Arrays Continued

Lecture 7

Announcements

- WS1 - Posted and Due Tuesday 2/12 at 11:59pm
- Quiz 2 – Thursday 2/14
- PS2 – Posted and Due Wednesday 2/18 at 11:59pm
  - Even though the due date is after the quiz, you should complete part 1 and part 2 through at least the equals function.

1. Fill in the blanks...

```
export let main = async () => {
   let a: ___A___ = ___B___();
   let b: ___C___ = ___D___(["hi"]);
};

let y = (s: ___E___): ___F___ => {
   return s.length;
};

let z = (): ___G___ => {
   return "hi";
};
```

2. What does the following expression evaluate to: **`foo([4, 8, 16], 4)`**

```
let foo = (a: number[], n: number): number => {
   for (let i = 0; i < a.length; i++) {
      if (a[i] > n) {
         return a[i];
      }
   }
   return -1;
};
```

3. PollEv – What is the *printed output* of this code listing?

```
01 export let main = async () => {
02         let anArray: number[] = [10];
03         append(anArray, 20);
04         append(anArray, 30);
05         print(anArray);
06 };
07
08 let append = (a: number[], n: number):void => {
09         a[a.length] = n;
10 };
11
12 main();
```

`void` functions return nothing.

- There are times when it's useful to have a function that performs a set of steps but doesn't actually result back in a value in your program.

- A function whose return type is **`void`** is often called a **procedure.**

- The **`print`** Function is the perfect example of a procedure
  - What does calling the **`print`** function return?
  - Nothing! It is a procedure the results in output to the screen.

- Procedures are commonly used to evoke effects *outside* itself
  - To make data or graphics appear on a screen
  - To save data to a file
  - To send data to another computer over the internet

Environment Diagrams **with Arrays**

- Now that we are working with arrays, the model of our environment diagram must expand to have a *heap* area in memory.

- The *heap* is also often called *dynamic memory.* It is an area in your program's working memory where *large* and *growing values* are kept.

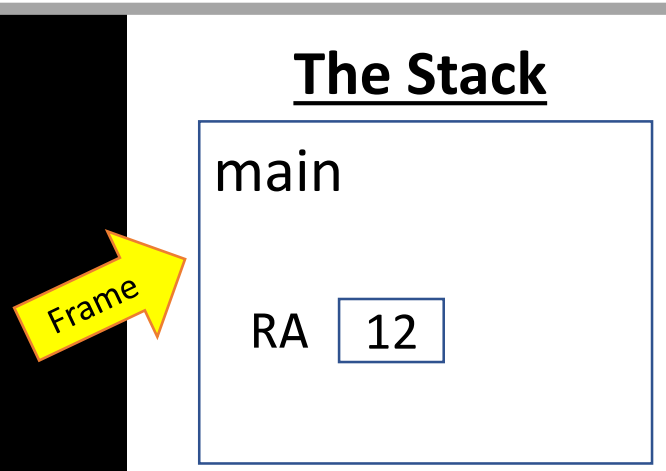
- Array variable names are still established in the current stack frame, however, they will ***refer*** to the actual array data on the heap with ***a pointer***.

- Why? An important reason is it would be time and memory intensive to copy large data structures (like arrays) around between function calls.
  - Each variable name referring to an array is just an address number to a place in the heap.
  - For now, we'll visualize this by drawing arrows! In COMP411, you'll get the nitty gritty.

Function Call - `main`

When a function call is encountered, a new **frame** is added to your stack. Label it with the function's name. Add its return address (RA). Establish parameters.

```
01 export let main = async () => {
02     let anArray: number[] = [10];
03     append(anArray, 20);
04     append(anArray, 30);
05     print(anArray);
06 };
07
08 let append = (a: number[], n: number):void => {
09     a[a.length] = n;
10 };
11
12 main();
```

