

# Initiation au C

## Cours n°1

Antoine Miné <sup>1</sup> Ozan Caglayan <sup>2</sup>

<sup>1</sup>École Normale Supérieure

<sup>2</sup>Université Galatasaray

19 février 2015



## Introduction

Premier programme en C

Premiers concepts en C

## Repères historiques

Généralités sur le C

# Repères historiques

---

# Repères historiques

## Origine

Denis Ritchie et Ken Thomson (Bell Labs) cherchent un langage pour reprogrammer UNIX de façon portable.  
Le C est dérivé du B (1969), BCPL (1966), CPL (1960), etc.

## Historique

- 1969 UNIX par Ken Thomson (assembleur pour DEC PDP-7)
- 1972 invention du C par Denis Ritchie
- 1973 UNIX en C par Denis Ritchie et Ken Thomson (PDP-11)
- 1978 *The C Programming Language* : C K&R
- 1989 1ère normalisation : ANSI C (C89), ISO C90
- 1999 2ème normalisation : ISO C99

# Le C aujourd'hui

## Le C est toujours très utilisé :

- systèmes d'exploitations : Linux  $\simeq$ 6,8 millions de lignes
- bibliothèques : GNU libc  $\simeq$ 1 million de lignes
- compilateur : gcc (C, C++, ada, etc.)  $\simeq$ 1.4 million de lignes
- Internet : Apache (serveur WEB)  $\simeq$ 250 000 lignes
- applications : GIMP (retouche d'images)  $\simeq$ 700 000 lignes

## Langages inspirés du C

- compatibles : C++, Objective-C
- de syntaxe similaire : Java, C#, etc.

## Introduction

Premier programme en C

Premiers concepts en C

Repères historiques

Généralités sur le C

# Généralités sur le C

---

# Un langage impératif

**Programme** = séquence de :

- déclarations (soit  $X\dots$ ),
- instructions (actions à effectuer),

chacune terminée par un point-virgule ;

## Paradigme impératif

Les instructions sont exécutées en séquence.

Autres paradigmes : langages logiques, orientés-objets, fonctionnels, multi-paradigmes, etc.

# Un langage structuré

**Bloc** = suite d'instructions délimitée par `{` et `}`

## Structures de contrôle

Contrôlent l'exécution d'un bloc :

- conditionnelles `if`, `else`,
- boucles `while`, `for`,

Il ne faut pas “sauter” d'une instruction à une autre (`goto`).

Les blocs et structures de contrôle peuvent s'imbriquer.

# Un langage procédural

## Fonction =

- bloc d'instructions,
- déclarée une fois, **réutilisable** de nombreuses fois,
- peut prendre des arguments et retourner une valeur,
- a un effet (modification de la mémoire, affichage à l'écran),
- peut être définie dans un autre module (bibliothèques).

## Fonctions prédefinies

- bibliothèque standard (affichage, fichiers, mémoire),
- bibliothèque mathématique, etc.

Il faut les importer par une directive spéciale **#include**.

# Un langage déclaratif

**Variable** = morceau de mémoire où stocker une valeur

## Attention

Toute variable doit être **déclarée** avant d'être utilisée !

## Durée de vie d'une variable

- variable globale,
- variable locale à un bloc ou une fonction,
- mémoire dynamique.

# Un langage typé

## Typage

Les variables sont **typées**.

Le type est fixé lors de la déclaration.

Le type d'une variable détermine :

- l'ensemble de valeurs possibles (entiers, flottants, etc.),
- la quantité de mémoire à réserver (`sizeof`),
- le codage utilisé en mémoire,
- la sémantique des opérations (division `/`),
- permet de vérifier la cohérence du programme.

# Un langage typé

## Les types du C

- types entiers (`int`, `unsigned`, `char`, etc.),
- types flottant : (`double`, `float`),
- types composés : tableaux, enregistrements,
- types pointeurs : adresse des variables en mémoire,
- types de fonctions (prototypes),
- types définis par l'utilisateur.

# Un langage (assez) bas niveau

Le C permet :

- des opérations mal définies ou invalides,
- l'accès direct à la mémoire (pointeurs),
- le contournement du système de types.

## Inconvénients

- dangereux
- peu d'abstraction

## Avantages

- rapidité
- contrôle total sur la mémoire

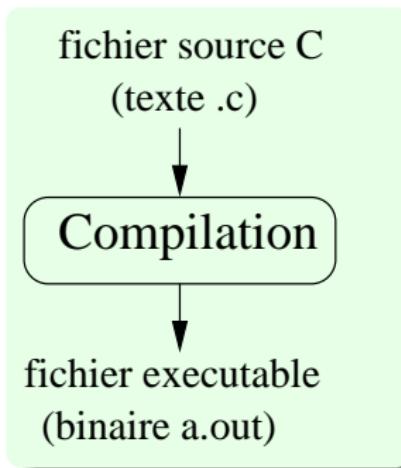
Le système est là pour protéger les autres programmes !  
(Segmentation Fault)

# Un langage compilé

**Source** = fichier .c contenant le texte du programme en C.

**Langage binaire** : seul compréhensible par l'ordinateur.

La **compilation** transforme le source en binaire.



## La compilation

Opération complexe :

- analyse syntaxique et typage,
- gestion des ressources très bas niveau (registres, adresses),
- découpage en instructions très élémentaires.

# Un langage compilé

Compilation et exécution :

- compiler une fois, exécuter une ou plusieurs fois,
- recompiler si le source change,
- recompiler pour un autre système / microprocesseur.

## Avantages de la compilation

- rapidité (exécution en langage machine),
- vérification complète du programme (syntaxe, typage),
- optimisation globale,
- liens avec d'autres langages,
- binaire autonome.

Different des langages interprétés (shell, Perl, BASIC, etc.) !

# Un langage normalisé

**Normalisation** imposée par le succès du langage C :

- disponible sur des systèmes très différents,
- chaque constructeur fournit son compilateur.

**Attention** : tout n'est pas normalisé !

- extensions non standard (dépendant du compilateur),
- bibliothèques non standard (dépendant du système),
- comportements indéfinis dans la norme.

## Avantage de la normalisation

Il est possible d'écrire des programmes portables  
avec un peu de soins !

# Premier programme en C

---

# Premier programme

```
bonjour.c
#include <stdio.h>

int main()
{
    printf("Bonjour tout le monde!\n");
    return(0);
}
```

Effet :

- affiche **Bonjour tout le monde!**,
- retourne le code 0 (tout s'est bien passé).

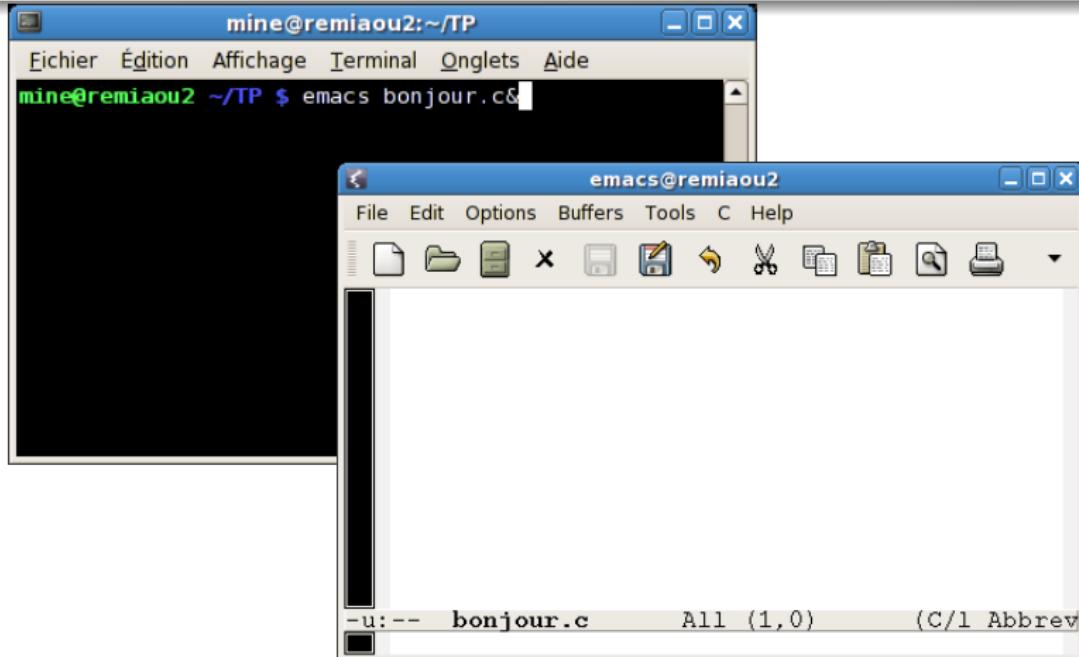
# Compilation et exécution



A screenshot of a terminal window titled "mine@remiaou2:~/TP". The window has a menu bar with "Fichier", "Édition", "Affichage", "Terminal", "Onglets", and "Aide". The main area shows a command line prompt: "mine@remiaou2 ~/TP \$ emacs bonjour.c &". The terminal window is set against a light gray background.

Lancement de l'éditeur en tâche de fond (**&**).

# Compilation et exécution



Lancement de l'éditeur en tâche de fond (**&**).

# Compilation et exécution

The screenshot shows a Linux desktop environment. In the top-left corner, there is a terminal window titled "mine@remiaou2:~/TP". The terminal has a menu bar with "Fichier", "Édition", "Affichage", "Terminal", "Onglets", and "Aide". The command "mine@remiaou2 ~/TP \$ emacs bonjour.c&" is visible in the terminal window. In the bottom-right corner, there is an Emacs editor window titled "emacs@remiaou2". The Emacs window has a menu bar with "File", "Edit", "Options", "Buffers", "Tools", "C", and "Help". Below the menu bar is a toolbar with various icons. The main buffer area contains the following C code:

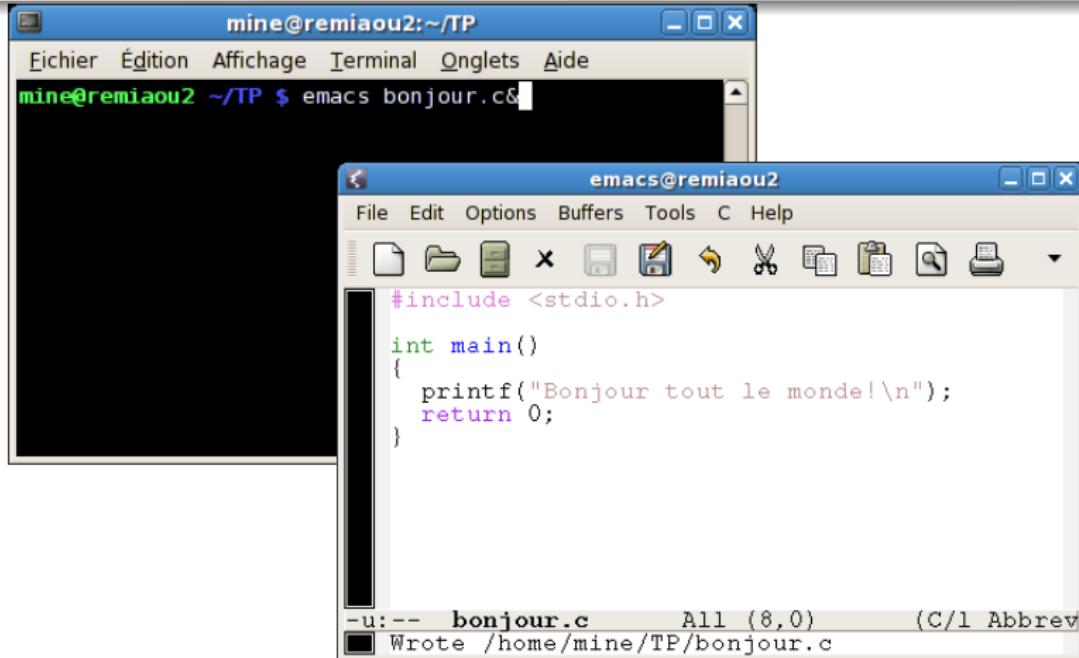
```
#include <stdio.h>

int main()
{
    printf("Bonjour tout le monde!\n");
    return 0;
}
```

The status bar at the bottom of the Emacs window displays "-u:\*\* bonjour.c All (8,0) (C/1 Abbrev)".

On tape le texte du programme.

# Compilation et exécution



Il ne faut pas oublier de sauvegarder.

# Compilation et exécution

The screenshot shows a Linux desktop environment with several windows open:

- A terminal window titled "mine@remiaou2:~/TP" containing the command "gcc bonjour.c -Wall".
- A file browser window titled "J2" showing a file named "bonjour.c".
- An Emacs editor window showing the source code of "bonjour.c".
- A small window at the bottom showing the status of the Emacs session.

The terminal output shows the compilation command being run:

```
mine@remiaou2 ~/TP $ gcc bonjour.c -Wall
```

The Emacs status bar at the bottom indicates:

```
-u:-- bonjour.c All (8,0) (C/l Abbrev)
Wrote /home/mine/TP/bonjour.c
```

Lancement de la compilation avec **gcc**.

# Compilation et exécution

The screenshot shows a Linux desktop environment with several windows open:

- A terminal window titled "mine@remiaou2:~/TP" containing the following command history:

```
mine@remiaou2 ~/TP $ emacs bonjour.c&
[2] 32667
mine@remiaou2 ~/TP $ gcc bonjour.c -Wall
mine@remiaou2 ~/TP $
```
- A file browser window titled "J2" showing a list of files.
- An Emacs editor window showing the source code of "bonjour.c":

```
bonjour()
{
    printf("Hello world!\n");
}
```
- A small window at the bottom showing the status of the Emacs session:

```
-u:-- bonjour.c      All (8,0)      (C/l Abbrev)
Wrote /home/mine/TP/bonjour.c
```

Si le compilateur ne dit rien, tout s'est bien passé.  
Un fichier **a.out** a été créé.

# Compilation et exécution

The screenshot shows a Linux desktop environment with a terminal window and an Emacs editor.

The terminal window (mine@remiaou2:~/TP) displays the following command sequence:

```
mine@remiaou2 ~/TP $ emacs bonjour.c&
[2] 32667
mine@remiaou2 ~/TP $ gcc bonjour.c -Wall
mine@remiaou2 ~/TP $ ./a.out
```

The Emacs editor window (bonjour.c) shows the source code:

```
e monde!\n");
```

A status bar at the bottom of the Emacs window indicates:

```
-u:-- bonjour.c      All (8,0)      (C/l Abbrev)
Wrote /home/mine/TP/bonjour.c
```

Lancement de l'exécutable.

# Compilation et exécution

The screenshot shows a Linux desktop environment. On the left, a terminal window titled "mine@remiaou2:~/TP" displays a session of the terminal. The user has run the command "emacs bonjour.c&" which starts an emacs editor in the background. Then they run "gcc bonjour.c -Wall" to compile the program. Finally, they run "./a.out" to execute it, producing the output "Bonjour tout le monde!". On the right, a file browser window titled "J2" is open, showing a list of files including "bonjour.c". At the bottom, a status bar in the terminal window shows "-u:-- bonjour.c All (8,0) (C/l Abbrev)" and "Wrote /home/mine/TP/bonjour.c".

```
mine@remiaou2:~/TP $ emacs bonjour.c&
[2] 32667
mine@remiaou2:~/TP $ gcc bonjour.c -Wall
mine@remiaou2:~/TP $ ./a.out
Bonjour tout le monde!
mine@remiaou2:~/TP $
```

-u:-- bonjour.c All (8,0) (C/l Abbrev)  
Wrote /home/mine/TP/bonjour.c

Le programme s'exécute et rend la main.

# Anatomie de bonjour.c

bonjour.c

```
#include <stdio.h>

int main()
{
    printf("Bonjour tout le monde!\n");
    return(0);
}
```

Tout programme C doit contenir une fonction appelée **main**.  
L'exécution commence au début de **main**.

# Anatomie de bonjour.c

bonjour.c

```
#include <stdio.h>

int main()
{
    printf("Bonjour tout le monde!\n");
    return(0);
}
```

Par convention, la fonction `main` renvoie un code de retour :

- il est de type `int` (entier),
- la convention est de retourner `0` si tout se passe bien,
- les parenthèses de `return` sont facultatives,
- le code de retour est exploitable depuis le shell.

# Anatomie de bonjour.c

bonjour.c

```
#include <stdio.h>

int main()
{
    printf("Bonjour tout le monde!\n");
    return(0);
}
```

La fonction **printf** permet d'écrire sur l'écran.

- elle fait partie de la bibliothèque C standard,
- elle doit être importée depuis l'en-tête **stdio.h**.

# Anatomie de bonjour.c

bonjour.c

```
#include <stdio.h>

int main()
{
    printf("Bonjour tout le monde!\n");
    return(0);
}
```

printf prend en argument une chaîne de caractères :

- tapée entre guillemets " ,
- \ sert à entrer des caractères spéciaux :  
\n signifie “retour à la ligne”.

# Exemple d'erreur

bonjour.c faux

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Bonjour tout le monde!\n")
6     return(0);
7 }
```

# Exemple d'erreur

bonjour.c faux

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Bonjour tout le monde!\n")
6     return(0);
7 }
```

## Résultat

```
$ gcc bonjour.c -Wall
bonjour.c: In function 'main':
bonjour.c:6: error: expected ';' before 'return'
bonjour.c:7: warning : control reaches end of ...
```

Il manque un ;. Aucun a.out n'est généré.

## Exemple d'avertissement

bonjour.c faux

```
1 int main()
2 {
3     printf("Bonjour tout le monde!\n");
4     return(0);
5 }
```

### Résultat

```
$ gcc bonjour.c -Wall
bonjour.c: In function 'main':
bonjour.c:3: warning: implicit declaration of function
'printf'
```

Il manque un `#include <stdio.h>`.

C'est un avertissement non fatal généré par `-Wall`.

## Les options `-Wall` et `-Wextra`

`-Wall` attire l'attention, entre autres, sur :

- les ouboris d'imports `#include`,
- les ambiguïtés syntaxiques courantes,
- les incohérences de types.

La norme est très laxiste ne considère pas ces points comme des erreurs !

`-Wextra` ajoute des avertissements supplémentaires.

Toujours compiler avec `-Wall` et `-Wextra` !

# Espacement

L'espacement et les sauts de lignes sont libres.

## Exemple correct mais illisible

```
# include <stdio.h>
int main()
{
    printf
    ("toto\n"
 );return(0)    ;}
```

## Exceptions

- **#include <stdio.h>** doit être sur une seule ligne,
- les sauts de ligne comptent dans les chaînes de caractères.

# Commentaires

**Commentaires** : tout ce qui est entre /\* et \*/ est ignoré.

## Exemple commenté

```
#include <stdio.h> /* pour avoir printf */

/* la fonction principale
*/
int main(/* rien ici */)
{
    printf("toto\n");
    return(0); /* OK */
}
```

**Conseils** : - indentez votre code (tabulation sous Emacs),  
- commentez votre code.

# Variables et expressions

---

# Déclaration de variables

Les variables doivent être déclarées avant d'être utilisées.

## Syntaxe

type nom;

### Exemple

```
int main()
{
    int x;
    double y;
    x = 12;
    y = x/3.0;
    return 0;
}
```

# Durée de vie et visibilité

## Variables locales :

- déclarées **en début de bloc** (sauf C99),
- créées quand le programme “entre” dans le bloc,
- détruites en fin de bloc,
- masquent les autres variables de même nom.

### Exemple (fragment)

```
{ int x;
    { int y;
        int x;
        y=x+1; /* il s'agit du dernier x */
    }
    y=x+1; /* erreur: y inconnu */
    int z; /* erreur: pas en début de bloc */
}
```

# Durée de vie et visibilité

## Variables globales :

- déclarées en dehors de tout bloc, fonction,
- existent toujours,
- surtout utiles quand on a plusieurs fonctions (cours suivant).

### Exemple

```
#include <stdio.h>
int x;
int main()
{
    x=12;
    return x;
}
```

# Identificateurs

**Identificateur** = nom de variable

- suite de lettres, chiffres ou soulignés : a-z A-Z 0-9 \_
- commence par une lettre ou souligné,
- pas d'accent, d'espace, de ponctuation,
- sensible à la casse.

## Exemples

```
int i;  
int I; /* différent de i */  
int Mon42emeEntier;  
double nombre_flottant2;
```

# Noms réservés

Certains noms sont réservés par le langage.

Une variable ne doit pas porter un nom réservé.

## Noms réservés

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

Tout nom commençant par \_ suivi d'une majuscule ou de \_

Par contre, on peut utiliser : Auto, break2, \_case.

# Types de base

## Types courants

<code>int</code>	entiers machine
<code>char</code>	caractères, également entiers
<code>double</code>	nombres flottants double précision

## Autres types entiers et flottants :

<code>unsigned</code>		entiers positifs
<code>short</code>	<code>unsigned short</code>	petits entiers
<code>long</code>	<code>unsigned long</code>	gros entiers
<code>long long</code>	<code>unsigned long long</code>	très gros entiers
<code>signed char</code>	<code>unsigned char</code>	très petits entiers
<code>float</code>		flottants simple précision

# Types de base

Pourquoi tous ces types ?

- compromis entre capacité et occupation mémoire,
- **interprétation dépendante de la machine !**

## Exemples courants

int	32-bits : [-2147483648 ; 2147483647]
unsigned	32-bits : [0 ; 4294967295]
char	8-bits : [-128 ; 127] ou [0 ; 255]
double	64-bits : magnitude $10^{-308}$ à $10^{308}$ , 16 chiffres significatifs

# Affectations et expressions

**Affectation** = modification de la valeur d'une variable

## Syntaxe

```
variable = expression;
```

**Expression** =

- constantes entières : 2, -45, ‘a’ (=141),
- constantes flottantes : 3.45, -4.5e-12 ( $=4.5 \times 10^{-12}$ ),
- variables,
- opérateurs,
- parenthèses : (, ).

# Opérateurs arithmétiques

## Opérateurs courants

- + addition
- soustraction (binaire) ou négation (unaire)
- \*
- / division (entière ou flottante)
- % reste de la division (entière)

## Priorités

Comme en mathématiques :

- \*, /, % prioritaires sur + et -,
- si même priorité, on évalue de gauche à droite,
- en cas de doute, **mettre des parenthèses.**

# Sémantique de l'affectation

Deux étapes :

- **évaluation** de l'expression en un entier ou flottant,  
utilise la valeur courante des variables,
- stockage du résultat dans la variable destination.

## Exemple

```
x = 2+3*4; /* ici, x vaut 14 */  
y = 2*(x+1); /* ici, y vaut 30 */  
x = x - 1; /* ici, x vaut 13 */
```

# Initialisation des variables

## Attention !

Le contenu d'une variable est indéfini avant la première affectation.

### Exemple

```
int x;  
int y;  
/* x et y sont aléatoires */  
y = 100 / x; /* opération dangereuse */
```

### Raccourcis : déclaration et initialisation combinée

```
int x = 12;  
int y = 2+3*x;
```

## Affichage sur écran

---

# La fonction printf

printf permet d'afficher :

- du texte,
- la valeur d'expressions entières ou flottantes.

## Arguments de printf

printf prend un nombre arbitraire d'arguments :

- 1er argument : texte à afficher,
- arguments suivants : expressions à évaluer.

Les arguments sont séparés par une virgule ,

## Effet

Dans le texte, chaque %x est remplacé par la valeur d'un argument.

# Utilisation de printf

## Exemple

```
#include <stdio.h>
int main()
{
    int x = 12;
    printf("x = %i\n",x);
    return 0;
}
```

## Résultat :

x = 12

# Utilisation de printf

## Exemple

```
#include <stdio.h>
int main()
{
    int x = 12;
    printf("1/3 vaut %f mais 3x vaut %i\n\n", 1.0/3.0, 3*x);
    return 0;
}
```

## Résultat :

1/3 vaut 0.333333 mais 3x vaut 36

# Caractères magiques dans printf

## Caractères magiques \ et %

\n passe à la ligne suivante

\" affiche "

\\ affiche \

%i affiche un entier passé en argument

%c affiche un caractère passé en argument

%f affiche un flottant passé en argument

%% affiche %

## Attention :

- il faut autant d'arguments supplémentaires que de %x,
- l'argument doit être entier pour %i et %c, flottant pour %f,
- l'option -Wall vérifie cela pour vous !

# Conditionnelles

---

# La construction if

## Syntaxe

```
if (expression) { instructions }
```

### Effet :

- l'expression est évaluée,
- le bloc suivant n'est exécuté que si l'expression est vraie.

