CS 31: Introduction to Computer Systems

13-14: Arrays, Pointers

March 24

SWARTHMORE COLLEGE

Recall: Arrays

- C's support for collections of values
  - Array buckets store a single type of value
  - Specify max capacity (num buckets) when you declare an array variable (single memory chunk)

Recall: Arrays

Static Allocation:

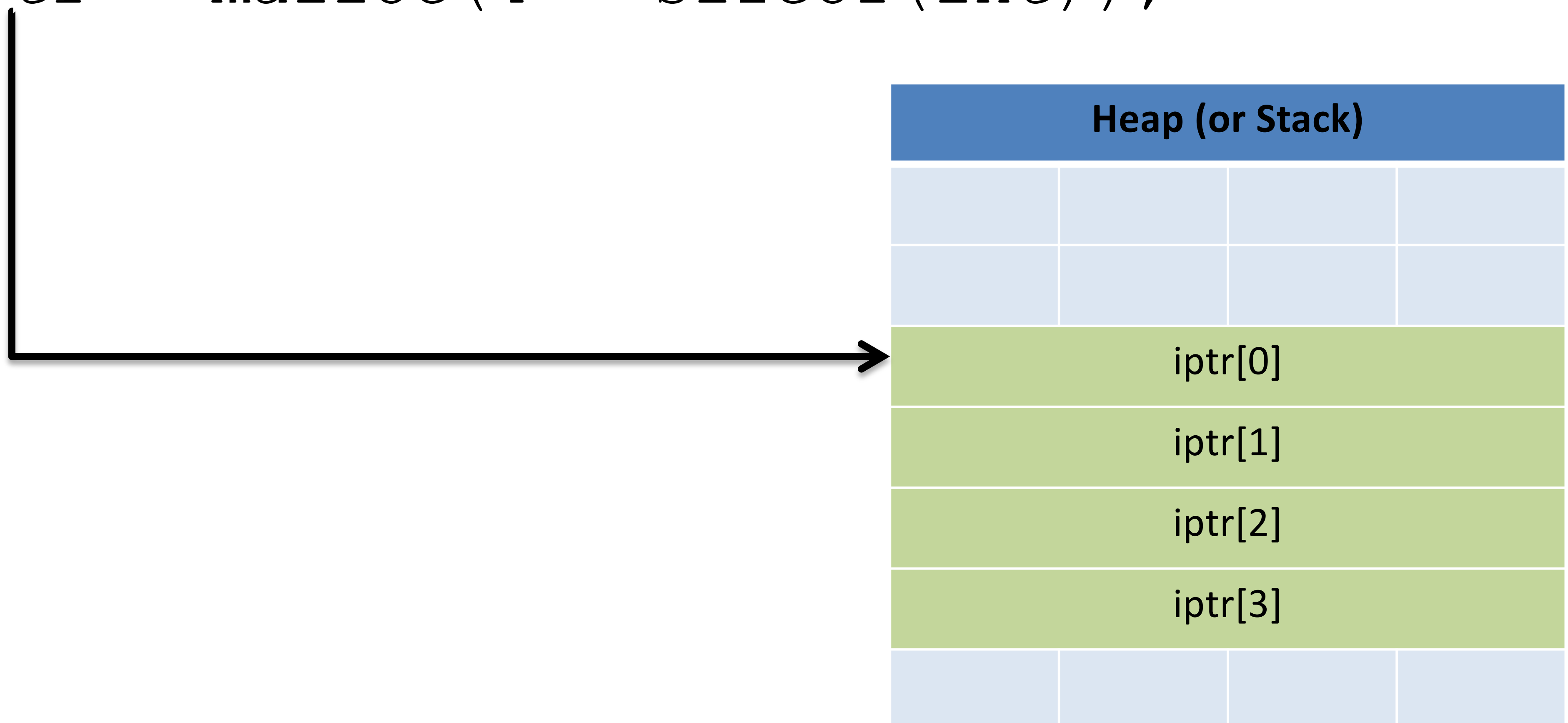
```
<type> <var_name>[<num buckets>]

int arr[5];
// an array of 5 integers

float rates[40];
// an array of 40 floats
```

Dynamic Allocation:

```
<type> <var_name>[<num buckets>]

int * arr =
malloc(sizeof(int)*5);
// an array of 5 integers

//initialize array
//free array
free(arr);
```

