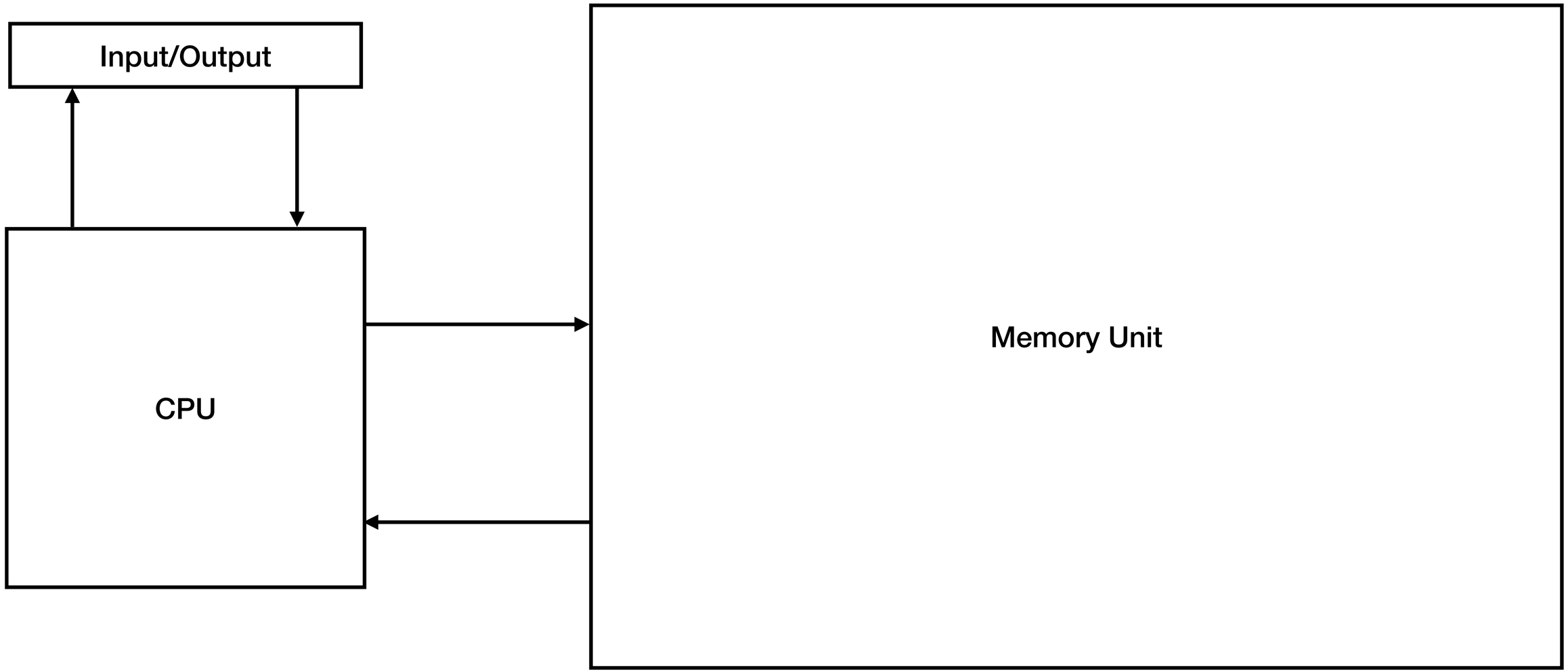
Pointers I

CS143: lecture 7

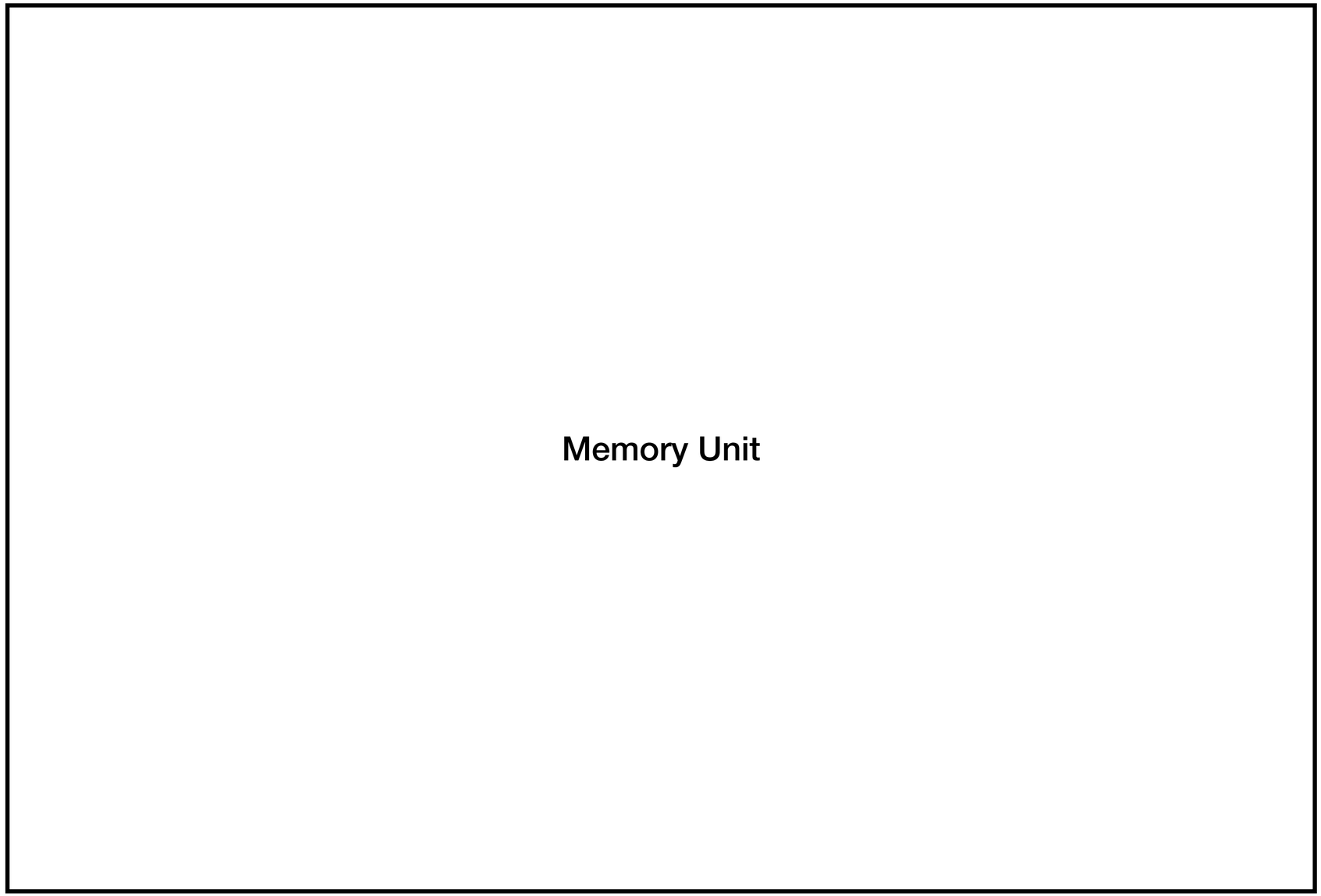
Konstantinos Ameranis, June 30

A Von Neumann Machine



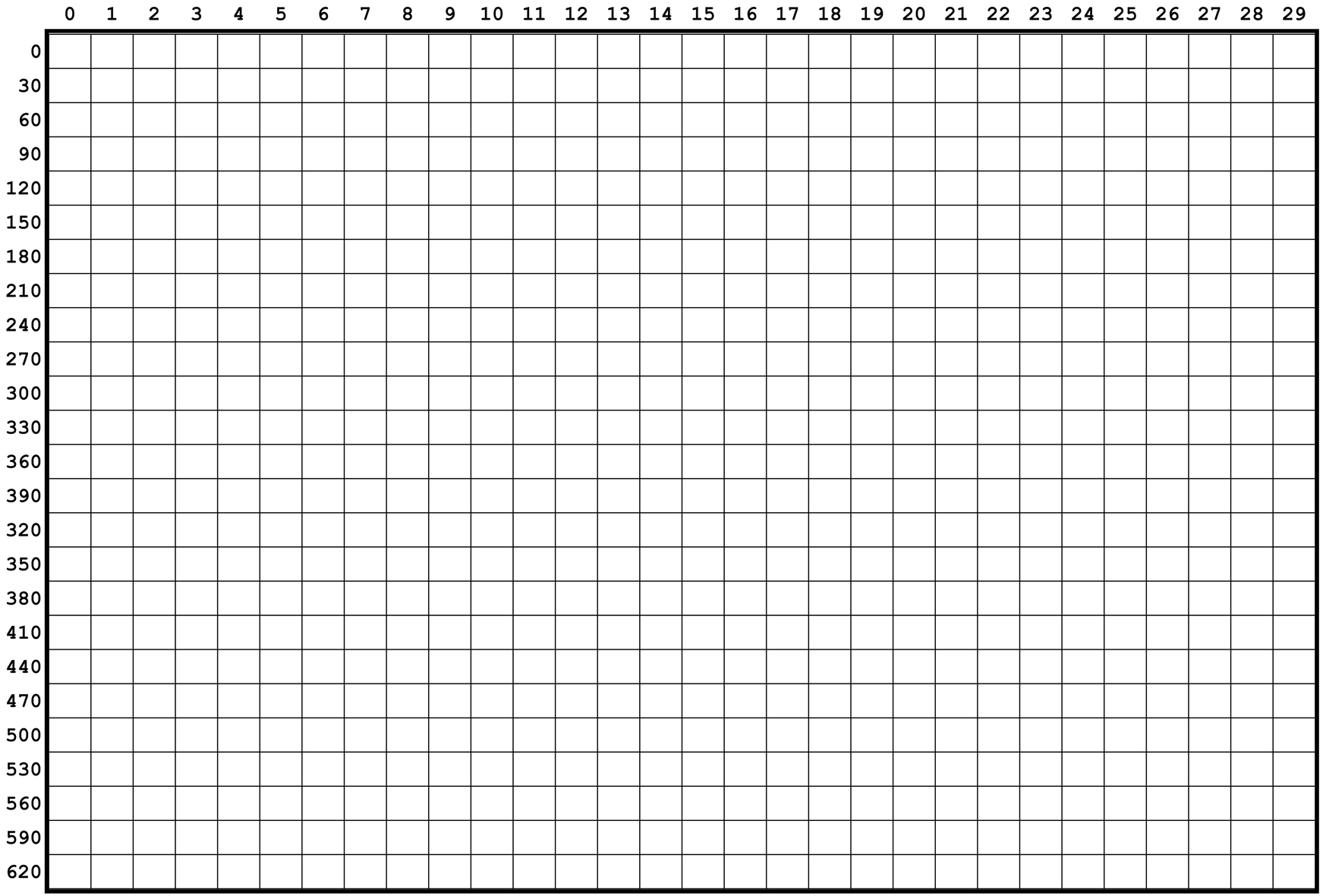
Variables

In Memory



Variables

In Memory



Variables

In Memory

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
0	11	119	115	102	115	31	58	127	49	108	161	128	120	139	49	138	165	8	173	1	167	160	18	148	131	240	49	227	116	84
30	28	77	126	184	77	46	7	95	127	117	145	164	77	6	98	12	232	252	238	143	39	49	225	210	142	227	28	182	82	81
60	119	209	67	25	13	220	39	84	249	190	14	10	23	72	200	64	173	216	129	195	171	37	251	217	205	210	229	41	198	78
90	99	36	153	109	28	194	94	28	170	94	224	34	24	87	70	112	107	213	48	195	164	117	202	169	104	238	240	56	30	142
120	220	135	65	195	236	211	117	131	206	198	96	83	142	32	50	165	64	177	18	205	210	143	210	165	221	227	238	132	14	131
150	4	103	109	183	254	173	250	88	34	47	176	173	193	72	233	103	233	174	234	98	25	130	10	149	230	103	54	59	107	99
180	142	20	16	142	38	19	198	85	164	89	134	171	213	75	8	112	20	197	29	204	64	210	47	244	188	237	99	150	207	109
210	168	161	208	83	103	221	140	239	166	233	93	153	173	180	69	27	69	136	210	93	80	58	47	102	1	41	67	21	74	79
240	32	63	212	256	52	168	244	19	75	228	210	121	119	98	133	226	77	43	105	72	55	227	140	169	6	98	188	222	108	98
270	97	140	8	67	239	106	155	82	110	244	93	183	172	154	137	57	50	158	107	81	67	123	227	220	240	30	119	11	119	243
300	206	46	218	50	170	218	247	34	188	65	34	247	192	117	35	67	146	26	120	150	48	166	174	33	138	50	197	120	155	163
330	114	152	68	122	254	126	76	131	131	73	183	86	144	107	232	247	41	31	32	169	97	32	146	216	197	167	90	1	20	231
360	152	107	192	60	137	191	4	240	122	76	101	251	118	197	71	183	12	89	90	10	19	190	204	17	170	178	236	108	16	26
390	206	171	202	187	191	230	20	148	198	183	243	71	136	125	204	230	72	239	151	128	63	126	205	111	74	248	95	21	170	152
320	224	250	208	211	99	72	232	60	8	38	108	5	41	15	71	195	26	67	84	139	100	173	199	206	219	60	235	141	39	208
350	183	72	75	12	99	165	165	246	0	117	204	38	181	2	129	111	121	88	28	127	191	64	244	205	76	107	103	146	147	181
380	163	161	77	44	145	98	89	166	180	151	152	99	225	169	130	73	41	238	20	17	253	139	6	166	157	45	89	138	159	34
410	203	9	70	34	43	19	169	72	160	21	250	34	197	157	113	192	27	52	26	35	84	95	39	171	99	246	234	45	214	47
440	39	195	93	109	19	185	68	206	192	224	235	187	140	68	205	129	55	41	30	143	198	222	78	25	2	199	94	102	149	0
470	150	28	18	225	182	94	46	81	198	66	252	90	110	65	161	241	36	17	151	225	130	89	67	151	160	15	195	160	212	117
500	133	40	139	147	250	50	122	4	99	11	201	8	6	225	104	153	146	103	67	1	155	165	103	158	55	89	23	110	133	58
530	118	33	145	42	247	93	21	202	196	21	203	63	248	155	96	185	138	179	150	134	139	37	36	132	140	7	211	167	11	99
560	187	210	219	52	121	39	24	125	27	61	171	110	26	169	192	6	148	249	190	24	8	9	229	107	10	244	191	166	137	27
590	167	65	33	66	149	133	173	128	151	196	207	41	108	219	46	208	13	209	105	154	19	175	135	42	63	36	143	254	32	215
620	215	40	157	139	175	235	235	92	197	203	68	109	241	194	92	231	168	173	30	206	113	97	81	150	145	163	237	208	133	36

Variables

Review

Variables

Review

- A variable has a type (thereby size) and a location in memory.

Variables

Review

- A variable has a type (thereby size) and a location in memory.
- Declaration and initialization

Variables

Review

- A variable has a type (thereby size) and a location in memory.
- Declaration and initialization
- Arrays and strings

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?

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 }
```

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?

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?

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?

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?

Variable Lifetime

An Example

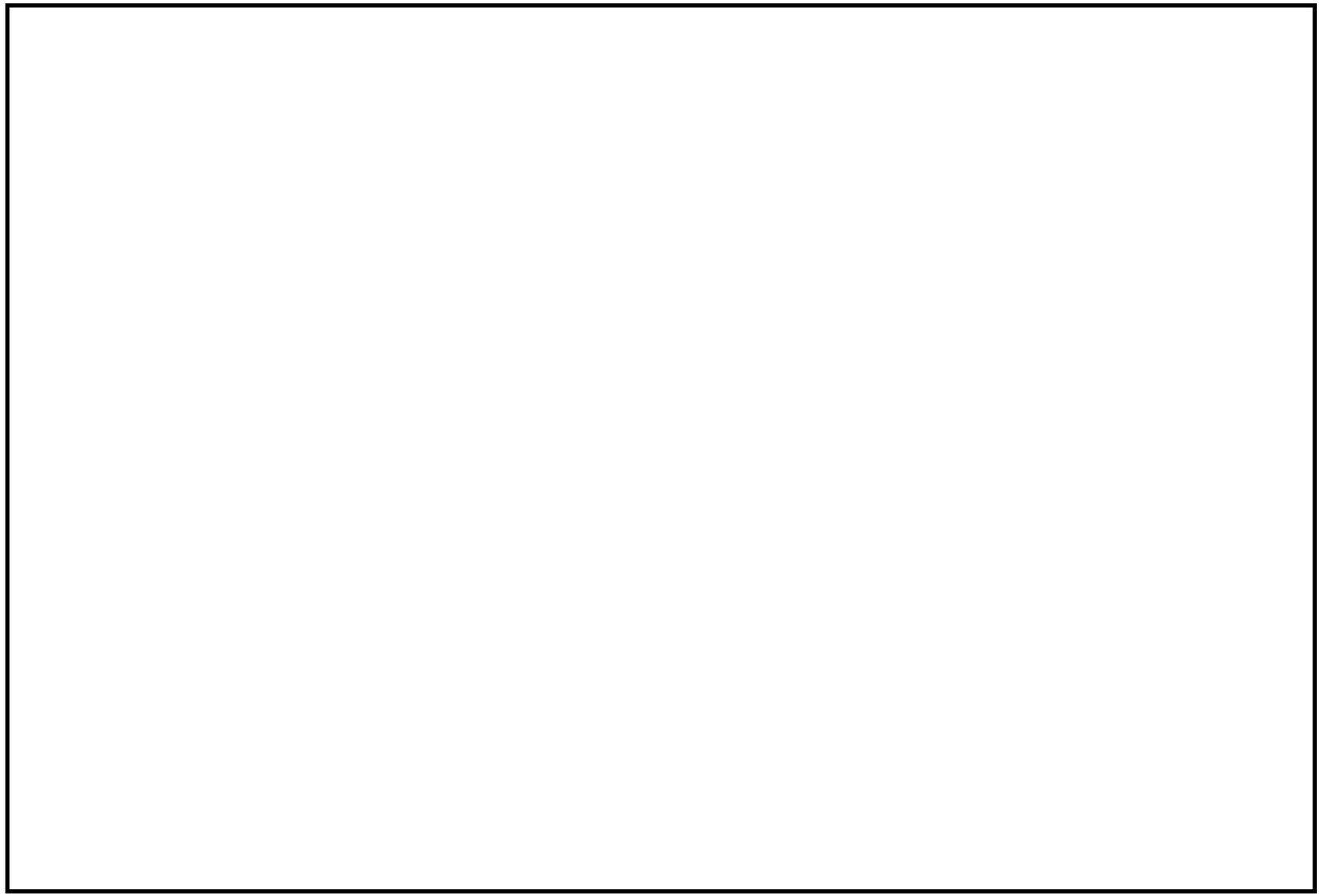
```
01 int f(int x)
02 {
03         int y = x * 2;
04         return y;
05 }
06
07 int main(void)
08 {
09         int a = f(10);
10         int b = f(a);
11         printf("%d\n", b);
12
13         return 0;
14 }
```

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

Variable Lifetime

An Example

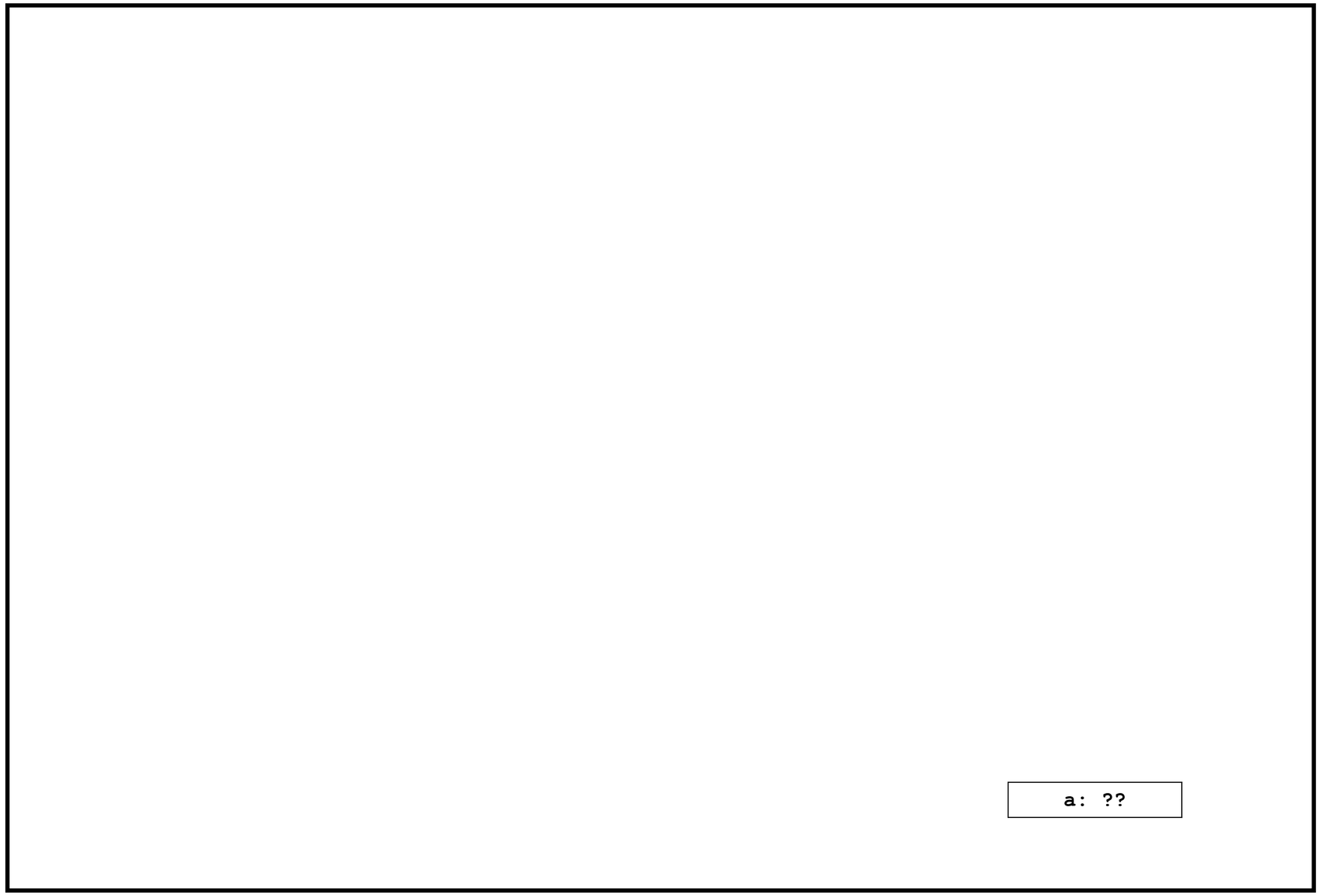
```
int f(int x)
{
        int y = x * 2;
        return y;
}

