Tut 2

Previous tut clarification:

- `.i` vs `.c` files -> `.i` files are pure C files (include no headers, macros, comments, conditional macros), `.c` files are normal C files. Every `.i` file is a `.c` file, but the reverse is not true. So the first preprocessing step does convert `.c` files to `.i` files, but they both are C code files. Also, extension of `.i` files doesn't matter, can be `.c` as well, and it will still go throught with the compilation producing the same executable
- The linker verifies that any references to names (symbols) in a .o file are present in other .o, .a, or .so files. For example, the linker will find the printf function in the standard C library (libc.so). If the linker cannot find the definition of a symbol, this step fails with an error stating that a symbol is undefined.
- Compilation example

Basic time complexity

What does O() denote?

O() denotes how the time for a particular function scales with respect to something

- O(n) means doubling n should double the execution time
- O(n^2) means doubling n should quadruple the execution time

```
for (int i = 0; i < N; ++i) {
    ...
}
```

O(n)

```
for (int i = 0; i < n; ++i) {
    for (int j = i+1; j < m; ++j) {
        ...
    }
}
```

O(nm), starting point of i doesn't matter, inner function still scales linearly with respect to m

```
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        ...
```

```
    }
}
for (int i = 0; i < n; ++i) {
    ...
}
```

O(n^2 + n) = O(n^2), since the overall execution time is going to be dominated by the first loop for big values of n

```
for (int i = 0; i < n; ++i) {
 ...
}
for (int j = 0; j < m; ++j) {
    ...
}
```

O(n + m), since n and m are independent

```
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
        ...
    }
}
for (int i = 0; i < m; ++i) {
    ...
}
```

O(n^2 + m) , since n and m are independent, we cannot say which of them will dominate later on.

Stacks

- Used to model FILO order (First in, Last out)
- Operations defined for a stack:
  - Push(s, x) -> Adds new element x on top of stack s
  - Pop(s) -> Removes the top element of the stack s and returns it
  - Empty(s) -> Checks if stack s is empty
  - Size(s) -> Returns the number of elements in stack s
  - top(s) -> Returns the top element of the stack s
- Time complexity for all operations is O(1)
- Can be implemented using:
  - Linked lists

- Arrays
- stack coding example: checking if various brackets are matched

---

Queue

- Used to model FIFO order (First in, First out)
- Operations defined for a queue:
  - Enqueue(q, x) -> Adds new element q to the back of the queue q
  - Dequeue(q) -> Removes element from the front of the queue q and returns it
  - Empty(q) -> Checks if queue q is empty
  - Size(q) -> Returns number of elements in the queue q
  - front(q) -> Returns the element in the front of the queue q
- Time complexity for all operations is O(1)
- Can be implemented using:
  - Linked lists
  - Arrays
- circular array implementation of queue (head and tail loops around, homework)
- queue coding example: Generating all binary numbers with length $\leq$ n

---

Bonus: Good coding practices

- Code should be reusable (because you are going to reuse it later in the course)
  - Eg: Do not store length of a vector as a global variable, then you can't declare multiple complex numbers with different lengths
- Code should be extendable (because you will need to modify the data structures, so it should be easy)
  - Eg: Modify the vector to store angles from each axes as well
- Keep code as simple and readable as possible, no need to do fancy one liners if they are not understandable
  - Use comments to explain what part of code does so you know what it does after you open the code later
- Follow a consistent (standard) coding style
  - Eg: "Google C++ style guide"
  - Check how your ide/editor can auto-format your code to follow that style

> Programs must be written for people to read, and only incidentally for machines to execute
> -MIT Professor Harold Abelson

Lab code explaination

- Diagrams