Recall: Pointers as Arrays

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

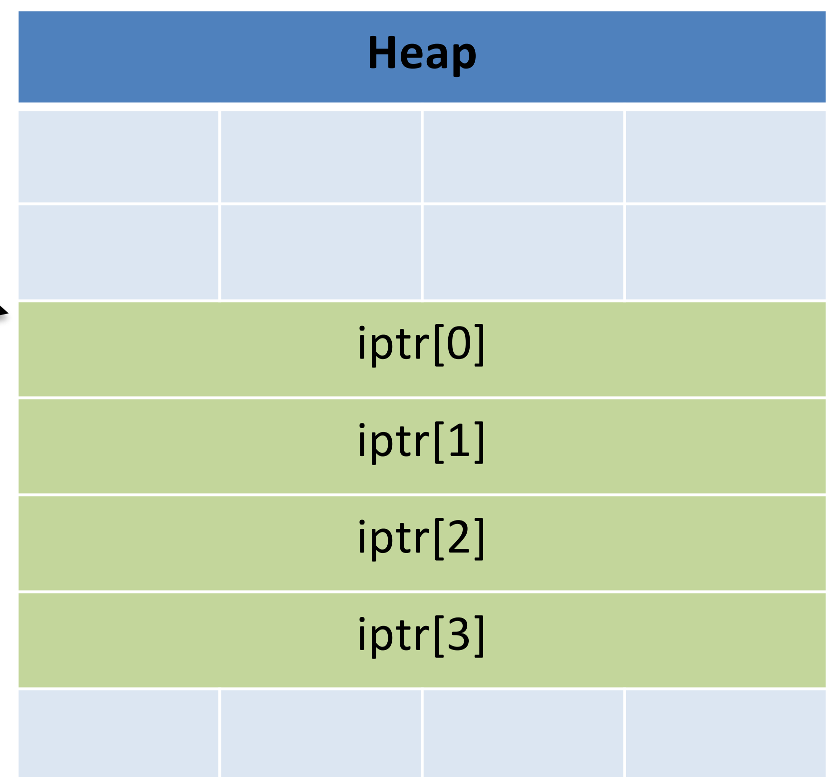


Pointers as Arrays

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

1. Start from the base of iptr.

```
iptr[2] = 7;
```



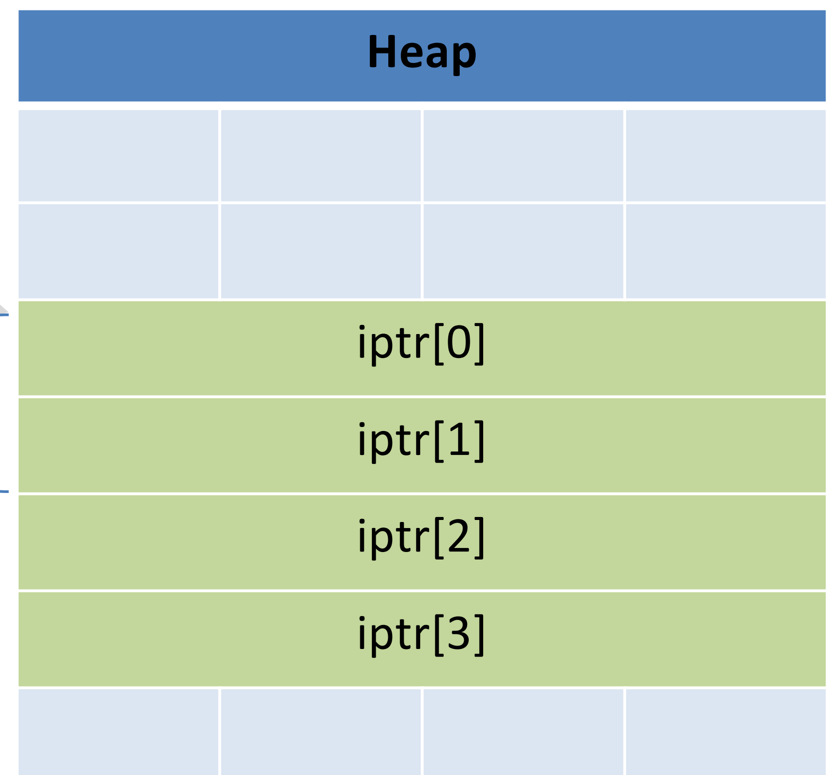
Pointers as Arrays

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

1. Start from the base of iptr.

```
iptr[2] = 7;
```

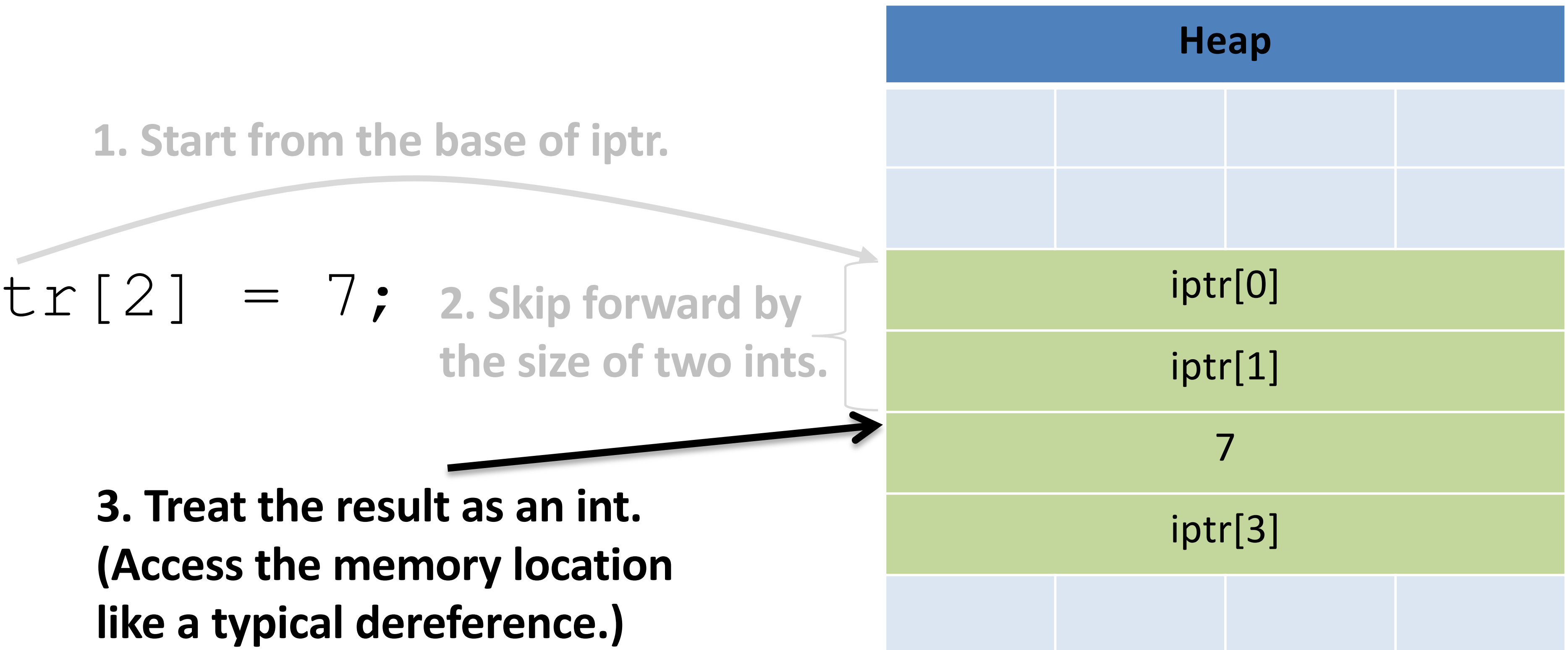
2. Skip forward by the size of two ints.



Pointers as Arrays

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

```
iptr[2] = 7;
```

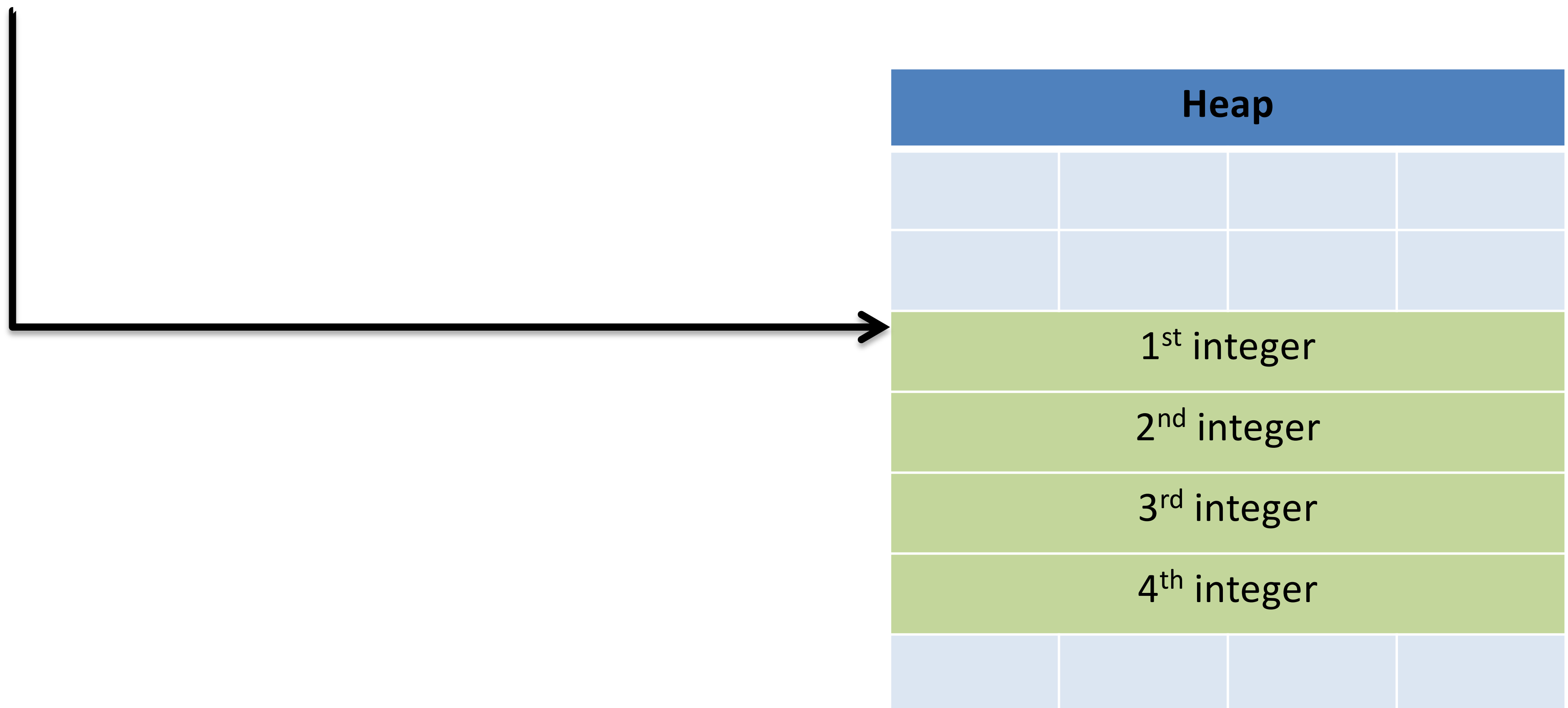


Pointer Arithmetic

- Addition and subtraction work on pointers.
- C automatically increments by the size of the type that's pointed to.

Pointer Arithmetic

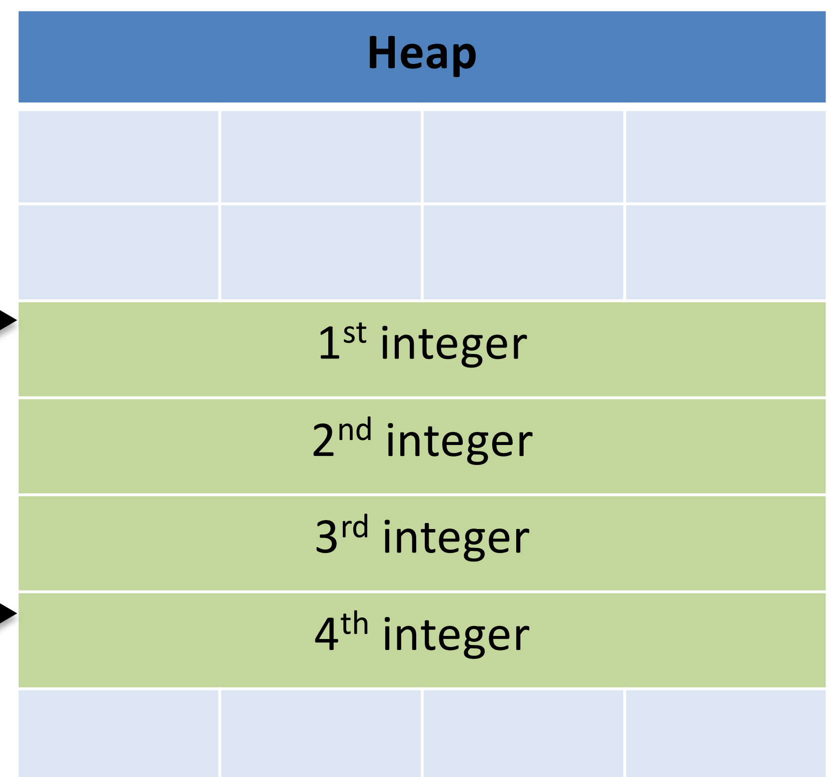
```
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
```



Pointer Arithmetic

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

```
int *iptr2 = iptr + 3;
```



Skip ahead by 3 times the size of iptr's type (integer, size: 4 bytes).

