CS 31: Intro to Systems
Arrays, Structs, Strings, and Pointers

Vasanta Chaganti & Kevin Webb

Swarthmore College

October 26, 2023

Overview

- Accessing *things* via an offset
  - Arrays, Structs, Unions
  - Connect accessing them in C with what we know about assembly

- How complex structures are stored in memory
  - Multi-dimensional arrays & Structs

So far: Primitive Data Types

- We’ve been using ints, floats, chars, pointers

- Simple to place these in memory:
  - They have an unambiguous size
  - They fit inside a register*
  - The hardware can operate on them directly

(*There are special registers for floats and doubles that use the IEEE floating point format.)

Composite Data Types

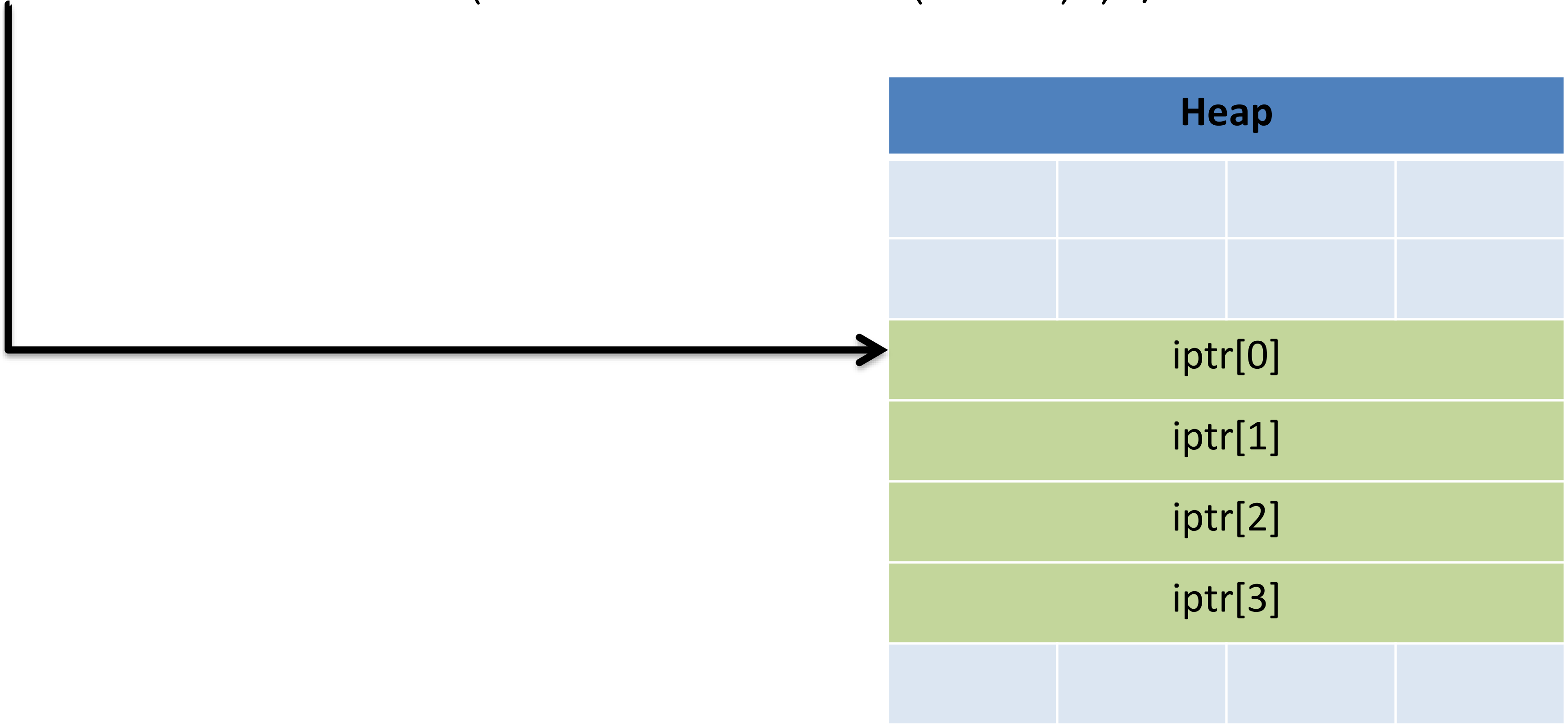
- Combination of one or more existing types into a new type. (e.g., an array of *multiple* ints, or a struct)

- Example: a queue
  - Might need a value (int) plus a link to the next item (pointer)

```
struct queue_node{
  int value;
  struct queue_node *next;
}
```

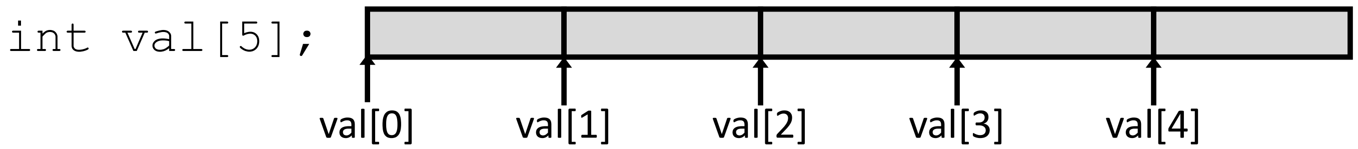
Recall: Arrays in Memory

```
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```



Base + Offset

- We know that arrays act as a pointer to the first element. For bucket [N], we just skip forward N.

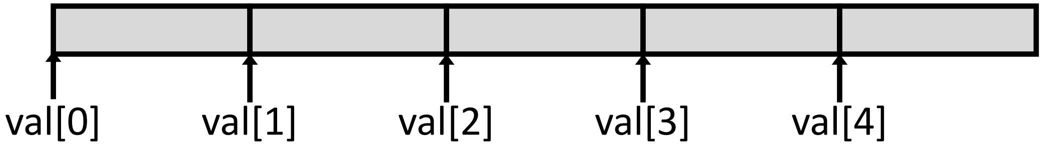


- “We’re goofy computer scientists who count starting from zero.”

Base + Offset

- We know that arrays act as a pointer to the first element. For bucket [N], we just skip forward N.

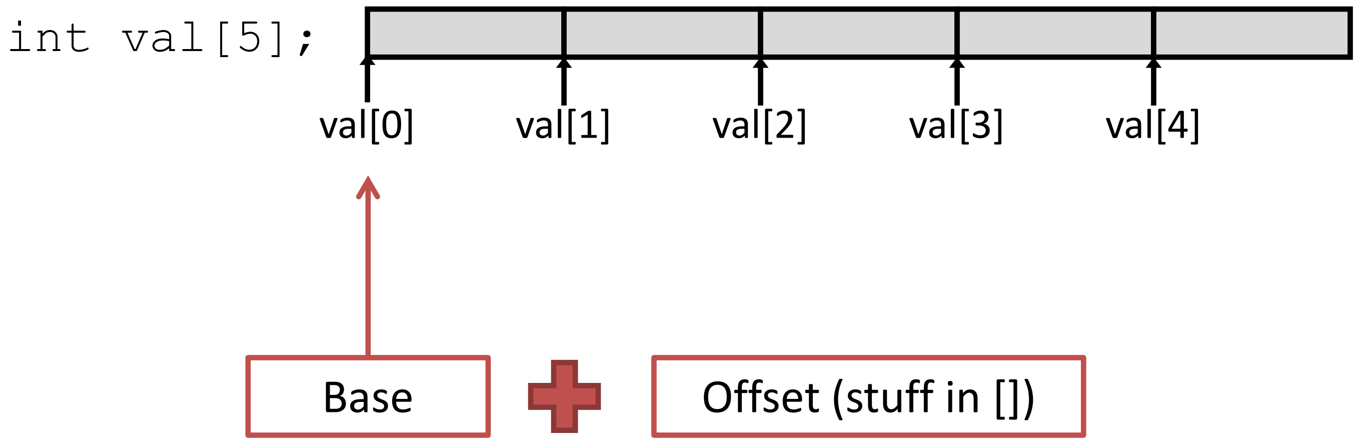
```
int val[5];
```



- ~~“We’re goofy computer scientists who count starting from zero.”~~

Base + Offset

- We know that arrays act as a pointer to the first element. For bucket [N], we just skip forward N.



This is why we start counting from zero!
Skipping forward with an offset of zero ([0]) gives us the first bucket…

Which expression would compute the address of iptr[3]?

What if this isn't known at compile time?

A. 0x0824 + 3 * 4

B. 0x0824 + 4 * 4

C. 0x0824 + 0xC

D. More than one (which?)

E. None of these

Heap			
0x0824:	iptr[0]		
0x0828:	iptr[1]		
0x082C:	iptr[2]		
0x0830:	iptr[3]		

Recall Addressing Mode: Memory

- Accessing memory requires you to specify which address you want.
  - Put the address in a register.
  - Access the register with () around the register's name.

```
mov (%rcx), %rax
```

  - Use the address in register %rcx to access memory, store result in register %rax

Recall Addressing Mode: Displacement

- Like memory mode, but with a constant offset
  - Offset is often negative, relative to %rbp

```
mov -24(%rbp), %rax
```

- Take the address in %rbp, subtract 24 from it, index into memory and store the result in %rax.

Addressing Mode: Indexed

- Instead of only using one register to store the base address of a memory address, we can use a base address register **and** an offset register value.

```
mov (%rax, %rcx), %rdx
```

  – Take the base address in %rax, add the value in %rcx to produce a final address, index into memory and store the result in %rdx.

Addressing Mode: Indexed

- Instead of only using one register to store the base address of a memory address, we can use a base address register **and** an offset register value.

