CSE 29
Lecture 16 Summary

February 26, 2026

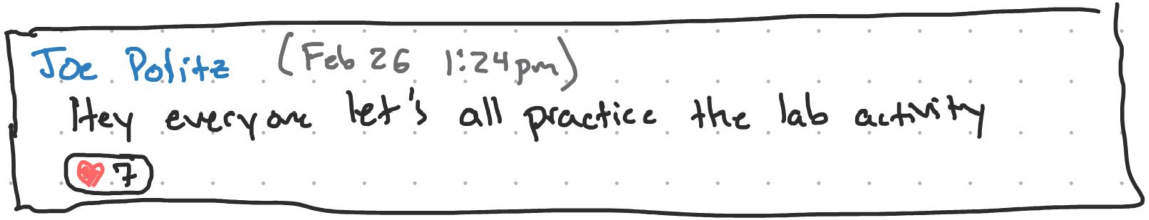
📣 Logistical Things

- Nothing today!

🧠 Review Questions

Answers in next slide!

Q1: Design a struct to represent a message/chat in a chat room

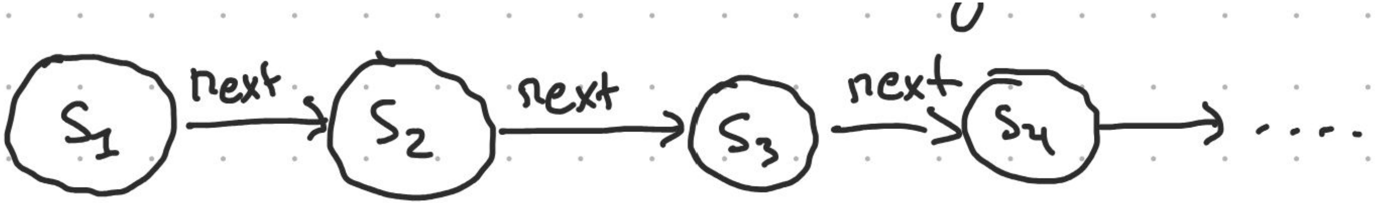


Joe Politz (Feb 26 1:24pm)

Hey everyone let's all practice the lab activity

❤ 7

Q2: Design a struct to represent a node in a linked list with string values



Q3: what is `sizeof` for each struct you designed?

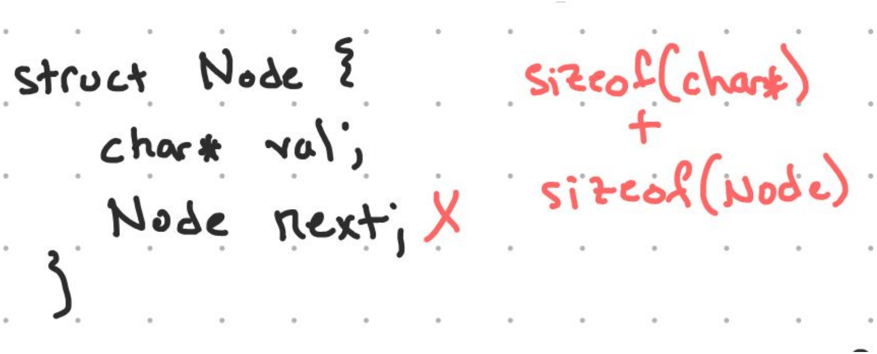
🧠 RQ 1 Answer

```
struct Chat {
    char* name;  8
    char date[16];  16   char time[16];  16
    char* message_content;  8
    int id;  4
    Reaction reactions[1024];
}
```

```
struct Reaction {   4
    char symbol[4];
    int count;  4
}
```

🧠 RQ 2 Answer

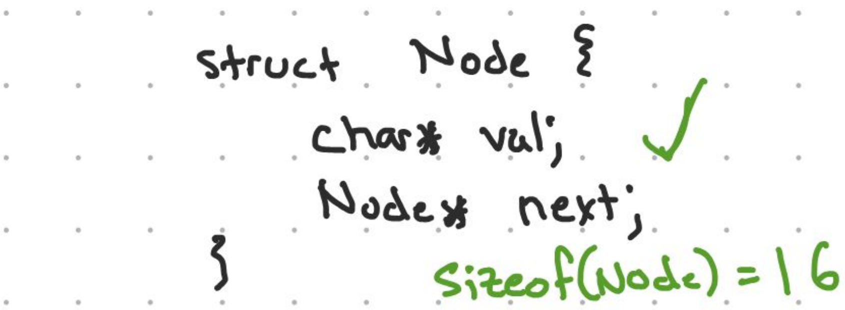
Will not work



```
struct Node {
    char* val;
    Node next; ✗
}
```

sizeof(char*)
+
sizeof(Node)

Will work

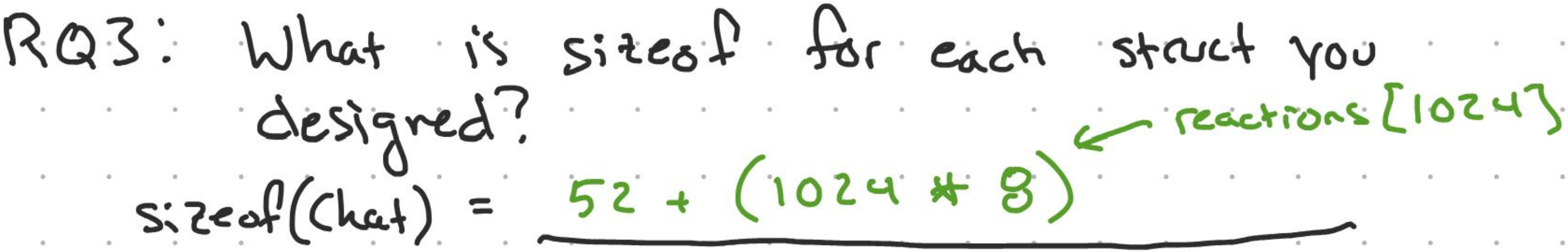


```
struct Node {
    char* val;     ✓
    Node* next;
}
```

sizeof(Node) = 16

A struct definition must have a determined size

🧠 RQ 3 Answer



RQ3: What is sizeof for each struct you designed?

sizeof(Chat) = 52 + (1024 * 8) ← reactions[1024]

This is unreasonably large for the size of each chat. Instead of storing an array of these we can store a pointer to a `linked list` (a list of nodes where each one has a pointer to the next in the list like RQ2)

Why do we use malloc?