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

        return 0;
}
```



Variable Lifetime

An Example

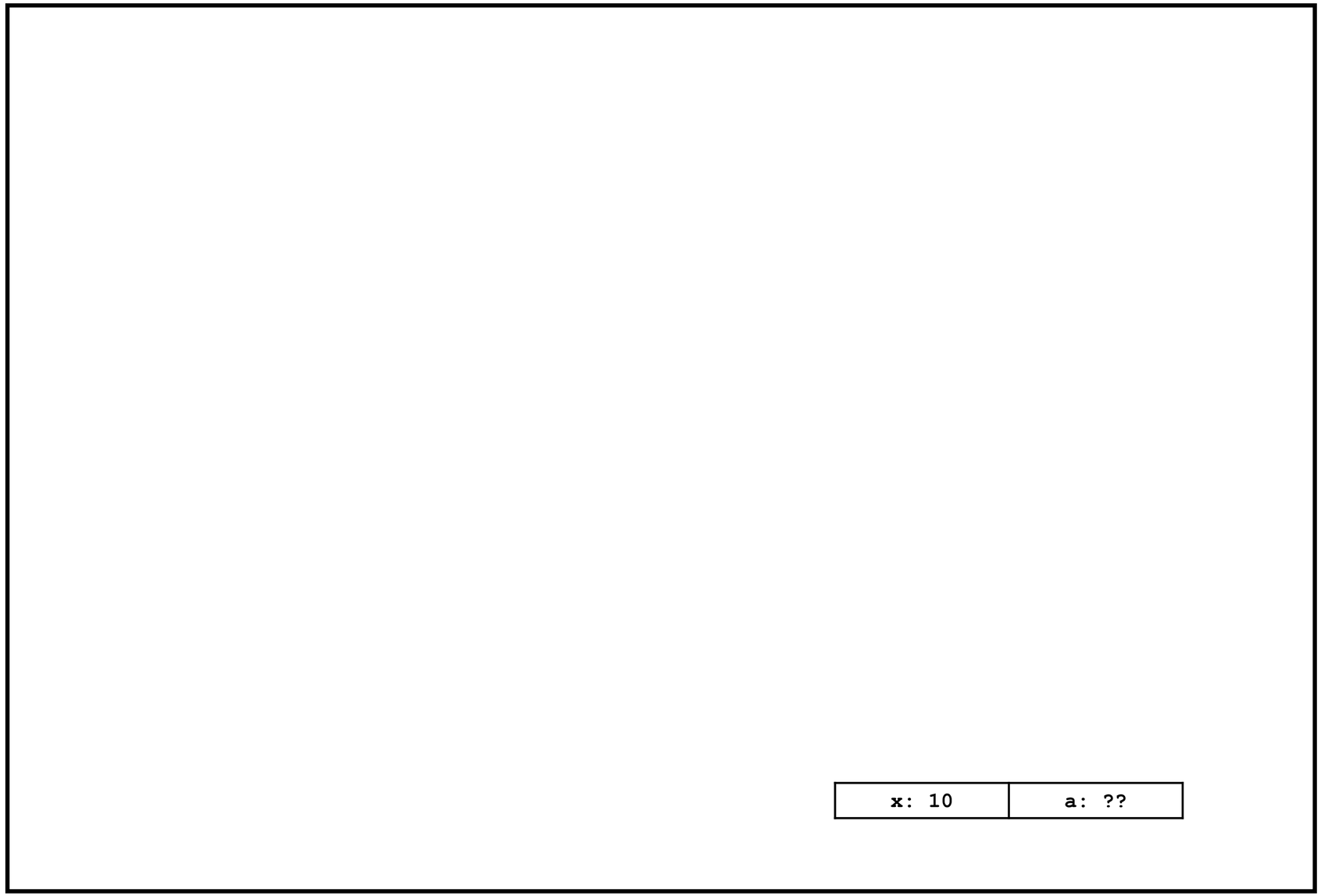
```
int f(int x)
{
        int y = x * 2;
        return y;
}

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

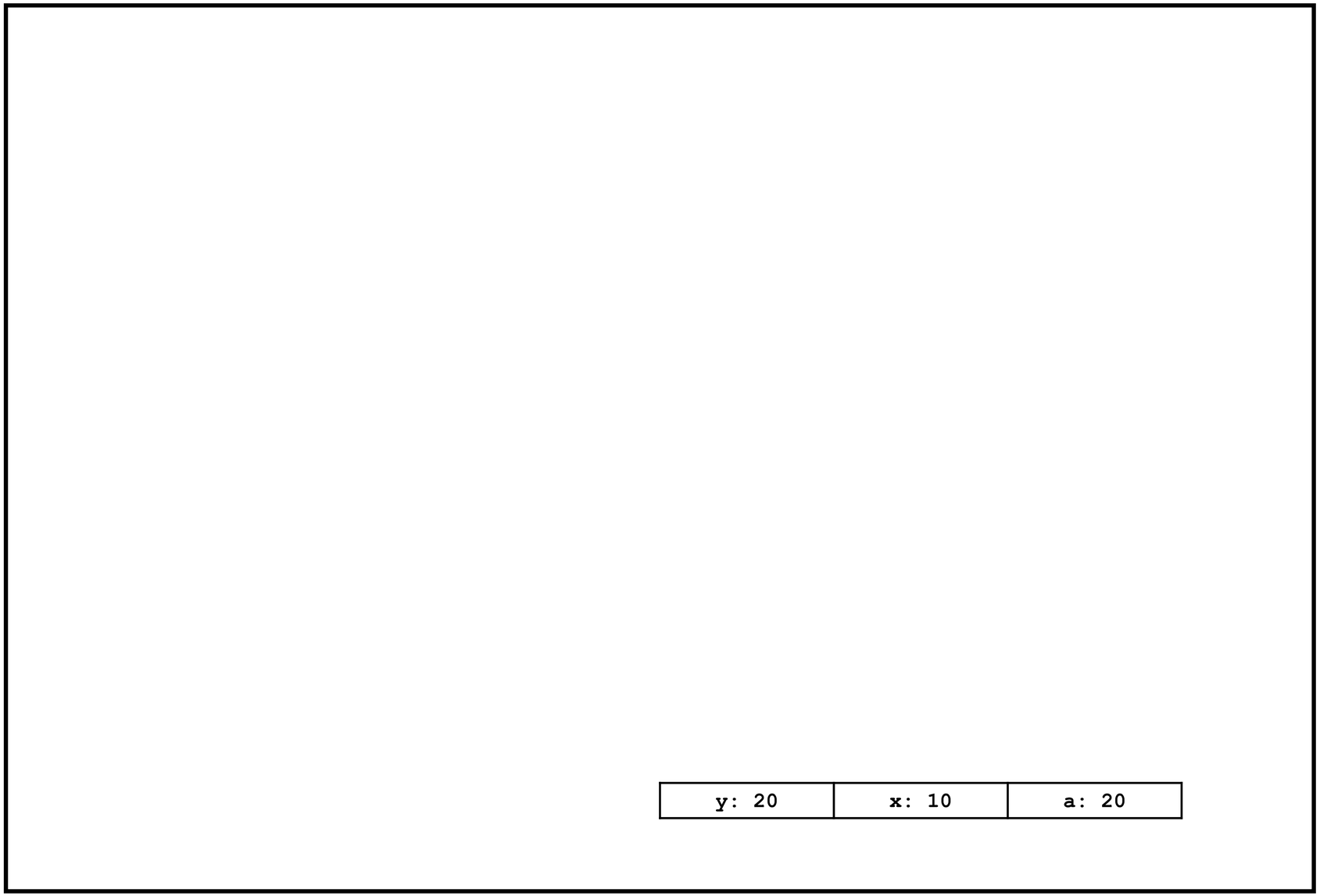
An Example

```
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

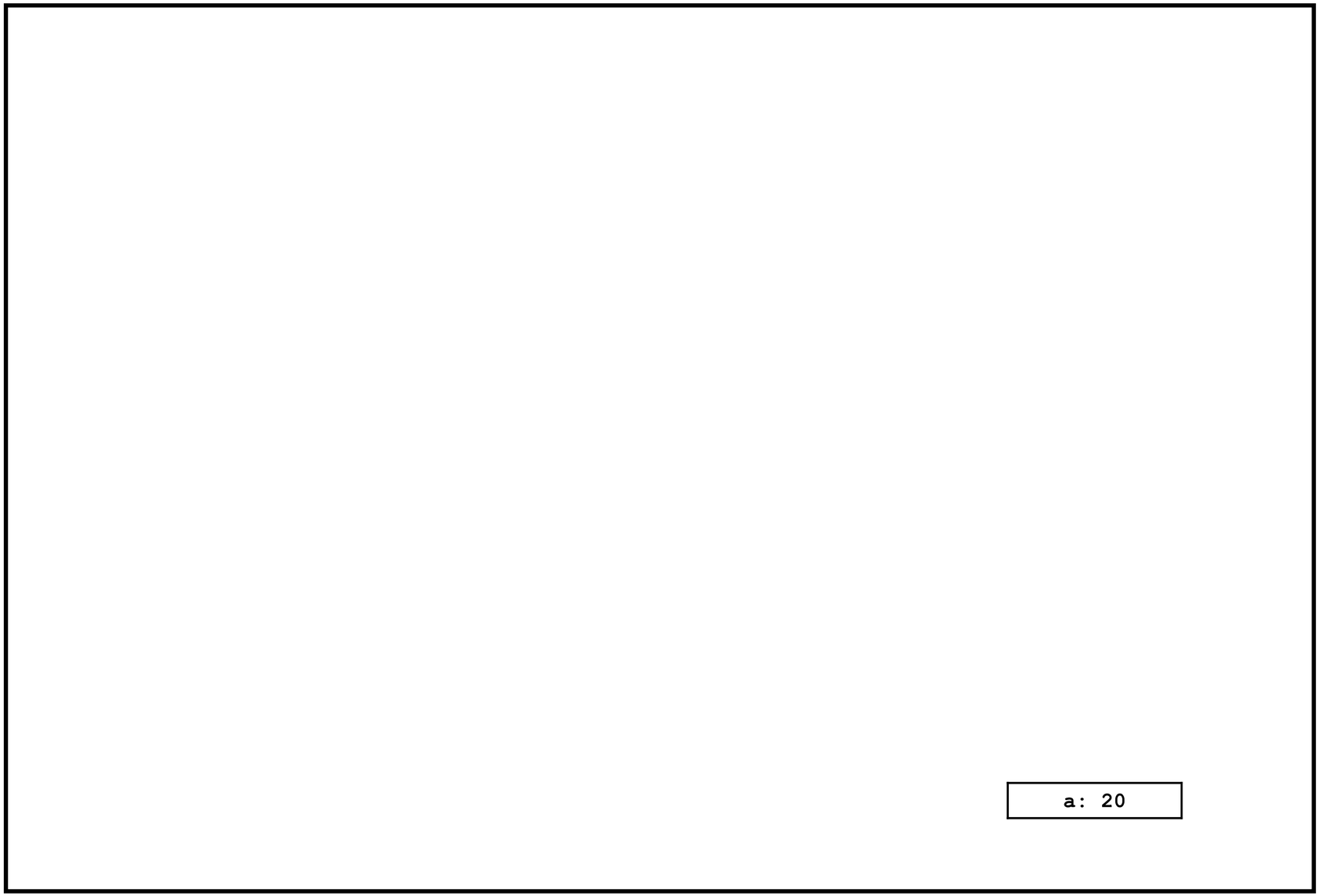
```
int f(int x)
{
        int y = x * 2;
        return y;
}

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

        return 0;
}
```



Variable Lifetime

An Example

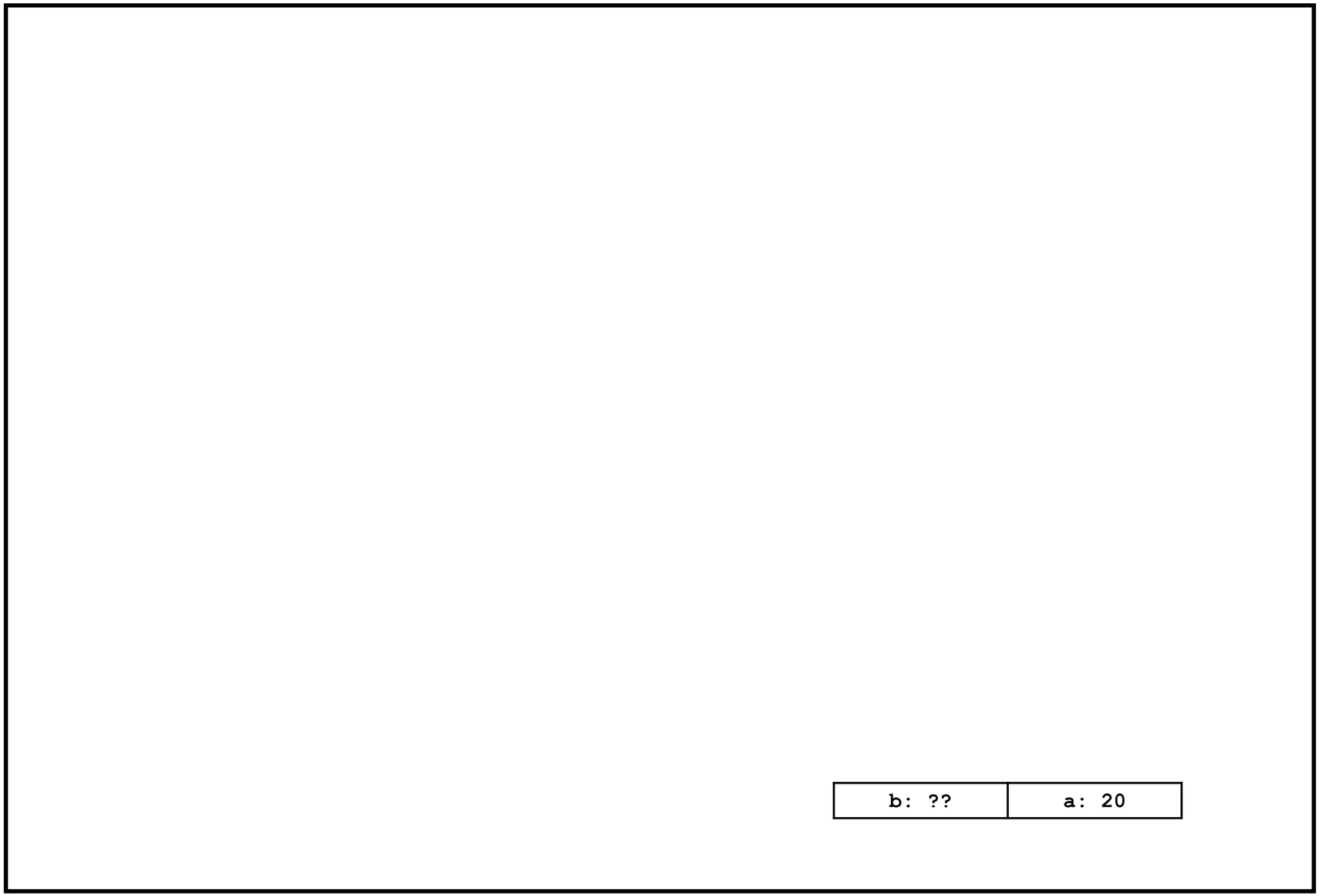
```
int f(int x)
{
        int y = x * 2;
        return y;
}

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

        return 0;
}
```



Variable Lifetime

An Example

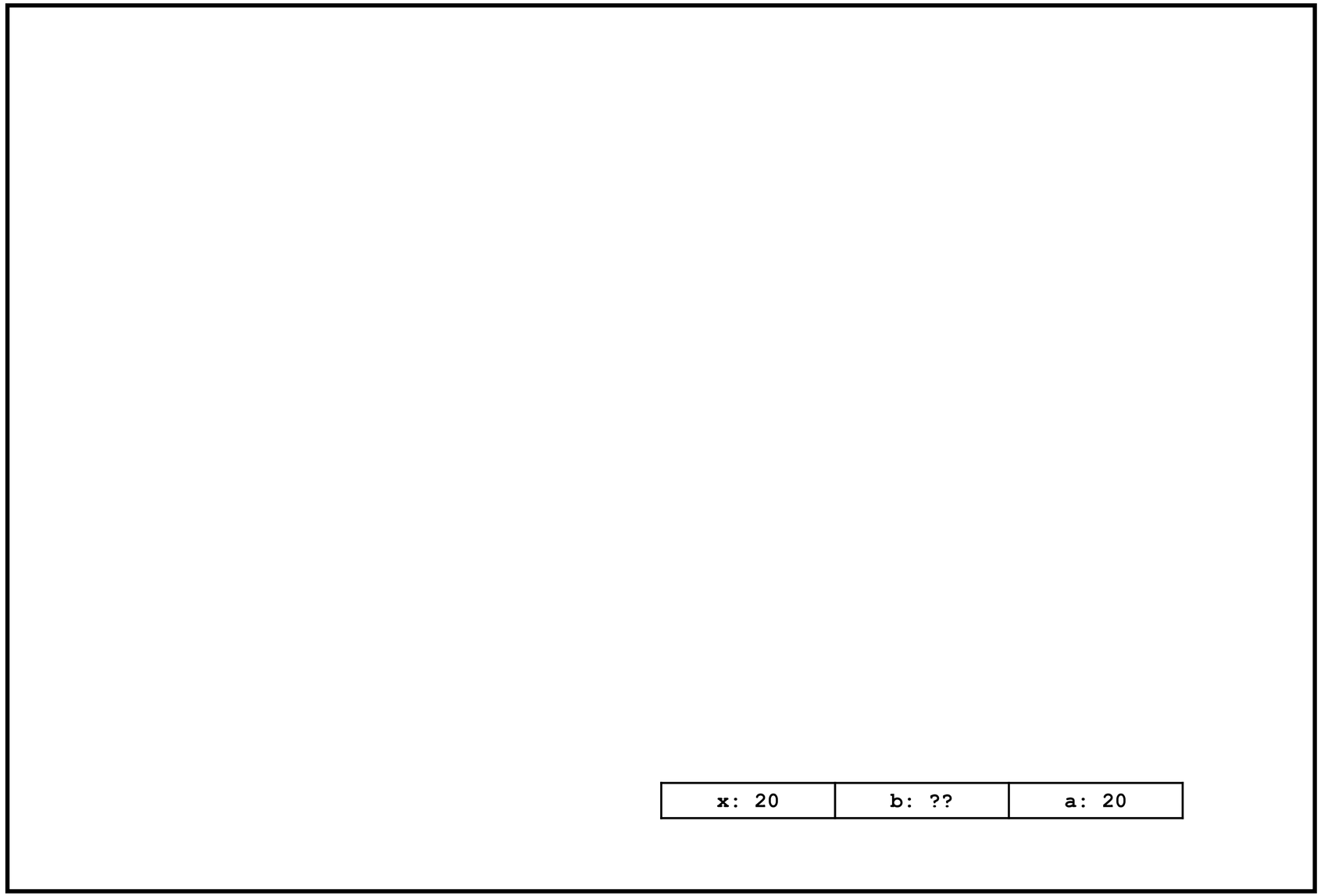
```
int f(int x)
{
        int y = x * 2;
        return y;
}

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

        return 0;
}
```



Variable Lifetime

An Example

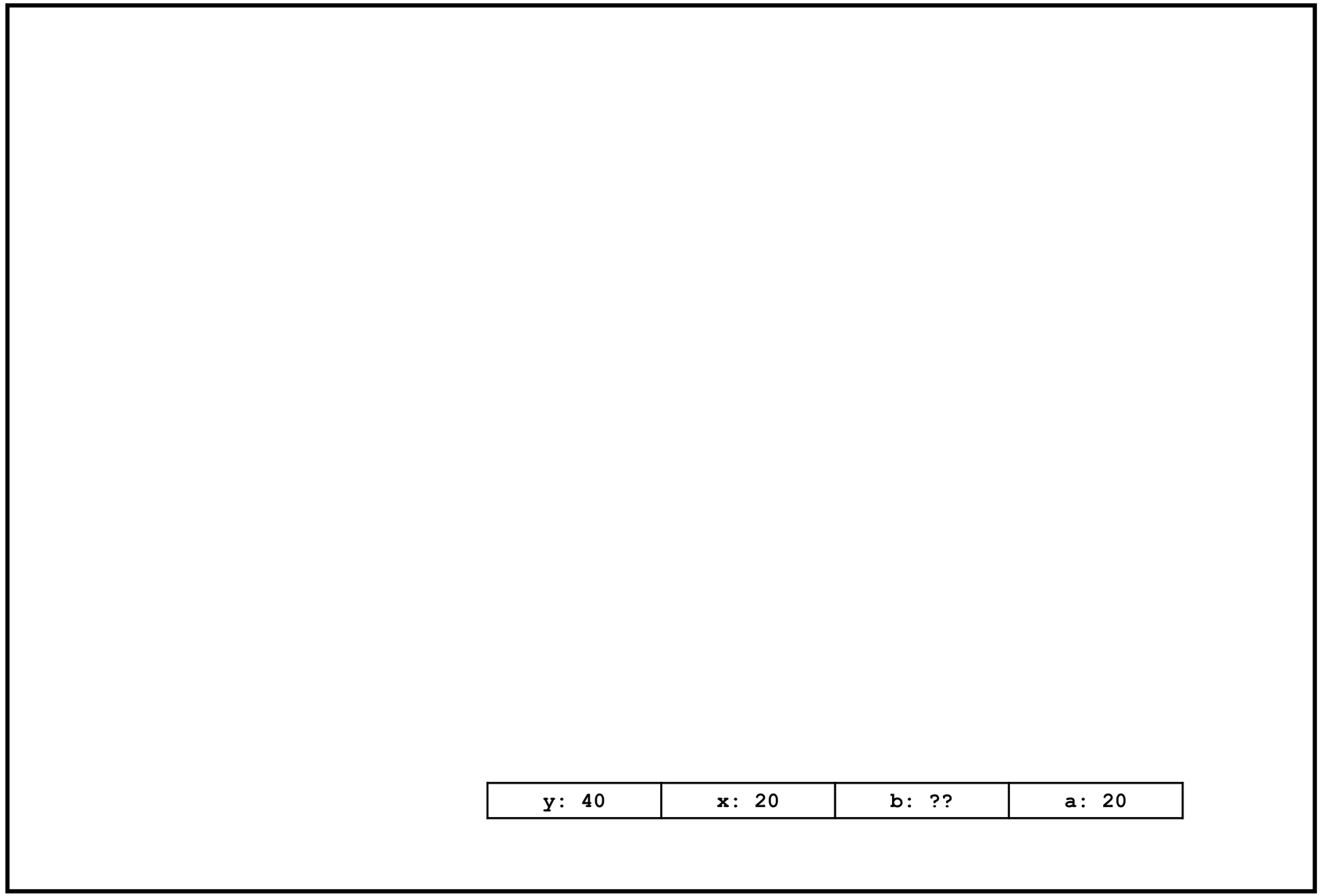
```
int f(int x)
{
        int y = x * 2;
        return y;
}

