Algorithms for GIS

Pointers in C

Laura Toma

Bowdoin College

Pointers

T* p;

- p is a pointer to something of type T (p is a T*)

Pointers

```
T*   p;
```

- Put differently, *p has type T (*p is a T)
- Some programmers actually prefer to write it as

```
T   *p;
```

Pointers

T* p;

- To dereference p, p has to store a valid address

Pointers

```
T* p;


T a;
p = &a;

p = (T*) malloc(10);


T* q;
q = p;
```

p stores the address of a

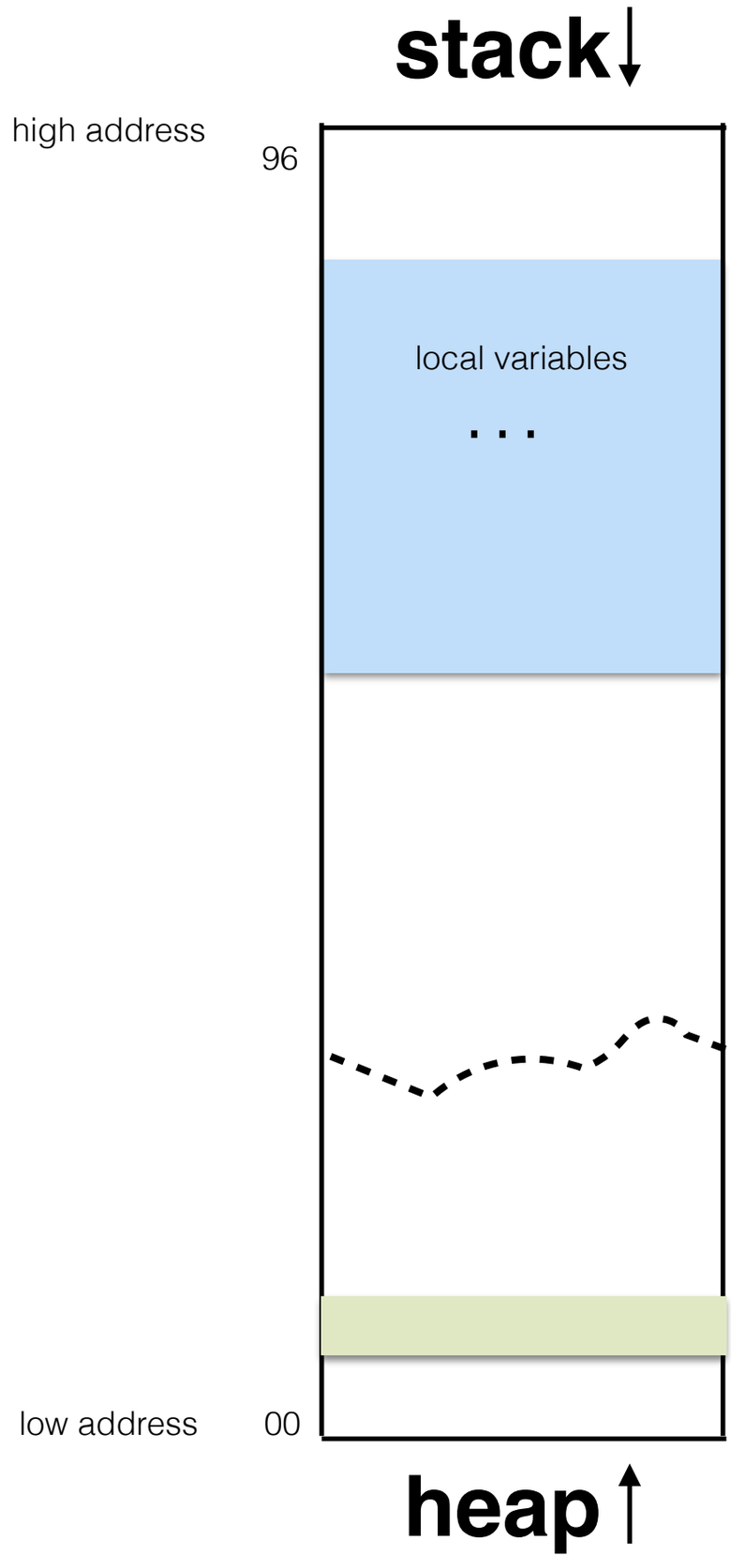
p stores the address of a chunk of 10B of memory on heap

q stores the same address as p

Pointers

T* p;

- What's a valid address?
  - the address of a variable
  - the address returned by malloc(), calloc() or realloc()



Pointers and arrays

- An array is a chunk of memory (on stack or heap)
- The name of an array is the address of the array

`int a[10];` ⟵ a is an int*
a points to a chunk of 10 ints (on the stack)

- `int a[]` is the same as `int* a`
- Operator [] implemented as
  - `a[0]` is the same as `*a`
  - `a[1]` is the same as `*(a + sizeof(int))`
  - `a[2]` is the same as `*(a + 2*sizeof(int))`
  - …

Pointers and arrays

T* a; ⟵ a is a T*

- You can think of a as an array
- Assuming a is a valid address, you can do *a or a[0]
- Assuming that a+sizeof(T) is a valid address, you can do *(a + sizeof(int)) or a[1]
- And so on
- Arrays provide convenience, but you can program without arrays

Pointers and arrays

- In fact, it's more efficient to avoid the [] operator and work directly on the pointer.. But don't do it!

```
/* return the length of a null-terminated string str */
unsigned int strlen(char*  str) {
        char* p = str;
        while (p && (*p != '\0')) p++;
        return p - str;
}
```

