Dynamic Memory Allocation: Arrays & Linked Lists

ELEC1201 Lecture 9

Enrico Costanza and Jeff Reeve

25 October 2013

Last edit: 20 October 2014
Labs & Quiz

• Lab 4 seems a bit of a struggle
  – Do not get discouraged, there is a lot to learn
  – As we move forward you need to rely on concepts from previous labs

• The mock quiz this afternoon will give you an idea of what you will face next week
Labs & Quiz (cont.)

• I expect many of you will find the quiz today difficult

• Programming is learned by practice!
  
  – Friendly advice: make sure you are well comfortable with all labs material
  
  – If you want more exercises you can find some here: https://secure.ecs.soton.ac.uk/notes/comp1010/labs 2-5 + lab 10
  
  – Don't be afraid to go back to early book chapters and early labs
We discussed pointers to string literals in the clinic last week, if you were not there, please ask someone who was to explain you, you will both learn from that.

In general my advice is: stay away from pointers to string literals! Use simple strings.

- The book uses them a lot, unfortunately

Use the lecture slides and the examples linked there.

- Some of you seem not to look at those
- (not just about strings and string literals)
Textbook

- Part of the material covered today is at the end of chapter 9 of the textbook

- For linked lists, please refer to Wikipedia: http://en.wikipedia.org/wiki/Linked_list and to the C5 lab script and related code

- Please go through this material and through chapter 10 of the textbook before the lab
Credits + feedback

• The quizzes in these slides are from http://www.cprogramming.com/

• The rest of the material is original, so feedback is welcome!
Memory Allocation and De-allocation

- Variables and arrays are stored in memory
- Through variables declaration memory is "reserved" for our program when it starts and it is "unreserved" when it quits, so that other programs can use it
- Variable declarations are fixed at compile time
- Sometimes it is useful or necessary to "reserve" memory at run-time, we refer to this as "allocation"
Dynamic Arrays

- So far we have always declared arrays of constant size, e.g.

  ```
  int myArray[10];
  or
  #define AL 100
  ...
  int arr[AL];
  ```

- Sometimes it can be useful or necessary to define the size of the array at run-time, i.e. through a variable
Allocating memory: `malloc`

- The function `malloc` can be used to allocate a memory block of a given size
- Easier to go by example: `allocDemo.c`
Allocating memory: malloc (cont.)

- The prototype for malloc:
  ```
  void *malloc( size_t size );
  ```

- `void *` is a *generic* pointer, a pointer of unspecified type; this is why it needs to be converted (or "casted") to a specific type, e.g. `int *`

- The argument is the size of memory in bytes

- The `sizeof()` operator returns the size in bytes of C types

- In general terms:
  ```
  <pointer> = (<type> *) malloc(<size>*sizeof(<type>));
  ```
Allocating Structures

• Besides arrays, it is common to allocate structures, e.g.:

```c
typedef struct
{
    char word[32];
    int frequency;
} entry;

entry * ptr;
ptr = (entry *) malloc(sizeof(entry));
ptr->frequency = 0;
...
free (ptr);
```
What can we allocate?

• In principle we can allocate just a variable
• However, to allocate a memory block we need a pointer
• A pointer requires memory itself, and it needs to be declared
• Generally it is practical to only allocate blocks of memory larger than a pointer: arrays and structures
De-allocating memory: free

- The memory allocated through malloc **MUST** be always de-allocated through free!
  - Memory used by variables is automatically unreserved when programs exit
  - This is **NOT** the case for the memory allocated through malloc!!!
- Failing to de-allocate the memory may cause the computer to run out of memory and crash
- To de-allocate call: free (ptr);
Running out of Memory

• A call to malloc may fail because there is not enough available memory to allocate
  – This is particularly likely on embedded platforms
• In this case the pointer value returned by malloc is NULL:

```c
ptr = (int *) malloc(length * sizeof(int));
if (ptr == NULL)
{
    printf("memory allocation failed\n");
}
```
Quiz to check your understanding

• Which of the following is correct?

  – A. `int * b = malloc(10*sizeof(int));`
  – B. `int * b = malloc(10);`
  – C. `int * b = (int *) malloc(10);`
  – D. `int * b = (int *) malloc(10*sizeof(int));`
Quiz to check your understanding

- Which of the following is correct?

  - A.  
    ```c
    char* buffer = (char *) malloc(10*sizeof(char));
    free(buffer, 10);
    ```

  - B.  
    ```c
    char* buffer = (char *) malloc(10*sizeof(char));
    free(buffer, 10*sizeof(char));
    ```

  - C.  
    ```c
    char* buffer = (char *) malloc(10*sizeof(char));
    free(buffer);
    ```

  - D.  
    ```c
    char* buffer = (char *) malloc(10*sizeof(char));
    free(10*sizeof(buffer));
    ```
Quiz to check your understanding

• Which of the following is **not** correct?

  - A. `int * buffer = (int *) malloc(10*sizeof(int));`
  - B. `double * buffer = (double*) malloc(10*sizeof(double));`
  - C. `entry * ePtr = (entry *) malloc(10*sizeof(entry));`
  - D. `entry * ePtr = (entry *) malloc(sizeof(entry));`
  - E. none, they are all correct
Valid & invalid pointers

• After calling free, the pointer becomes invalid
  – If you try to access the memory again you get a memory access violation (or "segmentation fault" or "bus error")

• A useful convention is to always set unused pointers to NULL to mean "not pointing anywhere"
  – e.g. always set pointers to NULL when they are declared and after calling free
More reasons to allocate structures

- Structures can contain pointers to structures of the same type; through these it is possible to create data structures
- Easier to look at an example: listDemo.c
Linking Structures Through Pointers

Figure 11.4  Linked structures.

Figure 11.5 A linked list.

Removing an Element

Important!
If this was dynamically allocated, it needs to be de-allocated!!!

Traversing a List

- Start from pointer to first structure
- Move to following structures:
  - `pointer = pointer->next;`
- Last structure has next set to NULL, so we can use this information to stop the iteration
  - `while(pointer != NULL)`
- Notice importance of first pointer

Structure for the entire List

- Store a pointer to the first item
- Store a pointer to the current item (for example for when we transverse)
- Size can be useful

```c
typedef struct list_struct
{
    int size;
    list_item * first;
    list_item * current;
} list;
```
Structure for a List Item

• Generic list item:
  ```c
  typedef struct list_item_struct
  {
      /* some data here */
      struct list_item_struct * next;
  } list_item;
  ```

• For example:
  ```c
  typedef struct list_item_struct
  {
      char word[32];
      int freq;
      struct list_item_struct * next;
  } list_item;
  ```
Functions for a List

- It can be useful to define functions to do common operations on the list:
  - Create a list
  - Create and append new items
  - Remove items and destroy list
  - Print items and entire list
  - Find items in the list
  - Insert items
Functions for a List

- Example code: list.h and list.c
- In the C5 lab you will have to use and modify these files
Other Data Structures

- Double linked list
  - Advanced exercise in the lab
- Trees
- Graphs
Summary

- Sometimes in programs it is useful or necessary to store variable amounts of data
- Using malloc and free we can allocate and de-allocate memory, such as:
  - Variable size arrays
  - Linked lists
  - Other data structures such as trees and graphs