

# UNIX and C Programming (COMP1000)

## Lecture 6: Structs

Updated: 20<sup>th</sup> August, 2018

Department of Computing  
Curtin University

Copyright © 2018, Curtin University  
CRICOS Provide Code: 00301J

## Textbook Reading (Hanly and Koffman)

For more information, see the weekly reading list on Blackboard.

- ▶ **Chapter 10: Structure and Union Types**

Unions (Section 10.6) are not important until lecture 9.

- ▶ **Sections 13.1 to 13.4 in Chapter 13 (“Dynamic Data Structures”)**

We’re only interested in linked lists here. Other structures (sections 13.5 onwards) are beyond the scope of the unit.

# Outline

Structs

Structs, Functions and Pointers

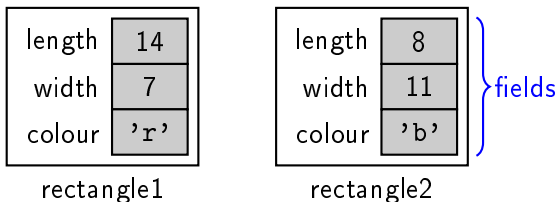
Structs and Arrays

Linked Lists

Linked List Operations

# Structs

- ▶ A composite datatype — a bucket of variables called “fields”.
- ▶ Each field has its own name and data type.
- ▶ First, declare a struct datatype, saying what fields it contains.
- ▶ Then, declare variables of that type. Each struct variable has its own copies of those fields.



Two variables of the same struct type

## Structs vs. Arrays

- ▶ Structs and arrays both have subcomponents that behave like variables.
- ▶ Array elements:
  - ▶ Accessed by an *index* (an integer “location” in the array).
  - ▶ Have the *same* data type.
- ▶ Struct fields:
  - ▶ Accessed by name.
  - ▶ Have different data types (typically).
- ▶ Arrays are *always* accessed via pointers.
- ▶ Structs are *often* accessed via pointers, but don't have to be.

## Declaration (the first way)

```
struct <tag> {  
    <field declarations>  
};
```

- ▶ Normally *outside* functions, and in a header file.
- ▶ Creates a new data type called “struct <tag>”, where “<tag>” can be any valid identifier.
- ▶ Fields are declared like variables.
  - ▶ *Except* “=” is not allowed in the declaration.

### Example

```
struct Result {  
    int mark;  
    char grade;  
};
```

## Struct Instantiation

- ▶ You can now create a struct variable (*inside* a function):

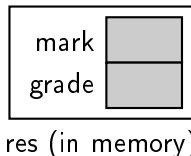
```
struct <tag> <variable-name>;
```

- ▶ “struct <tag>” is the data type.
- ▶ “<variable-name>” is the name of the variable (surprise!).

### Example

Declare a variable “res” of type “struct Result”:

```
int main(void)
{
    struct Result res;
    ...
}
```



## Instant Struct Variables (FYI)

You can also declare a struct variable directly:

```
int main(void) {  
    struct {  
        int mark;  
        char grade;  
    } res; /* One variable with a struct datatype. */  
    ...  
}
```

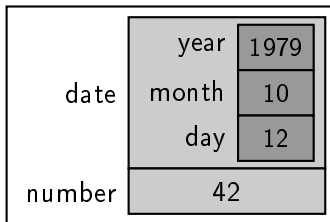
- ▶ The whole of “`struct{int mark; char grade;} ”` is just datatype, like “`int`” itself.
- ▶ It can (in principle) appear anywhere you need a datatype.
- ▶ Above, we declare the variable “`res`” of that type.
- ▶ *However...* what if we need many such variables?
- ▶ Much easier to give the datatype a name!



## Structs Within Structs

- ▶ Struct fields can have any data type.
- ▶ They can be structs themselves:

```
struct Answer {  
    struct {  
        int year;  
        int month;  
        int day;  
    } date;  
    int number;  
};
```



A variable of type struct Answer.

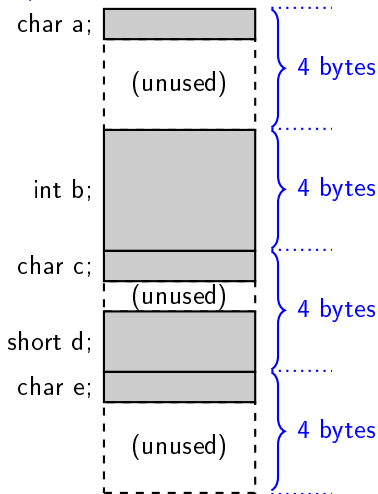
- ▶ struct Answer has 2 fields: date and number.
- ▶ date is also a struct, with 3 fields: year, month and day.