```
mov (%rax, %rcx), %rdx
```

  - Take the base address in %rax, add the value in %rcx to produce a final address, index into memory and store the result in %rdx.

One register to keep track of base address.

One register to keep track of offset from base address.

Addressing Mode: Indexed

- The offset can also be scaled by a constant.

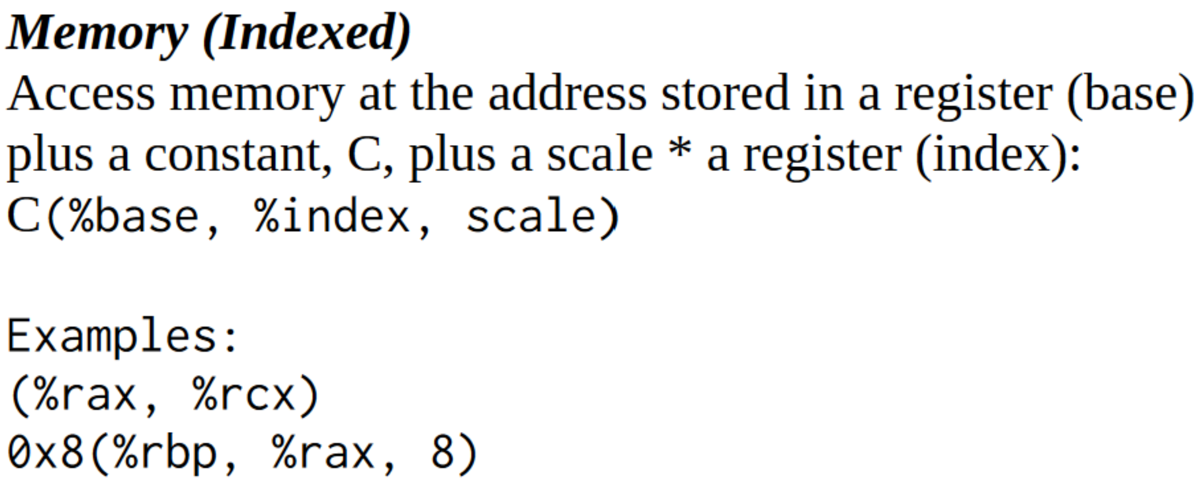
```
mov (%rax, %rcx, 4), %rdx
```

  – Take the base address in %rax, add (value in %rcx * 4) to produce a final address, index into memory and store the result in %rdx.

  – (If you don’t specify a scale constant, it defaults to 1)

Assembly Reference

- This mode has been on your assembly reference sheet all along:



Memory (Indexed)
Access memory at the address stored in a register (base) plus a constant, C, plus a scale * a register (index):

```
C(%base, %index, scale)

Examples:
(%rax, %rcx)
0x8(%rbp, %rax, 8)
```

Example

Suppose:

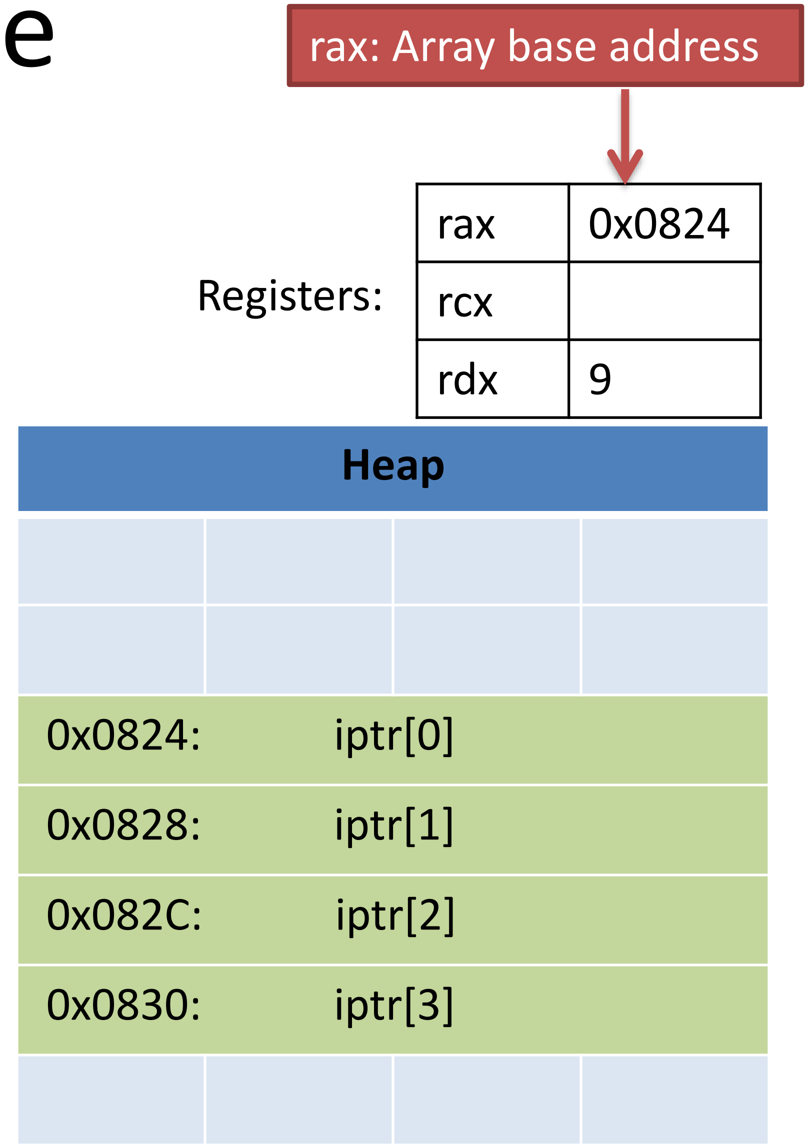
iptr is stored in register rax.

i is at rbp-8 and equals 2.

User says:

```
iptr[i] = 9;
```

Translates to:



rax: Array base address

Registers:

rax	0x0824
rcx	
rdx	9

Heap

0x0824:	iptr[0]
0x0828:	iptr[1]
0x082C:	iptr[2]
0x0830:	iptr[3]

Example

Suppose:

iptr is stored in register rax.

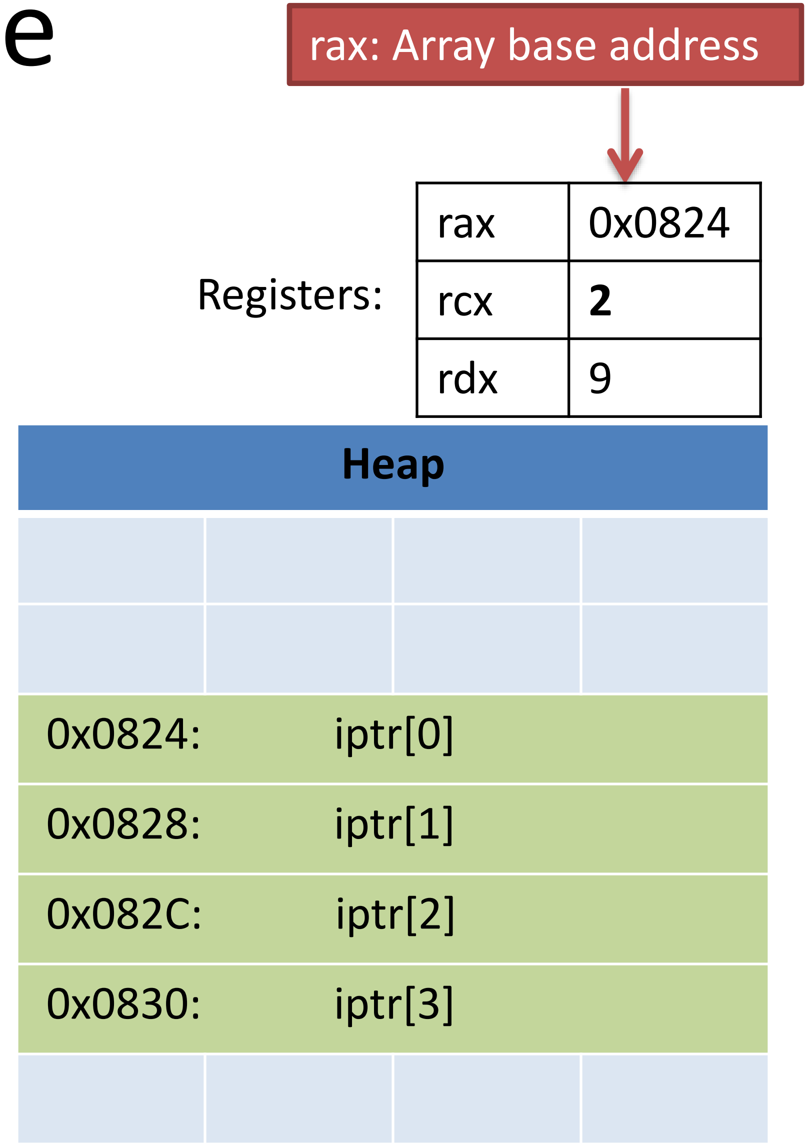
i is at rbp-8 and equals 2.

User says:

```
iptr[i] = 9;
```

Translates to:

```
mov -8(%rbp), %rcx
```



rax: Array base address

Registers:

rax	0x0824
rcx	**2**
rdx	9

Heap

0x0824:	iptr[0]
0x0828:	iptr[1]
0x082C:	iptr[2]
0x0830:	iptr[3]

Example

Suppose:

iptr is stored in register rax.

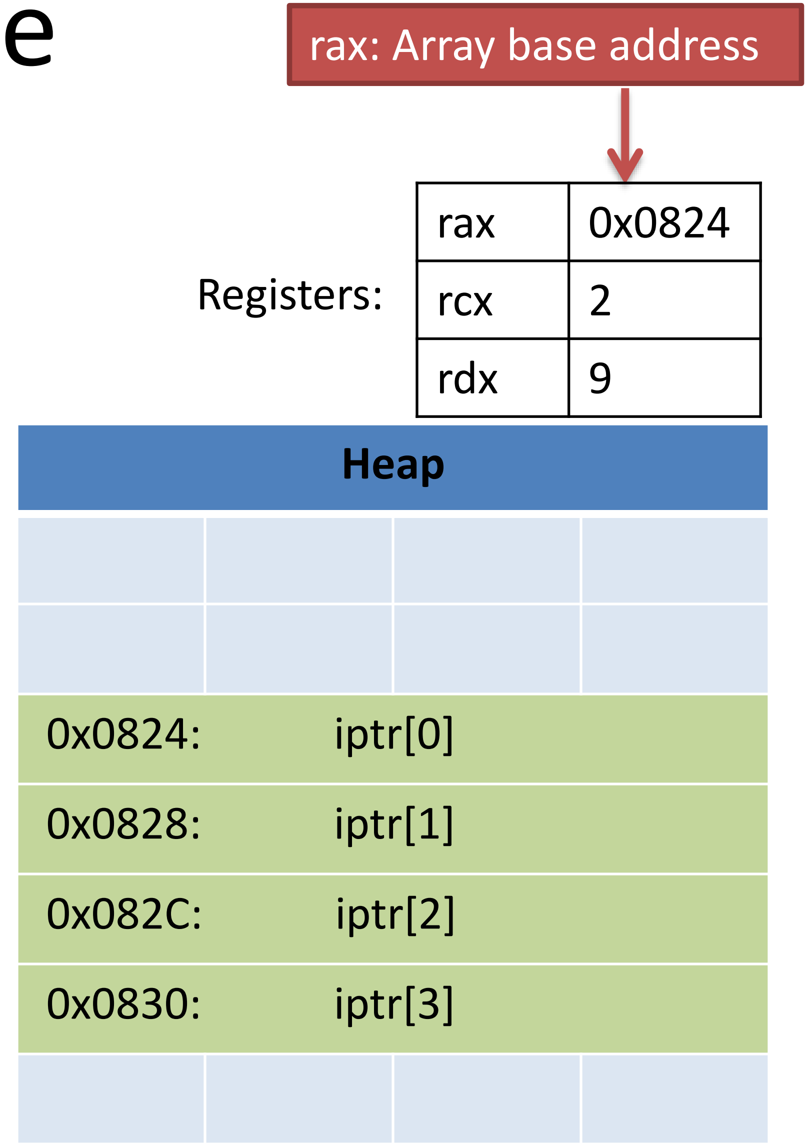
i is at rbp-8 and equals 2.

User says:

```
iptr[i] = 9;
```

Translates to:

```
mov -8(%rbp), %rcx
mov %rdx, (rax, rcx, 4)
```



rax: Array base address

Registers:

rax	0x0824
rcx	2
rdx	9

Heap

0x0824:	iptr[0]
0x0828:	iptr[1]
0x082C:	iptr[2]
0x0830:	iptr[3]

Example

Suppose:

iptr is stored in register rax.

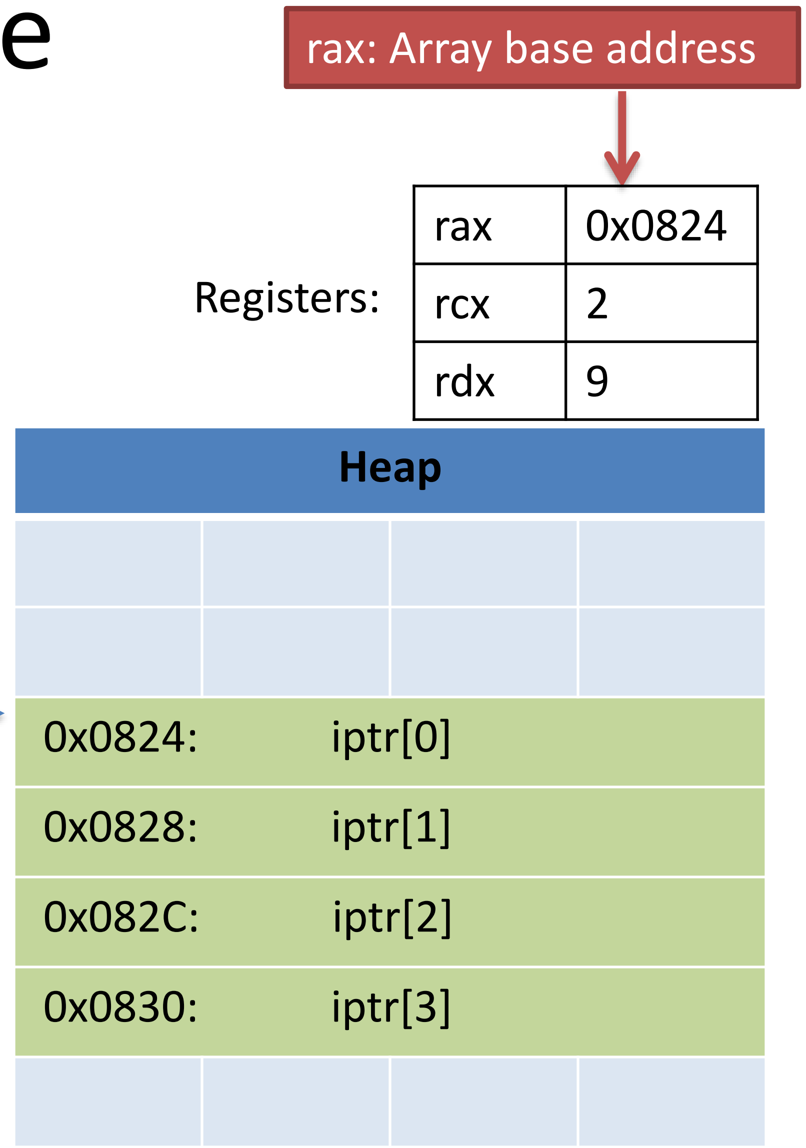
i is at rbp-8 and equals 2.

User says:

```
iptr[i] = 9;
```

Translates to:

```
mov -8(%rbp), %rcx
mov %rdx, (rax, rcx, 4)
```



rax: Array base address

Registers:

rax	0x0824
rcx	2
rdx	9

Heap

0x0824:	iptr[0]
0x0828:	iptr[1]
0x082C:	iptr[2]
0x0830:	iptr[3]

Example

Suppose:

iptr is stored in register rax.

i is at rbp-8 and equals 2.

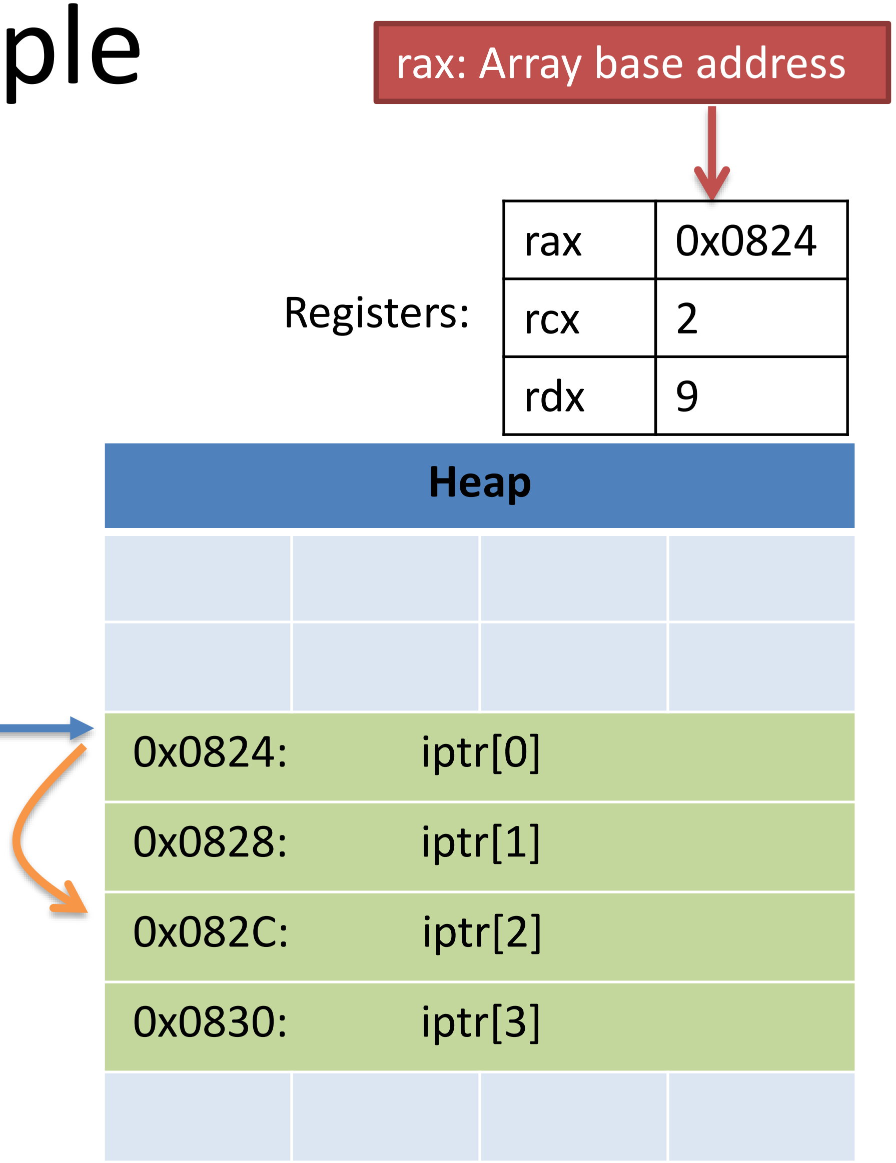
User says:

```
iptr[i] = 9;
```

Translates to:

```
mov -8(%rbp), %rcx
mov %rdx, (rax, rcx, 4)
```

+(2*4) = +8



rax: Array base address

Registers:

rax	0x0824
rcx	2
rdx	9

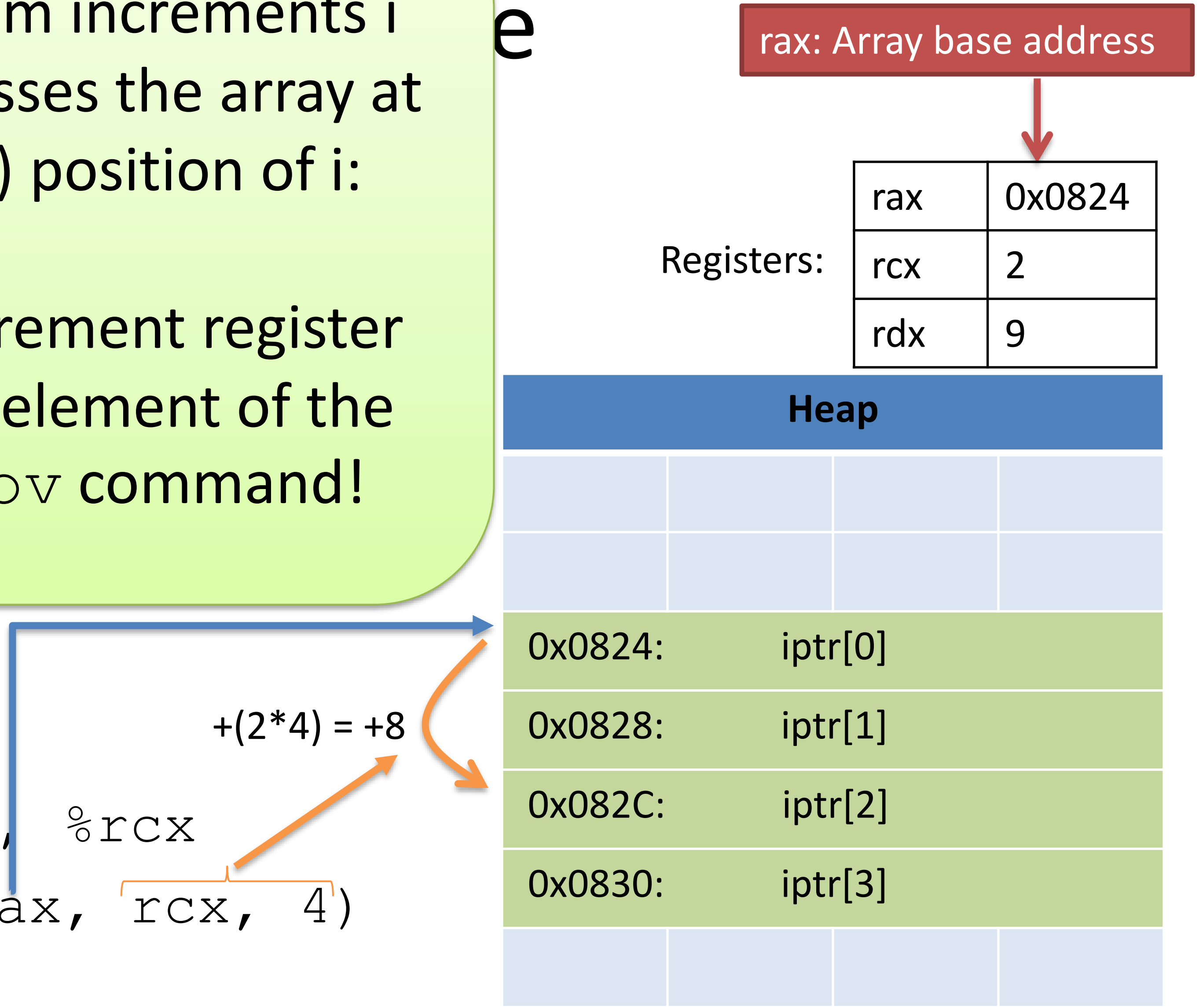
Heap	
0x0824:	iptr[0]
0x0828:	iptr[1]
0x082C:	iptr[2]
0x0830:	iptr[3]

From here, if the program increments i (e.g., in a loop) and accesses the array at the new (incremented) position of i:

Compiler can simply increment register rcx and access the next element of the array with the same `mov` command!

Translates to:

```
mov -8(%rbp), %rcx
mov %rdx, (rax, rcx, 4)
```



rax: Array base address

Registers:

rax	0x0824
rcx	2
rdx	9

Heap	
0x0824:	iptr[0]
0x0828:	iptr[1]
0x082C:	iptr[2]
0x0830:	iptr[3]

+(2*4) = +8

Structs

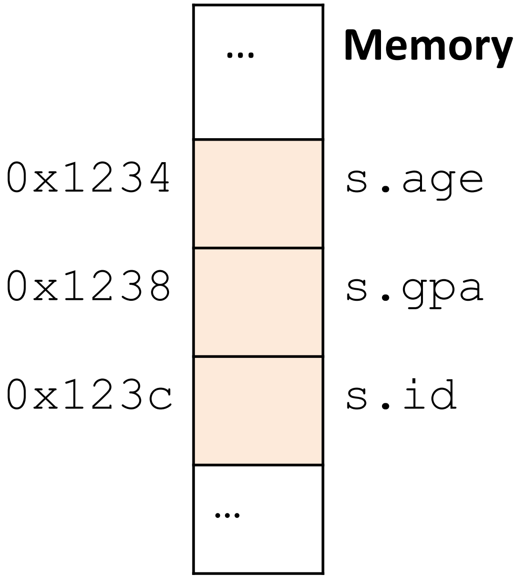
- Multiple values (fields) stored together
    - Defines a new type in C's type system

- Laid out contiguously by field (with a caveat we'll see later)
    - In order of field declaration.

Structs

- Laid out contiguously by field (with a caveat we'll see later)
  - In order of field declaration.

```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
```



Memory

…

0x1234 s.age

0x1238 s.gpa

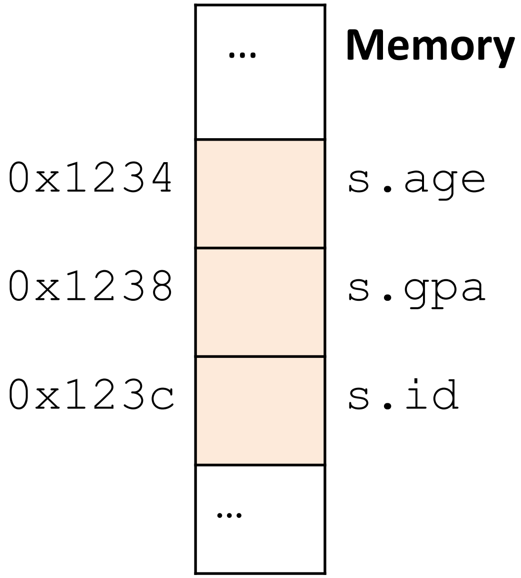
0x123c s.id

…

Structs

- Struct fields accessible as a base + displacement
  - Compiler knows (constant) displacement of each field

```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
```



Memory

…

0x1234 s.age

0x1238 s.gpa

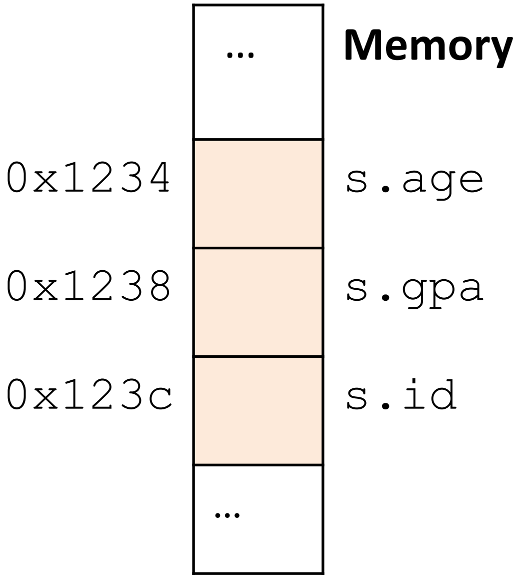
0x123c s.id

…

Structs

- Struct fields accessible as a base + displacement
  - Compiler knows (constant) displacement of each field

```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
s.id = 12;
```

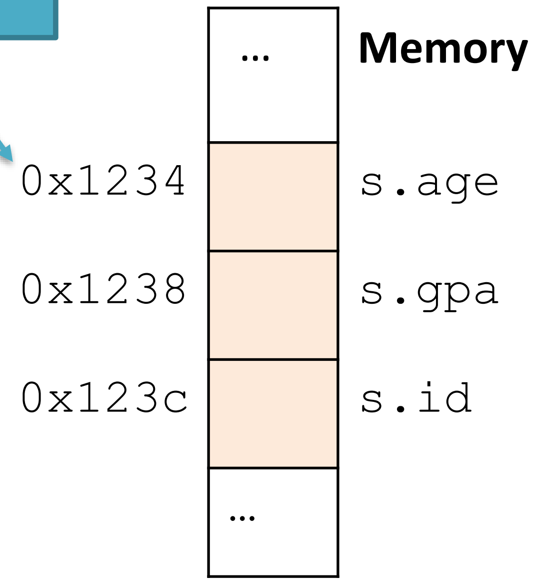


Structs

- Struct fields accessible as a base + displacement
  - Compiler knows (constant) displacement of each field

