CSE1142 – Command Line Arguments & File Processing & Dynamic Memory Allocation in C

Sanem Arslan Yılmaz

Some of the slides are from:
CMPE150 – Boğaziçi University
Deitel & Associates

Agenda

- Command Line Arguments
- File Processing
- Dynamic Memory Allocation
- Examples

Compiling C programs

- The most efficient way of developing a C program is to use a Unix environment:
  - Command-line
  - Compiler: gcc (GNU Compiler Collection)
    - It is free and available on many different platforms.
    - Command for compiling and running a C program on Unix:

```
$ gcc hello.c -o hello.out
$ ./hello.out
Hello, World!
```

  - To mimic a Unix programming environment on your own computer, you have to install cygwin on Windows.
    - **Dev-C++** automatically installs it.

Command Line Arguments

- Pass arguments to `main` on DOS or UNIX
  - Define main as

    ```
    int main( int argc, char *argv[] )
    ```

  - `int argc`
    - Number of arguments passed (including program name)
  - `char *argv[]`
    - Array of strings
    - Has names of arguments in order
      - `argv[ 0 ]` is first argument
  - Example:
    - `$ gcc myprog.c –o myprog.out`
    - `$ ./myprog.out input output`
      - `argc: 3`
      - `argv[ 0 ]: “./myprog.out "`
      - `argv[ 1 ]: "input"`
      - `argv[ 2 ]: "output"`

Files

- Storage of data in variables and arrays is temporary—such data is lost when a program terminates.
- Files are used for *permanent* retention of data.
- Computers store files on secondary storage devices, such as hard drives, CDs, DVDs and flash drives.

Input/Output Files and Text Streams

- text file
  - a named collection of characters saved in secondary storage
- input (output) stream
  - continuous stream of character codes representing textual input (or output) data
- stdin
  - system file pointer for keyboard's input stream
- stdout, stderr
  - system file pointers for screen's output stream

Files and Streams

- C views each file as a sequence of bytes
  - File ends with the *end-of-file marker*
    - Or, file ends at a specified byte
- Stream created when a file is opened
  - Provide communication channel between files and programs
  - Opening a file returns a pointer to a `FILE` structure (defined in `<stdio.h>`) that contains information used to process the file.
    - Example file pointers:
    - `stdin` - standard input (keyboard)
    - `stdout` - standard output (screen)
    - `stderr` - standard error (screen)

Creating a File

- `FILE *cfPtr;`
  - Creates a `FILE` pointer called `cfPtr`
- `cfPtr = fopen("clients.dat", "w");`
  - Function `fopen` returns a `FILE` pointer to file specifies
  - Takes two arguments – file to open and file open mode
  - If open fails, `NULL` returned

Accessing Files

- `feof ( FILE *fptr )`
  - Returns true if end-of-file indicator (no more data to process) is set for the specified file
  - The end-of-file indicator informs the program that there's no more data to be processed.

Operating system	Key combination
Linux/Mac OS X/UNIX	*<Ctrl> d*
Windows	*<Ctrl> z* then press *Enter*

End-of-file key combinations for various popular operating systems.

Accessing Files (cont.)

