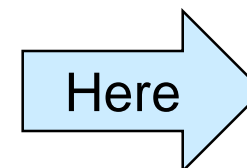
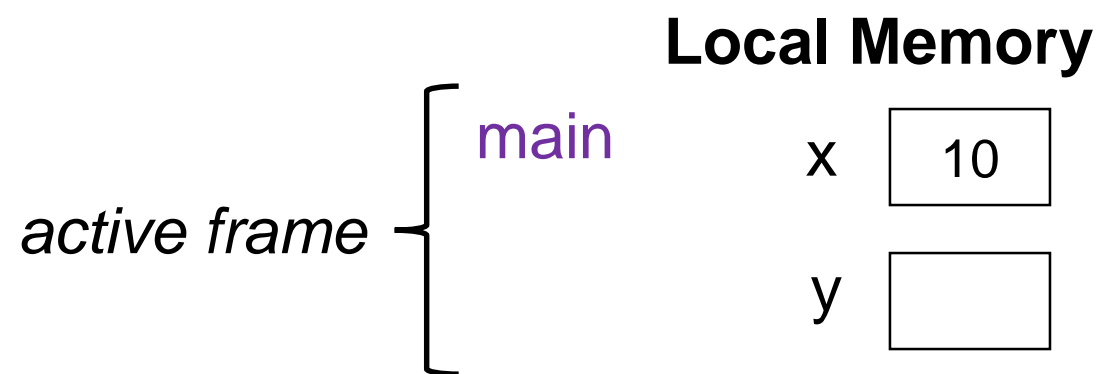


# Arrays

# Memory Model

# C0 Memory Model

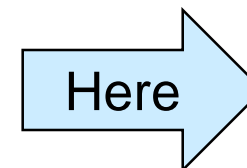
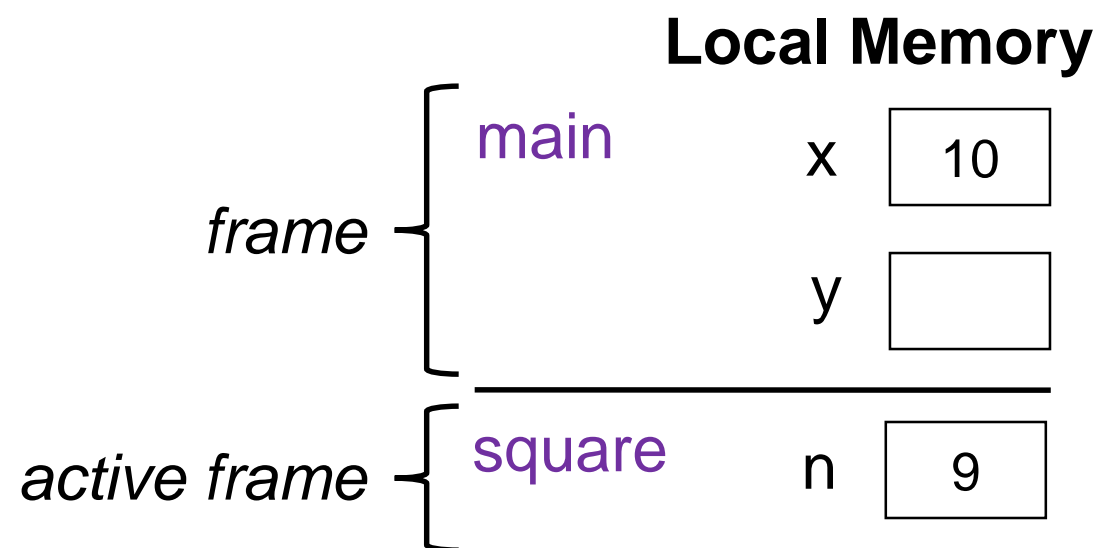
- Variables live in **local memory**
  - The variables of a function are grouped in a **frame**



```
int POW(int x, int y) {  
    if (y == 0) return 1;  
    return x * POW(x, y-1);  
}  
  
int square(int n) {  
    return n * n;  
}  
  
int main() {  
    int x = 10;  
    int y = square(x - 1);  
    //@assert y == POW(x-1,2);  
    return y;  
}
```

# C0 Memory Model

- Each function currently being called has its own frame
  - A function can only manipulate the variables in its frame

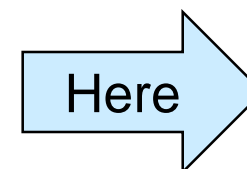
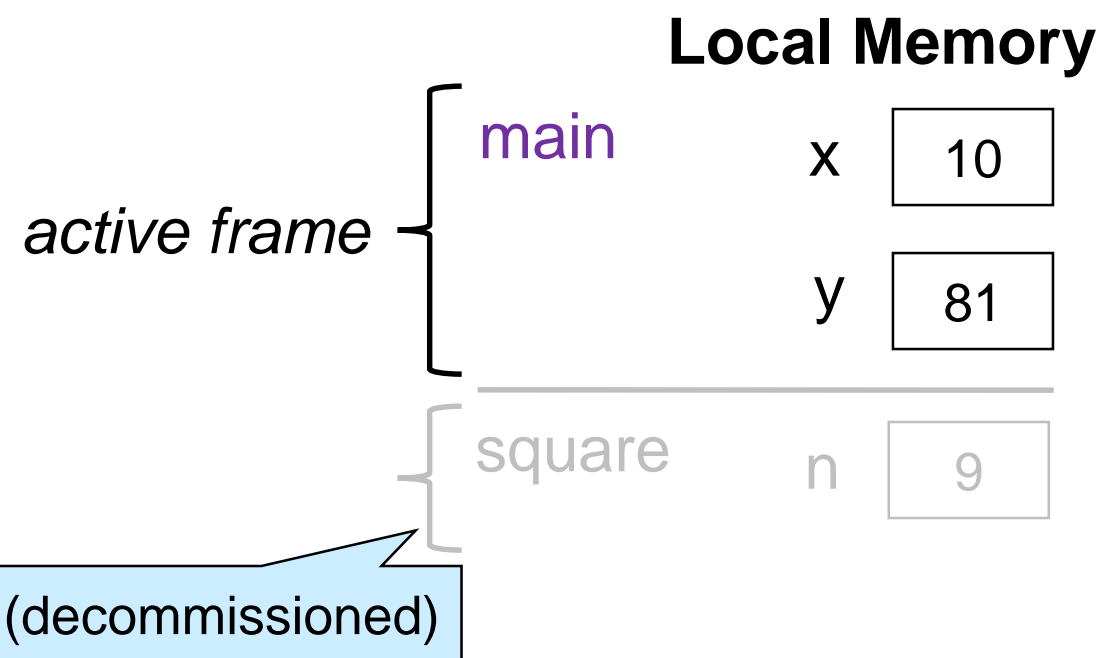


The only variable  
square can use is n

```
int POW(int x, int y) {  
    if (y == 0) return 1;  
    return x * POW(x, y-1);  
}  
  
int square(int n) {  
    return n * n;  
}  
  
int main() {  
    int x = 10;  
    int y = square(x-1);  
    //@assert y == POW(x-1,2);  
    return y;  
}
```

# C0 Memory Model

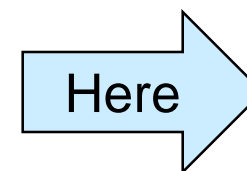
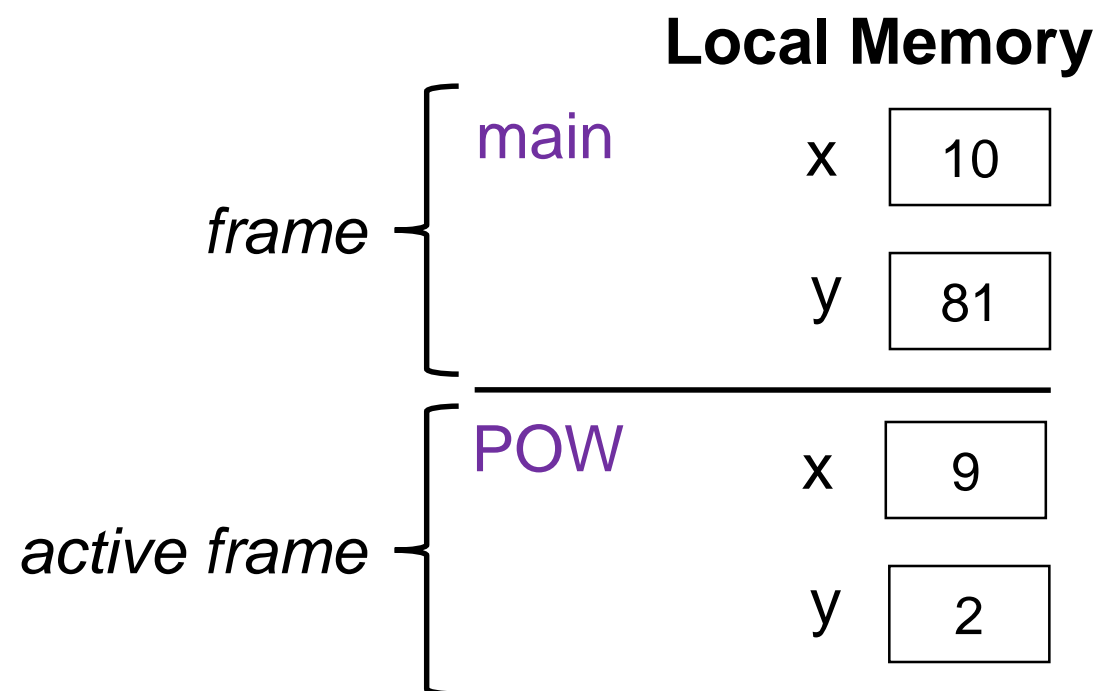
- The frame is de commissioned when the function returns



```
int POW(int x, int y) {  
    if (y == 0) return 1;  
    return x * POW(x, y-1);  
}  
  
int square(int n) {  
    return n * n;  
}  
  
int main() {  
    int x = 10;  
    int y = square(x - 1);  
    //@assert y == POW(x-1,2);  
    return y;  
}
```

