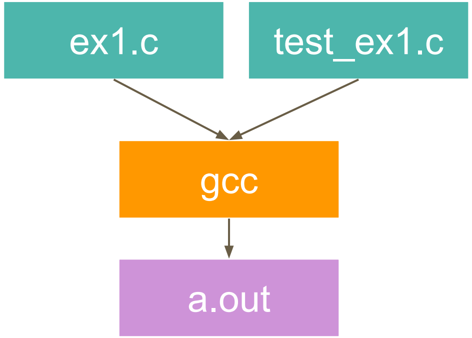
Lab 1

61C Summer 2023

Compiling a C Program

- `gcc` is used to compile C programs
- `gcc ex1.c test_ex1.c`



- Can specify the name of the executable file with the `-o` flag
  - `gcc -o ex1 ex1.c test_ex1.c`

Running a C Program

- To run an executable located in the current directory, use `./<executable_name>`
  - `./ex1`
- The dot refers to the current directory
- If you want to run a file in a different directory, specify the path after the dot
  - `./path/to/file/ex1`

Variable Types and Sizes

Guarantee: sizeof(long long) >= sizeof(long) >= sizeof(int) >= sizeof(short)

To know for sure what size your variable is, **use `uintN_t` or `intN_t` types.**

- `char` = 1 byte (8 bits)
- `short` = at least 2 bytes (16 bits), can be longer
- `int` = at least 2 bytes (16 bits), can be longer
  - `unsigned int`
- `float` = 4 bytes (32 bits)
- `double` = 8 bytes (64 bits)
- `long` = at least 4 bytes (32 bits), can be longer
- `long long` = at least 8 bytes (64 bits), can be longer

Defining a Function

Specify return type, function name, and function parameters.

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

     void nothing() {     return;     }
```

Conditionals

If-else

```
if (condition) {
    do this;
} else if {
    do this;
} else {
    do this;
}
```

Switch statements

```
switch (expression) {
    case constant1:
        do these;
        break;
    case constant2:
        do these;
        break;
    default:
        do these;
}
```

Loops

While loop

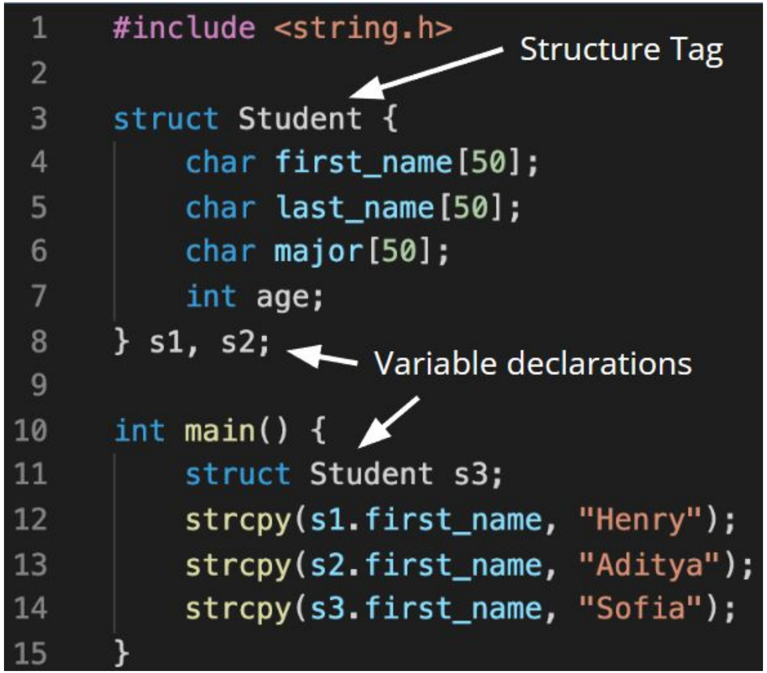
```
while (condition) {
    do this;
}
```

For loop

```
for (int i = 0; i < 10; i++) {
    do this;
}
```

Structs

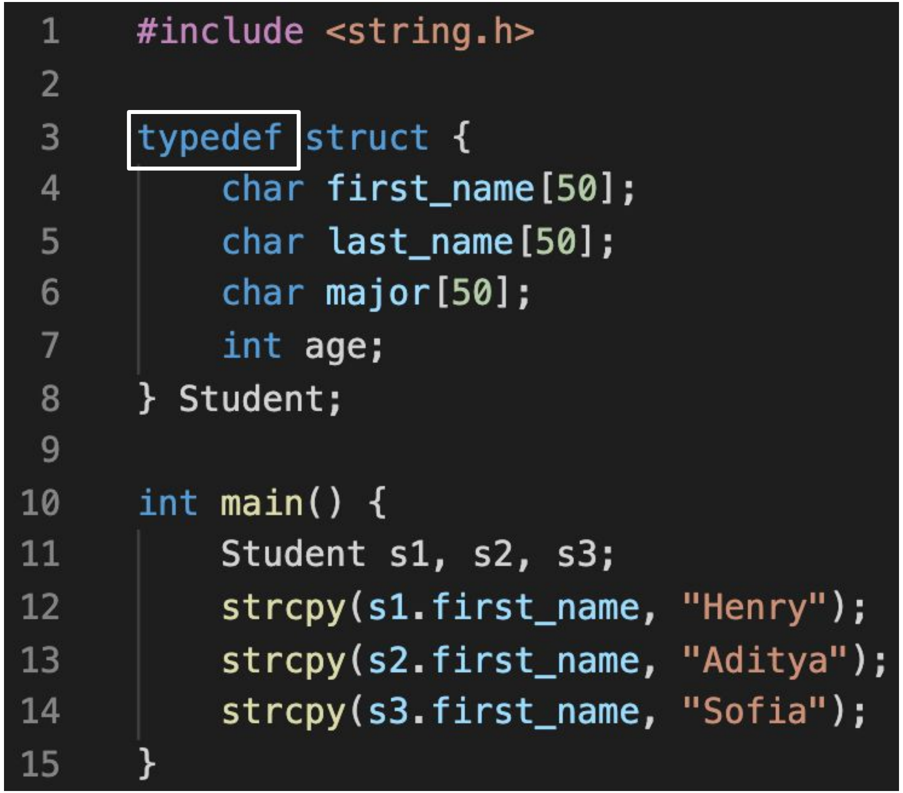
- Structs allow us to hold data items of different types in a single variable
- Structure Tag: optional, allows you to create new variables of this struct outside of the struct definition
- Two ways to declare struct variables
  - s1, s2: When you define the struct
  - s3: Using the struct tag
- Two ways to access members
  - Use dot operator(.) with structs
  - Use arrow operator(->) with pointers to structs



