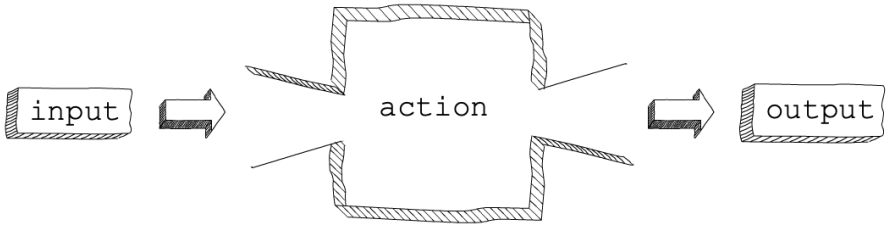
Chapter 11

More on Functions

Previously on LC101

FUNCTION ARE

- REUSABLE
- CALLABLE
- VERSATILE



They receive input and take an action to produce an output!

FUNCTION YOU ALREADY KNOW:

- CONSOLE.LOG
- TYPE CONVERSION
  - Number
  - String
  - Boolean
- String and Array Methods

FUNCTION SYNTAX

```
1        2                3
function myFunction (parameter1,.....,parameterN) {
    //function body4
}
```

1. function is a keyword that instructs JS to create a new function using the definition that follows.
2. function name is determined by the programmer
3. function parameters are variables that can be used only with the function itself
    a. Unlike other languages Javascript does not allow you to specify types of parameters
4. function body is everything within the curly brackets

Return Statements

To return a value from a function that we create, we must use a return statement

```
return someVal;
```

- Return statements are options
- Return statements terminate function execution
  - Safe bet most of the time is to put your return statement as the last part of your function
- Functions can have more than 1 return statement. But only one of them will return a value.
  - Ex:

```
1  function isEven(n) {
2    if (n % 2 === 0) {
3      return true;
4    } else {
5      return false;
6    }
7  }
8
9  console.log(isEven(4));
10 console.log(isEven(7));
```

Console Output

```
true
false
```

NAMING FUNCTIONS

1. Use Camel Case
2. Use Verb/Noun Pairs when Applicable
3. Use Descriptive Names

EXAMPLES

1. camelCase
   a. astronautCount
   b. fuelTempCelsius
2. Use Verb/Noun Pairs
   a. fillUpGasTank
   b. eatFood
3. Use Descriptive Names
   a. convertKilometersToMiles
   b. convertCelsiusToFahrenheit

FUNCTION COMPOSITION

USING FUNCTIONS TO BUILD OTHER FUNCTIONS

- Functions should do exactly one thing
  - Easier to debug
  - Easier to read
  - More reusable

Chapter Breakdown

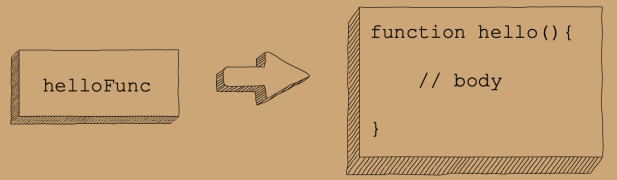
- 11.1 Functions as Values
- 11.2 Anonymous Functions
- 11.3 Passing Functions as Arguments
- 11.4 Receiving Function Arguments
- 11.5 Why Use Anonymous Functions
- 11.6 Recursion
- 11.7 Recursion Walkthrough: The Base
- 11.8 Making a Function Call Itself
- 11.9 Recursion Wrap-up

11.1 Functions as Values

Function Are Data

- Can be assigned as variables
- Variables storing a function can be thought of as an alias for that function.

- Functions may be assigned to variables.
- Functions may be used in expressions, such as comparisons.
- Functions may be converted to other data types.
- Functions may be printed using `console.log`.
- Functions may be passed as arguments to other functions.
- Functions may be returned from other functions.



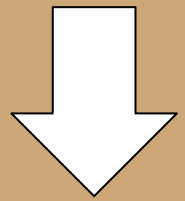
11.2 Anonymous functions

Anonymous Functions

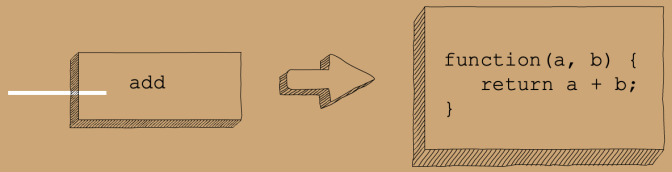
An anonymous function is a function that is not stored, but is associated with a variable. Anonymous functions can accept inputs and return outputs, just as standard functions do.