### Raccourcis : si il n'y a qu'une instruction :

```
if (expression) instruction;
```

Les parenthèses sont par contre obligatoires !

# Expressions booléennes

On peut comparer la valeur de deux expressions arithmétiques :

## Opérateurs de comparaison

- ==** égal
- !=** différent
- >** strictement supérieur
- <** strictement inférieur
- >=** supérieur ou égal
- <=** inférieur ou égal

## Exemple

```
if (x>2*y) { x=2*y; y=0; }
```

La priorité des opérateurs de comparaison est plus faible que celle des opérateurs arithmétiques.

# Opérateurs booléens

On peut combiner la valeur de vérité d'expressions booléennes :

## Opérateurs booléens

	ou logique	(binaire)
&&	et logique	(binaire)
!	négation logique	(unaire)

## Exemple

```
if ((x>0 && y>0) || (x<0 && y<0)) signe_xy=1;
```

**Conseil** : utilisez des parenthèses pour ne pas vous tromper dans les priorités.

# La construction if else

## Syntaxe

```
if (expression) { instructions1 }  
else { instructions2 }
```

## Effet :

- l'expression est évaluée,
- le premier bloc est exécuté si l'expression est vraie,
- le deuxième bloc est exécuté si l'expression est fausse.

# Exemple de test

## Exemple

```
if (age>99) printf("vous êtes trop vieux!\n");
else {
    if (age<18) printf("interdit aux mineurs!\n");
    else printf("vous avez %i ans\n",age);
}
```

# Opérateurs à ne pas confondre

## Ne pas confondre

l'opérateur d'affectation = et l'opérateur de comparaison ==

### Programme C "correct"

```
1 int main()
2 {
3     int x;
4     x==0;
5     if (x=0) {}
6     return 0;
7 }
```

**Conseil :** compiler avec -Wall -Wextra

y.c: In function ‘main’:

y.c:4: warning: statement with no effect

y.c:5: warning: suggest parentheses around assignment used as tr

## Boucles while

---

# La construction while

## Syntaxe

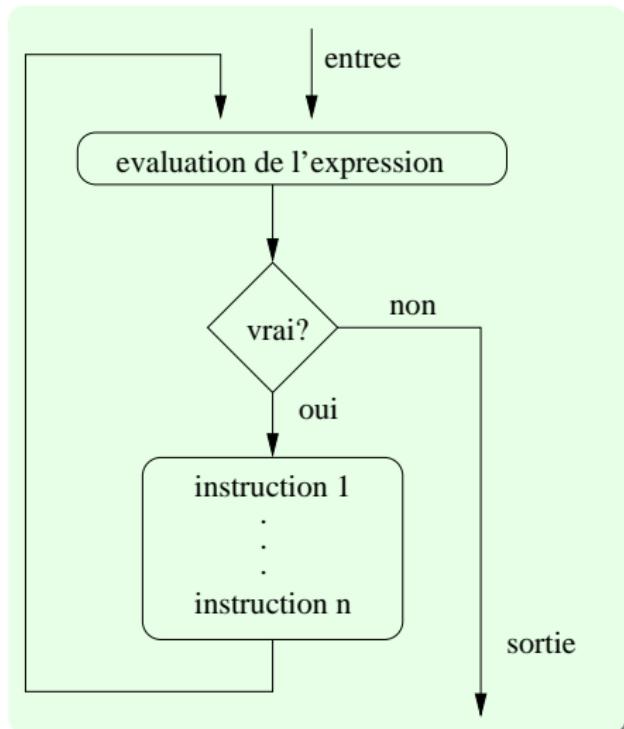
```
while (expression) { instructions }
```

**Effet :** tant que *expression* est vraie, le bloc est exécuté.

## Notes :

- si la condition est initialement fausse, le bloc n'est jamais exécuté,
- la condition est re-testée après chaque "tour" de boucle,
- les {} sont facultatives. mais les () obligatoires.

# Déroulement d'une boucle



# Exemple de boucle

## Exemple

```
#include <stdio.h>
int main()
{
    int pommes = 10;
    while (pommes > 0) {
        printf("j'ai %i pommes dans ma pochette\n",
               pommes);
        pommes = pommes-1;
    }
    printf("je n'ai plus de pommes!\n");
    return 0;
}
```

# Initiation au C

## Cours n°2

Antoine Miné<sup>1</sup> Ozan Caglayan<sup>2</sup>

<sup>1</sup>École Normale Supérieure

<sup>2</sup>Université Galatasaray

22 février 2007



# Plan du cours

- compléments sur les déclarations et expressions,
- les fonctions.

# Compléments sur les déclarations et expressions

---

# Déclarations multiples

Déclaration simultanée de plusieurs variables de même type.

## Syntaxe

```
type var1=expr1, var2=expr2, ..., varN=exprN;
```

- les variables sont séparées par des **virgules**,
- chacune est optionnellement initialisée grâce à **=**
- l'ordre de création et d'évaluation des initialiseurs est inconnu.

## Exemples

```
int x,y=2,z;      /* seul y est initialisé */  
int X=3, Y=2*X; /* invalide */
```

# Affectations combinées

Opération et affectation combinées *op=* :

## Opérateurs combinées

var <b><i>+=</i></b> expr;	équivalent à	var = var + (expr);
var <b><i>-=</i></b> expr;	"	var = var - (expr);
var <b><i>*=</i></b> expr;	"	var = var * (expr);
var <b><i>/=</i></b> expr;	"	var = var / (expr);
var <b><i>%=</i></b> expr;	"	var = var % (expr);

## Attention :

- pas d'espace entre l'opérateur *op* et le égal =,
- $x *= y+1$  est équivalent à  $x = x*(y+1)$   
et pas  $x = x*y+1$ .

## Résultat retourné par une affectation

Comme `+`, `-`, etc. toute affectation retourne une valeur :

- `var = expr;` retourne la valeur de `expr`,
- `var op= expr;` retourne la nouvelle valeur de `var`,
- l'affectation associe à droite.

### Exemples

```
x = (y = 12*z+3);    met la valeur de 12*z+3 dans x et y
x = y = 12*z+3;          "      "
x = (y+=2)+1;           met y+2 dans y et y+3 dans x
```

# Incrémantion et décrémentation

## Opérateurs

- `var++`; équivalent à `var = var + 1;`
  - `var--`; équivalent à `var = var - 1;`
- 
- pas d'espace entre les deux symboles,
  - espace possible entre la variable et l'opérateur.

L'opérateur peut se placer **avant** ou **après** la variable :

- même effet sur `var` mais valeur retournée différente,
- `var++` et `var--` renvoient la valeur de `var` **avant** l'affectation,
- `++var` et `--var` renvoient la valeur de `var` **après** l'affectation.

# Incrémantation et décrémentation

## Utilisation courante

```
int x = 0;  
while (x<10) {  
    printf("%i\n",x);  
    x++;  
}
```

## Exemples compliqués

```
x = 12;  
y = x++; /* ici, y=12 et x=13 */  
x = --y; /* ici, y=11 et x=11 */  
z = x++ + ++y; /* ici, z=23 */
```

**Question :** que signifie C++ ?

# Conflits d'effets de bord

Une unique instruction (terminée par ;) peut :

- lire plusieurs variables,
- modifier plusieurs variables.

L'ordre de ces opérations est indéfini !

## Conséquence

Une instruction ne doit pas :

- modifier deux fois la même variable,
- lire et modifier une même variable.

# Conflits d'effets de bord

## Exemples incorrects

```
x = (y=2) + (y=3); /* invalide: y modifié deux fois */  
x = (x=2) + 1;      /* invalide: x modifié deux fois */  
x = (y=2) + y;      /* invalide: y lu et modifié */  
y = x++ + x;        /* invalide: x lu et modifié */
```

**exception** : dans var=expr, var peut apparaître dans expr

```
x = x+1; /* correct */
```

## Conclusion

Se limiter à **une** affectation par instruction.

# Le “type” booléen

**booléen** = valeur de vérité : vrai ou faux.

Le C n'a pas de type booléen dédié :

- tout entier ou flottant **non nul** signifie **vrai**,
- **0** ou **0.0** signifie **faux**.

Les comparaisons et opérateurs booléens `== != > >= < <= && ||` renvoient toujours un entier :

- **0** pour **faux**,
- **1** pour **vrai**.

# Exemples

## Raccourcis classiques

`if (x) ...` est équivalent à `if (x!=0) ...`  
`if (!x) ...` est équivalent à `if (x==0) ...`

## Exemples plus complexes :

- `expr1 && expr2` est équivalent à `(expr1!=0)*(expr2!=0)`,
- `expr!=0` est équivalent à `!!expr`.

# Opérateurs booléens et court-circuits

expr1 **&&** expr2 et expr1 **||** expr2 :

- évaluent d'abord expr1,
- évaluent expr2 **uniquement si nécessaire** :
  - pour **&&**, si expr1 est fausse, expr2 n'est pas évaluée,
  - pour **||**, si expr1 est vraie, expr2 n'est pas évaluée.

## Application

```
if ( x!=0 && y/x>100 ) ...  
ne provoque jamais de division par zéro.
```

# Opérateur ternaire

**Opérateur d'alternative** ternaire :

## Syntaxe

```
expr1 ? expr2 : expr3
```

**Effet :**

- évalue expr1,
- si expr1 est vrai, évalue et renvoie expr2,
- si expr1 est fausse, évalue et renvoie expr3,
- un seul parmi expr2 et expr3 est évalué.

## Exemple

```
x = y ? 10/y : 99999;  
évite encore une division par zéro.
```

# Intermède

Que fait le programme suivant ?

```
je_compte_mal.c
#include <stdio.h>
int main()
{
    int x = 1;
    while (x<100) {
        if (x==2) printf("je n'aime pas 2\n");
        else      printf("%i\n",x);
        x++;
    }
    return(0);
}
```

# Solution

## L'erreur

```
while (x<100) {  
    if (x=2) printf("je n'aime pas 2\n");  
    else      printf("%i\n",x);
```

On a confondu = et == !

# Solution

## L'erreur

```
while (x<100) {  
    if (x=2) printf("je n'aime pas 2\n");  
    else      printf("%i\n",x);
```

On a confondu = et == !

À chaque itération :

- on met 2 dans x,

# Solution

## L'erreur

```
while (x<100) {  
    if (x=2) printf("je n'aime pas 2\n");  
    else      printf("%i\n",x);
```

On a confondu = et == !

À chaque itération :

- on met 2 dans x,
- comme 2 est vrai, on affiche toujours je n'aime pas 2,
- la branche else n'est jamais prise,

# Solution

## L'erreur

```
while (x<100) {  
    if (x=2) printf("je n'aime pas 2\n");  
    else      printf("%i\n",x);
```

On a confondu = et == !

À chaque itération :

- on met 2 dans x,
- comme 2 est vrai, on affiche toujours je n'aime pas 2,
- la branche else n'est jamais prise,
- en plus, on boucle indéfiniment !

# Les fonctions

---

# Définition de fonctions

**Fonction :** bloc d'instruction, nommé, avec arguments et valeur de retour.

## Syntaxe

```
type nom(type1 arg1, type2 arg2, ..., typeN argN)
{
    instructions
}
```

# Définition de fonctions

**Fonction :** bloc d'instruction, nommé, avec arguments et valeur de retour.

## Syntaxe

```
type nom(type1 arg1, type2 arg2, ..., typeN argN)
{
    instructions
}
```

Nom de la fonction.

N'importe quel identificateur non déjà utilisé convient.

# Définition de fonctions

**Fonction :** bloc d'instruction, nommé, avec arguments et valeur de retour.

## Syntaxe

```
type nom(type1 arg1, type2 arg2, ..., typeN argN)
{
    instructions
}
```

Nom des arguments formels de la fonction,  
séparés par des virgules ,

# Définition de fonctions

**Fonction :** bloc d'instruction, nommé, avec arguments et valeur de retour.

## Syntaxe

```
type nom(type1 arg1, type2 arg2, ..., typeN argN)
{
    instructions
}
```

Type de chaque argument formel.

# Définition de fonctions

**Fonction :** bloc d'instruction, nommé, avec arguments et valeur de retour.

## Syntaxe

```
type nom(type1 arg1, type2 arg2, ..., typeN argN)
{
    instructions
}
```

Type de la valeur renvoyée par la fonction,  
ou **void** si la fonction ne renvoie pas de valeur.

# Définition de fonctions

**Fonction :** bloc d'instruction, nommé, avec arguments et valeur de retour.

## Syntaxe

```
type nom(type1 arg1, type2 arg2, ..., typeN argN)
{
    instructions
}
```

Instructions à exécuter à chaque appel.

Les accolades { } sont obligatoires.

# Exemple de fonction sans valeur de retour

## Table de multiplication

```
#include <stdio.h>
void table(int x,int max)
{
    int y=1;
    while (y<max) {
        printf("%i x %i = %i\n",x,y,x*y);
        y++;
    }
}
int main()
{
    table(3,10);
    table(4,10);
    return 0;
}
```

# Appel de fonction sans valeur de retour

**Appel de fonction** : instruction terminée par ;

## Syntaxe

```
nom(expr1,expr2,...,exprN);
```

**Effet :**

- évalue les expressions expr1 à exprN,
- crée des variables locales arg1 à argN,
- initialise chaque argi à la valeur de expr<sub>i</sub> (conversion implicite éventuelle),
- exécute les instructions de la fonction,
- détruit les variables locales arg1 à argN.

# Position des définitions

Les définitions de fonctions apparaissent dans l'**“espace global”** avec :

- les directive `#include`,
- les déclarations de variables globales.

Chaque fonction définit un **“espace local”** contenant :

- des déclarations de variables locales,
- des instructions,
- des sous blocs (espace locaux imbriqués).

## Attention

Il n'existe pas de ‘fonctions locales’ en C

# Déclarations en avance

## Attention

Toute fonction doit être déclarée avant d'être appelée.

On peut déclarer une fonction sans la définir, par un **prototype**.

## Syntaxe

```
type nom(type1 arg1, type2 arg2, ..., typeN argN);
```

- le 'corps' de la fonction n'est pas fourni et remplacé par ;
- il faut quand même préciser tous les types,
- le prototype se place dans l'"espace global".

# Exemple de déclaration en avance

## Table de multiplication

```
#include <stdio.h>

void table(int x,int max);

int main()
{
    int i = 2;
    while (i<10) {
        table(i,10);
        i++;
    }
    return 0;
}

void table(int x,int max)
{
    int y=1;
    while (y<max) {
        printf("%i x %i = %i\n",
               x,y,x*y);
        y++;
    }
}
```

# Variables accessibles

Une fonction peut accéder :

- aux variables globales déjà déclarées,
- aux fonctions déjà déclarées,
- à **ses** variables locales,
- à **ses** arguments formels.

## Attention

Une fonction ne peut pas accéder aux variables locales d'une autre fonction !

# Vie et mort des variables locales

Chaque appel de fonction crée son lot de variables locales.

## Variables locales de l'appelé

- créées lors de l'appel,
- détruites quand la fonction se termine.

## Variables locales de l'appelant

- masquées durant l'appel,
- mais gardent leur valeur.

⇒ les valeurs sont perdues entre deux appels successifs,  
⇒ une fonction qui s'appelle elle-même a plusieurs copies de ses variables.

# Exemple

## Programme

```
1 #include <stdio.h>
2 void compte(int x,int y)
3 {
4     if (x>y) compte(y,x);
5     while (x<=y) {
6         printf("%i ",x);
7         x++;
8     }
9 }
10 int main()
11 {
12     int x=0;
13     compte(x+4,x+2);
14 }
```

## Résultat

2 3 4

# Exemple

## Programme

```
1 #include <stdio.h>
2 void compte(int x,int y)
3 {
4     if (x>y) compte(y,x);
5     while (x<=y) {
6         printf("%i ",x);
7         x++;
8     }
9 }
10 int main()
11 {
12     int x=0;
13     compte(x+4,x+2);
14 }
```

## État de la mémoire

main

l. 13  
x = 0

## Résultat

# Exemple

## Programme

```
1 #include <stdio.h>
2 void compte(int x,int y)
3 {
4     if (x>y) compte(y,x);
5     while (x<=y) {
6         printf("%i ",x);
7         x++;
8     }
9 }
10 int main()
11 {
12     int x=0;
13     compte(x+4,x+2);
14 }
```

## État de la mémoire

main

l. 13  
x = 0

compte

l. 4  
x = 4  
y = 2

## Résultat

# Exemple

## Programme

```
1 #include <stdio.h>
2 void compte(int x,int y)
3 {
4     if (x>y) compte(y,x);
5     while (x<=y) {
6         printf("%i ",x);
7         x++;
8     }
9 }
10 int main()
11 {
12     int x=0;
13     compte(x+4,x+2);
14 }
```

## État de la mémoire

main

l. 13  
x = 0

compte

l. 4  
x = 4  
y = 2

compte

l. 4  
x = 2  
y = 4

## Résultat

# Exemple

## Programme

```
1 #include <stdio.h>
2 void compte(int x,int y)
3 {
4     if (x>y) compte(y,x);
5     while (x<=y) {
6         printf("%i ",x);
7         x++;
8     }
9 }
10 int main()
11 {
12     int x=0;
13     compte(x+4,x+2);
14 }
```

## État de la mémoire

main

l. 13  
x = 0

compte

l. 4  
x = 4  
y = 2

compte

l. 8  
x = 3  
y = 4

## Résultat

2

# Exemple

## Programme

```
1 #include <stdio.h>
2 void compte(int x,int y)
3 {
4     if (x>y) compte(y,x);
5     while (x<=y) {
6         printf("%i ",x);
7         x++;
8     }
9 }
10 int main()
11 {
12     int x=0;
13     compte(x+4,x+2);
14 }
```

## État de la mémoire

main

l. 13  
x = 0

compte

l. 4  
x = 4  
y = 2

compte

l. 8  
x = 4  
y = 4

## Résultat

2 3

# Exemple

## Programme

```
1 #include <stdio.h>
2 void compte(int x,int y)
3 {
4     if (x>y) compte(y,x);
5     while (x<=y) {
6         printf("%i ",x);
7         x++;
8     }
9 }
10 int main()
11 {
12     int x=0;
13     compte(x+4,x+2);
14 }
```

## État de la mémoire

main

I. 13  
x = 0

compte

I. 4  
x = 4  
y = 2

compte

I. 8  
x = 5  
y = 4

## Résultat

2 3 4

# Exemple

## Programme

```
1 #include <stdio.h>
2 void compte(int x,int y)
3 {
4     if (x>y) compte(y,x);
5     while (x<=y) {
6         printf("%i ",x);
7         x++;
8     }
9 }
10 int main()
11 {
12     int x=0;
13     compte(x+4,x+2);
14 }
```

## État de la mémoire

main

l. 13  
x = 0

compte

l. 5  
x = 4  
y = 2

## Résultat

2 3 4

# Exemple

## Programme

```
1 #include <stdio.h>
2 void compte(int x,int y)
3 {
4     if (x>y) compte(y,x);
5     while (x<=y) {
6         printf("%i ",x);
7         x++;
8     }
9 }
10 int main()
11 {
12     int x=0;
13     compte(x+4,x+2);
14 }
```

## État de la mémoire

main

l. 14  
x = 0

## Résultat

2 3 4

# Fonctions avec valeur de retour

Si type n'est pas void, la fonction **doit** retourner une valeur.

Syntaxe  
`return expr;`

**Effet :**

- évalue expr,
- quitte immédiatement la fonction,
- renvoie la valeur de expr à l'appelant.

# Exemple

## Nombres premiers

```
int est_premier(int n)
{
    int i = 2;
    while (i<n) {
        if ( !(n%i) ) return 0;
        i++;
    }
    return 1;
}
```

Renvoie 1 si et seulement si n est premier.

# Utilisation de la valeur de retour

On peut appeler la fonction dans n'importe quelle expression.

## Affichage des nombre premiers

```
int i=2;
while (i<1000) {
    if (est_premier(i)) printf("%i\n",i);
    i++;
}
```

## Comptage des nombre premiers

```
int i=2, n=0;
while (i<1000)
    n = n + est_premier(i++);
printf("%i\n",n);
```

(On peut aussi ignorer la valeur de retour...)

# Notes sur return

## Notes :

- il peut y avoir plusieurs `return`,
- dans une fonction `void`,  
`return;` quitte immédiatement la fonction,
- dans une fonction non `void`,  
on doit toujours sortir de la fonction par `return expr;`

# Conflits d'effets de bord

## Attention

L'ordre d'évaluation dans une expression est indéterminé.

Cela peut être gênant si l'appel de fonction a un effet :

- affichage sur l'écran,
- modification d'une variable globale,
- modification d'un fichier, etc.

## Exemple dangereux

```
x = printf("a") + printf("b");
```

Affiche soit ab soit ba.

# Conflits d'effets de bord

## Exemple dangereux

```
int a = 0;

int compte()
{ return ++a; }

int main()
{
    printf("%i\n", compte() + a);
    return 0;
}
```

Cet exemple illustre aussi l'intérêt des variables globales pour :

- partager des variables entre fonctions,
- garder une valeur entre deux appels.

# Initiation au C

## Cours n°3

Antoine Miné<sup>1</sup> Ozan Caglayan<sup>2</sup>

<sup>1</sup>École Normale Supérieure

<sup>2</sup>Université Galatasaray

19 février 2015



# Plan du cours

- un peu plus sur les boucles : `for`, `do`, `break`,
- les tableaux,
- les constantes symboliques.

**Les boucles**  
Les tableaux  
Les constantes symboliques

Boucles while et do  
Boucles for  
break et continue

## Les boucles

---

# Boucles while (rappels)

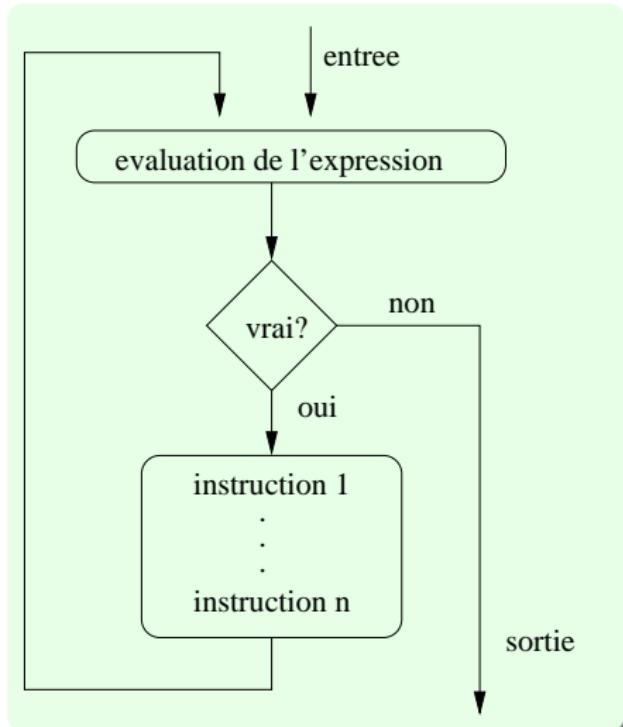
## Syntaxe

```
while (expression) { instructions }
```

## Effet :

- tant que *expression* est vraie, le bloc est exécuté,
- si la condition est initialement fausse,  
le bloc n'est **jamais** exécuté,
- la condition est re-testée **après** chaque "tour" de boucle.

# Déroulement d'une boucle while (rappels)



## Boucles do while

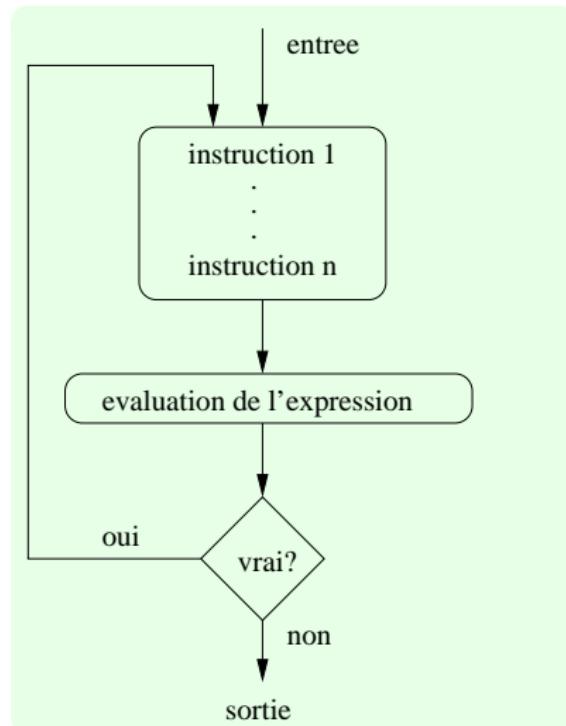
### Syntaxe

```
do { instructions } while (expression);
```

### Effet :

- le bloc est exécuté tant que *expression* est vraie,
- le bloc est **toujours** exécuté au moins une fois,  
même si la condition est initialement fausse,
- la condition est testée **après** chaque “tour” de boucle.

