/unc/comp211

## Systems Fundamentals

# Lifetimes!

# What is the *lifetime* of a memory address?

- Two questions to assess the ***lifetime*** of a memory address:
  - When is it ***safe*** to access or "read"?
  - When does it ***expire***?
  - Both answers depend on how the memory is *allocated* and *deallocated*!

- **Local variables** are **"automatic" variables** in C
  - Their space is *automatically* allocated/deallocated on function call/return

- The ***lifetime*** of a ***memory address*** of an **automatic/local/stack variable** is:
  - **Safe after initialization**.
  - **Expired once out of scope**.

# Why haven't you needed to worry too much about *lifetimes* before?

- In C, you can read memory addresses both *before it is safe to* and *after they expire*.
    - Very lucky: the compiler will emit warnings.
    - Somewhat lucky: the program will crash quickly and predictably.
    - Unlucky: program may not crash for days, months, years or predictably.
    - **C is *not*** a ***memory safe*** language.

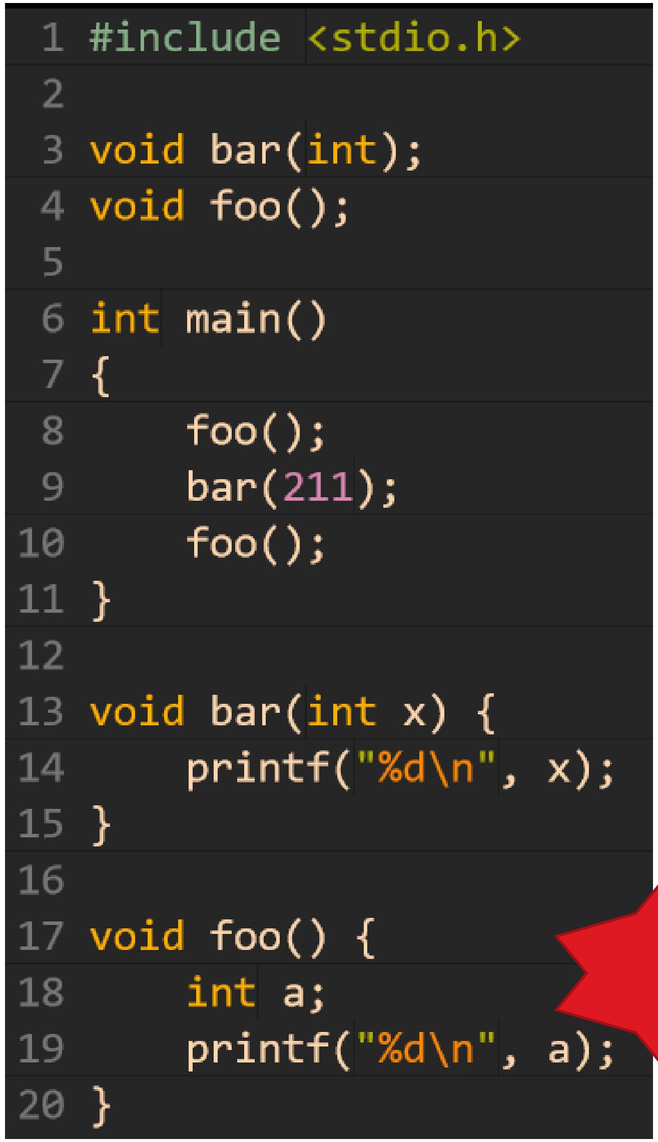
- Languages like Java, TypeScript, Python ***are memory safe***.

- When you access memory via a variable in a *memory safe* language you have some guarantees:
    1. If it's valid, you'll read the contents back directly.
    2. If it's invalid (null pointer, index out of bounds) an exception is always raised.

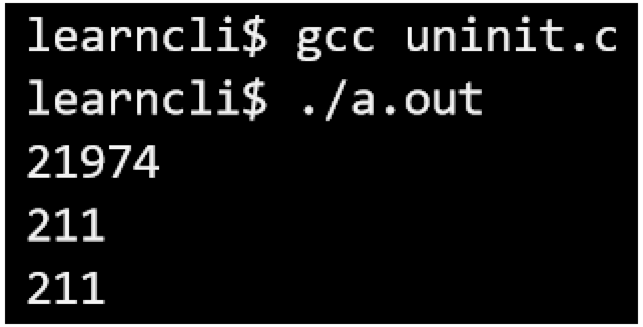
- There are trade-offs to achieve memory safety:
    - For memory-managed languages like Java, you can't pass references to stack values, your heap must be garbage collected, overhead in array access, etc. Generally less optimal in both time and space.
    - For modern systems languages like Rust, the trade-off is additional syntax for communicating lifetime guarantees to the compiler so that it can prove all memory accesses are valid.

