```
struct Point {
  int x, y;
};

void f(struct Point* p) {
  p->x = 22;
}

int main() {
  struct Point p = {1, 4};
  f(&p);
  printf("%d\n", p.x);
}
```

What does this program print?

A: 22

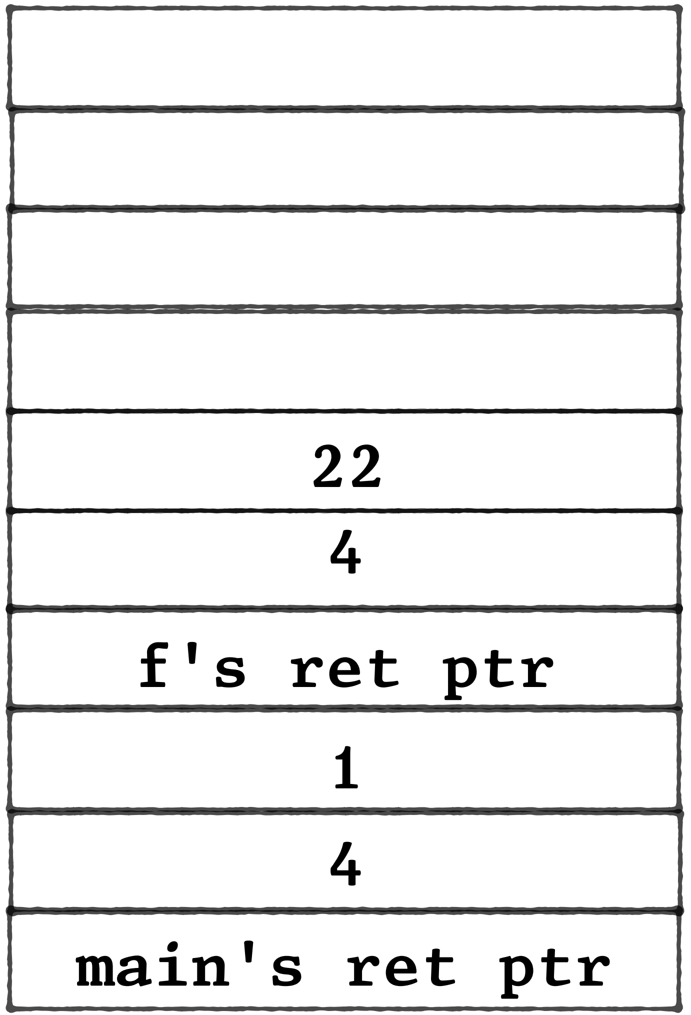
B: 1

```
struct Point {
  int x, y;
};

void f(struct Point* p) {
  p->x = 22;
}

int main() {
  struct Point p = {1, 4};
  f(&p);
  printf("%d\n", p.x);
}
```

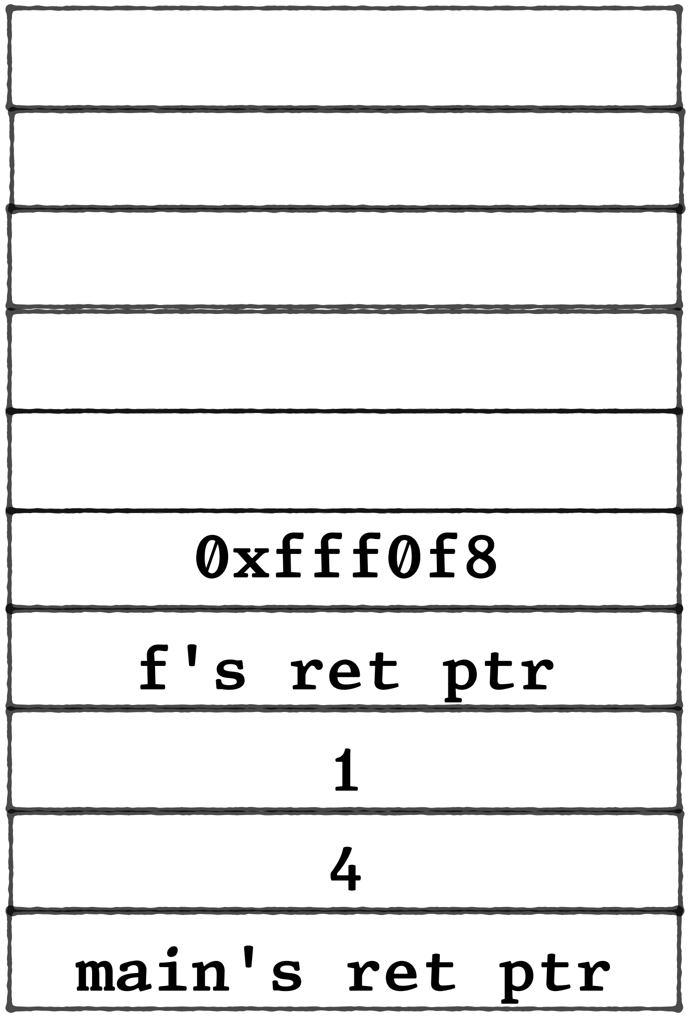
It prints 22. Which picture describes the stack layout best?

