Machine-Level Programming IV: Structured Data

CS 105: Computer Systems
Lecture 08

Melissa O'Neill

February 16, 2026

Learning Goals

- **Describe how one-dimensional and two-dimensional arrays are stored in memory and how to address an element in the array**
- **Reason about the layout of C `structs` and data alignment nuances**

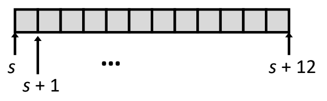
Quiz, due Today!
- **movq and leaq instructions – be able to use, given address specification format**
- **Control flow – determine condition code values, idea of how jumps work**
- **Procedures – what happens to the stack and PC when call and return from a function**

Static Array Allocation

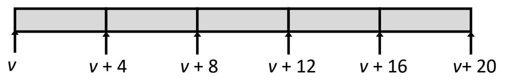
- **Basic Principle**

  T **`A`**`[`*L*`];`
  - Declare array of data type *T* and length *L*
  - Yields contiguously allocated region of *L* * **`sizeof`**`(`*T*`)` bytes in memory
- **Examples**

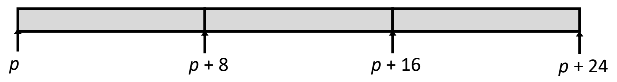
`char s[12];`



`int v[5];`



`char *p[3];`

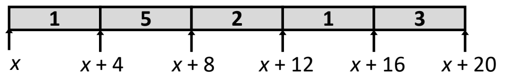


Array Access

- **Basic Principle**

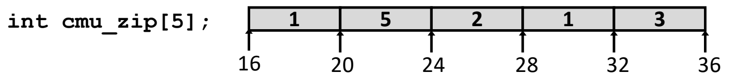
  T **`A`**`[`*L*`];`
  - Array of data type *T* and length *L*
  - Identifier **`A`** can be used as a pointer to array element 0: it has Type *T**
  - Suppose the array starts at address *x*

`int val[5];`



■ Expression	Type	Value
`val[4]`	`int`	3
`val`	`int *`	*x*
`val+1`	`int *`	*x* + 4
`&val[2]`	`int *`	*x* + 8
`val[5]`	`int`	??
`*(val+1)`	`int`	5
`val + `*`i`*	`int *`	*x* + 4 *i*

Array Accessing in Assembly: Example



```
int get_digit(int* z, int digit)
{
  return z[digit];
}
```

x86-64

```
  # %rdi = z
  # %rsi = 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`
- Uses memory reference `(%rdi,%rsi,4)`

Exercise

Below is the assembly code for `mystery`. What do you think `mystery` does?

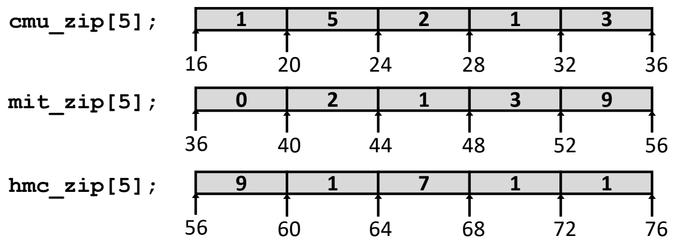
```
// z is starting address of a 5-element int array
void mystery(int* z) {
   [redacted]
}
```

```
  # %rdi = z
  movq    $0, %rax
  jmp     .L3
.L4:
  addl    $1, (%rdi,%rax,4)
  addq    $1, %rax
.L3:
  cmpq    $4, %rax
  jbe     .L4
  ret
```

`jbe`	**Below or Equal (unsigned) a≤b for** `cmpq b,a`

Multiple Arrays Example

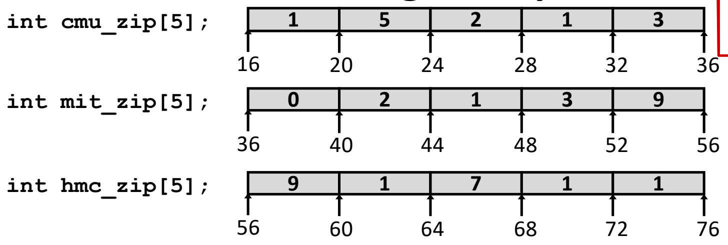
```
int cmu_zip[5] = { 1, 5, 2, 1, 3 };
int mit_zip[5] = { 0, 2, 1, 3, 9 };
int hmc_zip[5] = { 9, 1, 7, 1, 1 };
```



- **In this example, arrays happened to be allocated in successive 20 byte chunks**
  - **Not guaranteed to happen in general!**

Exercise: Referencing Examples

Code does not do any bounds checking!



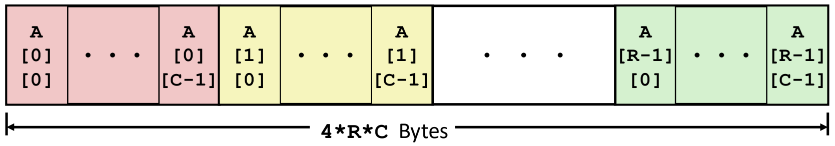
Reference	Address	Value	Value Guaranteed?
`mit_zip[3]`	`36 + 4* 3 = 48`	`3`	**Yes**
`mit_zip[5]`			
`mit_zip[-1]`			

Static Multidimensional (Nested) Arrays

- **Declaration**
  - *T* **A**[*R*][*C*];
  - 2D array of data type *T*
  - *R* rows, *C* columns
  - Type *T* element requires *K* bytes
- **Array Size in bytes**
  - *R* * *C* * *K* bytes
- **Arrangement in memory**
  - Row-Major Ordering

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

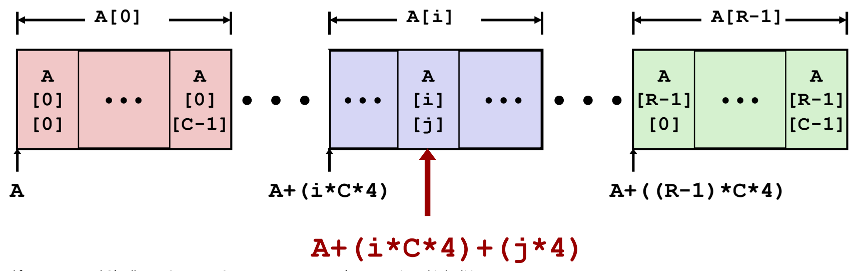
`int A[R][C];`



Nested Array: Element Access

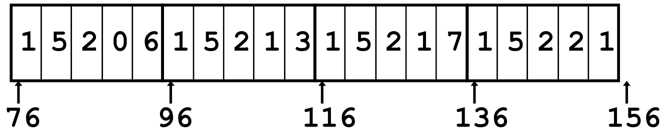
- **Array Elements**
  - **A[i][j]** is element of type *T*, which requires *K* bytes
  - Address: **A +** *i* * (*C* * *K*) + *j* * *K* = *A* + (*i* * *C* + *j*)* *K*

`int A[R][C];`



Exercise: Referencing Examples

`int zips[4][5];`



Reference	Address	Value	Value Guaranteed?
zips[2][5]			
zips[0][19]			
zips[0][-1]			

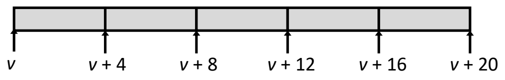
Dynamic Array Allocation

- **Basic Principle**

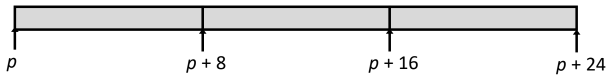
  T *** A = malloc(*L**sizeof(*T*));**

  - Array of data type *T* and length *L*
  - Contiguously allocated region of *L* * **sizeof** (*T*) bytes in memory
  - Can access as A[0]… A[L-1]
- **Examples**

```
int* v = (int *) malloc(5*sizeof(int));
```



```
char** p = (char **) malloc(3*sizeof(char *));
```

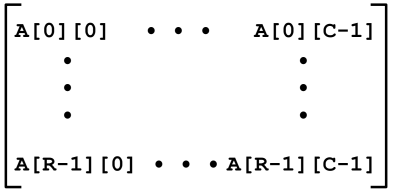


Dynamic Multidimensional Arrays

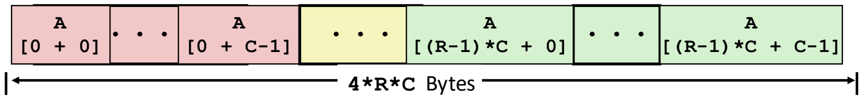
- **Nested**

  T *** A = malloc(*R**C*sizeof(*T*));**

  - "2D" array of data type *T*
  - *R* rows, *C* columns
  - Type *T* element requires *K* bytes

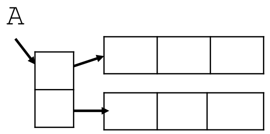


```
int* A = (int *) malloc(R*C*sizeof(int));
```



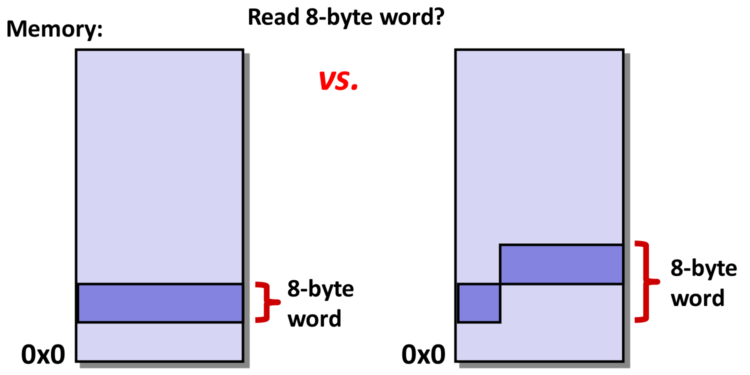
- **Non-nested**
  - Can access as elements as `A[i][j]`, but not all contiguous. E.g.:

```
int** A= (int**) malloc(2*sizeof(int*));
for (int i=0; i<2; i++)
   A[i] = (int *)malloc(3*sizeof(int));
```



Data Alignment: Intuition for Motivation

- **CPU fetches from memory in word-size chunks using an address that is a multiple of word size**
  - **8 bytes on a 64-bit architecture**



Alignment Principles

- **Aligned Data**
  - Primitive data type requires ***K*** bytes
  - Its address must be a multiple of ***K***
  - Required on some machines; advised on x86-64

E.g., `short int` is 2 bytes:
- **Lowest 1 bit of address must be 0_2**
- **i.e, a multiple of 2**

For `int`, lowest 2 bits of address must be 00_2 (multiple of 4)

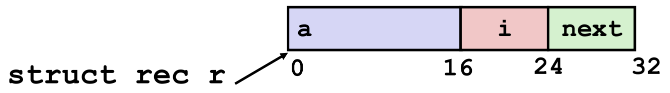
- **Motivation for Aligning Data**
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
  - Inefficient to load or store datum that spans quad word boundaries
- **Compiler**
  - Inserts gaps in a Struct to ensure correct alignment of its fields

Structure Representation

- **The C `struct` is a data type that groups objects of possibly different types**

```
struct rec {
    int a[4];
    unsigned long i;
    struct rec *next;
};
```

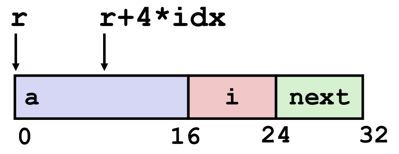
- **Represented as block of memory**
  - Big enough to hold all of the fields



- **Fields ordered according to declaration**
  - Even if another ordering could yield a more compact representation
- **Compiler determines overall size + positions of fields**
  - Machine-level program has no understanding of the structures in the source code

Generating Pointer to Structure Member

```
struct rec {
    int a[4];
    unsigned long i;
    struct rec *next;
};
```



- **Generating Pointer to Array Element**
  - Offset of each structure member determined at compile time
  - Compute as `r + 4*idx`

