

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Arrays

CSE 351 Autumn 2017

**Instructor:**  
Justin Hsia

**Teaching Assistants:**  
Lucas Wotton  
Michael Zhang  
Parker DeWilde  
Ryan Wong  
Sam Gehman  
Sam Wolfson  
Savanna Yee  
Vinny Palaniappan

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Administrivia

- ❖ Lab 2 due Friday (10/27)
- ❖ Homework 3 due next Friday (11/3)
- ❖ Lab 3 released next Wednesday (11/1)
- ❖ **Midterm** (Monday 10/30, 5:10-6:20pm)
  - 70 minute exam (longer than usual)
  - ID check, so come at 5pm
  - Midterm details Piazza post: [@163](#)
  - **Review session:** 5:30-7:30pm on Friday (10/27) in EEB 105
  - Take a look at midterm review packet before section

2

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Roadmap

**C:**

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

**Java:**

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
c.getMPG();
```

Memory & data  
Integers & floats  
x86 assembly  
Procedures & stacks  
Executables  
**Arrays & structs**  
Memory & caches  
Processes  
Virtual memory  
Memory allocation  
Java vs. C


**Assembly language:**

```
get_mpg:
    pushq   %rbp
    movq   %rsp, %rbp
    ...
    popq   %rbp
    ret
```


**Machine code:**

```
0111010000011000
100011010000010000000010
1000100111000010
11000001111101000011111
```

**OS:**



**Computer system:**



3

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Data Structures in Assembly

- ❖ **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- ❖ **Structs**
  - Alignment
- ❖ **Unions**

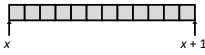
4

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

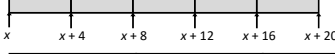
## Array Allocation

- ❖ **Basic Principle**
  - $T A[N]; \rightarrow$  array of data type  $T$  and length  $N$
  - *Contiguously* allocated region of  $N * \text{sizeof}(T)$  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];**)



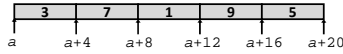
5

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Array Access

- ❖ **Basic Principle**
  - $T A[N]; \rightarrow$  array of data type  $T$  and length  $N$
  - Identifier  $A$  returns address of array (type  $T^*$ )

**int x[5];**



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

6

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Array Example

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

initialization

♦ typedef: Declaration "zip\_dig uw" equivalent to "int uw[5]"

7

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Array Example

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

zip_dig cmu:	1	5	2	1	3	
	16	20	24	28	32	36
zip_dig uw:	9	8	1	9	5	
	36	40	44	48	52	56
zip_dig ucb:	9	4	7	2	0	
	56	60	64	68	72	76

♦ Example arrays happened to be allocated in successive 20 byte blocks

- Not guaranteed to happen in general

8

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Array Accessing Example

```
typedef int zip_dig[5];
zip_dig uw;
int get_digit(zip_dig z, 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 memory reference (%rdi,%rsi,4)

9

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Referencing Examples

zip_dig cmu:	1	5	2	1	3	
	16	20	24	28	32	36
zip_dig uw:	9	8	1	9	5	
	36	40	44	48	52	56
zip_dig ucb:	9	4	7	2	0	
	56	60	64	68	72	76

Reference	Address	Value	Guaranteed?
uw[3]			
uw[6]			
uw[-1]			
cmu[15]			

♦ No bounds checking

♦ Example arrays happened to be allocated in successive 20 byte blocks

- Not guaranteed to happen in general

10

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Array Loop Example

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

$zi = 10 \cdot 0 + 9 = 9$   
 $zi = 10 \cdot 9 + 8 = 98$   
 $zi = 10 \cdot 98 + 1 = 981$   
 $zi = 10 \cdot 981 + 9 = 9819$   
 $zi = 10 \cdot 9819 + 5 = 98195$

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

11

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Array Loop Example

Original:

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

address just past 5<sup>th</sup> digit

Increments by 4 (size of int)

12

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Array Loop Implementation gcc with -O1

- Registers:
  - %rdi z
  - %rax zi
  - %rcx zend
- Computations
  - 
  -

```
int zd2int(zip_dig z)
{
    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 #
```

13

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## 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 variable looks like a pointer to the first (0<sup>th</sup>) element
  - ar[0] same as \*ar; ar[2] same as \*(ar+2)
- An array variable is read-only (no assignment)
  - Cannot use "ar = <anything>"

14

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## C Details: Arrays and Functions

- Declared arrays only allocated while the scope is valid:
 

```
char* foo() {
    char string[32]; ... BAD!
    return string;
}
```
- 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  
Must explicitly pass the size!

