This course has already ended.

Fifth round tasks as a `ZIP file.`

Usually in the software projects there are situations where we are at the programming stage and cannot know how much memory the program needs in advance. In this case, we need a **dynamic memory** where size can be defined during the execution of the program. Dynamic memory is allocated by operating system upon request and generally it will always be free. This section gives introduction to computer memory organization, dynamic memory management and **valgrind** (tool is used to check memory leaks in the program).

Note: In this section, the tasks use valgrind tool to check memory leaks. If valgrind memory leaks or warnings are present in the code, only half of the total task point is given. If both valgrind and warnings exist, then you will get only quarter of the total task score. Valgrind information is provided later in this section.

Memory organization¶

The modern computer systems typically use virtual memory, that provides each process a dedicated memory space: each process sees only its own virtual memory space, and cannot access the memory of other processes. Even though each process sees a full **64-bit** or **32-bit** address space, typically only small portion of it is in use. If program tries to use an address from the virtual memory that has not been allocated for it, " `segmentation fault` " follows and the program terminates immediately. The virtual memory is divided into different segments, as shown below:

../../_images/virt-memory.jpg

The program code is loaded from an ELF (Executable and Linking Format) formatted executable file into read-only code segment. The program code is produced as a result of compiling and linking process. When the program is started, the operating system starts executing the program from the code segment. Unlike Java, Python or some other languages, there is not interpreter software between the system and the executable.

Constant strings are also located in the read-only area of the memory space, and initialized when program is loaded. For example, when declaring `char *name = "Jaska";` , string “Jaska” is allocated from read-only memory (remember different ways of initializing a string). Trying to modify such strings would raise segmentation fault during program execution.

In addition to the code segment, memory space is allocated for initialized global data and uninitialized global data. Global data is for the variables that are declared outside the program functions, and can be accessed from anywhere in the program, or declared with the static qualifier inside or outside the functions. The size of these segments are determined in advance, and cannot be changed during program execution.

Heap is dynamically allocated memory: its usage varies based on the program execution. Program can allocate more memory from heap during its execution, and is expected to release the memory after it is not needed anymore. We have not yet used heap in our exercises and examples, but will see how heap is used in the next section.

Usage of the stack also varies during program execution, but it is allocated automatically by the computer system. Each time the execution enters a function, some space from stack is used: the return address to which the execution should return from the function is stored in stack. The function parameters are placed in stack. The space for local variables declared inside a function are also allocated from the stack. When function exits, the space it used from stack is released automatically. Therefore it is an error to refer to such memory from outside the function, even though use of pointers would technically allow this. The examples and exercises in section 1 and 2 have mainly used stack for the variables.

Here is an example which explains the above mentioned memory organization concept.

```c
#include <stdio.h>
#include <stdlib.h>

/* This variable is stored in the "initialized global data segment" */
int global_var1 = 10;

/* This variable is stored in the "uninitialized global data segment" */
int global_var2;

/* This variable is stored in the "uninitialized global data segment" */
static global_static_var1;

int main(void)
{
    /* This variable is stored in the "initialized global data segment" */
    static global_static_var2 = 500;

    /* This local variable is stored in "stack" and it is uninitialized */
    double local_var;

    /* This local variable is stored in "stack" and it is initialized */
    int local_var2 = 10;

    /* This memory is allocated from "heap" */
    char *dyn_ptr = malloc(10);

    /* Free heap memory */
    free(dyn_ptr);

    return 0;
}
```

There is a useful video (approx. 17 min) in YouTube about stack and heap dynamics with a C program. It is recommended that you check it out at some point.