Arrays Continued

Lecture 6

PollEv.com/compunc

Announcements

- PS2 - Posted last Thursday, Due Friday 1/7 at 11:59pm
  - If you have not completed at least 3 of the 9 functions, then you are at a high risk of not making the deadline.

- WS1 - Posts Tonight and Due ???

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;
};
```

`void` functions return nothing.

- There are times when it's useful to have a function that performs a set of steps that do not result in a returned value.

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

- The **`print`** Function is an 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
  - To modify or *mutate* a reference to an array or object

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

```
export let main = async () => {
        let anArray: number[] = [10];
        append(anArray, 20);
        append(anArray, 30);
        print(anArray);
};

let append = (a: number[], n: number): void => {
        a[a.length] = n;
};

main();
```

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.

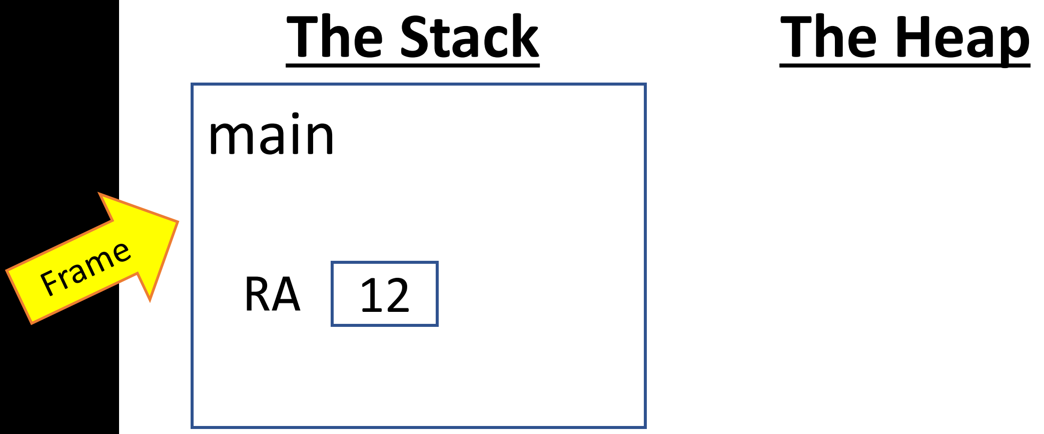
- Array variable names are still established in the current stack frame, however, they will ***refer*** to the actual array data on the heap with ***a pointer arrow***.

- 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 COMP211/311, you'll get the nitty gritty.

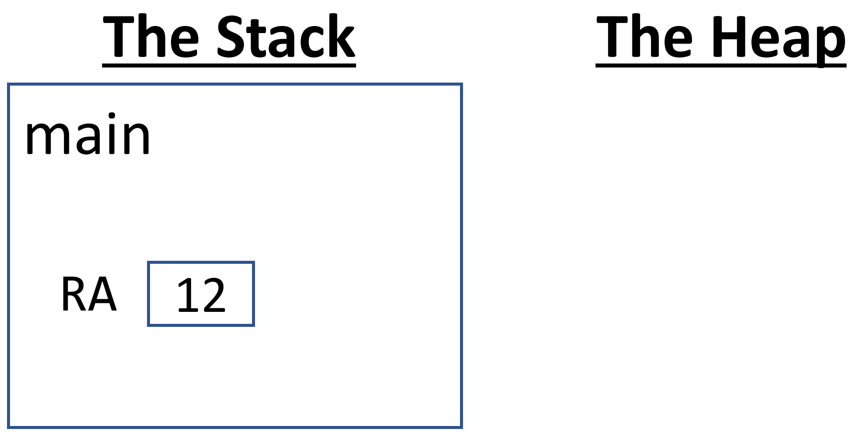
Function Call - `main`

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();
```



```
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();
```

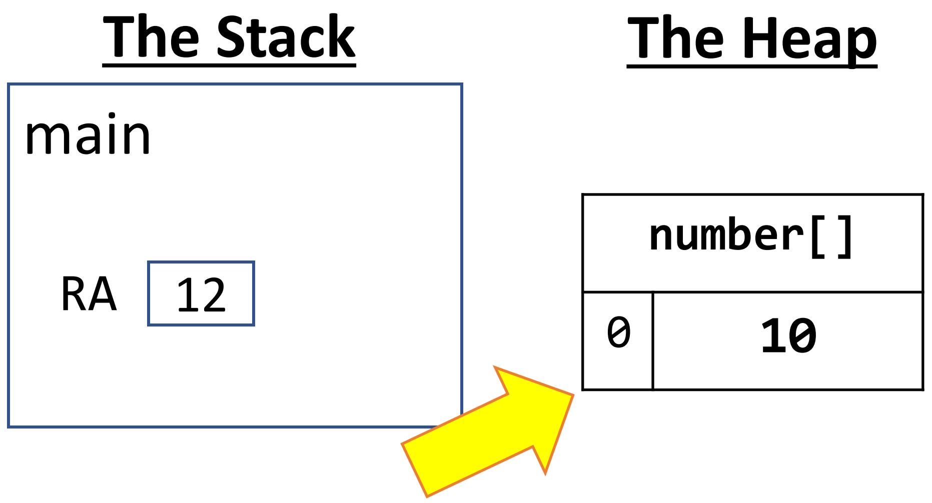


Variable Initialization – New Array

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

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

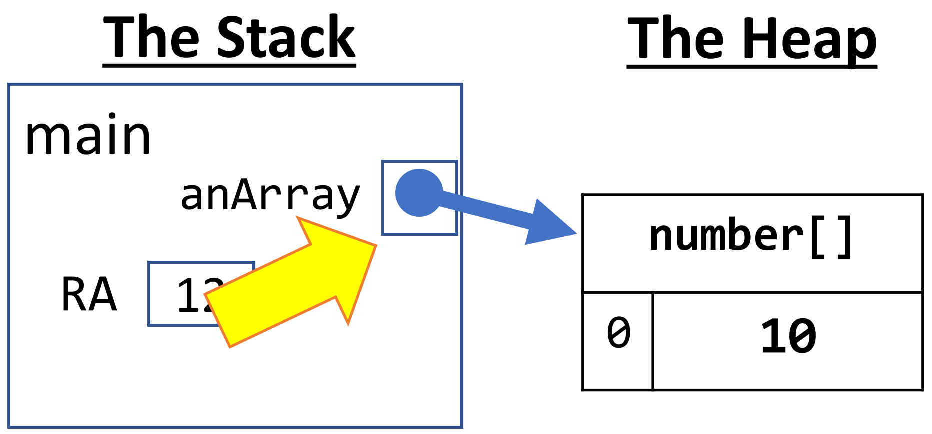
```
01 export let main = async () => {
02     let anArray: number[] = [10];
03     append(anArray, 20);
04     append(anArray, 30);
05     print(anArray);
06 };
07
08 let append = (a: number[], n: number):void => {
09     a[a.length] = n;
10 };
11
12 main();
```



