CS61C Spring 2010 **Week 3 - Memory** **Section 112/113**
TA: Bing Xia **cs61c-tb@imail.eecs**

Warm Up

1. How can we find the size of a given data type in bytes?
   `sizeof(type)`
2. How do you use `typedef`? Why use `typedef`?
   `typedef <type 1> <type 2>; // <type 2> is now equivalent to <type 1>`
   We use typedef to give new names to existing data types; for example, typedef-ing `size_t` to be an `unsigned int` because `size_t` is more descriptive.
3. Give a call to `malloc` that will return a pointer to an array of 17 `int*`s.
   `(int **) malloc(17 * sizeof(int *));`

Struct Practice

1. We want to add an inventory system to a text adventure game so that the player can collect items. First, we'll implement a *bag* data structure that holds *items* in a linked list. Each `item_t` has an associated weight, and each `bag_t` has a `max_weight` that determines its holding capacity (see the definitions below). In the left text area for `item_node_t`, define the necessary data type to serve as the nodes in a **linked list** of items, and in the right text area, add any necessary fields to the `bag_t` definition.

```
typedef struct item {
    int weight;
    // other fields not shown
} item_t;
```

```
typedef struct item_node {
    // (a) FILL IN HERE
    item_t *item;
    struct item_node *next;

} item_node_t;
```

```
typedef struct bag {
    int max_weight;
    int current_weight;
    // add other fields necessary
    // (b) FILL IN HERE
    item_node_t *items;


} bag_t;
```

2. Complete the add_item() function, which should add item into bag **only** if adding the item would not cause the weight of the bag contents to exceed the bag's max_weight. The function should return 0 if the item *could not* be added, or 1 if it succeeded. Be sure to update the bag's current_weight. You do not need to check if malloc() returns NULL. Insert the new item into the list wherever you wish.

```
int add_item(item_t *item, bag_t *bag) {
    if ( bag->current_weight + item->weight > bag->max_weight ) {
        return 0;
    }
    item_node_t *new_node = (item_node_t *) malloc(sizeof(item_node_t));
    // Add more code below…
    new_node->item = item;
    new_node->next = bag->items;
    bag->items = new_node;
    bag->current_weight += item->weight;
    return 1;
}
```

3. Finally, we want an empty_bag() function that frees the bag's linked list but **NOT** the memory of the items themselves and **NOT** the bag itself. The bag should then be "reset", ready for add_item. Assume that the operating system immediately fills any freed memory with garbage. Fill in the functions below.

```
void empty_bag(bag_t *bag) {
    free_contents( bag->items );
    // FILL IN HERE
    bag->items = NULL;
    bag->current_weight = 0;


}
```

```
void free_contents( item_node_t *node ) {
    // FILL IN HERE
    if(node->next != NULL)
        free_contents(node->next);
    free(node);

}
```

4. Now suppose that we care about the order of items in our bag. However, because we're clumsy, the only possible way for us to rearrange items is to reverse their order in the list.

```
void reverse_list(bag_t *bag) {
    item_node_t *head = bag->head, *new_list = NULL, *temp;
    while(head != NULL) {
        temp = head->next;
        head->next = new_list;
        new_list = head;
        head = temp;
    }

    bag->head = new_list;
}
```

Basic Memory Layout

Stack - grows down - holds local variables
Heap - grows up - where malloc() requests space
Static Data - fixed size - holds global variables
Code - fixed size - immutable - where instructions for program are

Questions 1 and 2 refer to the C code to the right.

```
#define val 16
char arr[] = "foo";
void foo(int arg){
    char *str = (char *) malloc (val);
    char *ptr = arr;
}
```

1. In which memory sections (code, static, heap, stack) do the following reside?

arg stack arr static *str heap val nowhere

2. Name a C operation that would treat `arr` and `ptr` differently: pointer assignment

3 Memory Allocation Schemes

Best-fit - choose the smallest block that satisfies the request
First-fit - choose the first block that satisfies the request starting from the front
Next-fit - choose the first block that satisfies the request starting from the where the last request finished

Exercise: Given a heap with an 16 byte capacity, generate a series of malloc()s and free()s for which each allocation scheme fails where others may succeed.

Best-Fit:

Best fit fails, first fit succeeds:

```
a = malloc(7)
b = malloc(1)
c = malloc(2)
d = malloc(6)
free(a)
free(c)
e = malloc(2)
free(b)
free(d)
f = malloc(14) // best fit fails
```

First-Fit:

First fit fails, best fit succeeds:

```
a = malloc(7)
b = malloc(7)
free(a)
c = malloc(2)
d = malloc(7) // first fit fails
```

Next-Fit:

Next fit fails, first fit succeeds:

```
a = malloc(5)
b = malloc(5)
free(a)
c = malloc(5)
d = malloc(6) // next fit fails
```