PURDUE
UNIVERSITY®

CS 240: Programming in C

Lecture 14: Doubly-Linked Lists
Pointers to Pointers

Prof. Jeff Turkstra

Announcements

- Hope you enjoyed last night
- Exam grading is underway
  - Target completion date is next Tuesday 3/12
- Don’t forget that Homework 7 is out
- Homework 6 due tonight!

Feasting with Faculty

- Tomorrow! 12pm – 1pm
  - Earhart private dining room B
- I’ll be there!
  - I hope!
- If you came last week and can make it, please try again
  - I’m still sorry I missed it!!

Homework 4

```
595 scores total…
100+:  (0)
 100: ===================== (565)
  90: = (1)
  80: = (8)
  70: = (4)
  60:  (0)
  50: = (1)
  40:  (0)
  30:  (0)
  20:  (0)
  10:  (0)
   0: = (16)
Average: 96.79
```

Homework 5

```
595 scores total…
100+:  (0)
 100: ===================== (522)
  90: = (16)
  80: = (9)
  70: = (5)
  60: = (3)
  50: = (4)
  40: = (1)
  30: = (2)
  20: = (1)
  10:  (0)
   0: == (32)
Average: 93.08
```

CS 240 malloc()

- We use our own malloc() library in this class
    - You’ll write your own in CS 252!
- It knows when you malloc() and do not free()
- It knows when you free() more than once
- It knows when you’ve been sleeping
- It knows when you’re awake
- It knows if you’ve been bad or good…

Valgrind

- Valgrind is a suite of tools for debugging and profiling programs
- Very useful for identifying memory leaks and errors

```
$ valgrind ./executable
$ valgrind --leak-check=full ./executable
```

