Part II Arrays as arguments and return types:

This discussion is related to class code Set12B. You should have that code open as you read through this.

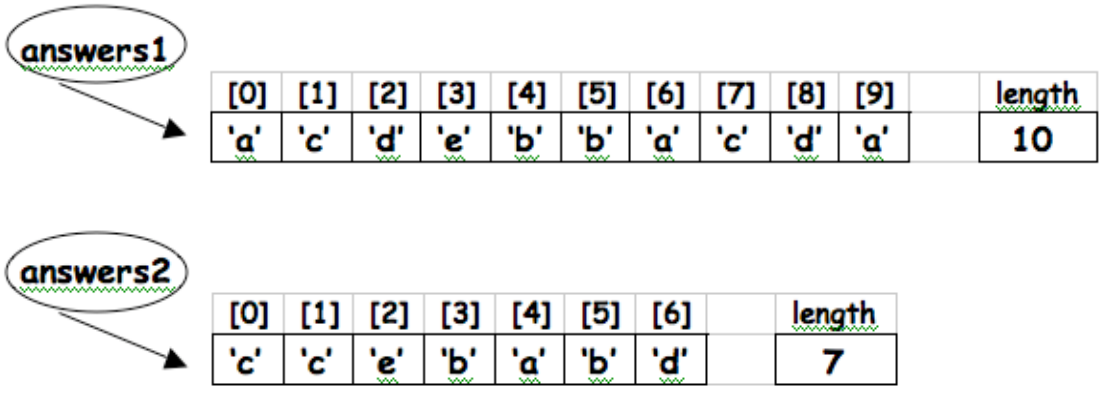
This set of notes focuses on using arrays as arguments and as return types. The last part is code for a type of search called a filter.

You should read over the board notes bn14PartA and revisit the first part of these notes if you are still foggy on how to declare and initialize an array and how to use a **for** *loop to traverse the array and what we can do when we visit each element of the array. These notes assume you know what those terms mean.*

We begin with two arrays of char:

```
char [ ] answers1 = { 'a', 'c', 'd', 'e', 'b', 'b', 'a', 'c', 'd', 'a' };
char [ ] answers2 = { 'c', 'c', 'e', 'b', 'a', 'b', 'd' };
```

Here is how we would draw Processing's view of these arrays:



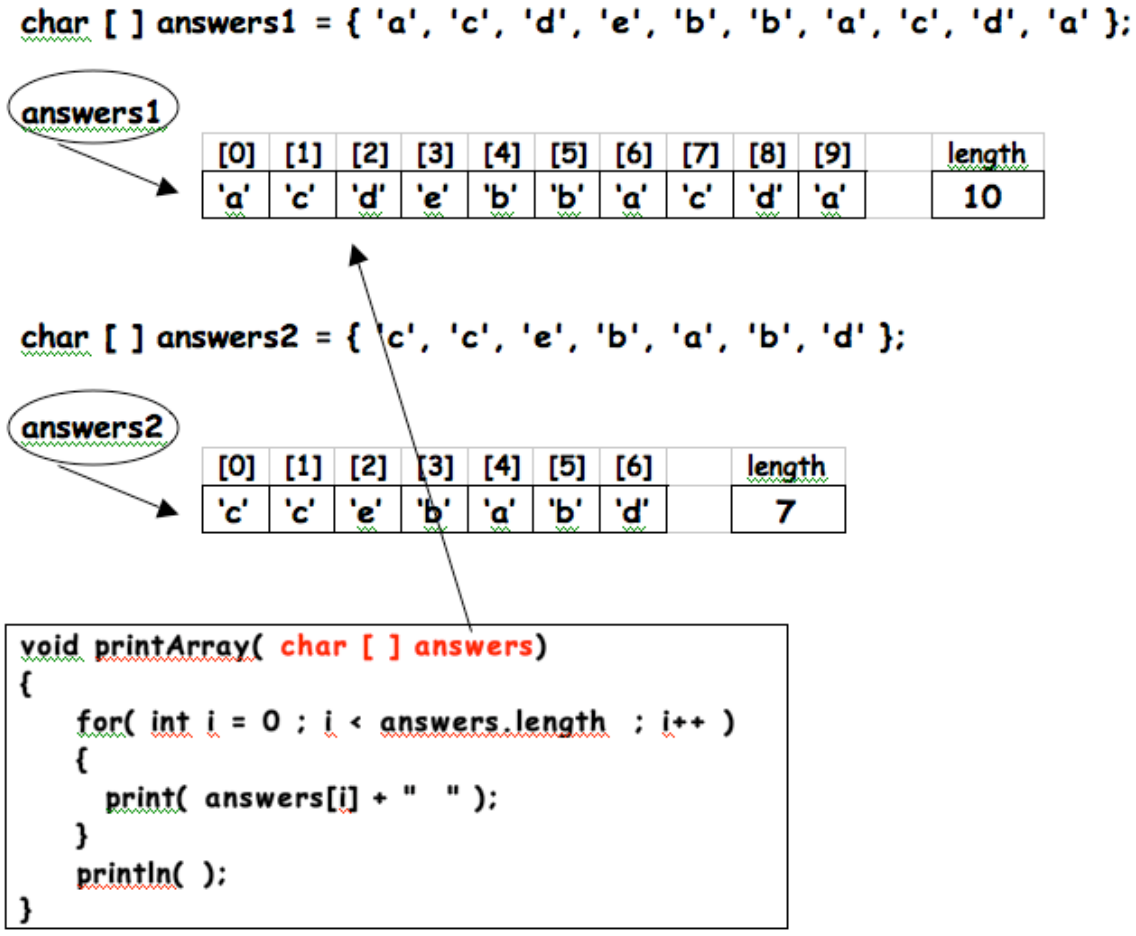
The first two function calls print each array:

```
printArray( answers1 );
printArray( answers2 );
```