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

        return 0;
}
```



Variable Lifetime

An Example

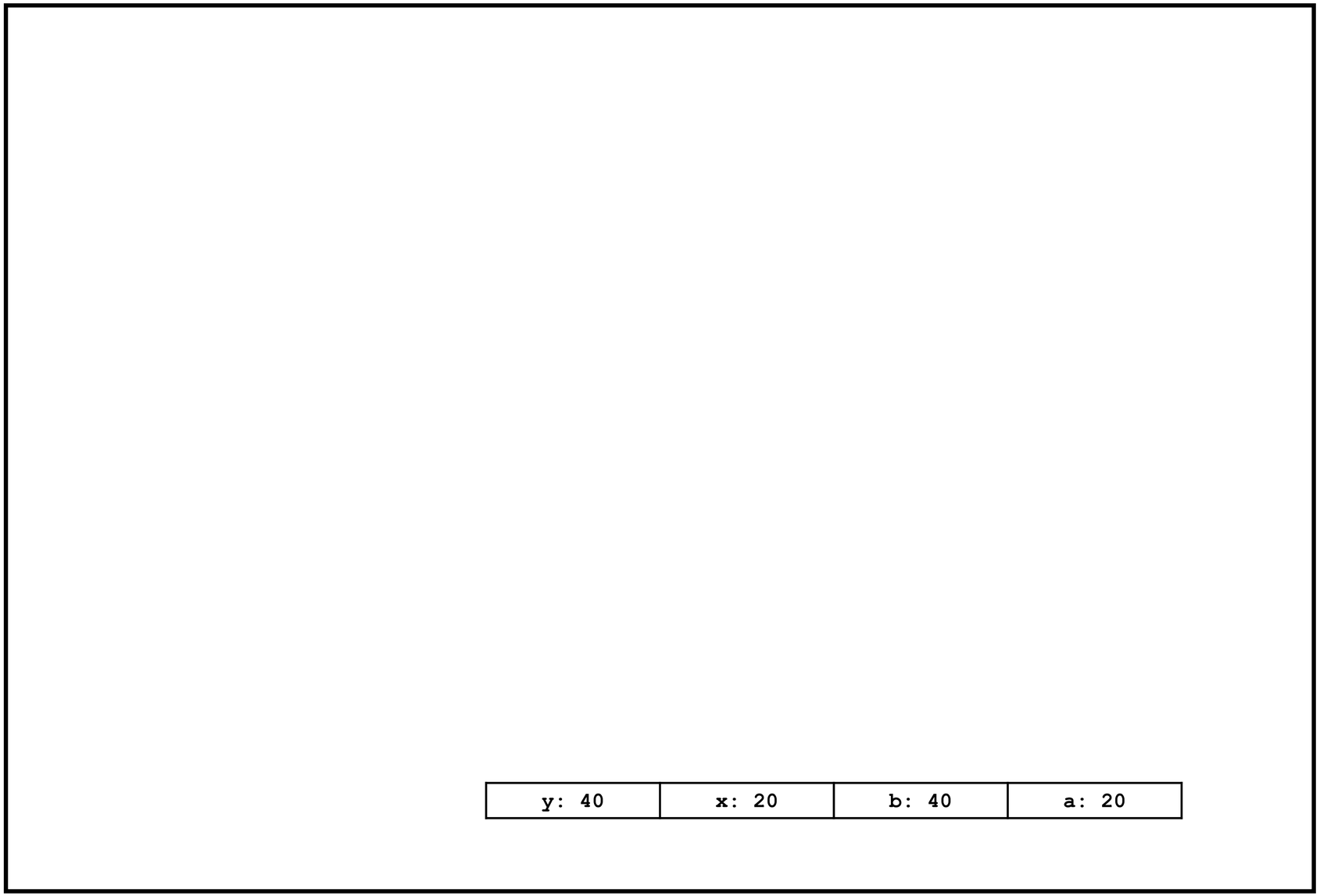
```
int f(int x)
{
        int y = x * 2;
        return y;
}

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

        return 0;
}
```



Variable Lifetime

An Example

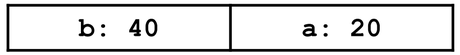
```
int f(int x)
{
        int y = x * 2;
        return y;
}

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

        return 0;
}
```



Variable Lifetime

An Example

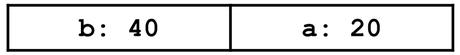
```
int f(int x)
{
        int y = x * 2;
        return y;
}

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

        return 0;
}
```



Variable Lifetime

An Example

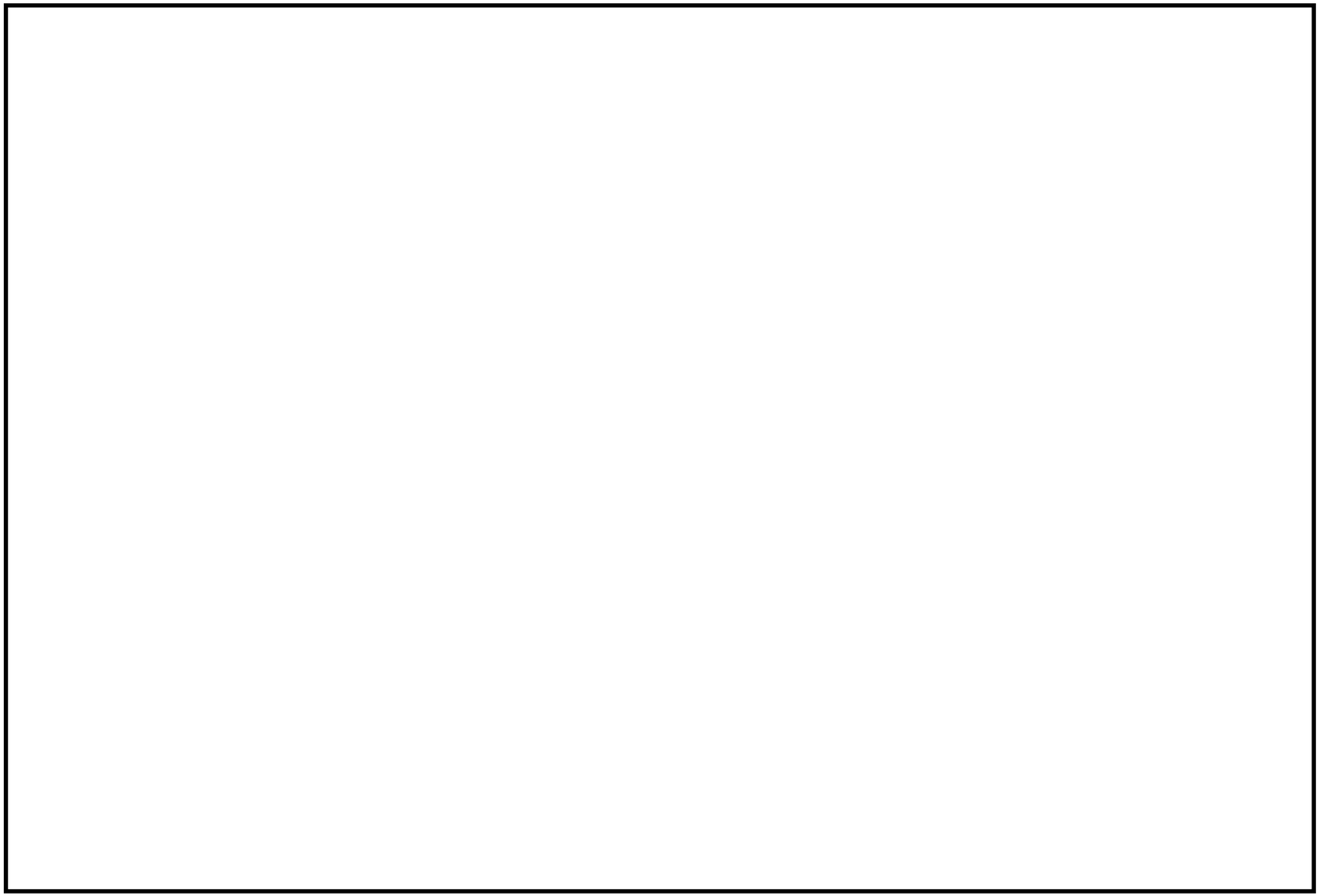
```
int f(int x)
{
        int y = x * 2;
        return y;
}

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

Variable Lifetime

An Example

- When a function returns, we can recycle the memory used by the variables declared inside the function.

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

Variable Lifetime

An Example

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

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

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)

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.

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.

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

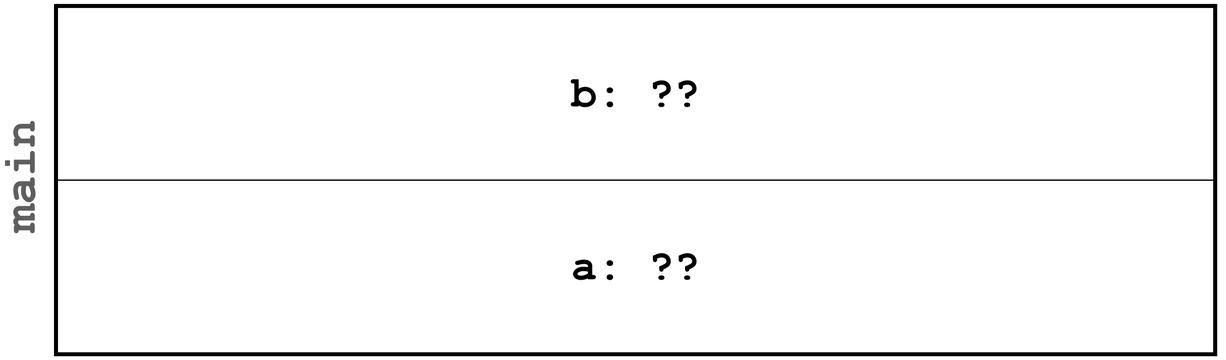
A more accurate picture

```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

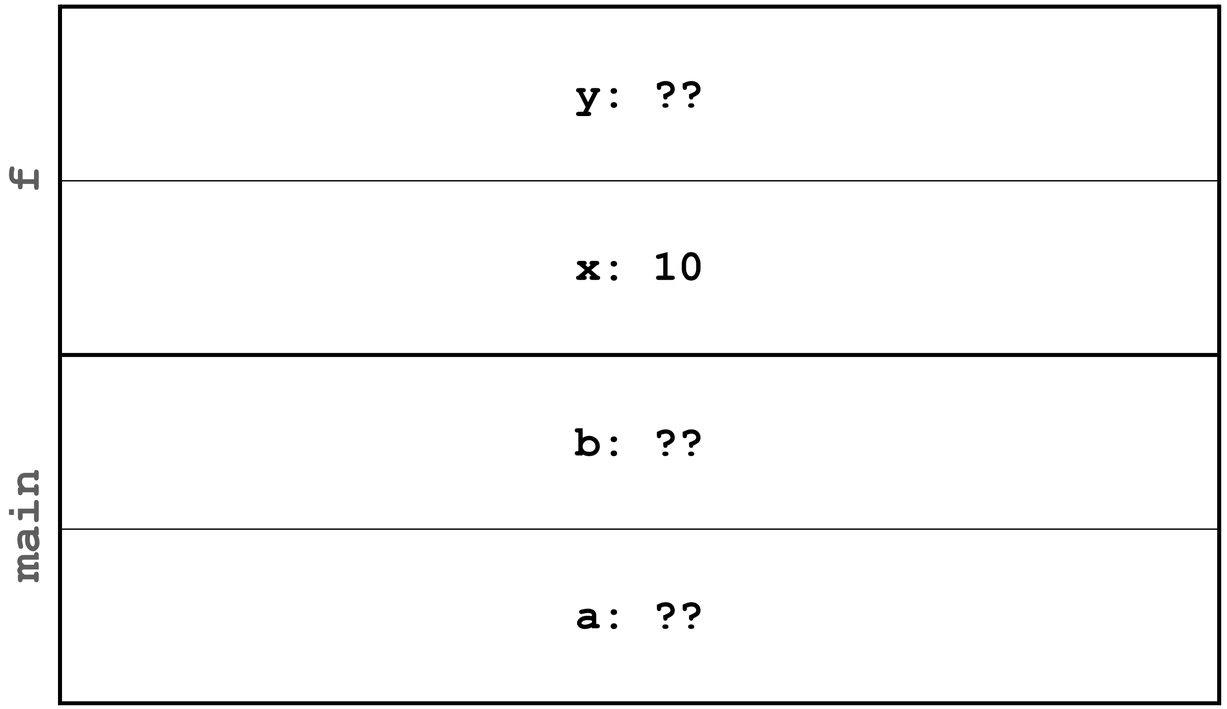
```c
int f(int x)
{
        int y = x * 2;
        return y;
}

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

        return 0;
}
```



Variable Lifetime

A more accurate picture

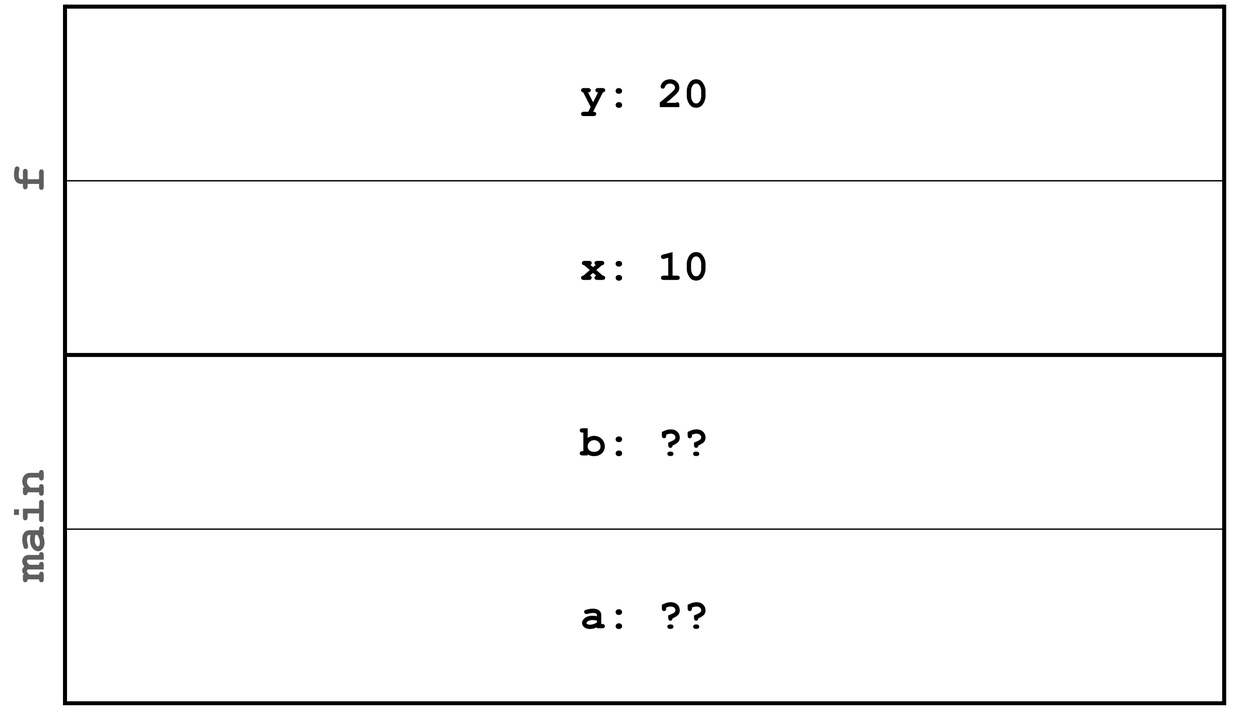
```
int f(int x)
{
        int y = x * 2;
        return y;
}

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

        return 0;
}
```



Variable Lifetime

A more accurate picture

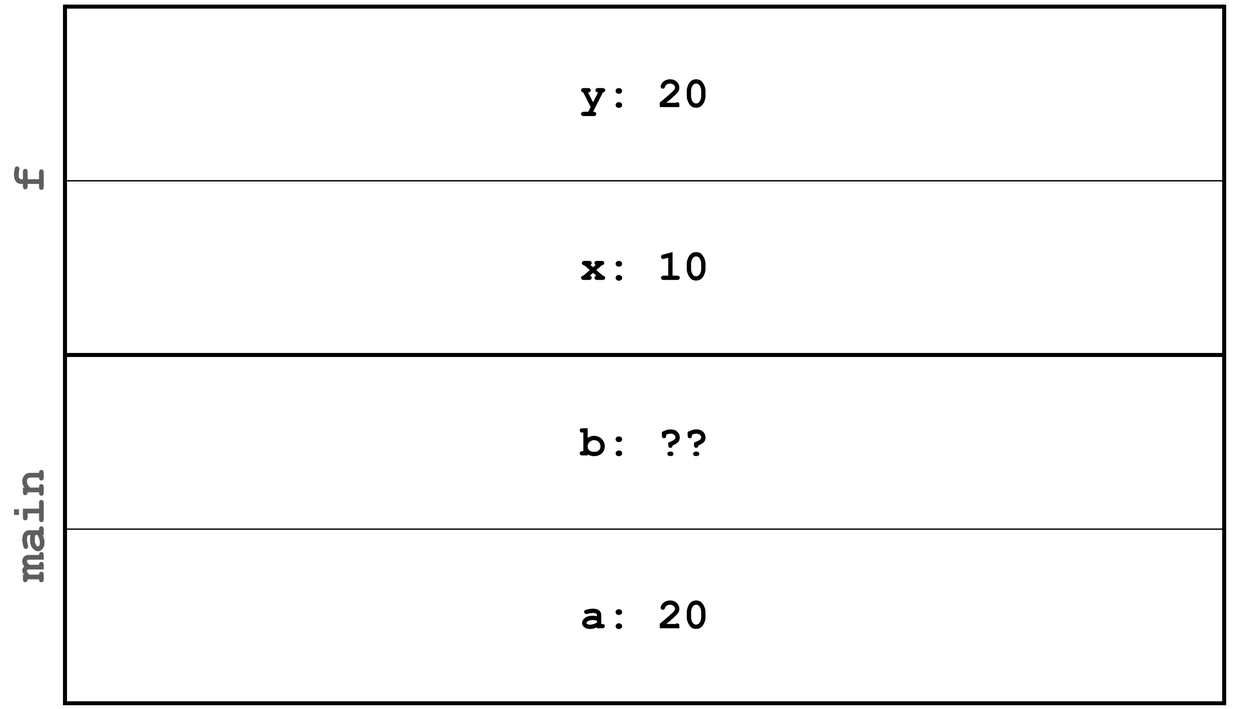
```
int f(int x)
{
        int y = x * 2;
        return y;
}

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

        return 0;
}
```



Variable Lifetime

A more accurate picture

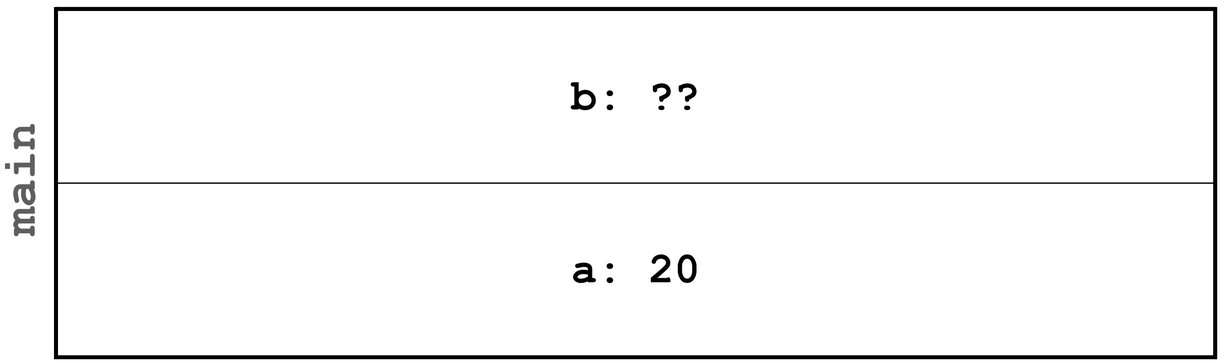
```
int f(int x)
{
        int y = x * 2;
        return y;
}

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

        return 0;
}
```



Variable Lifetime

A more accurate picture

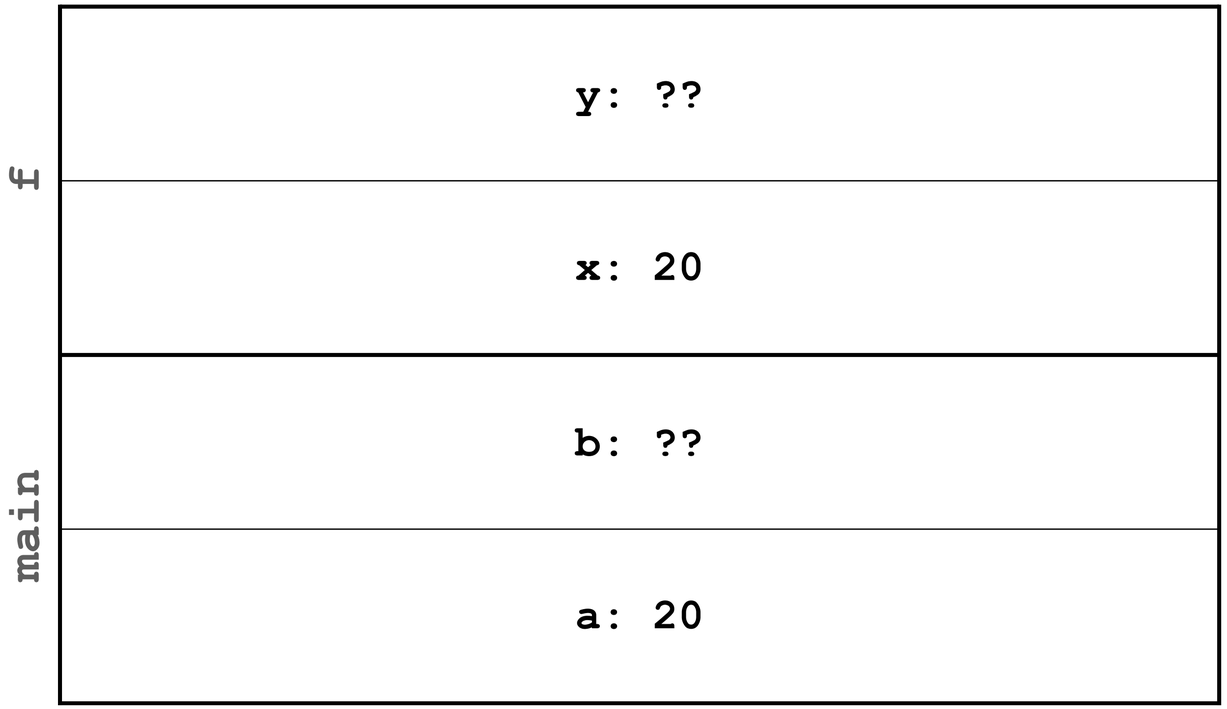
```c
int f(int x)
{
        int y = x * 2;
        return y;
}

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

        return 0;
}
```



Variable Lifetime

A more accurate picture

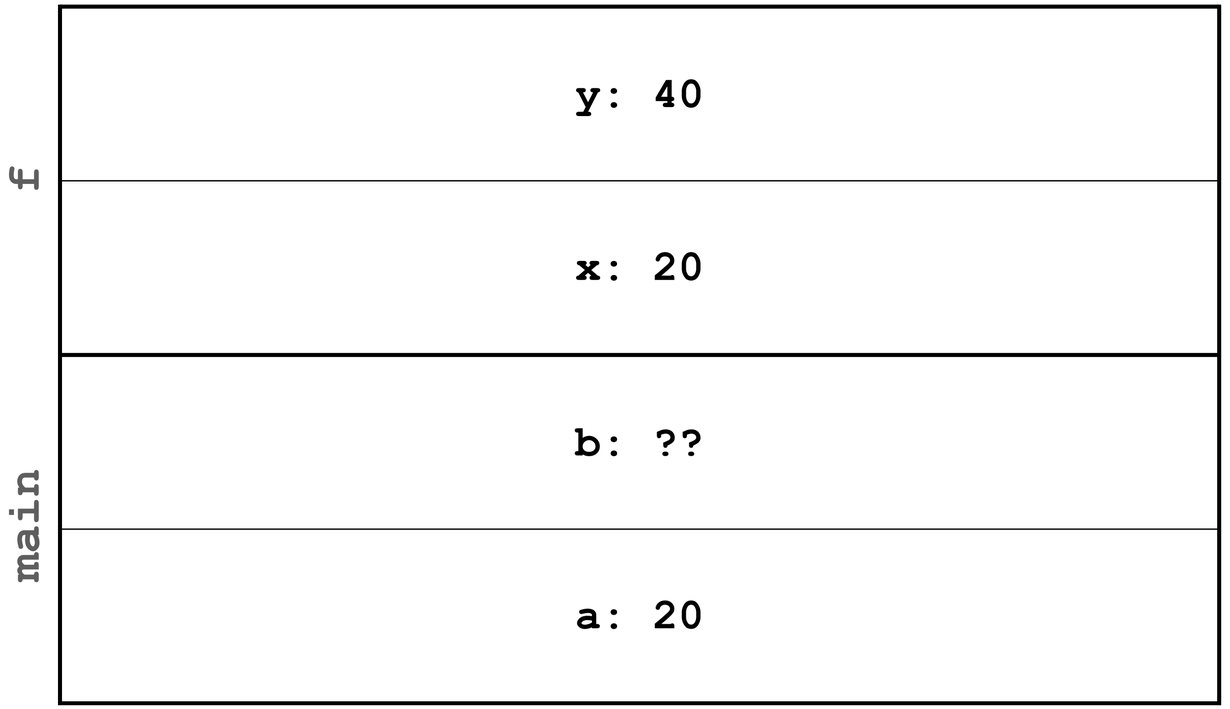
```
int f(int x)
{
        int y = x * 2;
        return y;
}

