Exercise 7-1: array_pointer.c

Maximilian Fernaldy - C2TB1702

Note: some links and other HTML-related objects may not work in pdf form. Consider reading the webpage format of the report here.

Introduction to "array pointers" and why we use array subscripts instead of pointer arithmetic notation

We have covered in the previous lecture that pointers are essentially variables that *point* to the memory address of another variable. In the case of arrays, in a sense, when we create an array, we have already created a pointer to it. By assigning a name to an array like so:

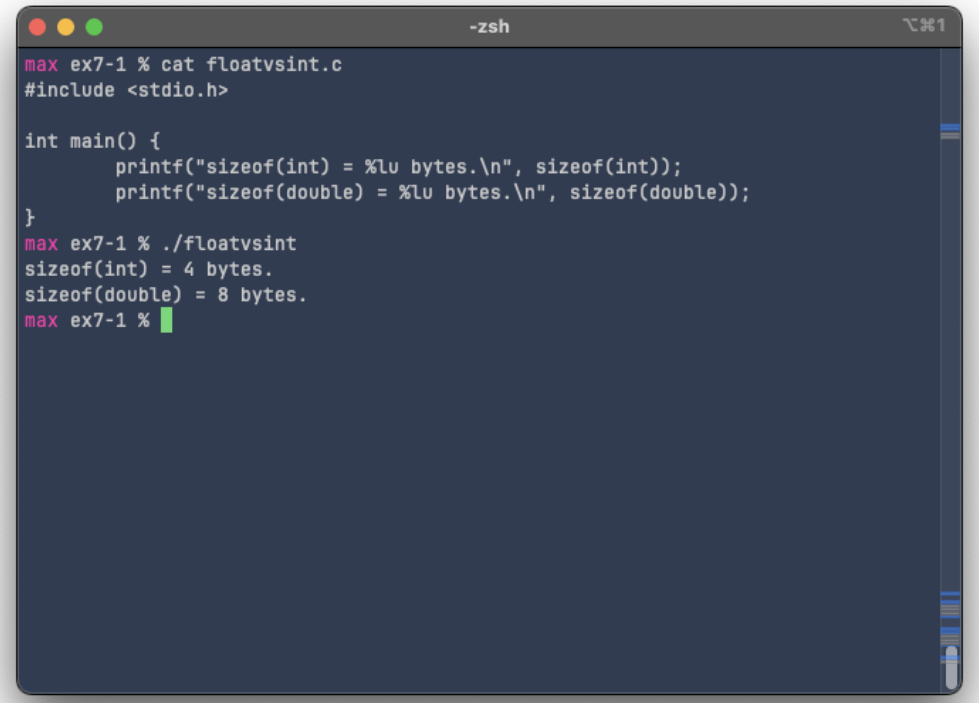
```
int array[100];
```

we have assigned a pointer called `array` to the first element in the array. Note that this does not mean all the elements in the array is stored in a single memory location/address. Memory addresses are *not* the same as pointers. Pointers *store* memory addresses, but they are not the same. To access the elements, we must use the *offset* or the *index* of the element we want to access. For example, if we want to access the i-th element of the array we created earlier using a pointer,

