Initiation au C
cours n°4

Antoine Miné

École normale supérieure

8 mars 2007

Plan du cours

- le manuel,
- les pointeurs et les références,
- les entrées au clavier avec scanf.

Les pages de man

Le manuel

man = manuel intégré à Unix

Mode d'emploi

- dans le terminal, on tape : `man` *mot-clé*
- navigation :

flèches	haut / bas
espace	page suivante
b	page précédente

p	début
q	quitter
/	rechercher

En fait, ce sont les commandes de `less`.

Contenu du man

Ce qui est documenté :

- les commandes Unix : *e.g.* `man gcc`,
- les fonctions de la bibliothèque C : *e.g.* `man printf`,
- les en-têtes de la bibliothèque C : *e.g.* `man stdio.h`,
- la commande `man` : `man man`.

Exemple de page de man

man cos (début)

```
$ man cos
COS(3)                          Linux Programmer's Manual


NAME
       cos, cosf, cosl - cosine function

SYNOPSIS
       #include <math.h>

       double cos(double x);
       float cosf(float x);
       long double cosl(long double x);

       Link with -lm.
```

Exemple de page de man

man cos (fin)

```
DESCRIPTION
       The  cos()  function returns the cosine of x,
       where x is given in radians.

RETURN VALUE
       The cos() function returns a value between
       -1 and 1.

CONFORMING TO
       SVr4, 4.3BSD, C99.  The float and long double
       variants are C99 requirements.

SEE ALSO
       acos(3), asin(3), atan(3), atan2(3), ccos(3),
       sin(3), tan(3)
```

Les sections du man

Les pages sont regroupées en sections.

Sections

1	Commandes UNIX
2	Appels systèmes en C
3	Bibliothèque standard C
4	Fichiers spéciaux `/dev/*`
5	Formats de fichiers, configuration
6	Jeux
7	Variés
8	Administration système
⋮	⋮
0p	En-têtes `*.h`
n	Bibliothèque Tcl

Options de man

`man` *mot-clé* : affiche la première page trouvée pour *mot-clé*.

Attention

Une page de `man` peut en cacher une autre !
(même nom, section différente)

options de `man` :

- `man -s` *section mot-clé* : cherche dans une section précise,
- `man -a` *mot-clé* : affiche toutes les pages pour *mot-clé*, tapez q pour passer à la page suivante,
- `man -f` *mot-clé* : liste les pages de titre *mot-clé*,
- `man -k` *mot-clé* : liste les pages contenant *mot-clé* dans leur titre ou leur description succincte.

Pointeurs et références

La mémoire

bit = chiffre binaire : 0 ou 1.

octet (byte) = 8 bits, donc 2^8 positions,

- "atome" de mémoire : tout est compté en octets,
- `unsigned char` : nombre dans [0 ;255],
- `signed char` : nombre dans [-128 ;127],
- `char` = `signed char` ou `unsigned char` (selon la machine)

(unsigned) int = taille "naturelle" selon la machine

- machine "16-bits" : 2 octets,
- machine "32-bits" : 4 octets,
- machine "64-bits" : heu, toujours 4 octets (pour compatibilité).

Modèle mémoire simplifié

Mémoire $\simeq$ tableau d'octets.
Chaque octet a une adresse en mémoire.

Modèle prédominant = mémoire plate.
L'adresse est un entier :

- machine "32-bits" : 4 octets $\Rightarrow$ 4 Go adressables
- machine "64-bits" : 8 octets $\Rightarrow$ 17 GGo adressables (!)

