Pointer I

CS143: lecture 4

Byron Zhong, June 20

Variables

Review

- A variable has a type (thereby size) and a location in memory.
- Declaration and initialization
- Arrays and strings
- But, how does a compiler find space for variables?
- But first, how long does a variable live?

Variable Lifetime

An Example

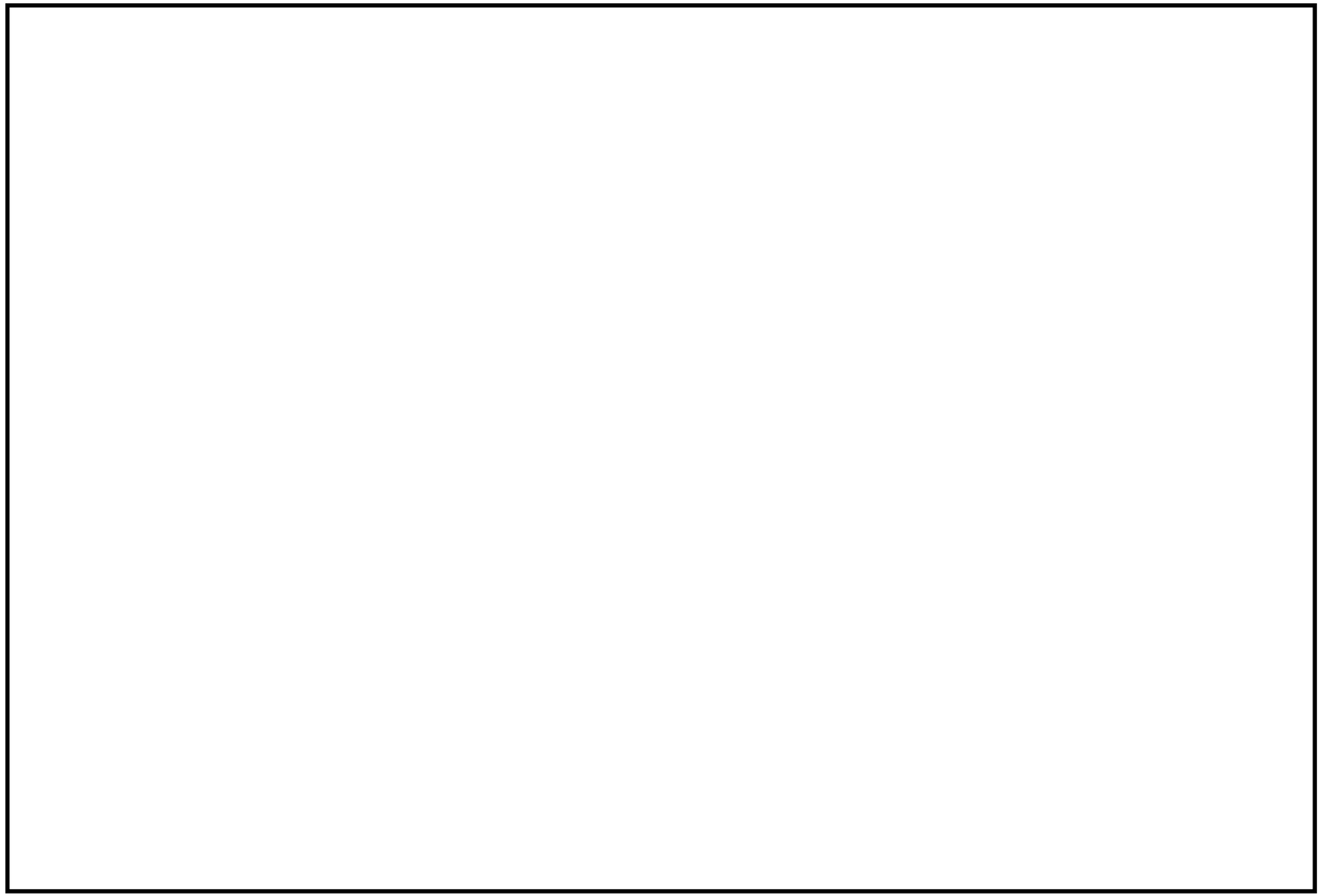
```
01 int f(int x)
02 {
03         int y = x * 2;
04         return y;
05 }
06
07 int main(void)
08 {
09         int a = f(10);
10         int b = f(a);
11         printf("%d\n", b);
12
13         return 0;
14 }
```

- When is y born?
- When is y dead?
- a?
- b?
- x?

Variable Lifetime

An Example

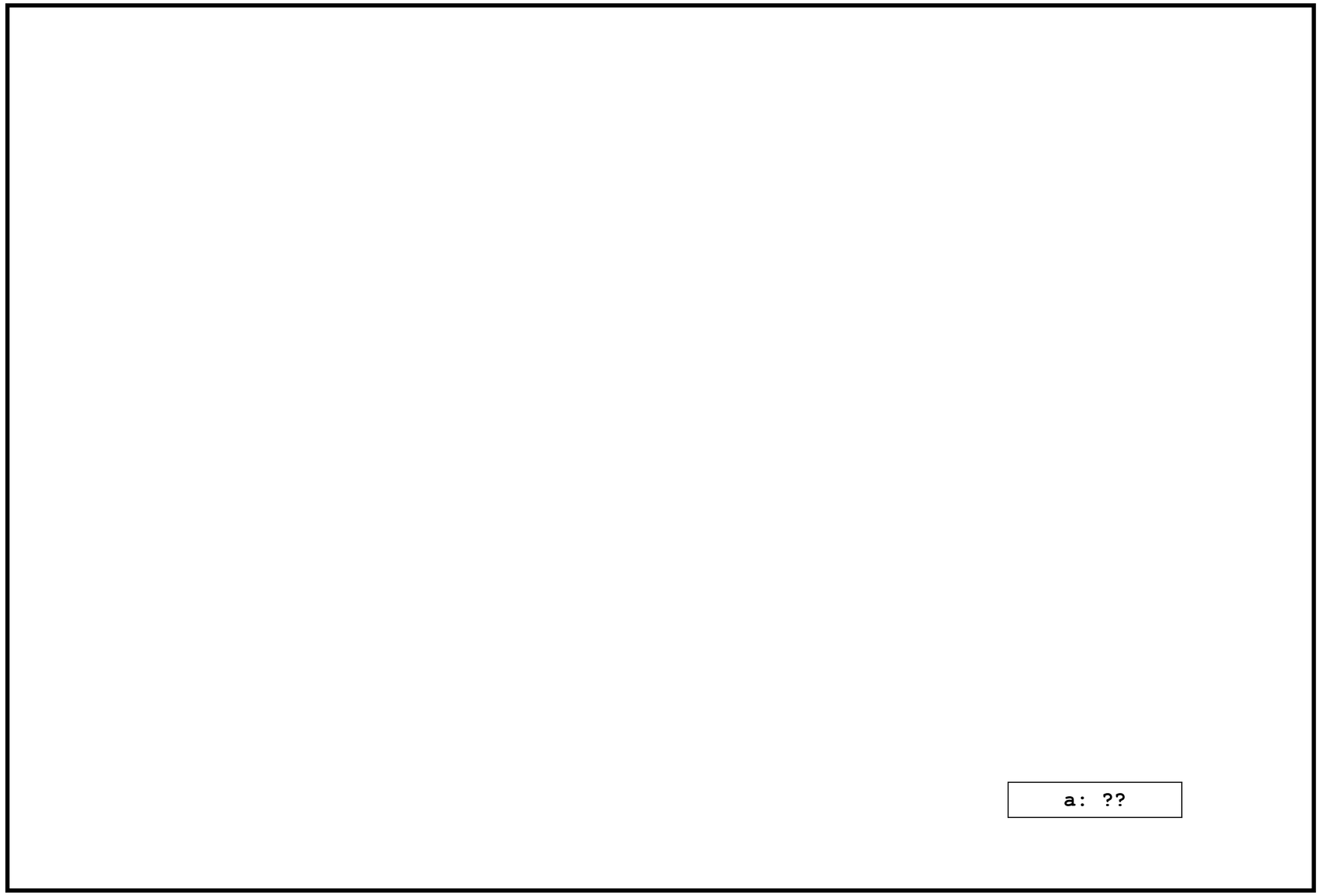
```
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```



Variable Lifetime

An Example

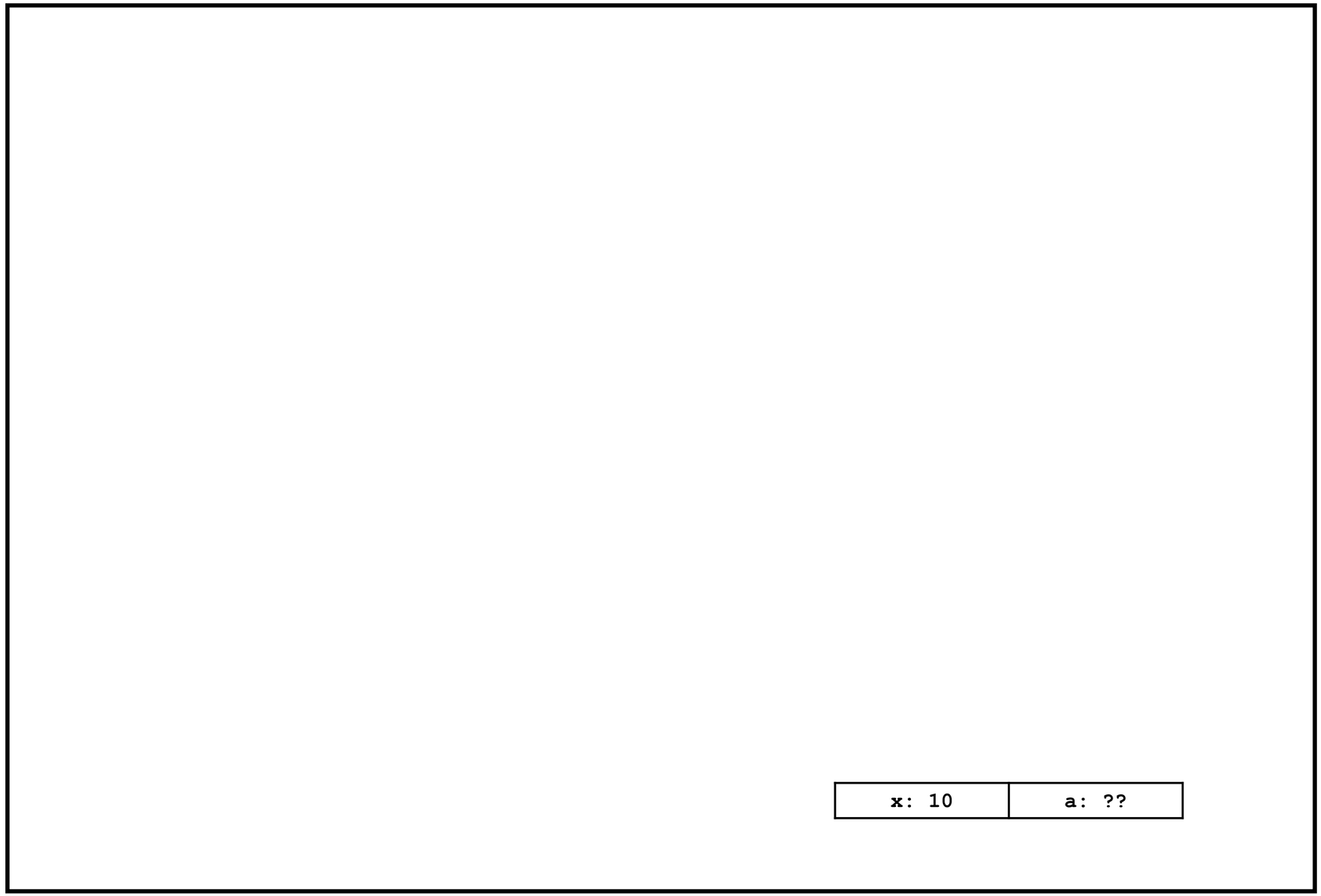
```
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```



Variable Lifetime

An Example

```
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```



Variable Lifetime

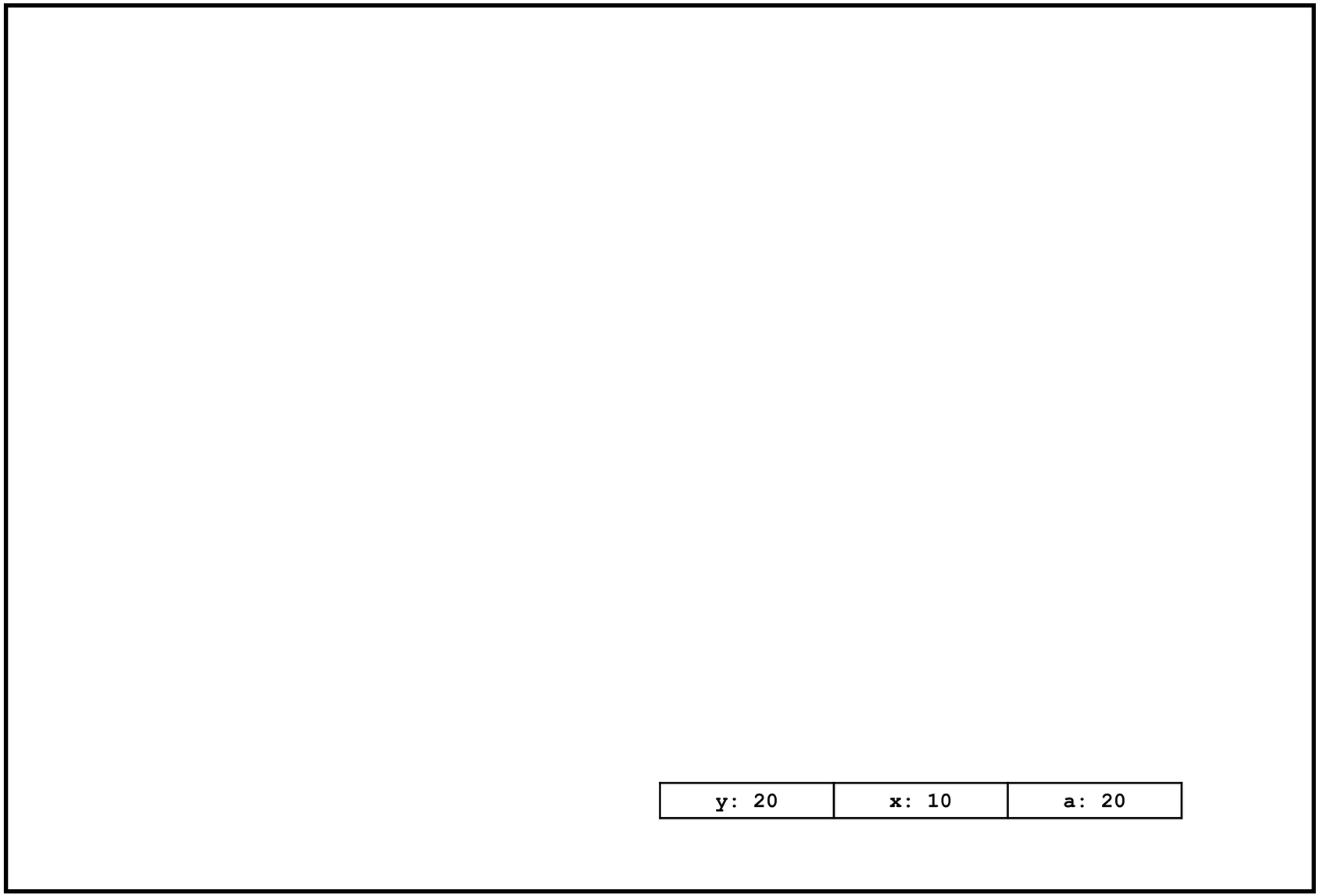
An Example