## Déroulement d'une boucle do while



# Boucles for

## Syntaxe

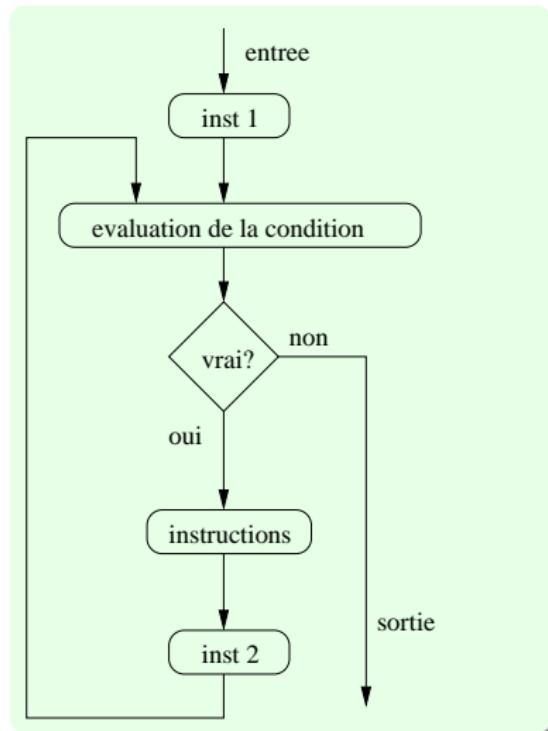
```
for (inst1; condition; inst2) { instructions }
```

- *inst1* et *inst2* : instructions simples,
- *condition* : expression booléenne,
- *instructions* : bloc d'instructions.

**Effet :** “boucle while avec initialisation”

- *inst1* est d'abord exécuté, **une seule fois**,
- tant que *condition* est vraie, exécute *bloc puis inst2*,
- la *condition* est testée :
  - après *inst1*,
  - après *inst2* à la fin de chaque “tour” de boucle.

# Déroulement d'une boucle for



## Exemples de boucle for

compte.c

```
int i;  
for (i=0;i<15;i++) printf("%i ",i);
```

Résultat : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

compte\_a\_rebours.c

```
int i;  
for (i=5;i>=0;i--) printf("%i ",i);
```

Résultat : 5 4 3 2 1 0

# Notes syntaxiques

## **inst1** et **inst2** :

- peuvent être **vides** ⇒ pas d'action,
- peuvent contenir **plusieurs** instructions,  
séparées par des virgules ,

## **condition** :

- peut être **vide** ⇒ boucle infinie.

## Extension C99 et C++ :

- **inst1** peut contenir une déclaration de variables
  - for (int i=0;i<10;i++)elles sont détruites en sortie de boucle.

## Exemples avancés de boucles for

boucle\_infinie.c

```
for (;;) printf("Je boucle\n");
```

table\_ASCII.c

```
int col,lin,c;
for ( lin=0, c=32; lin<6; lin++ ) {
    for ( col=0; col<16; col++, c++ )
        printf("%c ",c);
    printf("\n");
}
```

# L'instruction break

break sort immédiatement de la boucle la plus imbriquée (for, do ou while).

## Exemple

```
for (x=2;x<y;x++) {  
    if ( ! (y%x) ) {  
        printf("%i divise %i\n",x,y);  
        break;  
    }  
}  
/* on arrive ici après break */
```

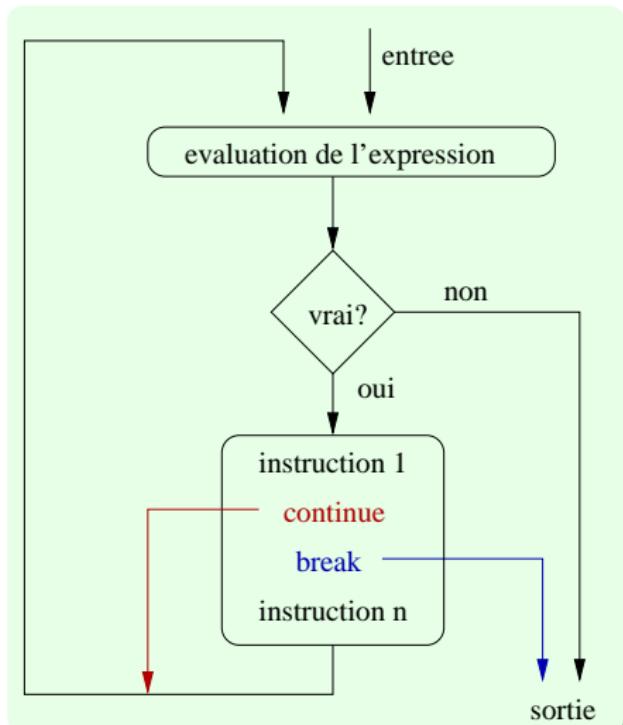
# L'instruction continue

**continue** saute à la prochaine itération de la boucle.  
La condition d'arrêt est testée.

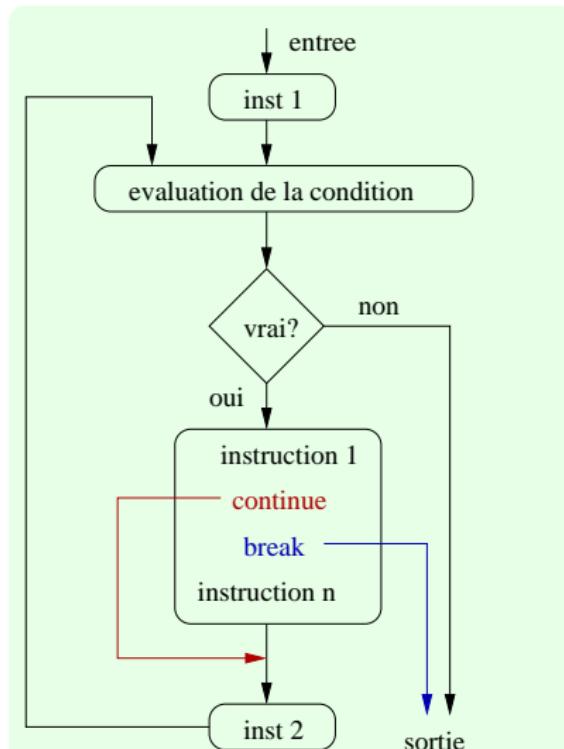
## Exemple

```
double x, y=10.0;
for (x=0;x<100;x++) {
    if (x==y) continue;
    printf("%f\n",1.0/(x-y));
}
```

## Effet sur une boucle while



## Effet sur une boucle for



## Les tableaux

---

# Déclarations de tableaux

**Tableau** = séquence de 'cases mémoires' de même type.

## Déclaration d'une variable-tableau

```
type var [taille];
```

## Exemples de déclaration

```
int      tab[10];           /* 10 entiers */  
double  Dalmatiens[10*10+1]; /* 101 flottants */
```

## Taille :

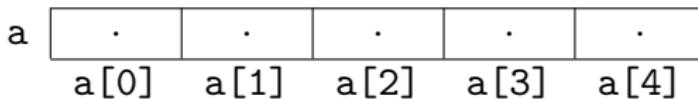
- en C ANSI, taille doit être une **expression constante** i.e., qui ne fait pas intervenir de variable,
- cette limitation est supprimée en C99 pour les variables locales.

## Accès dans un tableau

Chaque case contient une valeur indépendante.

Les cases sont indiquées **de 0 à taille-1**.

**Exemple :** int a[5] ;



Accès dans un tableau

tab [expr]

où

- tab est une variable de type tableau,
- expr est une expression entière.

# Accès dans un tableau

tab[expr] est modifiable !

**Lvalue** = expression qui peut se trouver à gauche de = :

- une variable ‘scalaire’,
- un accès dans une variable-tableau.

## Exemple

```
int a[10];
for (i=0; i<10; i++) a[i] = i*(i+1);
for (i=0; i<10; i++) printf("%i ",a[9-i]);
```

Par contre, une variable-tableau n'est pas une lvalue...

# Accès dans un tableau

**Puissance des tableaux** : on peut accéder à un indice *calculé* par une expression quelconque.

## Attention

Ne **jamais** accéder à un tableau en dehors de ses bornes !

- le langage C ne vérifie pas les accès dans les tableaux !
- le résultat d'un accès invalide est imprévisible :
  - peut planter le programme immédiatement avec **Segmentation fault**
  - peut ne rien faire de grave,
  - peut corrompre silencieusement la mémoire.

# Initialisation de tableaux

Avant affectation, le contenu du tableau est aléatoire.

## Déclaration avec initialisation

```
type var[taille] = { expr1, ..., exprN };
```

- les expressions sont séparées par des **virgules**,
- les accolades **{ }** et le point-virgule **;** sont obligatoires.

## Notes :

- on doit avoir  $N \leq \text{taille}$ ,
- si  $1 \leq N < \text{taille}$ , les initialiseurs manquants sont mis à 0,
- si **taille** est omise, elle est fixée à **N**.

# Initialisation de tableaux

## Exemples corrects

```
int a[3];           /* non initialise */  
int a[3] = { 2,1,0 }; /* complètement initialise */  
int a[3] = { 2,1 };  /* équivalent au précédent */  
int a[] = { 2,1,0 }; /* équivalent au précédent */
```

## Exemples incorrects

```
int a[2] = { 2,1,0 }; /* trop d'initialiseurs */  
int a[];               /* taille inconnue */  
a = { 1,2,3 };        /* pas une déclaration */
```

# Copies de tableaux

## Attention

On ne peut pas affecter un tableau entier.

Ne pas faire...

```
int a[10], b[10];  
a = b; /* erreur de syntaxe! */
```

...mais plutôt une boucle

```
int i;  
for (i=0;i<10;i) a[i] = b[i];
```

Il faut connaître à priori la taille du tableau,  
`sizeof(a) / sizeof(a[0])`

# Tableaux multidimensionnels

**Tableau multidimensionnel** = tableau de tableaux :

- matrices,
- images *bitmap*, etc.

## Déclaration

```
type var [taille1] [taille2] ... [tailleN];
```

Pas de limite au nombre de dimensions.

Seule limitation : la taille mémoire

$$\text{sizeof}(\text{var}) = \text{sizeof}(\text{type}) \times \text{taille1} \times \dots \times \text{tailleN}$$

e.g. : double cube[64][64][64] occupe 2 Mo.

# Accès dans un tableaux multidimensionnel

## Accès

var [expr1] [expr2] ... [exprN]

### Attention :

- il faut autant d'expressions qu'il y a de dimensions,
- chaque exprI doit être compris entre 0 et tailleI-1.

### Exemple : matrice identité

```
double mat[4][4];  
int i,j;  
for (i=0;i<4;i++)  
    for (j=0;j<4;j++)  
        mat[i][j] = (i==j) ? 1. : 0.;
```

# Représentation en mémoire

## Représentation en mémoire :

- taille1 tableaux de taille2 tableaux de ...  
de tableaux de tailleN éléments de type type.
- les éléments se succèdent en mémoire, sans 'trous'.

**Exemple :** int a[2][3];

.	.	.	.	.	.
a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]

$$\Rightarrow \begin{array}{lcl} \text{int a[2][3];} & \simeq & \text{int a[2*3];} \\ \text{a[i][j]} & \simeq & \text{a[(i)*3+(j)]} \end{array}$$

# Initialisation des tableaux multidimensionnels

**Déclaration avec initialisation** : { et } imbriqués.

## Exemple

```
int a[2][3] = { { 1,0,1 }, { 0,1,0 } };
```

**Copie de tableaux** : par des boucles imbriquées.

# Passage de tableaux en paramètres

## Attention

Les tableaux sont passés par référence, pas par valeur !

- pas de mémoire allouée pour l'argument,
- pas de copie du tableau passé en paramètre,
- un accès à l'argument modifie le tableau passé en paramètre.

## Attention

Une fonction ne peut pas retourner de tableau !

# Exemple

## Exemple de passage par référence

```
void zero(int a[4]);  
  
int main()  
{  
    int tmp[4];  
    zero(tmp);  
    return 1;  
}  
  
void zero(int a[4])  
{  
    int i;  
    for (i=0;i<4;i++) a[i] = 0; /* modifie tmp */  
}
```

## Les constantes symboliques

---

# Définition de constantes symboliques

Syntaxe (baroque)

`#define CST valeur`

- CST : identificateur, par convention en majuscules,
- valeur : texte arbitraire,
- doit occuper une ligne complète,
- pas de point-virgule ; final.

**Effet :** dans la suite du programme, CST est remplacé par valeur.

# Application des constantes symboliques

## Application :

écrire du code *paramétrique* là où le C interdit les variables.

### Exemple

```
#define N 10
int x[N], y[N];
void f()
{
    int i;
    for (i=0;i<N;i++)
        y[i] = x[i] + 1;
}
```

Pour changer la taille de **tous** les tableaux,  
il suffit de changer une seule ligne.

# Danger des constantes symboliques

Définition de constante symbolique  $\neq$  affectation de variable !

- affectation : évaluation,
  - constante symbolique : substitution littérale.
- ⇒ danger de “capture” syntaxique.

## Exemple d'erreur

```
#define N x+y
z = 3*N; /* signifie z = 3*x+y, pas z = 3*(x+y) */
           /* x et y peuvent aussi symboliques! */
```

## Correction

```
#define N ((x)+(y))      /* plus sûr */
```

# Initiation au C

## Cours n°4

Antoine Miné<sup>1</sup> Ozan Caglayan<sup>2</sup>

<sup>1</sup>École Normale Supérieure

<sup>2</sup>Université Galatasaray

19 février 2015



# 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 (!)

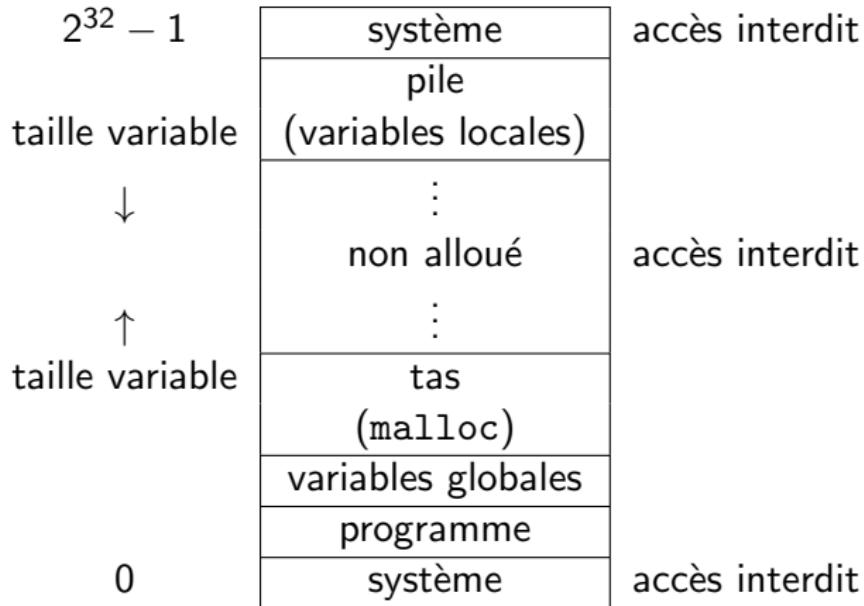
D'autres modèles existent : (segmentés, paginés, etc.)

**Virtualisation** = gestion et protection de la mémoire :

- chaque programme a son espace mémoire “logique”,
- l'OS gère la correspondance mémoire logique  $\rightarrow$  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  $t_1 \neq t_2$ , alors  $t_1*$  et  $t_2*$  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 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”  
⇒ 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,  
⇒ 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  $i \times \text{sizeof}(*p)$  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)++;

⇒ 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.

# Initiation au C

## Cours n°5

Antoine Miné<sup>1</sup> Ozan Caglayan<sup>2</sup>

<sup>1</sup>École Normale Supérieure

<sup>2</sup>Université Galatasaray

19 février 2015



# Plan du cours

- le débogage avec gdb,
- les chaînes de caractères.

# Le débogueur gdb

---

# Présentation de gdb

**GDB = débogueur** interactif permettant de :

- interrompre et reprendre l'exécution du programme,
- suivre l'exécution du programme pas à pas,
- poser des points d'arrêt,
- inspecter le contenu des variables,
- connaître la ligne exacte et le contenu des variables au moment d'un Segmentation fault.

# Lancement de gdb

## Préparation du programme

- compiler son programme avec l'option **-g**,
- éviter les options d'optimisation **-O2**, **-O3**, ...

**Exemple :** `gcc toto.c -g -Wall -Wextra`

## Lancement de gdb

Dans le shell, taper :

**gdb a.out**

gdb propose un *shell* interactif, d'invite (**gdb**).

# Exécution du programme sous gdb

## Lancement du programme

(gdb) **run**

L'exécution se termine ou est suspendue dès que :

- le programme **se termine** normalement :

Program exited with code XXX

- le programme **se termine** sur une erreur fatale :

Program received signal SIGSEGV, Segmentation fault

- l'utilisateur tape **contrôle+C** :

Program received signal SIGINT, Interrupt

- le programme passe par un **point d'arrêt** :

Breakpoint X, fun at file: line

# Points d'arrêt

## Placement d'un point d'arrêt

(gdb) **break nom de fonction**

(gdb) **break n° de ligne**

Suspend l'exécution à chaque fois que le programme :

- entre dans la fonction indiquée, ou
- arrive au début de la ligne indiquée.

⇒ on peut alors entrer de nouvelles commandes gdb.

### Notes :

- on peut placer des points d'arrêt avant run,
- on peut placer plusieurs points d'arrêt,
- un point d'arrêt reste actif jusqu'à sa destruction explicite :  
**delete** efface *tous* les points d'arrêt.

# Reprise de l'exécution

## Reprise de l'exécution (gdb) **continue**

Possible uniquement si l'exécution n'est que suspendue :

### Possible

- après contrôle+C,
- sur un point d'arrêt.

### Impossible

- avant run,
- après terminaison normale,
- après terminaison sur erreur.

**Note :** run **recommence** l'exécution au début.

# Exécution pas à pas

## Exécution d'une ligne

(gdb) **next**

(gdb) **step**

**Effet** : exécute une seule ligne et rend la main.

L'exécution peut être suspendue avant la ligne suivante :

- par contrôle+C,
- en cas de point d'arrêt,
- en cas d'appel de procédure pour **step**.

**Contraintes** : identiques à celles de continue.

Voir aussi : **finish**, **next n**, **step n**.

# Inspection des données

## Affichage d'une expression

(gdb) **print expr**

### Expressions autorisées :

- opérateurs C classiques, y compris déréférences \*, [] ,
- variables globales,
- variables locales de la fonction en cours d'exécution.

### Contraintes : le programme doit :

- soit être suspendu (point d'arrêt, next, etc.),
- soit s'être terminé sur une erreur fatale.

# Inspection de la pile

## Inspection de la pile

(gdb) **bt**            (concis)  
(gdb) **bt full**    (détailé)

**Effet :** affiche la pile d'appel et les arguments des fonctions.

# Déplacement dans la pile

On peut se “déplacer” dans la pile d’appel pour choisir une fonction.

## Déplacement dans la pile

- (gdb) **up** (monte vers l’appelant)
- (gdb) **down** (descend vers l’appelé)

## Effet :

- sur **print**  
indique de quelles variables locales on parle.
- sur **finish** :  
indique la fonction après laquelle gdb rend la main.

Aucun effet sur l’exécution du programme (`continue`, `next`,...).

# Exemple de session

## Programme débogué

```
int f(int x)          int main()
{
    int y = x+1;      {
    return y;          int x;
}                      x = f(12);
}                      return 0;
```

## Exemple de session

```
$ gcc toto.c -Wall -Wextra -g
```

```
$ gdb ./a.out
```

```
GNU gdb 6.6
```

```
Copyright (C) 2006 Free Software Foundation, Inc.
```

```
...
```

# Exemple de session

## Exemple de session (suite)

```
(gdb) break main
Breakpoint 1 at 0x4004b4: file toto.c, line 11.
(gdb) run
Starting program: /home/mine/ATER/2006/prog/a.out

Breakpoint 1, main () at toto.c:11
11      x = f(12);
(gdb) step
f (x=12) at toto.c:3
3      int y = x+1;
(gdb)
4      return y;
(gdb) print y
$1 = 13
```

# Exemple de session

## Exemple de session (fin)

```
(gdb) up
#1 0x00000000004004be in main () at toto.c:11
11      x = f(12);
(gdb) step
5      }
(gdb) print x
$2 = 12
(gdb) step
main () at toto.c:12
12      return 0;
(gdb) print x
$3 = 13
(gdb) quit
The program is running. Exit anyway? (y or n) y
$
```

# Récapitulatif des commandes gdb

quit (ou Ctrl+D)	quitte
<b>help</b>	aide intégrée
run	commence l'exécution
continue	reprend l'exécution
next (ou next <i>n</i> )	exécute une (ou <i>n</i> ) ligne(s)
step (ou step <i>n</i> )	" (mais s'arrête aux fonctions)
finish	exécute jusqu'au retour de la fonction
Ctrl+C	suspend l'exécution
break <i>n° ligne</i>	place un point d'arrêt
break <i>fonction</i>	"
delete	efface tous les points d'arrêts
print <i>expr</i>	affiche la valeur d'une expression
up	remonte dans la pile d'appels
down	descend dans la pile d'appels
bt (ou bt full)	affiche la pile d'appels

# Les caractères

---

# Les caractères

**Représentation** : par un code de caractère :

- historiquement : sur 7 bits  $\Rightarrow$  128 codes,
- de nos jours : sur **8 bits**  $\Rightarrow$  **256 codes**.

Les caractères regroupent :

- codes 0 à 127 : standard **ASCII** :
  - 32 à 126 : 95 **symboles affichables**,
  - 0 à 31 et 127 : **33 caractères de contrôle**,
- codes 128 à 255 : symboles affichables étendus : de nombreux standard existent :
  - **latin1** : ISO pour l'Europe de l'ouest,
  - **Mac-Roman** : Europe de l'ouest sur les Macintoshs,
  - **UTF-8** : standard universel Unicode,

**En C** : type **char**.

# Codes ASCII

**ASCII = American Standard Code for Information Interchange**  
 (1961).

## Codes ASCII affichables

code	caractères															
32	<i>esp</i>	!	"	#	\$	%	&	,	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	-
96	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	

**Note :** voir aussi **man ascii**.

# Caractères de contrôle

## Caractères de contrôle courants

code	sigle	signification	notation C
10	LF	saut de ligne	\n
13	CR	début de ligne	\r
8	BS	efface un caractère	\b
7	BEL	bip	\a
27	ESC	échappement	\033
0	NUL	fin de chaîne C	\0 (ou \000)

**Affichage** d'un caractère de contrôle : l'effet dépend du **terminal** !

# Constantes caractères en C

**Constante char :** un caractère entre ' ' :

- **caractère affichable** non '  
exemples : 'a' ' )' ' , '
- **séquence spéciale** commençant par '\  
exemples : '\n' '\, ' \\'
- **code numérique** en octal (base 8) de 1 à 3 chiffres  
exemples : '\033' (= 27) '\0' '\000'
- **code numérique** en hexadécimal (base 16) de 1 à 2 chiffres  
exemples : '\x0a' (= 10) '\x40' (= 65, ='A')

**Note :** on peut aussi utiliser un **nombre tout bête**

exemple : char c = 13;

# Arithmétique sur les caractères

## Remarques :

- char est un type entier,
- l'ASCII est organisé en plages logiques (A à Z, etc.)

⇒ on peut faire de l'**arithmétique sur les caractères**.

### Exemple : mettre en majuscule

```
char en_majuscule(char c)
{
    if ( c >= 'a' && c <= 'z' ) return c - 'a' + 'A';
    else                           return c;
}
```

# Les chaînes de caractères

---

# Représentation des chaînes de caractères

**Chaîne (string) = suite de caractères terminée par le caractère \0.**

En C, pas de type *chaîne* spécifique :

- **déclaration** : on utilise un **tableau de char** : `char []`,
- **en argument** : on passe un **pointeur** dans un tableau : `char*`.