Most of the time they are stored in a variable

```
function reverse(str) {
  let lettersArray = str.split('');
  let reversedLettersArray = lettersArray.reverse();
  return reversedLettersArray.join('');
}
```



```
let reverse = function(str){
  let lettersArray = str.split('');
  let reversedLettersArray = lettersArray.reverse();
  return reversedLettersArray.join('');
```



11.3 Passing Functions as Arguments

Passing functions as Arguments

Functions are data so they can be passed around just like other values!

```
setTimeout(func, delayInMilliseconds);
```

```
1  function printMessage() {
2      console.log("The future is now!");
3  }
4
5  setTimeout(printMessage, 5000);
```

11.4 Receiving Function Arguments

Receiving Function Arguments

You can write functions that can take other functions as an argument!

```
const input = require('readline-sync');

function getValidInput(prompt, isValid) {

   // Prompt the user, using the prompt string that was passed
   let userInput = input.question(prompt);

   // Call the boolean function isValid to check the input
   while (!isValid(userInput)) {
      console.log("Invalid input. Try again.");
      userInput = input.question(prompt);
   }

   return userInput;
}

// A boolean function for validating input
let isEven = function(n) {
   return Number(n) % 2 === 0;
};

console.log(getValidInput('Enter an even number:', isEven));
```

Sample Output

```
Enter an even number: 3
Invalid input. Try again.
Enter an even number: 5
Invalid input. Try again.
Enter an even number: 4
4
```

11.5 Why Use Anonymous Functions

Why Use Anonymous Functions?

- Can be Single Use
- Are Ubiquitous* in Javascript

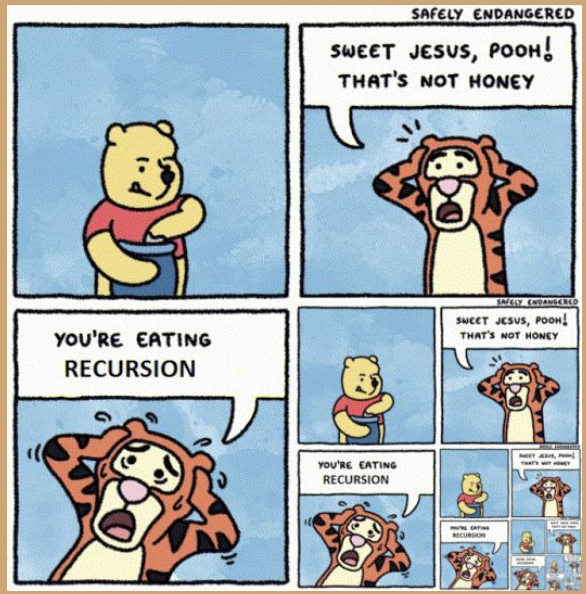
*present, appearing, or found everywhere.

Defining Functions anonymously, and directly within a function call, can reduce the number of names you need to create!

11.6 Recursion

Recursion

Allows for a function to be able to call itself instead of using a loop to solve a problem.



