Students:
Section 14.2 is a part of 1 assignment:
Programming Project 02 - Managing Dynamic Memory with C-structs (Building, Modifying, and Analyzing Dynamic Food Webs)

This assignment's due date has passed. Activity will still be recorded, but will not count towards this assignment (unless the due date is changed). See this article for more info.

Includes: 🟩 zyLab

Due: 09/26/2024, 11:59 PM CDT

Instructor created

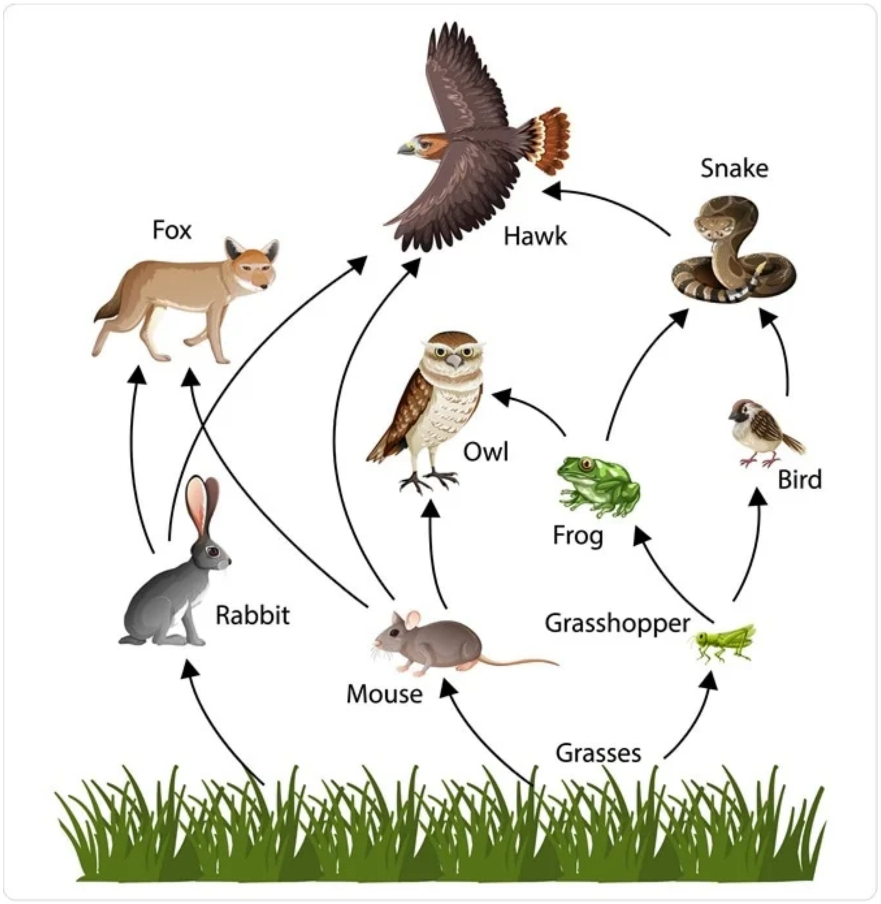
14.2 P02 - Dynamic Food Webs

Project 02 - Food Web Analysis with Dynamic Memory

Introduction

In this project, a food web is built from user-specified predator-prey relationships to build a dynamically allocated struct array of organisms, which is then used to identify the relationships between the organisms in the food web. Then, the user can modify the food web by adding new organisms (species expansion), adding new predator-prey relationships (food source supplementation), or removing organisms from the food web (species extinction). For these operations, the array of organisms AND the set of predator-prey relationships must both be rearranged and reallocated to either allow space for the new organism or remove all signs of the extinct species. At any stage, the user can choose to view food web characteristics, such as identifying the apex predators, the tastiest food source, the producers, the herbivores, the omnivores, and the carnivores. This requires a variety of food web analyses, which you will tackle one step at a time.

Figure 14.2.1: Sample Food Web



Organism Struct & Dynamic Arrays

For the purposes of this project, an organism is characterized by its struct subitems, as defined in the starter code as follows:

```
typedef struct Org_struct {
    char name[20];
    int* prey; //dynamic array of indices
    int numPrey;
} Org;
```

The **Org** struct contains the **name** of the organism as a character array (i.e. a *string*) and a dynamic int array representing its **prey** (i.e. *what it eats*) in the food web. The size of the **prey** array is stored as the **numPrey** subitem.

The organisms of the food web are stored in a dynamic array of **Org** structs, called **pWeb**, which has size **numOrgs**, which must be incremented or decremented as organisms are added or removed from the web.

The starter code includes a fully developed primary application in **main()**, which makes many calls to functions that need to be written. That is, you should NOT be developing code in **main()**; instead, start by reading through the primary application code in **main()** to get a full understanding of the different components of the program. In order to set you up for success in tackling the programming tasks to follow, the very first task of this project, i.e. Task #1, is to read, interpret, and fully-comment the provided primary application code in **main()**. Thus, the first task does not actually involve any coding at all!

The very beginning of **main()** is dedicated to the processing of command-line arguments, which set program mode parameters, specifically regarding whether *basic mode*, *debug mode*, and *quiet mode* are turned ON or OFF for each run of the program. More details are provided on the modes in the next section. Note that there is a call to the function **setModes()**, which is responsible for digging through the array of command-line arguments to set the mode parameters; writing **setModes()** is Task #2 of this project.

Once command-line arguments are processed, the Food Web Analysis Application formally begins by letting the user build an initial food web. The first step is to let the user enter an arbitrary number of organism names that will make up the initial food web. After each **name** is scanned in, a call to **addOrgToWeb()** is made in order to dynamically grow the food web to allow space for the additional organism; i.e. the **pWeb** array must be reallocated on the heap to allow for one additional **Org**. Throughout this project, reallocations must be done using only calls to **malloc()**, i.e. NO calls to **realloc()** are allowed. Writing **addOrgToWeb()** is Task #3 of this project.

