Université de Lille 1 **Année Universitaire 2011-2012**

Master Calcul Scientifique
Programmation in C

1 Preliminary: some hints about Emacs

To open a file `toto.txt`, type `$ emacs toto.txt`. If this file does not exist, that will open a new window with the given name. If emacs recognize the type of the file (and the corresponding mod is installed), then the text will be colored, and automatic indentations will be available (with the `TAB` key). For instance, a file beginning by `#! /bin/sh` will be recognized as a script. Same for a file with the extension `.sh`. A file with the extension `.c` will load the emacs C mode...
Remember that one can use a & to run it in the background, or the combination `CTRL+Z` then `bg` afterwards.
Some shortcuts (here, `^` stands for the `CTRL` key, and `M-` - called meta - stands for either the `ALT` or `ESC` key):

Running emacs

- `^x^c` quit emacs,
- `^x^f` opens a new file,
- `^x k` kills an opened file,
- `^x^v` opens a new file and close the current one,
- `^x^s` saves a new file,
- `^x b` moves to another buffer,
- `^x leftarrow` moves to the next left buffer,
- `^x rightarrow` moves to the next right buffer.

Moving into a file

- `M-<` moves to the beginning of the document,
- `M->` moves to the end of the document,
- `^s` searches forward,
- `^r` searches backward,
- `^l` centers the screen on the current line,
- `^leftarrow` moves the cursor to the beginning of the current/previous block,
- `^rightarrow` moves the cursor to the end of the current/next block,
- `^l` centers the screen on the current line,
- `M-x goto-line l` moves to the l-th line of the file.

Editing

- `TAB` indents current line,
- `^k` deletes the current line, starting from the cursor position,
- `^_` cancels the last operation,
- `M-%` search and replace,
- `^space` starts selecting a region from the cursor position,
- `M-w` copy,
- `^w` cut,
- `^y` paste,

Screen splitting

- `^0` Remove the current window from the screens,
- `^1` Make active window the only screen,
- `^2` Split screen horizontally,
- `^3` Split screen vertically,
- `^x o` Move to next screen.

Running commands

- `^g` cancels the current operation (to move back to the window),
- `M-$` check spelling of the word at cursor or the selected area,
- `M-!` run a shell command,
- `M-x shell` starts a shell within emacs,
- `M-x compile` starts the compiler for the active window,
- `M-x space` gives available commands,
- `^h` emacs helps,
- `^h t` runs the emacs tutorial,

2 What we do not want to do

Here are some examples coming from the International Obfuscated C Code Contest (`http://www.ioccc.org`):

Example 1.
```
m(char*s,char*t) {
return *t-42?*s?63==*t|*s==*t&&m(s+1,t+1):
         !*t:m(s,t+1)||*s&&m(s+1,t);
}
main(int c,char **v) { return!m(v[1],v[2]); }
```

Example 2.
```
#include <stdio.h>
int O,o,i;char*I="";
main(l){O&=l&1?*I:~*I,*I++||(l=2*getchar(),i+=O>8
?o:O?0:o+1,o=O>9,O=-1,I="t8B~pq`",l>0)?main(l/2):
printf("%d\n",--i);}
```
Example 3.
```
#define n ((e[++a]-42)/9-f[d+1])?
#define o printf("%c",c[" 01./:;|\\"]);

char e[]="**3<HRZcir+3@OXakt;=GOXds*\?HRZcir*7HNZ`i19JS\\p*H[m1:CJSz*>H[`mr25\
\?Hx,P,B2Gs-KTfzRdv1SeyCR-ISeu.<Ev+9+P,z,4PfzIdvO2*HRZcir6GPis=MU*3HRZcir*HZi\
1JS\\epy*>H[m1JSey*DH[m*3<HZiu-@P*3HZi<N]q1JS\\epy:[m1CJSeny06I[m*4\?HRZcir,\
\?*6HRZcir1J]q2K*H[m2K*H[m2@K]qtO@M2DK]q,]q1JS\\epy[m1:JSey+[m*3<HRZcir13Gt,\
=GVs*3<HRZcir1J]qz*HF*AH2;DK]qua0=G2:K]q]q1CJS\\pDVu1:JS*D!+3:BIOSY`egilqtxz\
\177.0249<==>EJMUY`ejov#$59@CJOXYZbfhlnrxy&+.57=@IMR[``bcfmnq!#),@",*f;

int main(int j,char *k[]) {
 int a,b,c,d,g,h,i=19;
 printf("       ");
 for(g=0[f=(char *)calloc(80+(h=atoi(1[k])),1)]=1; g<=h; g++) {
   if ((g>30)&&(f[i-2]+f[i-1]!=0)) i++;
   for(d=c=0; d<i; d++) {
   o f[d]=(e[b=c*9,b+=(c=d[f]),(((a=e[345+b]+b/19*85-33-b/40*12-b/80*4)[e]
                -42)/9-f[d+1])?n n n n n n n n 0:a:a:a:a:a:a:a:a:a]-42)%9; }
   o if (0[f]-1) printf("\n%6i ",g); else printf("\n       "); }
}$
```
If you understand them, this is good. But if you code like that, this is not.
The objective is to create understanble codes. Therefore, the objective is to separate the code in several files if needed, to declare the variable in header files (`.h`), and to use a *Makefile* for the compilation. Moreover, one should not hesitate to comment its code so that it is readable.

