**Lecture 27 - Chapter 12: C Data Structures – Mon Nov 27 or Tues Nov 28**

**Announcements**

Reading:

* Chapter 12

Assignments:

* Assign: Assignment #11 - due on **Dec 4** (MW class) or **Dec 5** (TR class) (no late assignments accepted)

**Today’s Goals**

1. Self-Referential Structures
2. Dynamic Memory Allocation
3. Linked Lists

**Today’s Terminology**

**Terminology**

* Self-Referential Structure
  + When a structure contains as a member that is a pointer to its same structure type
* Dynamic Data Structures
  + A data structure that can grow and shrink during execution time
* Dynamic Memory Allocation
  + Obtaining and releasing memory during execution time
* Linked Lists
  + Linear collection of self-referential structures!
  + Insertions and deletions are made anywhere in the list
* Stacks
  + Linear data structure - Last in First Out (LIFO)
  + Insertions and deletions are made only at the top of the stack
* Queues
  + Linear data structure - First in First Out (FIFO)
  + Insertions are made at the back (tail)
  + Deletions are made at the front (head)
  + Represents a waiting line
* Binary Trees
  + Nonlinear, two-dimensional data structure

**Self-Referential Structures**

**Structure Review**

* In chapter 10 we looked at structures:

**struct** student {

**char** name[50]; Remember this:

**double** grades[3]; 1) Did **not** reserve memory

}; 2) Only created a new data type!

**int** **main**(**void**) {

**struct** student csStudent1 = {"Ken Roczen", {65, 92, 72}};

**puts** ("CS Student1 Student");

**printf** ("Name = %s\n", csStudent1.name);

**for** (**int** k = 0; k < 3; k++) {

**printf** ("Grades[%d] = %.2f\n", k, csStudent1.grades[k]);

}

}

**Displays:**

CS Student1 Student

Name = Ken Roczen

Grades[0] = 65.00

Grades[1] = 92.00

Grades[2] = 72.00

**Self-Referential Structure**

* When a structure member is a pointer to its same structure type

**struct** student {

**char** name[50];

**double** grades[3];

**struct** student \*nextPtr;

};

* Visually
  + The structure should be viewed as two sections:
    - The 1st for the information
    - The 2nd for the pointer to another structure (the link!)

**1st (info) 2nd (next structure)**

|  |  |
| --- | --- |
| name  grades | nextPtr |

**Example Create students (structures)**

**struct** student student1 = {"Ken Roczen", {65, 92, 72}, NULL};

**struct** student student2 = {"Ryan Dungey", {98, 100, 78}, NULL};

**info next info next**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| name = Ken Roczen  grades = 65 92 72 | NULL |  |  | name = Ryan Dungey  grades = 98 100 78 | Null |

**Link the 2 structures together**

student1.nextPtr = &student2; **using the pointers**

**info next info next**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| name = Ken Roczen  grades = 65 92 72 |  |  |  | name = Ryan Dungey  grades = 98 100 78 | Null |

Show how structures are linked

together. Print each student’s

name using only the student1 variable!

**printf** ("Student 1 = %s\n", student1.name);

**printf** ("Student 2 = %s\n", student1.nextPtr->name);

**Displays**

Student 1 = Ken Roczen

Student 2 = Ryan Dungey

**Notes**

* Self-referential structures are used to construct data structures such as:
  + Linked Lists
  + Stacks
  + Queues
  + Trees

**Dynamic Memory Allocation**

**Static Data Structures**

* When the size **does not** change during execution

**int** numberList [50]; Container for 50 integer values

**struct** student {

**char** name[50];

**double** grades[3];

};

**int** **main**(**void**) { Container for array of 50

**struct** student student1 = {"Ken Roczen", {65, 92, 72}}; characters (string) and

} another array of 3 doubles

* Memory for static variables is allocated from the **stack**

**Dynamic Data Structures**

* When the size changes during execution
* Requires dynamic memory allocation
  + Obtaining and releasing memory during program execution
* Memory for dynamic data structures is allocate from the **heap**

**Dynamic Memory Allocation**

* Memory allocation functions in <stdlib.h>
* Functions that give us the ability to dynamically allocation and free memory:
  + **Malloc (memory allocation)**
    - Allocates specified amount of memory on the heap and returns pointer to that memory
    - Like **new operator** in Java, C++
    - May or may not initialize memory
    - General form:

**void\* malloc(size\_t size);**

Returns pointer to Amount of memory

memory of type void to allocate

Returns NULL if fails!

**Remember: a void pointer can be assigned to a variable of any pointer type**

* + - Example:
      * Allocate memory for 400 bytes of memory and **intPtr** points to 1st byte

**int** \*intPtr; Don’t need to typecast in C

intPtr = **malloc** (100 \* **sizeof**(**int**));

In C++ you need typecast!

intPtr = (**int** \*) **malloc** (100 \* **sizeof**(**int**));

* + - * Allocate memory for one student structure

Remember, this **does not** allocate

memory, it defines a new data type

**struct** student {

**char** name[50];

**double** grades[3];

};

This code allocates memory ***statically***!

**int** **main**(**void**) {

**struct** student student1 = {"Ken Roczen", {65, 92, 72}};

This code allocates memory ***dynamically***!

1st sizeof struct student is determined

2nd that number of bytes is allocates

3rd address of memory assigned to ptr

4th check if allocation was successful!

