3 Arrays

3.1 Introduction

A structured type of fundamental importance in almost every procedural programming language is the array.

> **Array**: A fixed length, ordered collection of values of the same type stored in contiguous memory locations; the collection may be ordered in several dimensions.

The values stored in an array are called **elements**. Elements are accessed by *indexing* into the array: an integer value is used to indicate the ordinal value of the element. For example, if a is an array with 20 elements, then $a[6]$ is the element of a with ordinal value 6. Indexing may start at any number, but generally it starts at 0. In the example above $a[6]$ is the seventh value in a when indexing start at 0.

Arrays are important because they allow many values to be stored in a single data structure while providing very fast access to each value. This is made possible by the fact that (a) all values in an array are the same type, and henokce require the same amount of memory to store, and that (b) elements are stored in contiguous memory locations. Accessing element $a[i]$ requires finding the location where the element is stored. This is done by computing $b+ (i \times m,)$ where m is the size of an array element, and b is the base location of array a. This computation is obviously very fast. Furthermore, access to all the elements of the array can be done by starting a counter at b and incrementing it by $m,$ thus yielding the location of each element in turn, which is also very fast.

Arrays are not abstract data types because their arrangement in the physical memory of a computer is an essential feature of their definition, and abstract data types abstract from all details of implementation on a computer. Nonetheless, we can discuss arrays in a "semi-abstract" fashion that abstracts some implementation details. The definition above abstracts away the details about how elements are stored in contiguous locations (which indeed does vary somewhat among languages). Also, arrays are typically types in procedural programming languages, so they are treated like realizations of abstract data types even though they are really not.

In this book, we will treat arrays as implementation mechanisms and not as ADTs.

3.2 Varieties of Arrays

In some languages, the size of an array must be established once and for all at program design time and cannot change during execution. Such arrays are called **static arrays**. A chunk of memory big enough to hold all the values in the array is allocated when the array is created, and thereafter elements are accessed using the fixed base location of the array. Static arrays are the fundamental array type in most older procedural languages, such as Fortran, Basic, and C, and in many newer object-oriented languages as well, such as Java.

Download free eBooks at bookboon.com

Some languages provide arrays whose sizes are established at run-time and can change during execution. These **dynamic arrays** have an initial size used as the basis for allocating a segment of memory for element storage. Thereafter the array may shrink or grow. If the array shrinks during execution, then only an initial portion of allocated memory is used. But if the array grows beyond the space allocated for it, a more complex *reallocation procedure* must occur, as follows:

1. A new segment of memory large enough to store the elements of the expanded array is allocated.
2. All elements of the original (unexpanded) array are copied into the new memory segment.
3. The memory used initially to store array values is freed and the newly allocated memory is associated with the array variable or reference.

This reallocation procedure is computationally expensive, so systems are usually designed to minimize its frequency of use. For example, when an array expands beyond its memory allocation, its memory allocation might be doubled even if space for only a single additional element is needed. The hope is that providing a lot of extra space will avoid many expensive reallocation procedures if the array expands over time.

Dynamic arrays are convenient for programmers because they can never be too small—whenever more space is needed in a dynamic array, it can simply be expanded. One drawback of dynamic arrays is that implementing language support for them is more work for the compiler or interpreter writer. A potentially more serious drawback is that the expansion procedure is expensive, so there are circumstances when using a dynamic array can be dangerous. For example, if an application must respond in real time to events in its environment, and a dynamic array must be expanded when the application is in the midst of a response, then the response may be delayed too long, causing problems.

3.3 Arrays in Ruby

Ruby arrays are dynamic arrays that expand automatically whenever a value is stored in a location beyond the current end of the array. To the programmer, it is as if arrays are unbounded and as many locations as are needed are available. Locations not assigned a value in an expanded array are initialized to `nil` by default. Ruby also has an interesting indexing mechanism for arrays. Array indices begin at 0 so, for example, `a[13]` is the value in the 14^{th} position of the array. Negative numbers are the indices of elements counting from the current end of the array, so `a[-1]` is the last element, `a[-2]` is the second to last element, and so forth. Array references that use an out-of-bound index return `nil`. These features combine to make it difficult to write an array reference that causes an indexing error. This is apparently a great convenience to the programmer, but actually it is not because it makes it so hard to find bugs: many unintended and erroneous array references are legal.

The ability to assign arbitrary values to arrays that automatically grow arbitrarily large makes Ruby arrays behave more like lists than arrays in other languages. We will discuss the List ADT later on.

