CSO-Recitation 06

CSCI-UA 0201-007

R06: Assessment 05 & Strings & Linked list

Today’s Topics

- Assessment 05
- Strings
- Linked list

Assessment 05

Q1 Basic C

Below are 4 C source files and their contents.

Q1.1 foo1.c

Which of the following statements are true?

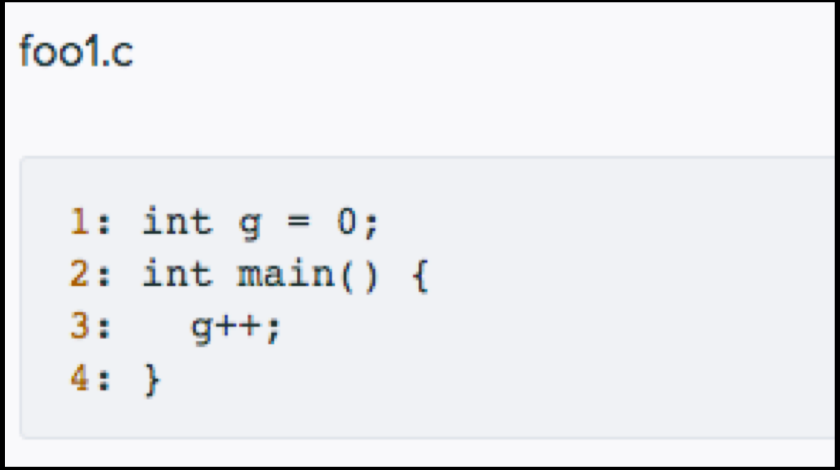
[✓ A.] The command gcc foo1.c creates a binary executable file called a.out

[✓ B.] The command gcc -c foo1.c creates a non-executable object file called foo1.o

C. The command gcc foo1.c results in an error.

[✓ D.] After executing line 3, variable g has value 1.

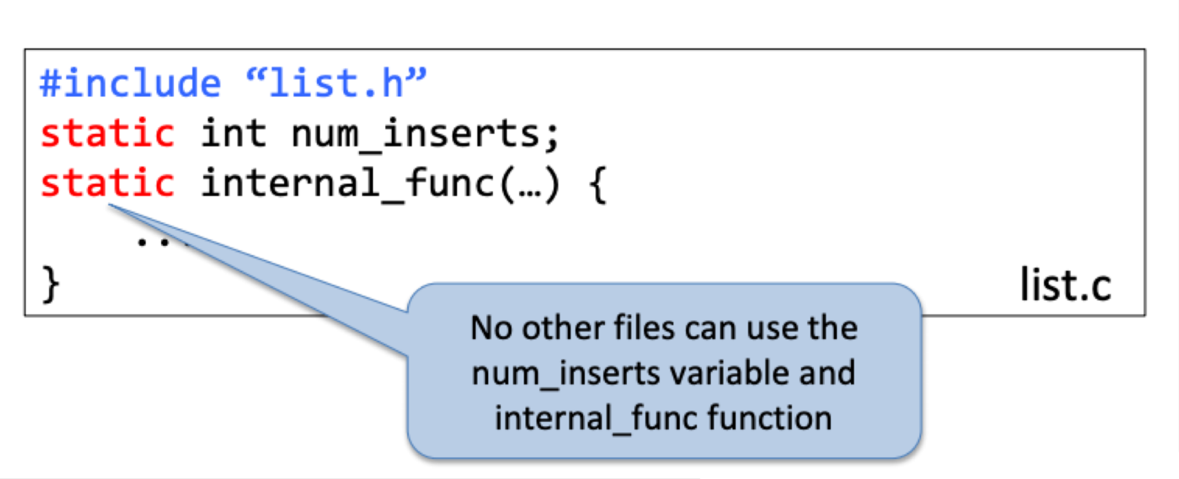
E. After executing line 3, variable g could have any value.



foo1.c

```
1: int g = 0;
2: int main() {
3:    g++;
4: }
```

Q2 Static and extern



```
#include "list.h"
static int num_inserts;
static internal_func(…) {
    ...
}
```

list.c

No other files can use the num_inserts variable and internal_func function

Q2.1 foo2.c

Which of the following statements are true?