```
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```

Variable Lifetime

An Example

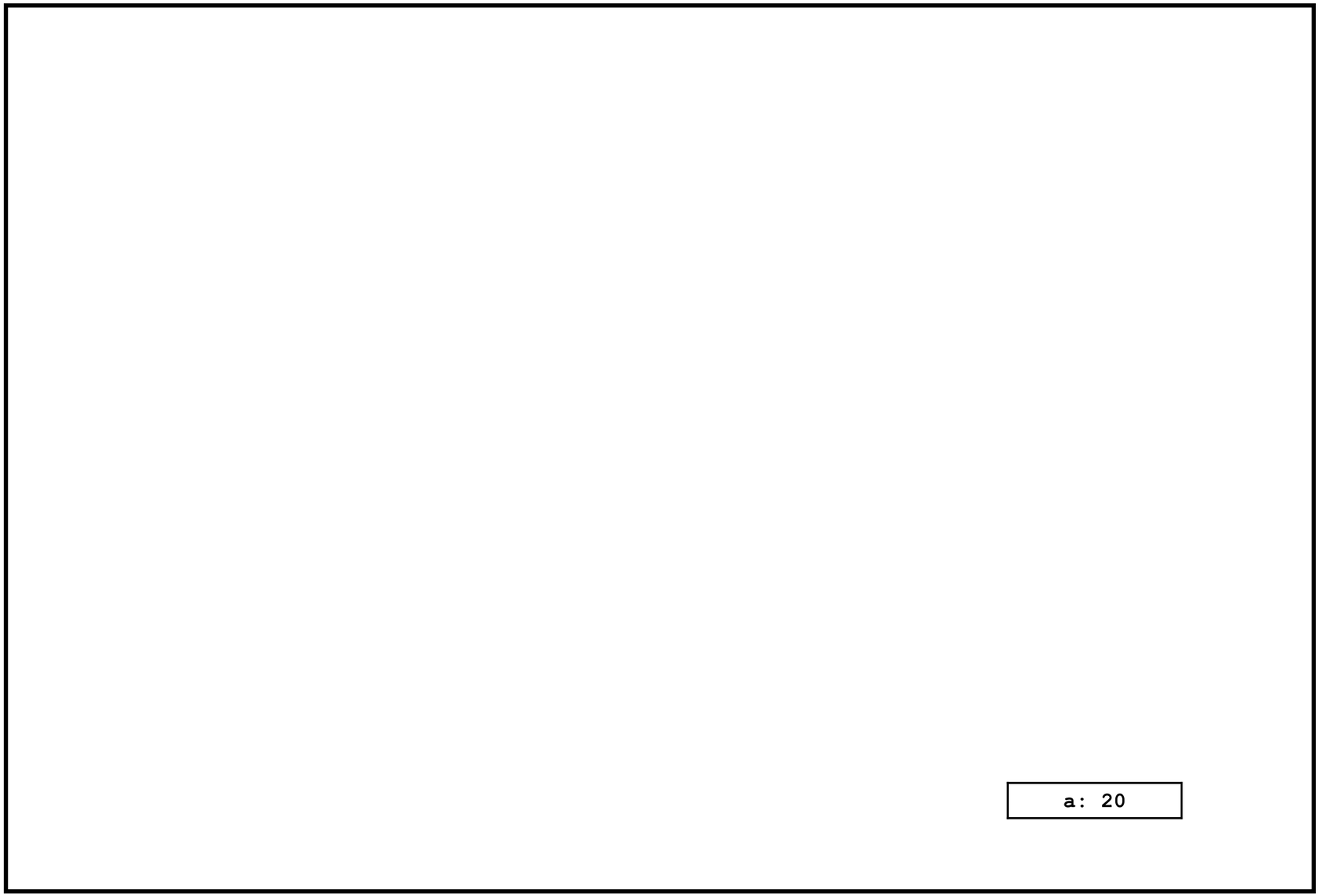
```
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```



Variable Lifetime

An Example

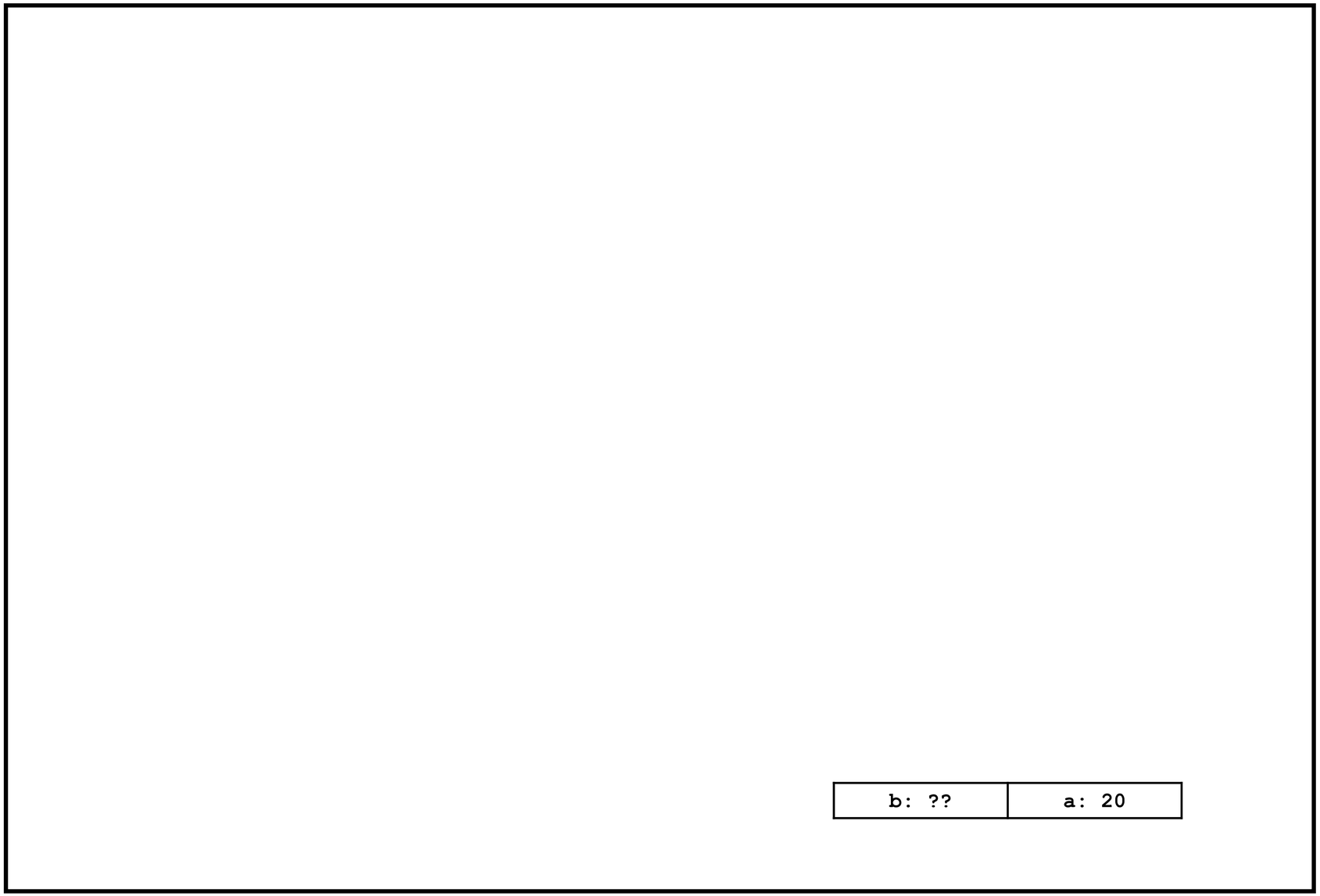
```
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```



Variable Lifetime

An Example

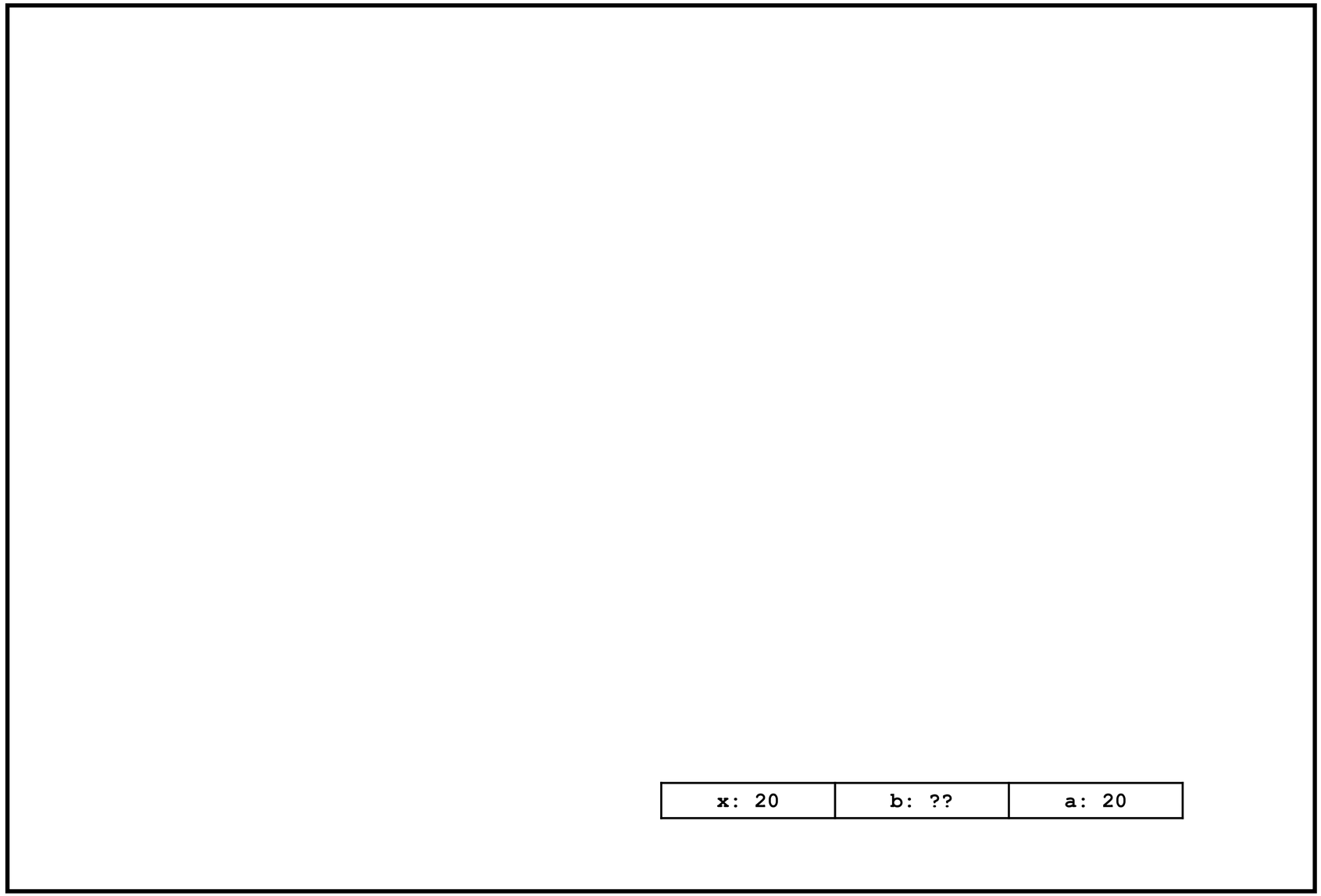
```
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```



Variable Lifetime

An Example

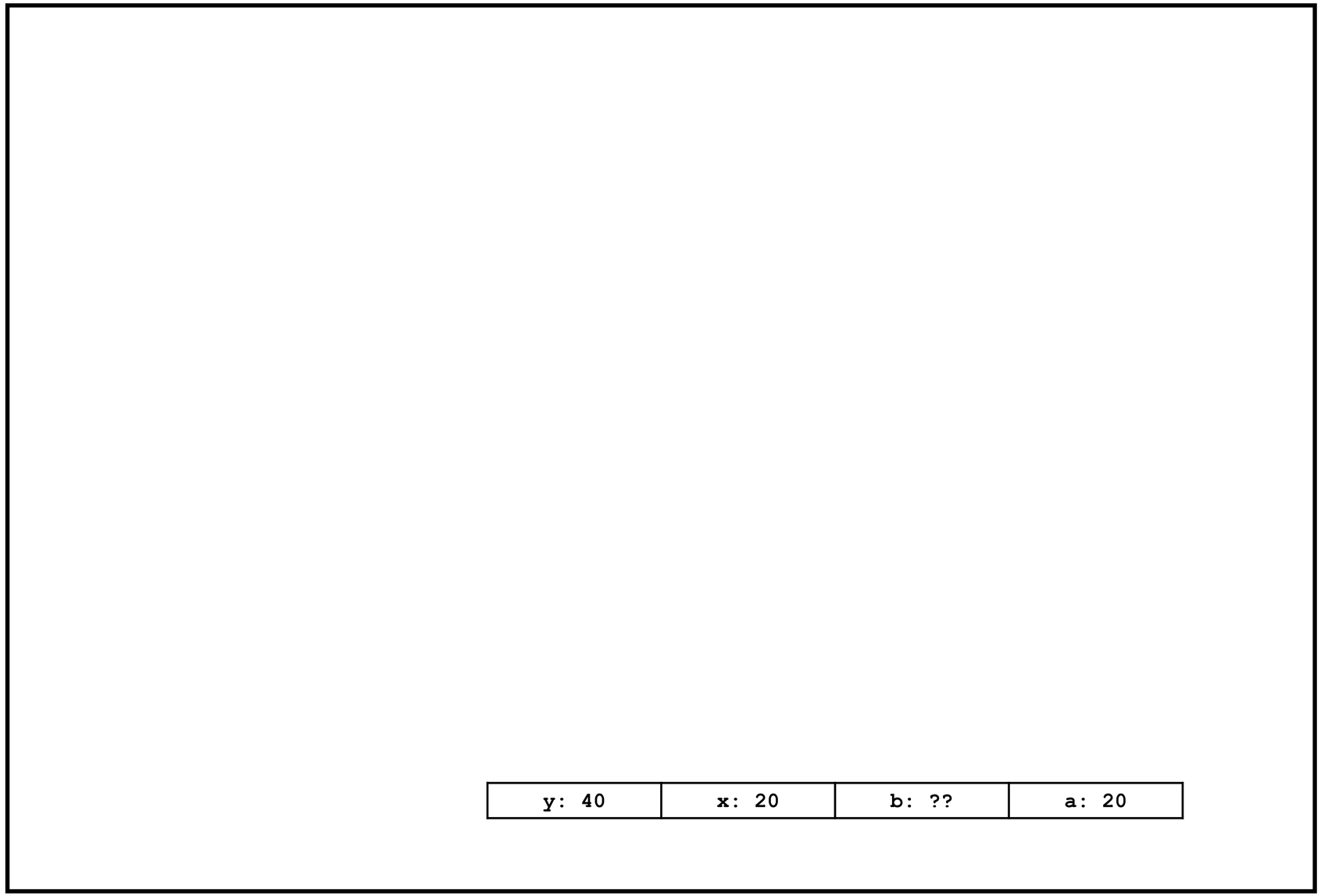
```
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```



Variable Lifetime

An Example

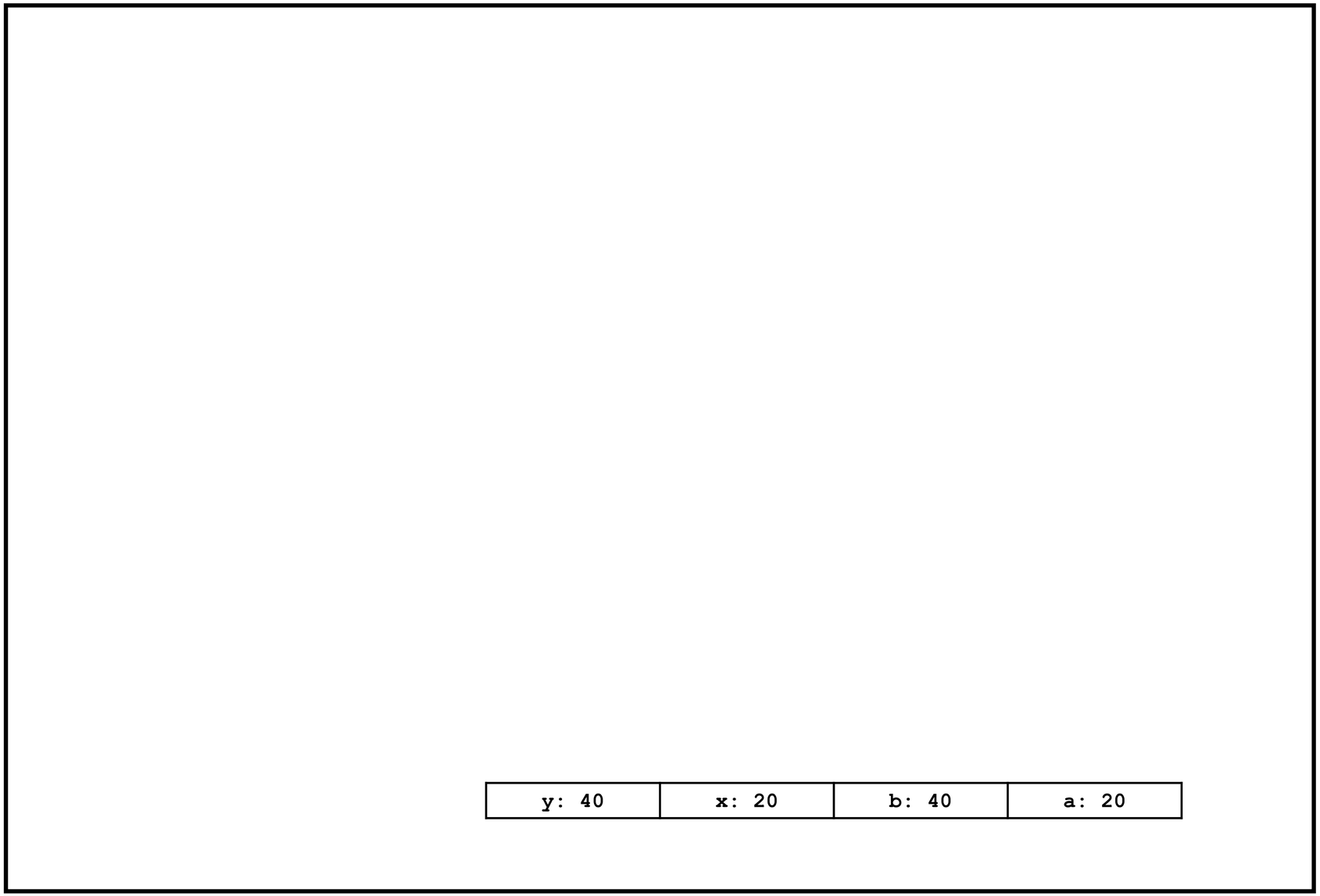
```
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```



Variable Lifetime

An Example

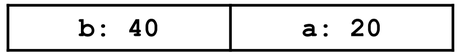
```
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```