B

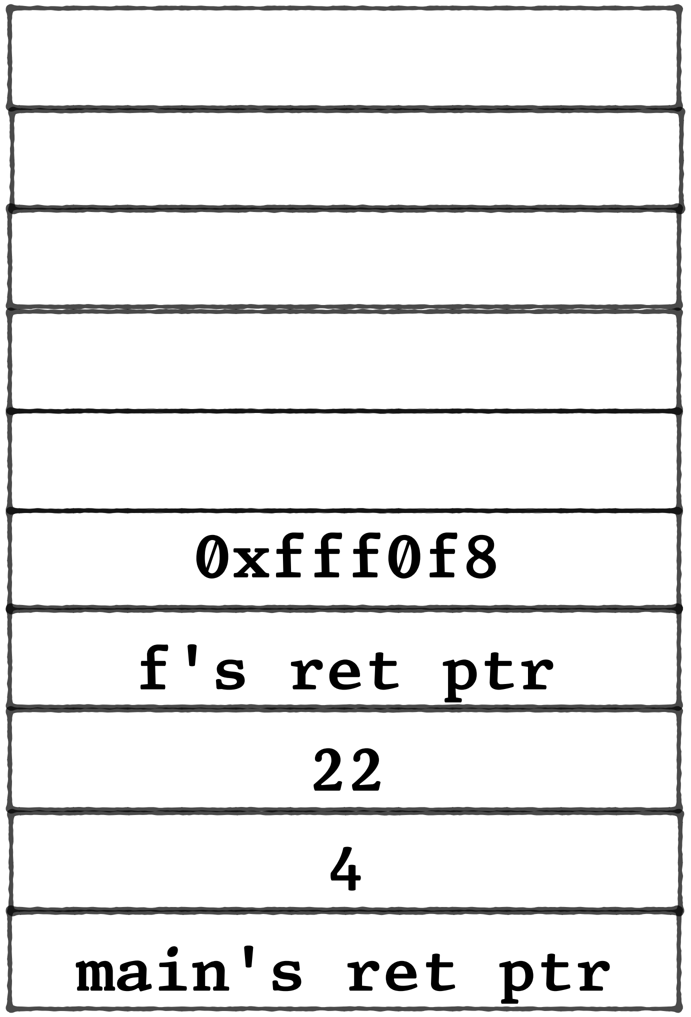
A



B



C



	A	B	C
0xfff0ec	22		
0xfff0f0	4	0xfff0f8	0xfff0f8
0xfff0f4	f's ret ptr	f's ret ptr	f's ret ptr
0xfff0f8	1	1	22
0xfff0fc	4	4	4
0xfff100	main's ret ptr	main's ret ptr	main's ret ptr

&x

Pronounced “address of x” or “get the address of x”

Evaluates to the *address* at which x is stored (on the stack)

If `x` has type `T`, then `&x` has type `T*`

```
int x = 10;
int* ptr_to_stack = &x;
```

