

CS 107

Lecture 28: Review and the void *: Generic stack

Friday, March 15, 2024

Computer Systems
Winter 2024
Stanford University
Computer Science Department

Reading: Reader: Ch 8, *Pointers, Generic functions with void **, and *Pointers to Functions*, K&R Ch 1.6, 5.6-5.9

Lecturer: Chris Gregg

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

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



Today's Topics

- Logistics
 - Final Exam on Monday, 3:30pm-6:30pm, CEMEX
- Review: Let's build a generic stack, void *



Review: Building a generic stack

Let's build a generic stack. We are going to be using `structs` extensively for this example, and they are fair game for the final exam. So, make sure you understand this example!

First, let's remind ourselves what the stack data structure does (back to CS 106B!):

1. A stack is a last-in-first-out data structure that can store elements. The first element in the stack is the last element out of the stack.
2. The *push* operation adds an element onto the stack
3. The *pop* operation removes an element from the stack.

Note, we are not talking about the program stack, but a generic version of the stack abstract data type!

Code at: `/afs/ir/class/cs107/lecture-code/lect28`



Example: Building a generic stack

Let's build a generic stack. We are going to be using `structs` extensively for this example, and they are fair game for the final exam. So, make sure you understand this example!

We'll start by defining a node that will hold a pointer to a "next" node, and some data:

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

A note on syntax: We are defining a type here (thus, `typedef`), and we are defining a node to be a "`struct node`". This is different from C++, where we can just define a struct and use its name. In C, without the `typedef`, we would constantly have to be referring to "`struct node`" every time we wanted to use it. We often do this in C, but having a `typedef` is nice.



Example: Building a generic stack

We'll start by defining a node that will hold a pointer to a "next" node, and some data:

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

We don't know anything about the type of thing that `data` will point to, although the stack itself will know its width.



Example: Building a generic stack

Next, let's build the `stack` type. It will have a defined width for each node, and it will also keep track of how many elements it holds. It will also keep track of the top of the stack. Again, we want to typedef it so we don't have to continually say "`struct stack`" when we want to use it.

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

Remember, a node is generic, so this stack can hold any type, although once it has a width defined, all elements you push must have that width.



Example: Building a generic stack

How do we create a default stack? We could do it manually:

```
stack s1;  
s1.width = sizeof(int); // store ints  
s1.nelems = 0;  
s1.top = NULL;
```

But let's create a function for it, in which case we should use a pointer:

```
stack *s = stack_create(...);
```



Example: Building a generic stack

Our stack creation function:

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

Let's investigate...



Example: Building a generic stack

Our stack creation function:

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

A particular stack must have a set width (otherwise, we would have to pass in the width each time, and this doesn't make sense for `pop` -- we wouldn't know what type we were popping off!)

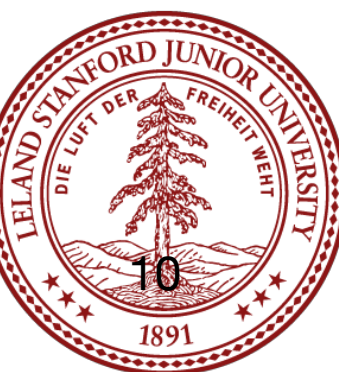


Example: Building a generic stack

Our stack creation function:

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

Get enough memory from the heap to create the stack.



Example: Building a generic stack

Our stack creation function:

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

Set the initial conditions.



Example: Building a generic stack

Our stack creation function:

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

Return the pointer to the memory we just requested and initialized.

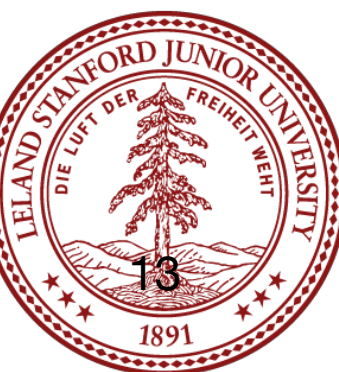


Example: Building a generic stack

Let's look at our `push` function:

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

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



Example: Building a generic stack

Let's look at our `push` function:

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

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

The `stack` function takes a `stack` as a parameter! The `stack` isn't an object, and it doesn't have functions built in. If we really wanted to, we could create a `stack` struct that has function pointers, but that is more advanced. A pointer to the data is also required.



Example: Building a generic stack

Let's look at our `push` function:

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

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

Each time we add an element to the stack, we need to create a `node`, and we get that off the heap, too.



