CS 261-020
Data Structures

Lecture 3
C Basics
Dynamic Array vs. Linked List
1/11/22, Tuesday
Odds and Ends

• My office hours posted

• Recitation 2 posted

• Assignment 1 Due Sunday midnight
Lecture Topics:

• C Basics
• Dynamic Array
Recap: C Basics – Pointers

• A pointer is a variable whose value is a memory address
• Every pointer points data of a specific data type
  • Ex.,

```c
int var = 20; //address of var: 0xffff0
int *p1 = &var; //address of p1: 0xffec
int **p2 = &p1; //address of p2: 0xffe4
```

What prints 20?
What prints 0xffff0?
What prints 0xffec?
What prints 0xffe4?
Recap: C Basics – Void Pointers (void*)

- A void pointer is a pointer represented by the type `void*`
- A void pointer is a generic pointer, it can point to **data of any data type**.
  - E.g., a void pointer points to an integer
    ```c
    int var = 20;
    void *v_ptr = &var;
    ```

- Why `void*`?
  - It allows the data structures to contain data of any type while remaining **type agnostic**
Recap: C Basics – malloc()

• Allocating memory on the heap
  • In C++: use `new` operator
  • In C: use `malloc()` ← requires `#include <stdlib.h>`

• `malloc()`:
  • Allocates a contiguous block of memory
  • Arguments: number of bytes
  • Return: `void*`
  
  ```
  void * allocated_memory = malloc(NUMBER_OF_BYTES);
  ```
C Basics – Free dynamic memory

• We have to **manually free** memory allocated on the heap
  • otherwise → memory leak!

• How?
  • In C++, we use `delete`
  • In C, we use `free()`
  • E.g.,
    ```c
    int* array = malloc(1000 * sizeof(int));
    ...
    free(array);
    array = NULL;
    ```

• Rule of thumb: For every call to `malloc()` you should have a corresponding call to `free()`
C Basics – valgrind

• Use valgrind to check if your program has memory issues:
  • valgrind ./prog [cmd_line args]

• To dig deeper into where memory was lost, pass the --leak-check=full:
  • valgrind --leak-check=full ./prog [args]

• Demo ...
C Basics – strings in C

• Unlike C++, there is no string objects in C
  • Thus, no `std::string` class

• Strings are represented in C as arrays of char values, i.e., `char*` type

• How do C strings indicate the end of the string?
  • Use a special character – null character (`\0`)
  • Thus, C strings also called null terminated strings
  • For example, the string “hello” would look like this in memory in C:

```
'h' 'e' 'l' 'l' 'o' \0
```
← array of 6 characters
C Basics – strings in C (cont. )

• The null character is important ➔ indicates the end of the string

• Functions rely on ‘\0’:  
  • printf() – know when to stop processing the string  
  • strlen() – returns the number of characters in a string  
    • Count until it finds a null character

• Allocating memory to store a string: num of char + null char  
  • Q: How many char can we store in the str?  
    char* str = malloc(64 * sizeof(char));
      63
      +1 for ‘\0’
C Basics – strings in C (cont.)

• Constant strings in C:
  
  ```c
  char* name = "Harry Potter";
  ```

• Constant strings are read-only, thus cannot be modified.
  
  ```c
  name[0] = 'l'; //illegal but no error message
  ```

• Best to mark it be constant
  
  ```c
  const char* name = "Harry Potter";
  name[0] = 'l'; //illegal with compiling error
  ```
C Basics – strings in C (cont.)

• Useful functions for C strings: \#include <string.h>
  • strlen() – returns the number of characters in the string
  • strncpy() – copy a specified number of characters from one string to another
  • snprintf() – “printing” content into a string, up to a specified number of characters
    • From <stdio.h>
  • strcmp() – compare two strings, returns 0 if they are equal

• And many more... check string.h
C Basics – Function pointers

• Function pointers allows us to store the memory address of a function in a variable and use that memory address to call the function being pointed to
  • Allows us to pass functions as arguments to other functions

• Why would we want to pass a function as an argument to another function?