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

        return 0;
}
```



Variable Lifetime

A more accurate picture

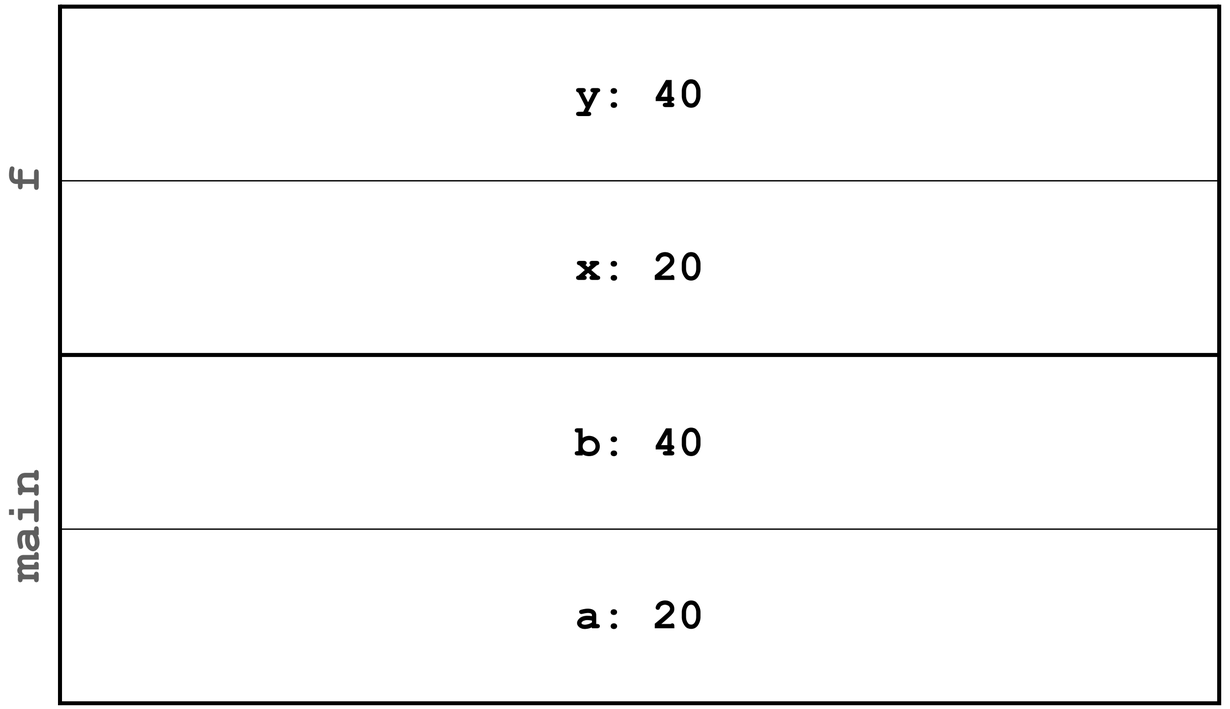
```
int f(int x)
{
        int y = x * 2;
        return y;
}

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

        return 0;
}
```



Variable Lifetime

A more accurate picture

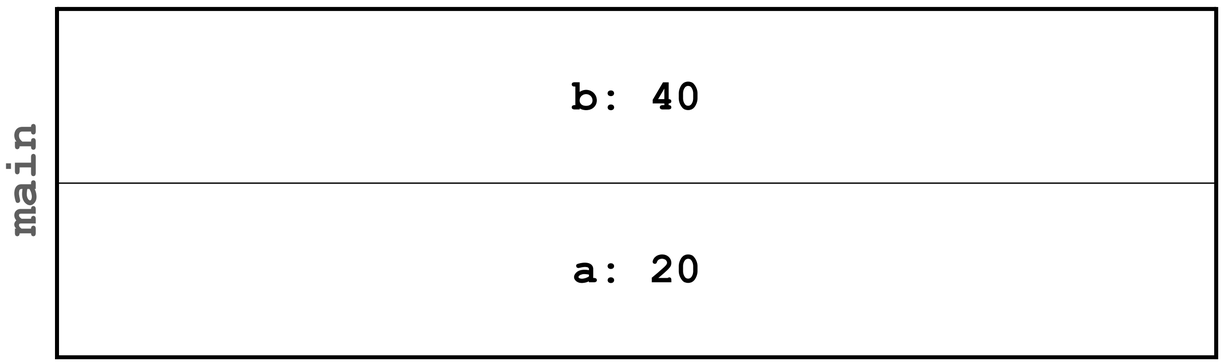
```
int f(int x)
{
        int y = x * 2;
        return y;
}

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;
}
```



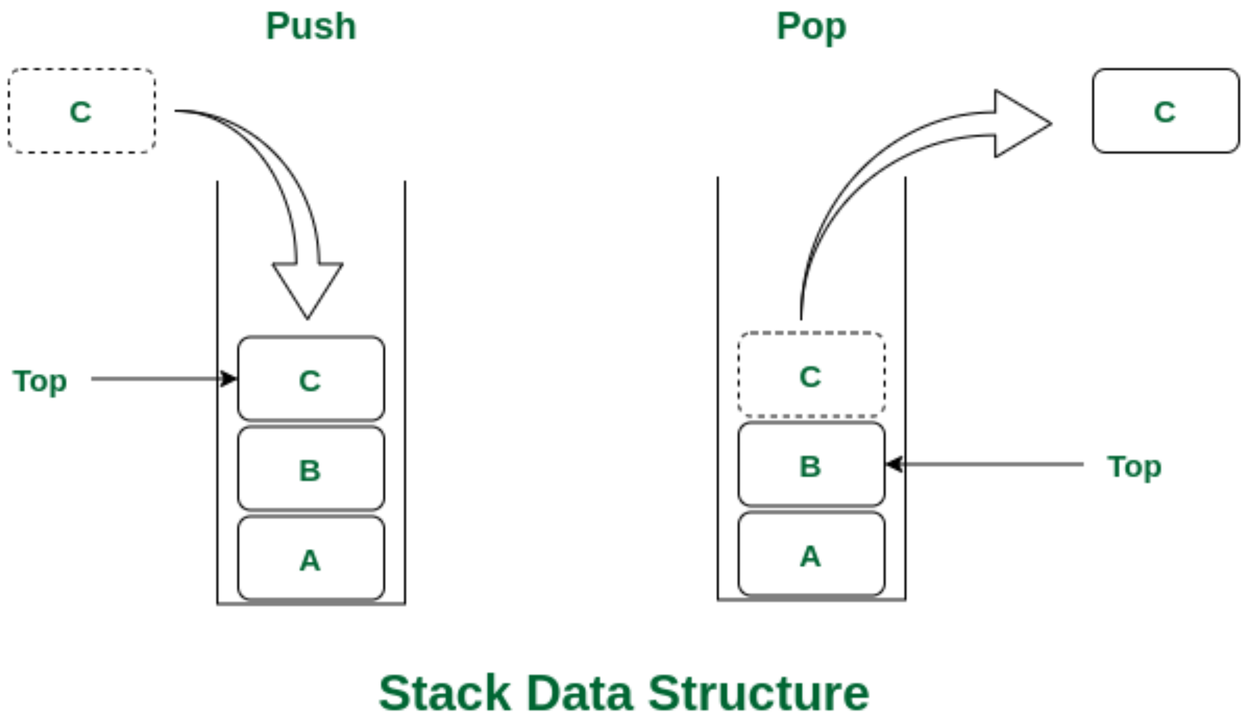
Variable Lifetime

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

Variable Lifetime

Call Stack

- C manages memory using a *stack* that lives in memory; internally, C remembers where the top of the stack is.

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.

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.

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.

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 Internal

Variable Lifetime

Call Stack Internal

- Because the sizes of variables are known, the size of a stack frame is also known.

Variable Lifetime

Call Stack Internal

- Because the sizes of variables are known, the size of a stack frame is also known.
- The runtime system keeps track of the *address* of the top of the stack (*sp*).

Variable Lifetime

Call Stack Internal

- Because the sizes of variables are known, the size of a stack frame is also known.
- The runtime system keeps track of the *address* of the top of the stack (*sp*).
- Pushing a frame is done by subtracting the size of the frame from *sp*.

Variable Lifetime

Call Stack Internal

- Because the sizes of variables are known, the size of a stack frame is also known.
- The runtime system keeps track of the *address* of the top of the stack (*sp*).
- Pushing a frame is done by subtracting the size of the frame from *sp*.
- Popping a frame is done by adding the size back to *sp*.

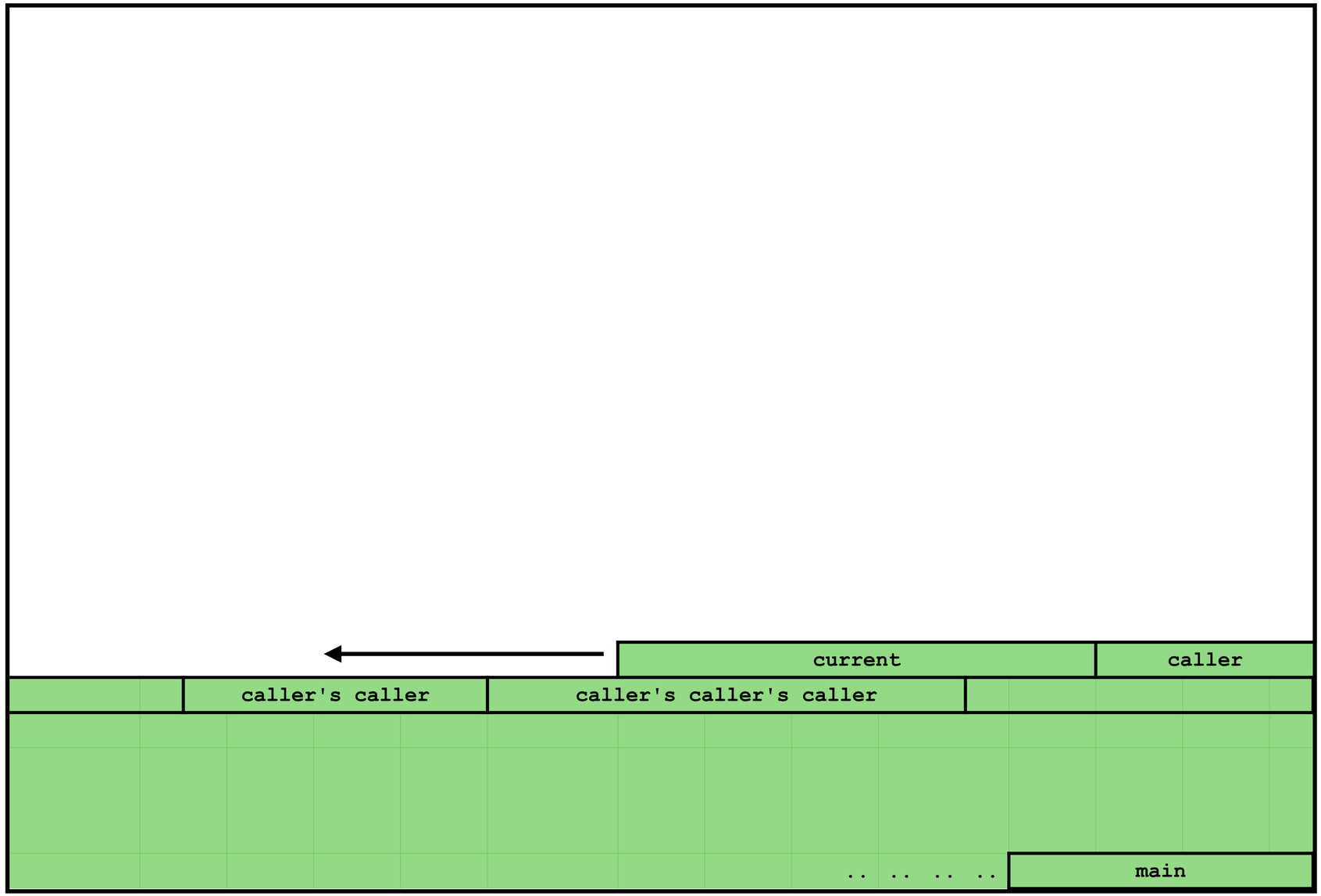
Variable Lifetime

Call Stack Internal

- Because the sizes of variables are known, the size of a stack frame is also known.
- The runtime system keeps track of the *address* of the top of the stack (*sp*).
- Pushing a frame is done by subtracting the size of the frame from *sp*.
- Popping a frame is done by adding the size back to *sp*.
- Local variables' addresses are calculated by *sp* + *offset.*

Variable Lifetime

Call Stack



Variable Lifetime

Call Stack

Variable Lifetime

Call Stack

- Stack has a fixed size determined by the OS.

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.

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

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

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

Variable Lifetime

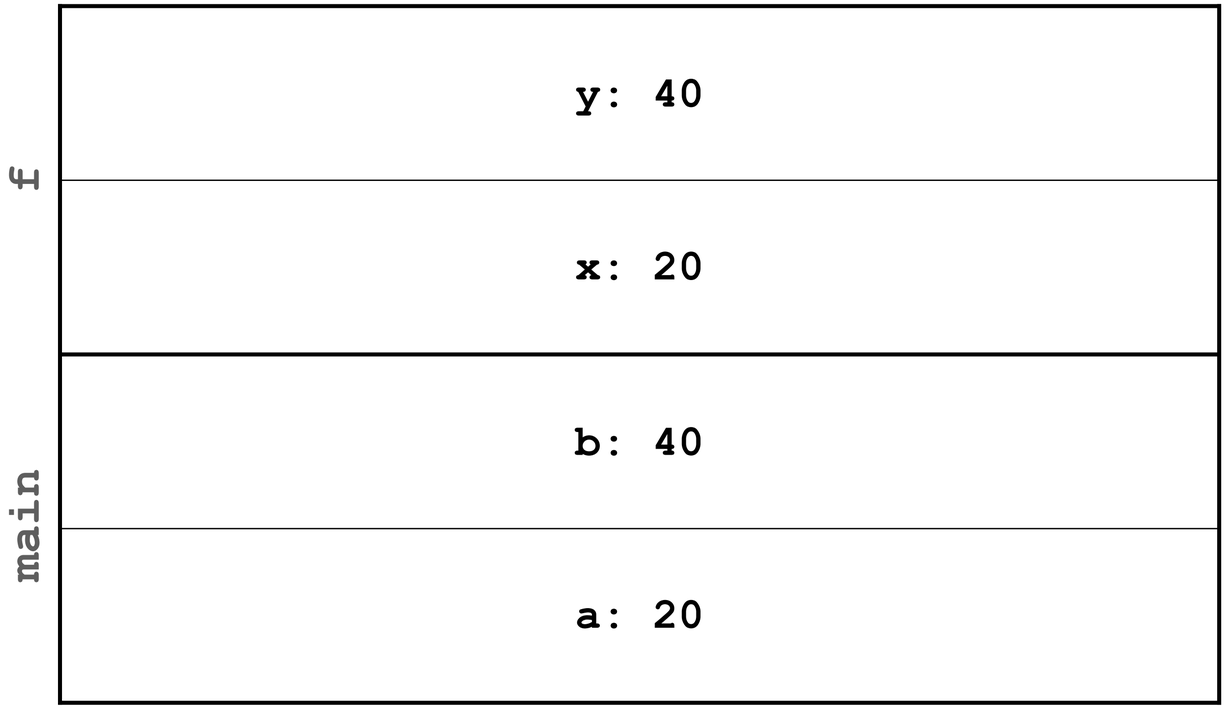
Call Stack

- 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;
}
```

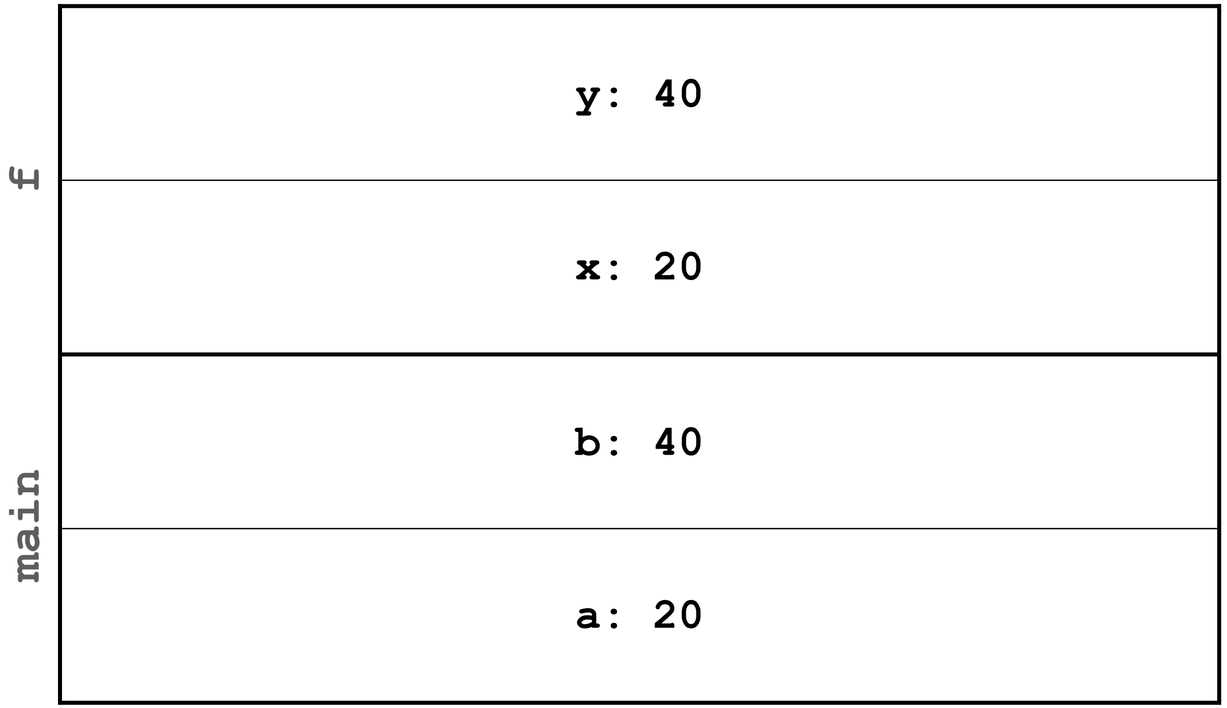


Variable Lifetime

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.

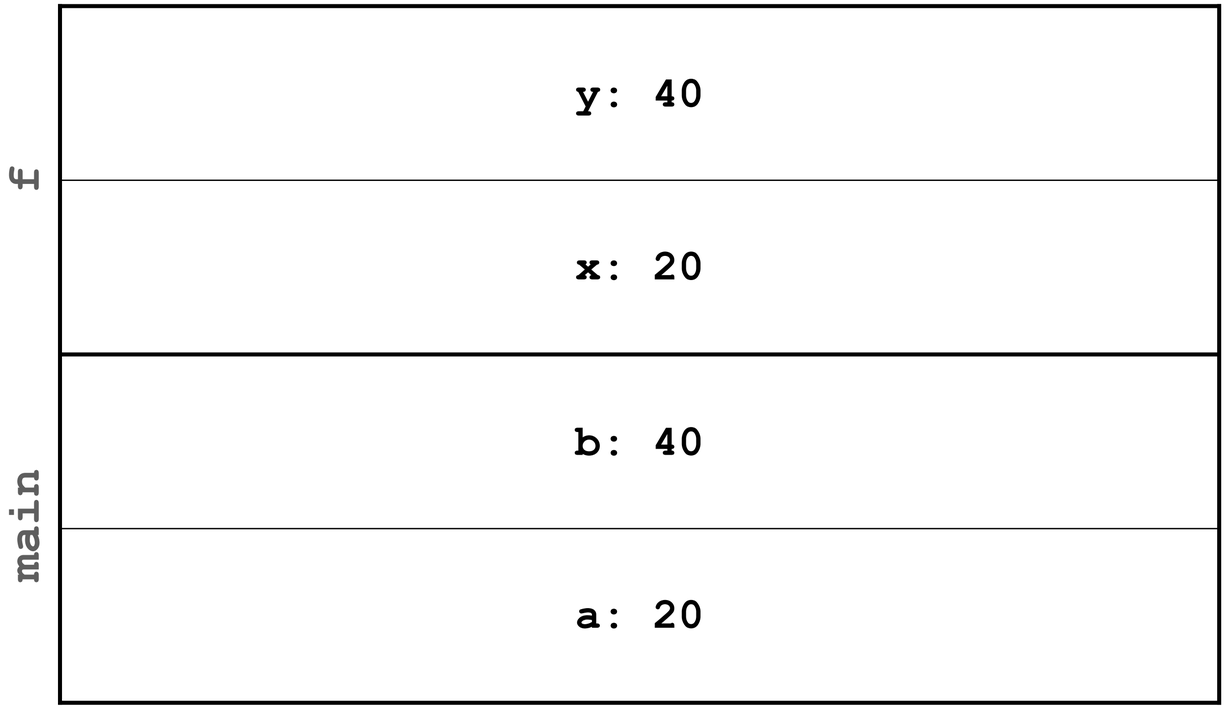


Variable Lifetime

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

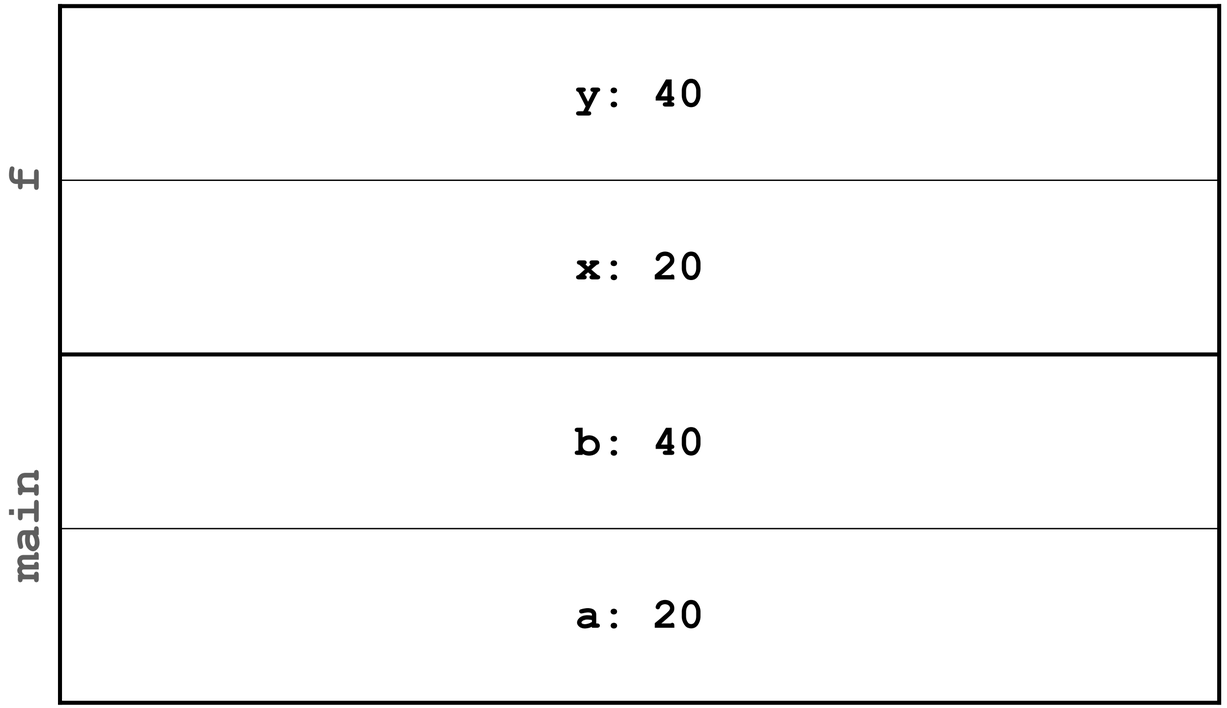


Variable Lifetime

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?