# C0 Memory Model

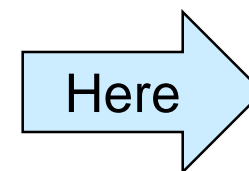
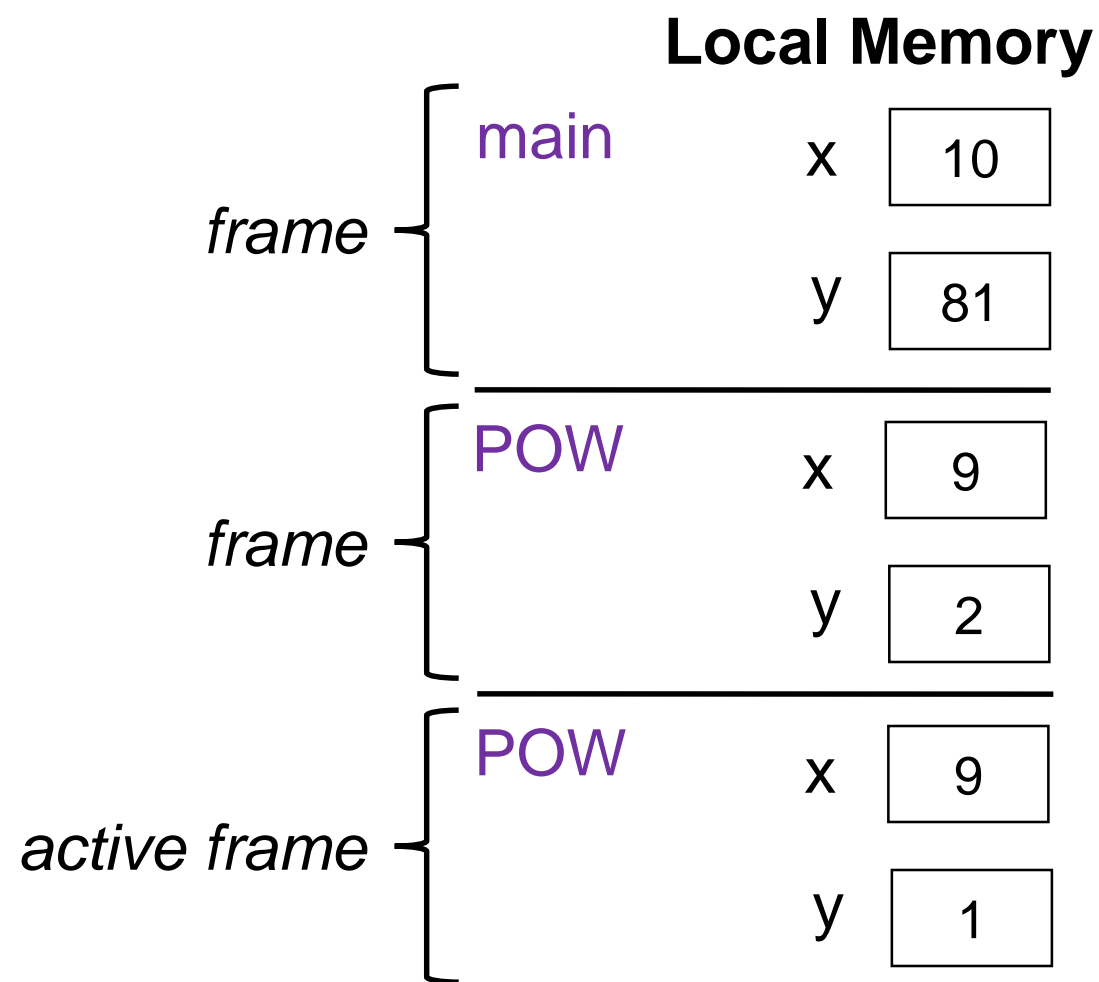
- The next function call adds a new frame
  - The variable names may be same as the caller
    - but the function can only manipulate the variables in **its** frame



```
int POW(int x, int y) {  
    if (y == 0) return 1;  
    return x * POW(x, y-1);  
}  
  
int square(int n) {  
    return n * n;  
}  
  
int main() {  
    int x = 10;  
    int y = square(x - 1);  
    //@assert y == POW(x-1,2);  
    return y;  
}
```

# C0 Memory Model

- The next function call adds a new frame
  - Recursive calls are treated the same way



```
int POW(int x, int y) {  
    if (y == 0) return 1;  
    return x * POW(x, y-1);  
}  
  
int square(int n) {  
    return n * n;  
}  
  
int main() {  
    int x = 10;  
    int y = square(x - 1);  
    //@assert y == POW(x-1,2);  
    return y;  
}
```

# Arrays



# Arrays

- Types so far
  - `int`, `bool`, `char`, `string`
- Arrays are collections of data of the same type
  - `int[]` is the type of arrays whose elements have type `int`
  - `string[]` is the type of arrays whose elements have type `string`
  - We can have arrays with elements of *any* type
    - even other arrays

# Creating an Array

- We create an array with

type of elements of the array

number of elements in the array

`alloc_array(int, 5)`

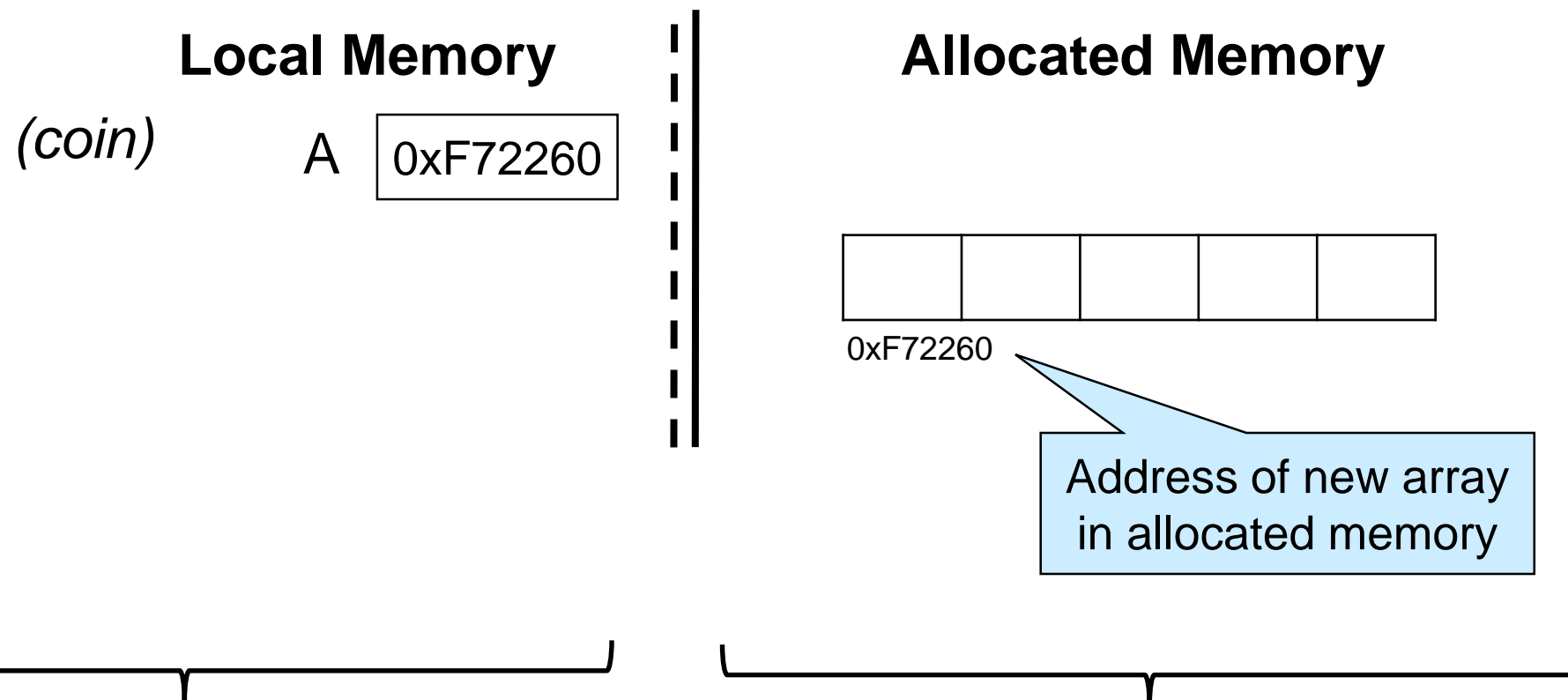
- This returns an `int[]`, an array of 5 `int`'s

```
Linux Terminal
# coin
C0 interpreter (coin) ...
...
--> int[] A = alloc_array(int, 5);
A is 0xF72260 (int[] with 5 elements)
-->
```

This is a **memory address**

# C0 Memory Model – Revisited

- Array contents live in **allocated memory**
  - A *new segment* of memory distinct from local memory
  - The variable *A* lives in local memory and
  - contains the address of the array in allocated memory

