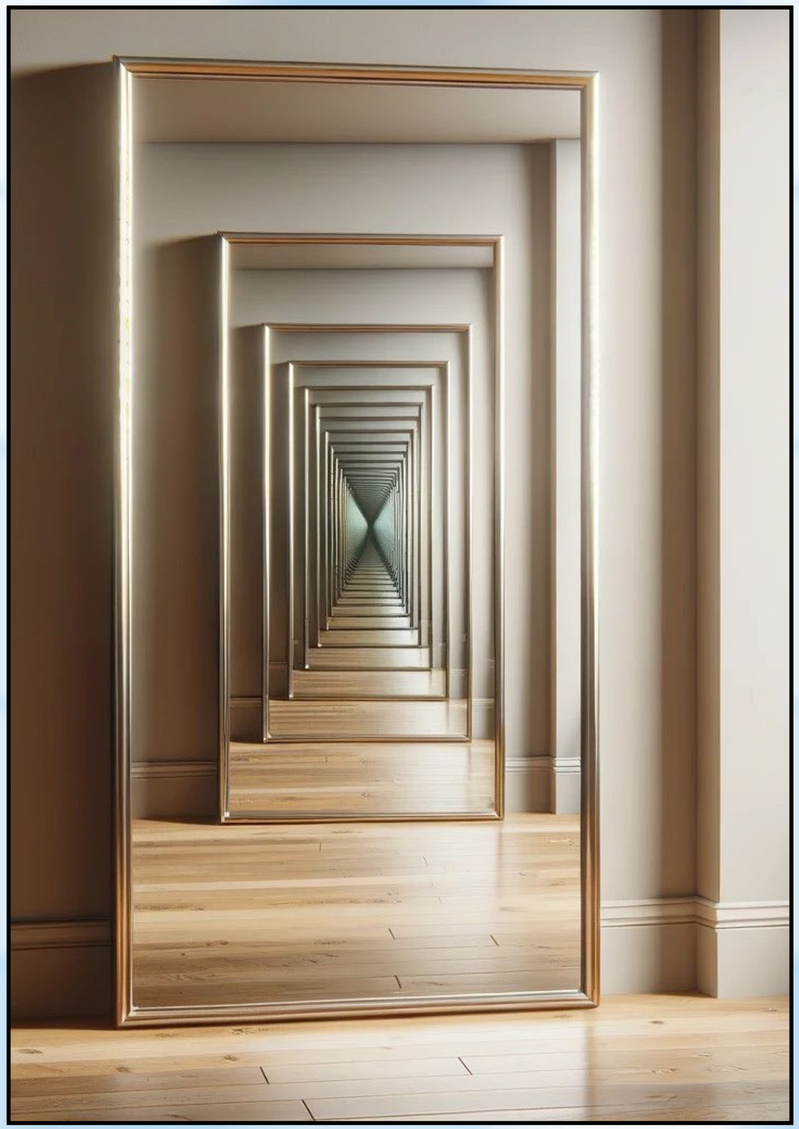
Introducing Recursion

RECURSION

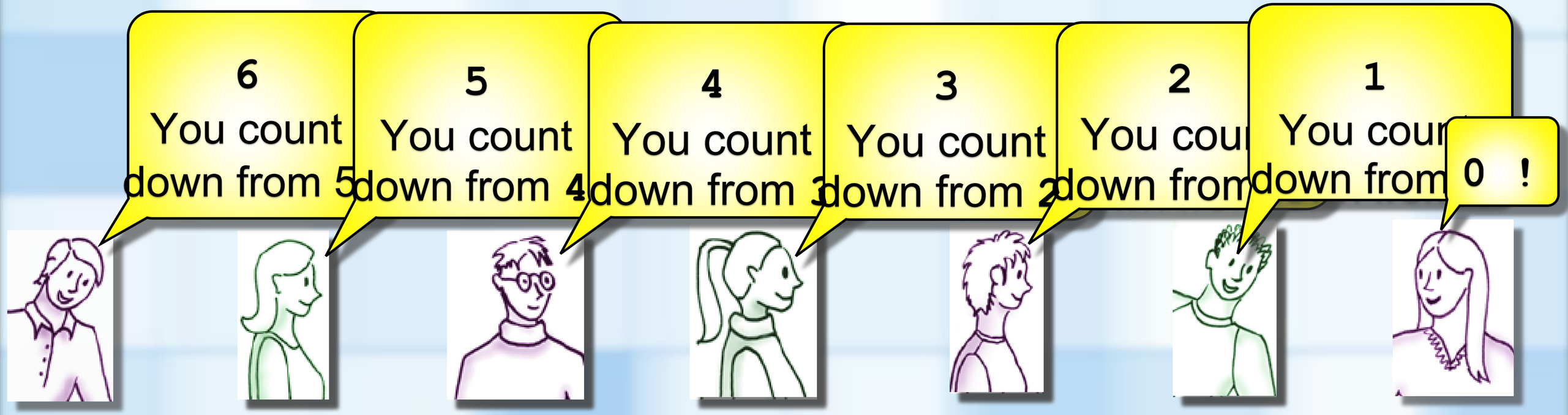
- **Powerful Problem-Solving Technique**
  - Breaks a problem into smaller, identical problems
    - Break each of these smaller problems into even smaller, identical problems
  - Eventually arrive at a stopping case
    - The Base Case
    - Problem cannot be broken down any farther