Download free eBooks at bookboon.com

Another interesting feature of Ruby arrays has to do with the fact that Ruby is a pure object-oriented language. This means (in part) that every value in Ruby is an object, and hence every value in Ruby is an instance of `Object`, the super-class of all classes, or one of its sub-classes. Arrays hold `Object` values, so *any* value can be stored in *any* array! For example, an array can store some strings, some integers, some floats, and so forth. This appears to be a big advantage for programmers, but again this freedom has a price: it much harder to find bugs. For example, in Java, mistakenly assigning a string value to an array holding integers is flagged by the compiler as an error, but in Ruby, the interpreter does not complain.

Ruby arrays have many interesting and powerful methods. Besides indexing operations that go well beyond those discussed above, arrays have operations based on set operations (membership, intersection, union, and relative complement), string operations (concatenation, searching, and replacement), stack operations (push and pop), and queue operations (shift and append), as well as more traditional array-based operations (sorting, reversing, removing duplicates, and so forth). Arrays are also tightly bound up with Ruby's iteration mechanism, which will be discussed later.

Download free eBooks at bookboon.com

Click on the ad to read more

3.4 Review Questions

1. If an array holds integers, each of which is four bytes long, how many bytes from the base location of the array is the location of the fifth element?
2. Is the formula for finding the location of an element in a dynamic array different from the formula for finding the location of an element in a static array?
3. When a dynamic array expands, why can't the existing elements be left in place and extra memory simply be allocated at the end of the existing memory allocation?
4. If a Ruby array `a` has `n` elements, which element is `a[n-1]`? Which is element `a[-1]`?

3.5 Exercises

1. Suppose a dynamic integer array `a` with indices beginning at 0 has 1000 elements and the line of code `a[1000] = a[5]` is executed. How many array values must be moved from one memory location to another to complete this assignment statement?
2. Memory could be freed when a dynamic array shrinks. What advantages or disadvantages might this have?
3. To use a static array, data must be recorded about the base location of the array, the size of the elements (for indexing), and the number of elements in the array (to check that indexing is within bounds). What information must be recorded to use a dynamic array?
4. State a formula to determine how far from the base location of a Ruby array an element with index *i* is when *i* is a negative number.
5. Give an example of a Ruby array reference that will cause an indexing error at run time.
6. Suppose the Ruby assignment `a=(1..100).to_a` is executed. What are the values of the following Ruby expressions? Hint: You can check your answers with the Ruby interpreter.
   a) `a[5..10]`
   b) `a[5...10]`
   c) `a[5, 4]`
   d) `a[-5, 4]`
   e) `a[100..105]`
   f) `a[5..-5]`
   g) `a[0, 3] + a[-3, 3]`

8. Suppose that the following Ruby statement are executed in order. What is the value of array `a` after each statement? Hint: You can check your answers with the Ruby interpreter.
   a) `a = Array.new(5, 0)`
   b) `a[1..2] = []`
   c) `a[10] = 10`

Download free eBooks at bookboon.com

d) `a[3, 7] = [1, 2, 3, 4, 5, 6, 7]`
e) `a[0, 2] = 5`
f) `a[0, 2] = 6, 7`
g) `a[0..-2] = (1..3).to_a`

3.6 Review Question Answers

1. If an array holds integers, each of which is four bytes long, then the fifth element is 16 bytes past the base location of the array.
2. The formula for finding the location of an element in a dynamic array is the same as the formula for finding the location of an element in a static array. The only difference is what happens when a location is beyond the end of the array. For a static array, trying to access or assign to an element beyond the end of the array is an indexing error. For a dynamic array, it may mean that the array needs to be expanded, depending on the language. In Ruby, for example, accessing the value of `a[i]` when `i` ≥ `a.size` produces `nil`, while assigning a value to `a[i]` when `i` ≥ `a.size` causes the array `a` to expand to size `i`+1.
3. The memory allocated for an array almost always has memory allocated for other data structures after it, so it cannot simply be increased without clobbering other data structures. A new chunk of memory must be allocated from the free memory store sufficient to hold the expanded array, and the old memory returned to the free memory store so it can be used later for other (smaller) data structures.
4. If a Ruby array `a` has `n` elements, then element `a[n-1]` and element `a[-1]` are both the last element in the array. In general, `a[n-i]` and `a[-i]` both access the same element.

Download free eBooks at bookboon.com