11.7 Recursion Walkthrough: The Base Case

Recursion Base Case

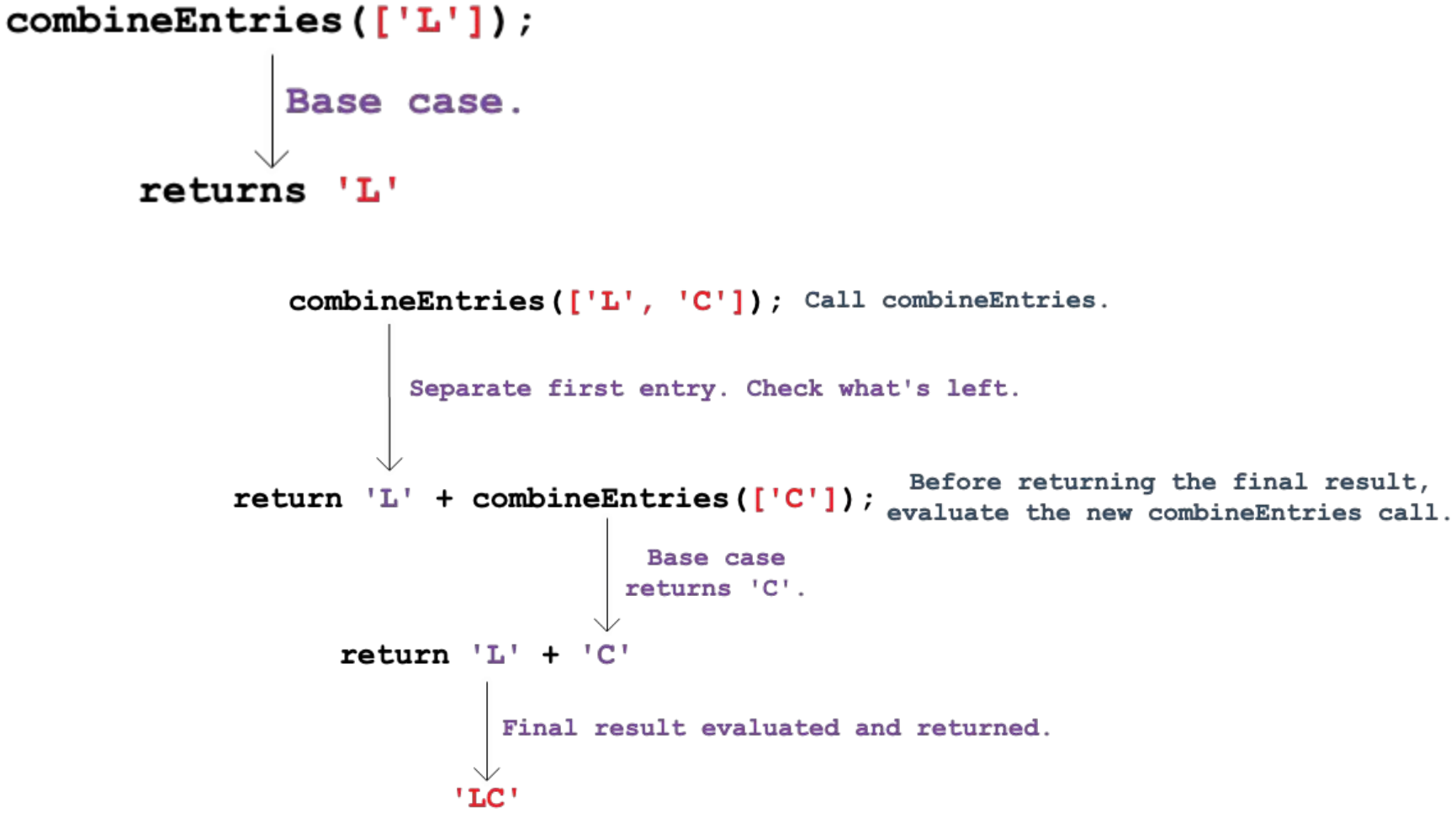
Since we don't know how many times the function will be called we need to make sure code stops at a proper time. The base case is the condition that ends the process