# Déclaration de chaînes

**Déclaration sans initialisation** (dangereux car pas de \0)

## Exemples

```
char buf[1024];
```

## Initialisation caractère par caractère

### Exemples

```
char cc[] = { 'H', 'e', 'l', 'l', 'o', '!', 0 };
char toto[4] = { 'H', 'i', '\0' };
char titi[4] = { 'H', 'i' };
```

Rappel des règles sur les initialisations de tableaux :

- avec [], la taille est calculée automatiquement,
- sinon, les initialiseurs manquants sont mis à 0.

# Déclaration de chaînes

**Initialisation par une chaîne littérale entre " " :**

## Exemples

```
char cc[] = "Hello!";  
char toto[4] = "Hi";
```

Avantages :

- plus léger et naturel,
- le \0 final est automatiquement ajouté,
- on peut utiliser les caractères spéciaux,  
Exemple : "toto\na \033\a".

# Rappels sur les pointeurs

**char\*** : référence un objet char en mémoire.

**Rappels** : si on se donne    `char x, t[5];`    alors

- `&x` pointe sur `x`,
- `&t[5]` pointe sur la 6ème case de `t`,
- `t` est équivalent à `&t[0]`,
- ces objets ont tous pour type `char*`.

**Déréférencement de pointeur** : par `[i]` ou `*` (`=[0]`).

**Arithmétique de pointeurs** : si `p` pointe sur la *i*ème case de `t`,

- `p+j` pointe sur la *i+j*ème case de `t`,
- `p-j` pointe sur la *i-j*ème case de `t`,
- `p++, p--` décalent `p` d'une case, etc.

## Passage en paramètre

Les chaînes sont passées aux fonctions **par référence** sous forme de **pointeur vers le premier caractère**.

Type de l'argument :

- **char\*** si la chaîne risque d'être modifiée par la fonction,
- **const char\*** si la chaîne n'est pas modifiée.

### Exemple

```
int longueur(const char* s)
{
    int i = 0;
    while (s[i]) i++;
    return i;
}
```

Exemple typique : on boucle du premier au dernier caractère (0).

# Passage en paramètre

## Autre exemple

```
char en_majuscule(char c)
{
    if ( c >= 'a' && c <= 'z' )
        return c - 'a' + 'A';
    else
        return c;
}

void chaine_en_majuscule(char* s)
{
    for ( ; *s; s++)
        *s = en_majuscule(*s);
}
```

# Fonctions standards sur les chaînes

**Fonctions standards** : documentées dans [man string](#).

## Quelques fonctions utiles

<code>strlen</code>	longueur d'une chaîne
<code>strcmp</code>	compare deux chaînes lexicographiquement
<code>strcpy</code>	copie de chaîne
<code>strcat</code>	concaténation de chaînes
<code>strncpy</code>	
<code>strncat</code>	vérions limitées à $n$ caractères
<code>strncmp</code>	
<code>strchr</code>	recherche d'un caractère dans une chaîne
<code>strstr</code>	recherche d'une sous-chaîne dans une chaîne

Il faut faire `#include <string.h>`.

# Utilisation des fonctions standards

## Exemples de prototypes

```
size_t strlen(const char *s);  
char *strcpy(char *dest, const char *src);  
char *strcat(char *dest, const char *src);
```

## Exemple d'utilisation

```
void make_path(const char* dir, const char* file)  
{  
    char buf[1024]  
    if ( strlen(dir) + strlen(file) + 2 > sizeof(buf) ) return;  
    strcpy(buf, dir);  
    strcat(buf, "/");  
    strcat(buf, file);  
    ...
```

# Chaînes constantes

On peut utiliser une chaîne entre " " en dehors des initialisations de tableaux.

## Exemples

```
printf("toto\n");
char* s = "toto\n";
```

## Effet :

- à la compilation : créé un tableau anonyme bien initialisé, le tableau est **global** et **constant**,  
(e.g. : const char anonymeXXX[] = "toto";)
- à l'exécution : renvoie un **pointeur** sur ce tableau.  
(e.g. : printf(anonymeXXX); char\* s = anonymeXXX;)

## Notes :

- à l'exécution, aucune allocation ni copie de chaîne n'a lieu,
- la tableau ne doit pas être modifié.

# Affichage de chaînes

On utilise le format **%s** avec printf.

## Exemple

```
char nom[] = "personne";
printf("mon nom est %s\n", nom);
```

## Attention

Ne jamais faire printf(**s**)...  
... et si s contient un caractère % ?

## Extensions :

- **%ns** : affiche au moins *n* caractères (complète par des blancs),
- **%.ms** : affiche au plus *m* caractères.

# Erreurs classiques

## Cause principale

Tout accès en dehors des bornes d'un tableau provoque une erreur non déterministe.

## Exemples d'erreurs

```
char buf[4];
char c[2] = { 'a', 'b' };
char* s;

strcpy( buf, s );           /* s pointe dans l'espace */
strcpy( buf, "abcd" );     /* buf trop petit */
strcpy( buf, c );          /* c non terminé par \0 */
s = "toto";
s[1] = 'a';                /* s pointe sur dans un tableau constant */
```

# Le latin1

**latin1** (ou **ISO 8859-1**) : l'ASCII étendu pour l'**Europe de l'ouest**.

- codes 128 à 159 : codes de contrôle,
- codes 160 à 255 : symboles, lettres accentuées

## Codes latin1 étendus

160	ı	¢	Ł	¤	¥	׀	§	„	©	¤	«	„	®	—	
176	°	±	²	³	’	µ	¶	.	,	º	»	¼	½	¾	ı
192	À	Á	Â	Ã	Ä	Å	Æ	Ç	É	É	Ê	Ë	Ì	Í	Ï
208	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ
224	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	ï
240	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ

**latin9** (ou **ISO 8859-15**) : ajout de œ, œ, Ÿ, Š, š, Ž, ž, €.

Voir : **man latin1**, **man latin9**.

# Utiliser le latin1

Pour utiliser des caractères latin1 étendus dans " et ', il faut s'assurer que :

- le source .c est sauvegardé au format latin1,  
(Options -> Mule dans Emacs)
- le terminal est configuré en latin1.

## Exemple : le retour des majuscules

```
char en_majuscule(char c)
{
    if ( c >= 'a' && c <= 'z' || 
        c >= 'à' && c <= 'ÿ' && c != 247 && c != 248 )
        return c - 32;
    return c;
}
```

# Unicode et UTF-8

**Unicode** (ou **ISO 10646**) : standard universel (**UCS**), 1M codes.

**UTF-8** : encodage de taille variable pour Unicode :

- codes 0 à 127 : sur 1 octet, identique à l'ASCII (0xxxxxxxx)
- codes 128 à 2047 : sur **2 octets** (110xxxxx 10xxxxxx) (11 bits)
- codes 2048 à 65535 : sur **3 octets** (16 bits)  
(1110xxxx 10xxxxxx 10xxxxxx)
- à partir de 65536 : sur **4 octets** (21 bits)  
(11110xxx 10xxxxxx 10xxxxxx 10xxxxxx)

**Avantages** : entre autres :

- compatible avec l'**ASCII**,
- compatible avec le C (pas d'ajout de \0),
- codage non ambigu, parcours de droite à gauche possible.

# Unicode et UTF-8

Pour utiliser l'UTF-8 en C, il faut s'assurer que :

- le source .c est sauvegardé au format UTF-8,
- le terminal est configuré en UTF-8.

## Attention

Un “caractère” unicode peut correspondre à plusieurs char en C.

Conséquences :

- 'à' n'a pas de sens (à occupe deux char en UTF-8),
- strlen et strchr ne fonctionnent pas comme on s'y attend,
- strcat et strcpy fonctionnent,  
mais attention à évaluer la taille des tableaux en char !

# Initiation au C

## Cours n°6

Antoine Miné <sup>1</sup> Ozan Caglayan <sup>2</sup>

<sup>1</sup>École Normale Supérieure

<sup>2</sup>Université Galatasaray

19 février 2015



# Plan du cours

Communications :

- les arguments en ligne de commande,
- les variables d'environnement,
- les fichiers, les flux d'entrée-sortie **FILE\***.

# Arguments en ligne de commande

---

# Arguments en ligne de commande

## Ligne de commande :

on peut passer un nombre **arbitraire** d'arguments à un programme.

Le *shell* effectue plusieurs opérations sur la ligne de commande :

- **Expansion** des caractères \*, ?, les **noms des fichiers** du répertoire courant sont utilisés.
- **Découpage** des arguments au niveau des **espaces**, on peut se protéger des caractères spéciaux par " ou \.

### Exemples

\$ ./a.out a b c	passe a, b et c	(3 arguments)
\$ ./a.out "a b" c	passe a b et c	(2 arguments)
\$ ./a.out *	passe la liste des fichiers	
\$ ./a.out \*	passe juste *	(1 argument)

# Lire les arguments depuis le C

En C : ils sont passés en argument à `main` sous forme de chaînes.

## Prototype de `main`

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

- `argc` : nombre d'arguments disponibles,
- `argv` : tableau de chaînes de caractères :
  - `argv[0]`              1er argument = **nom du programme**,
  - `argv[1]`              **1er argument supplémentaire**,
  - ⋮
  - `argv[argc-1]`      **dernier argument**,
  - `argv[argc]`          pointeur NULL (invalidé!).

# Exemple

## Exemple de programme

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    int i;
    printf( "%i args\n", argc );
    for ( i=0; i<argc; i++ )
        printf( "%i: %s\n",
                i, argv[i] );
    return 1;
}
```

## Exemple de session

```
$ gcc toto.c -Wall -Wextra
$ ./a.out hello world 42 !
5 args
0: ./a.out
1: hello
2: world
3: 42
4: !
$
```

## Attention

Un programme ne peut pas faire d'hypothèses sur argc.

# Passage d'arguments numériques

**Arguments numériques** : aussi passés comme des chaînes !

⇒ c'est au programme C de faire la conversion.

## Fonctions prédéfinies

```
#include <stdlib.h>
int    atoi (const char *nptr);
double atof (const char *nptr);
```

Exemple : int a = atoi(argv[2]);

## Les variables d'environnement

---

# Les variables d'environnement

**Environnement** : table d'association gérée par le système.

Permet d'affecter à une variable (chaîne) une valeur (chaîne).

L'environnement est :

- initialisé par le *shell*,

Exemple : `USERNAME = votre login.`

- modifiable par l'utilisateur,

Exemples : `export VAR=val`, fichier `.login` (pour bash).

- hérité par le programme exécuté,

⇒ utilisable par votre programme.

# Accès à l'environnement depuis le C

## Fonction prédefinie

```
#include <stdlib.h>
char* getenv(const char* name);
```

- retourne (un pointeur sur) une chaîne : la valeur de name,
- retourne **NULL** si name n'est pas définie,
- la chaîne retornnée ne doit pas être modifiée !

## Exemple

```
char* nom = getenv("USERNAME");
printf("Bonjour %s\n", nom ? nom : "X");
```

# Les flux d'entrée-sortie

---

# Les fichiers

**Fichier** = bloc de données, non volatile, stocké sur disque

- Chaque fichier a un nom et vit dans un répertoire,
- Données = suite d'octets **non structurée**,  
les programmes se mettent d'accord sur des formats d'échange.
- Par convention, la fin du nom indique le format,  
e.g. : .txt = texte brut, .png = image PNG, etc.

**Unix** = système multi-utilisateurs, donc chaque fichier a :

- un utilisateur et un groupe propriétaire,
- des droits : lecture, écriture, exécution,  
qui concernent : le propriétaire, le groupe, les autres.

(Voir ls -l et chmod.)

# Les flux en C

**flux** = objet C permettant d'accéder à un fichier.



Géré par la bibliothèque C,  
pour le programmeur, c'est un **type abstrait** (ne pas déréférencer).

Opérations sur les FILE\* :

- ouverture : fopen
- lecture : fgetc, fgets, fscanf, fread
- écriture : fputc, fputs, fprintf, fwrite
- navigation : ftell, fseek, feof
- fermeture : fclose

# Premier exemple

## Exemple

```
#include <stdio.h>
int main()
{
    FILE* f = fopen("toto.txt", "w");      /* ouverture */
    if (f==NULL) return 1;                  /* vérification */
    fputs("coucou\n", f);                 /* écriture */
    fclose(f);                           /* fermeture */
    return 0;
}
```

**Effet :** écrit la ligne coucou dans le fichier toto.txt.

Le pointeur FILE\* est passé en argument de toutes les fonctions.

# Ouverture des flux

**Note :** toutes les fonctions sont dans `stdio.h`.

## Prototype

```
FILE* fopen(const char* path, const char* mode);
```

## Arguments :

- `path` = nom (chemin) du fichier à ouvrir,
- `mode` = mode d'ouverture (lecture, écriture, ...).

## Valeur de retour :

- $\neq \text{NULL}$   $\Rightarrow$  nouveau flux représentant le fichier,
- $= \text{NULL}$   $\Rightarrow$  échec de l'ouverture,  
(fichier introuvable, droits insuffisants,..., voir `errno`)

$\implies$  toujours vérifier la valeur de retour !

# Chemins et noms de fichiers

path peut représenter un chemin complet :

- liste de répertoires séparés par / terminée par le nom de fichier,
- chemin **absolu**, commençant par /  
e.g. : /home/mine/truc.c,
- chemin **relatif**, ne commençant pas par /  
e.g. : truc.c, bidule/muche/truc.c,
- le répertoire .. représente le parent, e.g. : ../tmp/x.c.

**Sous Windows :**

le séparateur est \ au lieu de /

un chemin absolu commence par un lecteur, e.g. : c\truc\...

**Sous les vieux MacOS :**

le séparateur est : au lieu de /.

# Modes d'ouverture

mode = chaîne décrivant comment le fichier doit être ouvert :

"r"	lecture seule
"w"	écriture seule
"r+" ou "w+"	lecture et écriture
"a"	écriture à la fin du fichier

Si le fichier existe déjà :

- "r" et "r+" se placent en début de fichier,
- "w" et "w+" tronquent le fichier,
- "a" se positionne en fin de fichier.

Si le fichier n'existe pas :

- "r" et "r+" échouent,
- "w", "w+" et "a" créent le fichier.

Sous Windows : ajouter b pour indiquer un fichier binaire et non texte.

# Fermeture des flux

## Prototype

```
int fclose(FILE *fp);
```

### Effet :

- s'assure que les données sont écrites sur disque (buffering),
- ferme le flux pointé par fp,
- le flux pointé par fp ne doit plus être utilisé,  
mais on peut faire : `fclose(fp); fp = fopen(...);`

**Valeur de retour :** 0 si tout s'est bien passé (généralement le cas).

Les fichiers ouverts sont automatiquement fermés si le programme se termine normalement... pas en cas de Segmentation fault.

# Écritures simples

## Prototypes

```
int fputc(int c, FILE *stream);  
int putc(int c, FILE *stream);  
int fputs(const char *s, FILE *stream);
```

### Effet :

- `fputc` écrit un seul **caractère** `c`,
- `putc` identique à `fputc` (plus rapide),
- `fputs` écrit une **chaîne entière**, sans le `\0` final.

### Valeur de retour :

en cas d'erreur, la valeur spéciale **EOF** est retournée,  
sinon, une valeur  $\geq 0$ .

# Écritures formatées

## Prototype

```
int fprintf(FILE *stream, const char* format, ...);
```

S'utilise exactement comme printf, avec stream en plus :

- format : chaîne avec des caractères % magiques,
- on ajoute autant d'arguments que de %.

## Valeur de retour :

- nombre de caractères écrits,
- valeur négative en cas d'erreur.

## Exemple

```
fprintf(f, "%i * 2 = %i\n", x, x*2);
```

# Lecture de caractères

## Prototypes

```
int fgetc(FILE *stream);  
int getc(FILE *stream);
```

**Effet** : lit un seul octet.

getc est identique à fgetc (en plus rapide).

**Valeur de retour :**

- entier entre 0 et 255 (`unsigned char`),
- valeur spéciale **EOF** en cas de fin de fichier ou d'erreur.

# Exemple

## Copie de fichier caractère par caractère

```
void copie(const char* src, const char* dst)
{
    FILE* s = fopen( src, "r" );
    FILE* d = fopen( dst, "w" );
    if (!s || !d) { printf("erreur!\n"); exit(1); }
    while (1) {
        int x = getc(s);
        if (x == EOF) break;
        if (putc(x,d) == EOF) { printf("erreur!\n"); break; }
    }
    fclose(s);
    fclose(d);
}
```

# Lecture de lignes complètes

## Prototype

```
char* fgets(char* s, int size, FILE *stream);
```

## Effet :

- lit une ligne complète, terminée par \n,
- stocke la ligne dans la chaîne pointée par s,  
inclus le \n, puis un \0 final,
- retourne NULL si erreur ou plus rien à lire, s sinon.

## Cas particuliers : lignes non terminées par \n

- la dernière ligne du fichier ne se termine pas forcément par \n,
- lit au plus size caractères, avant d'ajouter le \0.

## Attention

Prévoir de la place pour size+1 octets dans s !

# Lectures formatées

## Prototype

```
int fscanf(FILE *stream, const char* format,...);
```

S'utilise exactement comme scanf, avec stream en plus :

- format : chaîne avec des caractères % magiques,
- on ajoute autant de pointeurs que de %.

## Valeur de retour :

- nombre d'éléments correctement lus,
- EOF en cas d'erreur ou de fin de fichier.

## Exemple

```
fscanf(f, "%i,%i,%i", &a, &b, &c);
```

# Erreurs sur les fichiers

En cas d'erreur, une valeur spéciale est retournée :

- **NULL** pour un type de retour pointeur,
- **EOF** pour un type de retour entier.

De plus, la variable globale entière **errno** est positionnée.

## Codes d'erreur courants

Valeurs symboliques définies dans **errno.h**

EACCES	fichier introuvable
EEXIST	fichier déjà existant
EISDIR	le fichier est un répertoire
EIO	erreur matérielle
EINVAL	position invalide

(errno n'est pas modifié en cas de succès)

## Fonctions sur les erreurs

Fonctions permettent de rendre ces codes d'erreur intelligibles.

### Prototypes

```
void perror(const char *s);  
char* strerror(int errnum);
```

#### Effet de perror :

affiche s et un message expliquant l'erreur de code errno.

#### Valeur de retour de strerror :

description textuelle de l'erreur de code errnum.

⇒ généralement appelée avec errno comme argument.

# Fin de fichiers

## Lecture en fin de fichier :

- on peut lire moins que ce qui est demandé (fgets, fscanf),
- si on ne peut rien lire, on retourne une erreur (NULL, EOF).

## Comment différencier vraie erreur et fin de fichier ?

### Prototypes

```
int feof(FILE *stream);  
int ferror(FILE *stream);
```

- feof retourne  $\neq 0$  si on est en fin de fichier, 0 sinon,
- ferror retourne  $\neq 0$  si il y a eu une erreur, 0 sinon.

## Écriture en fin de fichier : pas d'erreur, le fichier est agrandi.

# Exemple

## Exemple de gestion des erreurs

```
void echo(const char* fname)
{
    FILE* f = fopen(fname, "r");
    if (f) {
        while (1) {
            int c = getc(f);
            if (c == EOF) {
                if (ferror(f)) perror("échec de getc");
                fclose(f); return;
            }
            printf("%c", c);
        }
    } else printf("échec de open(%s): %s\n",
                 fname, strerror(errno));
}
```

# Position courante dans un flux

Chaque flux a une **position courante** en octets.

- initialisée à 0 à l'ouverture (sauf mode "a"),
- mise à jour à chaque lecture et écriture,
- un seul curseur pour la lecture et l'écriture.

## Prototype

```
long ftell(FILE* stream);
```

**Valeur de retour :**

- position en octets depuis le début du fichier,
- -1 en cas d'erreur.

## Positionnement dans un flux

Si le mode n'est pas "a", on peut changer la position du curseur.

### Prototype

```
int fseek(FILE *stream, long offset, int whence);
```

**Effet** : se déplace de **offset octets** dans le fichier.

La valeur symbolique **whence** indique un point de référence :

- **SEEK\_SET** à partir du début du fichier,
- **SEEK\_CUR** à partir de la position courante,
- **SEEK\_END** à partir de la fin du fichier.

⇒ **offset** peut être négatif,  
on peut se positionner au-delà du fichier !

# Exemple

Détermination de la taille d'un fichier

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    FILE* f;
    if (argc <= 1) return 1;
    f = fopen( argv[1], "r" );
    if (!f) printf("fichier %s introuvable\n", argv[1]);
    else {
        fseek(f, 0, SEEK_END);
        long p = ftell(f);
        printf("%li octets\n", p);
    }
}
```

# Opérations sur les fichiers

Opérations agissant directement sur le système de fichiers.

## Destruction de fichiers

```
#include <unistd.h>
int remove(const char *pathname);
```

## Renommage de fichiers

```
int rename(const char *oldpath, const char *newpath);
```

**Valeur de retour :** 0 en cas de succès, -1 en cas d'erreur (errno)

Pas d'ouverture de fichier  $\implies$  pas de flux FILE\*.

# Les flux stdin, stdout et stderr

Trois flux déjà ouverts sont prédéfinis par `stdio.h` :

- `stdin` entrée standard, en lecture seule,
- `stdout` sortie standard, en écriture seule,
- `stderr` sortie d'erreur, en écriture seule.

Par défaut, `stdin` est branché sur le **clavier**,  
`stdout` et `stderr` sont branchés sur l'**écran**.

Mais on peut les rediriger depuis le *shell* avec `<`, `>` et `2>`. (bash)

## Exemple

```
$ ./a.out < entree.txt > sortie.txt
```

# Les flux stdin, stdout et stderr

stdin, stdout et stderr sont de type FILE\*.

On peut les utiliser avec les fonctions d'entrée-sortie !

## D'ailleurs

printf(...) est un raccourci de fprintf(stdout,...)  
scanf(...) est un raccourci de fscanf(stdin,...)

## Attention :

- stdin, stdout et stderr n'ont pas de curseur,  
⇒ ftell et fseek ne fonctionnent pas... .
- feof fonctionne sur stdin, (frappe de contrôle-D)
- gets et puts ne sont pas équivalents à fgets et fputs... .

# Mémoire tampon

**Mémoire tampon (buffer) =**

zone de stockage temporaire en attendant un traitement.

Les entrées-sorties sont bufferisées. (raisons d'efficacité)

stdin :

le système attend d'avoir une ligne complète avant de la transmettre au programme.

stdout :

le C attend d'avoir une ligne complète avant de l'écrire à l'écran.

stderr : non bufferisé.

Flux sur des fichiers : bufferisés par blocs.

# Vidange

On peut **forcer l'écriture** du tampon.

## Prototype

```
int fflush(FILE *stream);
```

Exemple : `fflush(stdout);`

Autre méthode de vidange : **fseek**.

- marche pour le tampon d'écriture et **de lecture**  
    ⇒ entre une lecture et une écriture sur un même fichier,  
        on fait `fseek(f, 0, SEEK_CUR);`
- ne marche pas sur `stdout`, `stdin`...

# Initiation au C

## Cours n°7

Antoine Miné<sup>1</sup> Ozan Caglayan<sup>2</sup>

<sup>1</sup>École Normale Supérieure

<sup>2</sup>Université Galatasaray

19 février 2015



# Plan du cours

Les types de données **structurées** :

- **struct**,
- **union**.

Les *alias* de type : **typedef**.

# Rappels sur les types C

## Types scalaires :

- Types de base :
  - entiers : int, unsigned, char, long, etc.
  - flottants : float, double.
- Types pointeurs : type \*.

## Types composés :

- Types tableaux (homogènes) : type [] .
- **Types structures et unions** (hétérogènes).

# Les types struct

---

# Notion de structure

**Structure** : ou “enregistrement”.

Permet de **grouper plusieurs valeurs dans une seule variable**.

Exemple : compte (Suisse) = n° banque + n° compte + solde.

Une structure est composée d'un nombre fixé de **champs** :

- nommés, (banque, compte, solde)
- typés. (int pour banque et compte, float pour solde)

Les champs peuvent être de type différent.

Une **variable structurée** peut être manipulée soit :

- champ par champ, (lecture, mise à jour)
- comme un tout. (initialisation, copie, passage à une fonction)

# Déclarations de types structurés

**Déclaration de type** : obligatoire, on utilise le mot-clé `struct`.

## Syntaxe

```
struct mastuct {  
    type1 champ1;  
    :  
    typeN champN;  
};
```

**Effet** : déclare un nouveau `type` de structure

- de nom *mastuct*,
- de champs nommés *champ1* à *champN*,
- les champs ont pour type *type1* à *typeN*.