Processing does not make copies of the data for the arguments in the definition as it would for primitive variables. What Processing does is make copies the

arrow that connects the reference to the array. This sounds very confusing so let's make another drawing.

Here is how the program looks for this function call:
printArray(answers1);

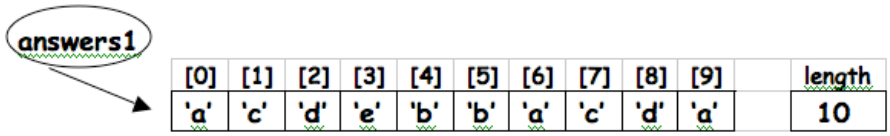


When **printArray** is called with **answers1** as the argument, the argument **answers** in the definition is assigned to reference the same array that **answers1** references. Using this diagram, when the function **printArray ( )** needs to know what the **length** of the array is, it "follows" the reference arrow to the array and looks up the **length** - which is 10. When the function is visiting an element and needs to know the value of an element, it does the same thing.

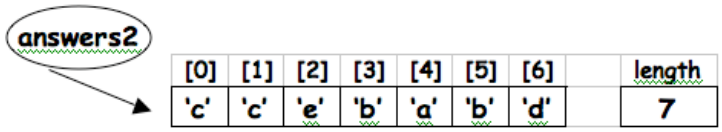
A similar picture forms for the second function call: Here is how the program looks for this function call:

printArray(answers2);

```
char [ ] answers1 = { 'a', 'c', 'd', 'e', 'b', 'b', 'a', 'c', 'd', 'a' };
```




```
char [ ] answers2 = { 'c', 'c', 'e', 'b', 'a', 'b', 'd' };
```




```
void printArray( char [ ] answers)
{
    for( int i = 0 ; i < answers.length  ; i++ )
    {
      print( answers[i] + "  " );
    }
    println( );
}
```

For the second call where the argument is **answers2**, *Processing assigns the argument,* **answers** *in the definition to reference the same array as* **answers2**. *When it needs to know the* **length** *of the array, it follows the arrow to the array and finds the* **length** *of this array to be 7.*

Some differences between arrays and primitive variables:

- *Arrays can have multiple values - primitive variables can have only one value.*
- *Arrays can have multiple references to them - primitive variables can have only one name.*

If you are not sure what the **for** *loop is doing or how it is working in this code, you must refer back to the previous set of class code or refer to Shiffman. The*

execution of the **for** loop with the array is explained and traced in detail.

Do not go on with this set of notes if you do not understand how the loop and the array work together. You will be wasting your time.

Next we have this line of code:

```
char [ ] answersAll = concat( answers1, answers2 );
```

The left side of the assignment operator is building a new array reference. There is no array - just a reference. The actual array is built and returned by the function **concat( )**. The **concat( )** function is part of the Processing API. The **concat( )** function builds a new array that contains the elements of the two arrays in the argument list. The two arrays in the argumentr list (**answers1, answers2**) are not modified or destroyed. Their values are copied into the elements of the new array. A reference to this new array is returned to the right side of the assignment operator where it is assigned to the array reference, **answersAll**.

There are a number of functions in the Processing API that return references to new arrays. Some of these might be useful to you in the last half of the semester. You should explore these to see what they do and how they work.

