# **Lecture #4** - Pointers - Introduction to linked lists

**Runtime Stack** - A place in memory where data can be saved while a program runs.

* When a function is called (invoked), values and memory addresses are pushed

on the run-time stack.

**When a function is called:** When a function is called, an activation record is placed on the stack.

When a function is called - An activation record is created, and the following items

are pushed on the runtime stack:

1.) **Return address** of the function call.

* + - Control returns to the calling block after the function executes.
    - The return address is the memory address of the next program instruction.

2.) **Actual Parameters** - Values (arguments) passed in the function call are pushed

on the stack.

3.) **Local variables** - Variables declared in the function block are pushed on the stack.

4.) **Return value** - If a function returns a value, it is first pushed on the stack, and later

popped off and returned to the function call.

Example: The following simple program demonstrates a value-returning function, a

**void-returning** function, the runtime stack, and return addresses after execution

#include <iostream> Output 🡪

Enter two numbers:

3

4

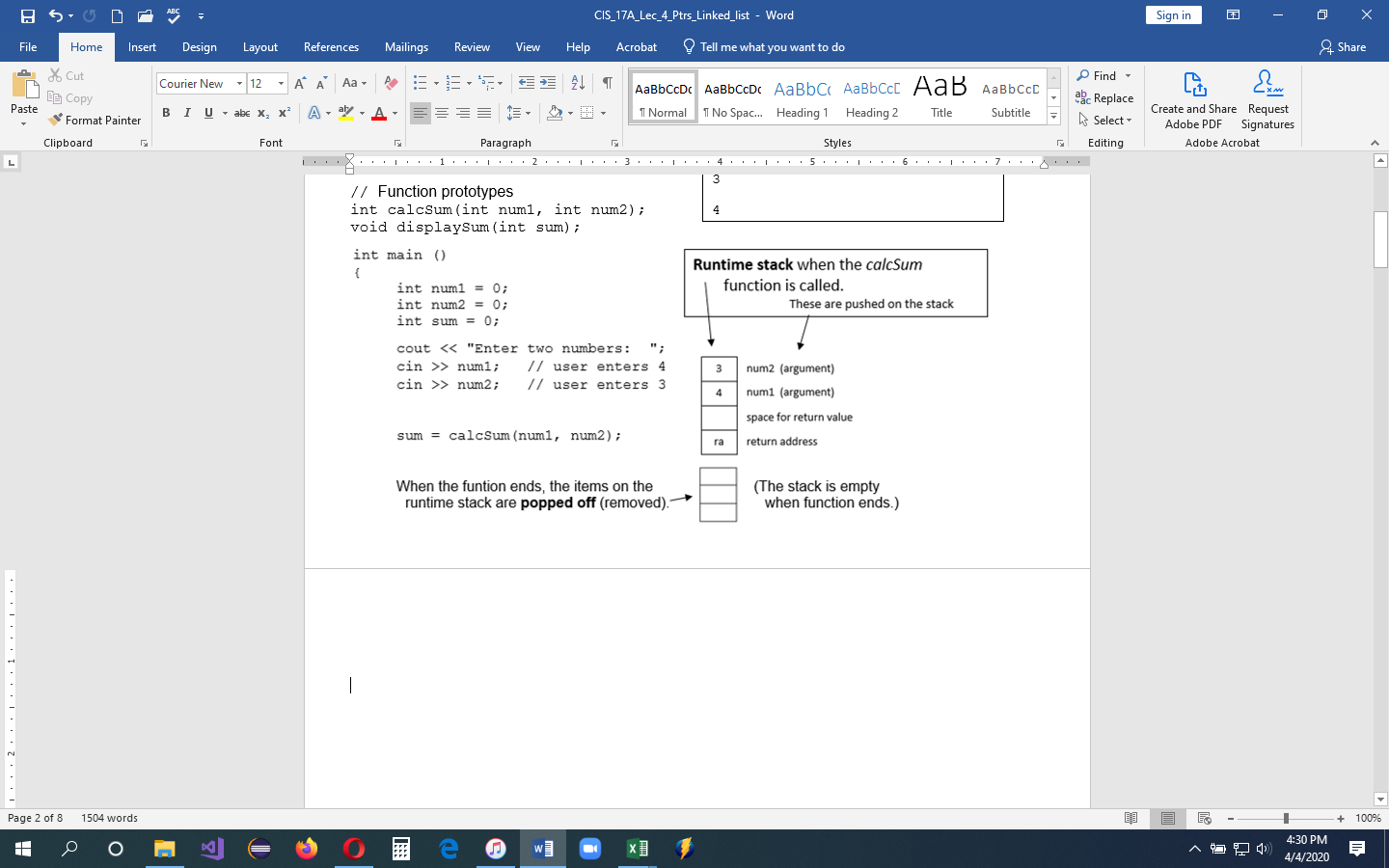
Press any key to continue **\*/**

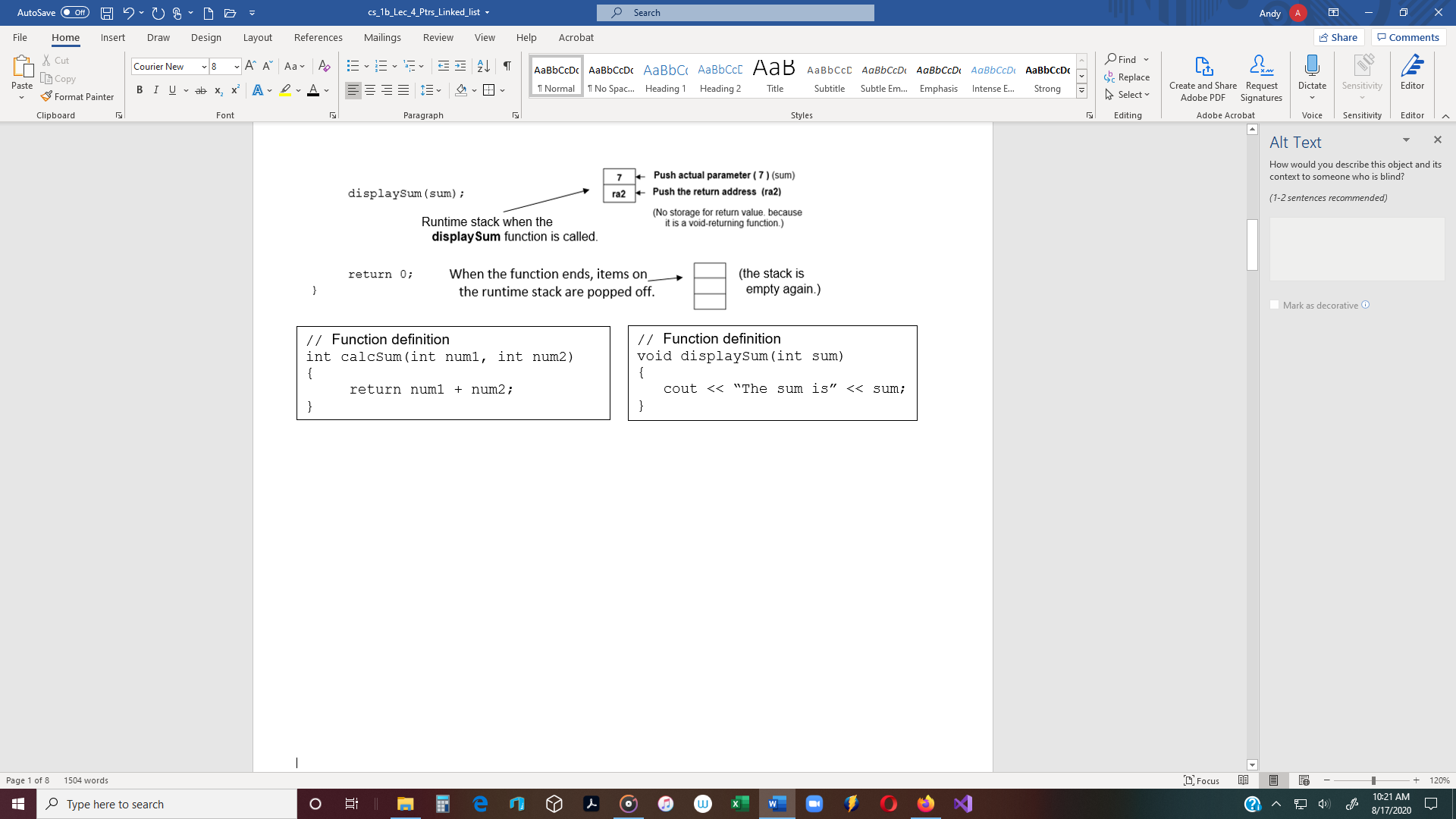
using namespace std;

// Function prototypes

int calcSum(int num1, int num2);

void displaySum(int sum);





**Dynamic memory** - Dynamic memory is memory that is allocated for variables and objects

at runtime.