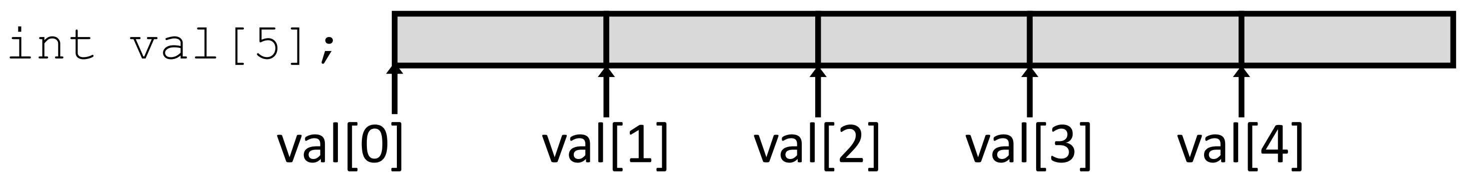
Pointer Arithmetic

- Addition and subtraction work on pointers.

- C automatically increments by the size of the type that's pointed to.

Base + Offset

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

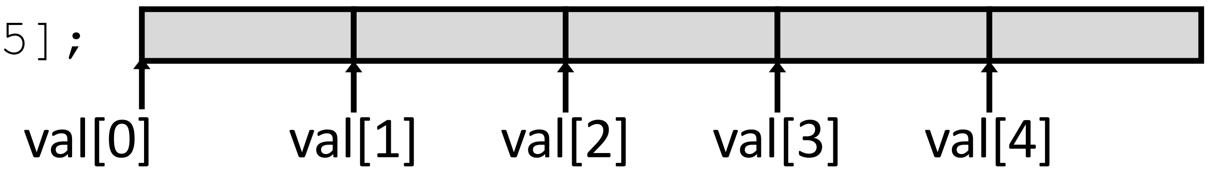


- “We’re goofy computer scientists who count starting from zero.”

Base + Offset

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

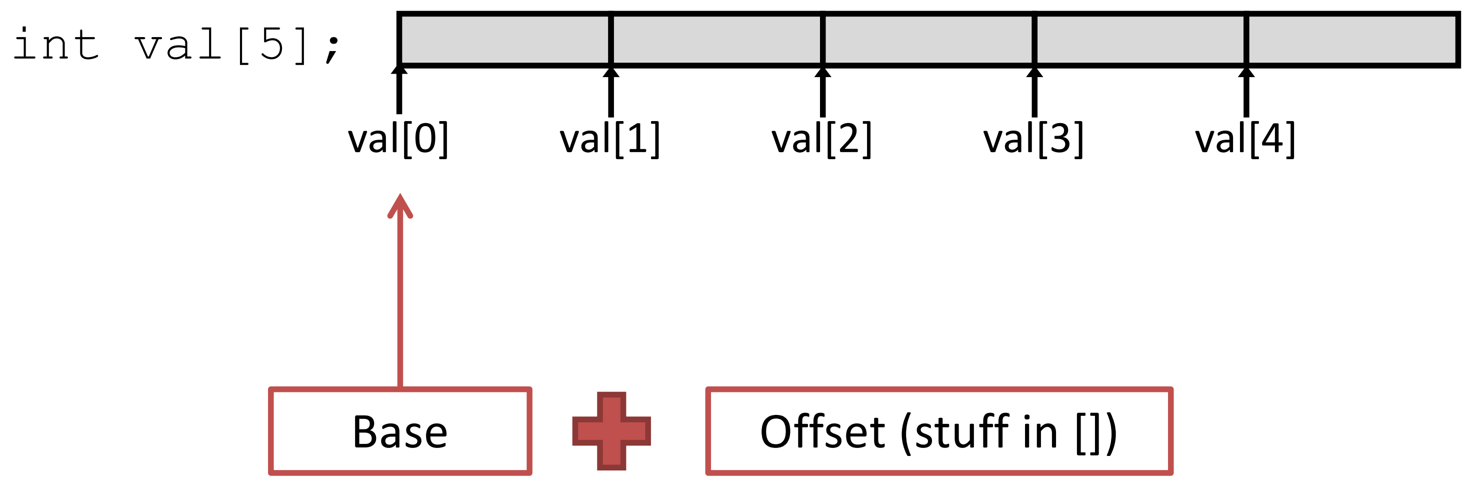
`int val[5];`



- ~~“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…

What is the memory address stored in iptr2?

```
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
int *iptr2 = iptr + 3;
```

A. Mem. address in iptr + 12 bytes
B. Mem. address in iptr + 3 bytes
C. Mem. address in iptr + 4 bytes
D. None of the above

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

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]		

What is the memory address stored in iptr2?

```
int *iptr = NULL;
iptr = malloc(4 * sizeof(int));
int *iptr2 = iptr + 3;
```

A. Mem. address in iptr + 12 bytes (3 buckets of size int)
B. Mem. address in iptr + 3 bytes
C. Mem. address in iptr + 4 bytes
D. None of the above

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

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]		

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

What if this isn't known at compile time?

