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

Mémoire

- les données manipulées par le programme sont stockées "quelque part" dans la mémoire
- pour nous, la mémoire est une grande étendue d'*octets* regroupés en *mots mémoire*
- en C, on est très rapidement conscient de cela
- 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 `scandate.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**

  pas si lamentable que ça, comme remarque

Pointeurs, exemple

- supposons que 47003 est stocké en mémoire comme valeur associée à la variable `a` [a | 47003]

```
(int)a      47003
(int *)a    0xb79b
&a          0xbfda4620
(int)(&a)   -1076214249
```

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`!!

  pour le coup, lamentable

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* 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`
- 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]`: nème case du tableau `t`

```
int fact(int n)
  {
    int r;
    for(r=n;n-1;r*=--n);
    return r;
  }
```

Pointer sur une structure

```
struct coup { char colonne; int ligne; };    /* un coup aux échecs */
```

colonne
ligne

```
int f(struct coup c){
  struct coup *a;

  a = &c;
  (*a).colonne = 'E';
  (*a).ligne = 2;
  ...
```

- `(*a).colonne` et non pas `*a.colonne`
- notation: `a->colonne` synonyme de `(*a).colonne`

```
a->ligne = a->ligne + 1
```

Pointer sur une structure – "récursivité dans les données"

```
struct personne
  {
  int age;
  struct personne *suivant;
  };
```

- manipulation:

```
struct personne adam = { 0, ???  };
```

- le pointeur invalide: `NULL`

constante définie dans `stddef.h`, sert en particulier pour signaler une erreur

Sur l'allocation

- DÉMO fichier `alloc.c`
- la fonction `malloc` attend une taille et renvoie un pointeur vers une zone mémoire nouvellement allouée

```
struct truc *p;
p = (struct truc *) malloc(sizeof(struct truc));
```

  - pour faire `malloc`, il faut `#include <stdlib.h>`
  - `void * malloc(size_t SIZE)`
    - `void *`: le type le plus général d'un pointeur
    - il est bon de transtyper (coercer) à chaque utilisation de `malloc`
  - ...et d'ailleurs le "contraire" de `malloc` s'appelle `free`

Deux petits trucs

- le préprocesseur de C
  - `#include <stdio.h>`
    lire les prototypes des fonctions fournies par la librairie `stdio`
  - `#define MAX 100`
    macro (abréviation)
  - majuscules et minuscules: indifférent, mais des coutumes
- définition de types
  `typedef` DÉMO `def_type.c`

La mémoire et les tableaux

On veut ranger une quantité d'entités de même type en mémoire: on les met *côte à côte*, et on appelle ça un *tableau*

Exemple: tableau d'entiers

k_0	k_1	...	k_{n-1}

↑ adresse de début du tableau

Un tableau, c'est une *"tranche de mémoire"*
ainsi qu'une "granularité de la tranche"

Tableaux – déclaration et manipulation

- déclaration `int mois[12];` 12 entiers
- accès aux "cases" du tableau, de `mois[0]` à `mois[11]`

```
mois[1] = 28;
mois[2] = mois[0];
```

- initialisation

```
char voyelles[5] = {'a', 'e', 'i', 'o', 'u'};
```

  - si on initialise partiellement, le reste est mis à zéro
  - pour mettre tout le tableau à `'i'`, on doit faire une boucle
- il ne faut pas sortir d'un tableau (mais pas de contrôle **statique**)
  - tout plante `Segmentation fault`
  - ...ou pas (c'est encore pire)

Le cas particulier des tableaux de caractères

- en C, les chaînes de caractères sont des tableaux de `char`

  pas besoin d'écrire
  `char s[6] = {'b','a','t','e','a','u'};`,
  on peut directement écrire `char s[] = "bateau";`

  mais ceci définit un tableau de 7 caractères, le dernier étant `'\0'`
- impression formatée:
  `printf("%s\n",x);` `x` doit être terminé par `'\0'`
  (sinon ça peut être spectaculaire)

  Rq: `'\0'` c'est `false`, c'est `0`...
- exemples
  DÉMO `longstr.c, str_len.c, autre_str_len.c, virer.c`

Qu'est-ce qu'un tableau?

- un tableau `t`:
  - une adresse de début (`t`, ou `&t[0]`)
  - une "granularité" (type du contenu)
- ce qu'est `t`
  - au moment de la création de `t`, la taille est associée au tableau *(allocation mémoire)*
  - après quoi elle est oubliée, et `t` est une *constante*, de type *tableau d'entiers*
  - du coup, lorsque l'on passe `t` à une fonction, on passe l'*adresse* de début du tableau
    - on passe l'adresse, pas vraiment "le tableau"
    - `void initialise (int t[N]){ ... };` est autorisé mais ne sert à rien, car `t` n'existe que comme adresse de début d'un tableau d'entiers
      ⇝ passer `N` comme argument supplémentaire de `initialise`
- la taille d'un tableau doit être déterminée *statiquement*:
  pas de `int n=10; int tab[n];`

Tableaux à deux dimensions

- tableaux de tableaux
  ex.: `int t[3][10]` 3 tableaux de 10 entiers

```
for (ligne=0; ligne<3; ligne++)
  for (colonne=0; colonne<10; colonne++)
    printf("%d ",t[ligne][colonne]);
```

- NB: `t[i][j]` et non `t[i,j]` !!
  (`i,j` c'est la séquence)
- on peut omettre *une* coordonnée:
  `void remplit(int t[ ][10])`
  - tableaux de tableaux et contiguïté: imbrication des boucles
  - les matrices sont rectangulaires
    - ≠ les `int array array` en OCaml
    - DÉMO `matrices.c`

Relations entre pointeurs et tableaux

soit `int t[10];`

- tableau `t`: adresse de `t[0]` + type du contenu de `t` + taille
- définissons un pointeur `int *p = t;`
  - on peut faire `p[0], p[1], p[2], ...`
  - ...**mais aussi** `*p, *(p+1), *(p+2)`
    = arithmétique sur les pointeurs
    - `p[i]` abrège `*(p+i)`
    - et renvoie *ici* la même chose que `t[i]` *(car `int *p = t;`)*
    - l'information "`int`" dans le type de `p` (et de `t`) sert à savoir la taille des pas lorsque l'on fait `p+i`
- le nom "`t`" dénote une expression constante de type `int *` *(pas de `t++`)*
  - un tableau est un tableau *à la déclaration*
  - après, c'est un 'pointeur constant'
- un tableau implémente une forme d'*"appel par adresse indexé par un entier"*

Arithmétique des pointeurs, suite

- on utilise volontiers un pointeur pour parcourir un tableau

```
for (p=&message[0]; *p !='\0'; p++)...
int *p;   p=&t[4];
```

- non seulement peut-on faire `*(p+i)`, mais on peut aussi faire `p-q`, si `p` et `q` sont deux pointeurs *pointant à l'intérieur d'un même tableau*, ce qui renvoie un `int` *(en fait, un ptrdiff_t)*
- *Say it loud:* si `t` est un tableau, on ne peut modifier `t`
  ...mais on peut modifier un pointeur

```
int premier_zero(int t[])
{
  int *p = t;
  while(*p!=0)p++;
  return (p-t)+1;
}
```

Debugger les programmes: gdb

Debugger avec gdb

- un debugger: programme qui "supervise" l'exécution d'un autre programme
- une option particulière pour la compilation:

```
gcc -Wall -g machin.c -o machin
```

- DÉMO `demo_gdb.c`