* **Heap** - Dynamically allocated memory is placed in a special part of memory called the

heap (also called the **free store**).

* The heap is reserved for variables and objects **created during program execution** – Not

at compile time.

Recall: When a function is called, a return address, arguments, local variables and return

value are pushed on the runtime stack.

* + The compiler sees the function calls and functions during compile time, before the program runs.
* **Dynamic memory** is part of RAM where variables and objects can be placed at runtime.

**Pointers** and the **new operator**

* **new operator** - Memory must be explicitly allocated during program execution using

the **new** operator.

* **Pointers** - Pointers are used with the new operator to access dynamically allocated

variables and objects.

Ex #6: int \* ptr = new int; // Dynamically allocates a new int variable

Ex #7: Car \* carPtr = new Car; // Dynamically allocates a new Car object

* A dynamically allocated object or variable does not have a name, and therefore, the only

way to access it is by using a pointer.

* To do this, the pointer must be assigned the address of the variable or object.
* Objects created on the heap do not have names.
* Therefore, pointers must be used to access dynamically-created objects.
* **delete operator** - The delete operator is used to delete a variable or object on the heap.

Ex #8: delete carPtr; // Note: The delete operator deletes the object, the

// pointer points to, not the pointer.

// (The Car object is deleted, not the pointer.)

**Linked List** - Create a linked list using pointers and the heap

**Linked list** - A list of class or struct objects can be linked together using pointers.

* Any group of items can be organized into a list.
* A list of employees or students, transactions,

struct Song

{

string title;

string artist;

Song \* next;

};

or songs, etc., can be put into a linked list.

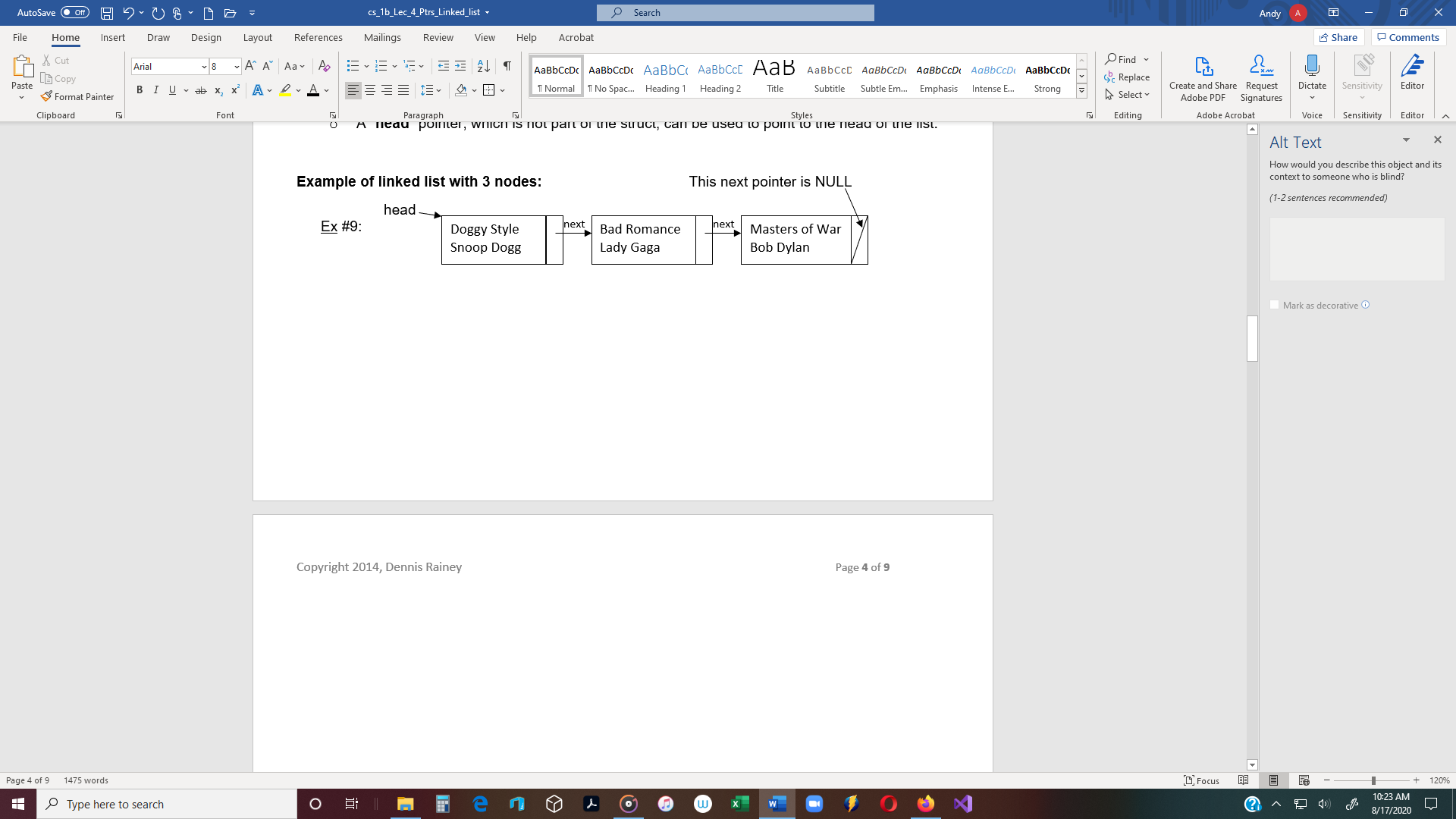
* **Object** – A linked list can be comprised of objects