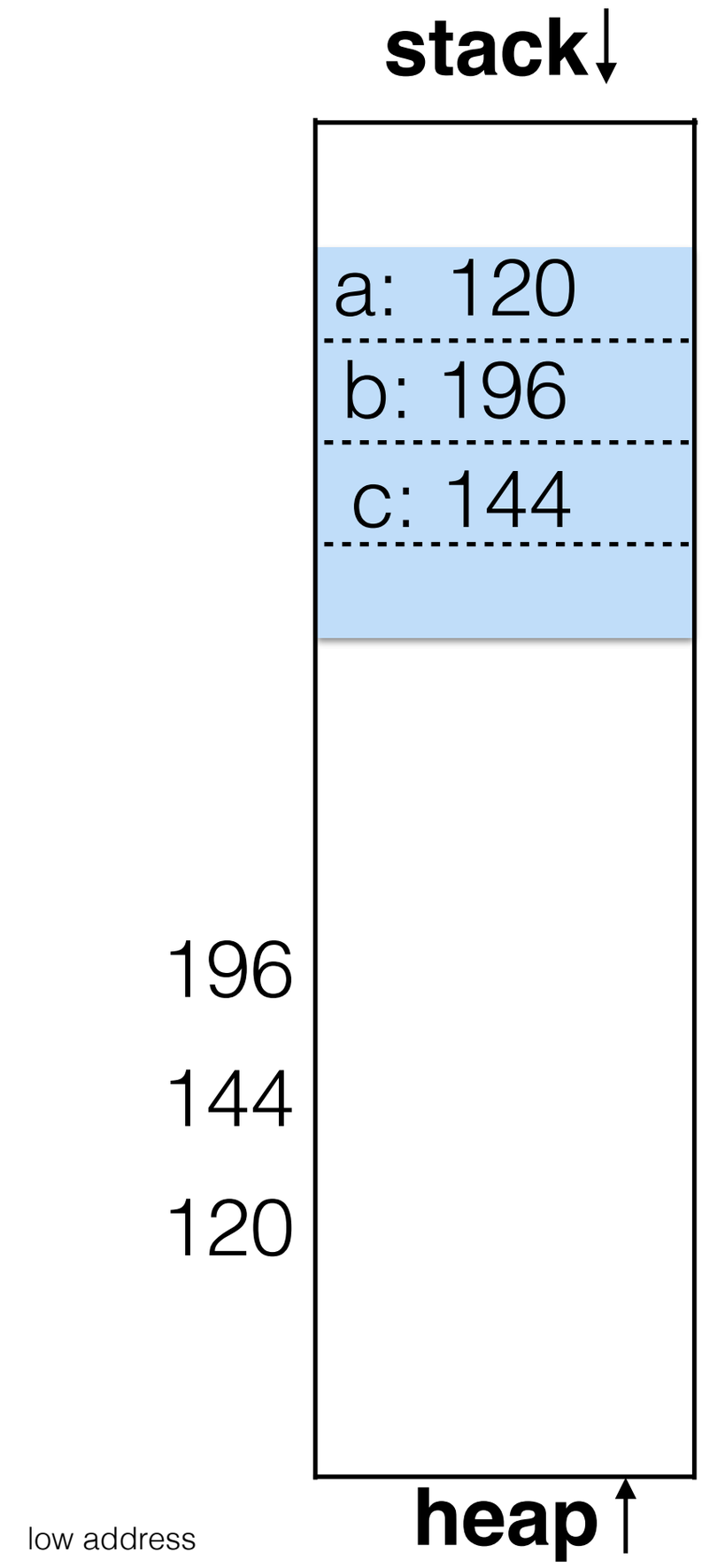
Pointers

```
int* a;

char* b;

double* c;
```

- How are they different?



Pointers

```
int* a;

char* b;

double* c;
```

- They are all addresses !
- The type matters only when you dereference
  - *a will read 4 bytes from address a and return it as an int
  - *b will read one byte from address b and return it as a char
  - *c will read 8 bytes from address c and return it as a double

Pointers

```
int* a;

char* b;

double* c;
```

- In fact C probably let's you do naughty things like casting pointers

```
char x = *((char*)a);
```

Void pointers

void* p; ⟵ p is an address to something

- Some functions work generically with void* and you have to cast them to the type that you need

```
NAME
       qsort, qsort_r - sort an array

SYNOPSIS
       #include <stdlib.h>

       void qsort(void *base, size_t nmemb, size_t size,
                  int (*compar)(const void *, const void *));

       void qsort_r(void *base, size_t nmemb, size_t size,
                  int (*compar)(const void *, const void *, void *),
                  void *arg);
```

```
typedef struct {
        float x, y;
} point2;
```

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is undefined.

```
int my_compare(const void * a, const void *b) {

   point2* p1 = (point2*)a;
   point2* p2 = (point2*)b;
   //now p1 and p2 are pointers to point2

   if (p1.x < p2.x) return -1;
   if (p1.x > p2.x) return 1;
   if (p1.y < p2.y) return -1;
   if (p1.y > p2.y> return 1;
   return 0;
}
```

What does this do?

C and pass-by-value

- When functions are called, their arguments are passed-by-value. That is, their values are copied.

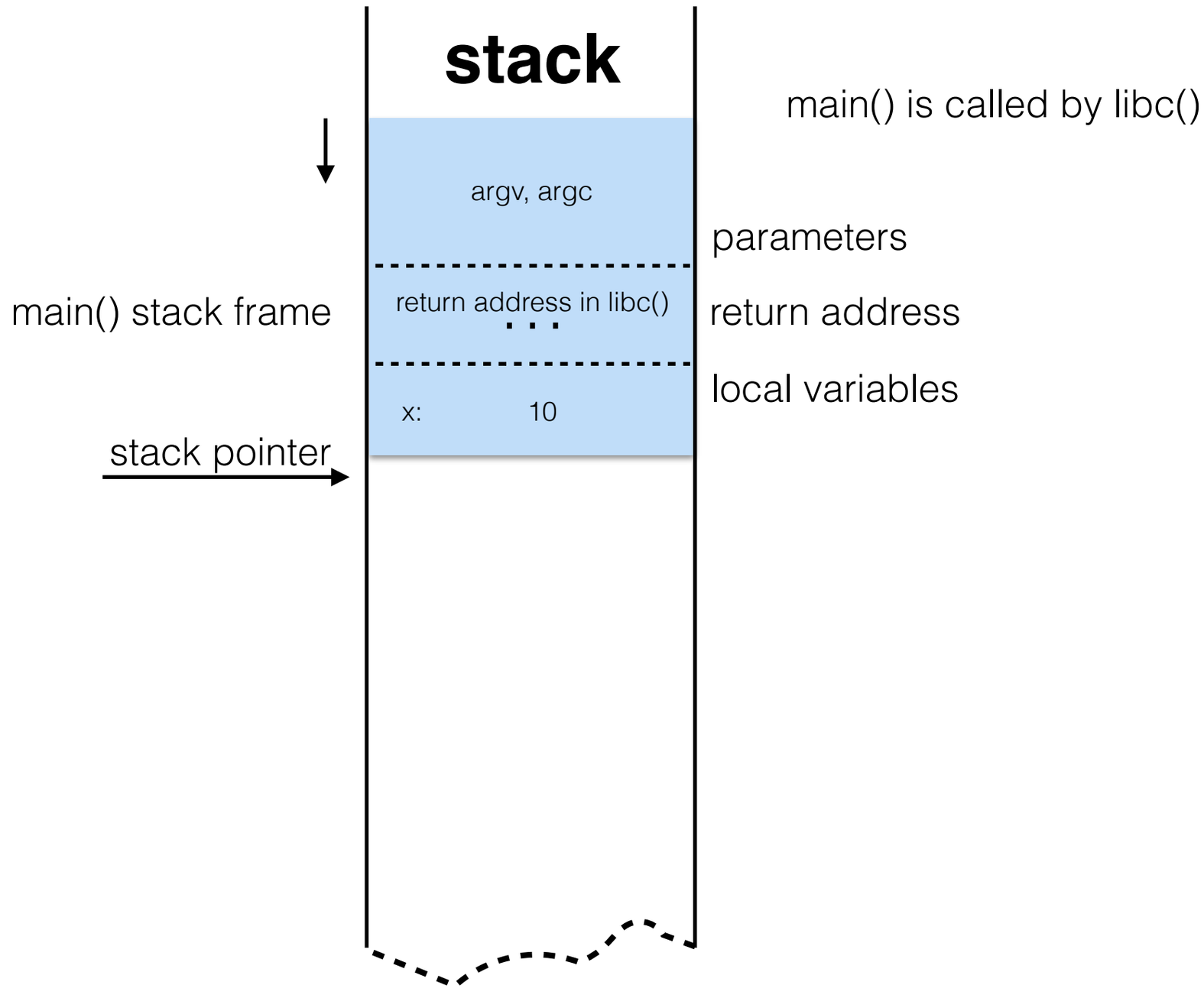
```
void fun(int a) {
    a = 3;
}
```

```
int main(..) {
    int x = 10;
    fun(x);
}
```

