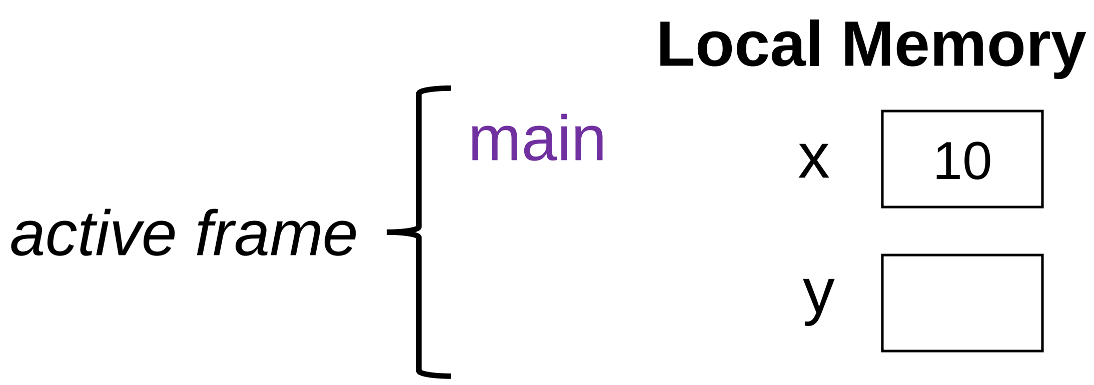
Arrays

Memory Model

C0 Memory Model

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

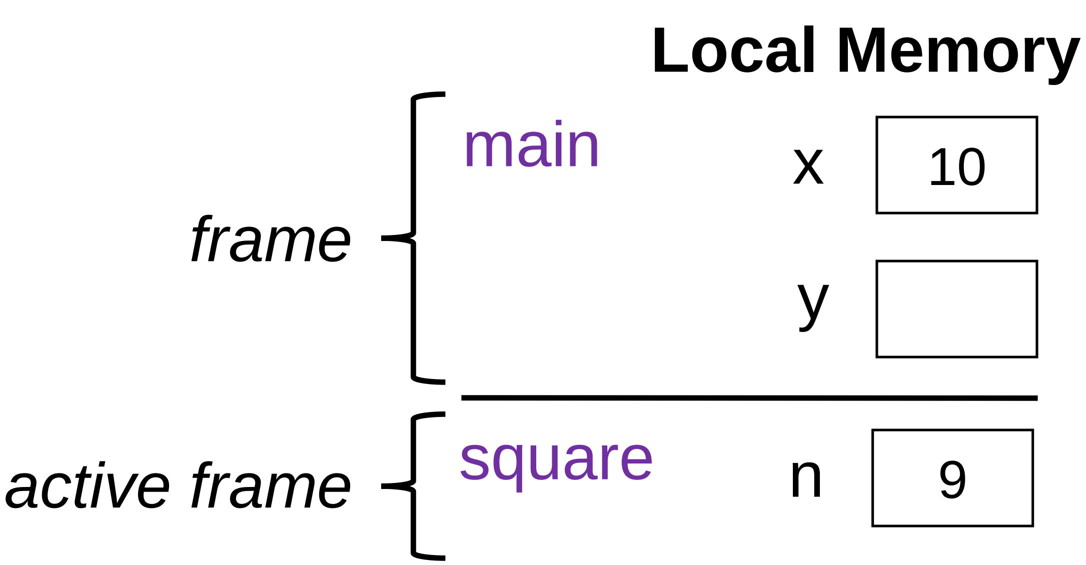



```
int POW(int x, int y) {
  if (y == 0) return 1;
  return x * POW(x, y-1);
}

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;
}
```

Here

C0 Memory Model

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



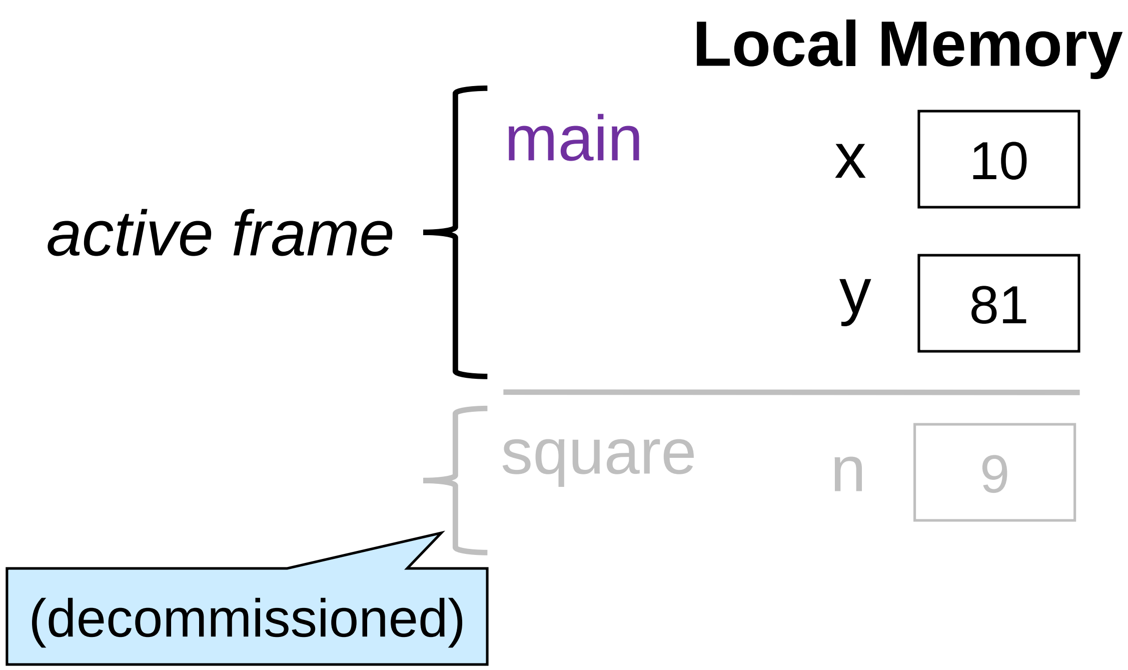
Here

```
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;
}
```

The only variable square can use is n

C0 Memory Model

- The frame is decommissioned when the function returns
  - It goes way



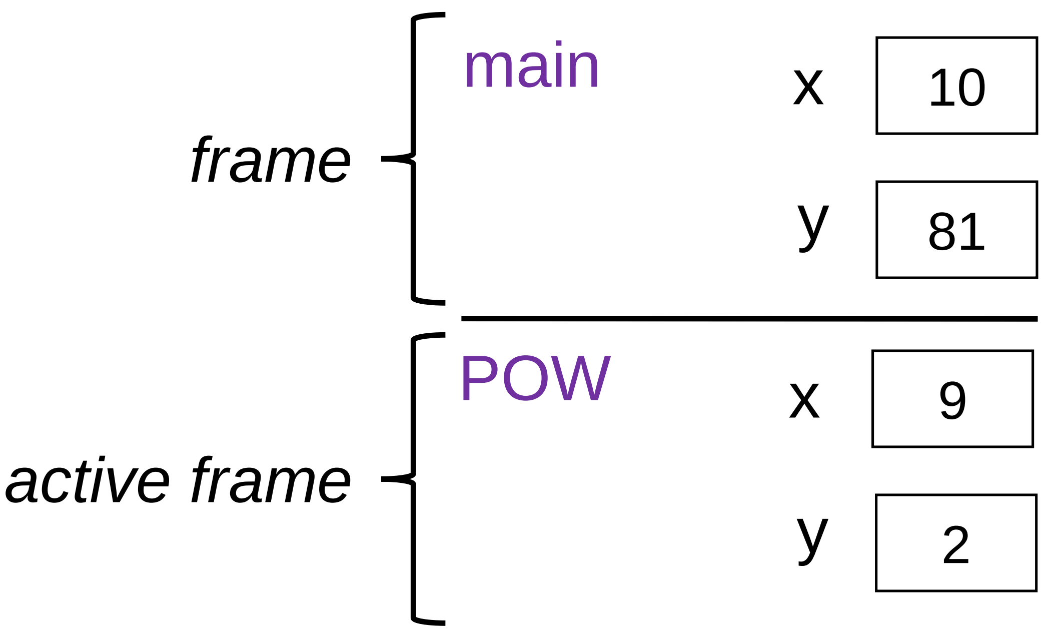

```
int POW(int x, int y) {
  if (y == 0) return 1;
  return x * POW(x, y-1);
}

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;
}
```