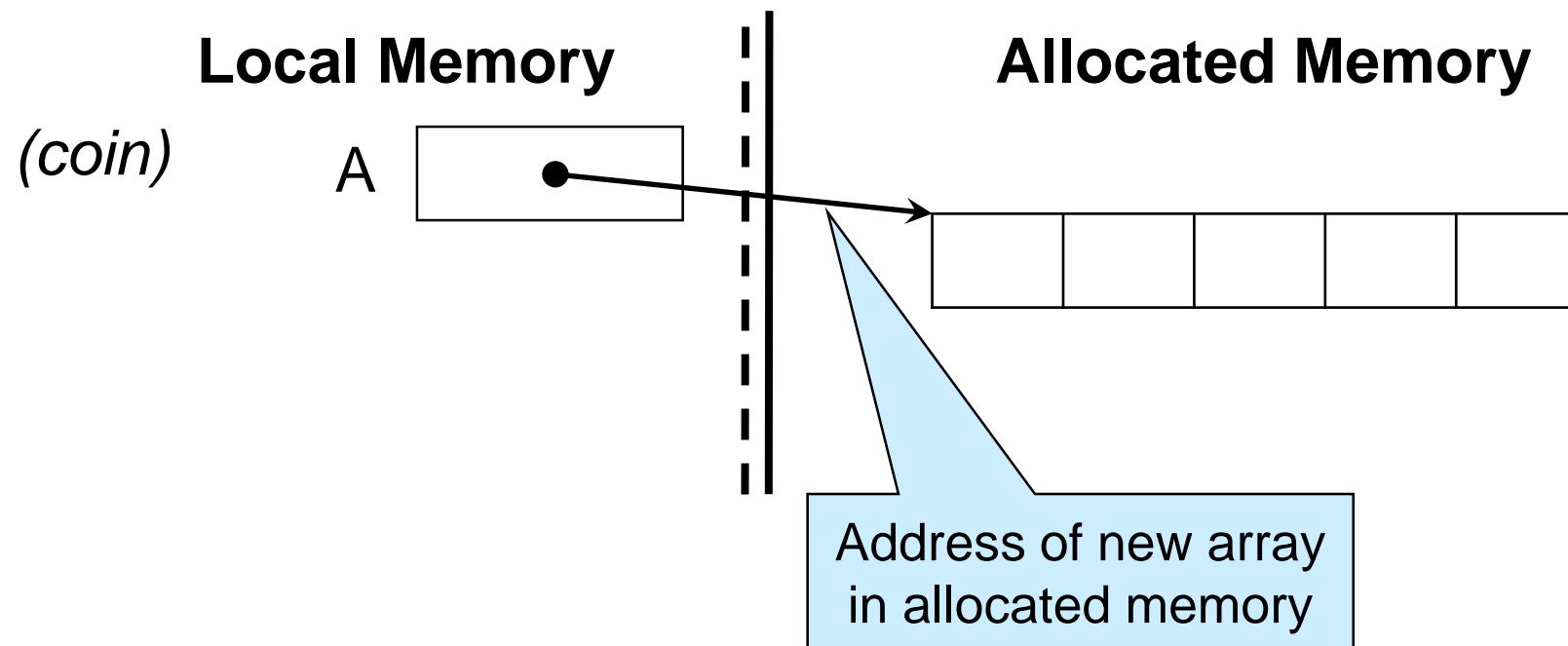


Contains the values of local variables  
*int*, *bool*, *char*, *string*, and **addresses**

Contains arrays themselves as we  
create them using ***alloc\_array***

# C0 Memory Model – Revisited

- Array addresses are invisible to the programmer
  - Except in coin
  - Different runs may result in different addresses
- We often abstract them as arrows

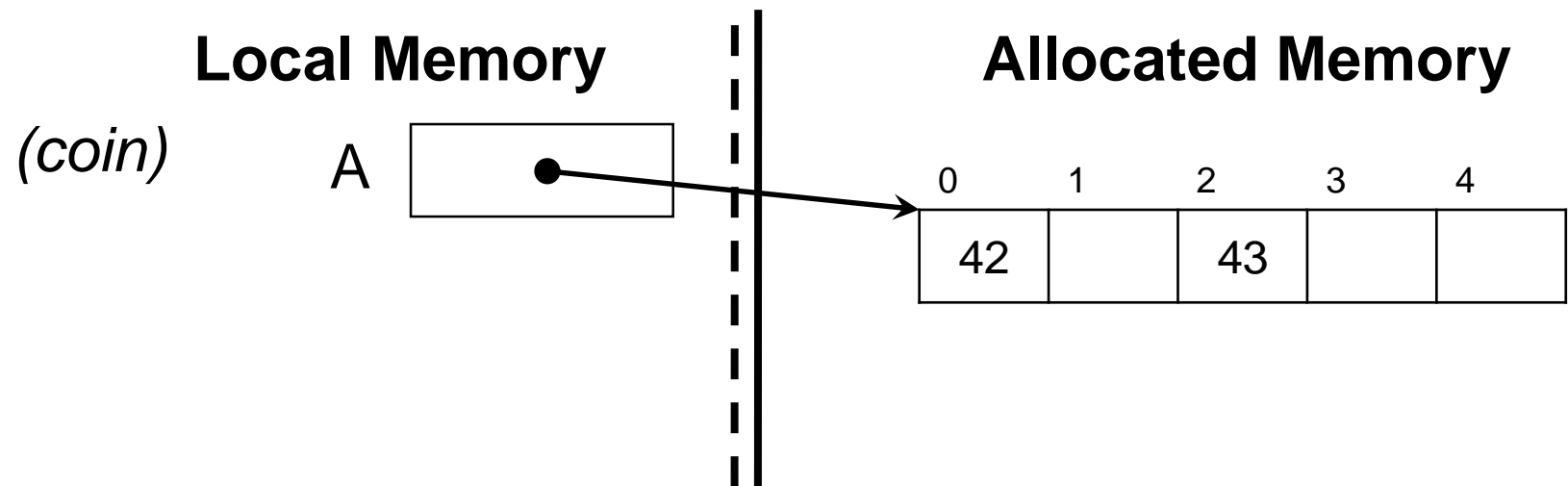


# Accessing Array Elements

- $i$ -th element of  $A$  is accessed as  $A[i]$ 
  - Indices start at 0

## Linux Terminal

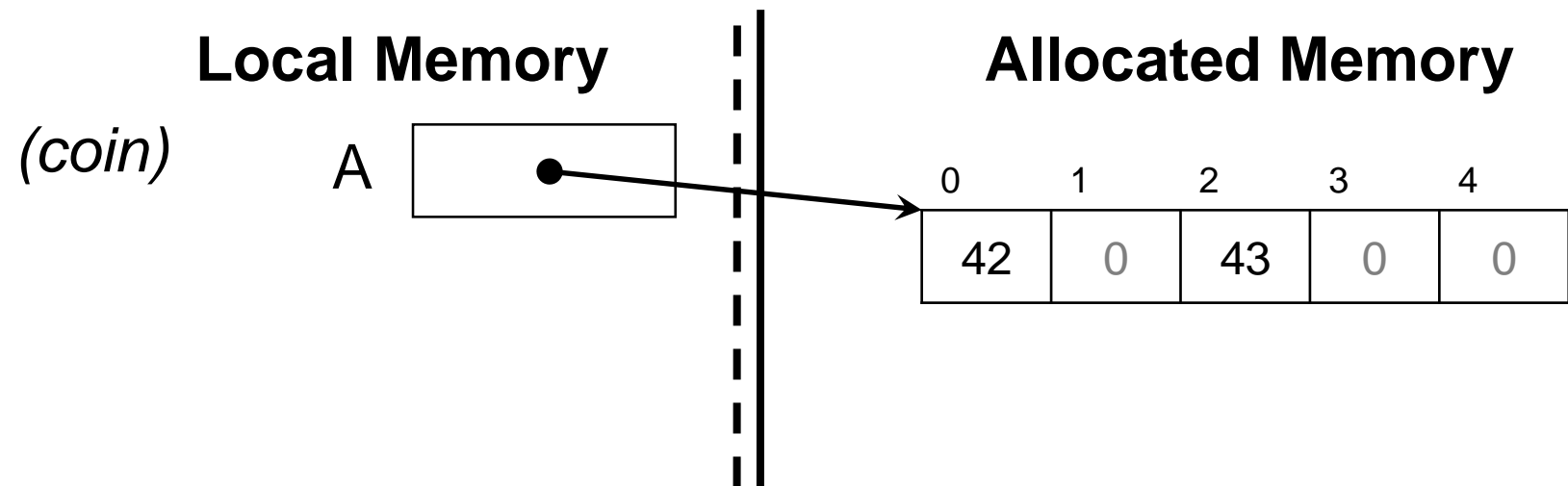
```
--> A[0] = 42;  
A[0] is 42 (int)  
--> A[0];  
42 (int)  
--> A[2] = A[0] + 1;  
A[2] is 43 (int)
```



# Accessing Array Elements

- Allocated memory is initialized with default values
  - 0 for `int`'s

```
Linux Terminal
--> A[1];
0 (int)
```