Example: Building a generic stack

Let's look at our `push` function:

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

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

Guess what? We also have to use heap memory to store the data! We are making a copy of the data, not just pointing to it!



Example: Building a generic stack

Let's look at our `push` function:

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

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

We copy the data pointed to into our node. This could be anything, but we know the width. If it is a pointer, we'll copy the pointer, but it could be integer data, or any other kind of data.



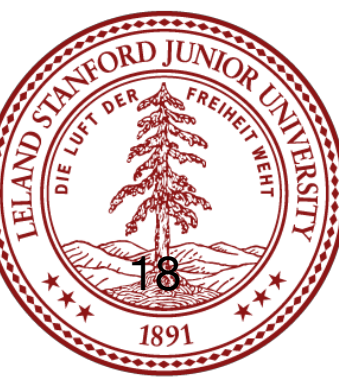
Example: Building a generic stack

Let's look at our `push` function:

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

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

We have to do some wiring here (kind of like linked lists). We are inserting this node before the top of the stack.



Example: Building a generic stack

Let's look at our `push` function:

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

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

Don't forget to update the number of elements.



Example: Building a generic stack

Let's look at our `pop` function. Pop will copy data back into a memory location we give it, instead of returning a pointer -- this preserves the encapsulation of our data.

```
bool stack_pop(stack *s, void *addr)
{
    if (s->nelems == 0) {
        return false;
    }
    node *n = s->top;
    memcpy(addr, n->data, s->width);
    // rewire
    s->top = n->next;

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



Example: Building a generic stack

Let's look at our `pop` function. Pop will copy data back into a memory location we give it, instead of returning a pointer -- this preserves the encapsulation of our data.

```
bool stack_pop(stack *s, void *addr)
{
    if (s->nelems == 0) {
        return false;
    }
    node *n = s->top;
    memcpy(addr, n->data, s->width);
    // rewire
    s->top = n->next;

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

Let's return a boolean value to say whether or not we had an element to return. In other words, if the stack is empty, return `false`; otherwise, return `true`.



Example: Building a generic stack

Let's look at our `pop` function. Pop will copy data back into a memory location we give it, instead of retiring a pointer -- this preserves the encapsulation of our data.

```
bool stack_pop(stack *s, void *addr)
{
    if (s->nelems == 0) {
        return false;
    }
    node *n = s->top;
    memcpy(addr, n->data, s->width);
    // rewire
    s->top = n->next;

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

Again, `pop` has a stack argument, and a pointer to a memory location to hold the data we are going to copy.



Example: Building a generic stack

Let's look at our `pop` function. Pop will copy data back into a memory location we give it, instead of retiring a pointer -- this preserves the encapsulation of our data.

```
bool stack_pop(stack *s, void *addr)
{
    if (s->nelems == 0) {
        return false;
    }
    node *n = s->top;
    memcpy(addr, n->data, s->width);
    // rewire
    s->top = n->next;

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

Check to see if the stack is empty.



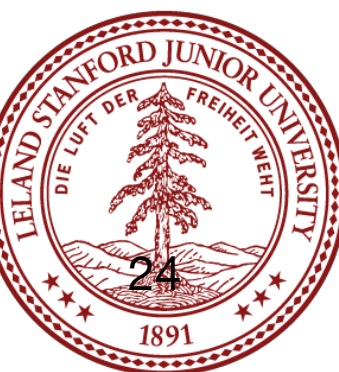
Example: Building a generic stack

Let's look at our `pop` function. Pop will copy data back into a memory location we give it, instead of retiring a pointer -- this preserves the encapsulation of our data.

```
bool stack_pop(stack *s, void *addr)
{
    if (s->nelems == 0) {
        return false;
    }
    node *n = s->top;
    memcpy(addr, n->data, s->width);
    // rewire
    s->top = n->next;

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

Might as well create a temporary pointer so we don't have to do a bunch of double "`->`" references.



Example: Building a generic stack

Let's look at our `pop` function. Pop will copy data back into a memory location we give it, instead of retiring a pointer -- this preserves the encapsulation of our data.

```
bool stack_pop(stack *s, void *addr)
{
    if (s->nelems == 0) {
        return false;
    }
    node *n = s->top;
    memcpy(addr, n->data, s->width);
    // rewire
    s->top = n->next;

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

We'll copy the data back to the memory location we were provided.