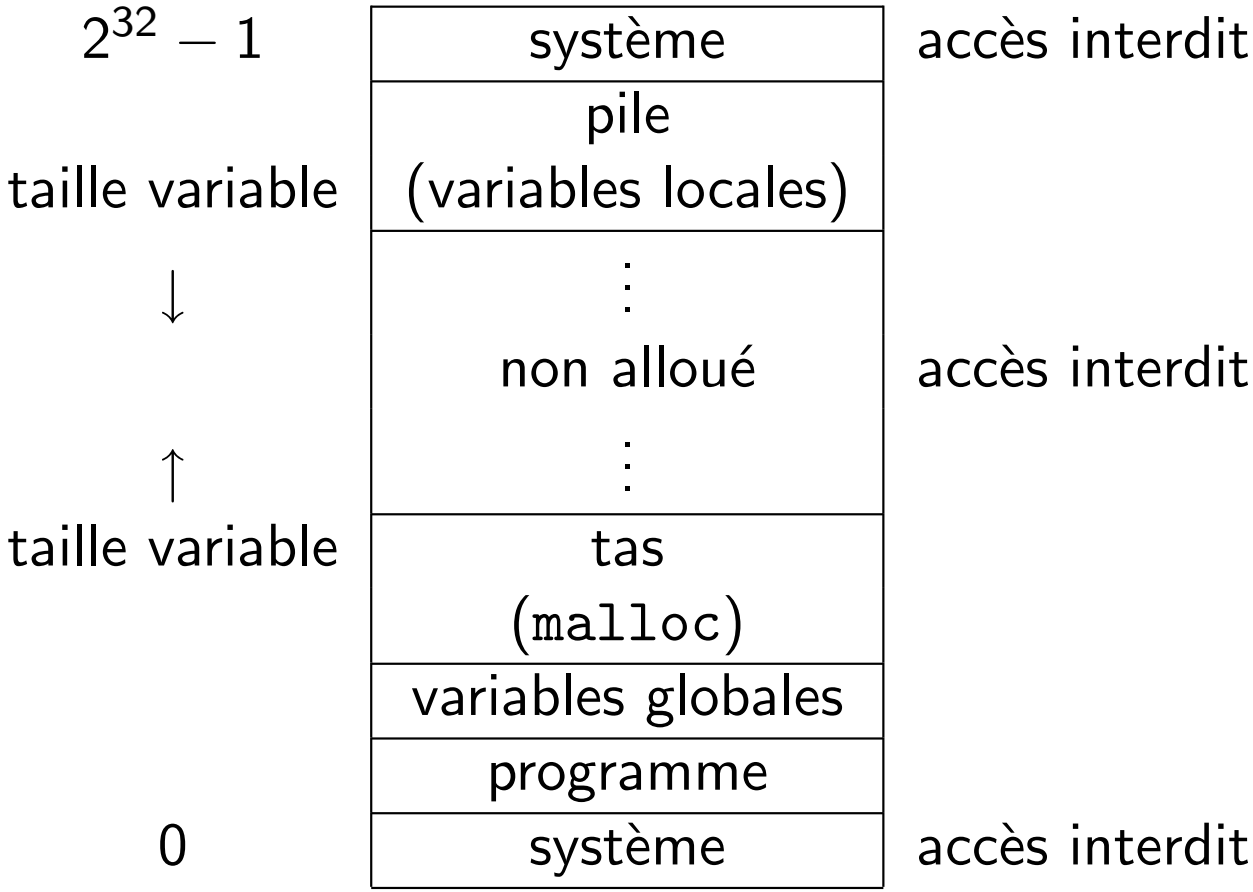
D'autres modèles existent : (segmentés, paginés, etc.)

Virtualisation = gestion et protection de la mémoire :

- chaque programme a son espace mémoire "logique",
- l'OS gère la correspondance mémoire logique $\longrightarrow$ physique.

Modèle mémoire simplifié

Exemple d'espace logique (Linux 32-bits) :



Les adresses et le C

Les **pointeurs** en C

Notion d'adresse :

- abstraite ($\neq$ entiers),
- typée.

En C, on peut :

- obtenir l'adresse d'une variable existante (&),
- accéder au contenu stocké à une adresse valide (*),
- passer des adresses en argument, les retourner, les copier (=),
- effectuer des opérations limitées sur les adresses (+, ==).

L'opérateur d'adresse &

Syntaxe

& *expr*

Renvoie l'adresse d'un "objet" en mémoire.

expr doit être une *lvalue* (*i.e.*, modifiable) :

- variable scalaire,
- case d'un tableau.

Exemples

```
int i, a[2];
&i                  /* adresse de i */
&(a[1])  &a[1]      /* adresse de a[1] */
&a   &(i+1)         /* invalides */
```

Les types pointeur

Types des pointeurs

`t*` = pointeur sur un objet de type `t`.

⇒ si `expr` a pour type `t`,
alors `&expr` a pour type `t*`.

Exemple : `&i` et `&a[1]` ont pour type `int*`.

Attention

Le type pointé est important !
Si `t1`≠`t2`, alors `t1*` et `t2*` sont incompatibles.

Les variables pointeurs

On peut déclarer des variables de type `t*`.

Déclaration d'un pointeur

```
t* var ;
```

`var` peut contenir l'adresse de tout objet de type `t`.

`var` ne peut pas contenir l'adresse d'un objet de type différent !

Exemple

```
int i;
float f;
int* p = &i;   /* p pointe sur i */
p = &f;        /* non défini */
p = &(i+1);    /* erreur de syntaxe */
```

L'opérateur de déréférencement *

Syntaxe

*expr

déréférencement = accès à l'objet pointé par l'expression `expr` :

- si `expr` est de type `t*`, `*expr` est de type `t`,
- inverse de `&` : `*&expr` $\simeq$ `&*expr` $\simeq$ `expr`,
- `*expr` est une *lvalue*, donc modifiable.