To complete the initial food web, predator-prey relationships among the organisms are then read-in from user-input in the format

```
[predator index] [prey index]
```

where the indices refer to the **pWeb** array. Organisms are attributed a **name** in the **Org** struct definition, but this is primarily for display purposes only. For the bulk of the program, the individual organisms are uniquely identified by their **index** in the **pWeb** array, which always maintains the order in which the organisms were read-in from user-input.

The predator-prey relationships are added to the food web by calls to **addRelationToWeb()**. For each predator-prey relationship, the **prey index** should be appended to the predator's **prey** array. That is, the organism at **predator index** in **pWeb** must have its **prey[ ]** array reallocated to allow an additional entry, specifically **prey index**. Once again, the dynamic growth of the **prey** arrays must be done using only calls to **malloc()**. Writing **addRelationToWeb()** is Task #4 of this project.

At this point, the initial food web is built. It is helpful to consider an example before moving forward with our primary application...

Example Food Web

Consider the Grassland Food Web (Basic), which contains the following 7 organisms (indices precede the organism name):

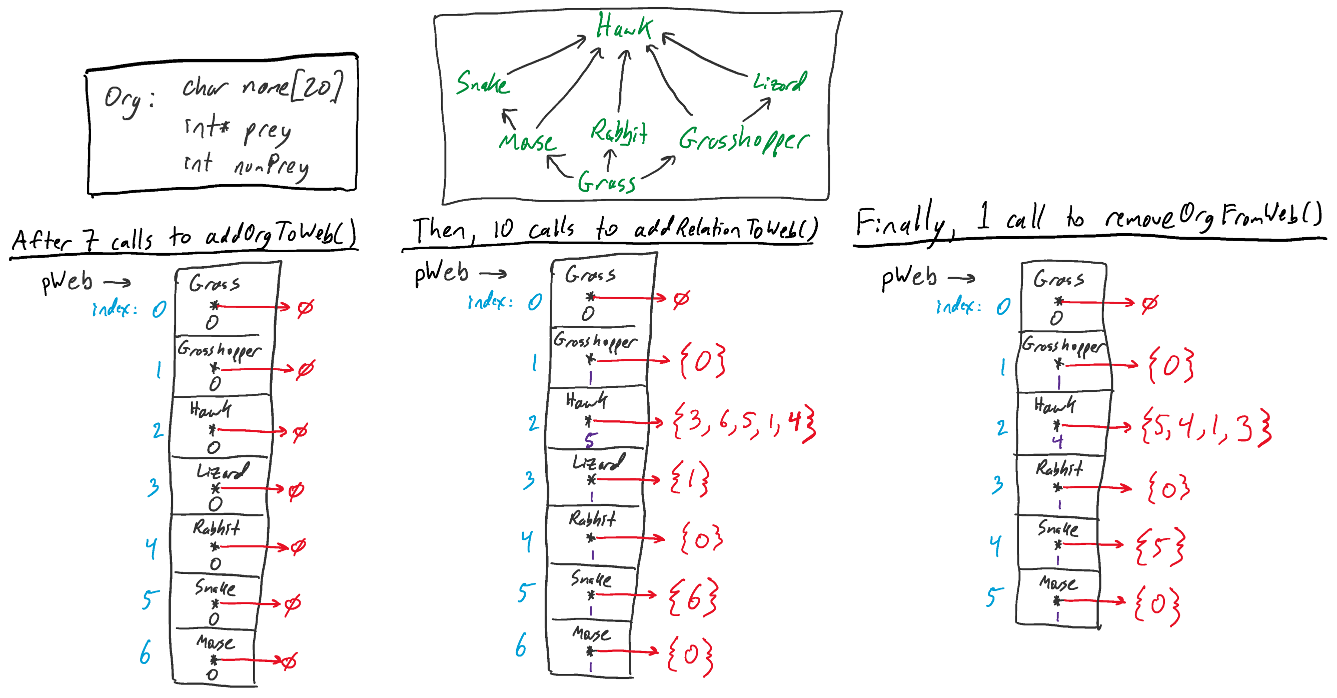
- 0: Grass
- 1: Grasshopper
- 2: Hawk
- 3: Lizard
- 4: Rabbit
- 5: Snake
- 6: Mouse

Now, consider the following 10 predator-prey relationships (with index relationship shown in parentheses):

- Mouse eats Grass (6 <- 0)
- Lizard eats Grasshopper (3 <- 1)
- Hawk eats Lizard (2 <- 3)
- Hawk eats Mouse (2 <- 6)
- Hawk eats Snake (2 <- 5)
- Rabbit eats Grass (4 <- 0)
- Hawk eats Grasshopper (2 <- 1)
- Snake eats Mouse (5 <- 6)
- Hawk eats Rabbit (2 <- 4)
- Grasshopper eats Grass (1 <- 0)

The figure below shows how the organisms are stored in a dynamically allocated **Org** struct array called **pWeb**, where each **Org** has a dynamically allocated int array called **prey**, which is the list of indices in **pWeb** for which that organism eats.

Figure 14.2.2: Sample Woodland Food Web and its Dynamic Org Arrays with Dynamic prey Arrays



After 7 calls to **addOrgToWeb()**, the **pWeb** food web is an array of **Org**s with size 7, where each **Org** has a **name**, and **prey** pointer initialized to **NULL**, and **numPrey** set to zero (since the **prey** array is empty). Also, notice that the order of the organisms in the **pWeb** array is determined by the order in which the organisms were added, where each addition is made to the back of the array.

Then, the 10 predator-prey relationships are set by calls to **addRelationToWeb()**. Each time a relationship is added, the prey index is added to the back of the predator's **prey** array, and the predator's **numPrey** subitem is incremented by one.

The food web on the far right of the figure shows the state of the **pWeb** array after a single call to **removeOrgFromWeb()**, which results in the extinction of the Lizard organism for this woodland food web. Notice that **pWeb** required a reallocation to reduce its total size, now that the total number of organisms in the web has decreased by one. Additionally, the organisms that had a larger index than Lizard (i.e. index 3) in the initial food web had their indices decrease by one, since those organisms were shifted up one in the newly allocated array. Furthermore, all instances of index 3 in any organism's prey array subitem from the initial food web must be removed, since the Lizard (who was at index 3) is now extinct. Thus, a reallocation of some prey arrays will likely be necessary. This only affects the Hawk in this example, since the Hawk was the only organism that had Lizard as prey. Lastly, we must keep all prey array elements synched to the indices of the **pWeb** array. Specifically, the prey array subitems are updated for all prey organisms that had their indices decrease by one (i.e.