when calling fun(x) the value of x is copied into the parameter a of fun()

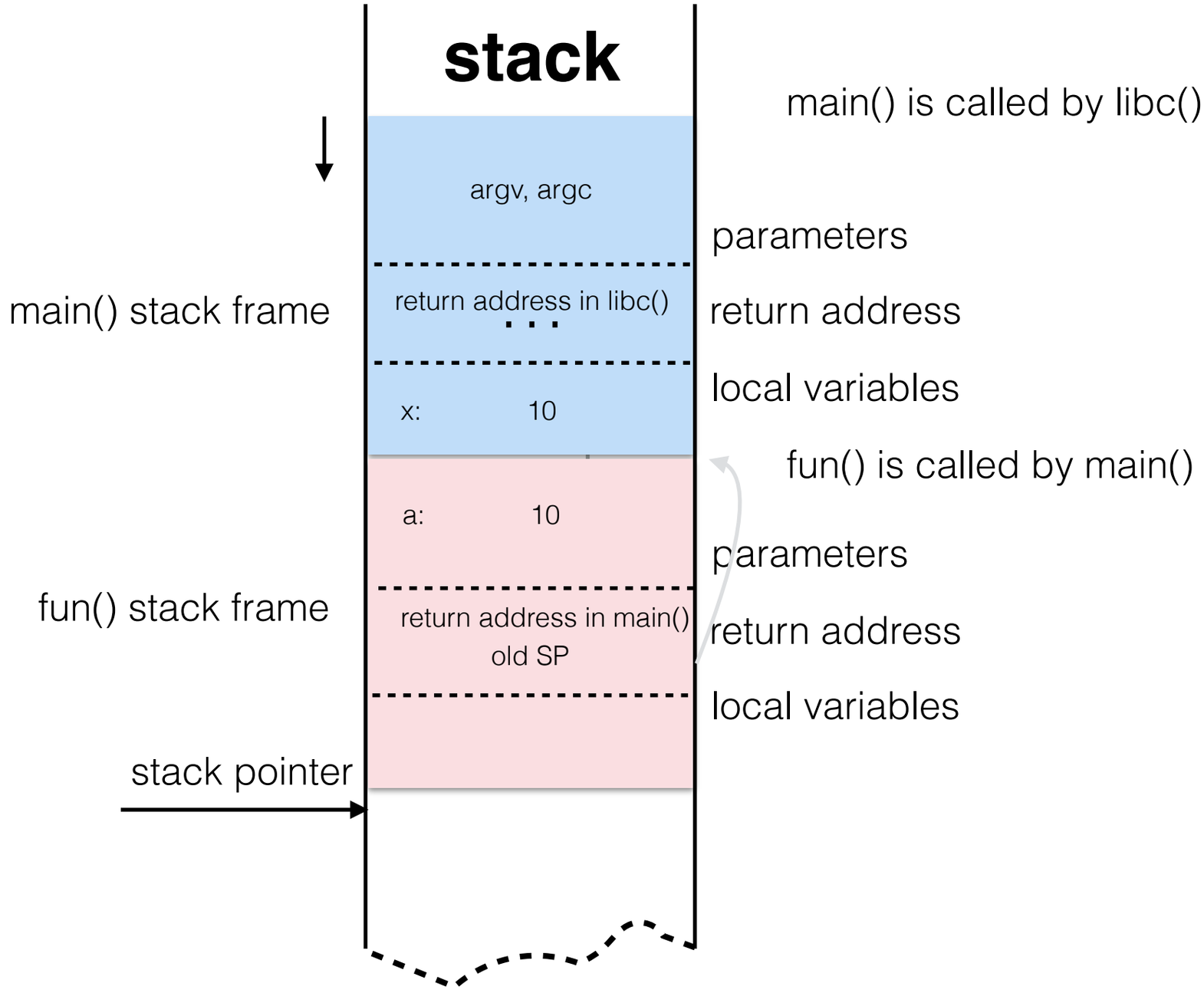
```
void fun(int a) {
    a = 3;
}
```

```
int main(..) {
    int x = 10;
    fun(x);
}
```