The next line of code is :

```
char [ ] answersOdd = buildAnswersOdd( answersAll );
```

Here again, the left side of the assignment operator builds a new array and returns a reference to it to be assigned to **answersOdd**. Unlike the previous line, there is no definition of **buildAnswersOdd( )** in the Processing API. We have to define it.

Here is the definition of **buildAnswersOdd( )**

```
char [ ]  buildAnswersOdd( char [ ] answers )
{
    char [ ] temp = { };
    for( int i = 1 ;i < answers.length  ;  i = i + 2 )
    {
       {
         temp = append( temp, answers[i] );
       }
    }
    return temp;
};
```

Here is a look at the parts of this definition:
`char [ ]` *This is the return type. OK - why is this function returning an array of char. The answer is not a guess or the result of some form of mystical reasoning. The answer is in the line of code where the function is called:*
`char [ ] answersOdd = buildAnswersOdd( answersAll );`

Read this from right to left. It reads as:
*"**buildAnswersOdd( )** will return something to be assigned to **answersOdd**."*
The next question is to ask is, "what is **answersOdd***?"*
We continue to read from right to left:
`char [ ] answersOdd`
*This reads as, "**answersOdd** is an array of **char**."*

This tells us that the array **buildAnswersOdd( )** *must return an array of* **char**.

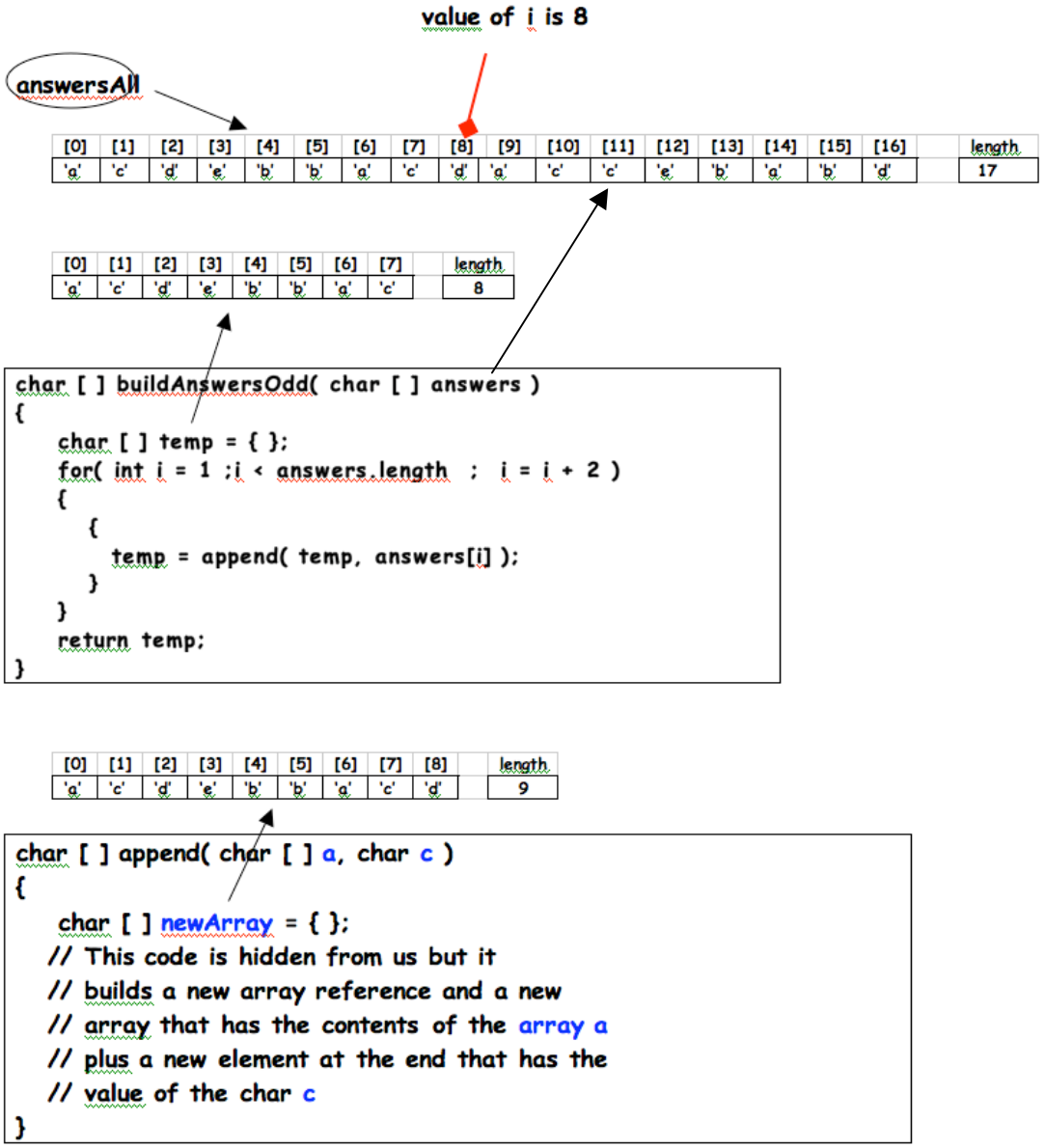
`char [ ] temp = { };` *If we are going to return an array of char, we must first build one. The reference* **temp** *is a local variable. Processing does not initialize local variables so we have to. This line of code initializes the array reference* **temp** *to an array of* **char** *that is empty. Its length at this point in the execution is* **zero**.

