# Compilation

## TP 4: Data layout

S. Filip & C. Alias

In the previous TP we have seen how to analyze the syntax of a program (TP 1 and 2), and how to check data types (TP 3). In this TP, we are interested by the actual code synthesis. The first step is to determine where and how to store data structures.

### Exercise 1. *Activation Records*

Lets study the following program:

```
void f(int[3] tab_dest, int[2] tab_source, int n)
{
  tab_dest[n] = tab_source[n];
}

void main()
{
  int[3] tab1;
  int[2] tab2;
  tab2[1] = 2;
  f(tab1,tab2,1);
}
```

**Question.** Run this program by showing side by side the evolution of the context and the stack.

### Exercise 2. *Functions (by hand)*

Lets get back to programs seen in the lecture:

```
void g()
{
  printf('g');
}

void f()
{
  printf('f');
  g();
}

void main()
{
  printf('m');
  f();
  g();
}
```

```
void f(int n)
{
  int m;
  m = n;
}

void main()
{
  f(0);
}
```

**Manip.**

- The file `simple0_mask.asm` contains the skeleton of the program of the *left hand side*. **Complete it** then **try it on** DigMIPS.
- The file `simple1_mask.asm` contains the skeleton of the program of the *right hand side*. **Complete it** then **try it on** DigMIPS.
- **Modify** `simple1_mask.asm` such that `f` returns `m`, which is then printed inside `main`.

## Exercise 3. *Functions (with* DigCC*)*

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:

**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).

**Program (lines 48 – 114).** Contains the functions `f()` (lines 48 – 68) and `main()` (lines 69 – 114).

**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`.

**Manip.**

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

## Exercise 4. *Data layout*

The following function (that does nothing) is considered:

```
void foo()
{
  struct { int[3] source; int[3] dest;} segment;
}
```

**Questions.**

- **Draw** its activation record.
- **Express, with details,** $[\![$ `struct { int[3] source; int[3] dest;}` $]\!]^t$.