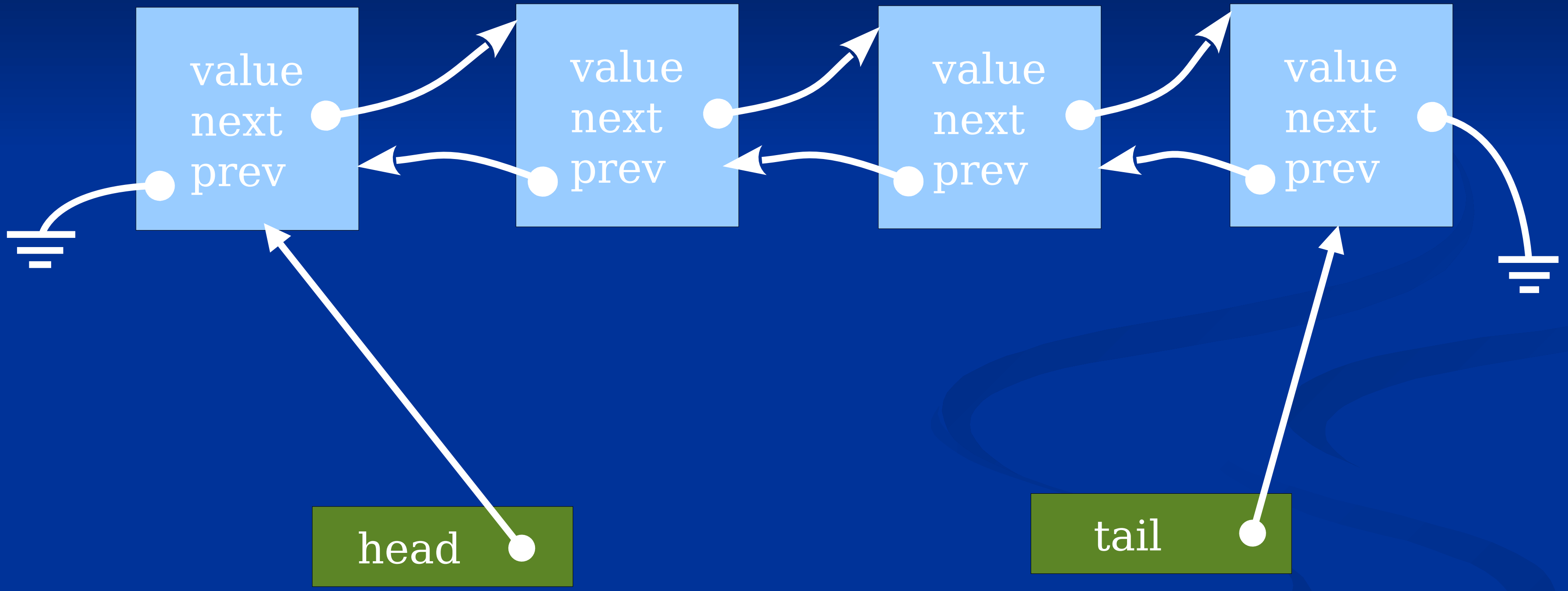
Tough questions

- It’s easy to traverse a list from head to tail
  - How about tail to head?
- Can you write a function that will exchange a specified structure in a linked list with the structure that follows it?
  - Without specifying the head of the list?
- Can you write a function that will prepend a structure before an arbitrary node in the list?
  - Without specifying the head of the list?

Doubly-linked list

- Without the head, the answers to the previous questions are ‘no.’
- The lists we’ve looked at so far are called singly-linked lists
- A doubly-linked list contains two pointers:
  - A “next” pointer
  - A “previous” pointer

Example of a doubly-linked list



Example of declaration

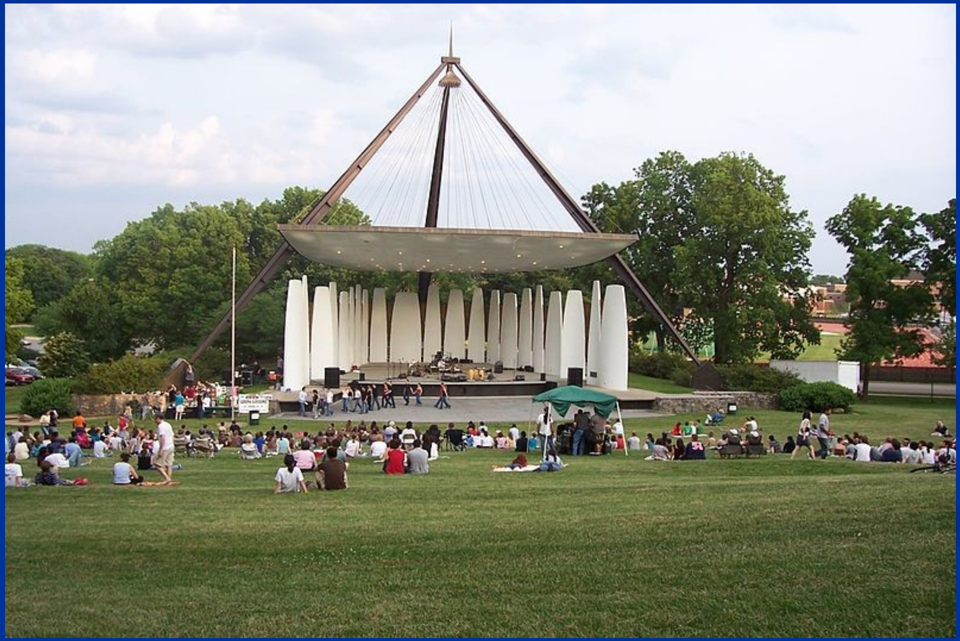
```
#include <stdio.h>
#include <malloc.h>
#include <assert.h>

struct double_l {
  int value;
  struct double_l *next_ptr;
  struct double_l *prev_ptr;
};
```