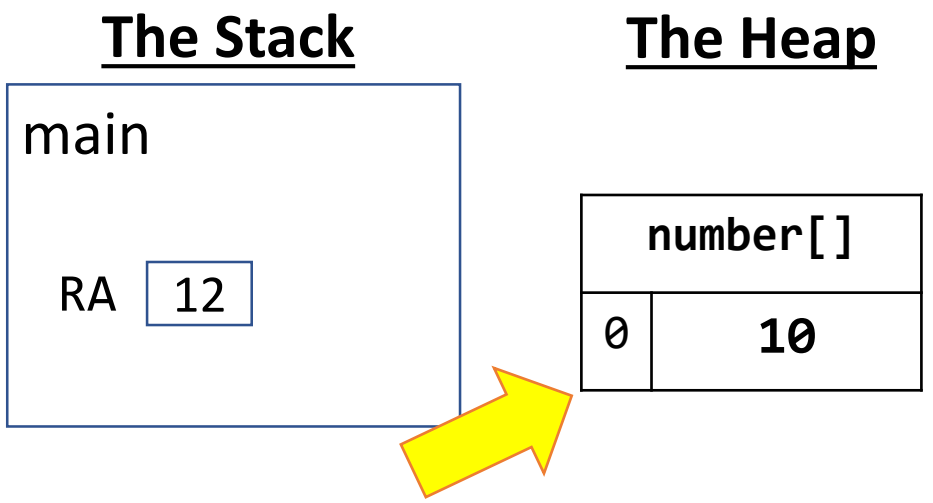


Variable Initialization – New Array

When a declaration and initialization is reached, evaluate the right hand side first.

When an *array literal* is evaluated, establish it on the heap.

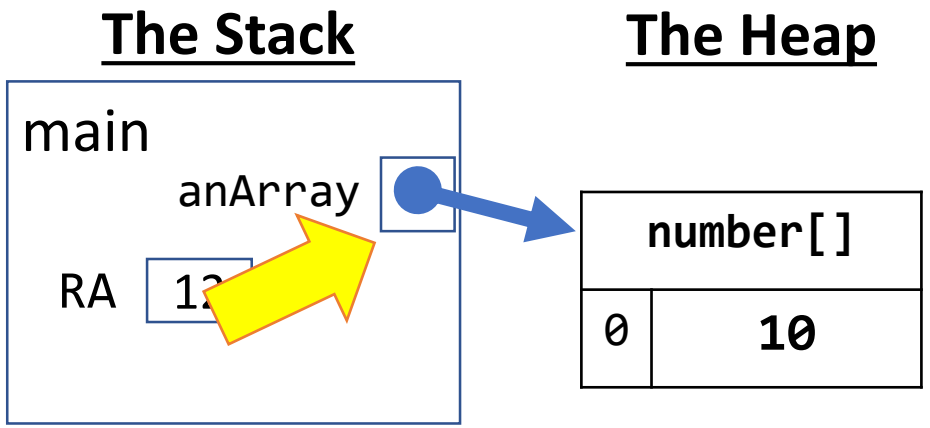
```
01 export let main = async () => {
02     let anArray: number[] = [10];
03     append(anArray, 20);
04     append(anArray, 30);
05     print(anArray);
06 };
07
08 let append = (a: number[], n: number):void => {
09     a[a.length] = n;
10 };
11
12 main();
```



Array Declaration and Assignment

When an array variable is declared and assigned, its label and is established in the current frame. ***Its value is a reference (pointer) to the heap value.***

```
01 export let main = async () => {
02     let anArray: number[] = [10];
03     append(anArray, 20);
04     append(anArray, 30);
05     print(anArray);
06 };
07
08 let append = (a: number[], n: number):void => {
09     a[a.length] = n;
10 };
11
12 main();
```



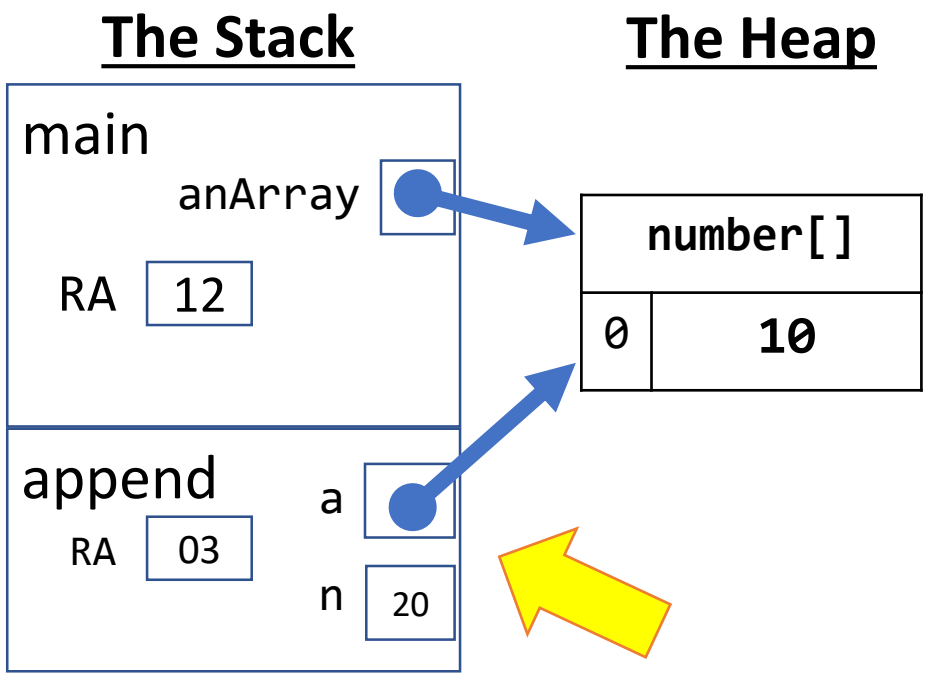
!!! This is a very important concept to understand. Array variables behave very differently from primitive variables because they're *references* to heap value.

