Lecture 25 — The Linked List

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

July 28, 2016

Acknowledgments: W.D. Bishop

Arrays and Collections

We've established that we would like to have a dynamic collection that does not rely on an array.

When using an array, we have a reference to the array.

- The reference tells us where the start of the array is in memory.
- The index tells us which entry of the array to access.

In a collection, the array serves the purpose of keeping the elements next to one another so that we can move from one to the next.

- To find the next object in an array, we increment the index by 1.

Linking Collection Elements

If the elements are not stored in an array, given that we know where the first object is, how might we get to the next object?

Idea: use a reference to link the objects together.

In the simplest case, each object has a reference to the next object.
A null reference indicates there is no next object.

Analogy: pirate treasure map.
Follow the map's directions; it leads to treasure and another map.
The 2^{nd} map says how to find another treasure and a third map.
... and repeat until we get to the final treasure (that has no map).

The Linked List

Linked lists are similar in function to arrays but are more flexible:

- The size of a linked list is not fixed
- Objects may be easily inserted or deleted from a linked list
- Objects may be sorted (if desired)

Linked lists are formed using self-referential structs (or later, classes):
A data type that has a pointer to another item of the same type.

For example, a class `ListEntry` can contain one or more data fields as well as a reference to another object of type `ListEntry`.

Linking Objects

Let's write a very simple `list_entry` struct.

```
struct list_entry
{
    public int data;
    public list_entry* next;
}
```

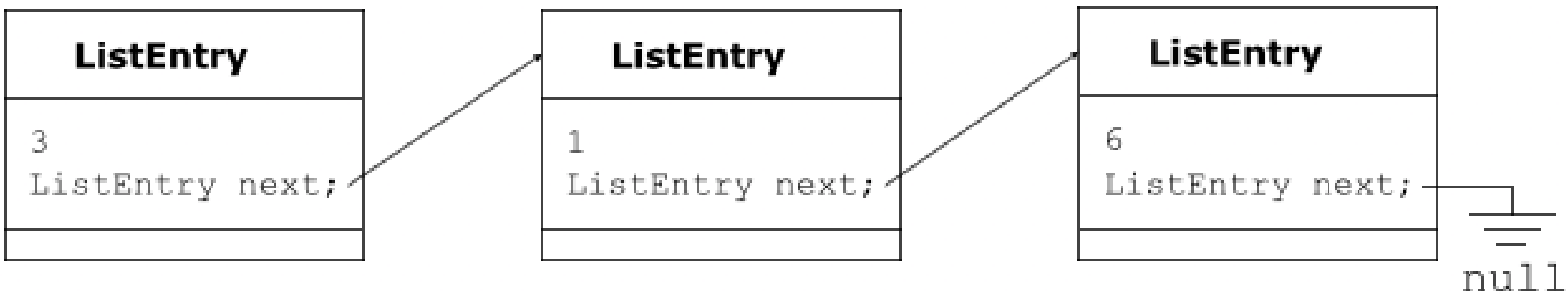
This works because `next` is a pointer.

The compiler can figure out how much memory `list_entry` needs because it knows the size of an `int` and the size of a pointer (`next`).

Creating a List of Integers

Using the struct previously defined, it is possible to create a collection of records known as a singly linked list.

For example, it is possible to define a list containing a set of integers {3,1,6} as follows:

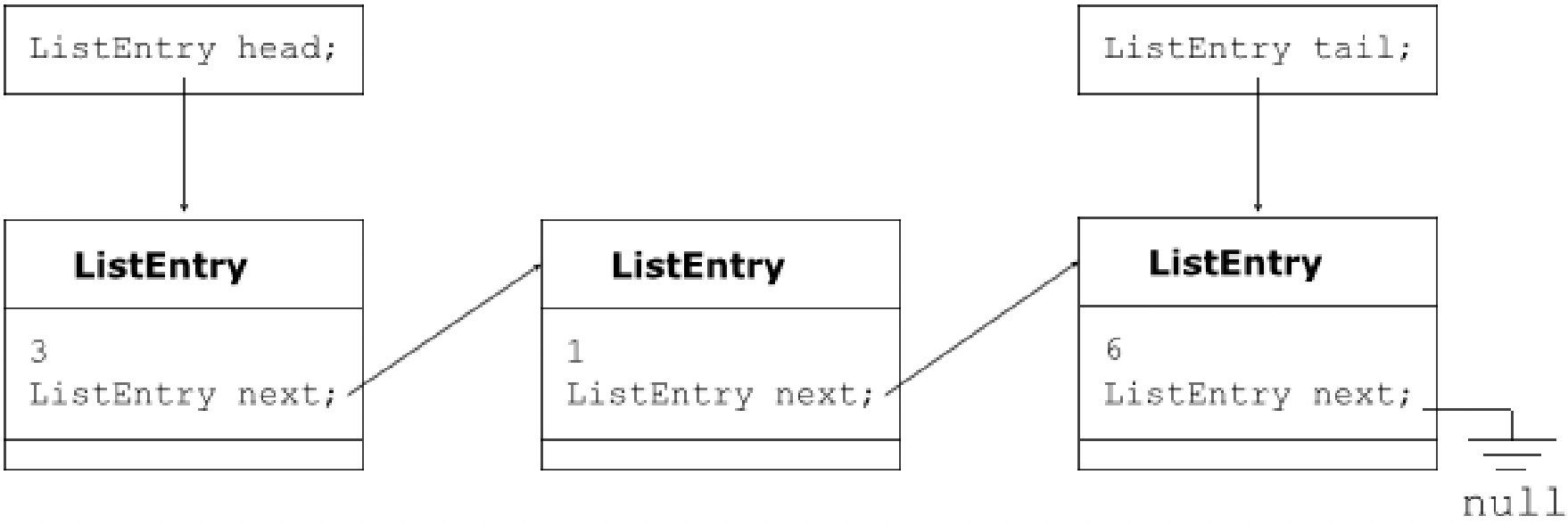


When a pointer is null, it is sometimes drawn as if it's an electrical connection to ground.

Accessing the Linked List

The linked list on the previous slide has no references pointing to it...

Developers often keep a pointer to the head and the tail of the list:



A tail pointer is not strictly needed; it's a perfectly functional linked list with only a head reference. A tail poster can be convenient.

The `linked_list`

```
struct linked_list {
    list_entry* head;
    list_entry* tail;
}
```

This is certainly a simple definition...

Now, how about some functions to manipulate it:

- `initialize`
- `append`
- `prepend`
- `remove_first`
- `clear`

Initialize

```
void initialize( linked_list* list ) {
    list->head = 0;
    list->tail = 0;
}
```

Append

```
void append( linked_list* list, int data ) {
    if (tail == 0) {
        list->head = new list_entry;
        list->tail = head;
        list->tail->data = data;
        list->tail->next = 0;
    } else {
        list_entry* entry = new list_entry;
        entry->data = data;
        entry->next = 0;
        list->tail.next = entry;
        list->tail = tail->next;
    }
}
```

Prepend

```
void prepend( linked_list* list, int data ) {
    list_entry* tmp = new list_entry;
    tmp->data = data;
    tmp->next = list->head;

    list->head = tmp;
    if (list->tail == 0) {
        list->tail = list->head;
    }
}
```

Remove First

```
list_entry* remove_first( linked_list* list ) {
    list_entry* result = list->head;
    if (list->head != 0) {
        list->head = list->head->next;
    }
    if (list->head == 0) {
        list->tail = 0;
    }
    return result;
}
```

Clear

```
void clear( linked_list* list ) {
    while( list-> head != 0 ) {
        list_entry* current = list->head;
        list->head = list->head->next;
        delete current;
    }
    list->tail = 0;
}
```

List Creation Demonstration

Let's work with this code and represent visually what the following series of statements would look like when executed.

```
1 initialize( list );
2 append( list, 3 );
3 append( list, 1 );
4 append( list, 6 );
5 prepend( list, 4 );
6 prepend( list, 5 );
7 remove_first( list );
8 prepend( list, 2 );
9 clear( list );
```

[Demo done on the board to facilitate understanding.]

Expanding on the Simple Linked List

A second reference in each `list_entry` that refers to the previous element can simplify deletion and other complex operations.

This is known as a doubly-linked list.

The use of dummy items (i.e., dummy head or dummy tail) can simplify some the termination conditions for some loops.

A dummy item is a `list_entry` that has a valid reference but never any valid data. It is constructed when the list is constructed.

A length member field can be added to keep track of the length of the list as items are added and removed.