A. The command gcc foo2.c creates a binary executable file called a.out.

[✓ B.] The command gcc -c foo2.c creates a non-executable object file called foo2.o.

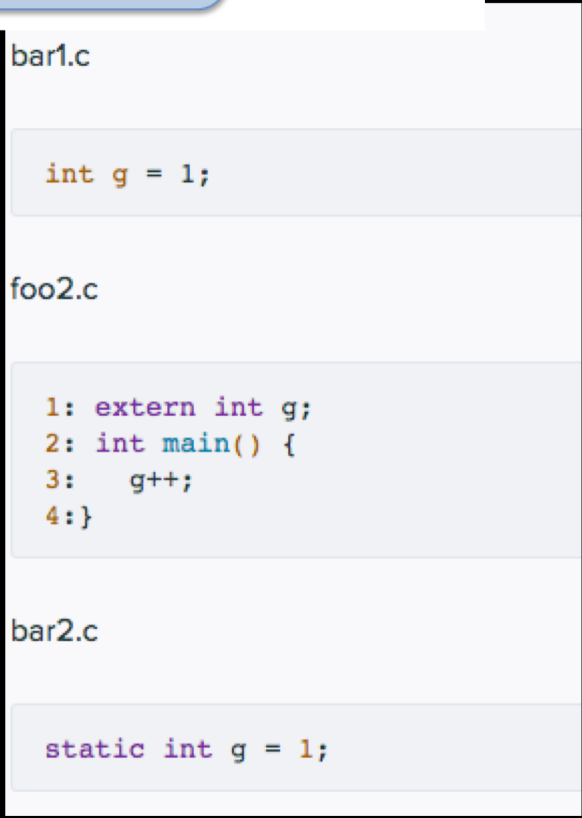
[✓ C.] The command gcc foo2.c results in an error.

[✓ D.] The command gcc foo2.c bar1.c creates a binary executable file called a.out.

E. The command gcc foo2.c bar1.c results in an error.

F. The command gcc foo2.c bar2.c creates a binary executable file called a.out.

[✓ G.] The command gcc foo2.c bar2.c results in an error.



bar1.c

```
int g = 1;
```

foo2.c

```
extern int g;
int main() {
   g++;
}
```

bar2.c

```
static int g = 1;
```

Q2 Static and extern

Q2.2

Suppose this command gcc foo2.c bar1.c bar2.c generates executable a.out, which of the following is true about executing a.out?

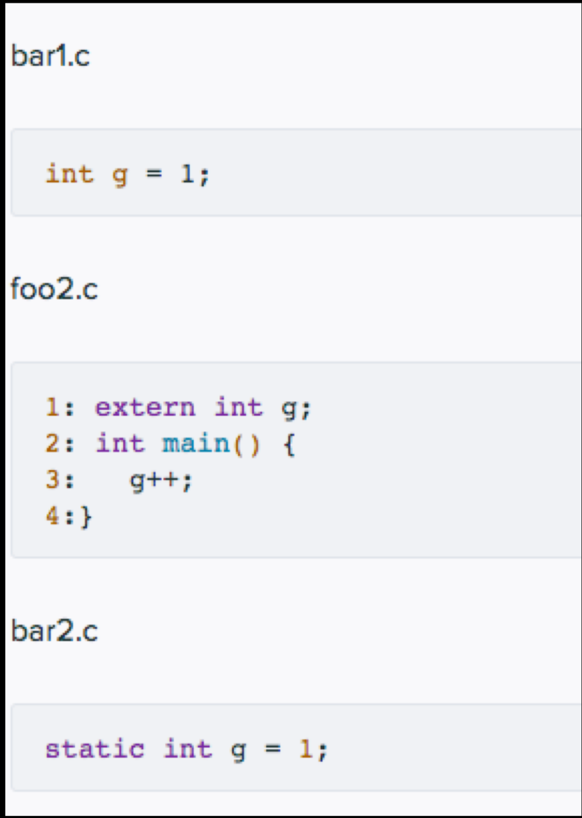
A. The global variable g in bar1.c and bar2.c have the same underlying same memory location.