A. 0x0824 + 3 * 4 (requires an extra multiplication step)

B. 0x0824 + 4 * 4

C. 0x0824 + 0xC

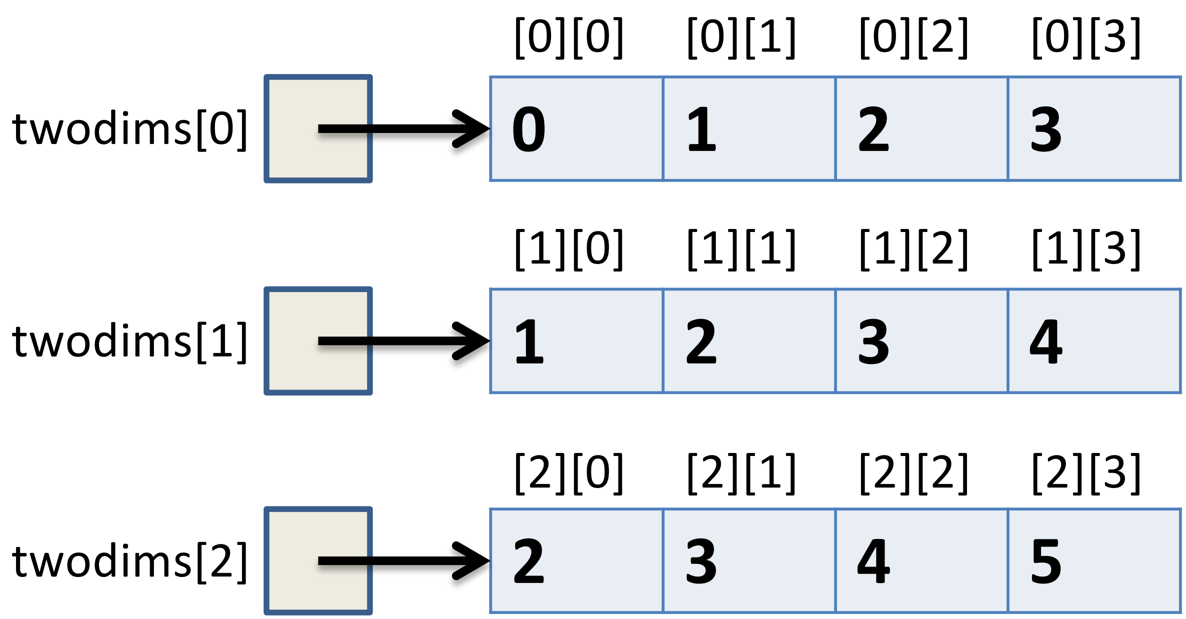
D. More than one (which?)

E. None of these

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

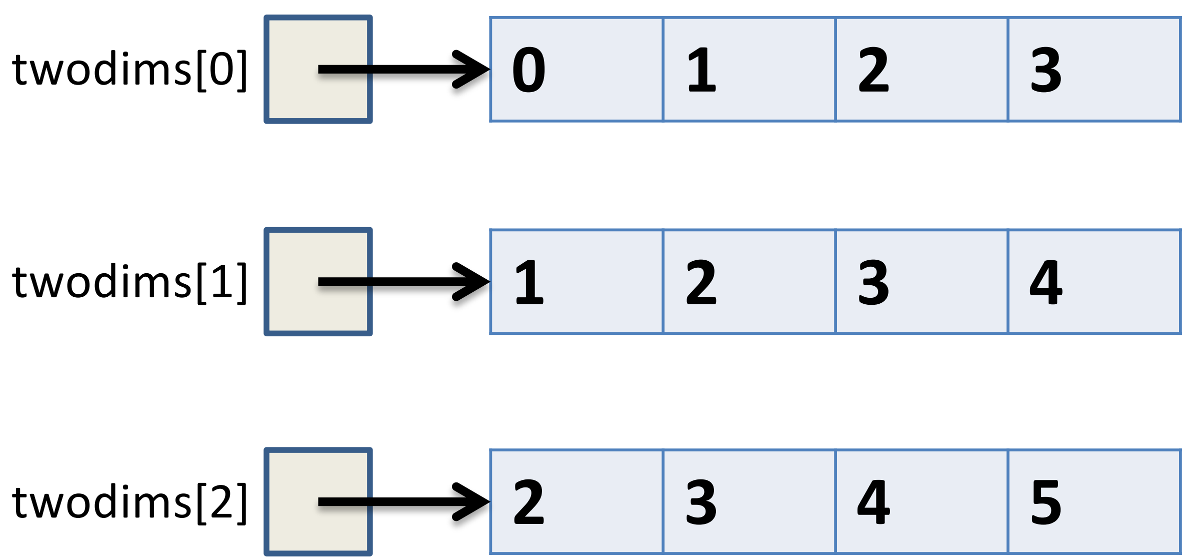
Two-dimensional Arrays

```
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**

Row Major Order:
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]:`

Find the memory index:

[row #][col #] = (row #) * ROWSIZE + col #
= 1 * 4 + 3
= 7

Address	Value	Element
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]:`

Converting mem index into a memory address:
= base_address + mem_index * sizeof(data)

base address = 0xf260 (hex)
mem index * sizeof(data) = 7*4 = 28 (decimal)
= 1c (hex)

= 0xf260 + 1c = 0xf27c

Address	Value	Element
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]

You do not need to convert mem index into an address for the lab!

Memory Layout

Matrix: 3 rows, 4 columns

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

`twodim[1][3]:`

Converting mem index into a memory address:
= base_address + mem_index * sizeof(data)

base address = 0xf260 (hex)
mem index * sizeof(data) = 7*4 = 28 (decimal)
= 1c (hex)

= 0xf260 + 1c = 0xf27c

Address	Value	Element
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]

You do not need to convert mem index into an address for the lab

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

Find the memory index:

[row #][col #] = (row #) * ROWSIZE + col #

Find the memory address:

base_address + mem_index * sizeof (datatype)

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

Find the memory index:
[row #][col #] = (row #) * ROWSIZE + col #

Find the memory address:
base_address + mem_index * sizeof (datatype)

Mem_index = 3*3+2 = 11
Mem. address = 0x3420 + 11*4 (2c) = 0x344c

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

Find the memory index:

[row #][col #] = (row #) * ROWSIZE + col #

Find the memory address:

base_address + mem_index * sizeof (datatype)

Mem_index = 3*3+2 = 11

Mem. address = 0x3420 + 11*4 (2c) = 0x344c

Composite Data Types

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

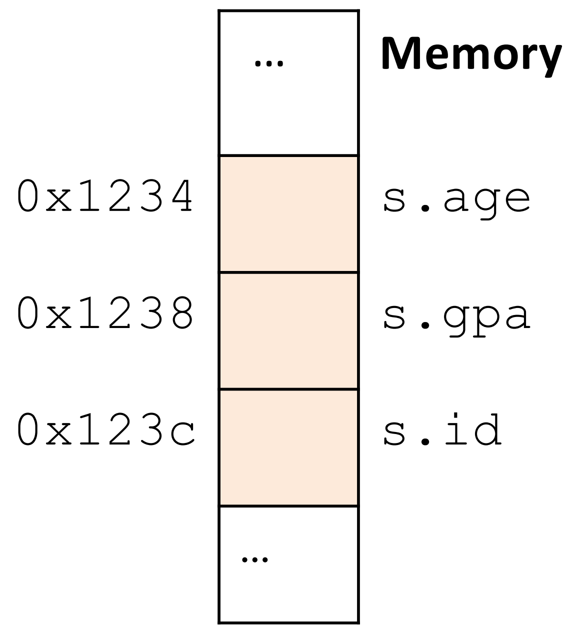
- Example: a queue
  - Might need a value (int) plus a link to the next item (pointer)

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

Structs

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

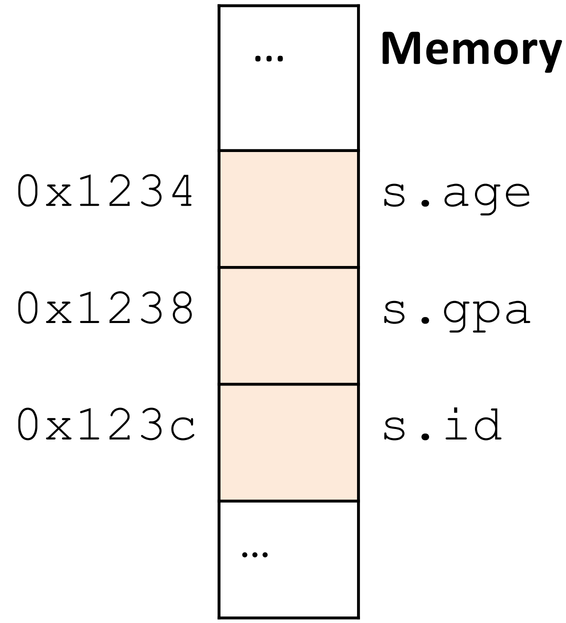
```
struct student{
   int age;
   float gpa;
   int id;
};