Rabbit, Snake, and Mouse). For example, Snake preys on Mouse. In the initial food web, Snake at index 5 had index 6 in its prey array because index 6 of pWeb is Mouse. However, after the extinction event, Mouse's new index is 5, so Snake's prey array must be updated to store index 5. Similarly, Hawk initially had a prey array of {3, 6, 5, 1, 4}, which means it preys on {Lizard, Mouse, Snake, Grasshopper, and Rabbit}. After the extinction event, Hawk's new prey array is {5, 4, 1, 3}, which is a result of Lizard being removed from the web, three prey organisms decreasing their index by one (Mouse, Snake, and Rabbit), and one prey organism with no change (Grasshopper). Writing **removeOrgFromWeb()** is Task #7, and is likely the most challenging task, of this project.

With the three functions (**addOrgToWeb**, **addRelationToWeb**, and **removeOrgFromWeb**) working properly, the user will be able to modify the food web any way they choose. A full menu of options for modifying the food web is provided to the user in the second half of the **main()** primary application. The user can also select to display a full set of food web characteristics whenever they choose; an example food web analysis output for the initial Grassland Food Web (Basic) (from the figure above) is shown below:

```
Food Web Predators & Prey:
  (0) Grass
  (1) Grasshopper eats Grass
  (2) Hawk eats Lizard, Mouse, Snake, Grasshopper, Rabbit
  (3) Lizard eats Grasshopper
  (4) Rabbit eats Grass
  (5) Snake eats Mouse
  (6) Mouse eats Grass

Apex Predators:
  Hawk

Producers:
  Grass

Most Flexible Eaters:
  Hawk

Tastiest Food:
  Grass

Food Web Heights:
  Grass: 0
  Grasshopper: 1
  Hawk: 3
  Lizard: 2
  Rabbit: 1
  Snake: 2
  Mouse: 1

Vore Types:
  Producers:
    Grass
  Herbivores:
    Grasshopper
    Rabbit
    Mouse
  Omnivores:
  Carnivores:
    Hawk
    Lizard
    Snake
```

Starter Code & Input Files

The starter code contains the **Org** struct definition, function headers for many required functions, and a fully-developed **main()** primary application as described in the previous section. Additional helper functions are allowed and will likely be required for efficient modularization of your program. Look for the TODO statements in the starter code to align them with the tasks below. Most of the functions you are tasked with writing contain helpful comments regarding input/output variables and general functionality guidance.

All required **scanf()** statements are provided in the starter code, to ensure all user-input is read into the program in the proper order and format. This allows us to use set predefined input files that contain all of the user input for a given food web. Thus, compile the code and use a file to input values to the **scanf()** calls directly using a redirection operator (<) as follows:

```
gcc main.c –o prog.out
./prog.out < GrasslandFoodWebBasic.txt
```

In the IDE file tree, you will find the code file main.c and 24 predefined food web input files; there are actually only 6 food webs, but each food web has 4 unique sets of user-inputs that utilize different aspects of the program. The 6 food webs are Simple Food Web, Grassland Food Web, Aquatic Food Web, Terrestrial Food Web, Another Food Web, and Mixed Food Web. Each one of the 6 food webs has a Basic input file (sets up a food web and prints out the characteristics ONLY, with no modifications later), a AddOne input file (builds an initial food web AND adds one additional organism and some relationships afterward), a OneExtinction input file (builds an initial food web AND removes one organism afterward), and a ManyMods input file (builds an initial food web AND then makes multiple organism additions, relationship additions, and organism removals afterward).

You can use the aforementioned input files for testing as you develop code for each of the tasks below. It is a good idea to start with small food webs, where you can easily predict the expected behavior. Then, move to testing with larger food webs once the smaller food webs appear to be functioning as expected. Thus, to test the initial food web construction, you should begin testing using **SimpleFoodWebBasic.txt**, then use **GrasslandFoodWebBasic.txt**, **AquaticFoodWebBasic.txt**, etc. You can also use the AddOne input files at this stage. Later, once you have developed the **removeOrgFromWeb()** function, you should begin testing with **SimpleFoodWebOneExtinction.txt** and **SimpleFoodWebManyMods.txt**. If the output for those matches expectations, then move on to **GrasslandFoodWebOneExtinction.txt**, **GrasslandFoodWebManyMods.txt**, **AquaticFoodWebOneExtinction.txt, AquaticFoodWebManyMods.txt**, etc. All of these input files are used by the autograder, which is another reason for using these files as you develop your code. Do not solely use the autograder (i.e. the Submit button) for testing your functionality. Instead, test as you develop in the terminal by using the various input files details above. Once you are confident your code for a given task is functioning correctly based on your own testing, then submit to the autograder. It is a good idea to ensure all autograder test cases are passing for a given task before moving on the subsequent task.

Task #1 - read, interpret, and comment the primary application - main()

The code in **main()** represents the primary Food Web Application, which has been fully developed for you. Within the code, there are many calls to functions that you must write in the tasks that follow. For this first warm-up task, slowly read through the code in **main()**, interpreting how the application works, checking on how and where functions are called, and adding relevant comments to show your understanding of the program. No changes should be made to the code in **main()**, as it will be needed to pass the autograded test cases.You are welcome to add debugging print statements when necessary, but they should all be removed or commented out prior to submitting to the autograder. Once you are comfortable with the provided code in **main()** and the overall structure of **main.c**, it is now time to start developing your own pieces of code for this Food Web Application.

