

CSc113

Advanced Programming

WK22: Linked Lists in C

Tony Chung

Today's Objectives

- Pointers vs. Values
- Passing by Value vs. Passing by Reference
- Memory Allocation
- Structures
- Object-Orientation in C
- Linked Lists Exercise

Pointers vs. Values

- A Value is the actual data held in memory.
 - Could be an integer, byte, etc
- A Pointer is the location in memory.
 - The address of the data.
- Any primitive in C can be accessed as a value or pointer.
 - For primitive types, the default is 'by value'.
 - To access the address (pointer) use the & symbol.
- You can instead declare a pointer (many reasons for this).
 - Declare like `int * name;`
 - You must tell it where the data is (manually allocating memory if needed using `malloc()`).
 - Access the data from the pointer using `*`

Passing by Value vs. Passing by Reference

- Take this function: `void f(int a, int * b){ ... }`
- It is called like this: `f(x, &b);`
 - Variable x is used directly and a copy of the value is passed.
 - Variable b must be passed by reference.
- The function `f()` can access the values of a and b like this:
 - `... int n = x; // Directly`
 - `... int m = *b; // Must de-reference first!`
- The function `f()` can update ONLY the value of b.
 - It knows where in memory b actually is.
 - `... *b = 9; // Don't forget *`
 - If we try to change a, we only change the local copy, not x.
- We thus have another way to return multiple items.

Memory Allocation

- If you have a local variable, like an int, C will do this automatically on the stack.
 - It's a local copy for the current function call.
 - Can pass the ptr around for other functions.
- You can also manually allocate memory. This is good if you don't know when compiling how big things will be.
 - `int * b = malloc(sizeof(int));`
 - `int * array = malloc(sizeof(int) * x);`
 - Gets you array[0] to array[x-1]...
- MUST free memory when done.
 - No Garbage Collector in C.
 - `free(b)` // No use of `*` as passing the ptr!
- Do not access a pointer after it is freed!
 - Set it to NULL!

Structures

- Structures are a convenient way of storing related data items.
 - Like a class, without methods.
 - Access members using the dot (.) operator as in Java.
 - Declare the type at the top (or in a header file) and declare as shown:
- ```
struct Person{
 char name[20];
 int age;
 Person * mother;
}
...
struct Person bob; // Declare an instance. Must use 'struct'
(hint: lookup 'typedef').
bob.age = 9; // Access member.
```

## Structures II

- Structures can also be manipulated using pointers.

```
struct Person * bill;
bill = memalloc(sizeof(struct Person));
bill->age = 65; // Note use of -> instead of .
```

- You can obviously get the address of a Struct:

```
struct Person * anne = &bob; // See last slide
```

## Object-Orientation in C

- C can be object oriented. You just do not get all the benefits!
- Use a structure and methods as an object.
- Can directly access members using .
- Write functions that take a structure pointer:
  - void setAge( struct Person \*, int age ){...} // In C Rather than...
  - public void setAge( int age ){...} // In Java
- Another example:
  - struct Person \* p = malloc( sizeof( struct Person ) );
  - setAge( p, 14 ); // Instead of...
  - Person p = new Person();
  - p.setAge( 14 );
- NB: You'd need to manually call a constructor function in C!

## Linked Lists Exercise

- Implement a Linked-List in C.
  - Make a Structure called LinkedListElement.
    - Contains an int and a pointer to the next one.
  - Write a loop that makes a four element LL.
    - Use malloc()...
    - Put numbers in it.
    - Keep only a reference to the first one.
  - Write a function called output()
    - void output( LinkedListElement \* l );
    - Outputs the number in the element.
    - Calls output on the next one.

## Finally

- There are two coursework deadlines this Friday, be sure to check if these apply to you...
  - Two people managed to submit coursework last term despite it not being applicable...