Call “Stack”

CS 1037a – Topic 7

Memory Organization in a Running Program

- Memory available to a running program is divided into three parts:
  - The portion that holds the machine-language version of the program
  - The *heap*: a pool of memory used for dynamic allocation of objects
  - The *call stack*: used for all parameters, local variables, statically-allocated data

Memory Organization in a Running Program

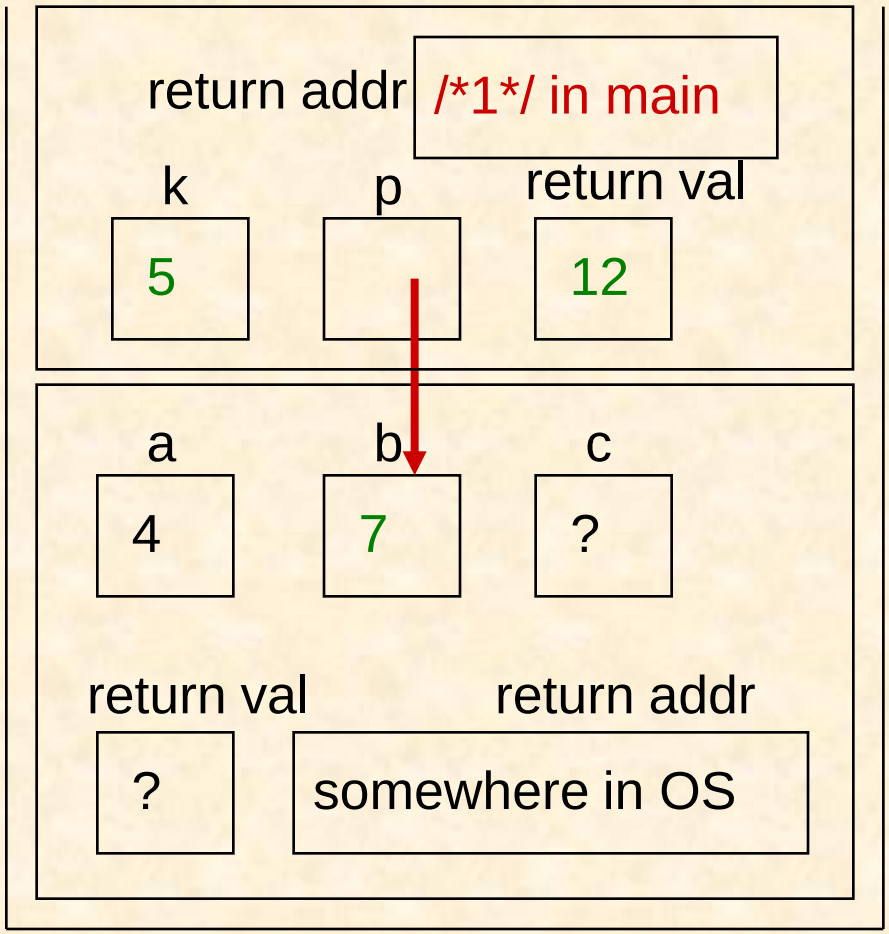
- Program section is protected so that it cannot accidentally be overwritten
- A special *register* in the CPU called the *program counter (PC)* holds the memory address of the next program instruction to be executed

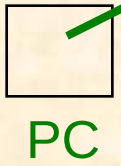
Call Stack

- A stack data structure
- Each item on the call stack is known as a *stack frame* (or *call frame*)
- Call stack manages all function calls and returns
- Top stack frame determines what variables, objects are currently accessible

Call Stack Example 1

After k+*p is evaluated, but before return is executed



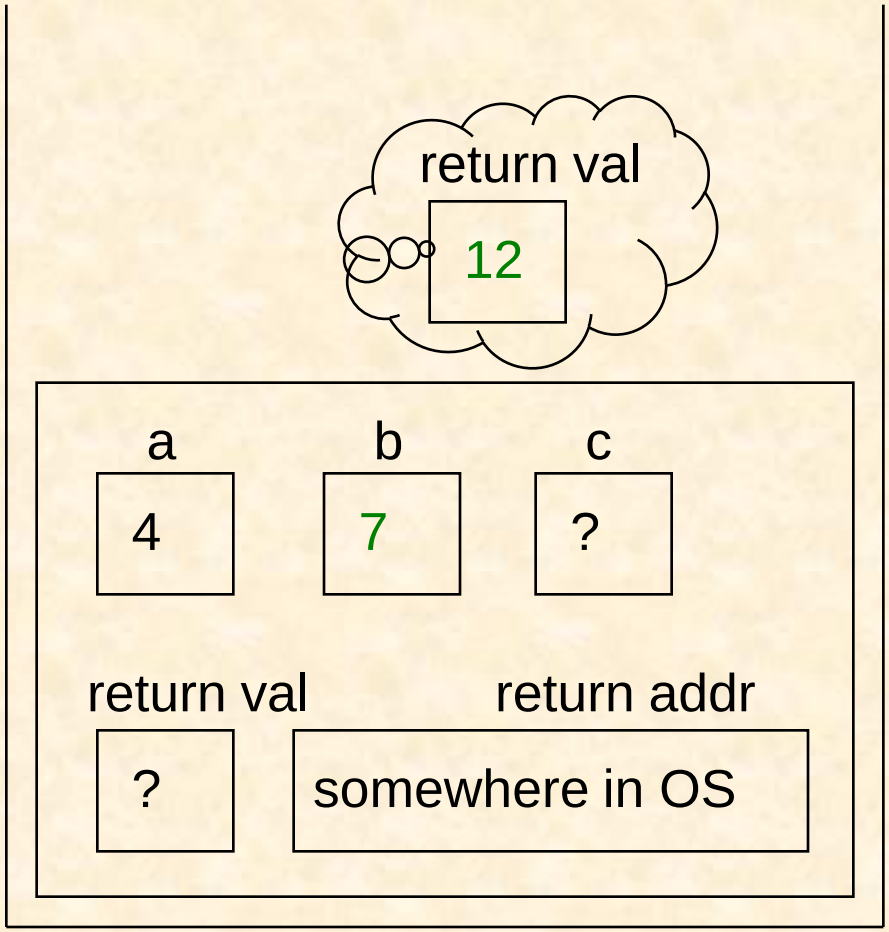



```
int fn( int k, int *p) {
    k = k + 1;
    *p = *p + 1;
    return k + *p;
}

int main( void ) {
    int a = 4, b = 6, c;
    c = /*1*/ fn( a, &b);
    // rest of program…
```

