

COMP201

Computer Systems & Programming

Lecture #15 – More Function Pointers, const, structs



KOÇ
UNIVERSITY

Aykut Erdem // Koç University // Fall 2020

Recap

- Generics So Far
- Motivating Example: Bubble Sort
- Function Pointers

Recap: Generics Overview

- We use **void *** pointers and memory operations like **memcpy** and **memmove** to make data operations generic.
- We use **function pointers** to make logic/functionality operations generic.

Plan for Today

- Function Pointers (cont'd.)
- `const`
- `struct`
- Generic stack

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

Lecture Plan

- Function Pointers (cont'd.)
- `const`
- `Struct`
- Generic stack

Function Pointers

- Function pointers can be used in a variety of ways. For instance, you could have:
 - A function to compare two elements of a given type
 - A function to print out an element of a given type
 - A function to free memory associated with a given type
 - And more...

Function Pointers

- Function pointers can be used in a variety of ways. For instance, you could have:
 - A function to compare two elements of a given type
 - **A function to print out an element of a given type**
 - A function to free memory associated with a given type
 - And more...

Demo: Generic Printing



print_array.c

Common Utility Callback Functions

- Comparison function – compares two elements of a given type.

```
int (*cmp_fn)(void *addr1, void *addr2)
```

- Printing function – prints out an element of a given type

```
void (*print_fn)(void *addr)
```

- There are many more! You can specify any functions you would like passed in when writing your own generic functions.

Demo: Count Matches



```
count_matches.c
```

Count Matches

- Let's write a generic function *count_matches* that can count the number of a certain type of element in a generic array.
- It should take in as parameters information about the generic array, and a function parameter that can take in a pointer to a single array element and tell us if it's a match.

```
int count_matches(void *base, int nelems,  
                  int elem_size_bytes, bool (*match_fn)(void *));
```

Count Matches

```
int count_matches(void *base, int nelems, int elem_size_bytes,  
                  bool (*match_fn)(void *)) {  
  
    int match_count = 0;  
  
    for (int i = 0; i < nelems; i++) {  
        void *curr_p = (char *)base + i * elem_size_bytes;  
        if (match_fn(curr_p)) {  
            match_count++;  
        }  
    }  
  
    return match_count;  
}
```

Function Pointers As Variables

In addition to parameters, you can make normal variables that are functions.

```
int do_something(char *str) {  
    ...  
}  
  
int main(int argc, char *argv[]) {  
    ...  
    int (*func_var)(char *) = do_something;  
    ...  
    func_var("testing");  
    return 0;  
}
```

Generic C Standard Library Functions

- **qsort** – I can sort an array of any type! To do that, I need you to provide me a function that can compare two elements of the kind you are asking me to sort.
- **bsearch** – I can use binary search to search for a key in an array of any type! To do that, I need you to provide me a function that can compare two elements of the kind you are asking me to search.
- **lfind** – I can use linear search to search for a key in an array of any type! To do that, I need you to provide me a function that can compare two elements of the kind you are asking me to search.
- **lsearch** - I can use linear search to search for a key in an array of any type! I will also add the key for you if I can't find it. In order to do that, I need you to provide me a function that can compare two elements of the kind you are asking me to search.

Generic C Standard Library Functions

- **scandir** – I can create a directory listing with any order and contents! To do that, I need you to provide me a function that tells me whether you want me to include a given directory entry in the listing. I also need you to provide me a function that tells me the correct ordering of two given directory entries.

Summary: Function Pointers

- We can pass functions as parameters to pass logic around in our programs.
- Comparison functions are one common class of functions passed as parameters to generically compare the elements at two addresses.
- Functions handling generic data must use *pointers to the data they care about*, since any parameters must have *one type* and *one size*.

Lecture Plan

- Function Pointers (cont'd.)
- **const**
- Struct
- Generic stack

const

- Use **const** to declare global constants in your program. This indicates the variable cannot change after being created.

```
const double PI = 3.1415;  
const int DAYS_IN_WEEK = 7;
```

```
int main(int argc, char *argv[]) {  
    ...  
    if (x == DAYS_IN_WEEK) {  
        ...  
    }  
    ...  
}
```

const

- Use **const** with pointers to indicate that the data that is pointed to cannot change.

```
char str[6];
```

```
strcpy(str, "Hello");
```

```
const char *s = str;
```

```
// Cannot use s to change characters it points to
```

```
s[0] = 'h';
```

const

Sometimes we use **const** with pointer parameters to indicate that the function will not / should not change what it points to. The actual pointer can be changed, however.

// This function promises to not change str's characters

```
int countUppercase(const char *str) {  
    int count = 0;  
    for (int i = 0; i < strlen(str); i++) {  
        if (isupper(str[i])) {  
            count++;  
        }  
    }  
    return count;  
}
```


const

By definition, C gets upset when you set a **non-const** pointer equal to a **const** pointer. You need to be consistent with **const** to reflect what you cannot modify.

```
// This function promises to not change str's characters
int countUppercase(const char *str) {
    // compiler warning and error
    char *strToModify = str;
    strToModify[0] = ...
}
```

const

By definition, C gets upset when you set a **non-const** pointer equal to a **const** pointer. You need to be consistent with **const** to reflect what you cannot modify. **Think of const as part of the variable type.**

```
// This function promises to not change str's characters
int countUppercase(const char *str) {
    const char *strToModify = str;
    strToModify[0] = ...
}
```

const

const can be confusing to interpret in some variable types.

```
// cannot modify this char
```

```
const char c = 'h';
```

```
// cannot modify chars pointed to by str
```

```
const char *str = ...
```

```
// cannot modify chars pointed to by *strPtr
```

```
const char **strPtr = ...
```