# 0. Consider the Following Code...

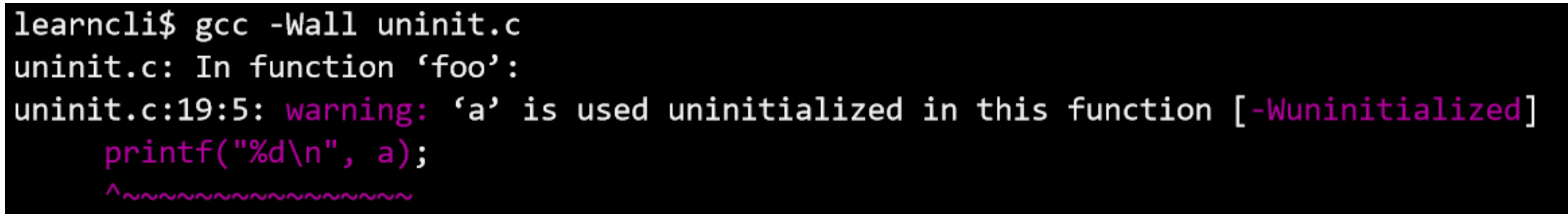
1. Does it compile?
2. Does it run?
3. What is its output?



```c
 1 #include <stdio.h>
 2 
 3 void bar(int);
 4 void foo();
 5 
 6 int main()
 7 {
 8     foo();
 9     bar(211);
10     foo();
11 }
12 
13 void bar(int x) {
14     printf("%d\n", x);
15 }
16 
17 void foo() {
18     int a;
19     printf("%d\n", a);
20 }
```



```
learncli$ gcc uninit.c
learncli$ ./a.out
21974
211
211
```



```
learncli$ gcc -Wall uninit.c
uninit.c: In function ‘foo’:
uninit.c:19:5: warning: ‘a’ is used uninitialized in this function [-Wuninitialized]
     printf("%d\n", a);
     ^~~~~~~~~~~~~~~~~
```

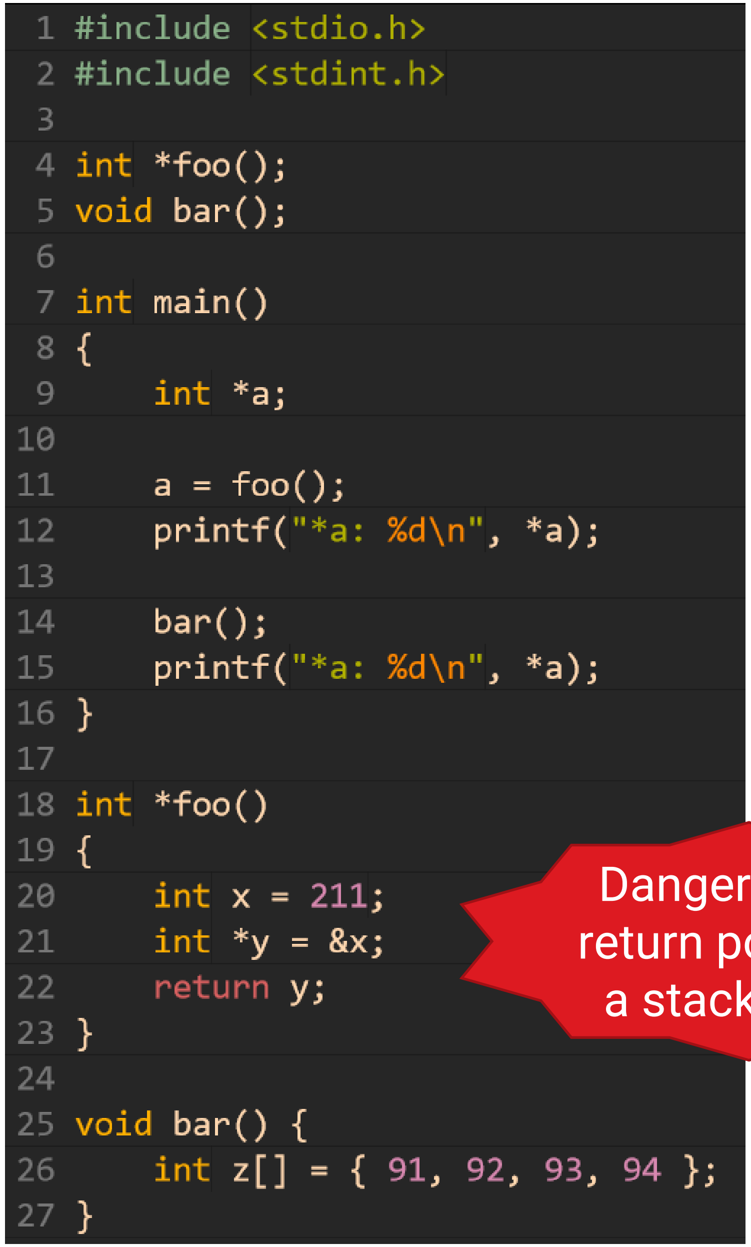
YIKES!
Always initialize
before Access!!!