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();
```



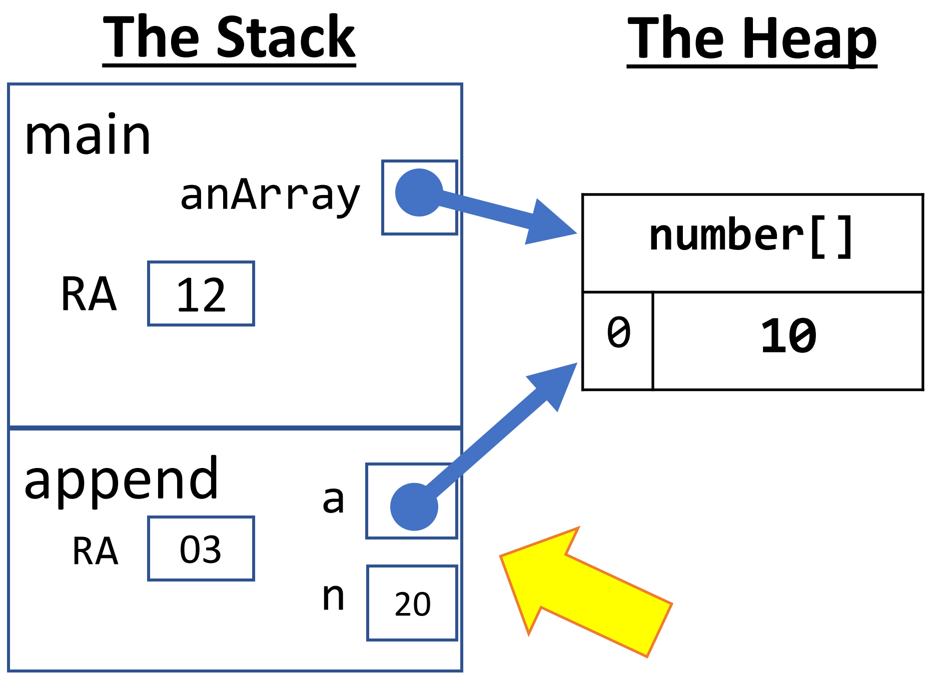
!!! This is a very important concept to understand. Array variables behave very differently from primitive variables because they're *references* to heap value.

Function Call – Establish Frame of Call

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

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

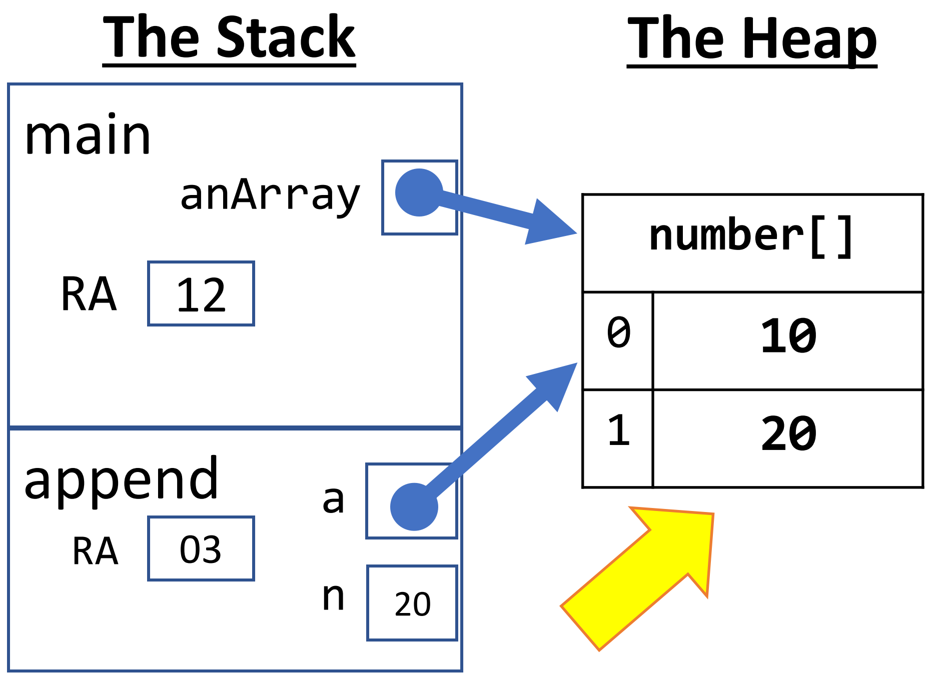
```
01 export let main = async () => {
02     let anArray: number[] = [10];
03     append(anArray, 20);
04     append(anArray, 30);
05     print(anArray);
06 };
07
08 let append = (a: number[], n: number):void => {
09     a[a.length] = n;
10 };
11
12 main();
```



Append Value to Array

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

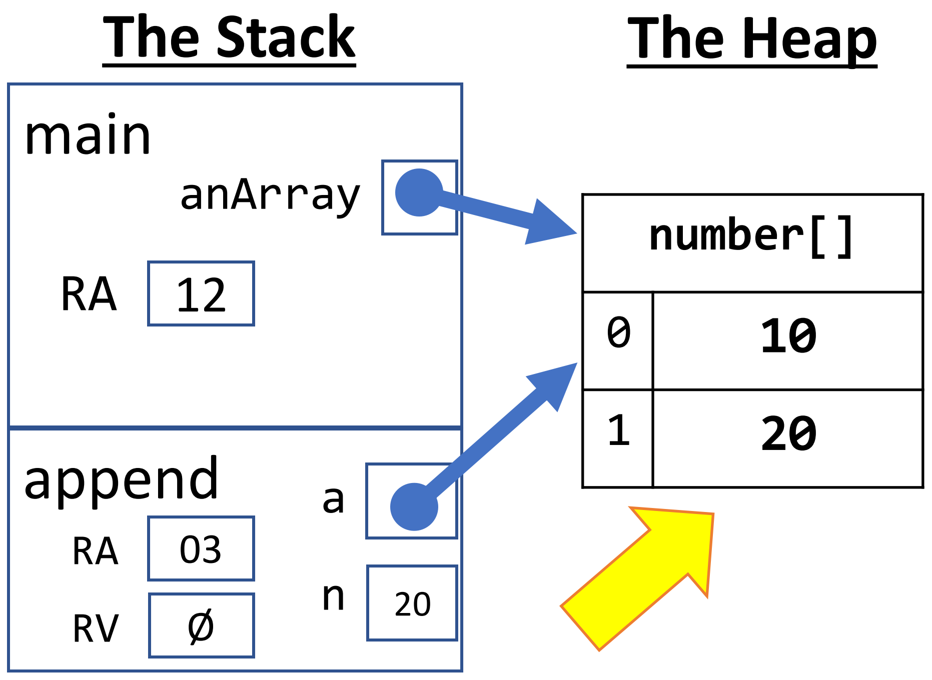
```
01 export let main = async () => {
02     let anArray: number[] = [10];
03     append(anArray, 20);
04     append(anArray, 30);
05     print(anArray);
06 };
07
08 let append = (a: number[], n: number):void => {
09     a[a.length] = n;
10 };
11
12 main();
```



"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();
```

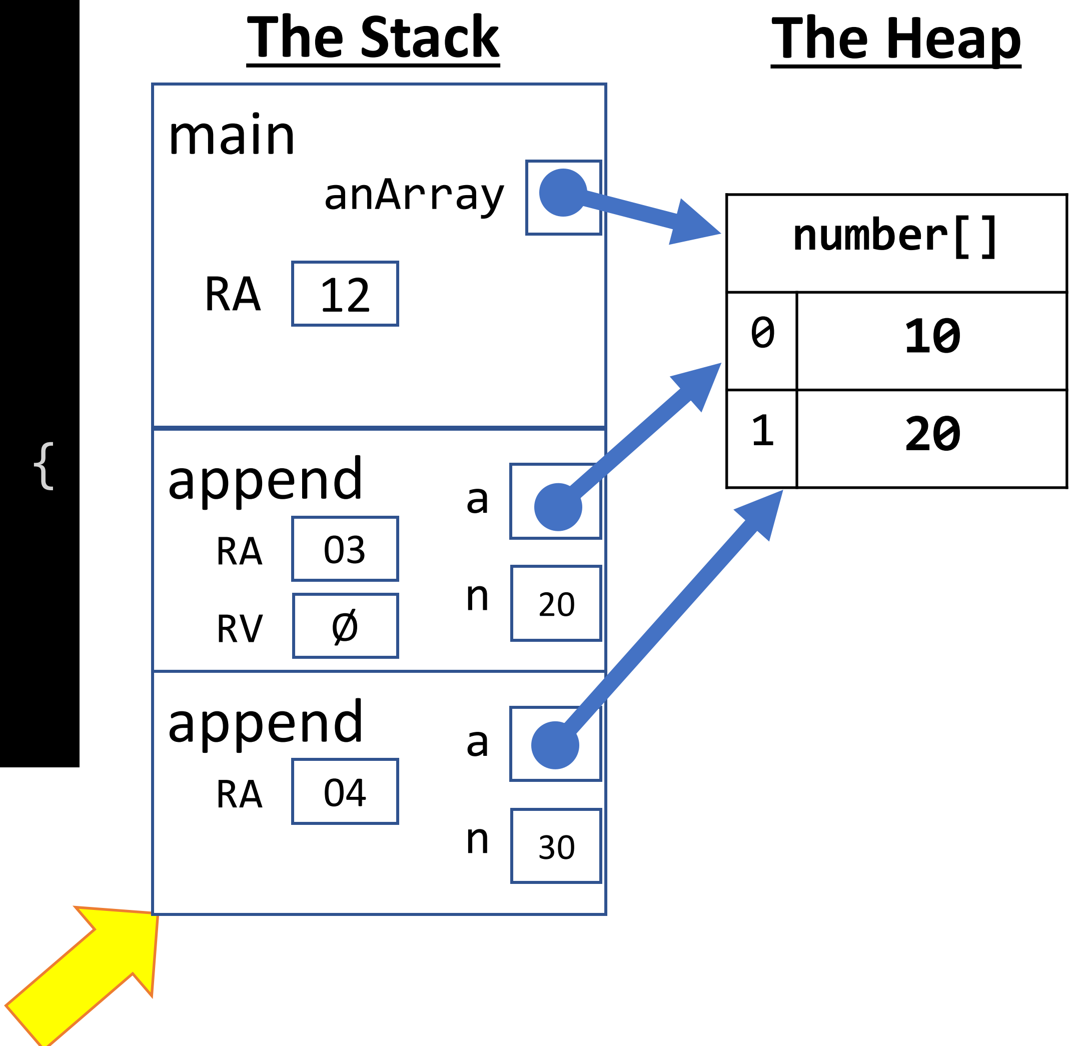


Function Call – Establish Frame of Call

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.