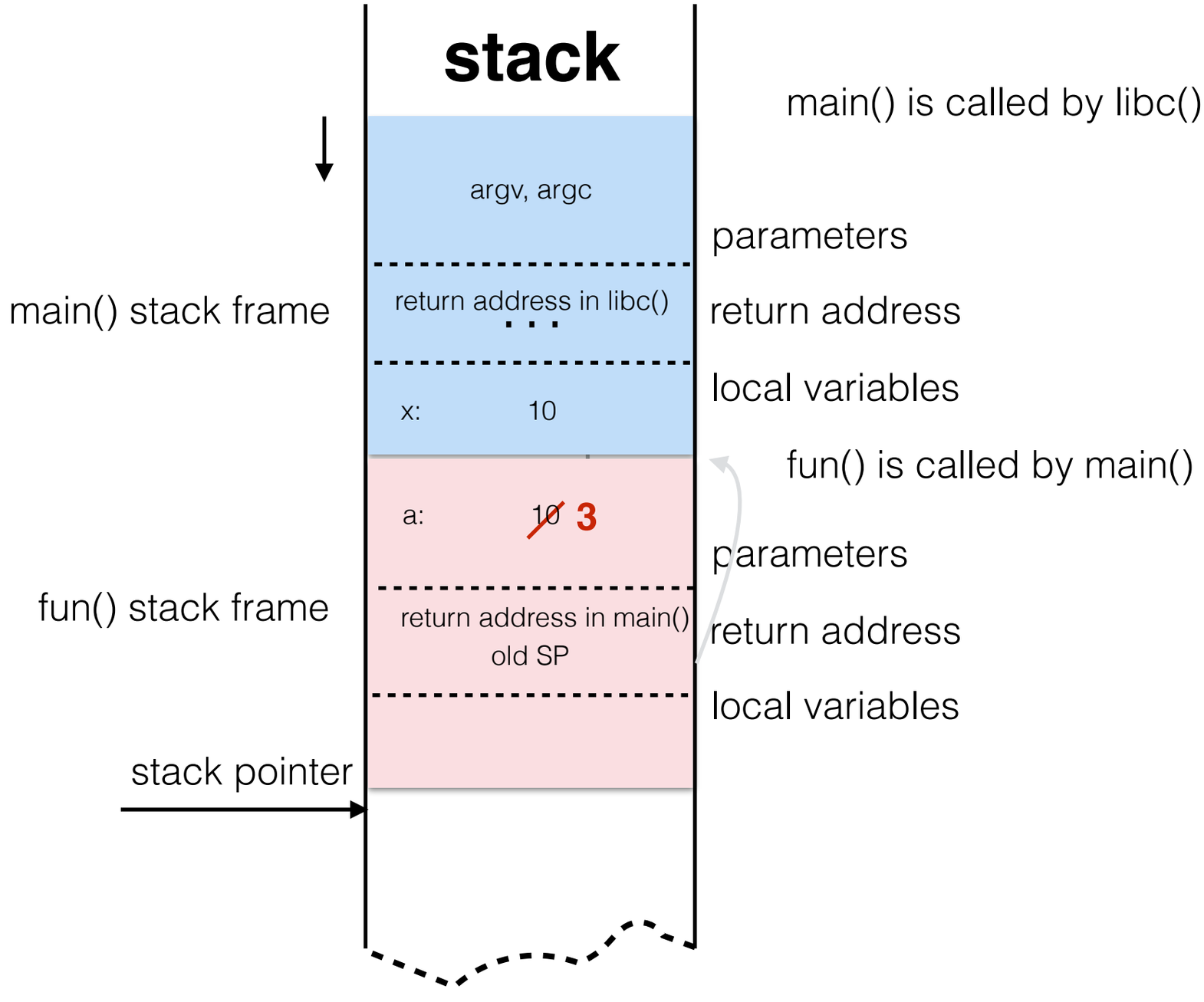
```
void fun(int a) {
    a = 3;
}
```

```
int main(..) {
    int x = 10;
    fun(x);
}
```



```
void fun(int a) {
    a = 3;
}
```

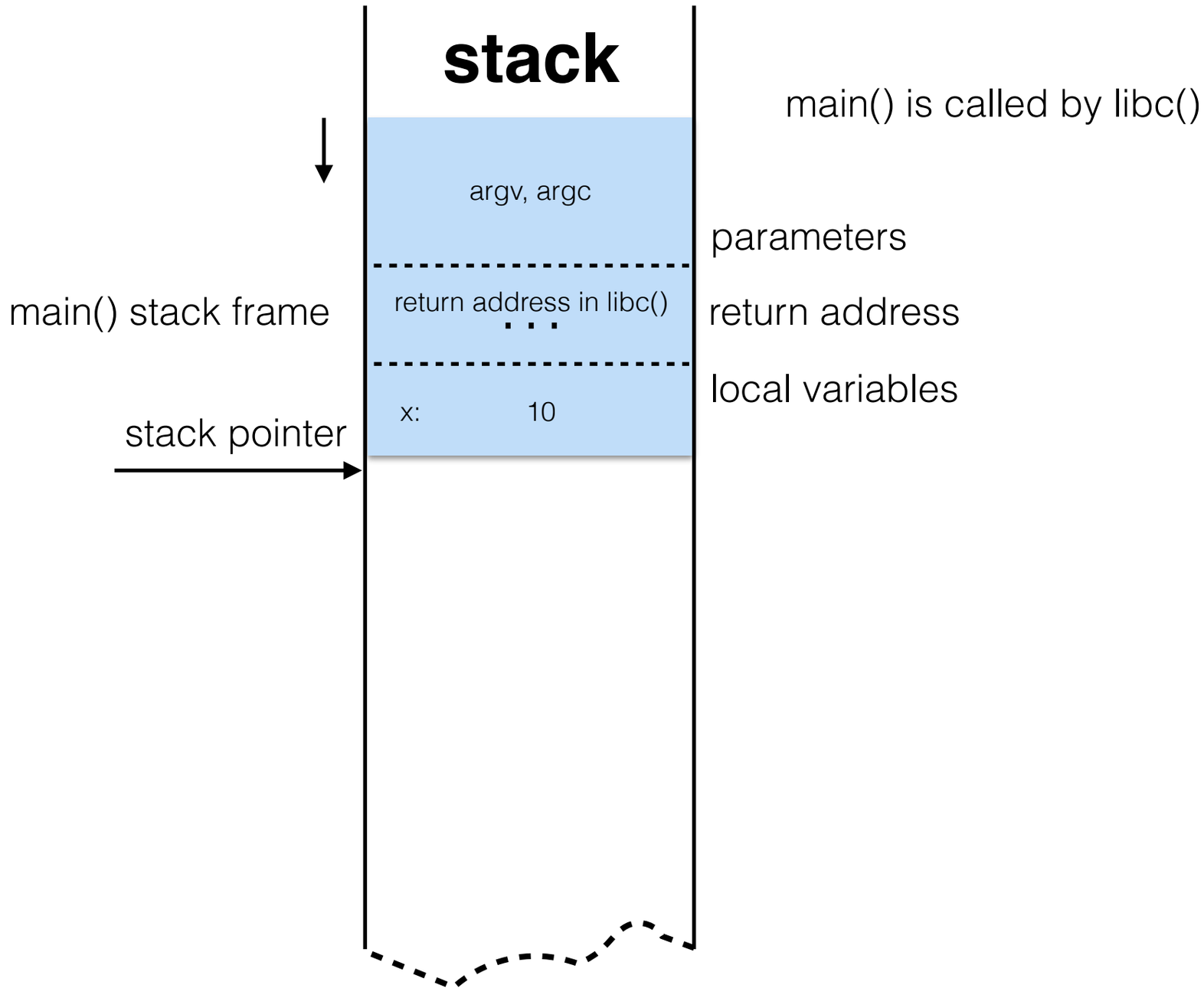
```
int main(..) {
    int x = 10;
    fun(x);
}
```



The value of x does not change!

```
void fun(int a) {
    a = 3;
}
```

```
int main(..) {
    int x = 10;
    fun(x);
}
```



To change a, fun() must get its address.