```c
#include <string.h>

struct Student {
    char first_name[50];
    char last_name[50];
    char major[50];
    int age;
} s1, s2;

int main() {
    struct Student s3;
    strcpy(s1.first_name, "Henry");
    strcpy(s2.first_name, "Aditya");
    strcpy(s3.first_name, "Sofia");
}
```

Structs

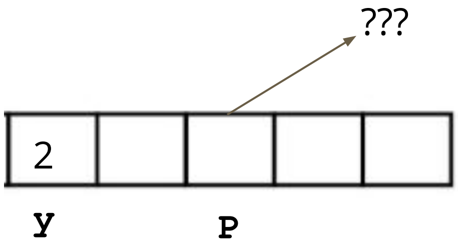
- typedef
  - Lets you avoid rewriting `struct` every time you want to declare a new struct variable
  - Can no longer declare variables in the struct definition



```c
#include <string.h>

typedef struct {
    char first_name[50];
    char last_name[50];
    char major[50];
    int age;
} Student;

int main() {
    Student s1, s2, s3;
    strcpy(s1.first_name, "Henry");
    strcpy(s2.first_name, "Aditya");
    strcpy(s3.first_name, "Sofia");
}
```

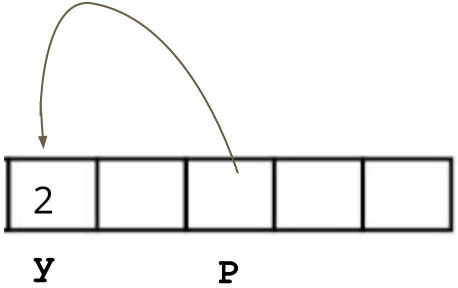
Pointers

- Pointers are variables that contain memory locations (addresses) of other variables
- int *p;
  - Variable p is a pointer to an int
  - p hasn't been initialized yet



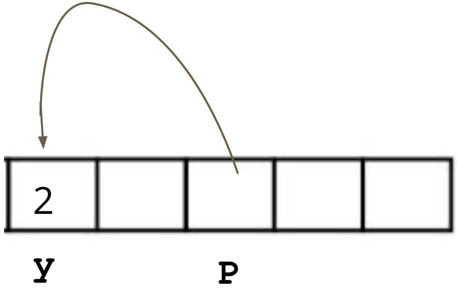
Pointers

- Pointers are variables that contain memory locations (addresses) of other variables
- int *p;
  - Variable p is a pointer to an int
- p = &y;
  - & called the “address operator”
  - p now points to (contains the address of) y

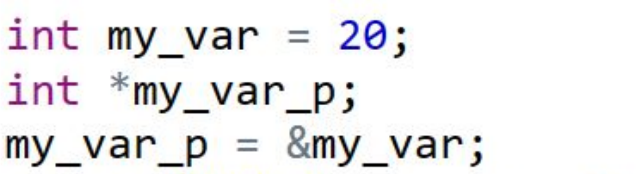


Pointers

- Pointers are variables that contain memory locations (addresses) of other variables
- int *p;
  - Variable p is a pointer to an int
- p = &y;
  - & called the “address operator”
  - p now points to (contains the address of) y
- *p
  - * called the “dereference operator”
  - *p == 2



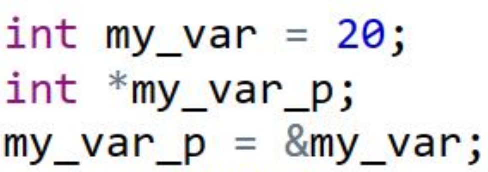
Pointers



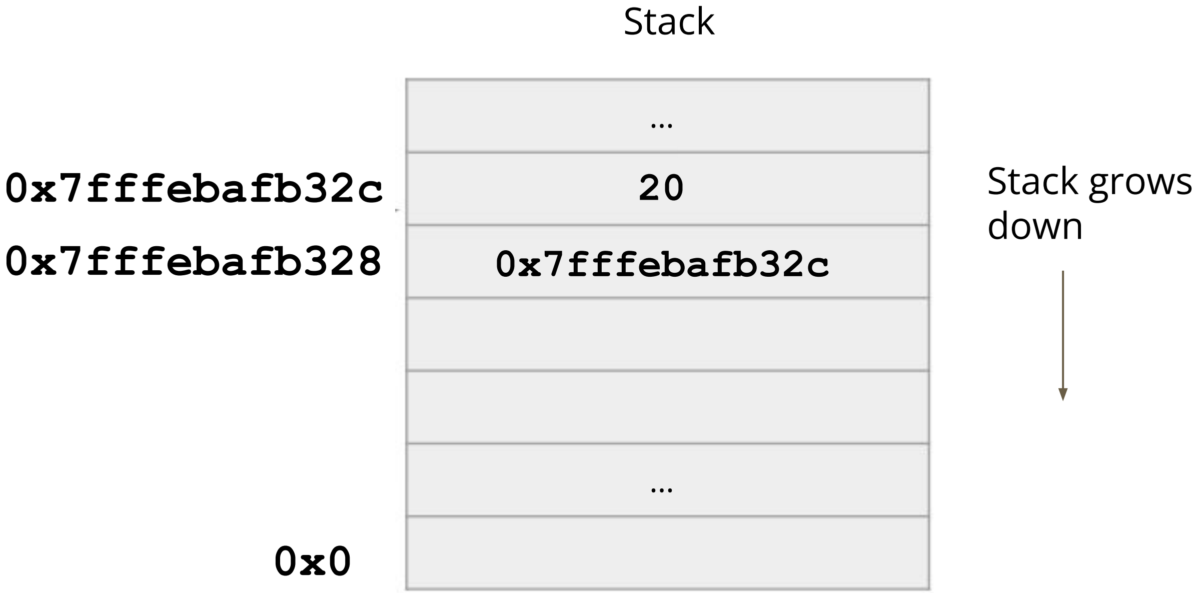
```
int my_var = 20;
int *my_var_p;
my_var_p = &my_var;
```

What does memory look like?

Pointers