```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
s.id = 12;
```

Given the starting address of a struct…



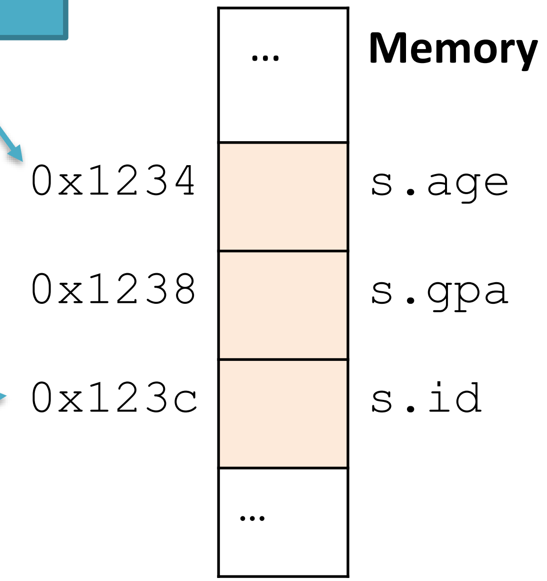
Structs

- Struct fields accessible as a base + displacement
  - Compiler knows (constant) displacement of each field

```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
s.id = 12;
```

Given the starting address of a struct…

The id field is always at an offset of 8 forward from the start.



Structs

- Struct fields accessible as a base + displacement
  - Compiler knows (constant) displacement of each field

In assembly:

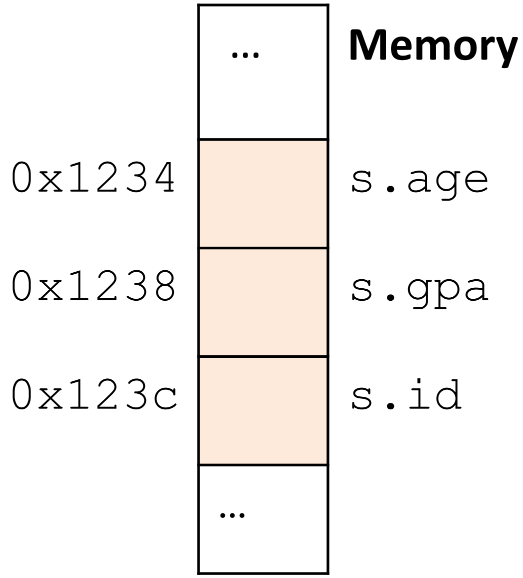
```
mov reg_value, 8(reg_base)
```

Where:

`reg_value` is a register holding the value to store (12)

`reg_base` is a register holding the base address of the struct

```
s.id = 12;
```

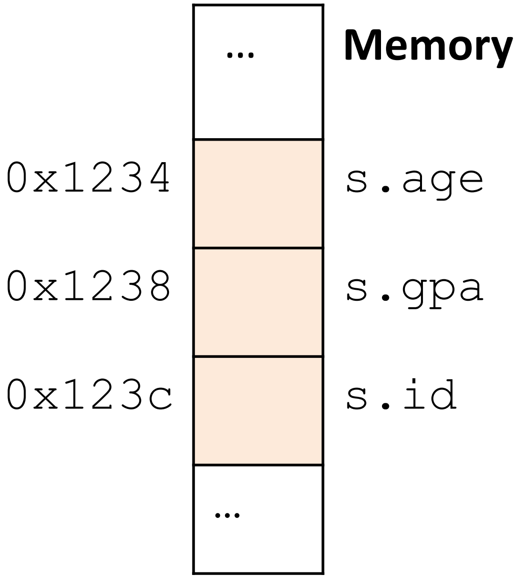


Memory

	…	
0x1234		s.age
0x1238		s.gpa
0x123c		s.id
	…	

Structs

- Laid out contiguously by field
  - – In order of field declaration.
  - – May require some padding, for alignment.



Data Alignment:

- Where (which address) can a field be located?

- char (1 byte): can be allocated at any address:

  0x1230, 0x1231, 0x1232, 0x1233, 0x1234, …

- short (2 bytes): must be aligned on 2-byte addresses:

  0x123**0**, 0x123**2**, 0x123**4**, 0x123**6**, 0x123**8**, …

- int (4 bytes): must be aligned on 4-byte addresses:

  0x123**0**, 0x123**4**, 0x123**8**, 0x123**c**, 0x124**0**, …

Why do we want to align data on multiples of the data size?

A. It makes the hardware faster.

B. It makes the hardware simpler.

C. It makes more efficient use of memory space.

D. It makes implementing the OS easier.

E. Some other reason.

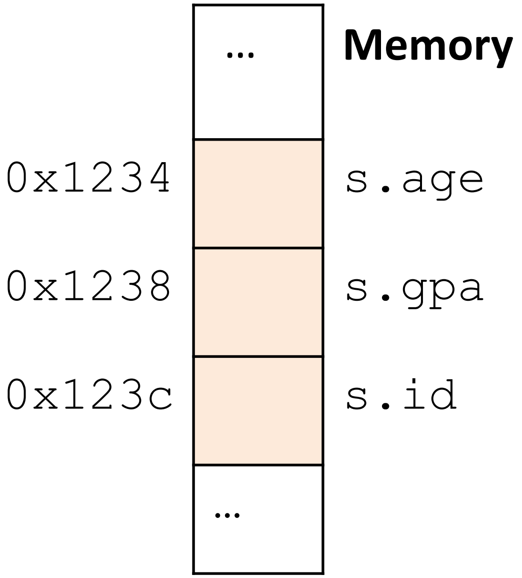
Data Alignment: Why?

- Simplify hardware
  - e.g., only read ints from multiples of 4
  - Don’t need to build wiring to access 4-byte chunks at any arbitrary location in hardware
- Inefficient to load/store single value across alignment boundary (1 vs. 2 loads)
- Simplify OS:
  - Prevents data from spanning virtual pages
  - Atomicity issues with load/store across boundary

Structs

- Laid out contiguously by field
  - In order of field declaration.
  - May require some padding, for alignment.

```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
```



Structs

```
struct student{
  char name[11];
  short age;
  int id;
};
```

How much space do we need to store one of these structures? Why?

```
struct student{
  char name[11];
  short age;
  int id;
};
```

A. 17 bytes

B. 18 bytes

C. 20 bytes

D. 22 bytes

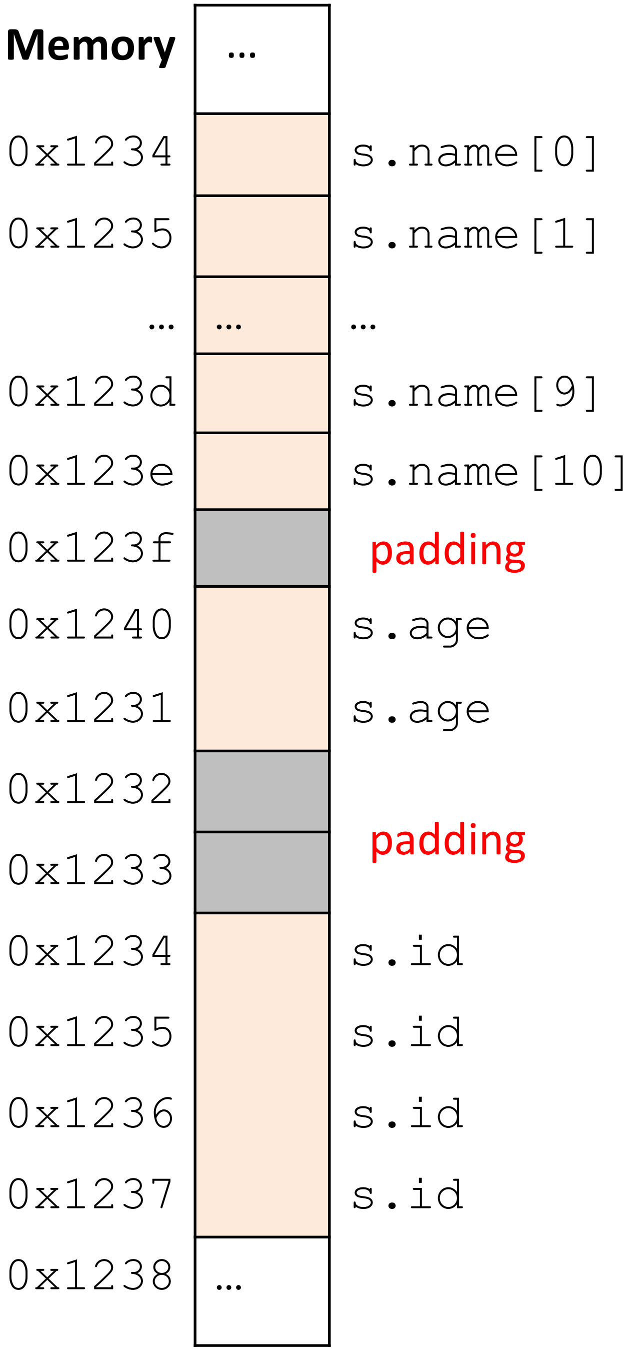
E. 24 bytes

Structs

```
struct student{
  char name[11];
  short age;
  int id;
};
```

- Size of data: 17 bytes
- Size of struct: 20 bytes

<u>Use sizeof() when allocating structs with malloc()!</u>



Memory	…	
0x1234		s.name[0]
0x1235		s.name[1]
…	…	…
0x123d		s.name[9]
0x123e		s.name[10]
0x123f		padding
0x1240		s.age
0x1231		s.age
0x1232		padding
0x1233		
0x1234		s.id
0x1235		s.id
0x1236		s.id
0x1237		s.id
0x1238	…	

Alternative Layout