Here

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

Local Memory

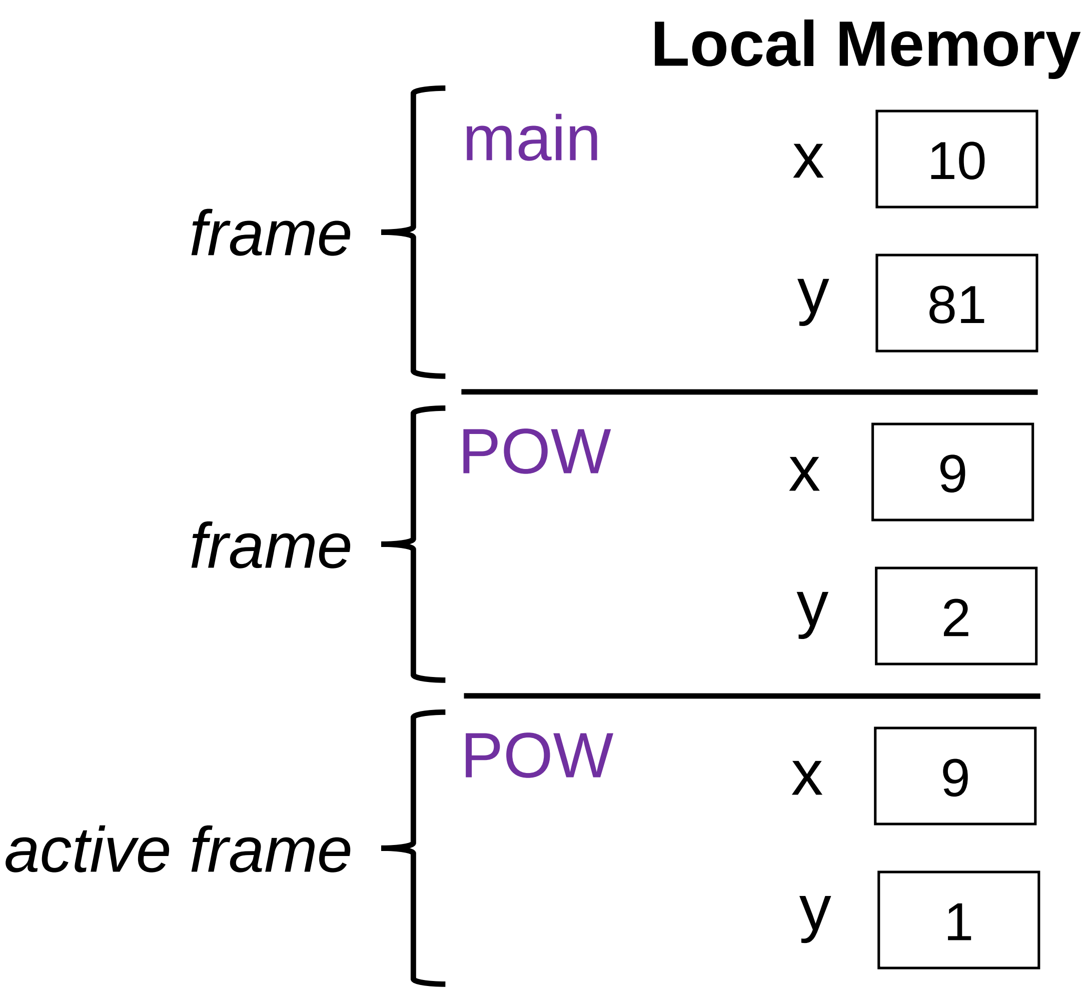


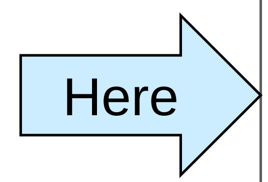
Here

```
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

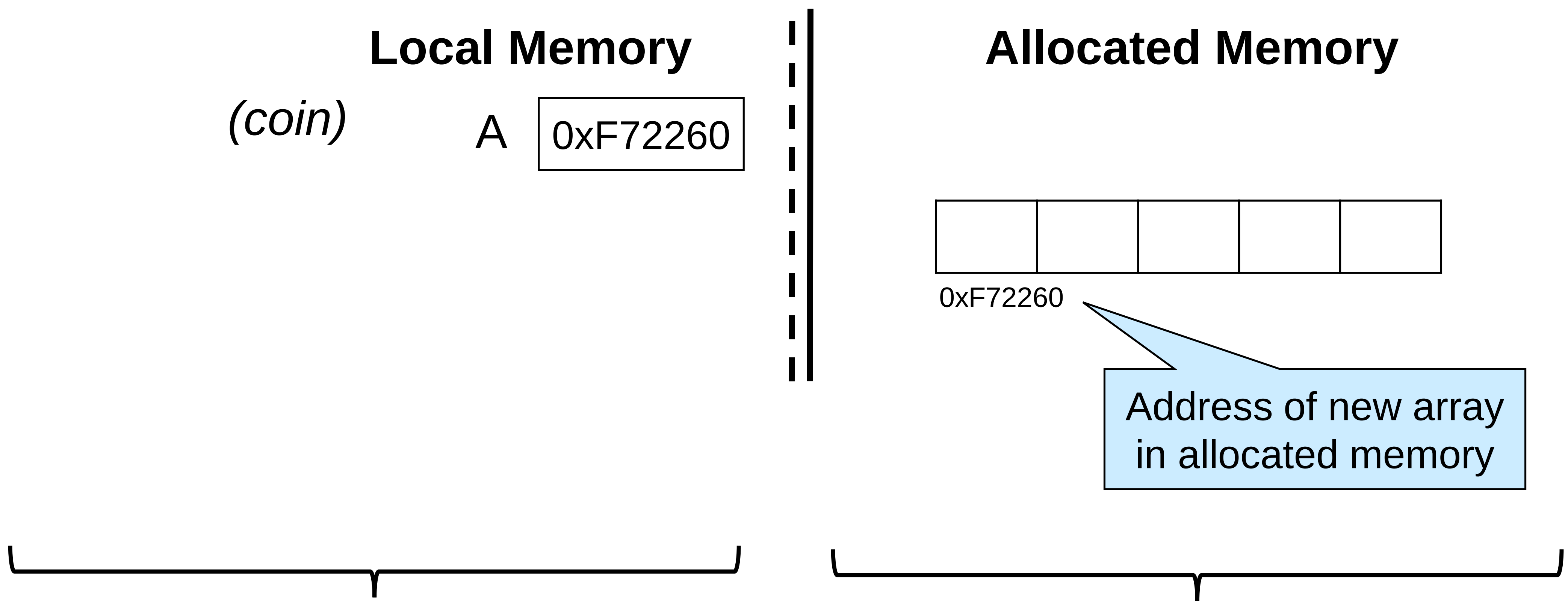
Linux Terminal