Creation routine

```
struct double_l *create(int value) {
  struct double_l *temp = NULL;

  temp = malloc(sizeof(struct double_l));
  assert(temp != NULL);

  temp->next_ptr = NULL;
  temp->prev_ptr = NULL;

  temp->value = value;

  return temp;
}
```

Purdue Trivia

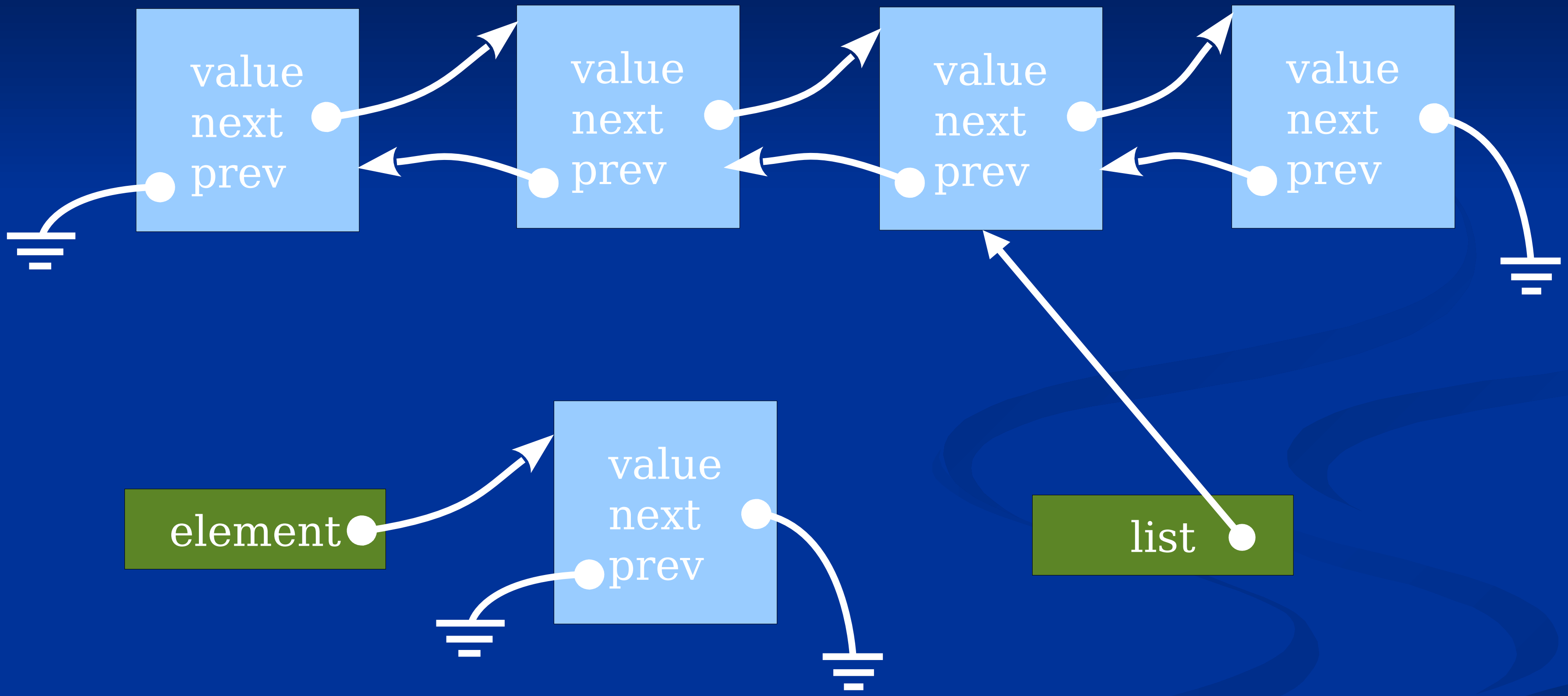
- Slayter Center of the Performing Arts
  - Completed in 1964, dedicated May 1, 1965
  - Gift from Dr. Games Slayter and wife Marie
  - Designed to reflect Stonehenge