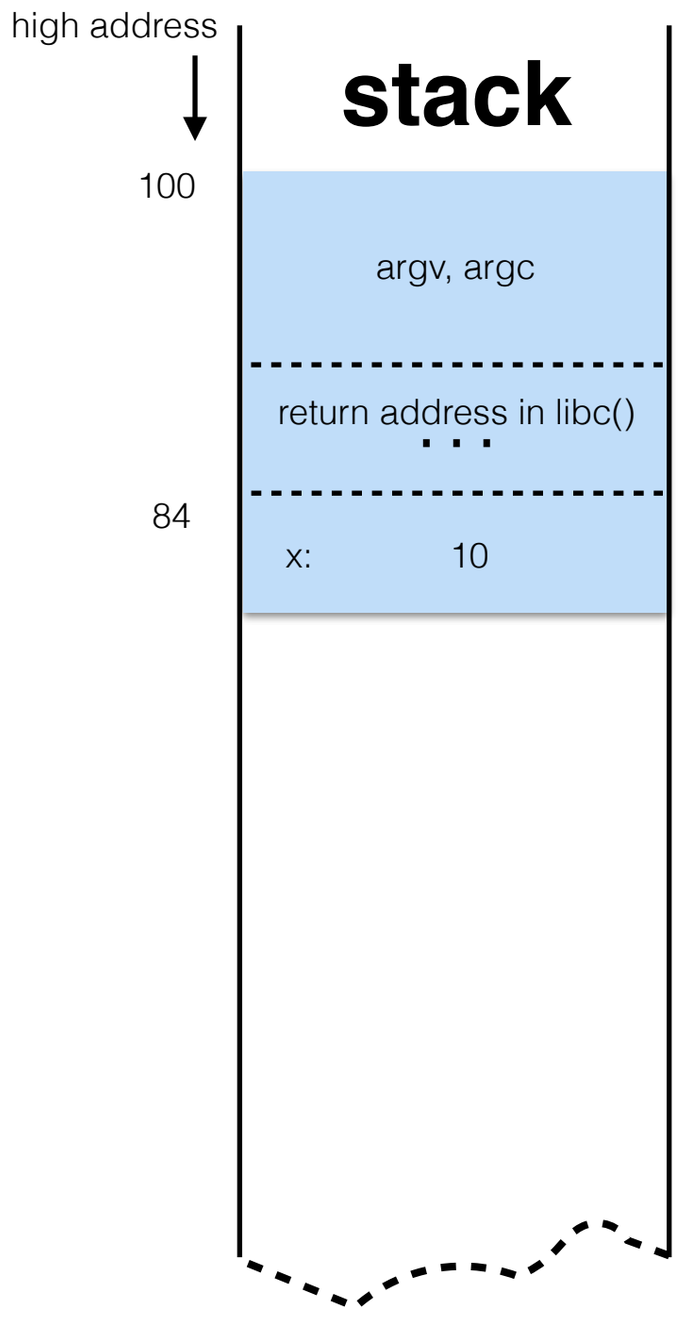
```
void fun(int * a) {
    *a = 3;
}
```

```
int main(..) {
    int x = 10;
    fun(&x);
}
```

To change a, fun() must get its address.

```
void fun(int * a) {
    *a = 3;
}
```

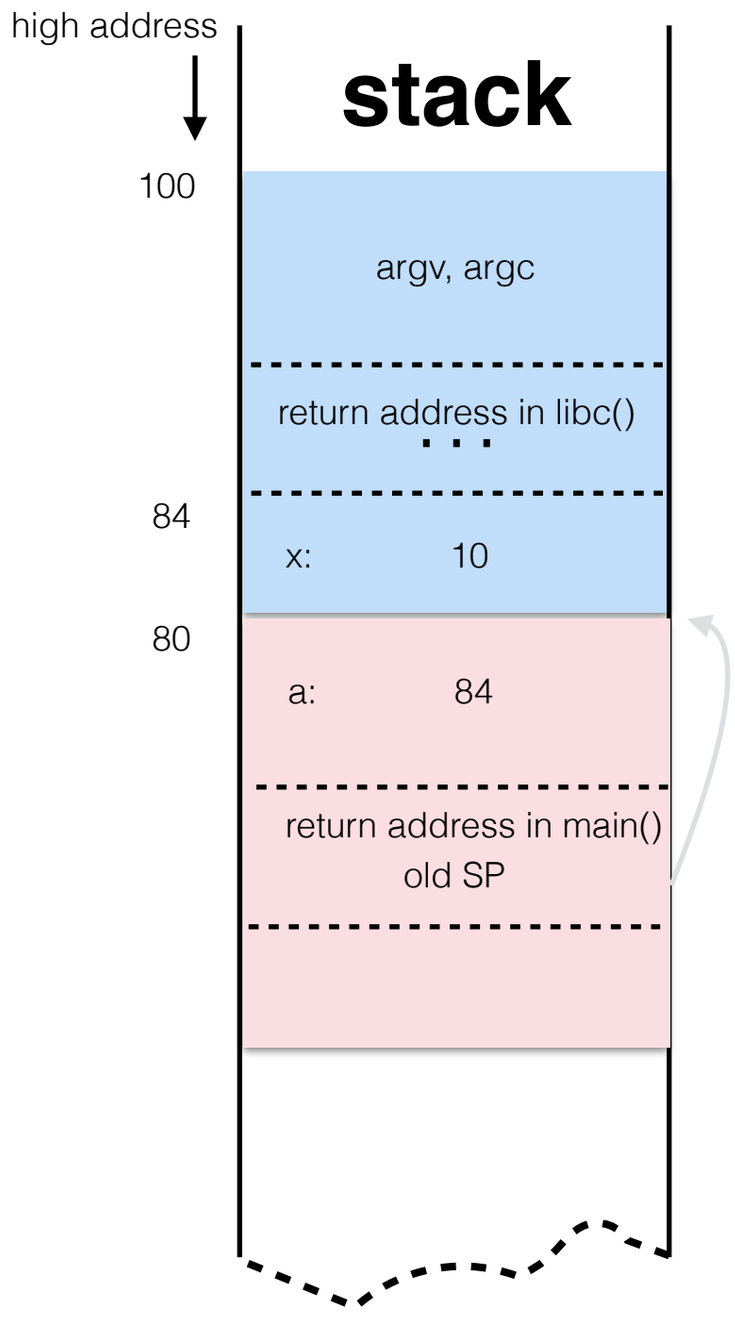
```
int main(..) {
    int x = 10;
    fun(&x);
}
```



