

CMSC 216

Introduction to Computer Systems

Lecture 11

Dynamic Memory Allocation

Jan Plane & Alan Sussman
{jplane, als}@cs.umd.edu

Administrivia

- Still working on Project 1 grades
- Project 2 due Friday
- Project 3 posted by Monday
- Read Chapter 11, Reek
- Exam #1 Thursday
 - for 12:30 and 2:00 lectures, exam is in CSIC 1115

(Not in the textbooks)

MAKE

A more complex makefile

```
# comments start with a pound sign and last until the end of the line

CC = gcc
CFLAGS = -ansi -pedantic-errors -Wall -Werror
PROGS = int_count table

```

macro definitions

```
all: $(PROGS)

```

builds the two top-level programs

```
int_count: int_count.o
    $(CC) -o int_count int_count.o

table: table.o main.o hash_table.o
    $(CC) -o table table.o main.o hash_table.o

```

note no \$(CFLAGS) in these

```
table.o: table.c table.h
hash_table.o: table.h
main.o: table.h
int_count.o: table.h

```

use implicit rules

```
clean:
    rm -f *.o $(PROGS)

```

not part of the main tree -
built by typing "make clean"

More about header files

- Note that header files should contain only declarations, with no executable code
- Header files are not directly compiled (for example, there were no rules to compile any `.h` files in the previous example); they just get included in source files, which are compiled
- Header files should contain declarations that the compiler needs to see to be able to compile executable code
 - examples include function prototypes, typedefs, and **extern** global variables

DYNAMIC MEMORY ALLOCATION

Stack vs. Heap Memory

- Stack memory is allocated and deallocated in a specific order
 - you can only remove from and add to the top of the stack
 - any item in the stack is always freed before the item below it
 - useful for local variables, since functions work in similar manner
- Heap memory
 - can be allocated and deallocated in any order while the program is running

Dynamically allocated memory

- Why use it?
 - to eliminate compile-time limits:
 - Often, the size of a data structure isn't known until runtime
 - Allocating a huge amount of space to fit every possible scenario can be error-prone at worst and a waste of space at best
- User-managed heap vs. Garbage collection
 - garbage collection:
 - frees programmer from burden of keeping track of objects (convenience)
 - slows down execution due to collection needing to be run periodically; collection is also expensive
 - user-managed heap
 - programmer can control the precise time to deallocate objects
 - requires a lot more care - errors can occur if the wrong thing is freed, or the right thing is freed at the wrong time

Memory management functions

- Allocating memory:

void *malloc(size_t amount);

- allocates **amount** bytes of memory (if available) from the heap and returns a pointer to the beginning of it
- no initialization of the space

void *calloc(size_t count, size_t obj_size);

- allocates **count** objects of size **obj_size** each (if memory is available), returns a pointer to beginning of it
- initializes all the space to 0

– both return **NULL** if the allocation fails

– both require **#include <stdlib.h>** since prototypes are contained there

- Note that both these functions return a **void ***

Memory management functions, cont.

- Deallocating memory

void free(void *ptr);

- returns the memory pointed to by **ptr** to the free pool in the heap
- memory **MUST** have been previously allocated from the heap
- does not change **ptr** (why not?), so you may want to set **ptr** to **NULL** after calling **free()** if you might accidentally use **ptr** again
- free(NULL);** does nothing - it's perfectly OK to do

More about **free()**

- Once you've freed memory, it goes back to the heap, meaning you can't trust its value to remain the same (or even stay valid!)
- It's up to you to ensure you don't dereference a freed pointer - otherwise, weird things can happen ...

Don't dereference freed pointers

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int x;
    int y;
} Point;

int main() {
    Point p = {3, 4}, *p1, *p2;
    p1 = malloc(sizeof(Point)); /* no NULL check! Baaad... */
    p1->x = 5;
    p1->y = 9;
    printf("%d %d\n", p1->x, p1->y);
    free(p1); /* now p1 points to invalid memory */
    p2 = malloc(sizeof(Point));
    *p2 = p;
    printf("%d %d\n", p1->x, p1->y); /* Printing out p1, not p2... */
    return 0;
}
```

Pointer aliases

- We can have two pointers pointing to the same address
- But what happens when we free one of them?

```
int *p, *q;  
p = malloc(sizeof(int));  
q = p;  
free(p);  
p = NULL;  
*q = 42; /* what is the value of q? */
```
- **q** is called a dangling pointer

Common errors

- Forgetting to check the return value from **malloc()** for **NULL**
- Not initializing the memory **malloc()** returns
- Not allocating enough space:

```
char string[] = "Inconceivable!";  
char *p = malloc(strlen(string));  
strcpy(p, string); /* Oops... */
```

Common errors, cont.

- Attempting to free non-heap memory:

```
int i, *p;  
p = &i;  
free(p); /* ??? Undefined operation */
```
- Freeing something that's not the **beginning** of a dynamically allocated block:

```
int *p = calloc(10, sizeof(int));  
free(p + 3);
```

Common errors, cont.

- Dereferencing a garbage pointer

```
int *p;  
*p = 42;
```
- Dereferencing a **NULL** pointer

```
int *p;  
p = NULL;  
*p = 23;
```
- Dereferencing a dangling pointer

```
int *p = malloc(...);  
free(p);  
...  
*p = 27;
```

Common errors, cont.

- Referring to data past the end of allocated space
 - Remember the string example?
- Using freed memory

```
List_node *ptr =  
    malloc(sizeof(List_node));  
...  
free(ptr);  
...  
ptr = ptr->next; /* Uh oh... */
```

Common errors, cont.

- What happens to the first **Stack** object in Java?
Stack s = new Stack();
...
s = new Stack();
- Losing a pointer to dynamically allocated memory causes memory leaks
 - not an immediately fatal error
 - can just be a waste of resources, but left to run for a while, it can keep your program from being able to do anything

Operating on blocks of memory

- Much as we have functions like **strcpy()** that operate on strings of characters, we can copy whole blocks of memory (or strings of bytes, if you like)
void *memcpy(void *dst, void *src, size_t n);
void *memmove(void *dst, void *src, size_t n);
- Both copy **n** bytes from the memory starting at **src** to the memory starting at **dst**, and return the **dst** pointer
- **memcpy()** cannot be used on overlapping arrays, though; but it's faster than **memmove()**
- Both prototypes contained in **string.h**

A possible **memcpy()** implementation

```
#include <stdio.h>  
/* not #including <string.h> because of conflict */  
  
void *memcpy(void *dst, void *src, size_t n) {  
    char *dp = dst;  
    char *sp = src;  
    while (dp - (char *) dst < n)  
        *dp++ = *sp++;  
    return dst;  
}  
  
int main() {  
    int i, j = 5;  
    memcpy(&i, &j, sizeof(int));  
    printf("%d\n", i);  
    return 0;  
}
```