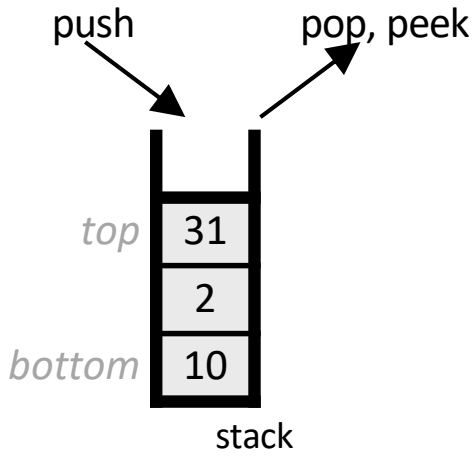
C Generics – Void *

Instructor: Jeeho Ryoo

Stacks

A **Stack** is a data structure representing a stack of things.
Objects can be ***pushed*** on top of or ***popped*** from the top of the stack.
Only the top of the stack can be accessed; no other objects in the stack are visible.



Main operations:

- **`push(value)`**: add an element to the top of the stack
- **`pop()`**: remove and return the top element in the stack
- **`peek()`**: return (but do not remove) the top element in the stack

What modifications are necessary to make a generic stack?

Stack Structs

```
typedef struct int_node {
    struct int_node *next;
    int data;
} int_node;
```

```
typedef struct int_stack {
    int nelems;
    int_node *top;
} int_stack;
```

How might we modify the Stack data representation itself to be generic?

Stack Structs

```
typedef struct int_node {
    struct int_node *next;
    int data;
} int_node;


typedef struct int_stack {
    int nelems;
    int_node *top;
} int_stack;
```

Problem: each node can no longer store the data itself, because it could be any size!

Generic Stack Structs

```
typedef struct int_node {
    struct int_node *next;
    void *data;
} int_node;


typedef struct stack {
    int nelems;
    int elem_size_bytes;
    node *top;
} stack;
```

Solution: each node stores a pointer, which is always 8 bytes, to the data somewhere else. We must also store the data size in the Stack struct.

Stack Functions

`int_stack_create()`: creates a new stack on the heap and returns a pointer to it

`int_stack_push(int_stack *s, int data)`: pushes data onto the stack

`int_stack_pop(int_stack *s)`: pops and returns topmost stack element

int_stack_create

```
int_stack *int_stack_create() {
    int_stack *s = malloc(sizeof(int_stack));
    s->nelems = 0;
    s->top = NULL;
    return s;
}
```

How might we modify this function to be generic?

From previous slide:

```
typedef struct stack {
    int nelems;
    int
elem_size_bytes;
    node *top;
} stack;
```

Generic stack_create

```
stack *stack_create(int elem_size_bytes) {
    stack *s = malloc(sizeof(stack));
    s->nelems = 0;
    s->top = NULL;
    s->elem_size_bytes = elem_size_bytes;
    return s;
}
```

int_stack_push

```
void int_stack_push(int_stack *s, int data) {
    int_node *new_node = malloc(sizeof(int_node));
    new_node->data = data;

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

How might we modify this function to be generic?

From previous slide:

```
typedef struct stack {
    int nelems;
    int
elem_size_bytes;
    node *top;
} stack;
```

```
typedef struct node
{
    struct node
*next;
    void *data;
} node;
```

Generic stack_push

```
void int_stack_push(int_stack *s, int data) {
    int_node *new_node = malloc(sizeof(int_node));
    new_node->data = data;

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

Problem 1: we can no longer pass the data itself as a parameter, because it could be any size!

Generic stack_push

```
void int_stack_push(int_stack *s, void *data) {
    int_node *new_node = malloc(sizeof(int_node));
    new_node->data = data;

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

Solution 1: pass a pointer to the data as a parameter instead.

Generic stack_push

```
void int_stack_push(int_stack *s, void *data) {
    int_node *new_node = malloc(sizeof(int_node));
    new_node->data = data;

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

Problem 2: we cannot copy the existing data pointer into new_node. The data structure must manage its own copy that exists for its entire lifetime. The provided copy may go away!