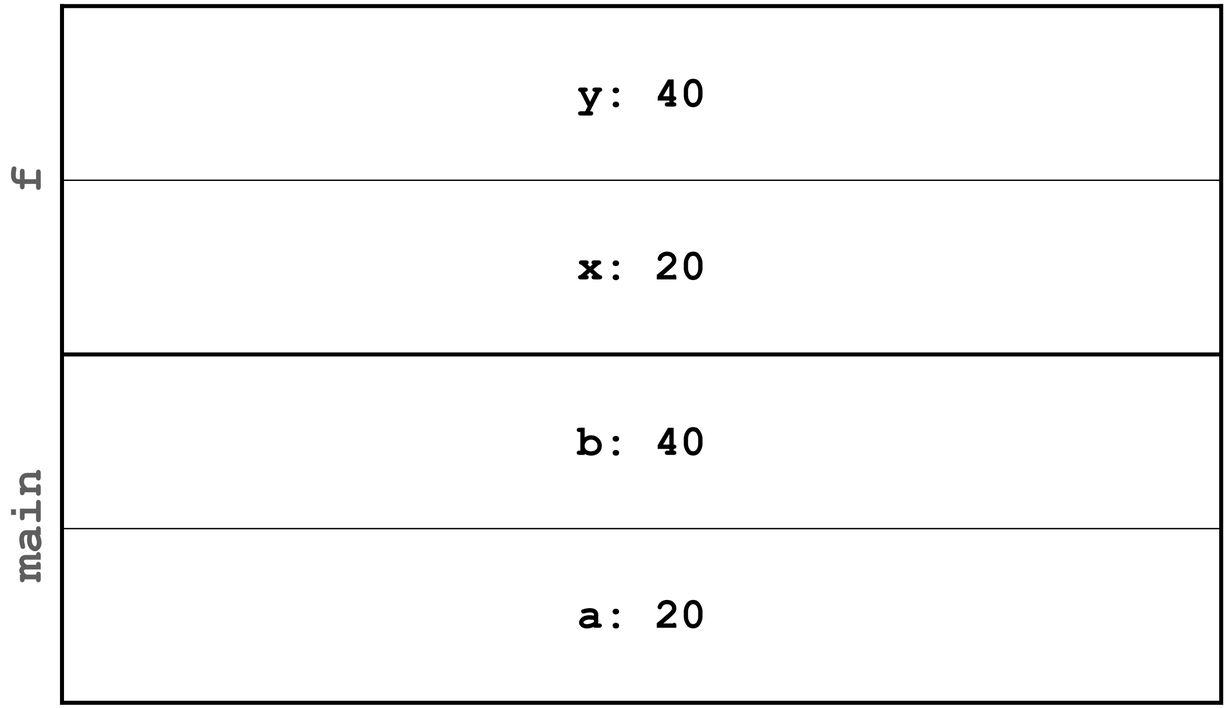


Variable Lifetime

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;
}
```

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?

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

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
```

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

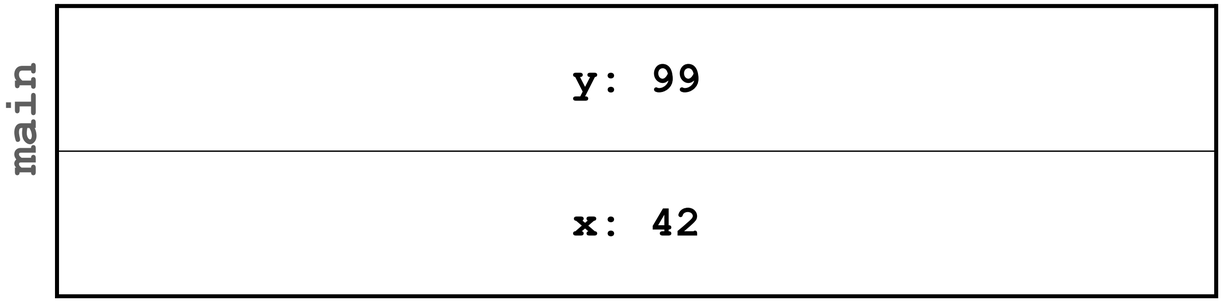
```
byron@MacBook-Pro solutions % ./swap
42 99
```

But why?

Pass by Reference

swap

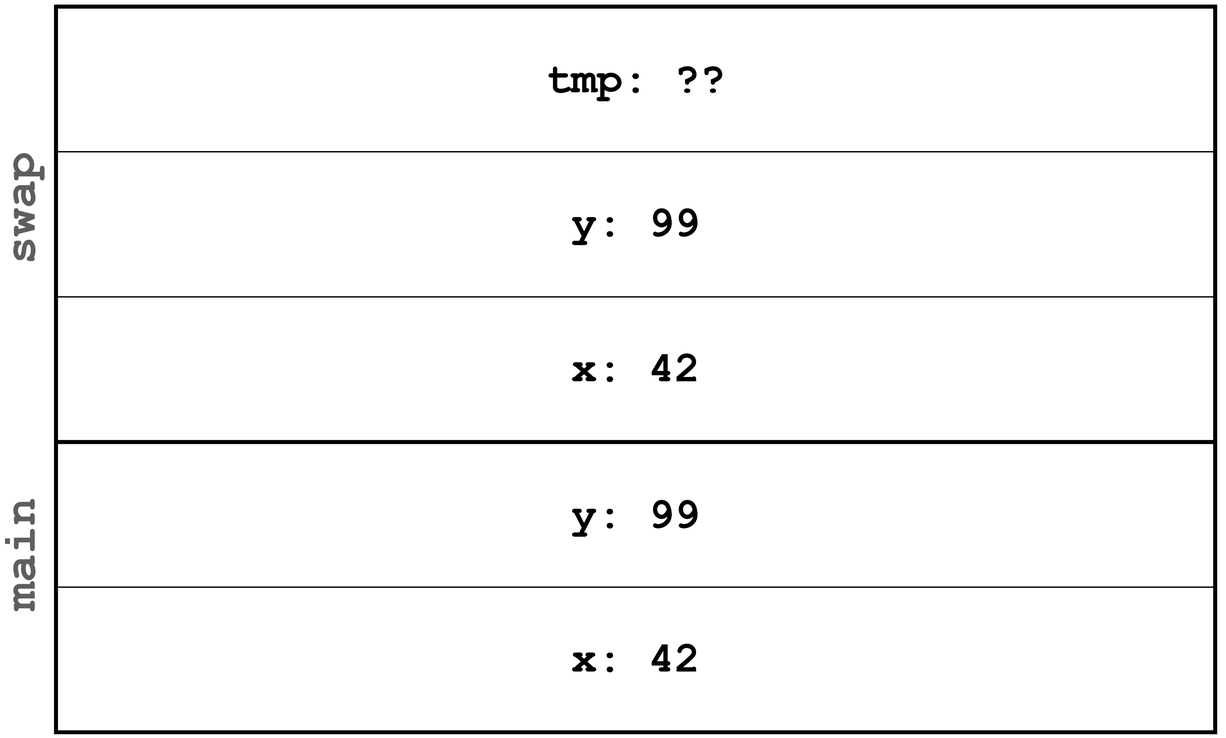
```c
#include <stdio.h>

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

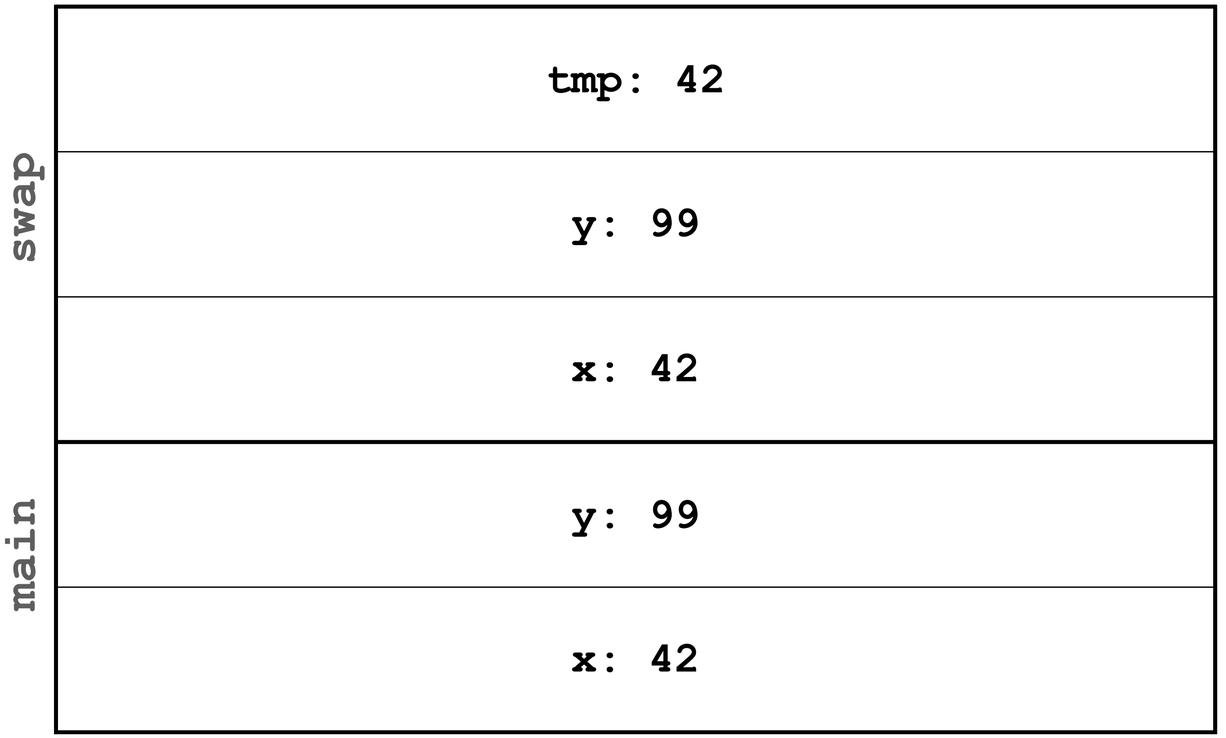
```c
#include <stdio.h>

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

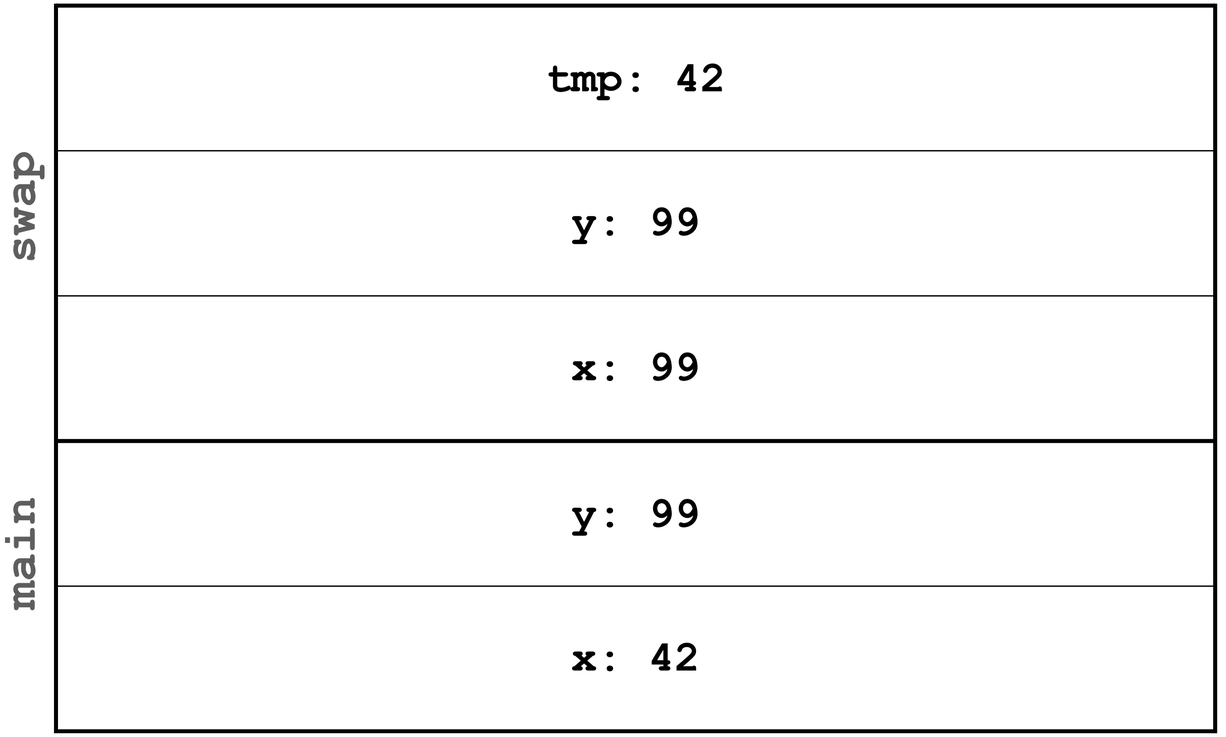
```c
#include <stdio.h>

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

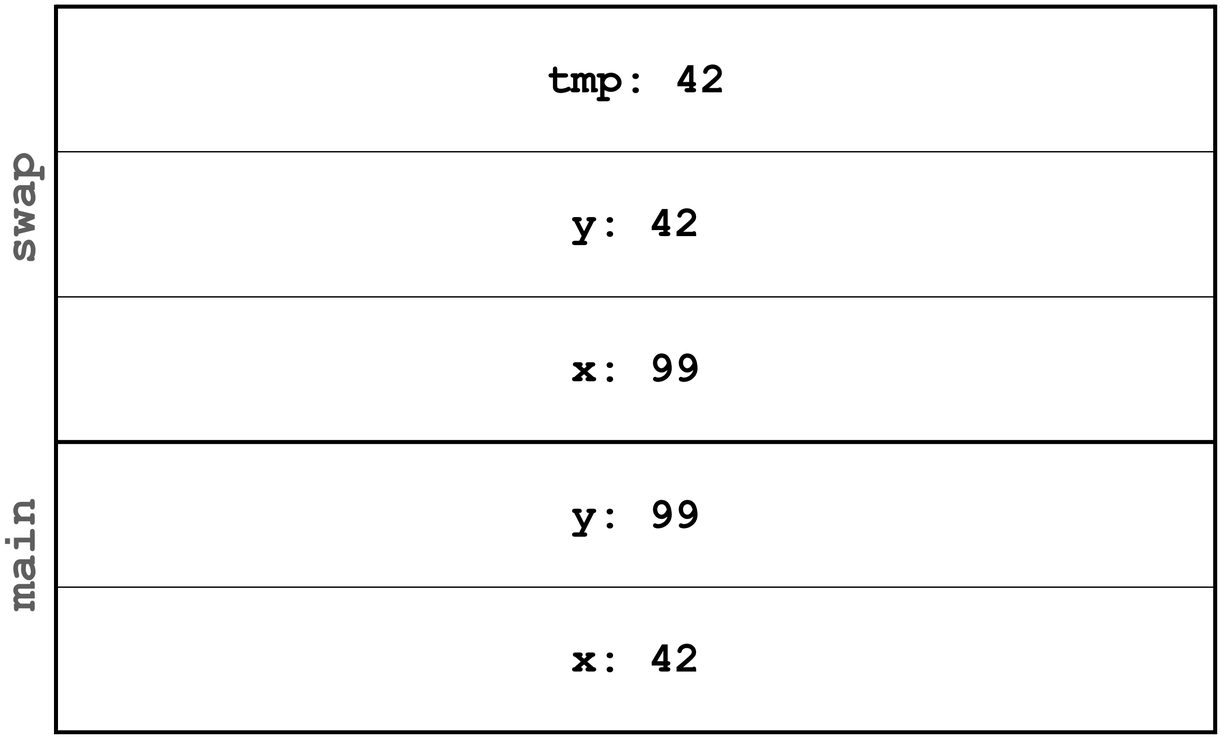
```c
#include <stdio.h>

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

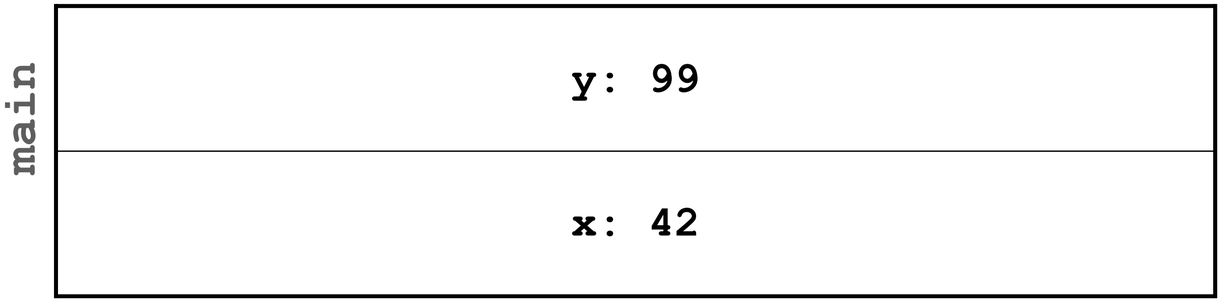
```c
#include <stdio.h>

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;
}
```

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?

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

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
```

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;
}
```

Does it work?

No

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

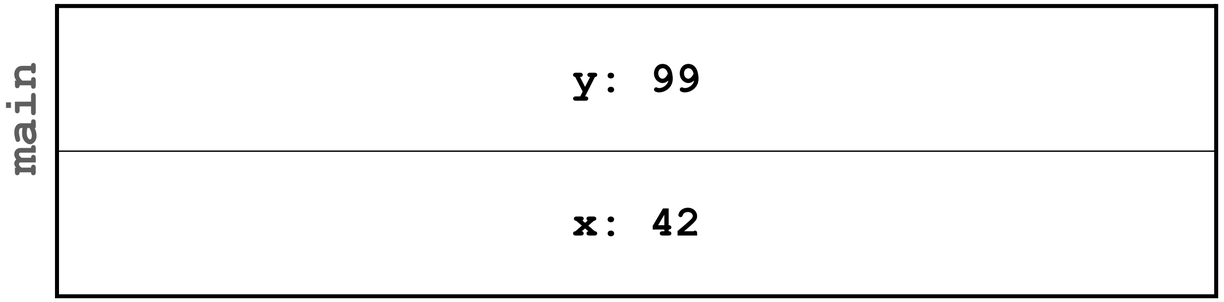
But why?

Function arguments are passed by *values*

Pass by Reference

swap

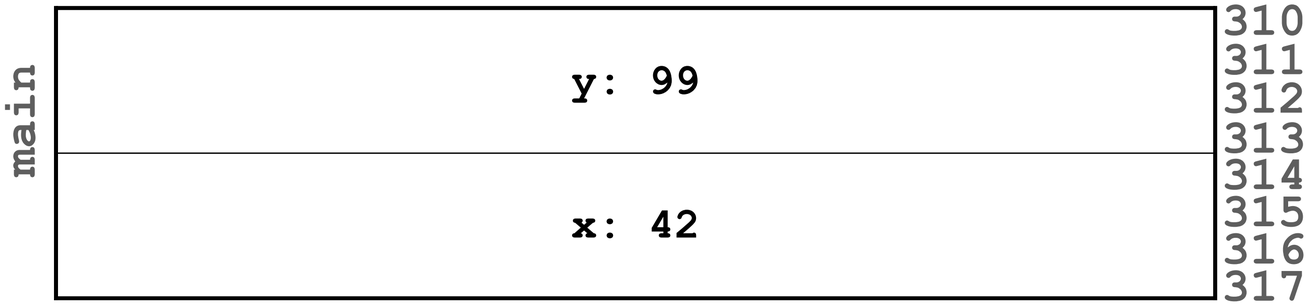
```c
#include <stdio.h>

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

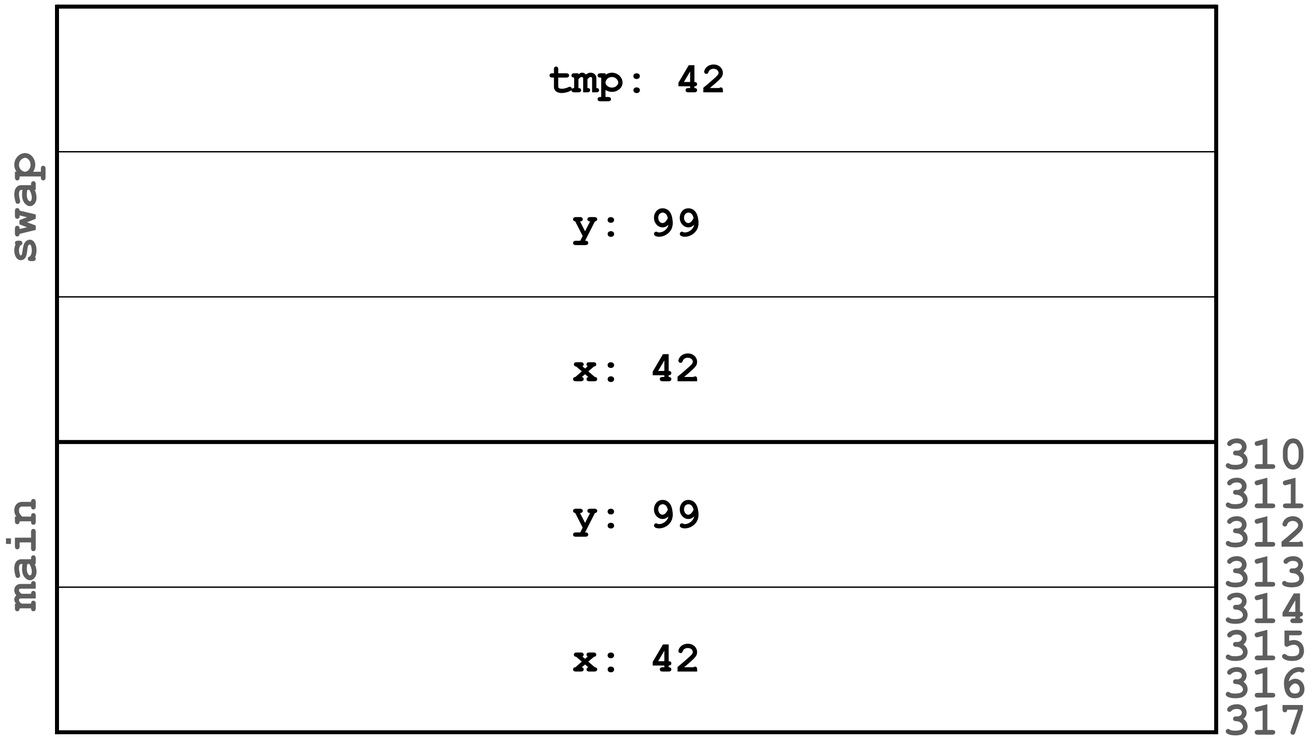
```c
#include <stdio.h>

