As you arrive:

1. Start up your computer and plug it in
2. ***Log into Angel*** and go to CSSE 120
3. Do the ***Attendance Widget*** – the PIN is on the board
4. Go to the course ***Schedule Page***
5. Open the ***Slides*** for today if you wish
6. Check out today's first project: `Session29_MallocSample`

Plus in-class time working on these concepts AND practicing previous concepts, continued as homework.

Dynamic Memory Allocation

- What it is
- How Python does it: garbage collection
- How C does it: malloc, free

Time to work on your project

Final Exam Facts

- Date: Thursday, November 18, 2010
- Time: 8 a.m. to noon
- Venue: **O167, O169, O157, O159 (sections 1 to 4, respectively)**
- 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 50% of the total.

Q1-2

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));`

1 byte = 8 bits

On our system:

char:	1 byte
int:	4 bytes
float:	4 bytes
double:	8 bytes
pointer:	4 bytes
student:	16 bytes

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

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

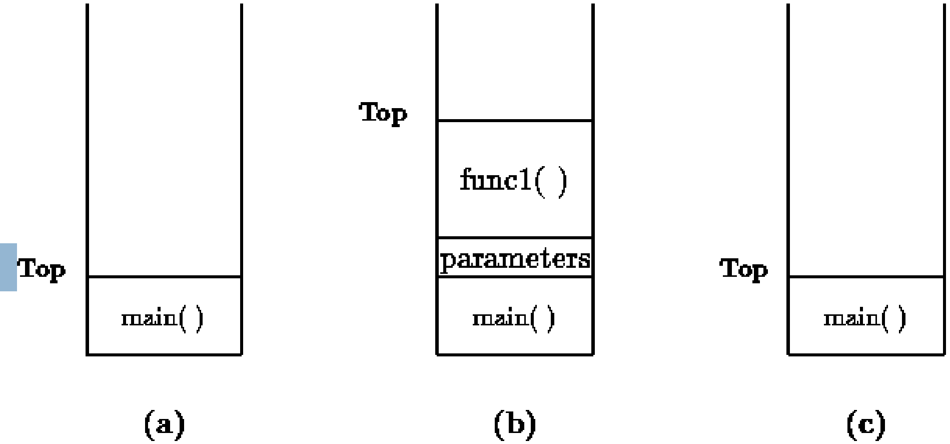
Examine the beginning of ***main_MallocSample*** of **Session29_MallocSample**.
Run it and use the results to answer Q3-5 of your quiz.
Ask about the questions that you are not sure of.

Q3-5

Memory Allocation

- In many programming languages, memory gets dynamically ***allocated as the need arises.***
- Example: In Python:
  - Lists grow and shrink as we add to or remove items from them.
  - Space for objects is allocated when the object is constructed
- In such languages, memory gets ***freed up when it is no longer needed.***
  - By the "garbage collector"
  - When is memory no longer needed? Answer: When nothing refers to it (also see 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 previously, 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>);
```

  - That is, `malloc` takes an integer that specifies *the size in bytes* to be allocated
- `malloc` 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 can store objects of **any type** (***void* pointer**).
- We give it the desired type by ***typecasting***.
  - That's the `(double *)`
    - Notation for typecasting: put the type to which to cast ***in parentheses***. The ***next expression*** is "converted" to that type.

WRONG! **z** becomes **2**, since C does *integer* division on integers

```
int x = 8;
int y = 3;
double z;
z = x / y;
```

```
int x = 8;
int y = 3;
double z;
z = ((double) x) / y;
```

RIGHT! Must ***cast*** the integers. **Necessary in AroundTheWorld!**

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:

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

   // You can use samples here just like an array, e.g.
for (k = 0; k < count; ++k) {
    ... samples[k] ...
}

free(samples);
```

Allocate the space.

Use the space.

When you are done with this storage (array), free up its space. Otherwise, you have a ***memory leak***.

Returning Arrays from Functions

- In ***main_MallocSample.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>**

Q6-7

Dynamically allocating an array

Typecast to desired pointer type

```
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;
}
```

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

OK to use array notation even though declared as a pointer

Q8-9

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".

Q10

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

Summary: Overcoming some array limitations with *dynamic memory allocation*

- `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:
  - `float *ptr;`
    `ptr = (float *) malloc(number_of_bytes_needed)`
  - `sizeof()`
  - `ptr = (float *) realloc(ptr, number_of_bytes_needed)`
  - `free(ptr)`

With a similar typecast for pointers to other types.

Q11-12

Your C Capstone Project

- Work on it the rest of today.
- Individual or with a partner.
- Due Saturday at 11:59 p.m.
- Strive to maintain ***at least*** this schedule:
  - In class Monday: ask your instructor to demo the project and explain it.
  - Monday night: read the project instructions and bring questions to class Tuesday.
  - Monday night: Sketch the organization of your project -- what functions will you need to write? What structures? Consider drawing a structure diagram.
  - Tuesday: Examine the **TestScores** project that we gave you.
    - You don't need to understand all of it at this point, but you DO need to know WHAT IT COULD HELP YOU WITH.
    - Refer to the TestScores project as needed for details on how to do some of the steps.
  - Tuesday and Wednesday: Begin implementing.
  - Thursday: Bring your questions to class.
  - Thursday through Saturday: Finish implementing.