Task #2 - process command-line arguments - setModes()

There are three program modes that affect the user experience and can be set by command-line arguments. By default, the program is setup with **basicMode = FALSE**, **debugMode = FALSE**, and **quietMode = FALSE**. The program should accept the following command-line arguments:

- **-b** or **-basic** or **-basicmode** or **-b***: basicMode = TRUE, which prevents the user from making any modifications to the initial food web. That is, the user can build a food web, for which the food web analysis will be completed and displayed, and then the program terminates, without any options provided to the user to modify the food web. This mode should be turned ON for any command-line argument string begins with **-b** as the first two characters.
- **-d** or **-debug** or **-debugmode** or **-d***: debugMode = TRUE, which prints out the full web as it is being built or modified, after each step of adding an organism to the web, adding a relationship to the web, or removing an organism from the web. This mode should be turned ON for any command-line argument string begins with **-d** as the first two characters.
- **-q** or **-quiet** or **-quietmode** or **-q***: quietMode = TRUE, which suppresses the printed prompt messages before each user-input scan. This is particularly helpful when running the program in a non-interactive fashion by redirecting user-input to a file that contains the full set of predefined user-inputs. This mode should be turned ON for any command-line argument string begins with **-q** as the first two characters.

The command-line arguments are processed in the function **setModes()**. The program mode parameters should be updated using the passed-by-pointer function parameters **pBasicMode**, **pDebugMode**, and **pQuietMode**. The function has a Boolean return. If an invalid or duplicate command-line argument is present, then the function should return **false**. Otherwise, the function should return **true** after updating the relevant passed-by-pointer mode parameters. Here are some examples:

- Example 1: no arguments (**basicMode** = **FALSE**, **debugMode** = **FALSE**, **quiteMode** = **FALSE**, **returntrue**)

```
-> gcc main.c -o prog.out
-> ./prog.out
```

- Example 2: two arguments (**basicMode** = **TRUE**, **debugMode** = **TRUE**, **quiteMode** = **FALSE, return true**)

```
-> gcc main.c -o prog.out
-> ./prog.out -b -d
```

- Example 3: three arguments (**basicMode** = **TRUE**, **debugMode** = **TRUE**, **quiteMode** = **TRUE, return true**)

```
-> gcc main.c -o prog.out
-> ./prog.out -q -basic -debugMode
```

- Example 4: invalid argument (**return FALSE**)

```
-> gcc main.c -o prog.out
-> ./prog.out -d -a -q
```

- Example 5: invalid duplicate argument (**return FALSE**)

```
-> gcc main.c -o prog.out
-> ./prog.out -d -b -debug
```

- Example 6: invalid extra argument (**return FALSE**)

```
-> gcc main.c -o prog.out
-> ./prog.out -d -b -q -x
```

Note: the command-line arguments CAN be used in conjunction with input file redirection, for example:

```
-> ./prog.out -d -q < MixedFoodWebManyMods.txt
```

Task #3 - species extension - addOrgToWeb()

In order to build a food web, we must first collect all the organisms of the food web in an array. Your task is to write the **addOrgToWeb()** function, which adds a organism to the food web. In the starter code, **addOrgToWeb()** is called from **main()** for each organism **name** entered by the user. Here are some important considerations for this function:

1. **New memory allocation**: the food web may be an empty **Org** array if there are no organisms in the food web yet; in this case, the input ***ppWeb** should be **NULL**, and a new heap allocation to store only a single **Org** is necessary.
2. **Memory reallocation**: otherwise, a new heap allocation must be made to store all the Orgs in the input web PLUS one additional Org space for the new organism. Do NOT use realloc(); instead, the reallocation should be performed manually by malloc'ing a new array, copying the elements to the new array, freeing up the old array, and reassigning the array pointer to the new array.
3. **Copying a character array, i.e. a string**: the string copy function can be used to copy the new organism name (which is an input parameter to the function), to the food web. For example, **strcpy((*ppWeb)[0].name, newOrgName)** copies **newOrgName** to the

name subitem of the **Org** at index 0.

4. **Initialize other subitems for the new Org**: make sure to also set default initial values for the **prey** subitem (should be **NULL**) and **numPrey** (should be **0**).
5. **Append the new organism**: add the new organism to the back of the *ppWeb array.
6. **Update the size of the web**: update the ***pNumOrg** parameter appropriately (if an organism was added, the web got bigger)

Make sure the memory allocated for the web array is ALWAYS the exact (i.e. no extra wasted memory, but not too little) amount of memory needed to store the organisms; and that ***pNumOrg** ALWAYS accurately represents the size of the ***pWeb** array.

Your task is to write the **addOrgToWeb()** function, which takes in a *pointer to the **pWeb** array (i.e. a double pointer, **Org** ppWeb**)*, a *pointer to the number of organisms* (***pNumOrg***) in the web, and a character array (i.e. a *string*) for the name for the organism to remove due to extinction. The number of organisms is passed-by-pointer since it will be changed in the function and passed back to **main()**. Similarly, the **pWeb** array itself is passed-by-pointer because the array will need reallocation and will have its location changed inside the function. Since **pWeb** is an array, it is a pointer. When we pass an array name by pointer to a function, we are actually passing *a pointer to a pointer* (i.e. a *double pointer*): **Org** ppWeb.**

Task #4 - food source supplementation - addRelationToWeb()

Your next task is to write the **addRelationToWeb()** function, which adds a predator-prey relation to the food web. In the starter code, **addRelationToWeb()** is called from **main()** for each **[predator index] [prey index]** pair read in from user-input, representing a single predator-prey relationship. This function requires a few parts:

1. **New memory allocation**: if the predator's **prey[ ]** array is empty, allocate memory for one index;
2. **Memory reallocation**: otherwise, reallocate the predator's **prey[ ]** array to allow one more additional index. Do NOT use realloc(); instead, the reallocation should be performed manually by malloc'ing a new array, copying the elements to the new array, freeing up the old array, and reassigning the array pointer to the new array.
3. **Append the new prey**: append the prey index as the last element of the predator's **prey[ ]** array
4. **Update size of prey[ ] array**: update the **numPrey** subitem for the predator appropriately

Make sure the memory allocated for each prey[ ] array is ALWAYS the exact (i.e. no extra wasted memory, but not too little) amount of memory needed to store the prey indices for each organism; and that **numPrey** ALWAYS accurately represents the size of the prey[ ] array.