struct student s;
```



Structs

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

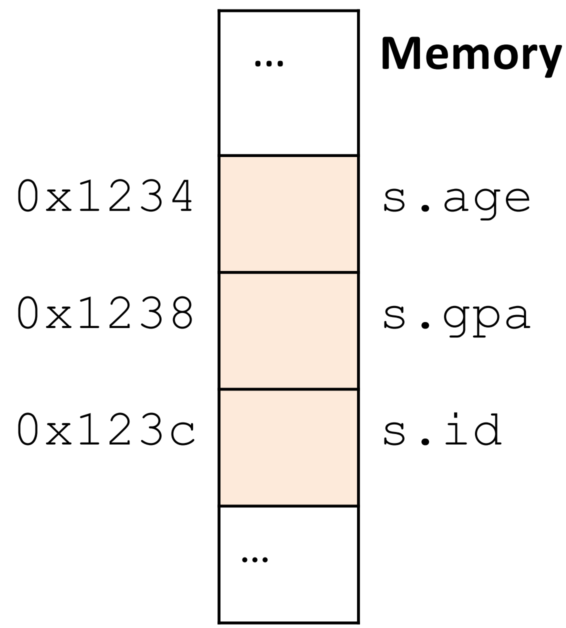
```
struct student{
   int age;
   float gpa;
   int id;
};

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.

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

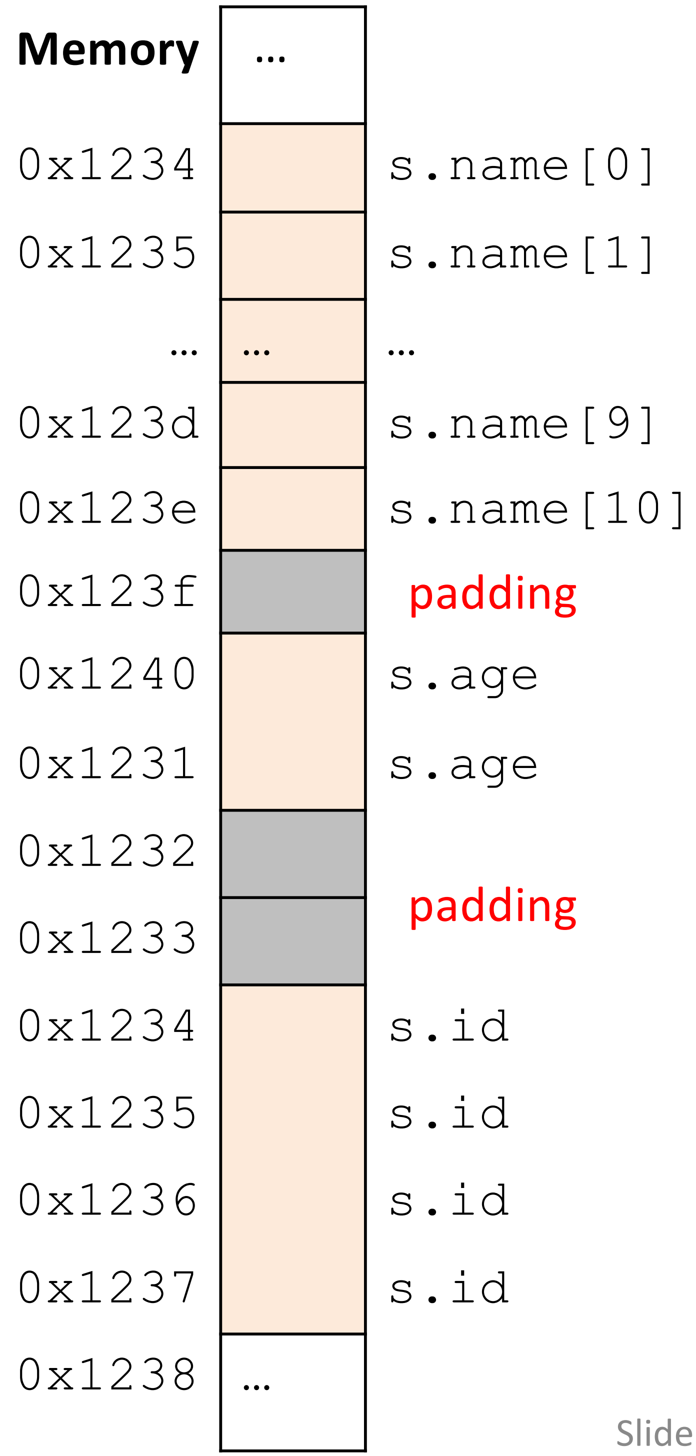
- Laid out contiguously by field
  - In order of field declaration.
  - May require some padding, for alignment

