

Short Lesson: Assignment 4

Dr. Ziad Kobti

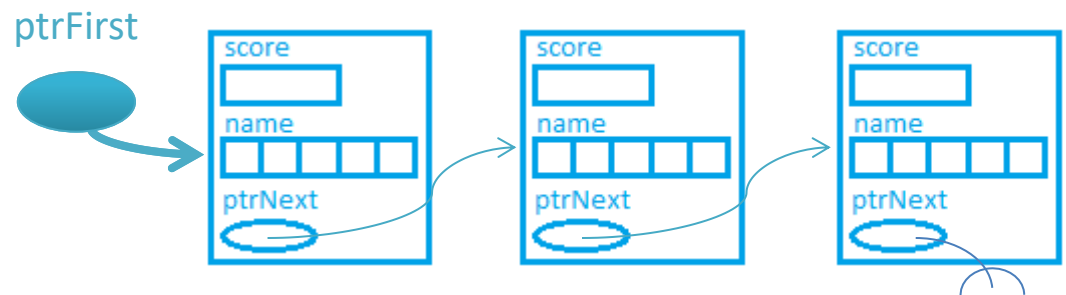
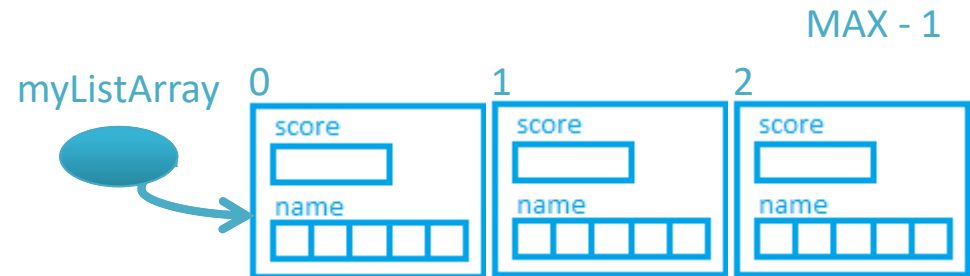
School of Computer Science, University of Windsor

© 2018 All Rights Reserved.

May not be redistributed without the permission of
the author. For education purposes only.

Dynamic Linked-List or Array?

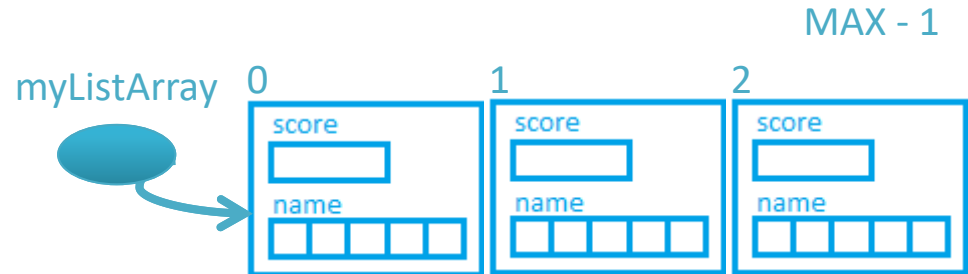
If I want to create a list how do I know whether I should use an **array** data structure or a **dynamic-Linked List**?



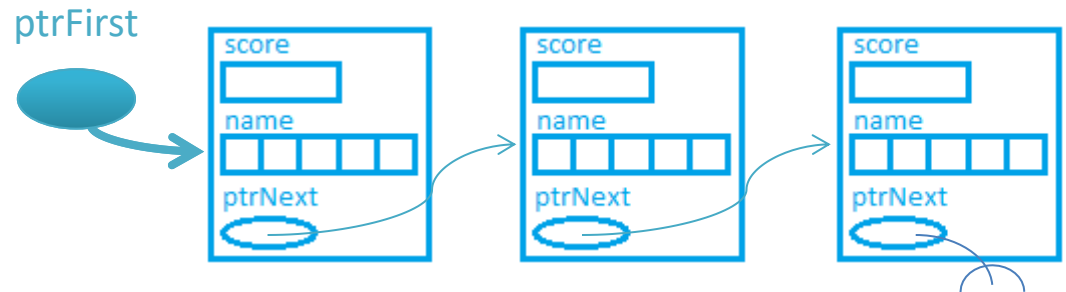
Dynamic Linked-List or Array?

A short answer:

If you don't know the size of the list in advance, it is best to use **dynamic linked list**.
Linked lists are better to handle memory space use efficiently.



Arrays are best if you know the size (MAX size) in advance at compile time and when you are not planning to resize them. The challenge is that when the size of the list is unpredictable you can either exceed the boundary of your array in one extreme, or waste a lot of unused storage at the other extreme.



Linked-lists are dynamic – meaning you can allocate at run time (while the program is running) the exact amount of memory you need. You can ask for more memory as you go, or free up and use less memory as needed. Hence they make better use of the memory.