This function has a Boolean return. The function should return **true** if the new relation was successfully added to the food web and return **false** if any issue occurs that prevents the new relation from being added. For example, if either the predator or prey index argument is not a valid index of the **pWeb** array, then print "Invalid predator and/or prey index. No relation added to the food web." and immediately leave the function with a **false** return. As another example, if the argument relation is already a part of the web, then print "Duplicate predator/prey relation. No relation added to the food web." and immediately leave the function with a **false** return.

Task #5 - free the dynamic memory - freeWeb()

Whenever we work with dynamic memory allocations, we must be careful to avoid memory leaks. Dynamically allocated arrays used solely inside of functions must be freed before leaving each function. Even if you have been careful with freeing dynamic arrays isolated inside functions, it is almost a guarantee that you have a memory leak associated with **pWeb**, since the food web is built, modified, and analyzed across multiple functions. Before the program terminates, we must ensure the dynamic heap-allocated memory for our food web is freed. This is done at the very end of main() with a call to **freeWeb()**. Your task is to write **freeWeb()** by completely freeing ALL **prey** array subitems AND the **pWeb** array itself. The autograder includes a memory leak check labeled as Task #5.

It is a good idea to ensure your program has no memory leaks at this point by passing all autograded test cases through Task #5. However, the tasks that follow will also require diligence to the practice of proper memory record keeping and freeing. For example, you must be VERY careful with memory management in **removeOrgFromWeb()** (Task #7) to ensure all malloc'ed memory is freed. Specifically, you must free the memory of a **prey** array subitem BEFORE removing an **Org** element from **pWeb**. Thus, there is a second autograder test case that checks for memory leaks labeled as ALL Tasks, since it also checks for memory leaks associated with Tasks #6 and #7. Going forward, make sure to free all memory dynamically allocated to the heap to prevent potential memory leaks.

Task #6 - Food Web Analysis - printWeb() and displayAll()

Now that the initial food web is built correctly, it is time to perform the following analyses:

- print all organisms with a list of their prey
- identify apex predators
- identify producers

- identify the most flexible eaters
- identify the tastiest food
- calculate the height of each organism in the food web
- categorize each organism as a producer, herbivore, omnivore, or carnivore

Each analysis is described in the following subsections. Each step should be decomposed into separate function(s), like **printWeb()**, and called from the labeled locations within **displayAll()**. This entire task will involve properly accessing elements of the **pWeb** array and the **prey** array subitems.

REMINDER: whereas dynamically allocated arrays are declared using a pointer, e.g.

```
Org* web = (Org*)malloc(numOrgs*sizeof(Org));
```

the square bracket access/assignment operator can be used just like a traditional array, e.g.

```
scanf("%s",web[i].name); //assigns name for ith Org in web using user-input
```

Note: all printed displays with multiple organisms should be in the order they were first entered by user. The predator-prey relationships should also be printed in the order they were read in from the user.

Print Food Web Organism and Their Prey

To check if the web is built correctly with debugMode ON, the **printWeb()** function is called after each step of building the food web. Printing the food web with a call to **printWeb()** is also the first step of the full display of web characteristics inside **displayAll()**. Write the **printWeb()** function to print the web in the format consistent with this sample output (for GrasslandFoodWebBasic.txt):

```
Food Web Predators & Prey:
  (0) Grass
  (1) Grasshopper eats Grass
  (2) Hawk eats Lizard, Mouse, Snake, Grasshopper, Rabbit
  (3) Lizard eats Grasshopper
  (4) Rabbit eats Grass
  (5) Snake eats Mouse
  (6) Mouse eats Grass
```

Note that all organisms are printed in the order they were read-in from user-input and stored in the **pWeb** array. Organisms that eat no other organisms are simply printed, whereas organisms with at least one prey are printed with a formatted list of what they eat, again in the order that the predator-prey relation was entered by the user. In the parentheses before each organism is its index in the **pWeb** array.

Identify Apex Predators

An apex predator is an organism that is not the prey of any other organism in the food web. For the Grassland Food Web (Basic) example, the output is as follows:

```
Apex Predators:
  Hawk
```

There may be more than one apex predator, in which case print them in the order they were first entered by the user. For the Aquatic Food Web (Basic), the output is as follows:

```
Apex Predators:
  Bird
  Fish
  Lobster
```

Identify Producers

A producer is an organism that does not eat any other organism in the food web (typically, producers are plants and get their energy directly from the sun). For the Grassland Food Web (Basic) example, the output is as follows:

```
Producers:
  Grass
```

There may be more than one producer, in which case print them in the order they were first entered by the user. For the Aquatic Food Web (Basic), the output is as follows:

```
Producers:
  Phytoplankton
  Seaweed
```

Identify the Most Flexible Eaters

The most flexible eater is the organism(s) that eat the greatest number of different organisms in the food web. For the Grassland Food Web (Basic) example, the output is as follows:

```
Most Flexible Eaters:
  Hawk
```

There may be a tie for the most flexible eater, in which case print them in the order they were first entered by the user. For the Mixed Food Web (Basic), the output is as follows:

```
Most Flexible Eaters:
  Hawk
  Owl
```

Identify the Tastiest Food

The tastiest food is the organism(s) that gets eaten by the greatest number of different organisms in the food web. For the Grassland Food Web (Basic) example, the output is as follows:

```
Tastiest Food:
  Grass
```

There may be a tie for the tastiest food, in which case print them in the order they were first entered by the user. For the Aquatic Food Web (Basic), the output is as follows:

```
Tastiest Food:
  Limpets
  Mussels
```

Food Web Heights

The height of an organism in the food web is defined as the longest path from any of the producers up to that organism. Thus, all producers have height 0. Any organism that eats only producers has height 1. All other organisms in the food web have a height that is one more than the maximum height of all their prey. This is innately a recursive definition, and can be calculated using recursion. However, recursion is not required. Another fine approach is to use repeated iteration as follows:

1. set all heights to 0
2. assume changes to the heights need to be made
3. visit all organisms in **pWeb**
    - for each organism, set its height as one more than its maximum prey height
    - if no changes to the heights were made, then we are done; otherwise, redo step 3

For the Grassland Food Web (Basic) example, the output is as follows:

```
Food Web Heights:
  Grass: 0
  Grasshopper: 1
  Hawk: 3
  Lizard: 2
  Rabbit: 1
  Snake: 2
  Mouse: 1
```
Note: the organisms and their heights are printed in the order they were first entered by the user.

