C, seconde séance

- le `if then else` n'est pas une expression, mais une instruction ($\neq$ Caml)
- se souvenir de:
  - `false`, c'est `0` (`'\0'` aussi)
  - test d'égalité: `==`
  - pour `printf`: `%d` entier, `%f` flottant, `%c` caractère, `%p` adresse (pointeur), `%s` chaîne de caractères (tableau de `char`)
- en mettant les initialisations avant la boucle, et les incréments dans le corps de la boucle, on transforme un `for` en un `while`

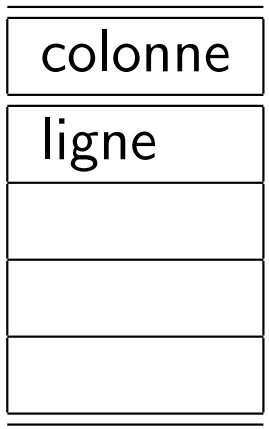
Deux petits trucs

- le préprocesseur de C
  - `#include <stdio.h>`
    lire les prototypes des fonctions fournies par la librairie `stdio`
  - `#define MAX 100`
    macro (abréviation)
- définition de types

  `typedef` DÉMO `def_type.c`

Pointer sur une structure

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




```
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

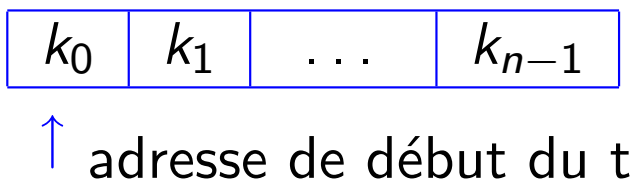
```
struct truc *p;
p = (struct truc *) malloc(sizeof(struct truc));
```

  - 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`

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



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)
  - DÉMO `listes.c`

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
      $\rightsquigarrow$ 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
    - $\neq$ 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]`
    - 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`