This is *quite* scary!

Only *with warnings* does it give you a warning you're accessing an uninitialized value. It still compiles! It still runs! The value is trash!

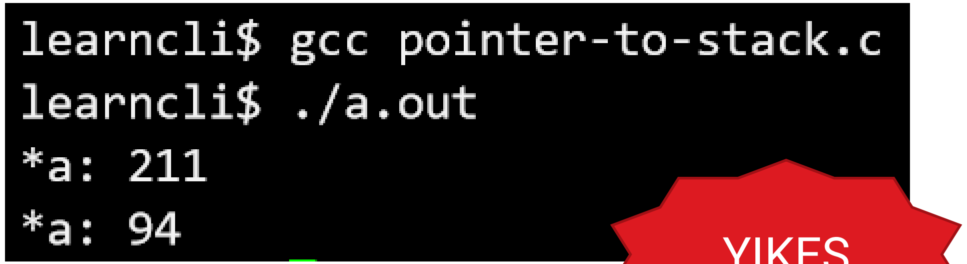
# 1. Consider the Following Code...

1. Does it compile?
2. Does it run?
3. What is its output?



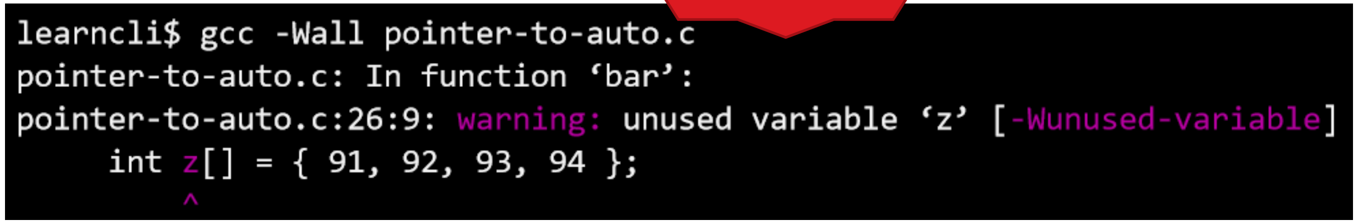
```c
 1 #include <stdio.h>
 2 #include <stdint.h>
 3
 4 int *foo();
 5 void bar();
 6
 7 int main()
 8 {
 9     int *a;
10
11     a = foo();
12     printf("*a: %d\n", *a);
13
14     bar();
15     printf("*a: %d\n", *a);
16 }
17
18 int *foo()
19 {
20     int x = 211;
21     int *y = &x;
22     return y;
23 }
24
25 void bar() {
26     int z[] = { 91, 92, 93, 94 };
27 }
```

Danger! Never return pointer to a stack value.



```
learncli$ gcc pointer-to-stack.c
learncli$ ./a.out
*a: 211
*a: 94
```