3 What's C ?

3.1 ANSI C: a very small and simple language

- 32 key words:

char	**double**	**enum**	**float**	**int**	**long**	**short**	**signed**
struct	**union**	**unsigned**	**void**	**const**	**volatile**	**break**	**case**
continue	**default**	**do**	**else**	**for**	**goto**	**if**	**switch**
while	**auto**	**register**	**static**	**extern**	**typedef**	**return**	**sizeof**

- 40 operators:

( )	[ ]	->	!	˜
++	−−	-	(type)	*()
&()	sizeof	,	∗	/
%	+	-	>>	<<
>	<	<=	>=	==
<<=	&	ˆ	|	&&
||	?:	+=	-=	*=
/=	%=	ˆ=	!=	>>=

That's all !

3.2 A low-level programming language

C language

- enables to manage data at a processor level,
- does not manage memory,
- does not have any instruction to manage character chains, structures…
- does not have any Input/Output function

To manage these points, we need libraries.

3.3 A first example

```
/* This is a comment */
#include <stdio.h>
int
main
(int argc, char **argv)
{
 printf("Hello world\n") ;
 return 0 ; /* i.e. EXIT_SUCCESS */
}
```

- `include` is a directive for the preprocessor to use what is needed for the function `printf` of the standard library;
- Instructions end by a semi-colon;
- The function `main` is needed to produce an executable (that will first run `main`). It is defined by the header of the function: return type, name, argument; the braces contain the instructions of the function;
- The function `printf` is declared in `stdio.h`.

3.4 Warning

In C, the coder is suppose to know what he is doing. Error test and type checking are only made during the compilation phase; this means that nothing of the sort is tested during the execution of the program (are we using an index that is bigger than the size of a table, is the conversion of type used ok…). Moreover, the compiler is lazy: if something has a tiny sense, it will be allowed; therefore, a program can run correctly for some data and give an error for other ones.

3.5 Principle of compilation

1. **Editing the source file:** text file containing the program ; requires a text editor (vi, emacs…).
2. **Preprocessing phase:** the source file is processed by a *preprocessor* that makes only purely text transformations (replacing character chains, including other source file….
3. **Compilation:** The file generated by the preprocessor is converted in *assembly language*, i.e. a sequence of instructions associated to the functionality of the microprocessor (make an addition…).
4. **Assembler:** transforms the assembly language in an *object* file understanble for the processor.

5. **Link edition:** In order to use function libraries already written, a program is separated in several source files. Once the source code is assembled, one must *link* the object files together. this creates an *executable*.

