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

Kevin Webb
Swarthmore College
October 23, 2018

Reading Quiz

Overview

- Accessing *things* via an offset
  - Arrays, Structs, Unions
- 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

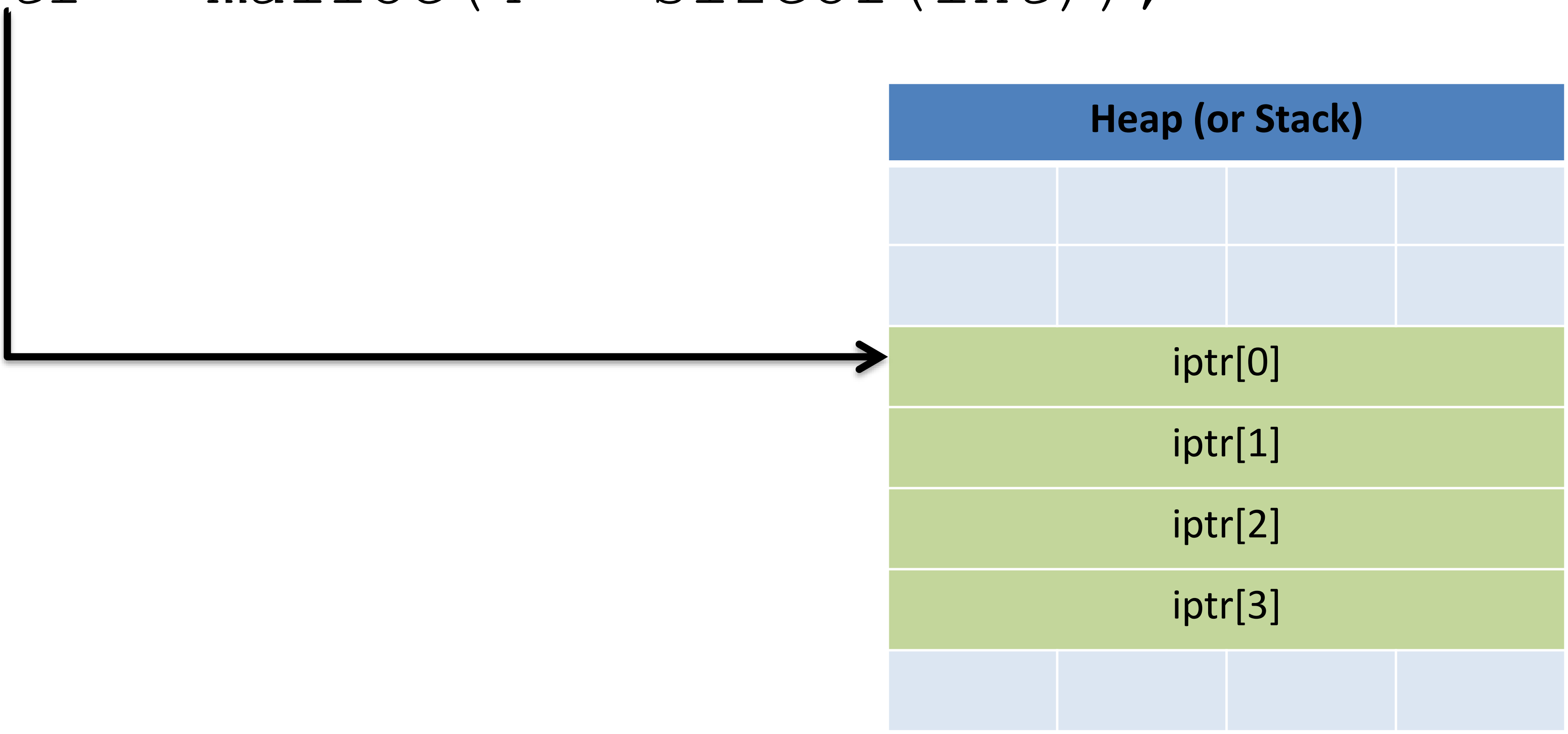
- Combination of one or more existing types into a new type. (e.g., an array of *multiple* ints, or a struct)

- 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));
```



Recall: Assembly While Loop

