Arrays

This lesson will introduce you to arrays. You will learn how arrays are defined and how to access array elements. In addition, you will also get to know about how C implements arrays and what precautions we have to take while accessing elements.

We'll cover the following

Just as in MATLAB, or Python, or any number of other high-level languages, C provides a data structure called an **array**. An array in C is a set of ordered items. You can think of this as a vector, a list, etc, there are many names. Typically in C we call these an array.

Defining an Array

As an example, let's say we want to store a list of 5 grades. We can define an array as follows:

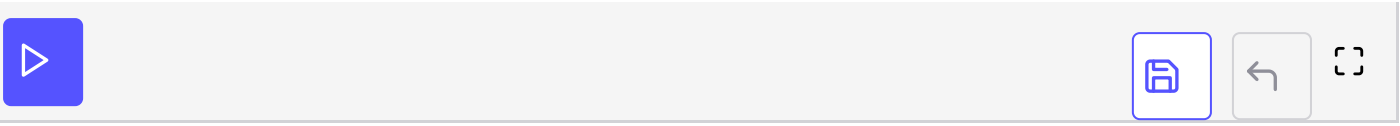
```
int grades[5];
```

This declaration says we want to set aside space in memory for 5 integer values, and we can refer to that block in memory using the variable name "grades". Note that we have only allocated the space in memory, we have not initialized any values of the array. Whatever values happened to be in memory at the locations set aside when we declared the grades variable, will still be there. We can have a look at what is there by **indexing** into the array like this:

```
#include <stdio.h>

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

```
  return 0;
}
```



> **Note a couple of things.**
>
> 1. The first index always starts with 0 (this is the same in Python, but in MATLAB for example, indices start at 1).
> 2. The values in the array right after declaring the variable will not be initialized, they will contain whatever values happened to be in memory at those locations before. (You will probably see different values when you run your code).

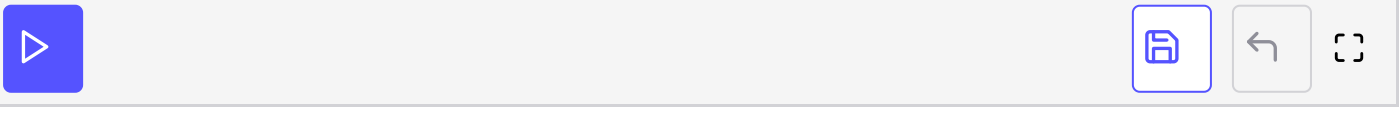
Indexing Array Elements

We have seen that array indices always start at 0 (not 1 as in MATLAB). It's also important to know that once an array of a given size is declared (and memory is allocated), the array size is fixed. That is, you cannot extend the array and make it larger (or make it smaller). We will see exceptions to this later, when we talk about dynamically allocated memory using `malloc()`, but for now, assume arrays are fixed in size once declared.

If we try to access an element of an array beyond its bounds, like for example accessing the 6th element of the `grades` array defined above, **sometimes C will not prevent us from doing that**.

```
#include <stdio.h>

int main ()
{
  int grades[5];
  int i;
  for (i=0; i<5; i++) {
    printf("grades[%d]=%d\n", i, grades[i]);
  }
  printf("grades[5]=%d\n", grades[5]);
  printf("grades[500]=%d\n", grades[500]);
  return 0;
}
```



In fact, **we will not even get a warning** from the compiler. This highlights a general difference in approach between C and other high-level languages (especially interpreted languages like Python and MATLAB) — C will do exactly what you tell it to.

To understand why asking for the 6th element of a 5-element array is not nonsensical, we have to understand some details about how C represents arrays in memory, and how accessing memory by indexing arrays works in C. When you declare a variable `grades` that is a 5-element array of integers, what C actually does, is two (main) things:

1. A consecutive block of memory is "set aside" (reserved for use). The amount of memory that is set aside is equal to the number of elements of the array multiplied by the size of the declared element type. So if we declare `int grades[5];`, then 20 bytes (5 x 4 bytes) of consecutive memory is set aside.

2. The variable name `grades` is assigned to the block of memory. In fact what actually happens, is that the variable name `grades` is assigned a **pointer** to the address in memory corresponding to the first element of the array (the beginning of the block of consecutive memory that was set aside). Section 5.3 of the Kernighan and Ritchie book will take you through this in more detail.

Then when you index into the `grades` array, for example like `grades[0]` or `grades[3]`, C will look at the appropriate place in memory, defined by the beginning of the array (the `grades` **pointer**) plus the appropriate number of "steps" into the memory block, as defined by the index, and the size of the basic type as defined in the original array declaration. So if you access the 3rd element of the `grades` array with `grades[2]`, what C actually does is jumps 8 bytes (2 x 4) into the block of memory that was set aside, and reads the value it finds there. So if you ask for the 500th element of the array, C simply reads the memory location 500 x 4 = 2000 bytes past the beginning of the memory block, **whatever that may be**.

As you can imagine, this lack of "protection" represents a prime opportunity to generate programming errors that can be difficult to debug. What's particularly dangerous, however, is that this feature of C also applies to **assigning** values to an array.

Now that we're able to define an array, let's give it values and play around with

Now that we're able to define an array, let's give it values and play around with them.