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

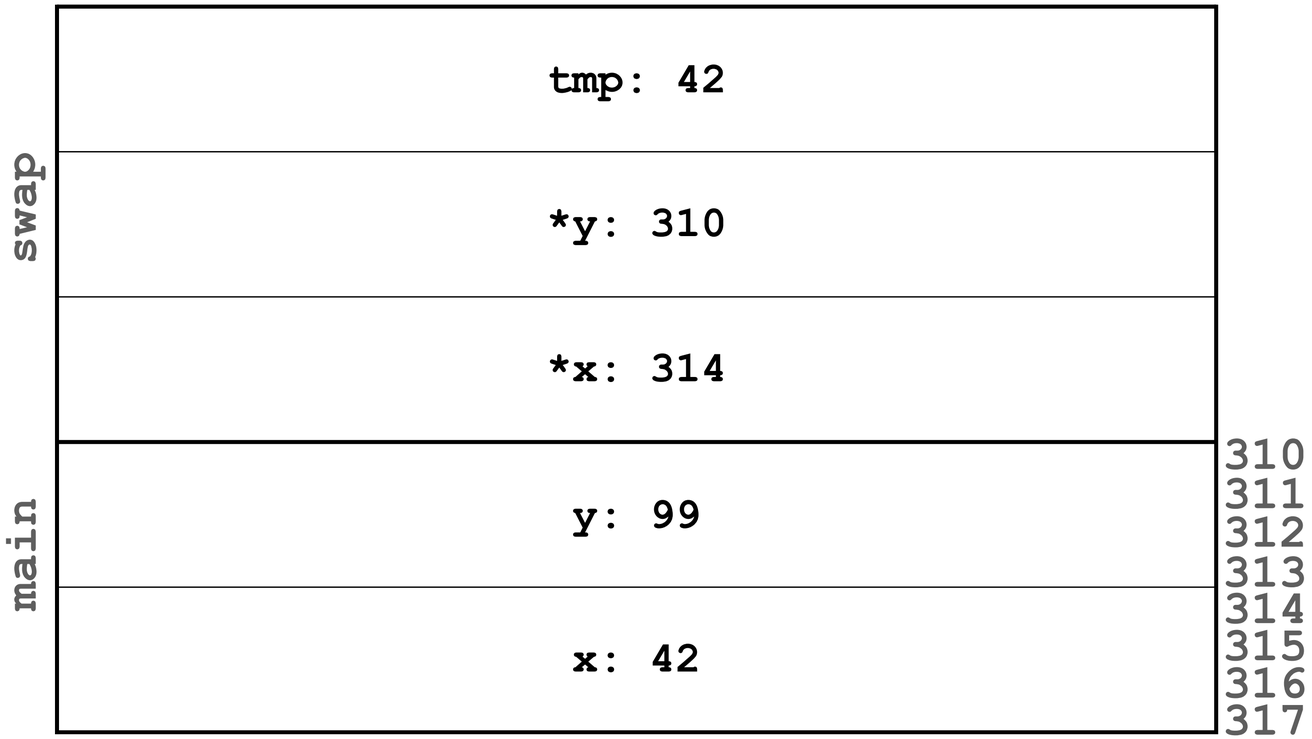
```c
#include <stdio.h>

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

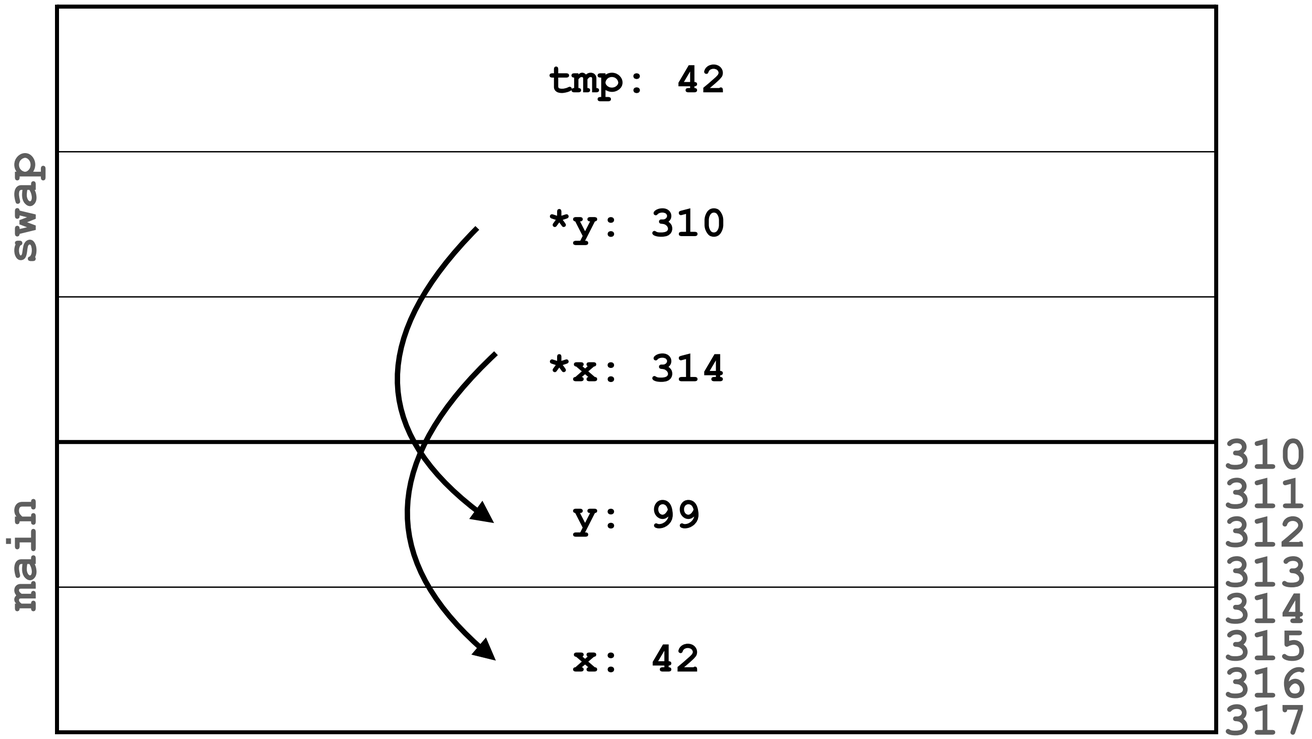
```c
#include <stdio.h>

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

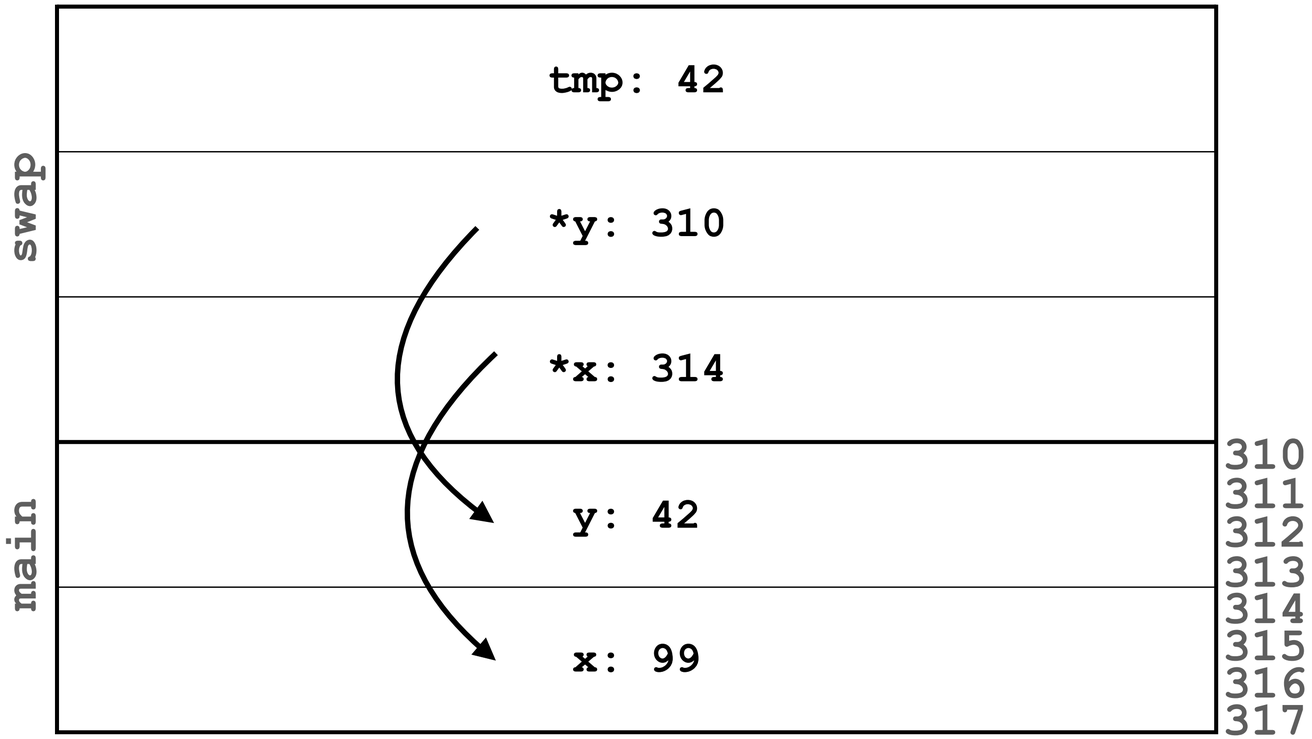
```c
#include <stdio.h>

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

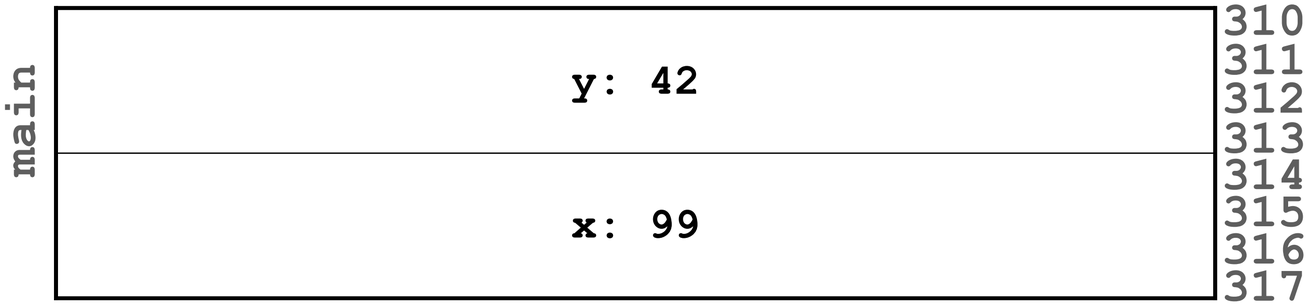
```c
#include <stdio.h>

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;
}
```



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;
}
```

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.

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;
}
```

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.

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).

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;
}
```

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)

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`:

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)

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`

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;
}
```

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`.

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...

Pointers

Obtaining Addresses

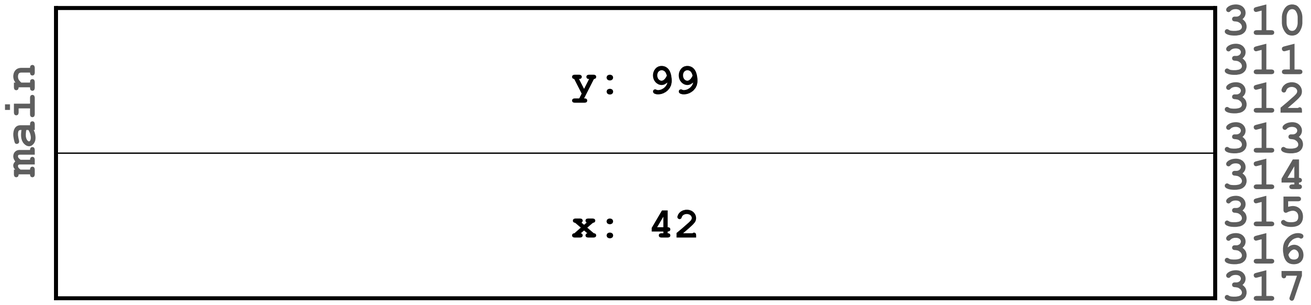
```
int main(void)
{
        int x = 42;
        int y = 99;

        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

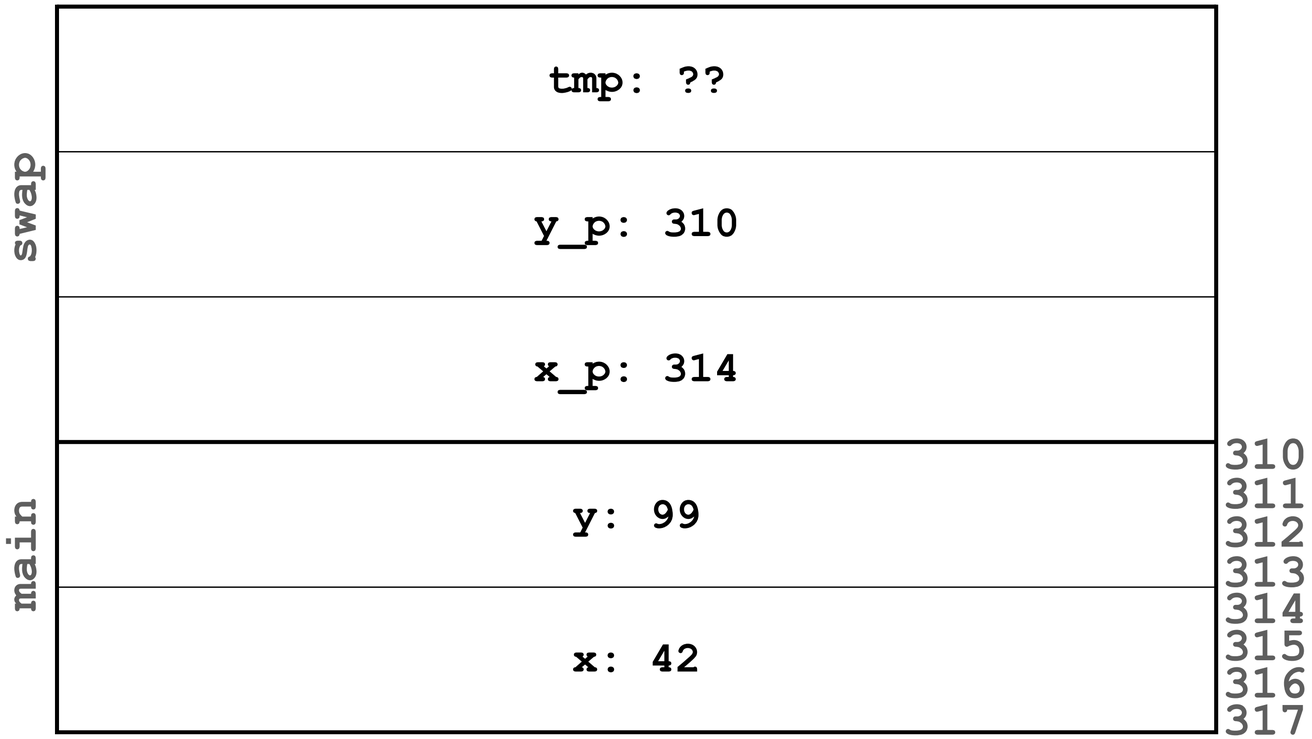
```c
#include <stdio.h>

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

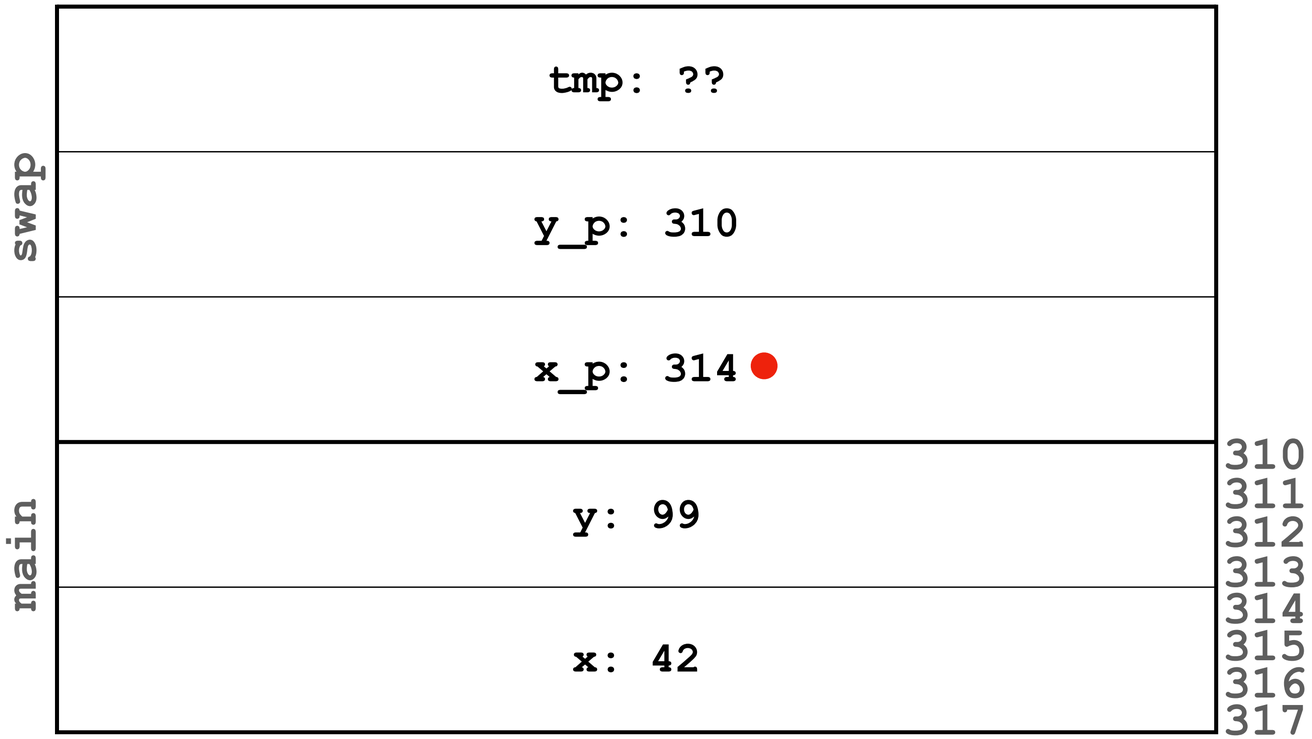
```c
#include <stdio.h>

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

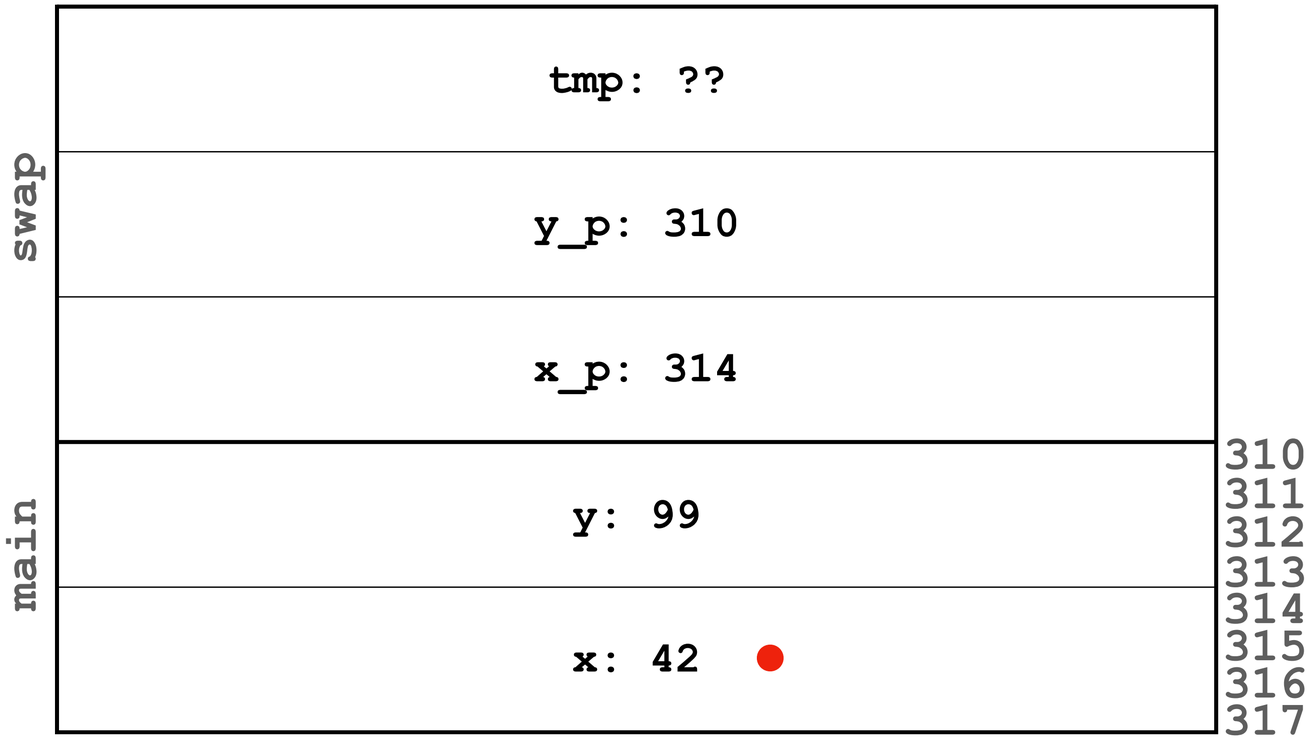
```c
#include <stdio.h>

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

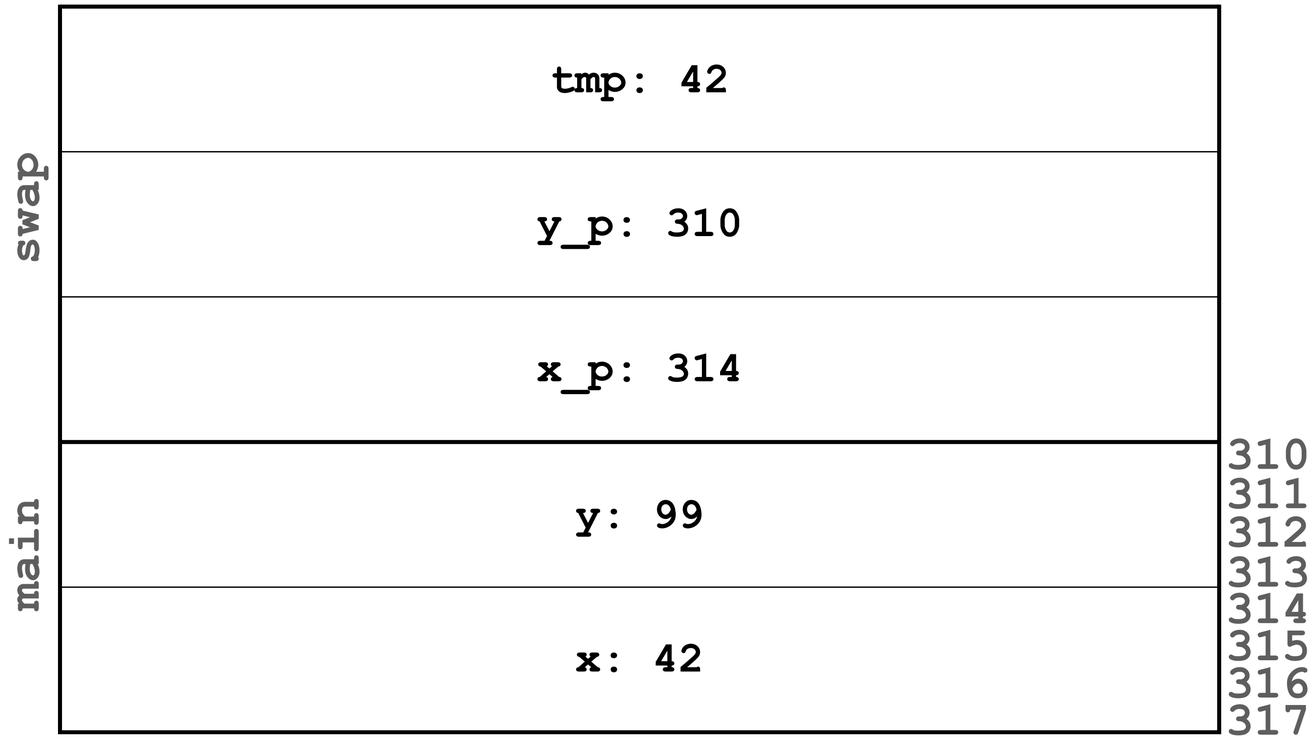
```c
#include <stdio.h>