```
    movl $0 eax
    movl $0 edx
loop:
    addl (%ecx), %eax
    addl $4, %ecx
    addl $1, %edx
    cmpl $5, %edx
    jne loop
```

Using (*dereferencing*) the memory address to access memory at that location.

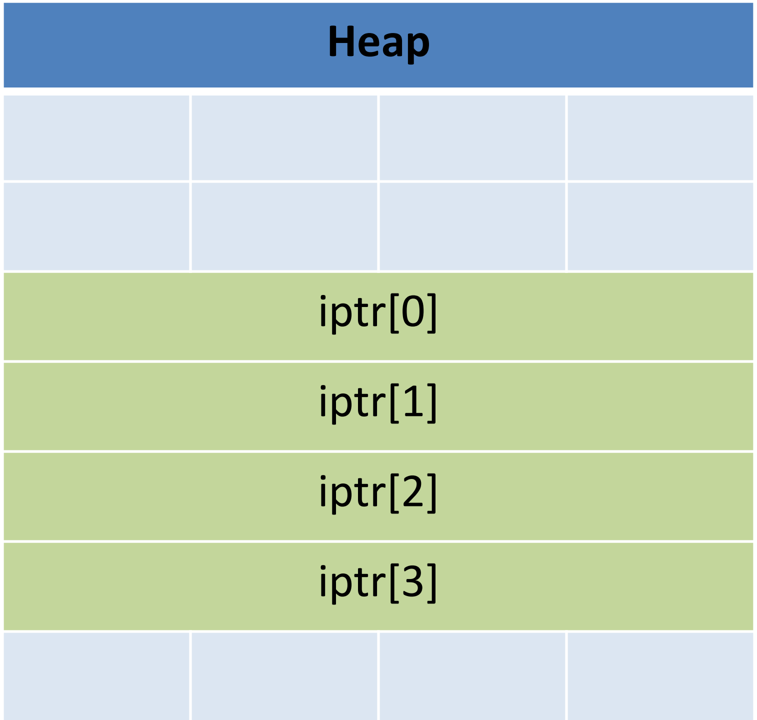
Manipulating the pointer to point to something else.

Note: This did NOT read or write the memory that is pointed to.

Pointer Manipulation: Necessary?

- Previous example: advance %ecx to point to next item in array.

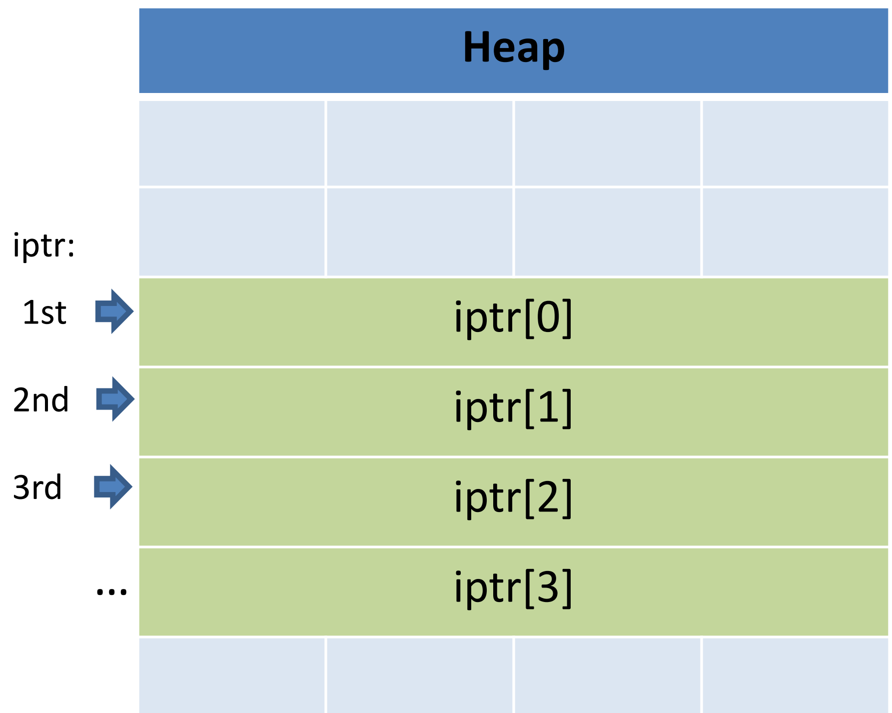
```
iptr = malloc(…);
sum = 0;
while (i < 4) {
  sum += *iptr;
  iptr += 1;
  i += 1;
}
```



Pointer Manipulation: Necessary?

- Previous example: advance %ecx to point to next item in array.

```
iptr = malloc(…);
sum = 0;
while (i < 4) {
   sum += *iptr;
   iptr += 1;
   i += 1;
}
```



Reminder: addition on a pointer advances by that many of the type (e.g., ints), not bytes.

Pointer Manipulation: Necessary?

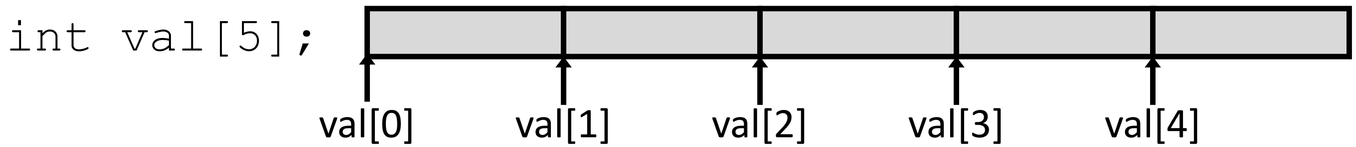
- Problem: `iptr` is changing!
- What if we wanted to free it?
- What if we wanted something like this:

```
iptr = malloc(…);
sum = 0;
while (i < 4) {
   sum += iptr[0] + iptr[i];
   iptr += 1;
   i += 1;
}
```

Changing the pointer would be really inconvenient now!

Base + Offset

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

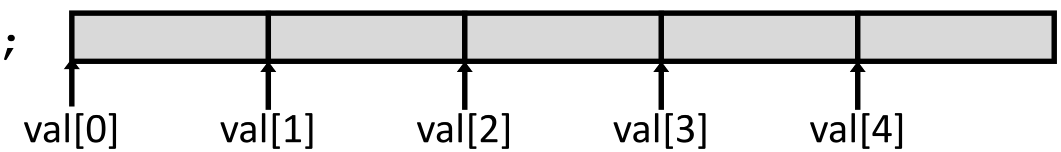


- "We're goofy computer scientists who count starting from zero."

Base + Offset

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

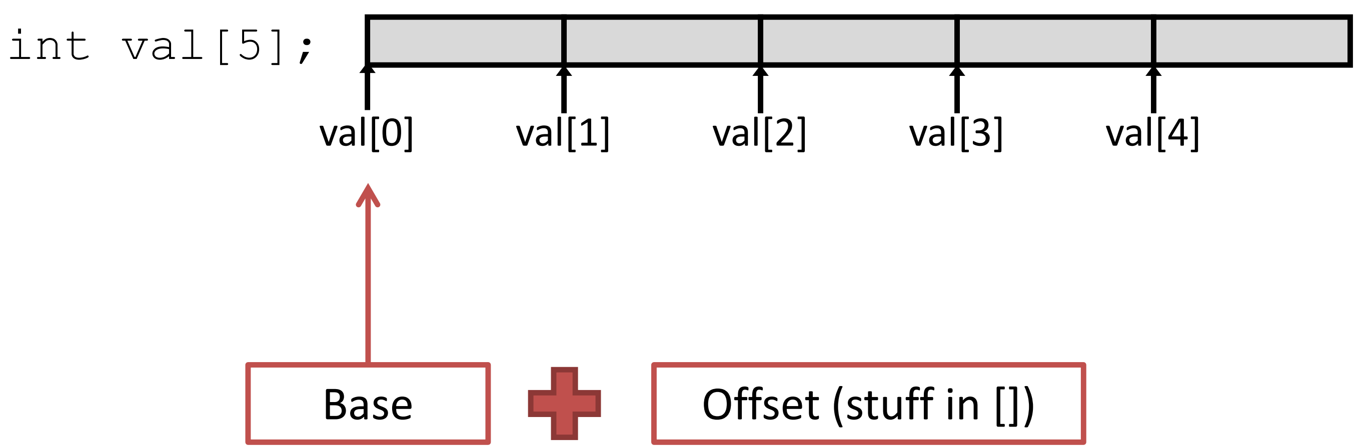
`int val[5];`



- ~~"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]

Indexed Addressing Mode

- What we'd like in IA32 is to express accesses like iptr[N], where iptr doesn't change – it's a base.
- Displacement mode works, if we know which offset to use at *compile time*:
  - Variables on the stack: -4(%ebp)
  - Function arguments: 8(%ebp)
  - Accessing [5] of an integer array: 20(%base_register)
- If we only know at run time?
  - How do we express i(%ecx)?

Indexed Addressing Mode

- General form:

  displacement(%base, %index, scale)

- Translation: Access the memory at address…
  - base + (index * scale) + displacement

- Rules:
  - Displacement can be any 1, 2, or 4-byte value
  - Scale can be 1, 2, 4, or 8.

Example

ECX: Array base address

Registers:

%ecx	0x0824
%edx	2

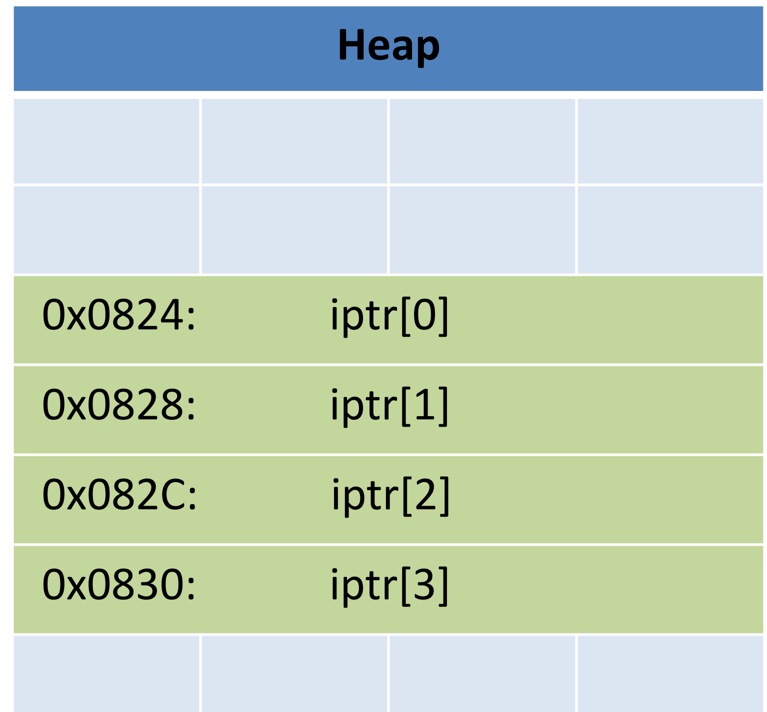
Suppose i is at %ebp - 8, and equals 2.

User says:

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

Translates to:

```
movl -8(%ebp), %edx
```



Example

Suppose i is at %ebp - 8, and equals 2.

Registers:

%ecx	0x0824
%edx	2

User says:

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

Translates to:

```
movl -8(%ebp), %edx
```

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

Example

Suppose i is at %ebp - 8, and equals 2.

Registers:

%ecx	0x0824
%edx	2

User says:

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

Translates to:

```
movl -8(%ebp), %edx
movl $9, (%ecx, %edx, 4)
```

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

Example

Suppose i is at %ebp - 8, and equals 2.

Registers:

%ecx	0x0824
%edx	2

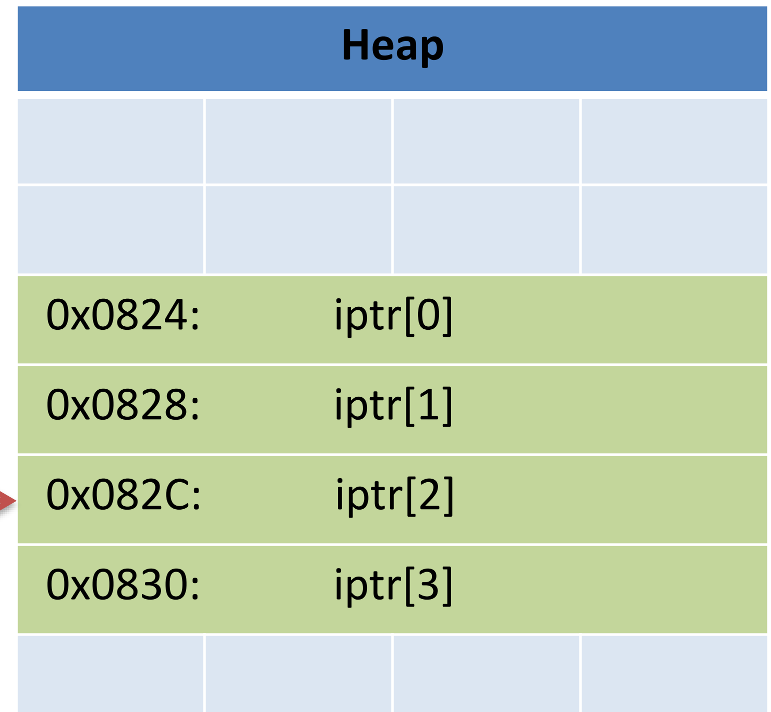
User says:

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

Translates to:

```
movl -8(%ebp), %edx
movl $9, (%ecx, %edx, 4)
```

```
0x0824 + (2 * 4) + 0
0x0824 + 8 = 0x082C
```



What is the final state after this code?