Function Call – Establish Frame of Call

Add name, return address, and copy in parameters to be prepared for your jump.

NOTICE! The pointer of anArray was copied, not the array on the heap itself.

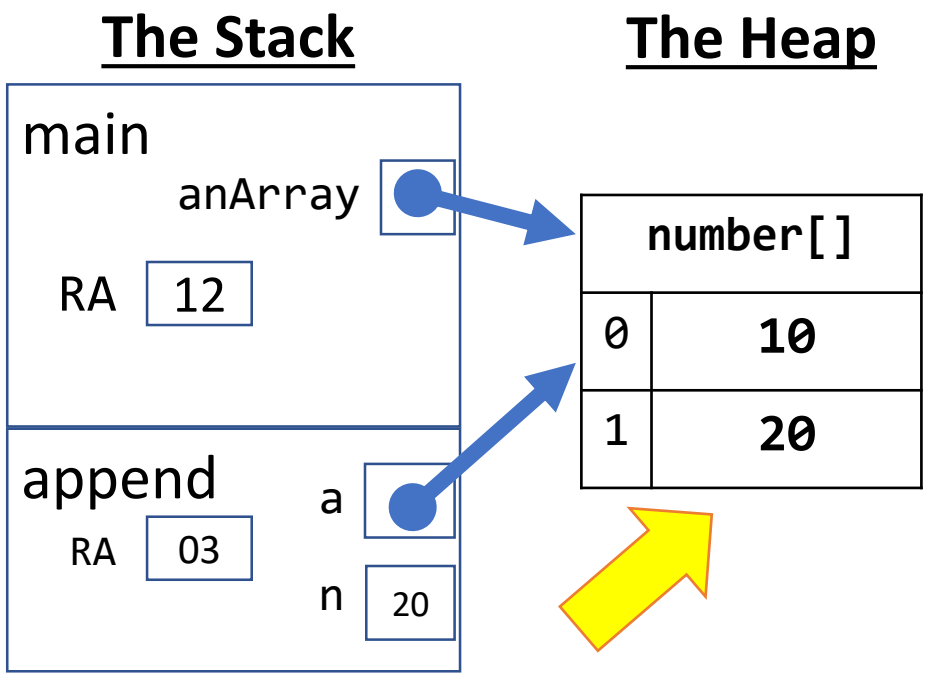
```
01 export let main = async () => {
02     let anArray: number[] = [10];
03     append(anArray, 20);
04     append(anArray, 30);
05     print(anArray);
06 };
07
08 let append = (a: number[], n: number):void => {
09     a[a.length] = n;
10 };
11
12 main();
```



Append Value to Array

Use name resolution to find n, a, and `a.length` to append `20` to the array a.

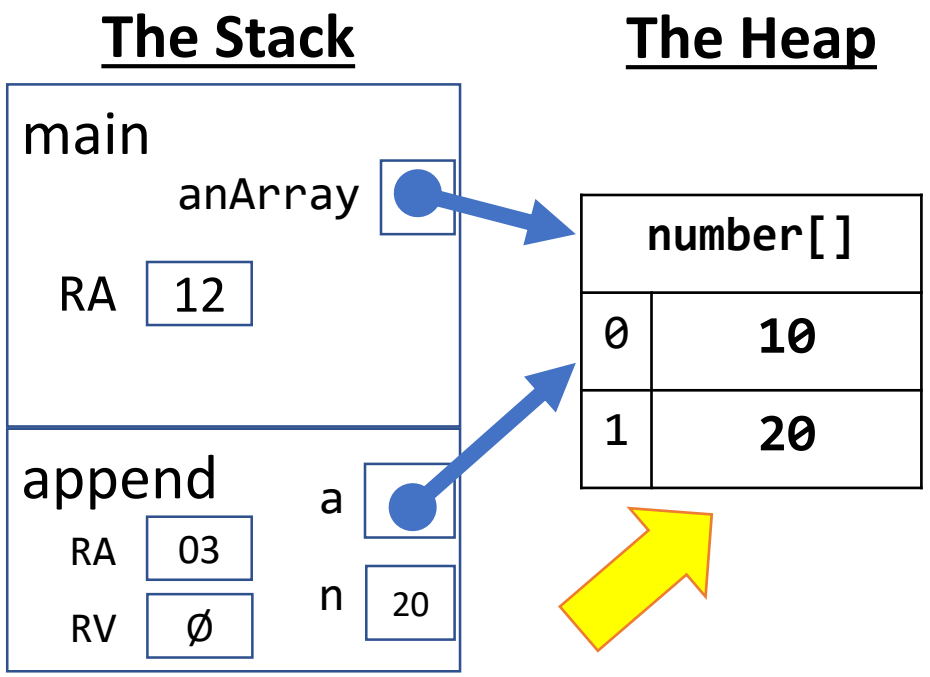
```
01 export let main = async () => {
02     let anArray: number[] = [10];
03     append(anArray, 20);
04     append(anArray, 30);
05     print(anArray);
06 };
07
08 let append = (a: number[], n: number):void => {
09     a[a.length] = n;
10 };
11
12 main();
```