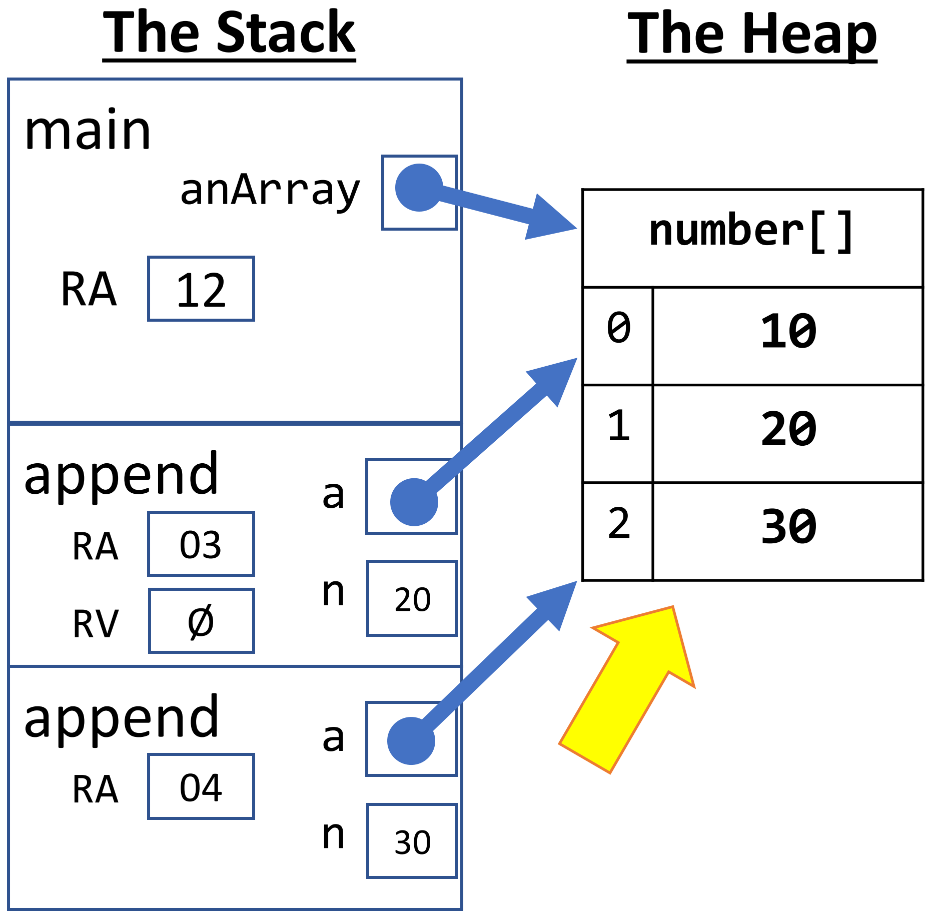
```
01 export let main = async () => {
02     let anArray: number[] = [10];
03     append(anArray, 20);
04     append(anArray, 30);
05     print(anArray);
06 };
07
08 let append = (a: number[], n: number):void => {
09     a[a.length] = n;
10 };
11
12 main();
```



