CS 31: Introduction to Computer Systems

13-14: Arrays and Pointers

March 5

SWARTHMORE COLLEGE

Reading Quiz

Today

- 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)

structs

- Treat a collection of values as a single type:
  - C is not an object oriented language, no classes
  - A `struct` is like just the data part of a class

- Rules:
  1. Define a new struct type outside of any function
  2. Declare variables of the new struct type
  3. Use dot notation to access the different field values of the struct variable

Struct Example

Suppose we want to represent a *student* type.

```
struct student {
    char name[20];
    int grad_year;
    float gpa;
};
// Variable bob is of type struct student
struct student  bob;
// Set name (string) with strcpy()
strcpy(bob.name, “Robert Paulson”);
bob.grad_year = 2019;
bob.gpa = 3.1;

printf(“Name: %s, year: %d, GPA: %f”, bob.name,
bob.grad_year, bob.gpa);
```

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:

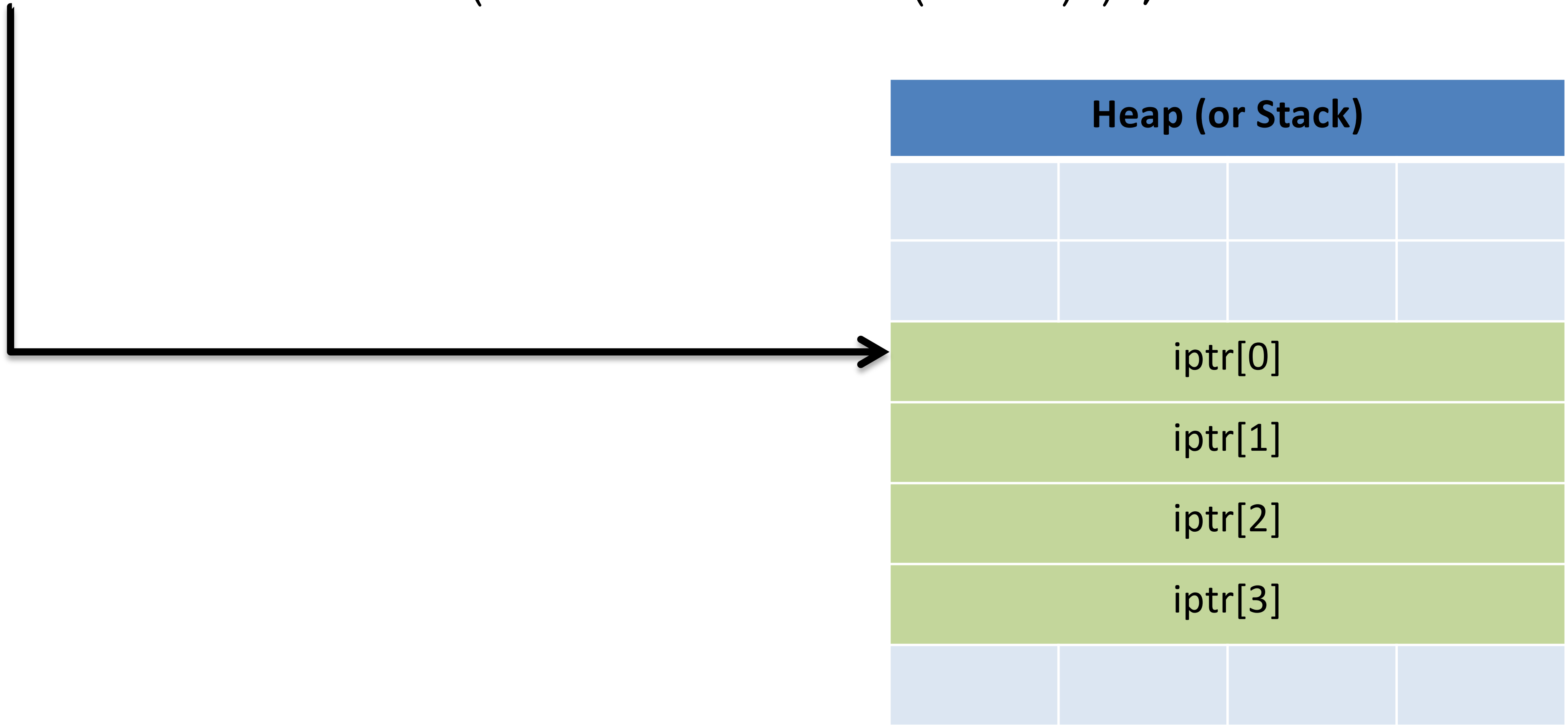
```
<type> <var_name>[<num buckets>]

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));
```



Heap (or Stack)

iptr[0]

iptr[1]

iptr[2]