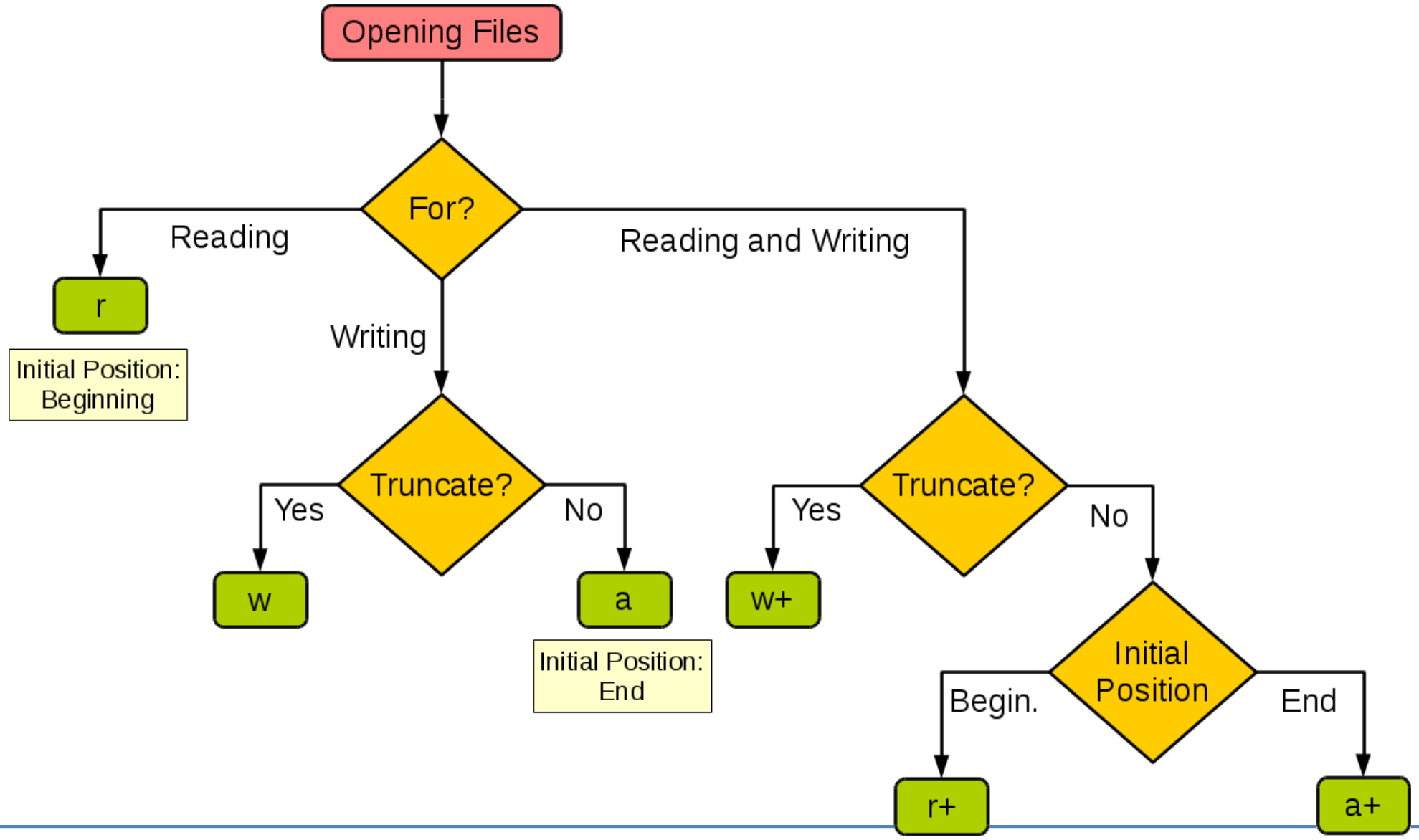
- `fclose( FILE *fpt )`
  - Closes specified file
  - Performed automatically when program ends
  - Good practice to close files explicitly
- Details
  - Programs may process no files, one file, or many files
  - Each file must have a unique name and should have its own pointer

File opening modes

Mode	Description
`r`	Open an existing file for reading.
`w`	Create a file for writing. If the file already exists, *discard* the current contents.
`a`	Open or create a file for writing at the end of the file—i.e., write operations *append* data to the file.
`r+`	Open an existing file for update (reading and writing).
`w+`	Create a file for reading and writing. If the file already exists, *discard* the current contents.
`a+`	Open or create a file for reading and updating; all writing is done at the end of the file—i.e., write operations *append* data to the file.

Fig. 11.5 | File opening modes. (Part 1 of 2.)

File opening modes (cont.)



*https://images.app.goo.gl/jFDjr8qjNAJa5duq7

Read/Write functions in standard library

- `int fgetc (FILE *fptr)`
  - Reads one character from a file
  - Takes a `FILE` pointer as an argument
  - `fgetc( stdin )` equivalent to `getchar()`
- `int fputc (int char, FILE *fptr)`
  - Writes one character to a file
  - Takes a `FILE` pointer and a character to write as an argument
  - `fputc( 'a', stdout )` equivalent to `putchar( 'a' )`
- `char *fgets (char *str, int n, FILE *fptr)`
  - Reads a line from a file
- `int fputs (const char *str, FILE *fptr)`
  - Writes a line to a file

Read/Write functions in standard library (cont.)

