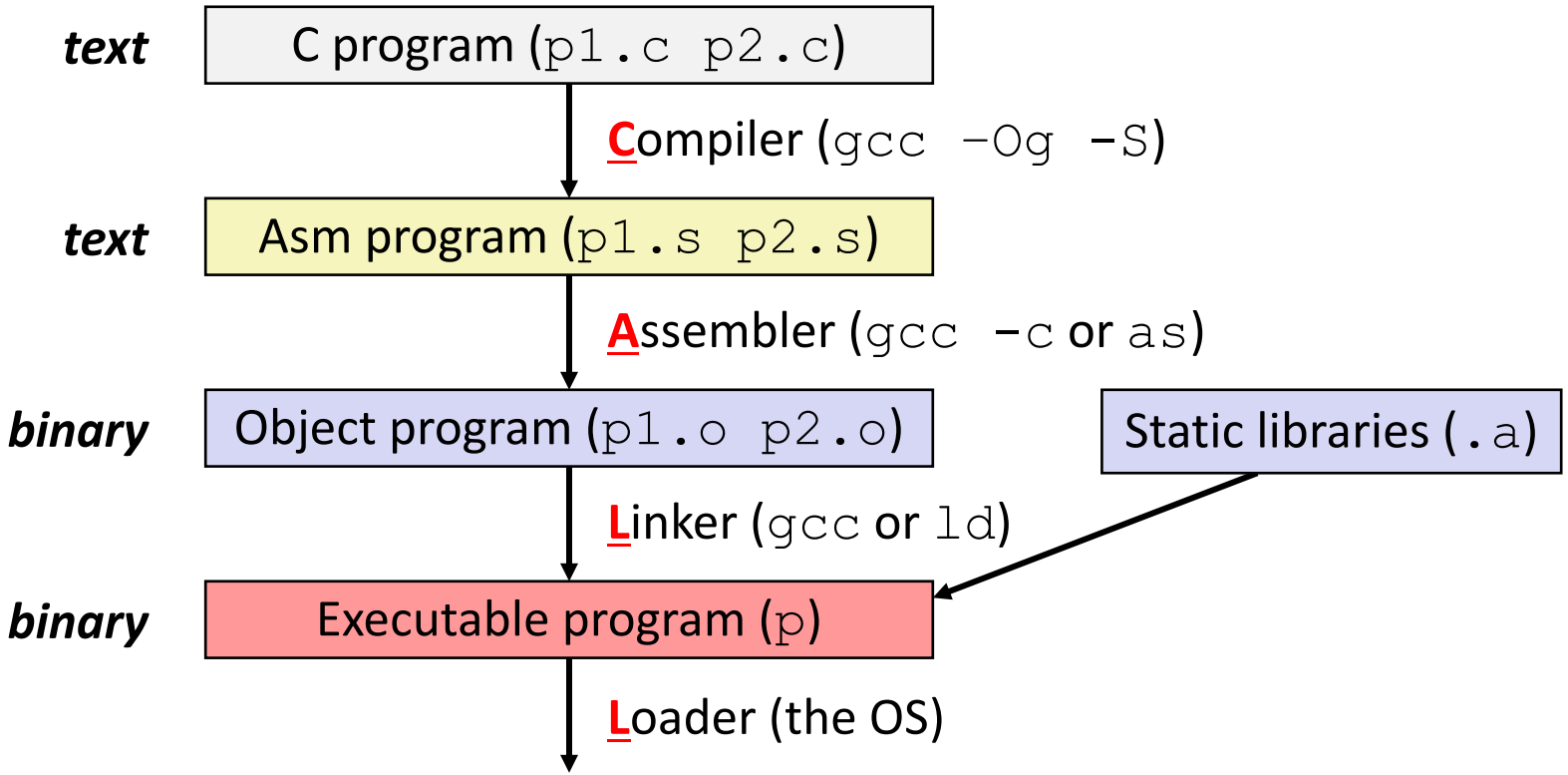
- Terminology:
  - CALL: compiler, assembler, linker, loader
  - Object file: symbol table, relocation table
  - Disassembly
  - Multidimensional arrays, row-major ordering
  - Multilevel arrays

- Questions from the Reading?