*mastuct*, *champ1* à *champN* doivent être des identificateurs.

# Déclarations de types structurés

## Exemple

```
struct compte {  
    int    banque;  
    int    compte;  
    float  solde;  
};
```

Sur un Intel 32 ou 64 bits,  
un objet de type compte occupera (`sizeof`) 12 octets.

# Déclaration de variables structurées

Le type associé à une structure est de la forme : ***struct mastruct***.

## Syntaxe

```
struct mastruct variable;
```

**Effet** : déclare une variable *variable* de type structure *mastruct* préalablement défini.

## Exemple

```
struct compte cb, pel;
```

Déclare deux variable structurées de type *compte*.

Ne pas confondre : nom de type, de variable, de champ.

# Accès aux champs

## Opérateur point .

### Syntaxe

*variable.champ*

**Effet :** référence le champ *champ* de la variable *variable*

- utilisable dans une expression,

Exemple : `float fric = cb.solde + pel.solde;`

- modifiable (*lvalue*).

Exemple : `cb.solde -= 200.25;`

# Initialisation de structures

**Initialisation** à la déclaration : comme pour un tableau.

## Syntaxe

```
struct mastruct variable = { expr1, ..., exprN };
```

- l'ordre des expressions est le même que celui des champs,
- les champs manquants sont initialisés à 0 ou NULL,
- on peut imbriquer les initialiseurs entre { et }.  
(structures et/ou tableaux imbriqués)

## Exemple

```
struct compte cb = { 99, 1345, 0.1 };
```

# Copies de structures

**Copie** : l'opérateur = peut être utilisé sur des variables structurées.

**Effet** : copie champ à champ.

## Exemple

```
new_cb = cb;
```

est équivalent à

```
new_cb.banque = cb.banque;  
new_cb.compte = cb.compte;  
new_cb.solde = cb.solde;
```

**Initialisation par recopie** :    struct compte visa = cb;

Ce n'était pas possible avec les tableaux !

# Passage de structures par valeur

## Appels de fonctions :

les variables structurées sont passées **par valeur**.

### Exemple

```
void anniversaire(struct compte c)
{ c.solde += 10.; }

struct compte cb;
anniversaire(cb);
```

- une nouvelle variable `c` est créée,
- les champs de `cb` sont copiés dans ceux de `c`,
- une modification de `c` **ne change pas** `cb` !

Comportement très différent de celui des tableaux !

# Retour de structures

**Retour** : une fonction peut retourner une structure.

## Exemple

```
struct compte nouveau_compte(int banque)
{
    struct compte c = { banque, lrand48(), 0. };
    return c;
}
```

## Applications :

- initialisation : `struct compte c = nouveau_compte(12);`
- copie : `cb = nouveau_compte(42);`

Là encore, un comportement très différent des tableaux.

# Passage de structures par référence

**Passage par référence** : peut être simulé grâce aux **pointeurs**.

## Exemple

```
void loyer(struct compte* c)
{ (*c).solde -= 999.9; }

struct compte cb;
loyer(&cb);
```

- il n'y a pas d'allocation ou de copie de structure,  
⇒ coût faible en mémoire et en temps,
- \*c et cb **référencent le même objet en mémoire**,  
⇒ (\*c).solde -= 999.9; modifie cb.solde.

# L'opérateur ->

Attention à la priorité des opérateurs

`*x.y` signifie `*(x.y)` et pas `(*x).y`

Comme on a souvent besoin de la construction `(*x).y`,  
le C propose un opérateur spécial : `->`. (tiret, supérieur)

## Syntaxe

variable `->` champ

- strictement équivalent à `(*variable).champ.`

# Application : affichage d'une structure

**Affichage** : il faut afficher les champs un par un, à la main !  
⇒ on définit souvent une fonction auxiliaire.

## Fonction d'affichage

```
void affiche_compte(const struct compte* c)
{
    printf("Compte\n");
    printf("-----\n");
    printf("Banque : %i\n", c->banque);
    printf("Numéro : %i\n", c->numero);
    printf("SOLDE : %f\n", c->solde);
}
```

- on opte pour un passage par référence pour éviter la copie,
- `*c` n'est pas modifié ⇒ on l'indique par `const`.

# Tableaux dans les structures

Des **champs tableaux** peuvent apparaître dans une structure :

## Exemple

```
struct id {  
    char nom[30];  
    int naissance[3];  
};
```

## Notes :

- une variable de type `struct id` occupe 42 octets,
- on peut imbriquer les initialiseurs :  
Ex. : `struct id u = { "Antoine", { 11, 10, 1977 } };`
- on peut accéder à un élément par **un chemin d'accès** :  
Exemple : `u.naissance[2]` : année de naissance.

# Tableaux dans les structures

- L'affectation =, l'initialisation, le passage en argument ou en retour de fonction copient récursivement les champs.

## Exemple

```
v = u;
```

est équivalent à

```
v.nom[0] = u.nom[0];
```

```
:
```

```
v.nom[29] = u.nom[29];
```

```
v.naissance[0] = u.naissance[0];
```

```
v.naissance[1] = u.naissance[1];
```

```
v.naissance[2] = u.naissance[2];
```

# Structures imbriquées

Des **champs structures** peuvent apparaître dans une structure :

## Exemple

```
struct fiche {  
    struct id      ident;  
    struct compte cb, visa;  
};
```

**Notes :** (similaires aux tableaux dans les structures)

- on peut imbriquer les initialiseurs,
- les sous-structures sont copiées récursivement,
- on accède à un champ par un chemin d'accès :  
Exemple : u.cb.numero : n° de compte CB.

# Exemples de structures imbriquées

## Exemple

```
void affiche_compte(const struct compte* c);

void affiche(const struct fiche* f)
{
    printf( "fiche de %s\n", f->ident.nom );
    affiche_compte( &f->cb );
    affiche_compte( &f->visa );
}
```

Les structures permettent :

- d'organiser ses données de manière hiérarchique,
- de réutiliser des fonctions.

# Tableaux de structures

On peut aussi faire des tableaux de structures...

## Exemple

```
struct entree {  
    char mot[TAILLE_MOTS];  
    int nombre_ok, nombre_spam;  
};  
  
struct entree corpus[NB_MOTS];
```

## Quizz :

- comment accéder au  $i$ -ème caractère du  $j$ -ème mot ?
- `corpus[i]` est-il passé par valeur ou référence ?
- `corpus` est-il passé par valeur ou référence ?

# Tableaux de structures

On peut aussi faire des tableaux de structures...

## Exemple

```
struct entree {  
    char mot[TAILLE_MOTS];  
    int nombre_ok, nombre_spam;  
};  
  
struct entree corpus[NB_MOTS];
```

## Quizz :

- accès au  $i$ -ème caractère du  $j$ -ème mot : `corpus[j].mot[i]`,
- `corpus[i]` est passé par **valeur** et peut être copié,
- `corpus` est passé par **référence** et ne peut pas être copié.

## Alias de type avec `typedef`

---

# Alias de type avec **typedef**

## Syntaxe

```
typedef type monalias;
```

**Effet :** *monalias* devient un *alias* de *type*.

## Exemples

```
typedef unsigned int uint;      uint i;  
typedef int compte_t;          compte_t numero;
```

**Applications :** permet de rendre un programme

- plus **concis**, (uint plus court que unsigned int)
- **paramétrique**, par abstraction du type réellement utilisé.  
(on peut facilement changer le type des numéros de compte)

# Application de `typedef` aux structures

On peut se servir de `typedef` pour éviter le mot-clé `struct`.

## Exemple

```
struct compte { ... };  
typedef struct compte compte_t;  
  
compte_t c = { ... };  
void affiche_compte(const compte_t* c);
```

**Note :** on peut utiliser un *alias* dans un *alias* de type.

## Exemple

```
typedef const compte_t* compte_constptr;  
void affiche_compte(compte_constptr c);
```

## typedef alternatifs

### Exemple

```
struct compte { ... };  
typedef struct compte compte;
```

On donne le même nom au type structuré et à l'*alias*.

Il n'y a pas d'ambiguïté entre `struct compte` et `compte`.

## typedef alternatifs

### Exemple

```
typedef struct compte { ... } compte;
```

Définit à la fois `struct compte` et `compte`.

On peut utiliser les deux types.

# `typedef` alternatifs

## Exemple

```
typedef struct { ... } compte;
```

Définit seulement `compte`, pas `struct compte`.

Le type structuré est anonyme.

# `typedef` dans la bibliothèque C

On a déjà vu des exemples de `typedef`... dans la bibliothèque C !

- `size_t`

utilisé, par exemple, comme type de retour de `strlen`,  
défini par la bibliothèque C comme un type entier,  
sur mon Linux 64-bit : `typedef unsigned long size_t`.

- `FILE`

type des flux ouverts par `fopen`,  
sur mon Linux 64-bit :

```
typedef struct _IO_FILE FILE;
struct _IO_FILE {
    int _flags;
    char* _IO_read_ptr;
    ...
};
```

## Les types union

---

# Notion d'union

**Union C = alternative.**

Exemple : nombre = entier ou flottant.

Une union est composée de champs :

- nommés,
- typés.

Les champs partagent le même emplacement en mémoire :

- on ne peut utiliser qu'un champ à la fois,
- l'union occupe la place nécessaire au plus gros champ.

# Déclarations de types unions

## Déclaration de type :

similaire à une structure, mais avec le mot-clé **union**.

### Syntaxe

```
union monunion {  
    type1  champ1;  
    :  
    typeN  champN;  
};
```

**Effet** : déclare un nouveau type d'union.

# Exemple d'union

## Exemple

```
union nombre {  
    int      entier;  
    double   flottant;  
};
```

Occupation en mémoire : 8 octets.

# Variables unions

Les unions se déclarent et s'utilisent comme des structures . . .

## Exemple

```
union nombre nb;  
nb.flottant = 12;  
nb.flottant *= 2;
```

. . . avec une différence importante :

## Attention

Écrire dans un champ rend invalide les autres champs.

## Exemple d'utilisation invalide

On ne doit pas écrire dans un champ puis lire depuis un autre !

### Exemple faux

```
union nombre nb;  
  
nb.flottant = 12.; /* OK */  
nb.entier    = 42;  /* OK */  
x = nb.flottant; /* erreur! */
```

⇒ il faut se souvenir du champ actif !

# Utilisation pratique des unions

En pratique, on se sert d'un **discriminant** pour se souvenir du champ actif.

## Exemple

```
typedef union { int ent; double flot; } val_nb;  
typedef struct { int type; val_nb val; } nombre;  
nombre nb;
```

**Mode d'emploi :** le champ entier **type** sert de discriminant

- si `nb.type==0`, alors on utilise `nb.val.ent`,
- si `nb.type==1`, alors on utilise `nb.val.flot`.

# Unions de structures

**Autre utilisation courante : champs structures.**

## Exemple

```
typedef struct { int type; int x; ... } truc;
typedef struct { int type; int y; ... } machin;
typedef union {
    int      type;
    truc    truc;
    machin  machin;
} machintruc;
machintruc b;
```

- on se sert de `b.type` pour indiquer si on a affaire à un `truc` ou un `machin`,
- `b.type`, `b.truc.type` et `b.machin.type` sont interchangeables et représentent le même objet mémoire.

# Initiation au C

## Cours n°8

Antoine Miné<sup>1</sup> Ozan Caglayan<sup>2</sup>

<sup>1</sup>École Normale Supérieure

<sup>2</sup>Université Galatasaray

19 février 2015



# Plan du cours

- Conversions de types : l'opérateur **(type)**.
- Allocation dynamique de mémoire : **malloc**, **realloc**, **free**.

# Conversions de types

---

# Rappels sur les types entiers

## Types entiers sur un Intel 32-bit

type	sizeof	ensemble représenté
char, signed char	1	[−128, 127]
unsigned char	1	[0, 255]
short	2	[−32768, 32767]
unsigned short	2	[0, 65535]
int, long	4	[− $2^{31}$ , $2^{31} - 1$ ]
unsigned, unsigned long	4	[0, $2^{32} - 1$ ]
long long	8	[− $2^{63}$ , $2^{63} - 1$ ]
unsigned long long	8	[0, $2^{64} - 1$ ]

Calculs dans les entiers :

- non-signés : calcul modulo  $2^{8 \times \text{sizeof}}$ ,
- signés : résultat aléatoire en cas de dépassement de capacité,
- erreur à l'exécution en cas de division par 0 (/ ou %).

# Rappels sur les types flottants

## Types flottants sur un Intel 32-bit

type	sizeof	min	max	précision
float	4	$10^{-45}$	$3 \times 10^{38}$	$10^{-7}$
double	8	$5 \times 10^{-324}$	$10^{308}$	$10^{-16}$
long double	12	$10^{-4933}$	$10^{4912}$	$10^{-19}$

Calculs dans les flottants :

- calculs arrondis à la précision du type,
- nombres spéciaux :  $+\infty$ ,  $-\infty$ ,  $\text{NaN}$ .

# Choix d'un type arithmétique

Le type donne un compromis entre l'expressivité et le coût (mémoire et temps).

Le type définit la sémantique des opérations.

## Exemple

```
int a,b,c;
double x,y,z;

c = a / b; /* division entière (troncature) */
z = x / y; /* division flottante */

c = a % b; /* modulo entier */
z = x % y; /* ERREUR */
```

La sémantique est donnée par le type des **arguments** de l'opérateur.

# Conversions explicites de types

Opérateur de conversion de type (*cast* en anglais).

## Syntaxe

(type) *expr*

**Effet** : convertit la valeur de *expr* dans le type précisé.

### Exemple

```
int x,y;  
double d;  
d = (double)x / (double)y; /* division flottante */
```

**Attention** à la forte priorité de l'opérateur (type) :

Exemple : (int)x+y signifie ((int) x)+y et non (int)(x+y).

# Effet sur les valeurs

## Règles de conversion :

- si la valeur est représentable exactement dans type,  
    ⇒ la valeur est **inchangée**,
- en cas de conversion flottant → entier,  
    ⇒ la valeur est **tronquée**,
- en cas de dépassement de capacité :
  - si type est flottant : le résultat est  $+\infty$  ou  $-\infty$ ,
  - si type est non signé : le résultat est pris **modulo  $2^{8 \times \text{sizeof}}$** ,
  - si type est signé : le résultat **dépend de la machine**.

### Exemples

```
(int) (1.5 + 0.5)      /* donne 2 */  
(int) (1.0 + 0.5)      /* donne 1 */  
(unsigned char) (-1)   /* donne 255 */
```

# Conversions implicites de types

**Conversions implicites** : ajoutées par le C.

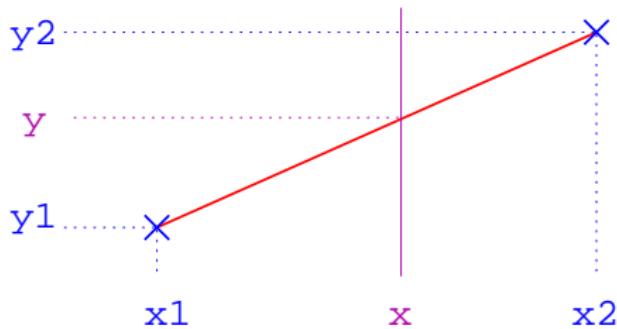
- lors des **affectations, initialisations** : vers le type de la *lvalue*,

Exemple

```
double d = 1;    conversion de 1 de int vers double
int x = d+0.5;  troncature de d+0.5 de double vers int
```

- lors des appels de fonctions : vers le **type des arguments**,  
Exemple : `fputf(2.5, f);` (exception : `printf`)
- lors des opérations binaires : **promotion vers un type commun**,  
`int < long < long long < float < double < long double`  
Exemple : `int x=6; x=x*0.5;` multiplication en double.

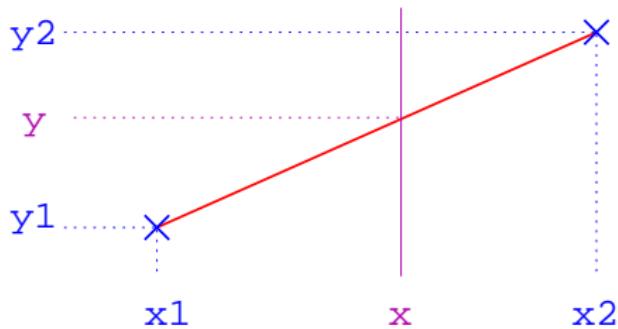
## Exemple : interpolation linéaire



### Calculs en entiers

```
int interpol(int x1, int y1, int x2, int y2, int x)
{
    return y1 + ((y2-y1) / (x2-x1)) * x;
}
```

## Exemple : interpolation linéaire



### Calculs en flottants

```
int interpol(int x1, int y1, int x2, int y2, int x)
{
    return y1 + ( (double) (y2-y1) / (x2-x1)) * x;
}
```

# Conversions de types pointeurs

On peut convertir d'un type pointeur en un autre type pointeur, avec la même syntaxe :

## Syntaxe

(type) expr

### Effet :

- ne change pas l'adresse pointée,
- change l'interprétation des opérateurs :
  - +, -, etc. décalent par unité de `sizeof(type)` octets,
  - \* extrait un objet de type `type`.

# Applications des conversions de types pointeurs

Deux applications principales :

- conversion avec le type générique **void\***,

Exemple : allocation de mémoire dynamique,

- conversion avec le type **unsigned char\***,

⇒ permet l'accès à la représentation mémoire en octets.

## Exemple

```
int x = 386, i;  
unsigned char* p = (unsigned char*) &x;  
for ( i=0; i<sizeof(x); i++ ) printf("%i ", p[i]);
```

donne sur un Intel 32-bit : 130 1 0 0.

# Allocation dynamique de mémoire

---

# Rappels sur la gestion de la mémoire

**Variables** : méthode la plus simple de réservation de mémoire.

Caractéristiques :

- variables globales : accessibles par tout le programme,  
variables locales : détruites en fin de bloc,
- la taille est fixée à la déclaration (par le **type**),
- tableaux globaux : la taille est une constante fixée “en dur”,  
tableaux locaux : la taille peut être une expression.

⇒ pas toujours suffisamment flexible !

Exemple : comment avoir un tableau

- dont la taille est calculée dans une fonction,
- dont la durée de vie dépasse celle de la fonction,
- dont la taille peut être changée *a posteriori* ?

# Les blocs de mémoire dynamique

Solution : **mémoire dynamique.**

- blocs de mémoire de taille arbitraire,
- alloués et libérés explicitement par l'utilisateur,
- redimensionnables (explicitement),
- accessibles par **pointeur**.

Tas (*heap*) = zone mémoire où est allouée la mémoire dynamique.

# Allocation d'un bloc avec malloc

## Syntaxe

```
#include <stdlib.h>
void * malloc(size_t size);
```

## Effet :

- alloue un nouveau bloc de mémoire de **size octets**,
- renvoie un pointeur sur le début (1ère octet) du bloc,
- renvoie **NULL** en cas d'erreur (mémoire insuffisante).

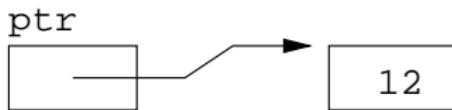
## Comment utiliser malloc :

- calculer la taille à allouer grâce à **sizeof()**,
- vérifier que la valeur de retour n'est pas NULL,
- stocker le pointeur dans une **variable pointeur du bon type**,
- accéder au bloc grâce au pointeur : **\* , [ ]**.

# Exemple d'allocation

## Exemple

```
double* ptr = malloc( sizeof( double ) );
assert( ptr!=NULL );
*ptr = 12.0;
*ptr += 2.0;
```



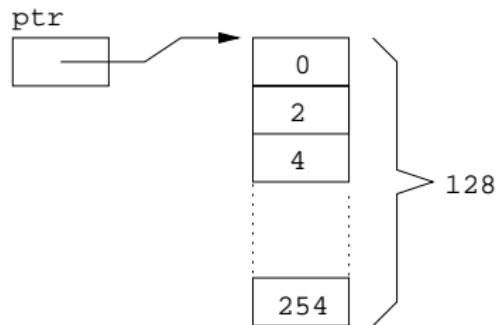
## Notes :

- la conversion `void*` → `double*` est implicite,
- on utilise `assert` pour vérifier que `ptr` n'est pas `NULL`.

# Exemple d'allocation de tableau

## Exemple

```
int i;  
double* ptr = malloc( sizeof( double ) * 128 );  
assert(ptr);  
for ( i=0; i<128; i++ ) ptr[i] = i * 2.0;
```



Attention

Ne pas dépasser de la taille allouée !

# Durée de vie d'un bloc

Le bloc reste alloué en mémoire jusqu'à être libéré explicitement.

## Tableau local

```
int* truc(int n)
{
    int x[n];
    ...
    return &x[0];
}
```

FAUX !

## Tableau dynamique

```
int* truc(int n)
{
    int* y = malloc(sizeof(int)*n);
    ...
    return y;
}
```

Correct.

## Ne pas confondre

Durée de vie du bloc et durée de vie du pointeur y.

# Allocation dynamique de chaînes

## Fonction

```
#include <string.h>
char* strdup(const char* s);
```

**Effet :** renvoie une copie de *s* dans un bloc alloué dynamiquement, ou NULL (mémoire insuffisante).

## Fonction équivalente à strdup

```
char* mon_strdup(const char* s)
{
    char* x = ????
    ???
    return x;
}
```

# Allocation dynamique de chaînes

## Fonction

```
#include <string.h>
char* strdup(const char* s);
```

**Effet :** renvoie une copie de *s* dans un bloc alloué dynamiquement, ou NULL (mémoire insuffisante).

## Fonction équivalente à strdup

```
char* mon_strdup(const char* s)
{
    char* x = malloc( strlen(s) + 1 );
    if (x) strcpy( x, s );
    return x;
}
```

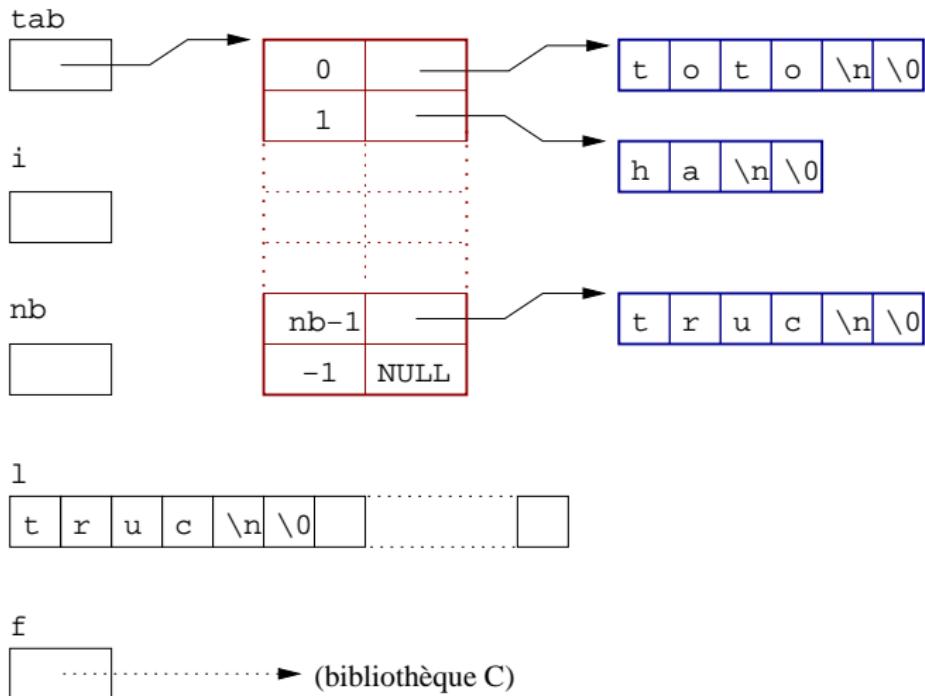
# Exemple complexe

## Exemple

```
typedef struct { int n; char* s; } ligne;

ligne* lit( FILE* f )
{
    char l[256];
    int i, nb;
    ligne* tab;
    fscanf( f, "%i", &nb );
    tab = malloc( sizeof(ligne) * (nb+1) );
    for ( i=0; i<nb; i++ ) {
        fgets( l, 256, f );
        tab[i].n = i; tab[i].s = strdup( l );
    }
    tab[i].n = -1; tab[i].s = NULL;
    return tab;
}
```

## Illustration de l'exemple complexe



# Libération d'un bloc avec free

La mémoire est libérée automatiquement en fin du programme.