Call Stack Example 1

After return is executed, but before assignment statement finishes executing

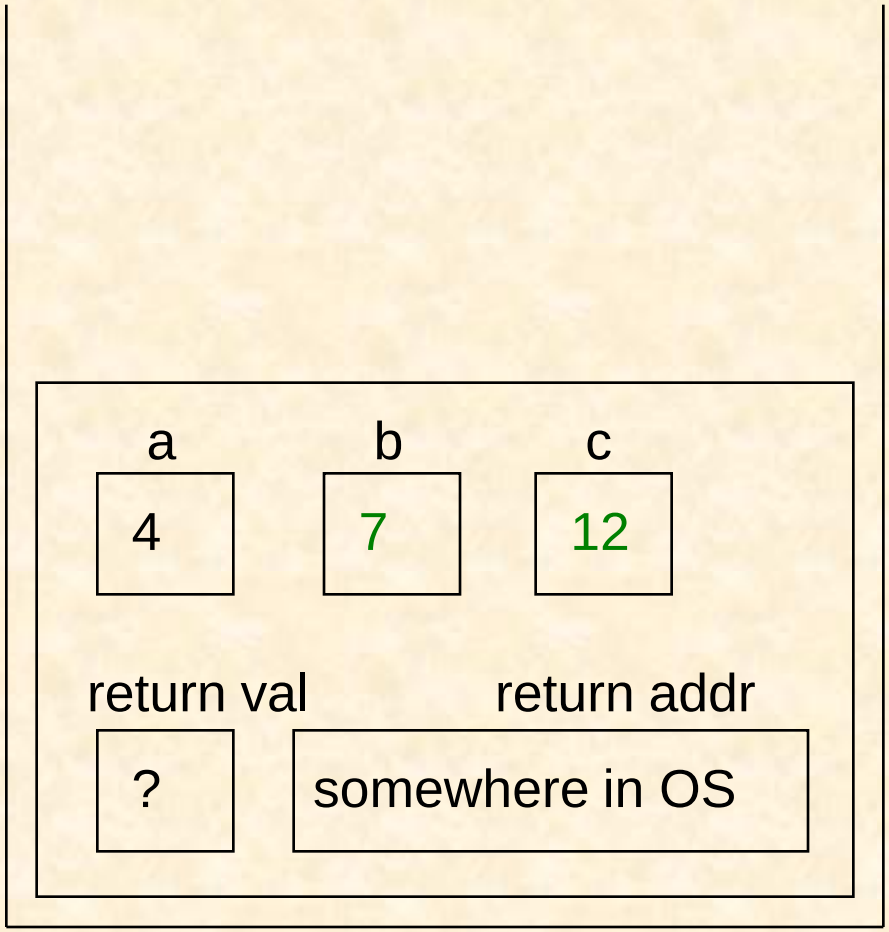


PC

```
int fn( int k, int *p) {
    k = k + 1;
    *p = *p + 1;
    return k + *p;
}
int main( void ) {
    int a = 4, b = 6, c;
    c = /*1*/ fn( a, &b);
    // rest of program…
```

Call Stack Example 1

After c=fn(a,&b); is executed



PC

```
int fn( int k, int *p) {
    k = k + 1;
    *p = *p + 1;
    return k + *p;
}
int main( void ) {
    int a = 4, b = 6, c;
    c = /*1*/ fn( a, &b);
    // rest of program…
```

Call Stack vs. Heap Memory

- Call stack is stored *contiguously* in memory (i.e.: in consecutive memory locations)
- “Popped” stack frame will be overwritten the next time a function is called
- Thus, values in a stack frame are accessible only while the corresponding function is being executed

Call Stack vs. Heap Memory

- In contrast, memory allocated from the heap is *persistent*: remains allocated for the entire program run, unless it is deallocated by, say, calling a destructor