Generic stack_push

```
void stack_push(stack *s, void *data) {
    node *new_node = malloc(sizeof(node));
    new_node->data = malloc(s->elem_size_bytes);
    memcpy(new_node->data, data, s->elem_size_bytes);

    new_node->next = s->top;
    s->top = new_node;
    s->nelems++;
}
```

Solution 2: make a heap-allocated copy of the data that the node points to.

int_stack_pop

```
int int_stack_pop(int_stack *s) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    int_node *n = s->top;
    int value = n->data;

    s->top = n->next;

    free(n);
    s->nelems--;

    return value;
}
```

How might we modify this function to be generic?

From previous slide:

```
typedef struct stack {
    int nelems;
    int
elem_size_bytes;
    node *top;
} stack;
```

```
typedef struct node
{
    struct node
*next;
    void *data;
} node;
```

Generic stack_pop

```
int int_stack_pop(int_stack *s) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    int_node *n = s->top;
    int value = n->data;

    s->top = n->next;

    free(n);
    s->nelems--;

    return value;
}
```

Problem: we can no longer return the data itself, because it could be any size!

Generic stack_pop

```
void *int_stack_pop(int_stack *s) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    int_node *n = s->top;
    void *value = n->data;

    s->top = n->next;

    free(n);
    s->nelems--;

    return value;
}
```

While it's possible to return the heap address of the element, this means the client would be responsible for freeing it. Ideally, the data structure should manage its own memory here.

Generic stack_pop

```
void stack_pop(stack *s, void *addr) {
    if (s->nelems == 0) {
        error(1, 0, "Cannot pop from empty stack");
    }
    node *n = s->top;
    memcpy(addr, n->data, s->elem_size_bytes);
    s->top = n->next;

    free(n->data);
    free(n);
    s->nelems--;
}
```

Solution: have the caller pass a memory location as a parameter and copy the data to that location.

Using Generic Stack

```
int_stack *intstack = int_stack_create();
for (int i = 0; i < TEST_STACK_SIZE; i++) {
    int_stack_push(intstack, i);
}
```

We must now pass the *address* of an element to push onto the stack, rather than the element itself.

Using Generic Stack

```
stack *intstack = stack_create(sizeof(int));
for (int i = 0; i < TEST_STACK_SIZE; i++) {
    stack_push(intstack, &i);
}
```

We must now pass the *address* of an element to push onto the stack, rather than the element itself.

Using Generic Stack

```
int_stack *intstack = int_stack_create();
int_stack_push(intstack, 7);
```

We must now pass the *address* of an element to push onto the stack, rather than the element itself.

Using Generic Stack

```
stack *intstack = stack_create(sizeof(int));
int num = 7;
stack_push(intstack, &num);
```

We must now pass the *address* of an element to push onto the stack, rather than the element itself.

Using Generic Stack

```
// Pop off all elements
while (intstack->nelems > 0) {
    printf("%d\n", int_stack_pop(intstack));
}
```

We must now pass the *address* of where we would like to store the popped element, rather than getting it directly as a return value.

Using Generic Stack

```
// Pop off all elements
int popped_int;
while (intstack->nelems > 0) {
    int_stack_pop(intstack, &popped_int);
    printf("%d\n", popped_int);
}
```

We must now pass the *address* of where we would like to store the popped element, rather than getting it directly as a return value.