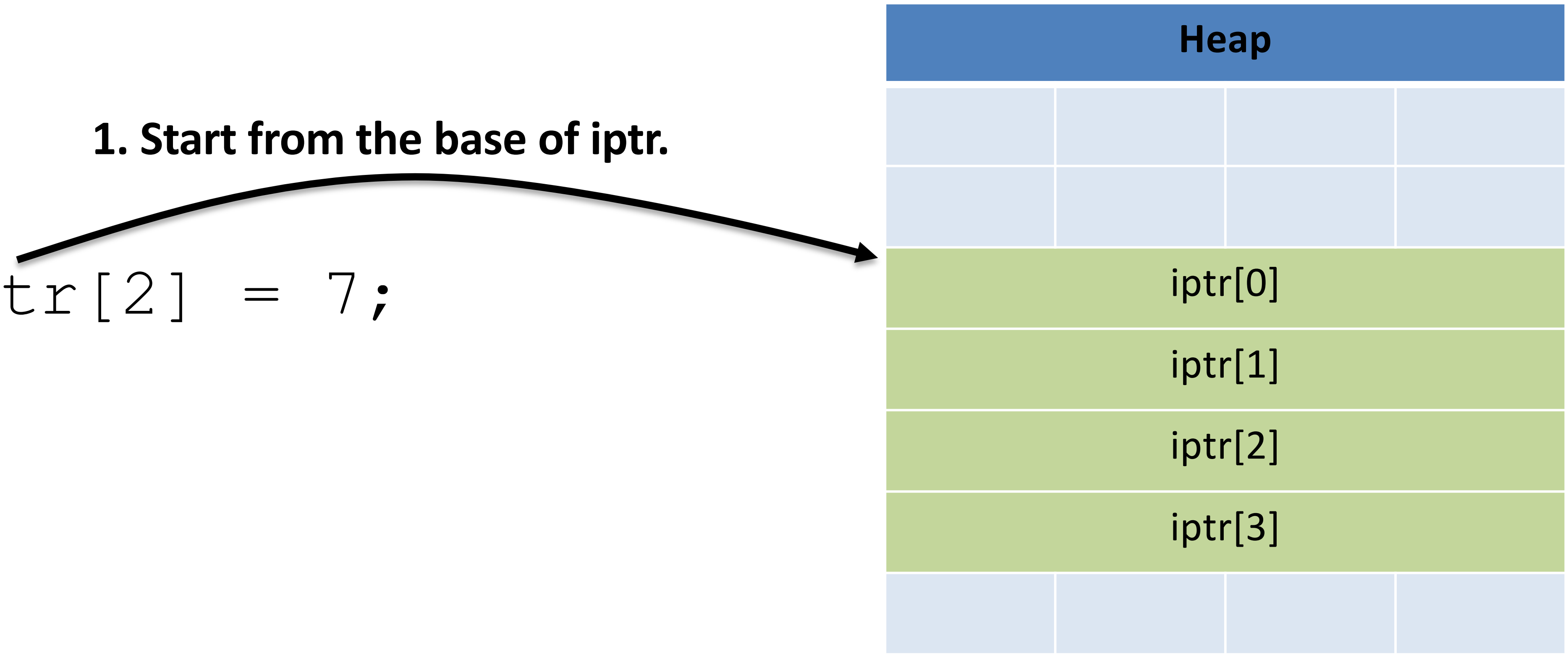
iptr[3]

Pointers as Arrays

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

1. Start from the base of iptr.

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



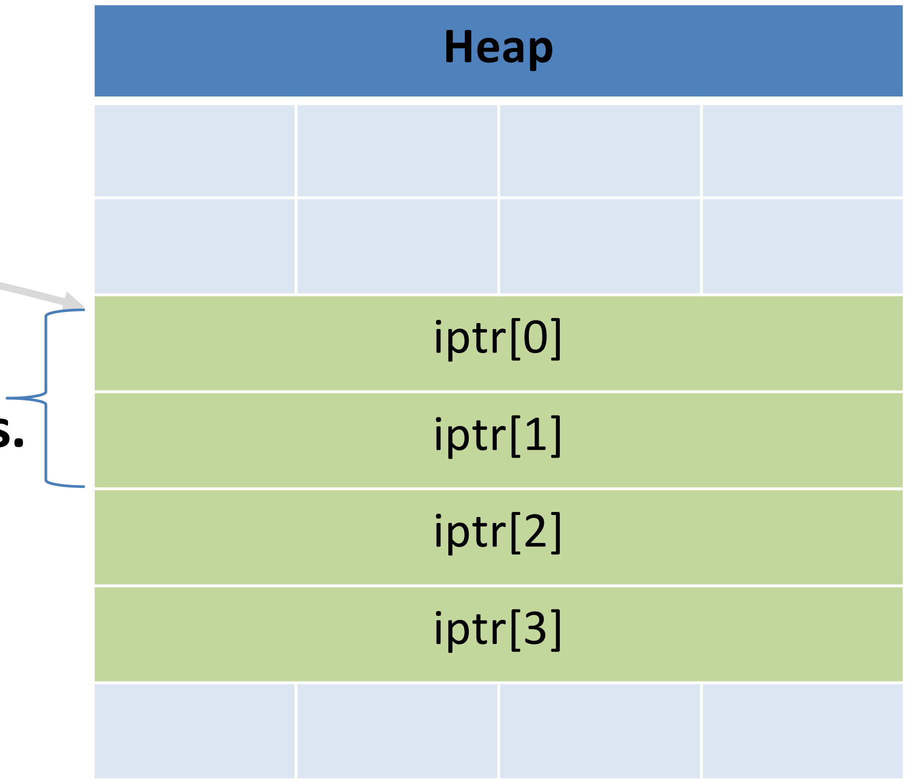
Pointers as Arrays

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

1. Start from the base of iptr.

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

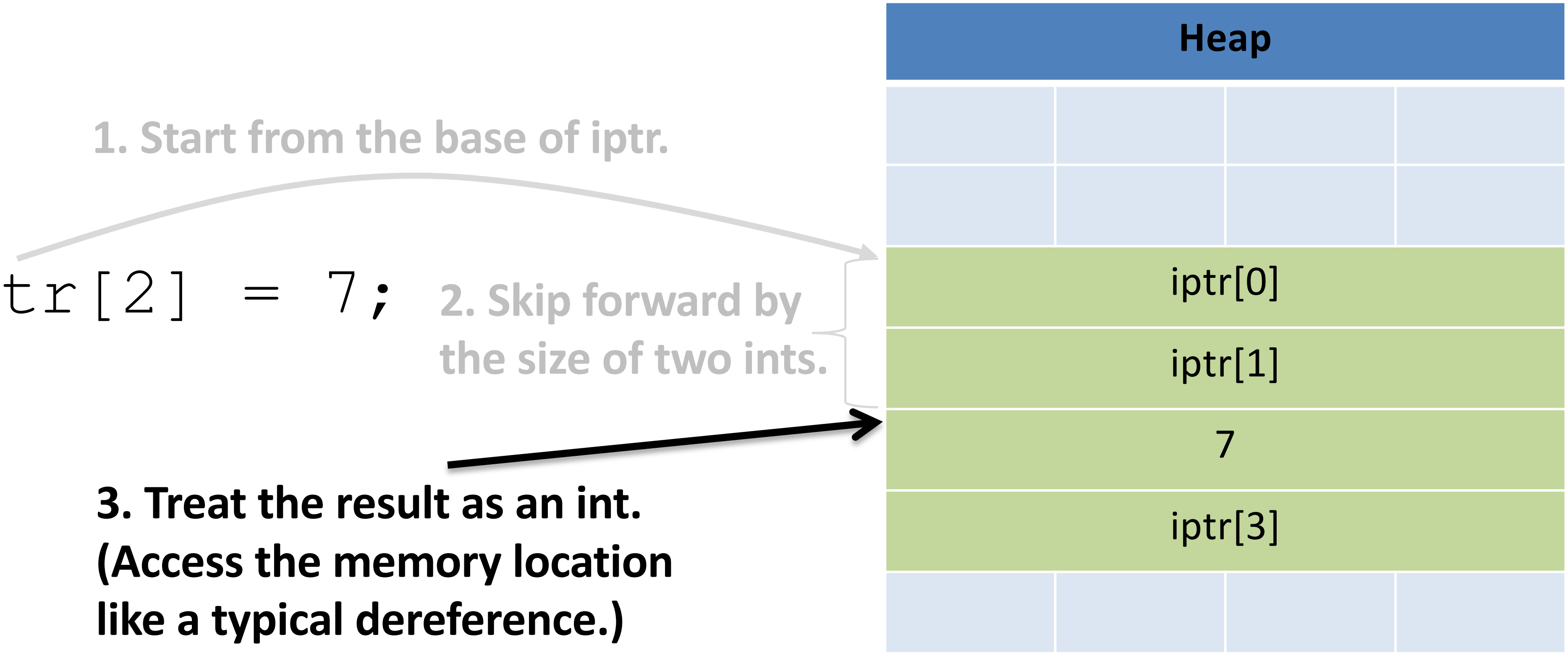
2. Skip forward by the size of two ints.



Pointers as Arrays

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

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



1. Start from the base of iptr.

2. Skip forward by the size of two ints.

3. Treat the result as an int. (Access the memory location like a typical dereference.)

Pointer Arithmetic

- Addition and subtraction work on pointers.

