CS1003: Intro to CS, Summer 2008

Lab #06

Instructor: Arezu Moghadam
arezu@cs.columbia.edu

Recap

- Debugging
- Pointers
Call by reference

- Pointers can be used as function arguments
- We have been typically using call by value
- Remember the swap function

```c
#include <stdio.h>

int swap (int a, int b); 
int main () {
    int x=3, y=7;
    printf("%d %d\n", x, y);
    swap (x,y);
    printf("%d %d\n", x, y);
    return 0;
}

int swap (int a, int b) {
    int tmp;
    tmp=a;
    a=b;
    b=tmp;
    return a;       // I can return only one value, what do I return?
} //ptrexample2.c
```

Call by reference II

- Note that the call-by-value has problems in that only the method’s local values are affected.
- Therefore we need something else
  - Pointers to the rescue
  - We call other functions and pass parameters by reference
  - New code looks like
Call by reference III

```c
#include <stdio.h>

int swap (int *, int *);

int main() {
    int x=3, y=7;
    printf("%d %d\n", x, y);
    swap (&x,&y);
    printf("%d %d\n", x, y);
    return 0;
}

int swap (int *p, int *q) {
    int tmp;
    tmp = *p;
    *p = *q;
    *q = tmp;
}

//ptrexample3.c
```

Storing an indeterminate amount of data

- How would you store an indeterminate amount of data?
- You create a bank, but you don’t know how many accounts you are going to have
- Two ways to fix this
  - Growable arrays
    - If the array fills up, create an array twice its size and copy all the elements over
  - Linked Lists
Pointers and linked lists

- Instead of statically declaring an array, we can create a bunch of nodes and link them together

```c
struct node {
    struct node *next_ptr;
    int value;
}
```

- If you wanted to create a large number of these nodes

```c
struct node node_1;
struct node node_2;
```

Pointers and linked lists II

- However, you can only declare a limited number of nodes.
  - Well, ok, so you can create a lot, but if you didn’t know how many you would need, then you have a problem.
- Therefore you can allocate memory dynamically
function malloc()

- malloc();
  - **usage**: void *malloc (unsigned int);
  - It allocates storage for a variable and returns a pointer.
  - It is used to create things out of thin air 😊
  - Up to now, we use pointers to point to predefined variables
  - With malloc we can allocate memory without having to predefine a variable
  - The void * mean that malloc returns a generic pointer

malloc examples

```c
#include <stdlib.h>
main() {
    char *string_ptr;
    string_ptr = malloc (80);
}
```

- This allocates storage for a character string 80 bytes long ('\0' included)
malloc examples

More precisely

```
#include <stdlib.h>
main() {
    char *string_ptr;
    string_ptr = malloc (80 * sizeof(char));
}
```

malloc examples II

- You may be allocating lots of variables of type struct, each of which has large arrays. Therefore you are allocating real space in memory for each instance.

```
#include <stdlib.h>
const int MAX_ENTRIES = 10;
struct mailing {
    char name[60];
    char address1[60];
    char address2[60];
    char city[40];
    char state[2];
    long int zip;
};
main() {
    struct mailing *mailing_list;
    mailing_list = malloc(MAX_ENTRIES * sizeof(struct mailing));
}
```
free()

- It is the opposite of malloc
- malloc allocates memory
- You can de-allocate it using free
- free takes a pointer as an argument, just as malloc returns a pointer
- **Usage**: free(pointer);
  - Here pointer is what was returned by malloc
- Not freeing / Double freeing is bad

free() example

```c
#include <stdlib.h>
main() {
    char *string_ptr;
    string_ptr = malloc(80);

    free(string_ptr);
    string_ptr = NULL;
}
```
- You typically NULL out the pointer as well
- If you don’t use free, you will keep eating the allocated memory every time you call the respective function
Heaps and Stacks

- How does all of this happen in memory?
- There are two ways that this is all stored in memory
  - Heaps
  - Stacks
- Stacks used for regular variables that you have seen so far
- Heaps used for malloc();

Heaps and Stacks II

- When you call a function, space for all the local function variables, etc. are created in memory, in a stack frame
  - When you leave the function, all that memory is cleaned up
- However, when you allocate space using malloc, it is allocated in a heap
  - It is not cleaned up when leaving a function
  - Therefore you have to use free
Dangling pointers

A dangling pointer is a surviving reference to an object that no longer exists at that address. Dangling pointers typically arise from one of:

- A premature free, where an object is freed, but a reference is retained;
- Retaining a reference to a stack-allocated object, after the relevant stack frame has been popped.

Bad code (preliminary free)

```c
int main(void) {
    int *result = malloc(sizeof(int));
    *result = 6;
    free(result);
    printf(“result is %d\n”, *result);
}
```
Bad code (stack memory)

```c
int main(void) {
    int *result = square(6);
    printf("result is %d\n", *result);
}

int *square(int i) {
    int j = i * i;
    return &j;
}
```

Back to linked lists

- So how does malloc help us here?
  ```c
  struct linked_list {
      char data[30];
      struct linked_list *next_ptr;
  }
  struct linked_list *first_ptr = NULL;
  ```
- So we want to use malloc instead of creating an array of linked lists that will limit the number of nodes in the linked list to the size of the array
- How can we do this?
Pointers and Linked Lists contd...

new_node_ptr = malloc(sizeof(struct linked_list));
☐ This created the new node and allocates the correct amount of memory
(*new_node_ptr).data = item;
☐ This will store the value of item into data
(*new_node_ptr).next_ptr = first_ptr;
☐ The node now points to first_ptr
first_ptr = new_node_ptr;
☐ The new element is now the first element

One other concept like malloc()

calloc()
  ■ Usage: void *calloc (int n, int size_of_n);
  ■ similar to malloc(), except that you give it that second argument of the number of elements followed by the size of each of those elements
  ■ Slightly cleaner than malloc(sizeof(foo) * nElements)
File I/O example – read file content

```c
#include <stdio.h>

int main() {
    char c; /* declare a char variable */
    FILE *file; /* declare a FILE pointer */
    file = fopen("numbers.txt", "r"); /* open a text file for reading */
    if(file==NULL) {
        printf("Error: can't open file.
        return 1;
    }
    else
    {
        printf("File opened successfully. Contents:

        while(1) {
            c = fgetc(file);
            if(c!=EOF) {
                printf("%c", c);
            }
            else {
                break;
            }
        }
        printf("Now closing file...
        fclose(file);
        return 0;
    }
```

File I/O example – copy a file

```c
/* Copy a file */
#include <stdio.h>

void main() {
    int ch; /* input character */
    char inputfile[80], outputfile[80];
    FILE *fpin, *fpout; /* input & output pointers */
    /* get the filenames from the user */
    printf("What file do you want to copy? ");
    scanf("%s", inputfile);
    printf("What file name do you want for the copy? ");
    scanf("%s", outputfile);
    /* open the files */
    fprintf(fpout, "%s", outputfile);
    /* copy the file char by char */
    while( (ch = fgetc(fpin)) != EOF) {  
        fputc(ch, fpout);
    }
    /* close the files */
    fclose(fpin);
    fclose(fpout);
}
Assignment

☐ Read chapter 17 from the practical C programming

☐ HW3