defined in a struct or class.

* + - Each object has the same data members.
    - In the case of the struct Song, each object of Song type

has three data members: ***title***, ***artist*** and ***next***.

* + - A "**next**" pointer can be used to link one object to another object.
    - A "**head**" pointer, which is not part of the struct, can be used to point to the head of the list.



**NULL pointer** - A pointer that is assigned NULL points to nothing.

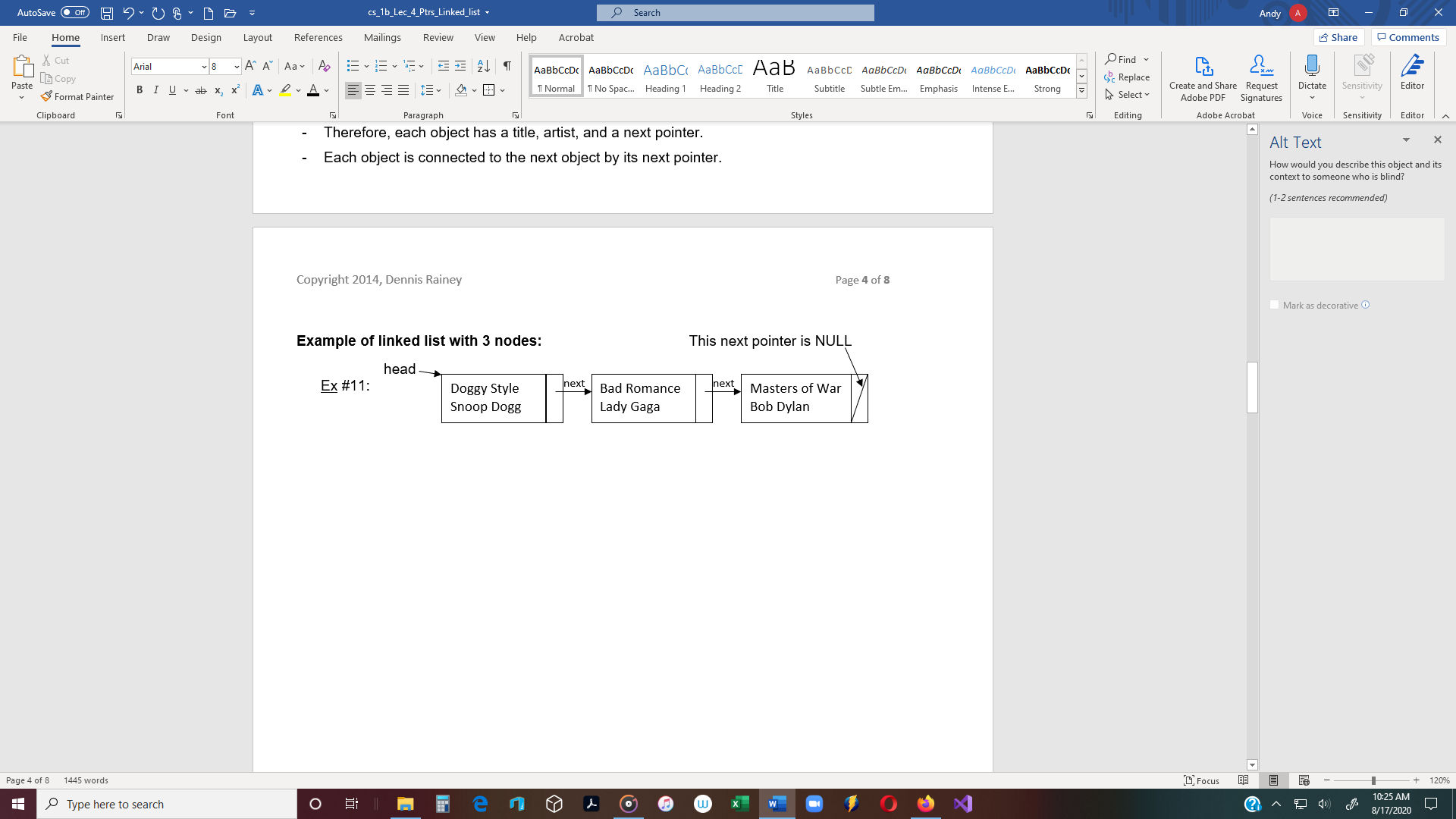
Ex #10: Song \*head = NULL; // Assign NULL (Uppercase)