[✓ B.] The global variable g in bar1.c and bar2.c have different underlying same memory locations.

[✓ C.] The variable g in foo2.c refers to the global variable g defined in bar1.c.

D. The variable g in foo2.c refers to the global variable g defined in bar2.c.

E. The command gcc foo2.c bar1.c bar2.c would result in an error.



bar1.c

```
int g = 1;
```

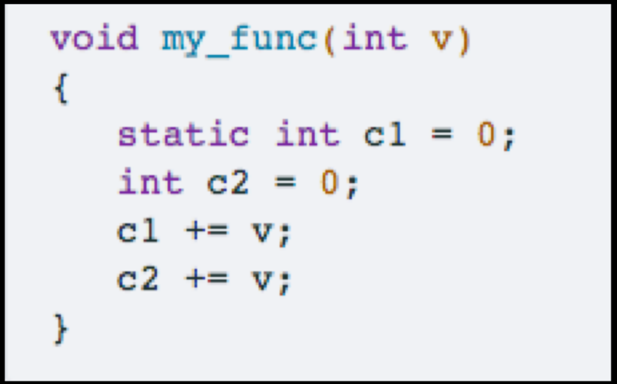
foo2.c

```
extern int g;
int main() {
   g++;
}
```

bar2.c

```
static int g = 1;
```

Q3 Static for local variable



```
void my_func(int v)
{
    static int c1 = 0;
    int c2 = 0;
    c1 += v;
    c2 += v;
}
```

The following shows the code for function my_func

Q3.1 basic

A. Local variable c1 is allocated upon each invocation of my_func and de-allocated upon its return.

[✓ B. Local variable c2 is allocated upon each invocation of my_func and de-allocated upon its return.]

C. Local variable c1 and c2 always have the same value right before the return of my_func.

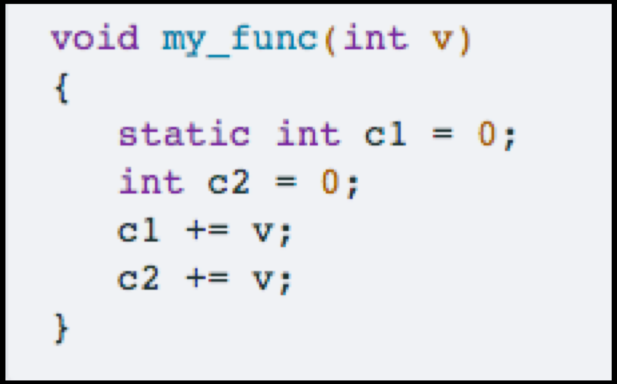
[✓ D. Local variable c1 has scope within function my_func and cannot be referred to from outside of this function.]

[✓ E. Local variable c2 has scope within function my_func and cannot be referred to from outside of this function.]

When "static" prefix local variables:

- Initialized once, never deallocated
- Any change persists across function invocations
- like a global variable, except with local scope

Q3 Static for local variable



```
void my_func(int v)
{
    static int c1 = 0;
    int c2 = 0;
    c1 += v;
    c2 += v;
}
```

Suppose one executes the following code snippet:

```
my_func(10);
my_func(20);
```

Which of the following statements are true?

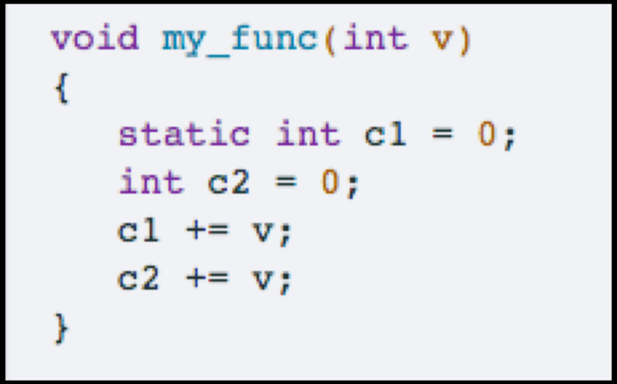
A. Right before returning from my_func(20), variable c1 has value 20.