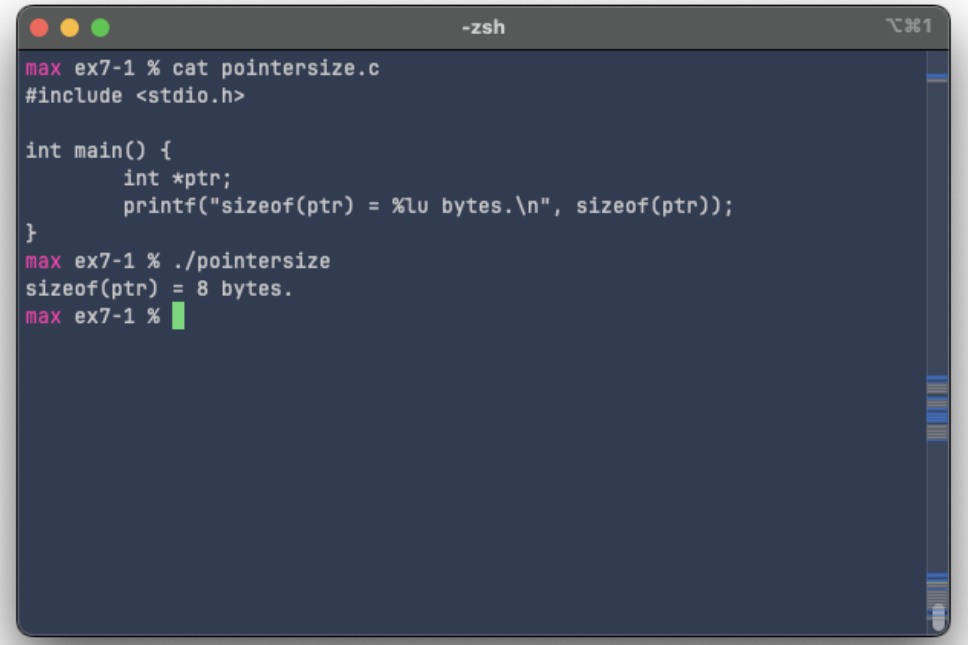
```
*(array+i)
```

will do it. We can think of it as there being a single pointer pointing to the first element of the array, and that pointer is `array`. When we add an offset `i` to the pointer and *then* dereference it, it's going to access the value that is stored in the i-th index of the array. Even though it might seem like something very obvious, there's something really important and interesting happening here, and it has something to do with data types and the space they take in memory.

Recall that different data types require different amounts of memory. The specific size depends from system to system, for example, an `int` type typically takes 4 bytes, and `double` typically takes 8 bytes on 32-bit and 64-bit systems. But even this is not always true, and the actual size can only be determined by using `sizeof(int)`. However, it always stands that some types that provide more precision/maximum size of stored data will require more memory. A `double` will always require more memory than an `int`:



Obviously, since pointers are just variables that store addresses, they too, have a size requirement:



So, if different types have different size requirements, how does the program know where the next element is when we use `*(array+i)` ? Well, remember that when we declare the pointer with `int array`, we are assigning a type to the variable. This type is immutable—it can't ever be changed unless we convert it into another type by assigning its value to a new variable, but that does not change the type of the variable, it creates a new variable with a different type. This behavior in languages is called "strongly typed", and strongly typed languages have numerous benefits over their *loosely typed* counterparts, especially in low-level memory allocation and optimization like the example above. As C is a strongly typed language, **it allows the compiler to know exactly**

what variable type will be stored in the memory space pointed to by the pointer, and this is beneficial for when we use `*(array+i)`, because if it knows the size of each element in the array, it knows where exactly each element starts and ends, and this is how it knows how to shift between elements in an array without us specifying how many bytes to shift.

But if we try reading some C code that other people write, it's almost immediately obvious that the notation `*(array+i)` is almost **never** used to access elements of an array using the pointer. This is because, as we have gone over earlier, **array "names" are pointers to the first element.** This means, when we use `array[i]` to access an element of an array, it's effectively doing the same thing as `*(array+i)`, just using a different operator—the *array subscript* operator, or the square brackets. The array subscript method is simply a feature of the C programming language to make it easier for programmers to read the code. Reading `array[i]` will *instantly* give you an intuitive understanding—it's the i-th element in `array[]`. Reading `*(array+i)`, however, might not be so obvious.