Example: **Linked List** of struct objects

* Assume that a pointer named "**head**" has already been declared and that it points to the

front of this existing linked list.

* Each object in the list is a struct Song object.
* Therefore, each object has a title, artist, and a next pointer.
* Each object is connected to the next object by its next pointer.

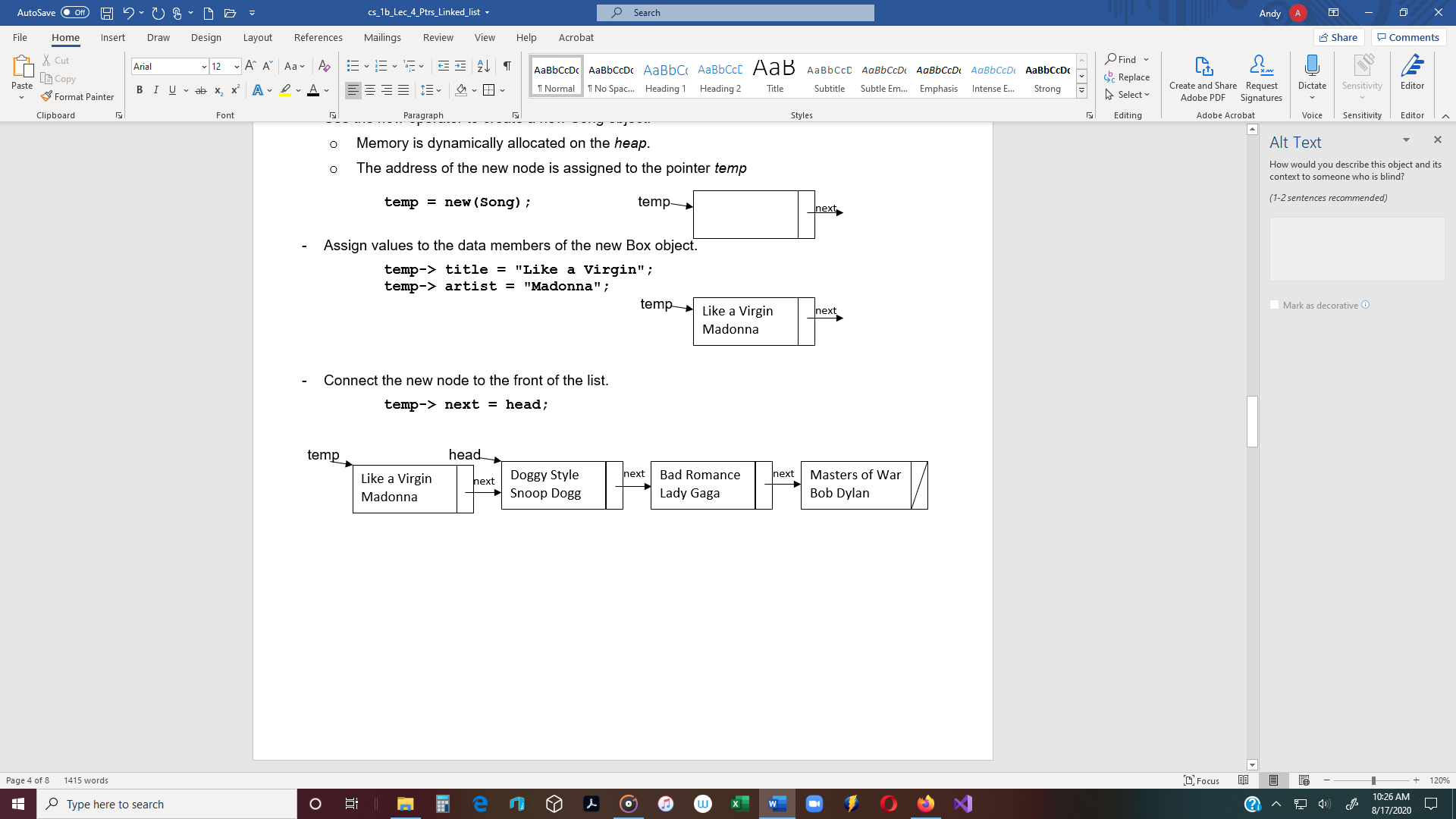