• Consider this...
C Basics – Function pointers

• Write a function to sort an array
• To make it work for any type,
  • each element is void*, and thus the pointer to the array is void**

```c
void sort(void** arr, int n); //can pass arr of any type into this function
```

• Question: How does sort() be able to compare the values in the array?
  • Use function pointers!
  • The function calling our sort() and passing data into it does know these things
C Basics – Function pointers

• Add a function pointer:
  • A function that compares two values from the array to be sorted and return a value indicating which is bigger/smaller
  • int cmp(void* a, void* b);

• So our sort() becomes to:
  void sort(void** arr, int n, int (*cmp)(void* a, void* b));
C Basics – Function pointers

• To use this function pointer
  • the calling function will need access to a function for *comparing* elements, i.e., integers
  • This function will have to match the prototype of the function pointer argument to our sort()
  • E.g.,

```c
int compare_ints(void* a, void* b) {
    int* ai = a, *bi = b; /* Cast void* back to int*. */
    if (*ai < *bi)
        return 0;
    else
        return 1;
}
```

• Function call will be:

```c
sort((void**)array_of_ints, number_of_ints, compare_ints);
```
C Basics – Function pointers

void sort(void** arr, int n, int (*cmp)(void* a, void* b));

• Within sort():
  • Whenever we need to compare two values from the array being sorted, we can just call cmp()

    if (cmp(arr[i], arr[j]) == 0) {
      /* Put arr[i] before arr[j] in the sorted array. */
    }
    else {
      /* Put arr[i] after arr[j] in the sorted array. */
    }

• Demo....
Lecture Topics:

• C Basics
• Dynamic Array
Abstract Data Type (ADT)

• Abstract Data Type (ADT) – a mathematical model for data types
• Specifies:
  • the type of data stored
  • the operations supported on them
  • the types of parameters of the operations.

• Why “abstract”?
  • an implementation-independent view of the data type
Dynamic Arrays

• Elements in an array are stored in a contiguous block of memory
• Allow random access (direct access)
  • i.e., time to access the 1st element = time to access the last element
  • By using array subscript ([]):
    ```c
    int* array = malloc(1000 * sizeof(int));
    array[0] = 0;
    array[999] = 0;
    ```

• Demo...
Dynamic Arrays (cont. )

• Basic operations:
  • get – Gets the value of the element stored at a given index in the array
  • set – Sets/updates the value of the element stored at a given index in the array
  • insert – Inserts a new value into the array at a given index.
    • Sometimes, dynamic array implementations limit insertion to a specific location in the array, e.g. only at the end.
  • remove – Removes an element at a given index from the array
    • Sometimes, dynamic array implementations avoid moving elements up a spot by only allowing the last element to be removed
Dynamic Arrays (cont.)

- **Drawbacks:**
  - Fixed size, must be specified when the array is created
    - For static array:
      ```c
      int array[50];
      ```
    - For dynamic array:
      ```c
      int *array = malloc (50 * sizeof(int));
      ```
  
  - Need to allocate more memory if we need to store more data
    - How?

- **Dynamic array DS doesn’t have a fixed capacity**
  - Has a variable size and can grow as needed
Dynamic Arrays (cont.)

• Need to keep track of three things:
  • data – underlying data storage array
  • size – number of elements currently stored in the array
  • capacity – number of elements data has space for before it must be resized

• How it works?
  • An array of known capacity is maintained by the dynamic array DS.
  • As elements are inserted, they are simply stored in data
  • If an element is inserted into the dynamic array, and there isn’t capacity for it in the underlying data storage array (data), the capacity of the underlying data storage array is doubled. Then the new element is inserted into this larger data storage array.
Dynamic Arrays

5 8
5 8 1
5 8 1 4
5 8 1 4 9
5 8 1 4 9 0
5 8 1 4 9 0 6
5 8 1 4 9 0 6 7
5 8 4 9 0 6 7
5 8 4 9 6 7
Inserting an element into dynarray

- **Case 1: if size < capacity**
  - At least one free spot in data
  - Insert the new element

- **Case 2: if size == capacity**
  - No free spot in data
  - Step 1: allocate a new array that has twice the capacity
  - Step 2: copy all elements from data to new array
  - Step 3: delete the old data array
  - Step 4: Insert the new element
Next Lecture:

• Linked List
• Complexity Analysis
  • Big O