# C Programming

## Recitation 6

April 2017

- Outline
- Dynamic Allocation
  - Malloc
  - Calloc
  - Realloc
  - Free
- Strings and Pointers
  - Strings
  - String Arrays vs String Pointers
  - Processing Strings
- 2D Arrays
  - Using 2D Arrays
  - Passing 2D Arrays
- Task

# HANDLING MEMORY ON RUN TIME

- Allows us to use and control memory on run time.
- Requires standard library (`stdlib.h`).
- Done by four functions: `malloc`, `calloc`, `realloc`, `free`.

# Malloc

- Given as `void* malloc (size_t size);`
- Allocates a block of `size` bytes of memory **without initializing it**.
- Returns a **void pointer** to the beginning of the block. A void pointer can be casted to any type of pointer.
- Returns `NULL` on failure.

```
/* Allocate memory large enough to hold 10 integers */
/* Suggested way */
int *array1 = (int*) malloc(10 * sizeof(int));
/* Bad way */
int *array2 = (int*) malloc(10 * 4);
```

# CALLOC

- Given as `void* calloc (size_t num, size_t size);`
- Allocates a block of memory for an array of `num` elements, each of them `size` bytes long.
- **Initializes** all its bits to zero.
- Returns a **void pointer** to the beginning of the block.
- Returns `NULL` on failure.

```
/* Allocate memory large enough to hold 10 integers and ↩
initialize them to 0 */
int *array = (int*) calloc(10, sizeof(int));
```

# REALLOC

- Given as `void* realloc(void *ptr, size_t size);`
- Reallocates the area of memory given with pointer `ptr`. It must be previously allocated and not yet freed.
- **Does not** change the existing data.
- Either **expands/shrinks** the block or **allocates** new block and **copies** the existing data there.
- Returns a **void pointer** to the beginning of newly allocated memory.
- On failure, returns `NULL` but the old memory is not freed.

```
/* Allocate memory large enough to hold 10 integers*/
int *array = (int*) malloc(10 * sizeof(int));

/* Reallocate the array to hold 100 integers */
array = (int*) realloc(array, 100 * sizeof(int));
```

# FREE

- Given as `void free (void* ptr);`
- Deallocates an already allocated memory block pointed by `ptr`.
- **Does not** do anything if `ptr` is `NULL`.
- **Does not** change `ptr`.

```
/* Allocate memory large enough to hold 10 integers*/
int *array = (int*) malloc(10 * sizeof(int));

/* Free the array */
free(array);
/* Good practice to reset the pointer */
array = NULL;
```

# Strings

- C does not have a built-in *string* data type, but uses null-terminated arrays of characters.
- *String constants* are surrounded by double quotes ("), e.g. "Hello world!".
- To create a *string variable*, you must allocate sufficient space for the number of characters in the string and the null character ’\0’.

```
/* Static array initialization */
char butler[7] = {'A', 'l', 'f', 'r', 'e', 'd', '\0'};
/* Adds the null character */
char villain[10] = "Joker";
char *hero = "Harvey Dent";
/* Print the string */
printf("%s\n", villain);
/* Assign new string to char pointer */
hero = "Batman";
/* Gives incompatible types error */
villain = "Batman";
```

# STRING ARRAYS VS STRING POINTERS

```c
char lover[] = "Harley Quinn";
char *iterator;
char* first;
int i = 0;
int size;

/* Find the size of the string by iterating with index */
while(lover[i] != '\0') i++;
size = i;

/* Find the size of the string by iterating with pointer */
first = lover;
iterator = lover;
while(*iterator != '\0') iterator++;
size = iterator - first;
```

- Example: *dynamic_strings.c*.
- Example: *reverse_string.c*.

# PROCESSING STRINGS

- The functions are provided in the library: `string.h`.
- Example: *string_functions.c*

| Method | Description |
|---|---|
| `size_t strlen(const char *s);` | Calculates the length of the string `s`. |
| `char *strcpy(char *dest, const char *src);` | Copies the string `src` to `dest` (including '\0') and returns `dest`. |
| `char *strncpy(char *dest, const char *src, size_t n);` | Copies at most `n` characters. |
| `char *strcat(char *dest, const char *src);` | Concatenates the string `src` to the end of `dest`, placing '\0' at the end of `dest`, returns `dest`. |
| `int strcmp(const char *s1, const char *s2);` | Compares the string `s1` and `s2`, returns -1, 0, 1 accordingly. |
| `char *strchr(const char *s, int c);` | Returns a pointer to the first occurence of the character `c` in the string `s`. |

# Using 2D Arrays

```
int myMatrix[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

- Physically, all elements are stored in a single block of memory.
- Array elements are stored in row major order.
- Indexing;
    - `myMatrix`: pointer to the first element of the 2D array.
    - `myMatrix[0]`: pointer to the first row of the 2D array.
    - `myMatrix[1]`: pointer to the second row of the 2D array.
    - `*myMatrix[1]`: the element `myMatrix[1][0]`.
    - `myMatrix[i][j]` is the same as;

```
*(myMatrix[i] + j)
(*(myMatrix + i))[j]
*((*(myMatrix + i)) + j)
*(&myMatrix[0][0] + 4*i + j)
```

# Passing 2D Arrays

```
int myMatrix[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

- While passing a multi-dimensional array, the first array size does not have to be specified. The second (and any subsequent) dimensions must be given.
- Example: *matrix_allocate.c*

```
#define ROWS 3
#define COLS 5

int addMatrix(int list[][COLS]);

int main() {
    int a[][COLS] = { {13, 22, 9, 23, 12}, {17, 5, 24, 31, 55}, {4, 19, 29, 41, 61} };
    printf("Sum = %d\n", addMatrix(a));
}

int addMatrix( int t[][COLS] ) {
    int i, j, sum = 0;
    for (i = 0; i < ROWS; i++)
        for (j = 0; j < COLS; j++)
            sum += t[i][j];
    return sum;
}
```

# Lab Demo

- Let's use what we have learned today and complete simple task in Moodle.