Usually, we suffix the files in the following way:

- `.c` for the source files,
- `.i` for the files created by the preprocessing phase,
- `.s` for the assembly language files,
- `.o` for the object files,
- `.a` for the object files that correspond to pre-compiled libraries,

The `UNIX` C compiler is called `cc`; we will rather use the one from the GNU project, `gcc`. A basic use is:

`$ gcc [options] file.c [-l`*libraries*`]`

If no option gives a name for it, the output will be an executable file `a.out`. If one gives libraries (let say `-llibrary`), they will be searched (precisely here, the file `liblibrary.a`) in the directory containing the precompiled libraries - usually `/usr/lib`. For instance, if we want to link the program with th mathematic library, we specify the option `-lm`, that will use the corresponding `libm.a` object file.
The main options of the compiler are the following:

- `-c` does not make the link edition ; this produces an object file,
- `-E` makes only the preprocessing phase,
- `-g` will give the symbolic informations used by the debugger,
- `-Idirectory` specifies a directory where to find the needed header files (the current directory is always included by default),
- `-Ldirectory` specifies the directory where to find libraries,
- `-o filename` gives the name of the file to produce (default is `a.out`),
- `-O, -O1, -O2, -O3` gives the level of optimization required ; without such option, the aim of the compiler is to reduce the compilation cost. With this option, will try to reduce the size and the executive time of the produced file. The number specifies the level of optimization,
- `-S` runs only the preprocessor and the compiler; therefore, this produces an assembly language file,
- `-v` prints the executed commands during the compilation,
- `-W` prints more warnings,
- `-Wall` prints all warnings.

4 More details

4.1 Operation priorities

16	`()` `[]` `->` `.`	L
15	`++` `--` (postfix)	R
14	`!` `~` `++` `--` (prefix) `-` (unary) `(type)`	R
	`*` (indirection) `&` (address) `sizeof`	R
13	`*` (multiplication) `/` `%`	L
12	`+` `-`	L
11	`<<` `>>`	L
10	`<` `<=` `>` `>=`	L
9	`==` `!=`	L
8	`&` (bitwise and)	L
7	`^`	L
6	`|`	L
5	`&&`	L
4	`||`	L
3	`?:`	R
2	`=` `+=` `-=` `*=` `/=` `%=` `>>=` `<<=` `&=` `^=` `|=`	R
1	`,`	L

Exercise Assume that we have:

$$A = 20 \quad B = 5 \quad C = -10 \quad D = 2 \quad X = 12 \quad Y = 15$$

Evaluate the valid expressions in the following expressions:

$$\begin{array}{llll}
5 * X + 2 * 3 * B/4 & A == B = 3 & A+ = X + 2 & A! = C* = -D \\
A\% = D + + & A\% = + + D & (X + +) * A + C & A - A == B - B \\
!(X - D + C)||D & A\&\&B||!0 & C = 12 - - & (1 << (2 << 1)) == ((1 << 2) << 1)
\end{array}$$

If an expression is not valid, add some brackets so that it becomes valid.

4.2 Identifiers

These are memory containing data (variables...) or code to execute (function). One should choose “goods” names for it. There are some mandatory rules:

- an identifier must not begin with an integer,
- it must not contain an operator,
- it cannot be a key word,
- small and capital letters are different,

There are also some conventions one should follow:

- non ASCII characters should not be used (portability),
- identifiers beginning by a _ are supposed to be only for the OS programs
- identifiers should be easily understandable ; usually, one use short names for local variable, more complete one for global variables (`surname_table` for instance), and capital letters are kept for constant values.

Example 4.

- *5, foo-1, 3knight, foo. are not correct,*
- *_foo_bar, tête, ___A_, _ are correct but should not be used,*
- *a is perfectly fine (usually for a local variable).*

4.3 Control structures

Conditional instruction. The syntax is the following:

```
if (expression)
  instruction1;
```

or

```
if (expression)
  instruction1;
else
  instruction2;
```

This evaluates the `expression`. If the value is different from 0, `instruction1` is executed; otherwise, it is `instruction2` (if it exists).

Multiple choice instruction. The syntax is the following:

```
switch (expression)
  case cte1:
    instruction1;
    break;
  case cte2:
    instruction1;
    break;
      .
      .
  case cteN:
    instructionN;
    break;
  default:
    instruction
```

This evaluates `expression`, compares it with `cte1`, `cte2`.... If `expression` is equal to `cteK`, `instructionK` is executed; the `break` instruction stops the execution of the switch, so that the next constants are not checked (this is a common but optional instruction). If none of the constant is equal to `expression` then `instruction` is executed (if it exists ; the `default` part is optional).

Remark: the constants must be different !

Condition-controlled loop There exist three kinds of iterative instructions:

- The instruction `while`:

  ```
  while (expression)
    instruction;
  ```

- One can also use the `do while` instruction, in the following way:

  ```
  do
    instruction;
  while (expression);
  ```

  Here `instruction` is executed at least once.

