Linked Lists

COMP1511 Week 8

Joanna Lin

Tutorial Outline

- In this tutorial, we will develop our toolbox of knowledge to analyse and solve more complex and abstract problems. Our focus will be on linked lists.
- We will learn about
  - struct pointers;
  - the *concept* of linked lists;
  - how to represent linked lists using structs and struct pointers
  - how linked lists compare to arrays — their benefits and downsides.

Struct Pointers

Struct Pointers

First Look

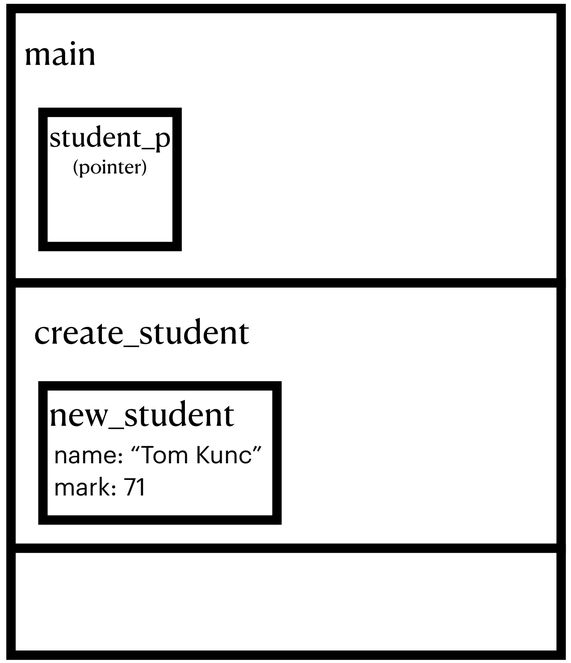
- Theoretically, they work like any other pointer.
  - we declare it by adding a `*` at the end of the struct type `struct student *student_p;`
  - struct pointers store memory addresses of structs.
  - we can dereference the pointer (change the struct at the memory address).
- C has syntactic sugar for accessing fields of the struct at the stored memory address.
  - Instead of `(*student_p).field_name`, we can write `student_p->field_name`.
- **Practically**, we create them quite differently from other pointers.
  - Initialising the fields of a struct is generally quite long and repetitive, so we often use functions as 'factories' to create a struct, initialise its fields and return a pointer to it.

Returning an Address: Bad

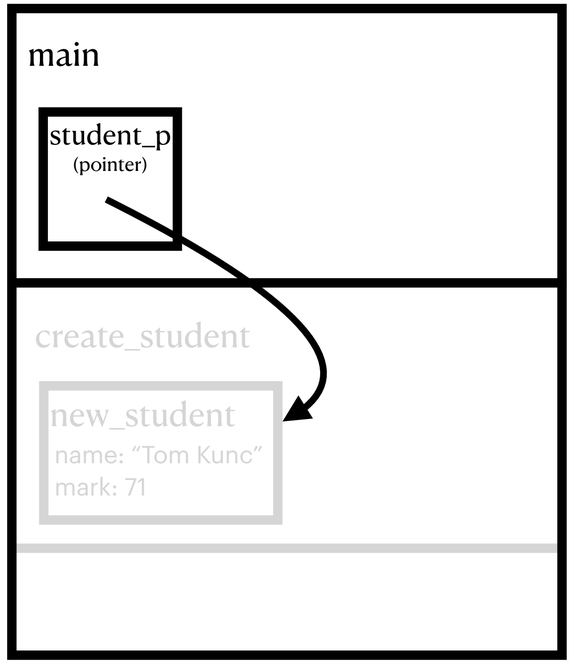
```c
#include <stdio.h>
#include <string.h>

#define MAX_NAME 50

struct student {
    char name[MAX_NAME];
    int mark;
};

struct student* create_student();

int main(void) {
    struct student *student_p = create_student();
    printf("My name is %s and my mark is %d\n", student_p->name, student_p->mark);
    return 0;
}

// Creates a default student and returns its address
struct student *create_student() {
    struct student new_student;
    strcpy(new_student.name, "Tom Kunc");
    new_student.mark = 71;
    return &new_student;
}
```

Returning an Address

Problem



`create_student` creates a `struct student` called `new_student` and initialises its fields.



when we exit `create_student`, its memory is deallocated and the address of `new_student` is stored in `student_p`.
now `student_p` is left pointing at memory that is unsafe to access.