[✓ B. Right before returning from my_func(20), variable c1 has value 30.]

[✓ C. Right before returning from my_func(20), variable c2 has value 20.]

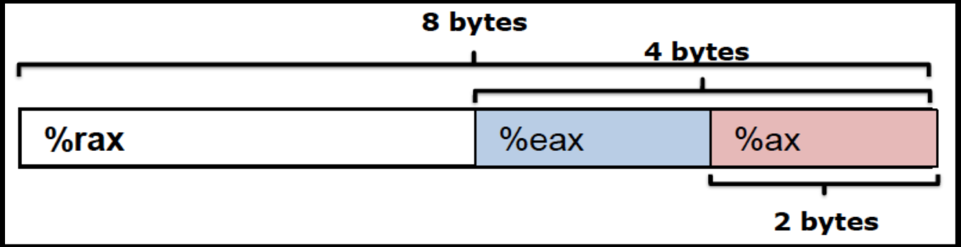
D. Right before returning from my_func(20), variable c2 has value 30.

Execution Breakdown



```
void my_func(int v)
{
    static int c1 = 0;
    int c2 = 0;
    c1 += v;
    c2 += v;
}
```

- c1 is allocated and assigned with value 0 // c1=0
- my_func(10)
  - c2 allocated and assigned with value 0 // c2=0
  - c1+=10 // c1=10
  - c2+=10 // c2=10
  - Function return (c2 is de-allocated)
- my_func(20)
  - c2 allocated and assigned with value 0 // c2=0
  - c1+=10 // c1=30
  - c2+=20 // c2=20



Q4 register

After x86 CPU executes instruction movq $0x12345678, %rax, which of the following is true?

[✓ A. The higher order 4-byte of register %rax are all zeros.]

B. The higher order 4-bytes of register %rax remain the same as before the movq instruction is executed.

C. Register %eax has value 0x00000000

[✓ D. Register %eax has value 0x12345678]

E. Register %eax is not changed by the movq instruction.

F. Register %ax has value 0x1234

[✓ G. Register %ax has value 0x5678]

Q5 mov

Suppose register %rax stores C variable long *x. Which of the following instruction corresponds to the C statement *x = 10;

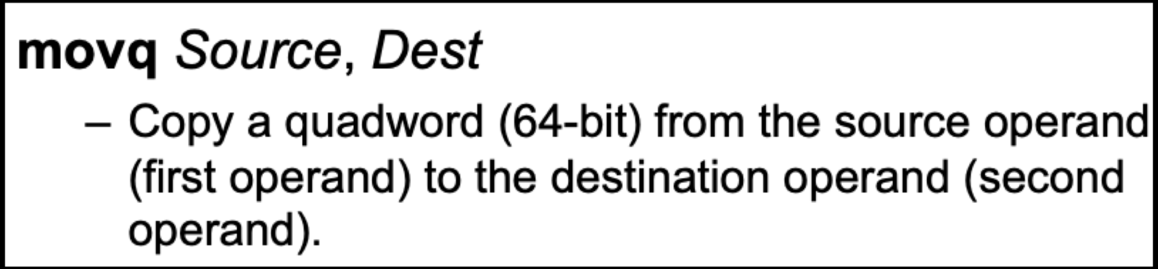
A. movq $10, %rax

[✓ B. movq $10, (%rax)]

C. movq (%rax), $10

D. movq %rax, $10

- long *x
  - x is a pointer to long (8 bytes, 64 bits)
- *x=10;
  - de-referencing x, assign the value 10
- x is an address stored in %rax, use (%rax) to deference it.



movq *Source*, *Dest*

– Copy a quadword (64-bit) from the source operand (first operand) to the destination operand (second operand).