"Return" from a `void` Function (Procedure)

When the end of a void function is reached, the RV is nothing/void and control jumps back to the Return Address (RA).

```
01 export let main = async () => {
02     let anArray: number[] = [10];
03     append(anArray, 20);
04     append(anArray, 30);
05     print(anArray);
06 };
07
08 let append = (a: number[], n: number):void => {
09     a[a.length] = n;
10 };
11
12 main();
```

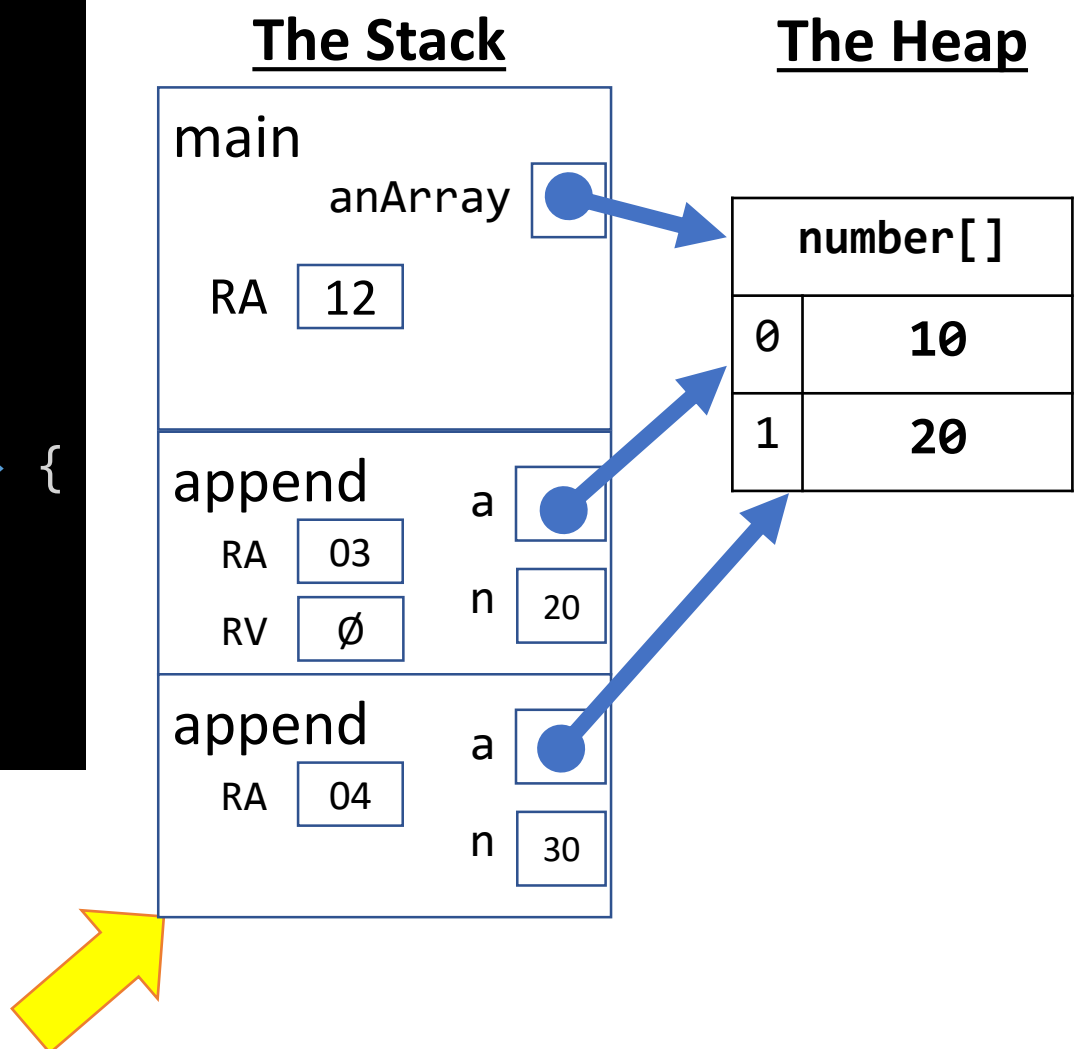


Function Call – Establish Frame of Call