Append Value to Array

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

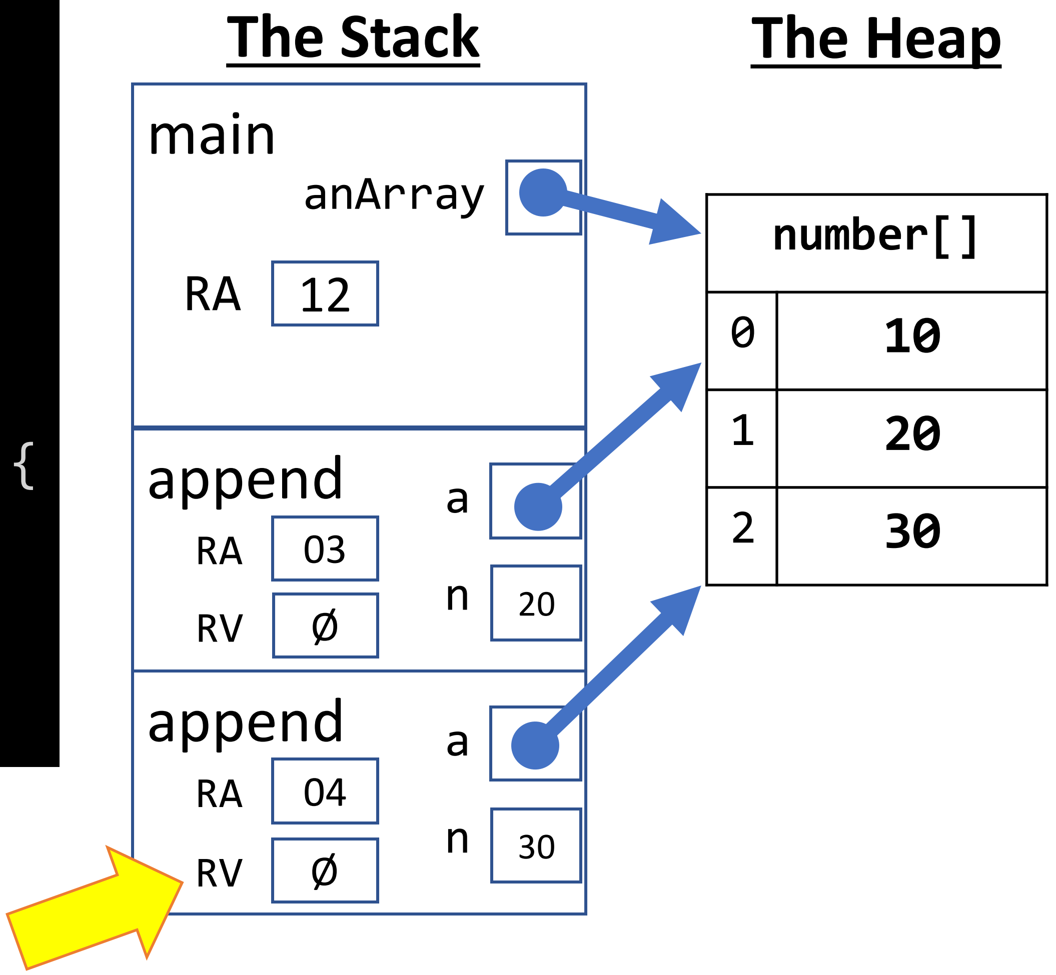
```
export let main = async () => {
    let anArray: number[] = [10];
    append(anArray, 20);
    append(anArray, 30);
    print(anArray);
};

let append = (a: number[], n: number):void => {
    a[a.length] = n;
};

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).

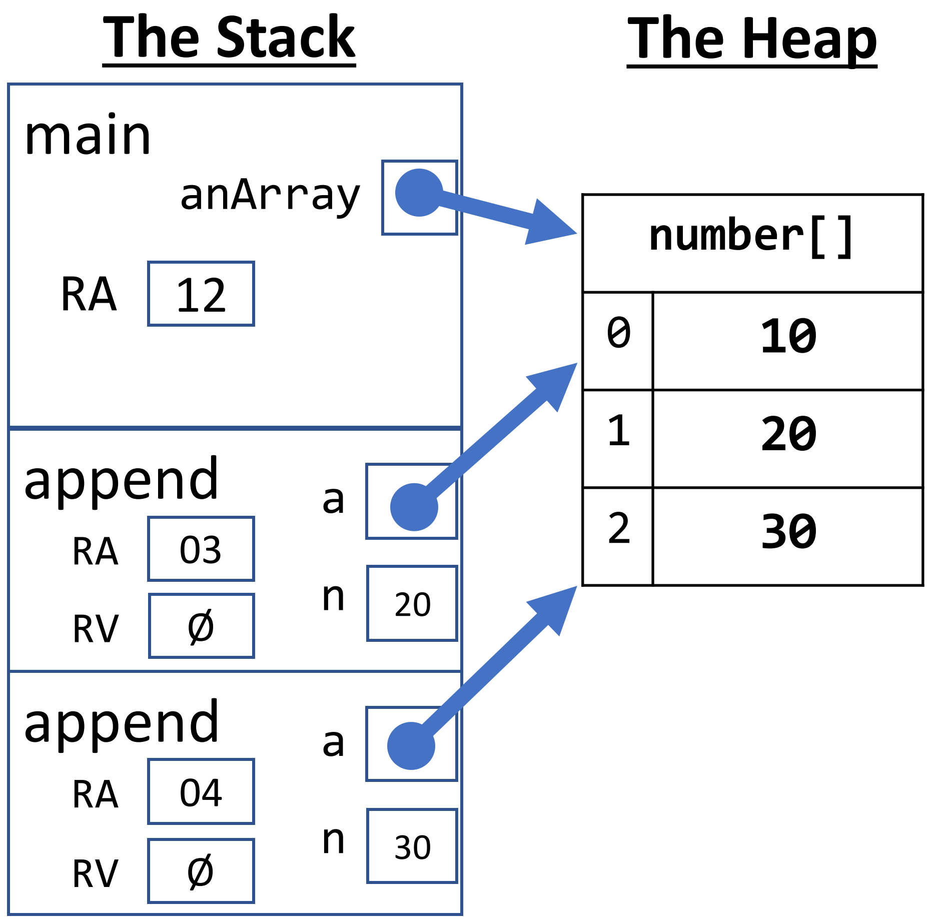
```
01 export let main = async () => {
02     let anArray: number[] = [10];
03     append(anArray, 20);
04     append(anArray, 30);
05     print(anArray);
06 };
07
08 let append = (a: number[], n: number):void => {
09     a[a.length] = n;
10 };
11
12 main();
```



Print Function Call

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

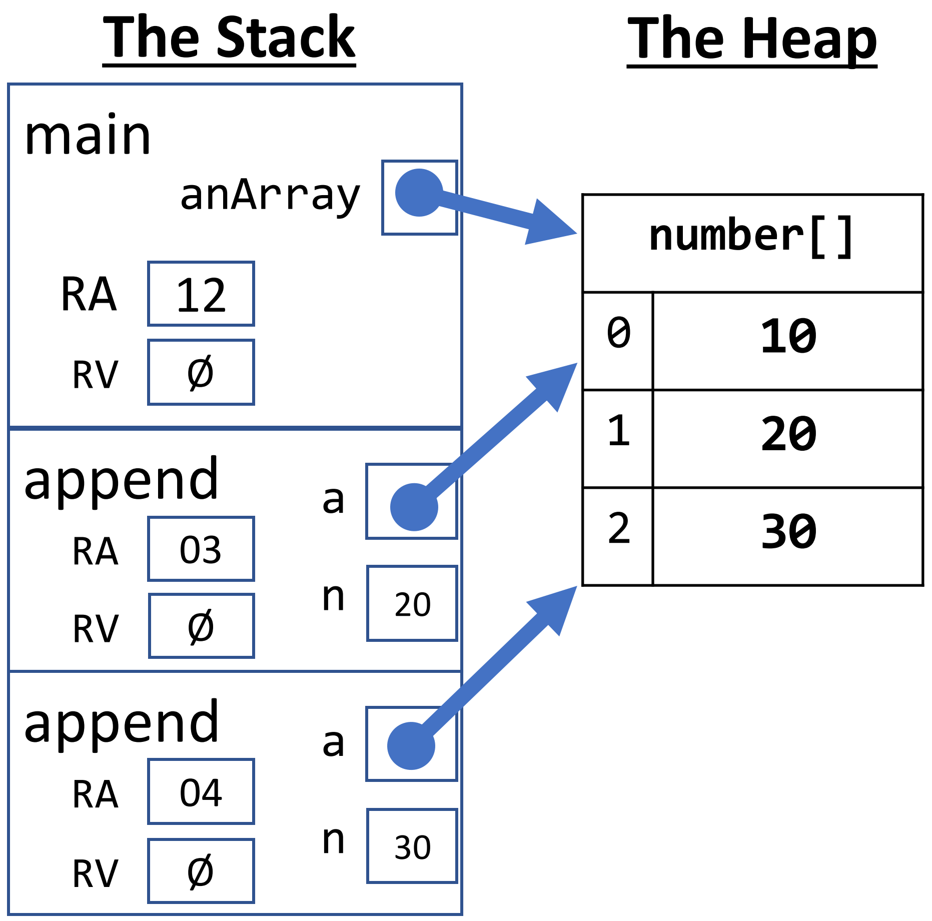
```
01 export let main = async () => {
02     let anArray: number[] = [10];
03     append(anArray, 20);
04     append(anArray, 30);
05     print(anArray);
06 };
07
08 let append = (a: number[], n: number):void => {
09     a[a.length] = n;
10 };
11
12 main();
```



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

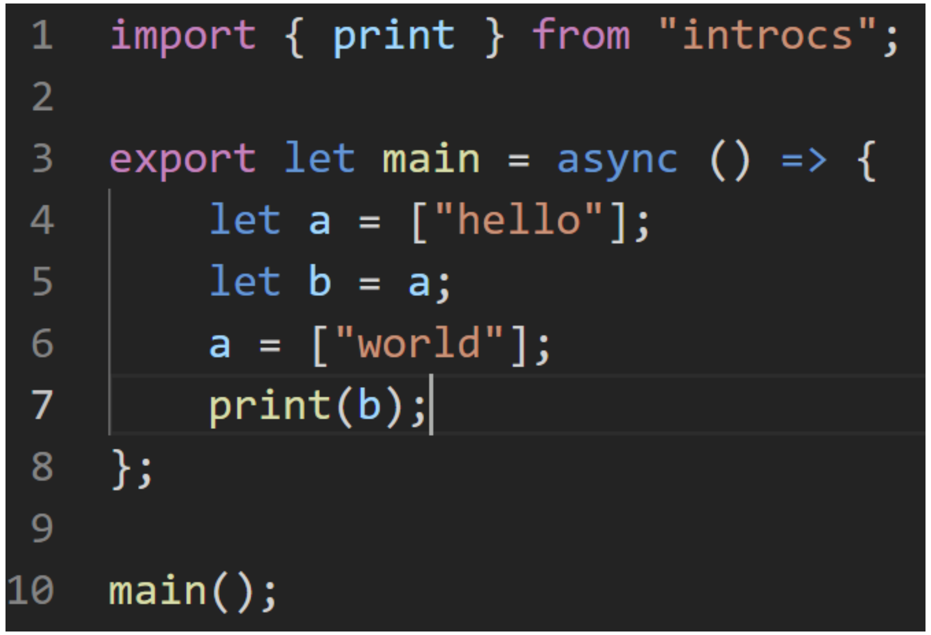
- Primitive types (number, boolean, string[1]) are **value types**
  - Variables hold *copies* of actual values.
  - Assigning one variable to another ***copies the value***.
  - Changing a copied variable's value does not impact original or vice-versa.

- 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.

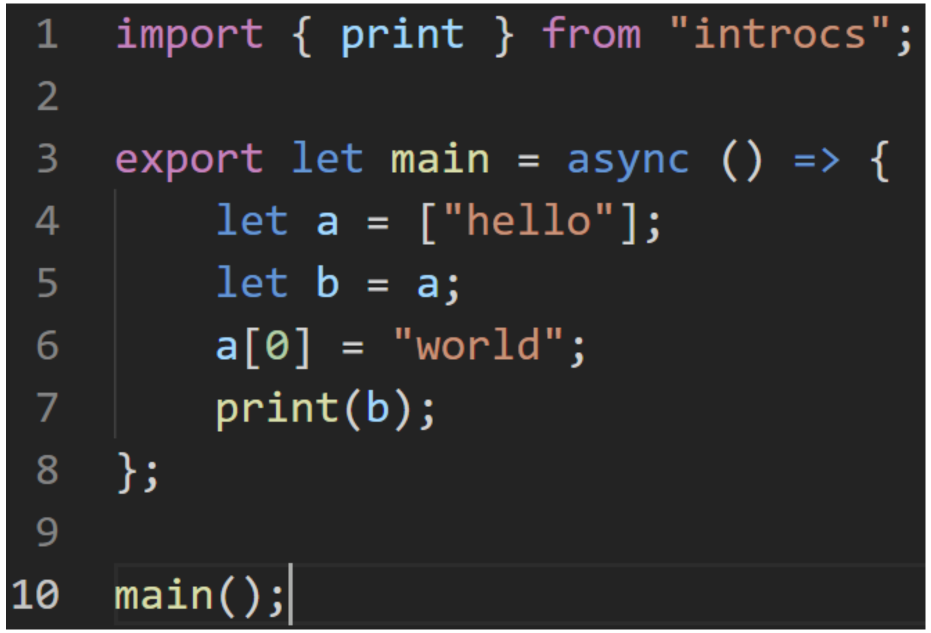
1. The true story of strings is more complicated than we're letting onto in 110. Technically, they're also reference types. However, since they're *immutable*, meaning we cannot change their values we can only establish new strings, they behave like value primitives.

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

4. Given the two code listings below, draw environment diagrams for each. Then respond on PollEv with whether b's value at line 7 is the same or different between examples. Finally, in the first example, what is b's value either way?