Building an Executable with C (Review)

- 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`

combine multiple source files into single executable



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! (CSE 401)
- 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;
}
```

emit assembly

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

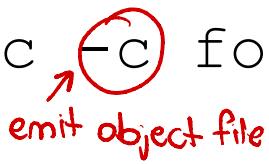
```
sumstore(long, long, long*):
  addq    %rdi, %rsi
  movq    %rsi, (%rdx)
  ret
```

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 e.g. global arrays, string constants)
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 (for GDB: compile with -g to include)

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

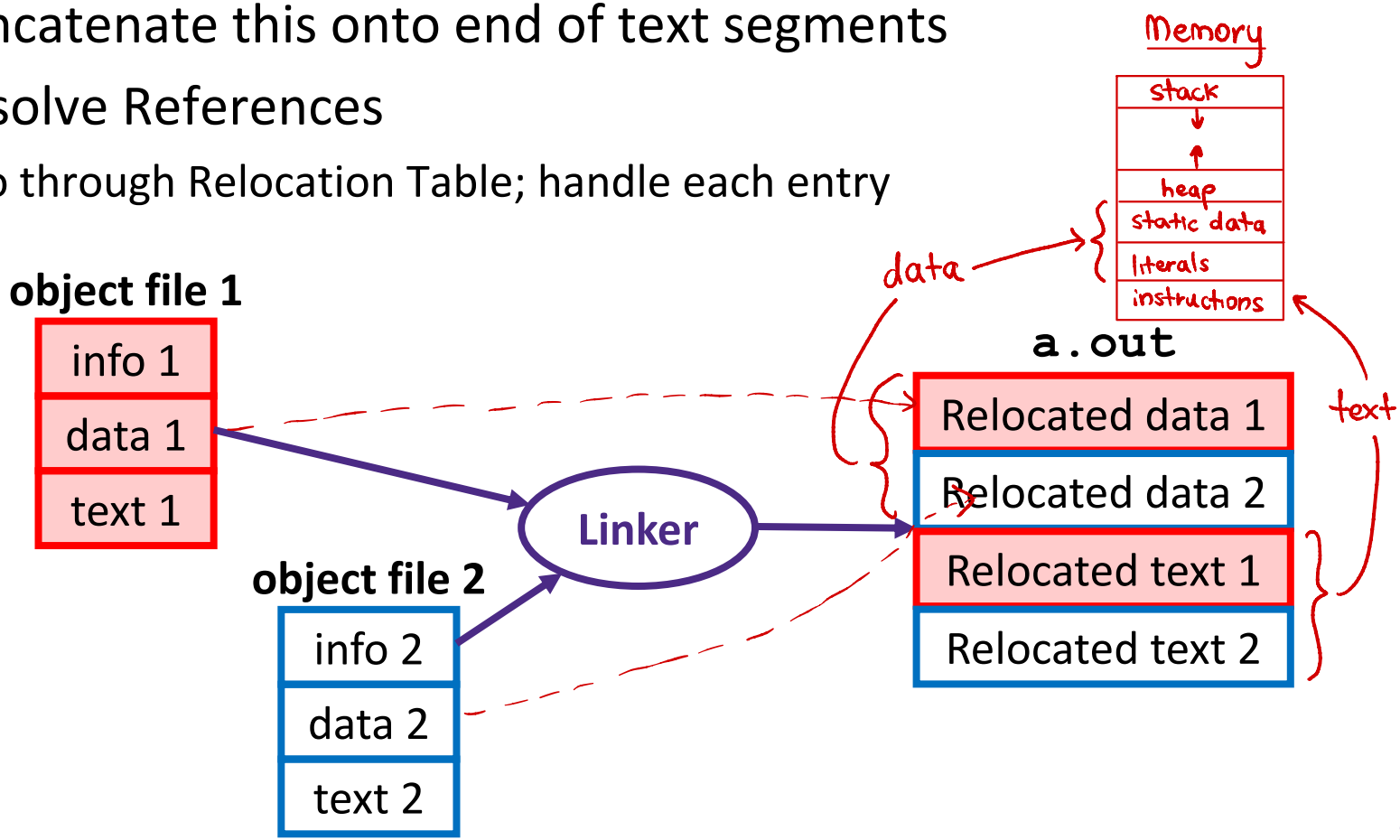
Linker (Review)

- **Input:** Object files (*e.g.*, ELF, COFF)
  - `foo.o`
- **Output:** executable binary program
  - `a.out` ← gcc 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>:
  400536:   48 01 fe          add     %rdi,%rsi
  400539:   48 89 32          mov     %rsi,(%rdx)
  40053c:   c3                retq
```

address of instruction | object code bytes | interpreted assembly

- **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?