```
addl $4, %eax
movl (%eax), %eax
sall $1, %eax
movl %edx, (%ecx, %eax, 2)
```

(Initial state)
Registers:

%eax	0x2464
%ecx	0x246C
%edx	7

Memory:

Heap	
0x2464:	5
0x2468:	1
0x246C:	42
0x2470:	3
0x2474:	9

Indexed Addressing Mode

- General form:

  displacement(%base, %index, scale)

- You might see some of these in your maze.

Two-dimensional Arrays

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

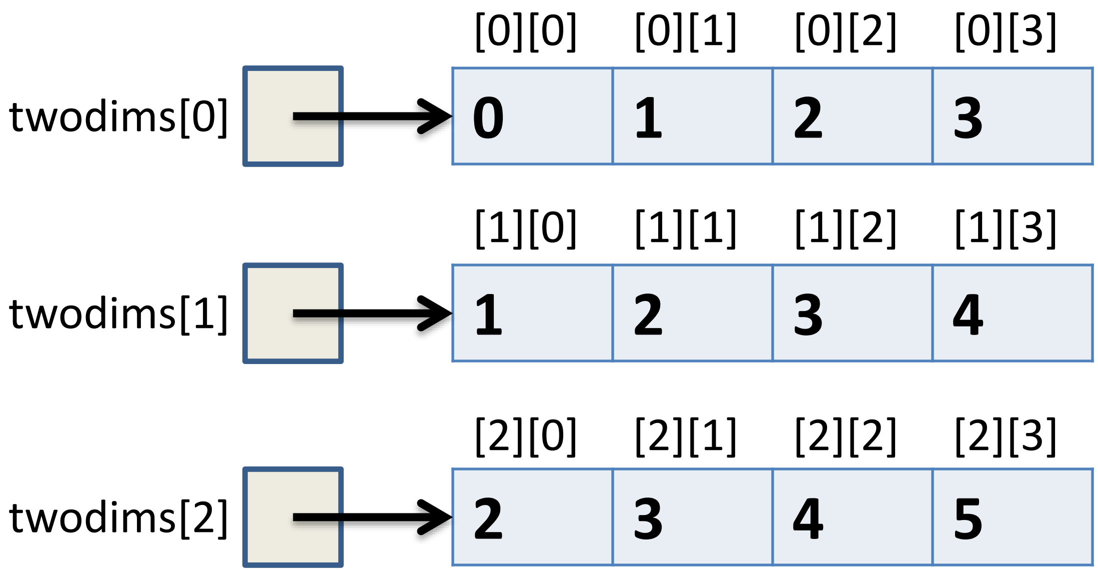
```
int twodims[3][4];
```

- "Give me three sets of four integers."

- How should these be organized in memory?

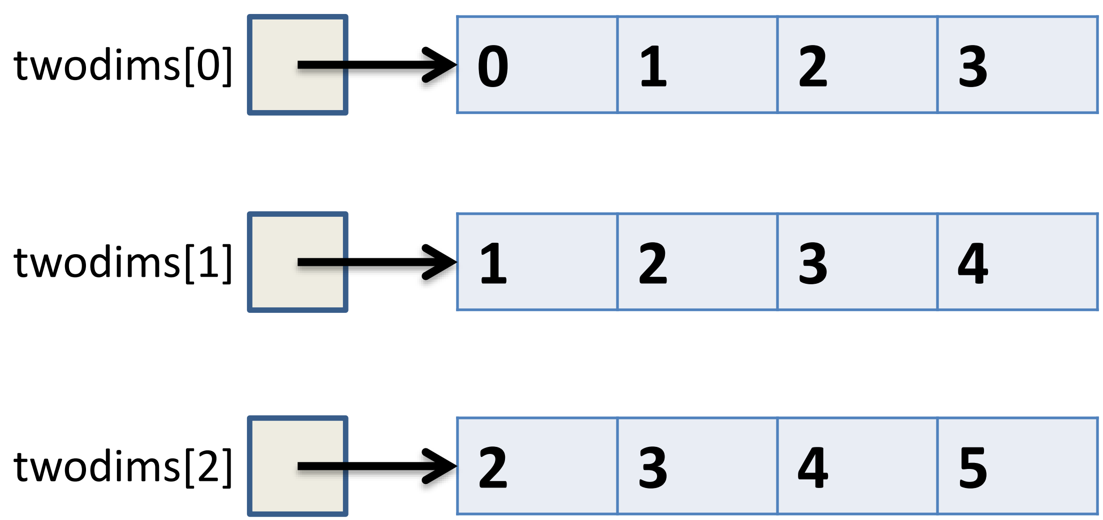
Two-dimensional Arrays

```
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;
   }
}
```



Memory Layout

- Matrix: 3 rows, 4 columns

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

<u>Row Major Order</u>:
all Row 0 buckets,
followed by
all Row 1 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

Composite Data Types

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

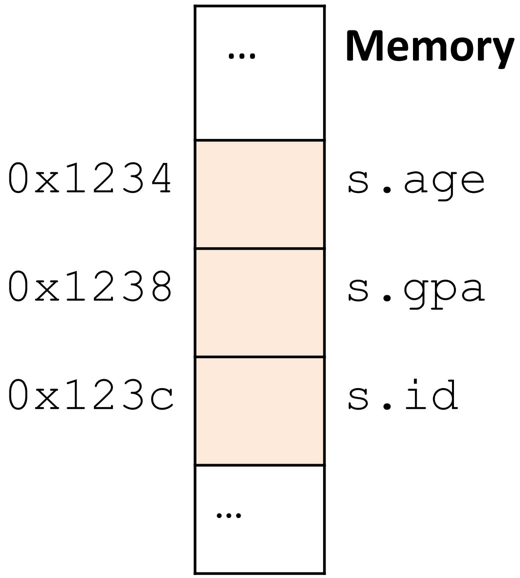
- Example: a queue
  - Might need a value (int) plus a link to the next item (pointer)

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

Structs

- Laid out contiguously by field
  - In order of field declaration.

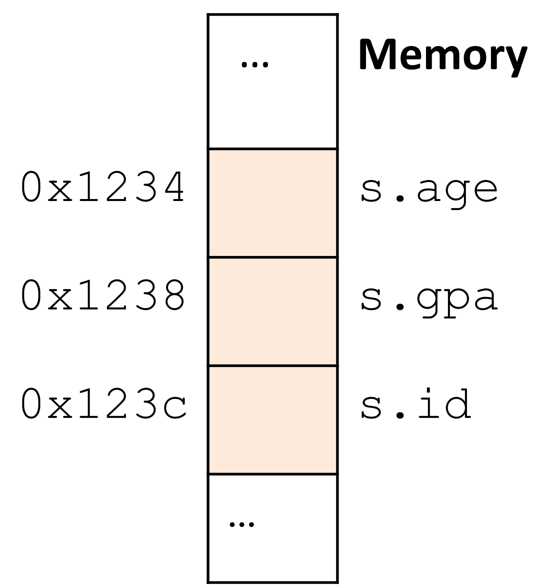
```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
```



Structs

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

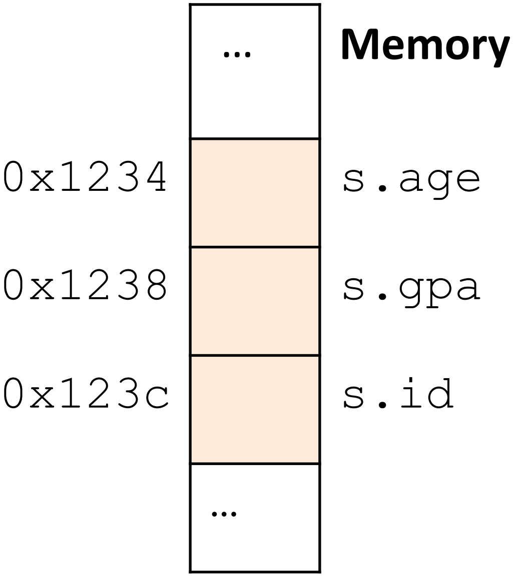
```
struct student{
  int age;
  float gpa;
  int id;
};

