CS 240: Programming in C

Lecture 15: Pointers to Pointers,
Internal Pointers,
Pointers to Functions

PURDUE UNIVERSITY

Announcements

- Midterm 1 grades are posted to the gradebook
  - Your estimated letter grade may have changed
  - Remember that cutoffs may change at the end of the semester

Pointers to pointers

- A pointer “points” to a variable
- But a pointer is itself a variable
- So, we can create a pointer that points to another pointer
  - Sometimes called a “double pointer”

Pointers to pointers

- A pointer “points” to a variable
- But a pointer is itself a variable
- So, we can create a pointer that points to another pointer
  - Sometimes called a “double pointer”

- Why would 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
- For example:

```
void my_free(struct dbl_node *ptr) {
  free(ptr);
  ptr = NULL;
}
```

- Why doesn't this work?

Why use pointers to pointers?

- In some cases, we haven't been able to get a single function to do everything we want
- For example:

```
void my_free(struct dbl_node *ptr) {
  free(ptr);
  ptr = NULL;
}
```

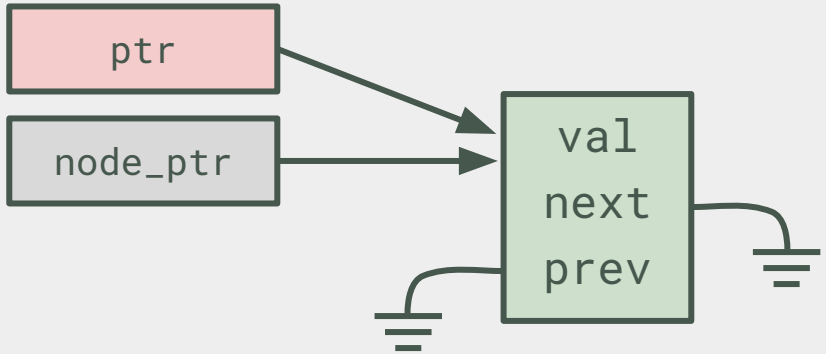
`ptr` is a *local variable*. Setting to NULL doesn't change it outside of this function

- Why doesn't this work?
- We need to be able to modify the pointer itself

Passing a pointer to a pointer

```
void my_free(struct dbl_node *ptr) {
  free(ptr);
  ptr = NULL;
}
```

```
my_free(node_ptr);
```



Passing a pointer to a pointer

- We can instead pass a pointer to the pointer

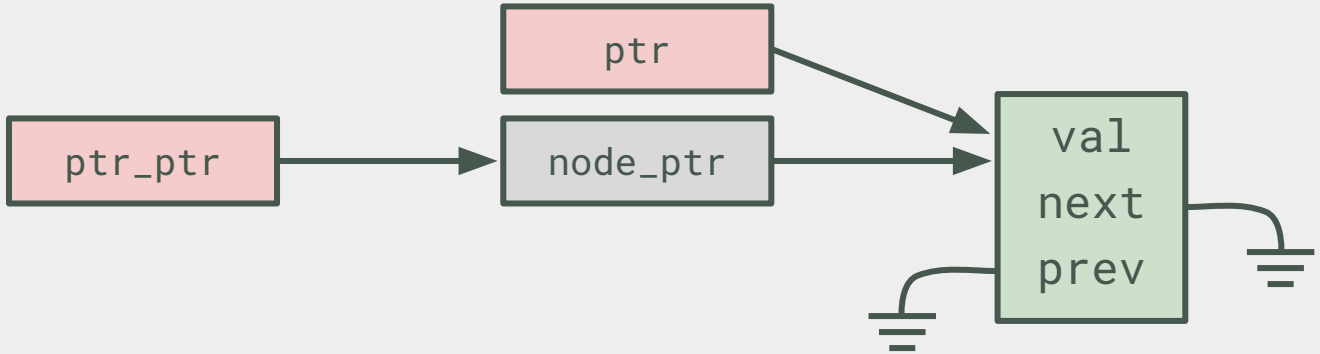
```
void my_free(struct dbl_node **ptr_ptr) {
  struct dbl_node *ptr = *ptr_ptr;
  free(ptr);
  *ptr_ptr = NULL;
}
```