Variable Lifetime

An Example

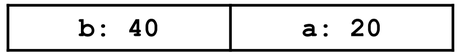
```
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```



Variable Lifetime

An Example

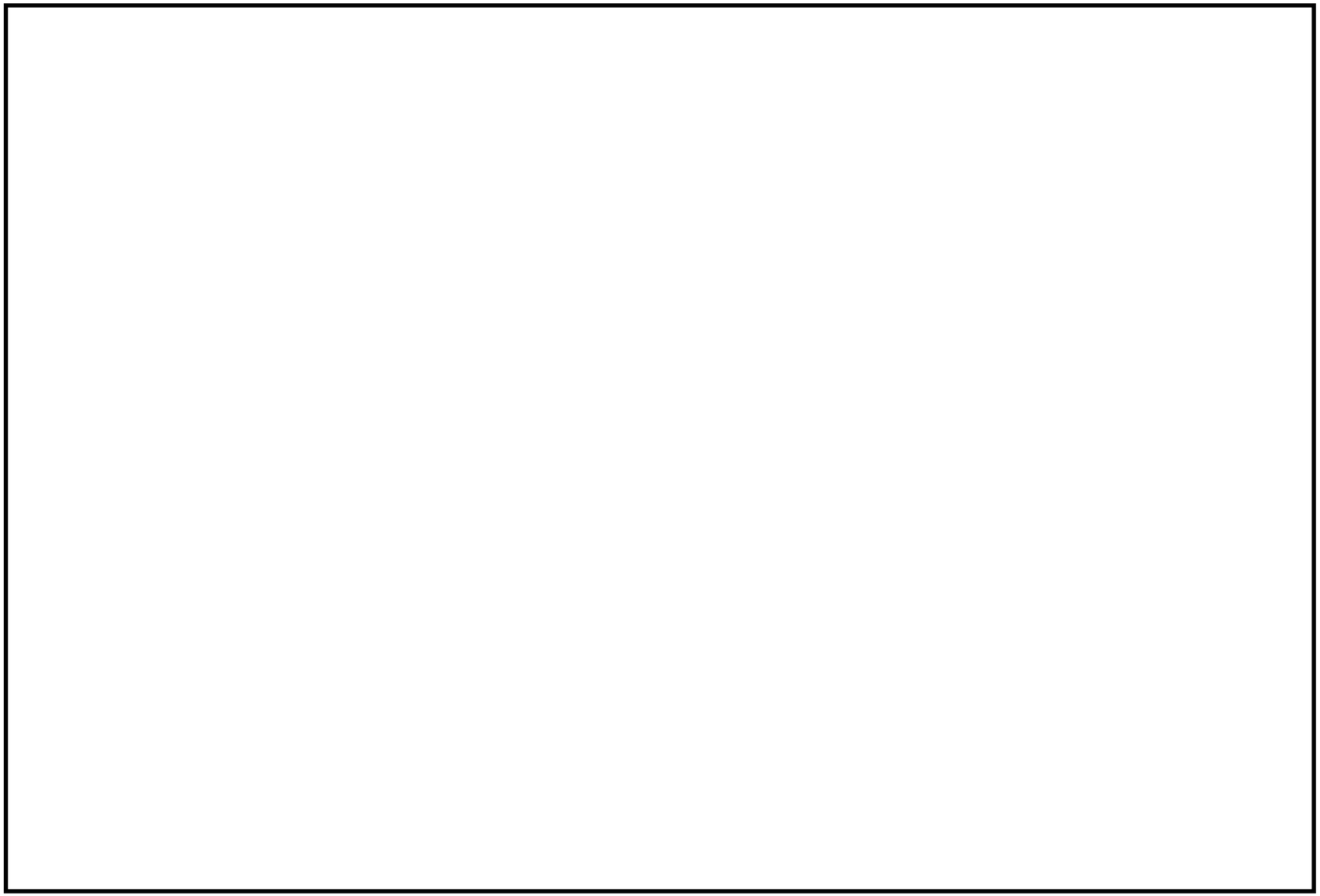
```
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```



Variable Lifetime

An Example

```
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```



Variable Lifetime

An Example

- When a function returns, we can recycle the memory used by the variables declared inside the function.
  - Variables declared in `{ .. }` can only be accessed in `{ .. }` (Scope)

Variable Lifetime

An Example

```c
#include <stdio.h>
int main(void)
{
        int len = 50;
        for (int i = 0; i < len; i++) {
                char c = 'a';
        }

        putchar(c);
        return 0;
}
```



```
ex.c:9:17: error: use of undeclared identifier 'c'
        putchar(c);
                ^
1 error generated.
```

Variable Lifetime

- When a function returns, we can recycle the memory used by the variables declared inside the function.
  - Variables declared in `{ .. }` can only be accessed in `{ .. }` (Scope)
- A variable is "born" when we declare it, and is "dead" when we leave its scope.
- In reality, all variables in a function are *allocated* at once when we call the function, and they are all *deallocated* at once when the function returns.
- We call this structure in memory composed of local variables and arguments a *frame*.

Variable Lifetime

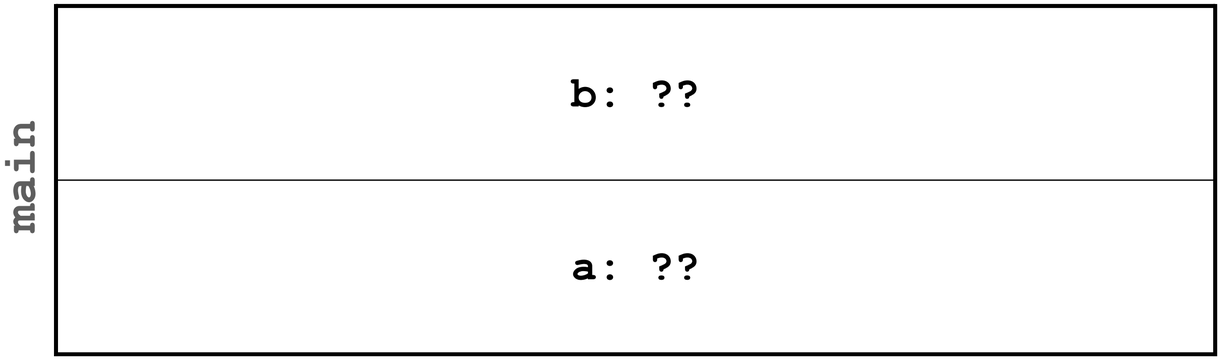
A more accurate picture

```c
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```

Variable Lifetime

A more accurate picture

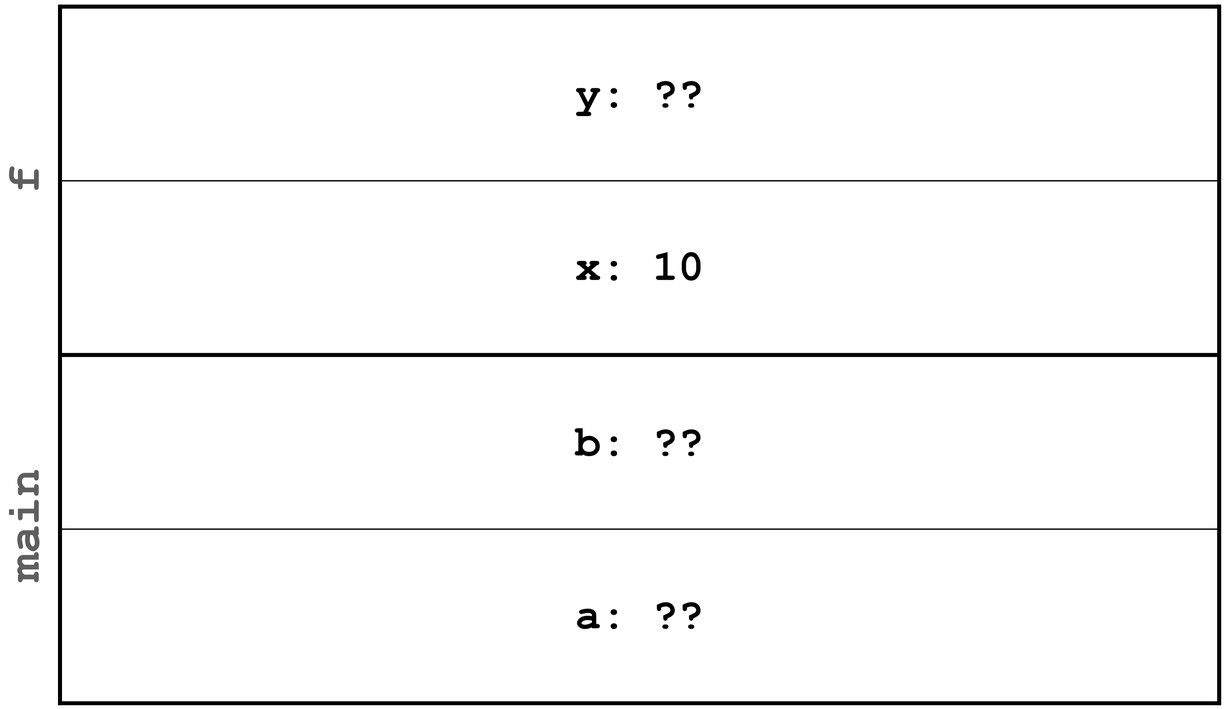
```c
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```



Variable Lifetime

A more accurate picture

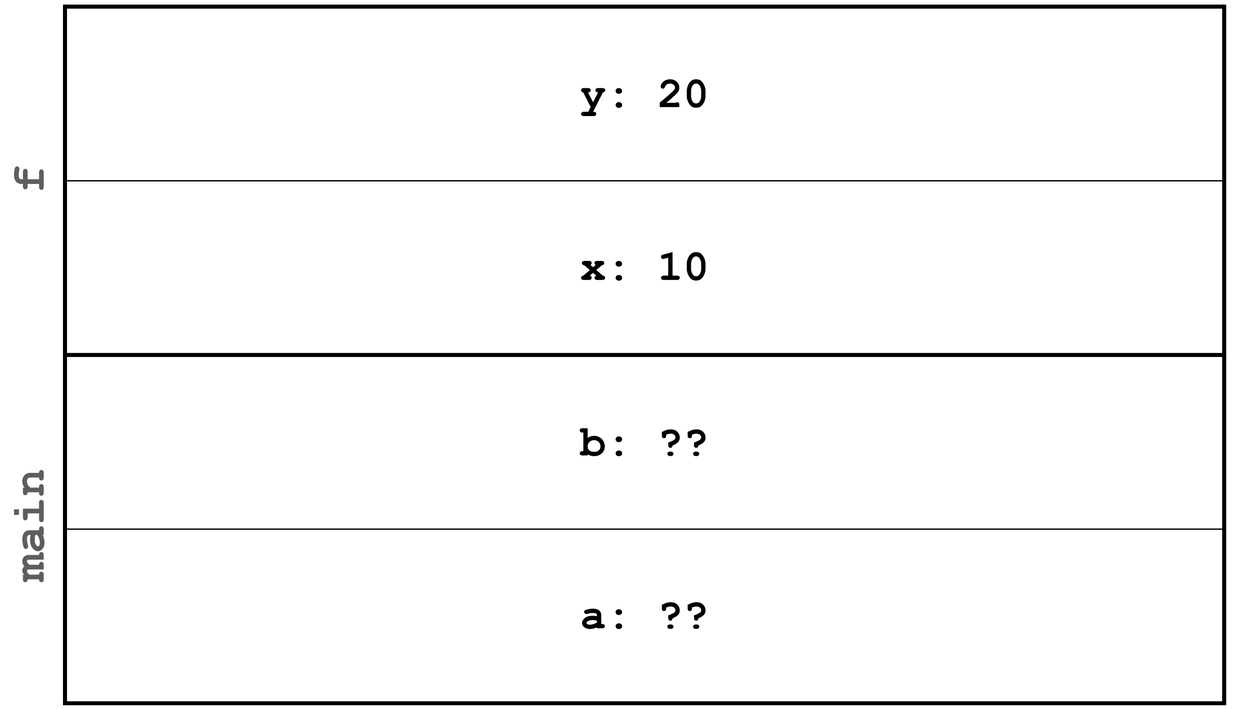
```
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```



Variable Lifetime

A more accurate picture

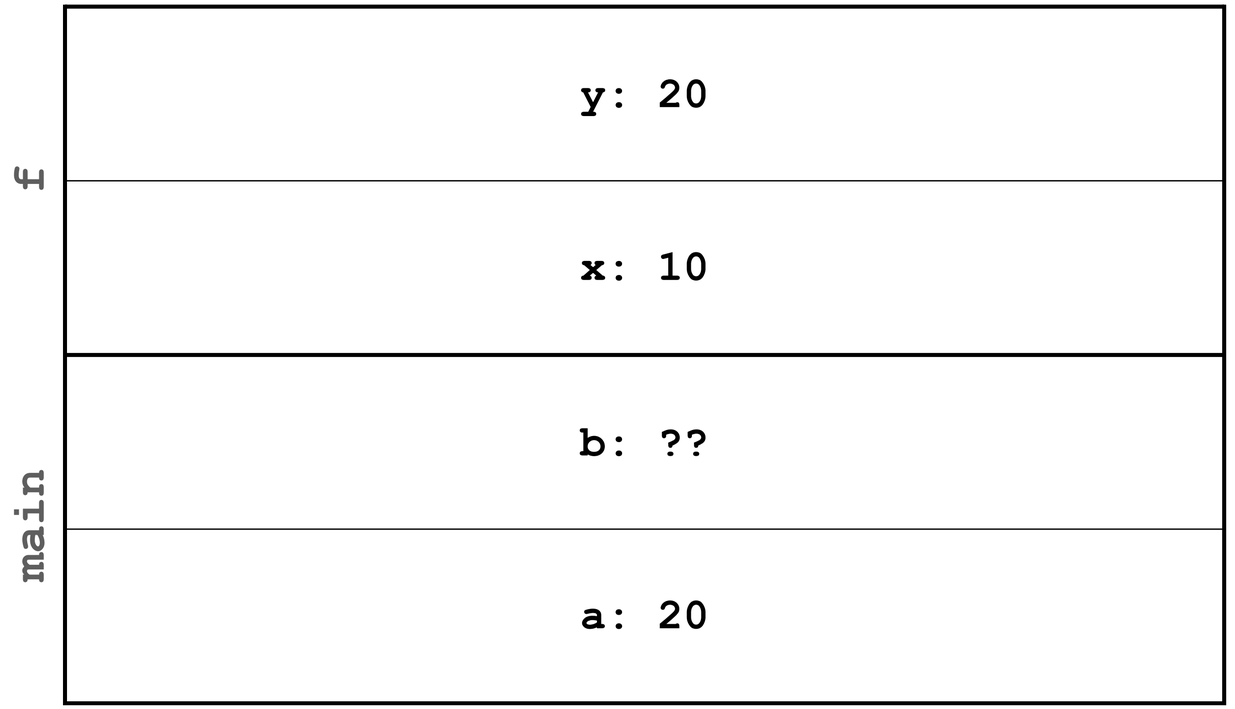
```
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```



Variable Lifetime

A more accurate picture

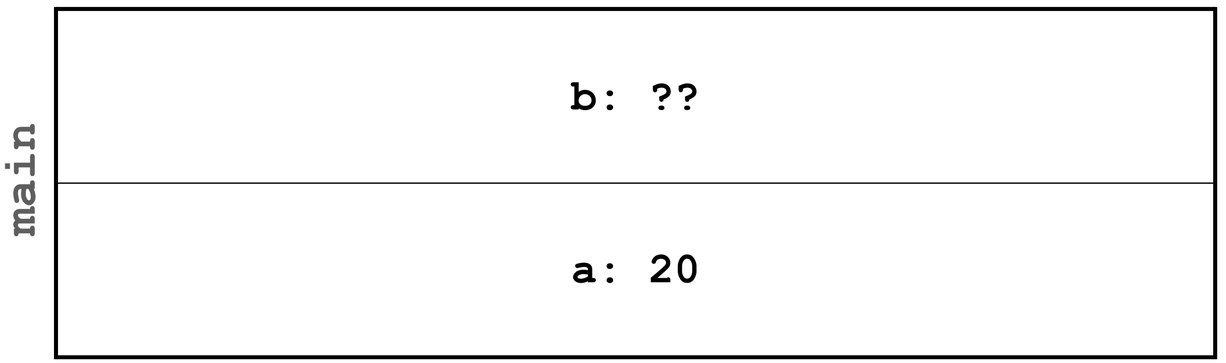
```
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```