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

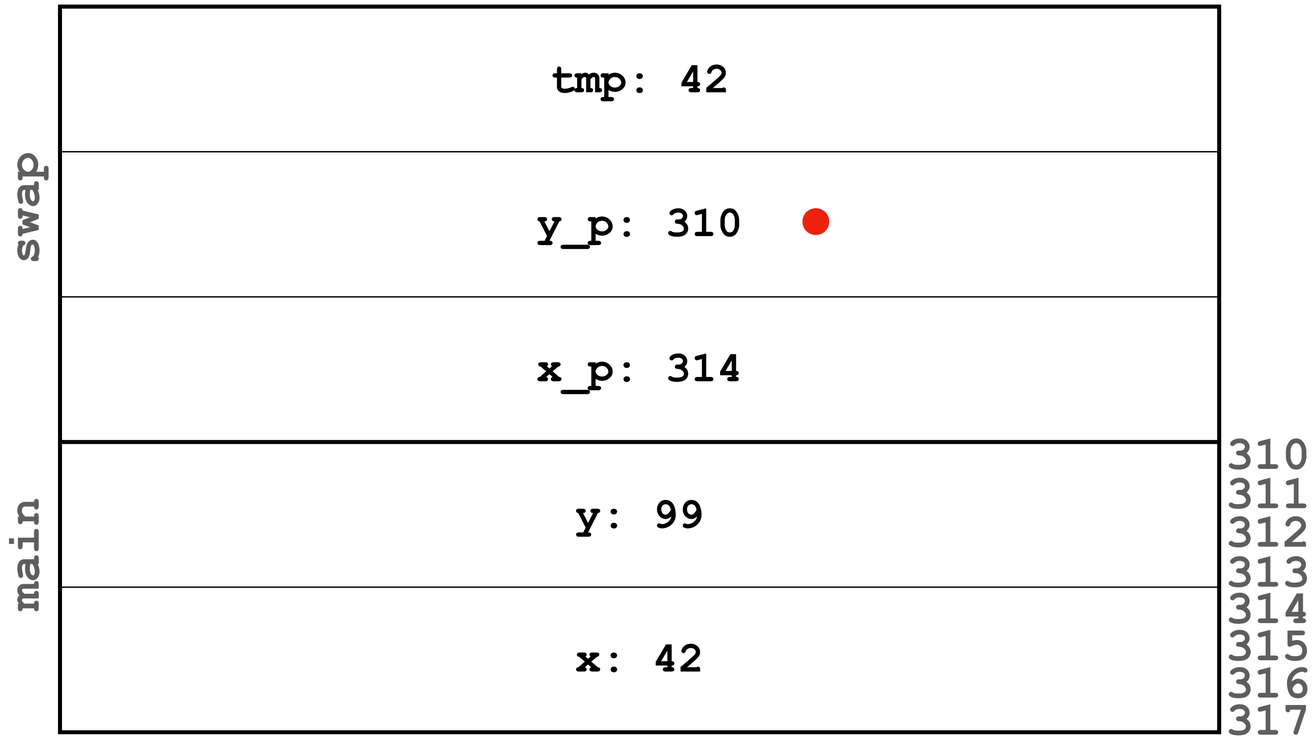
```c
#include <stdio.h>

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

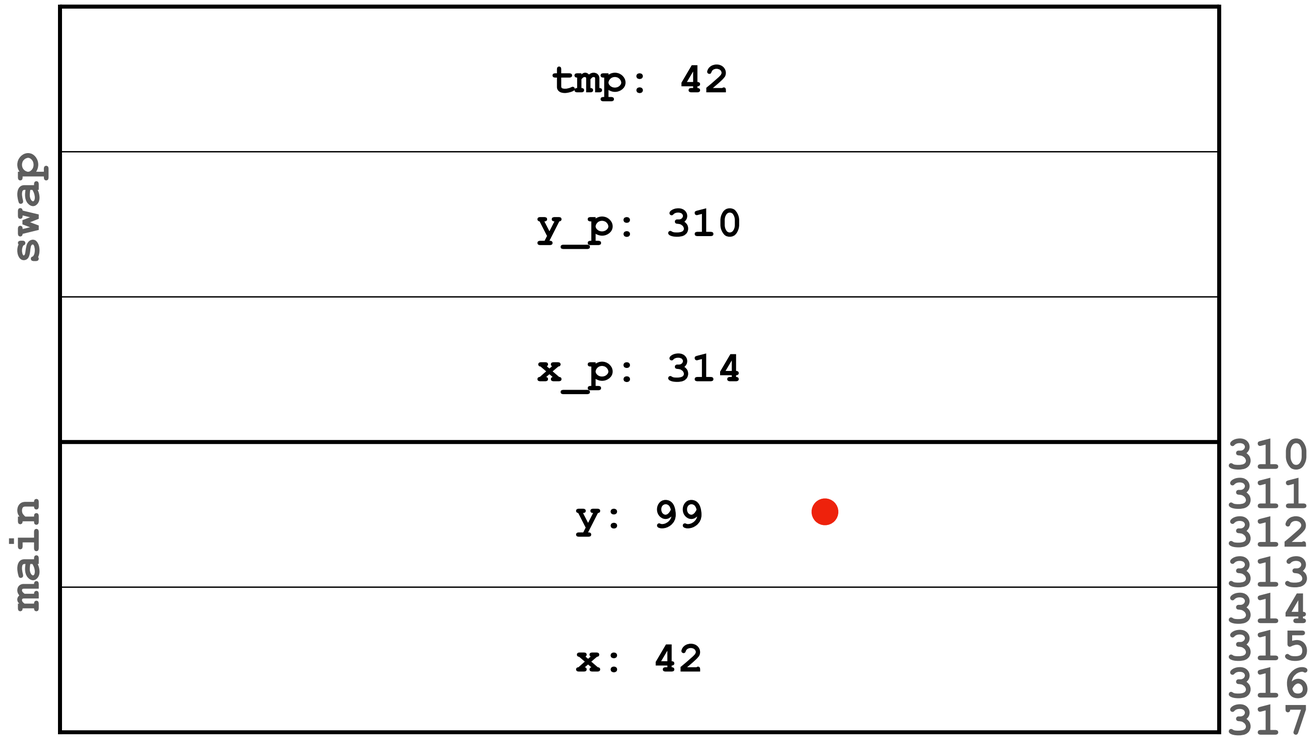
```c
#include <stdio.h>

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

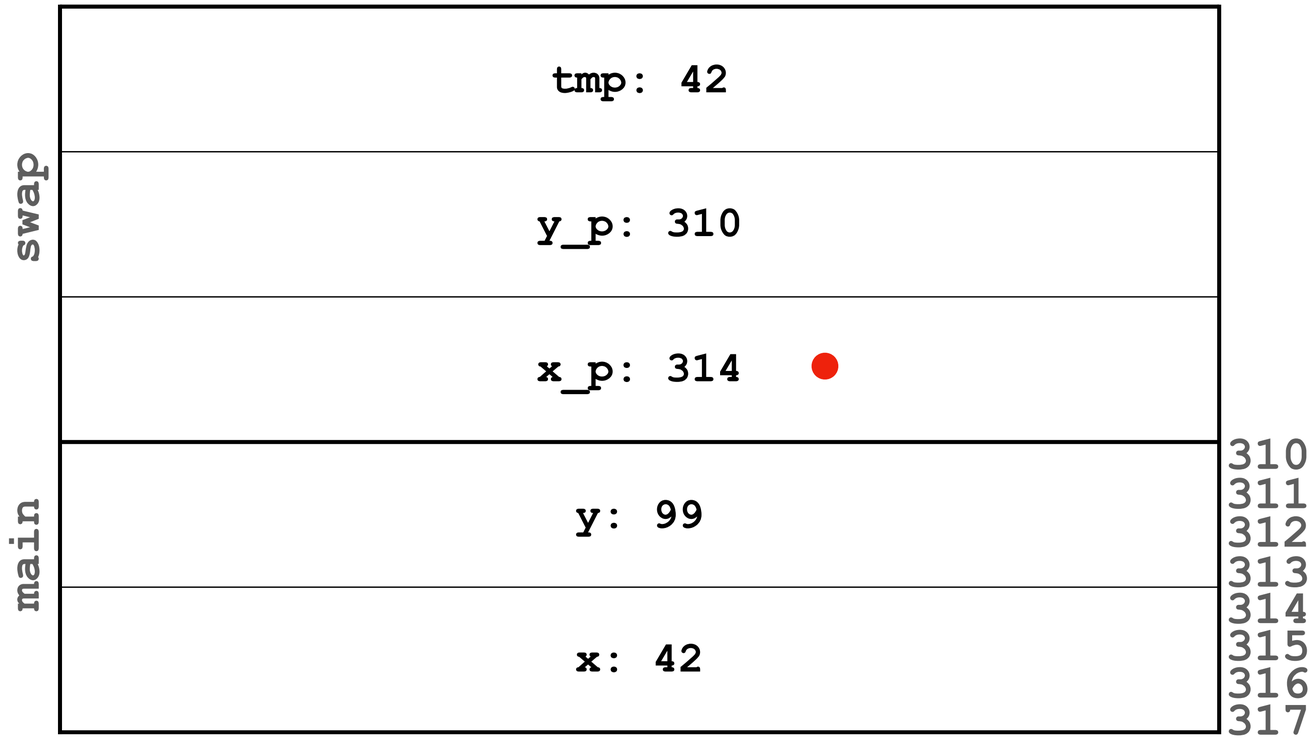
```c
#include <stdio.h>

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

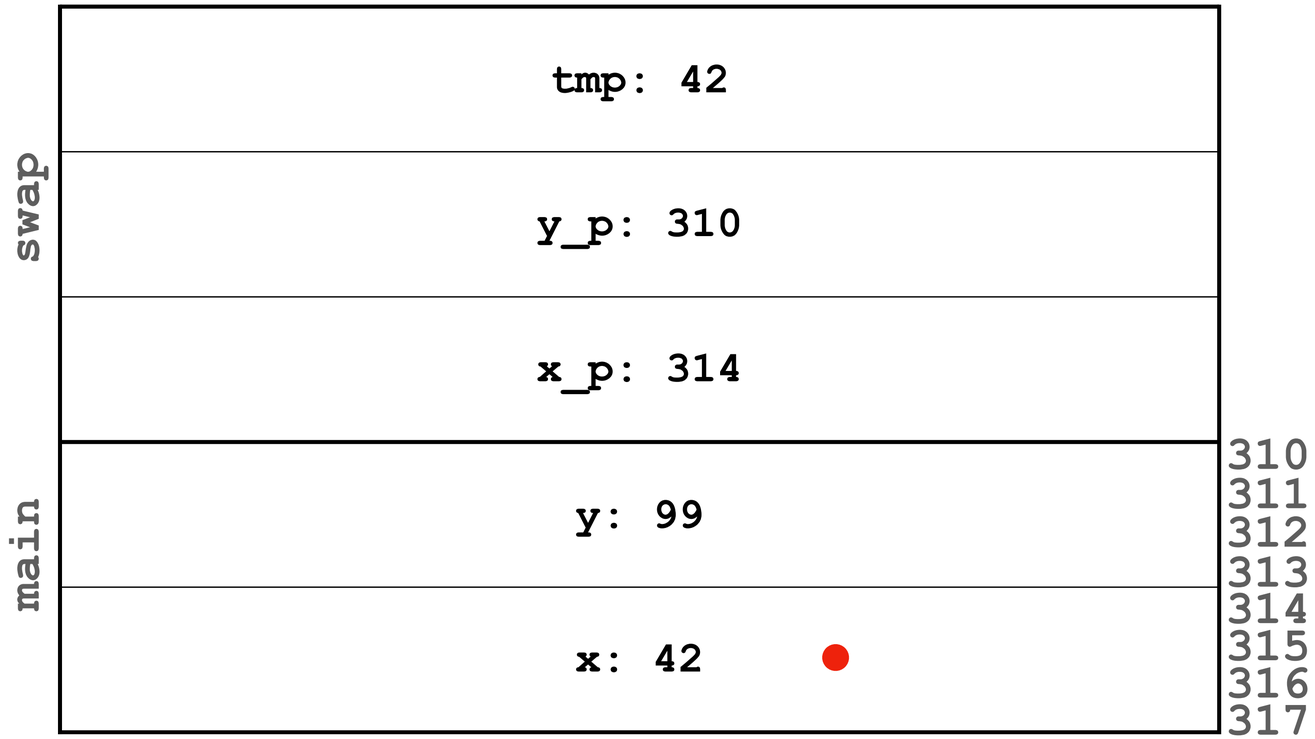
```c
#include <stdio.h>

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

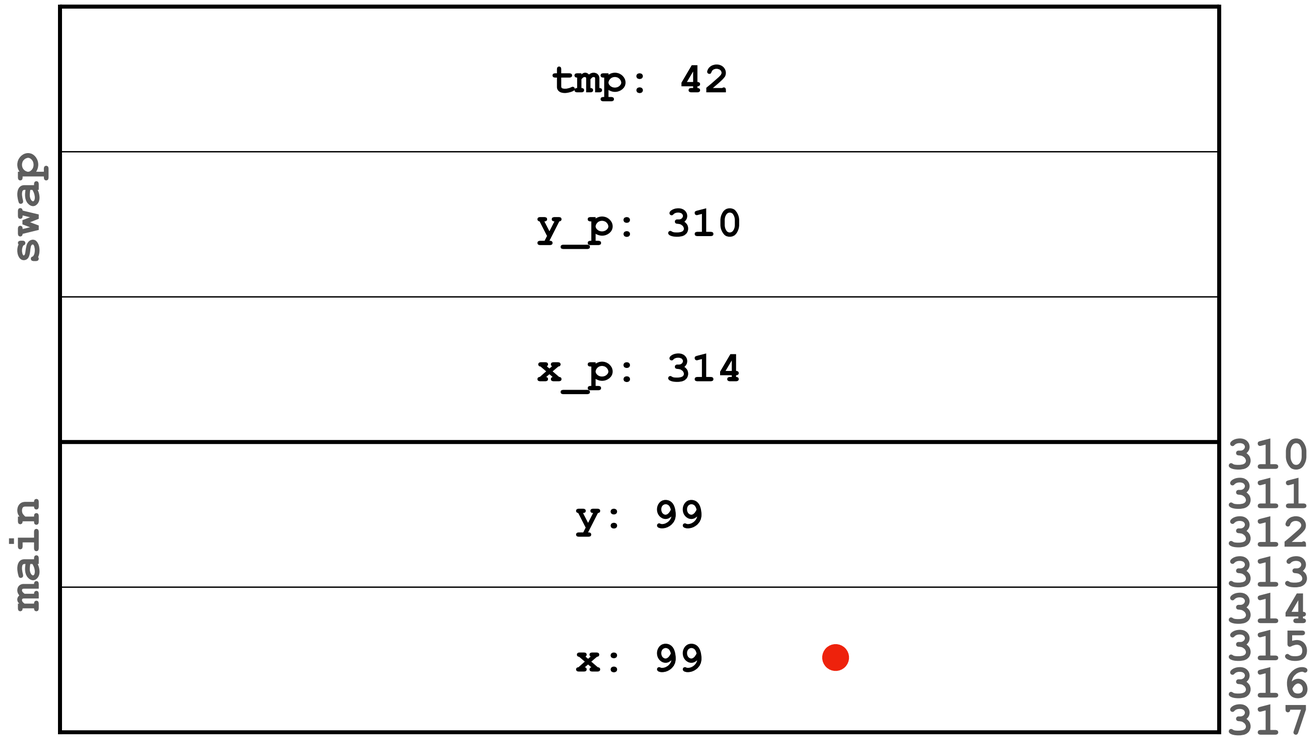
```c
#include <stdio.h>

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

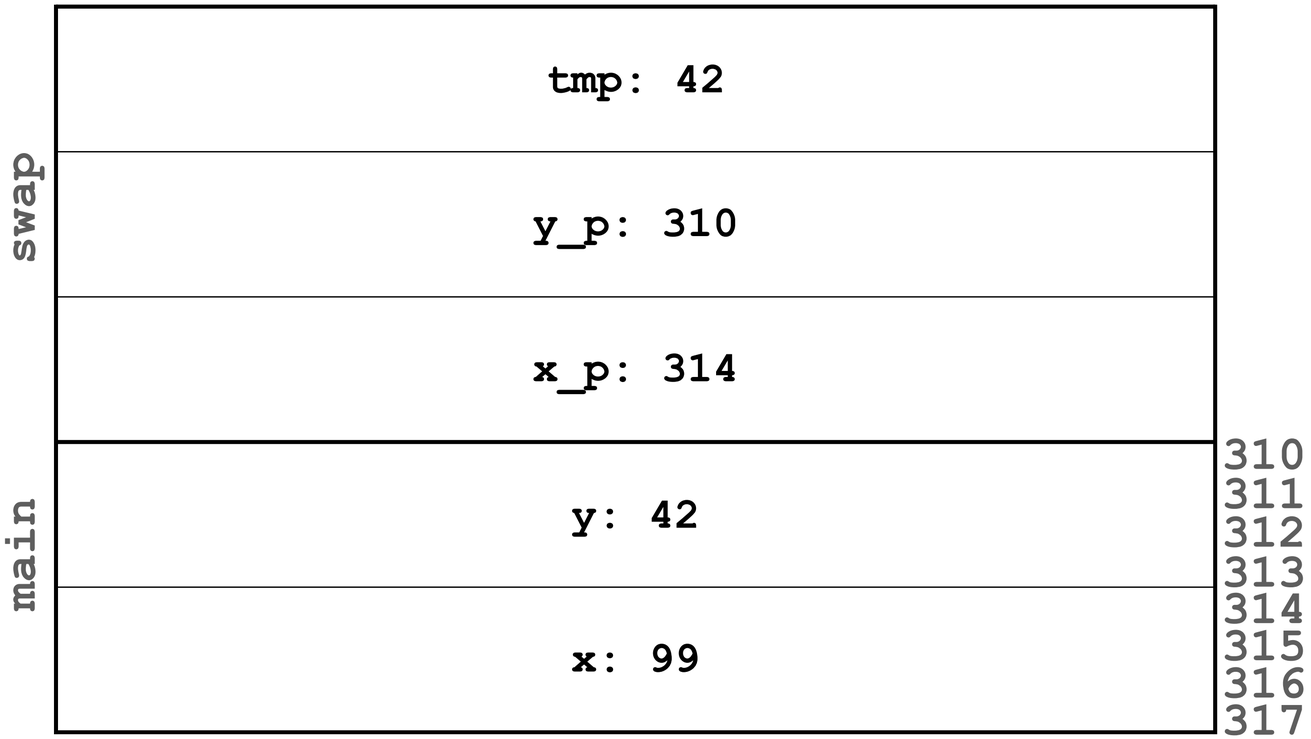
```c
#include <stdio.h>

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

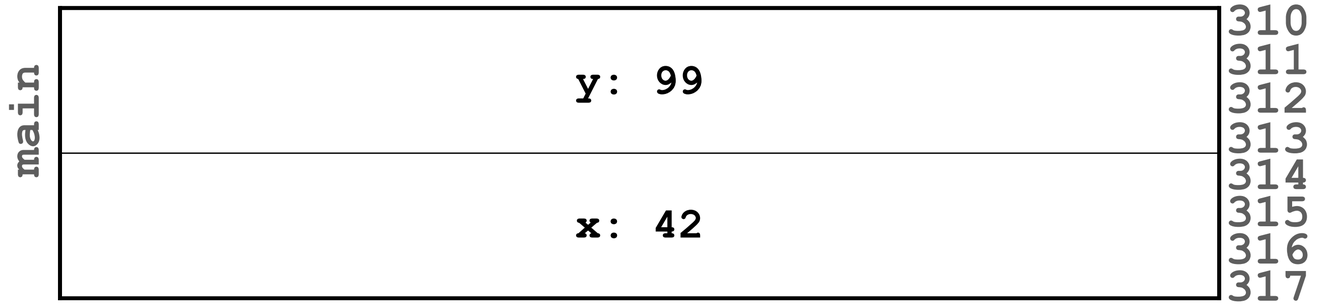
```c
#include <stdio.h>

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

Pointers

Checkpoint I

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

Pointers

Checkpoint I

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

```
int *x_p = x;
```

Pointers

Checkpoint I

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

```
int *x_p = x;
int *x_p = &x;
```

Pointers

Checkpoint I

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

```
int *x_p = x;
int *x_p = &x;
int z = *x_p;
```

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;
```

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;
```

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;
```

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;
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

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
```

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: 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: 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: 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;
}
```

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;
}
```

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;
}
```

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

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`

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

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

`struct`

Making your own types

`struct`

Making your own types

- Data placed in memory can be: `char`, `short`, `int`, `long`, `float`, `double`

`struct`

Making your own types

- Data placed in memory can be: `char`, `short`, `int`, `long`, `float`, `double`
- What if you want to store something other than a number?

`struct`

Making your own types

- Data placed in memory can be: `char`, `short`, `int`, `long`, `float`, `double`
- What if you want to store something other than a number?
  - Student?

`struct`

Making your own types

- Data placed in memory can be: `char, short, int, long, float, double`
- What if you want to store something other than a number?
  - Student?
  - Course?

`struct`

Making your own types

- Data placed in memory can be: `char`, `short`, `int`, `long`, `float`, `double`
- What if you want to store something other than a number?
  - Student?
  - Course?
  - House?

`struct`

Making your own types

- Data placed in memory can be: `char`, `short`, `int`, `long`, `float`, `double`
- What if you want to store something other than a number?
  - Student?
  - Course?
  - House?
  - ...

`struct`

Making your own types

- Data placed in memory can be: `char`, `short`, `int`, `long`, `float`, `double`
- What if you want to store something other than a number?
  - Student?
  - Course?
  - House?
  - ...
- You use numbers to represent them; you *digitize* them.

`struct`

Making your own types

- Data placed in memory can be: `char`, `short`, `int`, `long`, `float`, `double`
- What if you want to store something other than a number?
  - Student?
  - Course?
  - House?
  - ...
- You use numbers to represent them; you *digitize* them.
- In C, we can use *structures* to bundle data together.