- Finally, there is the `for` loop:

  ```
  for (expression1; expression2; expression3)
    instruction;
  ```

  which is equivalent to

  ```
  expression1;
  while (expression2)
    {
      instruction;
      expression3;
    }
  ```

Controlling loops iterations The `break` and `continue` enable to respectively terminate a loop body and skip the current loop instruction.
More precisely, the `break` instruction stops the first `for`, `while`, `do` or `switch` body structure. On an other hand, the `continue` instruction stops the current iteration of the current `for`, `while` or `do` loop, and starts the next one.

5 Makefile

When we separate a program in several files, one can improve the compilation phase (and save a lot of typing !) by using the Unix `make` command. The main idea of this command is to compile only what is necessary to create an executable. For instance, if only one of the source files has been modified, one do not want to recompile all the others. The `make` command searches in the current directory a file named `makefile`, or `Makefile` if it does not find it. This file specifies the dependencies between the different source files, objects and executable.

5.1 First examples

A `makefile` is a list of dependence rules in the following way:

```
target: dependency list
<TAB>     Unix commands
```

The first line specifies the target file, and then the list of files it depends on (separated by spaces), so that the next lines (which begins by a tabulation) are executed if and only if one of this file has been modified since the last modification of the target file.
For instance, one can write a `makefile` for computing the factorial of an integer in the following way (the main function, asking for the value of n, is defined in `test.c`):

```
test: test.c test.h fact.c fact.h
        gcc -o test test.c fact.c
```

Running the command `$ make test` will create the executable `test` by the command

```
$ gcc -o test test.c fact.c
```

If one does not provide any parameter to the command `make`, the first target file of the makefile will be used.
But this does not use completely the possibilities of `make`. Indeed, such direct compilation corresponds to 3 different steps:

1. the compilation of the source file `fact.c`, creating an object file `fact.o`,
2. the compilation of the source file `test.c`, creating an object file `test.o`,
3. the link editions between the two previous object files, creating the executable.

That leads to the following `makefile`:

```
test: test.o fact.o
	gcc -o test test.o fact.o

test.o: test.c test.h fact.h
	gcc -o test.o -c test.c

fact.o: fact.c fact.h
	gcc -o fact.o -c fact.c
```

If one run the make command for the first time, we get the following:

```
$ make
gcc -o test.o -c test.c
gcc -o fact.o -c fact.c
gcc -o test test.o fact.o
```

If we modify the file `fact.c`, the file `test.o` does not need to be modified. Therefore, only two step will be executed by `make`:

```
$ touch fact.c
$ make
gcc -o fact.o -c fact.c
gcc -o test test.o fact.o
```

Remark: There is an easy way to determine the dependencies of the different files we are using: the option `-MM` of `gcc`. For instance, one get:

```
$ gcc -MM fact.c test.c
fact.o: fact.c fact.h
test.o: test.c test.h fact.h
```

At this point, the makefile does not manage everything:

- one cannot generate several executables at once,
- Intermediary files created are not removed,
- we cannot force a full recompilation of the program.

We usually add the following rules:

- `all` (usually the first one of the file), that group all the executable,

- `clean` that removes all intermediary files,
- `mrproper` that removes everything that can be generated.

For instance, on our example, one would get:

```
all: test

test: test.o fact.o
        gcc -o test test.o fact.o

test.o: test.c test.h fact.h
        gcc -o test.o -c test.c

fact.o: fact.c fact.h
        gcc -o fact.o -c fact.c

clean:
        rm *.o

mrproper: clean
        rm test
```

5.2 Macro and abbreviations

To make the writing of a `makefile` easier, one can use some macro in the following way:

```
macro_name = what we want
```

When we use the command `make`, all the instance as `$(macro_name)` in the `makefile` will be replace with `what we want`.
The common macro one uses are the following:

- `CC` for the name of the compiler we use,
- `CFLAGS` for the options of compilation,
- `LDFLAGS` for the options of the link edition step,
- `DEBUGFLAGS` for the options of compilation using the debugger,
- `EXEC` for the list of executable files to generate

This leads to a file like this:

```
CC = gcc
CFLAGS = -ansi -Wall -pedantic -c
LDFLAGS =
DEBUGFLAGS = -g
EXEC = test

all: $(EXEC)

test: test.o fact.o
        $(CC) -o test test.o fact.o $(LDFLAGS)
```

```
test.o: test.c test.h fact.h
        $(CC) -o test.o test.c $(CFLAGS)

fact.o: fact.c fact.h
        $(CC) -o fact.o fact.c $(CFLAGS)

clean:
        rm *.o

mrproper: clean
        rm $(EXEC)
```

This enables to modify the `makefile` easily, by just changing the macros; for instance, if one want to change the compiler, the only line to change is the first one.
There exist a few macro already defined:

- `$@` stands for the name of the target,
- `$<` gives the name of the first dependency,
- `$^` is the list of all dependencies,
- `$*` designs the target file without any suffix.

One thus may write our `makefile` as:

```
CC = gcc
CFLAGS = -ansi -Wall -pedantic -c
LDFLAGS =
DEBUGFLAGS = -g
EXEC = test

all: $(EXEC)

test: test.o fact.o
        $(CC) -o $@ $^ $(LDFLAGS)

test.o: test.c test.h fact.h
        $(CC) -o $@ $< $(CFLAGS)

fact.o: fact.c fact.h
        $(CC) -o $@ $< $(CFLAGS)

clean:
        rm *.o

mrproper: clean
        rm $(EXEC)
```

Finally, one can create generic rules to manage a given suffix. This is done in the following way:

```
% .o: %.c
        commands
```

This gives:

```
CC = gcc
CFLAGS = -ansi -Wall -pedantic -c
LDFLAGS =
DEBUGFLAGS = -g
EXEC = test

all: $(EXEC)

% .o: %.c
        $(CC) -o $@ $(CFLAGS) -c $<

        % .do: %.c
        $(CC) -o $@ $(DEBUGLAGS) -c $<

test: test.o fact.o
        $(CC) -o $@ $^ $(LDFLAGS)

test.o: test.c test.h fact.h
fact.o: fact.c fact.h

test.db: test.do fact.do
        $(CC) -o $@ $^ $(LDFLAGS)

test.do: test.c test.h fact.h
fact.do: fact.c fact.h

clean:
        rm *.o *.do

mrproper: clean
rm $(EXEC)
```

Here the `.do` extension stands for the object files with information for the debugger, and `.db` for the files to use `gdb`

6 Preprocessor

- inclusion (`# include <file.h>`, `# include ``file.h''`),
- text substitution: `# define A 12`, `# undef A`...

  Macro with parameters:

  ```
  # define max(a,b) a>b?a:b

  max(3,4)
  ```

  after the preprocessing phase (`gcc -E`), we have

  ```
  3>4?3:4
  ```

  (that will be evaluated as `4`).

  Warning !

```
# define SQR(x) x*x
SQR(a+1)
```

will compute `a+1*a+1`, thus $2\,a + 1 \neq a^2 + 2\,a + 1$.
One can have some really weird results:

```
# define max(a,b) a>b?a:b

max(x=y,++z)
```

after the preprocessing phase (`gcc -E`), we have

```
x=y>++z?x=y:++z;
```

which is the same than

```
x = ( y>++z ? x=y : ++z)
```

different than

```
(x=y) > ++z ? x=y : ++z
```

Conclusion: Put brackets around variables (`# define max(a,b) (a)>(b)?(a):(b)`)

- condition structures:
  - `# if # endif`
  - `# if # else # endif`
  - `# ifdef # endif`
  - `# ifdef # else # endif`
  - `# ifndef # endif`

  Two examples:

```
# ifdef ERR
# define SQR(x) x*x
# else
# define SQR(x) ((x)*(x))
# endif
```

One can define a macro while computing: `gcc -D ERREUR file.c`.

```
# ifndef MYMACRO
# error "MYMACRO unknown"
# endif
```

(`# error` stops the compilation.)

7 Structures

7.1 Table

This code:

```
# include <stdio.h>

# define N 12
# define M 15

void mytabprint(int tab[], int size)
{
  int i;
  for (i=0; i<size; i++)
    printf("%d ",tab[i]);
  putchar('\n');
}

int main()
{
  int tab[N]={24, 32, 2, 4, 8, 16};
  mytabprint(tab,N);
  int tab2[M];
  mytabprint(tab2,M);
  return 0;
}
```

will print:

```
./a.out
24 32 2 4 8 16 0 0 0 0 0 0
-1075251788 134513128 -1075251800 -1215882668 0 -1216005304 1 0 1\
-1215883016 -1216094220 -1216351431 -1217219675 -1075251880 -1217320331
```

7.2 Struct

```
struct complex {
  float re;
  float im;
};

struct complex alpha;
alpha.re=4.5;
alpha.im=0.2;
```

One can initialize a variable directly:

```
struct adress{
  int num;
  char street[40];
  char code[8];
  char city[20];
};

struct people{
  char name[20];
  char first_name[25];
  int age;
  struct adress adr;
};

struct people john_doe = {"Doe", "John", 45, {39,
"London street", "N2J 1B9", "Waterloo"}};

printf("%s\n",john_doe.adr.code);
```

will print `N2J 1B9`.
One can create a "new type" (it is more a new name for another type):

```
typedef type synonym
```

For instance:

```
struct complex {
  float re;
  float im;
};

typedef struct complex complexe

complexe alpha;
alpha.re=4.5;
alpha.im=0.2;
```

This also works:

```
typedef struct complex {
  float re;
  float im;
} complex alpha = {4.5, 1.2};
```

7.3 Enumeration

```
enum color_e {CLUB, DIAMOND, HEART, SPADE, TRUMP, EXCUSE};
```

We can create an anonymous enumeration:

```
enum {JACK=11, KNIGHT, QUEEN, KING};
```

Here KNIGHT is equal to 12, QUEEN to 13…

```
typedef enum colors_e { RED, GREEN, BLUE = 5, YELLOW } colors paint_color;
```

Here the declaration is done directly. RED is equal to 0, GREEN to 1, BLUE to 5 and YELLOW to 6.

7.4 union

This enables to store in a same union different types: an object can be of one of the types defined in the structure union.

```
union day
{
  char letter;
  int number;
};
```

8 Pointers

8.1 What is a pointer ?

```
int i,j;
i=3;
j=i;
```

If the compiler put the variable `i` at the address 343211400 and the variable `j` at the address 343211404, we have:

object	address	value
`i`	343211400	3
`j`	343211404	3

```
int i=3;
int *p;
p=&i;
```

In this other program, the situation is:

object	address	value
`i`	343211400	3
`p`	343211404	343211400

```
# include <stdio.h>

int main()
{
  int i=3;
  int *p;
  p=&i;
  printf("*p = %d\n",*p);
  return 0;
}
```

will print `*p = 3`. Here we have:

object	address	value
`i`	343211400	3
`p`	343211404	343211400
`*p`	343211400	3

One can note that `i` and `*p` are identical: same adress and same value. If you change `*p`, you also change `i` !
Let consider the situation of the two next programs:

```
int main()
{
  int i=3, j=6;
  int *p1, *p2;
  p1=&i;
  p2=&j;
  *p1=*p2;
  return 0;
}

int main()
{
  int i=3, j=6;
  int *p1, *p2;
  p1=&i;
  p2=&j;
  p1=p2;
  return 0;
}
```

Before the last affectation, for both programs, the variables are:

object	adress	value
`i`	343211400	3
`j`	343211404	6
`p1`	343211984	343211400
`p2`	343212000	343211404

After the affectation `*p1=*p2;` of the first program, we have:

object	adress	value
`i`	343211400	6
`j`	343211404	6
`p1`	343211984	343211400
`p2`	343212000	343211404

whereas after the affectation `p1=p2;` of the second program, we have:

object	adress	value
`i`	343211400	3
`j`	343211404	6
`p1`	343211984	343211404
`p2`	343212000	343211404

8.2 Pointer arithmetic

A pointer is an integer, therefore, one can make some operations on them: one may

- add an integer to a pointer; the result is a pointer with the same type as the original pointer.
- subtract an integer to a pointer; the result is a pointer with the same type as the original pointer.

- make the difference between two pointers with the same type. The result is an integer.

(one cannot add two pointers).
Of course, one can use the increment ++ and decrement −− operations.
For instance,

```
# include <stdio.h>

int main()
{
  int i;
  int *p1, *p2;
  p1=&i;
  p2=p1+1;
  printf("p1 = %ld \t p2 = %ld\n",p1,p2);
  return 0;
}
```

will print `p1 = 354321840 \t p2 = 354321844`.
But if the pointers point on `double`:

```
# include <stdio.h>

int main()
{
  double i;
  double *p1, *p2;
  p1=&i;
  p2=p1+1;
  printf("p1 = %ld \t p2 = %ld\n",p1,p2);
  return 0;
}
```

will print `p1 = 354321800 \t p2 = 354321808`.
In particular, pointers are useful when dealing with tables:

```
# include <stdio.h>
# define N 5
int tab[N]={3,1,7,9,4};
int main()
{
  int *p;
  printf("Increasing order:\n";
  for (p=&tab[0]; p <= &tab[N-1]; p++
    printf("%d ",*p);
  printf("\nDecreasing order:\n";
  for (p=&tab[N-1]; p >= &tab[0]; p--
    printf("%d ",*p);
  return 0;
}
```

8.3 Dynamic allocation

When we want to use a pointer, specially when we want to use the indirection operator `*`, we have to initialize it. One can affect to the pointer the adress of another variable. One can also directly give a value to `*p`, but in such case we have to reserve a memory space of the correct size. The adress of this memory space will be the value of `p`. This is called *dynamic allocation*, and is with the `malloc` function (from `stdlib.h`):

```
malloc(number_of_octets)
```

This function returns a pointer of type `char *` that points to an object of size `number_of_octets`. If we want to use something else than an object of type `char`, we can do the following:

```
# include <stdlib.h>
int *p;
p=(int*) malloc(sizeof(int));
```

In the following example:

```
# include <stdlib.h>
int main()
{
  int i=3;
  int *p;
  p=(int*) malloc(sizeof(int));
  *p=i;
  return 0;
}
```

before the dynamic allocation, we have:

object	adress	value
i	343211400	3
p	343211404	0

Here, `*p` has no sense; if we try to use the variable `*p`, one will get a `segmentation fault`.
The dynamic allocation gives a value to `p`, and reserve at this adress a memory space equal to 4 octets. After that, we have:

object	adress	value
i	343211400	3
p	343211404	546318724
*p	546318724	? (int)

At this stage, `*p` is defined but not initialized. Its value can be any integer (the last one stored at this adress). At the end of the program, we thus have:

object	adress	value
i	343211400	3
p	343211404	546318724
*p	546318724	3

Note that the following program is different:

```
main()
{
  int i=3;
  int *p;
  p=&i;
}
```

Here we have:

object	adress	value
i	343211400	3
p	343211404	343211400
*p	343211400	3

Here `i` and `*p` are the same variable: any modification on one of them modifies the second. This is not the case in the previous example.

8.4 Pointers and tables

In C, a table is a constant pointer: if one has

```
int tab[10];
```

tab is a constant pointer, with for value the adress of the first element of the table. One can access to the element of index `i` of `tab` with `tab[i]`, or also `*(tab+i)` (these two ones are equivalent).
One can create a table with `n` elements where `n` is a variable of the program as follows:

```
# include <stdlib.h>
main()
{
  int n;
  int *tab;
    ...
  tab=(int *)malloc(n*sizeof(int));
    ...
  free(tab);
}
```

If we use the function `tab = (int*)calloc(n,sizeof(int));` instead of `malloc`, all the elements of `tab` are also initialized to 0.
One always have to allocate the memory of such table, and if we do not want any memory problem later on, one has to free the memory once used.
One can create a matrix with `k` lines and `n` columns as follows:

```
# include <stdlib.h>
main()
{
  int k, n,i;
  int **tab;
    ...
  tab=(int **)malloc(k*sizeof(int*));
  for (i=0; i<k ; i++)
    tab[i]=(int *)malloc(n*sizeof(int));
    ...
  for (i=0; i<k ; i++)
    free(tab[i]);
  free(tab);
}
```

It is also possible to have a different number of elements for each line.

```
for (i=0; i<k ; i++)
  tab[i]=(int *)malloc((i+1)*sizeof(int));
```

8.5 Pointers and structures

In C, structures have an adress. Therefore, if we want to modify one given as an input of a function, it is better to use a pointer on this structure. Also, if the structure takes a lot of memory space, it is better to use a pointer to this structure than the structure itself when we use it as a parameter of an intermediary function: as copies of the parameter are made when the function is called, it is better to copy a pointer (4 octets) than the structure (specially if it contains for instance a big table).
If `p` is a pointer on a structure, one can access to a member of this structure by the following expression:

```
(*p).member
```

The following line is equivalent:

```
p->member
```

For instance, one could replace `tab[i].date` by `(tab+i)->date`.

8.6 Linked list

It is very often useful to create a structure with one member that is a pointer to a structure with the same model, as for instance:

```
struct cell
{
int value;
struc cell *next;
};
```

This enables to create a linked list as a pointer that points to such a structure:

```
typedef struct cell *list;
```

With such a list, it is easy to insert an element:

```
list insert(int elt, list l)
{
  list l2;
  l2=(list) malloc(sizeof(struct cell));
  l2->value=elt;
  l2->next=l;
  return l;
}
```

One example: the following program create a list of integers

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

struct cell
{
  int value;
  struct cell *next;
};

typedef struct cell *list;

list insert(int elt, list l)
{
  list l2;
  l2=(list) malloc(sizeof(struct cell));
  l2->value=elt;
  l2->next=l;
  return l;
}

main()
{
  list l, p;
  l = insert(1,insert(2,insert(3,insert(4,NULL))));
  printf("\n The list is:\n");
  p = l;
  while (p != NULL)
    {
      printf("%d \t",p->value);
      p = p->next;
    }
}
```

A last example of structure: one can manage a binary tree in the following way:

```
struct node
{
  int value;
  struct node *left;
  struct node *right;
};

typedef struct node *tree;
```

9 Function and variables

9.1 Variables

- **Local variables:** they are defined inside a function, and are destroyed at the end of the function call.
- **Global variables:** these are the variables declared outside of any function. One make declaration in headers files, but we define them only once, in a .c file.

- **Static variables:** they take a place in the memory that will never change during all the program. The memory allocated is part of the data memory. These variables are initialized at 0 by default. This is the only way to keep a local variable from being lost at the end of a function call.
- **Constant variables:** a variable defined with `const` cannot be modified. One use it quite often for character chain.
  - `const char *p` is a pointer on a constant character.
  - `char * const p` is a constant pointer on a character.
- **volatile variable:** such variable cannot be modified by any optimisation of the compiler. This is used whenever we consider a variable that may be modified outside the program.

Some examples:

```
int n = 10, k=20;
void fonction();
void fonction()
{
  int n = 0;
  n++;
  printf("call %d\n",n);
  printf("global %d\n",k);
  return;
}
main()
{
  int i;
  for (i = 0; i < 3; i++)
    fonction();
  printf("oustside the function %d\n",n);
}
```

will print

```
call 1
global 20
call 1
global 20
call 1
global 20
outside the function 10
```

```
# include <stdio.h>

int n = 10;
void fonction();
void fonction()
{
  static int n;
  n++;
  printf("call %d\n",n);
  return;
}
main()
{
  int i;
  for (i = 0; i < 5; i++)
    fonction();
  printf("oustside the function %d\n",n);
}
```

will print

```
call 1
call 2
call 3
call 4
call 5
outside the function 10
```

9.2 the main function

The structure of the main function is the following:

```
int main(int argc, char* argv[]);
```

which can also be written as

```
int main(int argc, char** argv);
```

- `argc` is the number of arguments given as an input to the executable;
- `argv` is a table of pointers. `argv[0]` will be the name of the executable, `argv[1]` the first parameter given to the executive, `argv[2]` the second...

One example:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
  int a, b;
  if (argc != 3)
    {
      printf("\nError : bad number of arguments");
      printf("\nUsage: %s int int\n",argv[0]);
      exit(EXIT_FAILURE);
    }
  a = atoi(argv[1]);
  b = atoi(argv[2]);
  printf("\nThe product of %d and %d is : %d\n", a, b, a * b);
  exit(EXIT_SUCCESS);
}
```

After compilation, if we run

```
$ a.out 9 4
```

we will have the result:

```
The product of 9 and 4 is : 36
```

Here, the `atoi` function, from the standard library, converts a character chain into an integer.

9.3 Pointers on function

```
int operation(int, int, int(*)(int, int));

int operation(int a, int b, int (*f)(int, int))
{
return((*f)(a,b));
}
```