Rapide mise à niveau en C

C pour Camliens

Références

- Kernighan, Ritchie, *"Le langage C, norme ANSI"*
  - disponible à la bibliothèque
- Bernard Cassagne, *"Introduction au langage C"*
  - disponible sur la page www du cours
    (de même que ces transparents)

C: fonctions

- en C comme en Caml, un programme est fait de *fonctions*, comme

```
int f(int a, int b){
  int c;              /* les variables auxiliaires au début */

  c = 2*a + b;
  c = c+1;
  return c;    /* sortie de la fonction */
}
```

- un programme contient la fonction `main`: figure imposée

```
int main(void){
  :
  :
  return 0;
}
```

Comment compiler le C

- le source est dans un fichier `toto.c`
- `gcc -Wall toto.c -o toto`
  - `-Wall`: “tous les warnings” (le compilateur dit tout)
  - traiter les warnings *dans l'ordre*
  - `toto` est l'exécutable (taper `./toto`) pour lancer le programme
- jusque là les programmes ne font rien, il faut savoir faire des effets de bord

printf et les types

- une fonction pour la *sortie formatée*

```
#include <stdio.h>    /* printf déclarée dans le fichier stdio.h */
⋮
printf("un entier:  %d, un flottant:  %f\n",n,f);
printf("juste un message");
```

  - `printf`: fonction à nombre d'arguments variable
  - `#include <stdio.h>`: en tête du fichier pour pouvoir utiliser `printf`
  - `%d`,`%f`: séquences d'échappement

- les *types* en C:

`char`	`int`	`long int`	`float`	`double`	
1	4	4	4	8	octets
8	32	32	32	64	bits
`%c`	`%d`	`%d`	`%f`	`%f`	pour afficher

DÉMO `taille_types.c`

- pas de type bool: `0` signifie `false`, tout le reste c'est `true`

Types, coercions, structures

- les types de base “communiquent” en C
  - `(t)e`: l'expression `e` vue avec le type `t`
  - on appelle cela coercion, ou *cast*, ou transtypage

    DÉMO `cast.c`
- les enregistrements s'appellent en C des *structures*

```
struct coup
{
  char colonne;
  int ligne;
};
```

  - le type s'appelle `struct coup`
  - `struct coup c = {'E', 2};...c.ligne = 4;...struct coup c'=c;`
  - on ne peut pas comparer ou afficher les structures directement
  - DÉMO `structs.c`

Adresses, pointeurs

- tout objet est quelque part en mémoire, tout objet a une adresse
- si `a` désigne une entité en C, `&a` est l'*adresse* de `a`
- inversement, si `p` est une adresse (un pointeur), `*p` désigne ce qui est stocké à l'adresse `p` DÉMO `maxmin.c`
- l'"inverse" de `printf`: `scanf` DÉMO `scanagain.c`
- on peut faire une déclaration `int a, *p;`
  - `a` est un entier
  - `p` est un pointeur sur un entier

DÉMO `locate.c`

Au sujet du passage par adresse

- suivant que l'on passe à une fonction une valeur ou une adresse, on donne des droits différents (`f(x)` et `g(&y)`)
- ce n'est pas "à la carte", c'est prévu dans la définition de la fonction (le type le dit)

- fonctions en C: **toujours des parenthèses**

Instructions

- jusqu'ici on a les fonctions (et `return`), et l'affectation
  - NB: du code après un `return` est inutile

```
int f(void){
  return 32;
  printf("voici ce que je pense du cours de
programmation:\n");
}
```

- `printf`, c'était un appel de fonction, pas une instruction
- branchement conditionnel

```
if (temp>25) printf("il fait chaud\n");
             else printf("il fait doux\n");

if (richter==6) printf("badaboum\n");
```

- boucles: `for` (*pour*), `while` (*tant que*)

```
for (i=0; i<10; i=i+1) total = total + f(i);

while (r-x>epsilon) x = f(x);
```

- le test d'arrêt pour le `for` se lit comme dans un `while` (sur l'exemple, ce n'est pas `i==9`)
- des `;` pour séparer les constituants du `for`!!

Remarque: forme des instructions

```
if (temp>25) printf("chaud\n"); else printf("doux\n");
while (r-x>epsilon) x = f(x);
for (i=0; i<10; i=i+1) total = total + f(i);
```

`if` *(test) instruction;* `else` *instruction;* *(pas de `then`)*

`while` *(test) instruction;*

`for` *(instruction; test; instruction) instruction;*

- comparaison: égalité à zéro ($\neq 0$: vrai, $= 0$: faux)
- le test d'égalité s'écrit `if (a==12)...`!!
- les parenthèses sont importantes
- une instruction peut être remplacée par un *bloc:*
  { *instruction; . . . ; instruction;* }
  - c'est souvent le cas dans le corps des fonctions
  - un bloc peut *commencer* par la **déclaration**/définition de variables locales
    `{int a = 3; printf(...);...}`
    les gens peuvent exister sans avoir de valeur ($\neq$ Caml)

Les affectations

- une affectation: `g = d` (en OCaml, `g := d`)

  en C, une affectation est une *expression* *(eh oui, comme en Caml)*
  renvoyant la valeur du membre de droite
- ainsi, `g = g' = d;`
  - se lit `g = (g' = d);`
  - et "signifie" `g' = d; g = g';`
- abréviations:
  - `x = x+5` ⇝ `x += 5`
  - `x = x+1` ⇝ `x++`, ou plutôt `++x` DÉMO `abrev.c`
- il y a des expressions, il y a même la séquence (`;` en Caml)
  `a++,2` renvoie `2`

Tics de langage

- `while(i--)printf("je continue!\n");`
  - `i--` signifie `i--!=0`
- `for (i=j=0; i-2; i++,j+=12){...};`
  - insistons: des `;` séparent les éléments du `for`
  - utilisation de la séquence pour faire deux incrémentations
- `while(i<12)t[i++]='a';`
  - `t[n]`: `n`ème case du tableau `t`
-
```
int fact(int n)
{
  int r;
  for(r=n;n-1;r*=--n);
  return r;
}
```