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

Passing a pointer to a pointer

```
void my_free(struct dbl_node **ptr_ptr) {
  struct dbl_node *ptr = *ptr_ptr;
  free(ptr);
  *ptr_ptr = NULL;
}
```

```
my_free(&node_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);
  }
}
```

```
$ ./program a b c
Argument 1 is: a
```

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 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 use triple-level pointers
  - ...but if you do, you’re probably doing something wrong

List operations using double pointers

- Let’s look at another situation where using pointers to pointers makes sense:

```
void prepend_to_head(struct dbl_node **head_ptr,
                     struct dbl_node *new_node);
```

- Call it like: `prepend_to_head(&head, node_ptr);`

prepend_to_head()

```
void prepend_to_head(struct dbl_node **head_ptr,
                     struct dbl_node *new_node) {
  assert(head_ptr != NULL);

  if (*head_ptr == NULL) {
    *head_ptr = new_node;
  }
  else {
    new_node->next = *head_ptr;
    (*head_ptr)->prev = new_node;
    *head_ptr = new_node;
  }
}
```

Internal pointers

- We can have pointers inside list nodes that point to other structures
  - e.g., next, prev
- We can also have pointers pointing to arbitrary things

```
struct info {
  char *name;
  char *address;
  struct info *next;
};
```

Using internal pointers incorrectly

```
struct info *create_info(char *name, char *addr) {

  struct info *ptr = NULL;
  ptr = malloc(sizeof(struct info));
  assert(ptr != NULL);

  ptr->name = name;
  ptr->address = address;

  ptr->next = NULL;
  return ptr;
}
```

Using internal pointers incorrectly

- Why was the previous example incorrect?
- Consider the following code segment

```
printf("Enter name: ");
scanf("%s", name);
printf("Enter address: ");
scanf("%s", address);
head_ptr = create_info(name, address);

name[0] = '\0';
address[0] = '\0';

printf("Node:    name: %s\n", head_ptr->name);
printf("      address: %s\n", head_ptr->address);
```

Using internal pointers

- Make sure you allocate everything

```
struct info *create_info(char *name, char *addr) {
  struct info *ptr = NULL;
  ptr = malloc(sizeof(struct info));
  assert(ptr != NULL);

  ptr->name = malloc(strlen(name) + 1);
  assert(ptr->name != NULL);
  strcpy(ptr->name, name);

  ptr->address = malloc(strlen(address) + 1);
  assert(ptr->address != NULL);
  strcpy(ptr->address, address);

  ptr->next = NULL;
  return ptr;
}
```

Using internal pointers

- Also make sure you deallocate everything

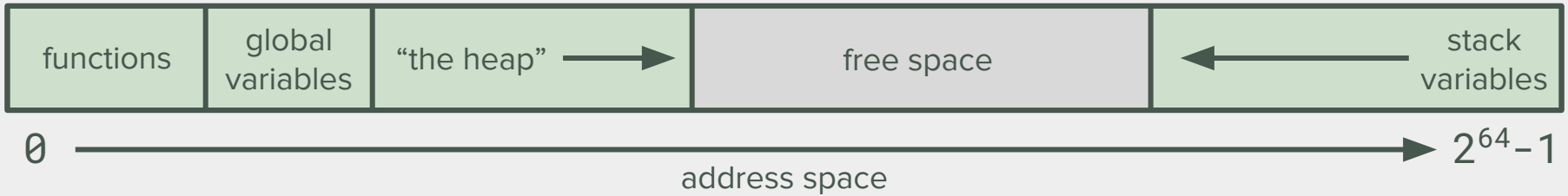
```
void delete_info(struct info **info_ptr_ptr) {
  assert(info_ptr_ptr != NULL);
  assert(*info_ptr_ptr != NULL);

  if ((*info_ptr_ptr)->name != NULL) {
    free((*info_ptr_ptr)->name);
    (*info_ptr_ptr)->name = NULL;
  }
  if ((*info_ptr_ptr)->address != NULL) {
    free((*info_ptr_ptr)->address);
    (*info_ptr_ptr)->address = NULL;
  }

  (*info_ptr_ptr)->next = NULL;
  free(*info_ptr_ptr);
  *info_ptr_ptr = NULL;
}
```

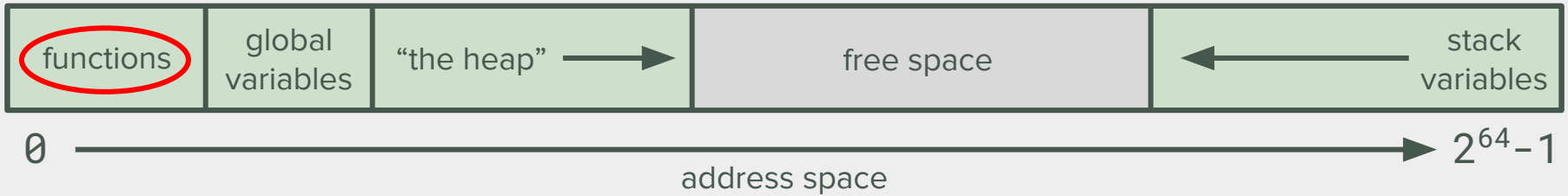
Function pointers

- Recall the memory layout map...



Function pointers

- Recall the memory layout map...



- Functions reside in memory. Therefore we can refer to their addresses
- We can call functions using their address!

Declaring a function pointer

- The difficult part of using function pointers is figuring out how to declare a pointer to a function
- Here is a pointer to a function that accepts two integers and returns an integer:

```
int (*ptr_to_func)(int x, int y);
```

- We could also initialize this pointer to NULL:

```
int (*ptr_to_func)(int x, int y) = NULL;
```

- We don’t need argument names:

```
int (*ptr_to_func)(int, int) = NULL;
```

Using a function pointer

```
int sum(int addend, int augend) {
  return addend + augend;
}

int main() {
  int result = 0;
  int (*ptr_to_func)(int, int) = NULL;

  ptr_to_func = sum;
  result = (*ptr_to_func)(3, 5);
  printf("result = %d\n", result);
  return 0;
}
```

Or like this...

```
int sum(int addend, int augend) {
  return addend + augend;
}

int main() {
  int result = 0;
  int (*ptr_to_func)(int, int) = NULL;

  ptr_to_func = sum;
  result = ptr_to_func(3, 5);
  printf("result = %d\n", result);
  return 0;
}
```

Passing a pointer to function

```
int do_operation(int (*pf)(int, int),
                 int value1,
                 int value2) {
  return pf(value1, value2);
}

int main() {
  int (*ptr_to_func)(int, int) = NULL;
  ptr_to_func = sum;
  printf("%d\n", do_operation(ptr_to_func, 3, 5));
  return 0;
}
```

What's this good for?

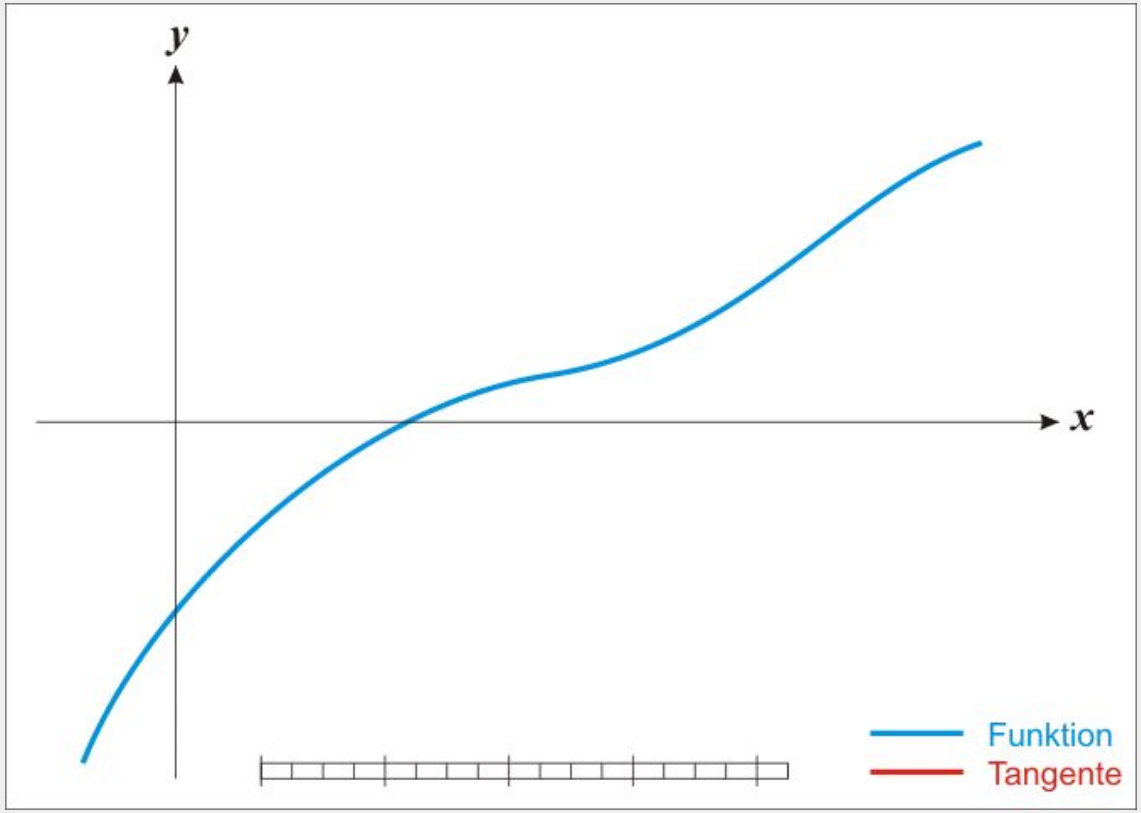
- Suppose we have a subroutine that uses Newton's Method to locate a root of a polynomial function:

```
float newton(float (*ptr_fn)(float x), float start);
```

- We might want to call the subroutine for different mathematical functions…

```
root1 = newton(func1, 5.3);
root2 = newton(func2, 2.9);
```

Newton's Method



Source: Wikipedia

Newton's Method

```
float newton(float (*function)(float), float start) {
  float x1, x2, y1, y2, tmp;
  x1 = start;
  x2 = x1 + 1.0;
  do {
    y1 = function(x1);
    y2 = function(x2);
    tmp = x1 - y1 / ((y1 - y2) / (x1 - x2));
    x2 = x1;
    x1 = tmp;
  } while (fabs(y1 - y2) > 0.001);
  return x1;
}
```

Example: find sqrt(23)

```
/* The positive root of this function
 * is the square root of 23.
 */
float func(float x) {
  return pow(x, 2) - 23.0;
}

int main() {
  float root = 0;
  root = newton(func, 1);
  printf("root of x^2 - 23 = %f\n", root);
  return 0;
}
```

Searching a list

- Suppose you have a list with many fields per node:

```
struct node {
  char *name;
  char *title;
  char *company;
  char *location;
  struct node *next;
};
```

- What if we wanted to be able to search the list by any one of them?

List search

```
struct node *list_search(
  int (*compare)(struct node *, char *),
  struct node *head_ptr,
  char *item) {

  while (head_ptr != NULL) {
    if (compare(head_ptr, item) == 0) {
      return head_ptr;
    }
    head_ptr = head_ptr->next;
  }
  return NULL;
}
```

Example comparison functions

```
int compare_name(struct node *ptr, char *item) {
  return strcmp(ptr->name, item);
}
```

```
int compare_title(struct node *ptr, char *item) {
  return strcmp(ptr->title, item);
}
```

```
ptr = list_search(compare_name, head_ptr, “Chris”);
```

For next lecture

- Work on Homework 7!!
- Study the examples in this lecture at home
- Practice the examples
- Modify the examples

Slides

- Slides are heavily based on Prof. Turkstra's material from previous semesters.