YIKES



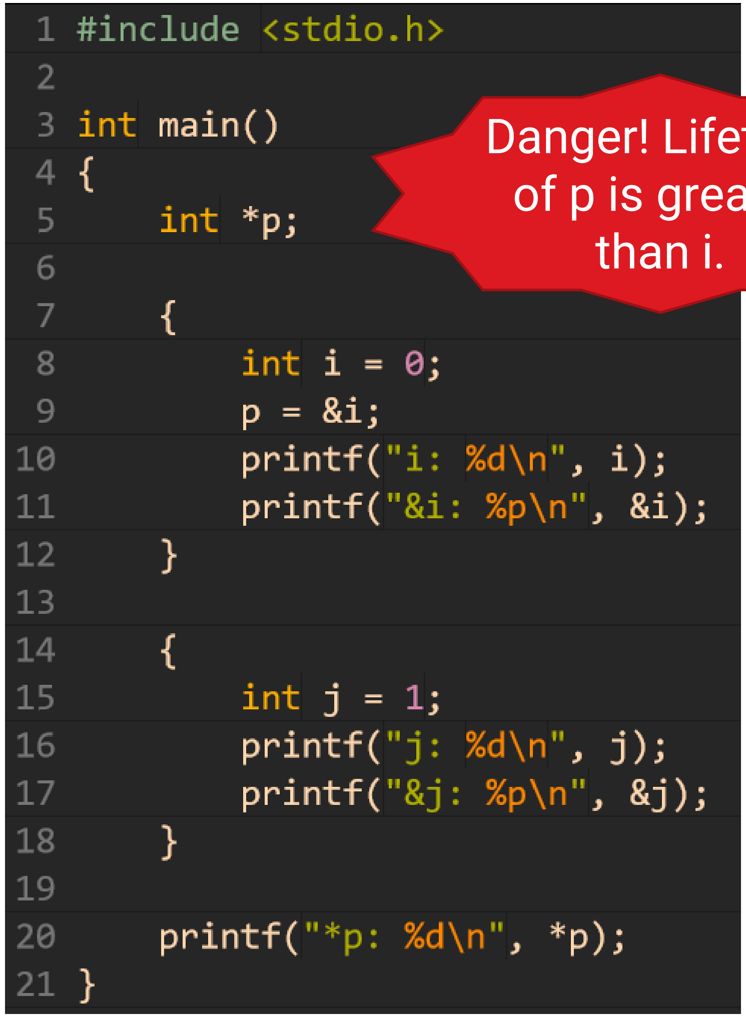
```
learncli$ gcc -Wall pointer-to-auto.c
pointer-to-auto.c: In function ‘bar’:
pointer-to-auto.c:26:9: warning: unused variable ‘z’ [-Wunused-variable]
     int z[] = { 91, 92, 93, 94 };
         ^
```

This is *quite* scary!

Even *with warnings* the warning generated isn't about the fundamental issue here and *ultimately it runs*. But this is fully insane and pathological.

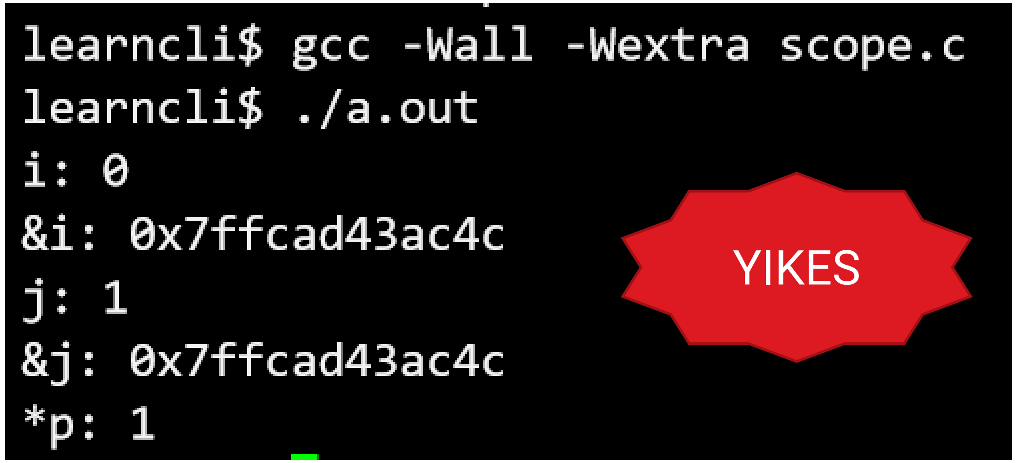
# 2. Consider the Following Code...