Questions from Exit Slips

- When and why do we use `malloc`?
  - With a class in Python or Java, every object we make is heap allocated
  - We need addresses, and we can't have stack allocated addresses, so we must do heap allocation
  - We need a persistent area of memory, which is the heap
- What is the type `time_t`?
  - It is the number of milliseconds since the new year in 1970. More on this in the PA5 writeup!
    - It is a 4 byte value and will overflow at 3:14:07 UTC on 19 January 2038
    - We will have to increase it to 8 bytes.

What a node would look like in other languages

```
struct Node {
    char* val;
    Node* next;
}
```

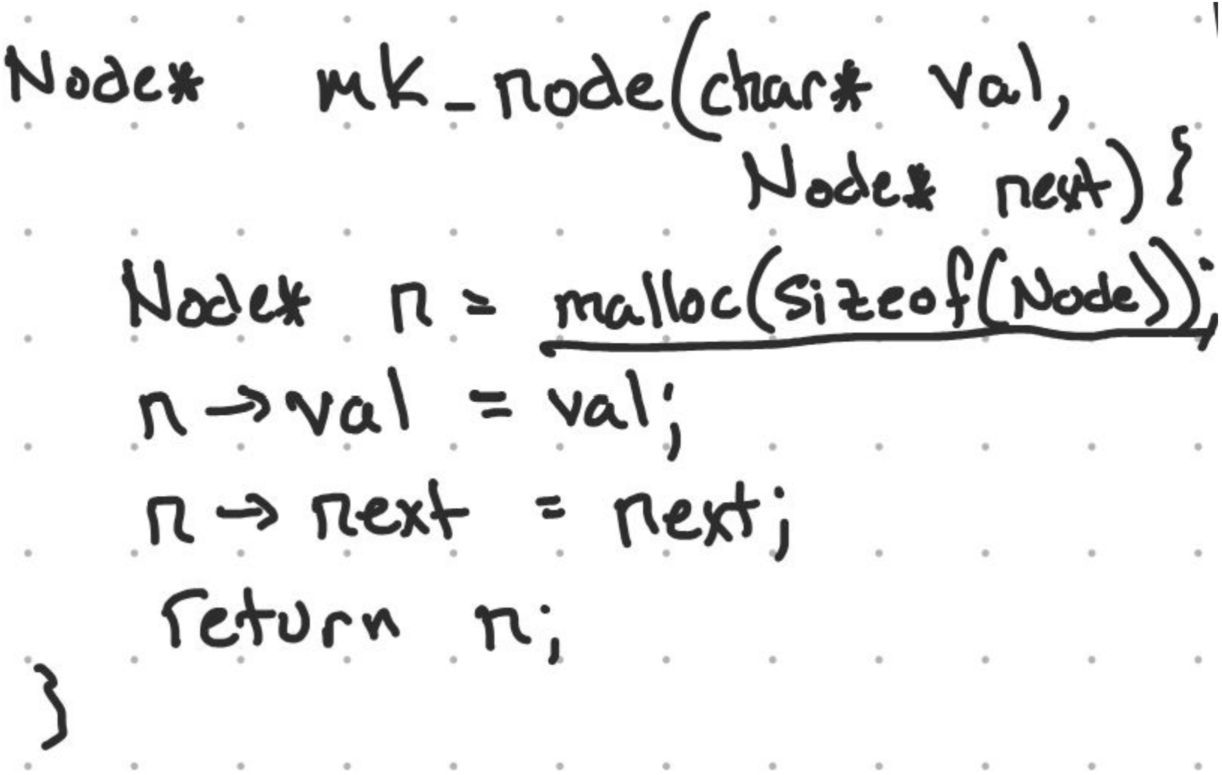
Python

```
class Node
    def _init_(self, val, next):
        ....

n = Node("abc", None)
```

Java

```
class Node {
    public Node (String val, Node next) {
        ...
    }
}

Node n = new Node("abc", null)
```

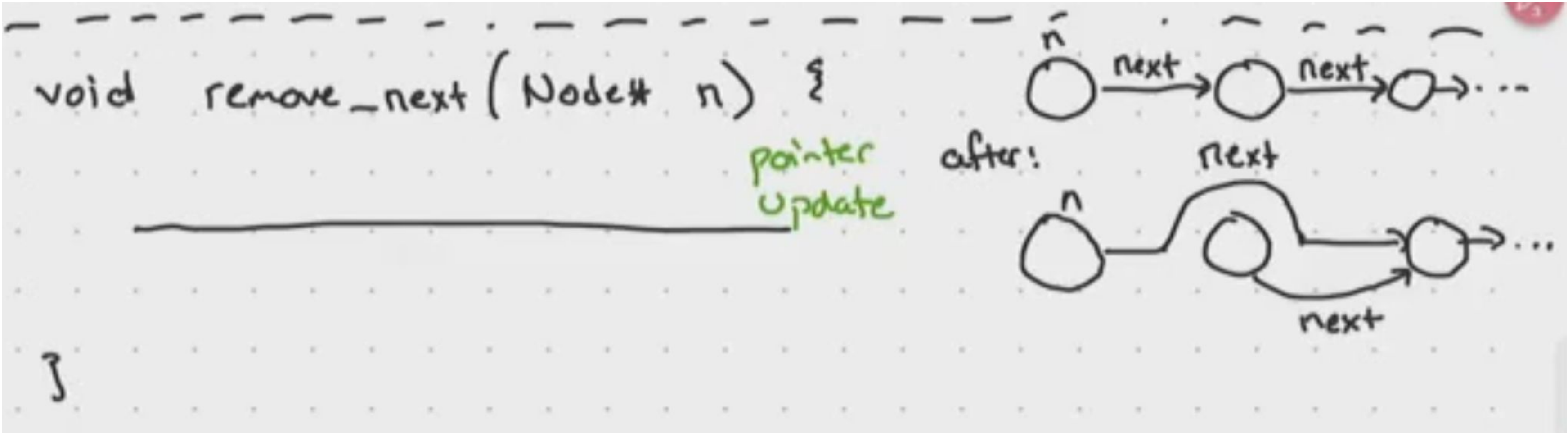
How to make a new node



```
Node*  mk_node(char* val,
                     Node* nest){
   Node* n = malloc(sizeof(Node));
   n->val  = val;
   n->next  = next;
   return n;
}
```