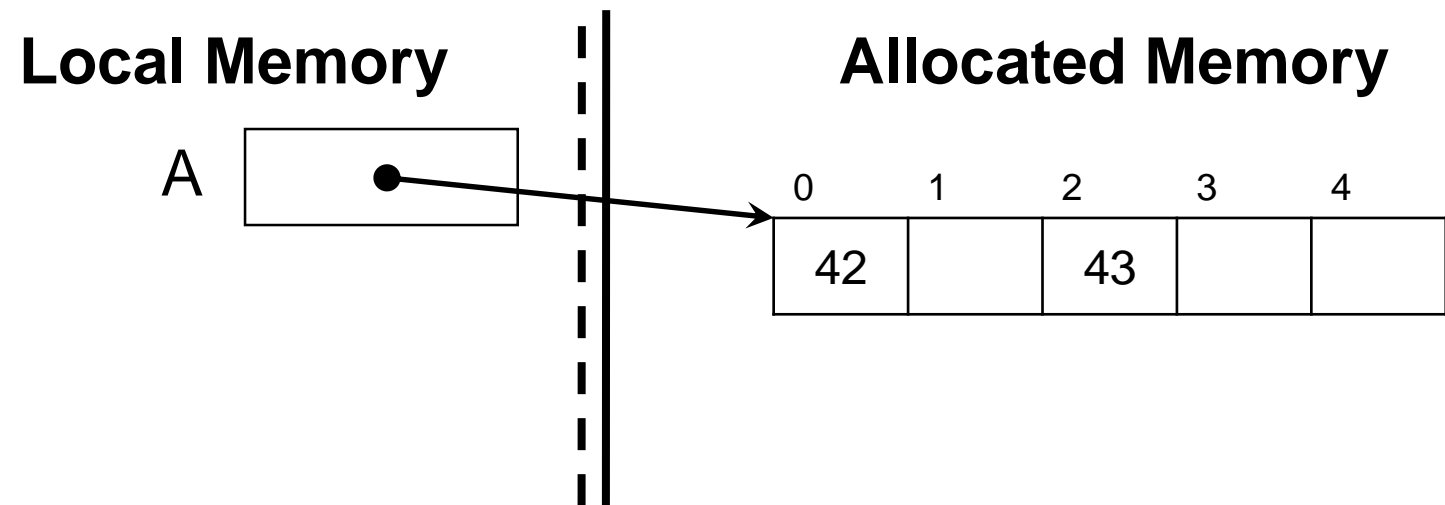


- For readability, we generally don't write default values

# Out-of-bound Array Accesses

Linux Terminal

```
--> A[-1];  
Error: accessing negative element in 5-element array  
--> A[100];  
Error: accessing element 100 in 5-element array  
--> A[5];  
Error: accessing element 5 in 5-element array
```



- Valid indices are only 0 up to length of the array
  - Anything else is **out of bounds**

# Preconditions of Array Operations

- Out-of-bound array accesses are **unsafe**
- Array operations have preconditions

```
alloc_array(type, n)  
//@requires n >= 0;
```

```
A[i]  
//@requires 0 <= i && i < 'length of A';
```

- When using array operations, we must **prove** these preconditions are met
- Arrays can have length 0



# Aliasing

## Linux Terminal

```
--> int[] B = A;  
B is 0xF72260 (int[] with 5 elements)  
--> B[2] = 7;  
B[2] is 7 (int)  
--> A[2];  
7 (int)  
--> A == B;  
true (bool)
```

## Local Memory

A	0xF72260
B	0xF72260

## Allocated Memory

0	1	2	3	4
42		7		

0xF72260

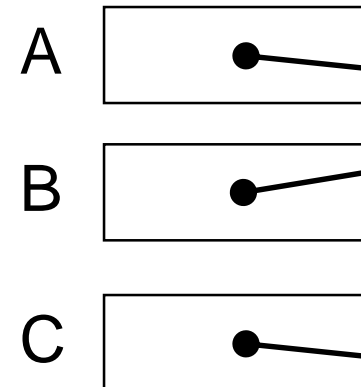
- A and B contain the same address
  - They refer to the *same array* in allocated memory
  - They are **aliases**
  - Modifying the array through one modifies it through the other

# Aliasing

## Linux Terminal

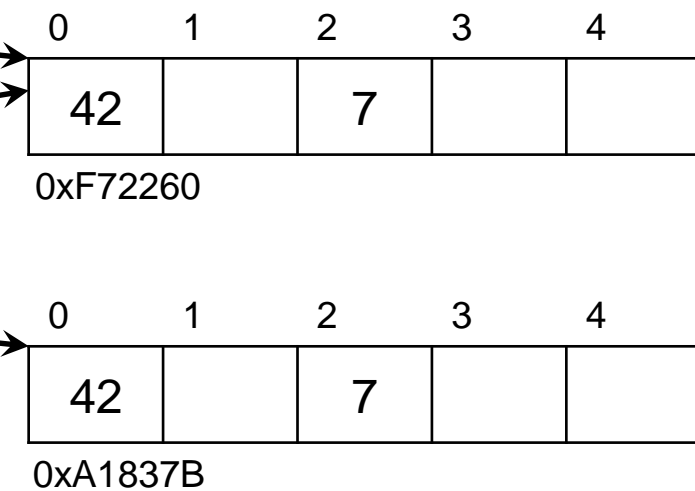
```
--> int[] C = alloc_array(int, 5);  
C is 0xA1837B (int[] with 5 elements)  
--> C[0] = 42;  
C[0] is 42 (int)  
--> C[2] = 7;  
C[2] is 7 (int)  
--> C == A;  
false (bool)
```

## Local Memory



Now using arrows

## Allocated Memory



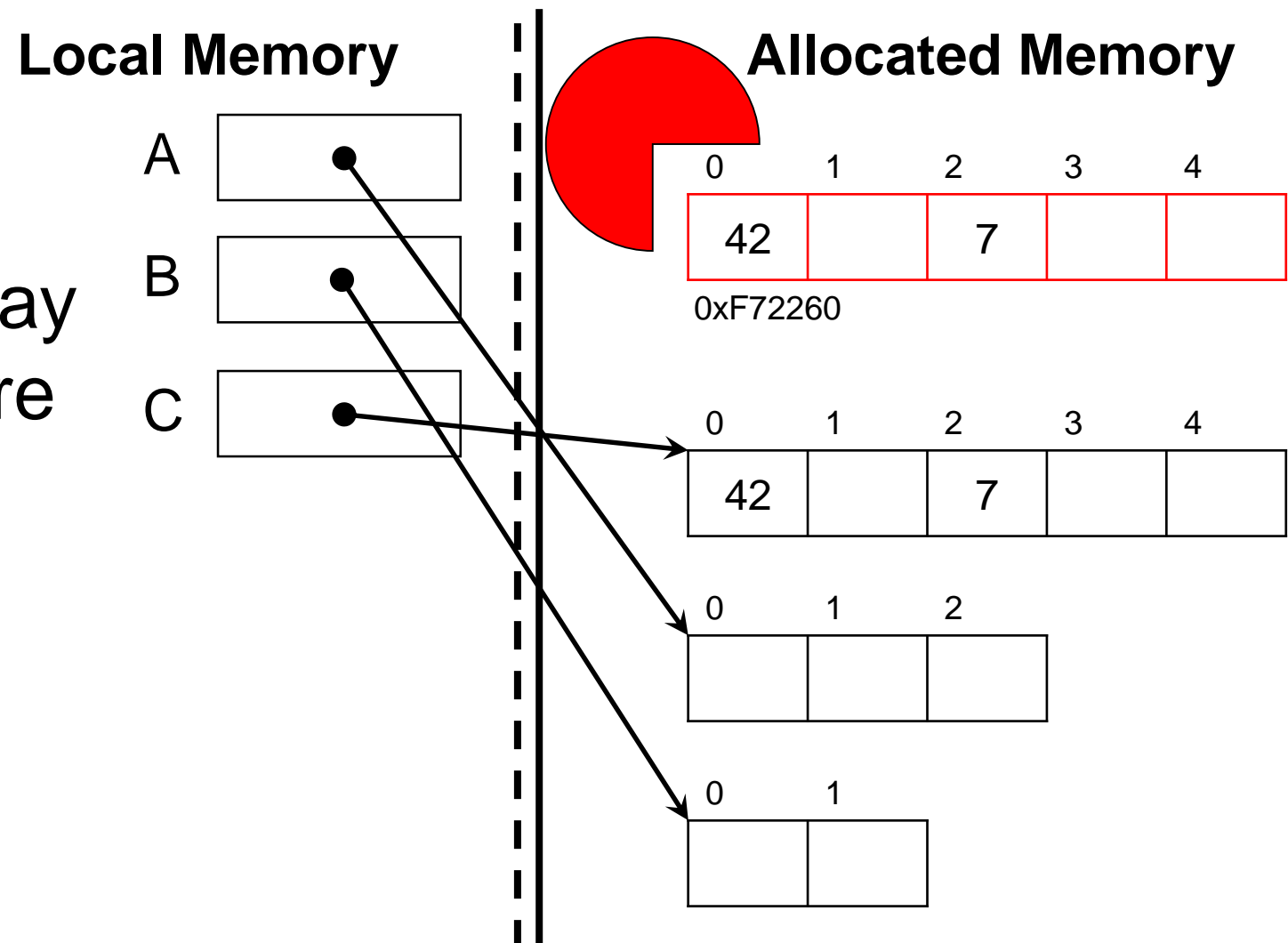
- A and C reference distinct arrays
  - which happen to have the same elements

# Garbage Collection

## Linux Terminal

```
--> A = alloc_array(int, 3);  
A is 0xF722C0 (int[] with 3 elements)  
--> B = alloc_array(int, 2);  
B is 0xF722F0 (int[] with 2 elements)
```

- Elements of the initial array (at address 0xF72260) are inaccessible
  - It will be automatically **garbage-collected**