1. Does it compile?
2. Does it run?
3. Output of line 20?



```c
 1 #include <stdio.h>
 2 
 3 int main()
 4 {
 5     int *p;
 6 
 7     {
 8         int i = 0;
 9         p = &i;
10         printf("i: %d\n", i);
11         printf("&i: %p\n", &i);
12     }
13 
14     {
15         int j = 1;
16         printf("j: %d\n", j);
17         printf("&j: %p\n", &j);
18     }
19 
20     printf("*p: %d\n", *p);
21 }
```

Danger! Lifetime of p is greater than i.



```
learncli$ gcc -Wall -Wextra scope.c
learncli$ ./a.out
i: 0
&i: 0x7ffcad43ac4c
j: 1
&j: 0x7ffcad43ac4c
*p: 1
```

YIKES

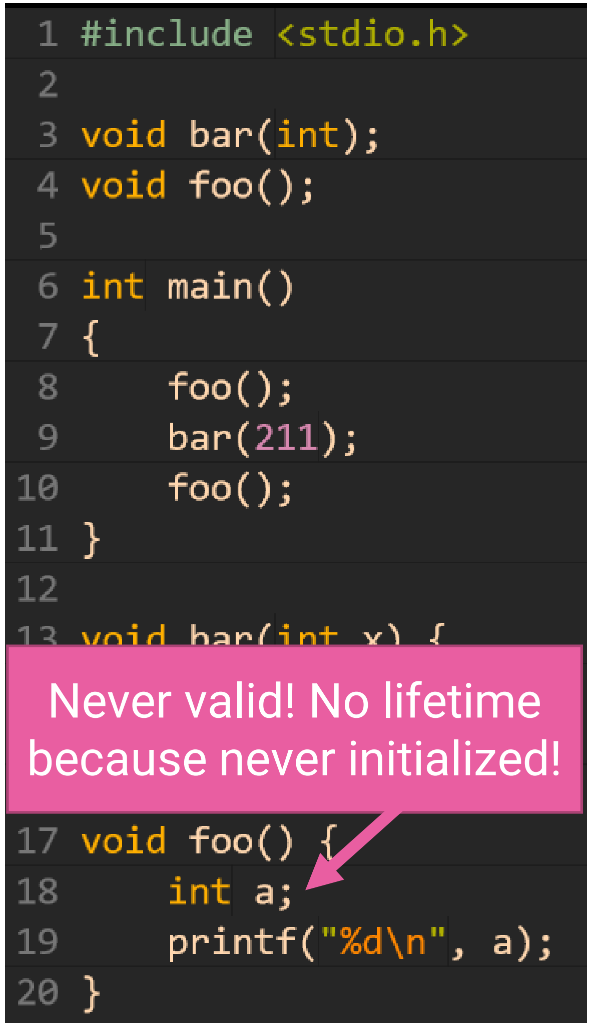
This is *quite* scary!

No warnings emitted! What's the fundamental issue here? We're assigning the address of ***i*** to the pointer ***p***, whose *lifetime* exceeds ***i***'s.

After a memory address' lifetime expires, the system is free (and wise!) to reuse that memory for other purposes, as you see happening here.