Variable Lifetime

A more accurate picture

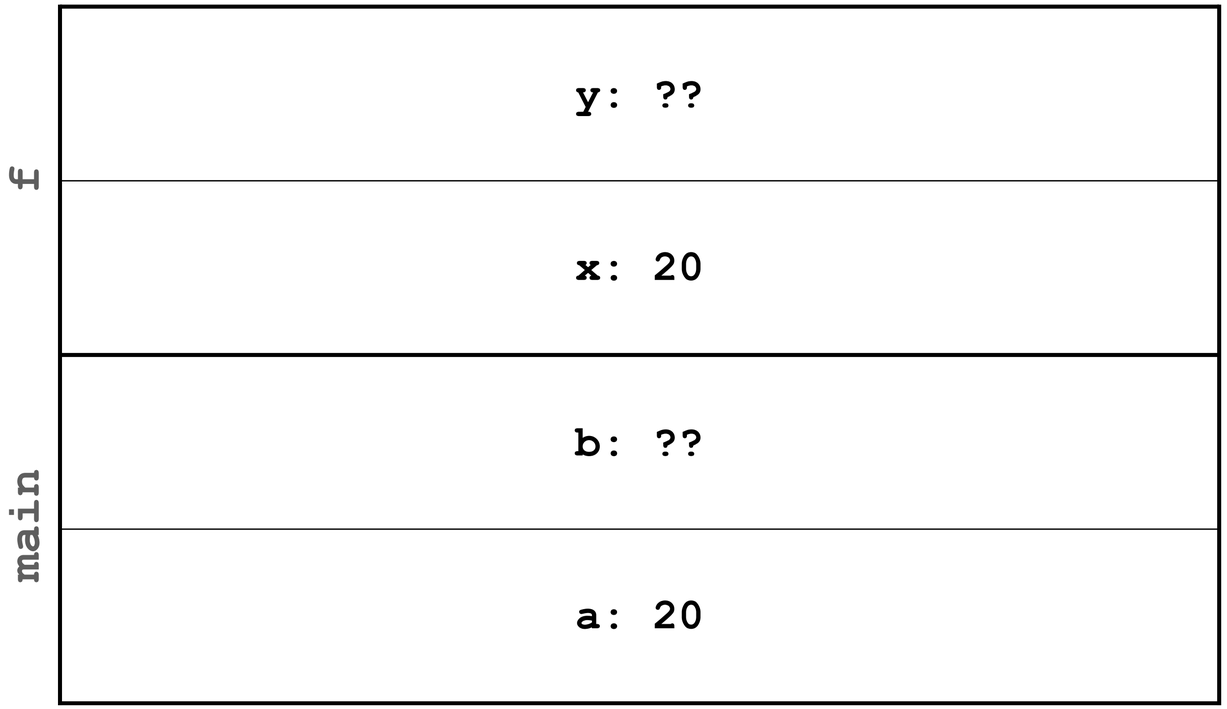
```c
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```



Variable Lifetime

A more accurate picture

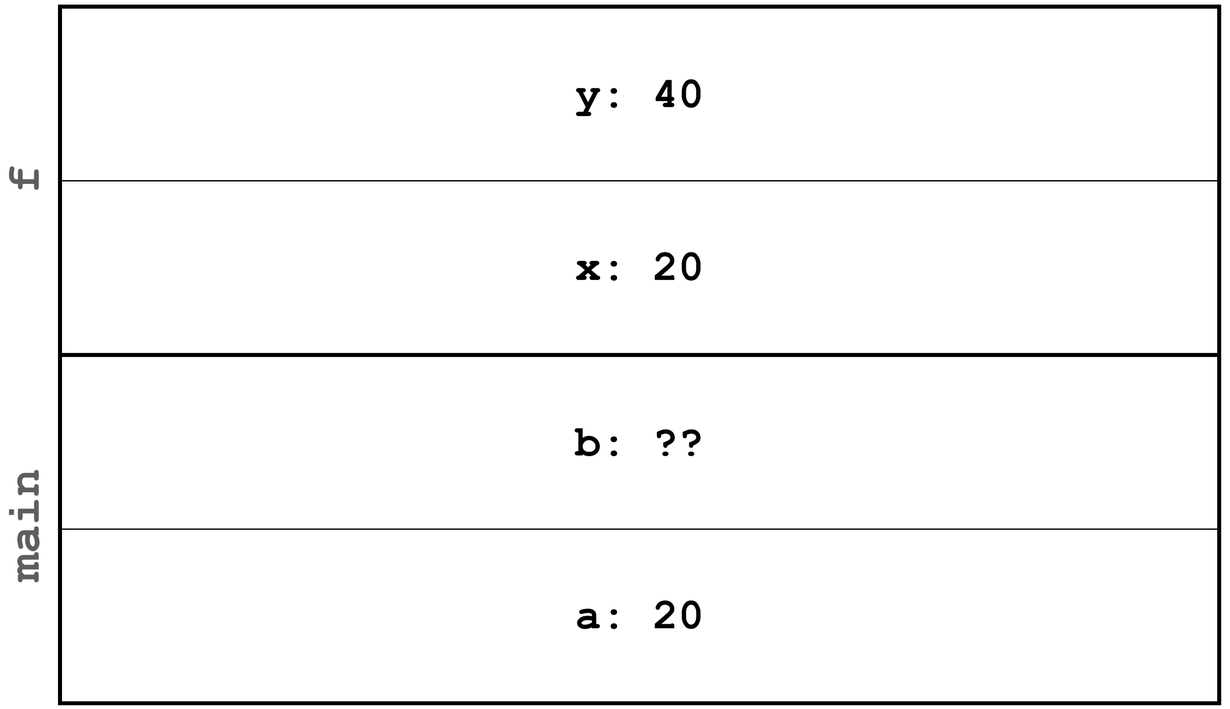
```
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```



Variable Lifetime

A more accurate picture

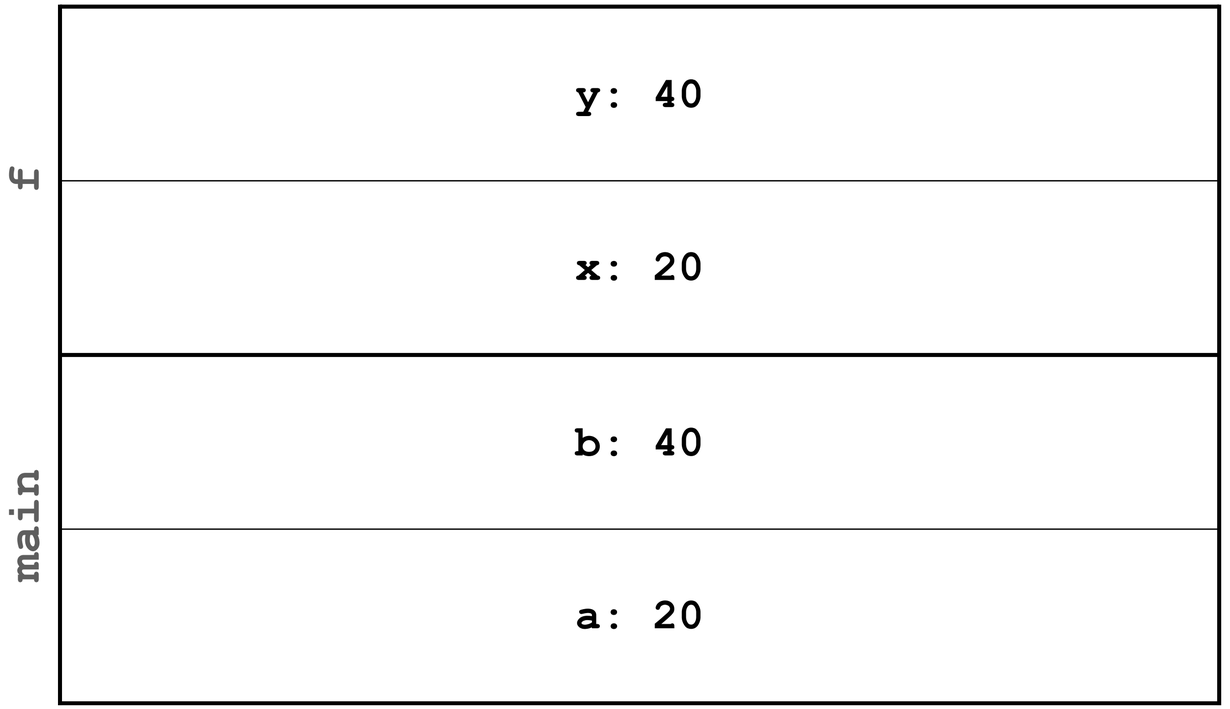
```
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```



Variable Lifetime

A more accurate picture

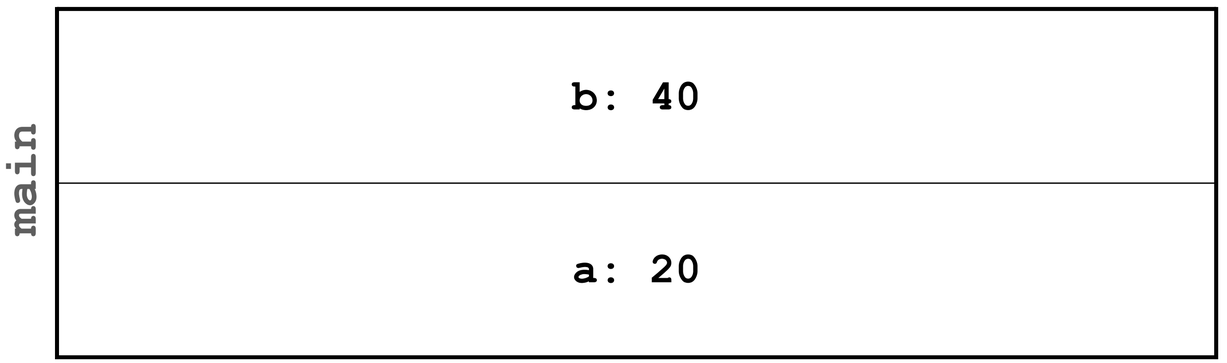
```
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```



Variable Lifetime

A more accurate picture

```c
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```



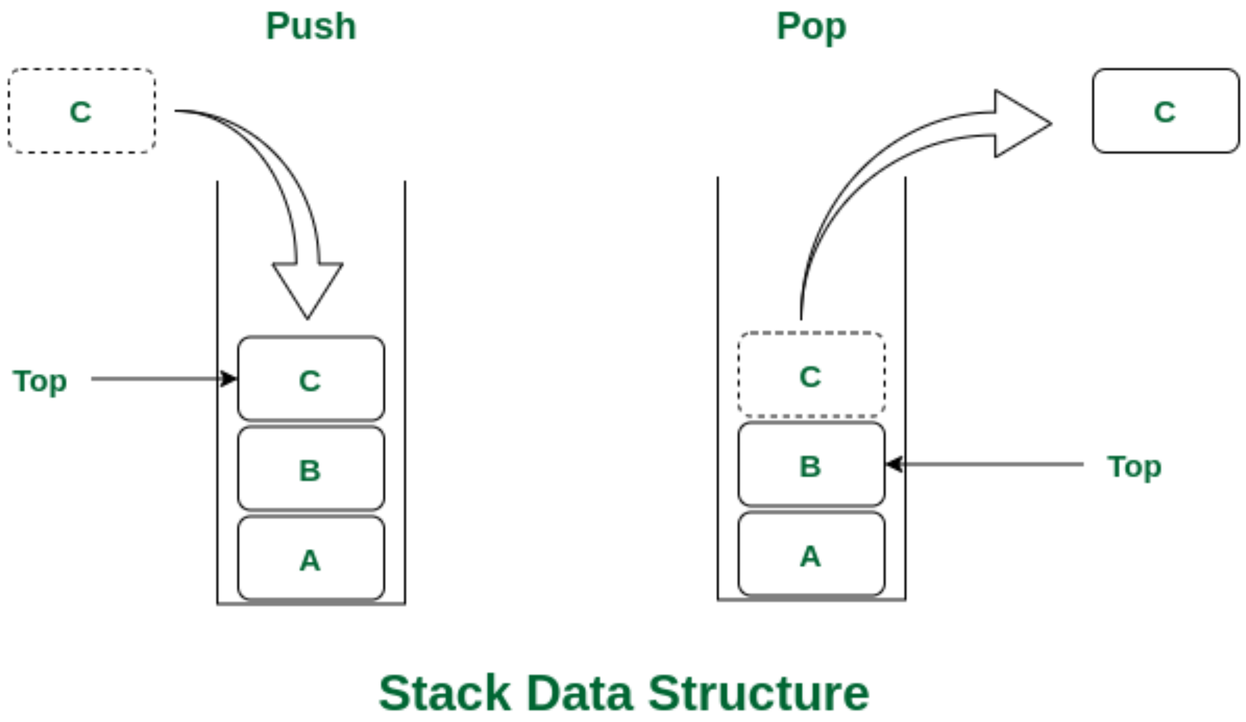
Variable Lifetime

A more accurate picture

```
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```

Variable Lifetime

Call Stack



Stack Data Structure

Source: https://www.geeksforgeeks.org/stack-data-structure/

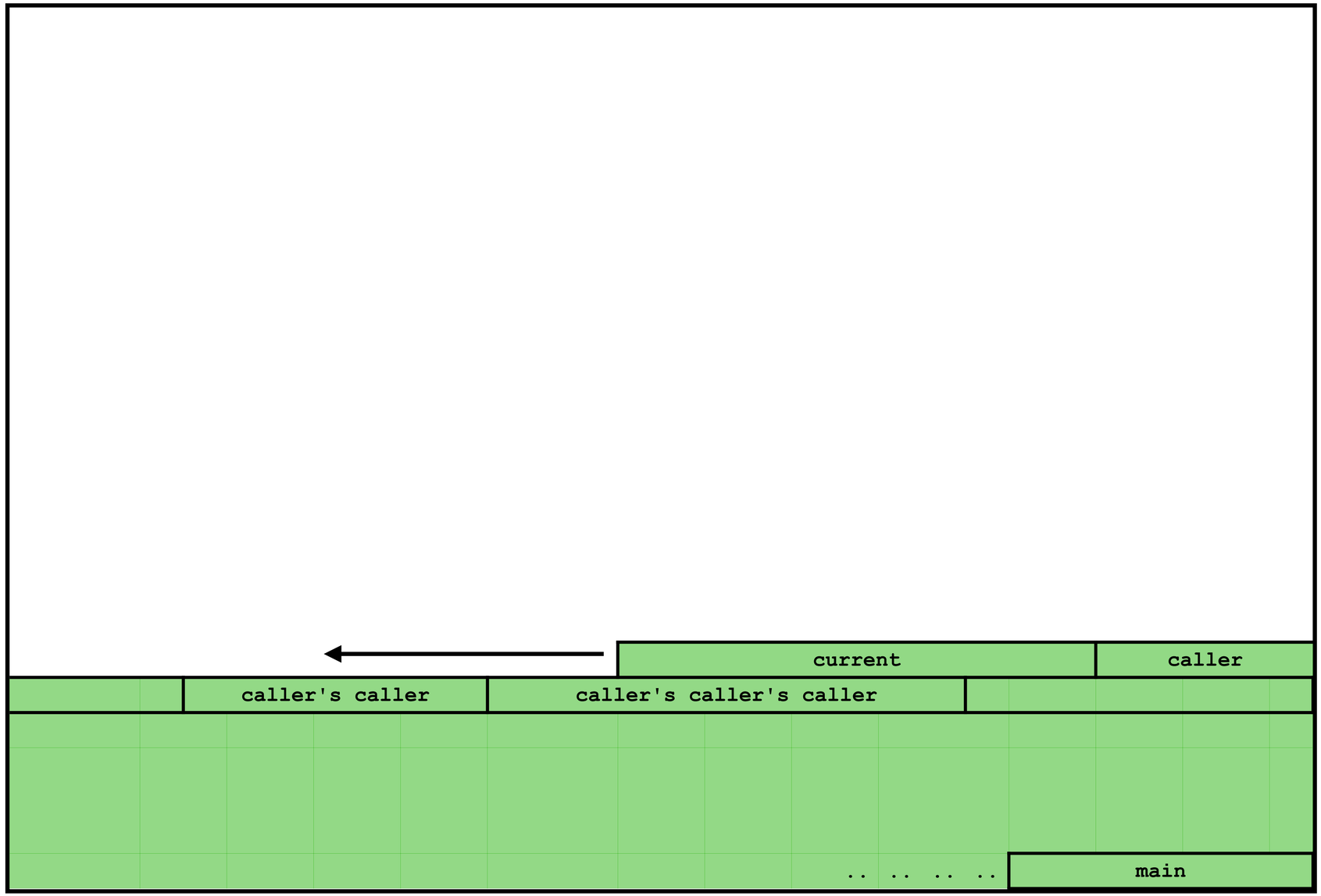
Variable Lifetime

Call Stack

- C manages memory using a *stack* that lives in memory; internally, C remembers where the top of the stack is.
- Local variables (in a frame) have addresses relative to the top of the stack.
- When we call a function, a frame is *pushed* to the stack.
  - A frame usually has all arguments, all local variables, and the return location.
- When we return from a function, a frame is *popped* from the stack.

Variable Lifetime

Call Stack



Variable Lifetime

Call Stack

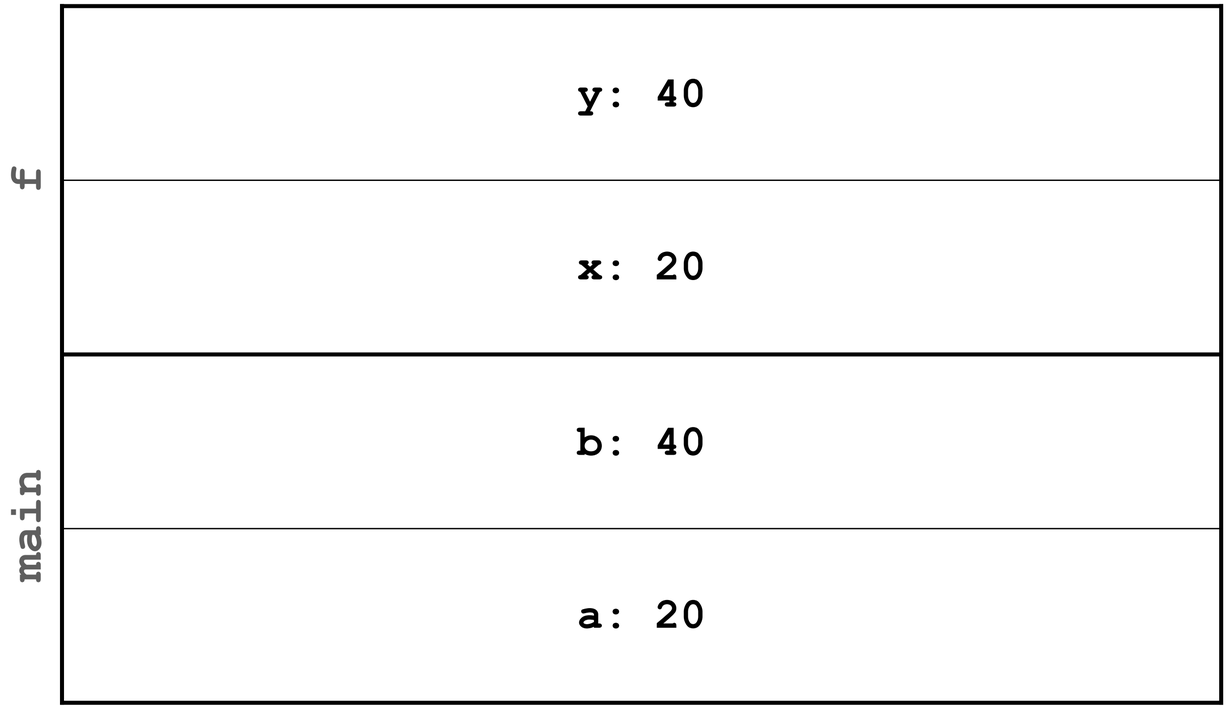
- Stack has a fixed size determined by the OS.
- We can only push a finite number of frames onto the stack.
  - Function calls have limited depth
  - Recursion can't be too deep
- When we run out of stack for frames, we say we "overflow" the stack
- Stack Overflow!