Exemple

```
int x,y;
int *p = &x;
*p = 2;       /* place 2 dans x */
 p = &y;
*p = *p+1;  /* incrémente y */
```

Copies de pointeurs

Soit : `int x = 1, y = 2;`
`int *p = &x, *q = &y;`

Que valent `x`, `y`, `p` et `q` après :

- `p = q; *p = -1;`

- `*p = *q; *p = -1;`

?

Copies de pointeurs

Soit : `int x = 1, y = 2;`
`int *p = &x, *q = &y;`

Que valent `x`, `y`, `p` et `q` après :

- `p = q; *p = -1;`
  `p` et `q` pointent sur `y`, (alias !)

- `*p = *q; *p = -1;`

?

Copies de pointeurs

Soit : `int x = 1, y = 2;`
`int *p = &x, *q = &y;`

Que valent `x`, `y`, `p` et `q` après :

- `p = q; *p = -1;`
  `p` et `q` pointent sur `y`, (alias !)
  `y = -1.` `x` est inchangé.
- `*p = *q; *p = -1;`

?

Copies de pointeurs

Soit : `int x = 1, y = 2;`
`int *p = &x, *q = &y;`

Que valent `x`, `y`, `p` et `q` après :

- `p = q; *p = -1;`
  `p` et `q` pointent sur `y`, (alias !)
  `y = -1.` `x` est inchangé.
- `*p = *q; *p = -1;`
  `*q = y = 2` est placé dans `*p = x`,

?

Copies de pointeurs

Soit : `int x = 1, y = 2;`
`int *p = &x, *q = &y;`

Que valent `x`, `y`, `p` et `q` après :

- `p = q; *p = -1;`
  `p` et `q` pointent sur `y`, (alias !)
  `y = -1.` `x` est inchangé.
- `*p = *q; *p = -1;`
  `*q = y = 2` est placé dans `*p = x`,
  puis `-1` est placé dans `*p = x.` `y` est inchangé.

!

Utilité des pointeurs

Les pointeurs peuvent servir à

- passer des variables par référence,
- simuler le retour de plusieurs valeurs,
- lire les des données entrées au clavier (fin de ce cours)
- traverser des tableaux,
- manipuler des chaînes de caractères (cours suivant),
- gérer des blocs de mémoire dynamique (pas tout de suite).

Passage par référence

Exemple

```
void mul2(double* d) /* par référence */
{
  *d *= 2;   /* double le contenu de d */
}

double power(double arg, int n) /* par valeur */
{
  for ( ; n>0; n-- ) mul2(&arg);   /* double arg */
  return arg;
}

void test()
{
  double f = 12.;
  double g = power(f, 10);   /* f non modifié */
}
```

“Retour” de plusieurs valeurs

Exemple

```
void divise(int a, int b, int* div, int* rem)
{
  *div = a / b;
  *rem = a % b;
}

void f()
{
  int x, y;
  divise( 100, 10, &x, &y );
}
```

Attention

C’est à l’appelant d’allouer la mémoire pour les valeurs de “retour”.

Le pointeur NULL

NULL = valeur pointeur spéciale :

- définie dans `#include <stdlib.h>`,
- de type générique `void*` non déréférençable,
- garantie de ne jamais pointer vers une adresse "valide"
  $\Rightarrow$ distinguable de toute `&x`.

Utilisations standard :

- utilisée comme valeur pour "non définie",
- renvoyée par une fonction pour indiquer une erreur,
- passée en argument pour indiquer qu'on n'est pas intéressé par une valeur de retour.

Le pointeur NULL

Exemple

```
#include <stdlib.h>
void divise(int a, int b, int* div, int* rem)
{
  if (div!=NULL) *div = a / b;
  if (rem!=NULL) *rem = a % b;
}
```

Notes :

- `if (p)` équivaut à `if (p!=NULL)`,
- `if (!p)` équivaut à `if (p==NULL)`.

Pointeurs valides et invalides

Avant de déréférencer un pointeur par *,
assurez-vous qu'il pointe vers un objet valide !

Pointeurs valides

- pointeur vers une variable globale,
- pointeur vers une variable locale existante.

Pointeurs invalides

- pointeur `NULL` ou non initialisé,
- pointeur en dehors des bornes d'un tableau,
- pointeur vers une variable locale détruite,
  $\Rightarrow$ ne jamais retourner un pointeur vers une variable locale !

Attention : la durée de vie d'une variable–pointeur peut dépasser celle de l'objet sur lequel elle pointe !

Exemples incorrects

Exemple incorrect

```
void g(int* x)
{
  *x = 2;
}

void main()
{
  int* z;
  g(z);    /* ERREUR: z non initialisé */
  {
    int k;
    z = &k;
    g(z);  /* OK, équivalent à g(&k) : g modifiera k */
  }
  g(z);    /* ERREUR: k n'existe plus, z est invalide */
}
```

Exemples incorrects

Exemple incorrect

```
int* f()
{
  int z = 12;
  return &z;
}

void main()
{
  int * x = f();
  *x = 13;        /* ERREUR: z n'existe plus */
}
```

Note : l'adresse d'une variable locale change entre deux appels d'une même fonction !