# Discern the *lifetimes* of each variable's memory.

1. What is the lifetime of ***a,*** the variable declared on on line 18?

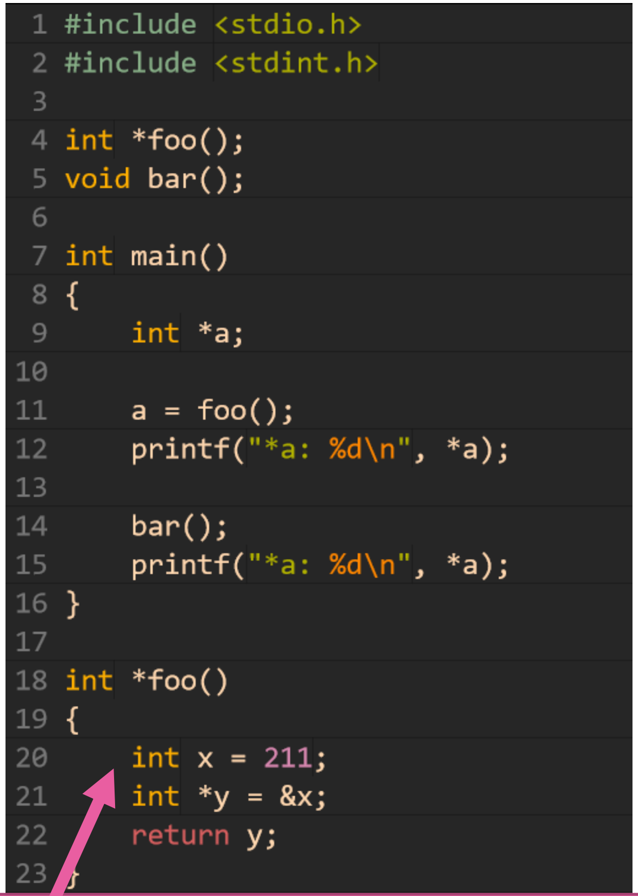


```
 1 #include <stdio.h>
 2
 3 void bar(int);
 4 void foo();
 5
 6 int main()
 7 {
 8     foo();
 9     bar(211);
10     foo();
11 }
12
13 void bar(int x) {
```

```
17 void foo() {
18     int a;
19     printf("%d\n", a);
20 }
```

Never valid! No lifetime because never initialized!

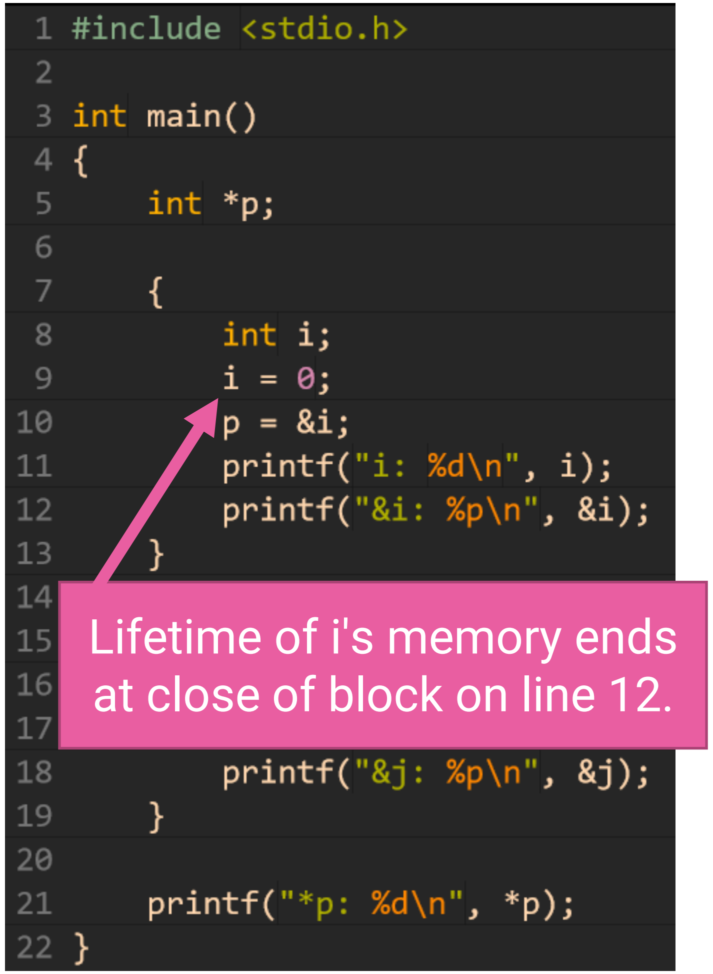
2. What is the lifetime of ***x,*** the variable declared on line 20?



```
 1 #include <stdio.h>
 2 #include <stdint.h>
 3
 4 int *foo();
 5 void bar();
 6
 7 int main()
 8 {
 9     int *a;
10
11     a = foo();
12     printf("*a: %d\n", *a);
13
14     bar();
15     printf("*a: %d\n", *a);
16 }
17
18 int *foo()
19 {
20     int x = 211;
21     int *y = &x;
22     return y;
23 }
```

Lifetime of x begins on line 20 and expires upon return at line 22.