# Alignment

- ▶ Struct fields are stored in the order of declaration.
- ▶ However, there are often gaps between them!
- ▶ The compiler tries to place fields on “word boundaries”.
- ▶ A “word boundary” is a memory address that happens to be a multiple of the word size — the natural size of data processed by the CPU.
  - ▶ On a 32-bit machine, words are 4-bytes (32 bits) long.
  - ▶ On a 64-bit machine, words are 8-bytes (64 bits) long.
- ▶ The CPU can access things on word boundaries more efficiently.
- ▶ The struct is also padded at the end, if necessary, to make it a multiple of the word size.
- ▶ This behaviour is compiler-dependent. It may work different ways on different systems!

## Alignment (32-Bit Example)

```
struct Example {  
    char a;  
    int b;  
    char c;  
    short d;  
    char e;  
};
```



Together a, b, c, d and e have 9 bytes, yet  
`sizeof(struct Example) == 16.`

## Accessing Struct Fields

- ▶ When you have a struct variable, you can access its fields with the “.” operator:

**<struct-variable> . <field-name>**

- ▶ For instance:

```
struct Result res;  
res.mark = 65;
```

- ▶ Note: we *cannot* simply say “mark = 65”.
  - ▶ There could be many copies of that field, each inside a different struct variable.
  - ▶ We must say *which* mark we want; e.g. “res.mark”.
- ▶ If the field is itself a struct, you can apply the “.” operator again:

```
struct Answer ans;  
ans.date.year = 1979;
```

## Accessing Struct Fields — Example

### Example

```
struct Result res;  
int* markPtr;  
char* gradePtr;  
  
res.mark = 85;  
res.grade = 'A';  
printf("%d, %c\n", res.mark, res.grade);  
  
/* Showing off with pointers. */  
markPtr = &res.mark;  
gradePtr = &res.grade;  
printf("%d, %c\n", *markPtr, *gradePtr);
```

## Struct Values

- ▶ A struct variable has a single composite value.
- ▶ You can copy an entire struct at once: `struct2 = struct1;`
- ▶ This implicitly copies all the struct's fields.
- ▶ You cannot do this with arrays.

## Copying Structs — Example

### Example

```
int main(void) {  
    struct Result res1;  
    struct Result res2;  
  
    scanf("%d, %c", &res1.mark, &res1.grade);  
  
    res2 = res1; /* Copy the entire struct */  
  
    printf("%d, %c\n", res2.mark, res2.grade);  
  
    return 0;  
}
```

## Structs and Typedef

- ▶ `typedef` is often used with struct declarations.
- ▶ Otherwise, you have to type “`struct`” everywhere.

### Without typedef

```
struct Result {  
    int mark;  
    char grade;  
};  
  
...  
  
struct Result res;
```

### With typedef

```
typedef struct {  
    int mark;  
    char grade;  
} Result;  
  
...  
  
Result res;
```

(Recall: `typedef` just creates an alias. In the right-hand case, it creates an alias for “`struct{int mark; char grade;}”.)`



## Structs and Pointers

- ▶ If you have a pointer to a struct, you must dereference the pointer before you can access the fields.
- ▶ Due to the order of operators, you must use brackets:

```
Result* res;  
...  
(*res).mark = 75;  
(*res).grade = 'D';
```

- ▶ Without brackets, the “.” operator would take precedence:

```
*res.mark = 75; /* Fail! This tries to  
                dereference 'res.mark'. */
```

## The “->” Operator (1)

- ▶ Combines the “\*” and “.” operators.
- ▶ If you write “structPtr->field”, you are:
  1. dereferencing “structPtr” (on the left), then
  2. accessing “field” (on the right).
- ▶ Exactly equivalent to “(\*structPtr).field”.
- ▶ Just syntactic sugar.

