Carleton University
Department of Systems and Computer Engineering
SYSC 2006 - Foundations of Imperative Programming

C Arrays and Python Lists

The C variable declaration:

```
type name[capacity];
```

allocates an array with the specified *name*. The array's *capacity* is an integer expression, and specifies the number of elements in the array. Each element in the array stores a value of the specified *type*. So, the declaration

```
int numbers[10];
```

declares an array named `numbers` that has 10 elements, each storing one signed integer.

An element in an array is accessed by specifying the array name and the element's position (index), which is given by an integer that ranges from `0` to *capacity*`-1`. For example, `numbers[0]` is the first element in array `numbers`, `numbers[1]` is the second element, and `numbers[9]` is the tenth element.

An array index does not have to be a literal integer; instead, we can use any expression that yields an integer. Often, the index is a variable of type `int`. As an example, here is a loop that initializes `numbers` with the first 10 even integers, starting with 0:

```
// initialize numbers to {0, 2, 4, 6, ..., 18}
int numbers[10];

for (int i = 0; i < 10; i += 1) {
    numbers[i] = 2 * i;
}
```
[1]

[1] Here is an equivalent Python loop that creates and initializes a list:

```
# initialize numbers to [0, 2, 4, 6, ..., 18]
numbers = [0] * 10   # create a list of 10 0's
for i in range(10):
    numbers[i] = 2 * i
```

A more common approach in Python is to create an empty list, then append the ten integers, one at a time:

```
# initialize numbers to [0, 2, 4, 6, ..., 18]
numbers = []
for i in range(10):
    numbers.append(2 * i)
```

C allows us to specify the initial values of an array's elements by providing an *initializer list* as part of the array declaration. For example, this statement:

```
int nums[] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18};
```

declares and initializes array `nums` with the first 10 even integers, starting with 0.[2] Notice that the declaration doesn't the array's capacity. The C compiler calculates the capacity, based on the number of values in the initializer list. In this example, the capacity of `nums` is 10 (signed integers).

Here's a C function that returns the sum of the first *n* values in an array of `int`s:

```
int sum_array(int arr[], int n)
{
    int sum = 0;
    for (int i = 0; i < n; i += 1) {
        sum = sum + arr[i];
    }
    return sum;
}
```
[3]

The first parameter, `arr`, refers to the array. Notice how this parameter is declared: the square brackets, `[]`, after the parameter name specify that it refers to an array. The declaration doesn't specify the array's capacity. As a result, the function will process any array with elements of type `int`, regardless of its capacity. (Of course, the sum of the array elements must not be greater

---

[2] This is equivalent to the Python statement:

```
nums = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

[3] The equivalent Python function is:

```
def sum_list(lst, n):
    sum = 0
    for i in range(n):
        sum += lst[i]
    return sum
```

If we want the function to sum all the elements in the list, we can delete parameter `n`:

```
def sum_list(lst):
    sum = 0
    for i in range(len(lst)):
        sum += lst[i]
    return sum
```

Python's `for` loop will iterate over all the elements in a list, so we can simplify this function:

```
def sum_list(lst):
    sum = 0
    for elem in lst:
        sum += elem
    return sum
```

than the largest `int` value.) It is the programmer's responsibility to ensure that the first *n* elements of the array have been initialized.

As an example of using an array as a function argument, here is how we call `sum_array` to sum all the integers in array `nums`:

```
int nums[] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18};

int capacity = sizeof(nums) / sizeof(nums[0]);

int total;
total = sum_array(nums, capacity);
```

Notice that the first argument is the name of the array, `nums`, and not `nums[]`.

We can call the same function to sum just the first five elements of array `numbers`; i.e., calculate `numbers[0] + numbers[1] + numbers[2] + numbers[3] + numbers[4]`:

```
int partial_sum;
partial_sum = sum_array(numbers, 5);
```

Because array parameters store references to arrays, functions can modify their array arguments. Here's a function that initializes the first *n* elements of the array referred to by `arr` to a specified integer value:

```
void initialize_array(int arr[], int n, int initial)
{
    for (int i = 0; i < n; i += 1) {
        arr[i] = initial;
    }
}
```
[4]

To initialize all 10 elements in `numbers` to 0, we call the function this way:

```
int capacity = sizeof(numbers) / sizeof(numbers[0]);

initialize_array(numbers, capacity, 0);
```

[4] The equivalent Python function is:

```
def initialize_list(lst, n, initial):
    for i in range(n):
        lst[i] = initial
```

Of course, an experienced Python programmer would dispense with the function and instead use a single statement to create an initialize the list:

```
numbers = [initial] * n
```

A Comparison of C Arrays and Python Lists

C arrays are similar to Python lists, in that the elements in arrays and lists are accessed by position, but there are several important differences:

- When we create a Python list, we don't specify its capacity. A Python list automatically grows (increases its capacity) as objects are appended or inserted. In contrast, the capacity of a C array is specified as part of its declaration. The array's capacity is fixed; there is no way to increase its capacity at run-time.

- We can determine the length of a Python list (that is, the number of objects stored in the list) by passing the list to Python's built-in `len` function. In contrast, C does not keep track of how many array elements have been initialized, and there is no function we can call to determine this. It is the programmer's responsibility to do this; for example, by using an auxiliary variable.

- Python generates a run-time error if you specify an invalid list index; for example, when executing this statement:

  ```
  elem = numbers[10]   # assuming len(numbers) <= 10
  ```

  Python will report an error along the lines of: `builtins.IndexError: list index out of range`.

  In contrast, C does not check for out-of-bounds array indices. For example, a C expression such as `numbers[10]` will compile without error. At run-time, this expression accesses memory outside the array.

  Python list elements can be accessed with negative indices between `-1` and `-len(`*`lst`*`)`, so `numbers[-1]` and `numbers[-len(numbers)]` are perfectly valid Python expressions. In C program, negative indices access memory outside the array.

- Python provides functions, methods and operators that perform several common operations on lists; for example, append an object to the end of a list, insert an item in a list, delete an item from a specified position in a list, remove a specified object from a list, determine if a specified object is in a list, find the largest and smallest objects in a list, etc.

  The only array operation C provides is the `[]` operator to retrieve or set the value at a specified index.

- In Python, when `lst2` is bound to a list, the statement `lst1 = lst2` doesn't make a copy of `lst2`; instead, it make initializes `lst1` so that it refers to the same list as `lst2`. (A shallow copy of `lst2` can be created by calling the copy method; e.g., `lst.copy()`, or by creating a slice of the entire list; e.g., `lst2[:]`.)

  C doesn't provide an operator that assigns one array to another; for example, this code won't compile:

  ```
  int arr[] = {1, 2, 3, 4};
  int arr2[4];
  arr2 = arr;
  ```

Last edited: Jan. 27, 2019