```
int *get_a_pointer
 (struct rec *r, unsigned long idx)
{
  return &(r->a[idx]);
}
```

```
  # r in %rdi, idx in %rsi
  leaq  (%rdi,%rsi,4), %rax
  ret
```

Exercise

- **Consider the redacted function `mystery` with its assembly below. What is `mystery` doing?**

```
void mystery(struct rec *r, int x){
  [redacted]
}
```

```
.L1:
  movq    16(%rdi), %rax
  movl    %esi, (%rdi,%rax,4)
  movq    24(%rdi), %rdi
  testq   %rdi, %rdi
  jne     .L1
```

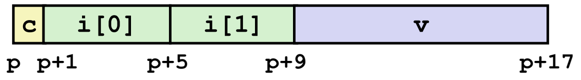
testq	`testq b,a`: computes `a&b` without setting destination
jne	Jumps if zero flag *not* set after `testq b,a`

Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy each element's alignment requirement
- **Overall structure placement**
  - Each structure has alignment requirement **K**
    - **K** = Largest alignment requirement of any element
  - Initial address & structure length must be multiples of **K**

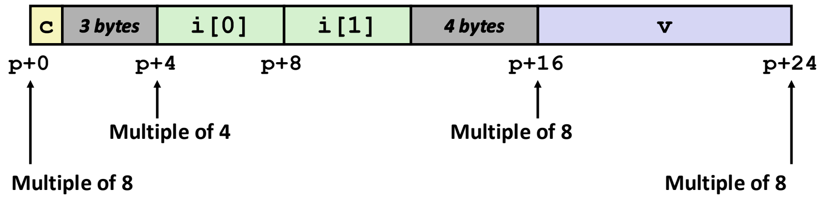
Structures & Alignment Within

- **Unaligned Data**



```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

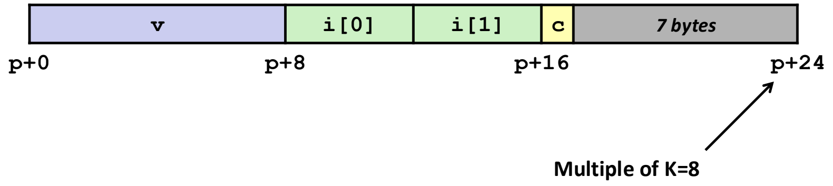
- **Aligned Data**
  - Primitive data type requires ***K*** bytes; address must be multiple of ***K***
    - In this example K = 8, due to `double` element



Meeting Overall Alignment Requirement

- **For largest alignment requirement K, overall structure must be multiple of K**

```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



Exercise

- **Draw the alignment for these two structs, with extra spaces as needed**

```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

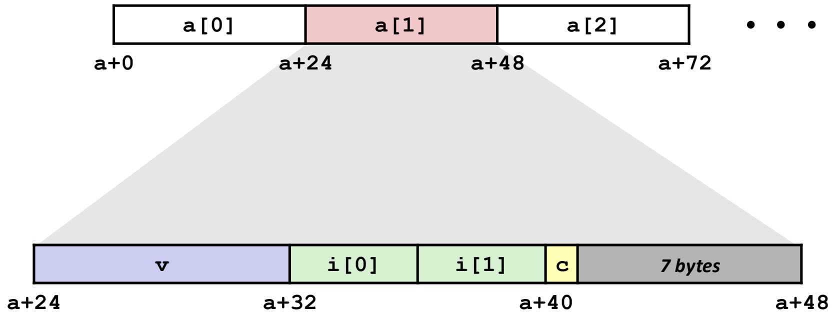
```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```

- **What do you notice? Can you think of a general rule?**

Arrays of Structures

- **Overall structure length multiple of K**
- **Satisfy alignment requirement for every element**

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```


a[0]
a[1]
a[2]
• • •
a+0
a+24
a+48
a+72
v
i[0]
i[1]
c
7 bytes
a+24
a+32
a+40
a+48