Variable Lifetime

Revisit

```c
int f(int x)
{
        int y = x * 2;
        return y;
}

int main(void)
{
        int a = f(10);
        int b = f(a);
        printf("%d\n", b);

        return 0;
}
```

- Frames are isolated.
- f can never touch main's variable
- Is it a good thing?
- Is there situation where this is not desirable?



Pass by Reference

swap

```c
#include <stdio.h>

void swap(int x, int y)
{
        int tmp = x;
        x = y;
        y = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(x, y);

        printf("%d %d\n", x, y);

        return 0;
}
```

Does it work?

No

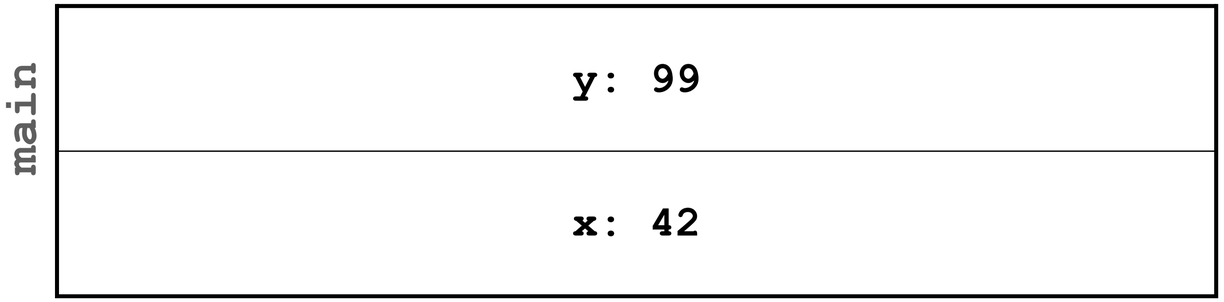
```
byron@MacBook-Pro solutions % ./swap
42 99
```

But why?

Pass by Reference

swap

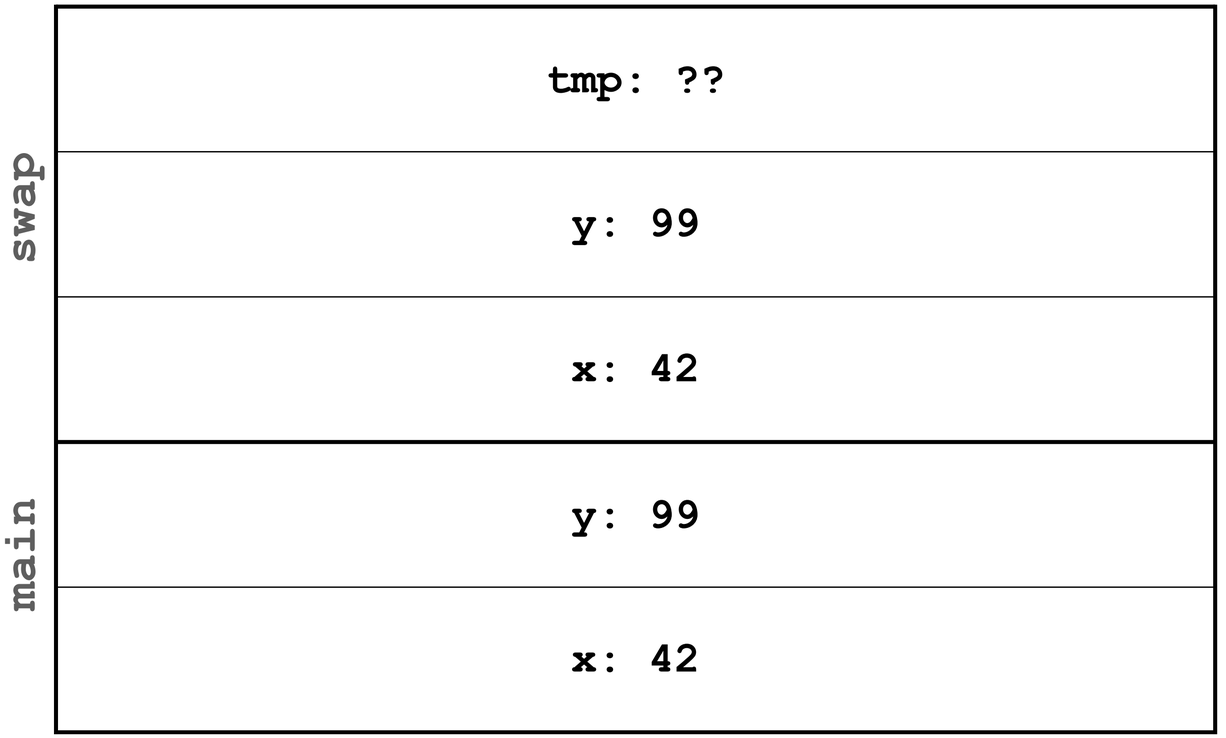
```c
#include <stdio.h>

void swap(int x, int y)
{
        int tmp = x;
        x = y;
        y = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(x, y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

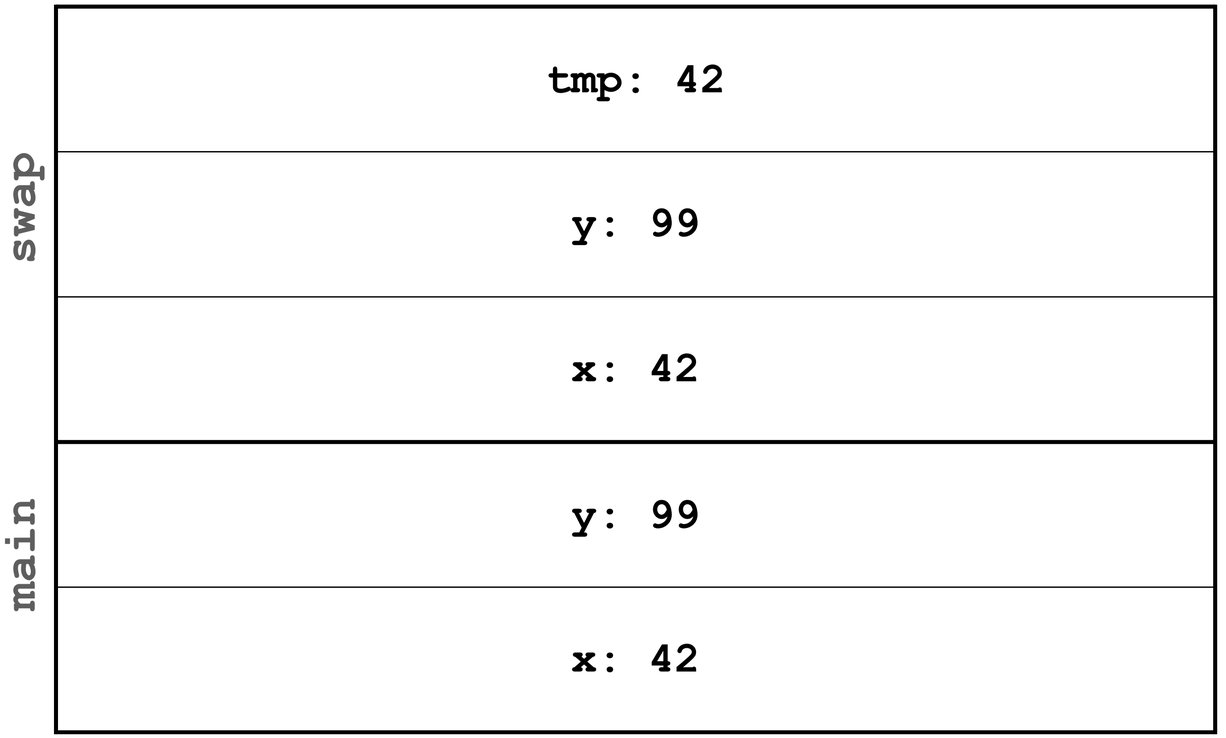
```c
#include <stdio.h>

void swap(int x, int y)
{
        int tmp = x;
        x = y;
        y = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(x, y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

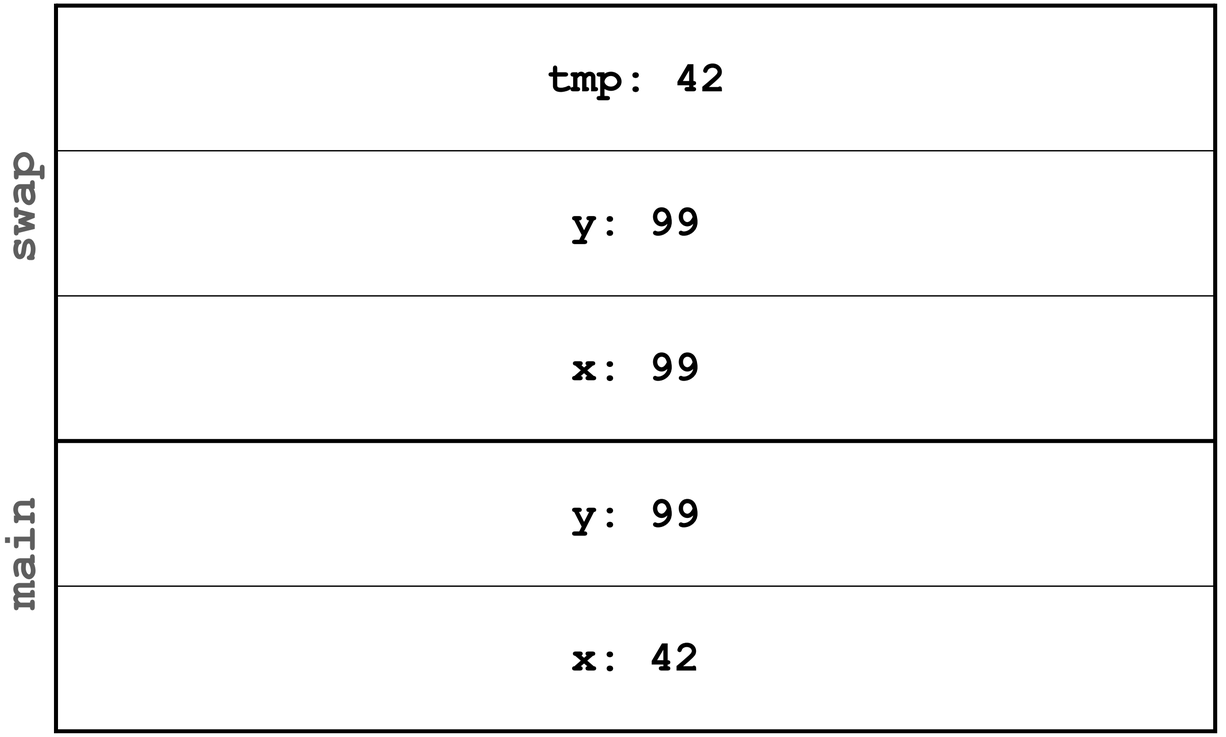
```c
#include <stdio.h>

void swap(int x, int y)
{
        int tmp = x;
        x = y;
        y = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(x, y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

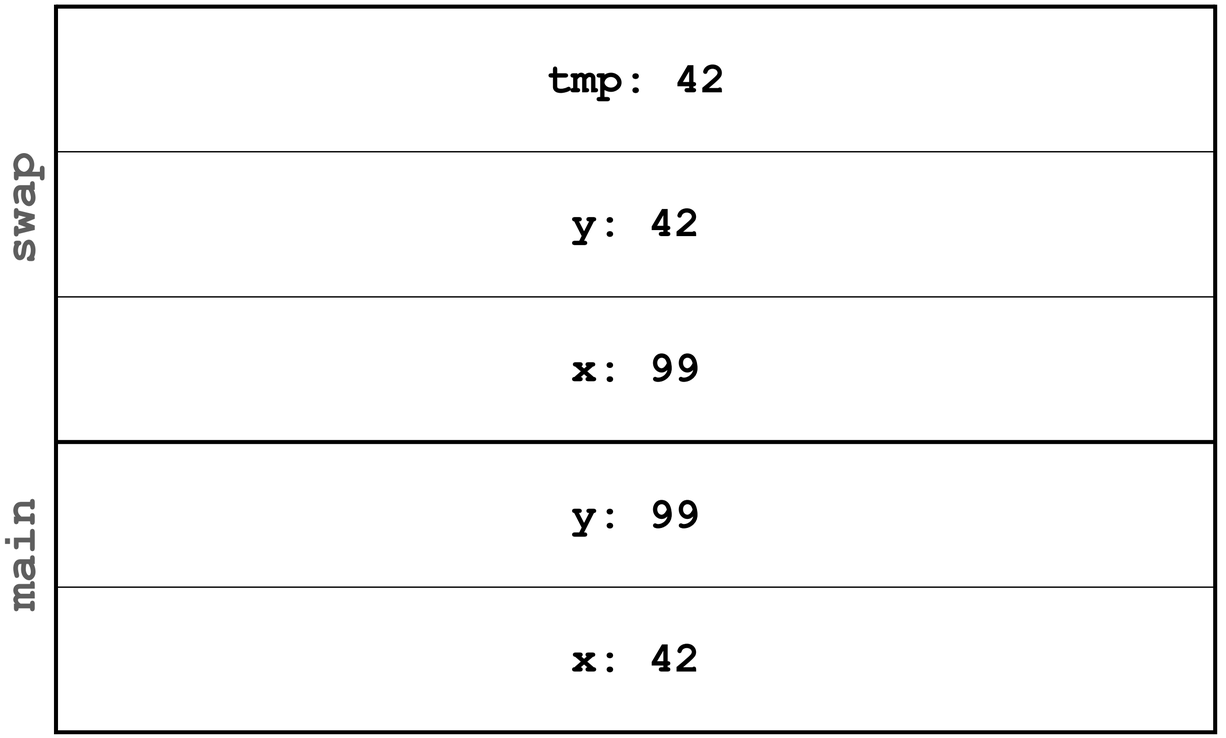
```c
#include <stdio.h>

void swap(int x, int y)
{
        int tmp = x;
        x = y;
        y = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(x, y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

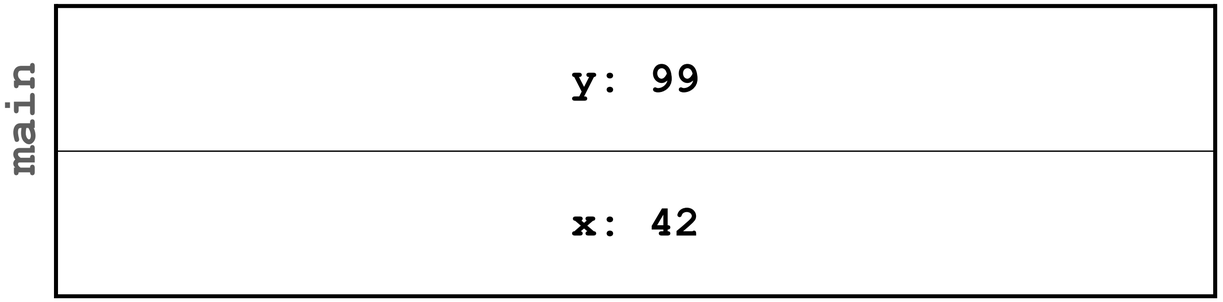
```c
#include <stdio.h>

void swap(int x, int y)
{
        int tmp = x;
        x = y;
        y = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(x, y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

```c
#include <stdio.h>

void swap(int x, int y)
{
        int tmp = x;
        x = y;
        y = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(x, y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

```c
#include <stdio.h>

void swap(int x, int y)
{
        int tmp = x;
        x = y;
        y = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(x, y);

        printf("%d %d\n", x, y);

        return 0;
}
```

Does it work?

No

```
byron@MacBook-Pro solutions % ./swap
42 99
```

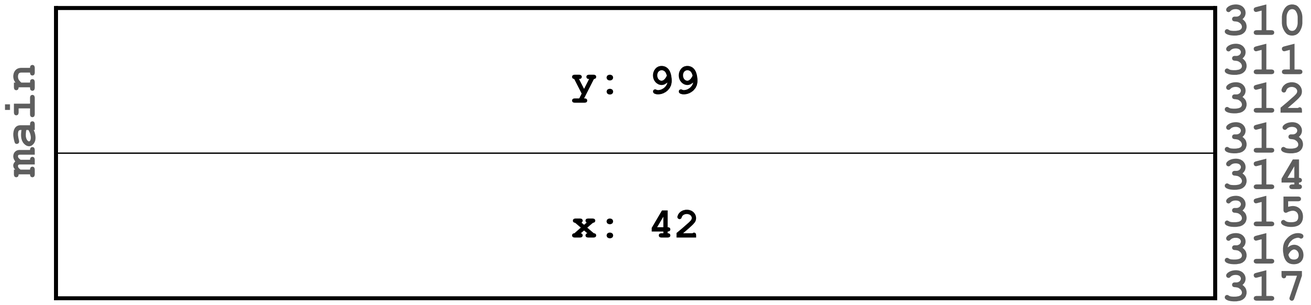
But why?