```
struct student{
   int id;
   short age;
   char name[11];
};
```

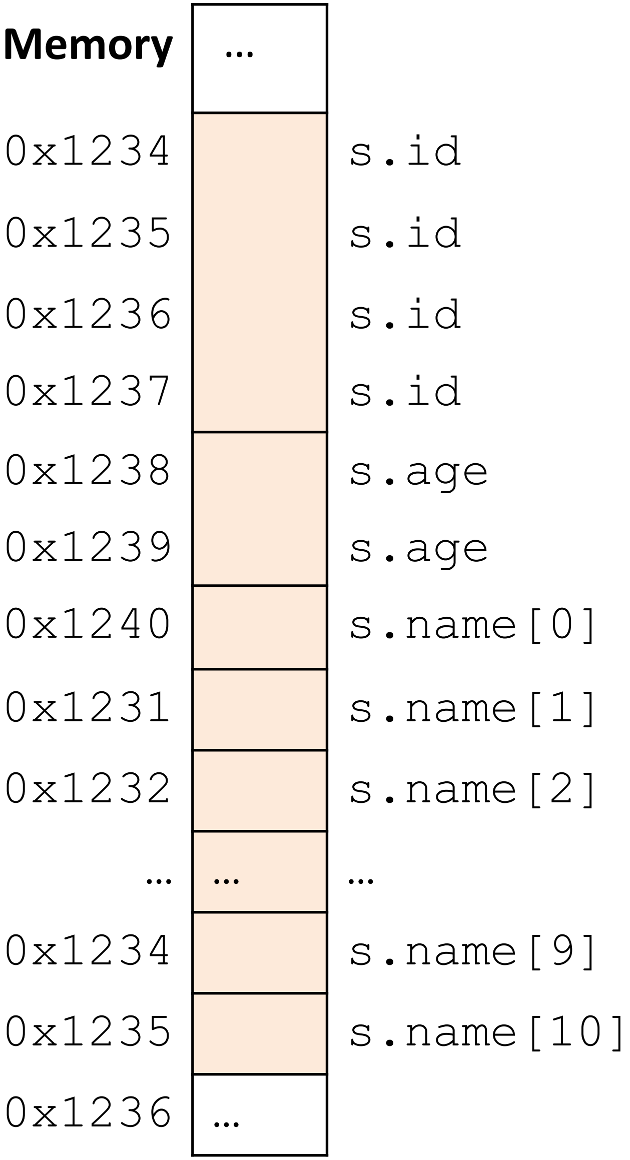
Same fields, declared in a different order.

Alternative Layout

```
struct student{
   int id;
   short age;
   char name[11];
};
```

- Size of data: 17 bytes
- Size of struct: 17 bytes!

In general, this isn't a big deal on a day-to-day basis. Don't go out and rearrange all your struct declarations.



Memory	…	
0x1234		s.id
0x1235		s.id
0x1236		s.id
0x1237		s.id
0x1238		s.age
0x1239		s.age
0x1240		s.name[0]
0x1231		s.name[1]
0x1232		s.name[2]
…	…	…
0x1234		s.name[9]
0x1235		s.name[10]
0x1236	…	

Aside: Network Headers

- In networks, we attach metadata to packets
  - Things like destination address, port #, etc.

- Common for these to be a specific size/format
  - e.g., the first 20 bytes must be laid out like …

- Naïvely declaring a struct might introduce padding, violate format.

Cool, so we can get rid of this struct padding by being smart about declarations?

A. Yes (why?)

B. No (why not?)

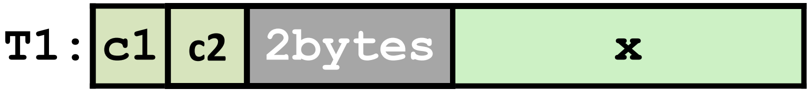
Cool, so we can get rid of this padding by being smart about declarations?

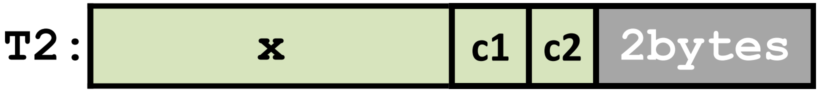
- Answer: Maybe.

- Rearranging helps, but often padding after the struct can’t be eliminated.

```
struct T1 {
     char c1;
     char c2;
     int  x;
};
```

```
struct T2 {
        int x;
        char c1;
        char c2;
};
```



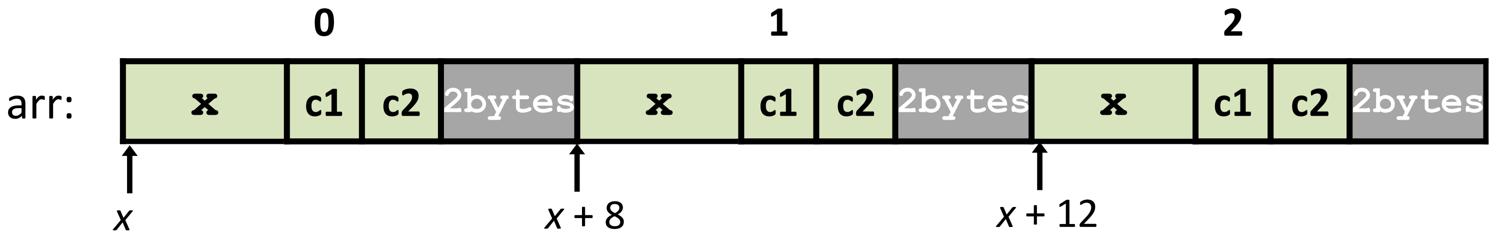


“External” Padding

- Array of Structs

Field values in each bucket must be properly aligned:

```
struct T2 arr[3];
```



Buckets must be on a 4-byte aligned address

Struct field syntax…

```
struct student {
  int id;
  short age;
  char name[11];
};
struct student s;

s.id = 406432;
s.age = 20;
strcpy(s.name, “Alice”);
```

Struct is declared on the stack.
(NOT a pointer)

Struct field syntax…

```
struct student {
  int id;
  short age;
  char name[11];
};
struct student *s = malloc(sizeof(struct student));
```

Not a struct, but a pointer to a struct!

```
(*s).id = 406432;
(*s).age = 20;
strcpy((*s).name, “Alice”);
```

This works, but is very ugly.

```
s->id = 406432;
s->age = 20;
strcpy(s->name, “Alice”);
```

Access the struct field from a pointer with ->
Does a dereference **and** gets the field.

Stack Padding

- Memory alignment applies elsewhere too.

```
int x;         vs.     double y;
char ch[5];            int x;
short s;               short s;
double y;              char ch[5];
```

In nearly all cases, you shouldn't stress about this. The compiler will figure out where to put things.

Exceptions: network headers, you're writing an OS and/or are optimizing for caches, etc.

Unions

- Declared like a struct, but only contains one field, rather than all of them.

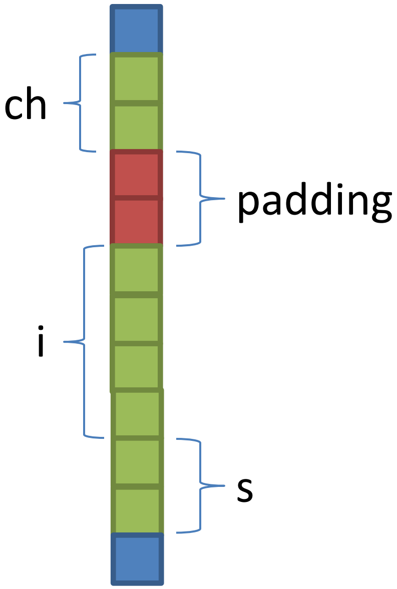
- Struct: field 1 and field 2 and field 3 …
- Union: field 1 or field 2 or field 3 …

- Intuition: you know you only need to store one of N things, don’t waste space.

Unions

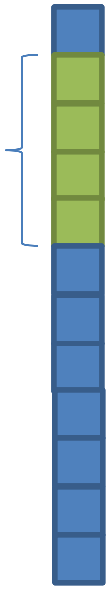
```
struct my_struct {
    char ch[2];
    int i;
    short s;
}
```

```
union my_union {
    char ch[2];
    int i;
    short s;
}
```



my_struct in memory

Same memory used for all fields!

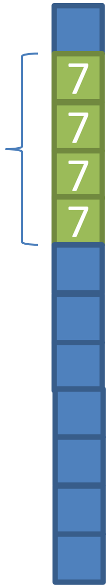


my_union in memory

Unions

```
my_union u;

u.i = 7;
```

```
union my_union {
    char ch[2];
    int i;
    short s;
}
```



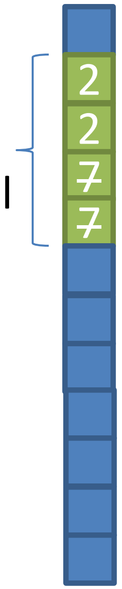
Same memory used for all fields!

my_union in memory

Unions

```
my_union u;

u.i = 7;

u.s = 2;
```

```
union my_union {
    char ch[2];
    int i;
    short s;
}
```



Same memory used for all fields!

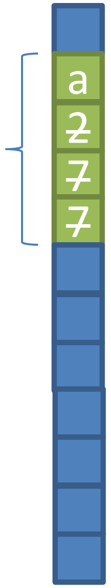
my_union in memory

Unions

```
my_union u;

u.i = 7;

u.s = 2;


u.ch[0] = ‘a’;
```

Reading i or s here would be bad!

```
union my_union {
    char ch[2];
    int i;
    short s;
}
```



Same memory used for all fields!

my_union in memory

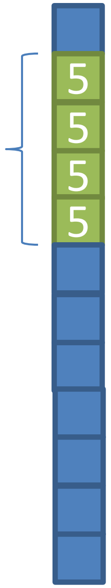
Unions

```
my_union u;

u.i = 7;

u.s = 2;

u.ch[0] = ‘a’;
```

Reading i or s here would be bad!