Microsoft Word!

```
% 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

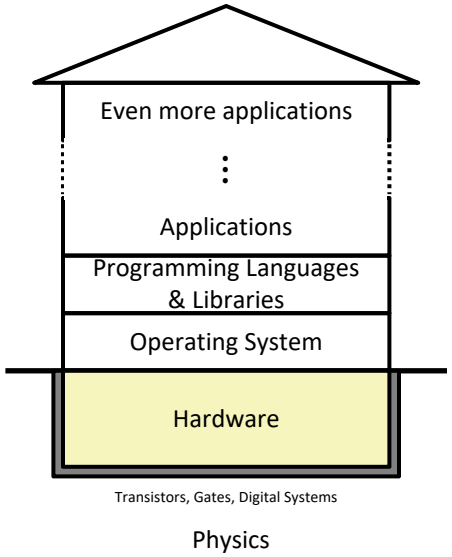
Loader (Review)

- **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



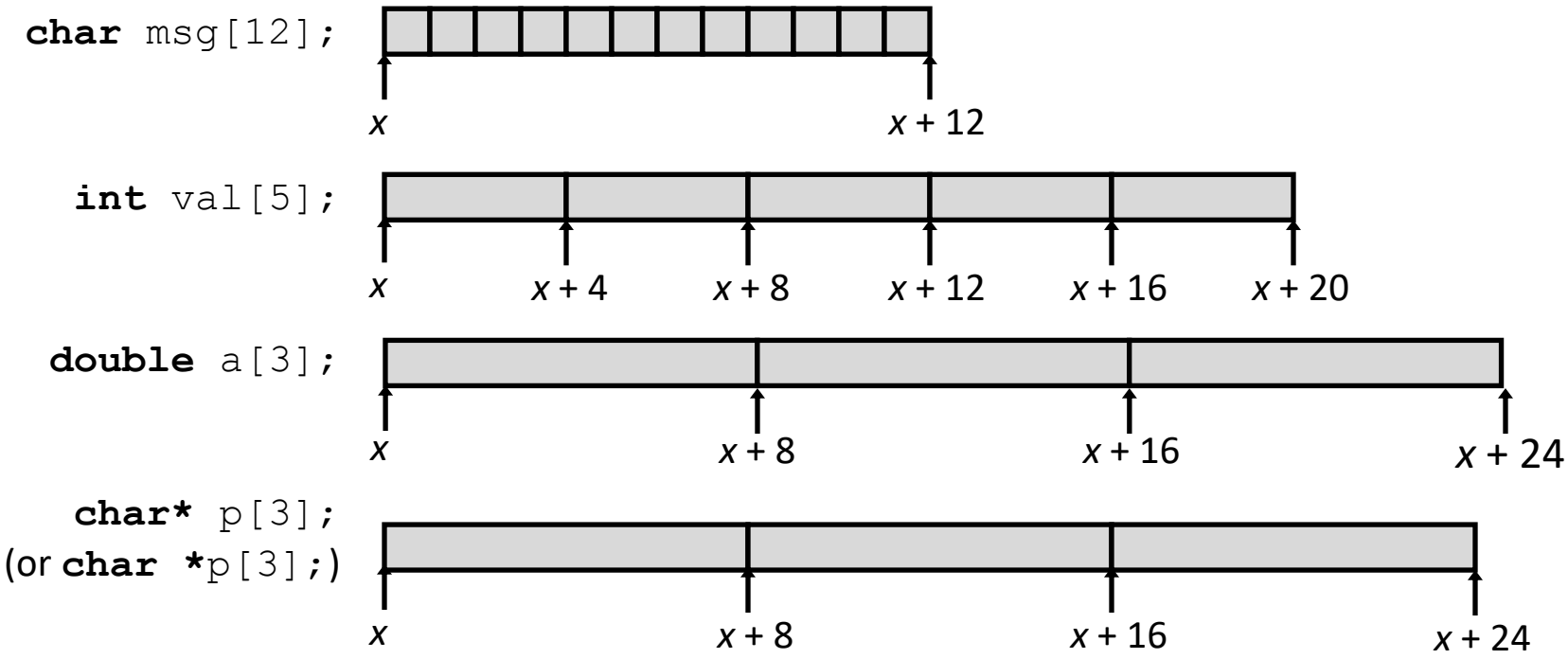
- How do we store information for other parts of the house of computing to access?
  - How do we represent data and what limitations exist?
  - What design decisions and priorities went into these encodings?

Data Structures in C

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