Function arguments are passed by *values*

Pass by Reference

swap

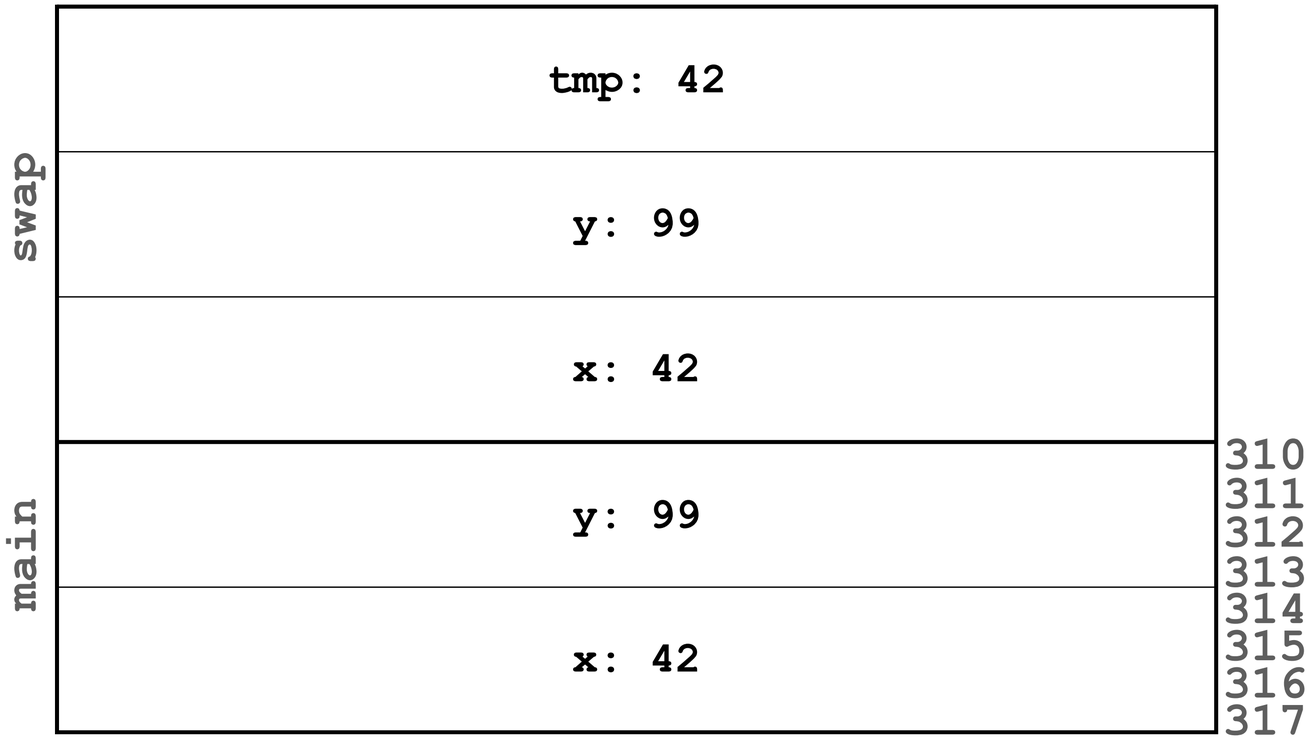
```c
#include <stdio.h>

void swap(int x, int y)
{
        int tmp = x;
        x = y;
        y = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(x, y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

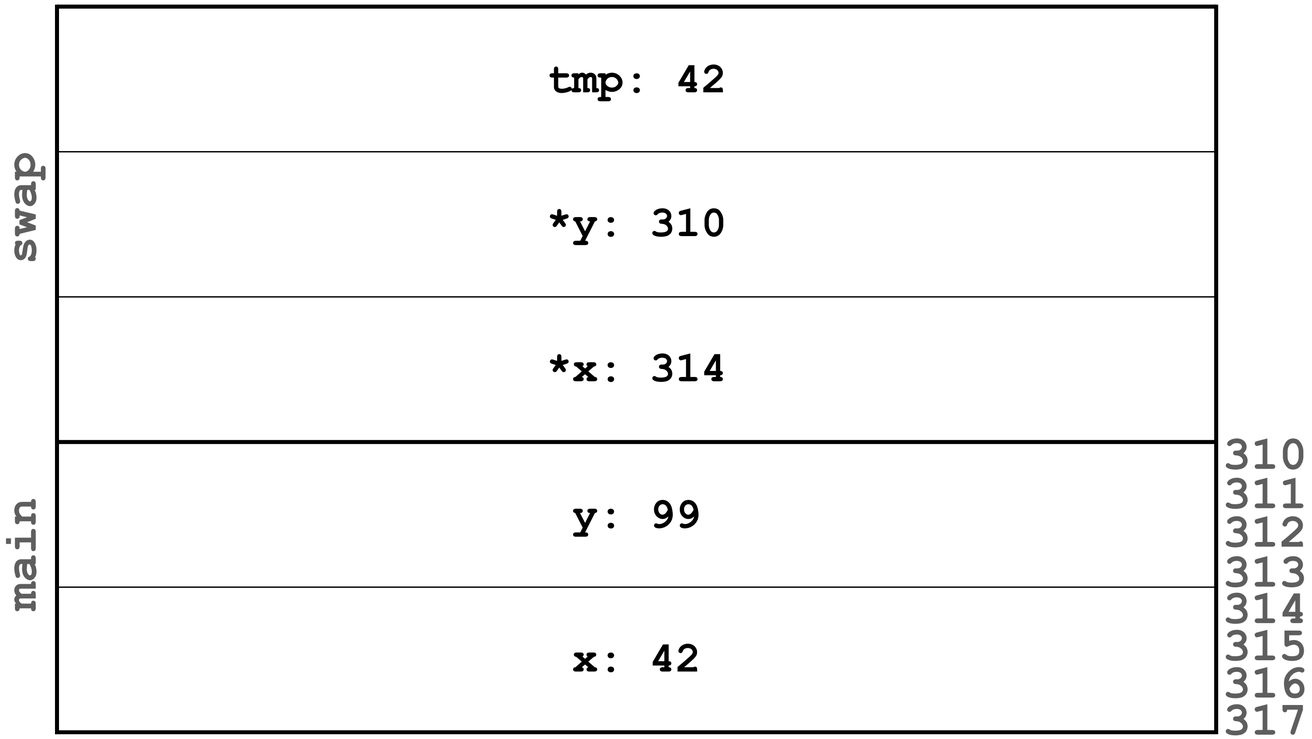
```c
#include <stdio.h>

void swap(int x, int y)
{
        int tmp = x;
        x = y;
        y = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(x, y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

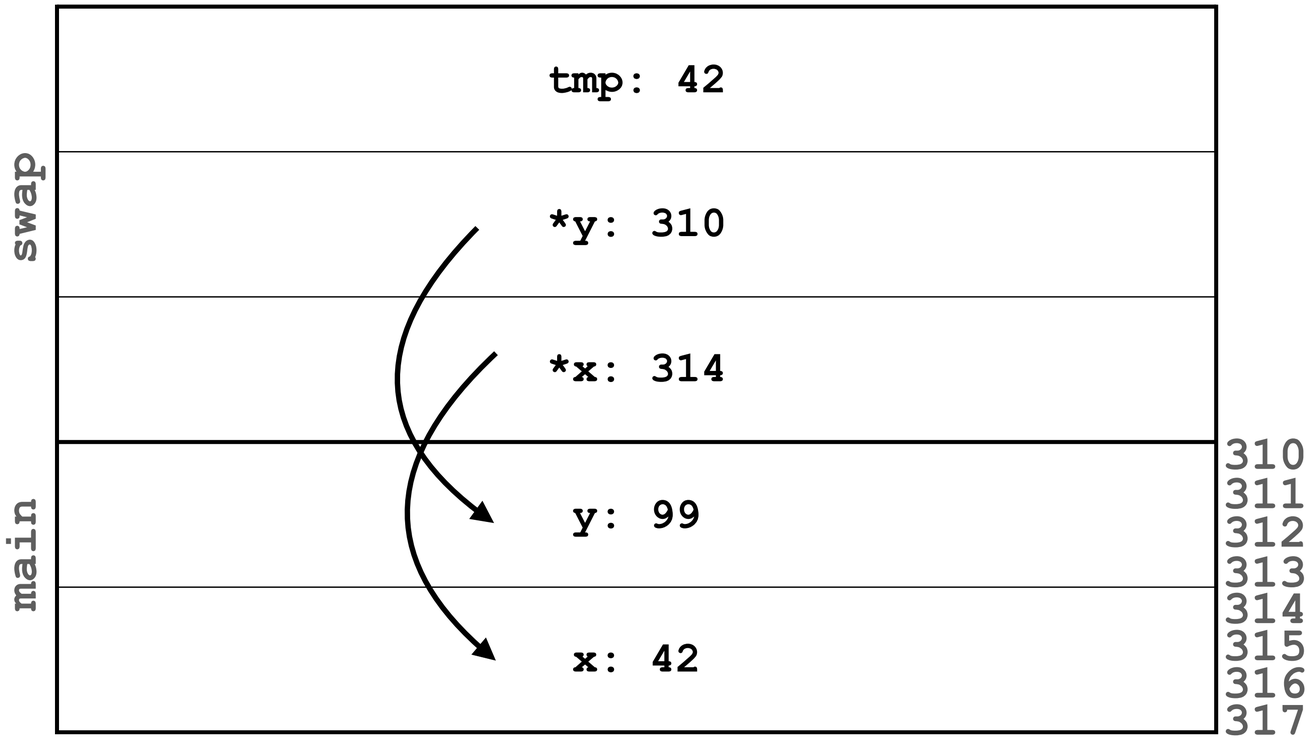
```c
#include <stdio.h>

void swap(int x, int y)
{
        int tmp = x;
        x = y;
        y = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(x, y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

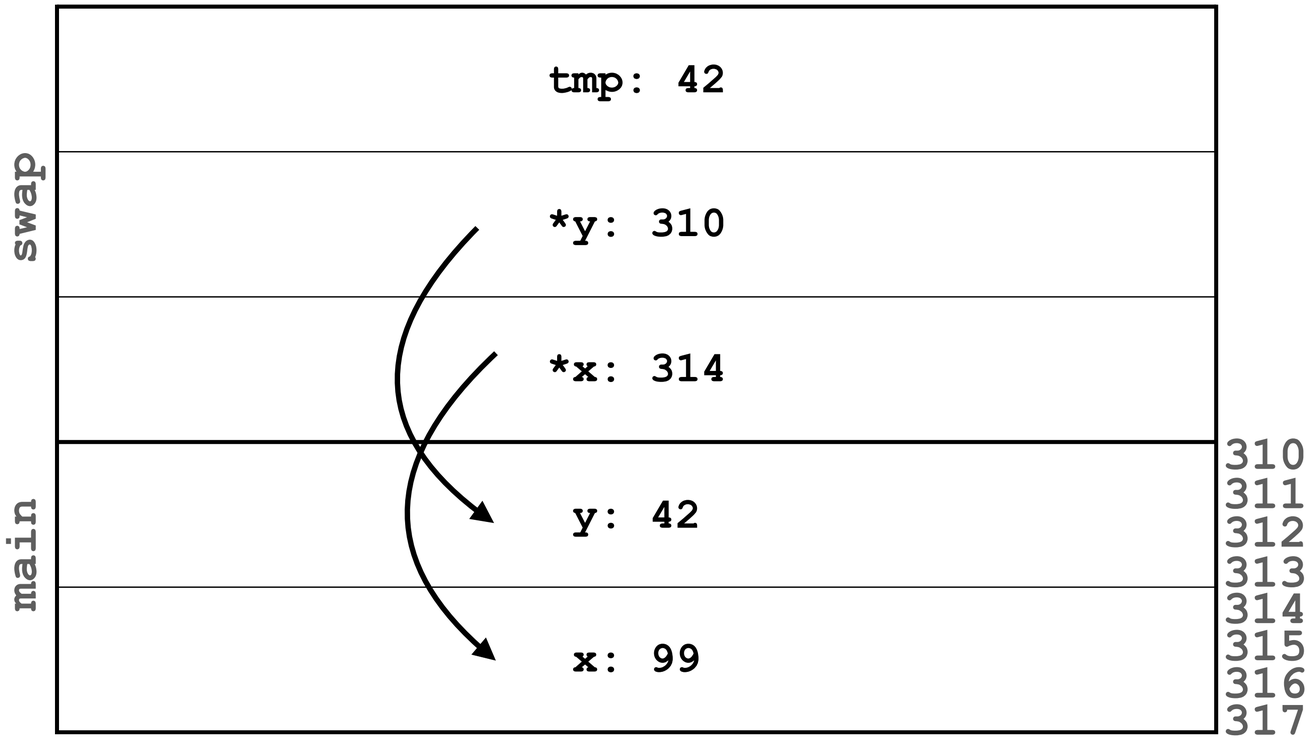
```c
#include <stdio.h>

void swap(int x, int y)
{
        int tmp = x;
        x = y;
        y = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(x, y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

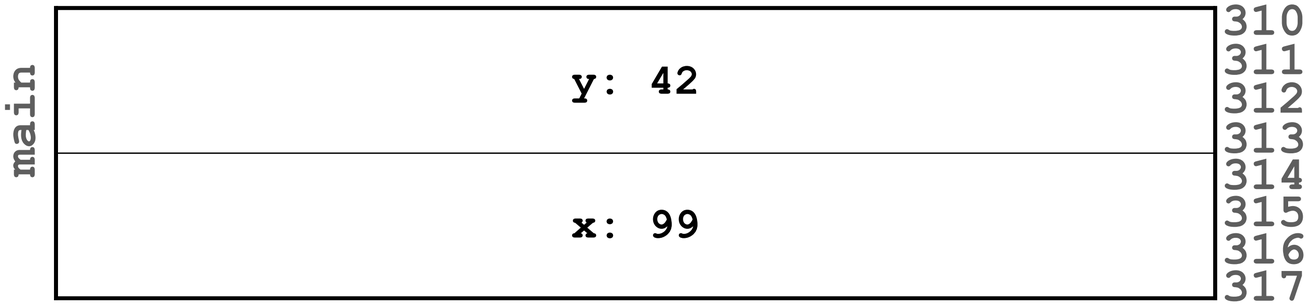
```c
#include <stdio.h>

void swap(int x, int y)
{
        int tmp = x;
        x = y;
        y = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(x, y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

```c
#include <stdio.h>

void swap(int x, int y)
{
        int tmp = x;
        x = y;
        y = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(x, y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pointers

Syntax

```
void swap(int x, int y)
{
        int tmp = x;
        x = y;
        y = tmp;
}
```

Pointers

Pointer Type

```
void swap(int *x_p, int *y_p)
{
        int tmp = x;
        x = y;
        y = tmp;
}
```

- `int *name` declares a *pointer* to int.
- `name` stores a *memory location* (to an integer).

Pointers

Pointer Access

```
void swap(int *x_p, int *y_p)
{
        int tmp = *x_p;
        *x_p = *y_p;
        *y_p = tmp;
}
```

- `int *name` declares a *pointer* to int.
- `name` stores a *memory location* (to an integer).
- `*name` tells the compiler we want to *follow* the pointer.

Pointers

Pointer Access

```
void swap(int *x_p, int *y_p)
{
        int tmp = *x_p;
        *x_p = *y_p;
        *y_p = tmp;
}
```

- `x_p` and `y_p` are still variables (of this weird pointer type) with their own locations. (lecture 1)
- `*x_p`:
  - look up address `a1` of `x_p` (in compiler's internal table)
  - read from `a1` and get another address `a2`
  - read/store from `a2` to get/put the value behind the pointer.

Pointers

Obtaining Addresses

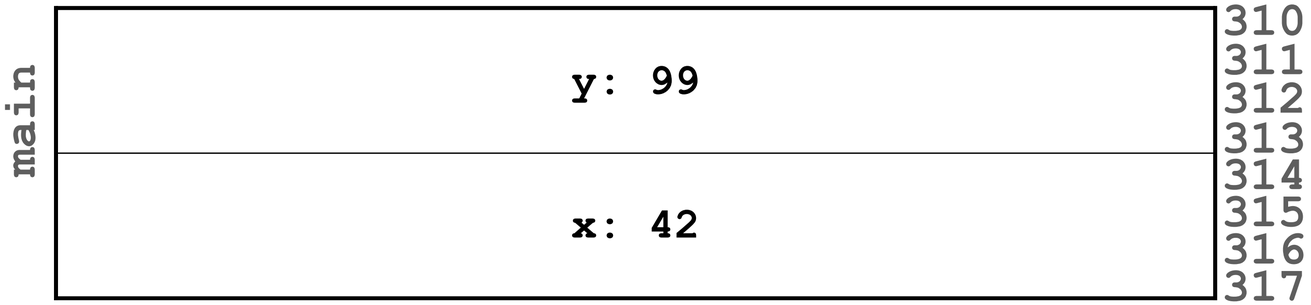
```
int main(void)
{
        int x = 42;
        int y = 99;

        swap(&x, &y);

        printf("%d %d\n", x, y);

        return 0;
}
```

- `&x` gets the *address* of `x`.
- Compiler has to know the address anyway...

