
OX
CB
BARE
METAL

```
int add(int x, int y) {
    x += y;
    return x;
}

int main(void){
    int a = 2;
    int b = 3;
    int sum = add(a, b);
    return 0;
}
```

- **x** and **y** are single values taking up 4 bytes each
- For function add, params **x** and **y** live on the *stack*
- Any manipulation done to the values occurs in the function
- **a** and **b** remain unaffected
- **x** and **y** are *copies* of **a** and **b**

```c
int main(void) {
    stdio_init_all();
    int numbers[10] = {1, 2, 3,
                       4, 5, 6, 7,
                       8, 9, 10};
    printf("%p", numbers);
    return 0;
}
```

Python

```
numbers = [1, 2, 3
           4, 5, 6, 7
           8, 9, 10]
```

What is this value?

And…why?

Arrays in C

- Must have declared size when created *or*
- Have a finite set of values provided
- Cannot be **append**ed or **pop**'d
- Must only store a single data type
- Cannot grow beyond initialized bounds

A *pointer*

What is this value?

And, again…why?

```
int sum(int *numbers) {
    printf("%p",numbers);
}

int main(void) {
    stdio_init_all();
    int numbers[10] = {1, 2, 3,
                       4, 5, 6, 7,
                       8, 9, 10};
    printf("%p", numbers);
    int total = sum(numbers);
    return 0;
}
```

Pointers

- Parameters in C are *pass by value*
- Pointer *values* are *references*
- These references (pointers) *point* to values stored in memory
  - Stack *or*
  - Heap

Pointers

"Dereferencing"

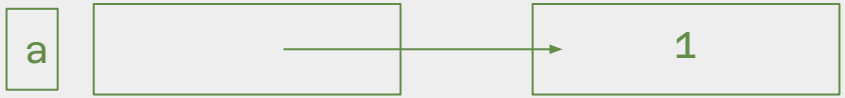
Resolve the memory location of a value *instead* of the value.

```
int *numbers
```

