Lab 4
Data Layout

Objective

- Understand the link between data and the stack.
- Understand how functions calls deal with the stack.
- Implement the stack inside DIGMIPS programs.

First, get the archive for the lab on `http://laure.gonnord.org/pro/teaching/capM1.html`.

4.1 Functions

EXERCISE #1 ▶ **Procedure calls**
Consider the following C program:

'simple0.c'

```c
#include <stdio.h>
void g()
{
  printf("%s","g");
}

void f()
{
  printf("%s","f");
  g();
}

void main()
{
  printf("%s","m");
  f();
  g();
  printf("%s","s");
  return;
}
```

The file `simple0_mask.asm` contains the skeleton of this program in the DIGMIPS format. Complete it, using the convention that register `r6` contains the first address of the stack and test it with the help of DIGEVAL.

EXERCISE #2 ▶ **Function calls, parameters, return**
Considering the following C program:

'simple1.c'

```c
#include <stdio.h>

int f(int n) {
  int m;
  m = n+1;
  return m;
}
```

```c
void main(){
  printf("%d\n",f(7));
}
```

1. Use the semantic rules of the course to give the traduction of $f : [\![f]\!]$ (compile by hand.)
2. Minimise the number of registers (4 should be enough, including r_7 and r_6).
3. In the prelude part, justify the fact that we do not need any slot for temporaries (local variables, intermediate results).
4. Complete `simple1_mask.asm`, test with DIGEVAL (First do it on paper!).

4.2 Data Structures

EXERCISE #3 ▶ **Arrays**
Consider the following program:

'arrays.c'

```c
void copy(int tab_dest[2], int tab_source[2], int n)
{
  tab_dest[n] = tab_source[n];
}

void main()
{
  int tab1[2]; // C code, our input would be   int[2] tab1
  int tab2[2];
  tab1[1] = 2;
  copy(tab2,tab1,1); //tab2[1] <-- tab1[1]
}
```

- How will you store the array in assembly language?
- Write the corresponding `asm` file and test it with DIGEVAL.

EXERCISE #4 ▶ **Structs**
Consider the following program:

'points.c'

```c
#include <math.h>

typedef struct { int x; int y; } point_t;
typedef struct {point_t A ; point_t B;} segment_t;

extern int sqr(int);

int len(segment_t s){
  return sqrt(sqr(s.A.x - s.B.x) + sqr(s.A.y - s.B.y));
}

int main(){
  segment_t s;
  s.A.x = 2;  s.A.y = 3;
  s.B.x = 4;  s.B.y = 5;
}
```

- How will the struct be stored ?
- (optional) Write the corresponding `asm` file and test it with DIGEVAL.

EXERCISE #5 ▶ **Lists - optional**
Consider the following program:

'struct.c'

```
typedef struct { int re; int im; } complex_t;
typedef struct { complex_t item; list_t* next; } list_t;

list_t* build(int n)
{
  list_t list;

  if(n==0)
    return NULL;
  else
    {
      list.next = build(n-1);
      list.item.re = n;
      list.item.im = n;
      return &list;
    }
}
```

- How will you store the list in assembly language?
- Write the corresponding `asm` file and test it with DIGEVAL.

EXERCISE #6 ▶ **Data: allocation in** DIGCC
Open `Type.cc` and study the `allocate()` function. This function will be called during the code production for functions:

'parser.ypp-cut'

```
function:
  type TK_ID TK_LPAR
    {
      add_function($2,$1);
      current_return_type = $1; //type checking
      reset_registers(); reset_variables();
    }
  declare_args TK_RPAR
  TK_LACC declare_local_vars
    {
      label($2);
      label_end_function = string($2);
      label_end_function += "_end";
      prelude();  // <----- HERE
    }
  stmt_list TK_RACC
    {
      label(label_end_function);
      postlude();
    }
;
```

'SymbolTable.cc-cut'

```
void prelude(){
  //Set ARP
```

```
  push_arp();
  mov_arp_sp();
  ...
  //for each local
  Type* variable_type = get_variable_type(local_variable);
  if(!(variable_type->is_base()))
  {
    int var_register = variable_type->allocate();
...
```

4.3 Cultural section: execution library

The file `simple1_compiled.asm` contains the assembly code produced by our compiler, DIGCC. The `.asm` file generated by DIGCC is composed of three parts:

1. **Initialization code (lines 1 – 47).** Initialization of the stack (line 1), initialization of the heap (a call to the `__init_heap()` function, line 16), and finally, a call to the `main()` function (line 38).
2. **Program (lines 48 – 114).** Contains the functions `f()` (lines 48 – 68) and `main()` (lines 69 – 114).
3. **Execution library (lines 115 – 397).** It's a rudimentary "operating system" layer:
   - `void __init_heap()` (lines 115 – 158). Initializes the heap. Always used inside the initialization code.
   - `void* malloc(int size)` (lines 159 – 222). Allocates `size` bytes inside the heap, and returns the memory address of the first byte (thus, a pointer).
   - `void free(void* data, int size)` (lines 223 – 265). Free a heap area of `size` bytes that begins at `data`. Do not use with wrong size!
   - `void print_char(char c)` (lines 266 – 288). Prints the character `c` on screen.
   - `void print_int(int n)` (lines 289 – 319). Prints the integer `n` on screen. We suppose $0 \leq n \leq 19$.
   - `void print_string(char* s)` (lines 320 – 350). Prints the character string `s` (ex. "bonjour"). Use with moderation.
   - `void print_newline()` (lines 351 – 372). Go to next line.
   - `char input_char()` (lines 373 – 397). Reads a keyboard character.

   This is obviously only the bare minimum. We could add arithmetic functions ($*$, /, $\sqrt{}$, sin, cos, tan, etc), drawing functions, etc.

   The execution library has itself been compiled (once and for all) from `runtime.c`.

EXERCISE #7 ▶ **Execution library - Optional exercise**

1. Inspect the code of the functions of the execution library (.c). In particular, the ones for handling the heap (`__init_heap`, `malloc`, `free`). Compares with what has been shown during the lecture.
2. How is *fragmentation* handled by `free()`?