DYNAMIC MEMORY ALLOCATION, POINTERS TO STRUCTS

CSSE 120—Rose Hulman Institute of Technology

Final Exam Facts

- Date: Monday, May 19, 2008
- Time: 6:00 to 10:00 PM
- Venue: A219
- Chapters: Zelle chapters 1 to 12 & C material (Angel → Course Resources → C Programming)
- Organization: A paper part and a **computer part (in C only)**, just as on the first 2 exams.
  - Computer access allowed for both parts of the exam since no C text

Sample Project for Today

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

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;
```

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 reused when function is done!
- If we want samples to persist, we need to allocate memory using "malloc".
  - Also need to **#include <stdlib.h>**

Dynamically allocating an array

```
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;
}
```

Typecast to desired pointer type

returns a void pointer (void *) to space of specified size or NULL if request fails. Space 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%0.1lf\n", sampleA[i] + sampleB[i]);
}

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

Don't forget to free the memory that was previously "malloced".

Recap: sizeof, malloc and free

- sizeof operator: gives the number of bytes needed to store a value
- void *malloc(size_t 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 be a space previously allocated.

Descriptions from K&R, p. 252

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 ?

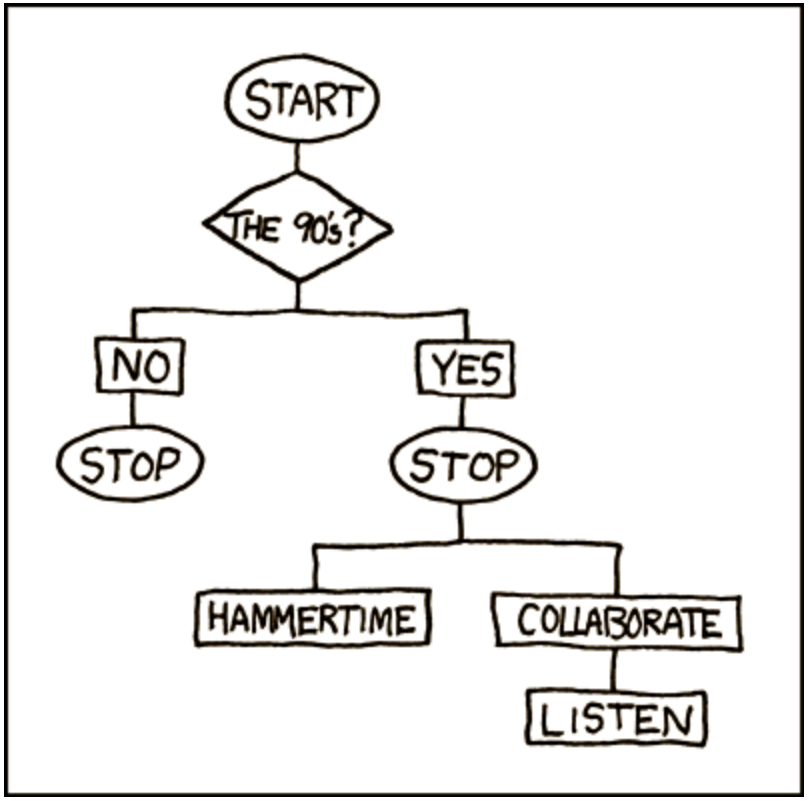
```
s2 = (char *) malloc((strlen(s1) + 1) * sizeof(char));
strcpy(s2, s1);
```

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

Cousins of malloc

- void *calloc(size_t n, size_t el_size): returns a pointer to contiguous space in memory allocated for an array of n elements, each element requiring el_size bytes. Returns NULL if the request cannot be satisfied. The space is initialized with all bits set to zero.
  - Example: **samples = (double *) calloc(count, sizeof(double));**
- void *realloc(void *ptr, size_t amount): changes the size of the block pointed to by ptr to amount bytes. The content of the space will be unchanged up to the lesser of the old and new size. Any new space is not initialized.
  - Example: **str = (char *) realloc(str, strlen(str) + 1);**

Break—90's Flowchart, xkcd.com



Freestyle rapping is basically applied Markov chains.

Dynamically Allocating Structs

- Can use **malloc** to dynamically allocate **struct**s
- 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