15

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Data Structures in Assembly

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structs
  - Alignment
- Unions

16

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Nested Array Example

```
typedef int zip_dig[5];
```

```
zip_dig sea[4] =
{{ 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?

same as:  
int sea[4][5];

17

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Nested Array Example

```
typedef int zip_dig[5];
```

```
zip_dig sea[4] =
{{ 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

sea[3][2];

Row 0	Row 1	Row 2	Row 3
9 8 1 9 5	9 8 1 0 5	9 8 1 0 3	9 8 1 1 5
76	96	116	136
			156

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

18

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Two-Dimensional (Nested) Arrays

- Declaration:  $\mathbf{T} \mathbf{A}[\mathbf{R}][\mathbf{C}];$ 

$A[0][0]$	...	$A[0][C-1]$
$\vdots$		$\vdots$
$A[R-1][0]$	...	$A[R-1][C-1]$

  - 2D array of data type  $\mathbf{T}$
  - $\mathbf{R}$  rows,  $\mathbf{C}$  columns
  - Each element requires  $\mathbf{sizeof}(\mathbf{T})$  bytes
- Array size?

19

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Two-Dimensional (Nested) Arrays

- Declaration:  $\mathbf{T} \mathbf{A}[\mathbf{R}][\mathbf{C}];$ 

$A[0][0]$	...	$A[0][C-1]$
$\vdots$		$\vdots$
$A[R-1][0]$	...	$A[R-1][C-1]$

  - 2D array of data type  $\mathbf{T}$
  - $\mathbf{R}$  rows,  $\mathbf{C}$  columns
  - Each element requires  $\mathbf{sizeof}(\mathbf{T})$  bytes
- Array size:
  - $\mathbf{R} * \mathbf{C} * \mathbf{sizeof}(\mathbf{T})$  bytes
- Arrangement: **row-major ordering**

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

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

4 \* R \* C bytes

20

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Nested Array Row Access

- Row vectors
  - Given  $\mathbf{T} \mathbf{A}[\mathbf{R}][\mathbf{C}]$ ,
    - $A[i]$  is an array of  $\mathbf{C}$  elements ("row  $i$ ")
    - Each element of type  $\mathbf{T}$  requires  $\mathbf{K}$  bytes
    - $A$  is address of array
    - Starting address of row  $i = A + i * (\mathbf{C} * \mathbf{K})$

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

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

A                      A+i\*C\*4                      A+(R-1)\*C\*4

21

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## 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  $\mathit{sea}[\mathit{index}]$ ?
- What is its starting address?

```
get_sea_zip(int):
movslq %edi, %rdi
leaq (%rdi,%rdi,4), %rdx
leaq 0,(,%rdx,4), %rax
addq %sea, %rax
ret
sea:
.long 9
.long 8
.long 1
.long 9
.long 5
.long 9
.long 8
...
```

22

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## 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  $\mathit{sea}[\mathit{index}]$ ?
- What is its starting address?

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

**Translation?**

23

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## 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
  - $\mathit{sea}[\mathit{index}]$  is array of 5 ints
  - Starting address =  $\mathit{sea} + 20 * \mathit{index}$
- Assembly Code
  - Computes and returns address
  - Compute as:  $\mathit{sea} + 4 * (\mathit{index} + 4 * \mathit{index}) = \mathit{sea} + 20 * \mathit{index}$

24

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## 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];
```

25

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## 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];
```

26

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## 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
  - Computes address as:  $sea + ((index*4*5) + digit)*4$
  - movl performs memory reference

27

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Strange Referencing Examples

```
typedef int zip_dig[5];
```

Reference	Address	Value	Guaranteed?
sea[3][3]	136	1	Yes
sea[2][5]	136	9	No
sea[2][-1]	136	9	No
sea[4][-1]	156	5	No
sea[0][19]	76	9	No
sea[0][-1]	76	9	No

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

28

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Data Structures in Assembly

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
    - Multi-level
- Structs
  - Alignment
- Unions

29

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Multi-Level Array Example

### Multi-Level 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};
```

### 2D Array Declaration:

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

Is a multi-level array the same thing as a 2D array? **NO**

One array declaration = one contiguous block of memory

30

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Multi-Level 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 ints

Note: this is how Java represents multi-dimensional arrays

31

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Element Access in Multi-Level 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
  - Element access `Mem[Mem[univ+8*index]+4*digit]`
  - Must do **two memory reads**
    - First get pointer to row array
    - Then access element within array
  - But allows inner arrays to be different lengths (not in this example)

32

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Array Element Accesses

### Nested array

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

### Multi-level 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]`

33

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## Strange Referencing Examples

Reference	Address	Value	Guaranteed?
<code>univ[2][3]</code>			
<code>univ[1][5]</code>			
<code>univ[2][-2]</code>			
<code>univ[3][-1]</code>			
<code>univ[1][12]</code>			

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

34

UNIVERSITY of WASHINGTON L13: Arrays CSE351, Autumn 2017

## 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

35