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

Structs

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

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



Use sizeof() when allocating structs with malloc()!

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	…	

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

```
struct student{
    char name[15];
     int id;
    short age;
 };
```

A. 17 bytes
B. 20 bytes
C. 21 bytes
D. 22 bytes
E. 24 bytes

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

A. Yes (why?)

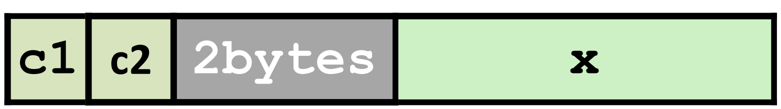
B. No (why not?)

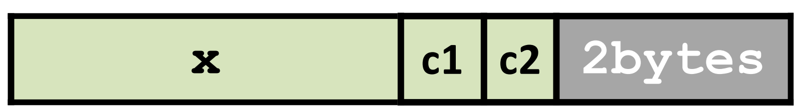
Cool, so we can get rid of this padding by being smart about declarations?

- Answer: Maybe.

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

```
struct T1 {                struct T2 {
    char c1;                     int x;
    char c2;                        char c1;
    int  x;                            char c2;
};                             };
```

T1:


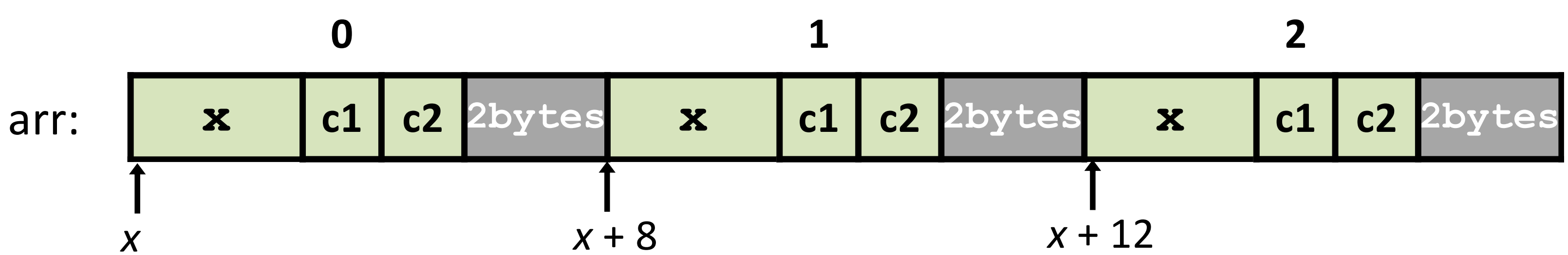
T2:


"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.

Arrays of Structs

```
struct student classroom[50];

strcpy(classroom[0].name, “Alice”);
classroom[0].grad_year = 2019;
classroom[0].gpa = 4.0;

strcpy(classroom[1].name, “Bob”);
classroom[1].grad_year = 2020;
classroom[1].gpa = 3.1

strcpy(classroom[2].name, “Cat”);
classroom[2].grad_year = 2021;
classroom[2].gpa = 3.4
```

Struct: Layout in Memory

`classroom:`

‘A’	‘l’	‘i’	‘c’	‘e’	‘\0’	…	‘B’	‘o’	‘b’	‘\0’	…	‘C’	‘a’	‘t’	‘\0’	…
2019							2020					2021				
4.0							3.1					3.4				

[0] [1] [2]

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.

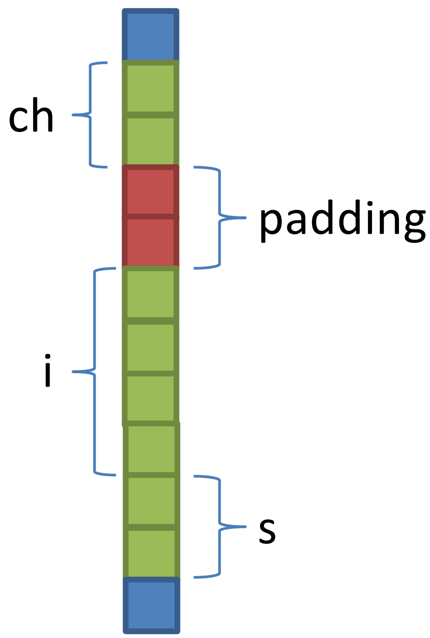
- Struct: field 1 and field 2 and field 3 …
- Union: field 1 or field 2 or field 3 …

