CSE 31
Computer Organization

Lecture 6 – Dynamic memory allocation and C structs

Announcements

- Labs
  - Lab 2 due this week (**with 7 days grace period** after due date)
    - Demo is REQUIRED to receive full credit
  - Lab 3 out this week
    - Due at 11:59pm on the same day of your next lab (with 7 days grace period after due date)
    - You must demo your submission to your TA within 14 days from posting of lab
    - Demo is REQUIRED to receive full credit
- Reading assignments
  - Chapter 4-6 of K&R (C book) to review C/C++ programming
  - Reading 01 (zyBooks 1.1 – 1.5) due 13-FEB
    - Complete **Participation Activities** in each section to receive grade towards Participation
    - IMPORTANT: Make sure to submit score to CatCourses by using the link provided on CatCourses
- Homework assignment
  - Homework 01 (zyBooks 1.1 – 1.5) due 20-FEB
    - Complete **Challenge Activities** in each section to receive grade towards Homework
    - IMPORTANT: Make sure to submit score to CatCourses by using the link provided on CatCourses

Announcements

- Project 01
  - Due 17-MAR
  - Can work in teams of 2 students
    - » Each team member must identify teammate in "Comments…" text-box at the submission page
    - » If working in teams, each student must submit code (can be the same as teammate) and demo individually
    - » Grade can vary among teammates depending on demo
  - Demo required for project grade
    - » No partial credit for submission without demo
  - No grace period
    - » Must complete submission and demo by due date.

Dynamic Memory Allocation (1/4)

- C has **`sizeof()`** which gives size in bytes (of type/variable)

- To assume the size of objects can be misleading and is bad style, so use **`sizeof(type)`**
  - Many years ago, an `int` was 16 bits, and programs were written with this assumption.
  - What is the size of integers now?

- **`sizeof()`** knows the size of arrays:

```
int ar[3]; // Or: int ar[] = {54, 47, 99}
sizeof(ar); // Should be 12
```

  - … as well of arrays whose size is determined at run-time:

```
int n = 3;
int ar[n]; // Or: int ar[funcThatReturns3()];
sizeof(ar) // Should be 12
```

Dynamic Memory Allocation (2/4)

- To allocate room for something new to point to, use `malloc()` (with the help of a typecast and `sizeof`):

```
ptr = (int *) malloc (sizeof(int));
```

  - Now, `ptr` points to a space somewhere in memory of size `(sizeof(int))` in bytes.
  - `(int *)` simply tells the compiler what will go into that space (called a typecast).

- `malloc` is almost never used for 1 value

```
ptr = (int *) malloc (n*sizeof(int));
```

  - This allocates an array of **`n`** integers.

Dynamic Memory Allocation (3/4)

- Once `malloc()` is called, the memory location can contain garbage, so don’t use it until you’ve initialized it.

- After dynamically allocating space, we must dynamically free it:

  ```
  free(ptr);
  ```

- Use this command to clean up.
  - Even though the program `frees` all memory on `exit` (or when `main` returns), don’t be lazy!
  - You never know when your `main` will get transformed into a subroutine!

Dynamic Memory Allocation (4/4)

- The following two things will cause your program to crash or behave strangely later on, and cause VERY VERY hard to figure out bugs:
  - `free()`ing the same piece of memory twice
  - calling `free()` on something you didn't get back from `malloc()`

- The runtime **does not** check for these mistakes
  - Memory allocation is so performance-critical that there just isn't time to do this
  - The usual result is that you corrupt the memory allocator's internal structure
  - You won't find out until much later on, in a totally unrelated part of your code!

C structures : Overview

- A `struct` is a data structure composed from simpler data types.
  - Like a class in Java/C++ but without methods or inheritance.

```
...
struct point {  /* type definition */
    int x;
    int y;
};

void PrintPoint(struct point p){

    printf(“(%d, %d)”, p.x, p.y);
}

int main() {
      struct point p1 = {0,10}; /* x=0, y=10 */
      PrintPoint(p1);
      ...
}
```

As always in C, the argument is passed by “value” – a copy is made.

C structures: Pointers to them

- Usually, more efficient to pass a pointer to the struct.
- The C arrow operator (->) dereferences and extracts a structure field (member) with a single operator.
- The following are equivalent:

```
struct point *p;
 (*p).x = 7; // or p->x = 7;
printf("x is %d\n", (*p).x);
printf("x is %d\n", p->x);
```