Tutorial 7

- Modularization.
- Arrays.
- Pointer arithmetic.
Modularization

In the last tutorial, we mentioned that the motivation for using modules was:

- Abstraction
- Maintainability
- Reusability

Today we are going to focus on each of these individually.
Abstraction: Client/Module Relationship

- It is helpful to think of yourself as a client when including a module.
- You have access to the functions included in the module, but not the implementation of those functions. The documentation in the interface is all you need to know about the code.
- For example, in seashell we’ve provided array_io.h. You are expected to use these functions without seeing the implementation.
Maintainability: Hiding the back-end

- As long as you don’t change how the code is used, things such as bug-fixes or changes to how the code runs can be done on the back-end without the client being affected.

- To show this point, we will be writing a function reverse in three different ways without modifying its interface.

```c
// reverse(a, len) reverse the elements
// of the array a given its len
// effects: reverses the elements of a
// requires; len > 0
void reverse(int a[], int len);
```
Reusability: The lazy CS major’s favourite

• Clients can use modules across multiple programs.
• This allows us to complete different tasks using the same module.
• stack.h is a good example of this concept in action; It will be used in this tutorial, and in the current assignment
Arrays

Arrays can be thought of as pointers to a block of memory. They can be used to store a **fixed number** of elements of the **same type**.

Example of array syntax:

```c
int my_array[3] = { 1, 2, 3 };  
int x = my_array[0];  // x = 1
```

In the above code, `my_array` is a pointer to the first element of the array. The `[i]` syntax **dereferences** the `(i + 1)th` element of the array.

The syntax `a[i]` is shorthand for the equivalent expression `*(a+i).`
Array Initialization

There are several ways to define an array:

```c
int a[3]; // array is not fully defined

int b[3] = { 1, 2, 3 }; // array length is explicit

int c[] = { 1, 2, 3 }; // array length is inferred
// c has length 3, and we cannot change it later

int d[3] = {0}; // array of length 3, filled with zeros

int e[8] = { 7, 4, 1 }; // {7, 4, 1, 0, 0, 0, 0, 0}
```
Array Exercise

//See reverse.h for general documentation.
void reverse(int arr[], int len);
Pointer Arithmetic

Certain arithmetic operations can be performed on pointers. An integer can be **added or subtracted** to a pointer, and pointers of the same type can be **subtracted** from one another.

```c
int a[10];
int *p = a;     // a is a pointer to first element
int *q = &a[9]; // address of 10th element
q = a + 9;      // equivalent
a[2] = q - p;   // set the value of 3rd element as 9
q = p + 1;      // now q == &a[1]
```

Addition of pointers is not allowed.
Array Notation vs Pointer Notation

If you ask the right people, you can start heated debates on whether to use `int a[]` or `int *a` in a parameter declaration.

_They are equivalent._

Some people will argue for the first style because it is clear that the function takes an array.

Others believe the first style is misleading, as the array isn’t actually passed by value and using `sizeof` on it is unintuitive.

_You should be familiar with both ways of passing arrays._
Array Exercise 2, Electric Boogaloo

Write Reverse again, now using pointer arithmetic
(hint, this code will be essentially identical to reverse)

//See reverse.h for general documentation.
void reverse(int arr[], int len);
ADTs : Abstract Data Types

• Formally, an ADT is a mathematical model for accessing data through operations

• In practice, this means that an ADT is a module that only allows access to the data through functions

• Since ADTs are usually modules they generally have the same motivations (Abstraction, Maintainability, and Reusability)
Array Episode 3, Revenge of the Stack

Write Reverse a third time, now using stack.h

//See reverse.h for general documentation.
void reverse(int arr[], int len);