DYNAMIC MEMORY ALLOCATION IN C

CSSE 120—Rose Hulman Institute of Technology

Final Exam Facts

- Date: Monday, May 24, 2010
- Time: 6 p.m. to 10 p.m.
- Venue: **O167 or O169 (your choice)**
- Organization: A paper part and a computer part, similar to the first 2 exams.
  - **The paper part will emphasize both C and Python.**
    - You may bring two double-sided sheets of paper this time.
    - There will be a portion in which we will ask you to **compare and contrast C and Python** language features and properties.
  - **The computer part will be in C.**
    - The computer part will be worth approximately 65% of the total.

Sample Project for Today

- Check out ***29-MallocSample*** from your SVN Repository
- Verify that it runs, get help if it doesn't
  - Don’t worry about its code yet; we’ll examine it together soon.

Memory Requirements

- Any variable requires a certain amount of memory.
- Primitives, such an `int`, `double`, and `char`, typically may require between 1 and 8 bytes, depending on the desired precision, architecture, and Operating System's support.
- Complex variables such as ***structs***, ***arrays***, and ***strings*** typically require as many bytes as their components.

How large is this?

- `sizeof` operator gives the number bytes needed to store a value
- `sizeof(char)`
- `sizeof(char*)`
- `sizeof(int)`
- `sizeof(float)`
- `sizeof(double)`
- `sizeof(student)`
- `sizeof(jose)`
-
```
printf("size of char is %d bytes.\n",
        sizeof(char));
```

```
typedef struct {
        char* name;
        int year;
        double gpa;
} student;
```

```
char* firstName;
int terms;
double scores;
student jose;
```

Examine the beginning of ***main*** of **29-MallocSample**. Run it and use the results to answer Q3-5 of your quiz. Ask about the questions that you are not sure of.

Q3-5

Memory Allocation

- In many programming languages, memory gets dynamically allocated as the need arises.
- Example: Lists in Python grow and shrink as we add or remove items from them.
- In Python, memory gets allocated as the need arises.
- Memory gets freed up when it is no longer needed.
  - By the "garbage collector"
  - When is memory no longer needed? (details on next slide)

Static Memory Allocation

- In C, we have the ability to manually allocate memory.
  - We typically do this when we know ahead of time the storage needs of a complex data-structure.
- We have seen this last time, when we did this:

```
char string[10];
```

  - We allocated ten bytes to store a string.
  - In some of the examples, we used all of the allocated bytes, in some, we did not.
- What memory is allocated by the example to the right? When? When is it returned to the system?

```
void foo(int x, char* p) {
    double y;
    char string[10];
    ...
}
```

  - This is called ***static allocation***. The memory is allocated from the ***stack***.

Dynamic Memory allocation in C

- We use the `malloc` command to dynamically allocate memory on the heap.
- The syntax is:

```
malloc(<size>);
```

- The command returns a pointer to a memory location.
- We typically want to store that pointer.

Example: Dynamic Memory allocation in C

- Suppose we want to reserve space for 10 doubles.
- We would do:

```
double* samples;

samples = (double*) malloc(count * sizeof(double));
```

- The memory returned to you can store objects of any type (void pointer).
- We give it the desired type by *typecasting*.
  - That's the **(double*)**

Deallocation of Dynamic Memory

- When we allocate memory, we also need to free it up when we are done with it.
- This is only necessary when we dynamically allocate memory (using constructs like **malloc()**).
  - Remember, ***static*** allocation allocates memory when the function is entered and deallocates memory when the function exits.
- Otherwise, we may well run out of the memory space allocated to us.

Memory Deallocation in C

- In order to deallocate memory, we use the `free` command
- The syntax is:

```
free(<pointer>);
```

- To continue our example, we would do:

```
free(result);
```

Returning Arrays from Functions

- In *maf-main*.c, remove the **exit()** call near the beginning.
- Run the program:
  - What happens?
  - Why?
- Original version of **getSamples()** just creates local storage that is recycled when function is done!
- If we want samples to ***persist beyond the function's lifetime***, we need to allocate memory using "malloc".
  - Also need to **#include <stdlib.h>**

Dynamically allocating an array

```
double *getSamples(int count) {
    double* samples;
    samples = (double *) malloc(count * sizeof(double));
    if (samples == NULL) {
        exit(EXIT_FAILURE);
    }


    int i;
    for (i = 0; i < count; i++) {
        samples[i] = gaussian(82.5, 7.1);
    }
    return samples;
}
```

Typecast to desired pointer type

returns a void pointer (void *) to memory of specified size or NULL if request fails. Memory is uninitialized

Exit program if out of memory or cannot allocate for another reason

Q9,10

Using Dynamically Allocated Array

```
double* sampleA;
double* sampleB;
int sampleCount = 5;

sampleA = getSamples(sampleCount);
sampleB = getSamples(sampleCount);

for (i = 0; i < sampleCount; i++) {
      printf(%0.1lf\n", sampleA[i] + sampleB[i]);
}

free(sampleA);
free(sampleB);
```

Don't forget to free the memory that was previously "malloc-ed".

Recap: sizeof, malloc and free

- sizeof operator: gives the number of bytes needed to store a value
- malloc(<amount>): returns a pointer to space for an object of size *amount*, or NULL if the request cannot be satisfied. The space is uninitialized.
- void free(void *p): deallocates the space pointed to by p; does nothing if p is NULL. p must point to memory that was previously dynamically allocated.

Descriptions from K&R, p. 252

Hidden: Dynamically allocating strings

- Consider:

```
char* s1 = "Sams shop stocks short spotted socks. ";
char* s2;
```

- What if we wanted to create a copy of s1 and store it in s2 ?

```
s2 = (char *) malloc((strlen(s1) + 1) * sizeof(char));
strcpy(s2, s1);
```

- free(s2) when s2 is no longer needed.

Dynamically Allocating Structs

- Can use **malloc** to dynamically allocate **struct**s
- Will need to use pointers to structs
  - student *zeb;
- Accessing elements of structs is different with pointers…

Pointers to Structs

- Direct reference

```
student debby = {"Deb", 2011, 2.9};
debby.gpa = 3.2;
printf("%s, Class of %d\n",
    debby.name, debby.year);
```

- Use dot when you have the struct directly

- Pointer reference

```
student *aaron;
aaron = (student *)
    malloc(sizeof(student));
aaron->name = "Aaron";
aaron->year = 2009;
aaron->gpa = 3.1;
printf("%s, Class of %d\n",
    aaron->name,aaron->year);
```

- Use "arrow" when you have a pointer to it

aaron->gpa is shorthand for (*aaron).gpa

Q12

Summary: Overcoming some array limitations

- **`malloc`** reserves space for variables or arrays in a separate location in memory called the heap
  - It allows the return type of a function to be an array
  - It allows arrays to be resized
- Keywords:
  - `ptr = malloc(number_of_bytes_needed)`
  - `sizeof()`
  - `free(ptr)`
  - `ptr = realloc(ptr, number_of_bytes_needed)`
- What does realloc do()

Q13,14