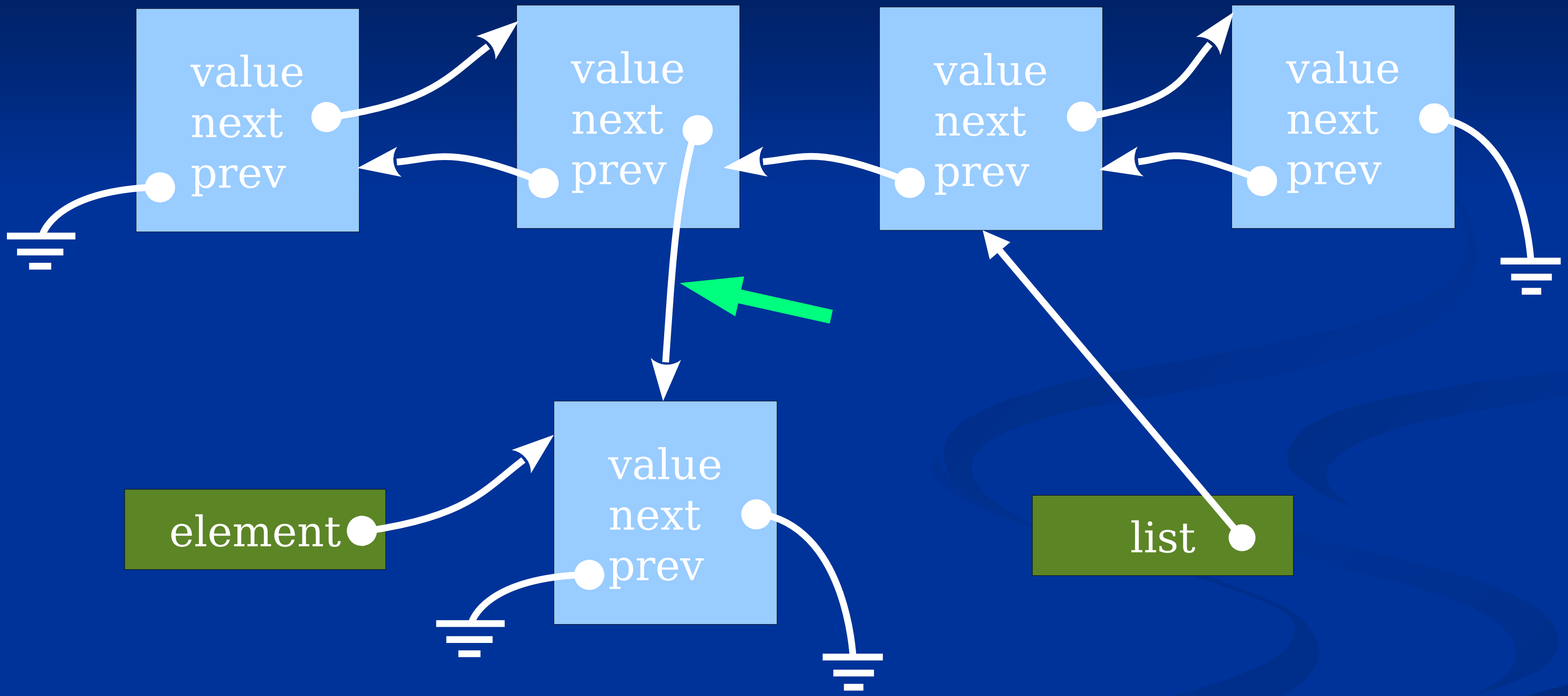
Prepend routine (Look this over carefully!)

```
void prepend(struct double_l *element,
             struct double_l *list) {

  if (list->prev_ptr != NULL)
    list->prev_ptr->next_ptr = element;   1

  element->prev_ptr = list->prev_ptr;   2

  element->next_ptr = list;   3

  list->prev_ptr = element;   4
}
```