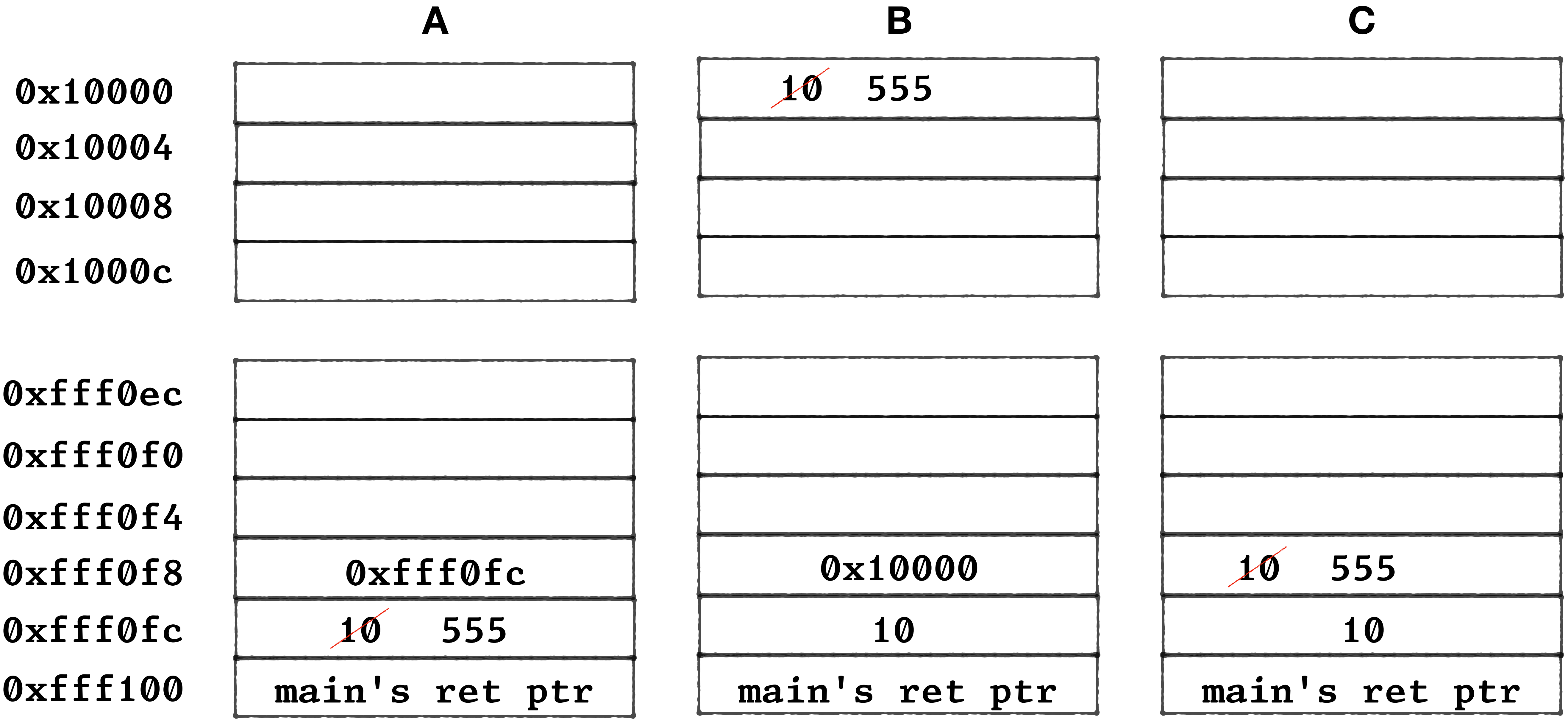
```
struct Point p = {1, 4};
struct Point* ptr_to_stack = &p;
```

```
int* x = malloc(sizeof(int));
int** ptr_to_stack = &x;
```

```
int main() {
  int x = 10;
  int* ptr_to_stack = &x;
  x = 555;
  printf("%d\n", *ptr_to_stack);
}
```

It prints 555. Which picture describes the stack layout best?

A



	A	B	C
0x10000		~~10~~ 555	
0x10004			
0x10008			
0x1000c			
0xfff0ec			
0xfff0f0			
0xfff0f4			
0xfff0f8	0xfff0fc	0x10000	~~10~~ 555
0xfff0fc	~~10~~ 555	10	10
0xfff100	main's ret ptr	main's ret ptr	main's ret ptr

```
int main() {
  int* x = malloc(sizeof(int));
  *x = 555;
  int** ptr_to_stack = &x;
  **ptr_to_stack = 333;
  printf("%d\n", *x);
}
```

It prints 333. Which picture describes the memory layout best?

B

	A	B	C
0x10000	~~555~~ 333	~~555~~ 333	555
0x10004			333
0x10008			
0x1000c			

	A	B	C
0xfff0ec			
0xfff0f0			
0xfff0f4			
0xfff0f8	0x10000	0xfff0fc	0x10008
0xfff0fc	0x10000	0x10000	0x10000
0xfff100	main's ret ptr	main's ret ptr	main's ret ptr

```
int main() {
  struct Point p = {1, 4};
  struct Point* ptr_to_stack = &p;
  p.x = 100;
  printf("%d\n", _______________)
}
```

Which picture describes the stack layout best?

A

	A	B	C
0x10000			~~1~~ 100
0x10004			4
0x10008			
0x1000c			
0xfff0ec			
0xfff0f0		~~1~~ 100	
0xfff0f4	0xfff0f8	4	
0xfff0f8	~~1~~ 100	1	0x10000
0xfff0fc	4	4	0x10000
0xfff100	main's ret ptr	main's ret ptr	main's ret ptr

```
int main() {
  struct Point p = {1, 4};
  struct Point* ptr_to_stack = &p;
  p.x = 100;
  printf("%d\n", _______________)
}
```

Which of the following could fill in the blank above to print 100?

B or E

A `(&p).x`

B `ptr_to_stack->x`

C `ptr_to_stack.x`

D `(&ptr_to_stack).x`

E `(*ptr_to_stack).x`

A common pattern in C libraries

```
time_t time( time_t *arg );
```

Returns the current calendar time encoded as a time_t object, and also stores it in the time_t object pointed to by arg

```
time_t t; // t is a stack-allocated time struct,
          // w/fields for minutes, seconds, etc.
time(&t); // t is filled in with current time info
```

http://en.cppreference.com/w/c/chrono/time

Where does & fit into this table?

	The type of x is…	The compiler generates code to…	Example

A	primitive (int, char)	Pass (copy) directly to callee	`int x = 10;`
			`f(x); // 10 passed in r0`

B	pointer	Pass (copy) directly to callee; copies an address	`int* x = malloc(sizeof(int));`
			`f(x); // address returned from`
			`      // malloc passed in r0`

C	array	Pass ***address*** of array directly to callee	`char cs[] = "abcd";`
			`f(cs); // address for start`
			`       // of cs passed in r0`

D	struct	***Copy*** struct contents to callee	`struct Point p = {1, 4};`
			`f(p); // 1, 4 copied to stack`
			`      // frame for f to use`

E **None of the above**

On the previous slide:

It's an intentionally open-ended question.

There's an argument for E – the & operator is just another way
to get an address, and doesn't have any specific rules for function calls

There's an argument for B, since & creates pointer-typed values,
and those values will be passed around as pointers.

There's an argument for C, since if we wanted to get similar behavior
for stack-allocated structs as we do for arrays, we'd need to use &.

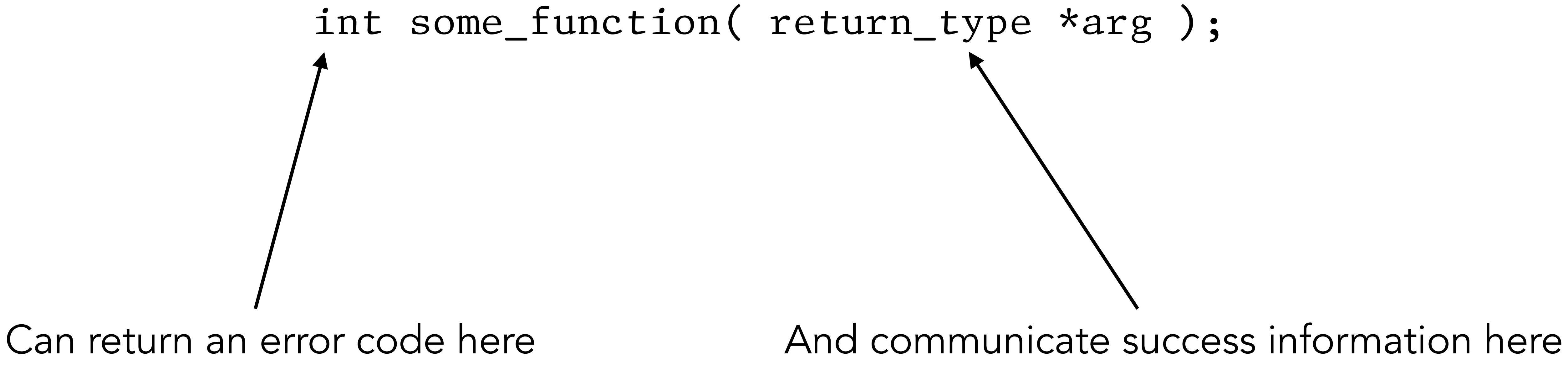
“Pass by value”

Two common vocabulary terms:

“Pass by reference”

The problem: these terms have different interpretations depending on the context and programming language.

When you hear these terms used, take it as a hint to start thinking about shallow vs. deep copy, and about changes to temporary (stack) memory vs long-lived (heap) memory



```
int some_function( return_type *arg );
```

Can return an error code here

And communicate success information here