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

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

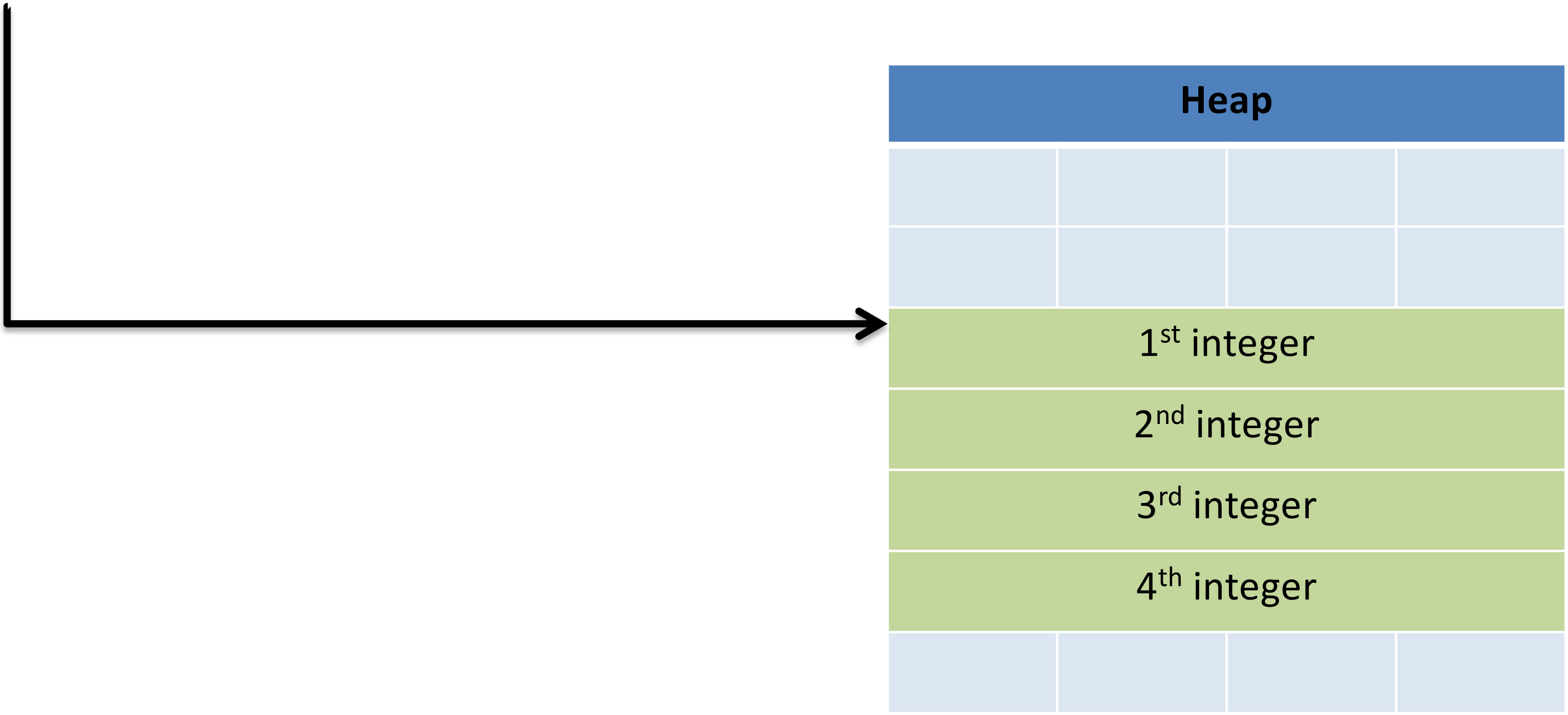
B. Mem. address in iptr + 3 bytes

C. Mem. address in iptr + 4 bytes

D. None of the above

Pointer Arithmetic

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



Pointer Arithmetic

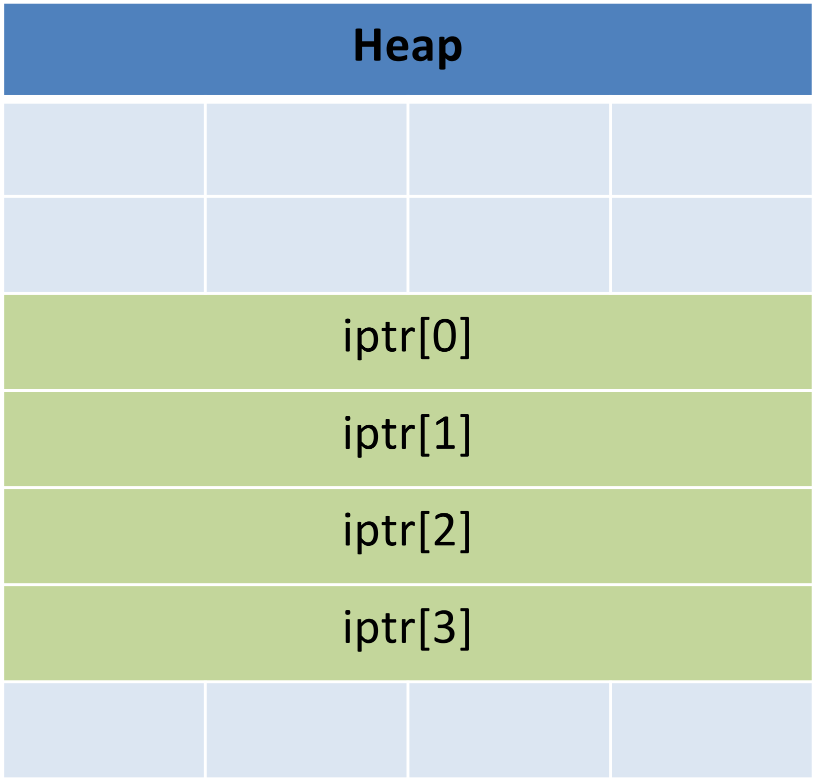
- Addition and subtraction work on pointers.

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

While Loop in C

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

moves +1 by size of the data type!



Let’s translate the while loop to assembly

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

Assume %ecx = base address of array

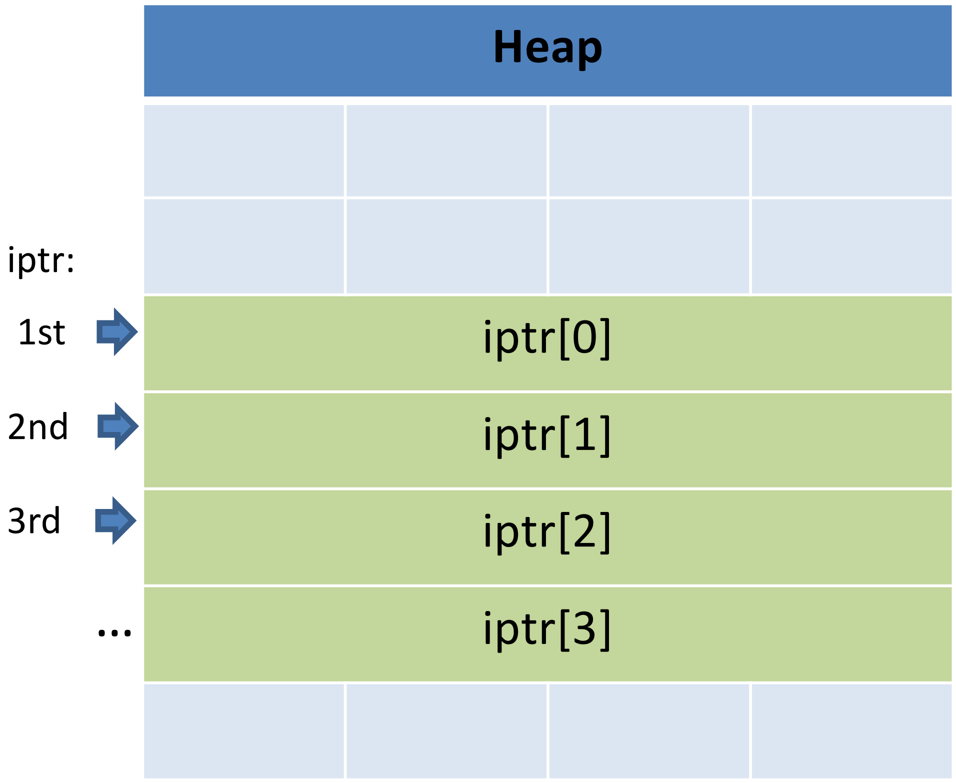
%eax = sum

%edx = loop index

```
   movl $0 eax
   movl $0 edx
loop:
   [fill instructions here]
   cmpl $5, %edx
   jne loop
```

While Loop in C

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



Heap

iptr:

1st

2nd

3rd

…

iptr[0]

iptr[1]

iptr[2]

iptr[3]

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?

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

cannot call free on iptr since it no longer references the base address of the array!

Pointer Manipulation: Necessary?