Example: Building a generic stack

Let's look at our `pop` function. Pop will copy data back into a memory location we give it, instead of retiring a pointer -- this preserves the encapsulation of our data.

```
bool stack_pop(stack *s, void *addr)
{
    if (s->nelems == 0) {
        return false;
    }
    node *n = s->top;
    memcpy(addr, n->data, s->width);
    // rewire
    s->top = n->next;

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

Re-wiring is pretty easy -- the top is now just the next element in the stack.



Example: Building a generic stack

Let's look at our `pop` function. Pop will copy data back into a memory location we give it, instead of retiring a pointer -- this preserves the encapsulation of our data.

```
bool stack_pop(stack *s, void *addr)
{
    if (s->nelems == 0) {
        return false;
    }
    node *n = s->top;
    memcpy(addr, n->data, s->width);
    // rewire
    s->top = n->next;

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

We have to clean up. First, we free the data (remember, we malloc'd it originally!)



Example: Building a generic stack

Let's look at our `pop` function. Pop will copy data back into a memory location we give it, instead of retiring a pointer -- this preserves the encapsulation of our data.

```
bool stack_pop(stack *s, void *addr)
{
    if (s->nelems == 0) {
        return false;
    }
    node *n = s->top;
    memcpy(addr, n->data, s->width);
    // rewire
    s->top = n->next;

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

Then, we free the node itself (because we malloc'd it!)



Example: Building a generic stack

Let's look at our `pop` function. Pop will copy data back into a memory location we give it, instead of retiring a pointer -- this preserves the encapsulation of our data.

```
bool stack_pop(stack *s, void *addr)
{
    if (s->nelems == 0) {
        return false;
    }
    node *n = s->top;
    memcpy(addr, n->data, s->width);
    // rewire
    s->top = n->next;

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

Don't forget to decrement the number of elements!



Example: Building a generic stack

Let's look at our `pop` function. Pop will copy data back into a memory location we give it, instead of retiring a pointer -- this preserves the encapsulation of our data.

```
bool stack_pop(stack *s, void *addr)
{
    if (s->nelems == 0) {
        return false;
    }
    node *n = s->top;
    memcpy(addr, n->data, s->width);
    // rewire
    s->top = n->next;

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

We did have an element to return, so we return `true`.



Example: Building a generic stack

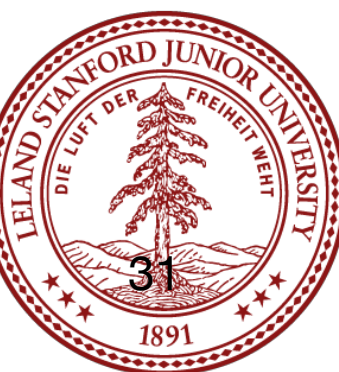
Now we can try it. Let's push on an array of `ints`, and then pop them all off:

```
int main(int argc, char **argv)
{
    // start with an int array
    int iarr[] = {0, 2, 4, 6, 8, 12345678, 24680};
    int nelems = sizeof(iarr) / sizeof(iarr[0]);

    stack *intstack = stack_create(sizeof(iarr[0]));
    for (int i=0; i < nelems; i++) {
        stack_push(intstack, iarr + i);
    }

    int popped_int;
    while (stack_pop(intstack, &popped_int)) {
        printf("%d\n", popped_int);
    }
    free(s); // clean up!
    return 0;
}
```

What is the size of each element?



Example: Building a generic stack

Now we can try it. Let's push on an array of `ints`, and then pop them all off:

```
int main(int argc, char **argv)
{
    // start with an int array
    int iarr[] = {0, 2, 4, 6, 8, 12345678, 24680};
    int nelems = sizeof(iarr) / sizeof(iarr[0]);

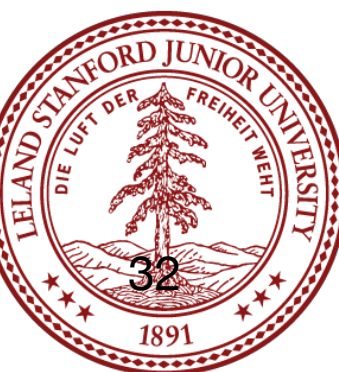
    stack *intstack = stack_create(sizeof(iarr[0]));
    for (int i=0; i < nelems; i++) {
        stack_push(intstack, iarr + i);
    }

    int popped_int;
    while (stack_pop(intstack, &popped_int)) {
        printf("%d\n", popped_int);
    }
    free(s); // clean up!
    return 0;
}
```

What is the size of each element?

4

(because we will be storing `ints` in the stack)



Example: Building a generic stack

Now we can try it. Let's push on an array of `ints`, and then pop them all off:

```
int main(int argc, char **argv)
{
    // start with an int array
    int iarr[] = {0, 2, 4, 6, 8, 12345678, 24680};
    int nelems = sizeof(iarr) / sizeof(iarr[0]);

    stack *intstack = stack_create(sizeof(iarr[0]));
    for (int i=0; i < nelems; i++) {
        stack_push(intstack, iarr + i);
    }

    int popped_int;
    while (stack_pop(intstack, &popped_int)) {
        printf("%d\n", popped_int);
    }
    free(s); // clean up!
    return 0;
}
```

```
$ ./stack
24680
12345678
8
6
4
2
0
7
```



Example: Building a generic stack

Let's try and push one more int onto the stack (assume we do this before the call to free:

```
int main(int argc, char **argv)
{
    ...
    int x = 42;
    stack_push(intstack, x);
}
```

Does this work? Recall:

```
void stack_push(stack *s, const void *data)
```



Example: Building a generic stack

Let's try and push one more int onto the stack (assume we do this before the call to free:

```
int main(int argc, char **argv)
{
    ...
    int x = 42;
    stack_push(intstack, x);
}
```

Does this work? Recall:

```
void stack_push(stack *s, const void *data)
```

This does **not** work -- we need a pointer to x. So, we should do:

```
stack_push(intstack, &x);
```



Example: Building a generic stack

Let's go ahead and use an array of `char *` pointers -- remember, our stack is generic, and will work for any pointer! Let's push all the command line args onto the stack:

```
stack *s = stack_create(sizeof(argv[0]));  
for (int i=1; i < argc; i++) {  
    stack_push(s, argv+i);  
}  
  
char *next_arg;  
while (stack_pop(s, &next_arg)) {  
    printf("%s\n", next_arg);  
}
```

What is the size of each element?

We're pushing on all but the program name.



Example: Building a generic stack

Let's go ahead and use an array of `char *` pointers -- remember, our stack is generic, and will work for any pointer! Let's push all the command line args onto the stack:

```
stack *s = stack_create(sizeof(argv[0]));  
for (int i=1; i < argc; i++) {  
    stack_push(s, argv+i);  
}  
  
char *next_arg;  
while (stack_pop(s, &next_arg)) {  
    printf("%s\n", next_arg);  
}
```

What is the size of each element?

8

because the size of a `char *` pointer is 8.

We're pushing on all but the program name.



Example: Building a generic stack

Let's go ahead and use an array of `char *` pointers -- remember, our stack is generic, and will work for any pointer! Let's push all the command line args onto the stack:

```
stack *s = stack_create(sizeof(argv[0]));  
for (int i = 1; i < argc; i++) {  
    stack_push(s, argv+i);  
}  
  
char *next_arg;  
while (stack_pop(s, &next_arg)) {  
    printf("%s\n", next_arg);  
}
```

We're pushing on all but the program name.

```
$ ./stack here are  
                some words  
words  
some  
are  
here
```



Example: Building a generic stack

Can we push on one more string?

```
...  
string *h = "hello";  
stack_push(s, h);
```

This should work, right? `h` is a pointer! Recall:

```
void stack_push(stack *s, const void *data)
```



Example: Building a generic stack

Can we push on one more string?

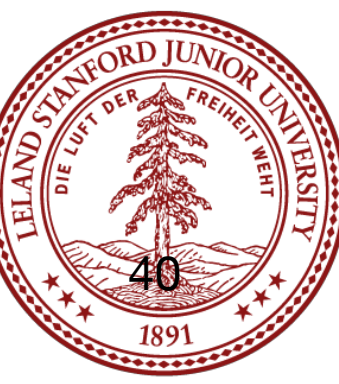
```
...  
char *h = "hello";  
stack_push(s, h);
```

This should work, right? `h` is a pointer! Recall:

```
void stack_push(stack *s, const void *data)
```

This doesn't work! We need a pointer *to the memory we are pushing onto the stack*. We aren't pushing string characters, we are pushing a string pointer! So, we need:

```
stack_push(s, &h); // &h is a char **
```



References and Advanced Reading

- **References:**

- K&R C Programming (from our course)
- Course Reader, C Primer
- Awesome C book: <http://books.goalkicker.com/CBook>
- Function Pointer tutorial: <https://www.cprogramming.com/tutorial/function-pointers.html>

- **Advanced Reading:**

- virtual memory: https://en.wikipedia.org/wiki/Virtual_memory



Extra Slides

