Lecture #11 Pointers, and Introduction to linked lists

Pointers and the **new operator**

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

```
struct  Song
{
    string  title;
    string  artist;
    Song * next;
};
```

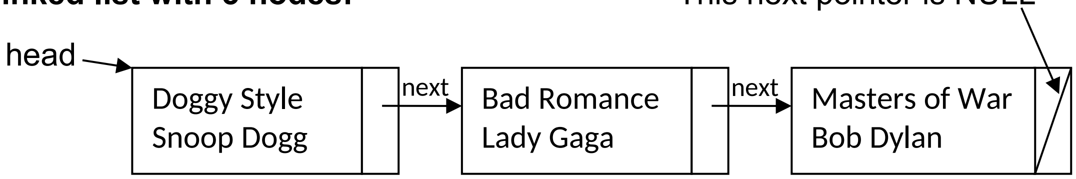
- **Object** – A linked list can be comprised of objects defined in a struct or class.
  - Each object has the same data members.

- 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.

Example of linked list with 3 nodes:

This next pointer is NULL

Ex #9:



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

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

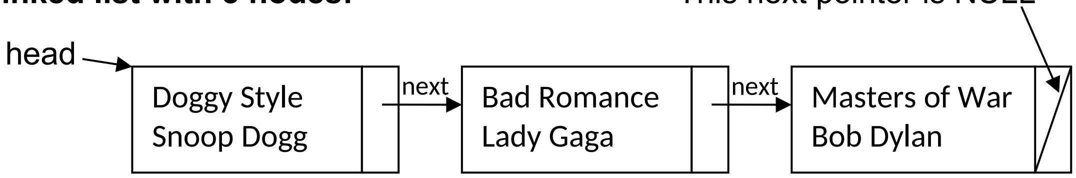
Example: **Linked List** of struct objects

- 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.

Example of linked list with 3 nodes:

This next pointer is NULL

Ex #11:



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

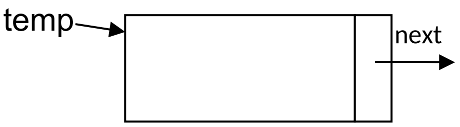
- Declare a new pointer named *temp* (initially it doesn't point to anything).

```
Song * temp;
```

temp ⟶

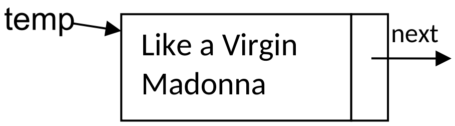
- Use the *new* operator to create a new Song object.
  - Memory is dynamically allocated on the *heap*.
  - The address of the new node is assigned to the pointer *temp*

```
temp = new(Song);
```



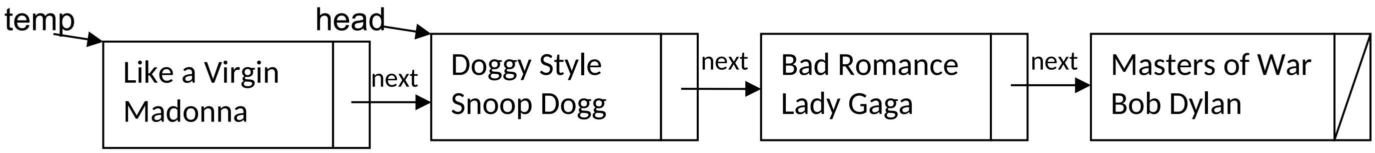
- Assign values to the data members of the new Box object.

```
temp-> title = "Like a Virgin";
temp-> artist = "Madonna";
```



- Connect the new node to the front of the list.