```
# 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

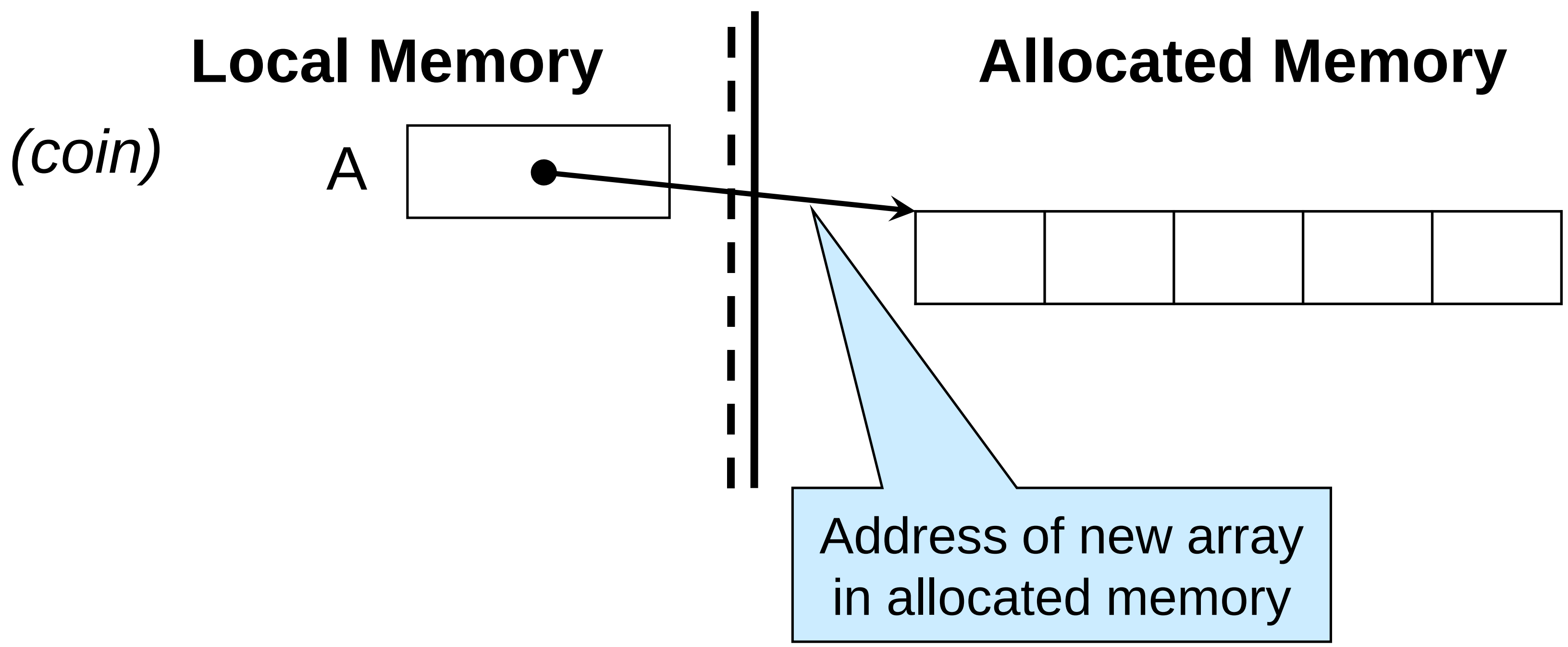


Contains the values of local variables int, bool, char, string, and ***addresses***

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



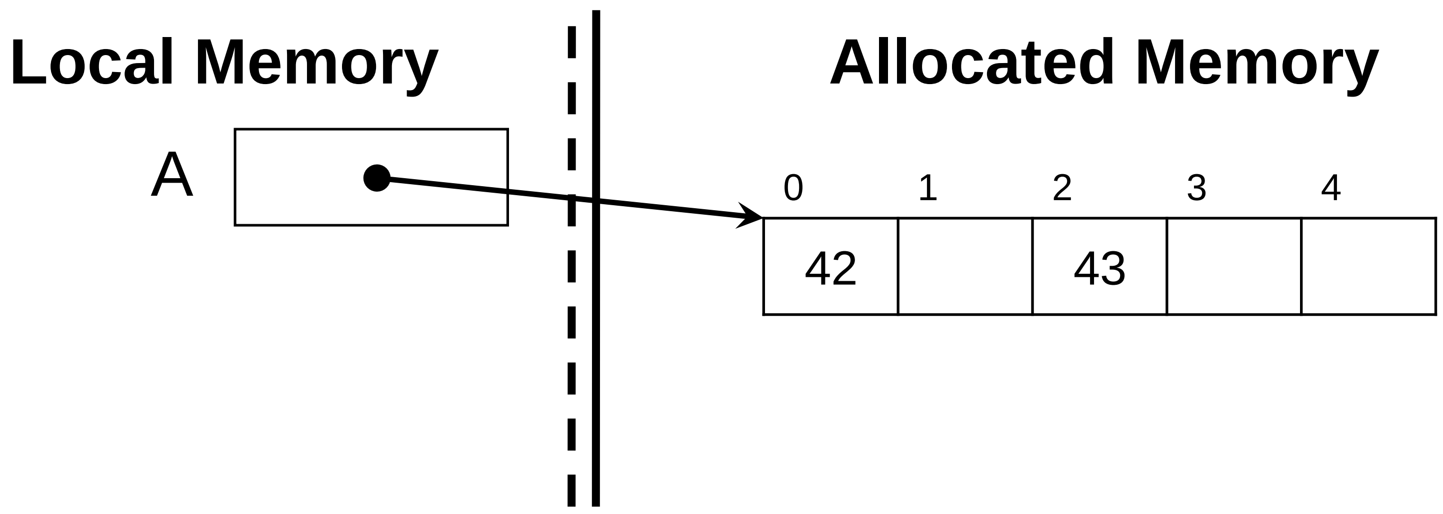
Accessing Array Elements

- *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)
```

(coin)

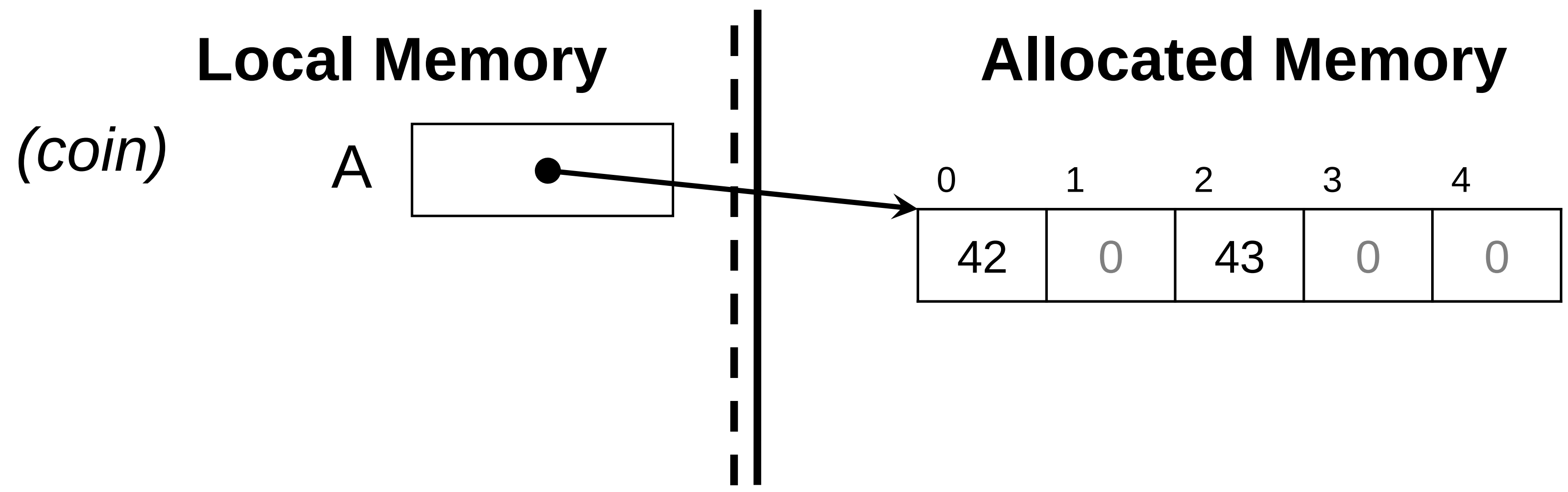


Accessing Array Elements

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

Linux Terminal

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

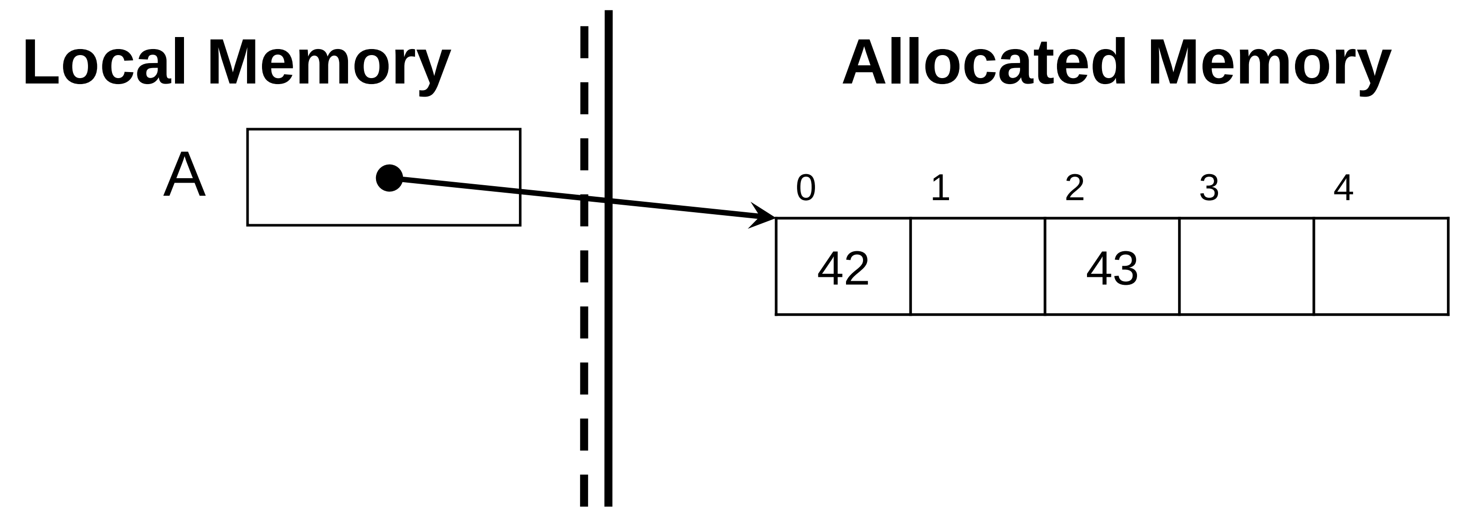


  - For readability, we won't write default values below

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'*;

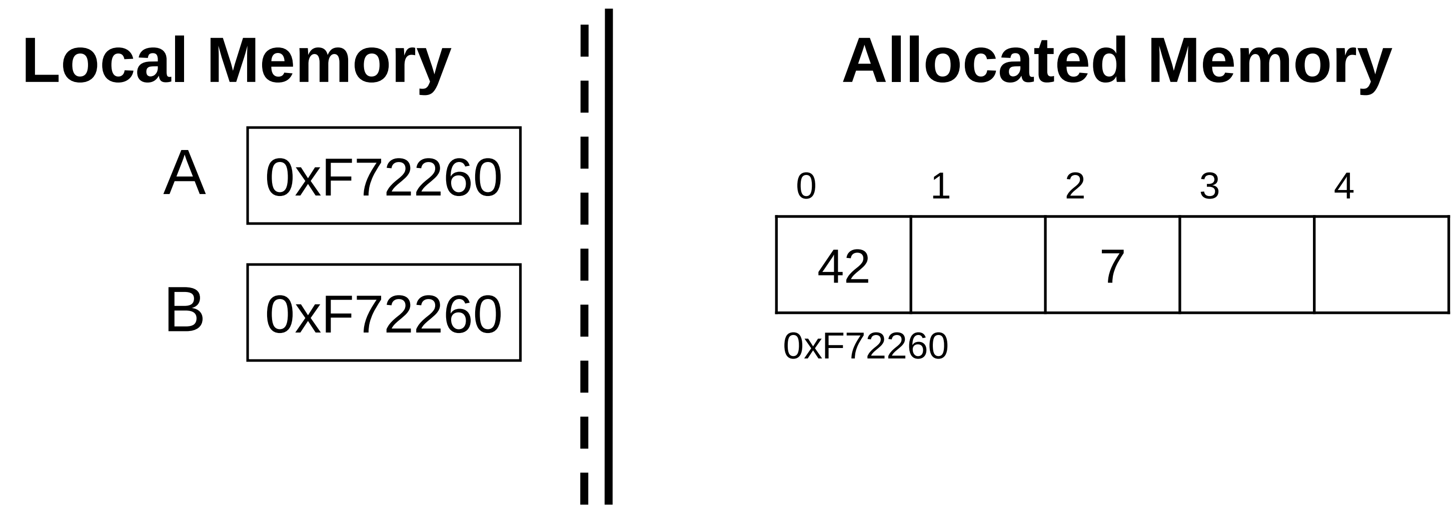
  - When using array operations, we must **prove** these preconditions are met

  - 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)
```