Add name, return address, and copy in parameters to be prepared for your jump.

NOTICE! The pointer of anArray was copied, not the array on the heap itself.

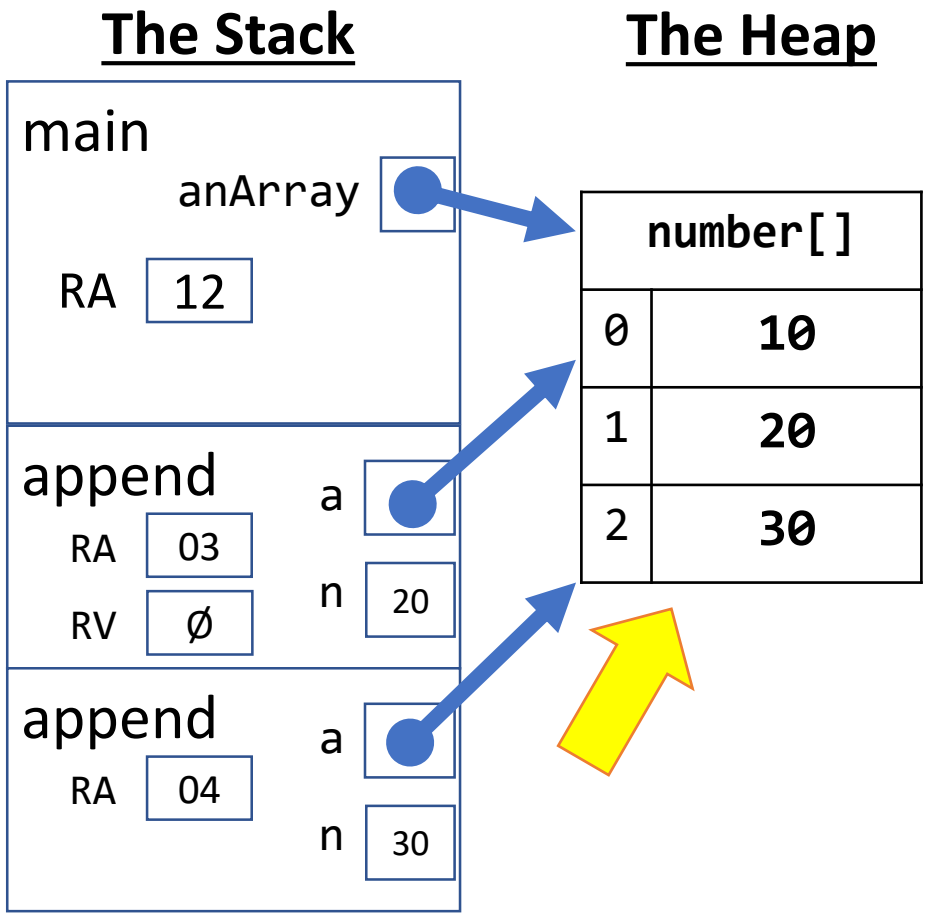
```
01 export let main = async () => {
02     let anArray: number[] = [10];
03     append(anArray, 20);
04     append(anArray, 30);
05     print(anArray);
06 };
07
08 let append = (a: number[], n: number):void => {
09     a[a.length] = n;
10 };
11
12 main();
```



Append Value to Array

Use name resolution to find n, a, and `a.length` to append `30` to the array a.