- `int fprintf (FILE *fptr, const char *format, ...)`
  - File processing equivalents of `printf`
  - `fprintf(stdout, "%d %s %.2f\n", account, name, balance);`

- `int fscanf (FILE *fptr, const char *format, ...)`
  - File processing equivalents of `scanf`
  - `fscanf(stdin, "%s %s %s %d", str1, str2, str3, &year);`

Comparison of I/O functions

Assume that 2 files are opened in the following way:

```
FILE *infilep, *outfilep;
infilep = fopen("data.txt", "r");
outfilep = fopen("out", "w");
```

TABLE 11.4 Comparison of I/O with Standard Files and I/O with User-Defined File Pointers

Line	Functions That Access stdin and stdout	Functions That Can Access Any Text File
1	`scanf("%d", &num);`	`fscanf(infilep, "%d", &num);`
2	`printf ("Number = %d\n", num);`	`fprintf(outfilep, "Number = %d\n", num);`
3	`ch = getchar();`	`ch = getc(infilep);`
4	`putchar(ch);`	`putc(ch, outfilep);`

Examples

- `fig11_01.c`
- `fig11_02.c`
- `fig11_06.c`

Example: fig11_02.c

```c
// Fig. 11.2: fig11_02.c
// Creating a sequential file
#include <stdio.h>

int main(void)
{
   FILE *cfPtr; // cfPtr = clients.txt file pointer

   // fopen opens file. Exit program if unable to create file
   if ((cfPtr = fopen("clients.txt", "w")) == NULL) {
      puts("File could not be opened");
   }
   else {
      puts("Enter the account, name, and balance.");
      puts("Enter EOF to end input.");
      printf("%s", "? ");

      unsigned int account; // account number
      char name[30]; // account name
      double balance; // account balance

      scanf("%d%29s%lf", &account, name, &balance);
```

Fig. 11.2 | Creating a sequential file. (Part 1 of 2.)

Example: fig11_02.c (cont.)

```
23
24          // write account, name and balance into file with fprintf
25          while (!feof(stdin)  ) {
26             fprintf(cfPtr, "%d %s %.2f\n", account, name, balance);
27             printf("%s", "? ");
28             scanf("%d%29s%lf", &account, name, &balance);
29          }
30
31          fclose(cfPtr); // fclose closes file
32       }
33    }
```

```
Enter the account, name, and balance.
Enter EOF to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z
```

Fig. 11.2 | Creating a sequential file. (Part 2 of 2.)

Example: fig11_06.c

```
1  // Fig. 11.6: fig11_06.c
2  // Reading and printing a sequential file
3  #include <stdio.h>
4
5  int main(void)
6  {
7     FILE *cfPtr; // cfPtr = clients.txt file pointer
8
9     // fopen opens file; exits program if file cannot be opened
10    if ((cfPtr = fopen("clients.txt", "r")) == NULL) {
11       puts("File could not be opened");
12    }
13    else { // read account, name and balance from file
14       unsigned int account; // account number
15       char name[30]; // account name
16       double balance; // account balance
17
18       printf("%-10s%-13s%s\n", "Account", "Name", "Balance");
19       fscanf(cfPtr, "%d%29s%lf", &account, name, &balance);
20
```

Fig. 11.6 | Reading and printing a sequential file. (Part 1 of 2.)

Example: fig11_06.c (cont.)

```
21          // while not end of file
22          while (!feof(cfPtr)  ) {
23             printf("%-10d%-13s%7.2f\n", account, name, balance);
24             fscanf(cfPtr, "%d%29s%lf", &account, name, &balance);
25          }
26
27          fclose(cfPtr); // fclose closes the file
28       }
29    }
```

```
Account    Name           Balance
100        Jones            24.98
200        Doe             345.67
300        White             0.00
400        Stone           -42.16
500        Rich            224.62
```

Fig. 11.6 | Reading and printing a sequential file. (Part 2 of 2.)

Dynamic Memory Allocation

- Dynamic memory allocation
  - Obtain and release memory during execution

- `malloc`
  - `void * malloc (int size)`
  - Takes number of bytes to allocate
    - Use `sizeof` to determine the size of an object
  - Returns pointer of type `void *`
    - A `void *` pointer may be assigned to any pointer
    - If no memory available, returns `NULL`
  - Example

```
newPtr = malloc( sizeof( struct node ) );
```

Dynamic Memory Allocation (cont.)