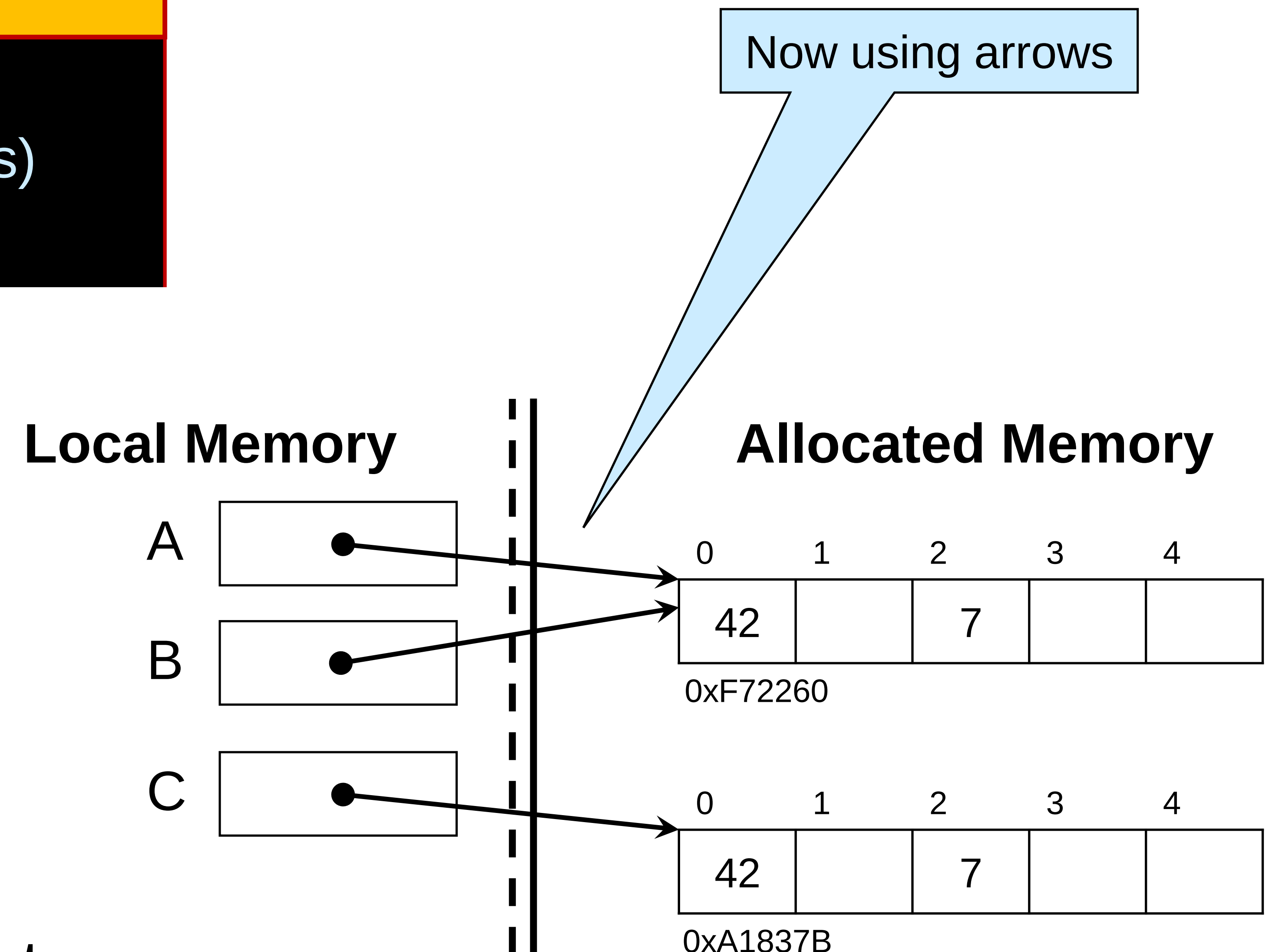


- A and B contain the same address
  - They refer to the *same array* in allocated memory
  - They are **aliases**
  - Modifying the array through one modifies it through the other