**To add a new node to the front of the list:**

* Declare a new pointer named *temp* (initially it doesn’t point to anything).

**Song \* temp;** temp

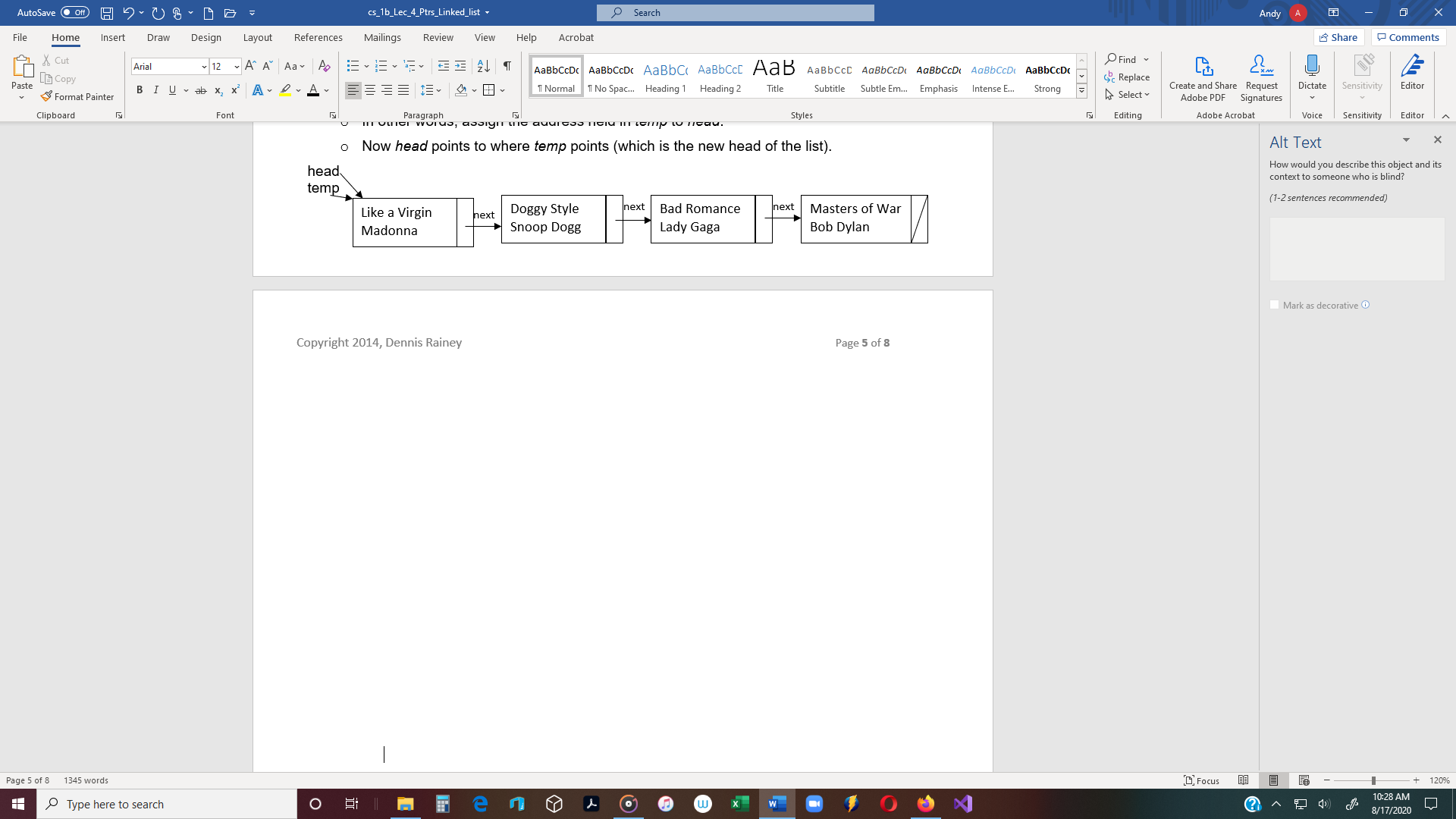
* Use the *new* operator to create a new Song object.
  + Memory is dynamically allocated on the *heap*.
  + The address of the new node is assigned to the pointer *temp*