- Example:

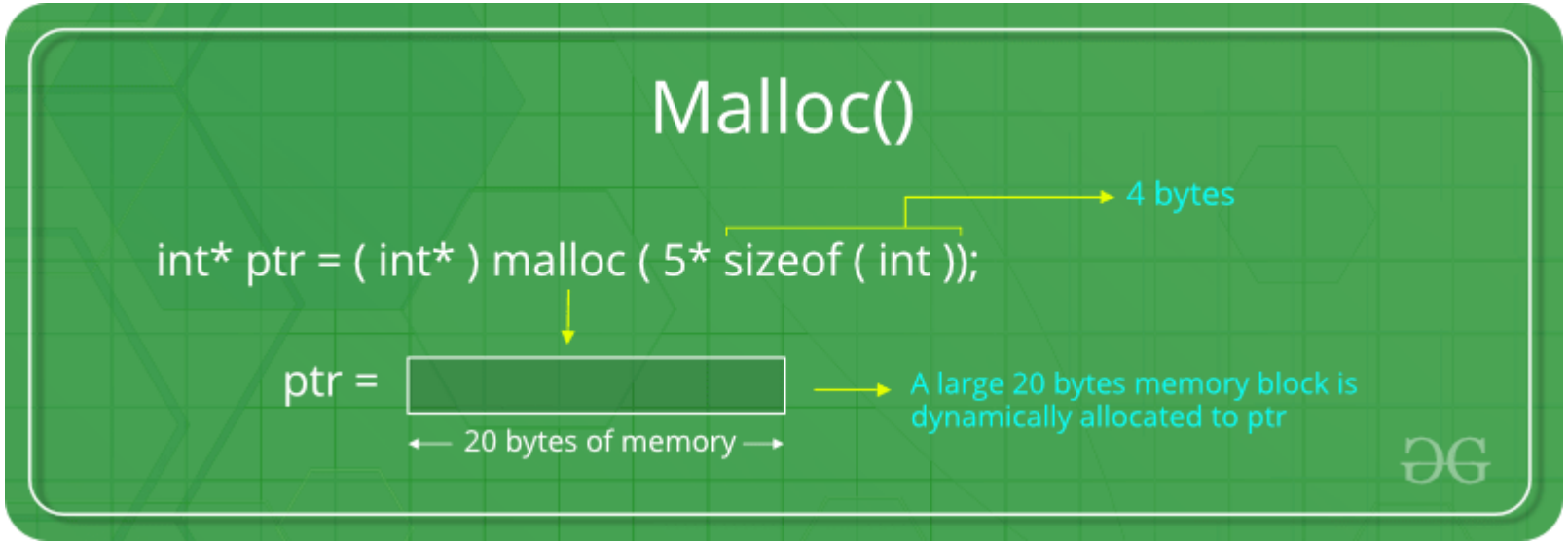
```
int *nump;
char *letp;
A_t *Ap;

nump = (int *)malloc(sizeof (int));
letp = (char *)malloc(sizeof (char));
Ap = (A_t *)malloc(sizeof (A_t));


*nump = 307;
*letp = 'Q';
 Ap->m = 5;
 Ap->n = 10;
```

Example - 1

```
#include <stdio.h>
#include <stdlib.h>
int main() {
        int* ptr;
        int n, i;

        printf("Enter number of elements: \n");
        scanf("%d", &n);

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

        if (ptr == NULL) {
                printf("Memory not allocated.\n");
                exit(0);
        }
        else {
                printf("Memory successfully allocated using malloc.\n");
                for (i = 0; i < n; ++i) {
                    ptr[i] = i + 1;
                }
                printf("The elements of the array are: ");
                  for (i = 0; i < n; ++i) {
                    printf("%d, ", ptr[i]);
                }
        }
      return 0;
}
```

Output:
Enter number of elements:
5
Memory successfully allocated using malloc.
The elements of the array are: 1, 2, 3, 4, 5,

malloc()


Malloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
A large 20 bytes memory block is dynamically allocated to ptr
20 bytes of memory

Dynamic Memory Allocation

- `free`
  - `void  free (void *ptr)`
  - Deallocates memory allocated by `malloc`
  - Helps to reduce wastage of memory by freeing it.
  - Takes a pointer as an argument
  - `free (newPtr);`

Dynamic Memory Allocation - Arrays

- Dynamic memory allocation
  - Can create dynamic arrays

- `calloc`
  - `void * calloc (int nmembers, int size)`
  - *nmembers* – number of elements
  - *size* – size of each element
  - Returns a pointer to a dynamic array

- All malloc/calloc/free functions found in <stdlib.h> header file.