The Heap

Self-Memory Management

- So far, all memory we've work with have been allocated on a part of computer memory called the **stack**.
  - As soon as a block of memory goes 'out of scope', it gets destroyed by the program.
  - This is good because we don't have to manage memory ourselves — the program does it for us and so memory usage stays efficient. However, it's not very helpful when we want to return an address!
- Instead, we take advantage of a separate block of memory called the **heap**.
  - Anything allocated on the heap stays allocated *forever* — the program will not manage that part of memory for us.
  - This means we must free (deallocate) memory on the heap when we no longer need that block of memory, otherwise we'll create **memory leaks**. This means that the block of memory can't be used by other processes even though we'll never use what is stored there ever again.
  - Luckily, most operating systems will automatically deallocate memory on the heap when your program terminates, but you should *not* rely on this. For every memory allocation on the heap, we must make sure we deallocate it!
  - We use the function `malloc` to allocate memory, and `free` to deallocate memory.

Returning an Address: Good

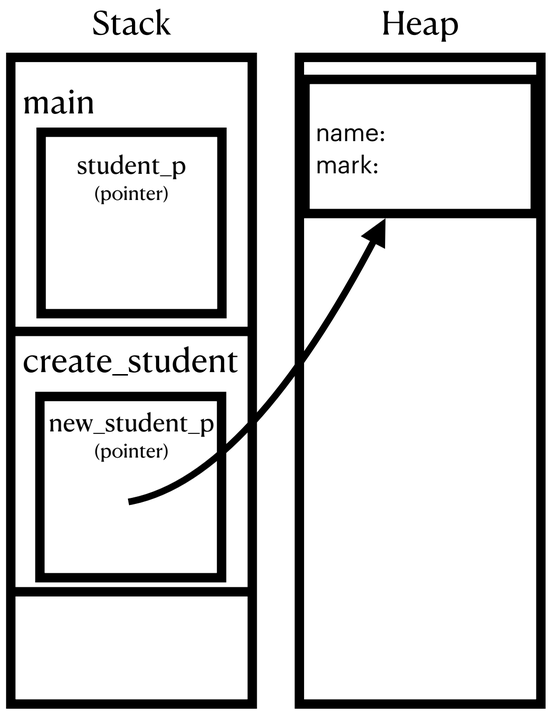
`free` and **`malloc`** come from **`stdlib.h`**

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX_NAME 50

struct student {
    char name[MAX_NAME];
    int mark;
};

struct student* create_student();

int main(void) {
    struct student *student_p = create_student();
    printf("My name is %s and my mark is %d\n", student_p->name, student_p->mark);
    free(student_p);
    return 0;
}