Pass by Reference

swap

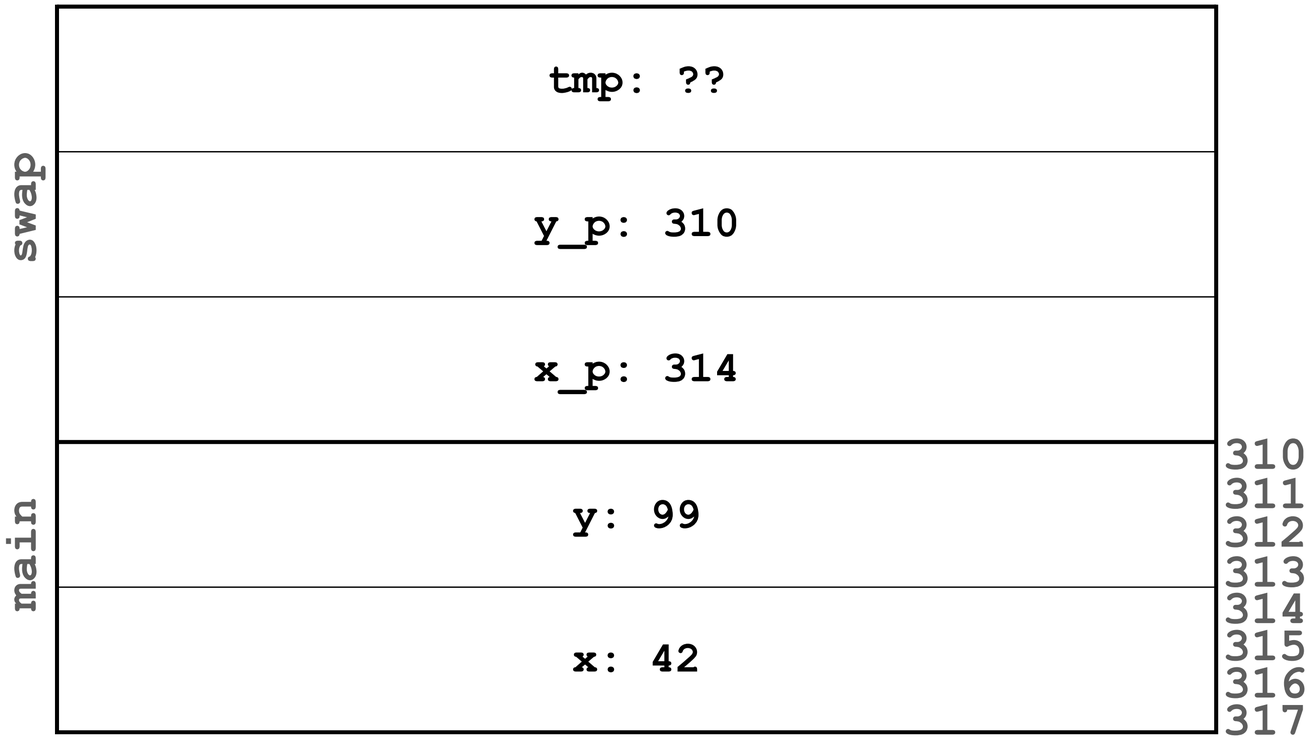
```c
#include <stdio.h>

void swap(int *x_p, int *y_p)
{
        int tmp = *x_p;
        *x_p = *y_p;
        *y_p = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(&x, &y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

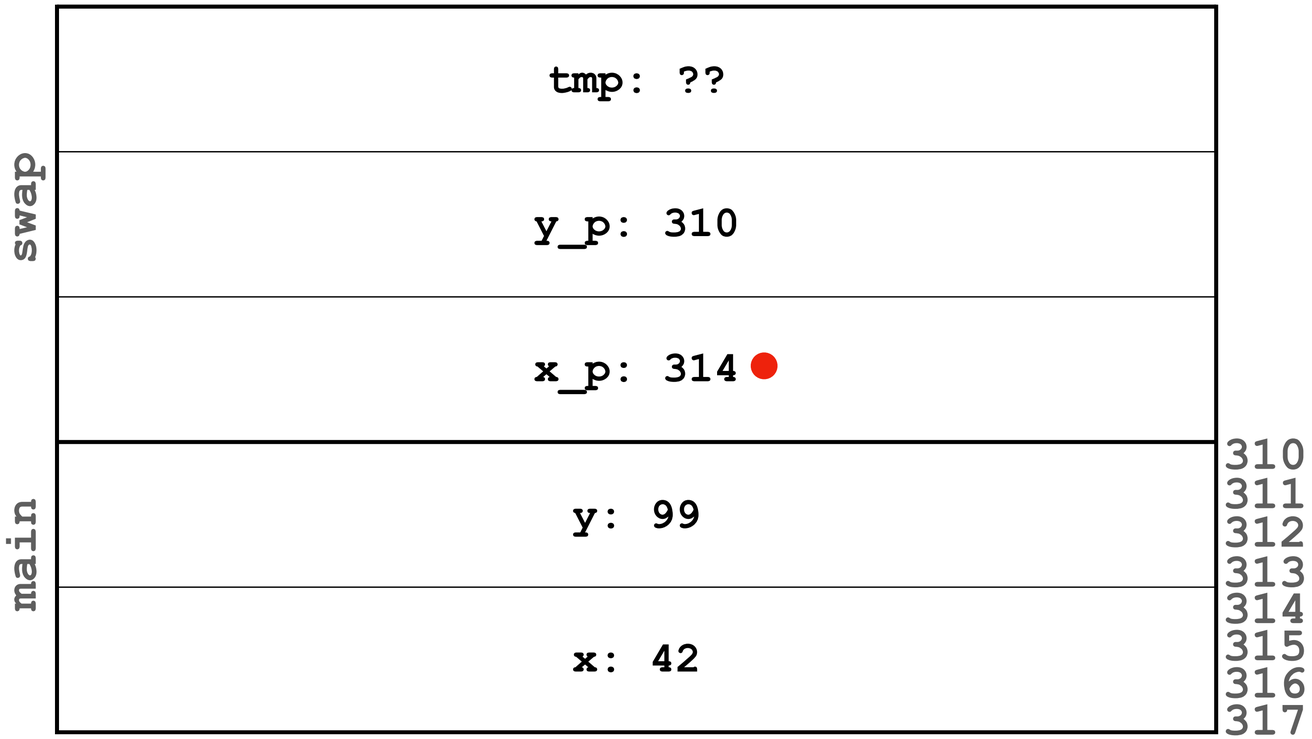
```c
#include <stdio.h>

void swap(int *x_p, int *y_p)
{
        int tmp = *x_p;
        *x_p = *y_p;
        *y_p = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(&x, &y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

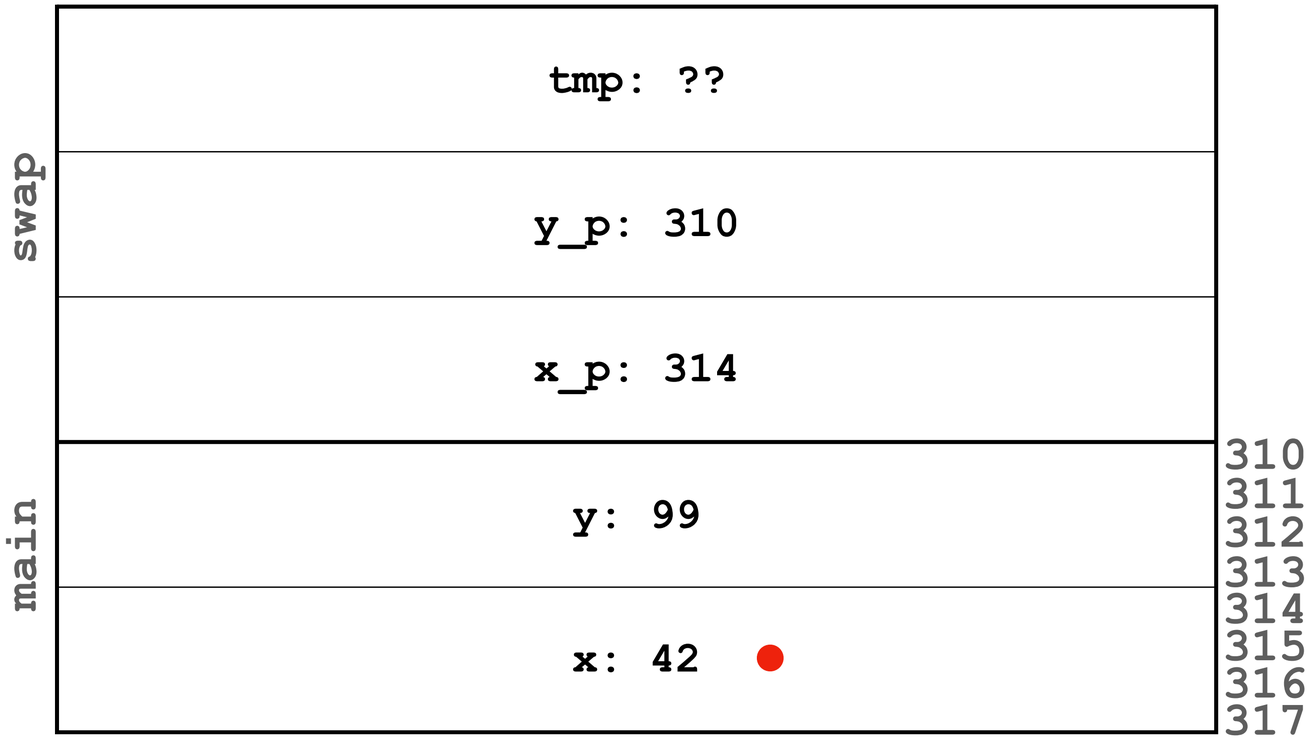
```c
#include <stdio.h>

void swap(int *x_p, int *y_p)
{
        int tmp = *x_p;
        *x_p = *y_p;
        *y_p = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(&x, &y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

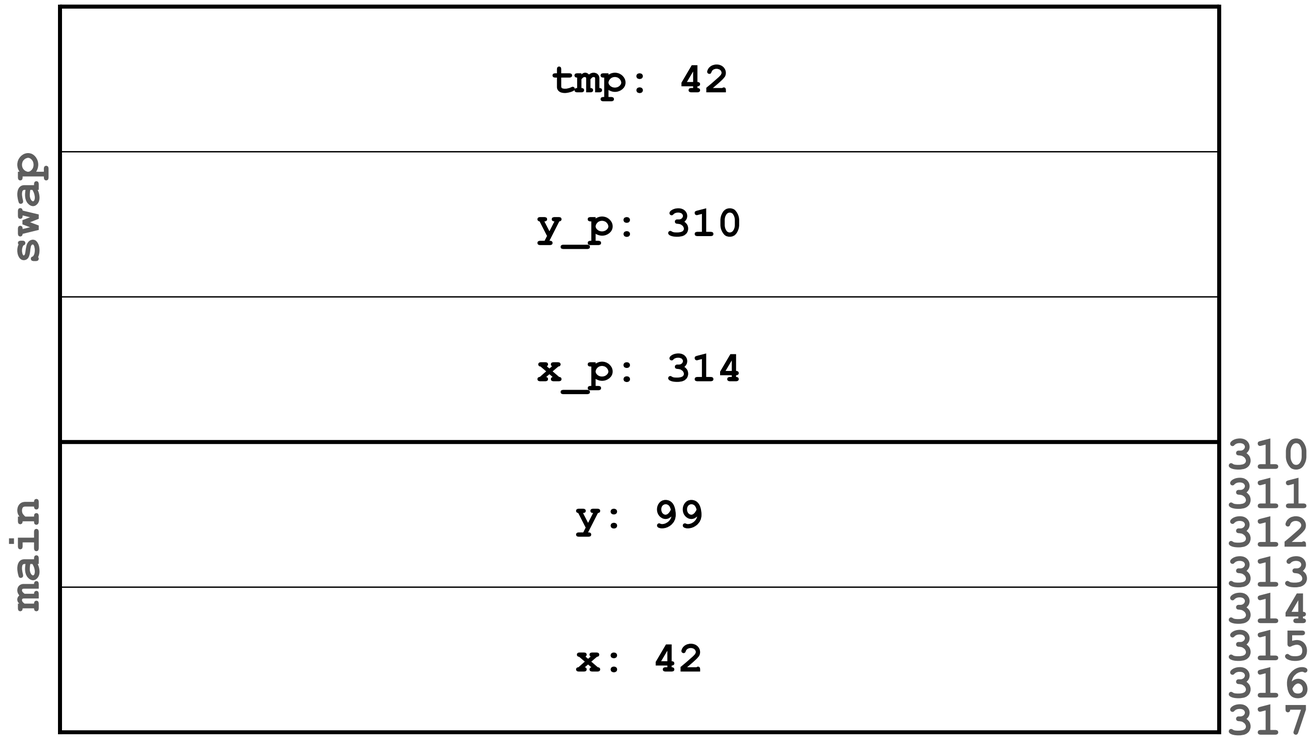
```c
#include <stdio.h>

void swap(int *x_p, int *y_p)
{
        int tmp = *x_p;
        *x_p = *y_p;
        *y_p = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(&x, &y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

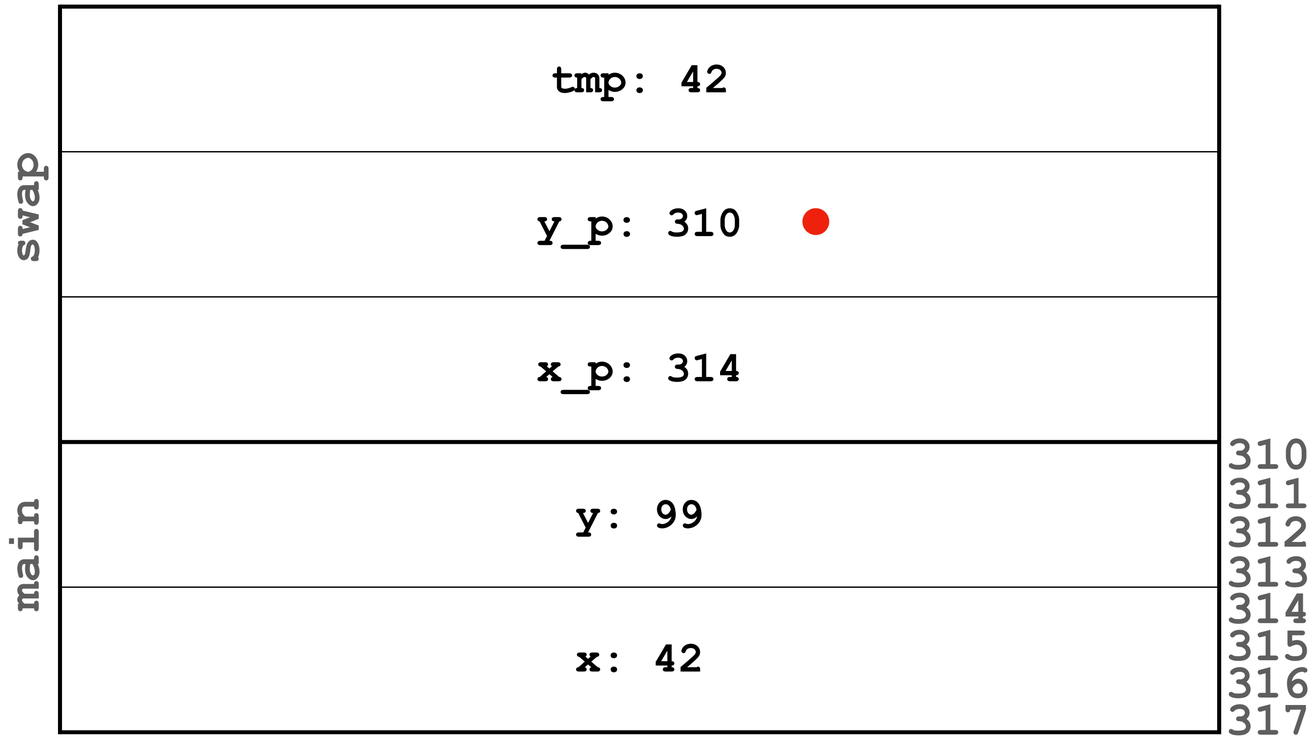
```c
#include <stdio.h>

void swap(int *x_p, int *y_p)
{
        int tmp = *x_p;
        *x_p = *y_p;
        *y_p = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(&x, &y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

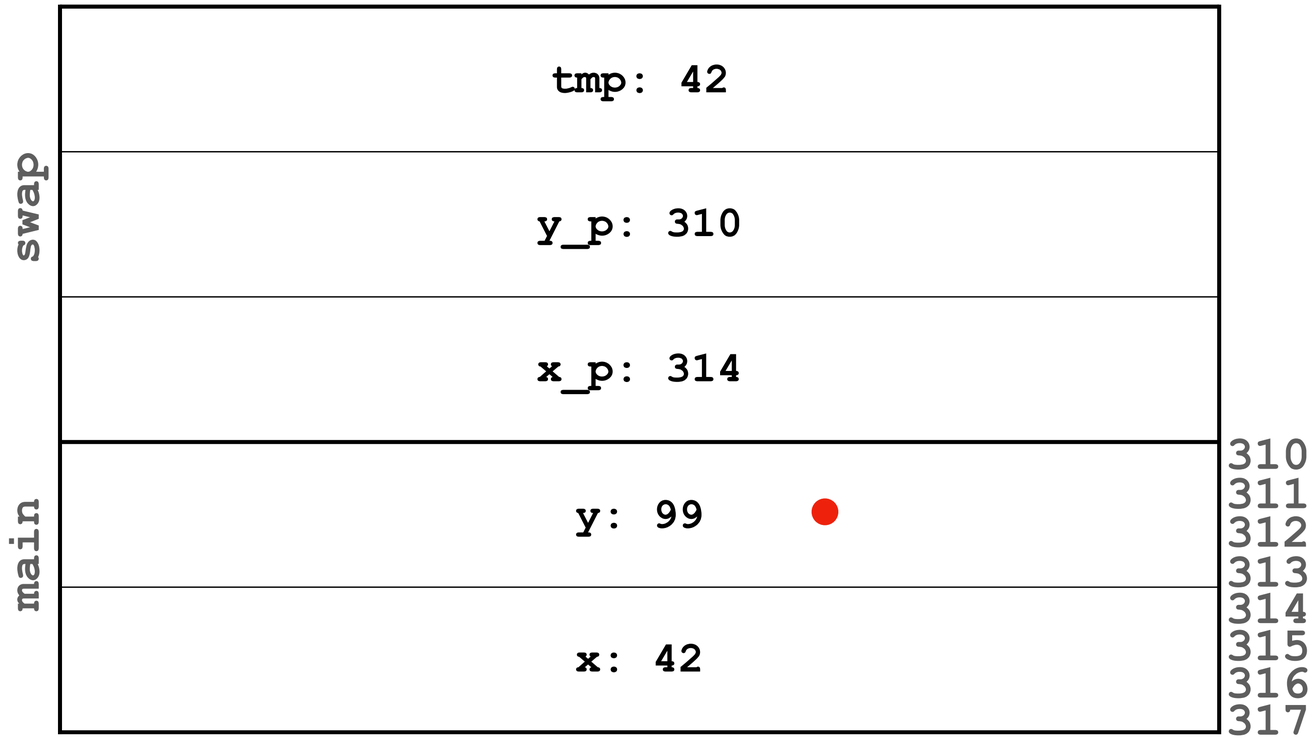
```c
#include <stdio.h>

void swap(int *x_p, int *y_p)
{
        int tmp = *x_p;
        *x_p = *y_p;
        *y_p = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(&x, &y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