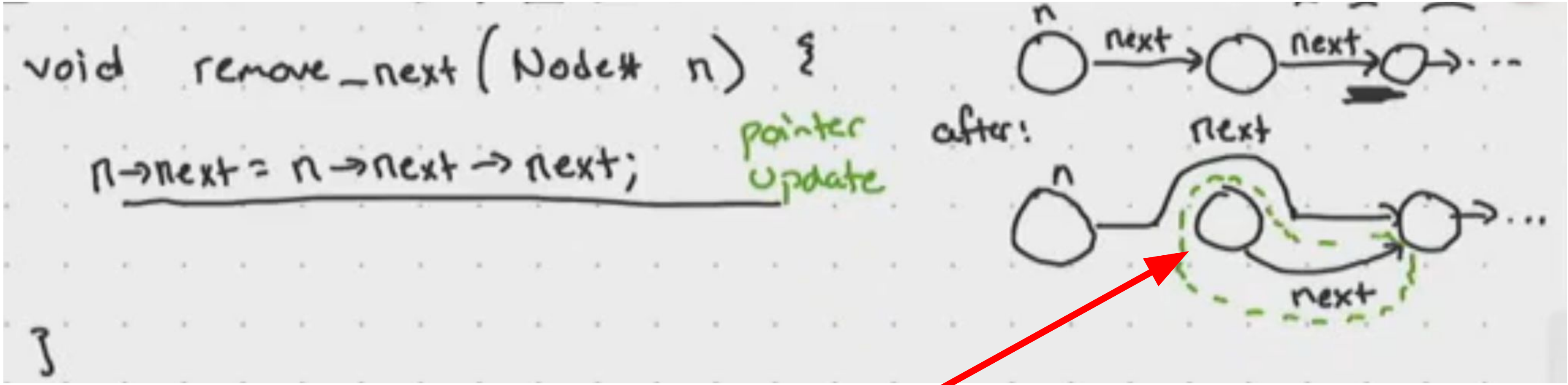
Removing a Node

Given Node* n, how can we remove the next Node from our linked list?



```
void remove_next(Node* n) {

    ______________________________ pointer update

}
```

Removing a Node leaky Solution

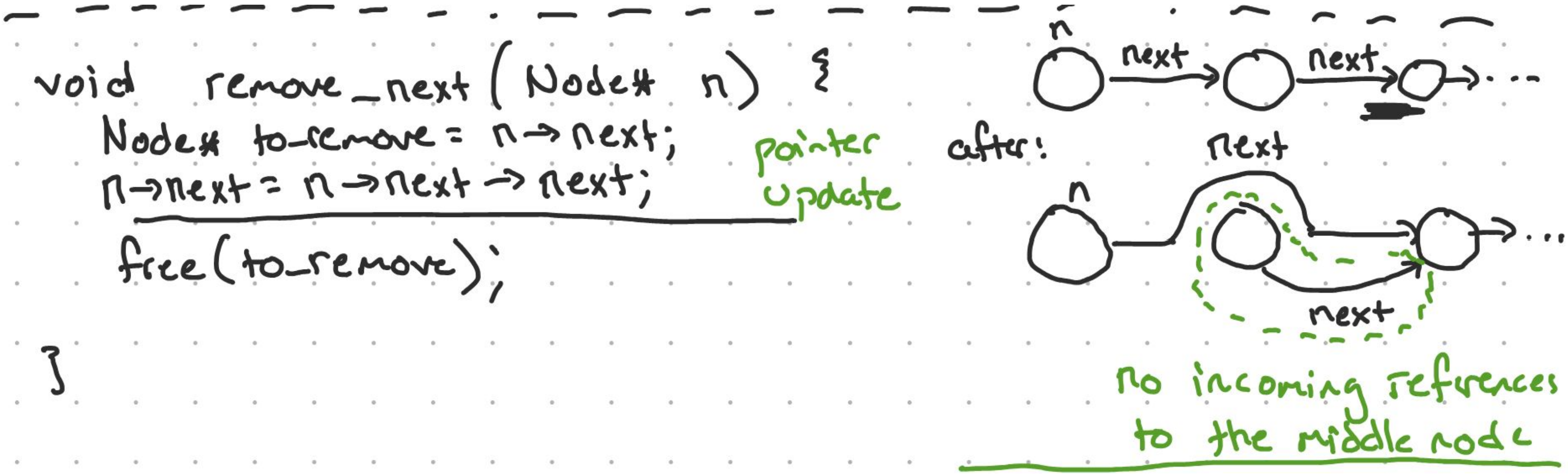


```
void remove_next(Node* n) {
    n->next = n->next->next;
}
```

Notice that this node still exists on the heap but we can no longer access it. It is a memory leak. How can we fix this? (Answer on next slide)

Removing a Node Solution

Insert solution screenshot



```
void remove_next(Node* n) {
    Node* to_remove = n->next;
    n->next = n->next->next;
    free(to_remove);
}
```

pointer update

no incoming references to the middle node

Questions

- Could we just add a number to find the next node?
    - No, there’s no guarantee that the nodes are adjacent with each other
- How does Java and Python manage memory?
    - Java uses garbage collection: loops over the stack to look at each pointer and then loops over the heap to and frees things that no longer has a reference to it and frees it
    - Python does reference counting: increments and decrements an integer related to number of references to an object and frees when the number of references is 0
- In order to just write “Node n;” and not “struct Node n;” don’t we need the `typedef`?
    - Yes, in actual code this would be necessary but to simplify the notes it has been omitted for concision.

Wrap Up

Python and Java look at things without incoming references and automatically frees it. This is automatic memory management

- Take CSE 131 with Joe if you’re interested!