### Example

```
Result* res;
```

```
...
```

```
res->mark = 75;
```

```
res->grade = 'D';
```

=

```
Result* res;
```

```
...
```

```
(*res).mark = 75;
```

```
(*res).grade = 'D';
```

## The “->” Operator (2)

Don't let it confuse you:

- ▶ “->” is *only* for pointers to structs (*not* pointers in general).
- ▶ “->” does *not* “make something point to something else”.
  - ▶ (You need “=” for that!)

## Allocating Structs

- ▶ Structs are often accessed via pointers, because...
- ▶ ...structs are often allocated on the heap using `malloc()`.
- ▶ You must eventually free this memory.

### Example

```
Result* res = (Result*)malloc(sizeof(Result));  
...  
free(res);
```

(Allocates a block of memory for a single struct.)

- ▶ Of course, you don't need `malloc()` to have a pointer.
- ▶ You *do* need `malloc()` to put a struct on the heap.

# Structs and Functions

- ▶ Structs can be passed by value, and even *returned* by value (unlike arrays).
- ▶ However, it's more common to pass by reference:
  - ▶ Structs are often quite large.
  - ▶ Passing/returning by value involves copying.
  - ▶ Unnecessary copying is inefficient.

## Passing Structs by Value

A copy is made of the entire struct — all its fields (just as in `struct2 = struct1;`)

```
void printResult(Result res) {  
    printf("%d, %c\n", res.mark, res.grade);  
}  
  
int main(void) {  
    Result res;  
    ...  
    printResult(res);  
    ...  
}
```

## Returning Structs

Similarly, we can return an entire struct, including all its fields:

```
Result getResult(void) {  
    Result res;  
    scanf("%d %c", &res.mark, &res.grade);  
    return res;  
}
```

Here, we're effectively returning two values, by wrapping them in a struct.

## Passing Structs by Reference

More conventional — only pass a pointer (saves copying).

```
void printResult(Result* res) {  
    printf("%d, %c\n", res->mark, res->grade);  
}  
  
int main(void) {  
    Result res;  
    res.mark = 75;  
    res.grade = '7';  
  
    printResult(&res);  
    ...  
}
```



## Struct Initialisation

- ▶ Recall that you can initialise an array like this:

```
int array[5] = {2, 4, 6, 8, 10};
```

- ▶ You can use a similar syntax to initialise structs:

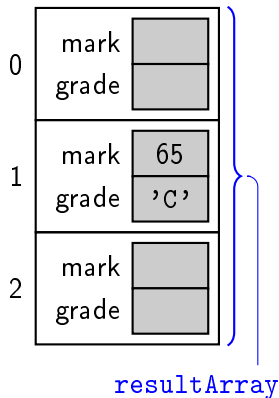
```
typedef struct {  
    int mark;  
    char grade;  
} Result;  
...  
Result res = {75, 'B'};
```

- ▶ The values are assigned to fields *in order*.
- ▶ Of course, the data types must match!

## Arrays of Structs

Arrays can contain any data type, including structs:

```
typedef struct {  
    int mark;  
    char grade  
} Result;  
  
int main(void)  
{  
    Result resultArray[3];  
    resultArray[1].mark = 65;  
    resultArray[1].grade = 'C';  
    ...  
}
```



## Arrays of Structs (2)

```
resultArray[1].mark = 65;
```

array

struct

int

- ▶ “resultArray” itself is an array.
- ▶ Adding “[1]” gives us array element 1, which is a struct.
- ▶ Adding “.mark” gives us a field inside that struct.
- ▶ Notice: the more “things” we add, the more specific we get.

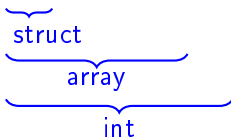
## Structs Containing Arrays

Structs can also contain any data type, including arrays:

```
typedef struct {  
    char surname[50];  
    int lottoNum[6];  
} Person;  
  
int main(void)  
{  
    Person per = {"Blackadder",  
                  {3, 14, 6, 17, 5, 2}};  
    ...  
    per.lottoNum[1] = 1 + per.lottoNum[2];  
    ...  
}
```

## Structs Containing Arrays (2)

```
per.lottoNum[1] = 1 + per.lottoNum[2];
```

  
struct  
array  
int

- ▶ “per” itself is a struct.
- ▶ Adding “.lottoNum” gives us the lottoNum field — an array.
- ▶ Adding “[2]” indexes that array, giving us an int.