Pointers

3 rules of a pointer

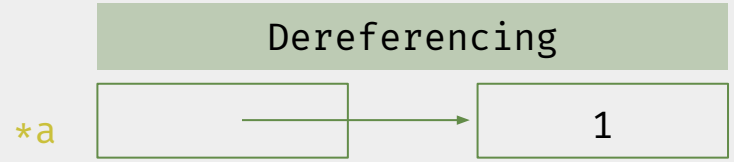
Pointers require a *pointee*



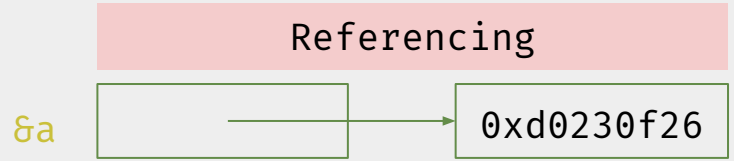
Pointers

3 rules of a pointer

Dereferencing always yields a value



Referencing yields a memory location

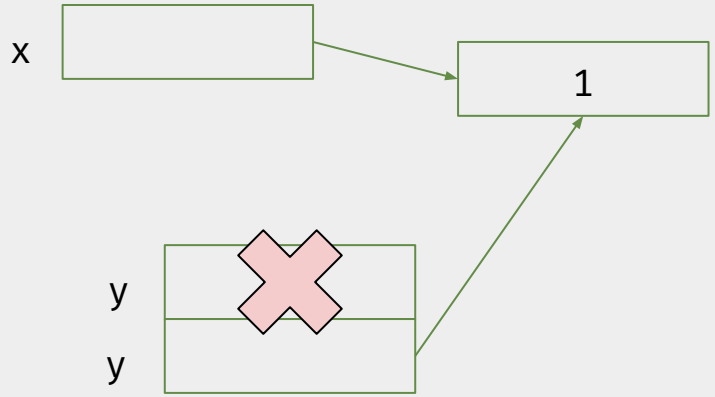


Pointers

3 rules of a pointer

Multiple pointers can point at the *same thing*

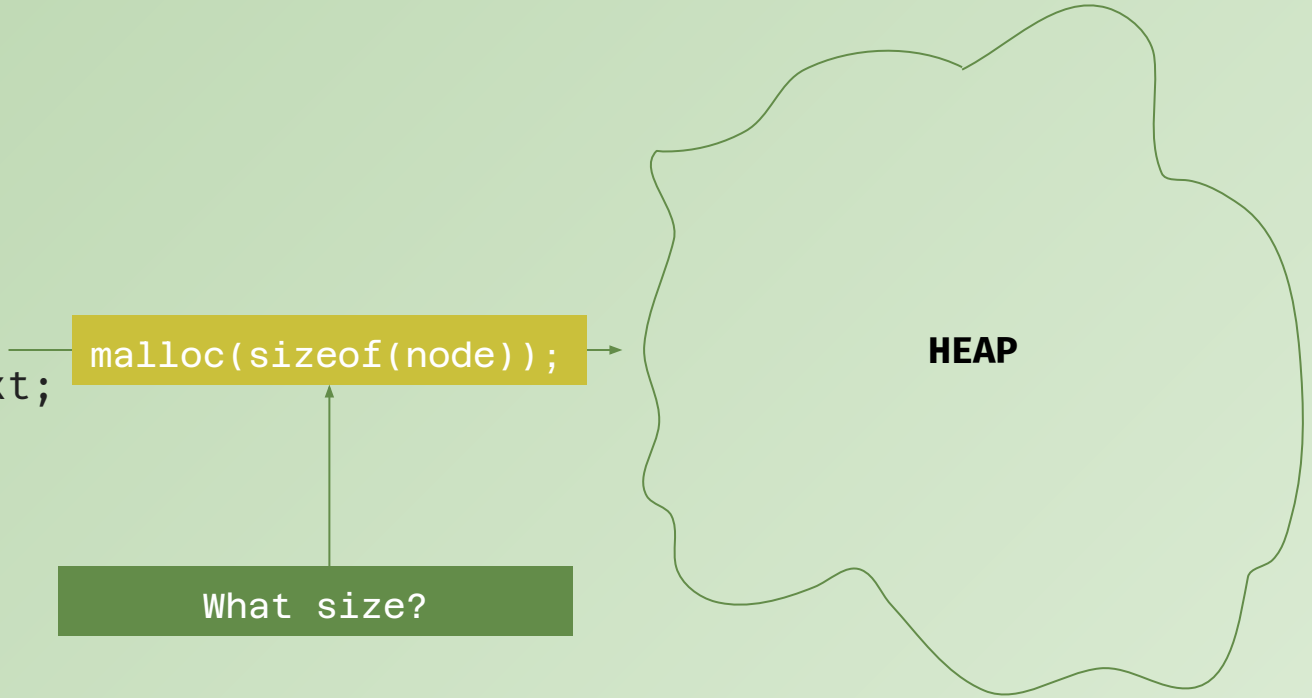
```
int *y;
*x = *y;
```



Why pointers?

- Allows programs to maintain one set of references to values and data
- Shares the same data between areas of a program
- Can allow us to create *ad hoc*, sizeable data structures

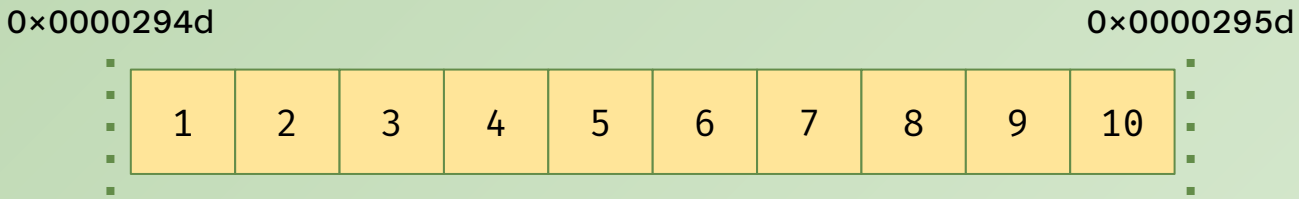
```
struct node {
    int id;
    struct node *next;
};
```



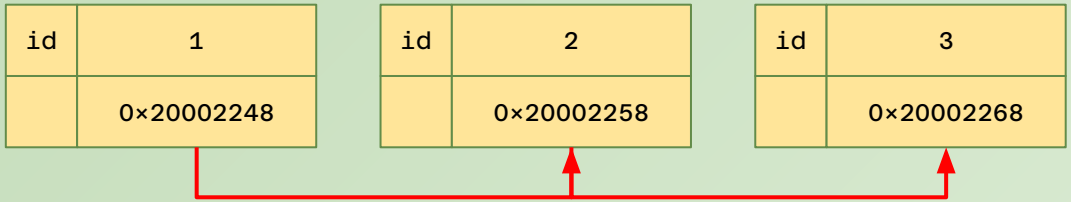
structs

- Can group data
- Can contain different kinds of data
- Are expandable and contractable
- Live in the heap (like arrays)

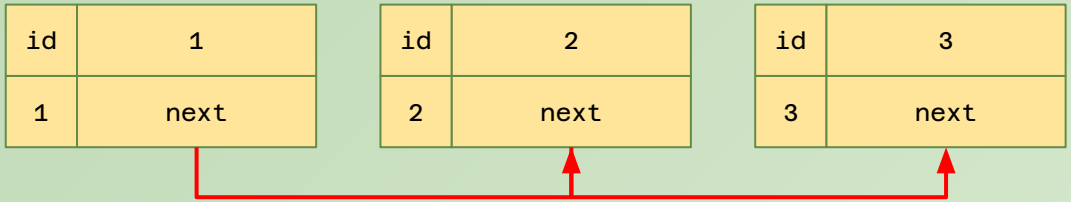
```
int numbers[10];
```




```
struct node {
    int id;
    struct node *next;
};
```



```
struct node {
    int id;
    struct node *next;
};
```




```
struct node *start;
start->id= 1
```