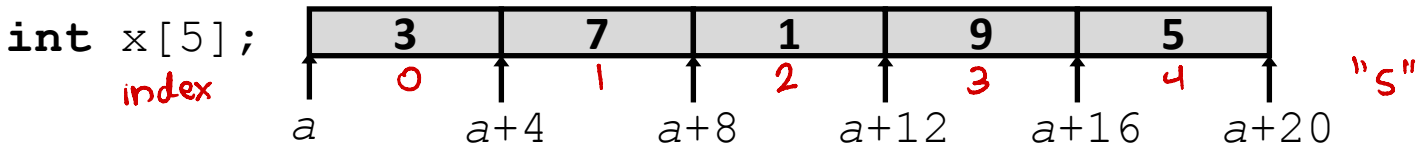
Array Allocation (Review)

- 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***)



- Reference / Type / Value

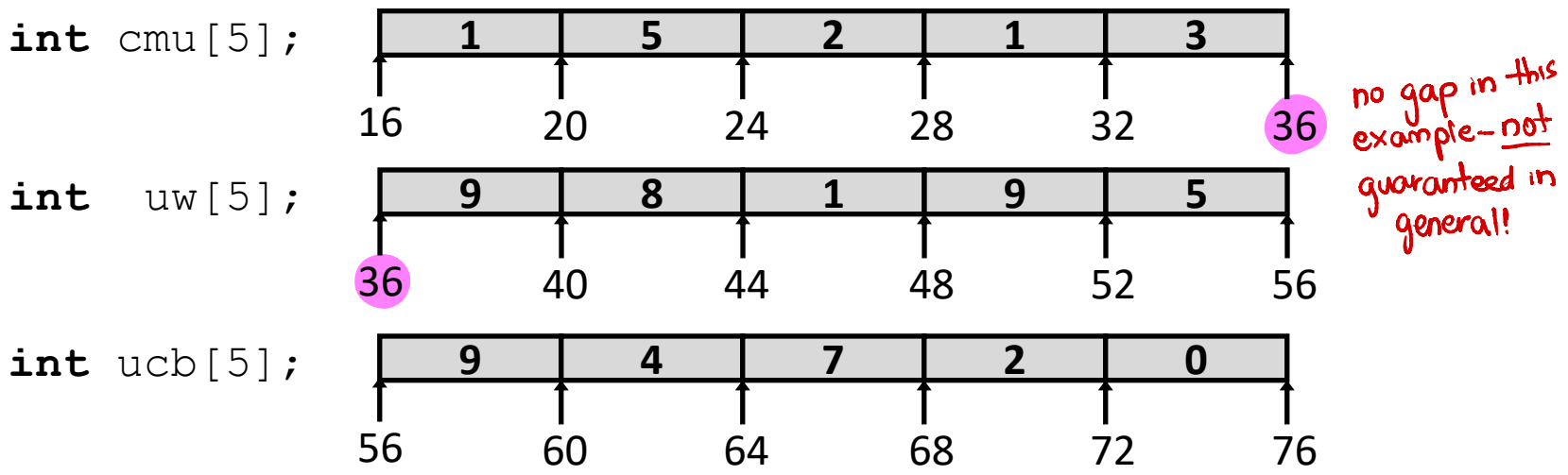
Reference	Type	Value
`x[4]`	**int**	`5`
`x`	**int***	`a`
`x+1` ← pointer arithmetic	**int***	`a + 4`
`&x[2]`	**int***	`a + 8`
`x[5]`	**int**	`??` (whatever's in memory at addr `x+20`)
`*(x+1)`	**int**	`7`
`x+i`	**int***	`a + 4*i`

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 };
```

