CSE 351: Section 10

Memory Allocation

Memory Allocation

- Must allocate any memory you need to use
- Allocating memory = designating a block of it as 'used'
  - This way, you don't overwrite your data
  - The memory isn't changed, but compiler knows that memory is allocated
- There are three general types of memory allocation
  - Static
  - Automatic
  - Dynamic

Static Memory Allocation

- Unlike the other two methods, static memory allocation occurs at compile-time
- Memory allocated statically exists for the duration of the program

Example:

```
int global_int_outside_main;
int main (int argc, char *argv[]){...}
```

Automatic Memory Allocation

- Occurs at run-time
- Automatically-allocated variables only last for the duration of the function call
- Automatically-allocated variables are stored on the stack

Example:

```
int myFunction(int a, int b){
    int local_var_on_the_stack = 5;
    ...
}
```

Dynamic Memory Allocation

- Also occurs at run-time
- However, the programmer has control over the lifespan of dynamically-allocated variables
- The x86 implementation of dynamic memory allocation is heap-based
  - Means that variables declared dynamically are stored on the heap
- In C/C++, the programmer is responsible for
  - Allocating space for dynamic variables [malloc()]
  - Freeing up that space when the variable is no longer needed [free()]

Advantages of Dynamic Allocation

- Say you want to create a linked list in C
- Example requirements:
  - You want this list to exist for roughly the duration of the program and be accessible to many different functions
  - You want to be able to add/remove elements from the list on the fly without worrying about exceeding a length bound
- Dynamic allocation is the only way to make this possible

Linked List C Implementation

- How will this work in C?
- Each node could look like this:

```
struct node {
   int data;
   struct node *next;
};
```

- Now...where do we put these structs?

Note: this is not the free list

Dynamic Allocation in C

- The C Standard Library provides several functions that allocate memory dynamically
- We will use the `malloc()` function to allocate space in the heap, and the `free()` function to free up the space when we no longer need it
- We will also use the `sizeof()` function to determine how many bytes are required for certain types

`malloc()`

- Parameter
  - The number of bytes to be allocated
- Returns:
  - Pointer to allocated block of memory or NULL if it fails
- Example: `int *int_ptr = (int *)malloc(4);`
  - Hard-coding a size is generally bad style, because the `int` datatype can vary across systems
  - `sizeof()` returns the size of a particular datatype (in bytes), so we can pass it as an argument to `malloc()`:

    ```
    int *int_ptr = (int *)malloc(sizeof(int));
    ```
  - It's good practice to cast the pointer from `malloc()`

`free()`

- Parameter:
  - A pointer to a block of memory to be freed
- Returns:
  - Nothing!
- Example:

```
int *t = (int *)malloc(sizeof(int));
free(t);
```

- The code snippet above would simply allocate a block of 4 bytes on the heap, then free it for the program to use later

Putting it together

```
struct node *list = NULL; // head of list, starts empty

void insert_front(struct node *head, int n) {
    struct node *new_node =
        (struct node *)malloc(sizeof(struct node));
    if(new_node == NULL){ printf("Error!"); return;}
    new_node->data = n;
    if (head == NULL)
        head = new_node;
    else {
        new_node->next = head;
        head = new_node;
    }
}
```

Dynamic Allocation Guidelines

- Always check for NULL from malloc()
- Always initialize what you get from malloc()
- Always free() what you malloc()
- After calling free(), set pointers to NULL
- Use casting to say how to view the memory

  Ex: `(int *) malloc(N * sizeof(int));`
- Never use more than you allocate
- Never free() something you didn't malloc
- Never free() twice

How `malloc()` works

- In the heap, it keeps a linked list of free blocks called a "free list"
- When a block needs to be allocated, it searches the free list for a block of the right size, breaks off a chunk of the necessary size, and adds the remainder back into the free list
- Thus, if you use more memory than you allocated, you will corrupt the linked list
- This is a basic description of the behavior you will need to implement in Lab 5!

GDB Linked List Demo

Source file:

http://www.cs.washington.edu/education/courses/cse351/12au/section-slides/tiny_linked_list.c

GDB command list:

http://www.cs.washington.edu/education/courses/cse351/12au/section-slides/gdb_linked_list.txt

Lab 5

- You will be responsible for implementing two functions
  - `mm_malloc()`, which allocates memory in the heap
  - `mm_free()`, which frees memory in the heap
- There is plenty of starter code, you just need to manipulate the free list correctly (which is easier said than done, so start early!)
- For exhaustive free list diagrams, see these slides put together last spring: http://www.cs.washington.edu/education/courses/cse351/12sp/section-slides/section-9.pdf