Categorize Organisms by Vore Type

Identify each organism in the food web as one of the following:

- **Producer**: eats no other organism in the food web; the assumption is that all producers are plants and all plants are producers.
- **Herbivore**: only eat producers (i.e. only plants)
- **Omnivore**: eats producers and non-producers (i.e. plants and animals)
- **Carnivore**: only eats non-producers (i.e. only animals)

For the Grassland Food Web (Basic) example, where there are no Omnivores, the output is as follows:

```
Vore Types:
  Producers:
    Grass
  Herbivores:
    Grasshopper
    Rabbit
    Mouse
  Omnivores:
  Carnivores:
    Hawk
    Lizard
    Snake
```
For the Aquatic Food Web (Basic) example, the output is as follows:

```
Vore Types:
  Producers:
    Phytoplankton
    Seaweed
  Herbivores:
    Limpets
    Zooplankton
  Omnivores:
    Mussels
  Carnivores:
    Bird
    Crab
    Fish
    Lobster
    Prawn
    Whelk
```

Task #7 - species extinction - removeOrgFromWeb()

Consider the situation where a species has gone extinct and must be removed from the food web. The organism flagged for removal is set using scanned user-input for the organism index that has gone extinct. Using the Grassland Food Web example, if the user enters 3 for the extinction index, then the Lizard must be removed from the food web. The figure above shows the updated **pWeb** array and updated **prey** array subitems for the extinction of the Lizard. Note the changes:

1. the **Org** struct with Lizard for a **name** is missing from **pWeb** and all **Org** elements after Lizard have been shifted forward one index; the size of the **pWeb** array is now one less
2. all organisms that had 3 (the initial index for Lizard) in their **prey** arrays (which is only Hawk in this example) now have one less element in their **prey** array; the index for the extinct organism is removed by shifting the remaining elements forward one;
3. additionally, all **prey** array elements greater than 3 must be decremented by one to account for the shift of organisms in the **pWeb** array; e.g. since Snake eats Mouse, the Snake's **prey** array used to store 6, which has been updated to 5 now that the Mouse has shifted forward one index in the **pWeb** array due to Lizard's extinction.

Remember that both the **pWeb** array and **prey** array subitems are dynamically allocated, so the removal of elements requires careful freeing of some memory and manual reallocation. Do NOT use realloc(); instead, the reallocation should be performed manually by malloc'ing a new array, copying the necessary elements to the new array, freeing up the old array, and reassigning the array pointer to the new array. **Edge case: if a food web only has one organism and it goes extinct, instead of malloc'ing an empty array, explicitly set the pointer to NULL; AND if a prey array becomes empty due to the extinction of its only prey, instead of malloc'ing an empty array, explicitly set the pointer to NULL.**

Your task is to write the **removeOrgFromWeb()** function, which takes in a *pointer to the **pWeb** array (i.e. a double pointer, **Org** ppWeb**)*, a *pointer to the number of organisms* (***pNumOrg***) in the web, and the ***index*** for the organism to remove due to extinction. Similar to Task #3, The number of organisms is passed-by-pointer since it will be changed in the function and passed back to **main()**. Similarly, the **pWeb** array itself is passed-by-pointer because the array will need reallocation and will have its location changed inside the function. Since **pWeb** is an array, it is a pointer. When we pass an array name by pointer to a function, we are actually passing *a pointer to a pointer* (i.e. a *double pointer*): **Org** ppWeb.**

The starter code includes some tips for the extinction function, which are reproduced here for convenience:

```
//      Remember to do the following:
//      1. remove organism at index from (*ppWeb)[] - DO NOT use realloc(), instead...
//          (a) free any malloc'd memory associated with organism at index; i.e. its prey
//          (b) malloc new space for the array with the new number of Orgs
//          (c) copy all but one of the old array elements to the new array,
//              some require shifting forward to overwrite the organism at index
//          (d) free the old array
//          (e) update the array pointer to the new array
//          (f) update numOrgs
//      2. remove index from all organisms' prey[] array subitems - DO NOT use realloc(),
//          (a) search for index in all organisms' prey[] arrays; when index is found:
//                [i] malloc new space for the array with the new number of ints
//               [ii] copy all but one of the old array elements to the new array,
//                    keeping the same order; some require shifting forward
//              [iii] free the old array
//               [iv] update the array pointer to the new array
//                [v] update the numPrey subitem accordingly
//          (b) update all organisms' prey[] elements that are greater than index,
//              which have been shifted forward in the web array
```

It is a good idea to harken back to Task #5, where the sole focus was ensuring proper memory management and that heap-allocated memory is freed when the memory is no longer in-user. You must be VERY careful with memory management in **removeOrgFromWeb()** to ensure all malloc'ed memory is freed. Specifically, you must free the memory of a **prey** array subitem BEFORE removing an **Org** element from **pWeb**. The autograder includes a test case to check for memory leaks for the full program output.

This function has a Boolean return. The function should return **true** if the organism was successfully removed from the food web and return **false** if any issue occurs that prevents the organism from being removed. For example, if either the organism index argument is not a valid index of the **pWeb** array, then print "Invalid extinction index. No organism removed from the food web." and immediately leave the function with a **false** return.

ALL Tasks - terminal testing, solution executable, and submitting to the autograder

Once the function is working properly and is fully tested (using your own code compilation/execution in the terminal as the primary testing approach and submission to the autograder as the last step), you can now run the full application (where the user can repeatedly choose how to build and modify their food webs) using input redirection and the wide variety of provided input files OR by creating your own food webs using interactive user-input.

