# Week 1 Notes, Tues May 9

## Course Introduction

**Sites to bookmark**: Canvas, Github

**Syllabus and marking scheme**: see Canvas

**TAs and office hours**: see Canvas

**Content overview**: arrays, linked lists, recursion, analysis tools (e.g. O-notation), stacks, queues, deques, trees, heaps, priority queues, hash tables, maps, search trees, sorting, graph algorithms, memory management, B-trees

A **data structure** is a way of organizing data. Arrays, lists, trees, tables, and graphs are all examples of data structures.

In this course, an **algorithm** is a description of sequence of actions on a data structure, usually to achieve some specific goal. Summing an array, printing all the “leaf” nodes in a tree, and finding a given string in a table are all examples of algorithms.

## Sum in a Dozen Different Ways

An overview of basic C++, including pointers, passing by constant reference, vectors, using a standard library function (accumulate), and a generic (templated) function.

See **sum.cpp**.

What version of summation do you like the most? Which one is most readable?

## Recursion

**Recursion** occurs when function that calls itself, either directly or indirectly. For some algorithms and data structures, recursion is a natural and useful way to write code, and so we will see it throughout the course.

**recursion.cpp** includes a few examples of recursive functions.

## Introduction to Linked Lists

Linked lists in pictures, without a class, with a class.

A basic **singly-linked list** of strings, with these basic operations:

* Test if the list is empty.
* Insert a new node at the front.
* Remove the node at the front.
* Count the number of nodes.
* Print the values of the nodes.
* Remove all elements.

Each “node” of a linked-list stores at least two things: it’s data value, and a pointer to the next node in the list. So we can represent a node like this:

struct Node {  
 string data;  
 Node\* next;  
};

The node stores a string, plus a pointer to the next node in the list. The next pointer for the last node of the list will have the value nullptr.

We need to know what node is at the start of the list, so will use a special Node pointer for that:

Node\* head = nullptr;

head always points to the first element of the list. When head is nullptr, the list is empty.

To create a new node for the list, we do this:

head = new Node{"hello", nullptr};

This creates a new Node, and also makes head point to it. This is now a linked list of length 1.

To delete the node, we can do this:

delete head;  
head = nullptr;

**Remember**: Every time we call new, there must, eventually, be a call to delete that de-allocates the new-ed memory. Otherwise you have a **memory leak**.

How do you add a new node to the front of the list?

How do you remove the node at the front of the list?

How do you count the number of nodes on the list?

How do you print the nodes on a list?

How do you remove all elements?