struct student s;
```



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;
```



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.

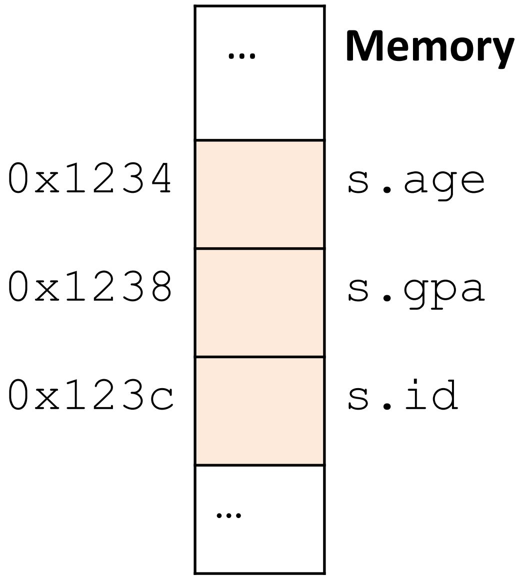
Data Alignment: Why?

- 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?

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

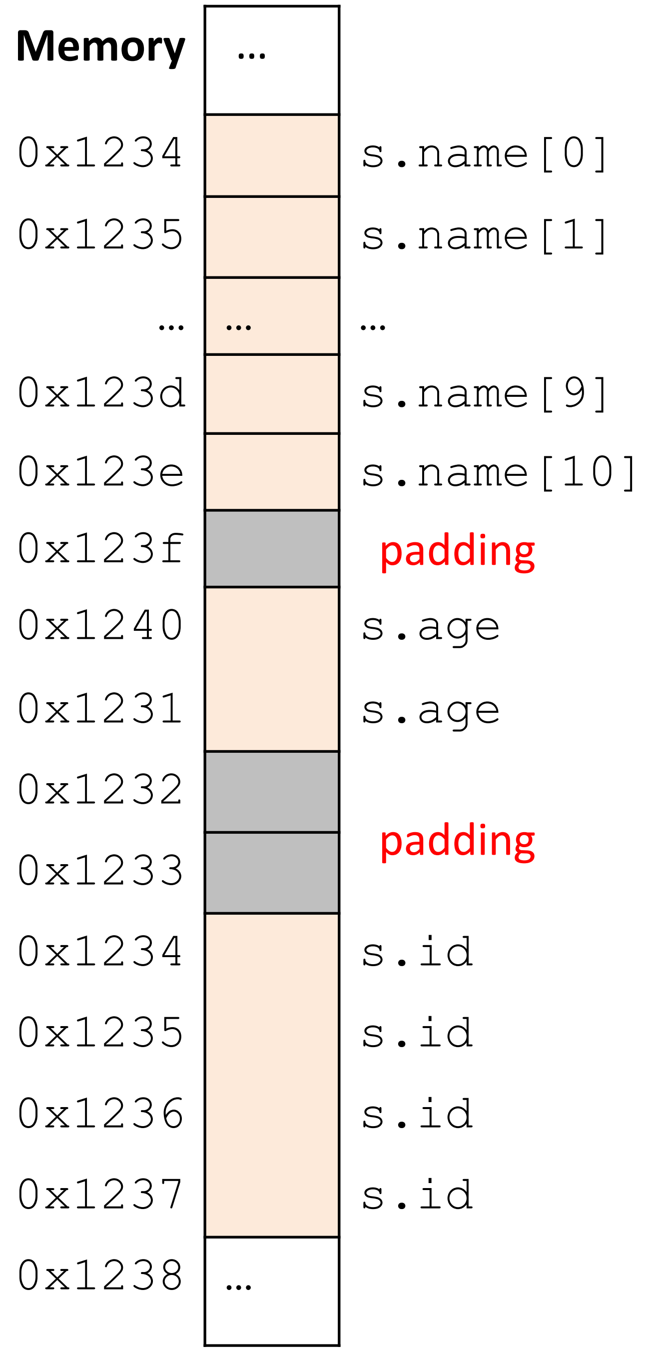
A. 17 bytes
B. 18 bytes
C. 20 bytes
D. 22 bytes
E. 24 bytes

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>



Alternative Layout

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

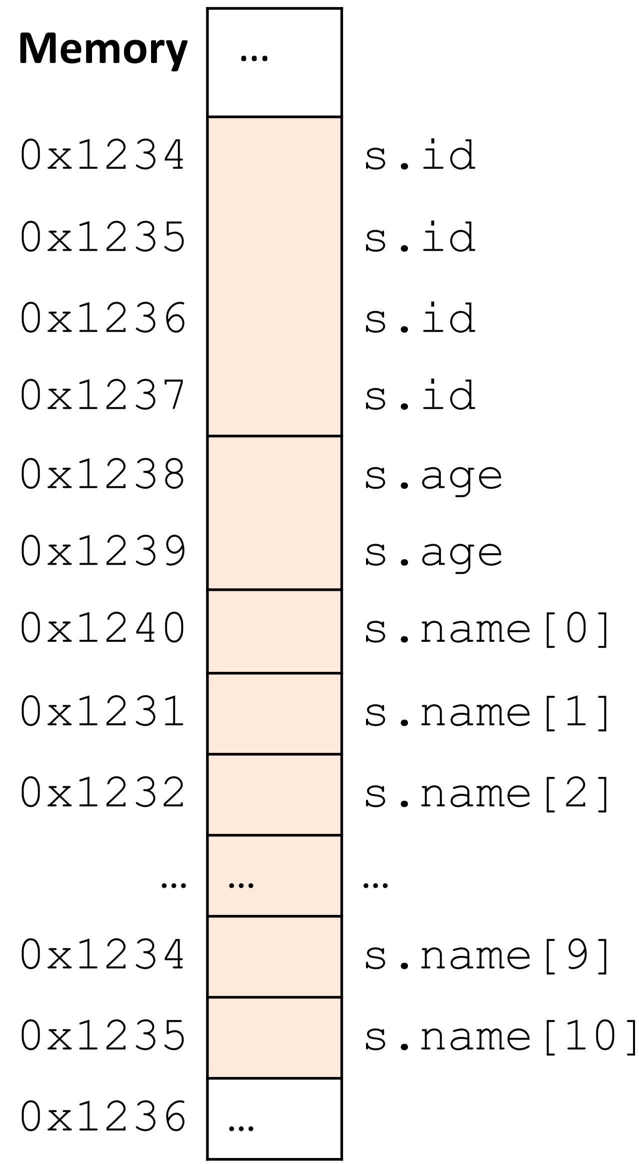
Same fields, declared in a different order.

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.



Cool, so we can get rid of this 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;
};
```

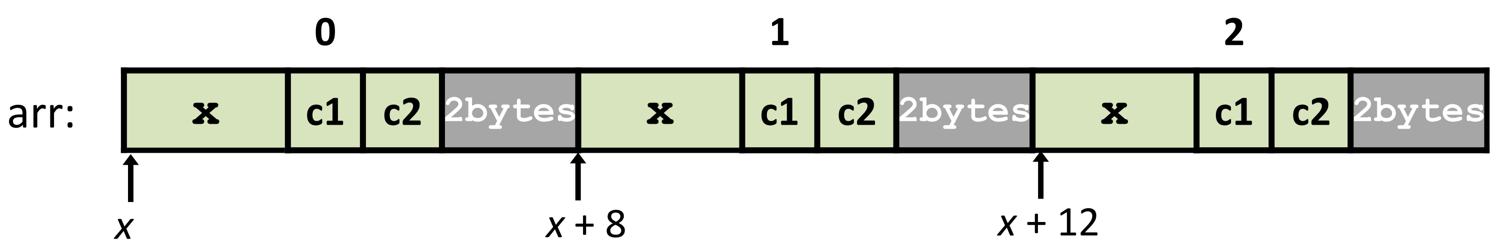
```
struct T2 {
    int x;
    char c1;
    char c2;
};
```

"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

A note on struct syntax…

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

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

A note on struct 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];`

Unions

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

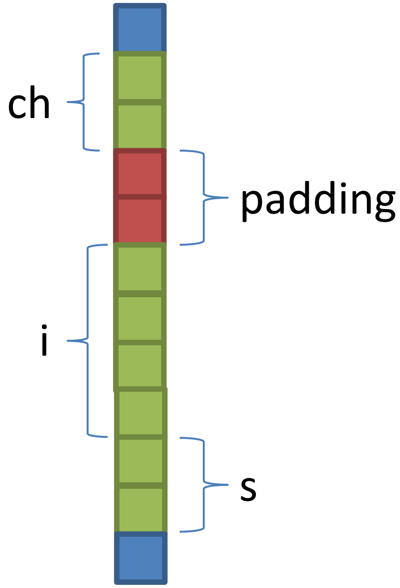
- Struct: field 1 <u>and</u> field 2 <u>and</u> field 3 …
- Union: field 1 <u>or</u> field 2 <u>or</u> field 3 …

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

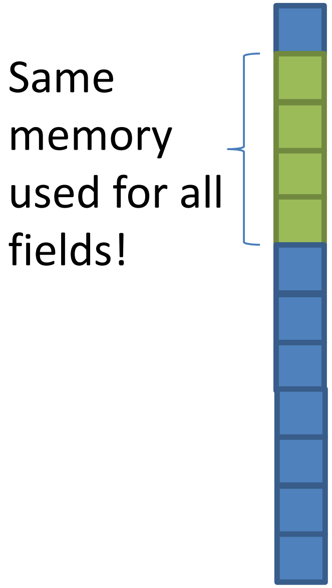
Unions

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

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



my_struct in memory



my_union in memory

Unions

```
my_union u;

u.i = 7;
```

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

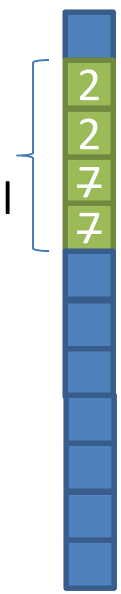
my_union in memory

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

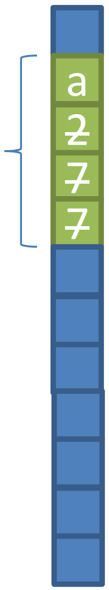
Unions

```
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;
```

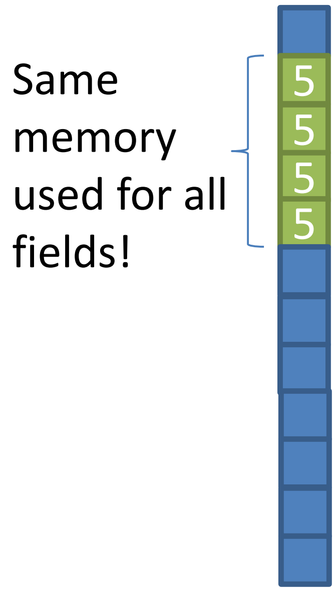
```
union my_union {
        char ch[2];
        int i;
        short s;
}
```

my_union in memory

Unions

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

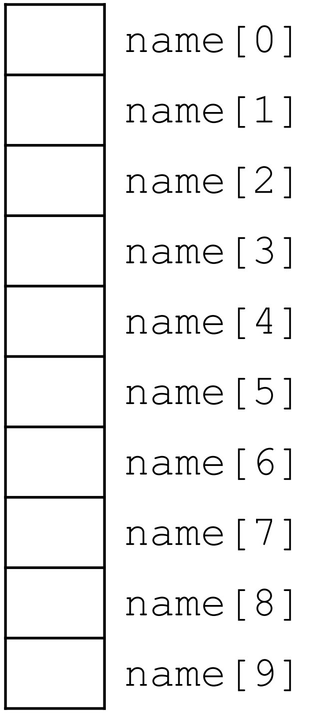
```
union my_union {
      char ch[2];
      int i;
      short s;
}
```



my_union in memory

Strings

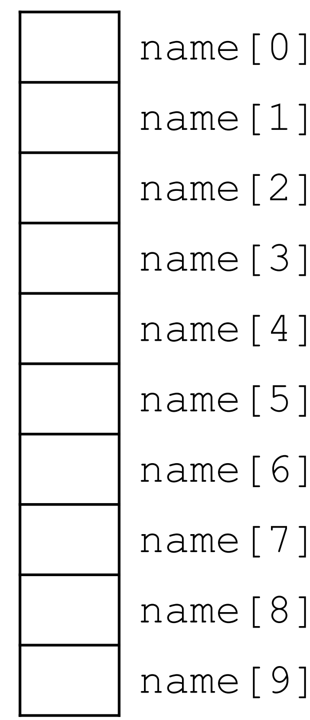
- Strings are *character arrays*
- Layout is the same as:
  - char name[10];
- Often accessed as (char *)



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");
```

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]

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

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 these functions in a future lab.