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)
```



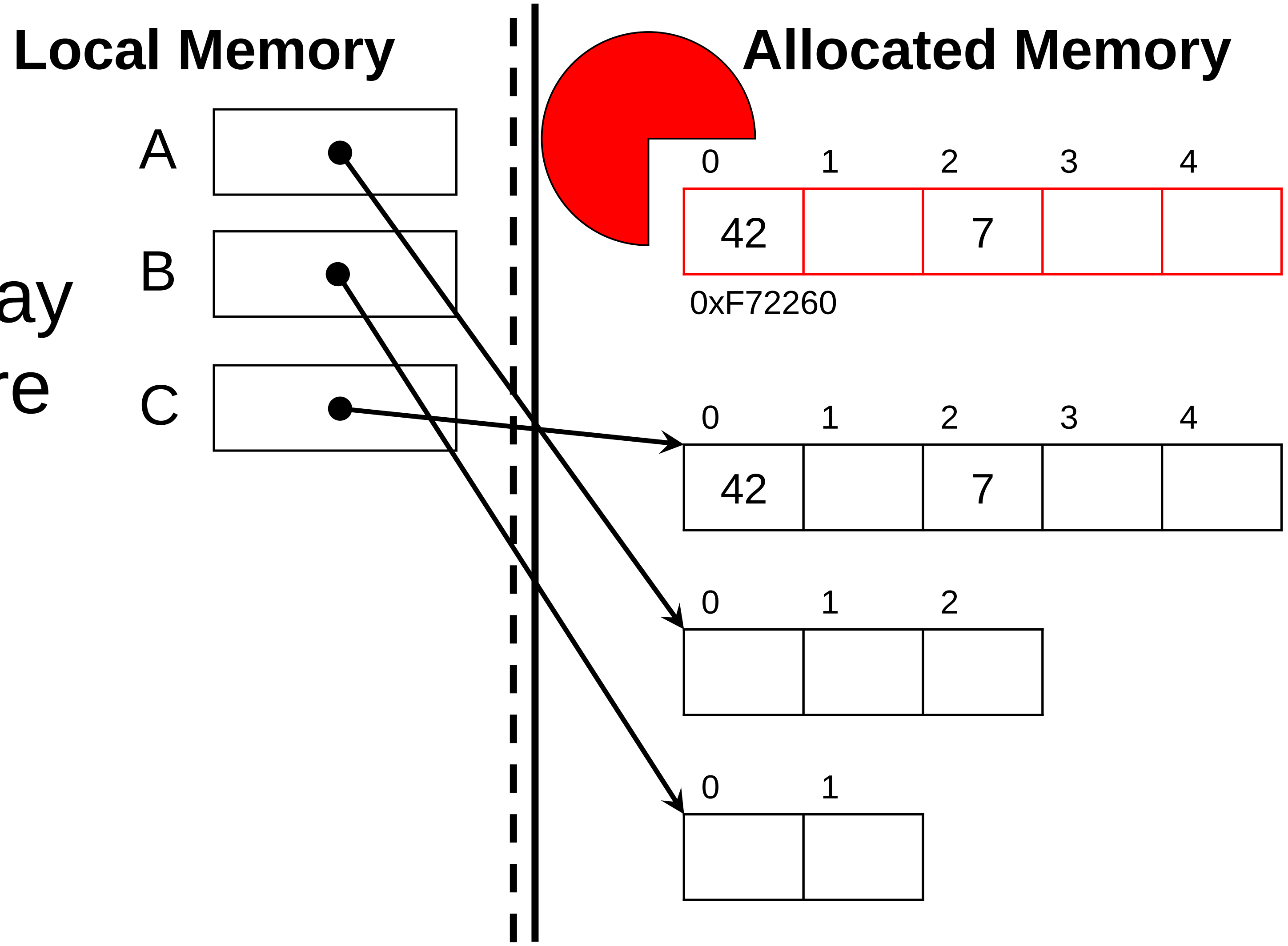
- A and C reference distinct arrays
  - which happen to have the same elements

Garbage Collection

Linux Terminal

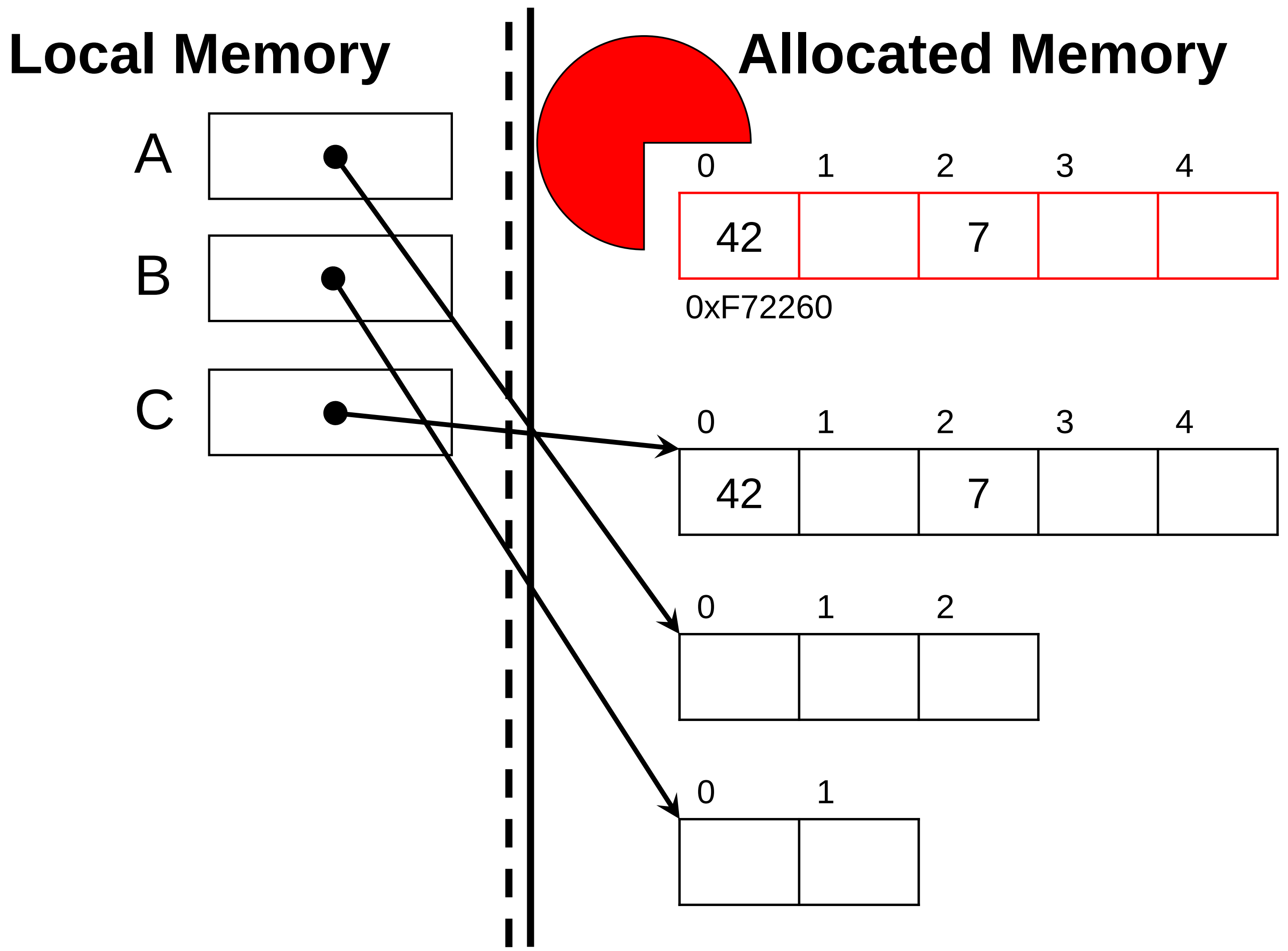
```
--> A = alloc_array(int, 3);
A is 0xF722C0 (int[] with 3 elements)
--> B = alloc_array(int, 2);
B is 0xF722F0 (int[] with 2 elements)
```

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



Garbage Collection

- Only allocated memory get garbage collected
  - eventually, when C0 feels it is appropriate
- Local memory does not get garbage collected
  - Function frames get decommissioned when the call returns



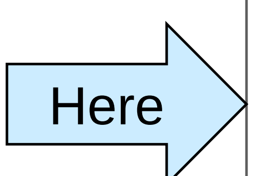
Garbage collection
≠
decommissioning

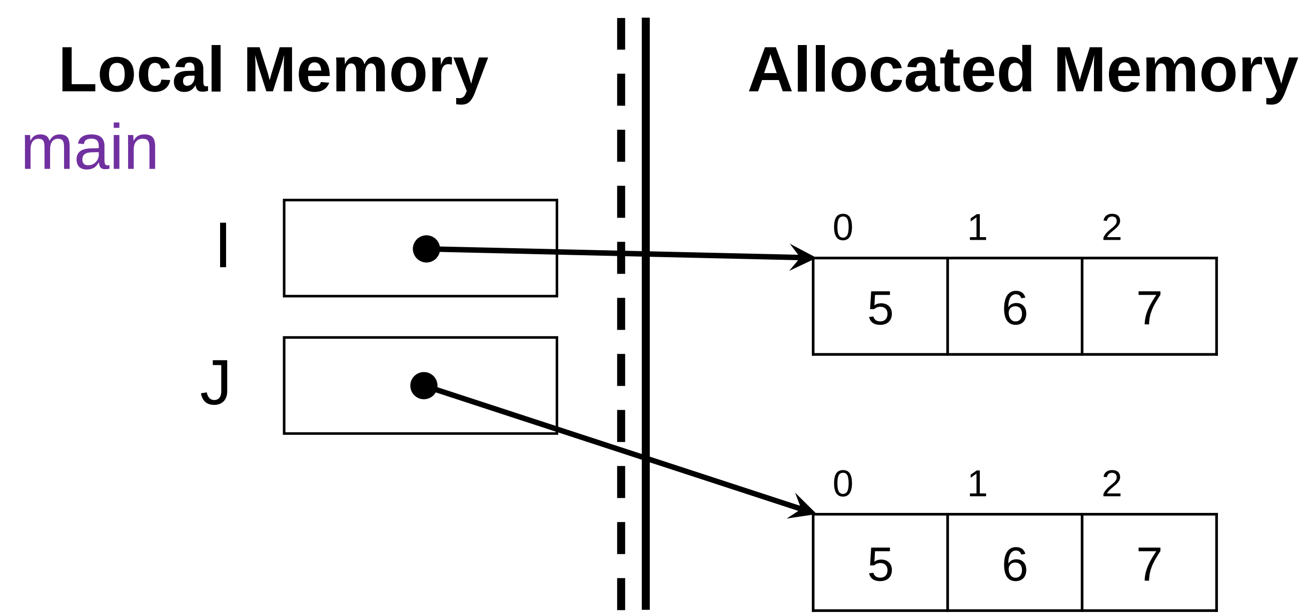
Coding with Arrays

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 its input
    - Not a alias!