Call Stack Example 2

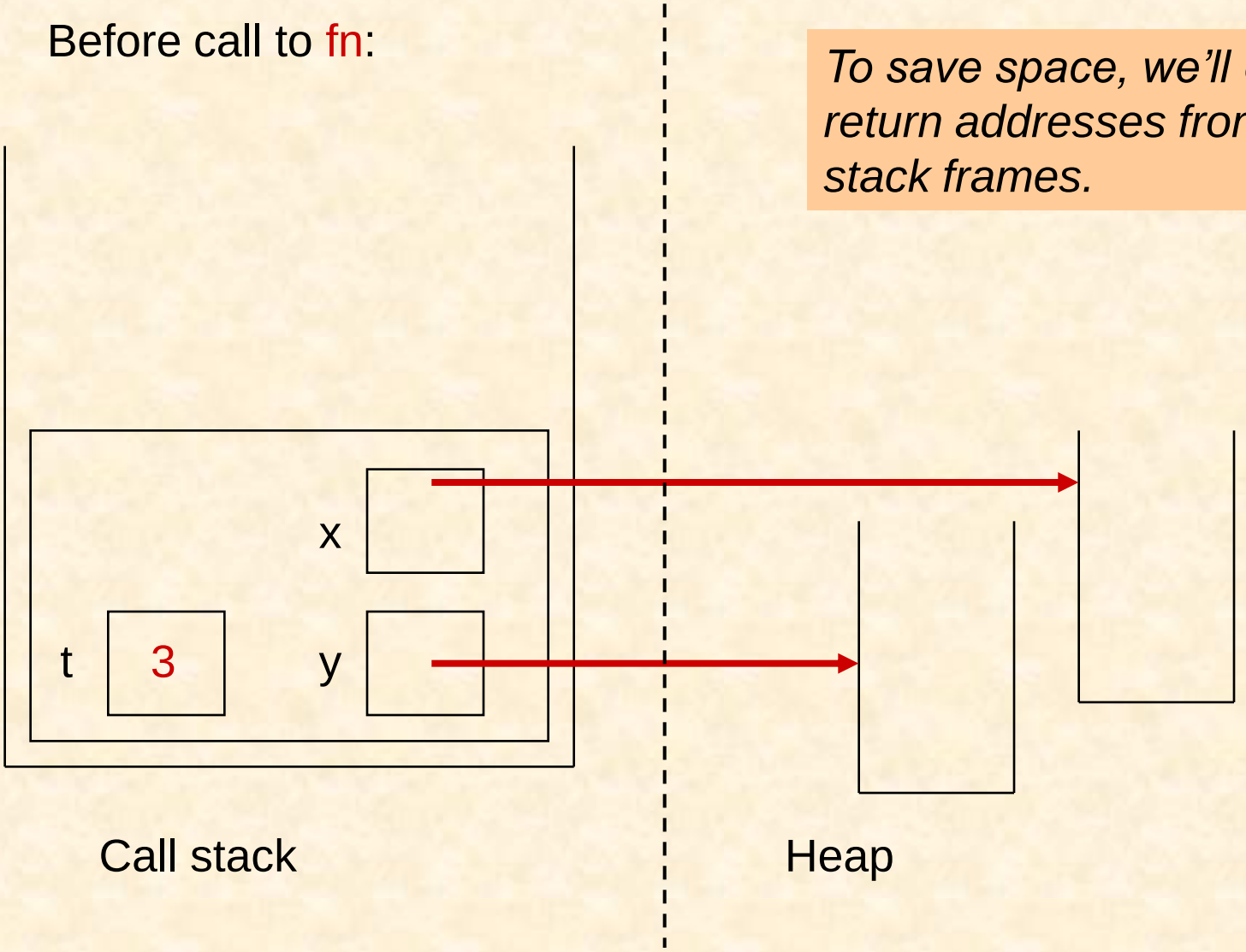
```
#include “Stack.h”
 void fn( Stack<int*> *s1,
        Stack<int*> *s2, int n) {
     int *j, *k;
     j = new int(n);
     s1->push( j );
     s2 = new Stack<int*>;
     n++;
     k = new int(n);
     s2->push( k );