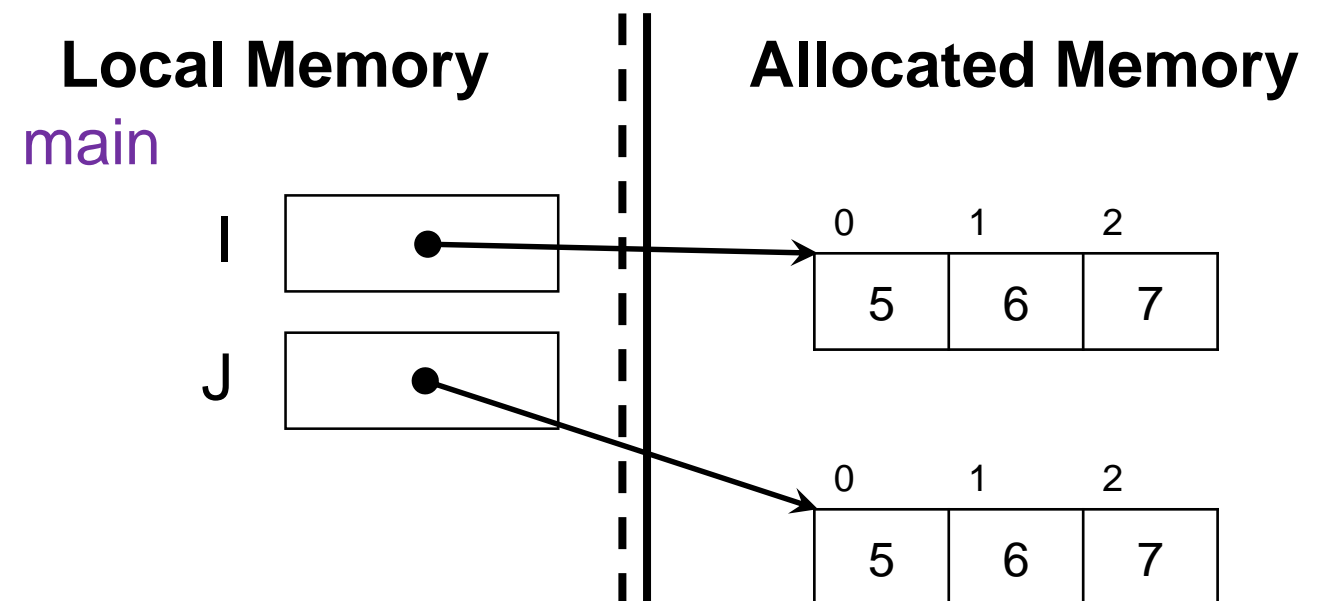
# **Coding with Arrays**

# array\_copy

- We want to write a function, `array_copy`, that returns a new array with the same elements as the array passed to it
  - `array_copy` returns a *deep copy* of input
    - Not a alias!

```
int[] array_copy(int[] A) {  
    ...  
}  
  
int main() {  
    int[] I = ... [5, 6, 7] ...;  
    int[] J = array_copy(I);  
    return 0;  
}
```

Here

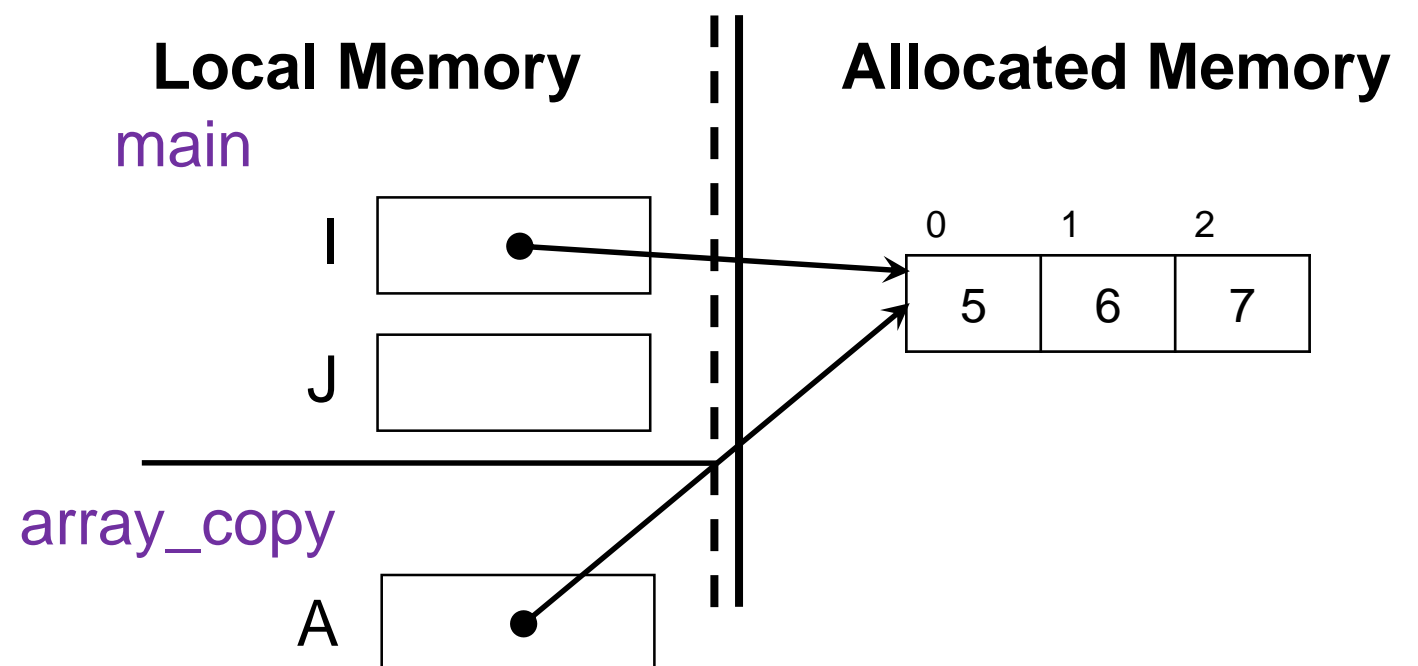


# First Attempt

- Calling a function with an array
  - copies the **address** of the array into its parameter
- Returning an array from a function
  - returns the **address** of the array to the caller

Here

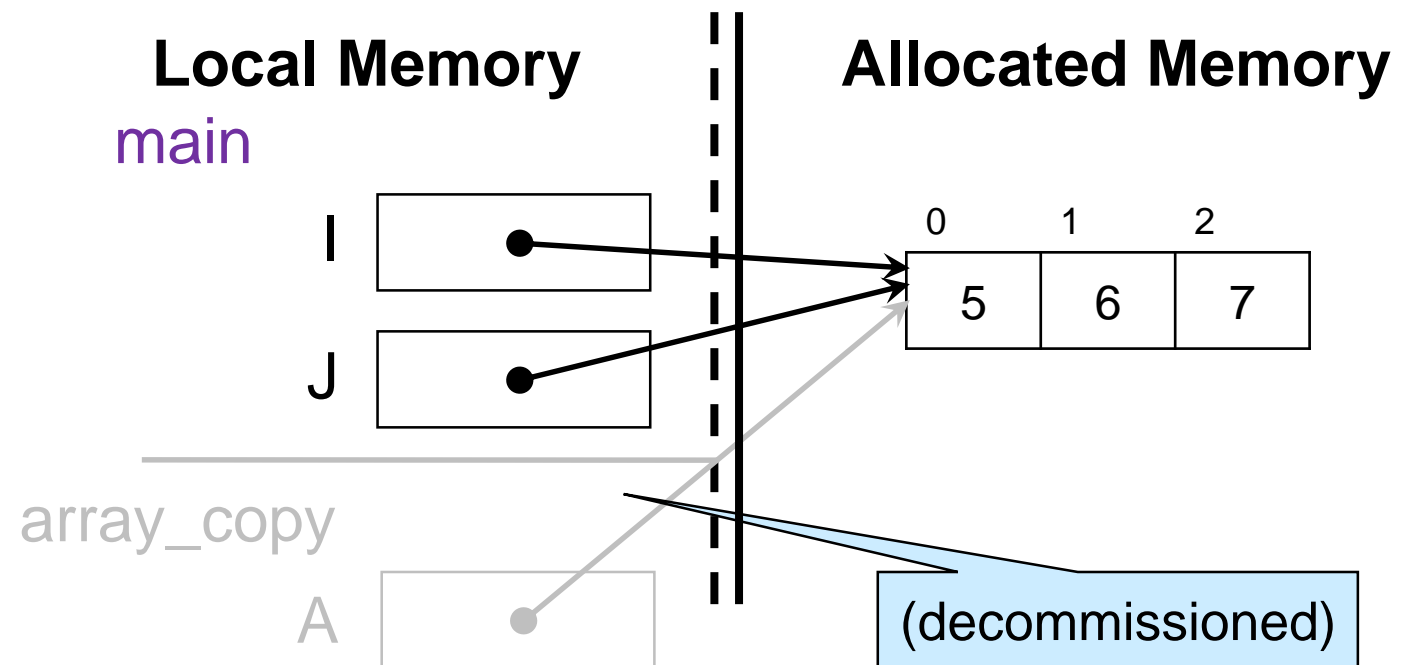
```
int[] array_copy(int[] A) {  
    return A;  
}  
  
int main() {  
    int[] I = ... [5, 6, 7] ...;  
    int[] J = array_copy(I);  
    return 0;  
}
```



# First Attempt

```
int[] array_copy(int[] A) {  
    return A;  
}  
  
int main() {  
    int[] I = ... [5, 6, 7] ...;  
    int[] J = array_copy(I);  
    return 0;  
}
```

Here



- We returned an *alias* to I
  - Not what we were aiming for!



# Second Attempt

- `array_copy` needs to *allocate* a new array

```
int[] array_copy(int[] A) {  
    int[] B = alloc_array(int, ??);  
    ...  
    return B;  
}  
  
int main() {  
    int[] I = ... [5, 6, 7] ...;  
    int[] J = array_copy(I);  
    return 0;  
}
```

- What length should B have?
  - There is **no way to get the length of an array in C0**
  - We need to pass it as an argument



# Second Attempt

- Pass the length of A as a second argument

```
int[] array_copy(int[] A, int n){
    int[] B = alloc_array(int, n);
    ...
    return B;
}

int main() {
    int[] I = ... [5, 6, 7] ...;
    int[] J = array_copy(I, 3);
    return 0;
}
```

- Is the call to **alloc\_array** safe?
  - No: we want  $n \geq 0$
  - Add precondition  
*//@requires n >= 0;*

# Second Attempt

- Is this enough to get intended behavior?