```
int my_var = 20;
int *my_var_p;
my_var_p = &my_var;
```

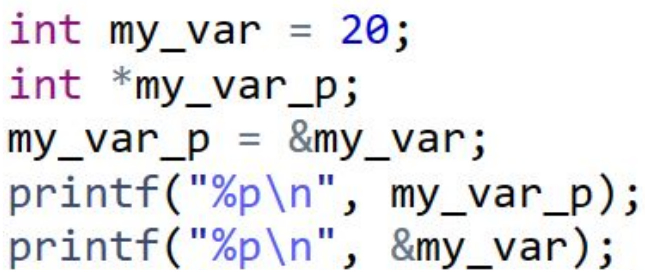


Stack

	…
0x7fffebafb32c	20
0x7fffebafb328	0x7fffebafb32c
	…
0x0	

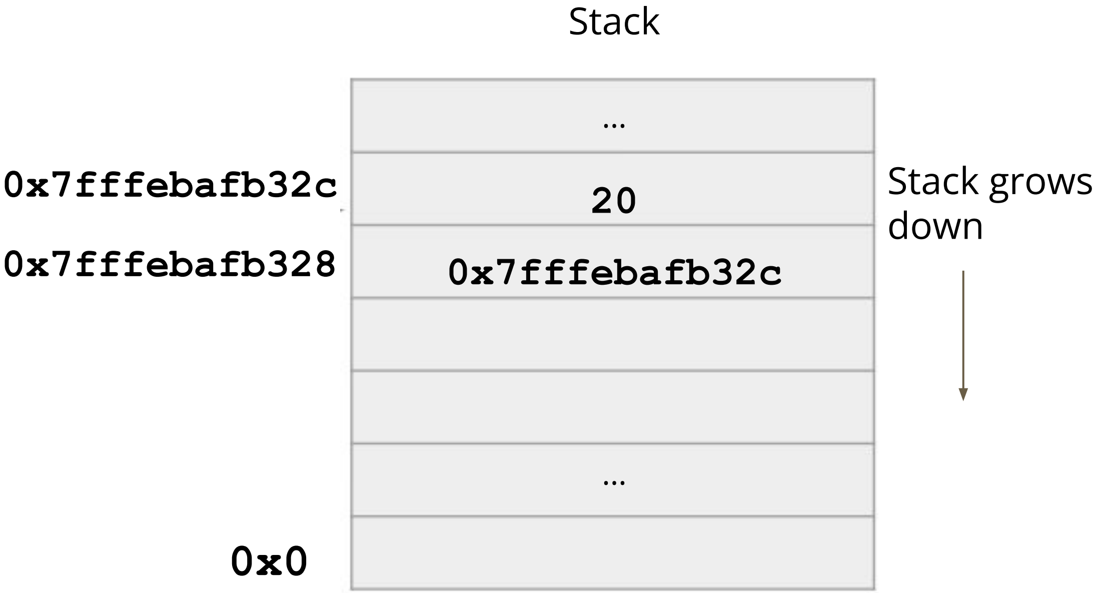
Stack grows down

Pointers

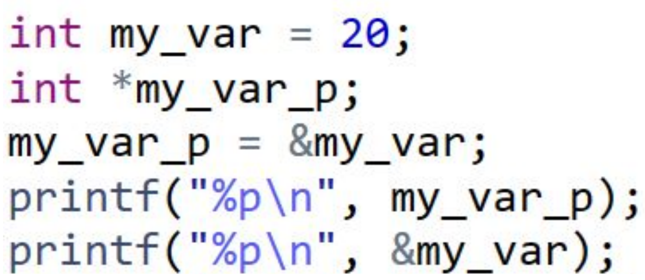