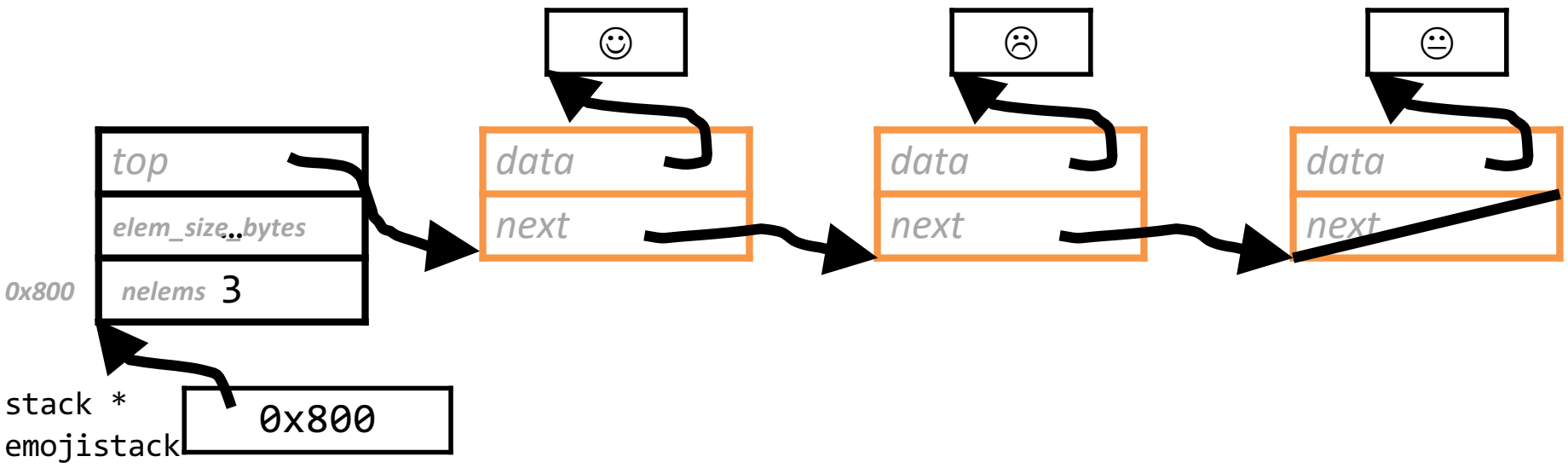
Demo: Generic Stack

`generic_stack.c`

More efficient generic stack

```
typedef struct stack {
    size_t nelems;
    size_t elem_size_bytes;
    node *top;
} stack;
```

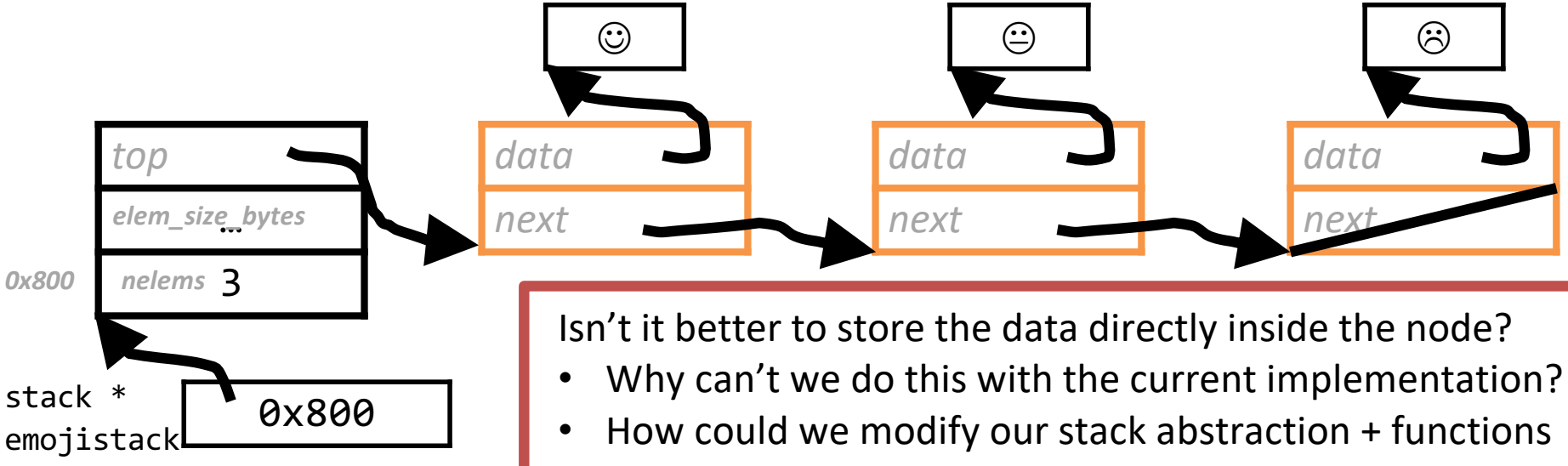
```
typedef struct node {
    struct node *next;
    void *data;
} node;
```



More efficient generic stack

```
typedef struct stack {
    size_t nelems;
    size_t elem_size_bytes;
    node *top;
} stack;
```

```
typedef struct node {
    struct node *next;
    void *data;
} node;
```