To change a, fun() must get its address.

```
void fun(int * a) {
    *a = 3;
}
```

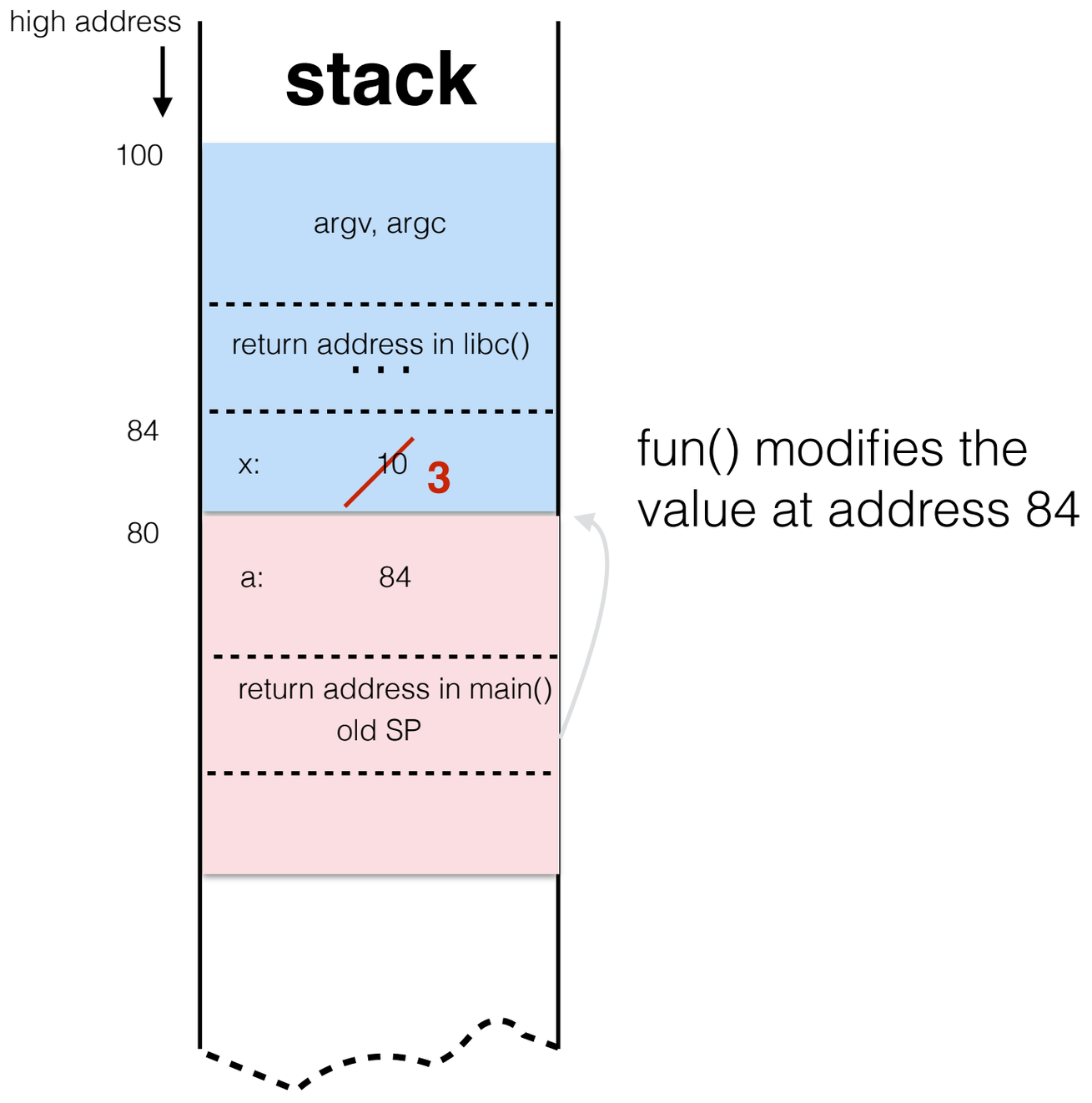
```
int main(..) {
    int x = 10;
    fun(&x);
}
```



To change a, fun() must get its address.

```
void fun(int * a) {
    *a = 3;
}
```

```
int main(..) {
    int x = 10;
    fun(&x);
}
```

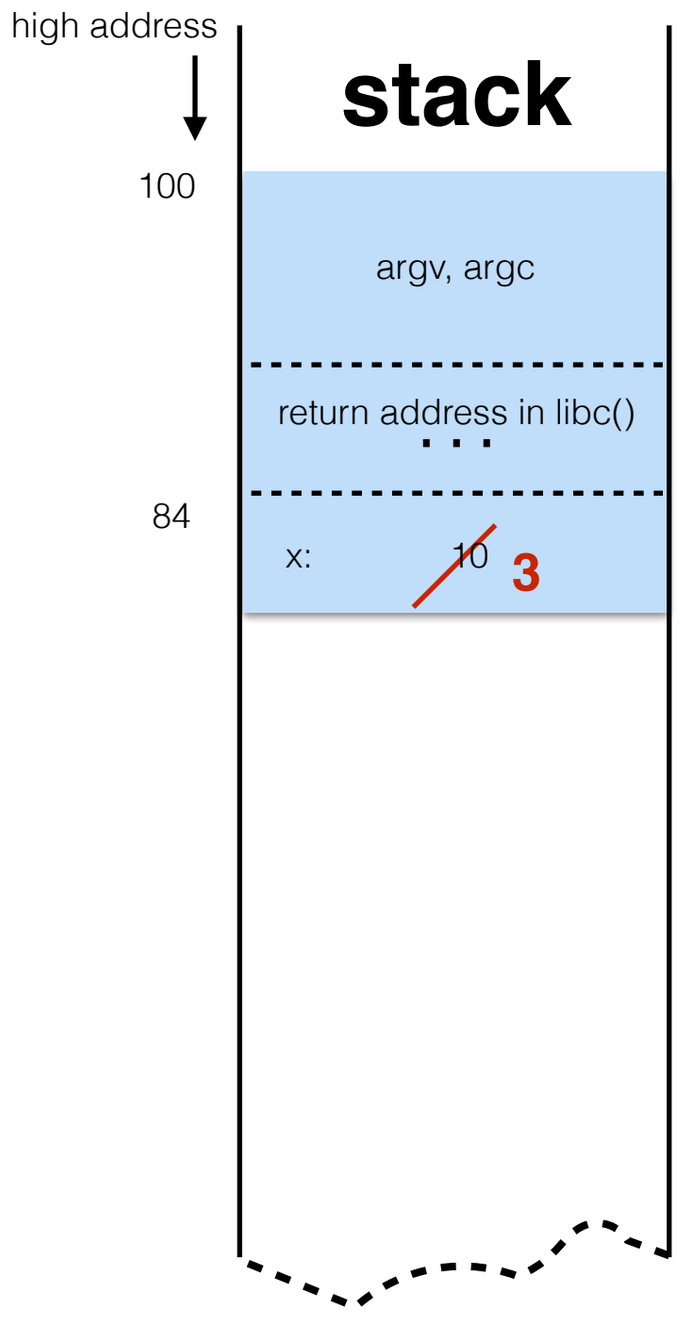


fun() modifies the value at address 84

To change a, fun() must get its address.

```
void fun(int * a) {
    *a = 3;
}
```

```
int main(..) {
    int x = 10;
    fun(&x);
}
```



