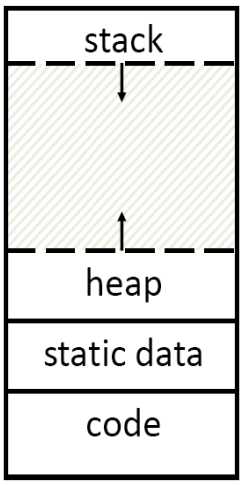
Discussion #3: Memory Management

Written by Justin Hsia (6/27/2011)

Memory Layout:



The address space is represented graphically on the left. The lowest address is at the bottom and the highest address is at the top. It contains the following four regions:

Stack: LIFO stack of frames (local variable environments). Holds local variables and grows downwards (advance by reducing address).

Heap: Holds dynamically allocated memory (malloc). Grows "upward," but not necessarily in a linear fashion (see section below).

Static Data: Global variables. Size does not change, but contents can.

Code: Keeps a copy of your code. Does not change.

Registers:

Although not technically a part of memory layout, it's good to remember that these exist. Most small operations will be handled entirely in registers, removing the need for memory access, which is slow.

Check: For the code on the left, which memory sections would the quantities on the right reside?

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

```
arg?
arr?
str?
*str?
val?
```

Check: A function call causes an additional frame to be added to the stack. Although elegant, what is a potential problem with recursive procedures?

Dynamic Memory Allocation:

Why is dynamic memory allocation desirable?

1) *Persistence* – allocated memory stays around and does not become "garbage" as the stack changes.
2) *Dynamic sizing* – we might not know beforehand how much space we need. Useful for dealing with inputs of unknown size (such as storing into a linked list).

`sizeof` operator:

The `sizeof` operator yields the number of bytes required to store an object of the type of its operand. You can pass it a variable such as `sizeof(i)` or pass it the name of the type, such as `sizeof(int)`. This is VERY useful for figuring out how much space to request from `malloc` especially for structs. The returned value will essentially be an unsigned integer.

Note: When passed the name of an array, `sizeof` will return the total size of the array. When passed a pointer, `sizeof` will return the size of the pointer. This is another difference between arrays and pointers!

`malloc` and `free`:

These are the two functions we use to dynamically allocate memory:

```
void *malloc(size_t size)
void free(void *p)
```

Don't fret about the void pointers: basically `free` can take any pointer, though the pointer returned by `malloc` must be cast into the proper type before storing. `malloc` returns `NULL` if the allocation request cannot be satisfied. `free` will only work if it is passed the exact same address that `malloc` returned when allocating the space. As usual, the space returned by `malloc` is uninitialized.

Fragmentation and allocation policies:

As mentioned before, dynamically-allocated memory does not "grow" the same way the stack does. Allocated memory cannot be moved because it must remain available to the user at the address supplied to it. After much use of `malloc`/`free`, we get *fragmentation* when the heap gets separated into small chunks.

~~This iteration of CS61C is brushing over allocation policies and different schemes for combatting fragmentation.~~ If you're interested in reading more about it for your own edification, K&R section 8.7 contains one possible implementation (good practice reading C code, too!).

Don't fret the details, but here are some notes on the general schemes just FYI:

Best-fit: Choose smallest block that is big enough to fulfill the request. Tries to limit wasted space, but takes time to examine ALL free blocks and also leaves lots of small blocks.

First-fit: Choose first block that is big enough. Fast, but potentially more wasted space. Also concentrates small blocks at the beginning.

Next-fit: Same as first-fit, but start search at last memory allocated instead of start of free list. Does not have concentration of small blocks that first-fit does.

Memory Leaks:

You have a limited amount of space to use for both the stack and the heap and you don't want to waste any of it. Dynamically allocated memory is persistent, so it remains in the heap even if you get rid of/change the pointer that pointed to it. Once this happens, the allocated memory essentially becomes unusable because you can't point to it and `malloc` thinks you're still using it.

Make sure you call `free` for every instance you call `malloc`!

Check: For the singly-linked list implementation below, fill out `free_ll`, which frees all of the memory allocated for the linked list. Try writing both a recursive and an iterative solution.

```
struct ll_node {
      struct ll_node *next;
      int *element;
}

void free_ll(struct ll_node *list) {
      /* YOUR CODE HERE */
}
```

What happens if element in the struct is an integer instead of a pointer? Does your code become easier or more complicated?