## Confused?

- ▶ Pointers, arrays and structs are often combined, leading to complex cases.
- ▶ This will test whether you have understood the “\*”, “.”, “[]” and “->” operators.
- ▶ If you truly understand them, this won't be a problem:

```
*(*var[4].fieldA)->fieldB[16][5]->fieldC = 4;
```

- ▶ You should be able to translate this into English.
- ▶ There is nothing new here. You just need to apply the rules!
- ▶ Think of this as a mathematical expression, just with different operators.

## Structs and header files

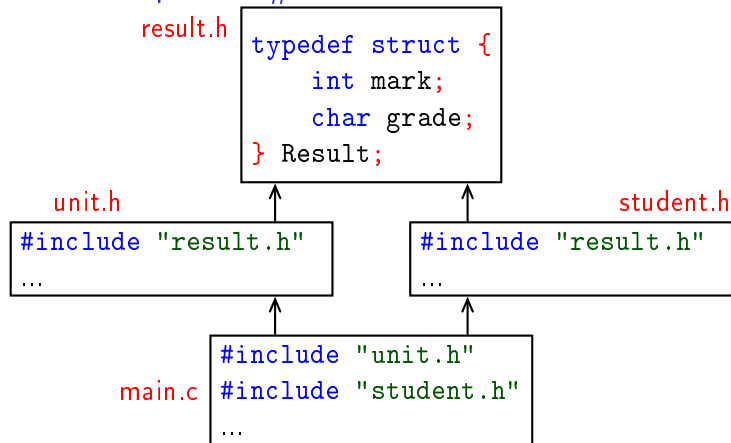
- ▶ Structs are often used to pass complex data between functions.
- ▶ So, they are often declared in header files:

result.h

```
typedef struct {  
    int mark;  
    char grade;  
} Result;  
  
Result getResult(void);  
void printResult(Result res);
```

- ▶ To call getResult() or printResult() from another file, we need both the struct and function declarations.

## Problem: Duplicate #includes



- ▶ The declaration of `Result` is included twice in `main.c`.
- ▶ This is a compile error.



## #Include Guards

- Use conditional compilation to avoid multiple declarations:

```
#ifndef RESULT_H
#define RESULT_H

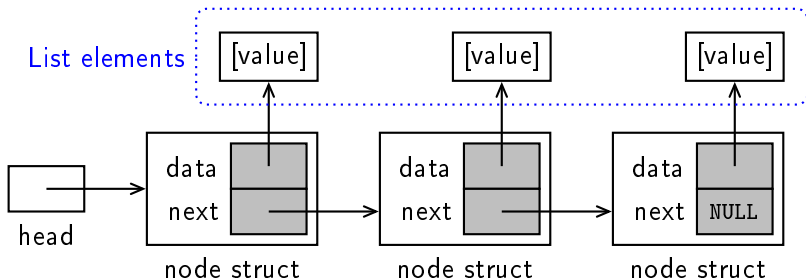
typedef struct {
    int mark;
    char grade;
} Result;

...
#endif
```

- On the first `#include`, the struct will be declared, along with `RESULT_H`.
- On each extra `#include`, `#ifndef` will skip everything.
- (You may also see the non-standard “`#pragma once`”.)

## Linked Lists

- ▶ A linked list is a chain of dynamically-allocated memory blocks.
- ▶ Each block — “node” — contains a pointer to the next.
- ▶ This produces a linear structure.
- ▶ Each node also contains a pointer to a value.
- ▶ These values are what we actually want to store!



## Self-Referential Structs

- ▶ In C, linked lists are best implemented with structs.
- ▶ We define a struct representing the node.
- ▶ This requires some (slightly) strange syntax:

```
typedef struct LinkedListNode {  
    void* data;  
    struct LinkedListNode* next;  
} LinkedListNode;  
  
typedef name
```

- ▶ Two names: the tag name and typedef name. Here, they're both the same (but they don't have to be).
- ▶ The typedef name can't be used inside the struct itself, but the tag name can.

## A Closer Look

We have our Node struct, as before:

```
typedef struct LinkedListNode {  
    void* data;  
    struct LinkedListNode* next;  
} LinkedListNode;
```

- ▶ Many instances of this struct go into making a linked list.
- ▶ For each instance:
  - `data` points to some arbitrary value (the actual contents of the linked list).
  - `next` points to the next node, or NULL if it's the last.
- ▶ You can replace `void*` with whatever data type you like.