Arithmétique de pointeurs

Arithmétique : pour se déplacer dans un tableau unidimensionnel.

Si `p` pointe sur une case d'un tableau :

- `p+i` ou `i+p` $\Rightarrow$ pointe *i* cases après `p`
- `p-i` $\Rightarrow$ pointe *i* cases avant `p`

  (ajouter `i` $\simeq$ se déplacer de `i` $\times$ `sizeof(*p)` octets...)

Note : les raccourcis `+=`, `-=`, `++`, `--` marchent également.

On obtient un pointeur invalide si :

- on dépasse des bornes du tableau,
- on déplace un pointeur sur un scalaire ($\simeq$ tableau de taille 1).

Impossible de "sauter" d'une variable à une autre par arithmétique.

Chaque variable est une île.

Comparaison de pointeurs

On peut comparer deux pointeurs pour :

- l'égalité `==` (pointent sur la même adresse ?)
- la différence `!=` (pointent sur des adresses différentes ?).

Si `p` et `q` pointent dans le même tableau, on peut de plus :

- comparer les indices des cases : `p < q`, `p <= q`, etc.
- calculer la distance en cases : `p - q`.

Pointeurs et tableaux unidimensionnels

Exemple

```
void cherche_zero(int* tab, int nb)
{
  for (; nb>0; nb--, tab++ )
    if ( *tab == 0 ) return 1;
  return 0;
}

void f()
{
  int a[100];
  ...
  if ( cherche_zero( &a[10], 15 ) ) ...
}
```

Avantage : remplace un couple tableau + indice.

Pointeurs et tableaux unidimensionnels

Dans une expression, tout tableau unidimensionnel est remplacé par un pointeur vers son premier élément.

Équivalences

`tab`	$\simeq$	`&tab[0]`	
`tab+i`	$\simeq$	`&tab[i]`	
`*tab`	$\simeq$	`tab[0]`	
`*(tab+i)`	$\simeq$	`tab[i]`	
`i[tab]`	$\simeq$	`tab[i]`	(!)

Exception : `sizeof(tab)` renvoie la taille du type de `tab`.
(attention si `tab` est un argument !)

Tableaux multidimensionnels : c'est plus complexe et moins utilisé.

Pointeurs complexes

Exemples complexes :

- `int** x ;` pointeur sur un pointeur sur un `int`,

  `*x` : pointeur sur un `int`,
  `**x` : `int`.

- `int *x[10] ;` tableau de 10 pointeurs sur des `int`,

  `x[1]` : pointeur sur un `int`,
  `*(x[1])` : `int`.

- `int (*x)[10] ;` pointeur sur un tableau de 10 `int`.

  (inutile : on préférera un pointeur sur un élément du tableau)

Priorité des opérateurs

Du plus prioritaire au moins prioritaire.

`[]`	accès dans un tableau
`++ --`	incrémentation et décrémentation
`*`	déréférencement de pointeur
`&`	prise d'adresse
`* / %`	opérateurs multiplicatifs
`+ -`	opérateurs additifs
`== < > ...`	opérateurs de comparaison
`&& ||`	opérateurs booléens
`= op=`	opérateurs d'affectation

Exemple : `*p++` signifie `*(p++)`, pas `(*p)++` ;

$\Rightarrow$ dans le doute : mettre des parenthèses.

Priorité dans les déclarations

Attention à la priorité de `*` et `,` dans les déclarations.

- `int *a,b ;`
  `b` a pour type `int`, pas `int*`.
- `int *a,*b ;`
  `a` et `b` ont le type `int*`.

Entrées au clavier

La fonction scanf

scanf : lit au clavier des entiers, flottants, etc :

- 1er argument : *format* entre " "
  $\simeq$ liste ce qui est attendu, avec le type de chaque élément,
- arguments suivants : pointeurs
  indiquent où stocker chaque objet lu.

Exemples

```
#include <stdio.h>

int x;
scanf( "%i", &x);

char c;
float a,b;
scanf( "%f %c truc %f", &a, &c, &b );
```

Le format de scanf

séquence	action	type du paramètre
%i	lit et retourne un entier	int*
%li	lit et retourne un entier	long*
%Li	lit et retourne un entier	long long*
%f	lit et retourne un flottant	float*
%lf	lit et retourne un flottant	double*
%Lf	lit et retourne un flottant	long double*
%c	lit et retourne un caractère	char*

espace	lit un ou des espace(s) ou \n	—
%%	lit un caractère %	—
toto	lit exactement le mot toto	—

scanf retourne le nombre d'éléments reconnus et stockés.