20B each



no gap in this example – not guaranteed in general!

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

(other variables could have been allocated 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>`"

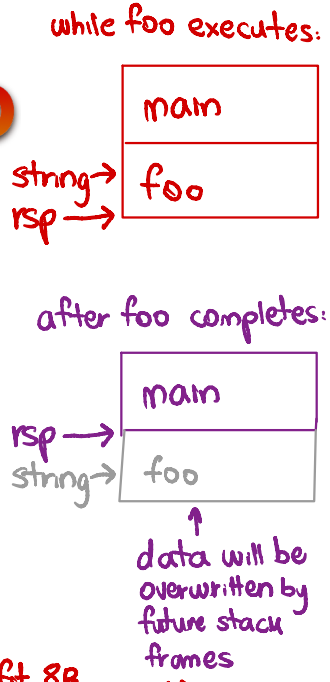
C Details: Arrays and Functions

- Declared arrays only allocated while the scope is valid:

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

allocates array on stack

returns stack address < rsp



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

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

Really `int* ar` (only fit 8B in %rdi)

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

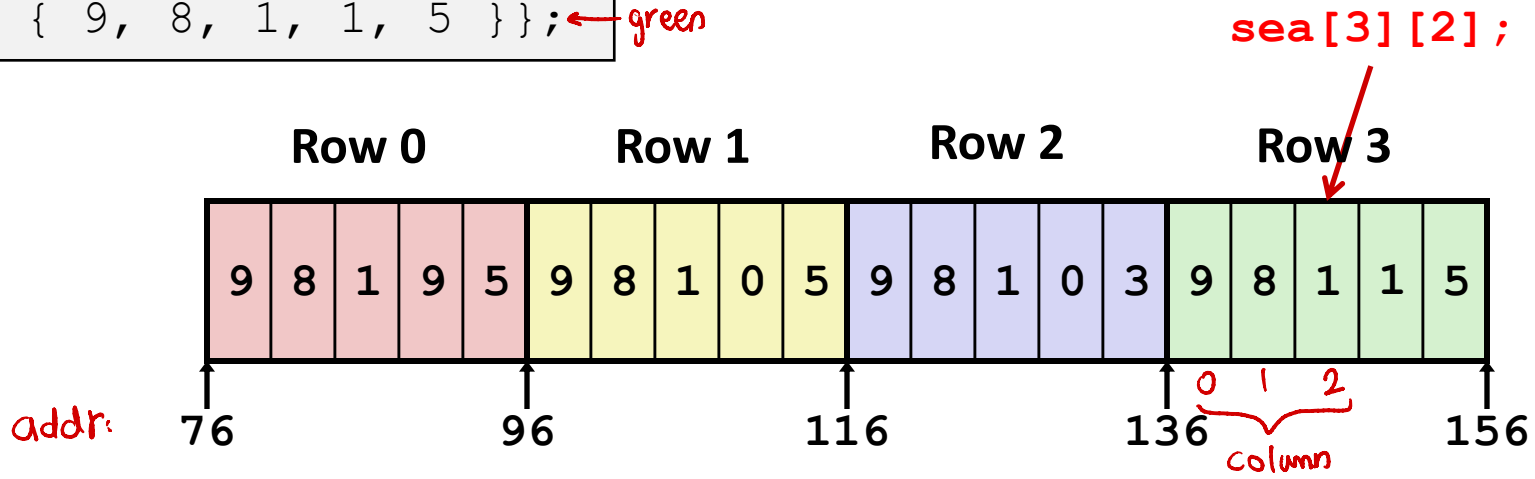
Remember, **T** `A[N]` is an array with elements of type **T**, with length `N`

- What is the layout in memory?

Nested Array Example

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

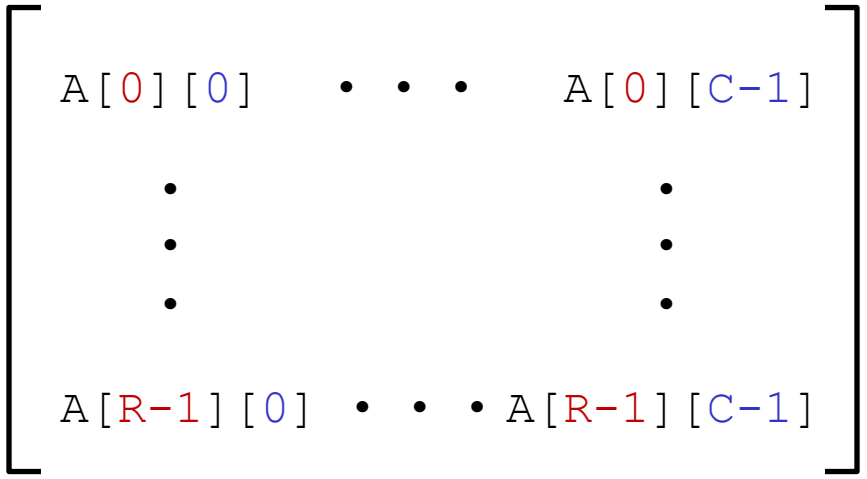
Remember, `T A[N]` is an array with elements of type `T`, with length `N`



- "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?