Example of ‘prepend’

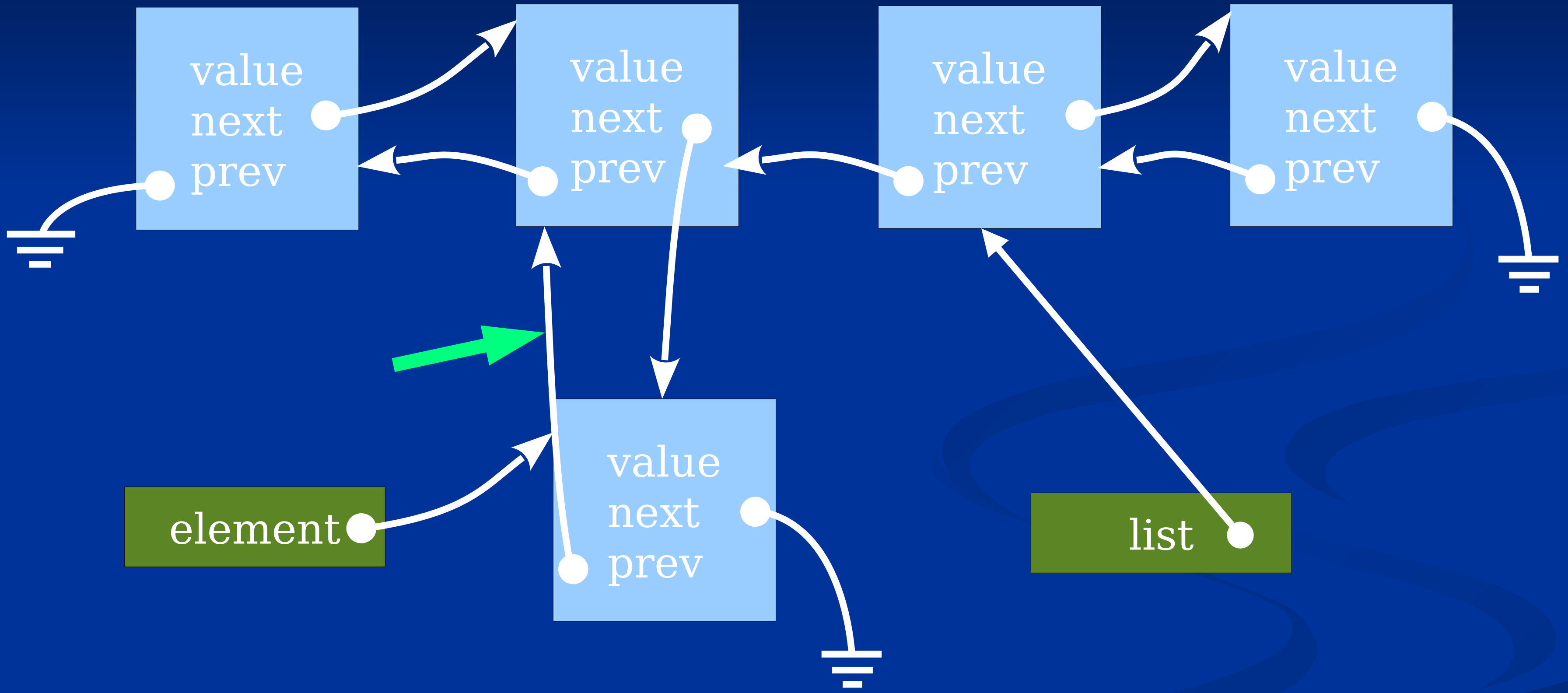


Example of ‘prepend’ Step 1