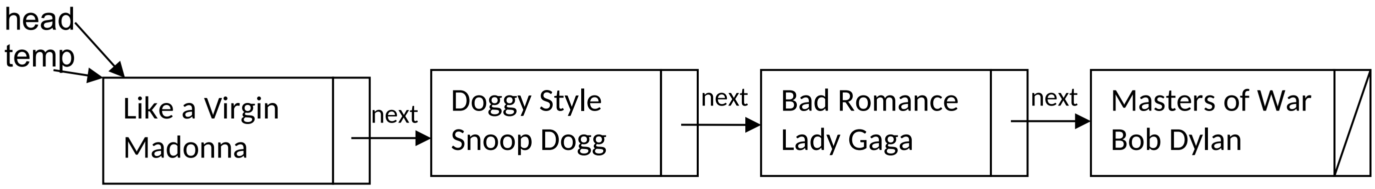
```
temp-> next = head;
```



- 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).



Exercise - Student Linked list

(THE STUDENT STRUCT CODE FROM LAB 6/25 GOES HERE)

Recursive display function – Displays nodes from right to left

```
void displayRecords(Student *&head)
{
     Student *temp = head;

     if (temp == NULL)
     {
          return;
     }
     displayRecords(temp->next);
     cout << temp->id << endl;
}
```

Doubly-linked list

```
// -----------------------------------------
// ---- Student.h --------------------------
// -----------------------------------------

#include <iostream>
#include <string>
using namespace std;

struct Student
{
     int id;
     string name;
     Student *next;
     Student *back;
};
// -----------------------------------------
// ---- main.cpp ---------------------------
// -----------------------------------------

#include "Student.h"

void insertStudent(Student *&head);
void displayStudents(Student *&head);
void displayInReverse(Student *&head);
void deleteStudent(Student *&head);

int main()
{
     char answer = 'Y';
     Student *head = NULL;

     while (toupper(answer) == 'Y')
     {
          insertStudent(head);

          cout << "Enter another record (Y or N)?  ";
          cin >> answer;
     }
     cout << "Here are the Student records:\n\n";
     displayStudents(head);
     deleteStudent(head);

     cout << "Here is the list after deleting one record:\n\n";
     displayStudents(head);

     cout << "\n\nHere is the list in reverse order:\n\n";
     displayInReverse(head);

     return 0;
}
```

```
// ----------------
void insertStudent(Student *&head)
{
     Student *temp = new Student;

     cout << "Enter ID:  ";
     cin >> temp->id;
     cin.ignore();

     cout << "Name:  ";
     getline(cin, temp->name);

     if (head == NULL)   // Check to see if list is empty
     {
          temp->next = head;
          temp->back = NULL;
          head = temp;
     }
     else
     {
          temp->next = head;
          temp->back = NULL;
          head->back = temp;
          head = temp;
     }
}
// ----------------
void displayStudents(Student *&head)
{
     Student *temp = head;

     while (temp != NULL)
     {
          cout << "ID:  " << temp->id << endl
               << "Name: " << temp->name << endl << endl;
          temp = temp->next;
     }
}
// ----------------
void deleteStudent(Student *&head)
{
     Student *lead = head;
     Student *follow = head;
     int id;

     // Check to see if the list is empty (don't delete if it is)
     if (head == NULL)
     {
          cout << "List is empty.\n\n";
               return;
     }
     cout << "Enter the ID:  ";
     cin >> boxID;

     cout << "Enter the ID of the record to be deleted:  ";
     cin >> id;

     // Check to see if the node to be deleted is at the front of the list
     if (lead->id == boxID)
     {
          head = head->next;
          head->back = NULL;
          delete lead;
          return;
     }
     while (lead->id != id)
     {
          follow = lead;
          lead = lead->next;
     }
     // If it’s not the first node, then check to see if it’s one of the other nodes in the list
     else if (lead->id == boxID)
     {
          follow->next = lead->next;
          lead->next->back = lead->back;
          delete lead;
     }
     // If it’s not the first node and it’s not any of the other nodes, then it’s not in the list.
     else
          cout << boxID << " is not in the list.\n\n";
}
// ----------------
void displayInReverse(Student *&head)
{
     Student *temp = head;

     while (temp->next != NULL)
     {
          temp = temp->next;
     }

     while (temp != NULL)
     {
          cout << temp->id << endl;
          temp = temp->back;
     }
}
// -----------------------------------------
```