```
int[] array_copy(int[] A) {
   …
}

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



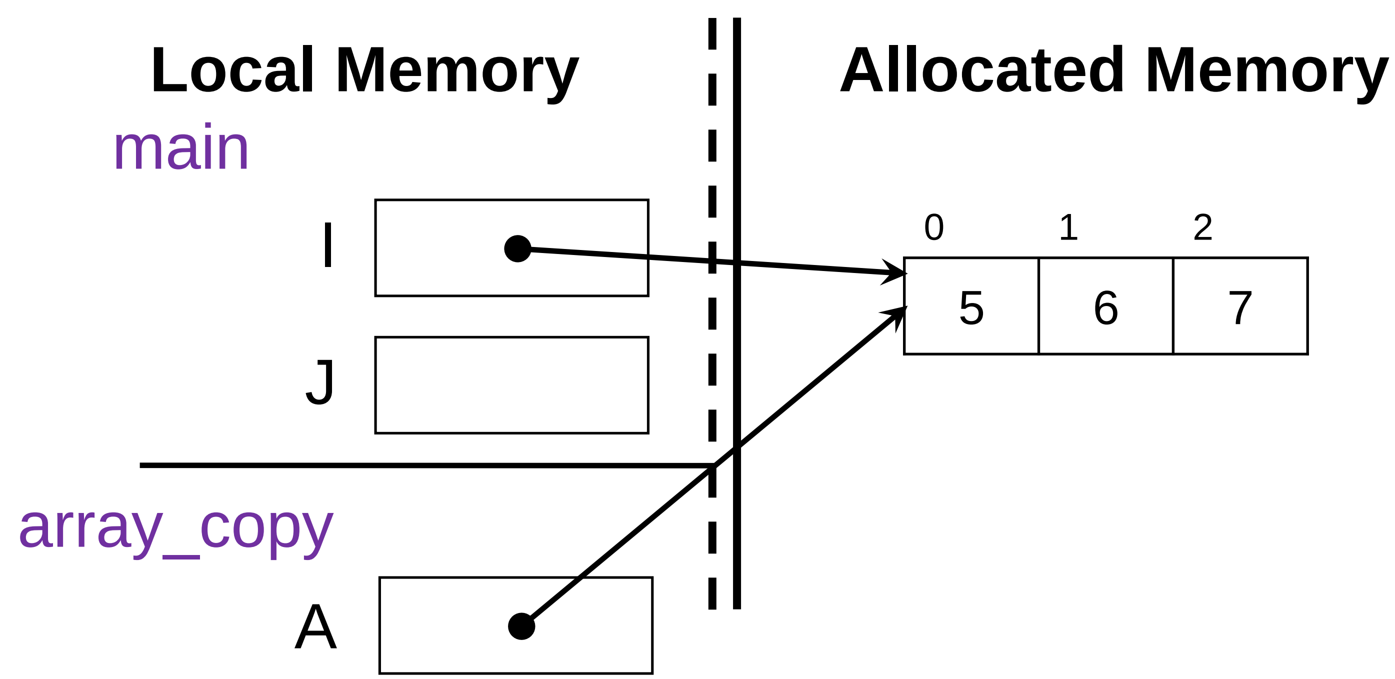


First Attempt

- 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

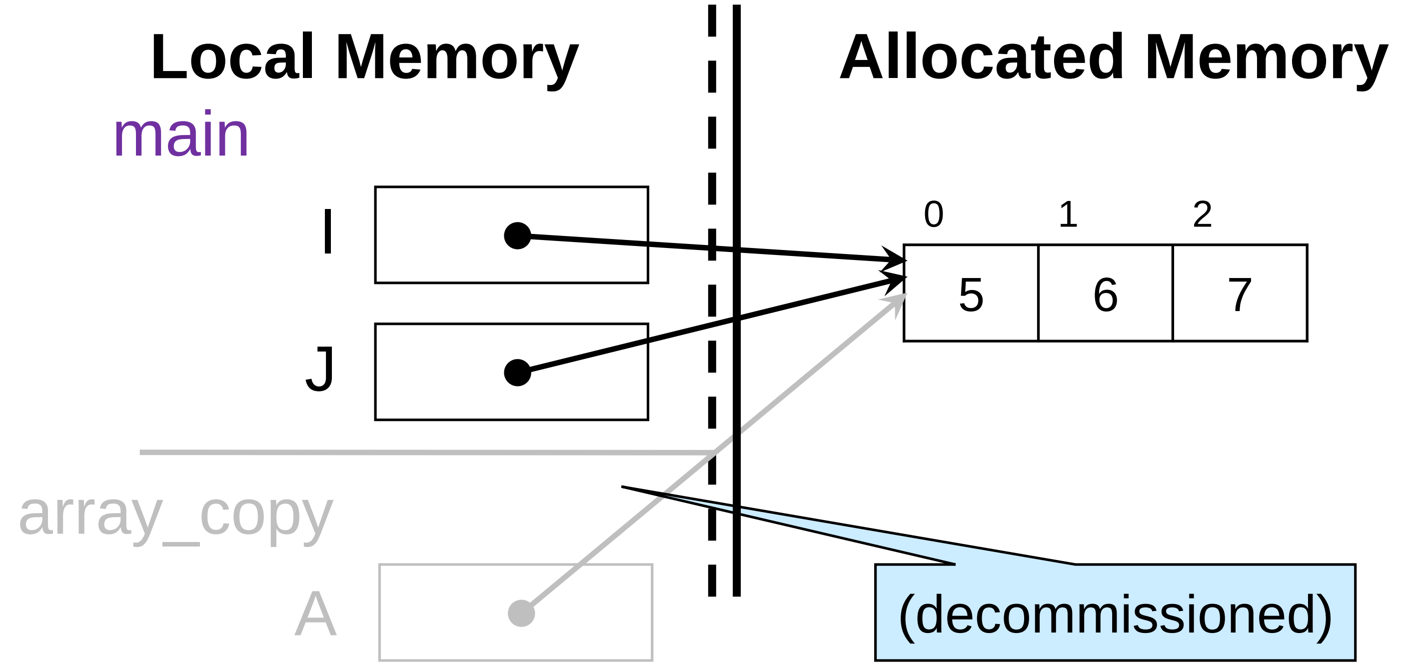
```
int[] array_copy(int[] A) {
  return A;
}

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



First Attempt

Here

```
int[] array_copy(int[] A) {
  return A;
}

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



- 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 ≥ 0
  - Add precondition

    //@requires n >= 0;

Second Attempt

- Is this enough to get the intended behavior?
  - No: n should be the length of A
  - But we can't get the length of an array

```
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;
}
```

- Special **contract-only** function:

  \length(A)

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

Contracts of Array Operations

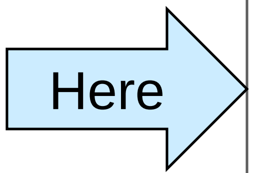
- We can now write strong contracts for the array operations
  - Better precondition of A[i]
  - Postcondition for **alloc_array**