Review

```
void fun(int a) {
    a = 3;
}
```

```
int main(..) {
    int x = 10;
    fun(x);
}
```

x is not changed

Review

```
void fun(int a) {
    a = 3;
}
```

```
int main(..) {
    int x = 10;
    fun(x);
}
```

x is not changed

Review

```
void fun(T a) {
    a = value of type T;
}
```

```
int main(..) {
    T x = value of type T;
    fun(x);
}
```

x is not changed

T is int*

```
void fun(T a) {
    a = value of type T;
}
```

```
void fun(int* a) {
    a = malloc(..);
}
```

```
int main(..) {
    T x = value of type T;
    fun(x);
}
```

x is not changed

```
int main(..) {
    int* x = NULL;
    fun(x);
}
```

x is not changed

To change a, fun() must get its address.

```
void fun(int* a) {
    a = malloc(..);
}
```

```
int main(..) {
   int* x = NULL;
   fun(x);
}
```

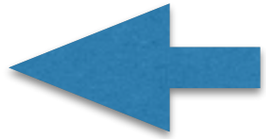
x is not changed

To change a, fun() must get its address.

```
void fun(int** a) {
    *a = malloc(..);
}
```

```
void fun(int* a) {
    a = malloc(..);
}
```

```
int main(..) {
    int* x = NULL;
    fun(&x);
}
```



```
int main(..) {
    int* x = NULL;
    fun(x);
}
```

x is not changed

Let's see how it works, again.

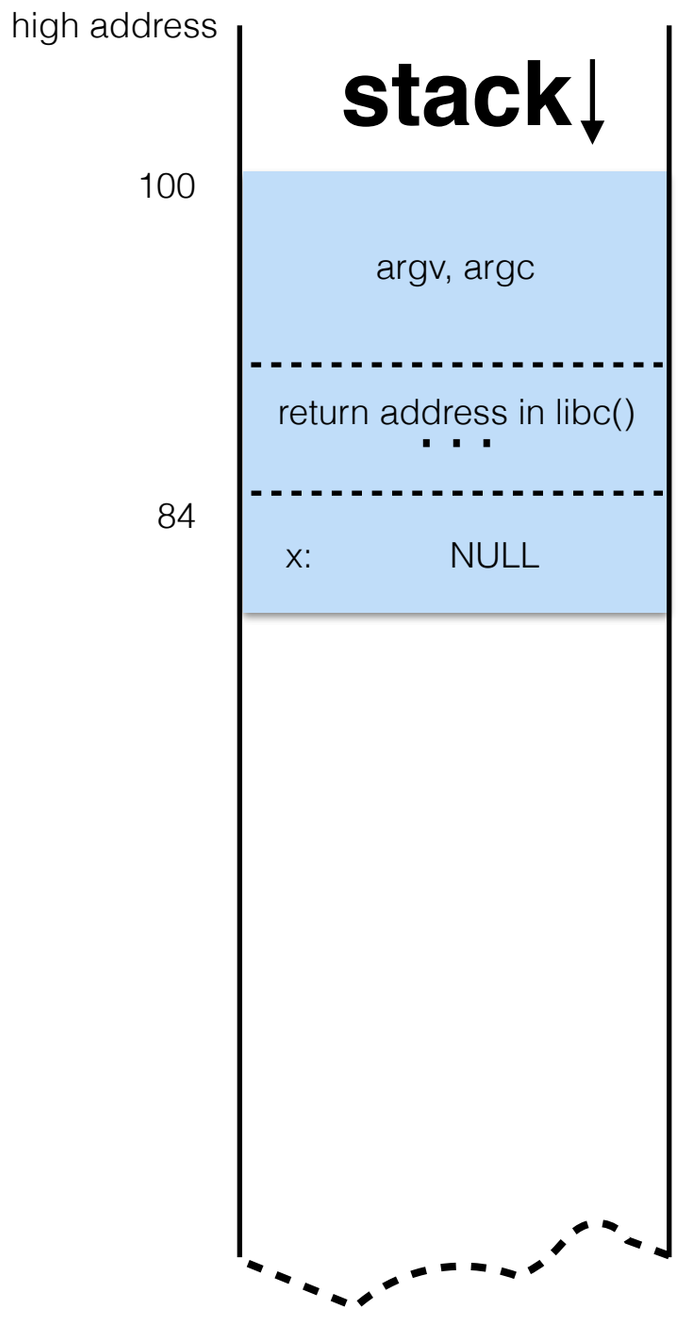
```
void fun(int** a) {
    *a = malloc(..);
}
```

```
int main(..) {
    int* x = NULL;
    fun(&x);
}
```

Let's see how it works, again.

```
void fun(int** a) {
    *a = malloc(..);
}
```