## The Head Pointer

- ▶ Somewhere, there must be a pointer to the first node in the list — the “head”.
- ▶ If this is NULL, then the list is empty.

```
int main(void)
{
    LinkedListNode* listHead = NULL;

    ... /* Create, manipulate & access the
         list using listHead. */
}
```

- ▶ An empty linked list takes up zero memory (except for the head pointer itself).

## “Current” Pointers

- ▶ Say you want to traverse a linked list (i.e. access each of its values).
- ▶ You start at the head, and follow each “next” pointer in turn.
- ▶ When you’re doing this, you need a “current” pointer, to keep track of the node you’re currently accessing.
- ▶ When you’re done with one node, you update the pointer.

```
current = head;
while(current != NULL) {
    ... /* Do something with current */
    current = (*current).next;
}
```

(Note: we could also say “`current = current->next;`”.)

## “Previous” Pointers

- ▶ It's also sometimes necessary to have a “previous” pointer, to keep track of the last node you visited.
- ▶ Used in conjunction with a “current” pointer.
- ▶ In general, `(*previous).next == current`.
- ▶ This is useful if you want to insert an element into the middle of a list.

```
current = head;
previous = NULL;
while(current != NULL) {
    ... /* Do something with current/previous */
    previous = current;
    current = (*current).next;
}
```

## Linked List Pointer Examples

Say “head” and “current” are LinkedListNode pointers:

Expression	Points to...
current	The “current” node.
<code>(*current).data</code>	The “current” value.
<code>(*current).next</code>	The next node after the current node.
head	The first node.
<code>(*head).data</code>	The first value (also <code>head-&gt;data</code> ).
<code>(*head).next</code>	The second node (also <code>head-&gt;next</code> ).
<code>(*(*head).next).data</code>	The second value (also <code>head-&gt;next-&gt;data</code> ).



## A Central Linked List Struct?

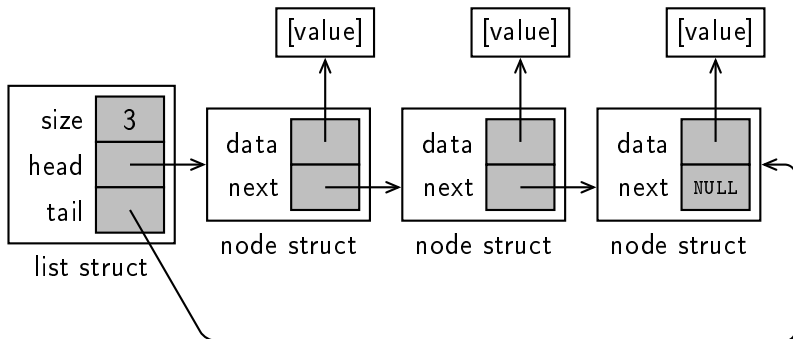
- ▶ It's often nicer to put the head pointer in its own struct:

```
typedef struct LinkedListNode {  
    void* data;  
    struct LinkedListNode* next;  
} LinkedListNode;
```

```
typedef struct {  
    LinkedListNode* head;  
} LinkedList;
```

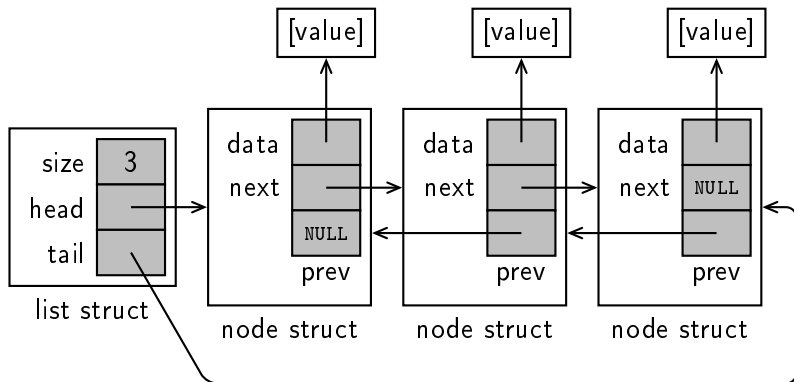
- ▶ There are other non-essential things you can store in `LinkedList`:
  - ▶ The size of the list.
  - ▶ A pointer to the *last* node — the “tail”.