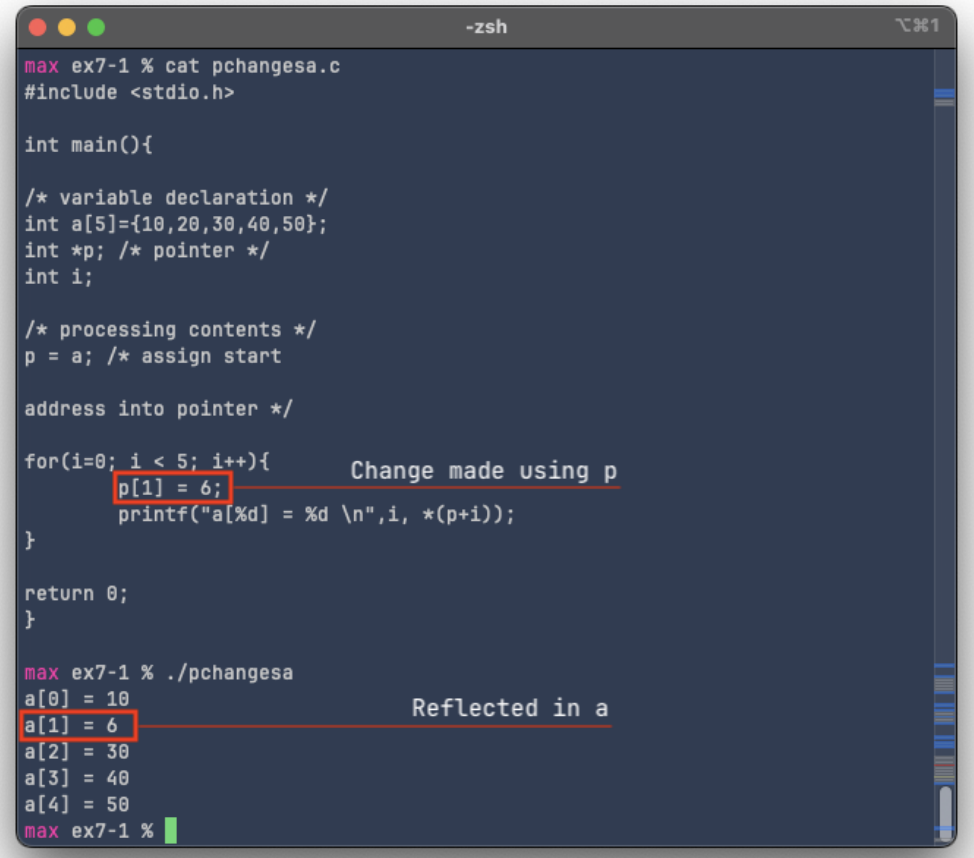
That doesn't mean the pointer arithmetic notation is useless, though. It's particularly useful when teaching concepts like the relation between arrays and pointers, and memory allocation. I hope I did a decent job at explaining my understanding of arrays and pointers in this report.

Exercise 7-1

Instead of accessing elements in `a` as usual, we are tasked to use a pointer `p` instead. To do this, we can simply use pointer arithmetic notation `*(p+i)` instead of the usual `a[i]`. They both mean exactly the same thing.

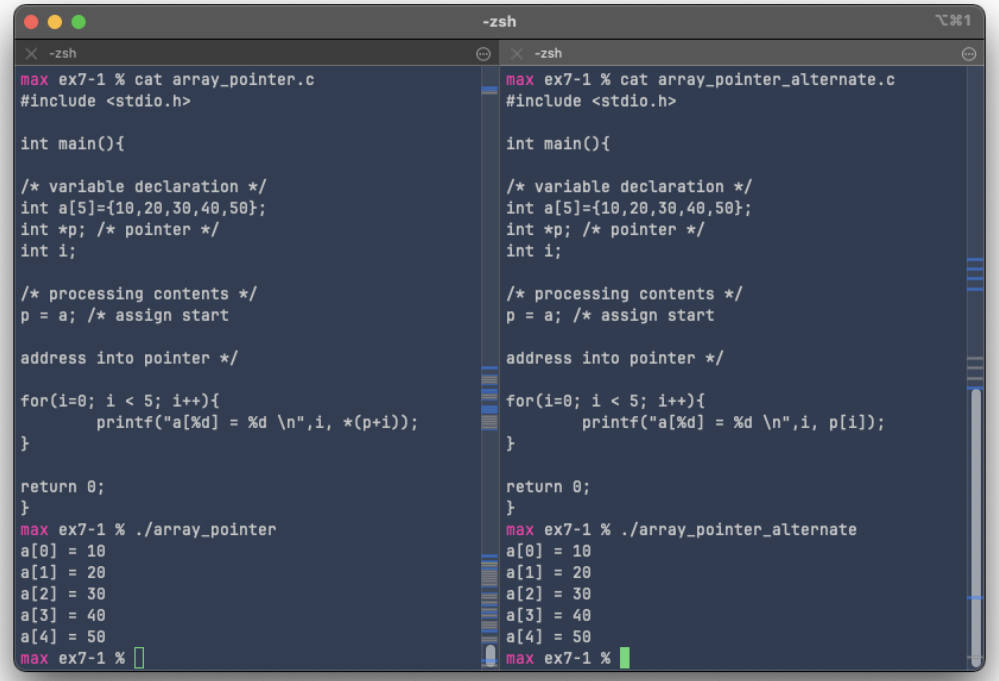
```
for(i=0; i < 5; i++){
    printf("a[%d] = %d \n",i, *(p+i));
}
```