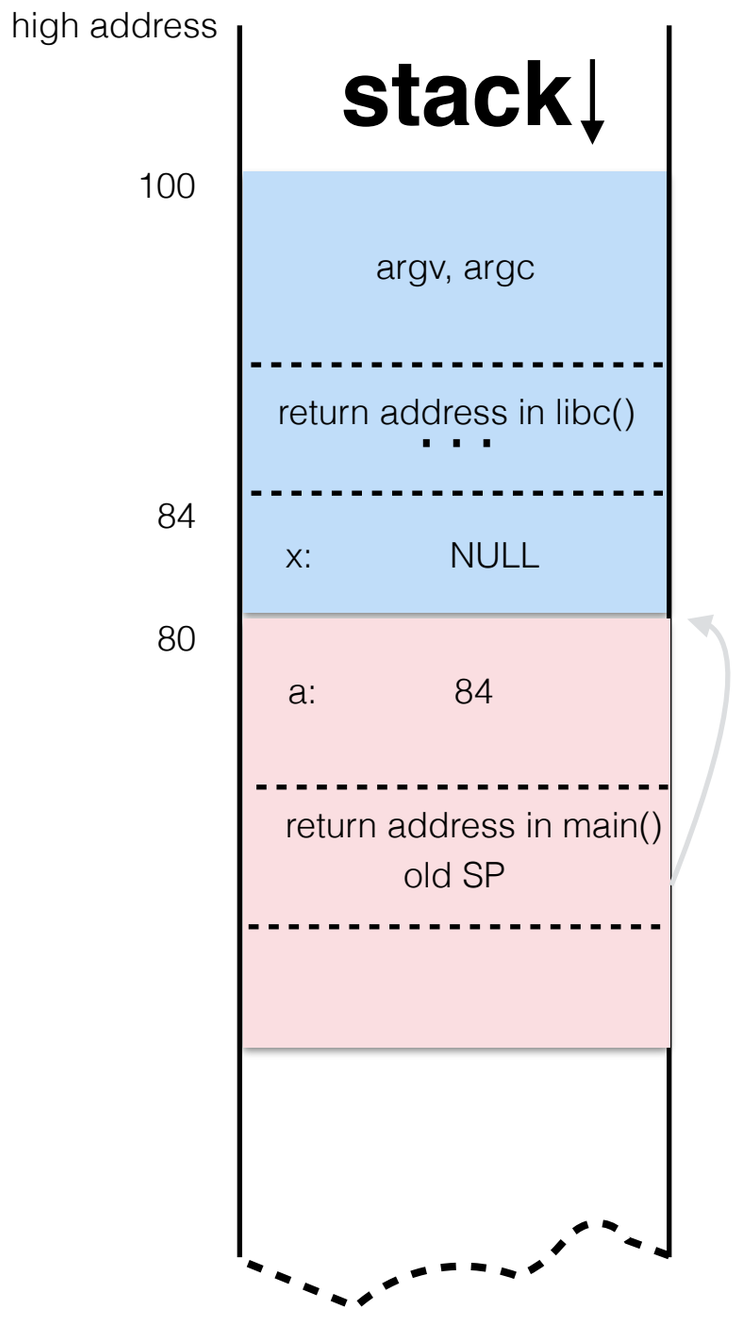
```
int main(..) {
    int* x = NULL;
    fun(&x);
}
```



Let's see how it works, again.

```
void fun(int** a) {
    *a = malloc(..);
}
```

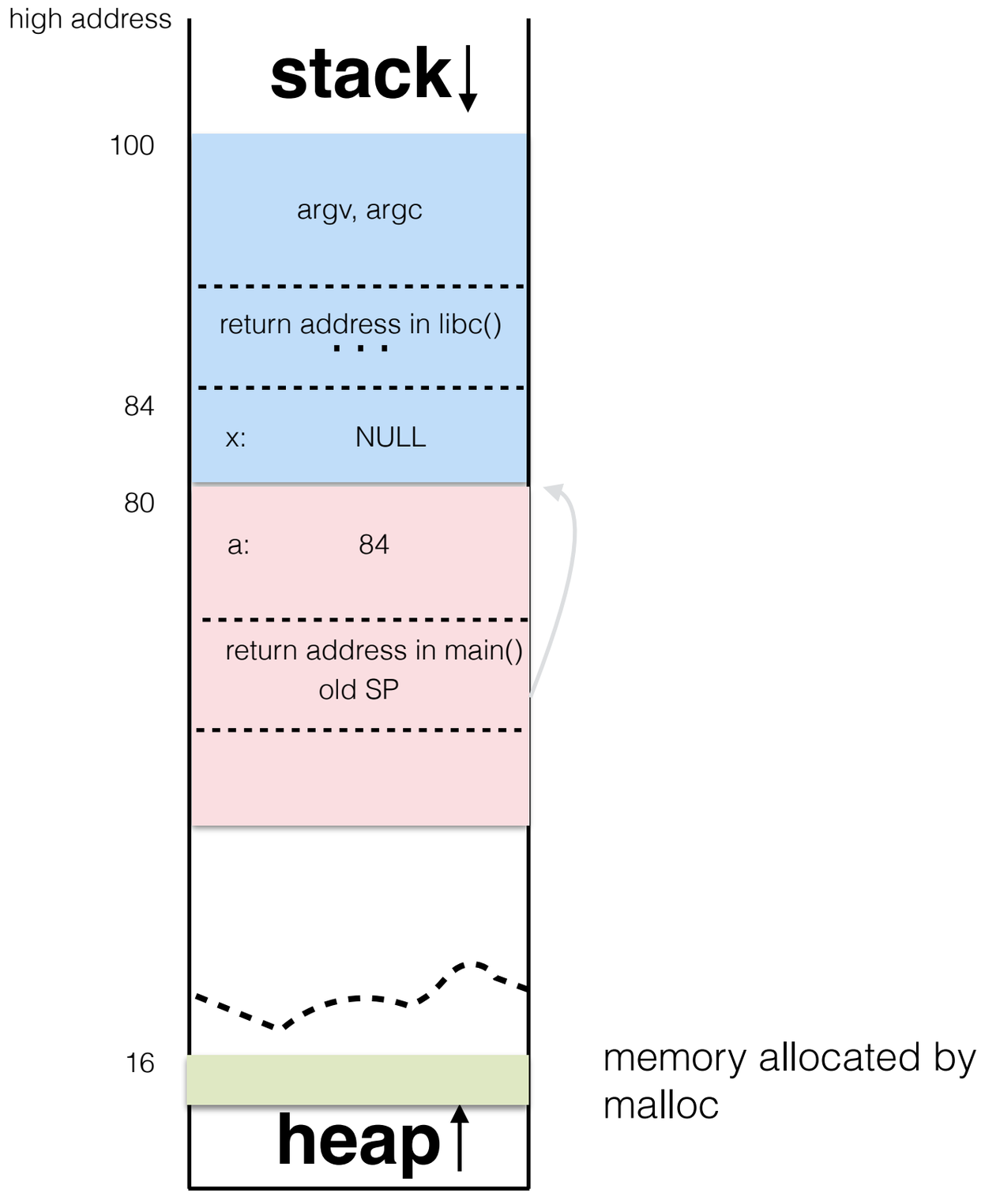
```
int main(..) {
    int* x = NULL;
    fun(&x);
}
```



Let's see how it works, again.

```
void fun(int** a) {
    *a = malloc(..);
}
```

```
int main(..) {
    int* x = NULL;
    fun(&x);
}
```



Let's see how it works, again.

```
void fun(int** a) {
    *a = malloc(..);
}
```

```
int main(..) {
    int* x = NULL;
    fun(&x);
}
```