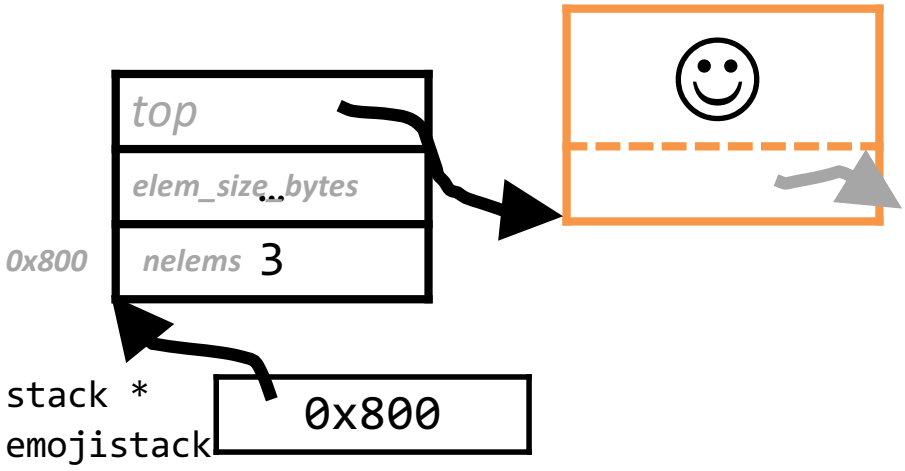


Isn't it better to store the data directly inside the node?

- Why can't we do this with the current implementation?
- How could we modify our stack abstraction + functions to do this?

More efficient generic stack

```
typedef struct stack {
    size_t nelems;
    size_t elem_size_bytes;
    void *top;
} stack;
```



If we *remove* the node struct:
We create nodes that are elem_size_bytes + 4B and ***directly*** store the data into our node.

A "node" just becomes contiguous bytes of memory storing (1) address of next node, and (2) data

⚠ **Tricky!** We will be working with `sizeof(void *)` and `(void **)`!!

More efficient generic stack

```
typedef struct stack {
    size_t nelems;
    size_t elem_size_bytes;
    void *top;
} stack;
```

Rewrite our `generic_stack.c` code without the node struct

Rewrite (as needed):

- `stack_create`
- `stack_push`
- `stack_pop`

(Don't touch main—a user of our stack should not know the difference)

stack_create

```
typedef struct stack {
    size_t nelems;
    size_t elem_size_bytes;
    void *top;
} stack;
```

```
1   stack *stack_create(size_t elem_size_bytes) {
2       stack *s = malloc(sizeof(stack));
3       s->nelems = 0;
4       s->top = NULL;
5       s->elem_size_bytes = elem_size_bytes;
6       return s;
7   }
```

✅ No nodes touched, nothing to change

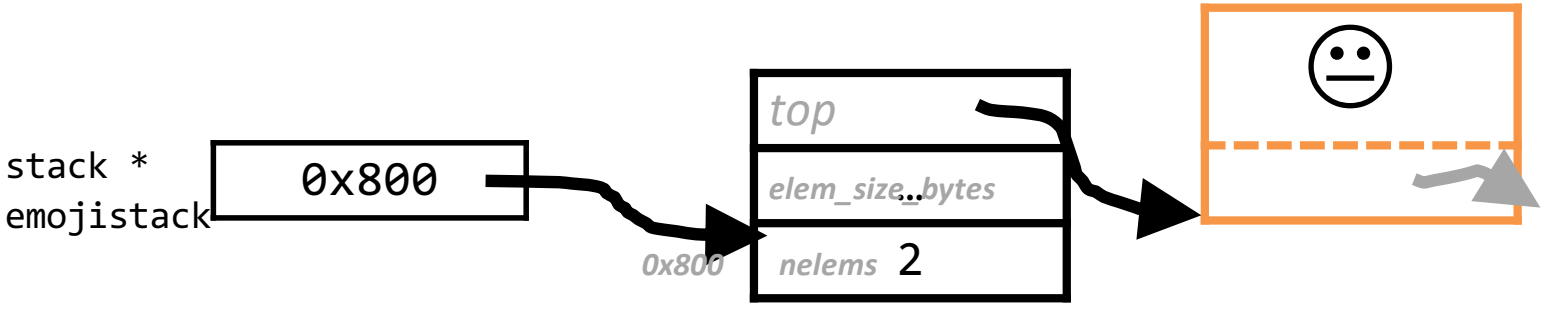
Old stack_push

```
1  void stack_push(stack *s, void *data) {
2      node *new_node = malloc(sizeof(node));
3      new_node->data = malloc(s->elem_size_bytes);
4      memcpy(new_node->data, data, s->elem_size_bytes);
5      new_node->next = s->top;
6      s->top = new_node;
7      s->nelems++;
8  }
```