3. What is the lifetime of ***i,*** the variable declared on line 8?



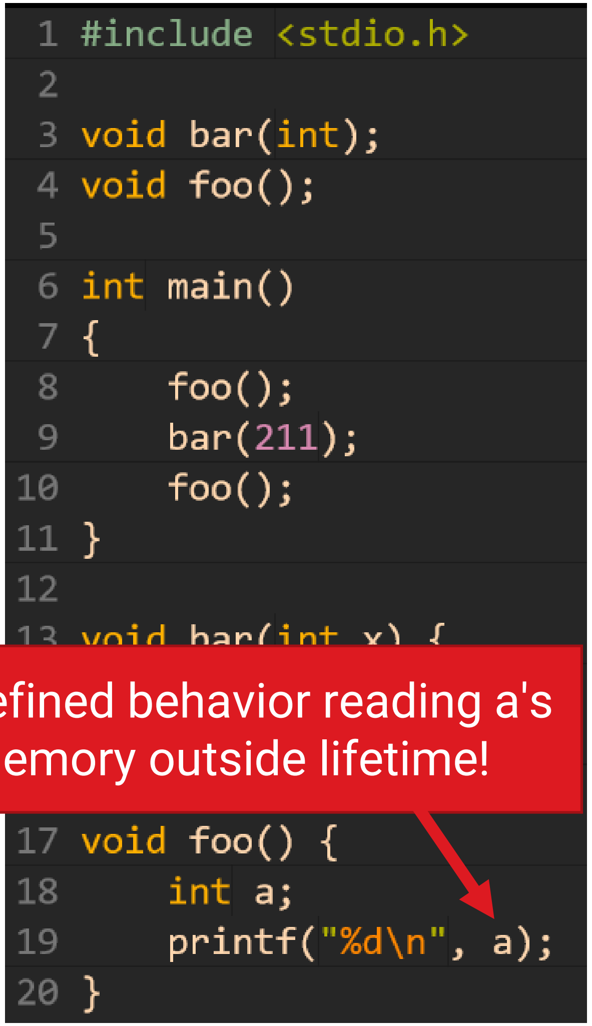
```
 1 #include <stdio.h>
 2
 3 int main()
 4 {
 5     int *p;
 6
 7     {
 8         int i;
 9         i = 0;
10         p = &i;
11         printf("i: %d\n", i);
12         printf("&i: %p\n", &i);
13     }
14
15
```

```
18         printf("&j: %p\n", &j);
19     }
20
21     printf("*p: %d\n", *p);
22 }
```

Lifetime of i's memory ends at close of block on line 12.

# Never access memory outside its *lifetime*!

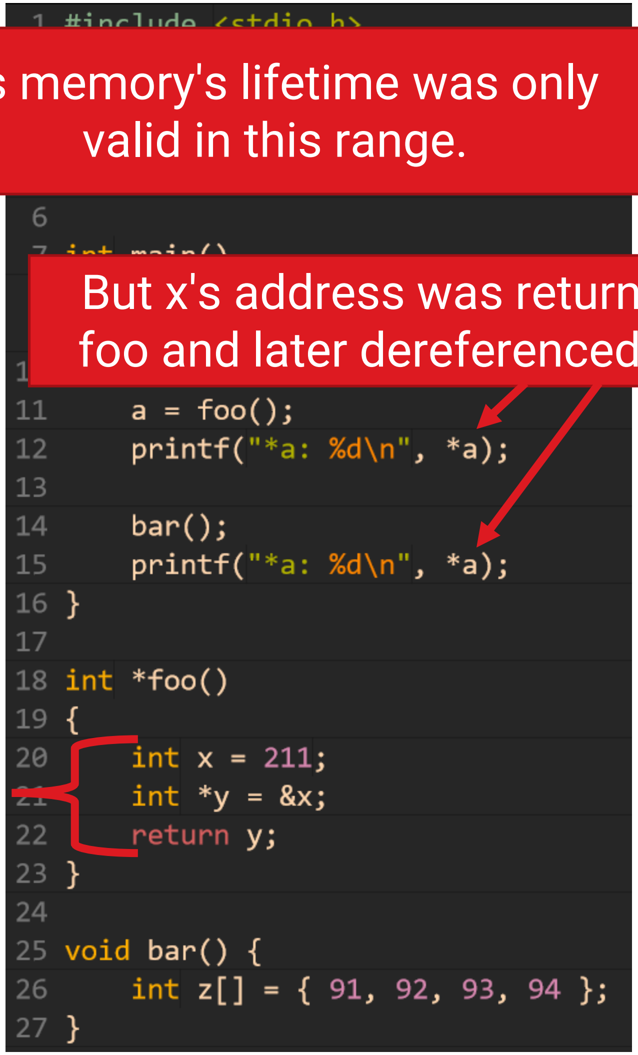
1. What is the lifetime of ***a,*** the variable declared on on line 18?



```
1  #include <stdio.h>
2
3  void bar(int);
4  void foo();
5
6  int main()
7  {
8      foo();
9      bar(211);
10     foo();
11 }
12
13 void bar(int x) {
17 void foo() {
18     int a;
19     printf("%d\n", a);
20 }
```

