Arrays

CSE 351 Winter 2018

Instructor:
Mark Wyse

Teaching Assistants:
Kevin Bi
Parker DeWilde
Emily Furst
Sarah House
Waylon Huang
Vinny Palaniappan

http://xkcd.com/1270/

Administrative

- Lab 2 due tonight by 11:59 pm!
- Homework 3 due next Friday (2/9)
- **Midterm** (Monday 2/5)
  - ID check, so come at 5pm
  - **Bring your UW Student ID (Husky Card)**
  - **Review session** 2:00-4:00pm on Saturday (2/3) in EEB 125

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

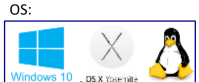
Assembly language:

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

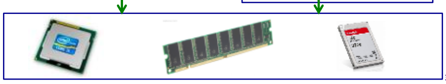
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:



Computer system:



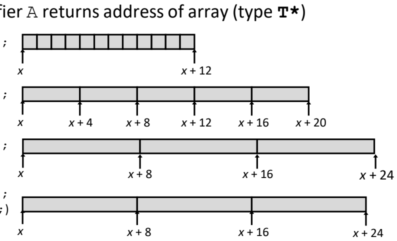
Data Structures in Assembly

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

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*`)

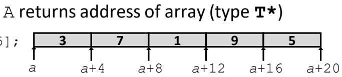
`char msg[12];`
`int val[5];`
`double a[3];`
`char *p[3];` (or `char* p[3];`)



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
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 a+20)
*(x+1)	int	7
x+i	int*	a + 4*i

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]`"

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

Array Accessing Example

`typedef int zip_dig[5];`

`zip_dig uw;`

9	8	1	9	5

36 40 44 48 52 56

```
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)`

Referencing Examples

`typedef int zip_dig[5];`

`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

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*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

`zip_dig uw;`

9	8	1	9	5

36 40 44 48 52 56

- 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 5th digit

Increments by 4 (size of int)

Array Loop Implementation

gcc with -O1

- Registers:
  - %rdi z
  - %rax zi
  - %rcx zend
- Computations
  - 10*zi + *z
  - z++

```
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
Init
  leaq  20(%rdi),%rcx          # %rcx = zend = z + 5
  movl  $0,%eax                # %rax = zi = 0
.L17:
Computation
  leal  (%rax,%rax,4),%edx     # %rdx = zi + 4*zi = 5*zi
  movl  (%rdi),%eax            # %rax = *z
  leal  (%rax,%rdx,2),%eax     # %rax = *z + 2(5*zi) = *z + 10*zi
  addq  $4,%rdi                # z++ (pointer arithmetic)
Jump if zend – z != 0
  cmpq  %rdi,%rcx              # zend – z
  jne   .L17                   # If != 0, goto Loop
```


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 (0th) element
  - `ar[0]` same as `*ar`; `ar[2]` same as `*(ar+2)`
- An array variable is read-only (no assignment)
  - 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;
}
```

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!

Data Structures in Assembly

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

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`

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

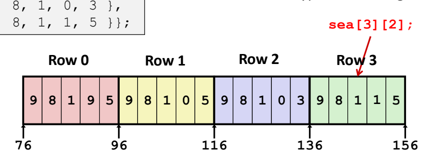
What is the layout in memory?

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 }};
```

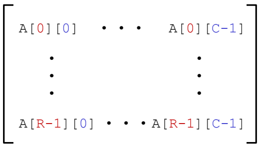
Remember, `T A[N]` is an array with elements of type `T`, with length `N`



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

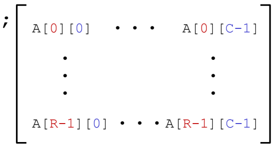
Two-Dimensional (Nested) Arrays

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

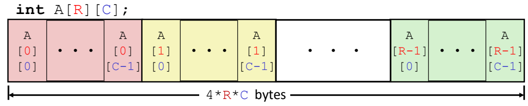


Two-Dimensional (Nested) Arrays

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



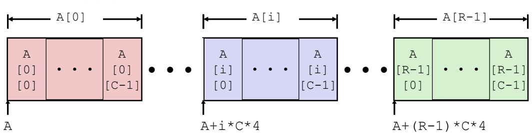
`int A[R][C];`



Nested Array Row Access

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

`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 }};
```

- What data type is `sea[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
...
```


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 starting address?