Pearson

Counting Recursively

- **Counting Down to Zero from a Number**
  - Say the number
  - Ask a friend to count down from the number minus one
  - When someone says zero, stop.



Characteristics of Recursion

- **A Recursive Function**
  - Is a function that ***calls itself*** and asks:
    - What part of the solution can you **contribute** directly?
    - What **smaller but identical problem** has a solution that, when taken with your contribution, provides the solution to the original problem?
    - When does the **process end**?
      - *What smaller but identical problem has a known solution (the base case)?*
    - How will you know when you have reached the base case?

```
void countDown(int number)
{
   if (number == 0)
      std::cout << number << std::endl;
   else
   {
      std::cout << number << std::endl;
      countDown(number - 1);
   } // end if
} // end countDown
```

Mechanics of Recursion

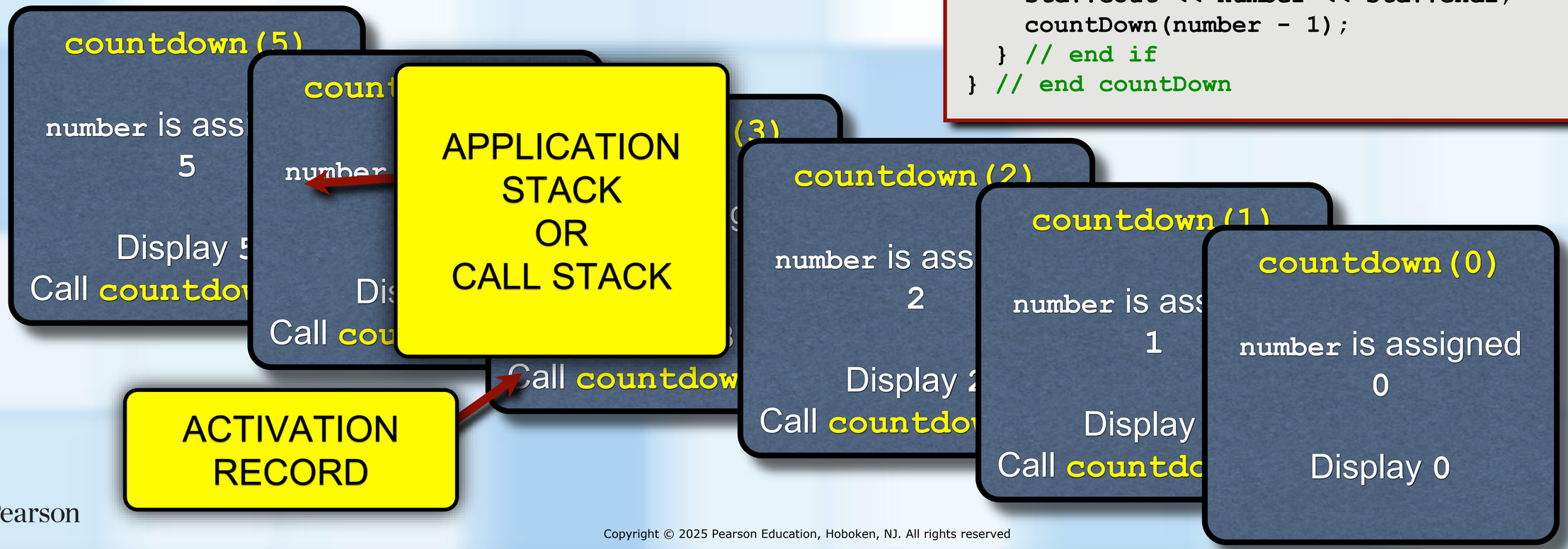
- **Recursive functions . . .**
  - Must be given an input value to start
    - Either as an argument or input from the user
  - Function logic must involve this value and use it to obtain different (smaller) cases
    - Often an `if` or `switch` statement is used
  - One of these cases must not require recursion to solve. These are the base cases.
  - One or more of these cases must include a recursive invocation of the function.
  - Each of these cases must be smaller versions of the problem that converge on the base case.

```
void countDown(int number)
{
  if (number == 0)
      std::cout << number << std::endl;

  else
  {
    std::cout << number << std::endl;

    countDown(number - 1);

  } // end if

} // end countDown
```