## Linked List with a Size and Tail Pointer



- ▶ We can jump instantly to the last node.
- ▶ Updating this structure is slightly more complicated.

## A Doubly Linked List



- ▶ We may decide to have a “previous” link inside each node.
- ▶ We can traverse this list forwards and backwards.

## Linked List Operations

- ▶ Here's where you have to **think for yourself!**
- ▶ You can do many things with a linked list.
- ▶ This lecture will try to cover some of them.
- ▶ However, there are many *other* ways of doing the same thing.
- ▶ Treat the following slides as **examples**.

### Do not...

- ▶ Memorise the following slides.
- ▶ Copy and paste from the following slides.

### Do...

- ▶ Understand the linked list structure.
- ▶ Use the following slides to see how that structure works.

# Linked List Operations

I mean it.

## Linked List Operations

- ▶ These are some typical linked list operations:
  - ▶ creating a list;
  - ▶ populating the list;
  - ▶ traversing the list;
  - ▶ freeing the list;
  - ▶ inserting elements;
  - ▶ accessing elements;
  - ▶ deleting elements;
  - ▶ ...and many more!
- ▶ There are no standard C functions for this — you have to implement them yourself.
- ▶ These operations are usually put inside their own functions (but they could be embedded directly in other code).

## Creating an Empty Linked List

To create a new, empty linked list:

- ▶ Just malloc the linked list struct.
- ▶ Set the head field to NULL, to indicate an empty list.

```
LinkedList* list;  
  
list = (LinkedList*)malloc(sizeof(LinkedList));  
(*list).head = NULL;
```

- ▶ Consider making a function for this!

## Inserting an Element at the Start

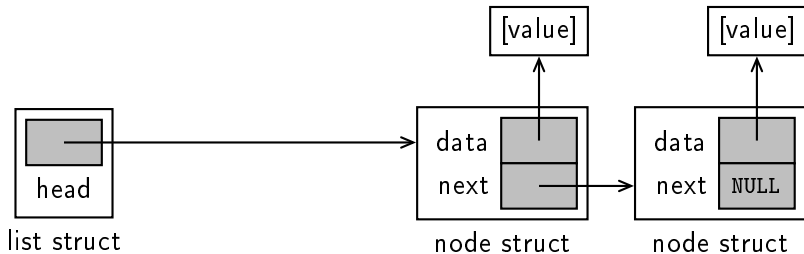
You can't do much with an empty list — we need to insert things into it:

- ▶ Create a new node (with `malloc()`).
- ▶ Point “data” to the thing we want to insert.
- ▶ Point “next” to the existing first node (which might be `NULL`)
  - ▶ Copy the value of “head”, which already points there.
- ▶ Point “head” to the new node.



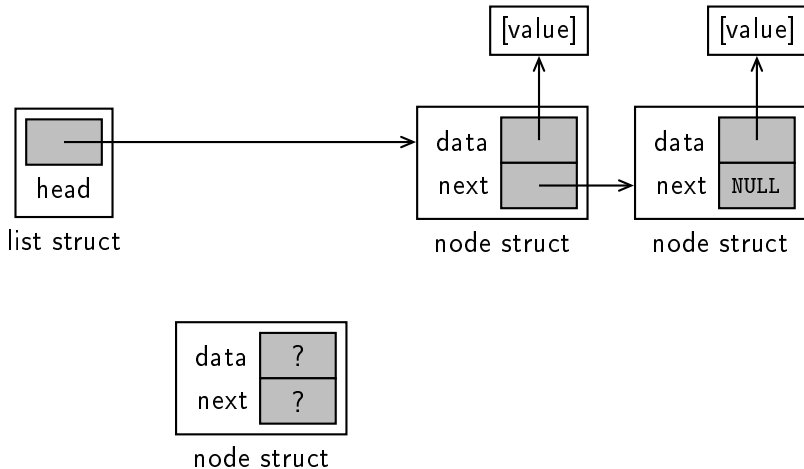
## Inserting an Element at the Start

Start with an existing linked list (which could be empty).