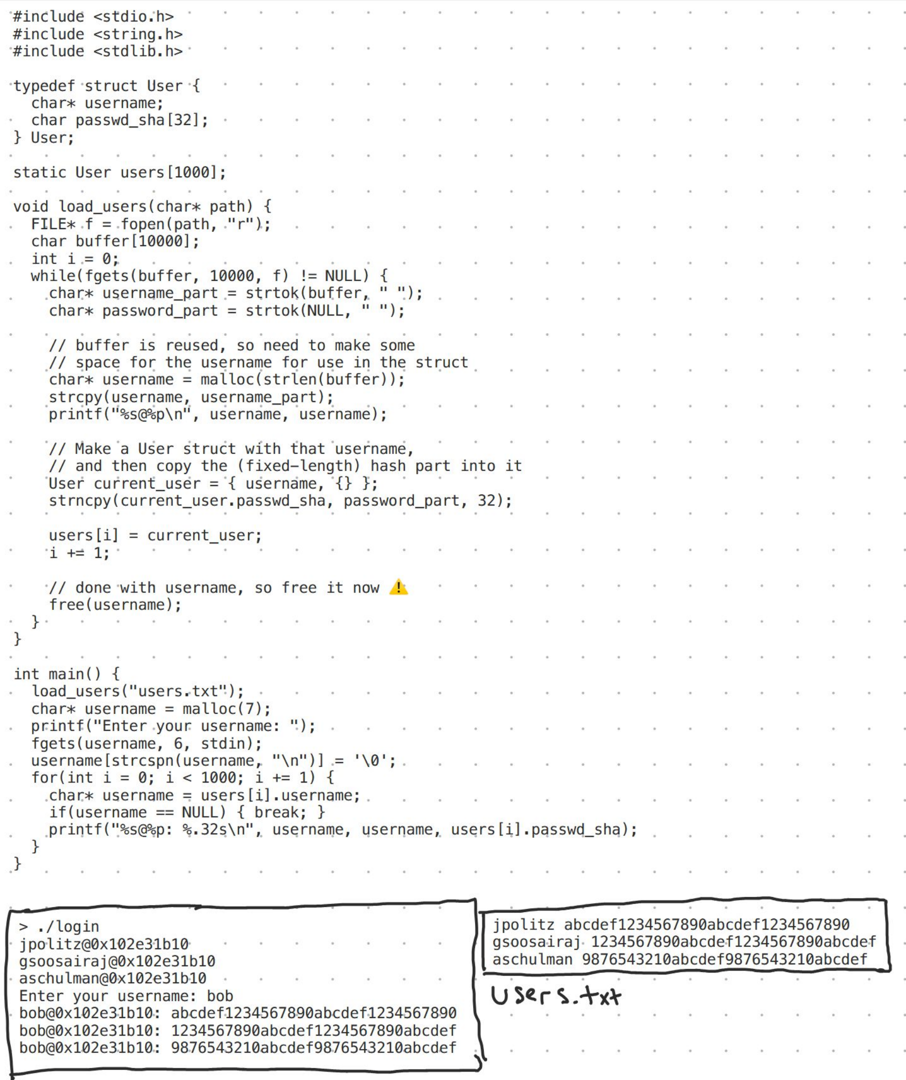
Use-After-Free Error

Let’s look at the handout

Look through and comprehend what the code is doing

https://ucsd-cse29.github.io/wi26/lec/02-26-badfree/lecture.pdf

(Blank Handout)



```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct User {
  char* username;
  char passwd_sha[32];
} User;

static User users[1000];

void load_users(char* path) {
  FILE* f = fopen(path, "r");
  char buffer[10000];
  int i = 0;
  while(fgets(buffer, 10000, f) != NULL) {
    char* username_part = strtok(buffer, " ");
    char* password_part = strtok(NULL, " ");

    // buffer is reused, so need to make some
    // space for the username for use in the struct
    char* username = malloc(strlen(buffer));
    strcpy(username, username_part);
    printf("%s@%p\n", username, username);

    // Make a User struct with that username,
    // and then copy the (fixed-length) hash part into it
    User current_user = { username, {} };
    strncpy(current_user.passwd_sha, password_part, 32);

    users[i] = current_user;
    i += 1;

    // done with username, so free it now ⚠️
    free(username);
  }
}

int main() {
  load_users("users.txt");
  char* username = malloc(7);
  printf("Enter your username: ");
  fgets(username, 6, stdin);
  username[strcspn(username, "\n")] = '\0';
  for(int i = 0; i < 1000; i += 1) {
    char* username = users[i].username;
    if(username == NULL) { break; }
    printf("%s@%p: %.32s\n", username, username, users[i].passwd_sha);
  }
}
```

```
> ./login
jpolitz@0x102e31b10
gsoosairaj@0x102e31b10
aschulman@0x102e31b10
Enter your username: bob
bob@0x102e31b10: abcdef1234567890abcdef1234567890
bob@0x102e31b10: 1234567890abcdef1234567890abcdef
bob@0x102e31b10: 9876543210abcdef9876543210abcdef
```

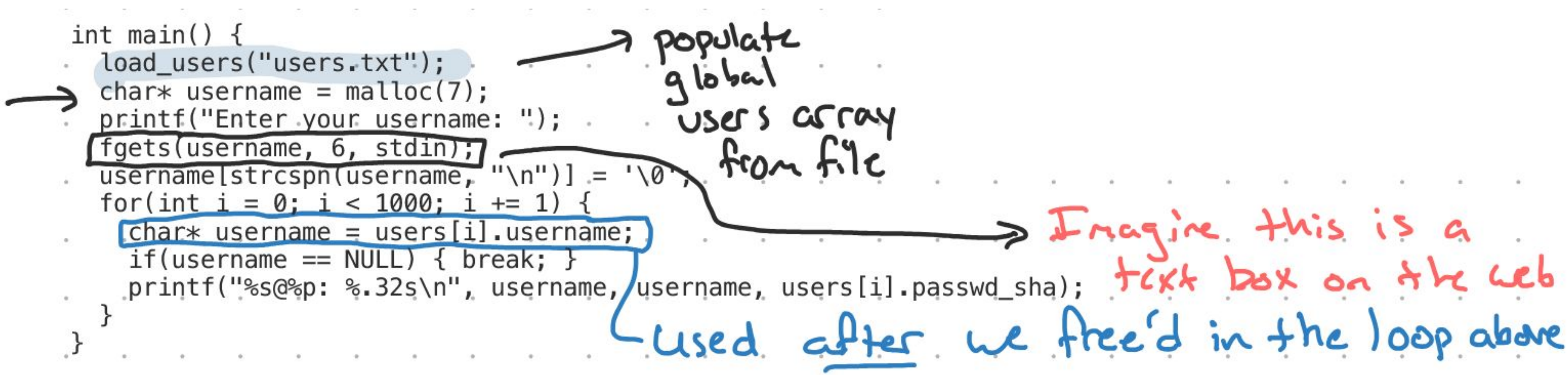
```
jpolitz abcdef1234567890abcdef1234567890
gsoosairaj 1234567890abcdef1234567890abcdef
aschulman 9876543210abcdef9876543210abcdef
```

Users.txt

Lets look at `main`

Populates global users array from file

Where user puts in username and logs in



```
int main() {
  load_users("users.txt");
  char* username = malloc(7);
  printf("Enter your username: ");
  fgets(username, 6, stdin);
  username[strcspn(username, "\n")] = '\0';
  for(int i = 0; i < 1000; i += 1) {
    char* username = users[i].username;
    if(username == NULL) { break; }
    printf("%s@%p: %.32s\n", username, username, users[i].passwd_sha);
  }
}
```

populate global users array from file

Imagine this is a text box on the web

Used after we free'd in the loop above

Let’s look at `load_users`

```
void load_users(char* path) {
  FILE* f = fopen(path, "r");
  char buffer[10000];
  int i = 0;
  while(fgets(buffer, 10000, f) != NULL) {
    char* username_part = strtok(buffer, " ");
    char* password_part = strtok(NULL, " ");

    // buffer is reused, so need to make some
    // space for the username for use in the struct
    char* username = malloc(strlen(buffer));
    strcpy(username, username_part);
    printf("%s@%p\n", username, username);

    // Make a User struct with that username,
    // and then copy the (fixed-length) hash part into it
    User current_user = { username, {} };
    strncpy(current_user.passwd_sha, password_part, 32);

    users[i] = current_user;
    i += 1;

    // done with username, so free it now ⚠️
    free(username);
  }
}

int main() {
  load_users("users.txt");
→ char* username = malloc(7);
  printf("Enter your username: ");
  fgets(username, 6, stdin);
```

read line by line

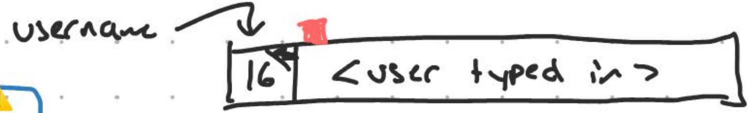
interior ptrs into buffer
internal

heap-allocated string for username

copying fixed 32-byte pw_hash

not done with the data at 🟥 0x...31b10

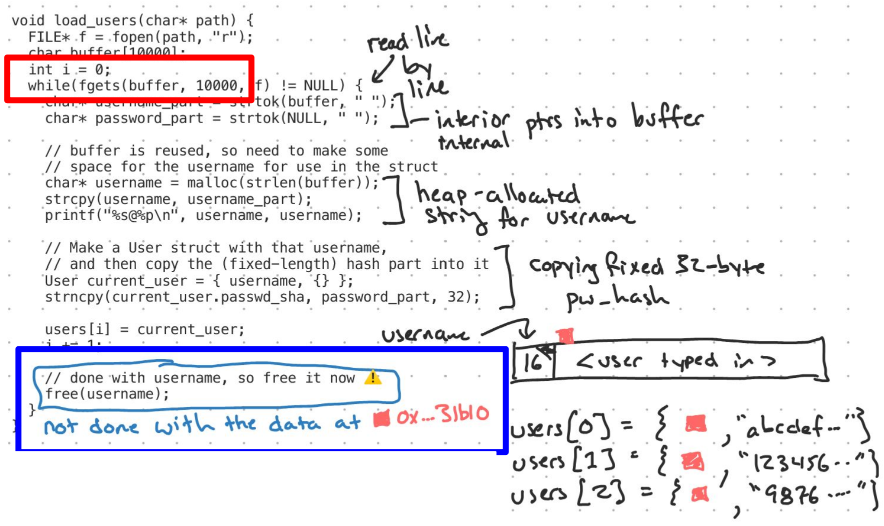
populate global users array



users[0] = { 🟥 , "abcdef..." }
users[1] = { 🟥 , "123456..." }
users[2] = { 🟥 , "9876...." }

Let’s look at `load_users`

`buffer` is a stack allocated array and so it could be overwritten later, so we don’t want to store pointers to the `buffer`



```
void load_users(char* path) {
  FILE* f = fopen(path, "r");
  char buffer[10000];
  int i = 0;
  while(fgets(buffer, 10000, f) != NULL) {
    char* username_part = strtok(buffer, " ");
    char* password_part = strtok(NULL, " ");

    // buffer is reused, so need to make some
    // space for the username for use in the struct
    char* username = malloc(strlen(buffer));
    strcpy(username, username_part);
    printf("%s@%p\n", username, username);

    // Make a User struct with that username,
    // and then copy the (fixed-length) hash part into it
    User current_user = { username, {} };
    strncpy(current_user.passwd_sha, password_part, 32);

    users[i] = current_user;
    i += 1;

    // done with username, so free it now ⚠
    free(username);
  }
}
```

read line by line
interior ptrs into buffer
internal
heap-allocated string for username
copying fixed 32-byte pw_hash
username
16
<user typed in>
not done with the data at 🟥 0x...31b10
users[0] = { 🟥, "abcdef--"}
users[1] = { 🟥, "123456.."}
users[2] = { 🟥, "9876----"}

This free seems like it makes sense, but it’s not actually good

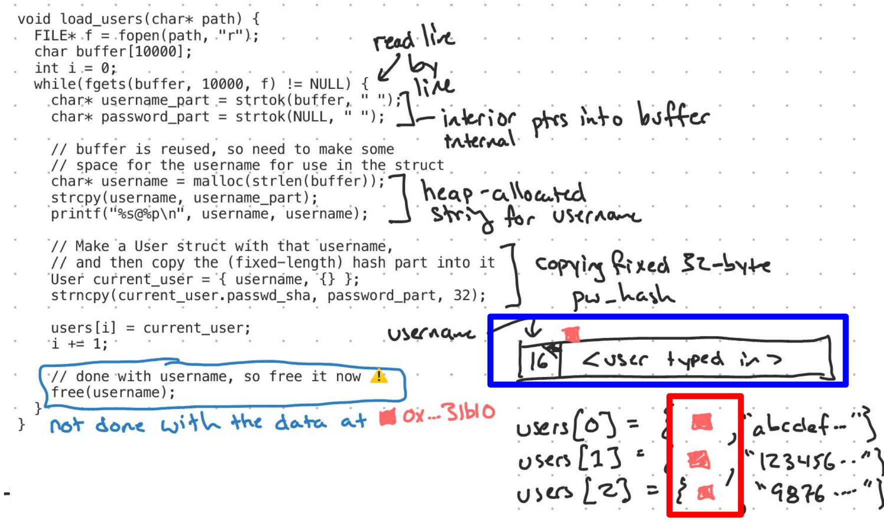
- When we free, we promise that we won’t use that address reference again