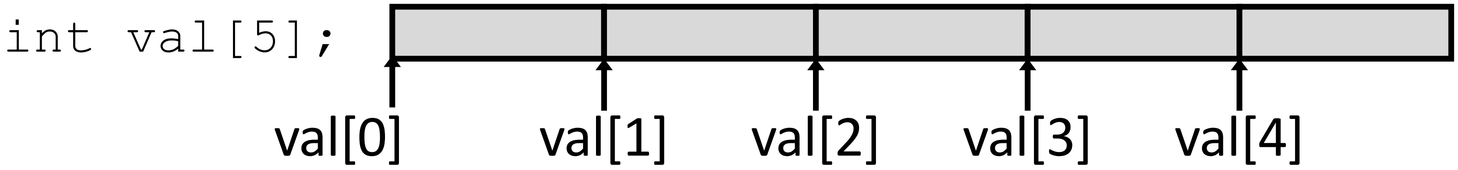
- Problem: `iptr` is changing!
- What if we wanted to free it?
- What if we wanted something like this:

```
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.

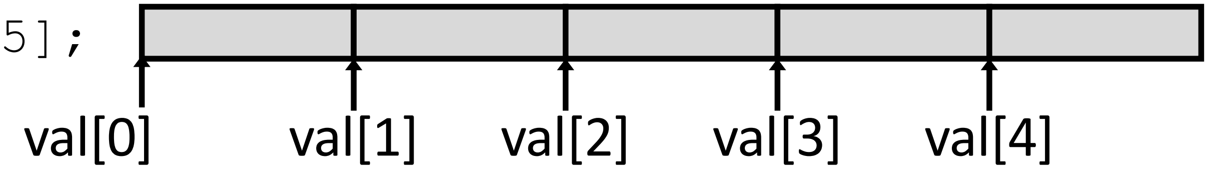


- “We’re goofy computer scientists who count starting from zero.”

Base + Offset

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

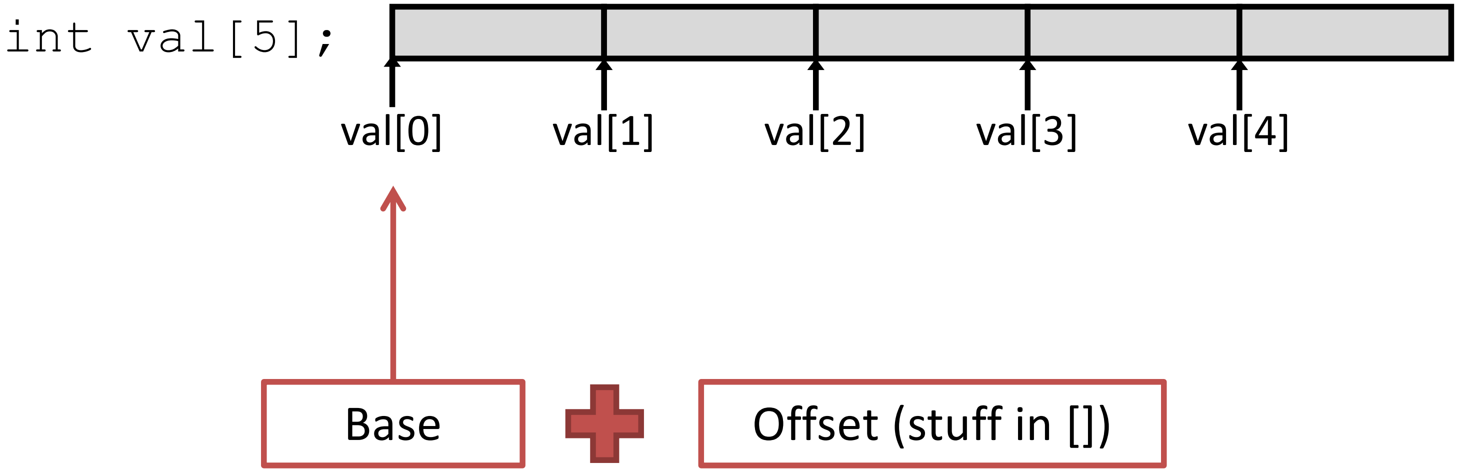
```
int val[5];
```



- ~~"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]?

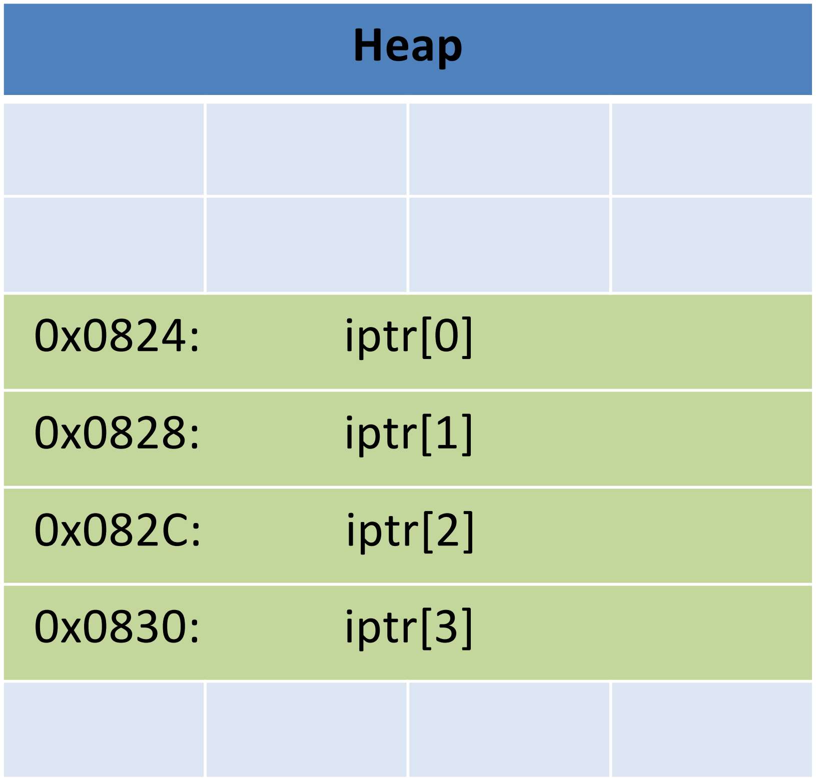
A. 0x0824 + 3 * 4

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

- We want 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

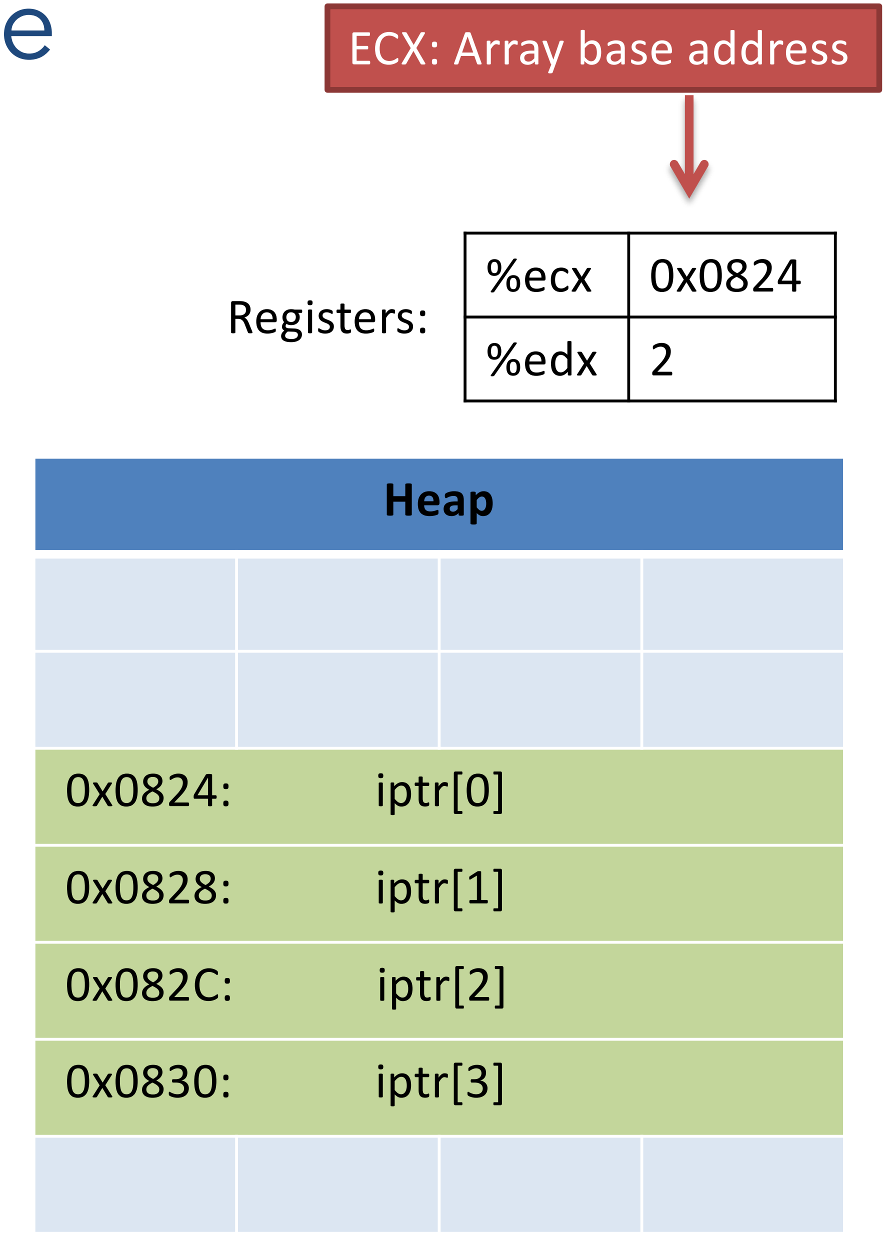
Suppose i is at %ebp - 8, and equals 2.

User says:

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

Translates to:

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



ECX: Array base address

Registers:

%ecx	0x0824
%edx	2

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

Example

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