What do we have to change from the old function? Check all functionality:

1. Allocate a node
2. Copy in data
3. Set new node's next to be top of stack
4. Set top of stack to be new node
5. Increment element count

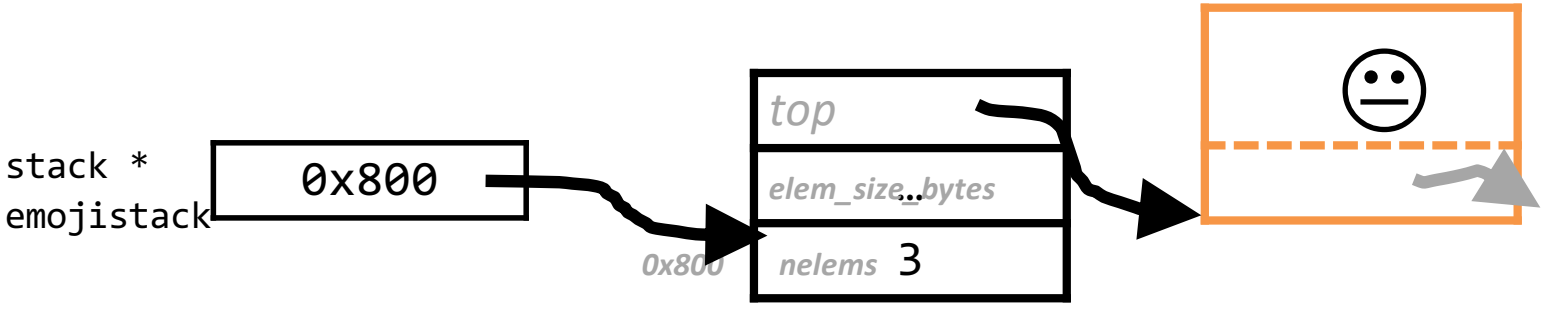
1. Allocate a node



In `stack_push`, we had: `node *new_node = malloc(sizeof(node));`

We no longer have a typedef struct node!
Our node is now just **contiguous bytes on the heap**.
How do we **rewrite** this line to handle
our new node representation?

1. Allocate a node



In `stack_push`, we had: `node *new_node = malloc(sizeof(node));`

We no longer have a typedef struct node!
Our node is now just **contiguous bytes on the heap**.
How do we **rewrite** this line to handle our new node representation?

```
void *new_node = malloc(sizeof(void *) + s->elem_size_bytes);
```

New stack_push

```
1  void stack_push(stack *s, void *data) {
2      void *new_node = malloc(sizeof(void *) + s->elem_size_bytes);
3      memcpy((char *) new_node + sizeof(void *),
             data, s->elem_size_bytes);
4      *((void **) new_node) = s->top;
5      s->top = new_node;
6      s->nelems++;
7  }
```

Check all functionality:

1. Allocate a node
2. Copy in data
3. Set new node's next to be top of stack
4. Set top of stack to be new node
5. Increment element count

New `stack_push`

- `sizeof(void *)` is the size of a pointer, which is always 4B in our class
- The dereference operation *(void **) ptr works!
  - `void * ptr = …;` **Declaration**: ptr stores an address, no idea what is at the address ptr
  - `(void **) ptr;` **Cast**: at the address ptr, there is an address
  - `*(void **) ptr;` **Dereference**: get the address stored at the address ptr

Old stack_pop

```
1   void stack_pop(stack *s, void *addr) {
2       if (s->nelems == 0) {
3           exit(1);
4       }
5       node *n = s->top;
6       memcpy(addr, n->data, s->elem_size_bytes);
7       s->top = n->next;
8       free(n->data);
9       free(n);
10      s->nelems--;
11  }
```

What do we have to change from the old function? Check all functionality:

1. Copy top node's data to addr buf
2. Set top of stack to top node's next
3. Free old top node
4. Decrement element count

New stack_pop

```
1   void stack_pop(stack *s, void *addr) {
2       if (s->nelems == 0) {
3           exit(1);
4       }
5       void *n = s->top;
6       memcpy(addr, (char *) n + sizeof(void *), s->elem_size_bytes);
7       s->top = *(void **) n;
8       free(n);
9       s->nelems--;
10  }
```

Check all functionality:

1. Copy top node's data to addr buf
2. Set top of stack to top node's next
3. Free old top node
4. Decrement element count