}         // end of function fn
```

```
int main (void) {
     int t = 3;
     Stack<int*> * x = new Stack<int*>;
     Stack<int*> * y = new Stack<int*>;
     fn( x, y, t );
     // rest of main …
```

Call Stack Example 2

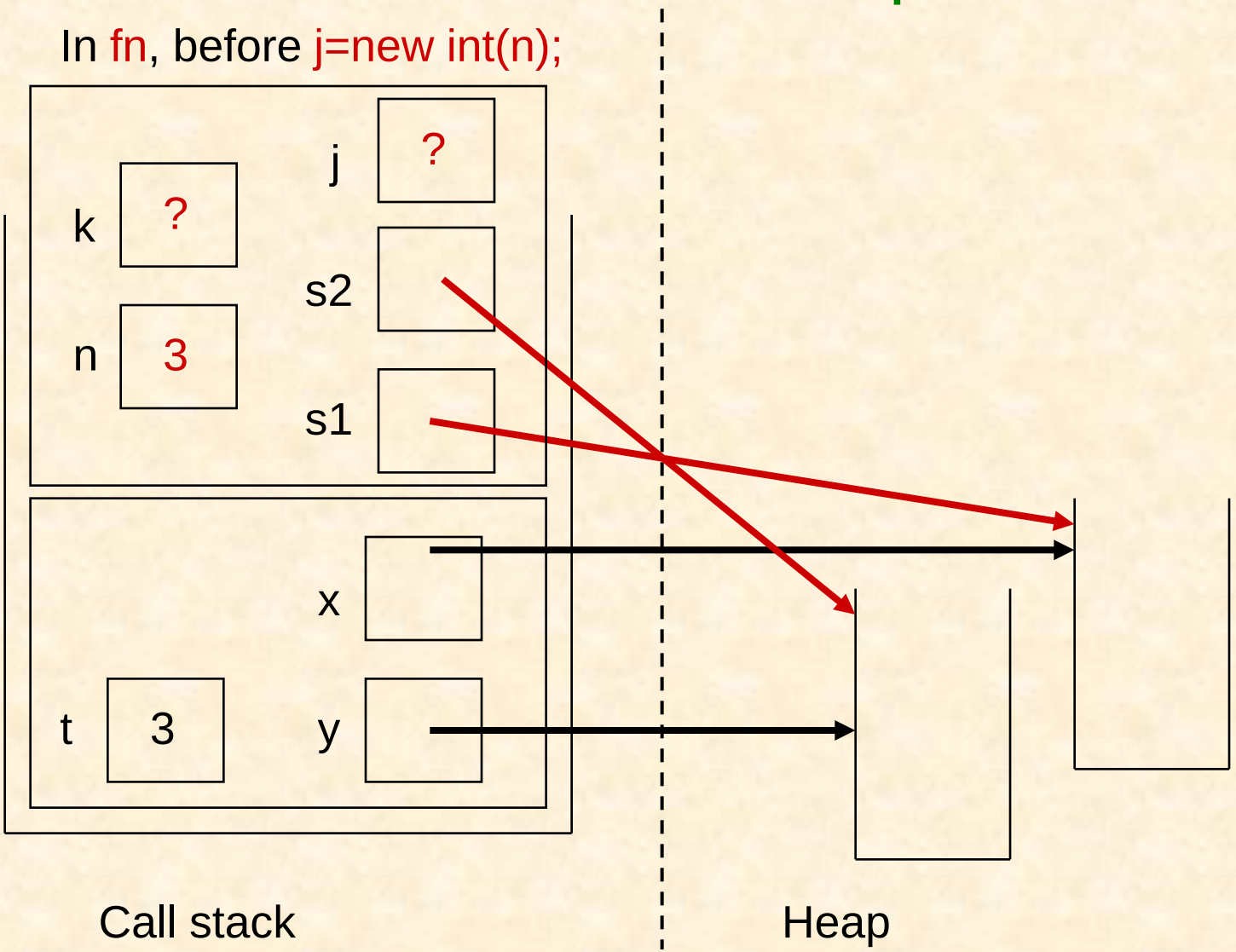
Before call to fn:

To save space, we'll omit return addresses from the stack frames.

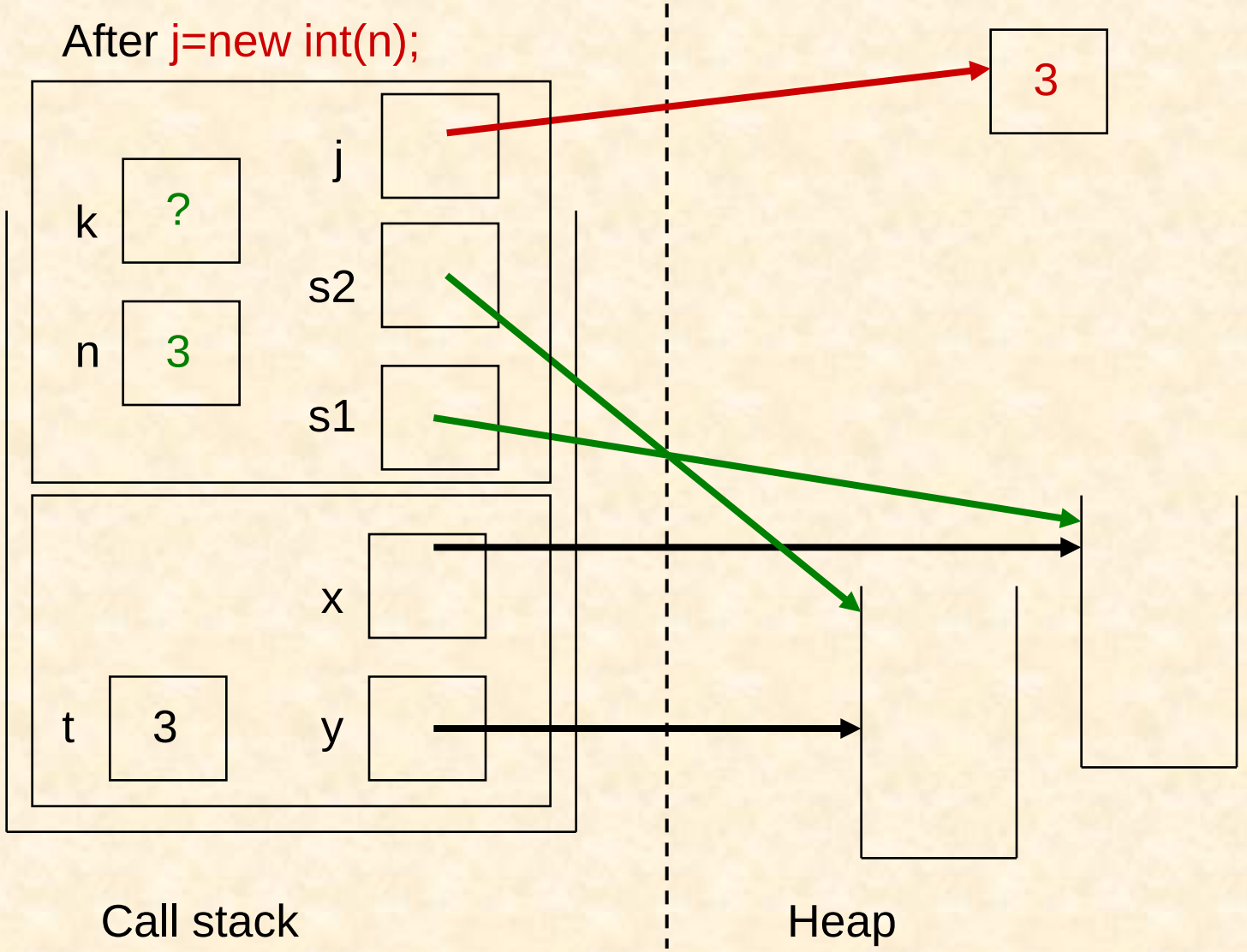


Call Stack Example 2

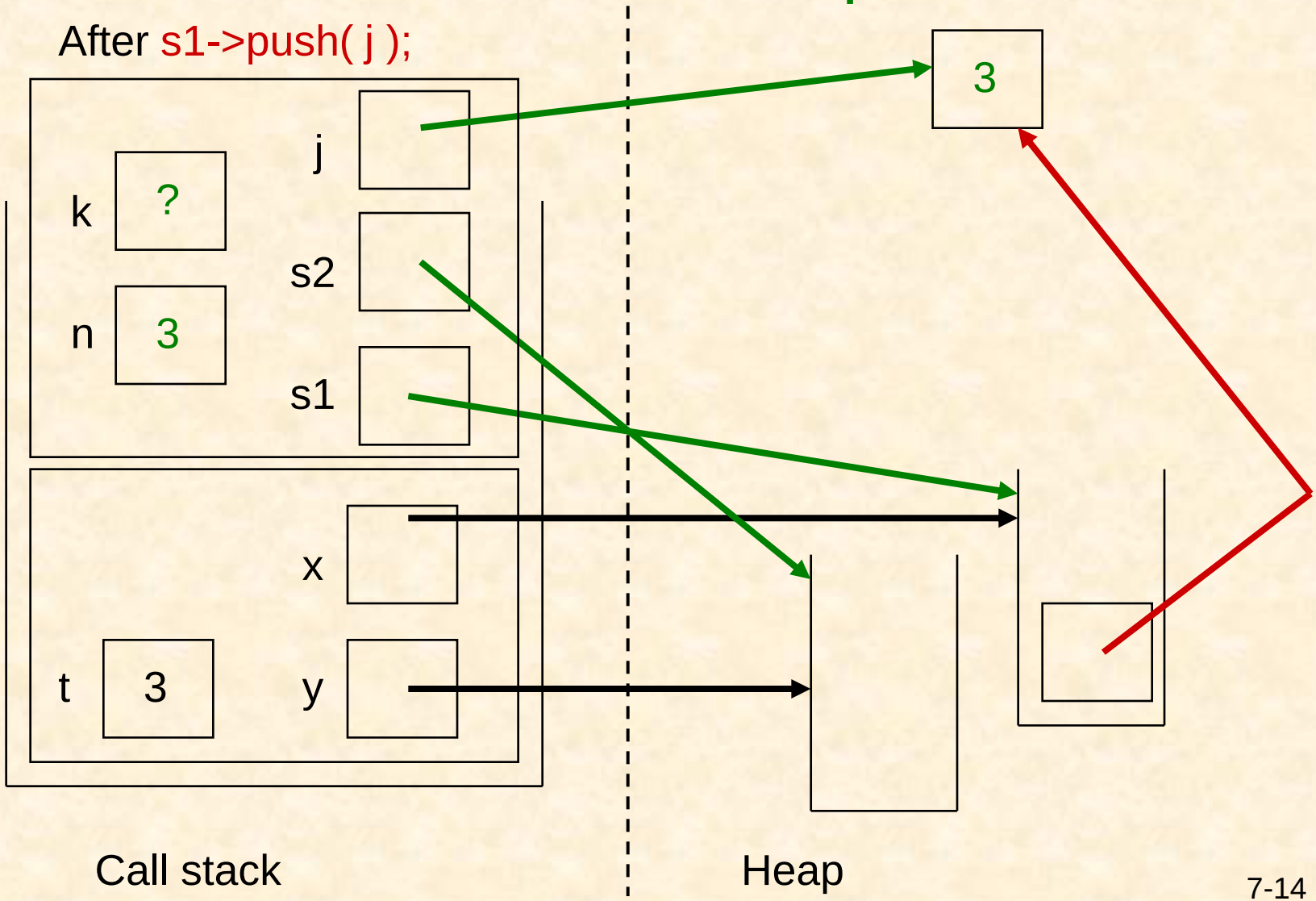
In fn, before j=new int(n);