```
u.i = 5;
```

```
union my_union {
    char ch[2];
    int i;
    short s;
}
```



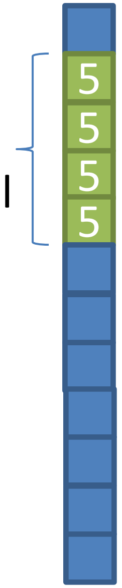
Same memory used for all fields!

my_union in memory

Unions

- You probably won't use these often.
- Use when you need mutually exclusive types.
- Can save memory.

```
union my_union {
    char ch[2];
    int i;
    short s;
}
```



Same memory used for all fields!

my_union in memory

Two-dimensional Arrays

- Why stop at an array of ints?
  How about an array of arrays of ints?

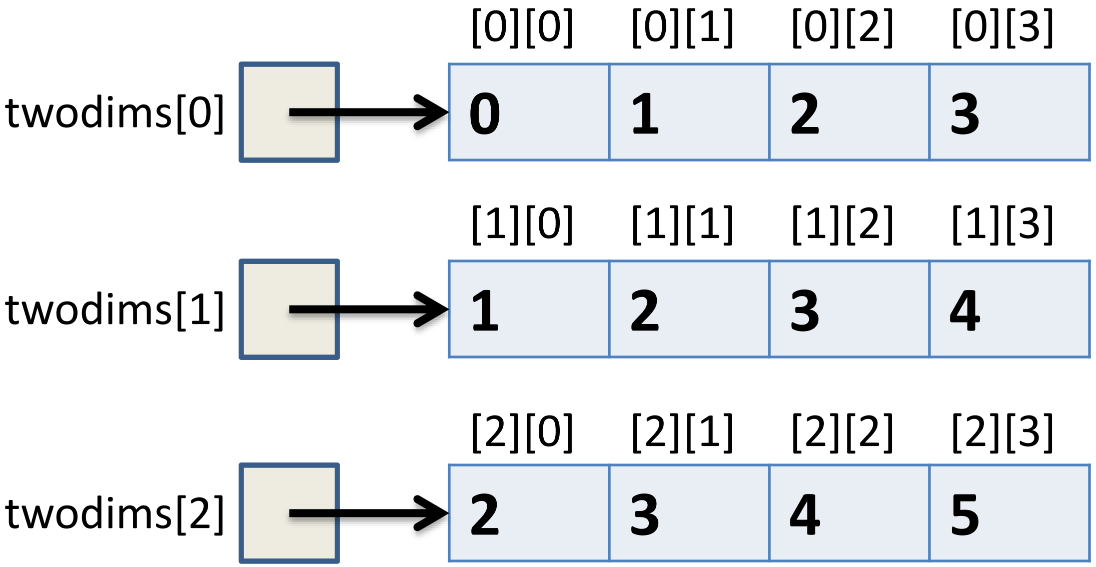
```
int twodims[3][4];
```

- “Give me three sets of four integers.”

- How should these be organized in memory?

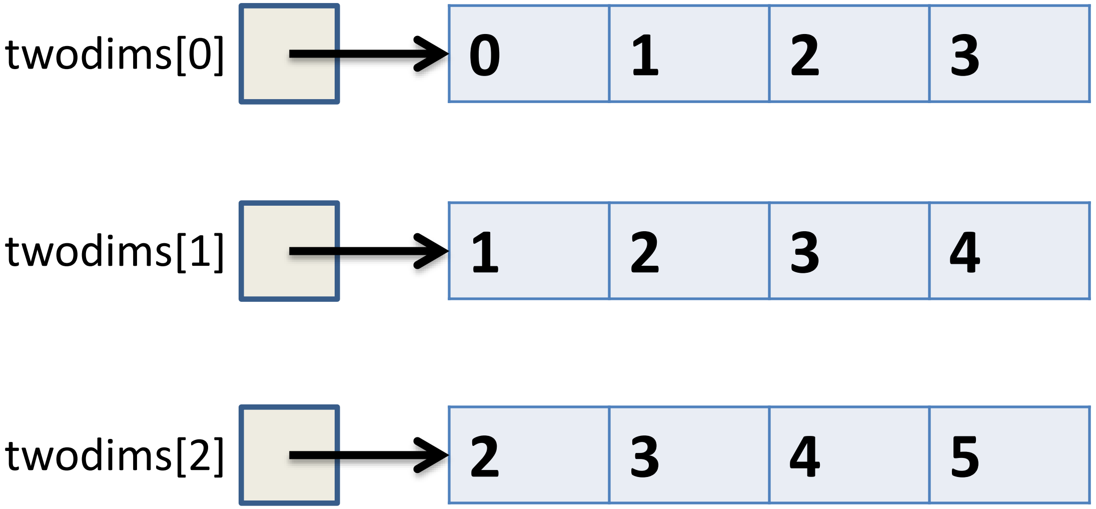
Two-dimensional Arrays

```
int twodims[3][4];
for(i=0; i<3; i++) {
  for(j=0; j<4; j++) {
     twodims[i][j] = i+j;
   }
}
```



Two-dimensional Arrays: Matrix

```
int twodims[3][4];
for(i=0; i<3; i++) {
  for(j=0; j<4; j++) {
     twodims[i][j] = i+j;
   }
}
```



twodims[0] 0 1 2 3

twodims[1] 1 2 3 4

twodims[2] 2 3 4 5

Memory Layout

- Matrix: 3 rows, 4 columns

0	**1**	**2**	**3**
1	**2**	**3**	**4**
2	**3**	**4**	**5**

Row Major Order:
all Row 0 buckets, followed by
all Row 1 buckets, followed by
all Row 2 buckets, …

`0xf260`	0	`twodim[0][0]`
`0xf264`	1	`twodim[0][1]`
`0xf268`	2	`twodim[0][2]`
`0xf26c`	3	`twodim[0][3]`
`0xf270`	1	`twodim[1][0]`
`0xf274`	2	`twodim[1][1]`
`0xf278`	3	`twodim[1][2]`
`0xf27c`	4	`twodim[1][3]`
`0xf280`	2	`twodim[2][0]`
`0xf284`	3	`twodim[2][1]`
`0xf288`	4	`twodim[2][2]`
`0xf28c`	5	`twodim[2][3]`

Memory Layout

- Matrix: 3 rows, 4 columns

0	**1**	**2**	**3**
1	**2**	**3**	**4**
2	**3**	**4**	**5**

`twodim[1][3]:`

base addr + row offset + col offset

```
twodim + 1*ROWSIZE*4 + 3*4

0xf260 + 16 + 12 = 0xf27c
```

`0xf260`	0	`twodim[0][0]`
`0xf264`	1	`twodim[0][1]`
`0xf268`	2	`twodim[0][2]`
`0xf26c`	3	`twodim[0][3]`
`0xf270`	1	`twodim[1][0]`
`0xf274`	2	`twodim[1][1]`
`0xf278`	3	`twodim[1][2]`
`0xf27c`	4	`twodim[1][3]`
`0xf280`	2	`twodim[2][0]`
`0xf284`	3	`twodim[2][1]`
`0xf288`	4	`twodim[2][2]`
`0xf28c`	5	`twodim[2][3]`

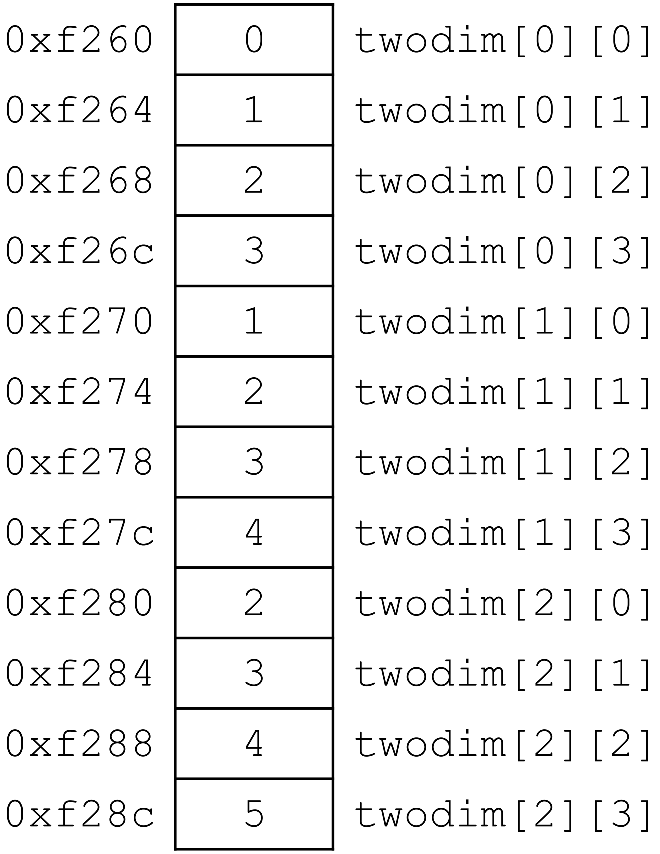
If we declared `int matrix[5][3];`, and the base of matrix is 0x3420, what is the address of `matrix[3][2]`?

A. 0x3438
B. 0x3440
C. 0x3444
D. 0x344C
E. None of these

Dynamic Two-dimensional Array

- Given the *row-major order* layout, a "two-dimensional array" is still just a contiguous block of memory:

- The malloc function returns… a pointer to a contiguous block of memory!



0xf260	0	twodim[0][0]
0xf264	1	twodim[0][1]
0xf268	2	twodim[0][2]
0xf26c	3	twodim[0][3]
0xf270	1	twodim[1][0]
0xf274	2	twodim[1][1]
0xf278	3	twodim[1][2]
0xf27c	4	twodim[1][3]
0xf280	2	twodim[2][0]
0xf284	3	twodim[2][1]
0xf288	4	twodim[2][2]
0xf28c	5	twodim[2][3]

Dynamic Two-dimensional Array

- For this example, with three rows and four columns:

0	1	2	3
1	2	3	4
2	3	4	5