```
# %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 }};
```

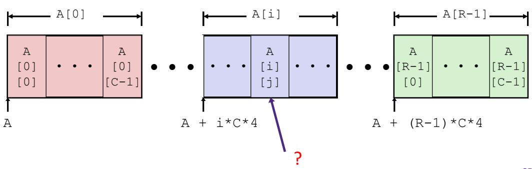
```
# %rdi = index
leaq (%rdi,%rdi,4),%rax   # 5 * index
leaq sea(,%rax,4),%rax    # sea + (20 * index)
```

- 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

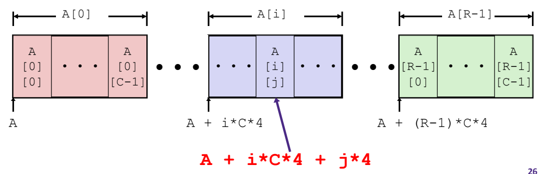
`int A[R][C];`



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 }};
```

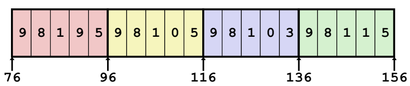
```
leaq  (%rdi,%rdi,4), %rax   # 5*index
addl  %rax, %rsi            # 5*index+digit
movl  sea(,%rsi,4),  %eax   # *(sea + 4*(5*index+digit))
```

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

Strange Referencing Examples

`typedef int zip_dig[5];`

`zip_dig sea[4];`



Reference	Address	Value	Guaranteed?
`sea[3][3]`			
`sea[2][5]`			
`sea[2][-1]`			
`sea[4][-1]`			
`sea[0][19]`			
`sea[0][-1]`			

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

Data Structures in Assembly

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

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};
```

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

2D Array Declaration:

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

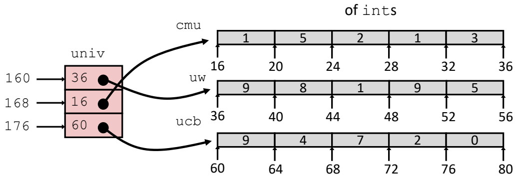
One array declaration = one contiguous block of memory

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};
```

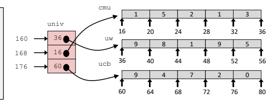
- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 8 bytes each
- Each pointer points to array of `int`s



Note: this is how Java represents multi-dimensional arrays

Element Access in Multi-Level Array

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



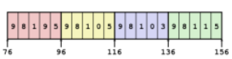
```
salq    $2, %rsi             # rsi = 4*digit
addq    univ(,%rdi,8), %rsi  # p = univ[index] + 4*digit
movl    (%rsi), %eax         # return *p
ret
```

- 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)

Array Element Accesses

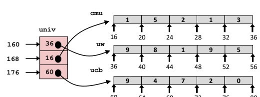
Nested array

```
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];
}
```

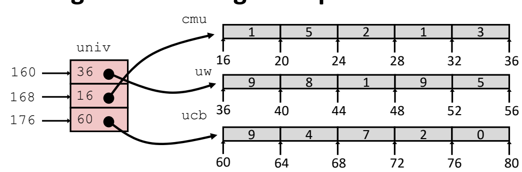


Access *looks* the same, but it isn't:

Mem[`sea+20*index+4*digit`]

Mem[Mem[`univ+8*index`]+`4*digit`]

Strange Referencing Examples



Reference	Address	Value	Guaranteed?
`univ[2][3]`			
`univ[1][5]`			
`univ[2][-2]`			
`univ[3][-1]`			
`univ[1][12]`			

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

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