U N I V E R S I T Y ®

CS 240: Programming in C

Lecture 15: Pointers to Pointers
The Many Faces of Zero
Pointers to Functions

Prof. Jeff Turkstra

Feasting with Faculty

- Tuesday this week!
  - No room reserved, will likely be in the main dining area somewhere (Earhart)
  - Still 12pm – 1pm
- No lunch Thursday this week!

Takehome Quiz 7

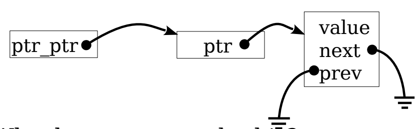
- **You must swap the nodes themselves (update the pointers). You will receive no credit for simply exchanging the values.**
- 1: Use typedef to define a type named “node.” “node” should be a structure with two fields: an integer (named “val”) and a pointer to a node structure (named “next”)
- 2: Write a function with the following prototype:
  `node *swap_with_next(node *, int);`
  - This function accepts a pointer to the head of a singly-linked list and the value of a node that, when found, should swap positions with its next node.
  - Traverse the list and find the node whose value equals the second argument. If it does not exist, return NULL. Once found, swap its position with its next node. Return a pointer to the head of the list on success.
  - Include any necessary assert()ion checks. #includes are not needed

Announcements

- No homework due over spring break
  - We suggest completing it before spring break, then you don’t have to worry about it
  - Homework 8 is due Wednesday, 3/26 9pm

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

List operations using pointers to pointers

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

```
void prepend_to_head(
               struct item **head_ptr,
               struct item *new_ptr);
```

- Call it like this:

```
prepend_to_head(&head, item_ptr);
```

prepend_to_head()

```
void prepend_to_head(struct item **head_ptr,
                struct item *new_item_ptr) {
  assert(head_ptr != NULL);

  new_item_ptr->prev_ptr = NULL;
  if (*head_ptr == NULL) {
    *head_ptr = new_item_ptr;
  }
  else {
    new_item_ptr->next_ptr = *head_ptr;
    (*head_ptr)->prev_ptr = new_item_ptr;
    *head_ptr = new_item_ptr;
  }
}
```


List elements containing pointers to other things

- We can have pointers inside list elements that point to other structures
  - E.g., next_ptr, prev_ptr
- We can have pointers inside list elements that point to other arbitrary things…

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


Using internal pointers incorrectly…

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

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

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


Why was the previous function 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 *address) {
  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->name != NULL);
  strcpy(ptr->address, address);

  ptr->next_ptr = 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_ptr = NULL;
  *info_ptr_ptr = NULL;
}
```

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_ptr = NULL;
  free(*info_ptr_ptr);
  *info_ptr_ptr = NULL;
}
```


Pointers to pointers – again

- Desire to change the ‘value’ of fp:

```
void my_close(FILE *file_ptr) {
  fclose(file_ptr);
  file_ptr = NULL;
}

int main() {
  FILE *fp = NULL;
  fp = fopen(“input”, “r”);
  my_close(fp);
  return (0);
}
```

file_ptr

fp

File

It didn’t work.

- Why wasn’t fp set to NULL?
  - The <u>value of fp was passed</u> to my_close()
  - The new value assigned inside my_close() was not stored back into fp
- In fact, there is no way for my_close() to modify the value of fp!
- What remained constant in both main() and my_close() with respect to the file pointer?
  - The address (memory location) of the file pointer – pointer to pointer

Pointers to pointers - correct

- my_close() now modifies fp:

```
void my_close(FILE **file_ptr_ptr) {
  fclose(*file_ptr_ptr);
  *file_ptr_ptr = NULL;
}

int main() {
  FILE *fp = NULL;
  fp = fopen(“input”, “r”);
  my_close(&fp);
  return (0);
}
```

*file_ptr_ptr

&fp

fp

File

Purdue Trivia

- The Stone Lions Fountain was dedicated in 1904 as a gift from the class of 1903
  - “Ran dry” (was turned off) somewhere between 1923-1931. Nobody knows why.
- Re-dedicated 4/22/2001

The many faces of ‘zero’

- Lots of people wonder what the difference between 0, 0.0, NULL, ‘\0’, NUL, 0x0, etc are and when/where to use them
  - 0 is an integer. sizeof(0) == 4
  - 0x0 is the same as 0 expressed in hexadecimal notation
  - ‘\0’ is a character. Its character code is NUL. Characters are one-byte integers. It is interchangeable with 0. sizeof(‘\0’) == 4
  - 0.0 is a floating point value.
    - Interchangeable with 0
  - NULL is literally: ((void *) 0)
    - Must assign to or compare against a pointer

The mark of zero

- Whenever you want to use a NULL, 0, or ‘\0’, you can just say: 0

```
void subroutine(char *char_ptr) {
  float value = 0;            /* 0.0 */
  char *new_ptr = 0;          /* NULL */
  if (char_ptr == 0) return;  /* NULL */
  if (*char_ptr == 0) return; /* ‘\0’ */
  new_ptr = (&char_ptr[1]);
  char_ptr[2] = 0;            /* ‘\0’ */
}
```

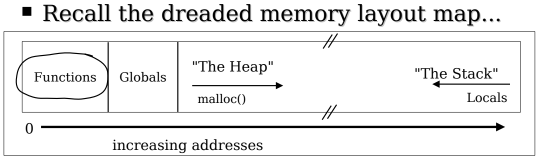
- But use the right symbol in the right place so that you’ll understand your code later

What to do about zero…

- If you can’t get your code to compile because of one of those ‘\0’, NULL, 0.0 problems, just use 0
- Then clean your code up for clarity…
  - Figure out whether it’s a pointer or not, then use NULL or 0 respectively
  - If it’s a character type, use ‘\0’
  - If it’s a float type, use 0.0
- Where does NULL come from?

Function pointers

- Recall the dreaded memory layout map…



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

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;
}
```

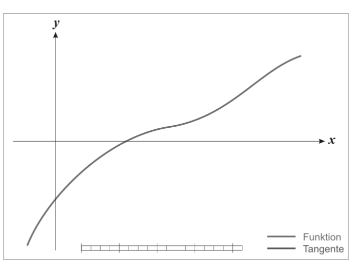
Passing a pointer to function

```
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

```
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 the square root of 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;
}
```


Pointers to functions and linked-lists

- Linked list manipulation routines we've looked at so far have assumed that one of the elements of the node was the key of the search/sort
- What if we had multiple items in the node structure and we wanted to be able to search by any one of them?
```
struct node {
  char *name;
  char *title;
  char *company;
  char *location;
  struct node *next;
};
```

Fields that we might search by...

New 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;
} /* list_search */
```


Example comparison function

```
/* Definition of comparison:
 * zero:     equal
 * negative: structure value ‘less than’ item
 * positive: structure value ‘greater than’ item
 */
int compare_name(struct node *ptr, char *item) {
  return strcmp(ptr->name, item);
}

/*
 * Example of calling list_search…
 */
ptr = list_search(compare_name, head_ptr, “Jeff”);
```


Strings

- When you use a string literal in C, that value is stored in “read-only” memory (the text/code segment)
  - It cannot be changed
- What’s the difference between this:
  `char *str = “Hello!”;`
  …and this:
  `char str[] = “Hello”;`

Strings

- `char *str = “Hello!”;`
  - Allocates a pointer on the stack
  - Points to an array in the read-only “code/text segment”
- `char str[] = “Hello”;`
  - Allocates an array on the stack and initializes it by copying values from the array in the read-only “code/text segment”

For next lecture

- Study the examples in this lecture at home
- Practice the examples
- Modify the examples

Boiler Up!