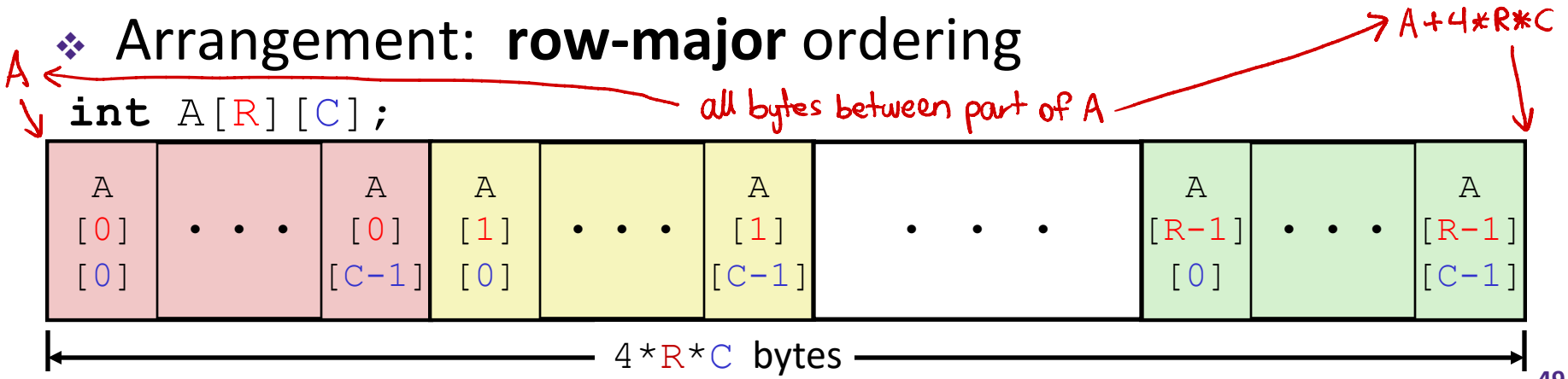


Two-Dimensional (Nested) Arrays

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

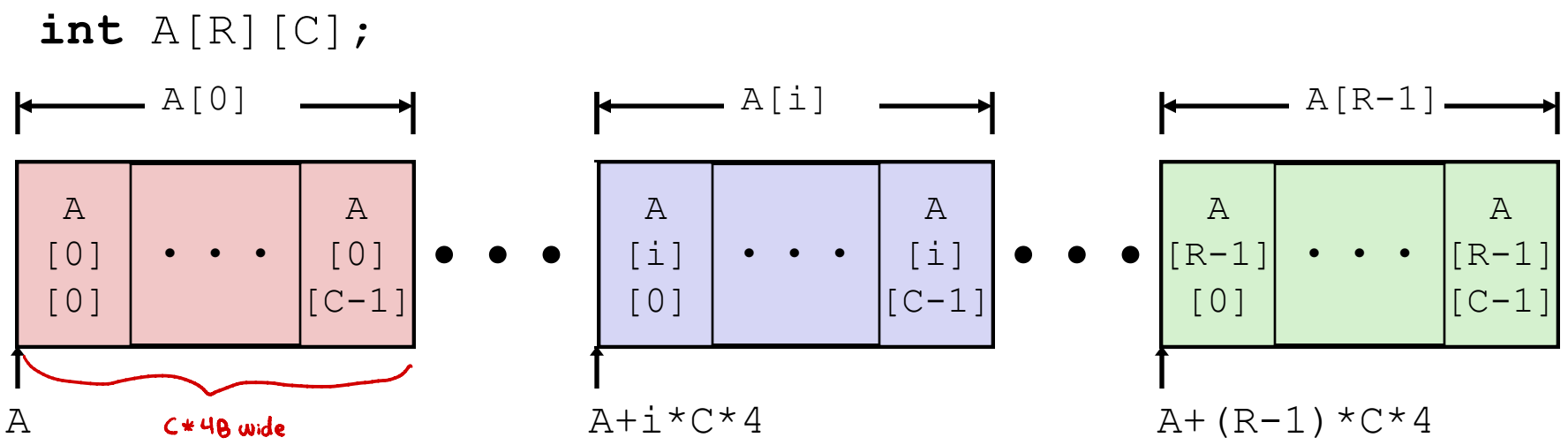
```
[ A[0][0]   . . .   A[0][C-1]
      .                 .
      .                 .
      .                 .
  A[R-1][0] . . . A[R-1][C-1] ]
```

- Array size:
  - R*C***sizeof**(**T**) bytes
- Arrangement: **row-major** ordering