Here’s what happening in the heap

The same block in the heap is being used for all three usernames: users[0], users[1], users[2] all have the same address for usernames

When we put in bob, it shows that all of the usernames are bob

We did a use-after-free



```
void load_users(char* path) {
  FILE* f = fopen(path, "r");
  char buffer[10000];
  int i = 0;
  while(fgets(buffer, 10000, f) != NULL) {
    char* username_part = strtok(buffer, " ");
    char* password_part = strtok(NULL, " ");

    // buffer is reused, so need to make some
    // space for the username for use in the struct
    char* username = malloc(strlen(buffer));
    strcpy(username, username_part);
    printf("%s@%p\n", username, username);

    // Make a User struct with that username,
    // and then copy the (fixed-length) hash part into it
    User current_user = { username, {} };
    strncpy(current_user.passwd_sha, password_part, 32);

    users[i] = current_user;
    i += 1;

    // done with username, so free it now ⚠
    free(username);
  }
}
```

read line by line
interior ptrs into buffer
internal
heap-allocated string for username
copying fixed 32-byte pw_hash
username
16
<user typed in>
not done with the data at 🟥 0x...31b10
users[0] = { 🟥, "abcdef..." }
users[1] = { 🟥, "123456..." }
users[2] = { 🟥, "9876...." }

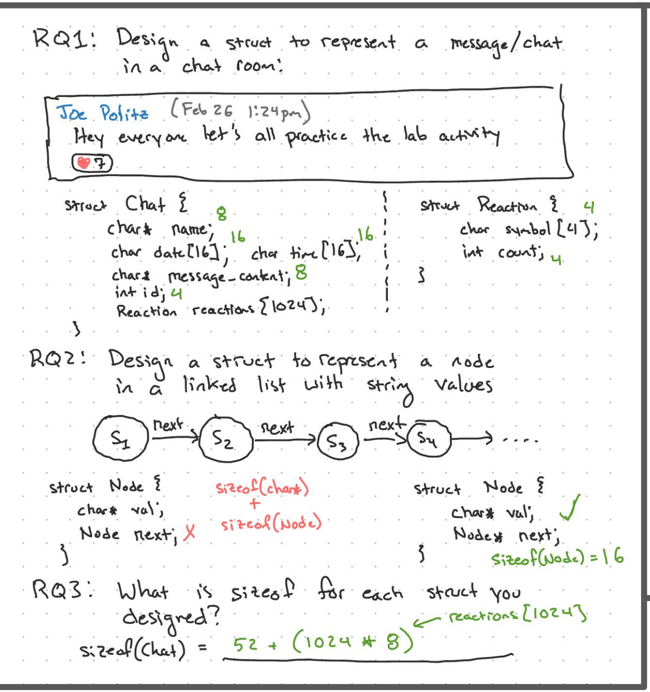
Side note about code quality: We probably shouldn’t have a global variable that has references to addresses on the heap

Takeaway: There’s no concrete place to always put free. You should put free in a place where you guarantee that you won’t use that reference anymore

Questions

- In this example what would be the right time to free?
  - If we wanted to call `load_users` again, right before that call we should free
  - It’s hard to determine the ‘lifetime” of a heap-allocated reference is
- How good is valgrind at catching stuff?
  - In this example it would report many errors
  - It catches when something is free’d then used.
  - It is pretty good
- Why are there errors made with memory leaks in the real world? Don’t these companies have policies against it?
  - Yes, but sometimes people don’t follow policies, and policies are imperfect.
  - There are many issues in the real world but it is still very impressive the things that we have made, such as browsers.

Joe’s Notes (11am)



RQ1: Design a struct to represent a message/chat in a chat room:

Joe Politz (Feb 26 1:24pm)
Hey everyone let's all practice the lab activity
♥ 7

```
struct Chat {
    char* name; 8
    char date[16]; 16  char time[16]; 16
    char* message_content; 8
    int id; 4
    Reaction reactions[1024];
}
```

```
struct Reaction { 4
    char symbol[4];
    int count; 4
}
```

RQ2: Design a struct to represent a node in a linked list with string values

S_1
next
S_2
next
S_3
next
S_4
....

```
struct Node {
  char* val;
  Node next; X
}
```

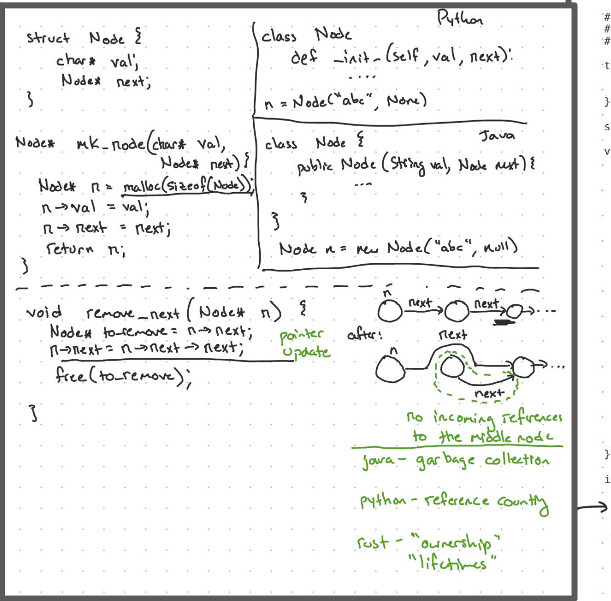
sizeof(char*) + sizeof(Node)

```
struct Node {
    char* val; ✓
    Node* next;
}
```

sizeof(Node) = 16

RQ3: What is sizeof for each struct you designed?

sizeof(Chat) = 52 + (1024 * 8) ← reactions[1024]



```
struct Node {
    char* val;
    Node* next;
}

Node* mk_node(char* val,
              Node* next) {
  Node* n = malloc(sizeof(Node));
  n->val = val;
  n->next = next;
  return n;
}
```

Python

```
class Node
    def __init__(self, val, next):
        ....

n = Node("abc", None)
```

