# Master M1 MOSIG, Grenoble Universities
# TD 1: Memory Management 1

Arnaud Legrand, Benjamin Negrevergne

2010

## I. Stack and Heap

The *stack* and the *heap* are two distinct memory segments defined by the system for each process so that they can store their data.

- The stack is used to *automatically* allocate memory for the variables defined within functions. Their size is known at compile time and for such variables, memory is automatically reserved when the program enter the functions and released when the program leaves the function. It is thus used for temporary storage of information and the use of this information is restricted to the function where they are defined.
- The heap is used to store data that will need to be visible across function calls or whose size is unknown at compile time. Memory management (allocation and deallocation) is the responsibility of the user.

Read the following code and try to figure out in which segment (stack or heap) each variable is allocated:

```
#include <stdio.h>

int min(int a, int b, int c){
  int tmp_min;
  tmp_min = a <= b ? a : b;
  tmp_min = tmp_min <= c ? tmp_min : c;
  return tmp_min;
}

int main(int argc, char *argv[]){
  int min_val = min(3, 7, 5);
  printf("The min is: %d\n", min_val);
  exit(0);
}
```

**Question I.1:** *In which memory segment are the variables* `a`, `b` *and* `c` *allocated ? When is the memory allocated to them released ? What about the* `tmp_min` *variable ?*

Let us now consider the following code:

```
int vect_sum(int *v1, int *v2, int size){
  int *r, i;
  r = malloc(sizeof(int) * size);
  for(i = 0; i < size; i++){
    r[i] = v1[i] + v2[i];
  }
  return r;
}

int main(){
  int v1[] = {1, 2, 4, 7};
  int v2[] = {3, 4, 9, 2};

  int *p_result = vect_sum(v1, v2, 4);
  print_vect(result, size); /* prints the content of the given vector
  exit(0);
}
```

**Question I.2:** *What value is contained by the* `r` *variable after the call to* `malloc()` *inside the* `vect_sum()` *function ? In which memory segment is this value stored ?*

**Question I.3:** *What is the exact meaning of the affectation:* `r[i] = v1[i] + v2[i];` *? This affectation results in a write instruction. In this program, in which memory segment does this write happen ?*

**Question I.4:** *Write a similar program that behaves in the same way without using* `malloc()`*. You may need to change the parameters of the* `sum_vect()` *function.*

**Question I.5:** *What is the life cycle of a stack-allocated variable ? of a heap-allocated variable ?*

## II. Illegal memory accesses

Correct memory allocation is required so that each variable lies at distinct places in the memory space. Pointers are very useful, but they also enable to access memory addresses that have not been allocated. If a program tries to read or write at a such address, it may be killed *by the system* with the `SEGFAULT` signal. A memory access that may raise a `SEGFAULT` signal is called an *illegal memory access*.

**Question II.1:** *Which one of theses lines are illegal memory accesses ? Which one would raise a warning using a "picky" compiler ? (One declared variable is available for the following lines).*

1. `int *pa = 2;`
2. `*pa = 34;`
3. `int b = 4, *pb = &b;`
4. `*pb = 5;`
5. `int *pc;`
6. `printf("pc is equal to %d\n", pc);`
7. `printf("*pc is equal to %d\n", *pc);`
8. `pc = malloc(sizeof(int));`
9. `*pc = -2;`
10. `pa = pc;`
11. `free(pa);`
12. `pc = -4;`

## III. Recursive functions

```
int power(int a, int n){
  if( n != 0 )
    return power(a * a, n - 1);
  else
    return 1;
}
```

**Question III.1:** *Make a rough estimation of the memory needed to compute* `power(2, 3)`*. Where is this memory allocated ?*

**Question III.2:** *Write and run a small program to validate your estimation. If your estimation was wrong, figure out why.*

## IV. Valgrind

`valgrind` is a tool used to track runtime errors. It simulates the execution of a given executable inside a virtual system, and records any illegal access to the memory as well as other errors.

Retreive and extract the archive at `mandelbrot: negreveb/ens/2010/mosig/ex.tgz`. The programs given in this archive are all syntactically correct C programs, but they all misbehave at run time.

**Question IV.1:** *Use* `valgrind` *to find and solve errors in the given* `C` *files.* `valgrind` *is usually very verbose, write down the* `valgrind` *errors that helped you and explain their exact meaning. (If programs are compiled with the* `-g` *option valgrind is able to provide the source file name and line where the error happened). You may also find the following options useful :*

- *–db-attach=yes (to launch gdb as soon as there is a problem);*
- *–leak-check=yes (to get information about memory that was never freed and are definitely lost);*
- *–show-reachable=yes (to get information about memory that was never freed and are still reachable).*