Call Stack Example 2

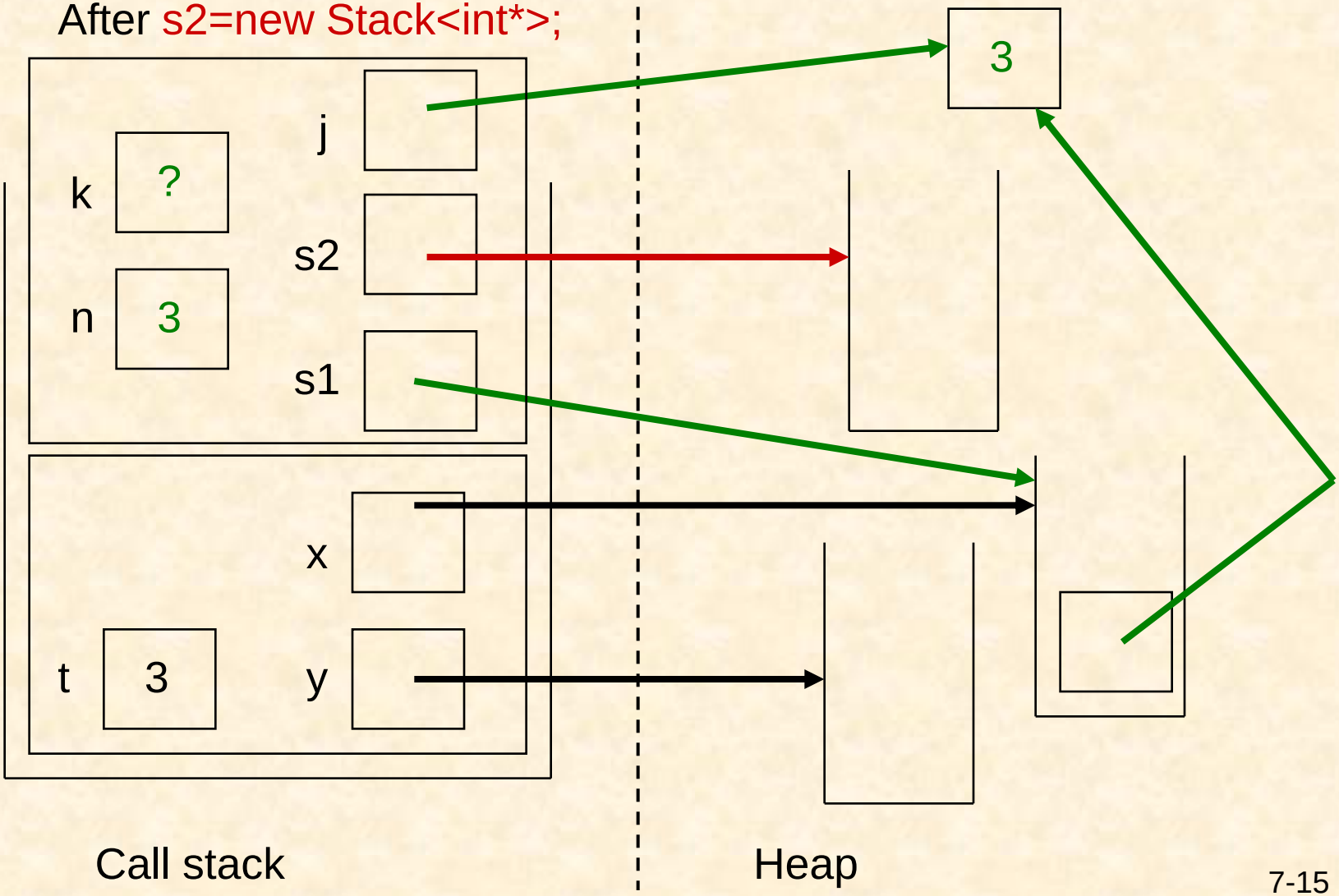


Call Stack Example 2

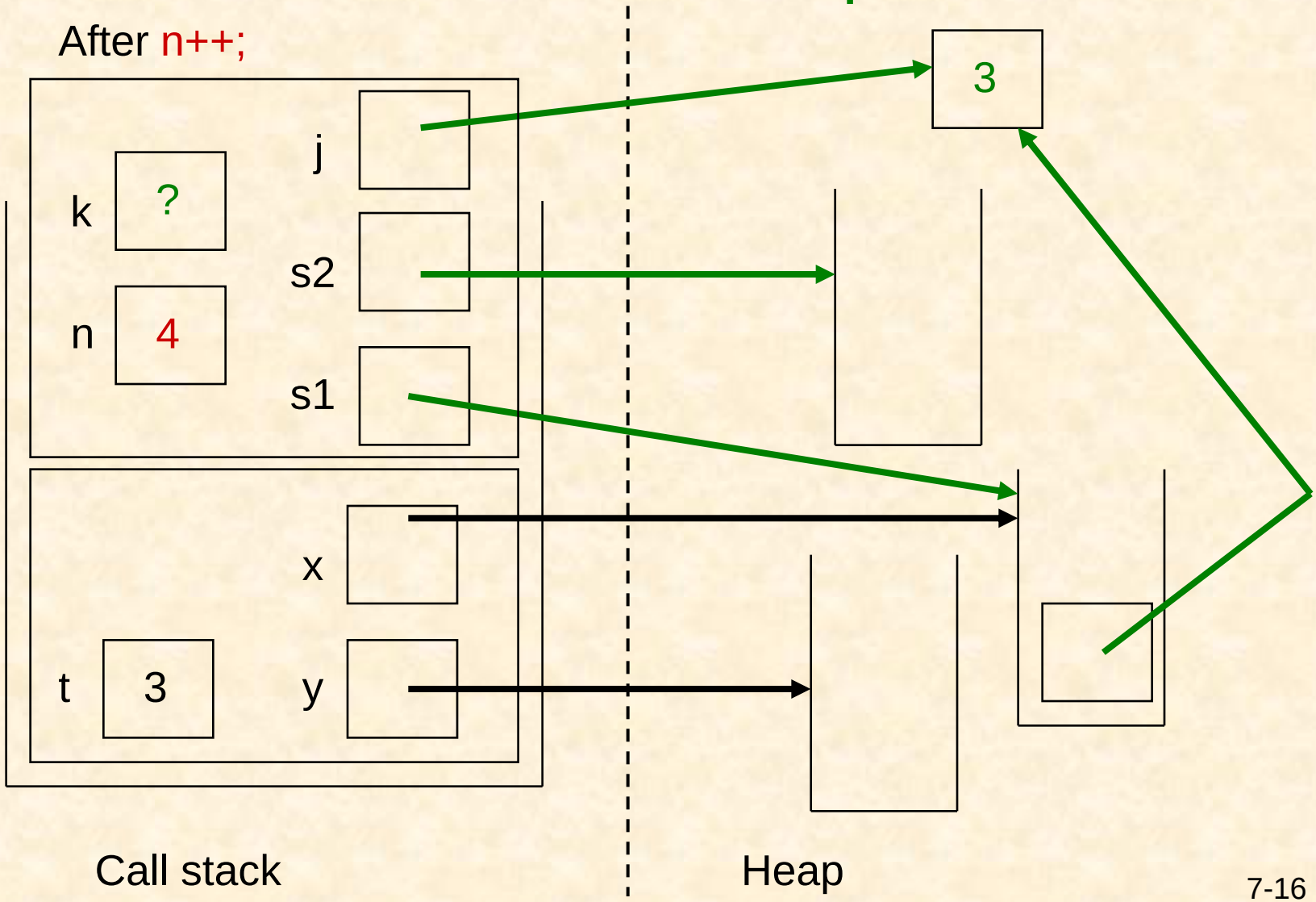


Call Stack Example 2

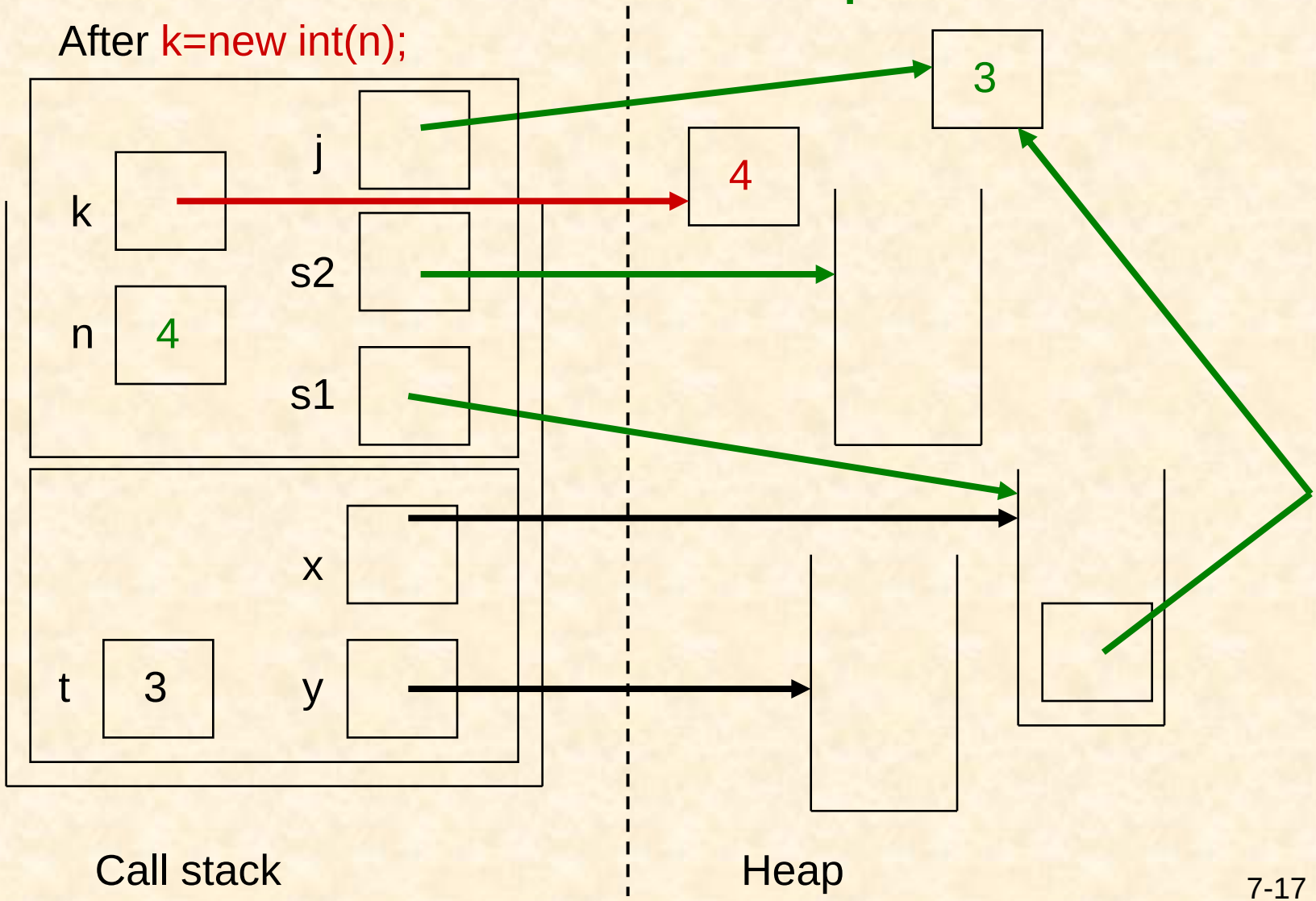
After s2=new Stack<int*>;