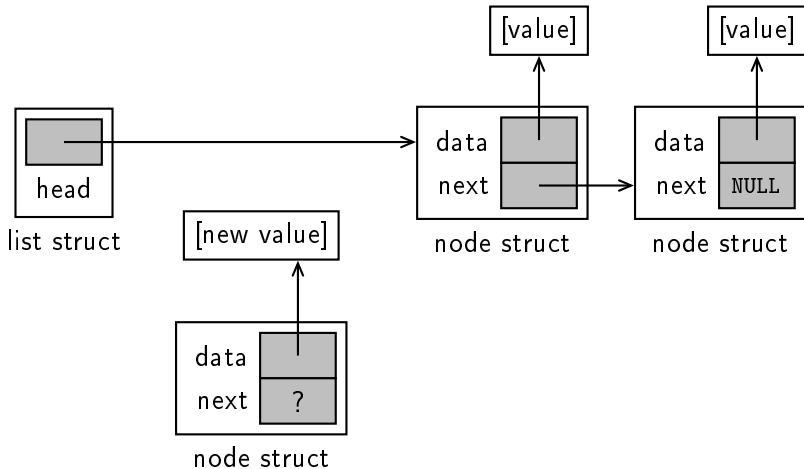
## Inserting an Element at the Start

Step 1: Create a new node.



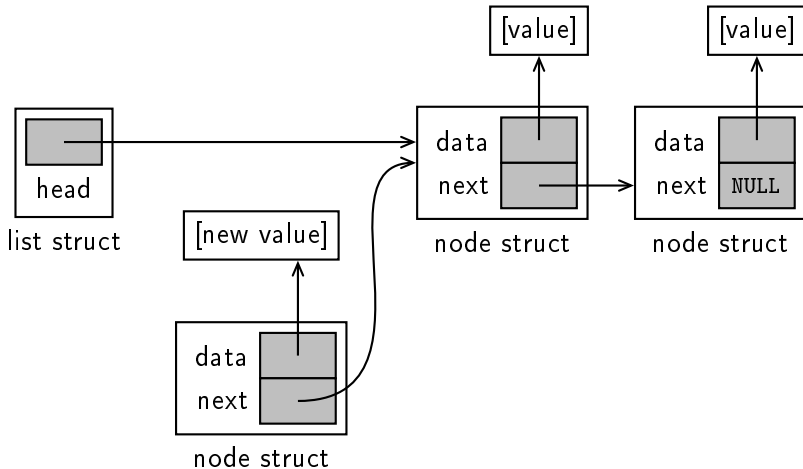
## Inserting an Element at the Start

Step 2: Point “data” to the thing we want to insert.



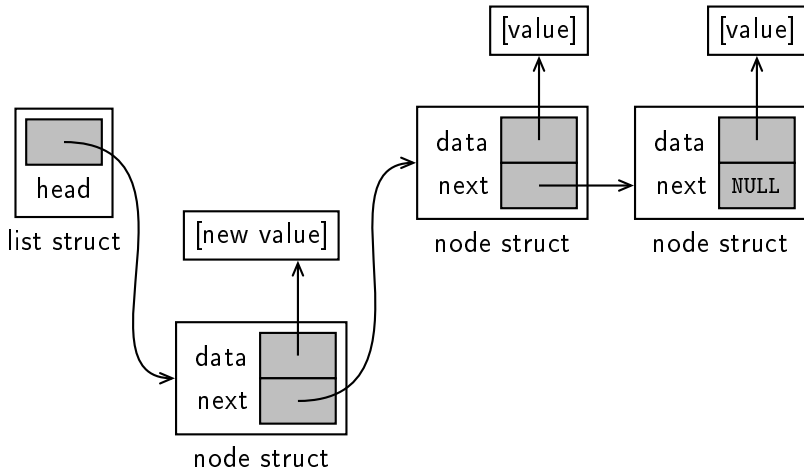
## Inserting an Element at the Start

Step 3: Point “next” to the existing first node.



## Inserting an Element at the Start

Step 4: Point “head” to the new node.



## Inserting an Element at the Start

```
/* Create a new node */  
newNode = (LListNode*)malloc(sizeof(LListNode));  
  
/* "Data" must point to the value to be inserted */  
(*newNode).data = ...;  
  
/* "Next" must point to the existing first node */  
(*newNode).next = head;  
  
/* Now head must point to this new node */  
head = newNode;
```

## Filling a Linked List

- ▶ To fill/initialise a linked list, simply insert multiple elements.
- ▶ Some sort of loop might be called for.
- ▶ We could insert data from:
  - ▶ calculated values
  - ▶ user input
  - ▶ an input file
  - ▶ anything else you can think up
- ▶ Keep in mind: inserting data only at the start will cause the elements to end up in **reverse order**.
- ▶ The first thing to be inserted will end up as the last element.
- ▶ (But you can often reverse the order of insertion to compensate!)