```
int *matrix = malloc(3 * 4 * sizeof(int));
```

Caveat: the C compiler doesn't know that you're planning to use this block of memory with more one index (i.e., row and column).

Can't access: `matrix[i][j]`

0xf260	0	matrix[?]
0xf264	1	matrix[?]
0xf268	2	matrix[?]
0xf26c	3	matrix[?]
0xf270	1	matrix[?]
0xf274	2	matrix[?]
0xf278	3	matrix[?]
0xf27c	4	matrix[?]
0xf280	2	matrix[?]
0xf284	3	matrix[?]
0xf288	4	matrix[?]
0xf28c	5	matrix[?]

Dynamic Two-dimensional Array

- For this example, with three rows and four columns:

0	**1**	**2**	**3**
1	**2**	**3**	**4**
2	**3**	**4**	**5**

```
int *matrix = malloc(3 * 4 * sizeof(int));

// Compute the offset manually
index = i * ROWSIZE + j;
matrix[index] = …
```

0xf260	0	matrix[0 + 0]
0xf264	1	matrix[0 + 1]
0xf268	2	matrix[0 + 2]
0xf26c	3	matrix[0 + 3]
0xf270	1	matrix[4 + 0]
0xf274	2	matrix[4 + 1]
0xf278	3	matrix[4 + 2]
0xf27c	4	matrix[4 + 3]
0xf280	2	matrix[8 + 0]
0xf284	3	matrix[8 + 1]
0xf288	4	matrix[8 + 2]
0xf28c	5	matrix[8 + 3]

Two-dimensional array alternative

- (Dynamically) Allocate an array of pointers.
  For each pointer, (dynamically) allocate an array.

- How do we get an array of pointers?

Two-dimensional array alternative

- If we want a dynamic array of ints:
  - declare `int *array = malloc(N * sizeof(int))`

- So… if we want an array of int pointers:
  - declare `int **array = malloc(…)`

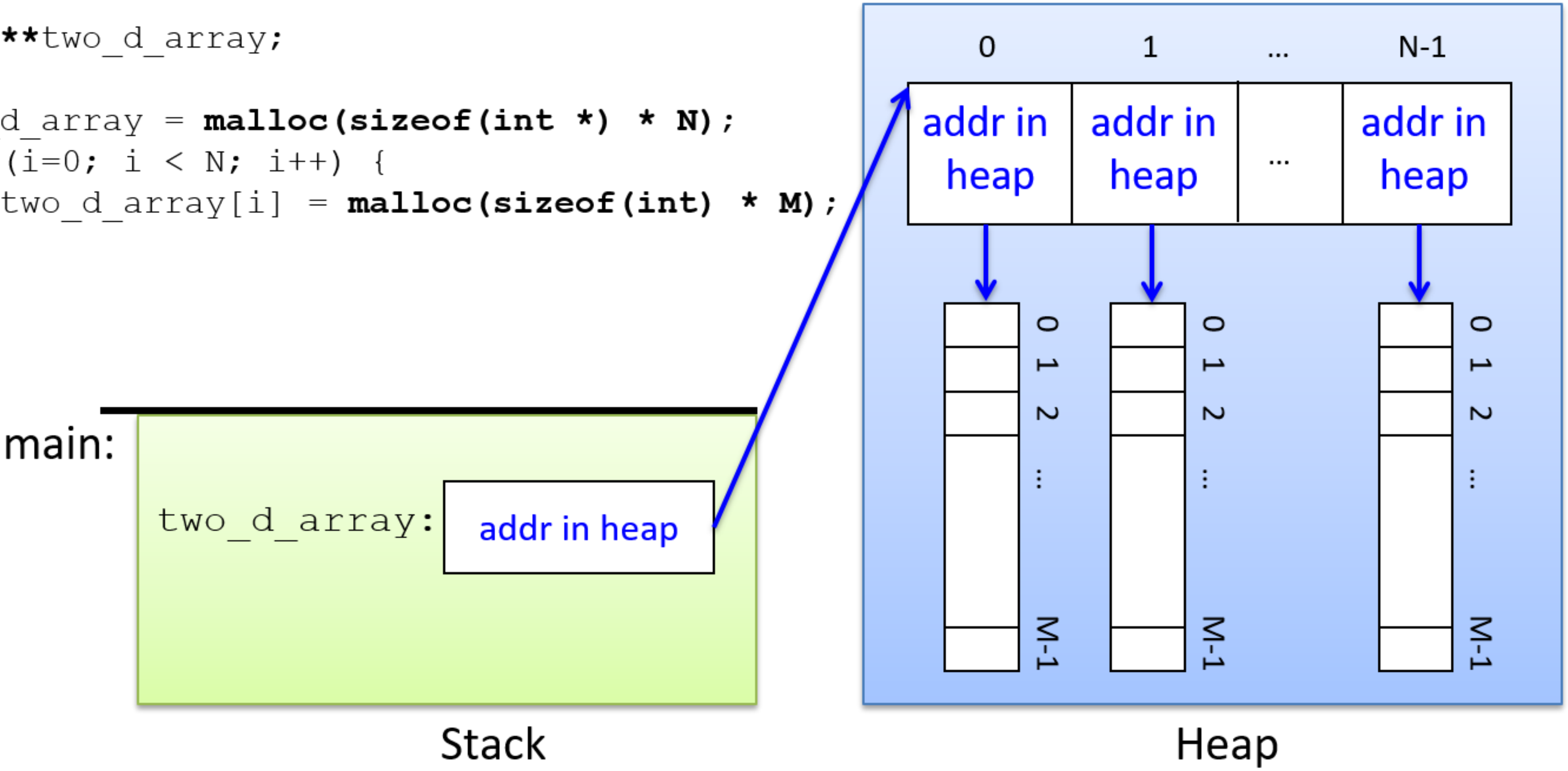
Two-dimensional array alternative

- If we want a dynamic array of ints:
  - declare `int *array = malloc(N * sizeof(int))`

- So… if we want an array of int pointers:
  - declare `int **array = malloc(N * sizeof(int *))`
  - The type of array[0], array[1], etc. is: `int *`
  - For each one of those, we can malloc an array of ints:
    - `array[0] = malloc(M * sizeof(int))`

Two-dimensional array alternative

```
int **two_d_array;

two_d_array = malloc(sizeof(int *) * N);
for (i=0; i < N; i++) {
    two_d_array[i] = malloc(sizeof(int) * M);
}
```



Two-dimensional arrays

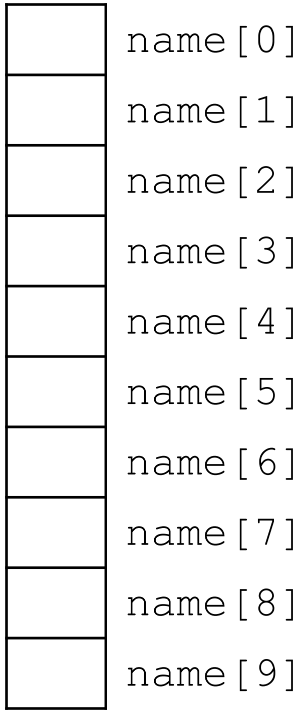
- We'll use BOTH methods in future labs.

Strings

- Strings are *character arrays*

- Layout is the same as:
  - char name[10];

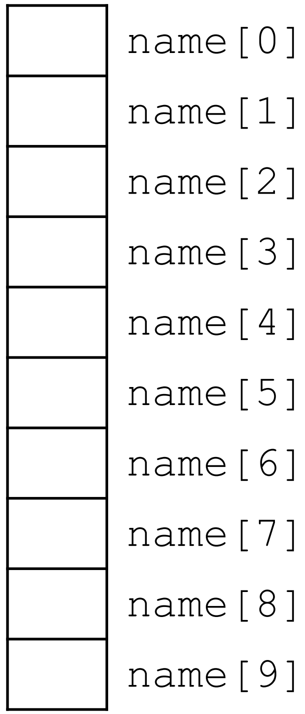
- Often accessed as (char *)



String Functions

- C library has many built-in functions that operate on char *’s:
  - strcpy, strdup, strlen, strcat, strcmp, strstr

```
char name[10];
strcpy(name, “CS 31”);
```

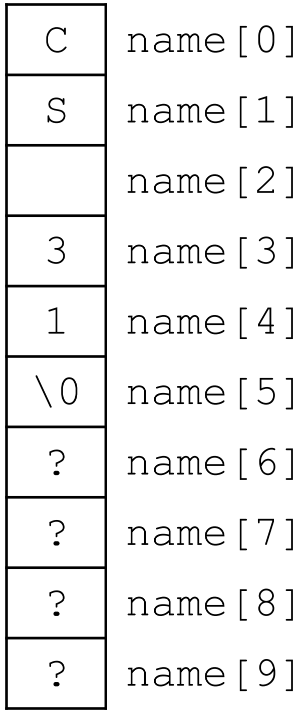


String Functions

- C library has many built-in functions that operate on char *’s:
  - strcpy, strdup, strlen, strcat, strcmp, strstr

```
char name[10];
strcpy(name, “CS 31”);
```

- Null terminator (\0) ends string.
  - We don’t know/care what comes after



C	name[0]
S	name[1]
	name[2]
3	name[3]
1	name[4]
\0	name[5]
?	name[6]
?	name[7]
?	name[8]
?	name[9]

String Functions

- C library has many built-in functions that operate on char *’s:
  - strcpy, strdup, strlen, strcat, strcmp, strstr

- Seems simple on the surface.
  - That null terminator is tricky, strings error-prone.
  - Strings used everywhere!

- You will ~~implement~~ use these functions in a future lab.

Up next…

- New topic: Storage and the Memory Hierarchy