CSI 333 – Lecture 11
Introduction to C: Part X
(Final Remarks on C)
Ref: Section 8.9 of Deitel & Deitel.

- Functions that manipulate and compare blocks of memory (just like strings).
- Useful for initializing, copying and comparing arrays.

**Function memset:**

```c
void * memset (void *s, int c, size_t n)
```

- Copies the character given by `c` into the first `n` bytes of `s`.
- Returns `s`.
**Function memcpy:**

```c
void * memcpy (void *s1, const void *s2, size_t n)
```

- Copies `n` bytes from `s2` to `s1`.
- Returns `s1`.

**Notes:**

- Function `memmove` is similar to `memcpy`, except that `memmove` is guaranteed to work correctly even if the two memory areas overlap. (Function `memmove` may not work correctly in that case.)

- Usually, `memcpy` is faster than `memmove`. So, if the two memory areas are known not to overlap, it is better to use `memcpy`.
**Function `memcmp`:**

```c
int memcmp (const void *s1, const void *s2, size_t n)
```

- Compares `n` bytes of `s1` and `s2`.
- Returns 0, negative or positive value depending upon whether in the (first) `n` bytes, `s1` is equal to, less than or greater than `s2`.

**Example:** See Handout 11.1.

**Dynamic Arrays Using `malloc`:**

- Dynamic arrays: Arrays whose size is known only at run time.
- Can be created and manipulated using `malloc`.

**Example:** See Handout 11.2.
Variants of `malloc`

**Ref:** Section 14.11 of Deitel & Deitel.

**Function `calloc`:**

```c
void * calloc (size_t nmemb, size_t size)
```

- Tries to allocate a number of bytes equal to `nmemb * size`.
- If allocation is successful, initializes all the bytes to zero and returns a pointer to the first initialized byte.
- Returns NULL if allocation is unsuccessful.
variants of malloc (continued)

**Function realloc:**

```c
void * realloc (void *oldptr, size_t newsize)
```

- Used to change the size of memory block pointed to by `oldptr`.

- `oldptr` must have been obtained by a previous call to `malloc`, `calloc` or `realloc`.

- If `oldptr` is `NULL` and `newsize` is greater than zero, then `realloc` works exactly like `malloc`.

- If `oldptr` is not `NULL` and `newsize` is zero, then `realloc` works exactly like `free`.
Function realloc (continued)

- If oldptr is not NULL and newsize is greater than zero, then realloc tries to allocate a new block of memory of size newsize bytes.

- If allocation is successful, a pointer to the new starting position is returned. (The newly allocated block may be somewhere else in memory.)

- The contents of the block are preserved up to the smaller of the old and new sizes.

- Returns NULL if allocation is unsuccessful. In this case, the value of oldptr remains unchanged.

Example: See Handout 11.3.
Function `sprintf` in `stdio.h`

```c
int sprintf (char *str, const char *fmt, arg1, ..., argn)
```

- Similar to `printf` (or `fprintf`) except that it writes the output into the string specified by `str`.
- The function automatically adds the ‘\0’ character.
- The string pointed to by `str` must have enough space to hold the output and the terminating null character; otherwise, the behavior of `sprintf` is unspecified.
- Useful in creating file names that depend on some values computed in a program.

**Example:** See Handout 11.4.
Global definitions/declarations

function_1 ( ... ) {
}

function_2 ( ... ) {
}

function_n ( ... ) {
Scopes of Identifiers

Example:

```c
#define SIZE 200
double x, y; int z[10];

int main() {
    int i; /* Local to main. */
    .
    .
} /* End of main. */

int compute(int r) {
    int j, k; /* Local to compute. */
    .
    .
    for (k = 0; k < r; k++) {
        .
        .
    }
} /* End of compute. */
```
Scopes of Identifiers (continued)

Notes:

- Variable i: Local to main; cannot be accessed from outside main.
- Variables j, k: Local to compute.
- Unlike Java (or C++) all variables in a C program must be declared before any executable statements.
- Variables i, j and k: Automatic variables.
  - Newly created every time the corresponding function is called.
  - Exist only as long as the corresponding functions are in execution.
  - Don’t retain their values between calls.
A Programming Error

```c
#define SIZE 100

char * get_name ( ... ) {

    char name_str[SIZE];
    .
    .
    return name_str;

} /* End of get_name. */
```

**Notes:**

- Since arrays are automatic variables, a function cannot return the name of a local array.
- If the type of `name_str` is `char *` and space is allocated to `name_str` using `malloc`, then the function can return `name_str`. 
The keyword static

- The keyword static has several interpretations in C.
- Its meaning depends on the context.

(a) **Static local variables:** *Retain* their values between calls.

**Example:**

```c
int main(void) {
    sample();
    sample();
}
```
Example (continued):

```c
void sample (void) {
    int x = 2; /* Automatic var. */
    static int i = 0;
    printf("x = %d\n", x++);
    printf("Call No. = %d\n", ++i);
}
```

Output:

```
x = 2
Call No. = 1
x = 2
Call No. = 2
```

Note: Function `strtok` (in string.h) is implemented using a static local variable.
(b) **Static global variables:** Visible only to functions within the file.

**Example:**

```c
static int val = 24;
int main(void) {
    .
    .
}
int compute( .... ) {
    .
    .
}
```

**Typical use:** Avoiding name conflicts across files.
(c) **Static Functions:** Visible only to functions within the file.

**Example:**

```c
static void sort (int x[], int n) {
    
    
    
}
```

**Typical use:** Similar to static global variables (i.e., avoiding name conflicts across files).
Register Variables

**Example:**

```c
register int x;
register float y;
```

**Notes:**

- Register attribute can be used only for variables which fit within a machine register (usually, one memory word).
- Use may speed up program since register access is faster than memory access.
- Compiler may ignore the register attribute.
- The address operator (&) can’t be used on a register variable.