Nested Array Row Access

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



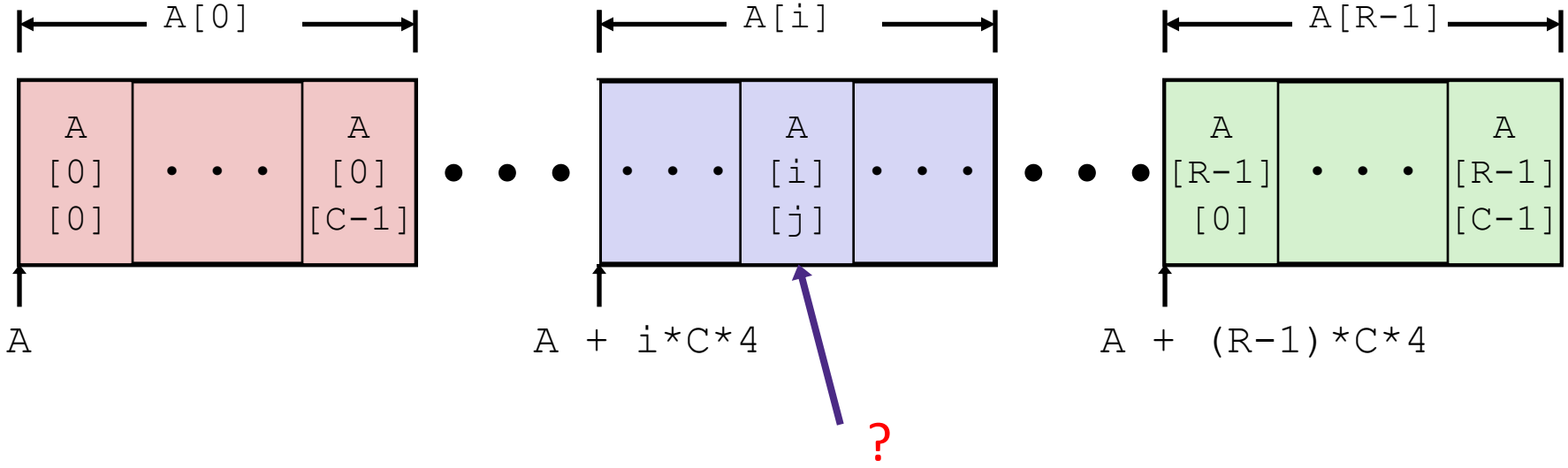
Nested Array Element Access

- Array Elements

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

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

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

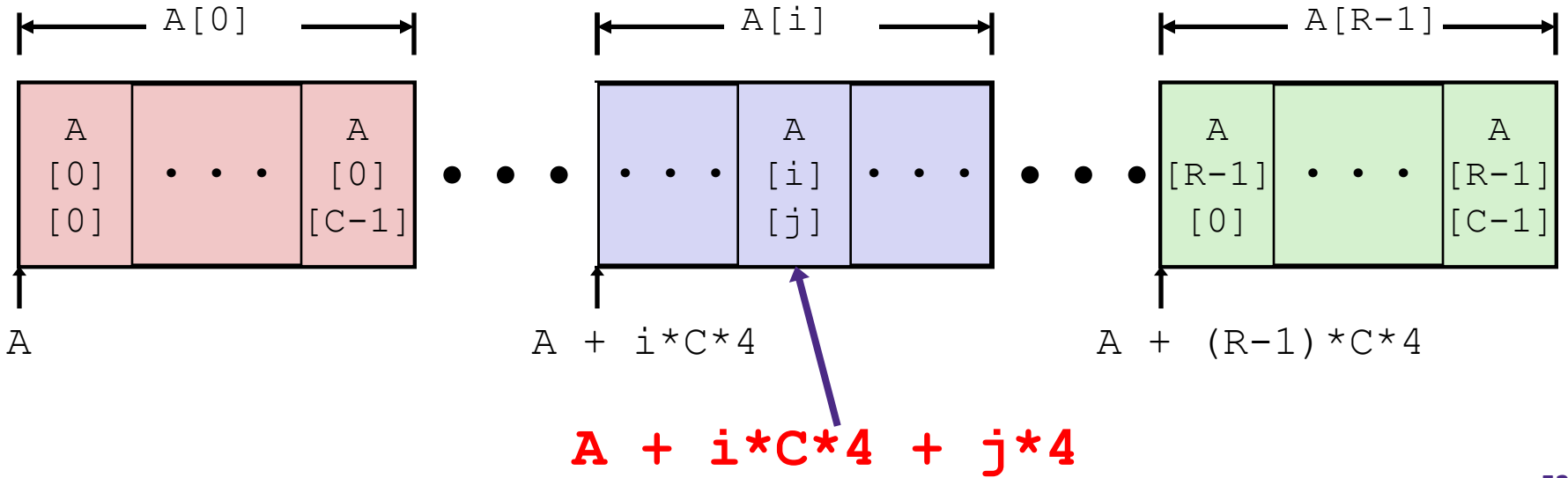


Nested Array Element Access

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

$$\texttt{A} + \texttt{i*(C*}t\texttt{)} + \texttt{j*t} = \texttt{A} + \texttt{(i*C} + \texttt{j)*}t$$

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



Data Structures in C

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

Multilevel Array Example

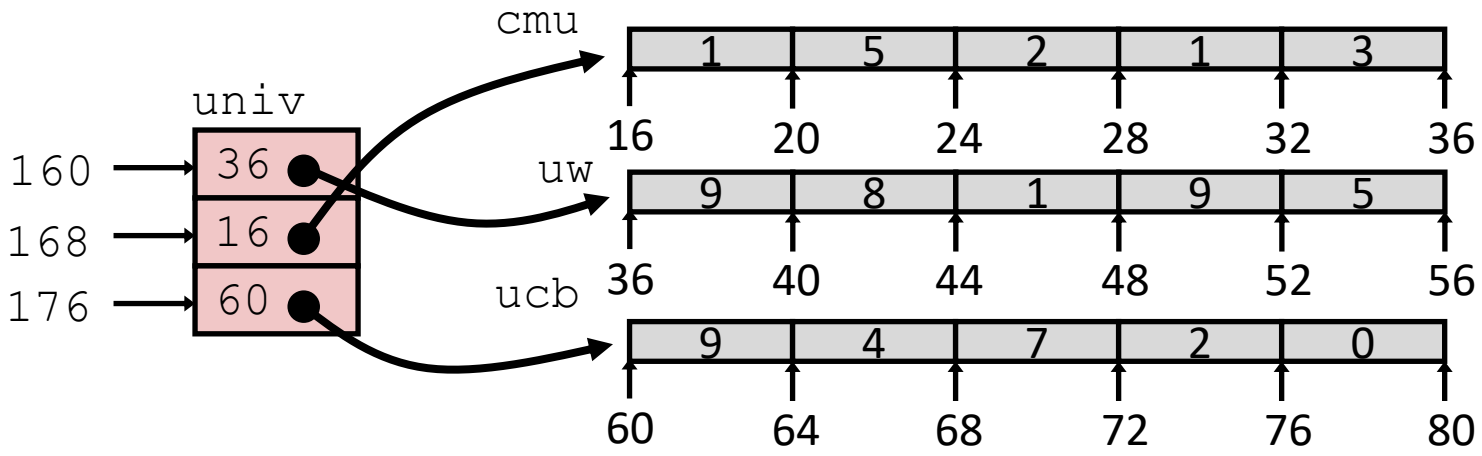
Note: this is how Java represents multidimensional arrays!