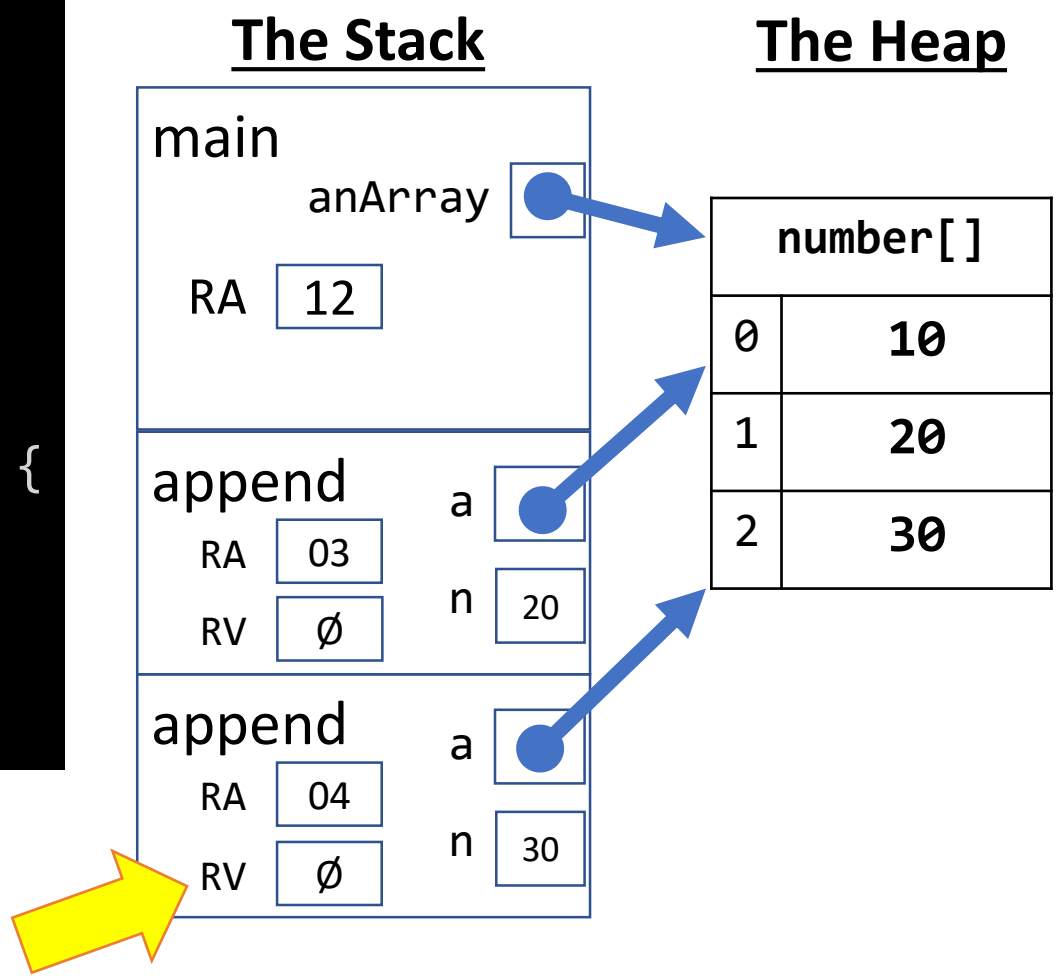
```
01 export let main = async () => {
02     let anArray: number[] = [10];
03     append(anArray, 20);
04     append(anArray, 30);
05     print(anArray);
06 };
07
08 let append = (a: number[], n: number):void => {
09     a[a.length] = n;
10 };
11
12 main();
```



"Return" from a `void` Function (Procedure)

When the end of a void function is reached, the RV is nothing/void and control jumps back to the Return Address (RA).