// Creates a default student and returns its address
struct student *create_student() {
    struct student* new_studentp = malloc(sizeof(struct student));
    strcpy(new_student_p->name, "Tom Kunc");
    new_student_p->mark = 71;
    return new_student_p;
}
```

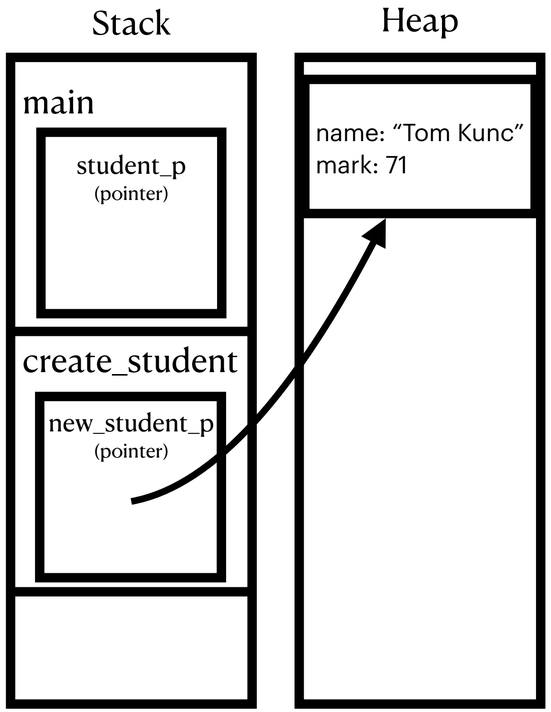
Frees (deallocates) memory from the heap once we no longer need so the memory can be used again

`memory allocate`:
Allocates memory sufficient for a **`struct student`** on the **heap**, and returns a pointer to the block of memory it allocated

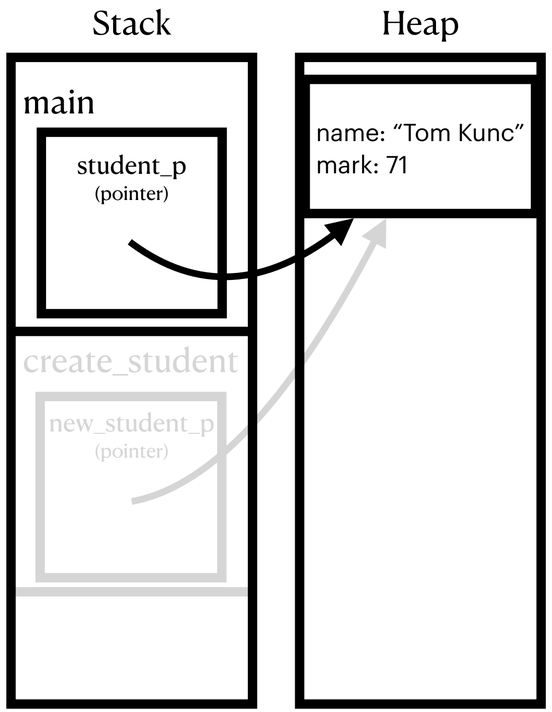
Memory Model



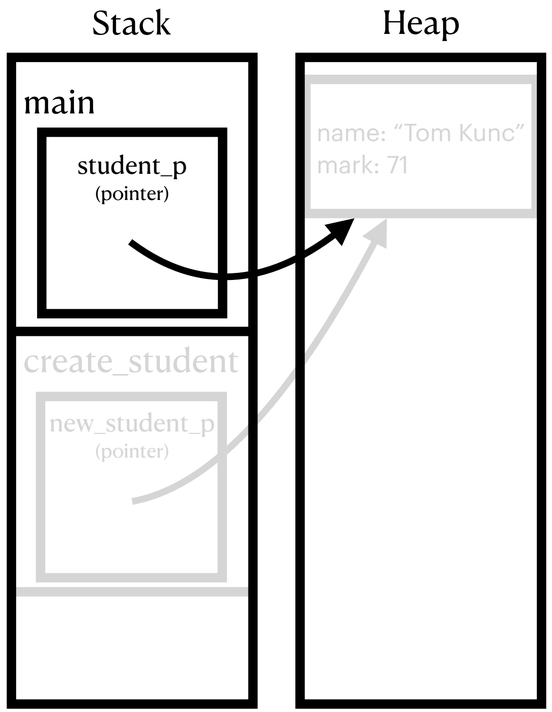
`malloc` allocates a block of memory on the heap for a `struct student`



The struct fields are initialised by dereferencing the `new_student_p` pointer



`create_student` returns the address to the heap-allocated struct, which is copied into the `student_p` variable. `student_p` points to the heap-allocated struct.



When **no longer needed**, we call `free` to deallocate memory on the heap so that other programs can use it.

<u>We should never access freed memory!</u>

- We can no longer guarantee that what is at the memory address is what we expect
- + security reasons etc...

Linked Lists

An Interesting Struct

Representing a 'Node'

- Recall how we can use structs in C, to represent things with characteristics (such as a student). We can also use it to represent something more abstract...
- Consider the following definition of a struct called a '**`struct node`**'

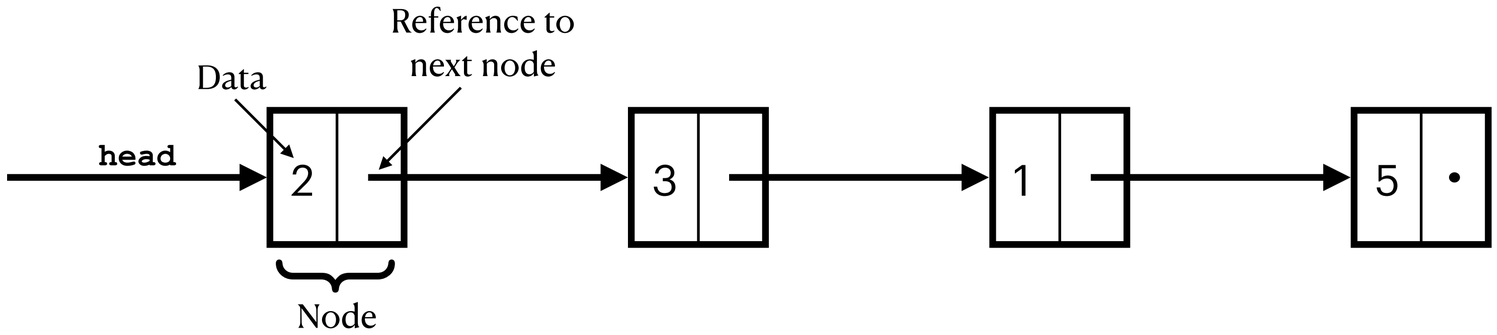
```
struct node {
    int data;
    struct node *next;
};
```

- It stores a pointer to a **`struct node`**!
  - We can make this **`struct node`** point to another **`struct node`**, which can then point to another **`struct node`**.
  - We can make something quite interesting with such a construction.

What is a Linked List

A Data Structure

- A linked list is an ordered collection of nodes. Every node contains piece(s) of data, and (with the exception of the last node) a reference to the next element in the list.
- We can visualise it like so



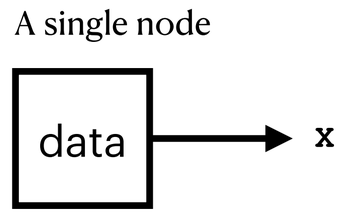
- Here, we have a *list* of 4 nodes *linked* together by references.
- We are usually given only where the first node of a list is through the **`head`** pointer. This is our means of accessing elements in the list.
  - To get to the *k*th node in the list, we will have to pass through all the nodes before it.
  - This is unlike an array where we are able to access any element immediately with an index.
  - Linked lists, however, offer other benefits like easy resizing and insertion at the start of a list.

Common Programming Patterns

Node Creation

- We mentioned at the beginning that we often use functions as a ‘factory’ for generating structs. We do the same for nodes in a linked lists. For example:

```c
// Creates a node initialised to the given data value
struct node *create_node(int data) {
    struct node* new_node = malloc(sizeof(struct node));
    new_node->next = NULL;
    new_node->data = data;
    return new_node;
}
```



- Why is it useful to return a pointer rather than the struct itself?
  - Makes it much easier to program operations on recursive data structures such as a linked list.
- We sometimes just call the node pointer the ‘node’.

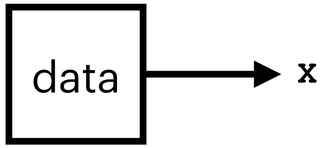
Problem Solving

- Whenever we are tasked with a problem to solve, we should always start by drawing a diagram.
- Suppose we wanted to write a function that inserts a node at the beginning of the list. We are given the list and a value to prepend
    1. We will need to create a node to prepend to the list.
    2. We will need to make that node point to the first element in the linked list.
    3. Readjust the head pointer.
- When we get to coding, we may need to work around some technical difficulties. Like
    - is there anything we've drawn that isn't practical in code? do we need to reorder some steps or perhaps figure out a workaround?
    - how do we tell the calling function that the head of the list has changed?
    - are there any 'edge' cases? That is, does our diagram assume something that doesn't work when the list is, say, empty or contains only one node etc.

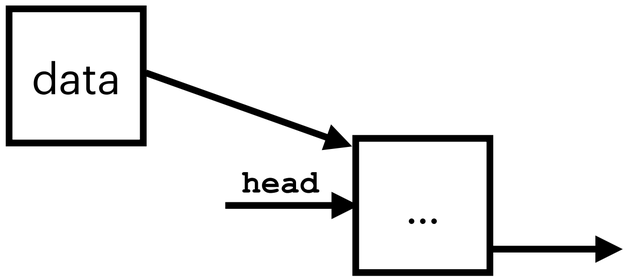
We have...

We should...

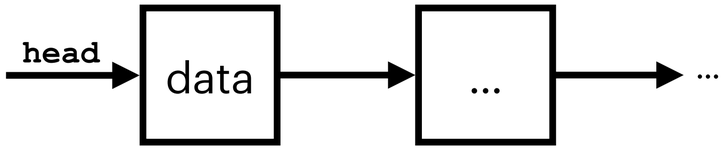
Step 1: Create the node to prepend



Step 2: Make the node point to the first element in the list



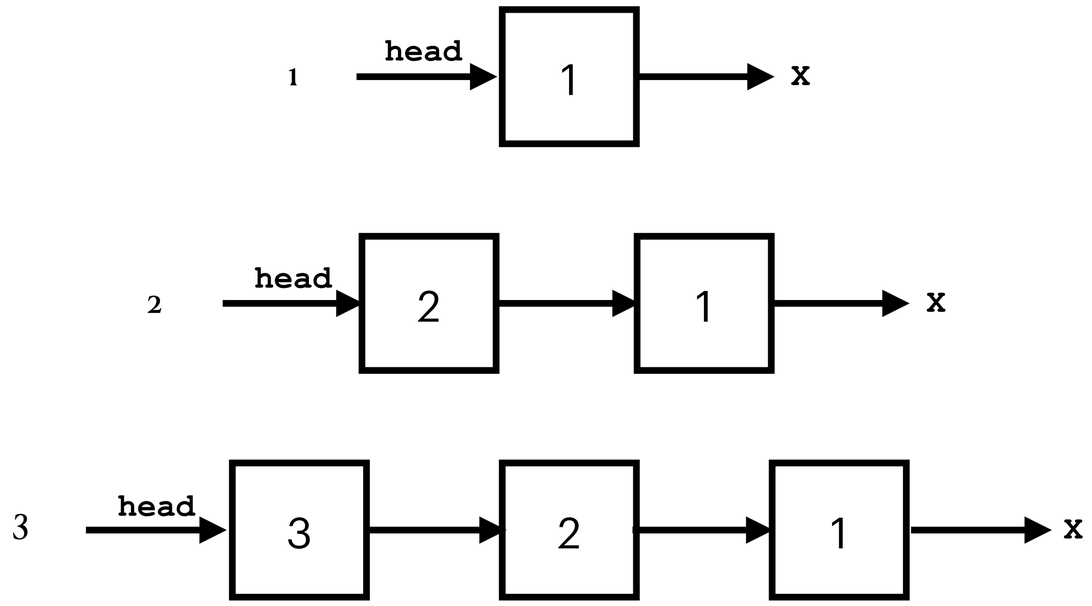
Step 3: Readjust the head pointer



Common Programming Patterns

Prepending a Node

- Functions dealing with linked lists often return the head of the list because the start of the linked list might have changed, and the program that called the function wouldn't know otherwise!