On peut toutefois libérer un bloc manuellement :

## Syntaxe

```
void free(void *ptr);
```

## Effet :

- `ptr` doit pointer sur un bloc créé par `malloc`,
- après `free`, le bloc n'est plus utilisable,
- la mémoire libérée est recyclée et réutilisable par `malloc`.

# Exemple d'utilisation de free

## Exemple

```
typedef struct { int n; char* s; } ligne;

void libere_tab( ligne* tab )
{
    int i;
    for ( i=0; tab[i].nb > -1; i++ ) free( tab[i].s );
    free( tab );
}
```

Si on oublie de libérer `tab[i].s`, on a des **fuites de mémoire** !

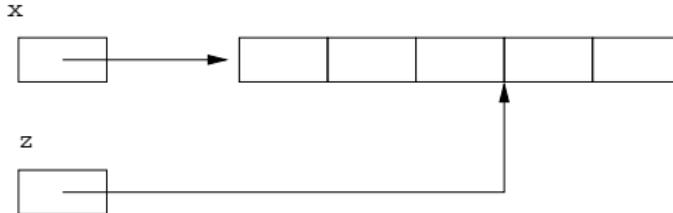
# Notion d'*aliasing*

## Aliasing

Pointeurs qui pointent dans le même bloc.

## Exemple

```
int* x = malloc( sizeof(int) * 5 );
int* z = x+3;
free(x);
*z = 12; /* erreur */
```



# Redimensionnement de bloc avec realloc

## Syntaxe

```
void* realloc(void *ptr, size_t size);
```

## Effet :

- ptr doit pointer sur un bloc créé par malloc,  
(ou retournée par realloc),
- change la taille du bloc en size,
- renvoie la nouvelle adresse du bloc (ou NULL),  
    ⇒ l'ancienne adresse ptr ne doit plus être utilisée,
- le contenu du bloc n'est pas changé.

## Exemple d'utilisation de realloc

### Exemple

```
int taille = 128;
int* tab = malloc( sizeof(int) * taille );
...
if ( i>=taille ) {
    taille = i+1;
    tab = realloc( tab, sizeof(int) * taille );
}
tab[i] = 12;
```

**Note :** il n'y a aucun moyen de connaître la taille d'un bloc,  
⇒ on s'en souvient dans une variable annexe taille.

# L'outil valgrind

valgrind = détecte (entre autres) les erreurs de mémoire :

- les accès en dehors des bornes d'un bloc dynamique,
- les free et realloc incorrects,
- les fuites de mémoire.

## Utilisation

```
$ gcc toto.c -Wall -Wextra -g  
$ valgrind ./a.out
```

## Notes :

- comme pour gdb, l'option de compilation -g est conseillée,
- contrairement à gdb, valgrind est non interactif.

# Initiation au C

## Cours n°9

Antoine Miné<sup>1</sup> Ozan Caglayan<sup>2</sup>

<sup>1</sup>École Normale Supérieure

<sup>2</sup>Université Galatasaray

19 février 2015



# Plan du cours

- compilation séparée,
- modularité,
- bibliothèques .a et .so,
- recompilation automatique avec `make`.

# Compilation séparée

---

# Étapes de la compilation

**Compilation** = génération d'un programme exécutable  
à partir d'un fichier .c.

Se décompose en plusieurs étapes.

## Étapes de la compilation

- pré-traitement (gestion de #include, #define),
- analyse syntaxique,
- typage,
- génération de code assembleur,
- assemblage,
- édition de liens (ajout de la bibliothèque C).

# Programmes utilisés lors de la compilation

Plusieurs programmes interviennent lors de la compilation :

## Chaîne de compilation

### Pré-traiteur : `cpp`

- pré-traitement (gestion de `#include`, `#define`),

### Compilateur C : `cc1`

- analyse syntaxique,
- typage,
- génération de code assembleur,

### Assembleur : `as`

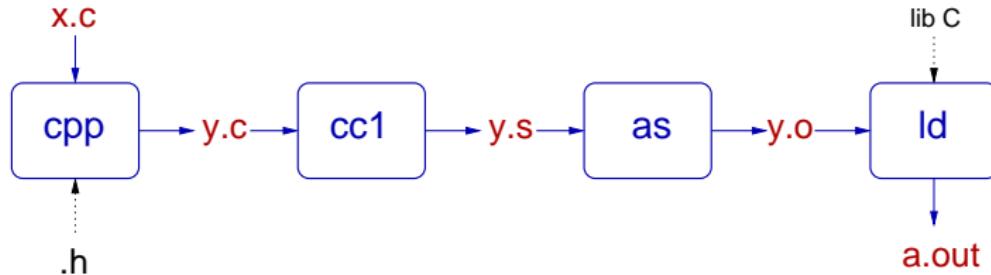
- assemblage,

### Éditeur de liens : `ld`

- édition de liens (ajout de la bibliothèque C).

# Chaîne de compilation

Ces programmes communiquent par des fichiers intermédiaires :



## Types de fichiers :

- **.c** : fichier source C, avant et après pré-traitement,
- **.s** : fichier source assembleur,
- **.o** : **fichier objet**, (.obj sous Windows),
- **a.out** : fichier exécutable (.exe sous Windows).

On parlera également des bibliothèques : **.a**, **.so**, **.dll**.

# Compilation simplifiée grâce à gcc

En pratique, on n'appelle pas cpp, cc1,... à la main, on utilise **gcc**.

**gcc** = interface sur la chaîne de compilation.

## Ligne de commande classique

```
$ gcc toto.c -Wall -Wextra
```

### Effet :

- pré-traite, compile, assemble et lie,
- détruit les fichiers intermédiaires .c, .s et .o après usage,  
⇒ on obtient automatiquement un exécutable.

**Note :** gcc détermine les actions à effectuer grâce à l'extension **.c**.

# Compilation séparée

**Compilation séparée** : on décompose en **deux** étapes

- 'compilation' = pré-traitement + compilation + assemblage,
- édition de liens.

## Exemple

```
$ gcc -c toto.c -Wall -Wextra  
$ gcc toto.o
```

## Effet :

- ➊ **gcc -c** compile et génère un fichier objet **toto.o**,  
(pas d'édition de liens, pas d'exécutable généré)
- ➋ **gcc** lie le **.o** en argument et génère un exécutable **a.out**.  
(pas de compilation, **toto.c** n'est pas examiné)

**Note** : le fichier **toto.o** n'est pas détruit à la fin du processus.

Compilation séparée

## Modularité

Automatisation de la compilation avec make

Compilation multi-fichiers

Utilisations des en-têtes

Les bibliothèques

# Modularité

---

# Programmes multi-fichiers

Un programme peut être composé de plusieurs sources .c.

## Avantages

- facilite l'écriture et la compréhension de gros programmes,
- facilite le travail à plusieurs,
- rend les recompilations plus rapides (compilation séparée),
- permet la modularité et l'abstraction,
- permet la réutilisation dans d'autres projets.

Exemple : noyau Linux 2.6.18

- 8531 fichiers .c,
- taille moyenne : 646 lignes,      taille totale : 5.5 Mlignes,
- taille maximale : 18227 lignes,      taille médiane : 330 lignes.

# Compilation multi-fichiers

**Méthode simple** : en une seule étape.

## Exemple

```
$ gcc main.c utils.c mon_print.c -Wall -Wextra
```

## Effet :

- pré-traite, compile, assemble et lie tous les .c,
- génère un exécutable a.out,
- efface tous les .o et autres fichiers intermédiaires.

## Notes :

- l'ordre des fichiers et des options n'est pas important,
- si un seul .c change, gcc recompile tout !

# Compilation séparée multi-fichiers

Méthode avancée : compilation **séparée**.

## Exemple

```
$ gcc -Wall -Wextra -c main.c
$ gcc -Wall -Wextra -c utils.c
$ gcc -Wall -Wextra -c mon_print.c
$ gcc main.o utils.o mon_print.o
```

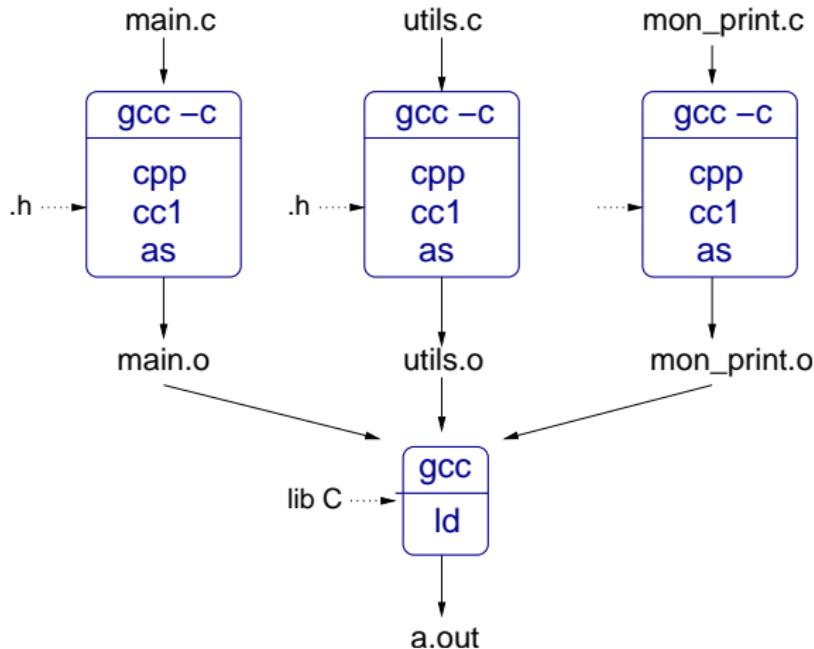
## Effet :

- compile chaque .c en un .o avec gcc -c,
- lie tous les .o en un exécutable a.out.

## Attention

Avec -c, on ne peut compiler qu'un fichier à la fois !

# Diagramme de compilation séparée multi-fichiers



# Bénéfice de la compilation séparée

## Avantage

Les fichiers .o restent disponibles.

En cas de modification du programme :

- seuls les fichiers .c modifiés doivent être recompilés,
- l'édition de liens doit également être refaite.  
⇒ gain de temps.

Exemple : après modification de main.c

```
$ gcc -Wall -Wextra -c main.c  
$ gcc main.o utils.o mon_print.o
```

(automatisation possible grâce à make)

# Contenu d'un fichier objet

Un fichier objet .o contient une **table de symboles** :

- **variables globales** avec leur taille et valeur d'initialisation,
- **fonctions** compilées en langage machine.

Le format des fichiers .o est :

- binaire, (consultable par objdump -Ds),
- non portable, (dépend du type de processeur et d'OS),
- standard pour tous les langages compilés,  
(inter-opérabilité avec le C++, le OCaml, l'assembleur, etc.)
- non typé,  
(types des variables et prototypes des fonctions perdu).

# Symboles non définis et édition de liens

Un fichier .o n'est pas un programme complet.

Il peut référencer des **symboles non définis**.

Exemples :

- fonction ou variable dans une **bibliothèque**,  
(e.g. : printf dans la bibliothèque standard,  
          sin     dans la bibliothèque mathématique)
- fonction ou variable définie dans **un autre fichier .o**.

**Édition de liens :**

- pioche dans les .o et les bibliothèques passés en argument,
- résout les symboles non définis,    (ou indique une erreur)
- **comportement indéfini en cas de définitions multiples**,
- génère un exécutable (presque) autonome.

# Notion d'unité de compilation en C

**Unité de compilation C** = source .c qui sera compilé en un .o.

Quelle que soit la méthode de compilation employée,  
chaque fichier .c est **compilé indépendamment** en un .o :

- pas d'accès aux autres fichiers sources .c,
- pas d'accès aux fichiers objets .o ni aux bibliothèques.

Les variables et fonctions non définies dans le .c se retrouvent comme symboles non définis dans le .o...

## Conséquence

Pour obtenir un .o correct, le .c doit préciser les **types** des variables et **prototypes** des fonctions utilisées mais non définies :  
à défaut de définir, il faut déclarer.

## Exemple incorrect

mon\_print.c

```
#include <stdio.h>
#include <stdlib.h>
void err(const char*s) {
    printf("%s!!!\n",s);
    exit(1);
}
```

utils.c

```
#include <stdio.h>
void lit(const char*s) {
    FILE* f = fopen(s,"r+");
    if (!f) err("open");
    while (fget(...))
        ...
}
```

main.c

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    if (argc<2) err("argc");
    lit(argv[1]);
    ...
}
```

- alerte de typage à la compilation (avec -Wall),
- pas d'erreur de liaison, mais programme généré incorrect !

# Premier essai de correction : ajout de prototypes

mon\_print.c

```
#include <stdio.h>
#include <stdlib.h>
void err(const char*s) {
    printf("%s!!!\n",s);
    exit(1);
}
```

utils.c

```
#include <stdio.h>
void err(const char*s);
void lit(const char*s) {
    FILE* f = fopen(s,"r");
    if (!f) err("open");
    while (fget(...))
        ...
}
```

main.c

```
#include <stdio.h>
void err(const char*s);
void lit(const char*s);
int main(int argc, char* argv[]) {
    if (argc<2) err("argc");
    lit(argv[1]);
    ...
}
```

# Meilleur correction : utilisation d'une en-tête .h

mon\_print.c

```
#include "header.h"
```

```
void err(const char*s) {
    printf("%s!!!\n",s);
    exit(1);
}
```

main.c

```
#include "header.h"
```

```
int main(int argc, char* argv[])
{
    if (argc<2) err("argc");
    lit(argv[1]);
    ...
}
```

utils.c

```
#include "header.h"
```

```
void lit(const char*s) {
    FILE* f = fopen(s,"r");
    if (!f) err("open");
    while (fget(...))
        ...
}
```

header.h

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

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

```
void err(const char*s);
void lit(const char*s);
```

# Utilisation des en-têtes .h

Inclusion d'une en-tête utilisateur :

## Syntaxe

```
#include "fichier"
```

## Effet :

- cherche *fichier* dans le **répertoire courant**,
- **remplace** la ligne `#include` par le **contenu** du fichier.

Similaire à `#include <...>`, sauf pour le répertoire de recherche.

## Avantages :

- pas de recopie inutile de prototypes,
- un seul endroit à mettre à jour en cas d'évolution des .c,
- une en-tête **documentée** donne un *résumé* des fonctionnalités.

# Contenu d'une en-tête

Une en-tête .h est un **fichier C** avec pour conventions tacites :

- l'en-tête n'est pas compilée, elle est #inclusée,
- l'en-tête contient :
  - des **définitions de types** et de macro-instructions,
  - des **déclarations de fonctions (prototypes)** et **variables**,
  - l'inclusion d'autres en-têtes, standards ou utilisateurs.

## Attention

Ne pas mettre de **définitions** de fonctions  
ou de variables dans une en-tête.

Si l'en-tête est inclusé par plusieurs .c, toute déclaration sera dupliquée dans plusieurs .o  $\Rightarrow$  problèmes à l'édition de liens !

# Déclarations de variables extern

Pour une déclarer une **variable globale** sans la définir,  
on utilise le mot-clé **extern**.

Exemple :

truc.h

```
/* déclaration */  
extern int toto;
```

truc.c

```
#include "truc.h"  
  
/* définition */  
int toto = 2;
```

main.c

```
#include "truc.h"  
...  
/* utilisation */  
toto = 12;
```

Dans l'unité de compilation truc.c,  
toto est déclaré **extern** et défini,  
⇒ permet à gcc de vérifier la cohérence.

**extern** est facultatif pour les déclarations de fonctions (prototypes).

# Variables globales et fonctions static

**Symbole statique** = local à l'unité de compilation :

- non visible depuis les autres unités de compilation,
- plusieurs unités de compilation peuvent avoir des symboles statiques de même nom,
- fonctionne pour les variables globales et les fonctions.  
(Pour les variables locales, static a un autre sens...)

Exemple :

truc.c

```
#include "header.h"
static int nb = 0;
void truc()
{
    nb++;
    ...
}
```

bidule.c

```
#include "header.h"
static int nb = 12;
void bidule()
{
    nb++;
    ...
}
```

header.h

```
void truc();
void bidule();
```

# Utilisation des en-têtes .h

Exemple : noyau Linux 2.6.18

- 8613 fichiers .h,
- taille moyenne : 154 lignes, total : 1.3 Mlignes.

L'utilisation des en-têtes est très libre :

- on peut utiliser une en-tête X.h pour chaque fichier X.c,  
ou une seule en-tête pour tout le projet,  
ou regrouper les déclarations dans des .h thématiques.
- l'en-tête peut omettre certains types et déclarations.

## Exemple d'application

---

# Rappels sur les types struct

**Rappel struct = types enregistrements.**

## Définition de type

```
struct s {  
    type1  champ1;  
    :  
    typeN  champN;  
};
```

**Effet :** définit un nouveau **type** de structure

- de nom *s*,
- de champs nommés *champ1* à *champN*,
- les champs ont pour type *type1* à *typeN*.

# Types struct incomplets

Il est possible de déclarer un type sans le définir.

Déclaration de type  
struct s;

Déclare l'existence du type **struct s** sans préciser ses champs.

Le type est **incomplet** :

- on ne peut pas déclarer de variable de type **struct s** :  
 e.g. : `struct s truc;`  
`void affiche_s(struct s truc);`
- on peut déclarer une variable de type **struct s \*** :  
 e.g. : `struct s* ptr;`  
`void affiche_s(struct s* ptr);`
- on ne peut pas déréférencer une variable de type **struct s \*** :  
 e.g. : `*ptr, ptr->titi.`

# Application à l'abstraction de types

compte.c

```
struct compte { int num; ... };

void affiche(struct compte* c)
{
    printf("%i\n", c->num);
    ...
}

struct compte* cree()
{
    struct compte* c;
    c = malloc(sizeof(*c));
    ...
    return c;
}
```

compte.h

```
struct compte;

void affiche
    (struct compte* c);

struct compte* cree();
```

main.c

```
#include "compte.h"
...
struct compte* x;
x = cree();
affiche(x);
free(x);
```

## Les bibliothèques

---

# Bibliothèques

**Bibliothèque** = archive de symboles compilés.

Ressemble aux fichiers objets .o, mais :

- regroupe plusieurs unités de compilation dans un seul fichier,  
    ⇒ facilite la distribution et l'utilisation,
- optimisée pour être liée de nombreuses fois.

Un fichier bibliothèque :

- a l'extension **.a** (statique) ou **.so** (dynamique),  
(.dll sous Windows)
- commence toujours par **lib** (e.g. : libm.so, libgmp.a),
- se trouve généralement dans le répertoire **/usr/lib**.

On expliquera ici l'utilisation des bibliothèques, pas leur création...

# Utilisation de bibliothèques prédéfinies

**Option d'édition de liens : `-l` *bibli* :**

*bibli* est le nom de la bibliothèque :

- sans l'extension,
- sans le préfixe lib.

**Cas particulier :**

la bibliothèque C (libc.so) est toujours liée par défaut.  
(option `-nostdlib` pour *ne pas* la lier)

**Note :** un fichier bibliothèque vient généralement accompagné de son lot d'en-têtes .h.

# Exemple : utilisation de la bibliothèque mathématique

**Bibliothèque mathématique** = libm.so ou libm.a :

Contient la définition des fonctions de **math.h**.  
(sin, cos, pow, etc.)

## Compilation séparée

```
$ gcc -c exemple.c -Wall -Wextra  
$ gcc exemple.o -lm
```

## Compilation en une passe

```
$ gcc exemple.c -Wall -Wextra -lm
```

# Options usuelles de compilation de gcc

## Options de compilation de gcc

<b>-c</b>	compilation seule, pas de liaison
<b>-Wxxxx</b>	alertes supplémentaires à la compilation e.g. : -Wall, -Wextra
<b>-Ox</b>	optimisation e.g. : -O, -O1, -O2, -O3, -Os
<b>-g</b>	ajout d'informations de débogage
<b>-I répertoire</b>	où #include cherche les en-têtes
<b>-Dvar</b>	équivalent à #define var
<b>-Dvar=val</b>	équivalent à #define var val

# Options usuelles de liaison de gcc

## Options de liaison de gcc

<b>-o <i>fichier</i></b>	l'exécutable s'appellera <i>fichier</i> au lieu de a.out
<b>-l<i>lib</i></b>	lie avec la bibliothèque <i>lib</i>
<b>-L <i>dossier</i></b>	où -L cherche les bibliothèques

# Exemple de compilation complexe

**Exemple :** programme utilisant la bibliothèque **gmp** installée chez l'utilisateur : gmp.h + libgmp.a.

## Compilation

```
$ gcc -c proj1.c -Wall -Wextra -I /users/mine/include
$ gcc -c proj2.c -Wall -Wextra -I /users/mine/include
$ gcc -o proj proj1.o proj2.o -L /users/mine/lib -lgmp -lm
```

**Note :** l'ordre des options -l peut être important...

# Automatisation de la compilation avec `make`

---

# L'outils make

**make** = utilitaire de compilation automatique :

- lit une liste de **règles de compilation** dans un fichier **Makefile**,
- compare les dates de dernière modification des fichiers,
- détermine ceux qu'il faut régénérer,
- n'effectue que le strict minimum d'actions.

## Utilisation de make

\$ make

# Format du fichier Makefile

## Forme des règles

*but : prérequis  
commande*

- *but* est le nom du fichier généré,
- *prérequis* est la liste des fichiers dont dépend *but*,
- *commande* est la commande à exécuter pour générer *but*.

**Attention** l'espacement compte dans les règles :

- *commande* est précédé d'un caractère **tabulation**,
- les lignes ne doivent pas être coupées.

# Exemple de Makefile

## Makefile

```
proj: proj1.o proj2.o  
        gcc -o proj proj1.o proj2.o -lgmp -lm -L ...  
  
proj1.o: proj1.c  
        gcc -c proj1.c -Wall -Wextra -I ...  
  
proj2.o: proj2.c  
        gcc -c proj2.c -Wall -Wextra -I ...
```

**Effet de make :** cherche à régénérer proj.

Si nécessaire, commence par régénérer proj1.o et proj2.o.

**Note :** on peut simplifier le Makefile en utilisant des variables et des règles génériques...

# Initiation au C

## Cours n°10

Antoine Miné<sup>1</sup> Ozan Caglayan<sup>2</sup>

<sup>1</sup>École Normale Supérieure

<sup>2</sup>Université Galatasaray

19 février 2015



# Plan du cours

- pointeurs de fonctions,
- structures de données dynamiques :
  - listes simplement chaînées,
  - listes doublement chaînées.

# Pointeurs de fonctions

---

# Rappels sur les prototypes de fonctions

**Prototype = déclaration** précisant :

- le **nom** de la fonction,
- son **type de retour**, ou **void**,
- le nombre et le **type de ses arguments**.

## Exemples

```
int additionne(int x, int y);  
void affiche(char*);
```

Différences avec une définition de fonction :

- le corps est omis (remplacé par ;),
- le nom des arguments est facultatif,
- une fonction peut être déclarée plusieurs fois.

# Rappels sur les prototypes de fonctions

## Utilité du prototype :

information nécessaire pour compiler un **appel** de fonction.

Si un prototype est donné, la fonction peut être définie

- après l'appel, ou
- dans une autre unité de compilation.

## Exemple :

a.c

```
int fun1(int x);

int fun2(void)
{ return fun1(1); }

int fun1(int x)
{ return x+1; }
```

b.c

```
int fun2(void);

int main()
{ return fun2(); }
```

# Pointeurs de fonctions

Chaque fonction a une adresse en mémoire.

**Pointeur de fonction = variable**

- contenant l'**adresse** d'une fonction,
- dont le type indique le **prototype de la fonction**.

Déclaration d'un pointeur de fonction **ptr**

**type-ret (\*ptr)(type1,...,typeN);**

**ptr** peut contenir l'adresse de toute fonction

- prenant N arguments, de types type1 à typeN, et
- retournant une valeur de type type-ret.

# Affectation de pointeur de fonction

**Affectation :** `ptr = f;` où

- `f` est le nom d'une fonction de prototype compatible avec `ptr`,
- `f` est un pointeur de fonction de même type que `ptr`.

## Exemples d'affectation

```
int truc(int x)
{ ... }

int bidule(int y);

void main()
{
    int (*ptr)(int);
    ptr = truc;
    ptr = bidule;
}
```

# Appel de fonction par pointeur

**Appel par pointeur** : identique à un appel de fonction classique

- `ptr(arg1,...,argN);`   ou
- `x = ptr(arg1,...,argN);`

## Exemple d'appel