In the previous section I mentioned that array "names" are simply pointers to the first element of the array. We can see this proven as we assigned `a` to `p` with `p = a;` to assign the start of the array to our pointer `p`. Notice that by doing `p = a`, we are simply copying the address of the first element of the array into another variable. This does not create another array, it just creates another "name" for the array that it can be accessed from. Any changes made to `p` will also reflect in `a`—this is an instance of *pass by reference*.

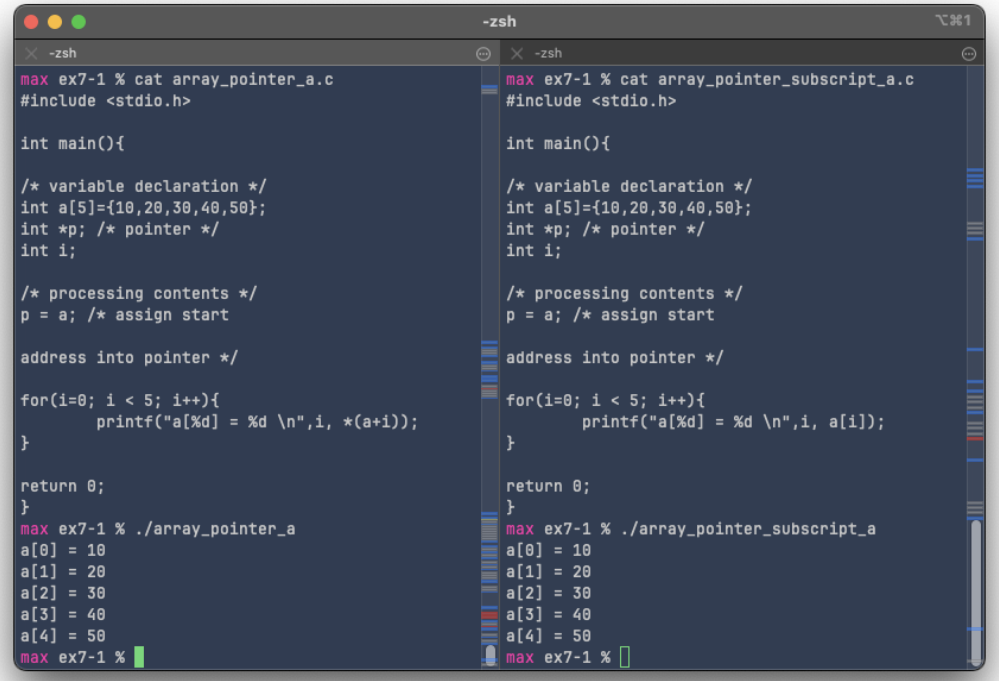


Additionally, since `p` is just a copy for the address of the first element of the array, we can infer that it can also be treated as another `name` of the variable. What this means is now, `p[i]` and `a[i]` do exactly the same thing. If we do the following, the output will be the same.

```
for(i=0; i < 5; i++){
    printf("a[%d] = %d \n",i, p[i];
}
```



and of course, `a[i]` and `*(a+i)` both work, too.



I think the purpose of exercise 7-1 is to demonstrate the relationship between arrays and pointers, and not necessarily an example of how to use pointers to modify or access arrays—creating alternate names like `p = a` is certainly useful, but for the use case of accessing elements of an array, `*(p+i)` is less preferable as it is much less readable than `p[i]` or `a[i]`.