Registers:

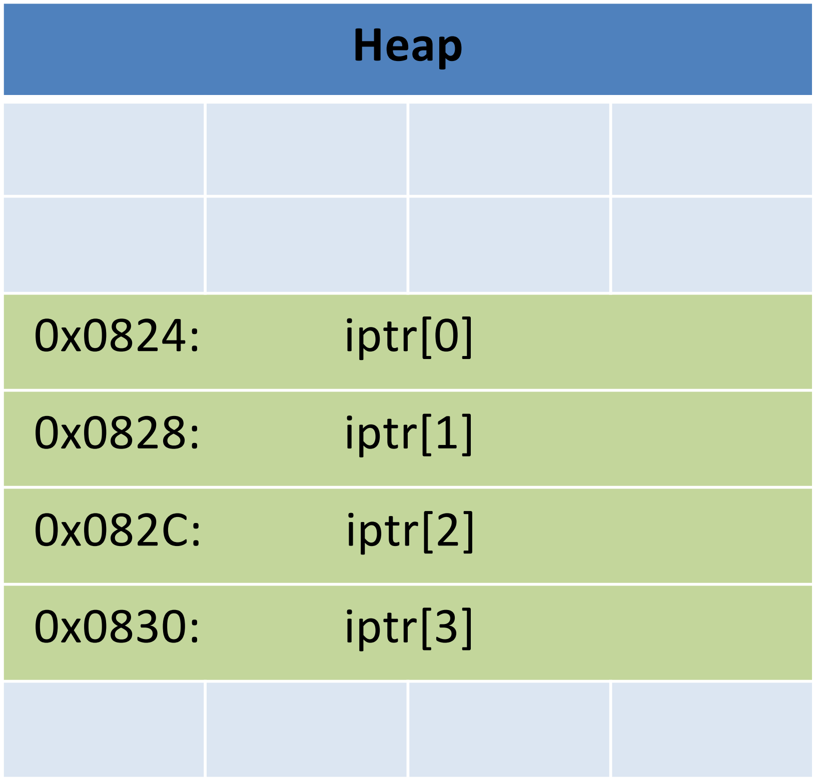
%ecx	0x0824
%edx	2

User says:

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

Translates to:

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



Heap

0x0824: iptr[0]

0x0828: iptr[1]

0x082C: iptr[2]

0x0830: iptr[3]

Example

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

Registers:

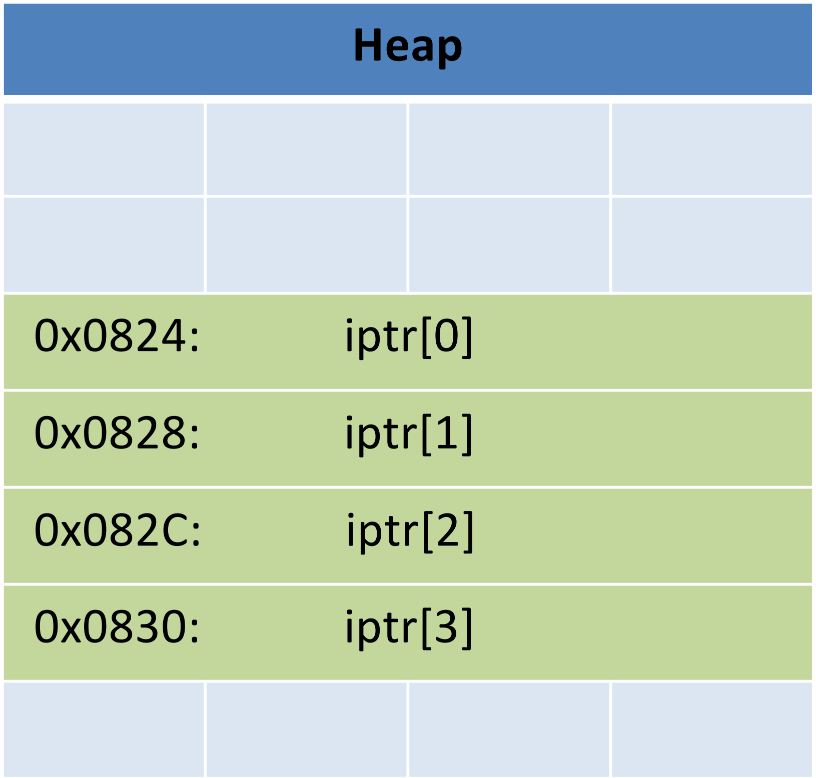
%ecx	0x0824
%edx	2

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

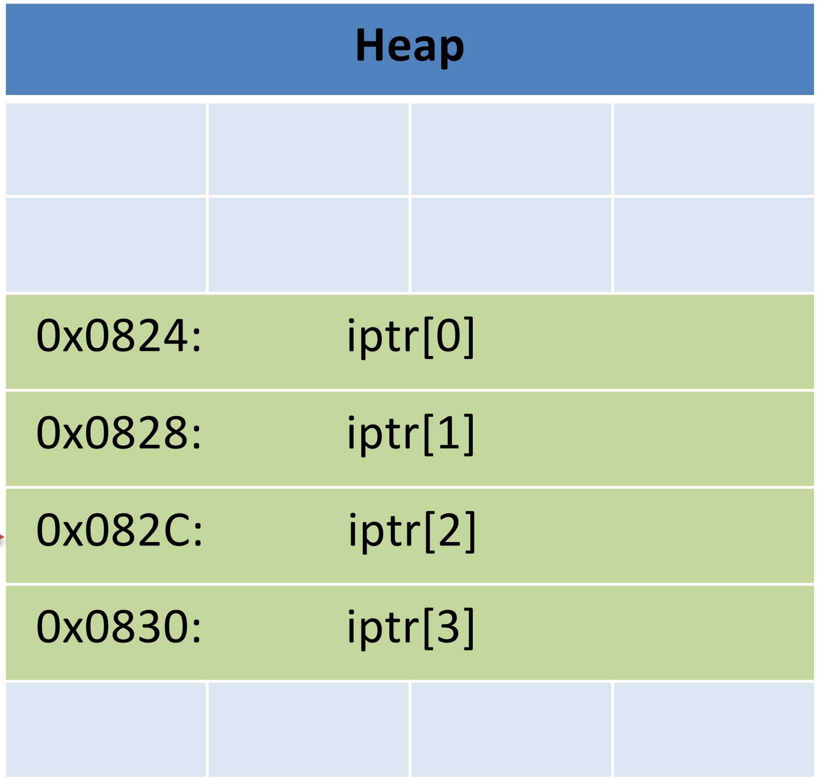
User says:

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

Translates to:

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

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



Heap

0x0824: iptr[0]

0x0828: iptr[1]

0x082C: iptr[2]

0x0830: iptr[3]

Example:

Allowed us to preserve ecx, and compute an offset without changing the pointer to the base of our array

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

Registers:

%ecx	0x0824
%edx	2

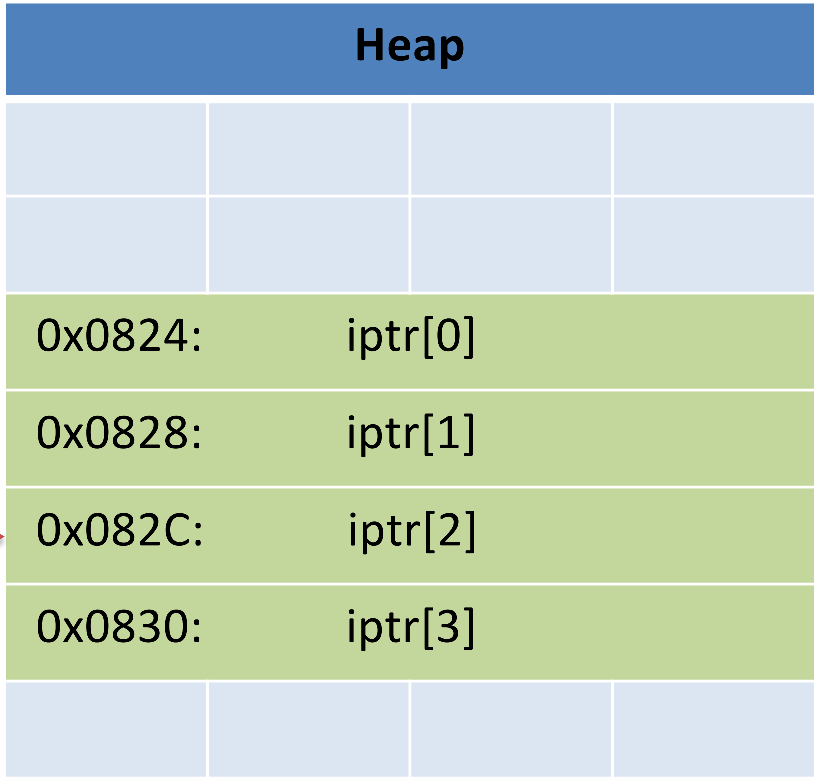
User says:

```
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)
```

displacement(%base, %index, scale)
base + (index * scale) + displacement

(Initial state)
Registers:

%eax	0x2464
%ecx	0x246C
%edx	7

Memory:

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

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 have seen these probably in your maze.

Two-dimensional Arrays

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

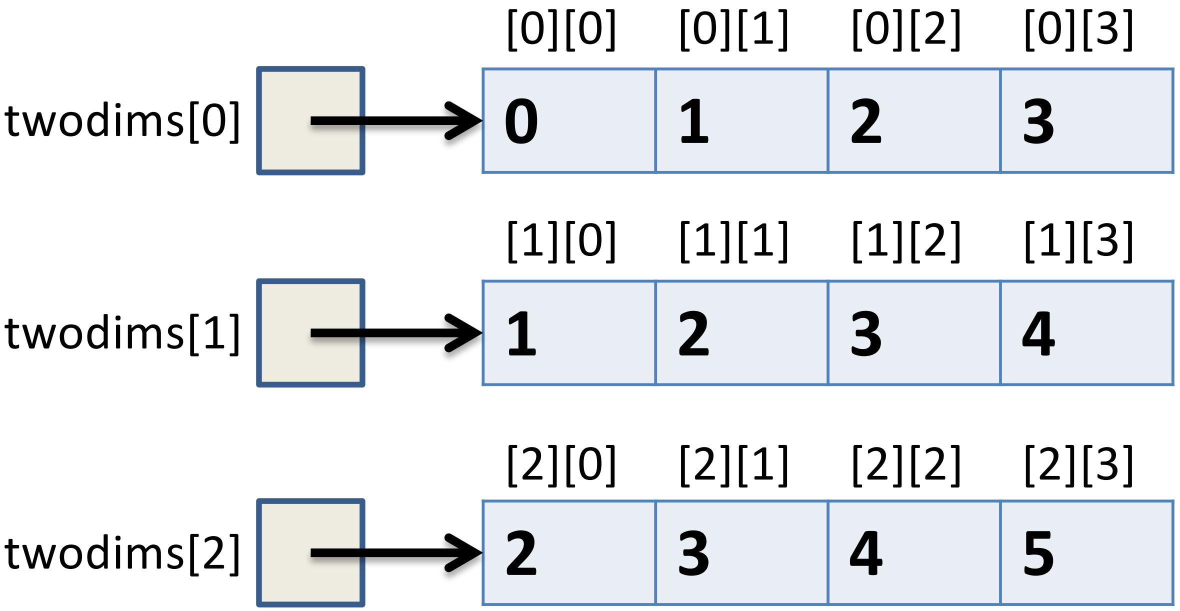
```
int twodims[3][4];
```

- “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;
   }
}
```