- Multilevel Array Declaration(s):

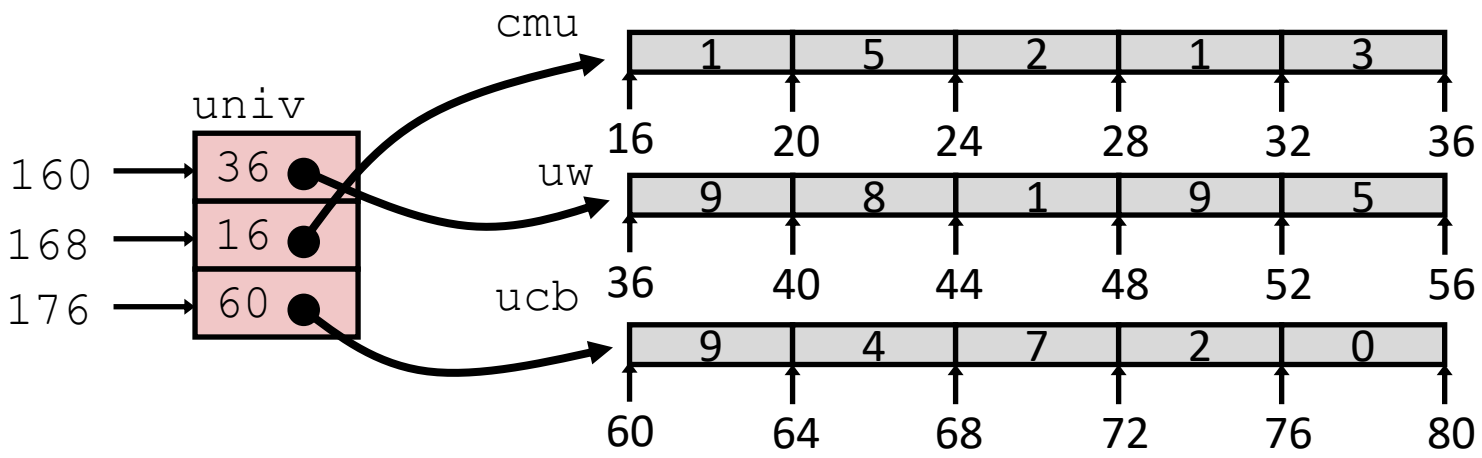
```
int cmu[5] = { 1, 5, 2, 1, 3 };
int  uw[5] = { 9, 8, 1, 9, 5 };
int ucb[5] = { 9, 4, 7, 2, 0 };
```

```
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



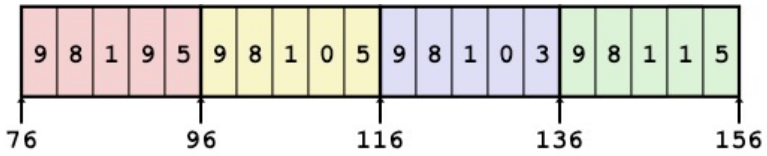

```
int get_univ_digit (int index, int digit) {
  return univ[index][digit];
}
```

- 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

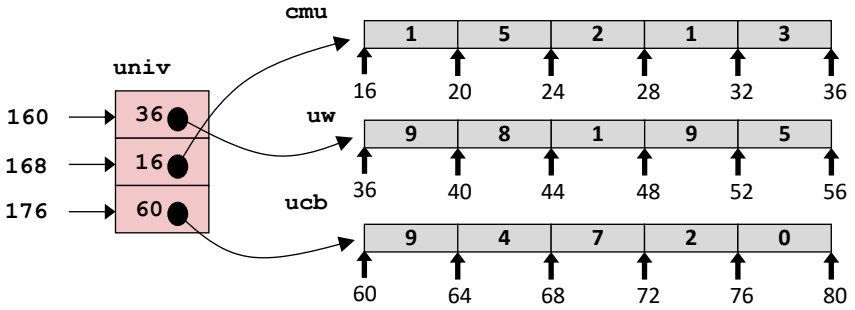
Multidimensional array:

```
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