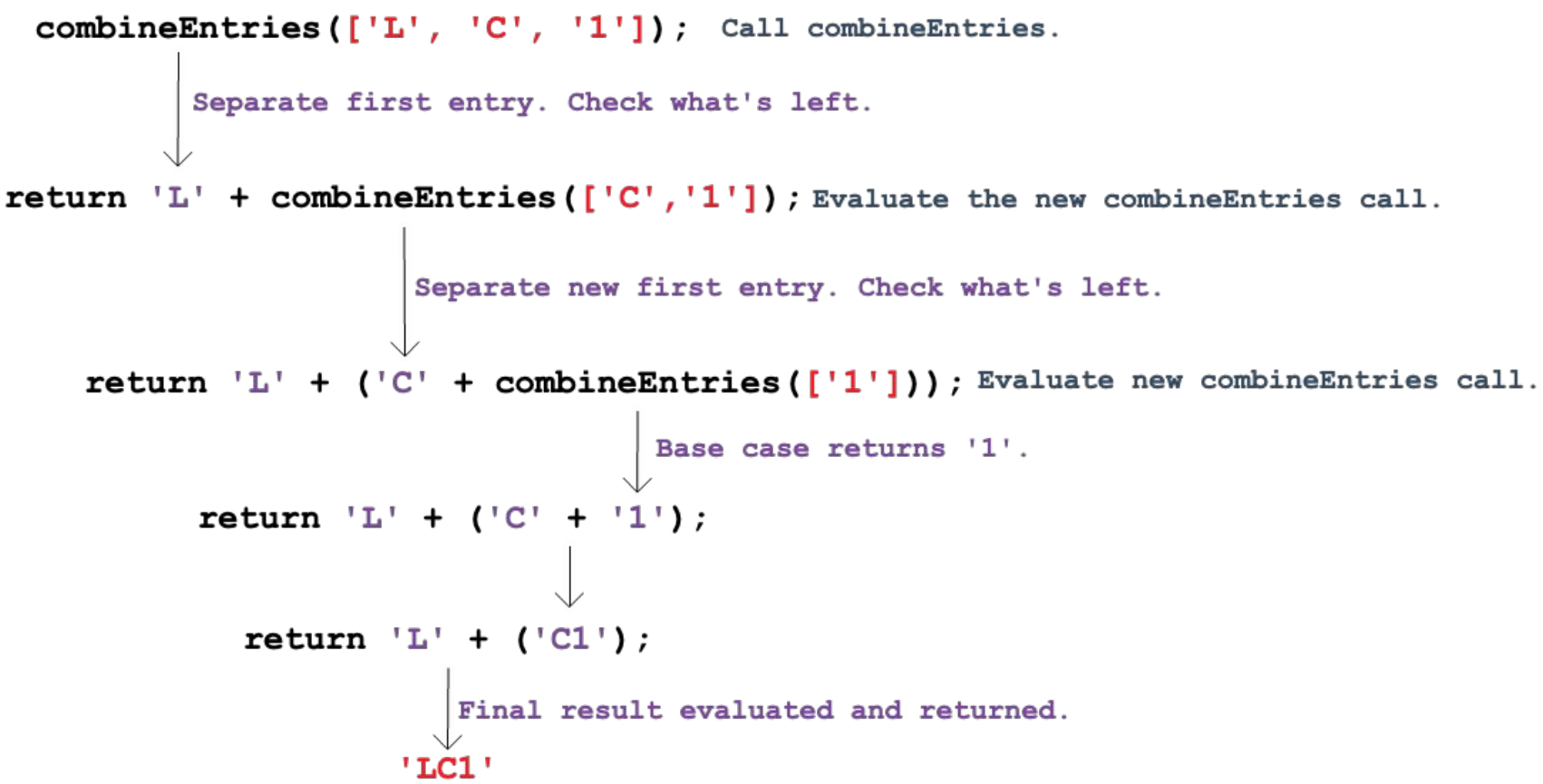
```
function combineEntries(arrayName){
   if (baseCase is true){
      //solve last small step
      //end recursion
   } else {
      //call combineEntries again
   }
}
```

```
function combineEntries(arrayName){
   if (arrayName.length <= 1){
      return arrayName[0];
   } else {
      //call combineEntries again
   }
}
```

This checks if the array only has 1 value and stops the recursion

11.8 Making a function call itself


combineEntries(['L']);
Base case.
returns 'L'
combineEntries(['L', 'C']); Call combineEntries.
Separate first entry. Check what's left.
return 'L' + combineEntries(['C']);
Before returning the final result, evaluate the new combineEntries call.
Base case returns 'C'.
return 'L' + 'C'
Final result evaluated and returned.
'LC'


combineEntries(['L', 'C', '1']); Call combineEntries.
Separate first entry. Check what's left.
return 'L' + combineEntries(['C','1']); Evaluate the new combineEntries call.
Separate new first entry. Check what's left.
return 'L' + ('C' + combineEntries(['1'])); Evaluate new combineEntries call.
Base case returns '1'.
return 'L' + ('C' + '1');
return 'L' + ('C1');
Final result evaluated and returned.
'LC1'

Function Calling Itself

If not the base case
returns value + function()

```
function combineEntries(arrayName){
   if (arrayName.length <= 1){
      return arrayName[0];
   } else {
      return arrayName[0]+combineEntries(arrayName.slice(1));
   }
}
```

For combineEntries(['L', 'C', '1', '0', '1']);, the sequence would be:

Step	Description
1.	First call: Combine 'L' with combineEntries(['C', '1', '0', '1']).
2.	Second call: Combine 'C', with combineEntries(['1', '0', '1']).
3.	Third call: Combine '1', with combineEntries(['0', '1']).
4.	Fourth call: Combine '0', with combineEntries(['1']).
5.	Fifth call: Base case returns '1'.

11.9 Recursion Wrap-up

Recursion in a Nutshell

Will you use it? Probably at some point.
Will it be often. No!

1. Build a single function to break a big problem into a slightly smaller version of the *exact same problem*.
2. The function repeatedly calls itself to reduce the problem into smaller and smaller pieces.
3. Eventually, the function reaches a simplest case (the *base*), which it solves.
4. Solving the base case sets up the solutions to all of the previous steps