`for( int i = 1 ;i < answers.length ; i = i + 2 )` The function must build a new array containing the elements of the odd indexes in the array referenced by the argument. We can do this in different ways. Two were done in class. In this example the loop starts at element `[1]` instead of element `[0]`. It traverses and visits every other element or the odd elements. Thi happens because of the way the loop increments the variable `i`. Instead of `i++` which is the "usual" pattern, `i` is incremented by 2 with this code: `i = i + 2`

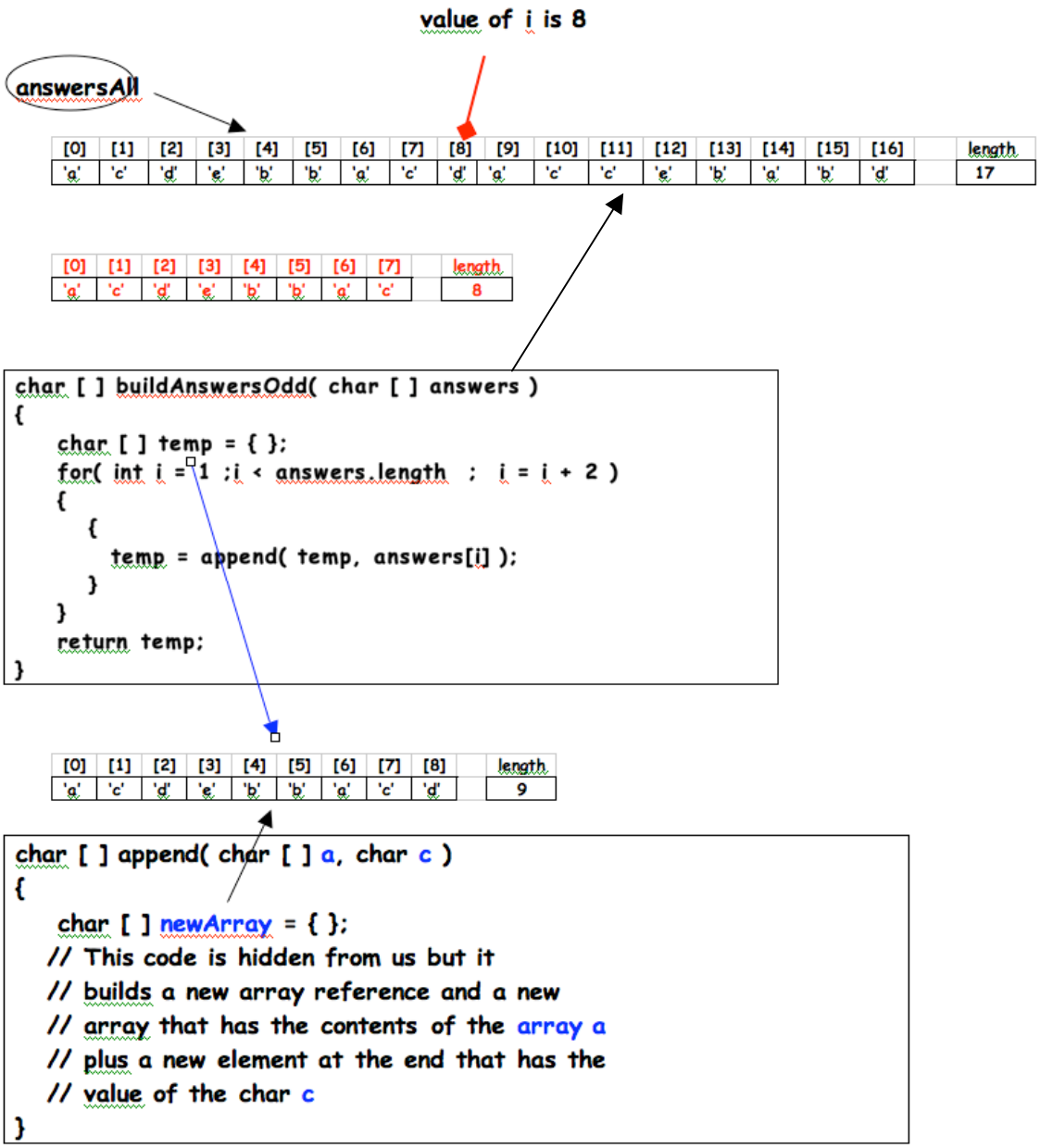
`temp = append( temp, answers[i] );` The `append( )` function is in the Processing API. `append( )` returns a reference to a new array that contains all of the elements in the array `temp` plus one new element at the end which has the value of the element in `answers[i]`. (Read this over several times). The reference to the new array is assigned to temp. `temp`'s old reference to the old array is lost forever. Since there are no references to the old array, it too, is lost forever.