```c
struct node* prepend(struct node* head, int data);

int main(void) {
    struct node* head = prepend(NULL, 1);
    printf("Value at head is: %d\n", head->data);
    head = prepend(head, 2);
    printf("Value at head is: %d\n", head->data);
    head = prepend(head, 3);
    printf("Value at head is: %d\n", head->data);
    return;
}

// Prepend the given data to the given list
// Returns new head
struct node* prepend(struct node* head, int data) {
    struct node* new_node = create_node(data);
    new_node->next = head;
    return new_node;
}
```

Common Programming Patterns

Traversing a Linked List

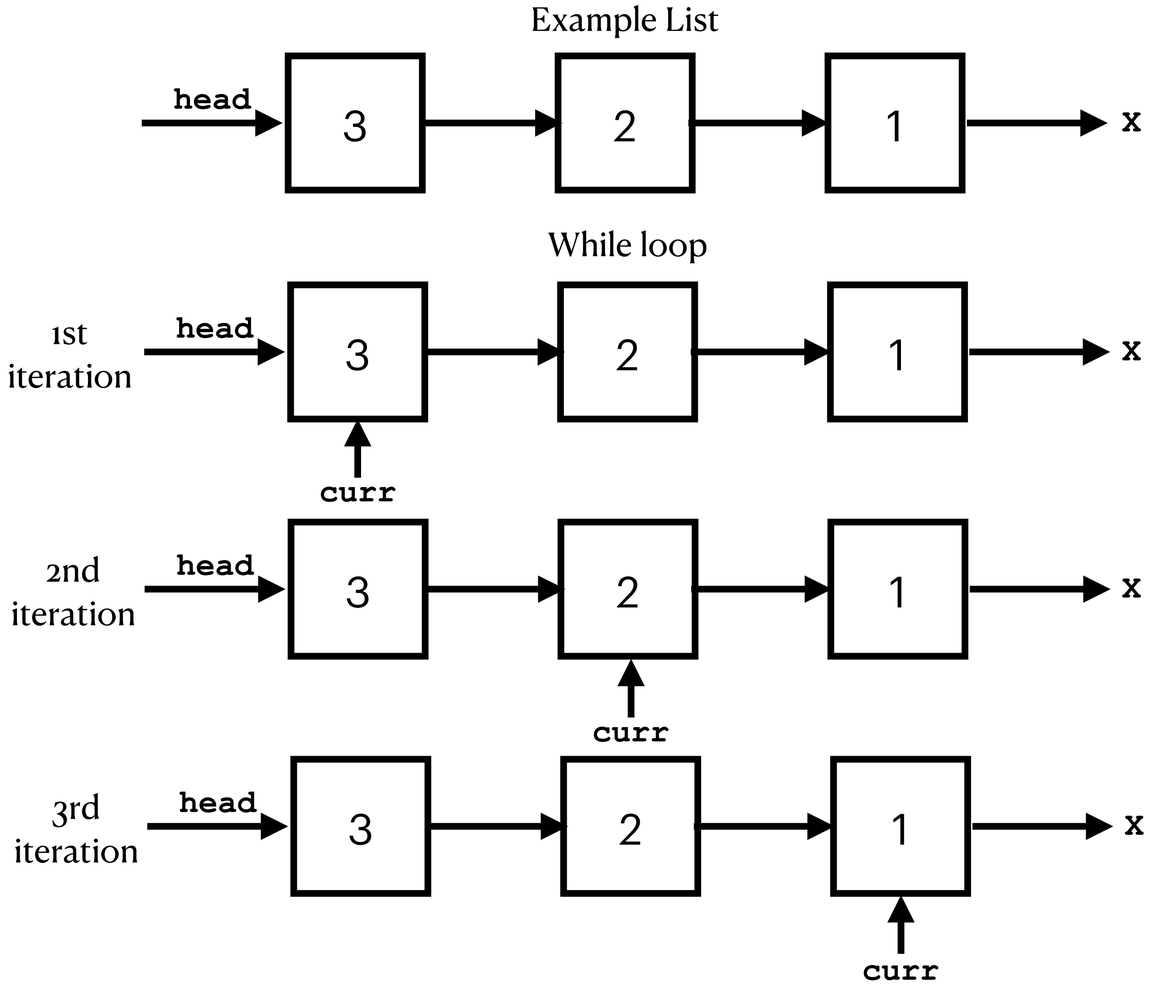
- We often want to traverse a linked list (go through all or some of its elements)
- We use a while loop

Initialise the variable we use to iterate over the list

```
struct node* curr = head;
while (curr != NULL) {
    printf("%d -> ", head->data);
    curr = curr->next;
}
printf("X\n");
```

Determine the loop condition

Determine step to take to get to the next iteration of the while loop



Linked Lists vs Arrays

Is one better than the other? It depends.

- We want to justify the need for inventing something new.
  - What's wrong with what we have?
  - Are there any limitations with linked lists that arrays don't have?

Array	Both	Linked Lists
• We are able to access any piece of data immediately using an index. • Memory efficient — we only need to store one thing at each index.	• We can store many variables that are associated with each other under one variable name. Through just one entity in our code, we can access multiple pieces of data.	• We can change the size of the list with ease. • Inserting at the start of the list is easy — no shifting of other elements is required.

Tips

How to Practice Linked Lists

- There are common programming patterns that arise when performing common operations on linked lists, such as creation, insertion, traversal of linked lists and deletion of nodes.
  - For example, one pattern we've seen is using a function to create nodes, instead of writing out the initialisation every time.
  - **Observe and learn** these patterns — don't try to reinvent the wheel and introduce unnecessary complexity (but do experiment to see why these operations cannot be simplified further!) This will allow you to offload these basic operations to muscle memory, and focus your attention on solving the problem at hand.
- Do and **repeat** lab exercises on linked lists after this week.
  - When you come back to the lab exercises, you'll likely find that you are able to simplify your solution and solve the problem more effectively!