Undefined behavior reading a's memory outside lifetime!

2. What is the lifetime of ***x,*** the variable declared on line 20?

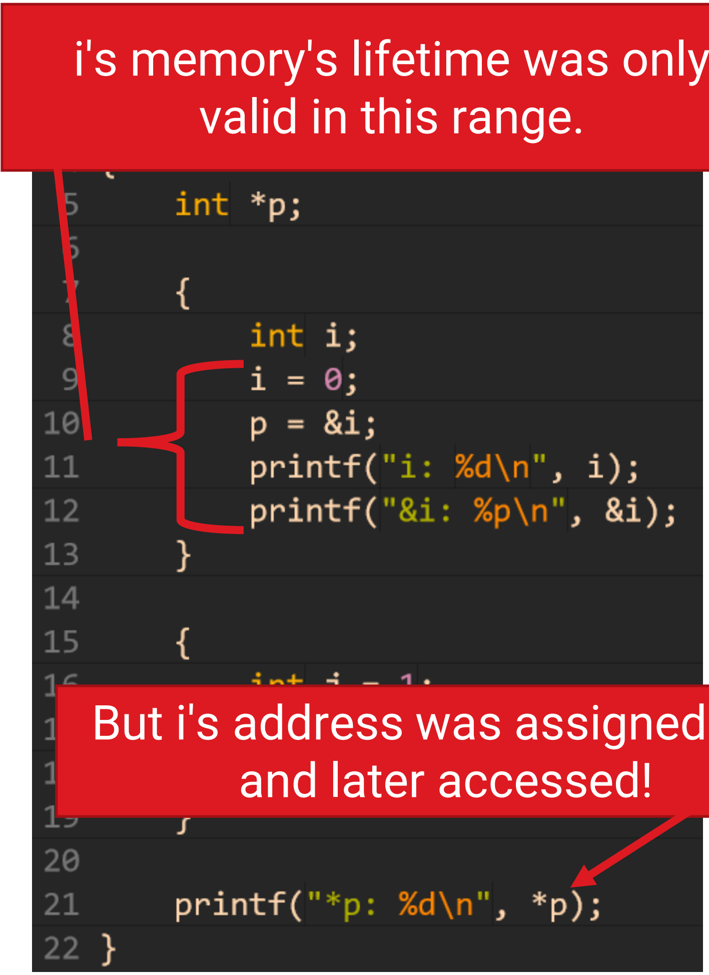


```
1  #include <stdio.h>
6
7  int main()
11     a = foo();
12     printf("*a: %d\n", *a);
13
14     bar();
15     printf("*a: %d\n", *a);
16 }
17
18 int *foo()
19 {
20     int x = 211;
21     int *y = &x;
22     return y;
23 }
24
25 void bar() {
26     int z[] = { 91, 92, 93, 94 };
27 }
```

x's memory's lifetime was only valid in this range.

But x's address was returned by foo and later dereferenced here!

3. What is the lifetime of ***i,*** the variable declared on line 8?



```
4  {
5      int *p;
6
7      {
8          int i;
9          i = 0;
10         p = &i;
11         printf("i: %d\n", i);
12         printf("&i: %p\n", &i);
13     }
14
15     {
16         int i = 1;
19     }
20
21     printf("*p: %d\n", *p);
22 }
```

i's memory's lifetime was only valid in this range.

But i's address was assigned to p and later accessed!