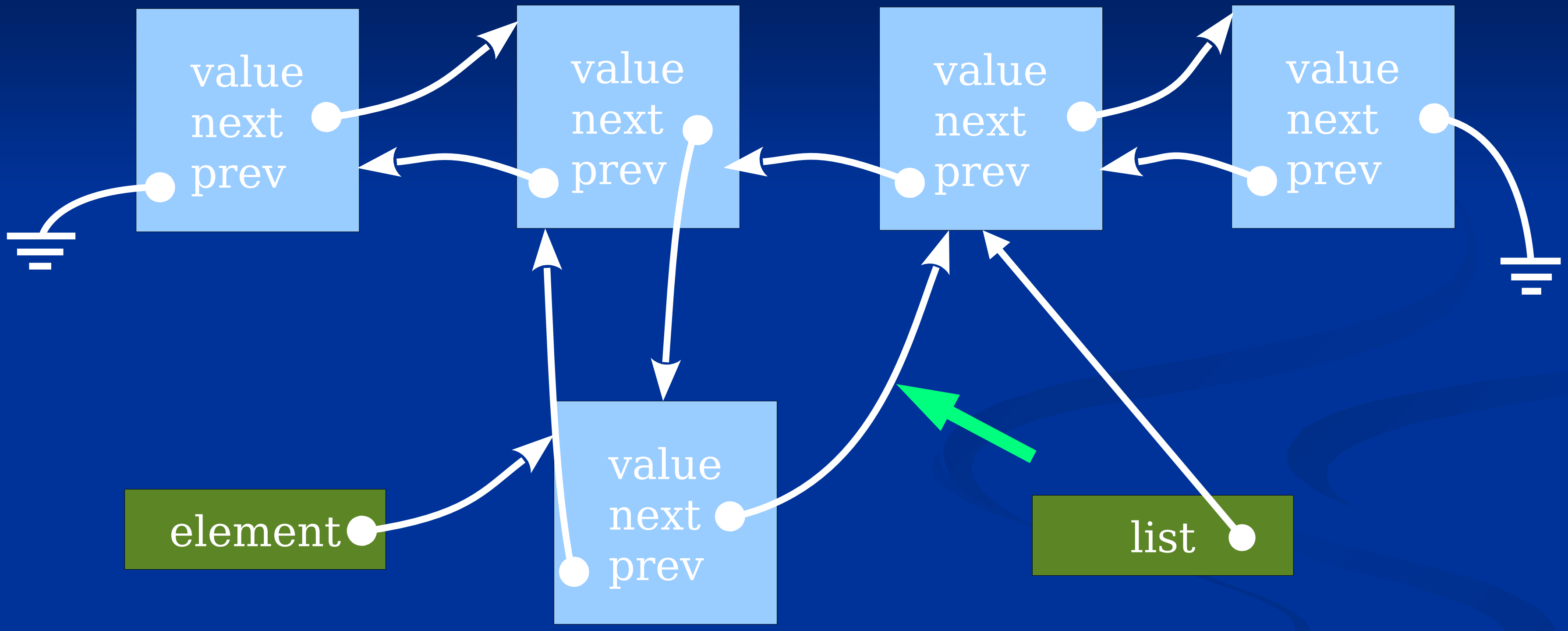
```
if (list->prev_ptr != NULL) {
  list->prev_ptr->next_ptr = element;
}
```