```
1   import { print } from "introcs";
2
3   export let main = async () => {
4       let a = ["hello"];
5       let b = a;
6       a = ["world"];
7       print(b);
8   };
9
10  main();
```



```
1   import { print } from "introcs";
2
3   export let main = async () => {
4       let a = ["hello"];
5       let b = a;
6       a[0] = "world";
7       print(b);
8   };
9
10  main();
```

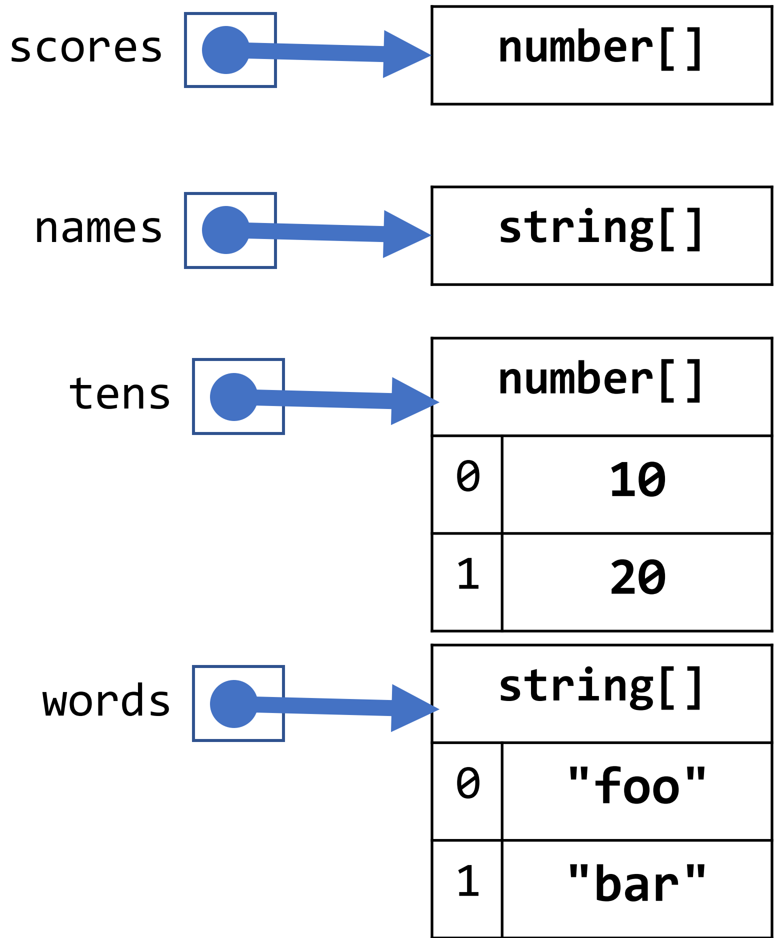
Array Literals Create New Arrays on Heap (1/2)

- Every time the processor reaches an *array literal* value a new array is constructed on the Heap
- Array literals can result in an *empty array*:

```
let scores: number[] = [];
let names: string[]  = [];
```

- Array literals can also *initialize* an array with values:

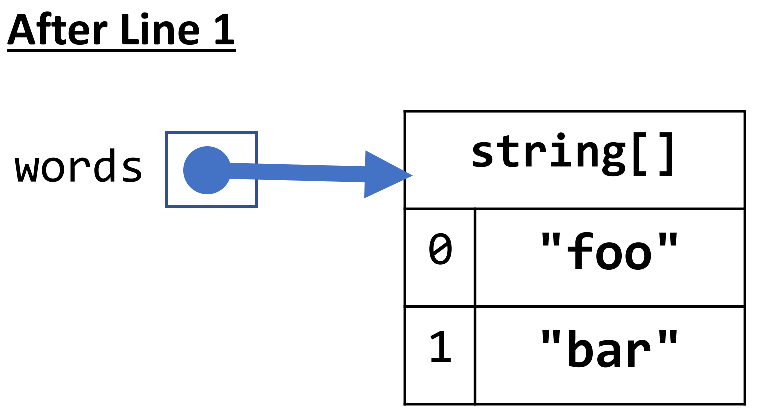
```
let tens: number[] = [10, 20];
let words: string[]  = ["foo", "bar"];
```

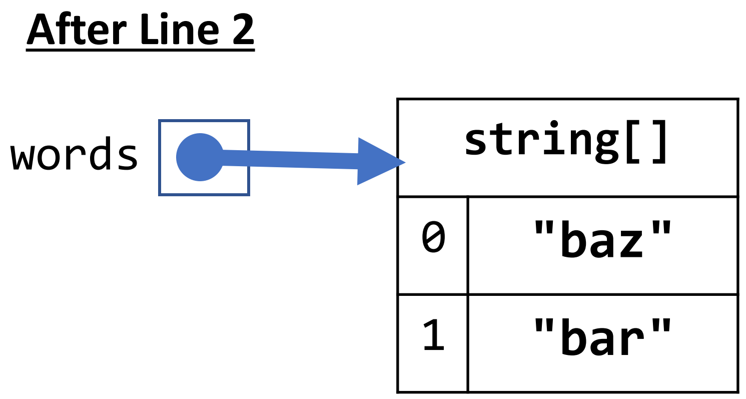


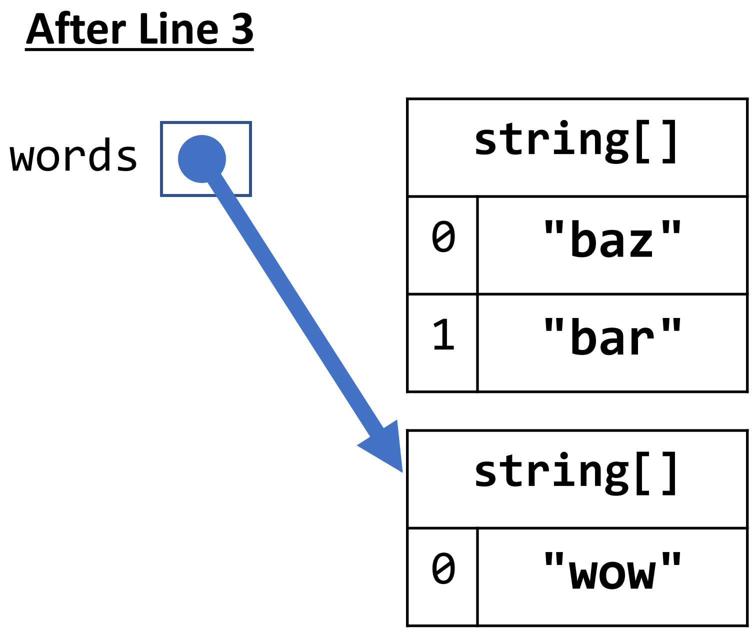
Reassignment of an Array Literal (2/2)

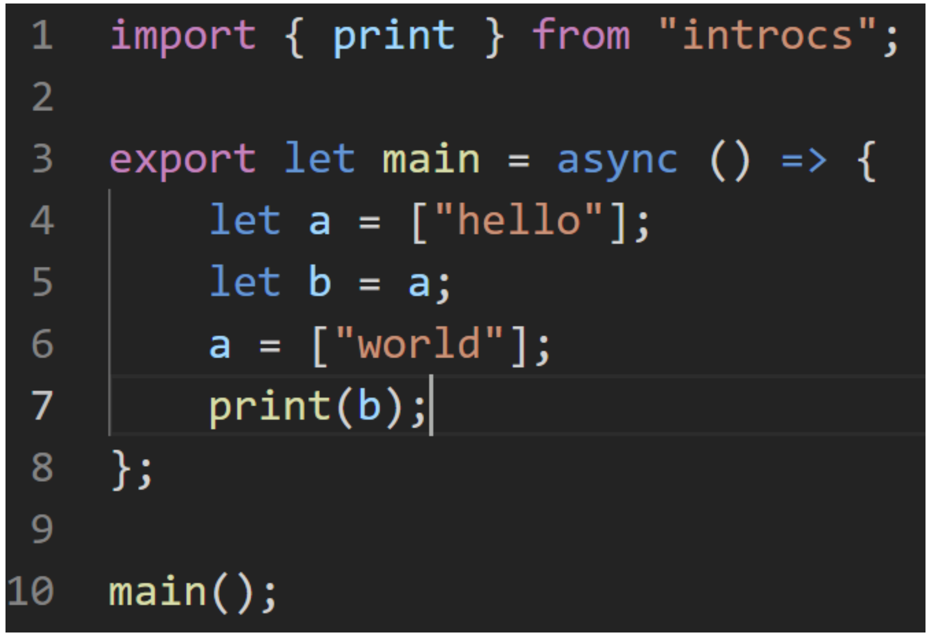
- Consider the following code:

```
Line 1) let words: string[]  = ["foo", "bar"];
Line 2) words[0] = "baz";
Line 3) words = ["wow"];
```

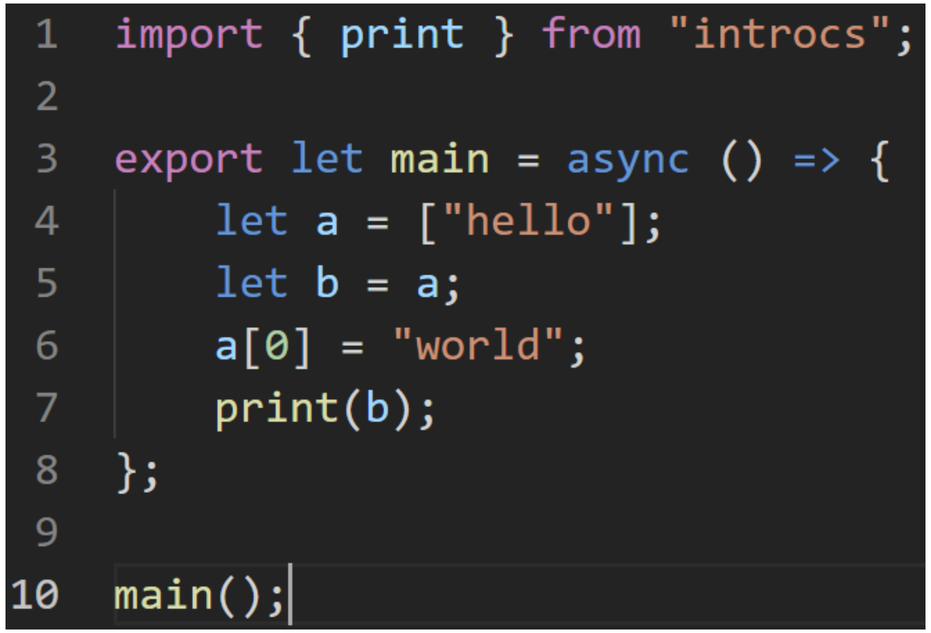








```
import { print } from "introcs";

export let main = async () => {
    let a = ["hello"];
    let b = a;
    a = ["world"];
    print(b);
};

main();
```



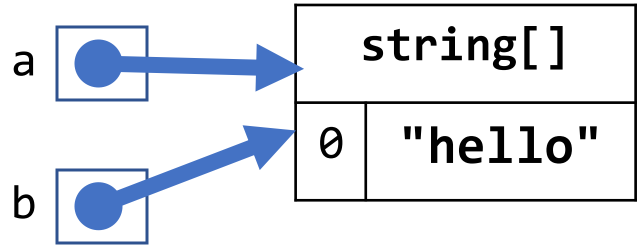
```
import { print } from "introcs";

export let main = async () => {
    let a = ["hello"];
    let b = a;
    a[0] = "world";
    print(b);
};