TRACING RECURSION

- **Each recursive call assigns new values to the parameters and local variables.**

```
void countDown(int number)
{
  if (number == 0)
    std::cout << number << std::endl;
  else
  {
    std::cout << number << std::endl;
    countDown(number - 1);
  } // end if
} // end countDown
```



Recursion and Iteration

- **Recursive Solutions**
  - Usually involve branching
    - **`if`** and **`switch`** statements
    - Deciding whether or not it has the base case
  - May involve a loop in addition to branching
- **Iterative Solutions**
  - Involve loops
    - **`while`**, **`for`** and **`do-while`** statements

Recursive Solution

```
void countDown(int number)
{
  if (number == 0)
    std::cout << number << std::endl;
  else
  {
    std::cout << number << std::endl;
    countDown(number - 1);
  } // end if
} // end countDown
```

Iterative Solution

```
void countDown(int number)
{
  while (number >= 0)
    std::cout << number << std::endl;
    number = number - 1);
  } // end while
} // end countDown
```

Returning a Value from Recursion

- **Recursive Functions that Return a Value**
  - Compute the product of the first ***n*** positive integers
- **Designing our Function**
  - Rewrite **`productOf`** in terms of itself
    - Make the problem smaller each time
  - What is the base case?
    - A problem that we know the answer
  - Function should have a parameter
    - The number to be processed

```
int productOf(int n)
{
  int product = 0;

  if (n == 1)
    product = 1;

  else
    product = n * productOf(n - 1);

  return product;

} // end productOf
```

`productOf(n) = n * (n-1) * ... * 3 * 2 * 1`

for any integer `n > 0`

`productOf(n) = n * productOf(n-1)`

`productOf(1) = 1`

SIMPLIFYING RECURSION

- **Simplifying the Functions**
  - Intermediate products do not need to be stored
  - Can use multiple **`return`** statements
    - Only one **`return`** will be executed

```
int productOf(int n)
{
  int product = 0;

  if (n == 1)
    product = 1;

  else
    product = n * productOf(n - 1);

  return product;

} // end productOf
```

```
int productOf(int n)
{
  if (n == 1)
    return 1;
  else
    return n * productOf(n - 1);

} // end productOf
```

Pearson

SIMPLIFYING RECURSION

- **Explicit Base Case**
  - A distinct option exists to handle the base case
- **Implicit Base Case**
  - A clear base case is not listed
  - Base Case is a "Do Nothing" option

```
public static void countDown(int number)
{
  if (number >= 0)
  {
    std::cout << number << std::endl;
    countDown(number - 1);
  }
  else
  {
     // Do Nothing
  }
} // end countDown
```

EXPLICIT BASE CASE

```
void countDown(int number)
{
  if (number == 0)
    std::cout << number << std::endl;
  else
  {
    std::cout << number << std::endl;
    countDown(number - 1);
  } // end if
} // end countDown
```

IMPLICIT BASE CASE

```
void countDown(int number)
{
  if (number >= 0)
  {
    std::cout << number << std::endl;
    countDown(number - 1);
  } // end if
} // end countDown
```