- No:  $n$  should be the length of  $A$
- But we can't get the length of an array

- Special **contract-only** function:

$\text{\textbackslash length}(A)$

- Can **only** be used in contracts
- Evaluates to the length of  $A$

```
int[] array_copy(int[] A, int n)
//@requires n == \length(A);
{
    int[] B = alloc_array(int, n);
    ...
    return B;
}

int main() {
    int[] I = ... [5, 6, 7] ...;
    int[] J = array_copy(I, 3);
    return 0;
}
```

# Contracts of Array Operations

- We can now write strong contracts for the array operations
  - Better precondition of `A[i]`
  - Postcondition for **`alloc_array`**

```
alloc_array(type, n)  
//@requires n >= 0;  
//@ensures \length(\result) == n;
```

```
A[i]  
//@requires 0 <= i && i < \length(A);
```

```
\length(A)  
//@ensures \result >= 0;
```

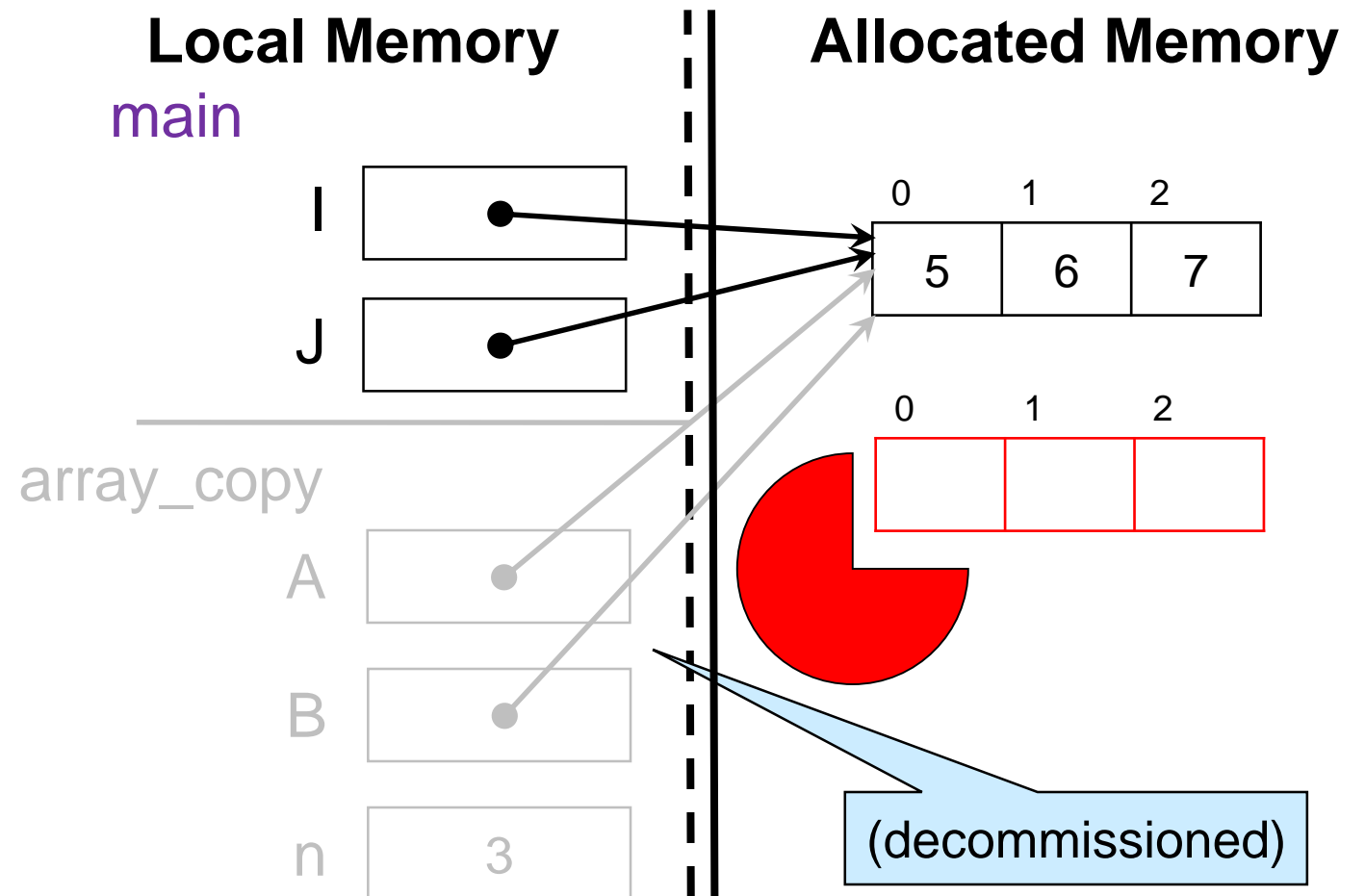
- We can use them in our proofs

# Second Attempt

```
int[] array_copy(int[] A, int n)
//@requires n == \length(A);
{
    int[] B = alloc_array(int, n);
    B = A;
    return B;
}

int main() {
    int[] I = ... [5, 6, 7] ...;
    int[] J = array_copy(I, 3);
    return 0;
}
```

Here



- B is aliased to A
  - Newly allocated array is garbage collected
  - We return an *alias* to I

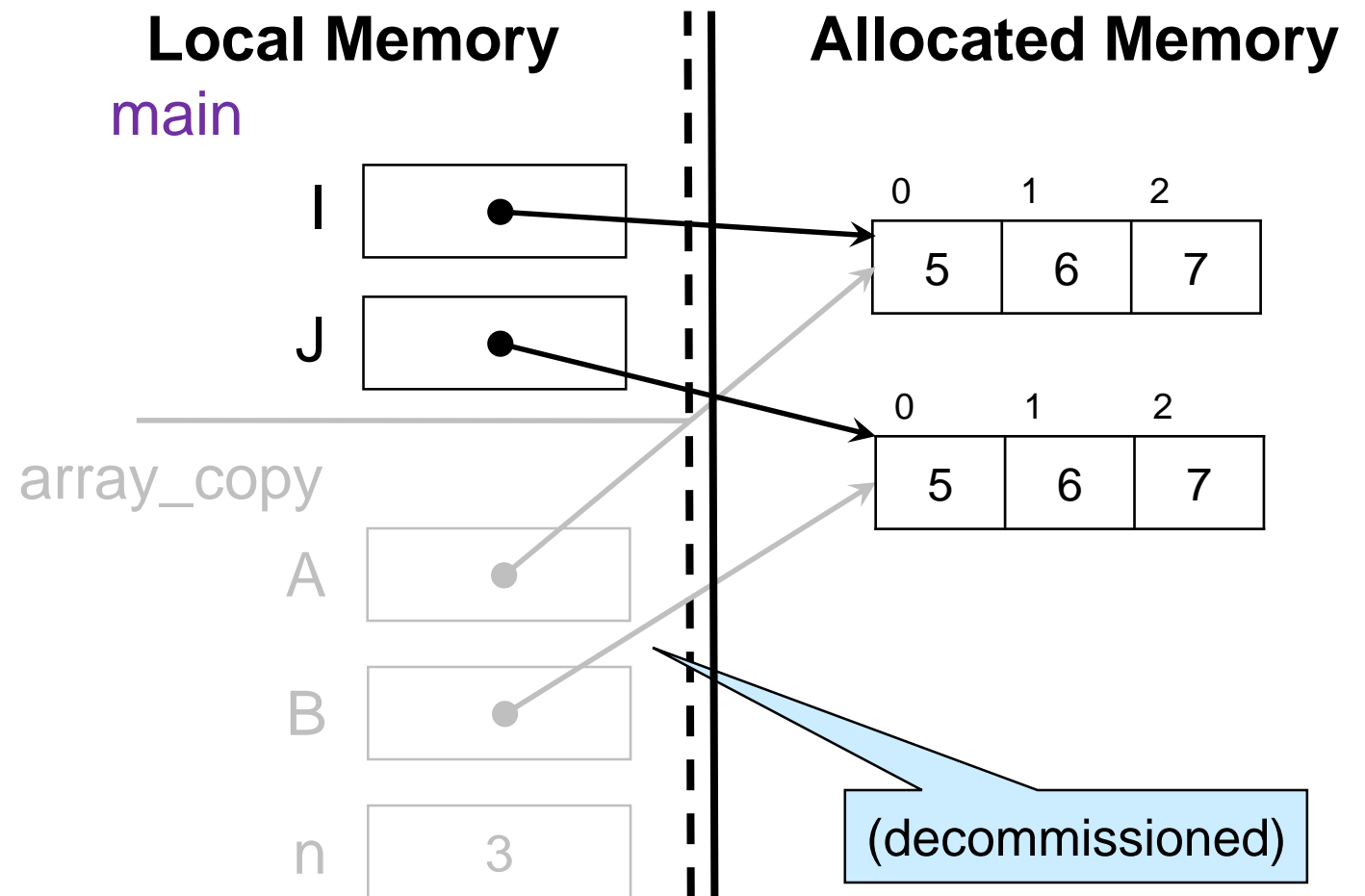
✗

# Third Attempt

```
int[] array_copy(int[] A, int n)
//@requires n == \length(A);
{
    int[] B = alloc_array(int, n);
    for (int i=0; i < n; i++) {
        B[i] = A[i];
    }
    return B;
}

int main() {
    int[] I = ... [5, 6, 7] ...;
    int[] J = array_copy(I, 3);
    return 0;
}
```

Here



- Works as expected ✓
  - `for`-loops are convenient to iterate through arrays
  - Local variable `i` is only defined inside the loop

# **Safety of Array Code**

# Safety of `array_copy`

## ● Is `array_copy` safe?

○ `alloc_array(int, n)` ?

➤ To show:  $n \geq 0$

○ `A[i]` ?

➤ To show:  $0 \leq i$   
and  $i < \text{length}(A)$

○ `B[i]` ?