At this point, you should begin to test your program for its full output, which comprises the last 25 test cases in the autograder suite. Once again, before jumping straight to submitting to the autograder, you are encouraged to compare your program output to a *fully-functioning* output by running the provided **demo.exe** in the starter file tree. You can run this executable to play around with building and modifying food webs using the provided input files and/or creating your own food webs. Note that program output does NOT need to exactly match the format of the the model solution output; i.e. the autograder does NOT check for exact matches in whitespaces. However, your program should produce nicely formatted displays.

To run **demo.exe**, you first must use the UNIX command **chmod** to change the permissions of the file to allow all users to execute it, as follows:

```
➜ chmod a+x demo.exe
➜ ./demo.exe
```

Additionally, **demo.exe** can be run with or without command-line arguments and with or without input redirection. Here are some valid ways to run **demo.exe**:

- interactive user-input to build a customized food web (i.e. no input redirection), but with debug mode ON:

```
➜ chmod a+x demo.exe
➜ ./demo.exe -d
```

- input redirection to build and modify the Terrestrial Food Web with Many Modifications, but all command-line modes OFF:

```
➜ chmod a+x demo.exe
➜ ./demo.exe < TerrestrialFoodWebManyMods.txt
```

- input redirection to build and modify the Grassland Food Web with One Extinction, and with debug mode AND quiet mode ON:

```
→ chmod a+x demo.exe
→ ./demo.exe -debug -quiet <
GrasslandFoodWebOneExtinction.txt
```

- input redirection to build and modify the Another Food Web with One Extinction, and with quiet mode ON:

```
→ chmod a+x demo.exe
→ ./demo.exe -q < AnotherFoodWebOneExtinction.txt
```

Upon running the final two example, notice how the Lizard's extinction has little impact on the Grassland Food Web organism heights, since it is near the top of the food web and the apex predator has many options. However, the extinction of the Spider has greater impact on Another Food Web organism heights, since it is closer to the bottom of the food web and is a key food source for many organisms. Do you see the identical behavior when you run your program???

Requirements

- The food web must be represented ONLY by the **Org** struct array as declared in main() of the starter code as **pWeb**. No copies of **pWeb** can be made, except during reallocation steps. Violations of this requirement will receive a manually graded deduction.
- Use the starter code as provided, adding code to complete functions and developing additional functions to complete the analysis steps, without making any structural changes: do NOT modify the **Org** struct definition (no name change, do not add subitems, do not remove subitems, do not modify subitem definitions, etc.), and do NOT change the function headers (no name changes, do not add parameters, do not remove parameters, do not modify parameter types, etc.). Violations of this requirement will receive a manually graded deduction.
- Solve each task and the program at large as intended, i.e. build the food web using the **Org** struct array called **pWeb** and fill out the **prey** array subitems with indices of the **pWeb** array; use this food web data structure to complete all food analysis steps. Additional arrays (static or dynamic) are allowed, as long as they don't represent the same data that is (or should be) stored in **pWeb**. Food web analysis can be done solely using indices, such that string analysis is not necessary (beyond one use of **strcpy** in

addOrgToWeb). Any analysis of C-arrays should be done directly on the characters; i.e. no string library functions should be used. Violations of this requirement will receive a manually graded deduction.

- The **pWeb** array and **prey** array subitems must be dynamically allocated to ALWAYS have the exact amount of memory needed. Both the **pWeb** array and the **prey** array subitems begin empty. As organisms and predator-prey relationships are added, the **pWeb** array and individual **prey** array subitems should be reallocated for each additional element. **Do NOT use realloc( )**; instead, manually malloc space for a new array with modified size, copy over necessary data, free the old array, update the array pointer, and update the size variable. During **removeOrgFromWeb**, both the **pWeb** array and **prey** array subitems must be reallocated, when necessary, to utilize the exact amount of memory needed to store the food web data after the species goes extinct. **Edge case: instead of malloc'ing an empty array, explicitly set the pointer to NULL.** Violations of this requirement will receive a manually graded deduction.
- All dynamically allocated memory must be freed to prevent possible memory leaks. Memory leak and array-out-of-bounds issues are checked by the autograder but may also receive a manually graded deduction.
- All input ***FoodWeb***.txt files have the organisms and predator-prey relationship are in a random order. When multiple organisms are printed, make sure the print order matches the order that the organisms and/or the predator-prey relationships were read-in from user-input. This is managed naturally by using arrays and then printing from index 0 working up in index. During the extinction of a species, make sure to shift remaining elements forward to remove the organism (i.e. do not get tricky by using a swap, just stick with shifting elements forward in the array to replace it); this keeps the order as read-in from user-input.
- Coding style issues are manually graded using deductions, worth up to 25% of the total project score. Style points are graded for following the course standards and best practices laid out in the syllabus, including a header comments, meaningful identifier names, effective comments, code layout, functional decomposition, and appropriate data and control structures.
- Programming projects must be completed and submitted individually. Sharing of code between students in any fashion is not allowed. Use of any support outside of course-approved resources is not allowed, and is considered academic misconduct. Examples of what is allowed: referencing the zyBook, getting support from a TA, general discussions about concepts on piazza, asking questions during lab or lecture, etc. Examples of what is NOT allowed: asking a friend or family member for help, using a "tutor" or posting/checking "tutoring" websites (e.g. Chegg), copy/pasting portions of the project description or starter code to an AI chatbot, etc. Check the syllabus for Academic Integrity policies. Violations of this requirement will receive a manually graded deduction and will be reported to the Dean of Students office.