Pearson

Recursively Processing Arrays

Recursive Processing - Arrays

- **Recursively displaying elements stored in an array**
  - Start with first element

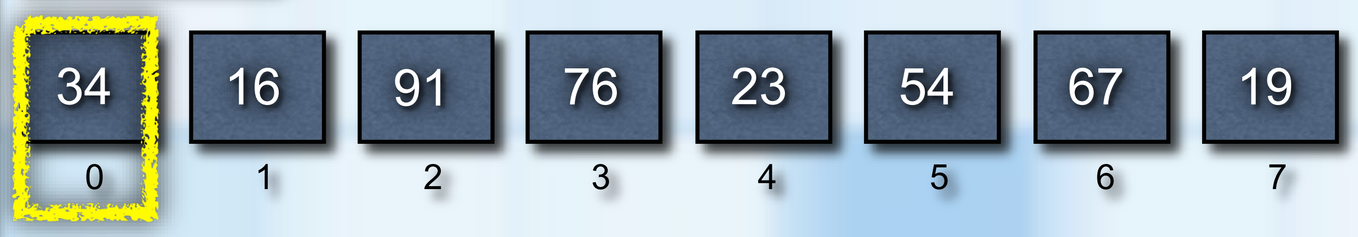
```
// Start with myArray[first]
void displayArray(int myArray[], int first, int last)
{
  std::cout << myArray[first] << " ";
  if (first < last)
    displayArray(myArray, first + 1, last);
} // end displayArray
```

This function has an implicit base case.

first
0

34	16	91	76	23	54	67	19

last
7



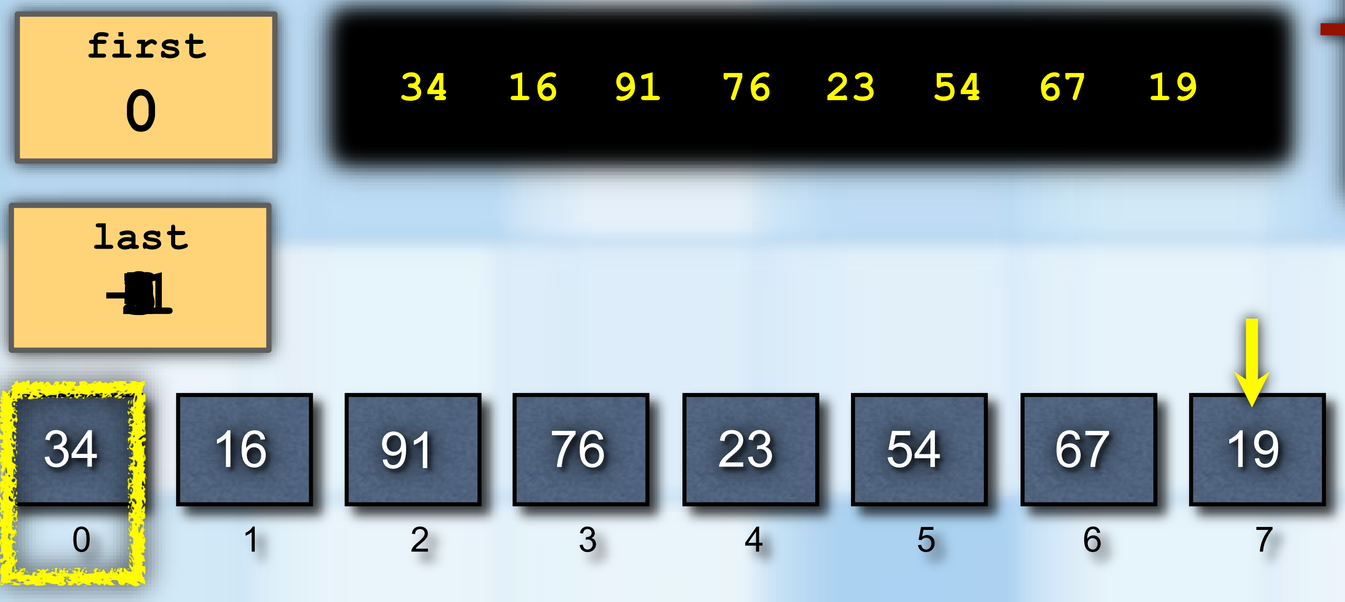
Pearson

Recursive Processing - Arrays

- **Recursively displaying elements stored in an array**
  - Start with first element
  - Start with last element

```
// Start with myArray[first]
void displayArray(int myArray[], int first, int last)
{
  std::cout << myArray[first] << " ";
  if (first < last)
    displayArray(myArray, first + 1, last);
} // end displayArray
```