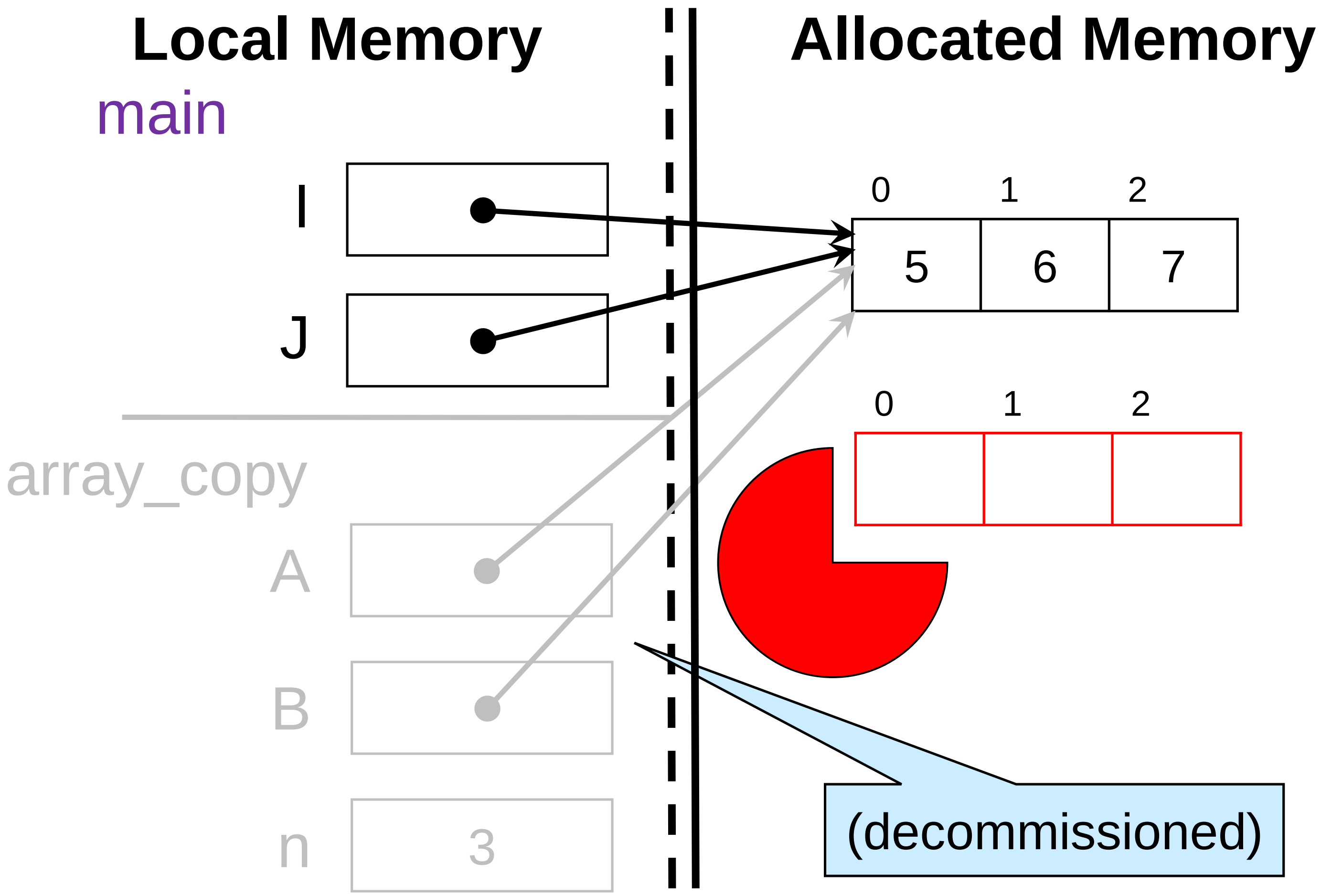
```
alloc_array(type, n)
//@requires n >= 0;
//@ensures \length(\result) == n;

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;
}
```



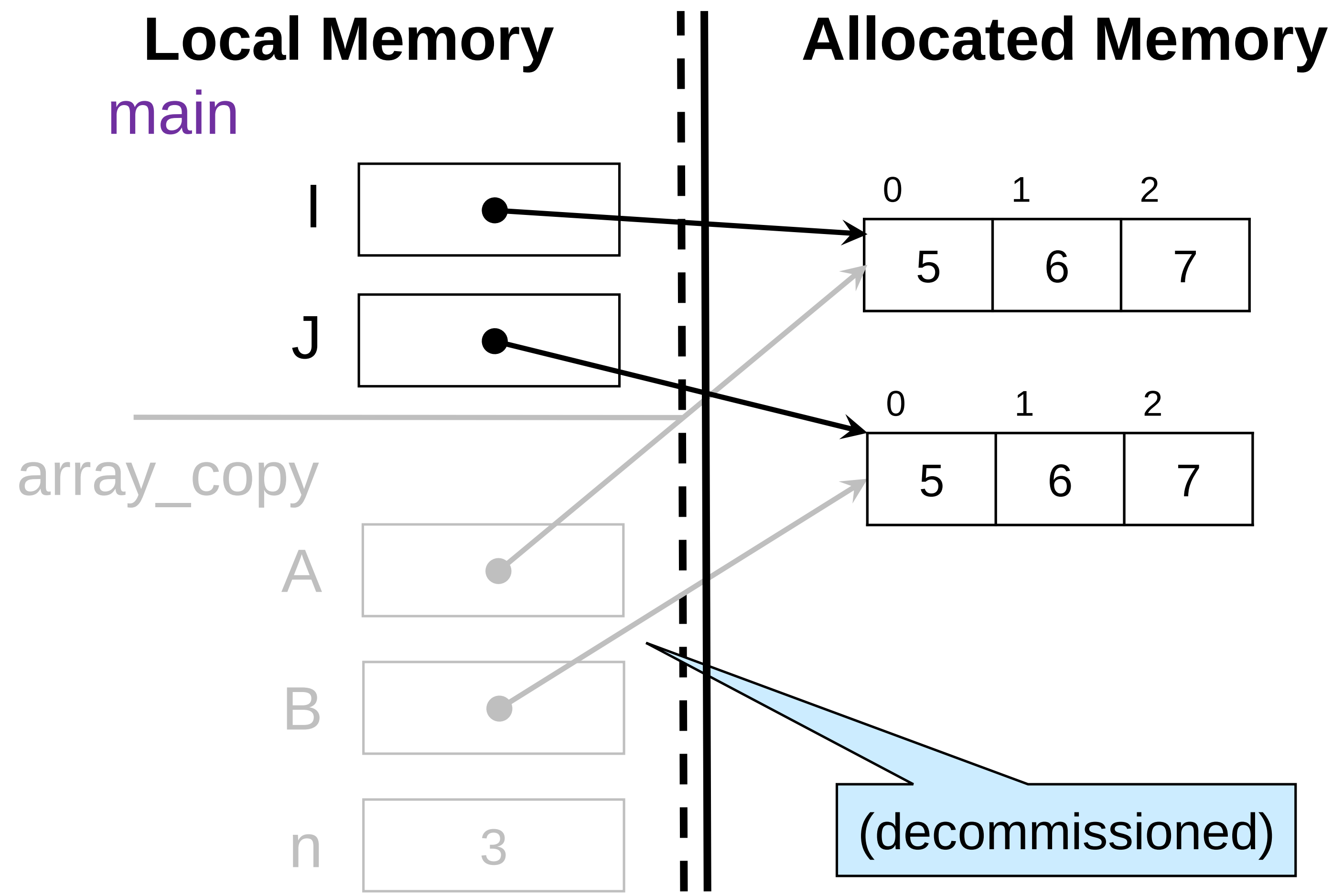


- B is aliased to A
  - Newly allocated array is garbage collected
  - We return an *alias* to I

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 ≥ 0
  - A[i] ?
    - **To show:** 0 ≤ i
      and i < \length(A)
  - B[i] ?
    - **To show:** 0 ≤ i
      and i < \length(B)

```
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;
}
```

Safety of array_copy

alloc_array(int, n)

- **To show:** n ≥ 0

A. n = \length(A) by line 2

B. \length(A) ≥ 0 by \length

C. n ≥ 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 < \length(A)

A. n = \length(A) by line 2
B. i < n by line 5
C. i < \length(A) by A and B

- **To show:** 0 ≤ 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 ≤ i && i < \length(B)

○ *Left as exercise*

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

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

Validity of the Loop Invariant

//@loop_invariant 0 <= i;

INIT:

- **To show:** 0 ≤ i initially

A. i = 0 by line 5

B. 0 ≤ 0 by math

C. 0 ≤ 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;            )
6.   //@loop_invariant 0 <= i;
7.   {
8.
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 ≤ i is preserved

- **To show:** if 0 ≤ i, then 0 ≤ i’

A.	i’ = i+1	by line 5
B.	0 ≤ i	assumption
C.	0 ≤ i+1	by math on B
	only if i ≠ int_max()	by two’s compl.
D.	i < n	by line 5
E.	i ≠ int_max()	by math on D
F.	0 ≤ i’	by A, 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.
10.  re
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)

- "*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

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

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

Safety of Calls to array_copy

Is array_copy(J, 3) safe?