➤ To show:  $0 \leq i$   
and  $i < \text{length}(B)$

```
1. int[] array_copy(int[] A, int n)
2. //@requires n == \length(A);
3. {
4.   int[] B = alloc_array(int, n);
5.   for (int i=0; i < n; i++) {
6.     B[i] = A[i];
7.   }
8.   return B;
9. }
10.
11. int main() {
12.   int[] I = ... [5, 6, 7] ...;
13.   int[] J = array_copy(I, 3);
14.   return 0;
15. }
```

# Safety of `array_copy`

`alloc_array(int, n)`

➤ To show:  $n \geq 0$

- A.  $n == \text{length}(A)$  by line 2
- B.  $\text{length}(A) \geq 0$  by `length`
- C.  $n \geq 0$  by A and B



```
1. int[] array_copy(int[] A, int n)
2. //@requires n == \length(A);
3. {
4.   int[] B = alloc_array(int, n);
5.   for (int i=0; i < n; i++) {
6.     B[i] = A[i];
7.   }
8.   return B;
9. }
10.
11. int main() {
12.   int[] I = ... [5, 6, 7] ...;
13.   int[] J = array_copy(I, 3);
14.   return 0;
15. }
```



# Safety of `array_copy`

`A[i]`

➤ To show:  $i < \text{length}(A)$

- A.  $n == \text{length}(A)$  by line 2
- B.  $i < n$  by line 5
- C.  $i < \text{length}(A)$  by A and B



➤ To show:  $0 \leq i$

- “*i starts at 0 and is **always** incremented*”
  - this is **operational reasoning**
- Nothing we can *point to*!



```
1. int[] array_copy(int[] A, int n)
2. //@requires n == \length(A);
3. {
4.   int[] B = alloc_array(int, n);
5.   for (int i=0; i < n; i++) {
6.     B[i] = A[i];
7.   }
8.   return B;
9. }
10.
11. int main() {
12.   int[] I = ... [5, 6, 7] ...;
13.   int[] J = array_copy(I, 3);
14.   return 0;
15. }
```

# Safety of `array_copy`

$A[i]$

➤ To show:  $0 \leq i$

○ Add it as a loop invariant

➤ We will need to show it is valid

A.  $0 \leq i$  by line 6



This is a common trick:

- if something is true by operational reasoning only
- turn it into a **contract** and prove it using point-to reasoning

```
1. int[] array_copy(int[] A, int n)
2. //@requires n == \length(A);
3. {
4.   int[] B = alloc_array(int, n);
5.   for (int i=0; i < n; i++)
6.     //@loop_invariant 0 <= i;
7.     {
8.       B[i] = A[i];
9.     }
10.  return B;
11.}
12.
13.int main() {
14.  int[] I = ... [5, 6, 7] ...;
15.  int[] J = array_copy(I, 3);
16.  return 0;
17.}
```

# Safety of `array_copy`

`B[i]`

➤ **To show:**  $0 \leq i \ \&\& \ i < \text{length}(B)$

○ *Left as exercise*

```
1. int[] array_copy(int[] A, int n)
2. //@requires n == \length(A);
3. {
4.   int[] B = alloc_array(int, n);
5.   for (int i=0; i < n; i++)
6.     //@loop_invariant 0 <= i;
7.     {
8.       B[i] = A[i];
9.     }
10.  return B;
11.}
12.
13.int main() {
14.  int[] I = ... [5, 6, 7] ...;
15.  int[] J = array_copy(I, 3);
16.  return 0;
17.}
```

# Validity of the Loop Invariant

//@loop\_invariant  $0 \leq i$ ;

## INIT:

➤ To show:  $0 \leq i$  initially

- A.  $i = 0$  by line 5
- B.  $0 \leq 0$  by math
- C.  $0 \leq i$  by A and B



```
1. int[] array_copy(int[] A, int n)
2. //@requires n == \length(A);
3. {
4.   int[] B = alloc_array(int, n);
5.   for (int i=0; i < n; i++)
6.     //@loop_invariant 0 <= i;
7.   {
8.     B[i] = A[i];
9.   }
10.  return B;
11.}
12.
13.int main() {
14.  int[] I = ... [5, 6, 7] ...;
15.  int[] J = array_copy(I, 3);
16.  return 0;
17.}
```

# Validity of the Loop Invariant

*//@loop\_invariant 0 <= i;*

**PRES:**  $0 \leq i$  is preserved

➤ **To show:** if  $0 \leq i$ , then  $0 \leq i'$

- A.  $0 \leq i$                       assumption
- B.  $i' = i+1$                   by line 5
- C.  $0 \leq i+1$                   by A  
    *only if  $i \neq \text{int\_max}()$*     by two's compl.
- D.  $i < n$                       by line 5
- E.  $i \neq \text{int\_max}()$             by math
- F.  $0 \leq i'$                     by B, C and E



```
1. int[] array_copy(int[] A, int n)
2. //@requires n == \length(A);
3. {
4.   int[] B = alloc_array(int, n);
5.   for (int i=0; i < n; i++)
6.     //@loop_invariant 0 <= i;
7.   {
8.     B[i] = A[i];
9.   }
10.  return B;
11.}
12.
13.int main() {
14.  int[] I = ... [5, 6, 7] ...;
15.  int[] J = array_copy(I, 3);
16.  return 0;
17.}
```

# Safety of Calls to `array_copy`

Is `array_copy(I, 3)` **safe**?

➤ **To Show:** `3 == \length(I)`

A. `\length(I) == 3` by line 14



```
1. int[] array_copy(int[] A, int n)
2. //@requires n == \length(A);
3. {
4.
5.
6.
7.
8.
9.
10. }
11.
12.
13. int main() {
14.   int[] I = alloc_array(int, 3);
15.   ... [5, 6, 7] ...;
16.   int[] J = array_copy(I, 3);
17.   int[] K = array_copy(J, 3);
18.   return 0;
19. }
```

# Safety of Calls to `array_copy`

Is `array_copy(J, 3)` **safe**?

➤ To Show:  $3 == \text{length}(J)$

- “*`array_copy` creates an array of the same length as its input*”
  - Looks at the code of a different function
  - This is **operational reasoning**!
  - We can only look at the *contracts* of other functions

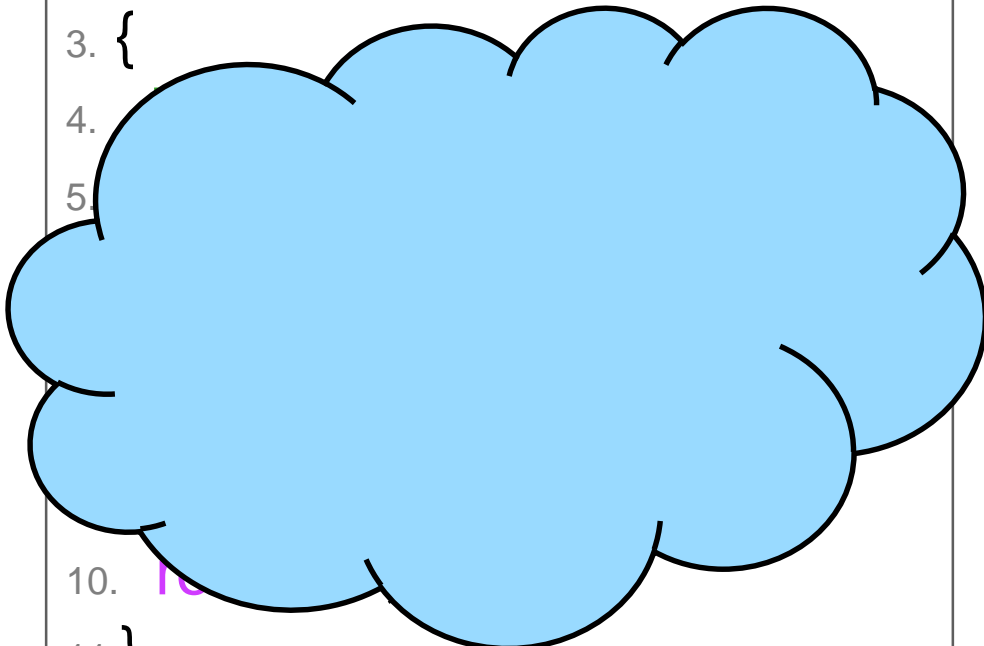
✗

- Add a postcondition to `array_copy`

Our trick again:

- if something is true by operational reasoning only
- turn it into a **contract** and prove it using point-to reasoning

```
1. int[] array_copy(int[] A, int n)
2. // @requires n == \length(A);
3. {
4.
5.
6.
7.
8.
9.
10. }
11.
12.
13. int main() {
14.   int[] I = alloc_array(int, 3);
15.           ... [5, 6, 7] ...;
16.   int[] J = array_copy(I, 3);
17.   int[] K = array_copy(J, 3);
18.   return 0;
19. }
```



# Safety of Calls to `array_copy`

Is `array_copy(J, 3)` safe?

➤ **To Show:** `3 == \length(J)`

- A. `\length(I) == 3` by line 15
- B. `\length(J) == 3` by lines 3 and 17
- C. `3 == \length(J)` by A and B



```
1. int[] array_copy(int[] A, int n)
2. // @requires n == \length(A);
3. // @ensures n == \length(\result);
4. {
5.   ...
6.   ...
11. }
12. }
13.
14. int main() {
15.   int[] I = alloc_array(int, 3);
16.           ... [5, 6, 7] ...;
17.   int[] J = array_copy(I, 3);
18.   int[] K = array_copy(J, 3);
19.   return 0;
20. }
```



# Is array\_copy correct?

➤ **To Show:** if  $n == \text{length}(A)$ ,  
then  $n == \text{length}(\text{result})$

- A.  $n == \text{length}(A)$                       assumption
- B.  $\text{length}(B) == n$                       by line 5
- C.  $\text{result} == B$                       by line 11
- D.  $n == \text{length}(\text{result})$               by A, B and C



- Does B contain the same elements as A in the same order?

