

DYNAMIC MEMORY ALLOCATION, POINTERS TO STRUCTS

Final Exam Facts

- **Date:** Wednesday, Feb 25, 2009
- **Time:** 8:00AM to 12:00 PM
- **Venue:** O267, O269, O231, O233
- **Chapters:** Zelle chapters 1 to 12, Assigned C readings from Schildt plus Web resources linked from ANGEL Resources page
- You may bring two sheets of paper this time.

Final Exam Facts

- **Organization:** A paper part and a computer part, similar to the first 2 exams.
- The paper and pencil part will emphasize the C materials.
- 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 less than 50% of the total.

Memory Requirements

- Any variable requires a certain amount of memory.
- Primitives, such as **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(int)`
- `sizeof(double)`
- `sizeof(char *)`
- `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;
```

How large is this?

- 32 bits = 4 bytes**
- int : 4 bytes**
- double : 8 bytes**
- char : 1 byte**
- pointer : 4 bytes**

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.

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.

Memory Deallocation

- When we allocate memory, we also need to free it up when we are done with it.
- Otherwise, we may well run out of the memory space allocated to us.

Memory allocation in C

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

```
malloc(<size>);
```
- The command returns a pointer to a memory location.
- We typically want to store that pointer.

Memory allocation in C - Example

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

```
double *result;  
result = (double *) malloc(count * sizeof(double));
```
- Memory is returned to as typeless.
- We give it a type by typecasting.

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);
```

Sample Project for Today

- Check out ***MallocSample*** from your SVN Repository
- Verify that it runs, get help if it doesn't

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

Typecast to desired pointer type

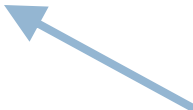
```
double *getSamples(int count) {  
    double *result;  
    result = (double *) malloc(count * sizeof(double));  
    if (result == NULL) {  
        exit(EXIT_FAILURE);  
    }  
    int i;  
    for (i=0; i<count; i++) {  
        result[i] = gaussian(82.5, 7.1);  
    }  
    return result;  
}
```

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

Using Dynamically Allocated Array

```
double *sampleA;  
double *sampleB;  
int sampleCount = 5;  
  
sampleA = getSamples(sampleCount);  
sampleB = getSamples(sampleCount);  
  
for (i=0; i<sampleCount; i++) {  
    printf%0.11f\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.

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 **structs**
- We'll use this to create an Array data type soon that's "smarter" than the basic C version
- 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

Stop, Practice Time

- Problem:
 - ▣ One nice feature of lists in Python is that they "know" their own length
 - ▣ Suppose we want that in C
- Solution:
 - ▣ Make our own `Array` type and helper functions!
- Homework:
 - ▣ Check out ***SmarterArrays*** from your SVN repository
 - ▣ See homework description linked from ANGEL