```
int bidule(int y);

int main()
{
    int (*ptr)(int);
    ptr = bidule;
    return ptr(2);
}
```

# Application : fonction d'ordre supérieur

Fonction paramétrée par une fonction.

⇒ on passe un pointeur de fonction en argument.

## Exemple

```
void affiche_tableau(char* tab[], void (*f)(char*)) {  
    int i;  
    for (i=0; tab[i]; i++) f(tab[i]);  
}  
  
void affiche(char* s)  
{ printf("%s\n", s); }  
  
void main() {  
    char* t[] = { "toto", "titi", NULL };  
    affiche_tableau(t, affiche);  
}
```

# Listes simplement chaînées

---

# Principe

**Liste** = **séquence ordonnée** d'éléments de même type.

Exemple : liste d'entiers (12, 43, 27, 9).

- l'ordre des éléments compte :  $(12, 43, 27, 9) \neq (12, 27, 43, 9)$
- la multiplicité des éléments compte  $(12, 43, 27, 9) \neq (12, 43, 27, 9, 9)$

**Liste simplement chaînée** =

représentation où chaque élément **pointe** sur le suivant.



# Représentation des listes

## Implantation en C =

structure de **cellule** pour représenter un élément.

### Type cellule

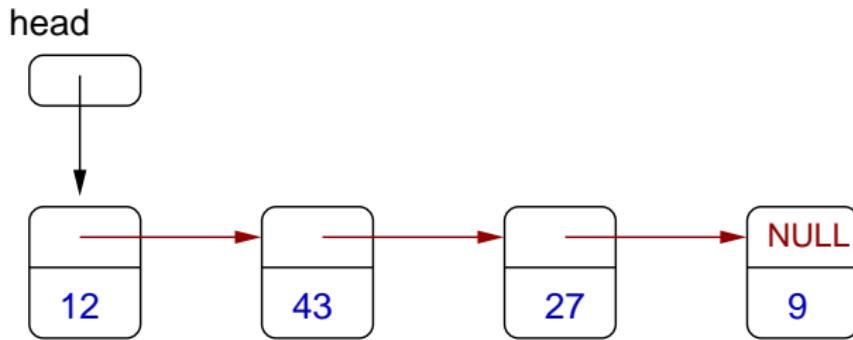
```
struct cell {  
    struct cell* next;  
    int         data;  
};
```

- **data** est le **contenu de la cellule**, (ici, un entier)
- **next pointe vers la cellule suivante**,  
ou vaut **NULL** (fin de liste).

Le type de **cell** est récursif (autoréférentiel).

## Représentation des listes

Une liste est représentée par un **pointeur de tête struct cell\***  
= pointeur sur la première cellule.



Tous les éléments sont **accessibles** depuis la tête de liste.

Par convention, head vaut NULL si **la liste est vide**.

# Opérations sur les listes

**Structure de données = type + algorithmes de manipulation.**

On va développer des fonctions pour les opérations suivantes :

- calcul de la longueur d'une liste,
- recherche d'un élément,
- insertion d'un élément,
- suppression d'un élément,
- concaténation de deux listes,
- destruction d'une liste.

Toutes nos fonctions prennent une tête de liste en argument.

# Calcul de la longueur d'une liste

## Longueur d'une liste

```
int longueur(struct cell* head) {  
    int i; struct cell* c;  
    for ( i=0, c=head; c; i++, c = c->next );  
    return i;  
}
```

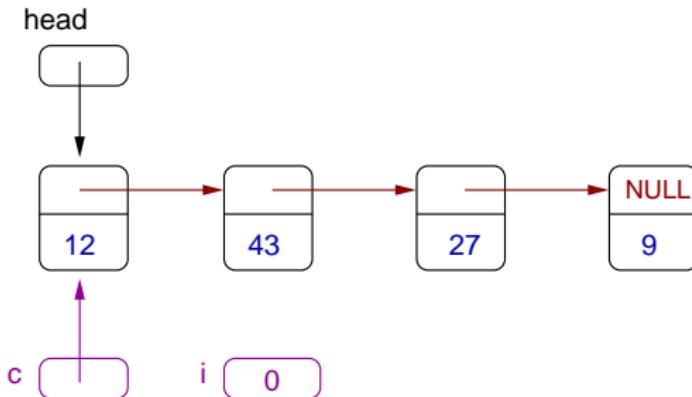
### Principe :

on suit les pointeurs next jusqu'à rencontrer NULL  
et on compte le nombre de cellules rencontrées.

# Calcul de la longueur d'une liste

## Longueur d'une liste

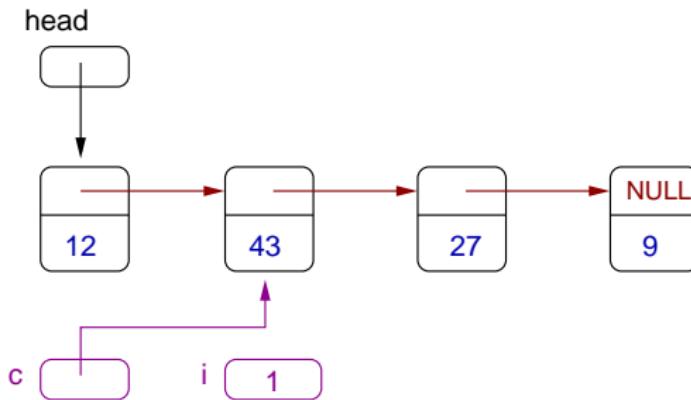
```
int longueur(struct cell* head) {  
    int i; struct cell* c;  
    for ( i=0, c=head; c; i++, c = c->next );  
    return i;  
}
```



# Calcul de la longueur d'une liste

## Longueur d'une liste

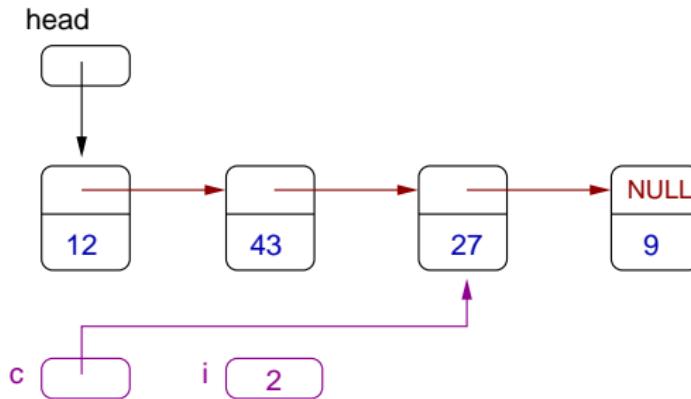
```
int longueur(struct cell* head) {  
    int i; struct cell* c;  
    for ( i=0, c=head; c; i++, c = c->next );  
    return i;  
}
```



# Calcul de la longueur d'une liste

## Longueur d'une liste

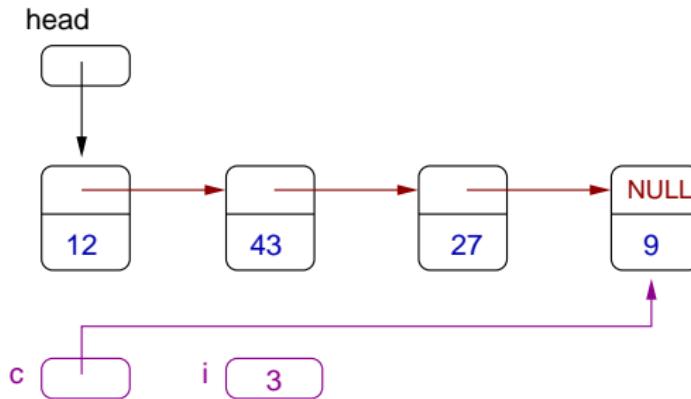
```
int longueur(struct cell* head) {  
    int i; struct cell* c;  
    for ( i=0, c=head; c; i++, c = c->next );  
    return i;  
}
```



# Calcul de la longueur d'une liste

## Longueur d'une liste

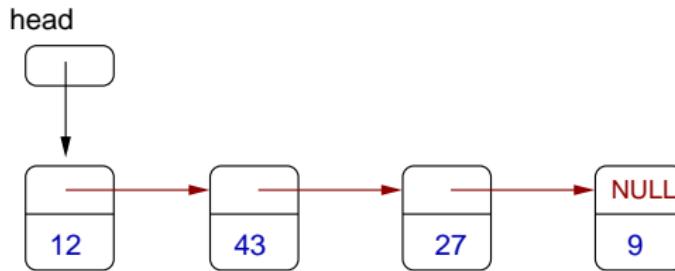
```
int longueur(struct cell* head) {  
    int i; struct cell* c;  
    for ( i=0, c=head; c; i++, c = c->next );  
    return i;  
}
```



# Calcul de la longueur d'une liste

## Longueur d'une liste

```
int longueur(struct cell* head) {  
    int i; struct cell* c;  
    for ( i=0, c=head; c; i++, c = c->next );  
    return i;  
}
```



c

NULL

i

4

# Recherche d'un élément

## Recherche de elem

```
int recherche(struct cell* head, int elem)
{
    while (head) {
        if (head->data == elem) return 1;
        head = head->next;
    }
    return 0;
}
```

## Effet :

renvoie 1 si la liste contient un élément égal à elem, 0 sinon.

# Rappels sur la mémoire dynamique

Pour plus de flexibilité, les cellules sont allouées dynamiquement.

## Gestion de la mémoire dynamique

```
#include <stdlib.h>
void* malloc (size_t size);
void free (void* ptr);
```

### Effet :

- `malloc` alloue un bloc de `size` octets,
- `free` libère le bloc,
- le bloc est accessible uniquement par pointeur.

## Création d'une liste (exemple à ne pas suivre)

Allocation de (12, 43, 27, 9)

```
struct cell* head;
head = malloc(sizeof(struct cell));
head->data = 12;
head->next = malloc(sizeof(struct cell));
head->next->data = 43;
head->next->next = malloc(sizeof(struct cell));
head->next->next->data = 27;
head->next->next->next = malloc(sizeof(struct cell));
head->next->next->next->data = 9;
head->next->next->next->next = NULL;
```

Peu pratique.

On préfère construire une liste par insertions successives.

# Insertion en tête de liste

## Insertion de e en tête de h

```
struct cell* insere(struct cell* h, int e) {  
    struct cell* c = malloc(sizeof(struct cell));  
    c->data = e;  
    c->next = h;  
    return c;  
}
```

## Utilisation

```
struct cell* head = NULL;  
head = insere(head, 9);  
head = insere(head, 27);  
head = insere(head, 43);  
head = insere(head, 12);
```

# Illustration de l'insertion en tête de liste

## Utilisation

```
struct cell* head = NULL;  
head = insere(head, 9);  
head = insere(head, 27);  
head = insere(head, 43);  
head = insere(head, 12);
```

## Insertion

```
struct cell* insere  
(struct cell* h, int e) {  
    struct cell* c = malloc(...);  
    c->data = e;  
    c->next = h;  
    return c;  
}
```

head

NULL

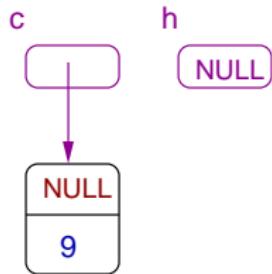
# Illustration de l'insertion en tête de liste

## Utilisation

```
struct cell* head = NULL;  
head = insere(head, 9);  
head = insere(head, 27);  
head = insere(head, 43);  
head = insere(head, 12);
```

## Insertion

```
struct cell* insere  
(struct cell* h, int e) {  
    struct cell* c = malloc(...);  
    c->data = e;  
    c->next = h;  
    return c;  
}
```



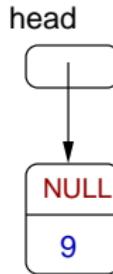
# Illustration de l'insertion en tête de liste

## Utilisation

```
struct cell* head = NULL;  
head = insere(head, 9);  
head = insere(head, 27);  
head = insere(head, 43);  
head = insere(head, 12);
```

## Insertion

```
struct cell* insere  
(struct cell* h, int e) {  
    struct cell* c = malloc(...);  
    c->data = e;  
    c->next = h;  
    return c;  
}
```



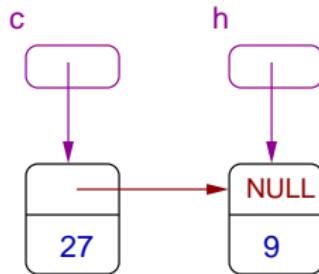
# Illustration de l'insertion en tête de liste

## Utilisation

```
struct cell* head = NULL;  
head = insere(head, 9);  
head = insere(head, 27);  
head = insere(head, 43);  
head = insere(head, 12);
```

## Insertion

```
struct cell* insere  
(struct cell* h, int e) {  
    struct cell* c = malloc(...);  
    c->data = e;  
    c->next = h;  
    return c;  
}
```



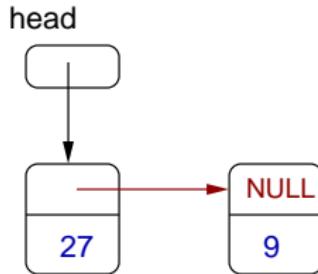
# Illustration de l'insertion en tête de liste

## Utilisation

```
struct cell* head = NULL;  
head = insere(head, 9);  
head = insere(head, 27);  
head = insere(head, 43);  
head = insere(head, 12);
```

## Insertion

```
struct cell* insere  
(struct cell* h, int e) {  
    struct cell* c = malloc(...);  
    c->data = e;  
    c->next = h;  
    return c;  
}
```



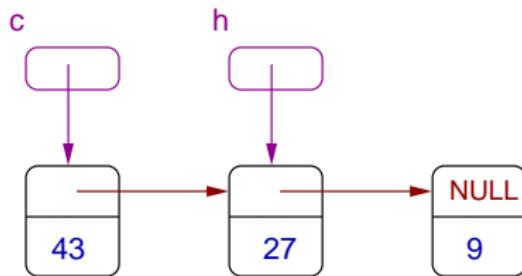
# Illustration de l'insertion en tête de liste

## Utilisation

```
struct cell* head = NULL;  
head = insere(head, 9);  
head = insere(head, 27);  
head = insere(head, 43);  
head = insere(head, 12);
```

## Insertion

```
struct cell* insere  
(struct cell* h, int e) {  
    struct cell* c = malloc(...);  
    c->data = e;  
    c->next = h;  
    return c;  
}
```



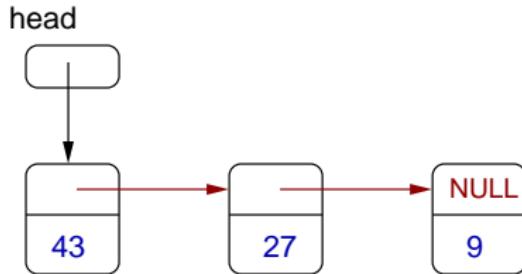
# Illustration de l'insertion en tête de liste

## Utilisation

```
struct cell* head = NULL;  
head = insere(head, 9);  
head = insere(head, 27);  
head = insere(head, 43);  
head = insere(head, 12);
```

## Insertion

```
struct cell* insere  
(struct cell* h, int e) {  
    struct cell* c = malloc(...);  
    c->data = e;  
    c->next = h;  
    return c;  
}
```



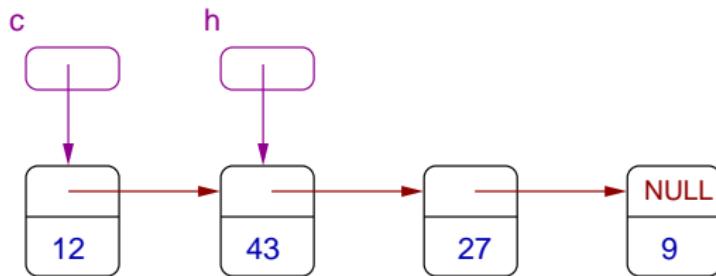
# Illustration de l'insertion en tête de liste

## Utilisation

```
struct cell* head = NULL;  
head = insere(head, 9);  
head = insere(head, 27);  
head = insere(head, 43);  
head = insere(head, 12);
```

## Insertion

```
struct cell* insere  
(struct cell* h, int e) {  
    struct cell* c = malloc(...);  
    c->data = e;  
    c->next = h;  
    return c;  
}
```



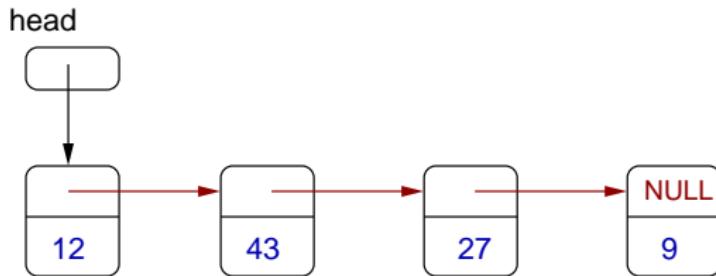
# Illustration de l'insertion en tête de liste

## Utilisation

```
struct cell* head = NULL;  
head = insere(head, 9);  
head = insere(head, 27);  
head = insere(head, 43);  
head = insere(head, 12);
```

## Insertion

```
struct cell* insere  
(struct cell* h, int e) {  
    struct cell* c = malloc(...);  
    c->data = e;  
    c->next = h;  
    return c;  
}
```



# Illustration de l'insertion en tête de liste

## Utilisation

```
struct cell* head = NULL;  
head = insere(head, 9);  
head = insere(head, 27);  
head = insere(head, 43);  
head = insere(head, 12);
```

## Insertion

```
struct cell* insere  
(struct cell* h, int e) {  
    struct cell* c = malloc(...);  
    c->data = e;  
    c->next = h;  
    return c;  
}
```

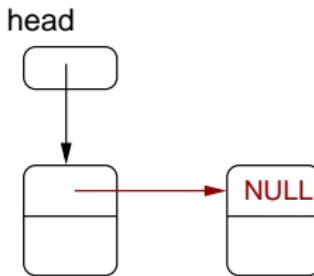
## Remarques :

- la liste est dans l'**ordre inverse** de celui des insertions,
- **insere** fonctionne sur une liste vide ou non-vide,
- la tête de liste est modifiée à chaque insertion,
- le coût d'une insertion est **constant**.

# Insertion en queue de liste

Deux cas à considérer :

- liste vide : on fait pointer **la tête de liste** sur la nouvelle cellule,
- liste non vide : on fait pointer **le champ next de la dernière cellule** sur la nouvelle cellule.



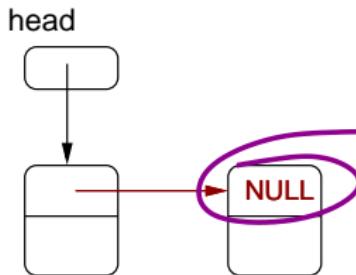
liste vide

liste non vide

# Insertion en queue de liste

Deux cas à considérer :

- liste vide : on fait pointer **la tête de liste** sur la nouvelle cellule,
- liste non vide : on fait pointer **le champ next de la dernière cellule** sur la nouvelle cellule.



liste vide

liste non vide

# Insertion en queue de liste

## Insertion en queue

```
struct cell* insere(struct cell* head, int elem)
{
    struct cell *c = malloc(...);
    c->data = elem;
    c->next = NULL;

    if (head) {
        struct cell* l = head;
        while (l->next) l = l->next;
        l->next = c;
        return head;
    }
    else return c;
}
```

# Exemple d'insertion en queue de liste

## Utilisation

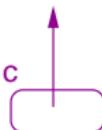
```
struct cell* head = NULL;  
head = insere(head, 12);  
head = insere(head, 43);  
head = insere(head, 27);  
head = insere(head, 9);
```

head

NULL

NULL

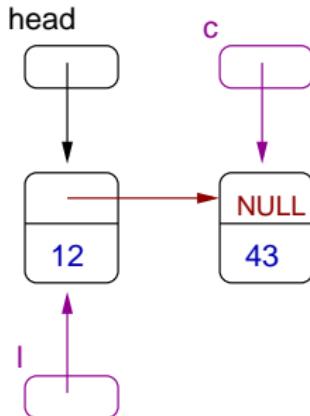
12



# Exemple d'insertion en queue de liste

## Utilisation

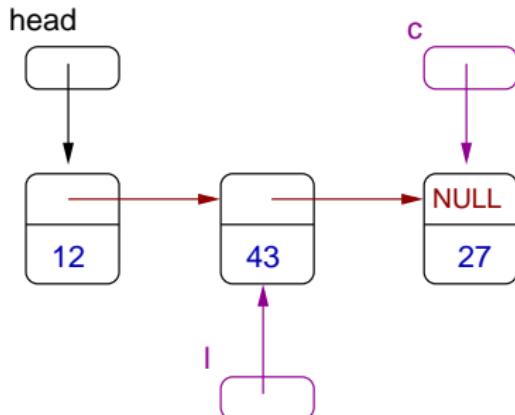
```
struct cell* head = NULL;  
head = insere(head, 12);  
head = insere(head, 43);  
head = insere(head, 27);  
head = insere(head, 9);
```



# Exemple d'insertion en queue de liste

## Utilisation

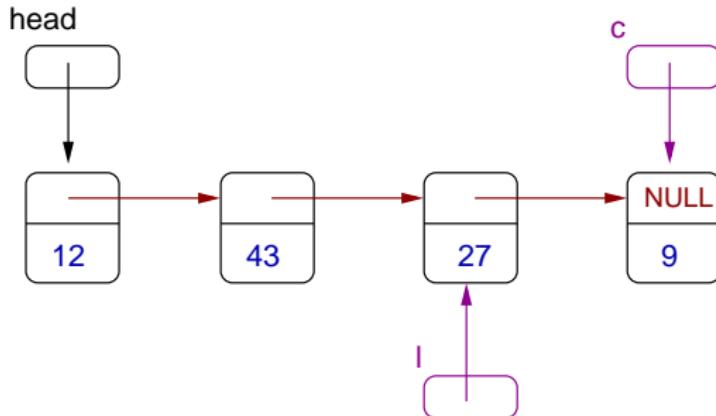
```
struct cell* head = NULL;  
head = insere(head, 12);  
head = insere(head, 43);  
head = insere(head, 27);  
head = insere(head, 9);
```



# Exemple d'insertion en queue de liste

## Utilisation

```
struct cell* head = NULL;  
head = insere(head, 12);  
head = insere(head, 43);  
head = insere(head, 27);  
head = insere(head, 9);
```



## Exemple d'insertion en queue de liste

### Utilisation

```
struct cell* head = NULL;  
head = insere(head, 12);  
head = insere(head, 43);  
head = insere(head, 27);  
head = insere(head, 9);
```

### Remarques :

- la liste est dans le **même ordre** que celui des insertions,
- la tête de liste n'est modifiée que lors de la première insertion,
- le coût d'une seule insertion est **linéaire**,  $\mathcal{O}(n)$   
le coût de la construction est **quadratique**.  $\mathcal{O}(n^2)$

# Insertion après un élément

Insertion facile si on a un pointeur vers la cellule précédente.

## Insertion après pred

```
void insere_apres(struct cell* pred, int elem) {  
    struct cell *c = malloc(...);  
    c->data = elem;  
    c->next = pred->next;  
    pred->next = c;  
}
```

**Effet :** insère elem après la cellule pointée par pred.

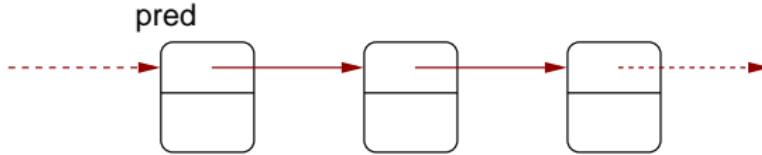
L'insertion *avant* une cellule donnée est plus complexe...

# Insertion après un élément

Insertion facile si on a un pointeur vers la cellule précédente.

## Insertion après pred

```
void insere_apres(struct cell* pred, int elem) {  
    struct cell *c = malloc(...);  
    c->data = elem;  
    c->next = pred->next;  
    pred->next = c;  
}
```

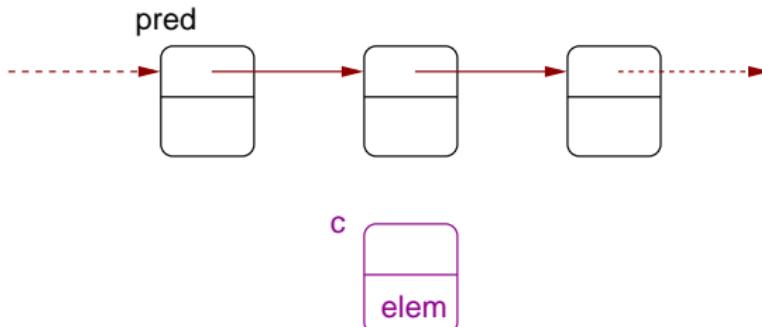


# Insertion après un élément

Insertion facile si on a un pointeur vers la cellule précédente.

## Insertion après pred