* The new node’s **next** pointer points to where **head** points, (which is the head of the list).
* Move **head** to the new front of the list.

**head = temp;**

* + In other words, assign the address held in *temp* to *head*.
  + Now *head* points to where *temp* points (which is the new head of the list).



**Example of a linked list of Song objects**

// ===============================

// **Song.h** - struct specification

// ===============================

#include <iostream>

#include <string>

using namespace std;

struct Song

{

int id;

string title;

float price;

Song\* next;

};

// ===============================

// ===============================

// **Source.cpp**

// ===============================

#include "Song.h"

void insertSong(Song\*& head);

void deleteSong(Song\*& head);

void showList(Song\*& head);

int main()

{

Song\* head = nullptr;

char answer = 'Y';

while (toupper(answer) == 'Y')

{

insertSong(head);

cout << "Insert another song(Y or N) ? ";

cin >> answer;

system("cls");

}

cout << "Here is the list of songs:\n";

showList(head);

if (head != nullptr)

{

cout << "Delete a song (Y or N)? ";

cin >> answer;

if (toupper(answer) == 'Y')

{

deleteSong(head);

}

}

else

{

cout << "The list is empty.\n\n";

}

cout << "Here is the list of songs after deleting:\n";

showList(head);

system("pause");

return 0;

}

// ------------

void insertSong(Song\*& head)

{

Song\* temp = new Song;

cout << "Enter ID: ";

cin >> temp->id;

cin.ignore();

cout << "Enter title: ";

getline(cin, temp->title);

cout << "Enter price: ";

cin >> temp->price;

temp->next = head;

head = temp;

}

// ------------

void showList(Song\*& head)

{

Song\* temp = head;

while (temp != nullptr)

{

cout << "ID: " << temp->id << endl

<< "Title: " << temp->title << endl

<< "Price: " << temp->price << endl << endl;

temp = temp->next;

}

}

// ------------

void deleteSong(Song\*& head)

{

Song\* lead = head;

Song\* follow = head;

int idNum;

// Check to make sure the list is not empty

if (head == nullptr)

{

return;

}

cout << "Enter the ID of the song to be deleted: ";

cin >> idNum;

// Search the list to find the ID number (idNum).

// The loop continues while the idNum is not found, and it’s not the end of the list.

// The loop stops when it finds idNum, or when it gets to the end of the list.

while (lead->id != idNum && lead->next != nullptr)

{

follow = lead;

lead = lead->next;

}

// Check to see if it's the first node

if (lead == head)

{

// Delete the first node, but first move head to the next node.

if (lead->id == idNum)

{

head = head->next;

delete lead;

}

else

{

cout << id << " is not in the list.\n\n";

}

}

// Check to see if it's the last node

else if (lead->next == nullptr)

{

// Check to see if it's the last node

if (lead->id == idNum)

{

follow->next = lead->next;

delete lead;

}

else

{

cout << "ID #" << idNum << " is not in the list.\n\n";

}

}

// Therefore it's in the list (but not the first or last)

else

{

follow->next = lead->next;

delete lead;

}

}