Example of ‘prepend’ Step 2



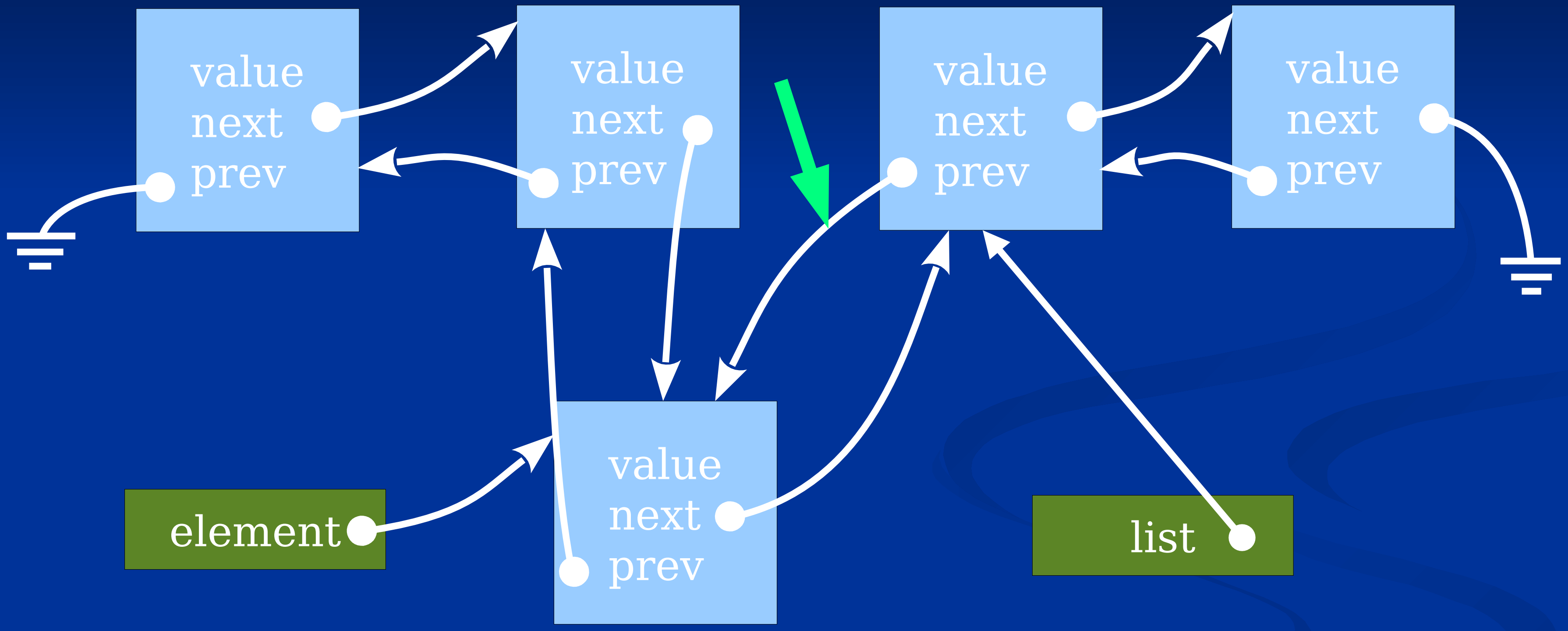
```
element->prev_ptr = list->prev_ptr
```

Example of ‘prepend’ Step 3



```
element->next_ptr = list;
```

Example of ‘prepend’ Step 4



```
list->prev_ptr = element;
```

Important points

- There are four steps.
- When you implement insert, prepend, append, etc you should always have four steps
- It is imperative to put those steps in the right order
  - Some steps are interchangeable; some are not!
- You should practice this on paper

Homework 8

- Practice everything on paper first
- Draw the boxes and reconnect the pointers
- Then write the code

Removing an element from the middle

- With a doubly-linked list, we can remove an element from anywhere within the list