**struct** student \*studentPtr = **malloc**(**sizeof**(**struct** student));

**if** (studentPtr != NULL) {

**printf** ("Address stored in studentPtr = %p\n", studentPtr);

}

}

* + **Free (memory deallocation)**
    - Deallocates previously **allocated memory by malloc**
    - Always free memory when no longer used - prevents memory leaks!
    - General form:

**void free (void \*ptr);**

Pointer to block of memory allocated

with malloc to be freed

* + - Example
      * When you free memory it is good practice to set the pointer to NULL

**free** (intPtr);

intPtr = NULL;

* There are also these two functions (chapter 14)
  + calloc and realloc
    - Used with **dynamic arrays**
  + calloc
    - Allocates a specified amount of memory for an array
    - Initializes the array elements to zeros
    - Returns pointer to memory (NULL is could not allocate memory)
  + realloc
    - Changes the size previously allocated memory using malloc or calloc

**Linked Lists**

**Array Review**

* With an array, once the size is set it is fixed
* When don’t know how big the array needs to be, we waste memory and make it extra big
* Memory is consecutive
* Accessing elements is easy through index notation
* Inserting new elements at the front of the array can be expensive
* Deleting elements also caused problems
* Array could become full

**Linked Lists**

* Collection of linked structures
* Dynamic data structure
  + No need to know size **before** execution time
  + No need to waste memory
  + Grows and shrinks dynamically during program execution
* Memory is general not consecutive
  + Blocks (i.e. nodes) are allocated and linked together with pointers
  + Not as easy to access as an array
* Can insert values anywhere into the list easily
  + Easier to keep list in sorted order!
* Can delete items from anywhere in the list easily
* Number of nodes only limited by amount of memory

**Singly Linked List**

* Create **nodes** Using self-referential structures
* Each node contains two fields
  + Data field
    - The information the list holds
  + Next field
    - A pointer to the next node in the list
* Nodes are allocated when needed on the heap with malloc

Pointer to 1st node

of the list (head)

Data Next Data Next Data Next

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **headPtr** |  |  |  |  |  |  |  |  |  |  | Null |

Each node stores the **data element** and **pointer** to next node Last node’s next pointer is set to NUL

Indicates end of list!

**Doubly Linked List**

* Each node contains three fields
  + Data field
    - The information the list holds
  + Previous field
    - A pointer to the previous node in the list
    - First node’s previous pointer is NULL
  + Next field
    - A pointer to the next node in the list
    - Last node’s next pointer is NULL

Pointer to 1st node

of the list (head)

Previous Data Next Previous Data Next Previous Data Next

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **headPtr** |  | NULL |  |  |  |  |  |  |  |  |  | Null |

Each node stores the **data element** and **pointer** to **previous node** and **next node**

**Singly Linked List Examples**

* Need a pointer to first node – called the **head** pointer
* Each node is created as needed
* Will do the following **Going step by step**
  + Insert **Tie everything together at end**
  + Delete **Covering steps books doesn’t**

**Insert Example #1:**

* Create a simple linked list with numbers => {2, 4, 6} Does easiest case 1st, insert at end of list
* Final result will look like this:

**Data Next Data Next Data Next**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **headPtr** |  | 2 |  |  | 4 |  |  | 6 |  |  | Null |

**struct** node { Define a new type for the node

**int** data;

**struct** node \*nextPtr;

};

**int** **main**(**void**) {

// Build a simple list -> {1, 2, 3}

// Create two node pointer variables

**struct** node \*headPtr = NULL;

**struct** node \*nodePtr = NULL;

Allocate memory for 1st node(heap)

nodePtr = **malloc**(**sizeof**(**struct** node)); Setup the 1st node

**if** (nodePtr != NULL) {

nodePtr->data = 1;

nodePtr->nextPtr = NULL;

}

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| nodePtr |  | 2 |  |  | NULL |

// Set the head to point to this first node

headPtr = nodePtr;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **headPtr**  nodePtr |  | 2 |  |  | NULL |

// Allocate memory for another node in the heap

nodePtr = **malloc**(**sizeof**(**struct** node));

**if** (nodePtr != NULL) {

nodePtr->data = 2;

nodePtr->nextPtr = NULL;

}

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| nodePtr |  | 4 |  |  | NULL |

// Connect the two nodes

headPtr->nextPtr = nodePtr;

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| headPtr |  | 2 |  |  | 4 |  |  | NULL |

// Allocate memory for another node in the heap

nodePtr = **malloc**(**sizeof**(**struct** node));

**if** (nodePtr != NULL) {

nodePtr->data = 3;

nodePtr->nextPtr = NULL;

}

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| nodePtr |  | 6 |  |  | NULL |

// Connect the nodes 2 and 3 Set 2nd node’s nextPtr to point

headPtr->nextPtr->nextPtr = nodePtr; to 3rd node

**headPtr**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 2 |  |  | 4 |  |  | 6 |  |  | NULL |

// Walk list and print values

**printf** ("The list is: ");

currentPtr = headPtr;

**while** (currentPtr != NULL) {

**printf** ("%d --> ", currentPtr->data);

currentPtr = currentPtr->nextPtr;

}

**puts** ("NULL");

} // main

**Displays**

The list is: 2 --> 4 --> 6 --> NULL