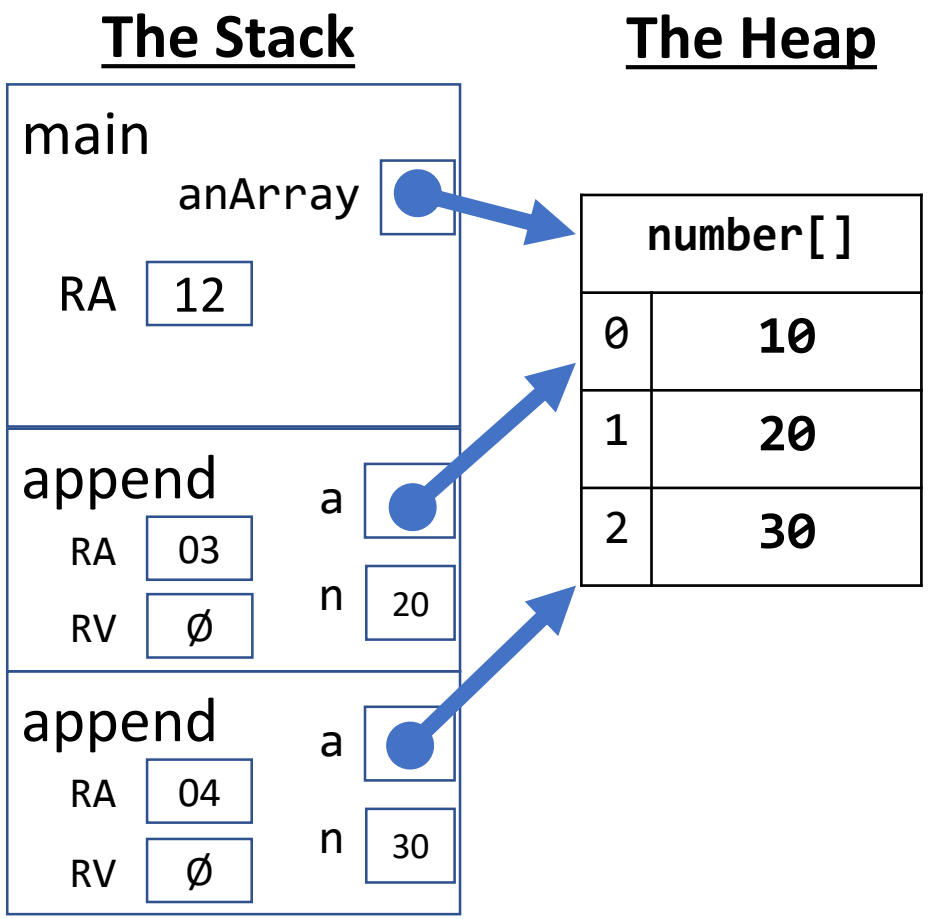
```
01 export let main = async () => {
02     let anArray: number[] = [10];
03     append(anArray, 20);
04     append(anArray, 30);
05     print(anArray);
06 };
07
08 let append = (a: number[], n: number):void => {
09     a[a.length] = n;
10 };
11
12 main();
```



Print Function Call

Output will be print's visual representation of 10, 20, 30.

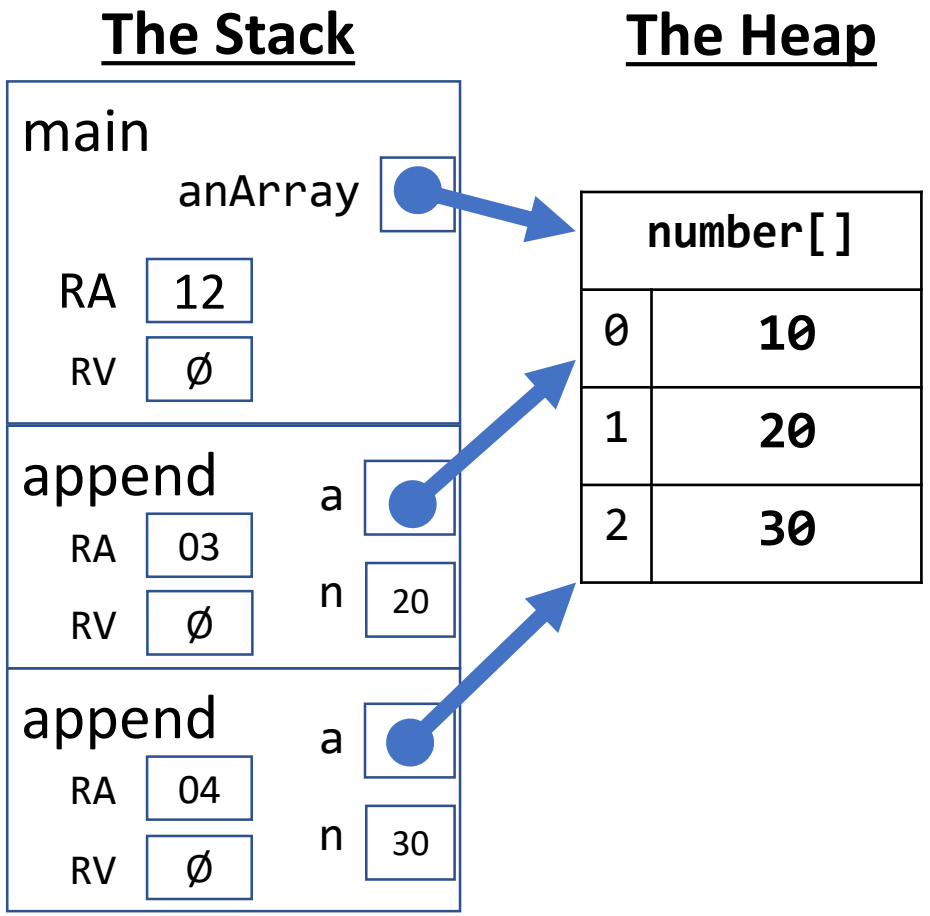
```
01 export let main = async () => {
02     let anArray: number[] = [10];
03     append(anArray, 20);
04     append(anArray, 30);
05     print(anArray);
06 };
07
08 let append = (a: number[], n: number):void => {
09     a[a.length] = n;
10 };
11
12 main();
```