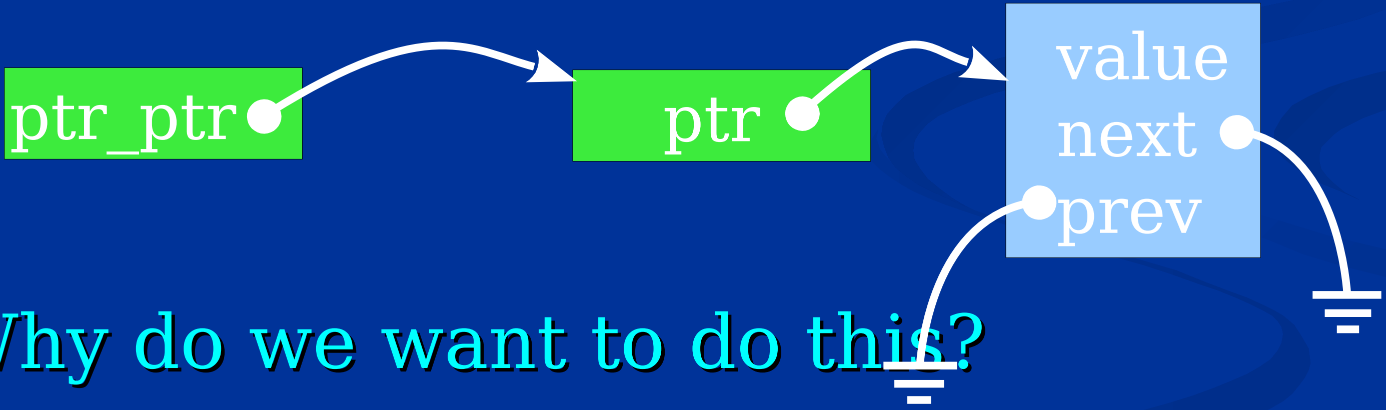
```
void remove_double(struct double_l *ptr) {
  if (ptr->next_ptr != NULL)
    ptr->next_ptr->prev_ptr = ptr->prev_ptr;

  if (ptr->prev_ptr != NULL)
    ptr->prev_ptr->next_ptr = ptr->next_ptr;

  ptr->next_ptr = NULL;
  ptr->prev_ptr = NULL;
}
```

Doubly-linked lists

- Any questions?
- Not covered in your textbook
- Ask TAs (or me) if you have questions
- Might be a good time to take a look at ddd
    - Can graphically display data structures

Pointers to pointers

- In the same way that we can create a pointer that points to an integer or a structure, we can also create a pointer that points to another pointer…



- Why do we want to do this?

Why use pointers to pointers?

- In some cases, we haven’t been able to get a single function to do everything we want. E.g.:
- We’d like to have a function free() a memory location and set the pointer to NULL.

```
        free(ptr);
        ptr = NULL;
```

- How can we create a function to (conveniently) do both of these operations?
- We need something that can modify the pointer in addition to what is pointed to…

Passing a pointer to a pointer

- Consider a function called my_free()…

```
void my_free(struct double_l **ptr_ptr) {
  struct double_l *ptr = NULL;
  assert(ptr_ptr != NULL);

  ptr = *ptr_ptr;
  free(ptr);
  *ptr_ptr = NULL;
}
```

- Call it like: `my_free(&ptr);`

Other uses

- The main() function is passed a pointer to pointers to char:

```
int main(int argc, char **argv) {
  char *temp = NULL;
  if (argc > 1) {
    temp = argv[1];
    printf(“Argument 1 is: %s\n”, temp);
  }
}
```

- Now you know what that argv thing is…

Rules for using pointers to pointers

- The issue of pointer type becomes just a little more important
  - You cannot assign pointers to each other that are not the right type
- Now you have more types to choose from
- You need to be sure what you are pointing to is something real (and that it’s still there)
  - More NULL conditions to check for…

Pointer problems

```
int main(int argc, char **argv) {
  int i = 0;
  int *pi = NULL;
  int **ppi = NULL;

  pi = &i;
  ppi = &pi;
  i = 5;

  printf(“i is %d\n”, **ppi);
  pi = NULL;
  printf(“i is %d\n”, **ppi);
  return *pi;
}
```

Rules of thumb…

- Don’t use more levels of indirection than you need
- Use multilevel pointers only when not doing so would be very inefficient or error prone
- You can triple-level pointers
  - …but if you do, you’re probably doing something wrong

For next lecture

- Work on Homework 8!!!
- Study the examples in this lecture at home
- Practice the examples
- Modify the examples

Boiler Up!