Java

```
class Node {
    public Node(String val, Node next) {
        ...
    }
}

Node n = new Node("abc", null)
```

```
void remove_next(Node* n) {
  Node* to_remove = n->next;
  n->next = n->next->next;
  free(to_remove);

}
```

pointer update

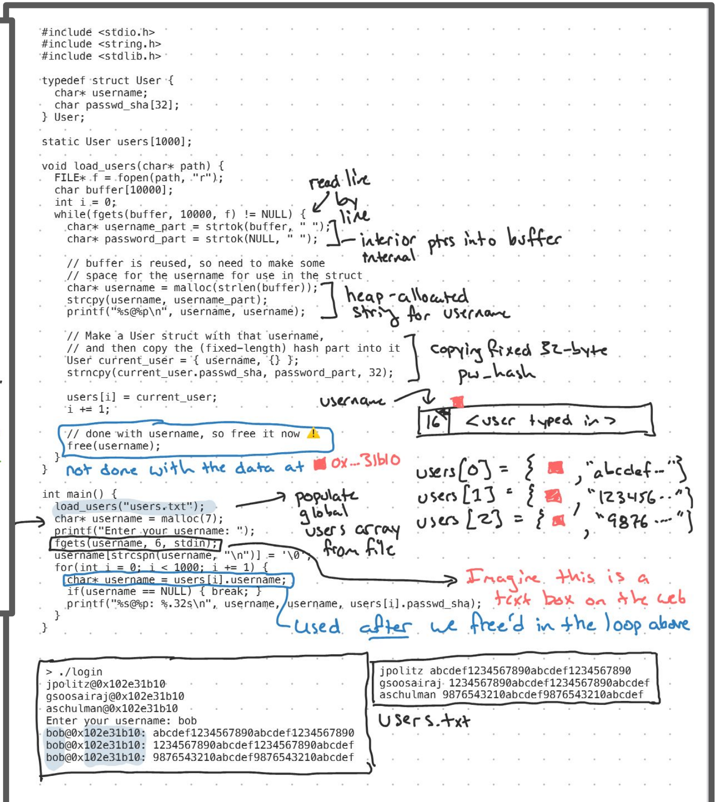
n
next
next
after:
n
next
next

no incoming references to the middle node

java - garbage collection

python - reference counting

rust - "ownership" "lifetimes"



```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct User {
  char* username;
  char passwd_sha[32];
} User;

static User users[1000];

void load_users(char* path) {
  FILE* f = fopen(path, "r");
  char buffer[10000];
  int i = 0;
  while(fgets(buffer, 10000, f) != NULL) {
    char* username_part = strtok(buffer, " ");
    char* password_part = strtok(NULL, " ");

    // buffer is reused, so need to make some
    // space for the username for use in the struct
    char* username = malloc(strlen(buffer));
    strcpy(username, username_part);
    printf("%s@%p\n", username, username);

    // Make a User struct with that username,
    // and then copy the (fixed-length) hash part into it
    User current_user = { username, {} };
    strncpy(current_user.passwd_sha, password_part, 32);

    users[i] = current_user;
    i += 1;

    // done with username, so free it now ⚠
    free(username);
  }
}

int main() {
  load_users("users.txt");
  char* username = malloc(7);
  printf("Enter your username: ");
  fgets(username, 6, stdin);
  username[strcspn(username, "\n")] = '\0';
  for(int i = 0; i < 1000; i += 1) {
    char* username = users[i].username;
    if(username == NULL) { break; }
    printf("%s@%p: %.32s\n", username, username, users[i].passwd_sha);
  }
}
```

read line by line

interior ptrs into buffer
internal

heap-allocated string for username

copying fixed 32-byte pw_hash

username

16 <user typed in>

not done with the data at 🟥 0x...31b10

populate global users array from file

users[0] = { 🟥 , "abcdef..." }
users[1] = { 🟥 , "123456..." }
users[2] = { 🟥 , "9876 ...." }

Imagine this is a text box on the web

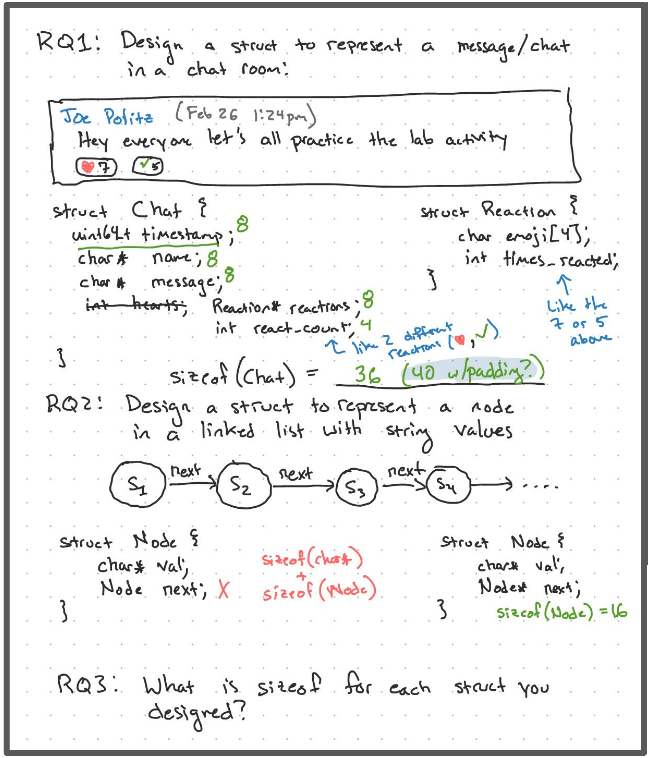
Used after we free'd in the loop above

```
> ./login
jpolitz@0x102e31b10
gsoosairaj@0x102e31b10
aschulman@0x102e31b10
Enter your username: bob
bob@0x102e31b10: abcdef1234567890abcdef1234567890
bob@0x102e31b10: 1234567890abcdef1234567890abcdef
bob@0x102e31b10: 9876543210abcdef9876543210abcdef
```

```
jpolitz abcdef1234567890abcdef1234567890
gsoosairaj 1234567890abcdef1234567890abcdef
aschulman 9876543210abcdef9876543210abcdef
```

users.txt

Joe’s Notes (12:30pm)



RQ1: Design a struct to represent a message/chat in a chat room:

Joe Politz (Feb 26 1:24pm)
Hey everyone let's all practice the lab activity
♥7 ✓5

```
struct Chat {
  uint64_t timestamp; 8
  char* name; 8
  char* message; 8
  ~~int hearts;~~ Reaction* reactions; 8
                  int react_count; 4
}
```

↑ like 2 different reactions (♥, ✓)

sizeof(Chat) = 36 (40 w/padding?)

```
struct Reaction {
  char emoji[4];
  int times_reacted;
}
```

↑ Like the 7 or 5 above

RQ2: Design a struct to represent a node in a linked list with string values

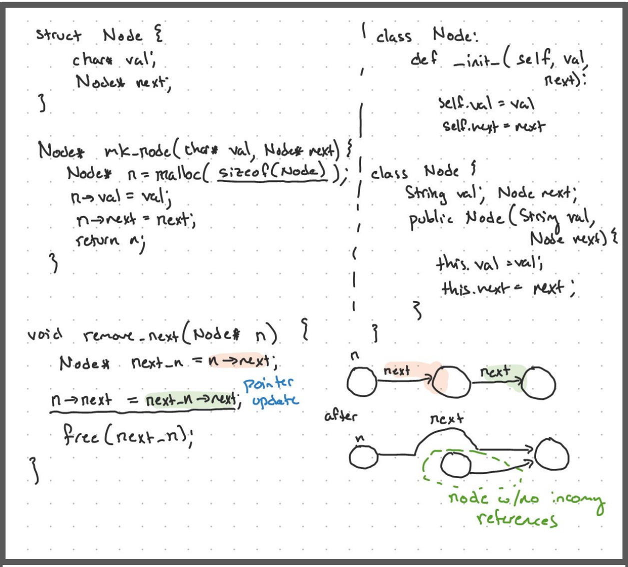
```
struct Node {
  char* val;
  Node next; X
}
```

sizeof(char*) + sizeof(Node)

```
struct Node {
  char* val;
  Node* next;
}
```

sizeof(Node) = 16

RQ3: What is sizeof for each struct you designed?



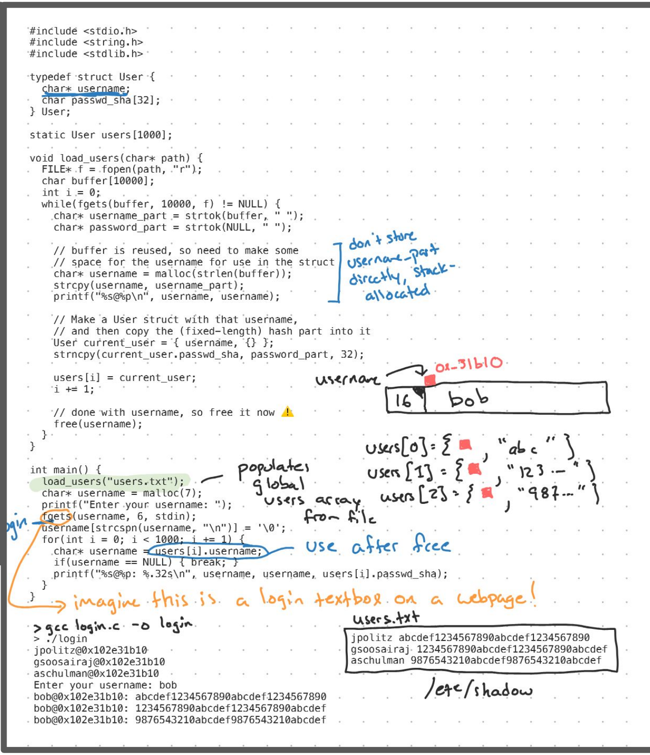
```
struct Node {
  char* val;
  Node* next;
}

Node* mk_node(char* val, Node* next) {
  Node* n = malloc( sizeof(Node) );
  n->val = val;
  n->next = next;
  return n;
}

void remove_next(Node* n) {
  Node* next_n = n->next;
  n->next = next_n->next;
  free(next_n);
}
```

pointer update

```
class Node:
  def _init_(self, val,
                   next):
    self.val = val
    self.next = next

class Node {
  String val; Node next;
  public Node(String val,
              Node next) {
    this.val = val;
    this.next = next;
  }
}
```

n
next
next
after
n
next

node w/no incoming references



```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef struct User {
  char* username;
  char passwd_sha[32];
} User;

static User users[1000];

void load_users(char* path) {
  FILE* f = fopen(path, "r");
  char buffer[10000];
  int i = 0;
  while(fgets(buffer, 10000, f) != NULL) {
    char* username_part = strtok(buffer, " ");
    char* password_part = strtok(NULL, " ");

    // buffer is reused, so need to make some
    // space for the username for use in the struct
    char* username = malloc(strlen(buffer));
    strcpy(username, username_part);
    printf("%s@%p\n", username, username);

    // Make a User struct with that username,
    // and then copy the (fixed-length) hash part into it
    User current_user = { username, {} };
    strncpy(current_user.passwd_sha, password_part, 32);

    users[i] = current_user;
    i += 1;

    // done with username, so free it now ⚠
    free(username);
  }
}

int main() {
  load_users("users.txt");
  char* username = malloc(7);
  printf("Enter your username: ");
  fgets(username, 6, stdin);
  username[strcspn(username, "\n")] = '\0';
  for(int i = 0; i < 1000; i += 1) {
    char* username = users[i].username;
    if(username == NULL) { break; }
    printf("%s@%p: %.32s\n", username, username, users[i].passwd_sha);
  }
}
```

don't store username_part directly, stack-allocated

users[0] = { ■, "abc" }
users[1] = { ■, "123 ..." }
users[2] = { ■, "987..." }

populates global users array from file

login

use after free

imagine this is a login textbox on a webpage!

```
> gcc login.c -o login
> ./login
jpolitz@0x102e31b10
gsoosairaj@0x102e31b10
aschulman@0x102e31b10
Enter your username: bob
bob@0x102e31b10: abcdef1234567890abcdef1234567890
bob@0x102e31b10: 1234567890abcdef1234567890abcdef
bob@0x102e31b10: 9876543210abcdef9876543210abcdef
```

users.txt

```
jpolitz abcdef1234567890abcdef1234567890
gsoosairaj 1234567890abcdef1234567890abcdef
aschulman 9876543210abcdef9876543210abcdef
```

/etc/shadow