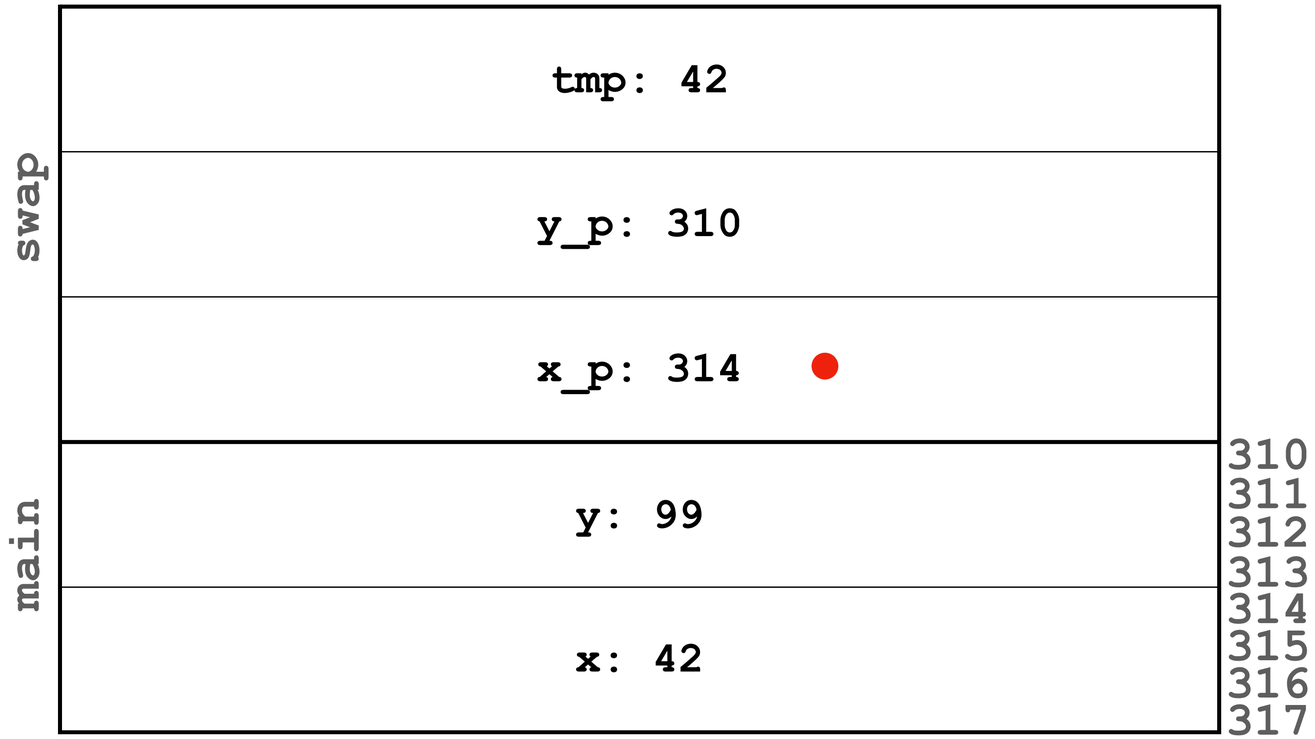
```c
#include <stdio.h>

void swap(int *x_p, int *y_p)
{
        int tmp = *x_p;
        *x_p = *y_p;
        *y_p = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(&x, &y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

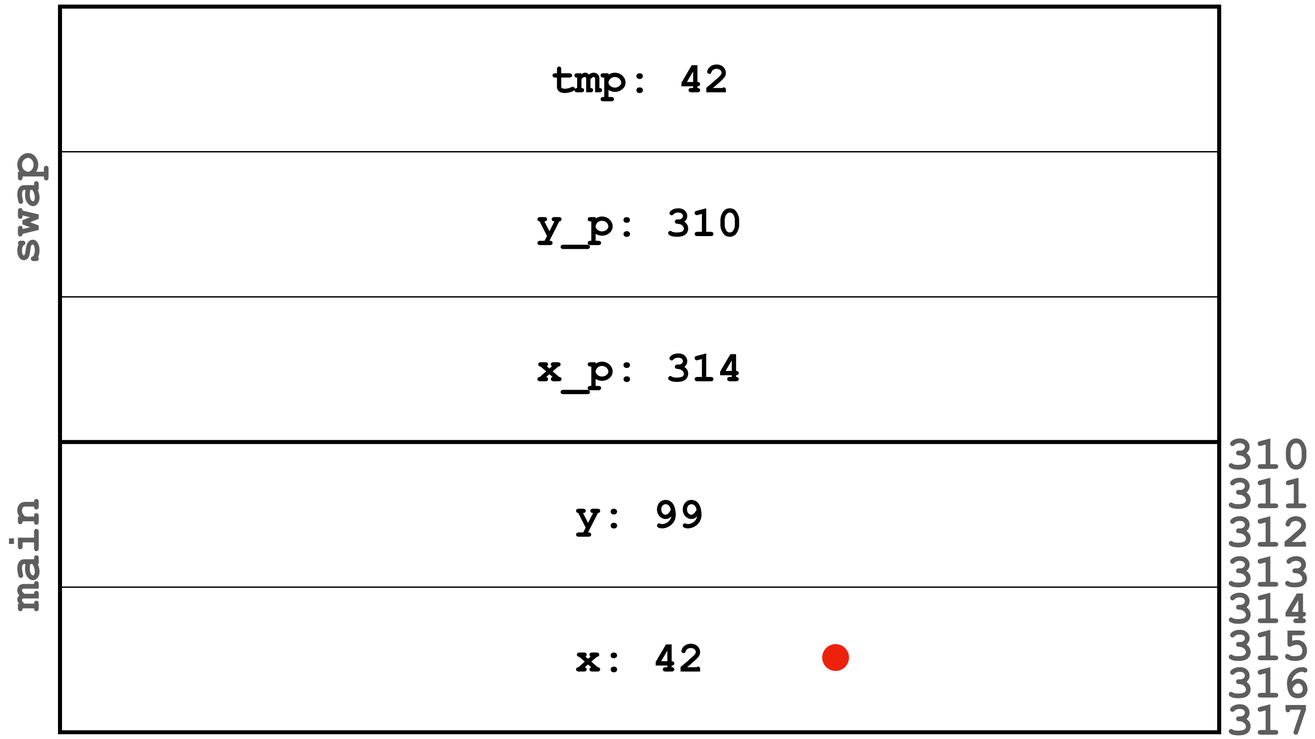
```c
#include <stdio.h>

void swap(int *x_p, int *y_p)
{
        int tmp = *x_p;
        *x_p = *y_p;
        *y_p = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(&x, &y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

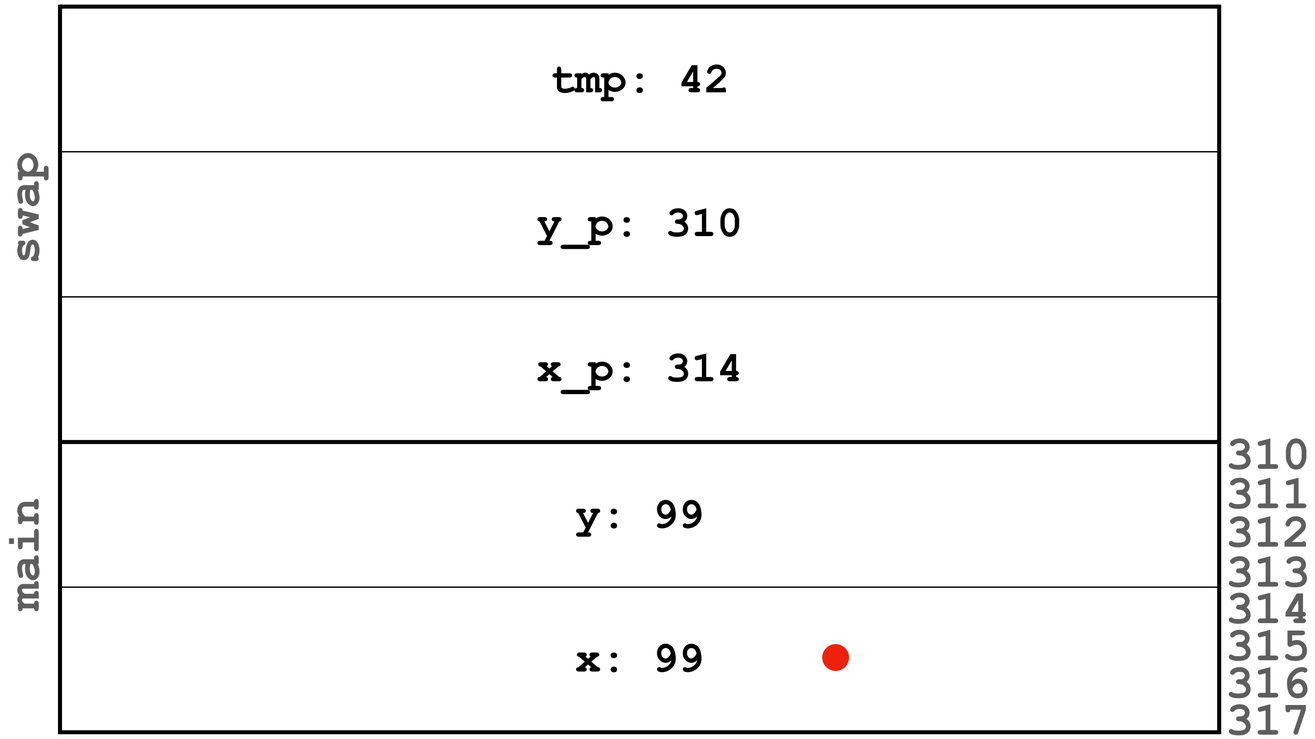
```c
#include <stdio.h>

void swap(int *x_p, int *y_p)
{
        int tmp = *x_p;
        *x_p = *y_p;
        *y_p = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(&x, &y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

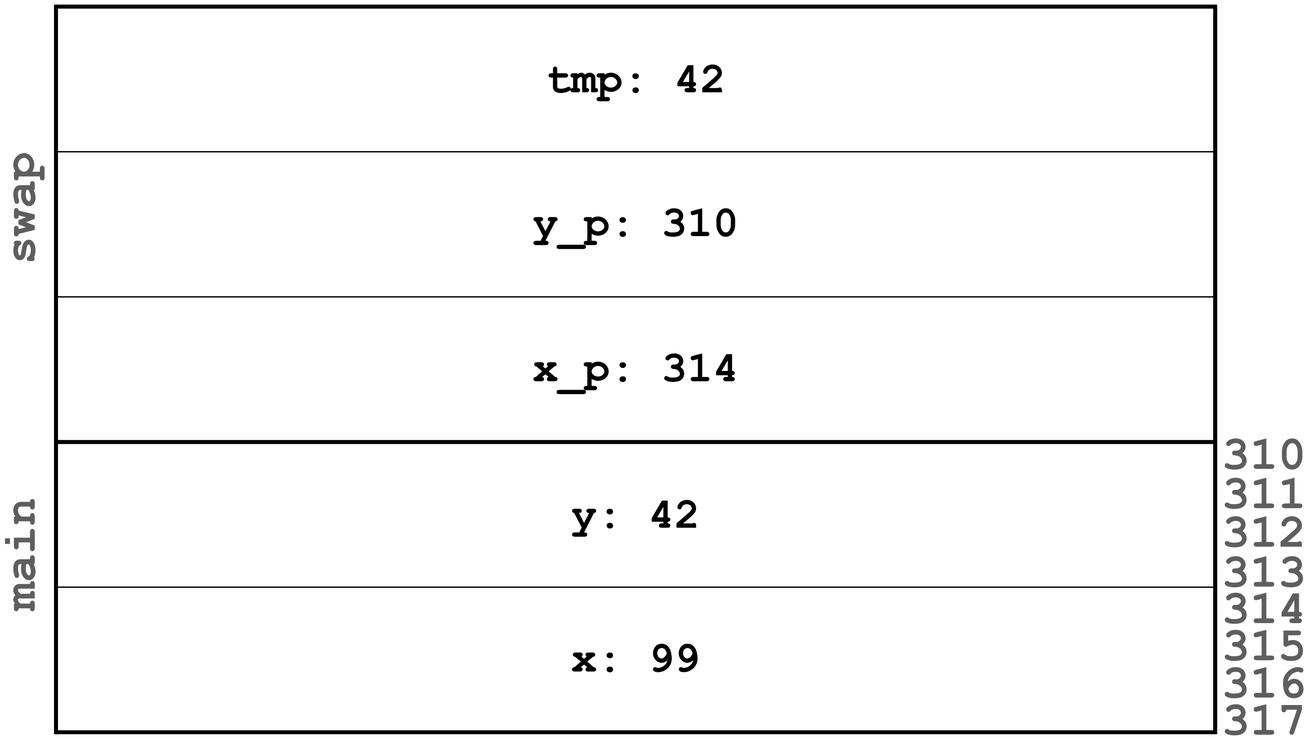
```c
#include <stdio.h>

void swap(int *x_p, int *y_p)
{
        int tmp = *x_p;
        *x_p = *y_p;
        *y_p = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(&x, &y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

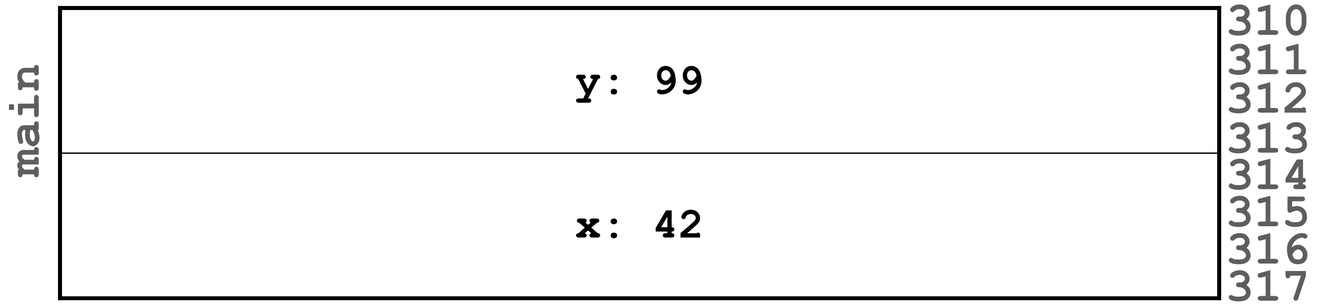
```c
#include <stdio.h>

void swap(int *x_p, int *y_p)
{
        int tmp = *x_p;
        *x_p = *y_p;
        *y_p = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(&x, &y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pass by Reference

swap

```c
#include <stdio.h>

void swap(int *x_p, int *y_p)
{
        int tmp = *x_p;
        *x_p = *y_p;
        *y_p = tmp;
}

int main(void)
{
        int x = 42;
        int y = 99;

        swap(&x, &y);

        printf("%d %d\n", x, y);

        return 0;
}
```



Pointers

Checkpoint I

Compile or error? `int x = 10, y = 10;`

```
int *x_p = x;
int *x_p = &x;
int z = *x_p;
*x_p = 30;
x_p = &y;
*x_p = 30;
int a = *&*&*&*&*&*&*&*&y;
```

Pointers

Checkpoint I

Compile or error? `int x = 10, y = 10;`

`int *x_p = x;`	🚫 x_p wants an address
`int *x_p = &x;`	✅ x_p has x's address
`int z = *x_p;`	✅ z has x's value
`*x_p = 30;`	✅ x is now 30
`x_p = &y;`	✅ x_p has y's address
`*x_p = 30;`	✅ y is now 30
`int a = *&*&*&*&*&*&*&*&y;`	✅ *& cancel out, DO NOT WRITE THIS

Pointers

Checkpoint I

- `type *name;` declares a variable of type "pointer to `type`"
- `*name` "dereferences" name -- following the address contained in `name` for reading or writing
- `&name` gets the address of name -- if name has type "`type`", `&name` has type "`type *`"
- Pointers can be used for passing arguments by references.
- Pointers enable sharing the same piece of data between functions.

Pointers

Example: Multiple return values

```
def divide(x, y):
    q = 0
    while y <= x:
        x -= y
        q += 1
    return q, x

q, r = divide(7, 3)
print(q, r) # 2, 1
```

- C functions can return at most 1 thing. :(

Pointers

Example: Multiple return values

```python
def divide(x, y):
    q = 0
    while y <= x:
        x -= y
        q += 1
    return q, x

q, r = divide(7, 3)
print(q, r) # 2, 1
```

```c
void divide(int x, int y, int *q_p, int *r_p)
{
        int q = 0;
        while (y <= x) {
                x -= y;
                q += 1;
        }
        *q_p = q;
        *r_p = x;
}

int main(void)
{
        int q, r;
        divide(7, 3, &q, &r);

        printf("%d %d\n", q, r); // 2, 1

        return 0;
}
```

Pointers

Example: Array

```c
int sum(int *arr, int n)
{
        int sum = 0;
        for (int i = 0; i < n; i++) {
                sum += arr[i];
        }
        return sum;
}

int main(void)
{
        int numbers[7] = { 0, 1, 2, 3, 4, 5, 6 };

        printf("%d\n", sum(numbers, 7));

        return 0;
}
```

- Even when `number` is a massive array, no copying is needed
- `&numbers[0] == numbers`
- Pitfall: `==` does pointer comparison between arrays, does not compare elements
  - use `for` loop