## Getting the Length of a Linked List

- ▶ Unlike arrays, the length of a linked list is stored, implicitly.
- ▶ You can find the length by counting the nodes, until you reach NULL:

```
int length = 0;
current = head;

while(current != NULL) {
    length++;
    current = (*current).next;
}
```

- ▶ Alternatively, you could store the length as a field inside the `LinkedList` struct.
- ▶ Update the length whenever you insert or delete an element.
- ▶ This would allow you to know the length instantly.



## A Brief Diversion into Recursion

- ▶ Time to demonstrate a very useful programming principle:

```
int listLength(LinkedListNode* node) {  
    int length = 0;  
    if(node != NULL)  
        length = 1 + listLength((*node).next);  
    return length;  
}
```

- ▶ This calculates the length of a linked list...without a loop.
- ▶ How?
- ▶ The function *calls itself*. Then the second copy of the function calls itself *again*, and *that* copy calls itself again, and so on.
- ▶ Each running copy of the function gets the next link in the list.
- ▶ Each copy gets the result of the next copy, and adds one.
- ▶ The first copy ends up with the total length.

## Deleting the First Element

- ▶ Make a copy of the head pointer.
- ▶ Set the head pointer to the second element.
- ▶ Use the copied pointer to free the first element (and its value if you need to).

```
temp = head;
head = (*head).next;

/* Free the first value. Think carefully about
whether you actually want to do this! */
free((*temp).data);

/* Free the first node. */
free(temp);
```

## Free or Return a Removed Element?

When you remove an element from the list:

- ▶ You can choose to:
  - ▶ immediately free/destroy the element itself (as well as the node); OR
  - ▶ keep the element around for some other purpose.
- ▶ You must choose! There's no "safe" answer.
- ▶ What you decide depends on the context.
- ▶ Does the list "own" its elements, or just "keep track" of them?
- ▶ (If you're using a linked list to implement a *stack*, you may not want to immediately destroy elements you pop off.)

## Other Insertion/Deletion Operations

Other common operations (covered in Data Structures & Algorithms – COMP1002) include:

- ▶ Inserting and deleting the last element.
- ▶ Inserting and deleting the  $n$ th element.

## Freeing a Linked List

- ▶ To deallocate a linked list, we must:
  - ▶ free each node; then
  - ▶ free the main list struct.
- ▶ You have to traverse the list while destroying it.
- ▶ This has to be done carefully, in a particular order.
- ▶ Very easy to create memory leaks here!
- ▶ As soon as you free a struct, you lose track of its contents, including any pointers.

## Freeing a Linked List (Iteratively)

You must free all the node structs, then the linked list struct:

```
void freeLinkedList(LinkedList* list) {  
    LinkedListNode *node, *nextNode;  
  
    node = (*list).head;  
    while(node != NULL) {  
        nextNode = (*node).next;  
        free((*node).data); /* <-- Maybe! */  
        free(node);  
        node = nextNode;  
    }  
  
    free(list);  
}
```

## Freeing a Linked List (Recursively)

```
/* Frees the whole list. */  
void freeLinkedList(LinkedList* list) {  
    freeNode((*list).head);  
    free(list);  
}  
  
/* Frees a given node and everything after it. */  
void freeNode(LinkedListNode *node) {  
    if(node != NULL) {  
        freeNode((*node).next); /* Recursive call. */  
  
        free((*node).data); /* <-- Maybe! */  
        free(node);  
    }  
}
```

## Linked Lists vs. Arrays

- ▶ Linked lists and arrays serve similar purposes.
- ▶ However, they are very different structures.
- ▶ Organisation:
  - ▶ An array is a single block of memory.
  - ▶ A linked list has many small blocks.
- ▶ Speed:
  - ▶ Insertion is faster with a linked list.
  - ▶ Accessing an element is faster with an array (except for the first element).
- ▶ Memory Usage:
  - ▶ Linked lists take up more memory per element (an additional two pointers, at least).
  - ▶ Storing small elements in a linked list is very inefficient.



# Coming Up

- ▶ The next lecture is about debugging.
- ▶ Start doing your assignment.