`struct`

Syntax

struct

Syntax

```
struct student {
```

struct

Syntax

```
struct student {
        char first_name[32];
```

struct

Syntax

```
struct student {
        char first_name[32];
        char last_name[32];
```

struct

Syntax

```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
```

struct

Syntax

```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};
```

struct

Syntax

```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};
```

Don't forget the ;

struct

Syntax

```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};
```

Don't forget the ;

```
int main(void)
{
```

struct

Syntax

```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};
```

Don't forget the ;

```
int main(void)
{
        struct student john;
```

struct

Syntax

```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};
```

Don't forget the ;

```
int main(void)
{
        struct student john;
```

Don't forget the word struct

struct

Syntax

```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};
```

Don't forget the ;

```
int main(void)
{
        struct student john;

        strcpy(john.first_name, "John");
        strcpy(john.last_name, "Doe");
```

Don't forget the word struct

struct

Syntax

```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

int main(void)
{
        struct student john;

        strcpy(john.first_name, "John");
        strcpy(john.last_name, "Doe");
```

Don't forget the ;

Don't forget the word struct

Assignment (=) doesn't work with arrays

struct

Syntax

```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

int main(void)
{
        struct student john;

        strcpy(john.first_name, "John");
        strcpy(john.last_name, "Doe");
        john.gpa = 3.0;
```

Don't forget the ;

Don't forget the word struct

Assignment (=) doesn't work with arrays

struct

Syntax

```c
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

int main(void)
{
        struct student john;

        strcpy(john.first_name, "John");
        strcpy(john.last_name, "Doe");
        john.gpa = 3.0;

        printf("%s %s: %.2f\n", john.first_name, john.last_name, john.gpa);

        return 0;
}
```

Don't forget the ;

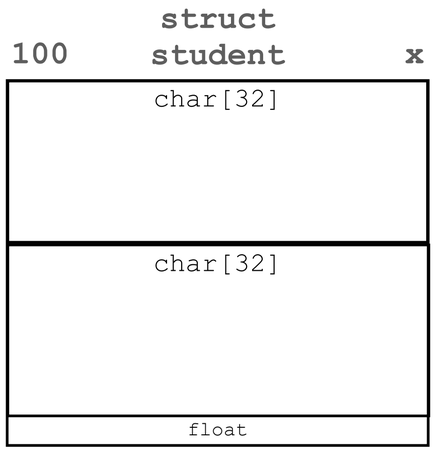
Don't forget the word struct

Assignment (=) doesn't work with arrays

struct

Syntax

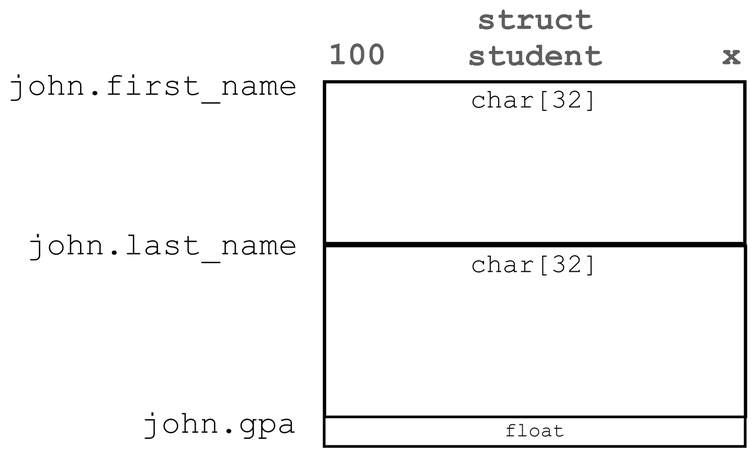
```c
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

int main(void)
{
        struct student john;

        strcpy(john.first_name, "John");
        strcpy(john.last_name, "Doe");
        john.gpa = 3.0;

        printf("%s %s: %.2f\n", john.first_name, john.last_name, john.gpa);

        return 0;
}
```



struct

Syntax

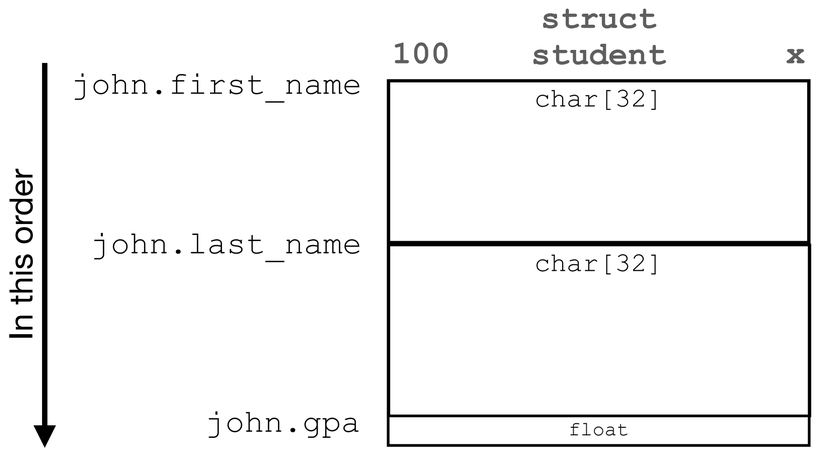
```c
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

int main(void)
{
        struct student john;

        strcpy(john.first_name, "John");
        strcpy(john.last_name, "Doe");
        john.gpa = 3.0;

        printf("%s %s: %.2f\n", john.first_name, john.last_name, john.gpa);

        return 0;
}
```



struct

Syntax

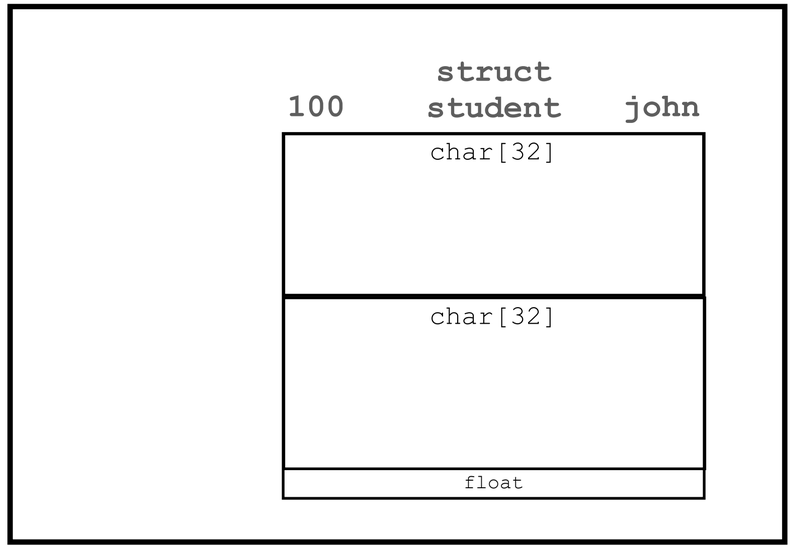
```c
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

int main(void)
{
        struct student john;

        strcpy(john.first_name, "John");
        strcpy(john.last_name, "Doe");
        john.gpa = 3.0;

        printf("%s %s: %.2f\n", john.first_name, john.last_name, john.gpa);

        return 0;
}
```



struct

Structures are passed by value

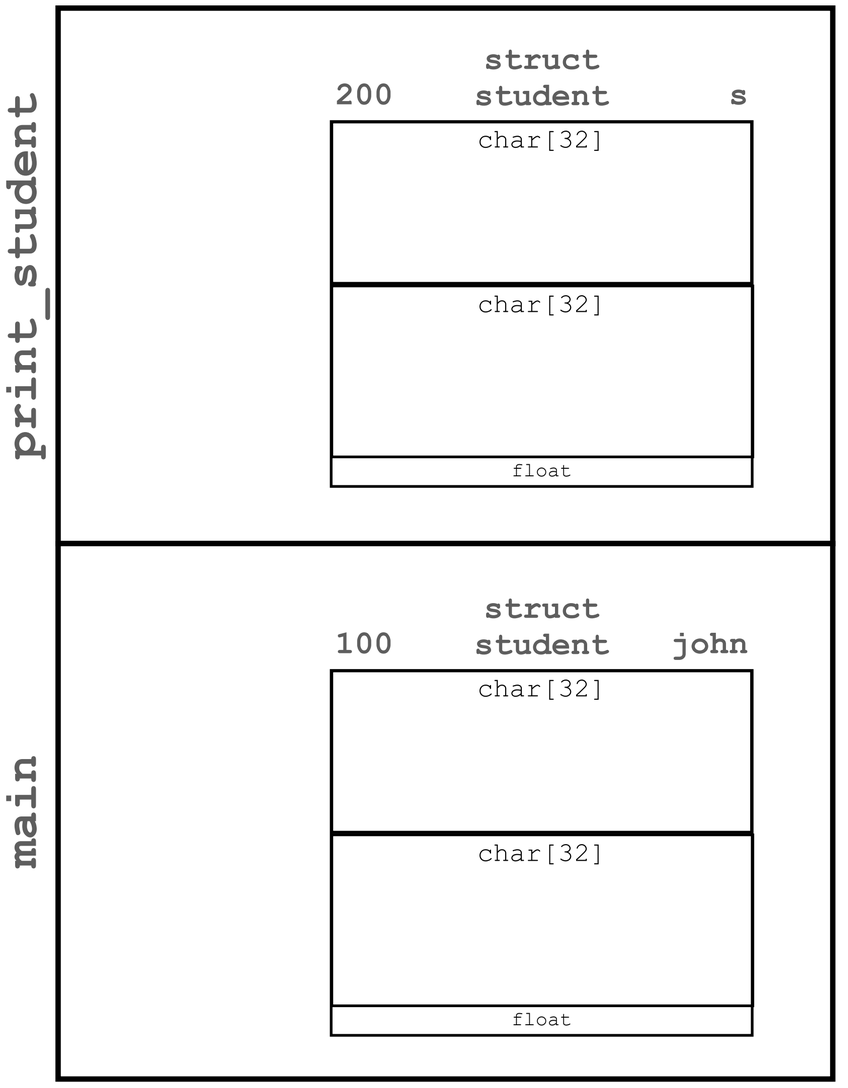
```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

void print_student(struct student s)
{
        printf("%s %s: %.2f\n", s.first_name, s.last_name, s.gpa);
}


int main(void)
{
        struct student john;
        ...
        print_student(john);

        return 0;
}
```



struct

Structures are passed by value

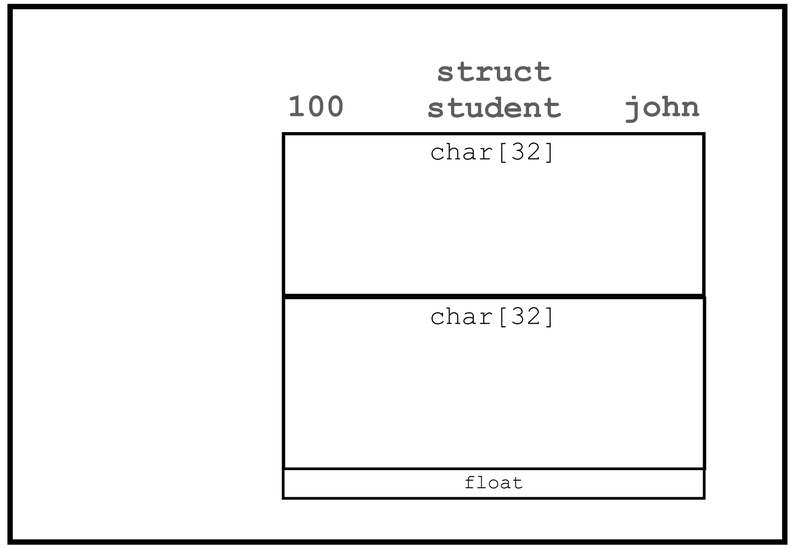
```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

void print_student(struct student s)
{
        printf("%s %s: %.2f\n", s.first_name, s.last_name, s.gpa);
}


int main(void)
{
        struct student john;
        ...
        print_student(john);

        return 0;
}
```



struct

Structures are passed by value

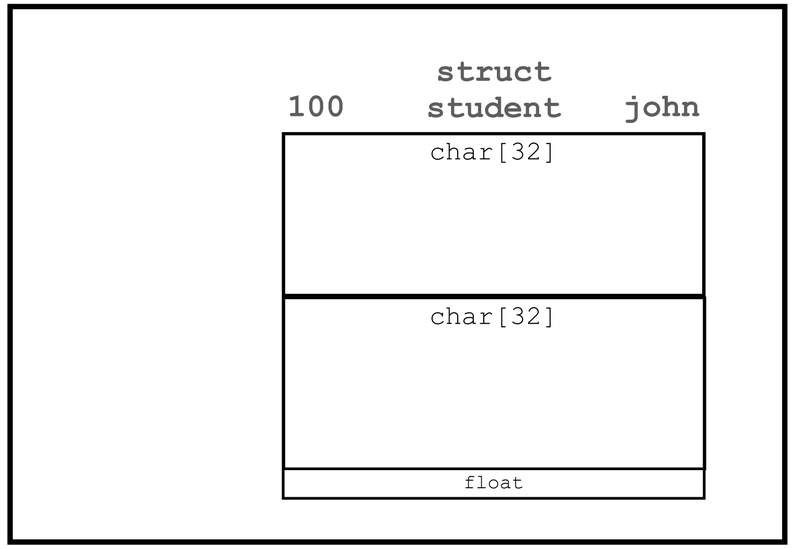
```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

void print_student(struct student s)
{
        printf("%s %s: %.2f\n", s.first_name, s.last_name, s.gpa);
}


int main(void)
{
        struct student john;
        ...
        print_student(john);

        return 0;
}
```



struct

Structures are passed by value

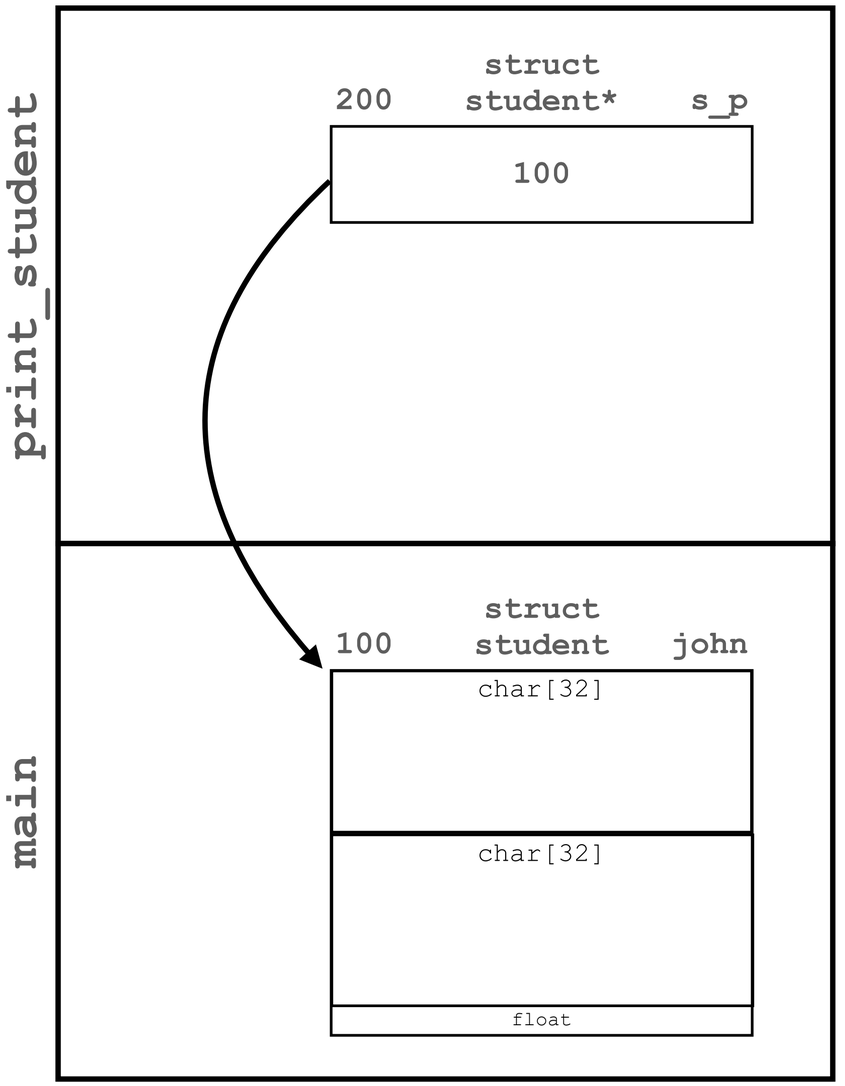
```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

void print_student(struct student *s_p)
{
        printf("%s %s: %.2f\n", (*s_p).first_name,
                                (*s_p).last_name,
                                (*s_p).gpa);
}

int main(void)
{
        struct student john;
        ...
        print_student(&john);

        return 0;
}
```



struct

Structures are passed by value

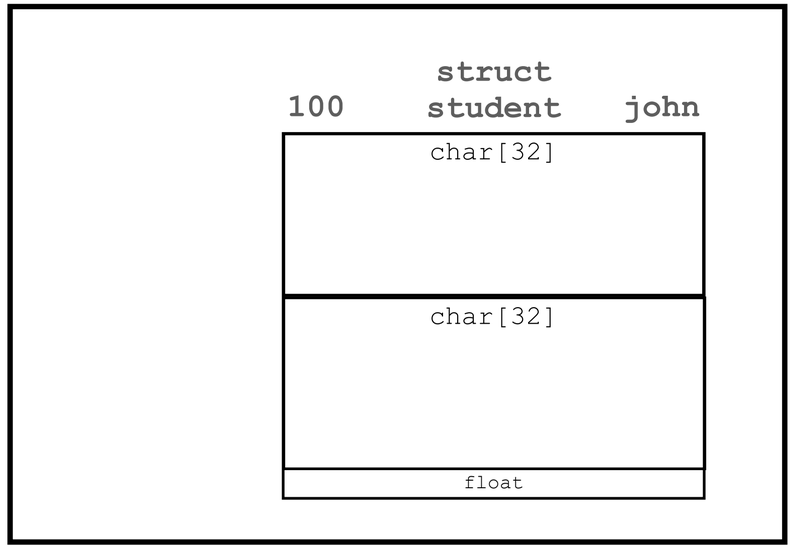
```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

void print_student(struct student *s_p)
{
        printf("%s %s: %.2f\n", (*s_p).first_name,
                                 (*s_p).last_name,
                                 (*s_p).gpa);
}

int main(void)
{
        struct student john;
        ...
        print_student(&john);

        return 0;
}
```



struct

Structures are passed by value

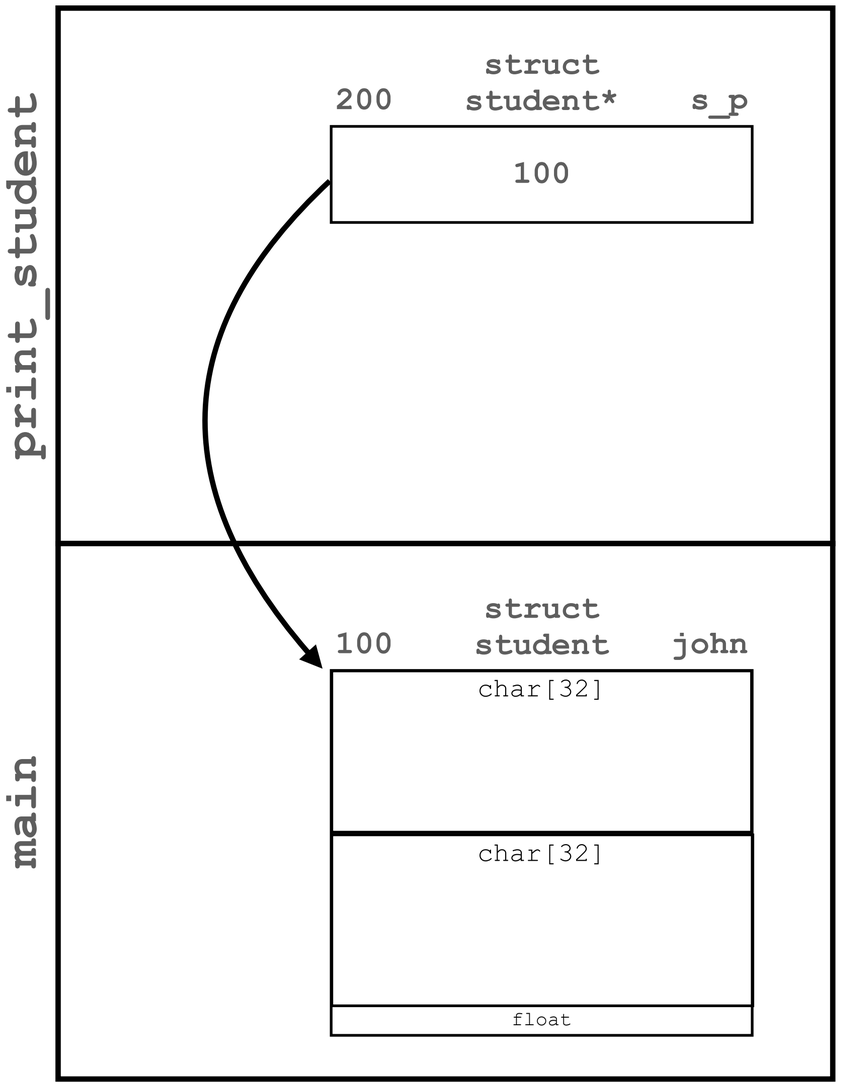
```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

void print_student(struct student *s_p)
{
        printf("%s %s: %.2f\n", (*s_p).first_name,
                                (*s_p).last_name,
                                (*s_p).gpa);
}

int main(void)
{
        struct student john;
        ...
        print_student(&john);

        return 0;
}
```



struct

Structures are passed by value

```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

void print_student(struct student *s_p)
{
        printf("%s %s: %.2f\n", s_p->first_name,
                                s_p->last_name,
                                s_p->gpa);
}

int main(void)
{
        struct student john;
        ...
        print_student(&john);

        return 0;
}
```

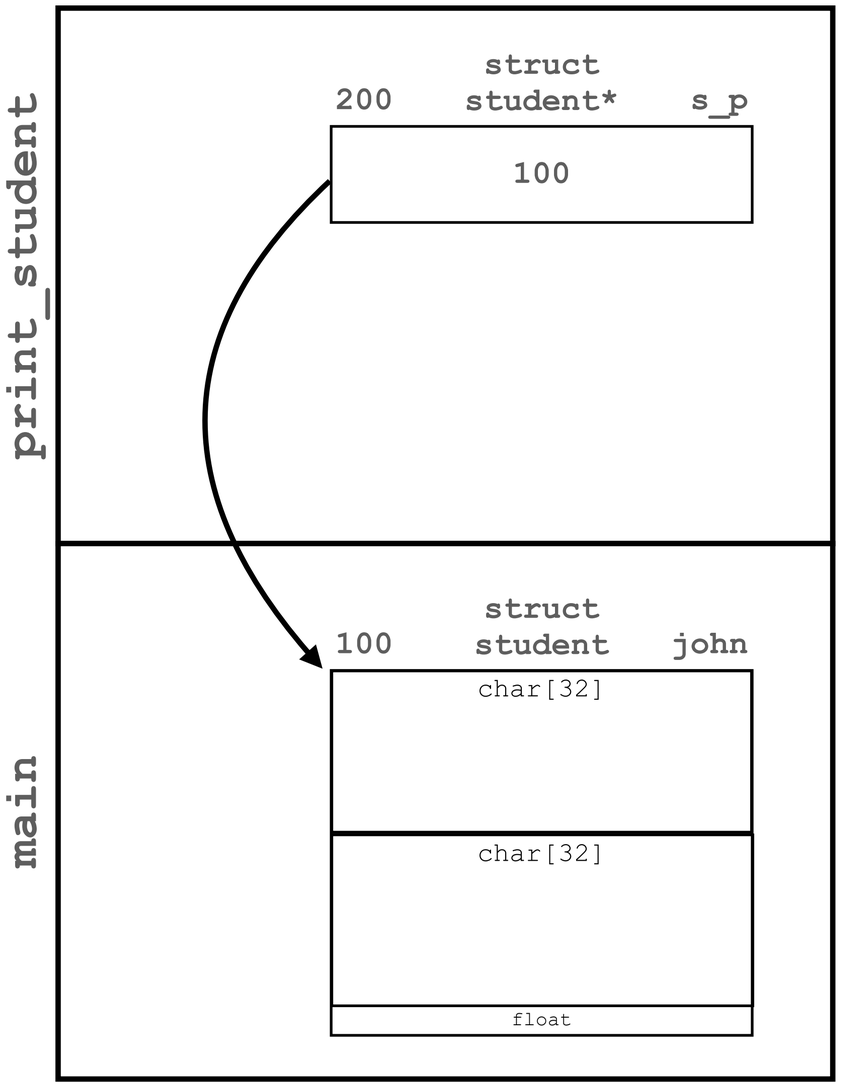


struct

Structures are passed by value

```
struct student {
        char first_name[32];
        char last_name[32];
        float gpa;
};

void print_student(struct student *s_p)
{
        printf("%s %s: %.2f\n", s_p->first_name,
                                s_p->last_name,
                                s_p->gpa);
}

int main(void)
{
        struct student john;
        ...
        print_student(&john);

        return 0;
}
```

```
s_p->gpa is a shorthand
for (*s_p).gpa

Btw, *s_p.gpa is read as
*(s_p.gpa), which is an
error
```



File I/O

- Memory is *volatile*. It loses data when power is removed.
- Files are stored on *non-volatile* storage media such as hard drives. It does not need power to preserve data.
- Memory supports *random access*. One can access memory at any address directly.
- Hard drives only support *sequential access*.
- C abstracts file system access via `FILE *`.