The next page shows the structure of the arrays when the value of the loop variable is `8` BUT before the reference is returned:

Remember, this is before the **append( )** *function returns the reference to the new array:*



The next page shows the structure of the arrays after the **append( )** *function has returned the reference to the new array:*



The array reference **temp** *is no longer pointing to the 8 element array now shown in red. It is pointing to the new array that contains all of the elements in the old array plus one more - the value of element* **[8]** *in the array referenced by* **answers**. *The old array now shown in red that* **temp** *used to reference now has no*

references. We can no longer access any of the values stored in it. It is lost to us forever.[1]

This is somewhat complex. It is explained here to show you that a reasonable set of events do occur when we work with arrays. The more of this you understand, the easier your work with arrays will be in the coming weeks.

The last set of function calls:

```
println( "The number of a answers in answers1 is "    + countLetter( answers1,    'a' ) );
println( "The number of b answers in answers2 is "    + countLetter( answers2,    'b' ) );
println( "The number of c answers in answersOdd is " + countLetter( answersOdd, 'c' ) );
```

call a function that demonstrates one form of array search called a filter. An array search occurs when we traverse an array looking for something specific in the array. There are two general types of array searches:

1. *We are looking for one specific value or the first occurrence of a specific value in the array. It may or may not be there. If, and when we find it, we can stop looking. This search is similar to you looking for your keys or your id card in your room when you have misplaced it. You stop looking when you find it*
2. *We are looking for all occurrences of a specific value in an array. We must traverse the entire array and check every element. We cannot stop when we find the first value. This is similar to your searching for dirty laundry before heading down to the laundry room. You do not stop when you find the first pair of dirty socks. You have to look everywhere in your room.*

The second search is often called a filter. We are filtering the array picking out certain values.

---

[1] In the "old" days of programming, this was called a memory leak. If this occurred too often, the program could crash because it ran out of available memory. Processing runs a program called the garbage collector (yes, that is what it is called) that goes around collecting up unreferenced arrays and returns the memory to the operating system so our program will not run out of memory.

The function **countLetter( )** *must traverse the array and count the number of times a specific character is in the array. It is filtering the characters in the array looking or a specific character.*

Here is the code:

```
int countLetter( char [ ] answers, char letterToCount )
{
    int letterCount = 0;
    for( int i = 0 ; i < answers.length  ; i++ )
    {
      if ( answers[i] == letterToCount )
      {
         letterCount++;
      }
    }
    return letterCount;
}
```

countLetter( ) *must return a count of a specific character. Since it is counting characters, the type of the count should be* **int**. *We should not find half of an* **'x'**.

`int letterCount = 0;` *We declare a variable to store the count and initialize it to zero.*

`for( int i = 0 ; i < answers.length  ; i++ )` *We use the* **for** *loop to traverse the entire array since the character we are counting might be in any element.*

`if ( answers[i] == letterToCount )` *We visit each element of the array and "ask" if it is the letter we are counting. To ask the question, we use the* **if** *control structure. If the* **[i]***th element is equal to the letter we are looking for, the expression evaluates to* **true**.

`letterCount++;` If the expression is `true`, we increment the variable we are using to store the count. If it is `false`, we do nothing.

`return letterCount;` Finally we return the count of the character we are looking for back to the call where it is printed on the screen.

Final Thoughts on this part:

There is a lot here - a lot. Arrays are the basis of the second exam. You need to work through this and the code of these same days. If you do not understand any part of all of this, please come to office hours held by Jim or the CAs and do this soon. Week 9 or 10 is too late.