CS 241 | Lecture Handout #4
January 27, 2016

From Monday's lecture:

```
int                                     int **result
// Sum an array of positive numbers, storing
// the result in `result` (by ref)
~~void~~ mysum(const int *ptr, int ~~*result~~) {
  assert(ptr != NULL);
 assert(result != NULL);
  *result = malloc(    sizeof(int)       );
  assert(*result != NULL);
  **result = 0;
  while ( *ptr ) {
   assert(*ptr > 0);

    ~~sum += *(ptr++);~~
    **result = *(ptr+
              +);
  }

        *result
  return ~~sum~~;
}
```

Puzzle #1: Create a custom string concatenation function

```
char *mystrcat(char *dest, char *src) {
    assert(dest != NULL);
     assert(src != NULL);
     while (*dest) { dest ++;}
char* n = dest;
   while (*dest)
  { *dest = *src;
              dest++;
              src++;
}          }
   *dest = '\0';
     return n;
```

Using `read()`:

```
ssize_t read(int fd, void *buf, size_t count);
```

...what type of call is read?

...how would we use it?

```
     int fd = open("", "");

   char * buf = malloc (…);
   read(fd, buf, …);
```

Using `scanf()`:

```
int scanf(const char * format, ...);
```

In `scanf`, the format string is the same as `printf` except that every type must be passed by reference to be written into by `scanf`:

Specifier:	d i	u o x	f	c s	p
Type:	int *	unsigned int *	float *	char *	void *

Return value?

number of format parameter filled

Example:

```
int num;   char c;
int result = scanf("%d %c", &num, &c);
printf("Values: %d %c\n", num, c);
printf("Return value: %d\n", result);
```

...what is the return value of the input: `7 hello`

return: 2, num: 7, c: h

...what is the return value of the input: `6` *(...followed by an EOF)*

return: 1, num: 6, c: unchanged

Using `getline()`:

```
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

The C-string passed by reference as `lineptr` will store the line; the size of the memory allocated in `lineptr` must be stored in `n` (to avoid overflow). Additionally:

> If `*lineptr` is set to `NULL` and `*n` is set 0 before the call, then `getline()` will allocate a buffer for storing the line. This buffer should be freed by the user program even if `getline()` failed.

...found in `man getline`

Example usage:

```
 1: char *s = NULL;   int n = 0;
 2: getline(&s, &n, stdin);
...  ...
 n: free(s);
```

Processes: *"I'm a nightmare dressed like a daydream"*
A process is the base computation container on Linux; multiple processes allow for multiple separate (and parallel) execution.

Q: System call to make a new process?

`fork();`

Environmental Variables
Process-specific dictionary that stores information about the execution environment:

- Command line: `env`
- C programming: `getenv(char *)`

Meta Example: *"Let is snow, let it snow!"*
`snowflake.c` attempts to create a snowstorm where every snowflake is a process *(found in /_shared/ in the CS 241 svn)*. Screen cursor logic is provided, simple API is:

- `int rows`: contains the number of rows of the terminal/console
- `int cols`: contains the number of columns of the terminal/console
- `gotoxy(x, y)`: moves cursor to a given **x**, **y** position

The key function, `snowflake()`:

```
void snowflake() {
  srand((unsigned)time(NULL));
  int col = rand() % cols;
  int row = 0;

  while (row < rows) {
    gotoxy(row, col);
    fprintf(stderr, "*");
    usleep(200000);
    gotoxy(row, col);
    fprintf(stderr, " ");
    row++;
  }
}
```

Fix #1:

Fix #2:

Fix #3:

Fix #4: