# Agenda

1. Homework review
2. Arrays in C
   - Strings as array type
   - Addressing elements of the array
3. Memory
   - Memory types
   - Stack and heap
   - Globals and constants
4. Pointers
   - Typed pointers
   - Untyped pointers and casting
   - volatile

# Home work review

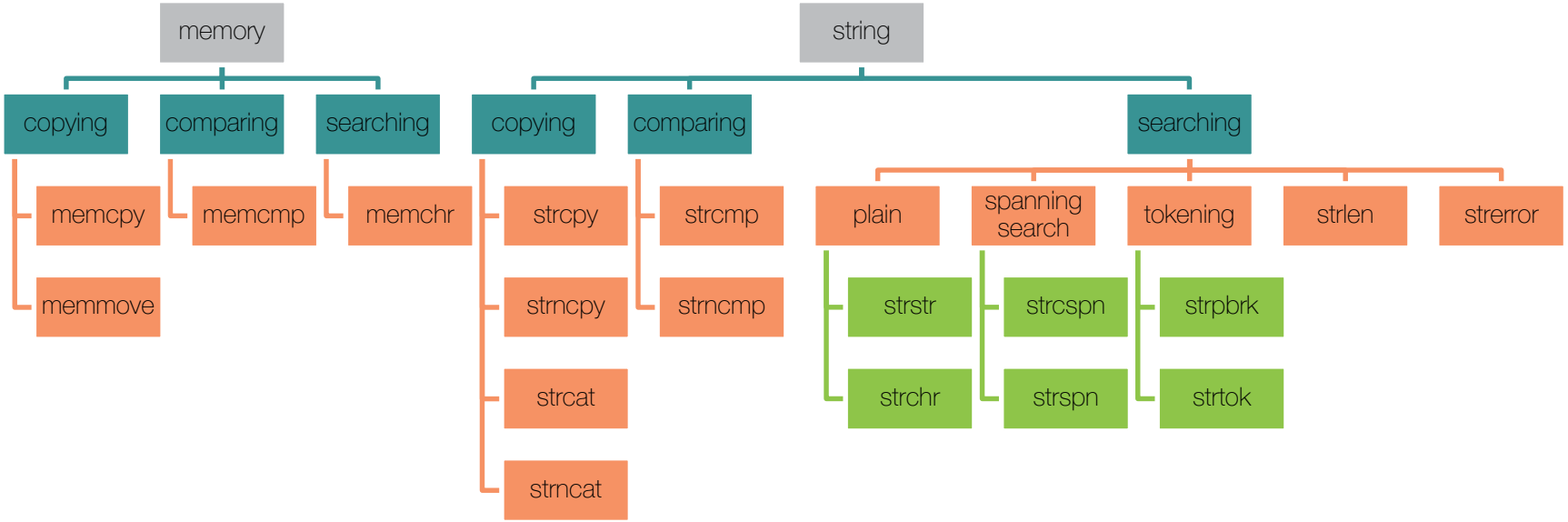
# Advanced data types in C

# Strings in C

- You already used strings in C with printf and scanf
- Strings are char arrays with terminating \0
- Strings elements are addressed with s[index] notation
- char is a usually one-byte signed type, Unicode support is done via wchar.
  - It's forbidden to mix wchar and char operations in one application
- **C++**: C doesn't support overloading of operators, therefore [] is not an operator, it's plain pointer arithmetics (see below)

```
const char s[6] = “abcd”;
const char s2[] = “another_string”;
const char s3[] = { ‘a’, ‘b’, ‘c’, ‘d’ };

sizeof(“template”) == 9;
sizeof(s3) == 4; /*< Not null terminated */

const char s3[sizeof(“template”)];

strcmp(s, s2);
strcmp(s, s3); /*< Bug here */

strncmp(s, s3, sizeof(s) > sizeof(s3) ?
    sizeof(s3): sizeof(s));
```

# <string.h>


memory
string
copying
comparing
searching
copying
comparing
searching
memcpy
memcmp
memchr
strcpy
strcmp
plain
spanning search
tokening
strlen
strerror
memmove
strncpy
strncmp
strstr
strcspn
strpbrk
strcat
strchr
strspn
strtok
strncat

# Arrays in C

- As strings are arrays, you already encountered array definition.
  - Initialization is different, though
- There is no native list/tree/map in C, however kernel, for example, provides a framework for them
- Arrays are linear, there are no holes in them
  - Size of array is a constant, they can't be resized
  - Array size can defined during variable definition, when using const value of the upper scope
- sizeof returns amount of chars needed for the whole array, not amount of elements

```
#define GLOBAL_BUF_SIZE_MACRO 4
const size_t GLOBAL_BUF_SIZE = 3;

int int_array[5];
int int_array2[5] = { 2, 3, 4 };
int int_array[GLOBAL_BUF_SIZE]; /*< error */
int int_array[GLOBAL_BUF_SIZE_MACRO]; /*< ok */

void f(const size_t BUF_SIZE)
{
    const size_t ANOTHER_BUF_SIZE = 3;
    int buf[BUF_SIZE];
    int buf2[ANOTHER_BUF_SIZE]; /*< error */
    int int_array[GLOBAL_BUF_SIZE]; /*< ok */
    //some code
}
```

# memory functions from arrays in C

- Always initialize arrays in some way
- memset is extremely efficient, but still more CPU consuming, than static initialization
  - This is affordable price for more precise bug reproducibility
- memcpy used for memory copying, it has same syntax as strncpy: dst, src, n
- memcpy can't be used for intercepting memory regions
  - use memmove instead
- memcmp is like stcmp, but doesn't stop on \0
- memchr is the same as strchr
- memmem is non-standard extension, don't use it

```
/* Not initialized memory */
int global_array[5];

/* Initialized from global scope constant */
int int_array2[5] = { 0 };

/* Initialized dynamically */
size_t f(void) {
    int int_array3[5];

    memset(int_array3, 0, sizeof(int_array3));

    return sizeof(global_array) /
        sizeof(global_array[0]);
}
```

# enum

- Enumerations are untyped constants lists
  - In fact they have sizeof(int)
- Enumeration allow to encapsulate lists of constants to a symbolic names
  - They are very efficient at magic numbers list specification
  - Code with enums looks much cleaner and easily understandable
- Enumerations works perfectly with switch-case
- **J**: enums are not classes, they don't have methods, instead you can assign them values in the text

```
enum my_enumeration
{
     MY_ENUM_CONSTANT1, /* default: 0*/
     MY_ENUM_CONSTANT2, /* MY_ENUM_CONSTANT1+1 */
     MY_ENUM_CONSTANT3, /* MY_ENUM_CONSTANT2+1 */
     MY_ENUM_CONSTANT4 = 7,
};

void f(enum my_enumeration e) {
     switch (e) {
          case MY_ENUM_CONSTANT1:
               //action
               break;

          case MY_ENUM_CONSTANT2:
               //action
               break;

          default:
               break;
     }
}
```

# More examples on arrays

- Enums works perfectly with arrays during initialization
    - Common pattern for map organization
- **Py**: arrays are typed in C
- **C++**: C allows more specific type to initialize array (check example)
- **J**: As in Java, you can initialize arrays statically

```
enum ieee80211_ac_numbers {
    IEEE80211_AC_VO      = 0,
    IEEE80211_AC_VI      = 1,
    IEEE80211_AC_BE      = 2,
    IEEE80211_AC_BK      = 3,
};
enum ieee802_1_d_traffic_class {
    IEEE802_1_TC_BACKGROUND       = 0,
    IEEE802_1_TC_BACKGROUND_HI    = 1,
    IEEE802_1_TC_BEST_EFFORT      = 2,
    IEEE802_1_TC_EXCELLENT_EFFORT = 3,
    IEEE802_1_TC_CONTROLLED_LOAD  = 4,
    IEEE802_1_TC_VIDEO            = 5,
    IEEE802_1_TC_VOICE            = 6,
    IEEE802_1_TC_NETWORK_CONTROL  = 7,
};

const int map_802_1d_tc_to_802_11_ac[] = {
    [IEEE802_1_TC_BACKGROUND]       = IEEE80211_AC_BE,
    [IEEE802_1_TC_BACKGROUND_HI]    = IEEE80211_AC_BK,
    [IEEE802_1_TC_BEST_EFFORT]      = IEEE80211_AC_BK,
    [IEEE802_1_TC_EXCELLENT_EFFORT] = IEEE80211_AC_BE,
    [IEEE802_1_TC_CONTROLLED_LOAD]  = IEEE80211_AC_VI,
    [IEEE802_1_TC_VIDEO]            = IEEE80211_AC_VI,
    [IEEE802_1_TC_VOICE]            = IEEE80211_AC_VO,
    [IEEE802_1_TC_NETWORK_CONTROL]  = IEEE80211_AC_VO
};
```

# struct

- Structures is a complex type, containing lots of fields of different types
  - C++: struct is not a class, it can't have method
  - C++: struct can be initialized more efficiently in C
- struct can contain other structures
- structure fields are aligned, so it can be accessed faster
  - Some architecture doesn't support struct fields access if they are not aligned
  - There are two specific compiler attribute in order to align fields to another boundary, or pack the structure (align to 1 boundary)
  - default fields alignment is multiplier of their size
  - sizeof for struct returns sum of sizeof for its fields and padding in betwen

```
struct mystruct {
    int i;
    char c;
} __attribute__((__packed__));

struct complex_struct {
    int j;
    struct mystruct s;
};

/* Explicit initialization, C++ style */
struct mystruct s2 = { 2, 'a' };

/* Implicit initialization, good style */
struct mystruct s2 = {
    .i = 2,
    .c = 'a'
};
```

# union

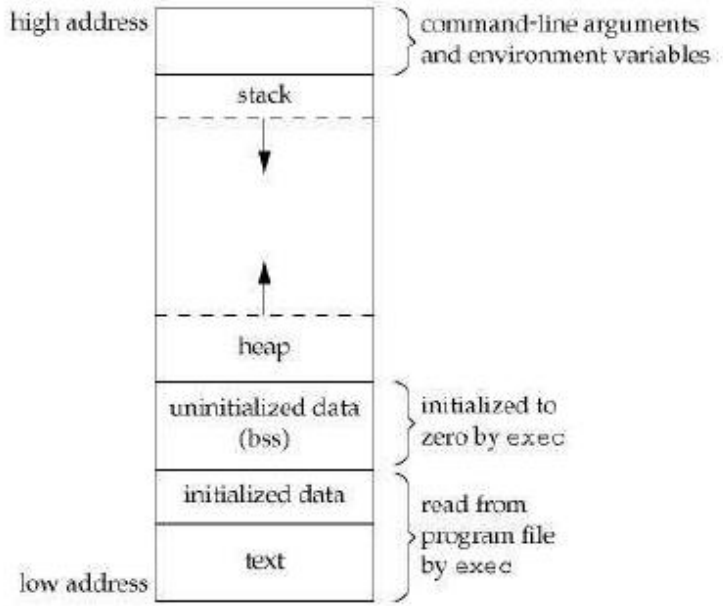
- Union is another complex data structure
  - **J, Py**: there are no unions in Java and Python
- All of the fields in union are compressed in the same memory region, and therefore they are mixed
- unions are used mainly for memory saving
- C99/C11 allows usage of anonymous unions
  - which is very convenient, as they are transparent for use

```
struct request {
    unsigned request_code;
    unsigned arg;
};

struct response {
    int error_code;
};

struct request_response_buffer {
    uint8_t type;
    union {
        struct request req;
        struct response resp;
    };
} __attribute__((__packed__));
```

# Memory and pointers

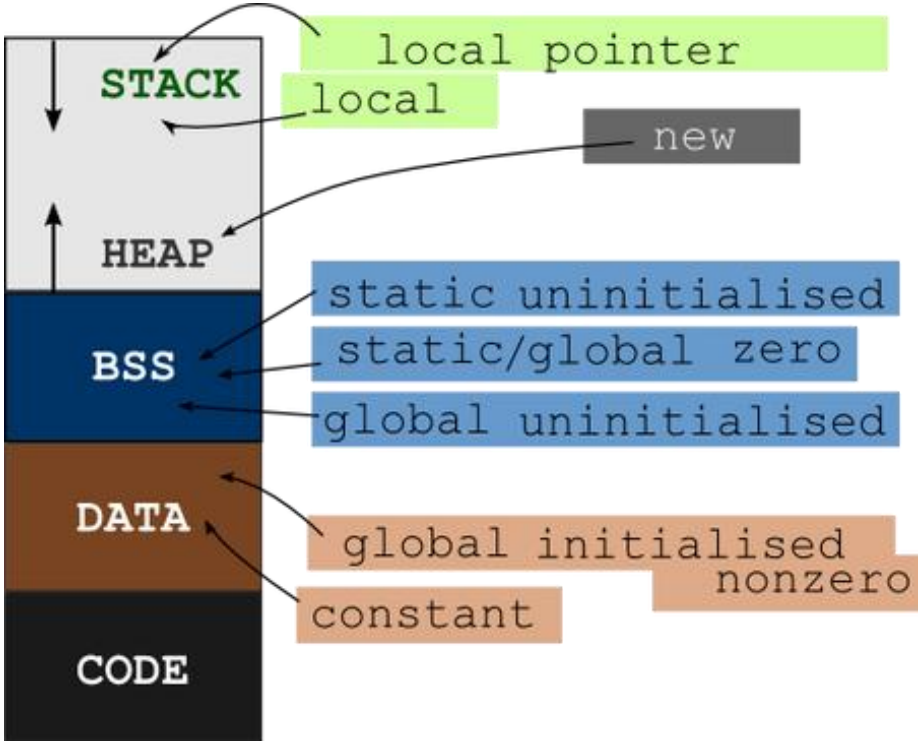
# Memory types and their structure

- Here we are talking only about virtual memory
- There are several memory types:
  - stack – for temporary variables, function calls, etc.
  - heap – for dynamically allocated data
  - static data (.bss and .data) – for global/static duration variables
  - .text – for the binary code
- Memory regions are allocated at binary start
  - For firmwares they are preallocated during linking
  - Kernel modules shares segments between different modules
    - it's oversimplification



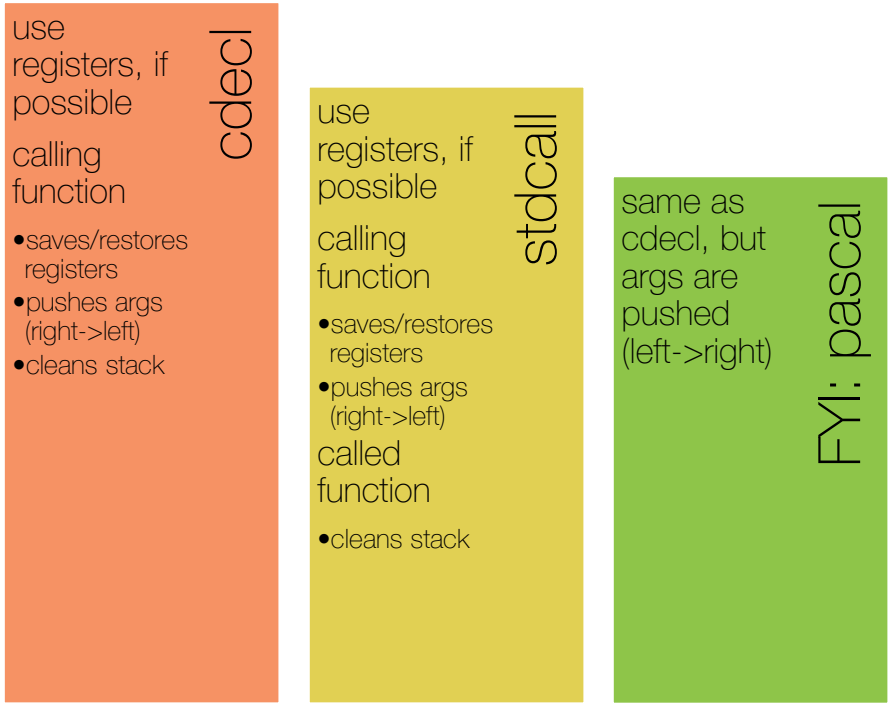
# Global scope memory and .text

- Variables located in the global scope are allocated before user space binary starts
- .data/.bss is used in the same way by the whole application
  - multithreaded application can produce race conditions, this memory should be treated carefully
- Breaking segment protections causes segmentation fault
  - .data/.bss is normally execution-protected, br
  - .text segment is normally write-protected, writing to this segment normally produced segmentation fault
  - writing/reading to non-allocated segments produces segmentation faults
  - memory outside of the allocated segments is also read/write-protected



# Stack

- Stack grows down
- All of temporary variables are allocated on the stack
- Stack is automatically cleaned after function return
  - Wrong arguments number leads to either excessive allocation in stack (stack leak), …
  - or to stack corruption
- Stack is relatively small
  - ~1Mb to 8Mb
  - Kernel have only 4K stack
  - Recursion aggressively consumes stack (each function return address + arguments + temporary variables consume stack)
  - ulimit –s regulate stack size



# Pointers

- Pointer is pretty unsuccessful name
  - Pointer is kind of pointing to somewhere, but in fact it's not connected to the actual type or value of the variable it's pointing to
  - **J:** pointers are renamed to references in Java, but "reference by value" sounds worse, than even pointer
  - **C++:** references are the pointer without memory arithmetic
  - **Py:** welcome to hell
- Pointer is a variable containing another memory region address
- Pointer exist in order to share memory
- Pointers have fixed size (it's memory address), therefore they are very efficient during arguments passing

```
int i = 3;
int *pointer_to_i = &i;
int *uninitialized_pointer;
int *pointer_to_nowhere = NULL;

int* another_syntax_for_pointer_definition;

/* You can modify pointer, but not the memory it
points to */
const int *pointer = &i;

/* You can modify memory, it points to, but not the
pointer */
int *const pointer = &i;
```

# More examples

- You can change the pointer value, pointed memory is not modified
- You can also access the pointed memory and perform some operations on it
- Referencing and dereferencing is very cheap operation
- &variable produces pointer of variable type to the memory, where the variable is stored
- *pointer accesses memory the pointer is referencing

```
int i = 3;
int *pointer_to_i = &i; /* referencing variable */

int f(void)
{
    int j = 5;

     /* assignment of new valu to the pointer */
    pointer_to_i = &j;

    /* dereferencing pointer,
       accessing pointed memory */
    *pointer_to_i = 3;

    return 2 + *pointer_to_i;
}
```

# Pointer arithmetics

- You can change the pointer value, pointed memory is not modified
- +/- operations on pointer are performed by the pointed type size
- Pointer doesn't know about memory accessibility
  - Pointer arithmetic is very efficient, but it's very unsafe
  - **C++, Java**: that's why these languages are encapsulating of the pointer arithmetic into references
- It's hardly possible to work at low level without pointer arithmetic

```
int array[] = { 3, 2, 5 };

/* array name is a pointer, i.e. 0x8010034a0 */
int *pointer_to_i = array;

/* referencing to the second element of array */
int *pointer_to_i = &array[1]; /* 0x8010034a4 */

printf("Value is %d, %p\n", *pointer_to_i,
    pointer_to_i);

pointer_to_i++;
/* Now it points to array[2] */
/* 0x8010034a8 */

printf("Value is %d, %p\n", *pointer_to_i,
    pointer_to_i);

printf("Value of array: %p\n", array);
```

# Pointers casting

- Pointers keep information about the type it's pointing to
- Assignment of the pointers of different types bring warning
- In order to perform such assignment, you need an implicit cast
  - Be careful
  - Implicit cast looks ugly intentionally
    - **C++**: FYI Stroustroup intentionally insisted on ugliness of static_cast<>(), etc.
- Pointers to custom types are not compatible to the pointers of the base type, even if the custom type is directly defined from it (see example)

```
/* type definition: source type => name */
typedef int myint;

int i = 5;
myint j = 3;

int *pi = &i;
myint *pj = &j;

pi = pj; /* Warning */
```

# void *

- Untyped pointer is an important construction
  - Though it's pretty unsafe
- Every pointer cast can be implicitly casted to void*
- void * is used for encapsulation of the data type passed
  - It's a base of concept oriented paradigm, also known as generic or templates.
- void* also demonstrates, that function is not really interested in the addressed memory
  - it's either a brilliant design, see memmove, …
  - or extremely bad (more common case)
- Think twice about void* usage
- void* behaves as char* during pointer arithmetic

```
int i = 5;
double d = 2.0e7;

/* Warning: initialization int* to double* */
int *pi = &d;

/* Just a pointer, no info on the addressed data */
void *untyped_pointer = &i;

/* No warning, unsafe operation */
untyped_pointer = &d;

pi = &i;

/* No warning, very unsafe */
untyped_pointer = pi;
/* No warning */
pi = (int *)&d;
```

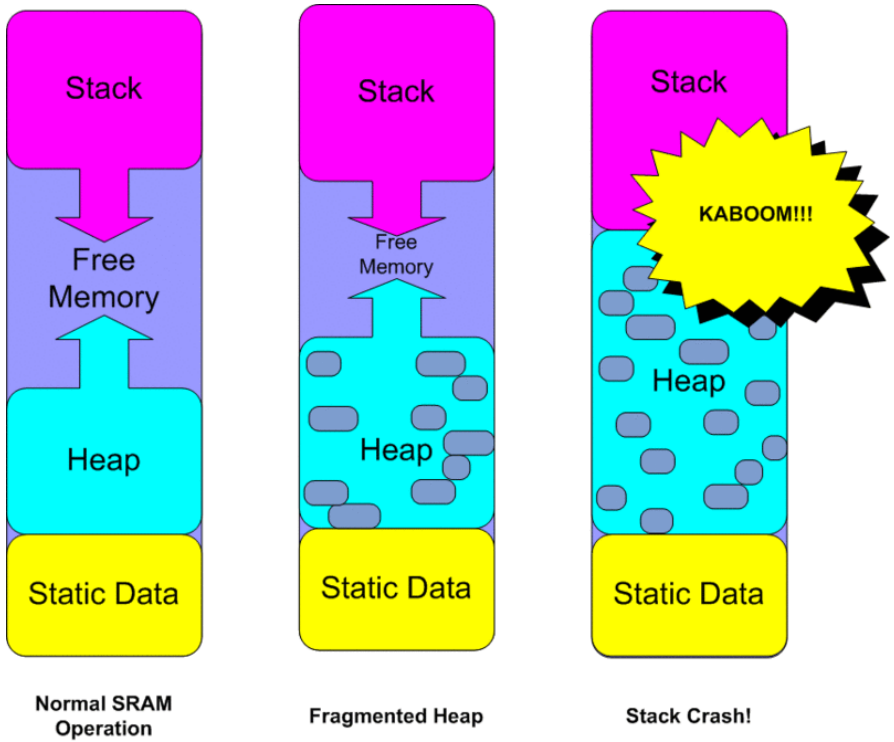
# Pointers to functions

- **Pointer to function is a key element of polymorphism in C/OOP and a base of concept-programming in C**
- Function name is a pointer as well
  - You can take pointers from the functions
  - You can pass functions as arguments with pointers
- Dereferncing function doesn't actually dereferencing, it's just assignment
  - Don't do double dereferencing, it make no sense
- There are two conventions on such function arguments calling:
  - Call function via pointer implicitly *myfunc(arg1, arg2);
  - Call function via pointer explicitly myfunc(arg1, arg2);

```
void qsort(void *base, size_t nmemb, size_t size,
     int (*compar)(const void *, const void *));

/* Defining function pointer type */
typedef int (*comp)(const void *, const void *);

/* You can write like this, but it's K&R style */
/* Don't use it in definitions */
comp my_compare_function_name
{
}

int array[] = {2, 3};
qsort(array, sizeof(array), my_compare_function_name);

/* Same as above */
qsort(array, sizeof(array), &my_compare_function_name);
```

# Heap

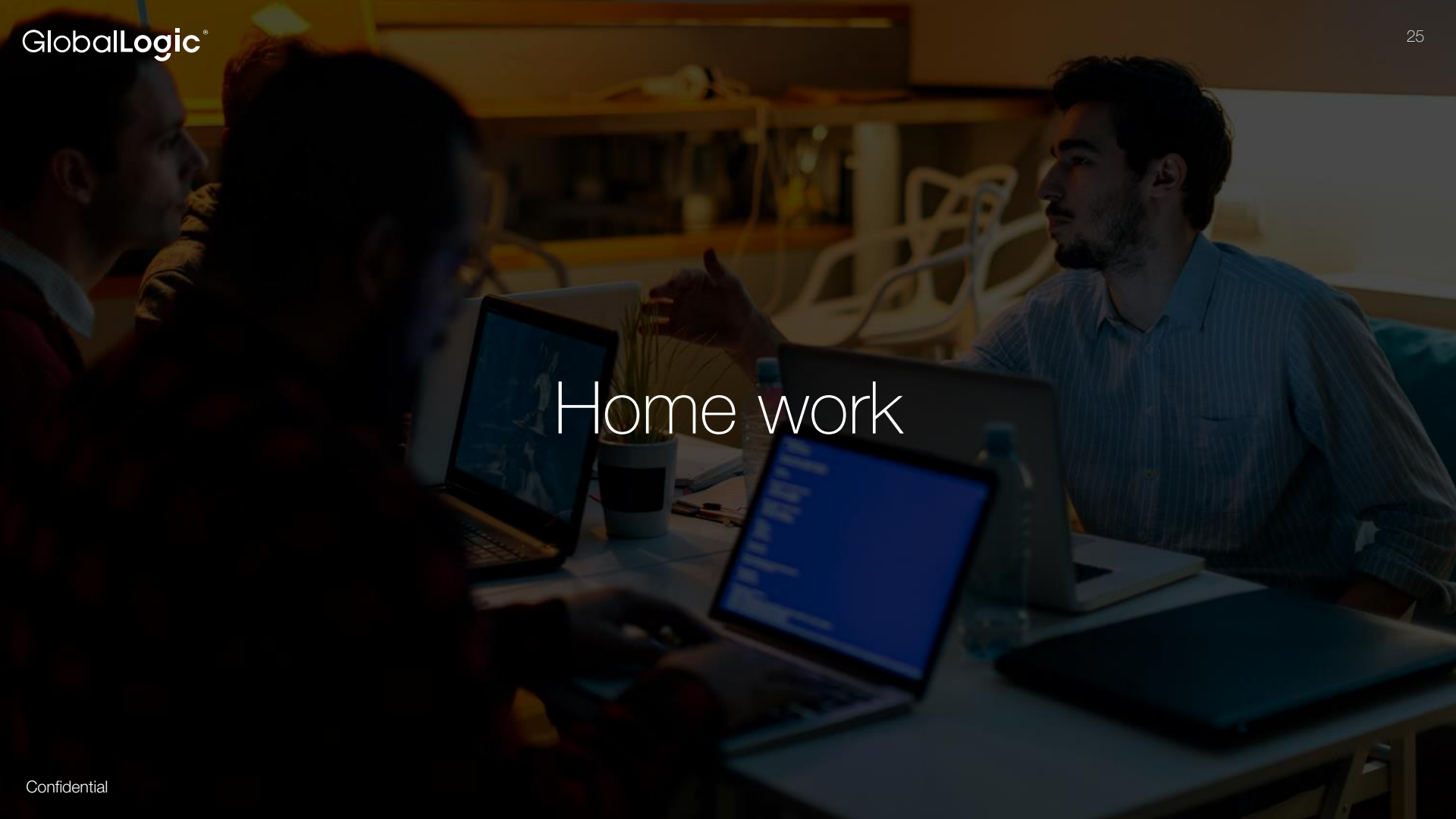
- Heap is a dynamically used memory
- You can allocate memory regions in heap
  - Don't forget to deallocate this memory memory
- Heap is pretty big, but inaccurate use of it may lead to massive memory leaks
  - And heap fragmentation, showed to the right
- Any significant data structures should be located in the heap (images, files)
  - avoid using of the heap, if you can, stack fits most of the temporary memory needs



# Heap allocation

- Most of the errors in C are connected to memory allocation and pointer arithmetics
- Basic function for memory allocation: malloc
  - Use calloc where applicable
- Free memory with free
  - Double free brings segmentation error, be carefull
  - Not freeing memory causes memory leak
- **Don’t work with dynamic memory, if it’s not needed**
  - **If it’s needed, don’t hesitate**
- You can use alloca for allocations on stack (no need to free this memory), however
  - its usage is usually sign of improper function splitting

```
const int N = 5;

int *dynamic_array =
    (int *)malloc(sizeof(int) * N);

uint8_t *dynamic_uint8_t_array =
    (uint8_t *)calloc(sizeof(uint8_t), N);


if (dynamic_array == NULL)
    /* always check */

free(dynamic_array);
```



# Home work

# Quiz

Answer each of the following:

a) A pointer variable contains as its value the _____ of another variable.

b) The three values that can be used to initialize a pointer are _____, _____ and _____.

c) The only integer that can be assigned to a pointer is _____.

State whether the following are *true* or *false*. If the answer is *false*, explain why.

a) The address operator (&) can be applied only to constants, to expressions and to variables declared with the storage-class *register*.

b) A pointer that's declared to be void can be dereferenced.

c) Pointers of different types may not be assigned to one another without a cast operation.

Do each of the following:

a) Write the function header for a function called *exchange()* that takes two pointers to unsigned integer numbers x and y as parameters and does not return a value.

b) Write the function prototype for the function in part (a).

c) Write the function header for a function called evaluate() that returns an integer and that takes as parameters integer

X and a pointer to function poly(). Function poly() takes an integer parameter and returns an integer.

d) Write the function prototype for the function in part (c).

Find the error in each of the following program segments. Assume

```
int *zPtr;
/* zPtr will reference array z */
int *aPtr = NULL;
void *sPtr = NULL;
int number, i;
int z[5] = { 1, 2, 3, 4, 5 };
sPtr = z;
```

a) `++zptr;`

b)
```
/* use pointer to get first value of array; assume zPtr is initialized */
number = zPtr;
```

c)
```
/* assign array element 2 (the value 3) to number; assume zPtr is  initialized */
number = *zPtr[ 2 ];
```

d)
```
/* print entire array z; assume zPtr is initialized */
for (i = 0; i <= 5; ++i) {
printf("%d ", zPtr[ i ] );
}
```

e)
```
/* assign the value pointed to by sPtr to number */
number = *sPtr;
```

f) `++z;`

# Home task

- Define variable pc of pointer to unsigned char type and assign (with typecast) address of memset() function from <string.h> header file. Try to write value 0x90 to address stored in pc. What is happening? Why?
- Declare pointer to the void *memset(void *s, int c, size_t n) function with typedef. Define global variable memset_s of such type with volatile qualifier and assign address of memset() function from <string.h> to it.
  Call memset() via memset_s pointer in main() to fill memset_s pointer with zeros (i.e. this simulates memset_s = NULL statement). Print memset_s value after filling to stdout with printf() function.

- Implement following functions, similar to corresponding C standard library functions:

```
size_t _strlen(const char *s);
size_t _strnlen(const char *s, size_t maxlen);

char *_strchr(const char *s, int c);
char *_strrchr(const char *s, int c)
```

  First two takes pointer to '\0' terminated string in s and return number of symbols in string excluding '\0', _strnlen() also scans no more than maxlen bytes in s.

  Last two search given symbol c in string s and return pointer to it and NULL when no c is found in entire s, _strrchr() starts search from right.
- Implement function size_t _strlcpy(char *d, const char *s, size_t dsize) which copies no more than dsize - 1 characters from string s to d and ensures that string d always terminated with '\0'. Function returns length of the s.
- Implement functions char *_strtolower(char *s, size_t len) and char *_strtoupper(char *s, size_t len), which converts all upper/lower case chars in string to lower/upper case.

# Clean code is important

C doesn't restricts you on the way you put spaces and the way you names the identifiers.

If the code is supposed to be read at least twice, you have to write it clean, especially if the code will be shared