- **To Show:** 3 = \length(J)

A. \length(I) = 3 by line15

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.   i                                    ;
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 = \length(A),
  then n = \length(\result)

A. \length(B) = n — by line 5

B. \result = B — by line 11

C. n = \length(\result) — by A and B

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

This what we expect

Correctness:
postconditions are met
whenever preconditions hold

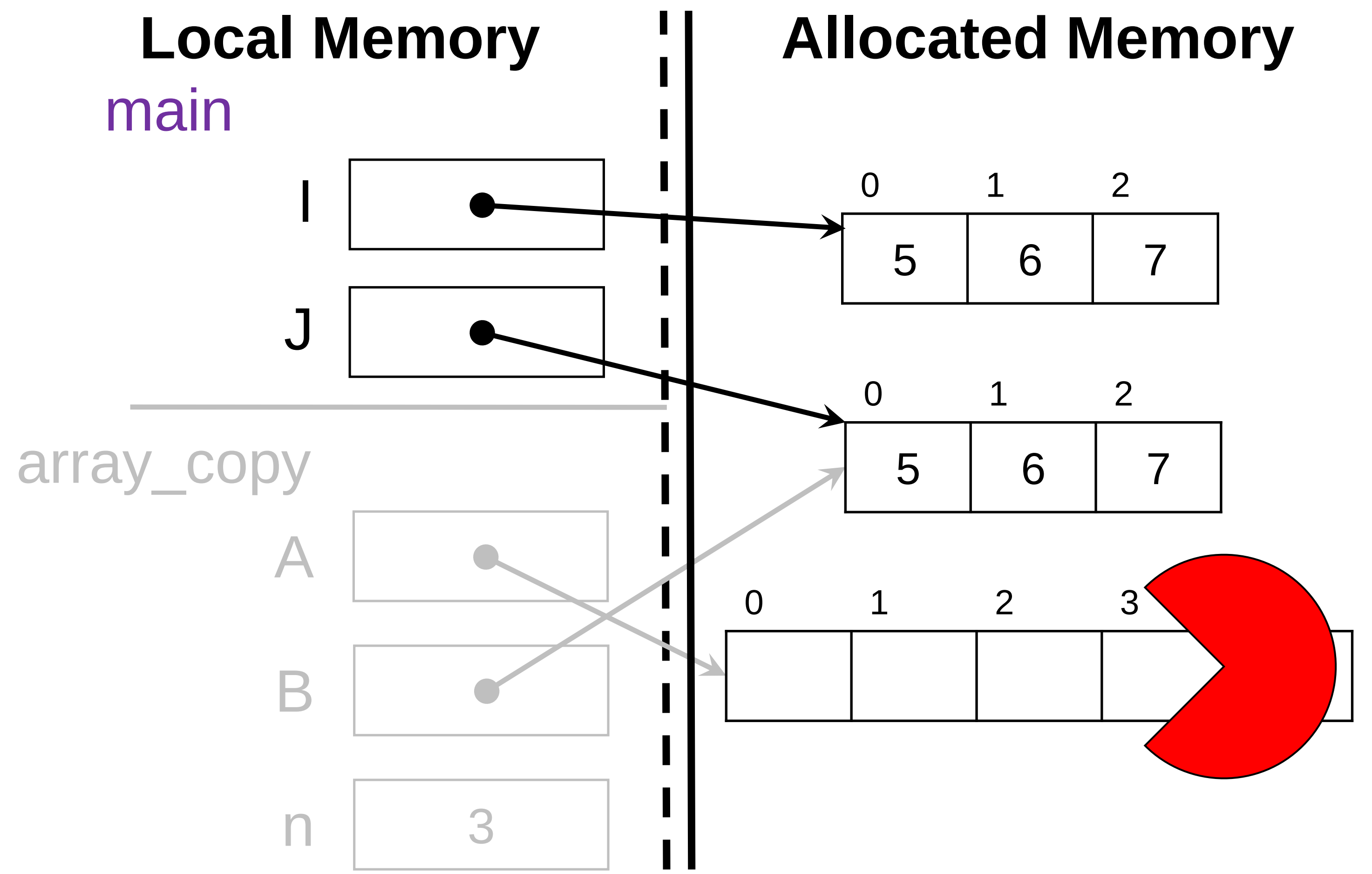
```
1. int[] array_copy(int[] A , int n)
2. //@requires n == \length(A);
3. //@ensures n == \length(\result);
4. {
5.   int[] B = alloc_array(int, n);
6.   for (int i=0; i < n; i++)
7.   //@loop_invariant 0 <= i;
8.   {
9.      B[i] = A[i];
10.  }
11.  return B;
12.}
13.
14.int main() {
15.    …
16.}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

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;
}
```

Here

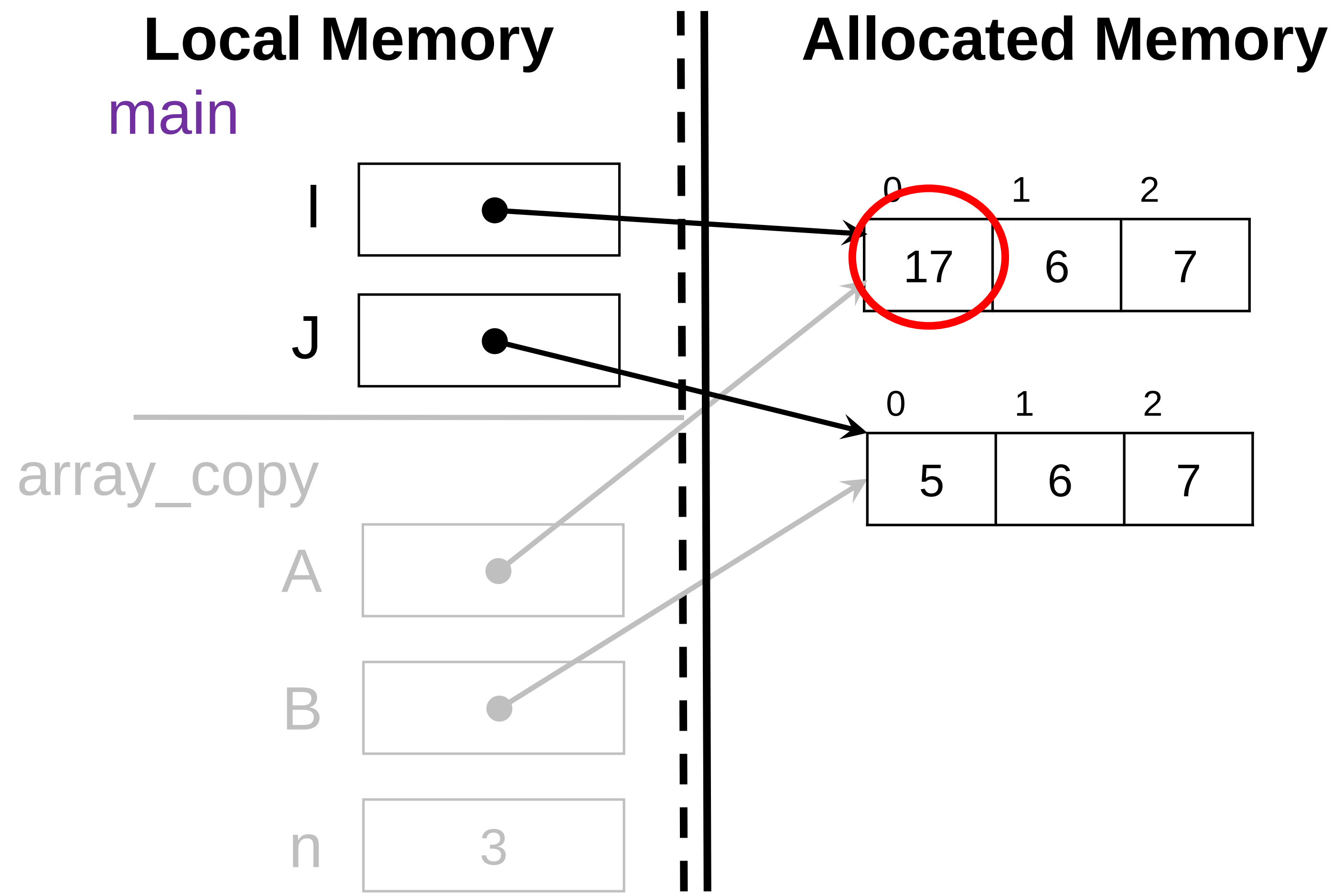


- Only the value of A in array_copy changes
  - The value of I is unchanged
  - The change is **not visible to caller**

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;
}
```

Here



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

Effects on the Caller

- A function can communicate with it caller by
  - returning a value
  - modifying the contents of shared allocated memory