Q6 mov

Suppose register %rax stores C variable int x. Which of the following instruction corresponds to the C statement x = 10;

[✓ A. movl $10, %eax]

B. movq $10, %eax

C. movl $10, (%rax)

D. movq $10, (%rax)

- int x
  - x is an integer with 4 bytes
- x=10;
  - assign the value 10 to x
- x is a variable stored in %eax

Q7 mov

Given instruction movl $eax, (%rbx), what are likely data types for the variable stored in %rbx?

A. long

B. unsigned long

C. int

D. unsigned int

[✓ E. int*]

[✓ F. unsigned int*]

G. long*

H. unsigned long*

- (%rbx)
  - Deference %rbx => %rbx stores a pointer
- movl
  - 4 bytes => %rbx stores a pointer which points to a data of 4 bytes
- long is 8 bytes
- the movl instruction does not distinguish between signed/unsigned

Q8 mov

Which of the following statements are true?

A. During a program's execution, its instructions are stored on disk while its program data is stored in the memory.

[✓ B.] During a program's execution, both its instructions and program data are stored in the memory.

C. Compilers must generate explicit instructions to increment PC (aka %rip)

[✓ D.] CPU automatically increments PC (aka %rip) as instructions are executed.

E. An executable file compiled for ARM can be directly executed by an x86 CPU.

[✓ F.] An executable file compiled for ARM can not be directly executed by an x86 CPU.

Strings

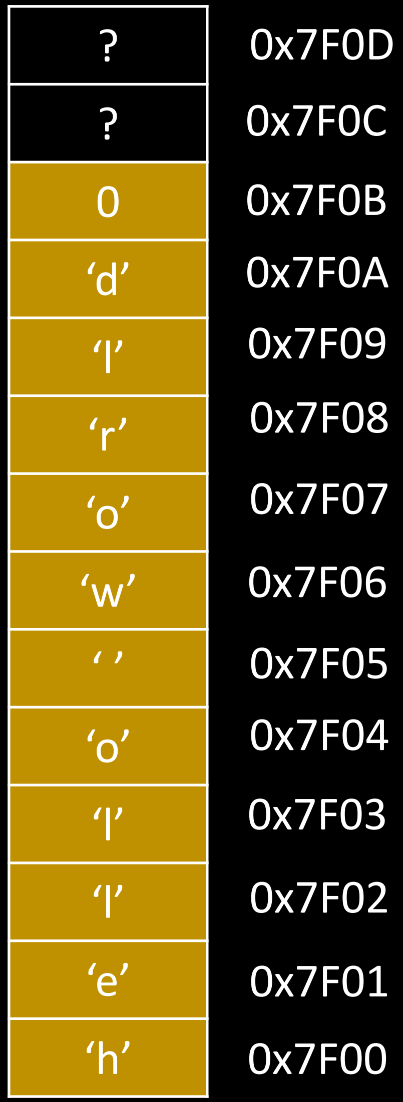
Arrays of chars

What are strings?

- They are arrays of the type *char*, which is typically one byte
- Char literals are in single quotes ‘ ’
- String literals are in double quotes “ “
- Unlike other arrays, strings have a way of knowing the length even at runtime
    - Strings are stored with the last byte set to 0 (or ‘\0’)
        - C strings are called “null terminated”
        - So you can find the length by looping over the string, keeping a counter, and stopping when you find a char equal to zero
    - There is also a standard library function for this, *strlen*

Defining a string

- char *arr = “hello world”;
- char arr[12] = “hello world”;
- The literal “hello world” includes the null-terminator.

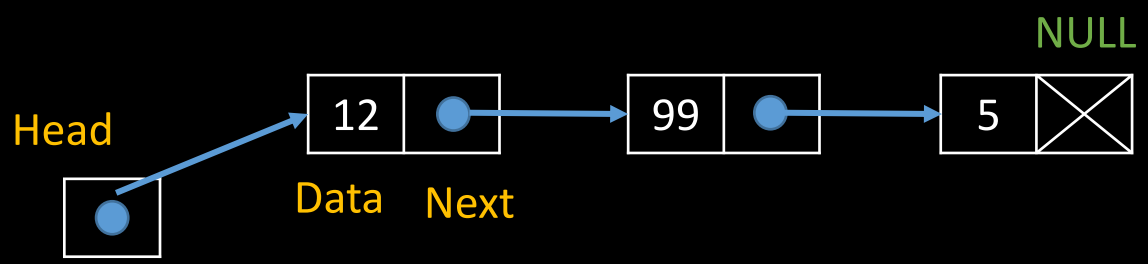


?	0x7F0D
?	0x7F0C
0	0x7F0B
‘d’	0x7F0A
‘l’	0x7F09
‘r’	0x7F08
‘o’	0x7F07
‘w’	0x7F06
‘ ’	0x7F05
‘o’	0x7F04
‘l’	0x7F03
‘l’	0x7F02
‘e’	0x7F01
‘h’	0x7F00

Linked list

A linear data structure

Why linked list?



- Like arrays, Linked List is a linear data structure.
- Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.
- Arrays have limitations:
    - The size of the arrays are fixed (pre-defined)
    - Inserting (Deleting) a new element in an array of elements is expensive
        - because the room has to be created for the new elements and existing elements have to be shifted.

Advantages and Drawbacks

- Advantages over arrays:
    - Dynamic size
    - Ease of insertion/deletion
- Drawbacks:
    - Random access is not allowed
        - We have to access elements sequentially starting from the first node. (Traverse)
    - Extra memory space for a pointer is required with each element of the list.
    - Not cache friendly
        - Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Linked list

- A linked list is represented by a pointer to the first node of the linked list
  - It is called the *head*
  - If the linked list is empty, then the value of the head is NULL
- Each node in a list consists of at least two parts:
  - data
  - Pointer (or Reference) to the next node
- In the case of the last node in the list,
  - the next field contains NULL - it is set as a null pointer.
- In C, we can represent a node using struct
  - nodes are defined as (e.g.) node using *typedef*
  - *node *head*