```
int my_var = 20;
int *my_var_p;
my_var_p = &my_var;
printf("%p\n", my_var_p);
printf("%p\n", &my_var);
```

What will be printed?



Pointers

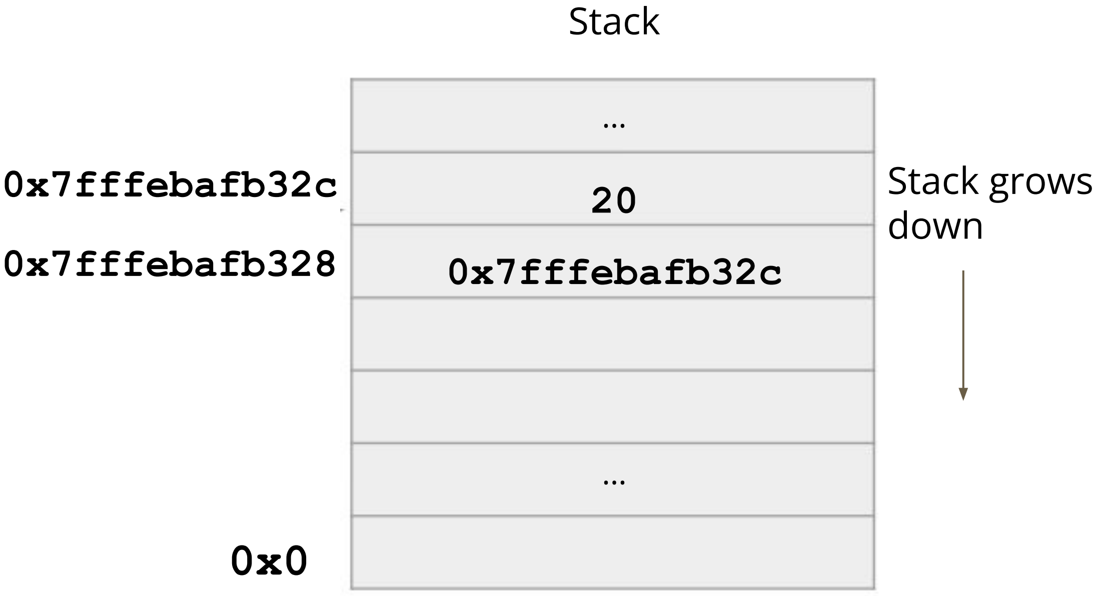


```
int my_var = 20;
int *my_var_p;
my_var_p = &my_var;
printf("%p\n", my_var_p);
printf("%p\n", &my_var);
```

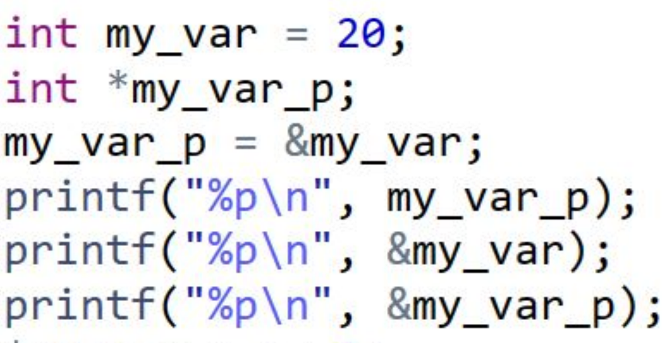
What will be printed?

Output:

```
0x7fffebafb32c
0x7fffebafb32c
```



Pointers

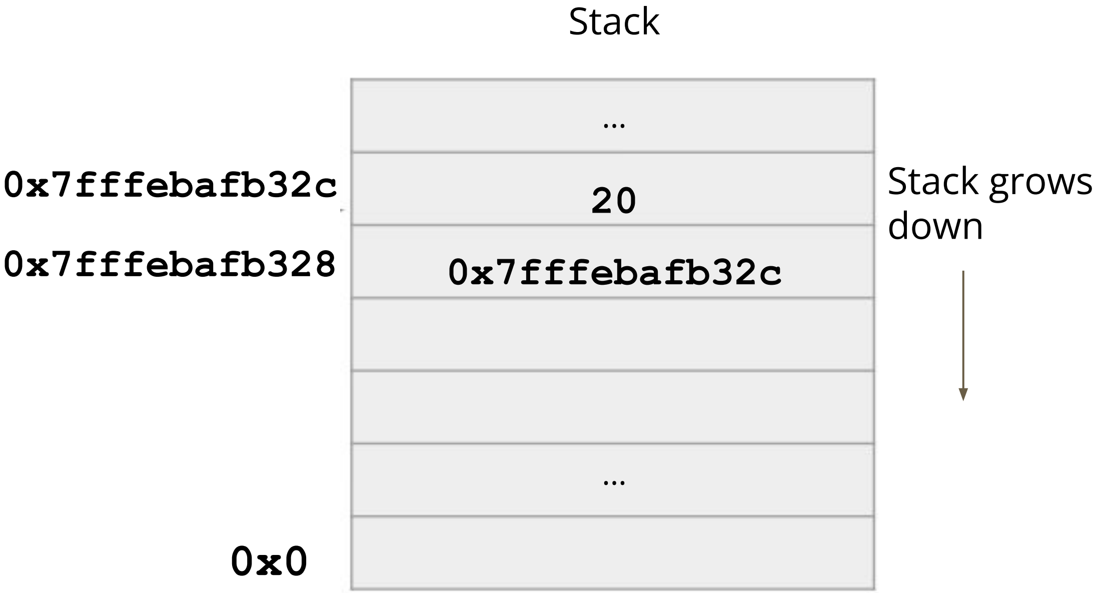


```
int my_var = 20;
int *my_var_p;
my_var_p = &my_var;
printf("%p\n", my_var_p);
printf("%p\n", &my_var);
printf("%p\n", &my_var_p);
```

What will be printed?

Output:

```
0x7fffebafb32c
0x7fffebafb32c
```



Stack

...

0x7fffebafb32c
20

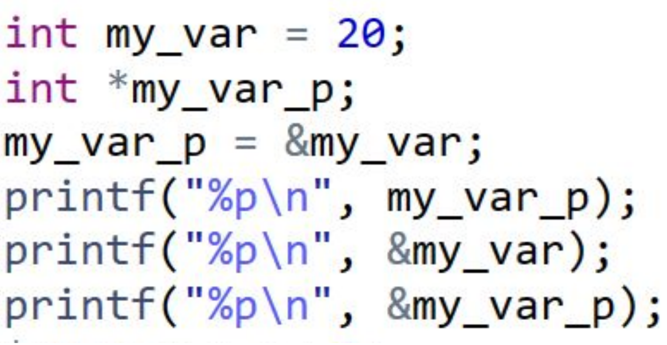
0x7fffebafb328
0x7fffebafb32c

...

0x0

Stack grows down

Pointers

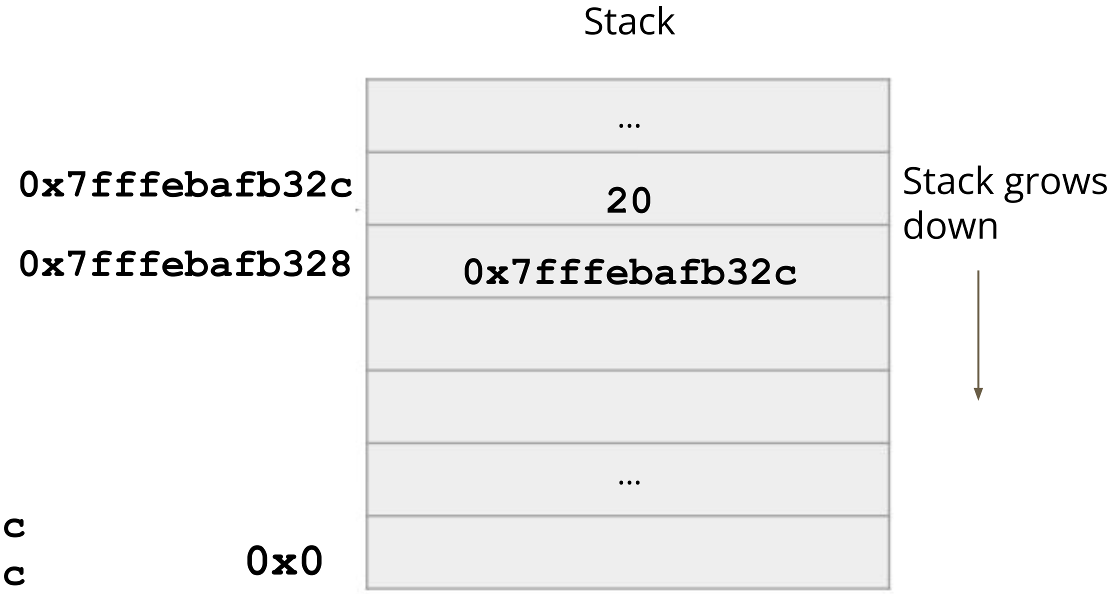


```c
int my_var = 20;
int *my_var_p;
my_var_p = &my_var;
printf("%p\n", my_var_p);
printf("%p\n", &my_var);
printf("%p\n", &my_var_p);
```

What will be printed?

Output:

```
0x7fffebafb32c
0x7fffebafb32c
0x7fffebafb328
```

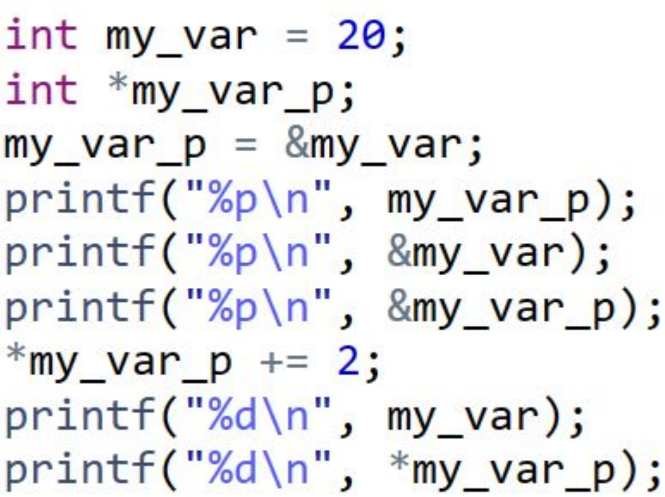


Stack

	...
0x7fffebafb32c	20
0x7fffebafb328	0x7fffebafb32c
	...
0x0	

Stack grows down

Pointers

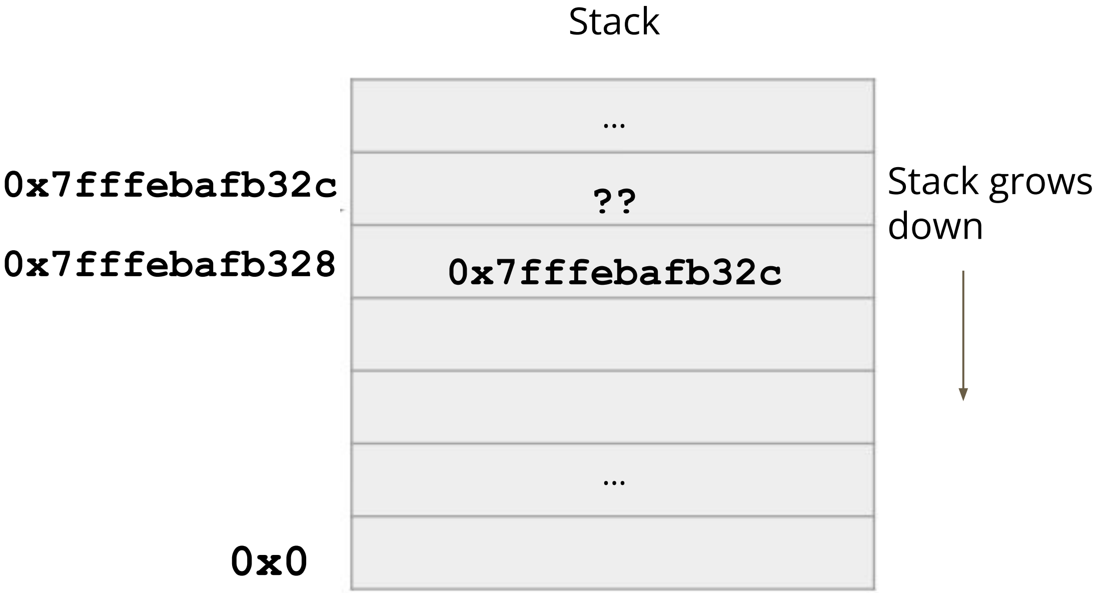


