COMP201
Computer Systems & Programming
Lecture #16 – structs
Good news, everyone!

• Assg3 will be out today (due Nov 27)

• Online Mid-Semester Course Evaluations are now available via Koç Üniversitesi Mobil
Recap

• Function Pointers (cont’d.)
• const
Plan for Today

• struct
• Generic stack

Disclaimer: Slides for this lecture were borrowed from
—Nick Troccoli's Stanford CS107 class
Lecture Plan

• struct
• Generic stack
A **struct** is a way to define a new variable type that is a group of other variables.

```c
struct date {    // declaring a struct type
    int month;
    int day;       // members of each date structure
};
...
```

```c
struct date today;  // construct structure instances
    today.month = 1;
    today.day = 28;

struct date new_years_eve = {12, 31};  // shorter initializer syntax
```
**Structs**

Wrap the struct definition in a **typedef** to avoid having to include the word **struct** every time you make a new variable of that type.

```c
typedef struct {
    int month;
    int day;
} date;
...

date today;
today.month = 1;
today.day = 28;

date new_years_eve = {12, 31};
```
Structs

If you pass a struct as a parameter, like for other parameters, C passes a **copy** of the entire struct.

```c
void advance_day(date d) {
    d.day++;
}

int main(int argc, char *argv[]) {
    date my_date = {1, 28};
    advance_day(my_date);
    printf("%d", my_date.day); // 28
    return 0;
}
```
If you pass a struct as a parameter, like for other parameters, C passes a copy of the entire struct. Use a pointer to modify a specific instance.

```c
void advance_day(date *d) {
    (*d).day++;  
}

int main(int argc, char *argv[]) {
    date my_date = {1, 28};
    advance_day(&my_date);
    printf("%d", my_date.day); // 29
    return 0;
}
```
The **arrow** operator lets you access the field of a struct pointed to by a pointer.

```c
void advance_day(date *d) {
    d->day++;
    // equivalent to (*d).day++;
}

int main(int argc, char *argv[]) {
    date my_date = {1, 28};
    advance_day(&my_date);
    printf("%d", my_date.day);
    // 29
    return 0;
}
```
C allows you to return structs from functions as well. It returns whatever is contained within the struct.

date create_new_years_date() {
    date d = {1, 1};
    return d;  // or return (date){1, 1};
}

int main(int argc, char *argv[]) {
    date my_date = create_new_years_date();
    printf("%d", my_date.day);  // 1
    return 0;
}
**Structs**

`sizeof` gives you the entire size of a struct, which is the sum of the sizes of all its contents.

```c
typedef struct date {
    int month;
    int day;
} date;

int main(int argc, char *argv[]) {
    int size = sizeof(date);  // 8
    return 0;
}
```
Arrays of Structs

You can create arrays of structs just like any other variable type.

```c
typedef struct my_struct {
    int x;
    char c;
} my_struct;

my_struct array_of_structs[5];
```
Arrays of Structs

To initialize an entry of the array, you must use this special syntax to confirm the type to C.

```c
typedef struct my_struct {
    int x;
    char c;
} my_struct;

my_struct array_of_structs[5];
array_of_structs[0] = (my_struct){0, 'A'};
```
Arrays of Structs

You can also set each field individually.

```c
typedef struct my_struct {
    int x;
    char c;
} my_struct;

... my_struct array_of_structs[5];
array_of_structs[0].x = 2;
array_of_structs[0].c = 'A';
```
Lecture Plan

• struct
• Generic stack
Stacks

• C generics are particularly powerful in helping us create generic data structures.
• Let’s see how we might go about making a Stack in C.
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
Stacks

A stack is often implemented using a **linked list** internally.

- "bottom" = tail of linked list
- "top" = head of linked list *(why not the other way around?)*

```c
Stack<int> s;
s.push(42);
s.push(-3);
s.push(17);
```

**Problem:** C is not object-oriented! We can’t call methods on variables.
Demo: Int Stack

int_stack.c
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!
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

• \texttt{int\_stack\_create()}: creates a new stack on the heap and returns a pointer to it

• \texttt{int\_stack\_push(int\_stack \,*s, int data)}: pushes data onto the stack

• \texttt{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;
}

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

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

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

```c
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, const 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

```c
void int_stack_push(int_stack *s, const 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

```c
int main() {
    stack *int_stack = stack_create(sizeof(int));
    add_one(int_stack);
    // now stack stores pointer to invalid memory for 7!
}

void add_one(stack *s) {
    int num = 7;
    stack_push(s, &num);
}
```
Generic stack_push

```c
void stack_push(stack *s, const 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

```c
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

```c
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

```c
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

```c
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

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

We must now pass the \textit{address} of an element to push onto the stack, rather than the element itself.
Using Generic Stack

```c
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
Recap

• struct
• Generic stack

Next Time: Compiling C programs