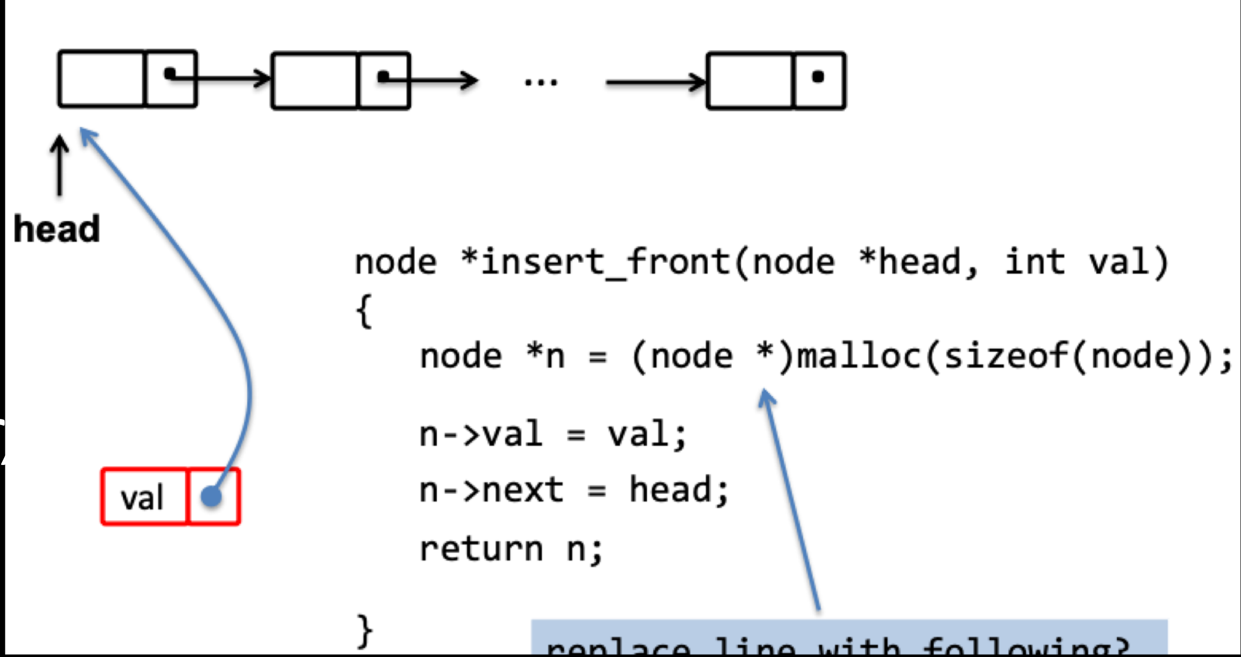
Initialize the linked list

- The list is initialized by creating a *node *head* which is set to NULL
- The variable *head* is now a pointer to NULL, but as nodes are added to the list, *head* will be set to point to the first node
- In this way, *head* becomes the access point for sequential access to the list.

Linked list

- Linked list insertion
- Linked list Deletion
- Search an element in a linked list
- Traverse a linked list
- Find length of a linked list
- …

Linked list

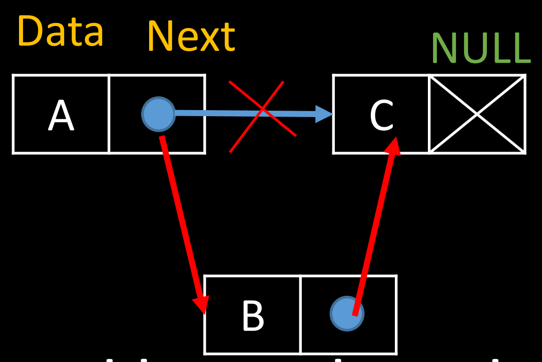


```
node *insert_front(node *head, int val)
{
    node *n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = head;
    return n;
}
```

replace line with following?

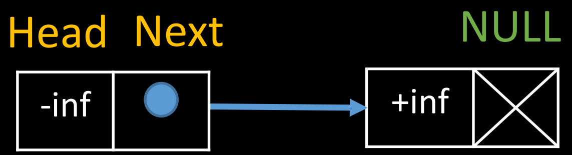
- In class, we pass the header pointer,
  - ask it to return a new head
  - the caller is responsible for updating it itself
- In lab-2, we pass a pointer to pointer parameter (pointer to the head pointer),
  - to allow changing the head pointer directly instead of returning the new one
  - note that there’s no return value; It’s not needed.

Inserting a node

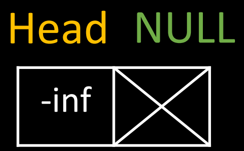


- How can we insert a node in a linked list sorted by each node's data?
- Assume data are all unique
- Four cases
  - List is empty: insert_front
  - Smaller than the head: insert_front
  - Larger than some node with data A but smaller than A's next node (data C):
    - Insert a node after A before C
  - Larger than all nodes:
    - Insert a node at the end of the linked list
- Too many corner cases! Any tricks to simplify it (to one case)?
  - Sentinel node

Inserting a node



Q: What if we only add one sentinel node with –inf, instead of both –inf and +inf?



Four cases

~~List is empty: insert_front~~

~~Smaller than the head: insert_front~~

Larger than some node with data A but smaller than A's next node (data C):

Insert a node after A before C

~~Larger than all nodes:~~

~~Insert a node at the end of the linked list~~

Dynamic memory allocation

- Each time you need to manually allocate data, use *malloc*
    - void *malloc(size_t size);
- If you need to manually de-allocate
    - void free(void *ptr);

More on linked list

- Implement a hash table
  - see clear instructions on our website lab-2 page
- A hash table is an array of linked lists with a hash function
  - A hash function basically just takes things and puts them in different “buckets” (hash table’s array of entries)
  - Each “bucket" just points to a linked list here