```
void insere_apres(struct cell* pred, int elem) {  
    struct cell *c = malloc(...);  
    c->data = elem;  
    c->next = pred->next;  
    pred->next = c;  
}
```

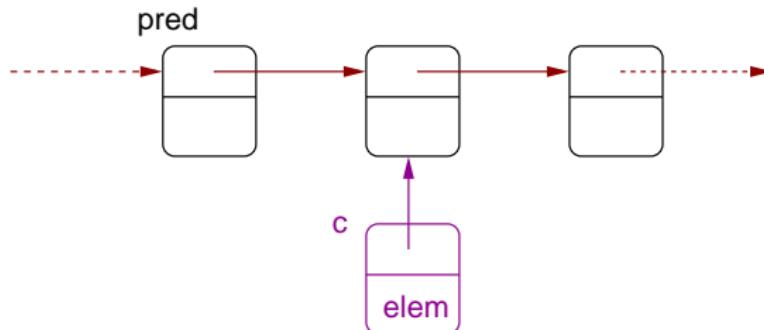


# Insertion après un élément

Insertion facile si on a un pointeur vers la cellule précédente.

## Insertion après pred

```
void insere_apres(struct cell* pred, int elem) {  
    struct cell *c = malloc(...);  
    c->data = elem;  
    c->next = pred->next;  
    pred->next = c;  
}
```

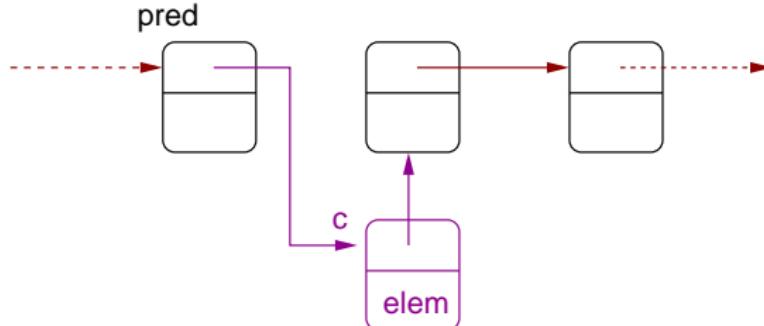


# Insertion après un élément

Insertion facile si on a un pointeur vers la cellule précédente.

## Insertion après pred

```
void insere_apres(struct cell* pred, int elem) {  
    struct cell *c = malloc(...);  
    c->data = elem;  
    c->next = pred->next;  
    pred->next = c;  
}
```



# Insertion après un élément

Insertion facile si on a un pointeur vers la cellule précédente.

## Insertion après pred

```
void insere_apres(struct cell* pred, int elem) {  
    struct cell *c = malloc(...);  
    c->data = elem;  
    c->next = pred->next;  
    pred->next = c;  
}
```

## Notes :

- coût constant, hors calcul de pred,
- pred peut être obtenu par une variante de recherche,
- on suppose qu'on n'insère pas en première position.  
(⇒ on suppose que la liste n'est pas vide)

# Suppression d'un élément

Suppression facile **si on a un pointeur vers la cellule précédente.**

## Suppression après pred

```
void supprime_apres(struct cell* pred) {  
    struct cell *c = pred->next;  
    pred->next = c->next;  
    free(c);  
}
```

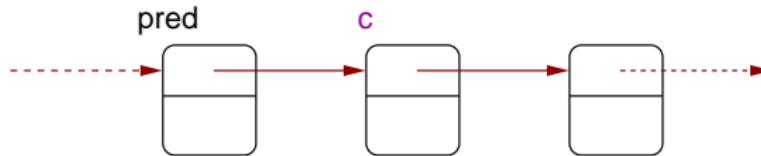
**Effet :** supprime la cellule suivant pred.

# Suppression d'un élément

Suppression facile si on a un pointeur vers la cellule précédente.

## Suppression après pred

```
void supprime_apres(struct cell* pred) {  
    struct cell *c = pred->next;  
    pred->next = c->next;  
    free(c);  
}
```

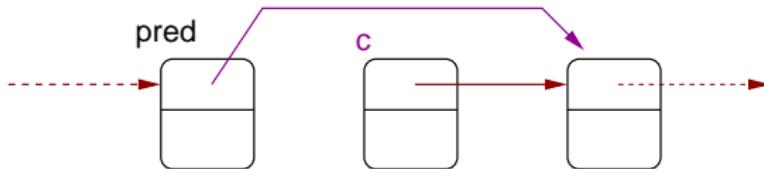


# Suppression d'un élément

Suppression facile si on a un pointeur vers la cellule précédente.

## Suppression après pred

```
void supprime_apres(struct cell* pred) {  
    struct cell *c = pred->next;  
    pred->next = c->next;  
    free(c);  
}
```

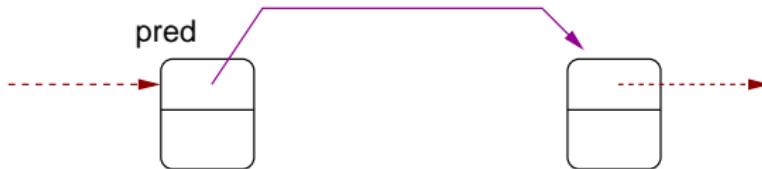


# Suppression d'un élément

Suppression facile si on a un pointeur vers la cellule précédente.

## Suppression après pred

```
void supprime_apres(struct cell* pred) {  
    struct cell *c = pred->next;  
    pred->next = c->next;  
    free(c);  
}
```



# Suppression d'un élément

Suppression facile **si on a un pointeur vers la cellule précédente.**

## Suppression après pred

```
void supprime_apres(struct cell* pred) {  
    struct cell *c = pred->next;  
    pred->next = c->next;  
    free(c);  
}
```

## Notes :

- coût constant, hors calcul de pred,
- on suppose qu'on ne détruit pas en première position,  
( $\Rightarrow$  on suppose que la liste n'est pas vide)
- on suppose que pred a un suivant, ( $\text{pred}-\text{next} \neq \text{NULL}$ )  
(on peut par contre avoir  $c-\text{next} = \text{NULL}$ ).

# Suppression d'un élément (version complète)

On propose maintenant une version plus complète de supprime de prototype :

Prototype de la suppression

```
struct cell* supprime(struct cell* head, int elem);
```

**Effet** : supprime le premier élément égal à elem,

- calcule automatiquement pred,
- gère les **cas limites** :
  - liste vide, liste à un seul élément,
  - elem en tête ou en fin de liste,
  - elem non présent dans la liste,
- la tête de liste peut changer,
- coût linéaire au pire, à cause de la recherche de pred.

# Suppression d'un élément (version complète)

## Suppression

```
struct cell* supprime(struct cell* head, int elem)
{
    struct cell* pred = head, *c;

    if (!head) return head;
    if (head->data==elem)
        { c = head->next; free(head); return c; }

    while (pred->next && pred->next->data!=elem)
        pred = pred->next;

    if (pred->next)
        { c = pred->next; pred->next = c->next; free(c); }

    return head;
}
```

# Concaténation de listes

## Concaténation

```
struct cell* concat(struct cell* head1,
                     struct cell* head2) {
    struct cell*c;
    if (!head1) return head2;
    for (c=head1; c->next; c=c->next);
    c->next = head2;
    return head1;
}
```

**Effet :** concatène la 2ème liste au bout de la 1ère.

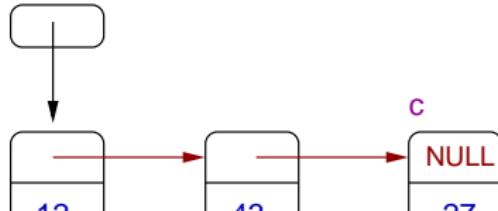
**Utilisation :** head1 = concat(head1, head2);

# Concaténation de listes

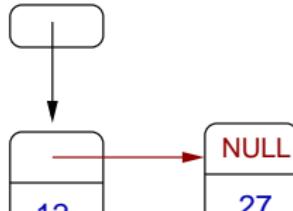
## Concaténation

```
struct cell* concat(struct cell* head1,
                     struct cell* head2) {
    struct cell*c;
    if (!head1) return head2;
    for (c=head1; c->next; c=c->next);
    c->next = head2;
    return head1;
}
```

head1



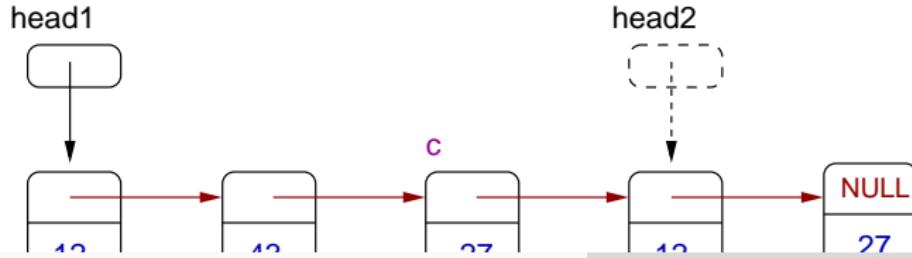
head2



# Concaténation de listes

## Concaténation

```
struct cell* concat(struct cell* head1,
                     struct cell* head2) {
    struct cell*c;
    if (!head1) return head2;
    for (c=head1; c->next; c=c->next);
    c->next = head2;
    return head1;
}
```



# Concaténation de listes

## Concaténation

```
struct cell* concat(struct cell* head1,
                     struct cell* head2) {
    struct cell*c;
    if (!head1) return head2;
    for (c=head1; c->next; c=c->next);
    c->next = head2;
    return head1;
}
```

## Remarques :

- coût linéaire (parcours complet de la 1ère liste),
- la tête de la première liste peut changer,
- il vaut mieux ne plus accéder à la liste par head2...

# Destruction totale d'une liste

## Destruction

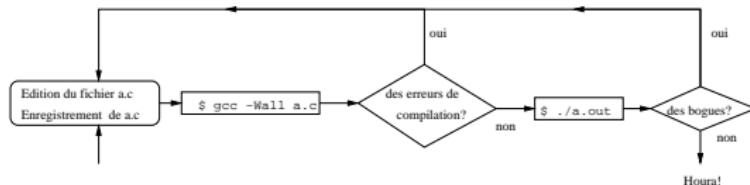
```
void detruit(struct cell* head)
{
    struct cell *c;
    while (head) {
        c = head->next;
        free(head);
        head = c;
    }
}
```

**Effet :** détruit totalement la liste.

**Note :** on a besoin d'une variable temporaire c.

# Erreurs courantes sur les listes

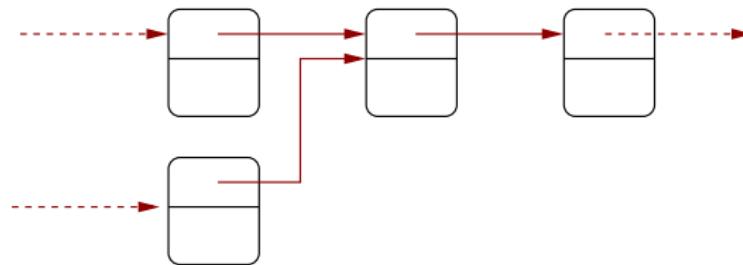
- Erreurs courantes sur les pointeurs et la mémoire dynamique :
  - déréférencer un pointeur NULL,
  - utiliser un bloc après l'avoir libéré,
  - libérer deux fois le même bloc,
  - oublier de libérer un bloc (fuites de mémoire).
- Introduction de *cycles* :



(génère des boucles infinies lors des parcours,  
cause des fuites de mémoire, ...)

## Erreurs courantes sur les listes

- Partage de cellules entre plusieurs listes :



(effets de bord lors de la modification d'une liste,  
cause des libérations multiples de blocs, . . . )

- Oubli des cas limites :

- liste vide, listes à un élément,
- insertion/suppression en première/dernière position,
- etc.

# Comparaison entre listes et tableaux

Coût comparé des listes simplement chaînées et des tableaux.

	listes	tableaux
recherche d'une valeur	$\mathcal{O}(n)$	$\mathcal{O}(n)$
accès par indice	$\mathcal{O}(n)$	$\mathcal{O}(1)$
insertion en tête	$\mathcal{O}(1)$	$\mathcal{O}(n)$
insertion en queue	$\mathcal{O}(n)$	$\mathcal{O}(n)$
insertion au milieu	$\mathcal{O}(1)$ / $\mathcal{O}(n)$	$\mathcal{O}(n)$

Les listes sont particulièrement efficaces pour :

- l'insertion et la suppression en tête de liste,
- l'insertion et la suppression en milieu de liste,  
si la cellule précédente est connue.

# Les listes d'association

**Liste d'association** = associe une valeur à chaque clé.

## Cellule

```
struct cell {  
    struct cell* next;  
    int          key;  
    float        val;  
};
```

## Recherche

```
float recherche(struct cell* head, int key) {  
    for (; head; head = head->next)  
        if (head->key == key) return head->val;  
    return 0.;  
}
```

**Effet** : renvoie la valeur associée à une clé, ou 0.

## Listes avec sentinelle

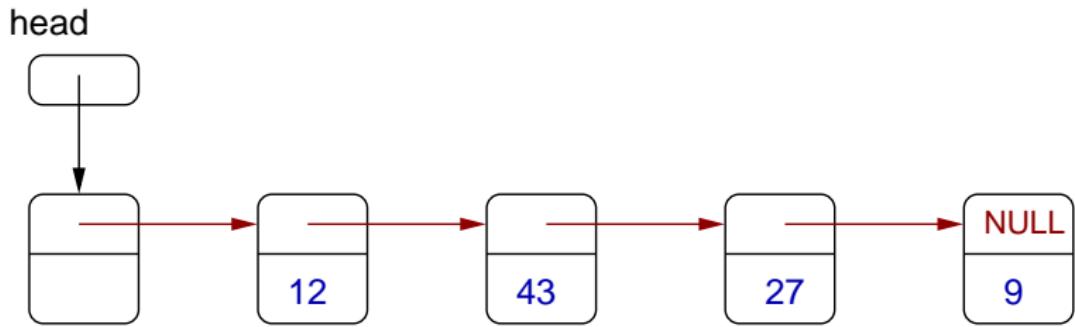
**Idée** : ajout d'une **cellule sentinelle** en tête de liste :

- `head->next` pointe sur la première cellule de la liste,
- `head->data` n'est pas utilisé,
- toutes les fonctions prennent en argument un **pointeur vers la sentinelle**.

**Avantage** : simplifie la gestion des cas limites :

- cas où la liste est vide,  
(la liste contient toujours au moins une cellule)
- cas où le premier élément de la liste est modifié.  
(`head->next` est modifié, pas de tête de liste à retourner)

## Illustration d'une liste avec sentinelle



# Opérations sur les listes avec sentinelle

## Exemples d'opérateurs

```
struct cell* create()
{
    struct cell* c = malloc(sizeof(struct cell));
    c->next = NULL;
    return c;
}

void insere_tete(struct cell* head, int data)
{
    struct cell* c = malloc(sizeof(struct cell));
    c->data = data;
    c->next = head->next;
    head->next = c;
}
```

# Opérations sur les listes avec sentinelle

## Exemples d'opérateurs

```
void insere_queue(struct cell* head, int data)
{
    struct cell* c = malloc(sizeof(struct cell));
    c->data = data;
    c->next = NULL;
    while (head->next) head = head->next;
    head->next = c;
}

void concat(struct cell* head1, struct cell* head2)
{
    while (head1->next) head1 = head1->next;
    head1->next = head2->next;
    free(head2);
}
```

## Listes doublement chaînées

---

# Listes doublement chaînées

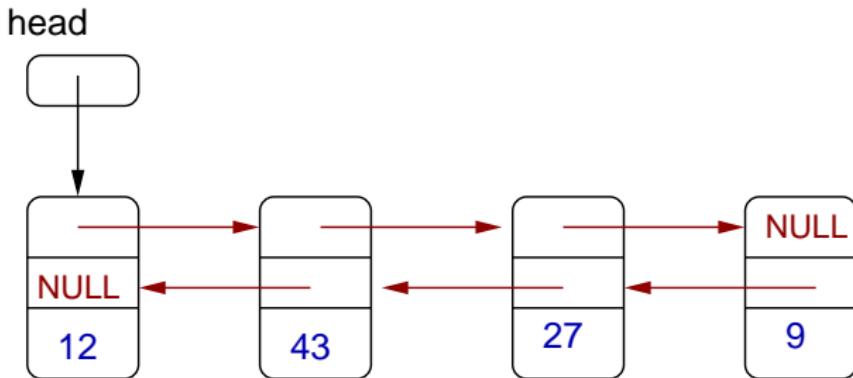
**Liste doublement chaînée =** on maintient :

- un pointeur vers la cellule suivante,
- un pointeur vers la cellule précédente.

## Type cellule

```
struct cell {  
    struct cell* next;  
    struct cell* prev;  
    int          data;  
};
```

## Illustration d'une liste doublement chaînée



- head pointe vers la première cellule,
- c->next=NULL pour la dernière cellule,  
sinon c->next->prev=c,
- c->prev=NULL pour la première cellule,  
sinon c->prev->next=c.

# Insertion dans une liste doublement chaînée

## Insertion après prev

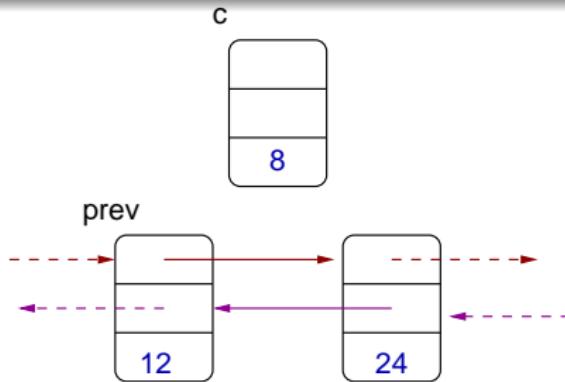
```
void insere_apres(struct cell* prev, int elem) {  
    struct cell *c = malloc(...);  
    c->data = elem;  
    c->next = prev->next; prev->next = c;  
    c->next->prev = c;      c->prev = prev;  
}
```

**Effet :** insère une cellule **après** prev.

# Insertion dans une liste doublement chaînée

## Insertion après prev

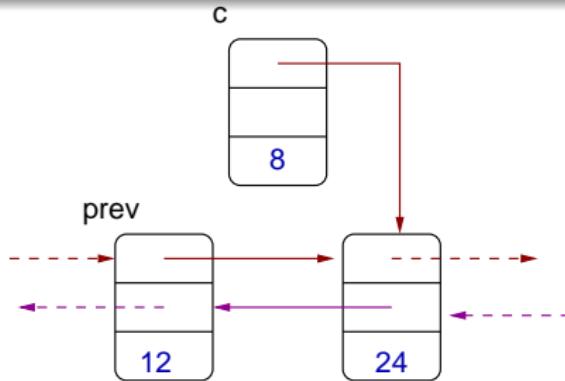
```
void insere_apres(struct cell* prev, int elem) {  
    struct cell *c = malloc(...);  
    c->data = elem;  
    c->next = prev->next; prev->next = c;  
    c->next->prev = c;      c->prev = prev;  
}
```



# Insertion dans une liste doublement chaînée

## Insertion après prev

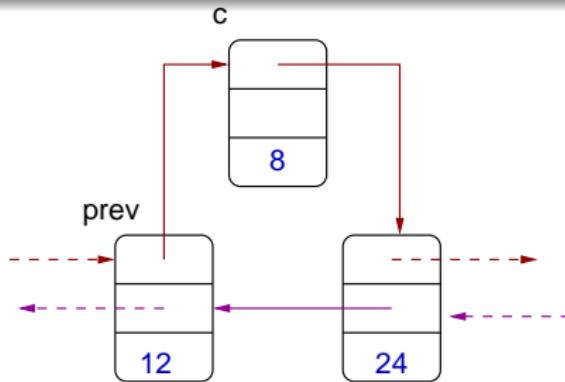
```
void insere_apres(struct cell* prev, int elem) {  
    struct cell *c = malloc(...);  
    c->data = elem;  
    c->next = prev->next; prev->next = c;  
    c->next->prev = c;      c->prev = prev;  
}
```



# Insertion dans une liste doublement chaînée

## Insertion après prev

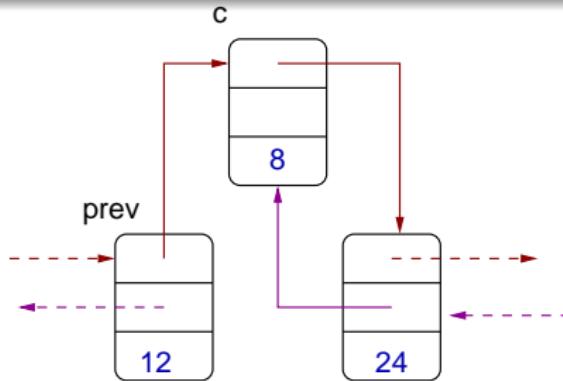
```
void insere_apres(struct cell* prev, int elem) {  
    struct cell *c = malloc(...);  
    c->data = elem;  
    c->next = prev->next; prev->next = c;  
    c->next->prev = c;      c->prev = prev;  
}
```



# Insertion dans une liste doublement chaînée

## Insertion après prev

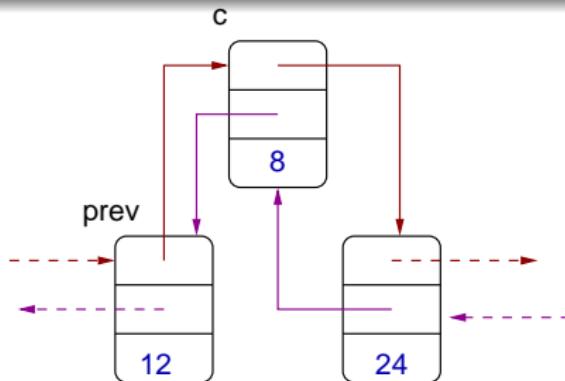
```
void insere_apres(struct cell* prev, int elem) {  
    struct cell *c = malloc(...);  
    c->data = elem;  
    c->next = prev->next; prev->next = c;  
    c->next->prev = c;      c->prev = prev;  
}
```



# Insertion dans une liste doublement chaînée

## Insertion après prev

```
void insere_apres(struct cell* prev, int elem) {  
    struct cell *c = malloc(...);  
    c->data = elem;  
    c->next = prev->next; prev->next = c;  
    c->next->prev = c;      c->prev = prev;  
}
```



# Insertion dans une liste doublement chaînée

## Insertion après prev

```
void insere_apres(struct cell* prev, int elem) {  
    struct cell *c = malloc(...);  
    c->data = elem;  
    c->next = prev->next; prev->next = c;  
    c->next->prev = c;      c->prev = prev;  
}
```

## Remarques :

- coût constant, sans compter le calcul de prev,
- ne marche pas pour insérer en première position,
- pour permettre l'insertion en dernière position :  
 changer    `c->next->prev=c;`  
 en        `if (c->next) c->next->prev = c;`

# Insertion dans une liste doublement chaînée

## Insertion avant next

```
void insere_avant(struct cell* next, int elem) {  
    struct cell *c = malloc(...);  
    c->data = elem;  
    c->prev = next->prev; next->prev = c;  
    c->prev->next = c;      c->next = next;  
}
```

**Effet :** insère une cellule **avant** pred.

**Remarques :**

- on a simplement inversé les mots prev et next,
- ne marche pas pour insérer en dernière position,
- facilement modifiable pour permettre l'insertion en première position.

## Comparaison des types de listes

Listes doublement chaînées vs. listes simplement chaînées :

- on doit maintenir deux pointeurs par cellule au lieu d'un :
  - chaque opération est légèrement plus coûteuse,
  - plus complexe, risques de bug accrus,
- certaines opérations sont plus faciles.  
(e.g., parcours en arrière, insertion avant une cellule)

Schéma classique en informatique : l'ajout de redondance permet des calculs plus rapides, au prix d'une maintenance plus complexe.

Variantes : listes avec sentinelles, listes circulaires, etc.