Building a Dynamic Linked-List

Step 1: Define a self-referential structure

Recall the structure from Assignment 3:

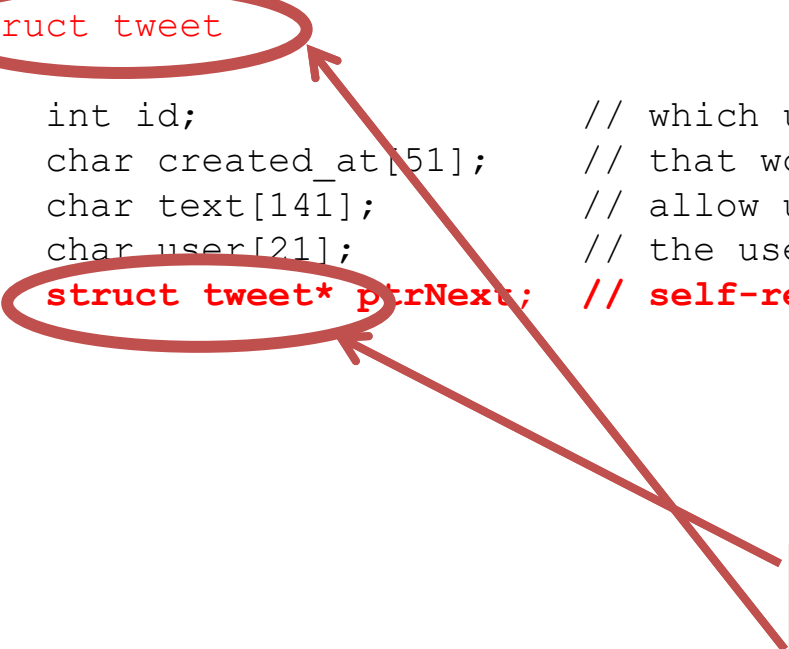
```
// Structure definition of an event aggregate data type
struct tweet
{
    int id;                // which uniquely identifies the simple tweet
    char created_at[51];   // that would store the UTC time and date.
    char text[141];        // allow up to 140 characters.
    char user[21];         // the user name of the person who posted the tweet
    struct tweet* ptrNext; // self-referential pointer
};
```

Add a pointer, typically named for its purpose, here ptrNext because we intend to use it to point to the next record (or item or node) in the list.

Building a Dynamic Linked-List

Step 1: Define a self-referential structure

```
// Structure definition of an event aggregate data type
struct tweet
{
    int id; // which uniquely identifies the simple tweet
    char created_at[51]; // that would store the UTC time and date.
    char text[141]; // allow up to 140 characters.
    char user[21]; // the user name of the person who posted the tweet
    struct tweet* ptrNext; // self-referential pointer
};
```



Note the data type of the self-referential pointer – it is a pointer variable of the same type as the structure it is in.

Building a Dynamic Linked-List

Step 2: Define the alias (helpful but not required in general)

```
// Structure definition of an event aggregate data type
struct tweet
{
    int id;                // which uniquely identifies the simple tweet
    char created_at[51];   // that would store the UTC time and date.
    char text[141];        // allow up to 140 characters.
    char user[21];         // the user name of the person who posted the tweet
    struct tweet* ptrNext; // self-referential pointer
};

typedef struct tweet Tweet;
```

Building a Dynamic Linked-List

Step 3: Declare the list pointer(s)

```
// Structure definition of an event aggregate data type
struct tweet
{
    int id;                // which uniquely identifies the simple tweet
    char created_at[51];   // that would store the UTC time and date.
    char text[141];        // allow up to 140 characters.
    char user[21];         // the user name of the person who posted the tweet
    struct tweet* ptrNext; // self-referential pointer
};
```

```
typedef struct tweet Tweet;
```

```
Tweet* ptrFirst = NULL;
```

```
Tweet* ptrLast  = NULL;
```

Make sure you include
#include <stdlib.h>

This pointer **ptrFirst** is intended to always point to the first element of the list.

By definition, if this pointer is NULL, it means the list is empty.

This pointer **ptrLast** is intended to always point to the last element of the list. This pointer is not required, but very helpful if you need to access the last item in the list quickly; for instance, it makes adding to the end of the list easier. By definition, if this pointer is NULL, it means the list is also empty. We will use it in this tutorial.

Building a Dynamic Linked-List

Step 4: Plan the functions you need

1. Open the file for reading
 1. If you can't open the file successfully then quit the program
2. Is there a record to read?
 - If yes,
 - a. Read one record to a new memory space
 - b. Add the record to the list in such a way that it remains in sorted order
 - c. Repeat, back to step 2.
 - If no, you are done reading, close the file
3. Open the file for writing
4. Save all the records to the file
5. Close the file
6. Clear the records from memory

Always make sure you free() all the records from memory. For every call to malloc() there must be a call to free(). Don't assume if you free the first pointer in the list this will free the rest of the list! You need to free every single record.

Top down approach

- Remember that your functions need to be very specific in what they do and don't cram too many features (functionality) into one function. Generally, one purpose for one function.
- There are several examples here:
<http://cs141.cs.uwindsor.ca/tutorials>
For you to read and learn from to inspire your code.