End of main

Reaching the end of main causes control to resume just after the call to main and our program has completed.

```
01 export let main = async () => {
02     let anArray: number[] = [10];
03     append(anArray, 20);
04     append(anArray, 30);
05     print(anArray);
06 };
07
08 let append = (a: number[], n: number):void => {
09     a[a.length] = n;
10 };
11
12 main();
```



Value Types vs. Reference Types

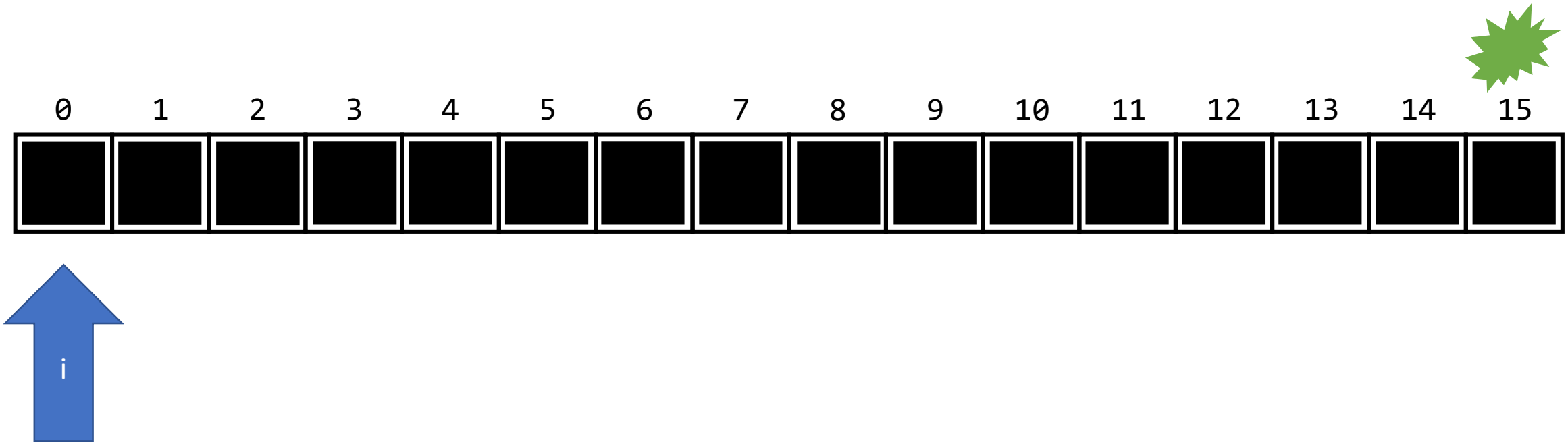
- Primitive types (number, boolean, string[1]) are **value types**
  - Variables hold *copies* of actual values.
  - Assigning one variable to another ***copies the value***.
  - Changing a copied variable's value does not impact original or vice-versa.

- Composite types (arrays, objects[2]) are **reference types**
  - Variables hold *references* to actual values.
  - Assigning one variable to another ***copies the reference***. Both variables now *refer* to the same value in memory.
  - Modifying a referenced value will impact all references to it.

1. The true story of strings is more complicated than we're letting onto in 110. Technically, they're also reference types. However, since they're *immutable*, meaning we cannot change their values we can only establish new strings, so they behave like the other primitives.

2. We'll discuss objects in the next unit. They're another composite data type.

The **Linear Search** Algorithm



Does the word “Yes” exist in this array of Strings?

The **`indexOf`** Function

- Given an array of number values, and a number to search for:

1. Return the index of the number's first occurrence in the array.
2. When the number does not exist in the array, return -1.

- Use Cases:

```
indexOf([1, 2, 3], 1) should return 0
indexOf([1, 2, 3], 2) should return 1
```

- Edge Cases:

```
indexOf([1, 2, 3], 4) should return -1
indexOf([1, 2, 2], 2) should return 1
```

Hands-on: Implementation of **`indexOf`**

- In indexOf.ts

1. Loop through every element of array **`a`**, starting from index **`0`**
2. If an element is equal to **`n`**, then return its **index**
3. Otherwise, return **`-1`**.

- Check-in when you have the test cases passing. pollev.com/compunc

4. PollEv – What is the printed output when the main function below is evaluated?

```
export let main = async () => {
    let a: number[] = [10, 20, 30];
    let b: number = a[1];
    let c: number[] = a;
    b = 100;
    c[2] = 1000;
    print(a);
};
```