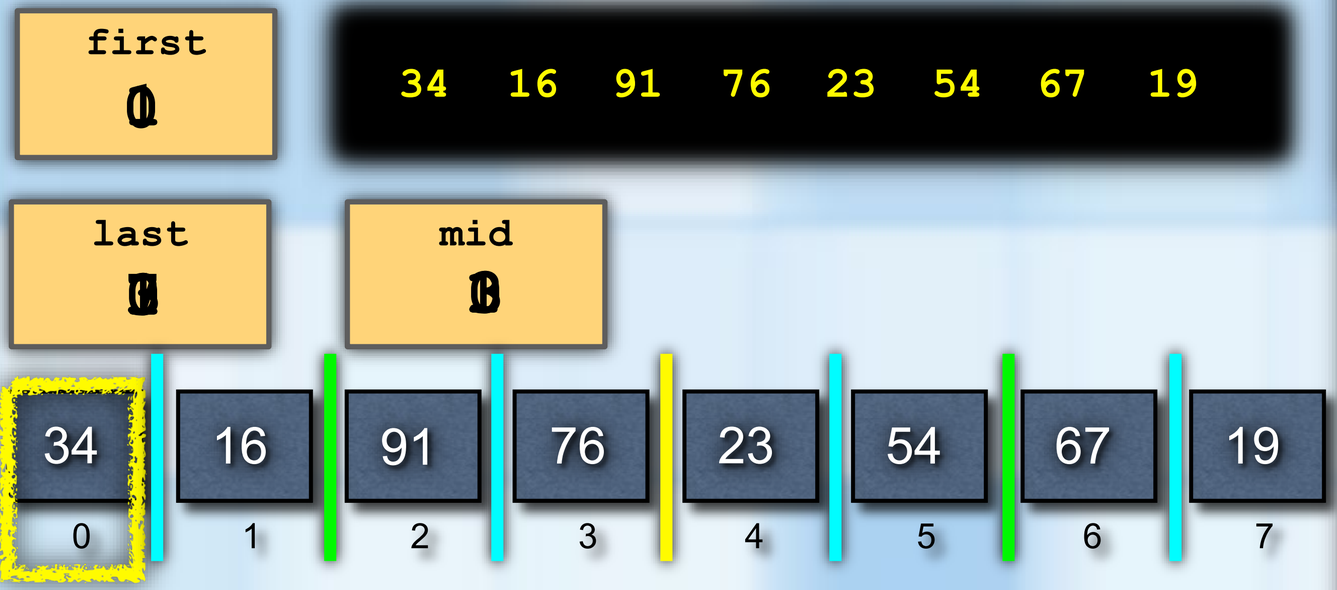
```
// Start with myArray[last]
void displayArray(int myArray[], int first, int last)
{
  if (first <= last)
  {
    std::cout << myArray[last] << " ";
    displayArray(myArray, first, last - 1);
  } // end if
} // end displayArray
```



This function has an implicit base case.

Recursive Processing - Arrays

- **Recursively displaying elements stored in an array**
  - Start with first element
  - Start with last element
  - Divide the array in half




```
// Start with myArray[first]
void displayArray(int myArray[], int first, int last)
{
  std::cout << myArray[first
  if (first < last)
    displayArray(myArra, fi
} // end displayArray
```

```
// Start with myArray[last]
void displayArray(int myArra
{
  if (first < last)
  {
    displayArray(myArray, fi
    std::cout << myArray[last] <<
  } // end if
} // end displayArray
```

```
// Dividing the array in half
void displayArray(int myArray
{
  if (first == last)
    std::cout << myArray[first
  else
  {
    int mid = first + (last -
    displayArray(myArray, firs
    displayArray(myArray, mid
  } // end if
} // end displayArray
```

These two functions each have an implicit base case so we test for when we should perform the recursion.

This function has an explicit base case to test for when we should not perform recursion.

Pearson