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

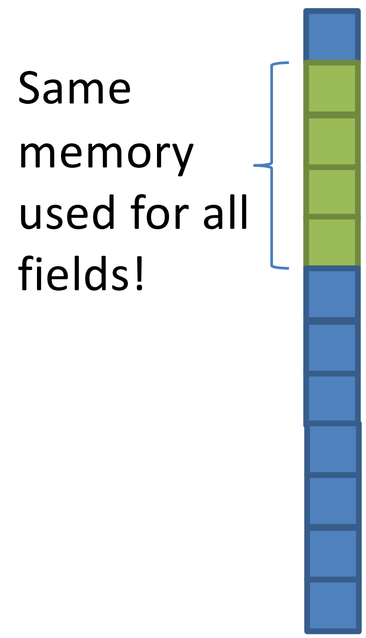
Unions

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

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



my_struct in memory

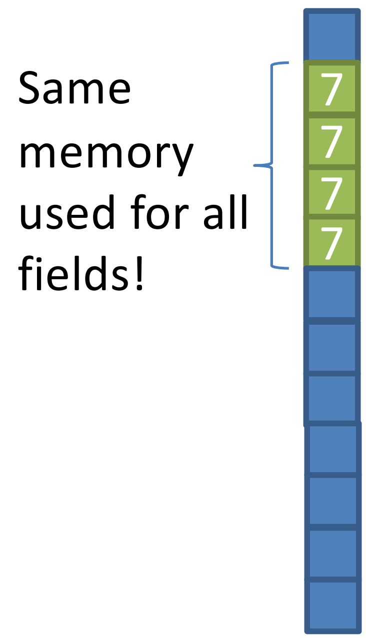


my_union in memory

Unions

```
my_union u;

u.i = 7;
```

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

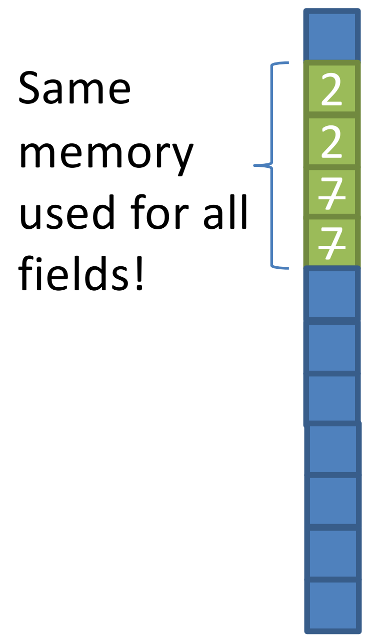


my_union in memory

Unions

```
my_union u;

u.i = 7;

u.s = 2;
```

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



my_union in memory

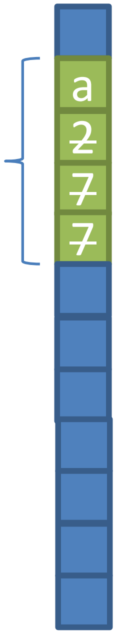
Unions

```
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

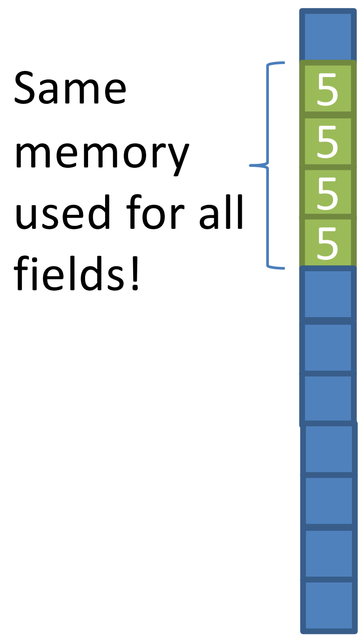
Unions

```
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;
}
```

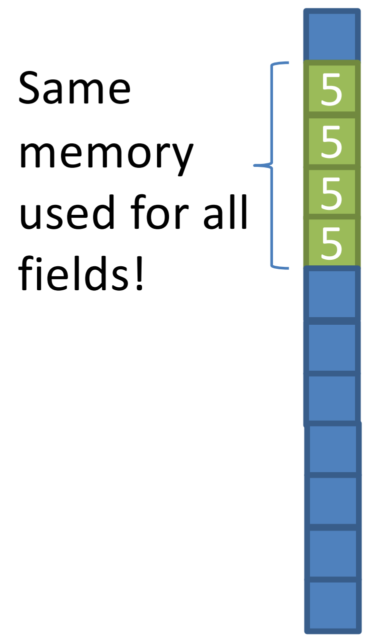


my_union in memory

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;
}
```



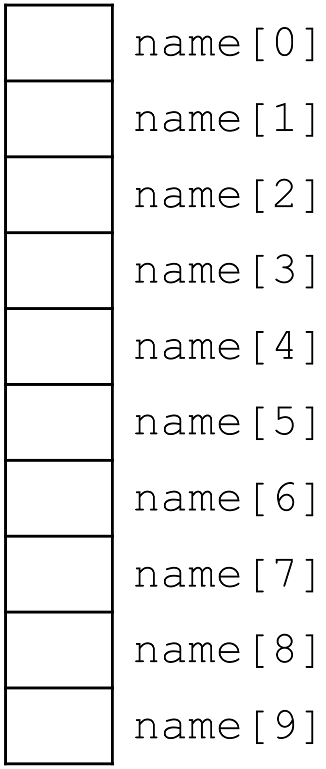
my_union in memory

Strings

- Strings are *character arrays*

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

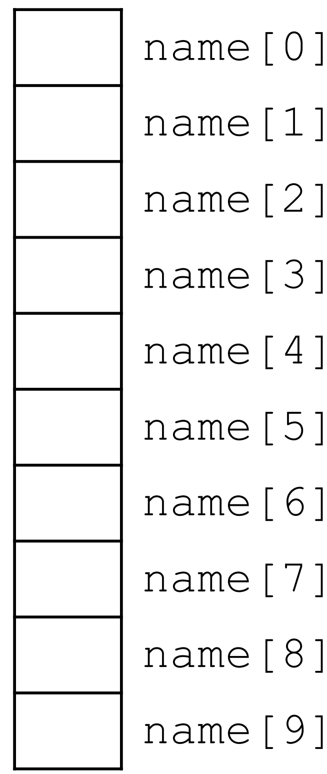
- Often accessed as (char *)



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!