main();
```

References *ALWAYS* refer to a value in the Heap

- When you access a reference, its value is its pointer (arrow).

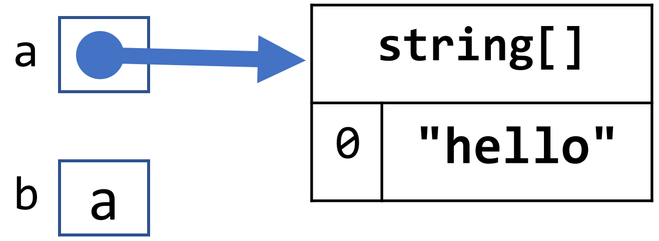
```
let a = ["hello"];
let b = a;
```

Results in:

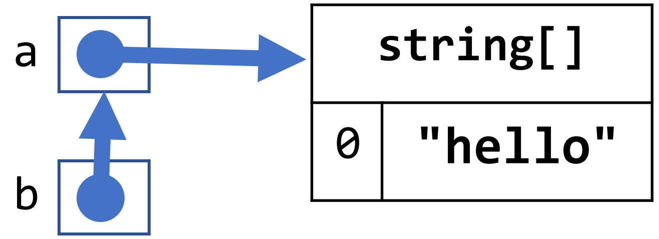


Why? Notice a's value *is the pointer.* So when we assign a's value to b, b's value is *the same pointer*.

Wrong! Never do this:

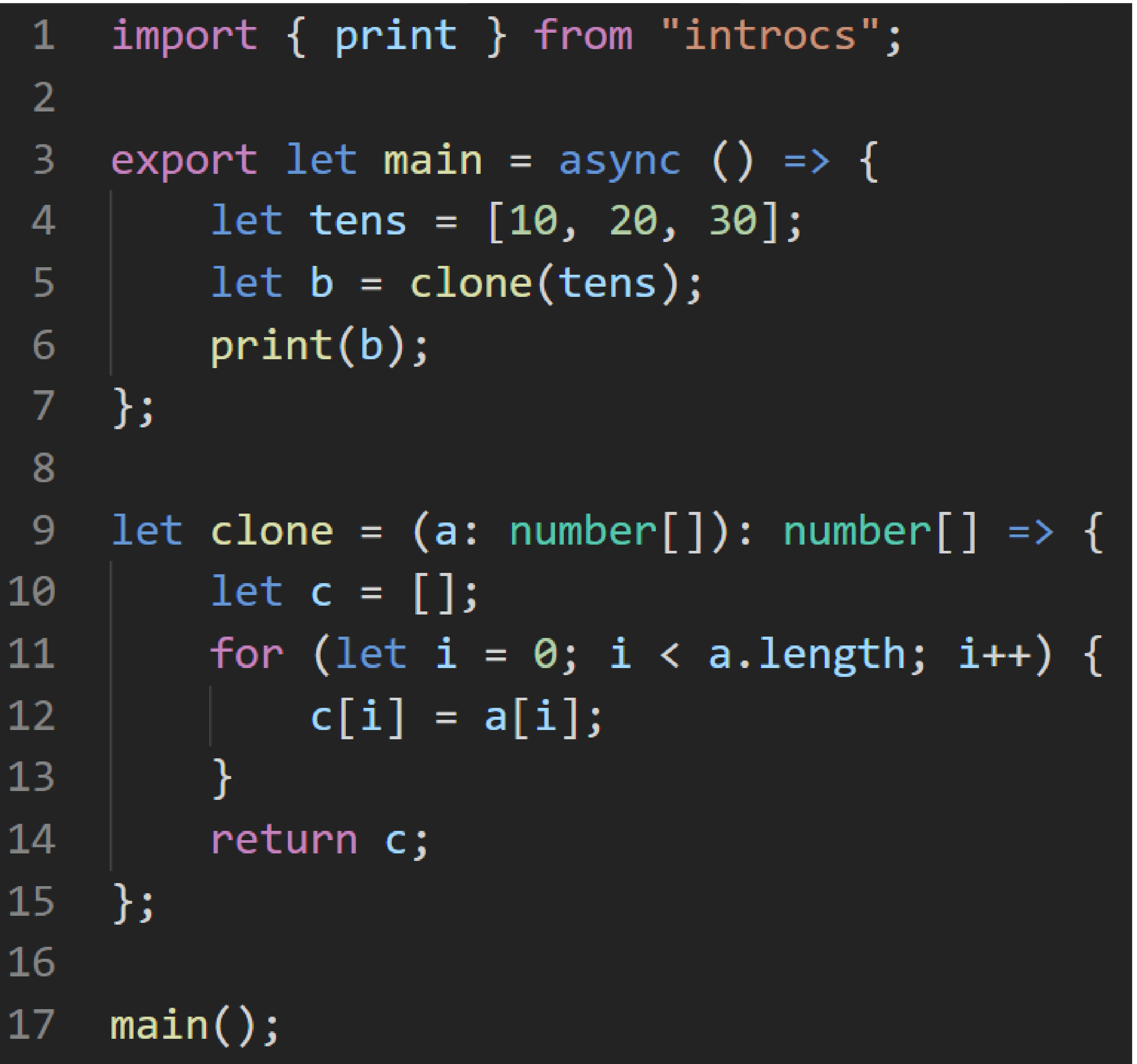


nor this:



Disclaimer for future CS courses: There are lower-level programming languages (C, C++, Rust) where this is possible.

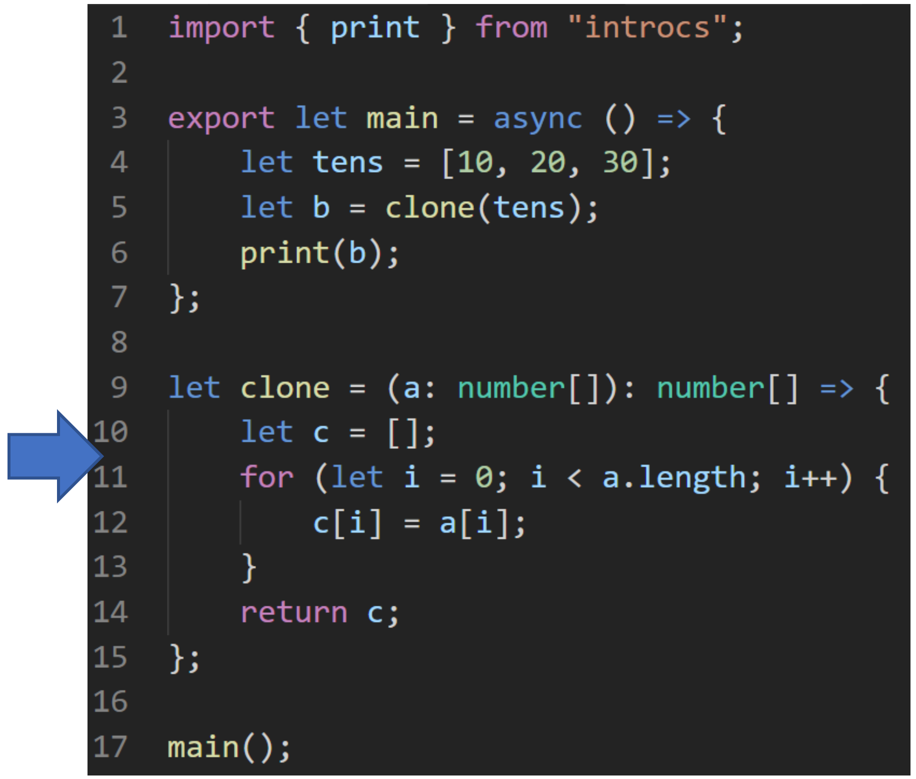
5. Trace an Environment Diagram for the Program Listing Below



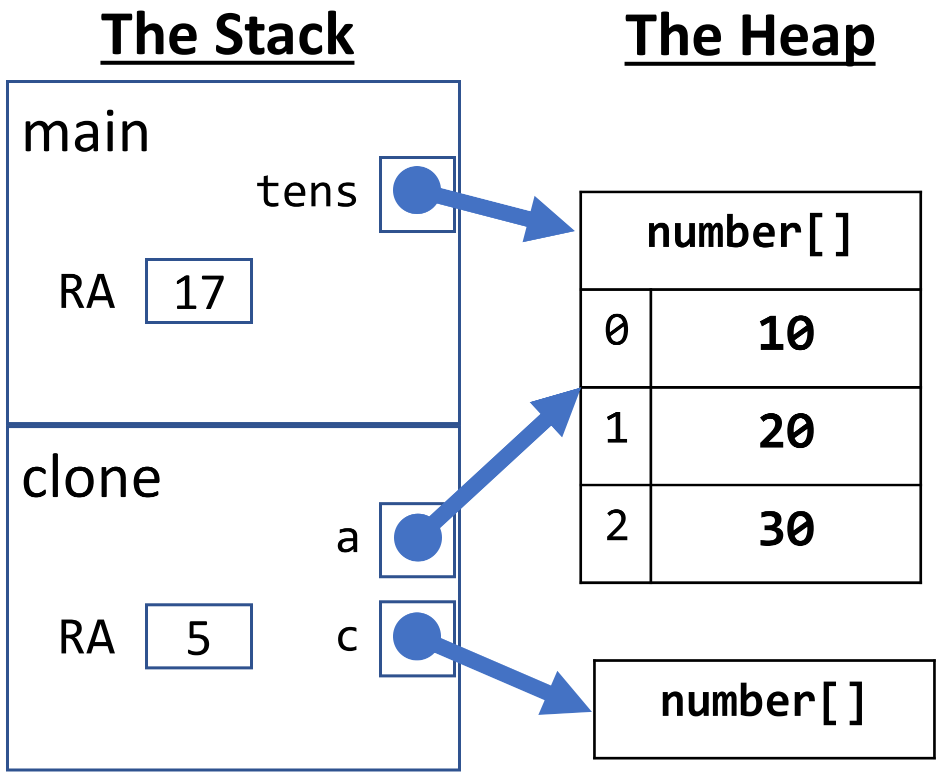
```
import { print } from "introcs";

export let main = async () => {
    let tens = [10, 20, 30];
    let b = clone(tens);
    print(b);
};

let clone = (a: number[]): number[] => {
    let c = [];
    for (let i = 0; i < a.length; i++) {
        c[i] = a[i];
    }
    return c;
};