Key Tips for Success with this Programming Project

- **Diagramming is key for the trickiest task involving careful memory management -** the programming tasks begin with simple processing of command-line arguments, building a dynamic food web structure from user-input, and a variety of statistic calculations. Most of the early tasks are extra practice with fundamental programming concepts in C. However, the trickiest task of the program is the extinction step, which requires working with structs and multiple manual reallocations of heap-allocated arrays. Thus, it is important that you understand all the ways in which *extinction* affects a food web from the descriptions above AND have sketched a diagram of the extinction process for a sample food web, before attempting to implement the extinction code. The species expansion (i.e. **addOrgToWeb**) and food source supplementation (i.e. **addRelationToWeb**) tasks should be good practice for the species extinction (i.e. **removeOrgFromWeb**) task, which is the trickiest aspect of the project.
- **NOT all tasks/functions are created equal -** the *Programming Tasks* for this project are primarily to develop functions to support the primary application in **main()**, all of which is provided for you in the starter code. The level of scaffolding is designed to lead you through the program development that leaves room for creatively applying course concepts and tools. A natural consequence is that..
  - some Tasks are straightforward and some Tasks are challenging;
  - some Tasks ask you to do one thing specifically and some Tasks are purposefully open-end with multiple components;
  - some Tasks test your ability to follow instructions verbatim and some Tasks require you to practice problem-solving and critical-thinking skills;
  - some Tasks may only take you a few minutes and some Tasks may require an initial attempt followed by a break to do something else only to come back multiple times to tackle the challenge;
- **Continual Testing -** you should continue building the best-practice habit of regular testing functionality as you develop code. There is no formal required structure to the testing you do, but continual testing in small chunks of code is essential for efficient code development. Testing should begin by compiling and running the program in the terminal with a set of simple interactive user inputs OR using input redirection with the Simple Food Web. Effective testing strategies also include comparing your output to the fully-functioning output provide with **demo.exe**. Submitting to the autograder should be the last step in testing your implementation for any given task. The autograder test cases are labeled for each Task. You are strongly recommended to pass all test cases for each task before moving to the next task.
- **No superfluous print statements -** The vast majority of console output is handled in the primary application in **main()**, all of which is already provided in the starter code. Of

course, as you develop, test, and debug your code, you may introduce some print statements to achieve those purposes. These added print statements must be removed and/or commented out before you submit to the autograder. In the event that you experience issues with the zyLab IDE crashing or not loading due to extraneous output in your program (which should not be a problem since the starter code output is limited) we cannot provide support and you will need to contact the zyBooks team for assistance.

- **Gradescope submission -** at the bottom of this description there are instructions for submitting your work (code file main.c) to Gradescope. Your project grade will be determined using the zyLab autograder score AND your Gradescope submission. You must do BOTH to earn credit for you work. Your project submission timestamp is formally determined from the Gradescope submission time.

Submission & Grading

Develop your program in the IDE below. Do NOT use the "Run" button; instead, compile your code (e.g. "**gcc main.c**") and run the resulting executable (e.g. "**./a.out -q < SimpleFoodWebBasic.txt**") in the terminal to test your code interactively as you develop your program. Use the "Submit for grading" button to test your code against the suite of autograded test cases, which also submits your program for functionality grading.

Formally, you must submit your final program (**main.c**) to Gradescope, where your code will be manually inspected for style (details in the syllabus) and project requirements (details listed above). The official code submission is the version you submit to Gradescope (look for *Project02*).

The standard deadline for the project is **Thursday, September, 26th**. As detailed in the syllabus, early submissions receive +1 extra credit point/day early, up to a maximum of +3 points. Also, students are allocated a certain number of late days throughout the term, where no late penalty is applied. Students can use their late days whenever they choose, simply by sumbitting their work to Gradescope after the deadline. However, no more than two late days are allowed for each project.

The autograder has a test case suite to check functionality with the following breakdown:

- Task 1 - comments in main(): **0 points (manually inspected after submission)**
- Task 2 - command-line arguments - setModes(): **10 points**
- Task 3 - species expansion - addOrgToWeb(): **10 points**
- Task 4 - food source supplementation - addRelationToWeb(): **10 points**

- Task 5 - memory management & freeing dynamic memory - freeWeb(): **10 points**
- Task 6 - food web analysis - printWeb() & displayAll(): **21 points**
- Task 7 - species extinction - removeOrgFromWeb: **10 points**
- ALL Tasks - full program output for a wide variety of input food web files: **24 points**
- ALL Tasks - extension of Task 5 - memory management for full program output: **5 points**

Assignment Inspiration

The overall idea for this project, food web analysis steps, and the food web data was greatly inspired by Ben Stephenson and Jonathan Hudson from the University of Calgary.

LAB ACTIVITY | 14.2.1: Food Web Analysis with Dynamic Memory

100 / 100

Run History Tutorial

main.c

```
/*--------------------------------------------
Program 2: Food Web Analysis
Course: CS 211, Fall 2024, UIC
Author: Shreya Ganguly
--------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

typedef struct Org_struct {
```

```c
typedef struct org_struct {
    char name[20]; // organism name
    int* prey; // dynamic array of indices
    int numPrey; // number of preys
} Org;

// Function definitions
void printApex(Org* pWeb, int numOrg);
void printProducer(Org* pWeb, int numOrg);
```

DESKTOP CONSOLE

Submit for grading

Coding trail of your work [What is this?]

```
9/22 U10 M10,10,10,10,10,20,0,20,20,20,20,20,20,20,20
,20,20,30,45,48,48,48,48,48,51,45,45,51,0,51,51 T20,35
,35,57,57,0,57,57,0,57,57,0,42,0,57,57,57,57,57,57,60
,42,60,0,0,60,60,60,0,60,0,60 R63,63,63,45,45,63,42
,63,63,63,73,73,0,68,68,68,68,68,0,0,73,0,68,68,68,0
,68,73,68,0,0,40,68,68,69,69,69,70,0,0,70,70,80,90,90
,90,90,73,100,100 min:324
```

Latest submission - 9:44 PM CDT on 09/26/24 | Submission passed all tests ✓ | Total score: 100 / 100

✓ Only show failing tests (40 tests hidden) **Open submission's code**

5 previous submissions

5 previous submissions

9:23 PM on 9/26/24	100 / 100	View
9:22 PM on 9/26/24	73 / 100	View
9:13 PM on 9/26/24	90 / 100	View
9:07 PM on 9/26/24	90 / 100	View
9:02 PM on 9/26/24	90 / 100	View

Trouble with lab?

Feedback?

Activity summary for assignment: Programming Project 02 - Managing Dynamic Memory with C-structs (Building, Modifying, and Analyzing Dynamic Food Webs)

100 / 100 points

Due: 09/26/2024, 11:59 PM CDT

This assignment's due date has passed. Activity will still be recorded, but will not count towards this assignment (unless the due date is changed). See this article for more info.

Completion details