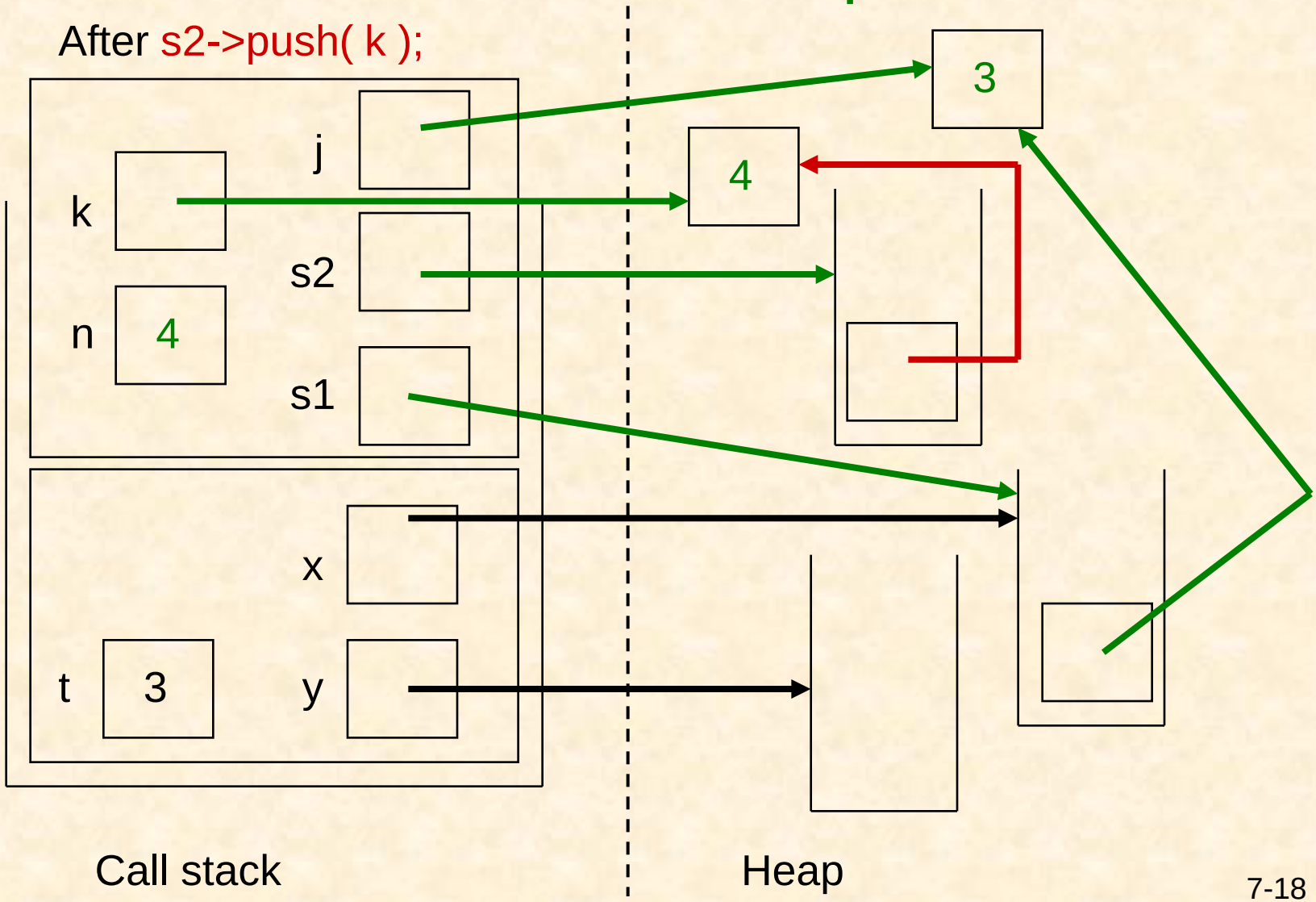
Call Stack Example 2



Call Stack Example 2

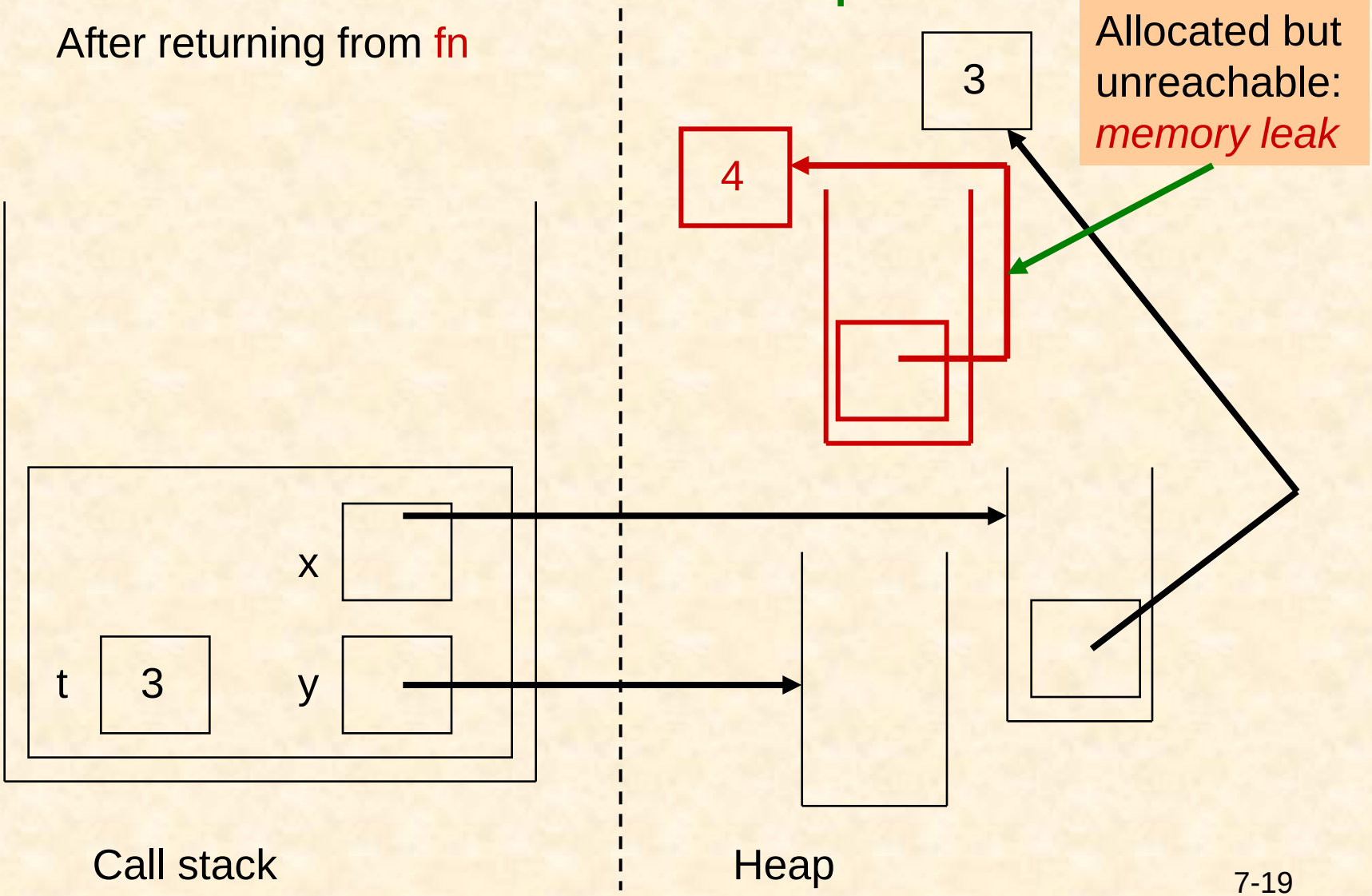


Call Stack Example 2



Call Stack Example 2

After returning from fn



Things You Can't Do

- Exercise: Explain what's wrong with the statements in the following function that are in colour:

```
int * fn ( int ** n ) {
    int j = 7, k = 5;
    *n = &j;
    return &k;
}
```