main();
```

Array Arguments Pass their Pointer (Arrow)

- An array argument results in the parameter having the same pointer.
  - Just like one array variable assigned to another gets a copy of its pointer.



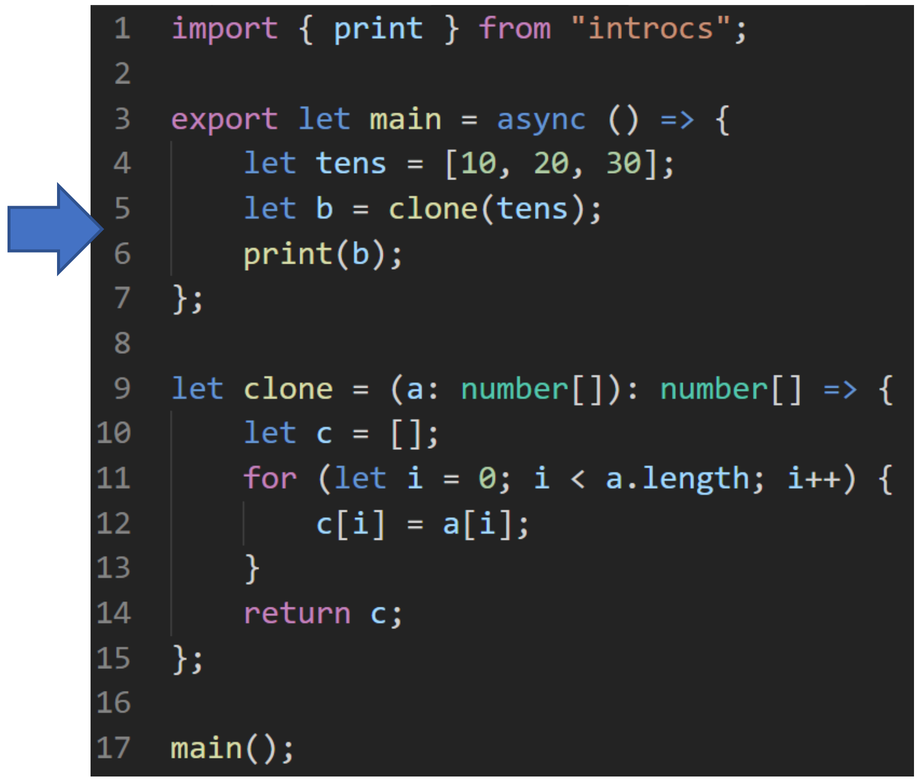
```
 1  import { print } from "introcs";
 2
 3  export let main = async () => {
 4      let tens = [10, 20, 30];
 5      let b = clone(tens);
 6      print(b);
 7  };
 8
 9  let clone = (a: number[]): number[] => {
10      let c = [];
11      for (let i = 0; i < a.length; i++) {
12          c[i] = a[i];
13      }
14      return c;
15  };
16
17  main();
```



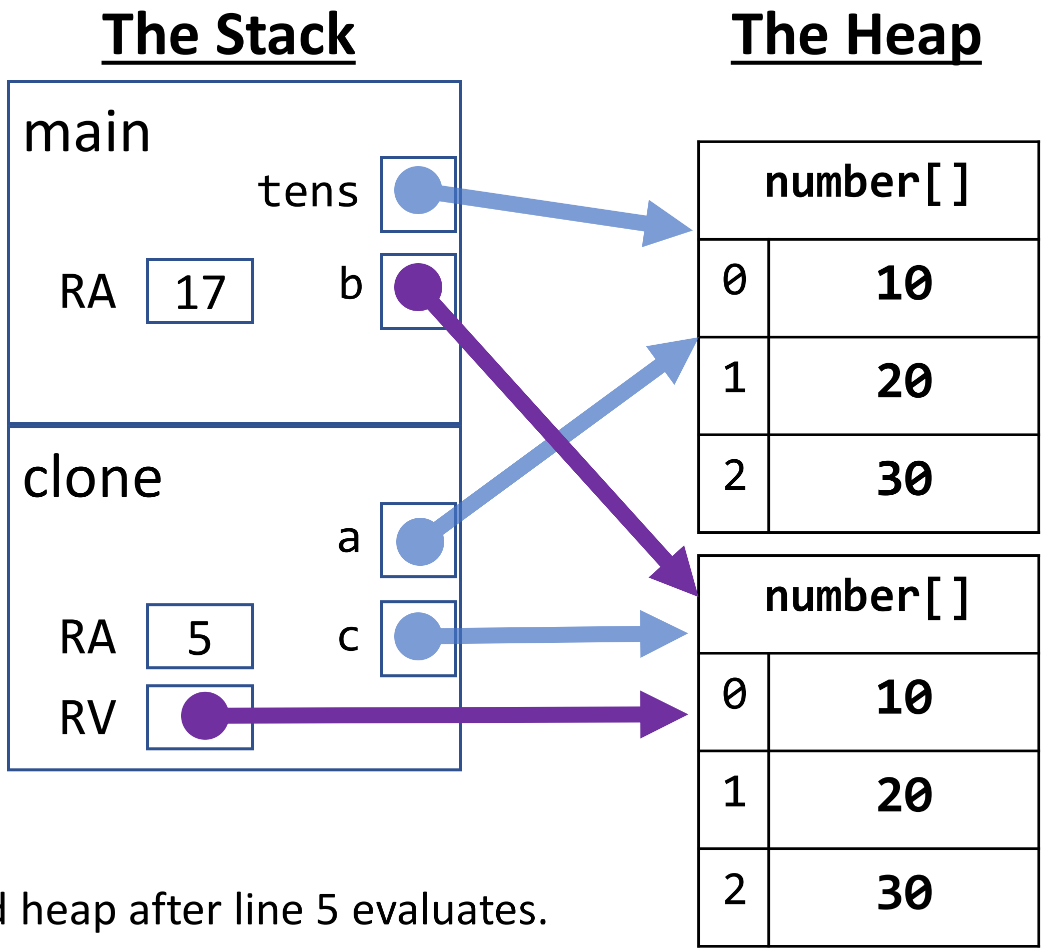
The state of the stack and heap after line 10 evaluates.

Returning References

- Functions returning an array return a pointer arrow to the array referenced.



```
 1  import { print } from "introcs";
 2
 3  export let main = async () => {
 4      let tens = [10, 20, 30];
 5      let b = clone(tens);
 6      print(b);
 7  };
 8
 9  let clone = (a: number[]): number[] => {
10      let c = [];
11      for (let i = 0; i < a.length; i++) {
12          c[i] = a[i];
13      }
14      return c;
15  };
16
17  main();
```



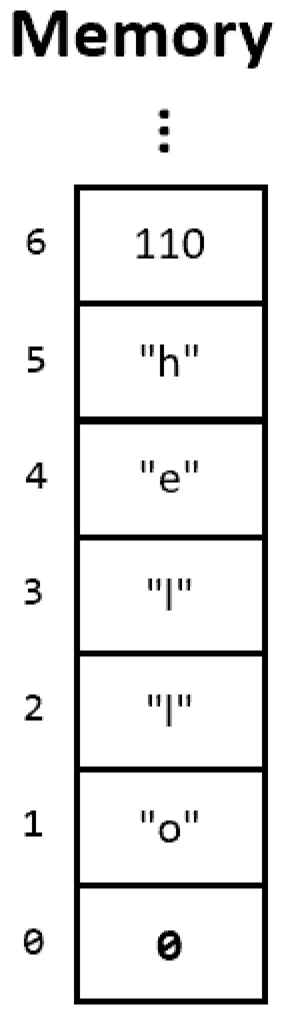
The state of the stack and heap after line 5 evaluates.

6. What is printed?

```
let s = "abc";
print(s[1]);
print(s.length);
```

Strings are Arrays of Characters

- In the first video of the semester, your computer's memory was introduced with a diagram to the left.
- Notice cells 1-5 store individual characters... *not strings.*
- A string is an array of single characters underneath the hood.
  - We haven't needed to worry over this detail thanks to ***data abstraction!***
- We can "poke through" the abstraction!
  - Access individual characters with **`stringName[index]`**
  - Access the length of a string with **`stringName.length`**



string vs string[]

```
let a = ["1", "2", "3"];
let s = "123";
```

There is a very important difference between a string value and an array of single character strings:

A string's elements cannot be changed. An array's can be.

- You cannot reassign a character like `s[1] = "9";`
- You cannot append new characters to the end of it.
- More precisely, a string is an *immutable values* because its contents cannot change.
- When you concatenate two strings you are producing a third, new string value.

Operation Assignment Operators

- Consider the following assignment statements:

```
i = i + 10;
s = s + "!";
```

- Increasing a variable, concatenating to a variable, and so on, are so common that there are built-in shorthand operators:

Operator	Syntax	Example	Equivalent To
Addition Assignment	`+=`	`i += 10;`	`i = i + 10;`
Subtraction Assignment	`-=`	`i -= 10;`	`i = i - 10;`
Multiplication Assignment	`*=`	`i *= 10;`	`i = i * 10;`
Division Assignment	`/=`	`i /= 10;`	`i = i / 10;`
Remainder Assignment	`%=`	`i %= 10;`	`i = i % 10;`
Concatenation Assignment	`+=`	`s += "!!!";`	`s = s + "!!!";`

Challenge Question #7 - pollev.com/compunc

- What is the result of calling: **`lol(3)`**

```
let lol = (force: number): string => {
    let s = "";
    for (let i = 1; i < force; i++) {
        s += "h";
        for (let h = 0; h < i; h += 1) {
            s += "e";
        }
    }
    return s;
};
```

Notes on Nested Loops

- **General Rule:** When the closing curly brace of a loop is encountered, the loop jumps back to the start of **its matching condition**.

- An inner loop will jump back up to the inner loop's condition and an outer loop will jump back up to the outer loop's condition.

- Thus, an inner loop must complete all of its **iterations** for *each* single iteration of an outer loop.