```
int my_var = 20;
int *my_var_p;
my_var_p = &my_var;
printf("%p\n", my_var_p);
printf("%p\n", &my_var);
printf("%p\n", &my_var_p);
*my_var_p += 2;
printf("%d\n", my_var);
printf("%d\n", *my_var_p);
```

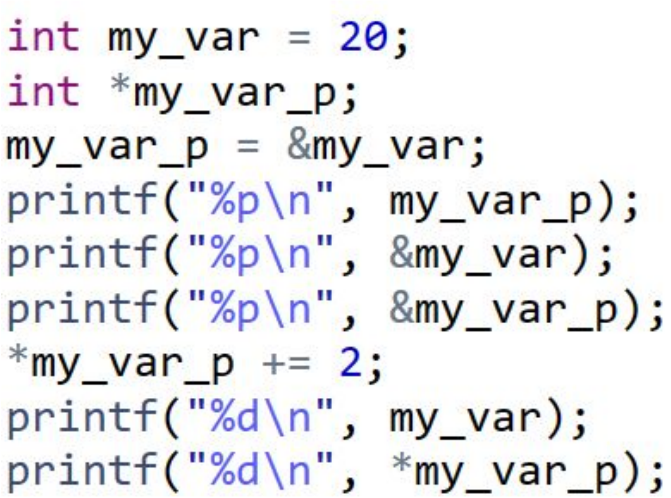
What will be printed?

Output:

```
0x7fffebafb32c
0x7fffebafb32c
0x7fffebafb328
```



Pointers

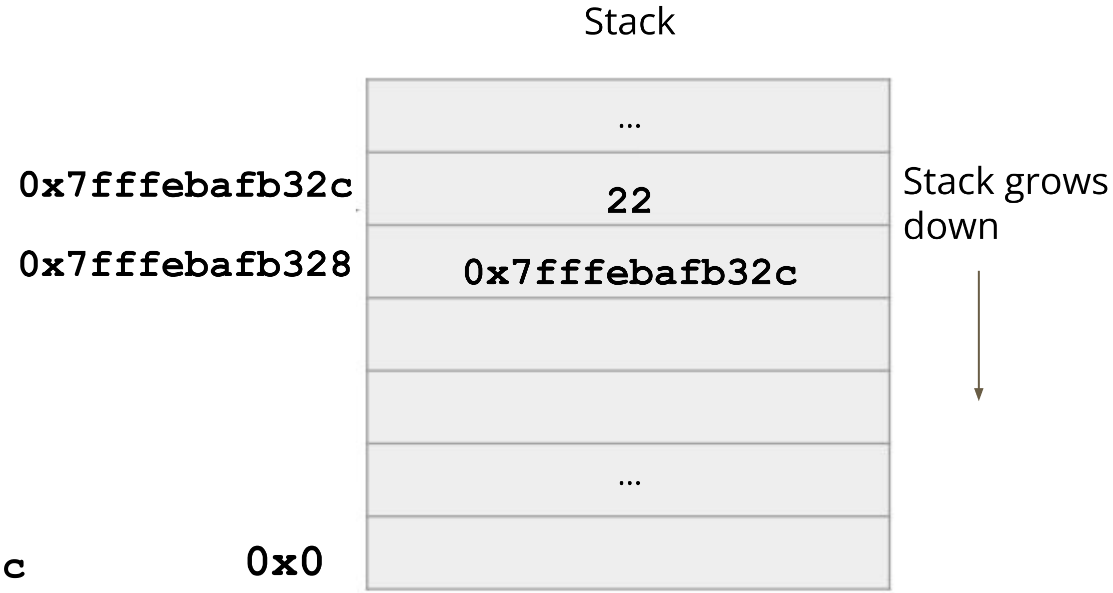


```c
int my_var = 20;
int *my_var_p;
my_var_p = &my_var;
printf("%p\n", my_var_p);
printf("%p\n", &my_var);
printf("%p\n", &my_var_p);
*my_var_p += 2;
printf("%d\n", my_var);
printf("%d\n", *my_var_p);
```

What will be printed?

Output:

```
0x7fffebafb32c
0x7fffebafb32c
0x7fffebafb328
22
22
```

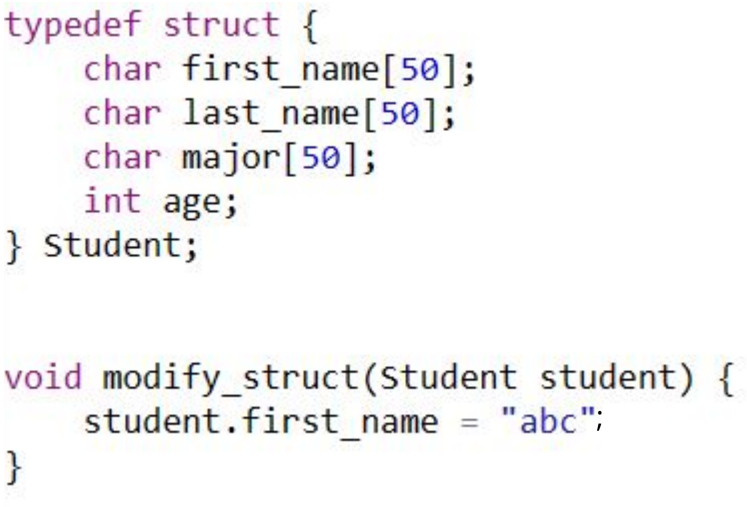


Pointers to Structs

- How to change struct’s fields in a function?

Pointers to Structs

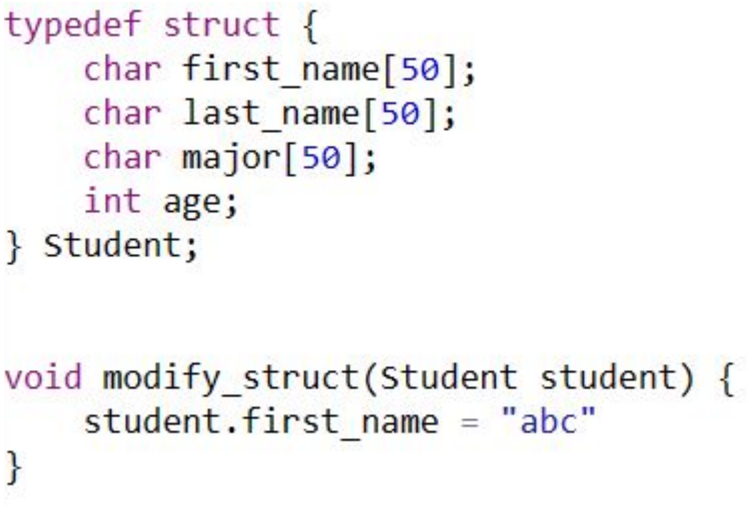
- How to change struct’s fields in a function?
  - Why doesn’t this work?



```
typedef struct {
    char first_name[50];
    char last_name[50];
    char major[50];
    int age;
} Student;


void modify_struct(Student student) {
    student.first_name = "abc";
}
```

Pointers to Structs

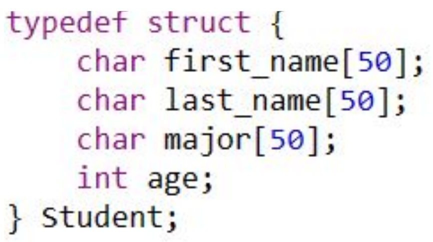
- How to change struct’s fields in a function?
    - Why doesn’t this work?
    - C is ‘pass by value’ so we would only be modifying copies of the struct



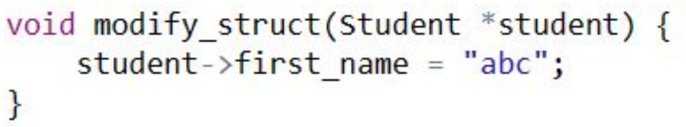
```c
typedef struct {
    char first_name[50];
    char last_name[50];
    char major[50];
    int age;
} Student;


void modify_struct(Student student) {
    student.first_name = "abc"
}
```

Pointers to Structs

- How to change struct’s fields in a function?
  - Why doesn’t this work?
  - C is ‘pass by value’ so we would only be modifying copies of the struct
- Correct way is to pass pointer to struct
  - student->first_name
  - (*student).first_name