[0][0] [0][1] [0][2] [0][3]

twodims[0] → 0 1 2 3

[1][0] [1][1] [1][2] [1][3]

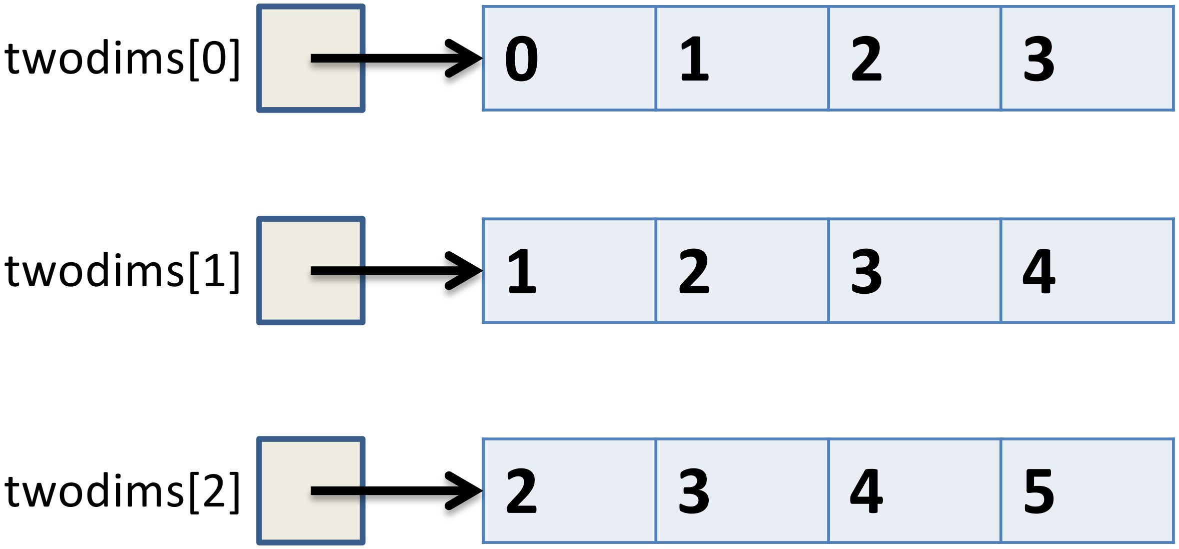
twodims[1] → 1 2 3 4

[2][0] [2][1] [2][2] [2][3]

twodims[2] → 2 3 4 5

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**

<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]

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]

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

base addr + row offset + col offset

or

base addr

+ num cols * data size

+ col offset