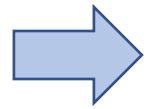
Practice

const

```
1 char buf[6];  
2 strcpy(buf, "Hello");  
3 const char *str = buf;  
4 str[0] = 'M';  
5 str = "Mello";  
6 buf[0] = 'M';
```

Which lines (if any) above will cause an error due to violating `const`?
Remember that `const char *` means that the characters at the location it stores cannot be changed.

const

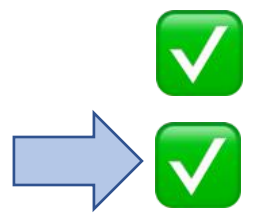


```
1 char buf[6];  
2 strcpy(buf, "Hello");  
3 const char *str = buf;  
4 str[0] = 'M';  
5 str = "Mello";  
6 buf[0] = 'M';
```

Line 1 makes a typical modifiable character array of 6 characters.

Which lines (if any) above will cause an error due to violating `const`?
Remember that `const char *` means that the characters at the location it stores cannot be changed.

const

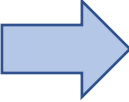


```
1 char buf[6];  
2 strcpy(buf, "Hello");  
3 const char *str = buf;  
4 str[0] = 'M';  
5 str = "Mello";  
6 buf[0] = 'M';
```

Line 2 copies characters into this modifiable character array.

Which lines (if any) above will cause an error due to violating `const`?
Remember that `const char *` means that the characters at the location it stores cannot be changed.

const



✓	1	char buf[6];
✓	2	strcpy(buf, "Hello");
✓	3	const char *str = buf;
	4	str[0] = 'M';
	5	str = "Mello";
	6	buf[0] = 'M';

Line 3 makes a const pointer that points to the first element of buf. We cannot use str to change the characters it points to because it is const.

Which lines (if any) above will cause an error due to violating **const**?
Remember that `const char *` means that the characters at the location it stores cannot be changed.

const

✓	1	char buf[6];
✓	2	strcpy(buf, "Hello");
✓	3	const char *str = buf;
→ ✗	4	str[0] = 'M';
	5	str = "Mello";
	6	buf[0] = 'M';

Line 4 is not allowed – it attempts to use a const pointer to characters to modify those characters.

Which lines (if any) above will cause an error due to violating **const**?
Remember that `const char *` means that the characters at the location it stores cannot be changed.

const

✓	1	char buf[6];
✓	2	strcpy(buf, "Hello");
✓	3	const char *str = buf;
✗	4	str[0] = 'M';
→ ✓	5	str = "Mello";
	6	buf[0] = 'M';

Line 5 is ok – `str`'s type means that while you cannot change the characters at which it points, you can change `str` itself to point somewhere else. `str` is not `const` – its characters are.

Which lines (if any) above will cause an error due to violating `const`?
Remember that `const char *` means that the characters at the location it stores cannot be changed.

const

✓	1	<code>char buf[6];</code>
✓	2	<code>strcpy(buf, "Hello");</code>
✓	3	<code>const char *str = buf;</code>
✗	4	<code>str[0] = 'M';</code>
✓	5	<code>str = "Mello";</code>
→ ✓	6	<code>buf[0] = 'M';</code>

Line 6 is ok – `buf` is a modifiable char array, and we can use it to change its characters. Declaring `str` as `const` doesn't mean that place in memory is not modifiable at all – it just means that you cannot modify it using `str`.

Which lines (if any) above will cause an error due to violating `const`?
Remember that `const char *` means that the characters at the location it stores cannot be changed.

Lecture Plan

- Function Pointers (cont'd.)
- `const`
- **`struct`**
- Generic stack

Structs

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

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

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.

```
typedef struct date {  
    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.

```
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;  
}
```

Structs

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.**

```
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;  
}
```

Structs

The **arrow** operator lets you access the field of a struct pointed to by a pointer.

```
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;  
}
```

Structs

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.

```
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.

```
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.

```
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.

```
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

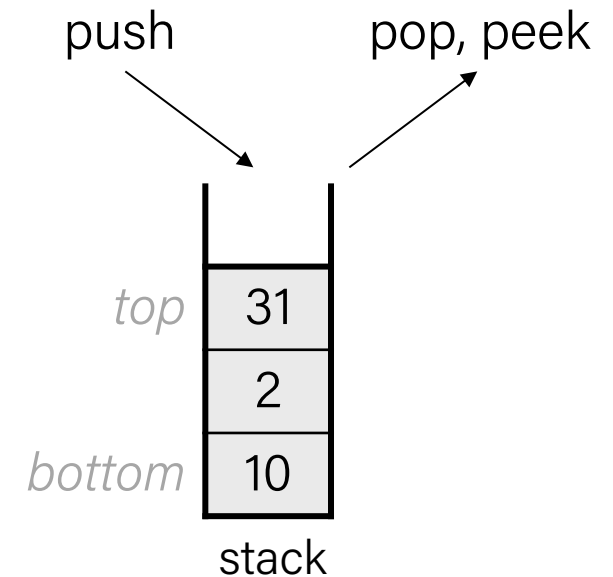
- Function Pointers (cont'd.)
- `const`
- `struct`
- Generic stack

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

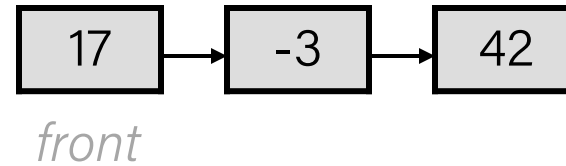


Stacks

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?*)

```
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!

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, 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

```
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

```
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

```
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

```
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

Recap

- Function Pointers (cont'd.)
- **const**
- **struct**
- Generic stack

Next Time: Compiling C programs