```
typedef struct {
    char first_name[50];
    char last_name[50];
    char major[50];
    int age;
} Student;
```



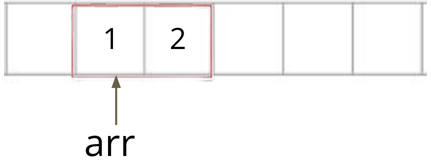
```
void modify_struct(Student *student) {
    student->first_name = "abc";
}
```

Arrays

- A block of memory: size is **static**
  - `int arr[2];`
  - `int arr[] = {1, 2};`
- Accessing elements: array indexing
  - `arr[1]`

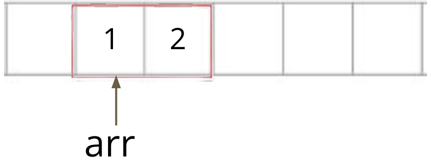
Arrays

- A block of memory: size is **static**
  - `int arr[2];`
  - `int arr[] = {1, 2};`
- Accessing elements: array indexing
  - `arr[1]`
- An array variable is a "pointer" to the first element.



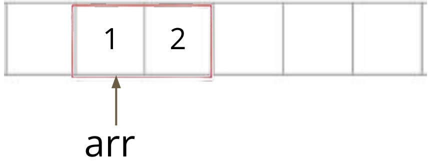
Arrays

- A block of memory: size is **static**
  - `int arr[2];`
  - `int arr[] = {1, 2};`
- Accessing elements: array indexing
  - `arr[1]`
- An array variable is a “pointer” to the first element.
  - You can use pointers to access arrays!
  - `arr[0]` is the same as `*arr`
  - Can use pointer arithmetic to move the pointer
    - Each operation automatically moves the size of one whole “type” that ptr points to
    - `arr[1]` is the same as `*(arr + 1)`



Arrays

- A block of memory: size is **static**
  - `int arr[2];`
  - `int arr[] = {1, 2};`
- Accessing elements: array indexing
  - `arr[1]`
- An array variable is a “pointer” to the first element.
  - You can use pointers to access arrays!
  - `arr[0]` is the same as `*arr`
  - Can use pointer arithmetic to move the pointer
    - Each operation automatically moves the size of one whole “type” that ptr points to
    - `arr[1]` is the same as `*(arr + 1)`
  - However, arr++ or arr = arr + x wouldn’t work because you can’t increment arrays
    - Instead, initialize another variable to point at arr and increment it: int *ptr = arr
    - Can also use array indexing with pointers so ptr[1] == arr[1]



Arrays (Cont.)

- Arrays aren’t exactly traditional pointers because they don’t occupy separate space aside from the block of memory allocated to the array itself
  - Consequently, arr == &arr; and &arr is also the memory location of the first element

Strings

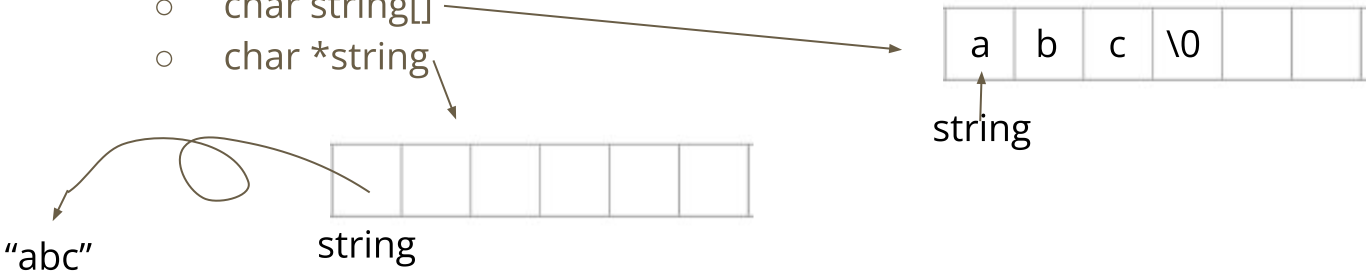
- In C, Strings are just char arrays with a ‘\0’ (null terminator).
  - Functions like strlen use the null terminator to determine where the array ends to calculate length
  - Strcpy also copies one character at a time from one location to another until the null terminator

Strings

- In C, Strings are just char arrays with a '\0' (null terminator).
  - Functions like strlen use the null terminator to determine where the array ends to calculate length
  - Strcpy also copies one character at a time from one location to another until the null terminator
- char *string = "abc" or char string[] = "abc"
  - char string[] creates a char array that contains {'a', 'b', 'c', '\0'}while char *string creates a pointer to a string literal in static portion of memory

Strings

- In C, Strings are just char arrays with a ‘\0’ (null terminator).
  - Functions like strlen use the null terminator to determine where the array ends to calculate length
  - Strcpy also copies one character at a time from one location to another until the null terminator
- char *string = “abc” or char string[] = “abc”
  - char string[] creates a char array that contains {‘a’, ‘b’, ‘c’, ‘\0’}while char *string creates a pointer to a string literal in static portion of memory
  - char string[]
  - char *string



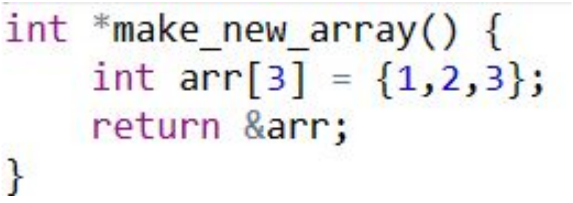
Dynamic Memory

- Pointers can be allocated using malloc that persist in the heap
- Heap is a memory space separate from other variables declared in the stack
- Always remove dynamically allocated pointers by calling free(ptr); after use

Dynamic Memory

- Pointers can be allocated using malloc that persist in the heap
- Heap is a memory space separate from other variables declared in the stack
- Always remove dynamically allocated pointers by calling free(ptr); after use

Why is this wrong

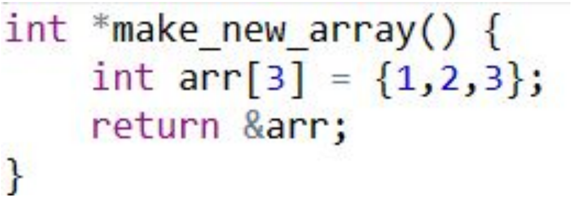


```
int *make_new_array() {
    int arr[3] = {1,2,3};
    return &arr;
}
```

Dynamic Memory

- Pointers can be allocated using malloc that persist in the heap
- Heap is a memory space separate from other variables declared in the stack
- Always remove dynamically allocated pointers by calling free(ptr); after use

Why is this wrong



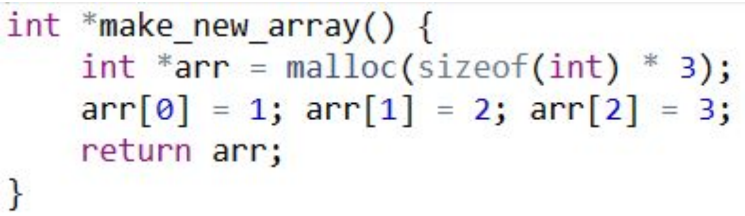
```
int *make_new_array() {
    int arr[3] = {1,2,3};
    return &arr;
}
```

arr will be thrown away after the function returns and stack pointer is moved above the function frame

Dynamic Memory

- Pointers can be allocated using malloc that persist in the heap
- Heap is a memory space separate from other variables declared in the stack
- Always remove dynamically allocated pointers by calling free(ptr); after use

Correct



```
int *make_new_array() {
    int *arr = malloc(sizeof(int) * 3);
    arr[0] = 1; arr[1] = 2; arr[2] = 3;
    return arr;
}
```