This what we expect

**Correctness:**  
postconditions are met  
whenever preconditions hold

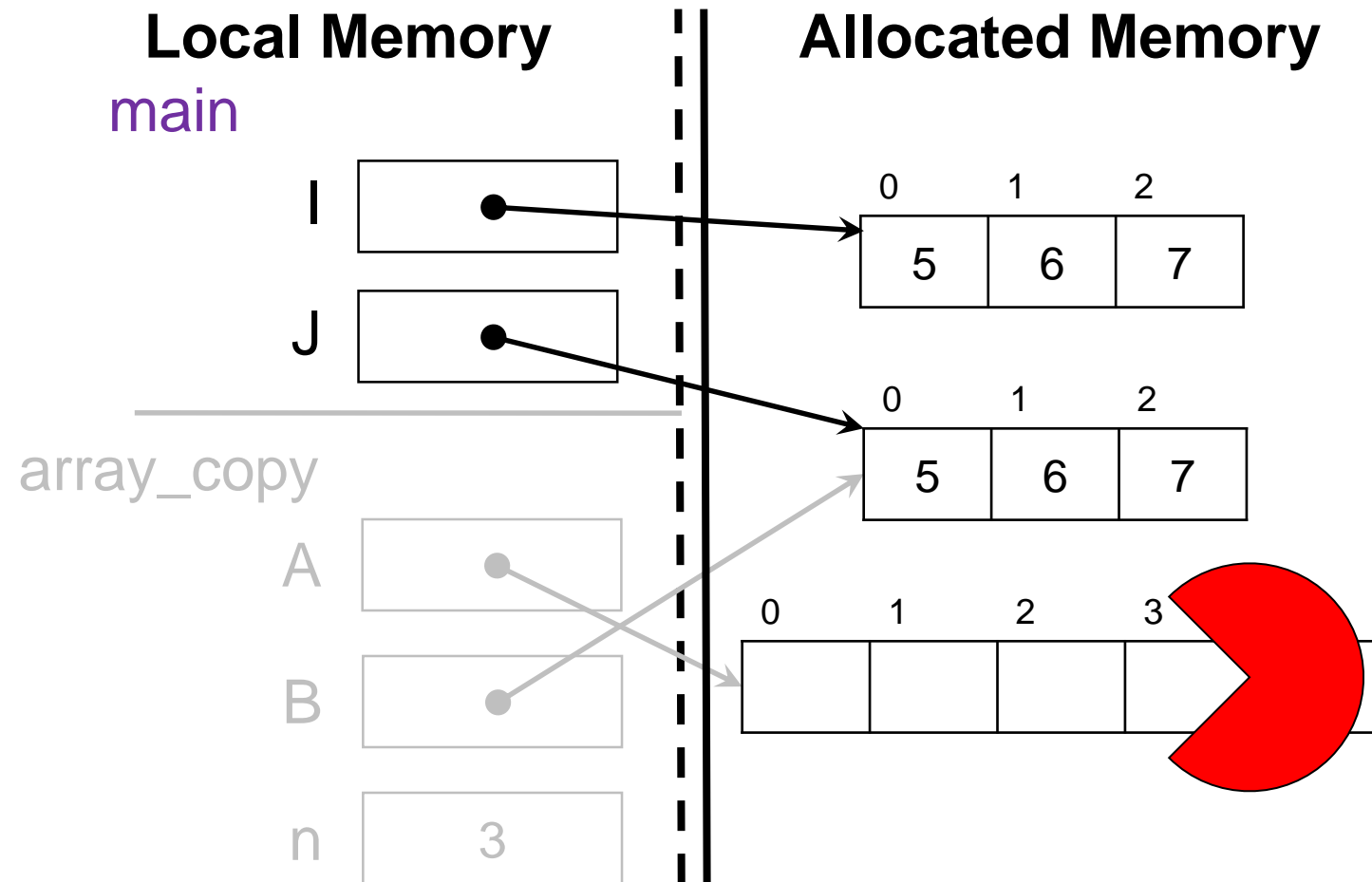
```
1. int[] array_copy(int[] A, int n)
2. //@requires n == length(A);
3. //@ensures n == length(result);
4. {
5.   int[] B = alloc_array(int, n);
6.   for (int i=0; i < n; i++)
7.     //@loop_invariant 0 <= i;
8.     {
9.       B[i] = A[i];
10.    }
11.   return B;
12. }
13.
14. int main() {
15.   ...
16. }
```

# **Effects of Array Code**

# Modifying Parameters

```
int[] array_copy(int[] A, int n)
//@requires n == \length(A);
//@ensures n == \length(\result);
{
    int[] B = alloc_array(int, n);
    for (int i=0; i < n; i++)
        //@loop_invariant 0 <= i;
        {
            B[i] = A[i];
        }
    A = alloc_array(int, 5);
    return B;
}

int main() {
    int[] I = ... [5, 6, 7] ...;
    int[] J = array_copy(I, 3);
    return 0;
}
```

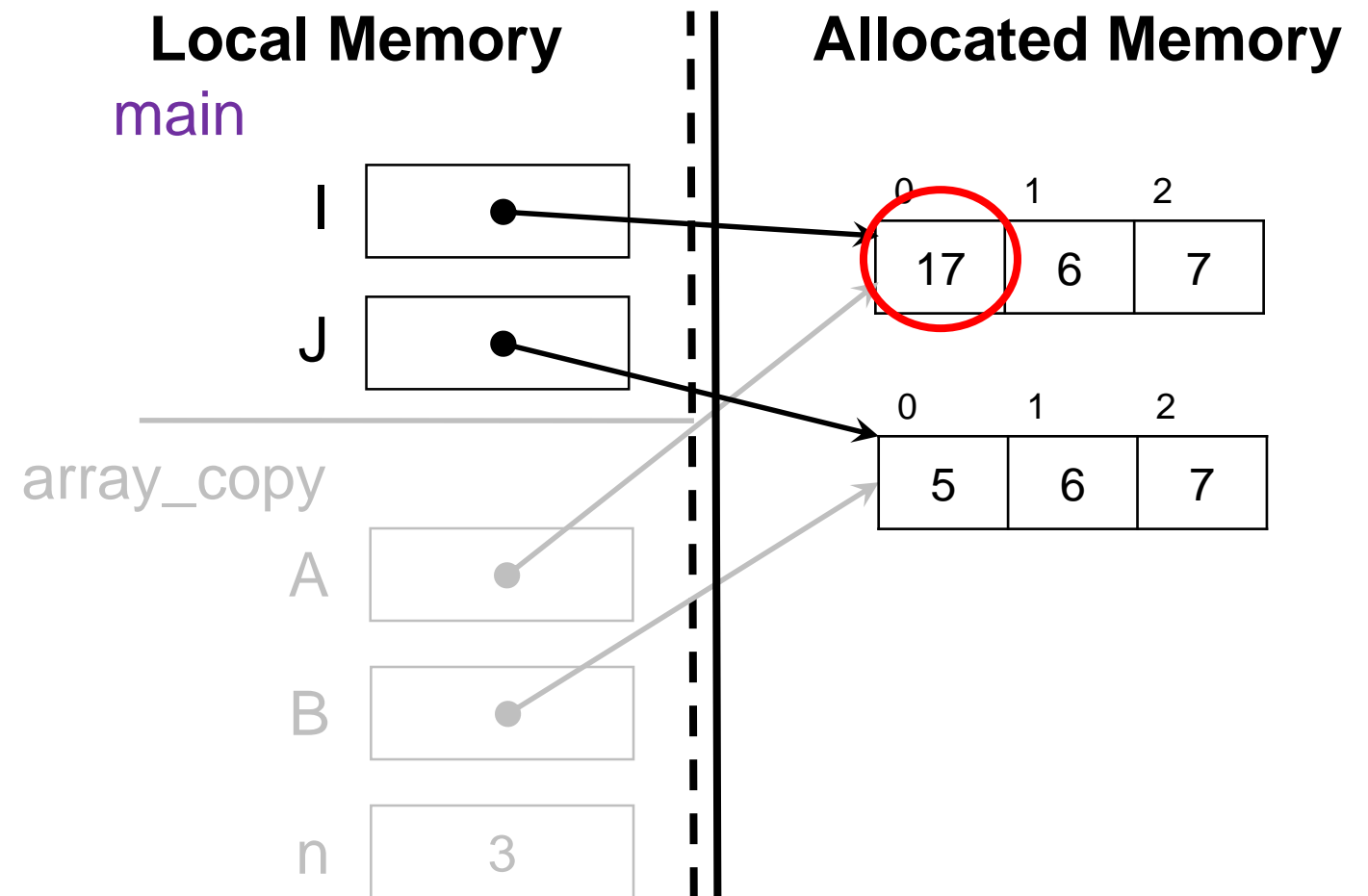


- Only the value of `A` in `array_copy` changes
  - The value of `I` is unchanged
  - The change is **not visible to caller**

# Modifying Array elements

```
int[] array_copy(int[] A, int n)
//@requires n == \length(A);
//@ensures n == \length(\result);
{
    int[] B = alloc_array(int, n);
    for (int i=0; i < n; i++)
        //@loop_invariant 0 <= i;
        {
            B[i] = A[i];
        }
    if (n > 0) A[0] = 17;
    return B;
}

int main() {
    int[] I = ... [5, 6, 7] ...;
    int[] J = array_copy(I, 3);
    return 0;
}
```



- The array contents is **shared** between caller and callee
  - The value of `I[0]` is changed
  - The change is **visible to caller**

Here