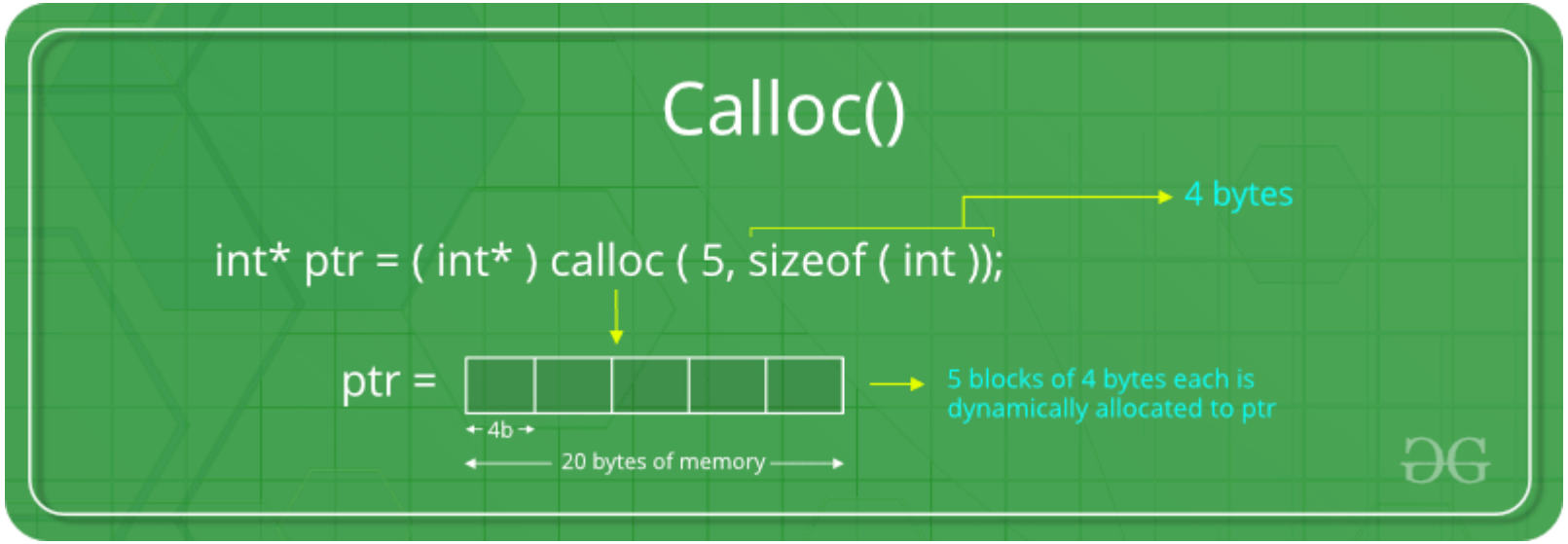
Example - 2

```
#include <stdio.h>
#include <stdlib.h>
int main() {
        int* ptr;
        int n, i;

        printf("Enter number of elements: \n");
        scanf("%d", &n);

        ptr = (int*)calloc(n, sizeof(int));

        if (ptr == NULL) {
                printf("Memory not allocated.\n");
                exit(0);
        }
        else {
                printf("Memory successfully allocated using calloc.\n");
                for (i = 0; i < n; ++i) {
                    ptr[i] = i + 1;
                }
                printf("The elements of the array are: ");
                  for (i = 0; i < n; ++i) {
                    printf("%d, ", ptr[i]);
                }
        }
      return 0;
}
```

Output:

Enter number of elements:
5
Memory successfully allocated using calloc.
The elements of the array are: 1, 2, 3, 4, 5,

calloc ()


Calloc()
4 bytes
int* ptr = (int*) calloc (5, sizeof (int));
ptr =
5 blocks of 4 bytes each is dynamically allocated to ptr
4b
20 bytes of memory

malloc and calloc Return Value

- malloc and calloc both return the address of the newly allocated block of memory
  - However, they are not *guaranteed* to succeed!
    - Maybe there is no more memory available
  - If they fail, they return NULL
  - You should always check for NULL when using malloc or calloc.
  - Ex:

```
int *arr = (int *)malloc(10 * sizeof(int));
if (arr == NULL)
        printf("Out of Memory! ");
```

malloc() vs. calloc()

- malloc and calloc both allocate memory
- calloc has slightly different syntax
- Most importantly:
  - calloc() zeros out allocated memory,
  - malloc() does not.

Example

```
#include<stdlib.h>
int *foo(int n){
        int i[10];
        int *j;
        j = (int *) malloc (n*sizeof(int));
        // alternatively
        // j = (int *) calloc (n, sizeof(int));
        return j;
} // i's memory deallocated here; j's not

void bar(){
        int *arr = foo(10);
        arr[0] = 10;
        arr[1] = 20;
        // do something with arr
        free(arr); //deallocate memory
}
```

Examples

- structExample.c
- unionExample.c