

Programmation C

Histoire du C

- Créé en 1972 par Ken Thompson et Dennis Ritchie pour le développement d'Unix.
- 1978 : le livre de Kernigham & Ritchie est la première « norme » du C.
- 1989 : normalisation par l'ANSI (puis par l'ISO en 1990). C'est la norme C89/C90.
- 1999 : ajout de fonctionnalités ; norme C99
- 2011 : ajout de fonctionnalités ; norme C11
- 2018 : corrections techniques dans la norme C18



Avantages

- Langage simple avec des concepts algorithmiques et manipulation de bas-niveau.
- Assez bonne portabilité.
- Code très efficace car proche des instructions processeur.
- Fortement interfacé avec le système Unix.

Inconvénients

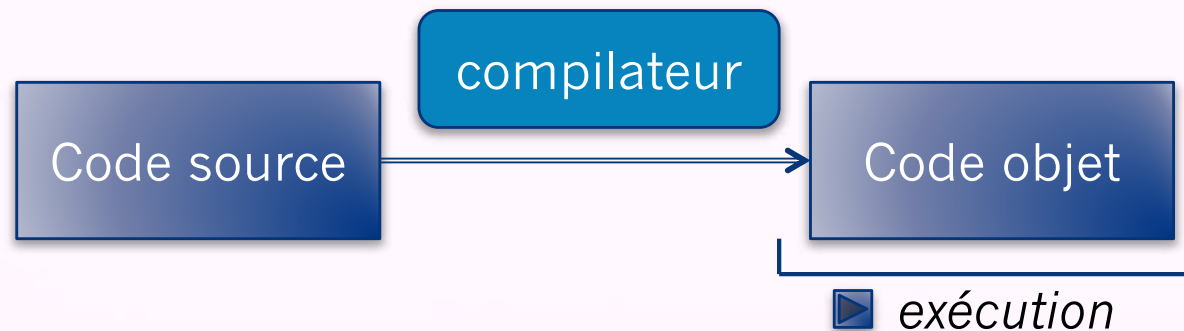
- Code efficace mais parfois illisible.
- Grande liberté dans la syntaxe ce qui amène fréquemment des bugs d'exécution.
- Le code est efficace car il n'y a pas de vérifications lors de l'exécution → bugs fréquents
- Simplicité → peu de structures de haut-niveau (liste/pile/file/table hachage). Il faut le faire à la main ou à l'aide de bibliothèques.
- Pas de GUI standard.

Pourquoi faire du C ?

- Source de nombreux autres langages.
- Ancienneté du langage → on rencontrera un jour ou l'autre du code C (maintenance, docs, algorithmes).
- Interfaçage avec de nombreuses bibliothèques existantes (ex: traitement du signal, banques...)
- Indispensable lors des interactions avec le système/matériel : systèmes embarqués, programmation système...
- Permet de mieux comprendre les mécanismes matériels et logiciels sous-jacents

Compilation

- Le C est un langage compilé (comme ADA, C++, Fortran, Pascal...).



- Le compilateur vérifie le code source et génère les instructions processeur pour la plateforme cible dans le code objet qui est ensuite exécuté.

Avantages

- Des erreurs sont détectées dès la compilation.
- L'exécution du code est très rapide (directement des instructions processeur).
- Le code exécutable se suffit à lui-même.

Inconvénients

- Le code exécutable est limité à une plateforme (CPU+OS).
Pour en changer, une recompilation est nécessaire.
- Le temps de compilation est non négligeable pour les gros projets.

Interprétation

- D'autres langages sont interprétés (PHP, Javascript, Perl, langages mathématiques, langages de script shell...)



- Le code source est analysé dans l'interpréteur et immédiatement exécuté en simulant son fonctionnement sur l'ordinateur.

Avantages

- Le temps de développement est réduit car l'exécution est immédiate.
- Quelle que soit la plateforme, il suffit d'avoir l'interpréteur.

Inconvénients

- C'est plus lent car c'est l'interpréteur qui tourne, pas l'application.
- L'occupation mémoire est plus importante (application + interpréteur).

Environnements de développement

- Simple
 - Utiliser un éditeur de texte pour le code source (avec coloration syntaxique).
 - Compilateur en ligne de commande : `gcc`
 - Shell pour lancer l'exécutable.
- IDE (*Integrated Development Environment*)
 - Regroupe tous ces outils avec une interface graphique unifiée.
 - Exemple : Eclipse, NetBeans, Code::Blocks, Xcode, Visual Studio Code
- En ligne : www.onlinegdb.com

Composition d'un programme

- C est un langage impératif : un programme est une succession d'ordres à exécuter par le processeur.
- Un programme C regroupe :
 - Des variables globales : zone de stockage de données accessible de tout le programme.
 - Un ensemble de fonctions effectuant des tâches et s'appelant les unes les autres.
 - Une des fonctions s'appelle `main()` ; c'est le point d'entrée du programme.
- Le code peut se répartir dans plusieurs fichiers source.

Un premier programme

```
/* Ce programme affiche "Hello world" */  
  
#include <stdio.h>  
  
int main(int argc, char **argv) {  
    printf("Hello world\n");  
    return 0;  
}
```

```
$gcc -o hello hello.c -Wall
```

```
$/hello
```

```
/* Ce programme affiche "Hello world" */
```

Commentaire

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

Inclusion d'un fichier, dans lequel est déclarée la fonction printf()

```
int main(int argc, char **argv) {  
    printf("Hello world\n");  
    return 0;  
}
```

Début

Chaîne à afficher

Retour à la ligne

Arguments que le shell passe au programme

Fonction "système" effectuant l'affichage

Délimiteurs de la fonction

```
$gcc -o hello hello.c -Wall
```

Option d'affichage des avertissements

```
./hello
```

Nom du fichier source

Nom de l'exécutable à générer

Commentaires

- On ajoute des commentaires dans du code C en l'encadrant par `/* ... */`
- Attention, pas de commentaires imbriqués. On ne peut donc pas commenter toute une zone de code contenant déjà des commentaires.
- `//` permet, suivant les compilateurs, de commenter une ligne de code
 - Ce n'est pas du C90
 - C'est dans la norme C99
 - Les compilateurs l'acceptent comme extension de toute façon.

Syntaxe de base

- Indentation, espacement et retour à la ligne laissés au choix du programmeur
- Jeu de caractères : ASCII sans accent !
 - Impératif dans les identificateurs
 - Si les caractères accentués sont utilisés dans les chaînes, la portabilité n'est pas assurée.
- Identificateur : suite de lettres majuscules/minuscules + chiffres + souligné (_)
 - Éviter le (_) en premier [« réservé » au système]

Variables & typage

- Toutes les variables utilisées doivent être préalablement déclarées.
- Chaque variable a un type fixé à la déclaration
 - Utilisé par le compilateur pour calculer la taille mémoire nécessaire au stockage de la variable.
 - Lui permet de vérifier les cohérences des expressions.
 - Lui permet de détecter les fautes de frappe.

- Une valeur numérique stockée en mémoire ne porte pas d'indication de type.
 - C'est le contexte d'exécution (mis en place par le compilateur) qui indique comment traiter la valeur.
- Si le programmeur demande à faire n'importe quoi, le programme fera n'importe quoi.

Types entiers

<code>char</code>	1 octet	$[-128 ; 127]$ (code ASCII)
<code>short</code>	2 octets	$[-32768 ; 32767]$
<code>long</code>	4 octets	$[-2.10^9 ; 2.10^9]$
<code>long long</code>	8 octets	(C99 ou extension)
<code>int</code>	2 ou 4 octets	(maintenant toujours 4)

- La norme n'impose pas de taille aux types mais de simples recommandations (souvent respectées)

II. Types

- Des constantes indiquent les valeurs extrêmes.
 - Des types artificiels (C99) forcent la taille.
 - On peut indiquer une valeur numérique non-signée (c'est-à-dire toujours positive) en précédant le type par **unsigned**
 - Se mélange mal avec les types signés
 - Amène des bugs
- Préférable de ne pas l'utiliser

Choix d'un type

« Quelles valeurs numériques vont être mises dans ma variable ? »

« Vont-elles changer au fur et à mesure de l'évolution du programme ? »

- Cela indique la place mémoire minimum à occuper et donc le type.

Types réels

- Représentation approchée en virgule flottante :

<code>float</code>	4 octets (simple précision)	$\sim 10^{38}$
--------------------	-----------------------------	----------------

<code>double</code>	8 octets (double précision)	$\sim 10^{308}$
---------------------	-----------------------------	-----------------

- Il existe le `long double` sur 8/10 octets.
- Calculs approchés ; erreurs d'arrondi...

Autres types

- Énumération : valeurs numériques limitées à quelques valeurs simples.
- Tableau : suite indexée d'éléments de même type.
- Pointeur : référence à une case mémoire, par exemple celle d'une autre variable.
- Structure : regroupement de variables de types différents manipulable comme un tout.
- Booléen et complexe en C99.

Constantes

- Caractère : entre apostrophes
 - Directement : `'a'` ; `'A'` ; `'1'`
 - Code ASCII en octal : `'\101'` ; `'\065'`
 - Code ASCII en hexa : `'\x41'` ; `'\x6b'`
 - Séquence d'échappement :
`'\n'` ; `'\t'` ; `'\\'` ; `'\''` ; `'\"'` ...
- Il y aura toujours en mémoire le code ASCII sur un octet.

II. Types

- Entière
 - En décimal, en octal (précédée de 0) ou en hexa (précédée de 0x).
 - Ex : 36, -47, 0674, -0xaf3e
- Réelle : notation scientifique
 - Ex : 1.2 ; 1e5 ; 1.3e2 ; 0.25e-3 ; .5 ; 12. ; .2e4
- Chaîne : entre guillemets
 - Ex : "A", "Hello\n", "Ici\040\n\"truc\t"
Question : combien de caractères dans cette dernière chaîne ?
 - La fin d'une chaîne est indiquée par un octet nul (le caractère '\0', de code ASCII 0), ajouté par le compilateur.
→ Occupe un octet de plus en mémoire.

Déclaration

- On déclare une variable en faisant précéder son identificateur par son type :

```
int i;
```

```
double dd;
```

```
char c;
```

```
unsigned long long very_big;
```

- Déclarations multiples :

```
int i, j, k;
```

```
double r, f;
```

II. Types

- On peut initialiser la variable avec une expression constante :

```
int i=0, k=2;  
double d=.5e4;  
char c='A'+3;
```

- **const** devant le type indique une variable non-modifiable. Permet au compilateur de vérifier et d'optimiser son utilisation.

Initialisation en même temps obligatoire !

```
const double PI = 3.141592;  
const char UN = '1';
```

Tableaux

- Suite indexée (de taille fixée) d'éléments de même type. On peut ainsi accéder à chaque cellule individuelle.
- Déclaration : on fait suivre l'id. de la taille entre []

```
int tabEntiers[10]; /* tableau de 10 entiers */  
double M[50]; /* tableau de 50 doubles */  
char str[25]; /* tableau de 25 caractères */
```

II. Types

- Normalement, la taille doit être constante et connue à la compilation.
- Les tableaux de taille connue à l'exécution sont possibles en C99 ou grâce à une extension autorisée du compilateur.

```
int n;
```

```
...
```

```
n = ... /* calcul */
```

```
double T[n];
```

- Mais une fois définie, on ne peut pas changer la taille d'un tableau.

II. Types

- On accède à un élément en indiquant son indice entre `[]` ; il va de `0` (premier élément) à *taille-1* (dernier élément).

```
tabEntiers[2] = 3;
```

```
M[0] = 3.2;
```

```
str[5] = 'a';
```



IL N'Y A AUCUN CONTRÔLE DE L'INDICE
AVANT L'ACCÈS !

II. Types

- Le nom du tableau est « équivalent » à l'adresse du premier élément ; il ne représente pas le tableau en tant qu'entité globale.
- Ainsi, l'affectation directe d'un tableau à un autre n'est pas autorisé !

```
int a[5], b[5];
```

```
a = b; /* interdit, erreur de compilation */
```

Initialisation d'un tableau

- L'initialisation peut se faire au moment de la déclaration :

```
int T[5] = {10, 20, 30, 40, 50};
```

- S'il y a moins d'éléments que la taille, les éléments restants sont initialisés à une valeur nulle :

```
int T1[100] = {0};
```

- Avec l'initialisation, la taille n'est plus nécessaire :

```
int T[] = {10, 20, 30, 40, 50};
```

- En C90 strict, les éléments doivent être constants.

Chaînes

- Une chaîne est un tableau de caractères dont le dernier élément « utile » est suivi d'un octet nul.
- On peut l'initialiser comme un tableau ou comme une chaîne :

```
char str[] = {'h', 'e', 'l', 'l', 'o', '\0'};  
char str[] = "hello"; /* taille = 6 */
```
- Dans tous les cas, on peut modifier les éléments.
- Pas de traitement direct, il faut passer par la bibliothèque `<string>`.

Bibliothèque string

- Quelques fonction utiles :
 - `strlen()` renvoie la longueur d'une chaîne
 - `strcmp()` compare deux chaînes
 - `strcpy()` recopie une chaîne dans une autre (Attention, une place suffisante doit être prévue !)
 - `strncpy()` recopie avec une taille maximum
 - `strchr()` cherche un caractère dans une chaîne
 - `strstr()` cherche une sous-chaîne dans une chaîne
 - `strtok()` découpe une chaîne en tokens séparés par un ou plusieurs délimiteurs
- Ne pas oublier `#include <string.h>`

Expressions

- Toute expression a une valeur et un type, fonction des types des opérandes.
- Il n'y a pas de type booléen en C :
 - Une valeur nulle (ou **NULL**) → FAUX
 - Une valeur non-nulle → VRAI

Opérateurs

- Arithmétiques :

+ - *

/ : division entière si opérandes entiers sinon réelle

ex : `var_entiere/10.0` \neq `var_entiere/10`

% : modulo pour opérandes entiers

- Sur bits (pas portable → dépend du matériel)

& (ET) | (OU), ^ (XOR), ~ (NON)

<< : décalage de bits vers la gauche (`10 << 3` = 80)

>> : décalage de bits vers la droite (`10 >> 2` = 2)

III. Expressions

- Booléens

&& : ET logique entre deux expressions

- `expr1 && expr2` renvoie 0 (faux) ou 1 (vrai)
 - Si `expr1` est nulle, `expr2` n'est pas évaluée et la valeur de l'expression est 0.
 - Sinon, si `expr2` est nulle, la valeur est 0, et sinon c'est 1.

|| : OU logique entre deux expressions

- `expr1 || expr2` renvoie 0 (faux) ou 1 (vrai)
 - Si `expr1` est non-nulle, `expr2` n'est pas évaluée et la valeur de l'expression est 1.
 - Sinon, si `expr2` est non-nulle, la valeur est 1, et sinon c'est 0.
- L'ordre des expressions est important.
- Ne pas confondre `&/&&` et `|/||`

III. Expressions

! : opérateur logique de négation

== : opérateur d'égalité

!= : opérateur de différence

>=, <=, >, < : opérateurs de comparaison

- Conditionnel

`expr_logique ? expr1 : expr2`

- `expr_logique` est évaluée.
- Si elle est non-nulle, `expr1` est évaluée et devient la valeur de l'expression ;
- si elle est nulle, `expr2` est évaluée et devient la valeur de l'expression.

Ex : `max = (a>b) ? a : b;`

Affectations

- Forme générale : *Ivalue* = *expression*
 - *Ivalue* peut être un variable scalaire (nombre, pointeur) ou structurée mais PAS un tableau.
 - Une affectation est une expression qui renvoie la valeur affectée.
 - Ex : `a = b = c = 0;`
 - La *Ivalue* et l'expression doivent être de même type (structures ou pointeurs) ou compatible (types numériques).

- Forme composée : *lvalue OP= expression*
 - Est équivalent à *lvalue = lvalue OP expression*
 - Valable pour les opérateurs arithmétiques et sur bits.
 - Exemples :
`i += 2`
`i ^= 1`
`var <<= 2`

- Incrémentation/Décrémentation

Ivalue $++$ est équivalent à *Ivalue* $+= 1$

Ivalue $--$ est équivalent à *Ivalue* $-= 1$

- Incrémente (resp. décrémente) la variable et renvoie la valeur de la variable AVANT incrémentation (resp.).

- $++Ivalue$ et $--Ivalue$ font de même mais renvoient la valeur APRÈS l'opération.

→ L'affectation est la même, la valeur renvoyée est différente.

Conversions de types

- Dans un calcul, le compilateur convertit les opérandes numériques du type le plus petit vers le plus grand.
 - Lors d'une affectation, l'expression à droite du = est converti au type de la *lvalue*.
- Attention à la perte éventuelle de précision.

Conversions explicites

- On peut convertir le type d'une expression (*cast*) :
(nv_type) expr

- Exemple :

```
int a=5, b=2;
```

```
float res;
```

```
res = a/b;           /* 2.0 */
```

```
res = (float)(a/b);  /* 2.0 */
```

```
res = ((float) a)/b; /* 2.5 */
```

Priorités des opérateurs

- Les priorités sont classiques avec quelques défauts.
- Les tests d'égalité sont plus prioritaires que les opérateurs sur bits :

$a \& b == c$ est $a \& (b == c)$

- Les décalages sont moins prioritaires que les opérateurs arithmétiques :

$a \ll b + c$ est $a \ll (b + c)$

Ordre d'évaluation

- L'ordre d'évaluation des opérandes d'un calcul dépend du compilateur (non-déterministe) !
- Il est spécifié pour `&&`, `||` et `?:`, dans les autres cas, on ne peut rien supposer.

`a[i]+b[i++]` → est-ce `a[1]+b[1]` ou `a[2]+b[1]` ?

`a[i]=b[i++]` → affecte `b[1]` à `a[1]` ou `a[2]` ?

E/S simples

- La fonction `printf()` permet d'afficher une chaîne dans laquelle certains éléments (spécification de format) auront été remplacés par la valeur d'expressions passées en paramètre.

```
int i = 10;
```

```
printf("La valeur de i est %d\n", i);
```



```
La valeur de i est 10
```

Remplacement

Affichage
écran

- Ne pas oublier `#include <stdio.h>`

IV. E/S simples

```
printf("Hello world\n");
```

```
int i=1, j=2;
```


```
printf("La somme de i(%d) et j(%d) vaut %d\n", i, j, i+j);
```

- Les spécifications de format simples sont :
 - %c : affiche un caractère
 - %d : affiche un entier
 - %f : affiche un réel
 - %s : affiche une chaîne
- On peut y ajouter des indications de formatage, de précision, de taille minimale d'affichage, de taille de l'expression (pour être cohérent avec son type)...

- La fonction `scanf()` effectue l'opération inverse à `printf()` : lecture d'une chaîne formatée et affectation à des variables en fonction de spécifications de format.

```
int i;
```

```
scanf("%d", &i);
```



On indique l'adresse
de la variable

→ On entre une valeur entière au clavier et sa valeur numérique sera mise dans la variable `i`.

`%c` : lecture d'un caractère

`%d` : lecture d'un entier (variable de type `int`)

`%f` : lecture d'un réel (variable de type `float`)

`%s` : lecture d'une chaîne sans espacement

→ `scanf()` est compliquée d'utilisation si on veut récupérer plusieurs valeurs en un appel.

Autres fonctions d'E/S

`puts(const char[]);`

→ Affiche la chaîne passée en paramètre suivie d'un '`\n`'. Attention que la chaîne soit bien terminée par `0`.

`gets(char []);`

→ Lit une chaîne au clavier et remplace le '`\n`' final par `0` puis la met dans le tableau.



`gets()` ne vérifie pas que la chaîne n'est pas plus grande que le tableau.

→ Risque de débordement mémoire !!

Privilégier `fgets(s, n, stdin)` qui indique une limite.

Exercice

- Écrire un programme qui récupère 2 entiers fournis par l'utilisateur et
 - affiche chaque nombre décalé
 - 3 fois à droite
 - 2 fois à gauche
 - affiche le résultat des opérateurs sur bits (& | ~) et des opérateurs booléens (&& || !) appliqués aux nombres
 - affiche le 3^e bit de poids faible de chaque nombre
 - affiche la décomposition de chaque nombre en 4 octets (de 0 à 255).

Structures de contrôle

- Il y a les structures habituelles : test, sélection, boucle.
- Il n'y a pas de type booléen en C :
 - Une valeur nulle (ou `NULL`) → FAUX
 - Une valeur non-nulle → VRAI
- Un bloc est une suite d'instructions encadrée par `{ ... }`. C'est équivalent à une instruction simple.

if ... else

```
if (expr)
    instr/bloc
```

- Exemples :

```
if (b>c)
    x=3;
```

```
if (a==0) {
    x=3;
    b=5;
} else
    i++;
```

```
if (expr)
    instr/bloc
else
    instr/bloc
```

- La condition est vraie si l'expression est non-nulle.
Exemples :

```
c = getchar()
(f=truc()) != -1
!a
```

while & do...while

while (expr)
 instr/bloc

- Répète le bloc intérieur tant que l'expression est vraie (non-nulle).
- Si la condition est d'emblée fausse, le bloc n'est jamais exécuté.

do
 instr/bloc
while (expr);

- Répète le bloc intérieur tant que l'expression est vraie (non-nulle).
- Le bloc est exécuté au moins une fois.

for

```
for (expr1; expr2; expr3)  
    instr/bloc
```

- `expr1` est l'initialisation, effectuée avant l'entrée dans la boucle.
- `expr2` est le test de continuation de la boucle ; il est évalué avant l'exécution du corps de la boucle. La boucle s'arrête s'il est faux.
- `expr3` est évaluée à la fin du corps de la boucle.

- Exemples :

```
int i;  
for (i = 0; i < 10; i++)  
    printf("%d ", i);  
printf("\n");
```

```
char c;  
for (; (c=getchar()) != EOF;)  
    printf("%c", c);
```

break, continue et return

break;

- break provoque l'abandon immédiat d'une boucle ou d'une sélection. Exemple :

```
while (1) {  
    ...  
    if (condition d'arrêt)  
        break;  
    ... /* suite de la boucle */  
}
```

`continue;`

- `continue` provoque l'abandon de l'itération en cours et passe immédiatement à l'itération suivante.

Exemple :

```
for (i = 0; i < n; i++) {  
    if (a[i] < 0)  
        continue;  
    a[i] = ...  
    ... /* suite de la boucle */  
}
```


`return;` ou `return expr;`

- **return** provoque l'abandon de la fonction en cours et le retour à la fonction appelante (en transmettant éventuellement une valeur).

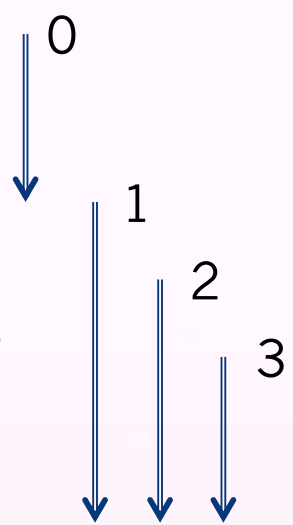
switch

```
switch (expr) {  
    case v1: instructions;  
    case v2: instructions;  
    ...  
    case vn: instructions;  
    default: instructions;  
}
```

- Si une des valeurs entières constantes indiquées dans les 'case' est égale à l'expression, on poursuit l'exécution à ce niveau (et tout ce qui suit).

- Si aucune valeur ne correspond, on poursuit au niveau du 'default' (s'il est présent).
- On peut mettre un break pour sortir.

```
switch (i) {  
    case 0 :    j = 0;  
                break;  
    case 1 :  
    case 2 :    x = 'a';  
    case 3 :    y = 25;  
                break;  
    default:    error();  
}
```



Fonctions

Suite autonome d'instructions que l'on peut appeler depuis d'autres endroits du programme et même récursivement.

- Évite de dupliquer du code dans le programme.
- Permet de partager du code entre programmes.
- Permet de conceptualiser une tâche :
 - écriture et tests plus faciles

VI. Fonctions

- Écriture de la fonction :

```
type Id(arg 1, ..., arg n) {  
    variables locales & instructions  
}
```

- Le type du résultat renvoyé est indiqué en premier.
- Le nom de la fonction (**Id**) est unique (pas deux fonctions avec le même nom, pas de variable globale identique).
- Chaque argument est indiqué par une déclaration de variable.
- Le corps est un bloc de variables locales et instructions. Pas de fonctions imbriquées !

Type fonction/args

- Une fonction peut renvoyer un type numérique, un type pointeur, un type structure mais pas un tableau (mais simplement sa référence).
- Si la fonction ne renvoie rien, mettre `void`.
 - *Sinon, le compilateur suppose `int`.*
- Si la fonction n'a pas d'argument, mettre `void` dans la liste d'arguments.
 - *Sinon le compilateur ne vérifie pas les appels à la fonction.*

- `return expr;` permet de terminer l'appel d'une fonction et de renvoyer une valeur.
 - Exemple :

```
int somme(int x, int y) {  
    int resultat = x + y;  
    return resultat;  
}
```

```
int absDiff(int x, int y) {  
    return (x>y ? x-y : y-x);  
}
```

- On peut avoir plusieurs `return` dans une fonction, renvoyant des valeurs différentes.

VI. Fonctions

- On peut s'en servir pour renvoyer une valeur spéciale (si c'est possible) en cas d'erreur.
- L'oubli d'un **return** n'est signalé que par un avertissement → source de bugs.
- Bonnes pratiques :
 - Donner un nom expressif à la fonction et aux variables.
 - En commentaire, mettre une documentation pour la fonction : ce qu'elle fait, que passer en argument, ce qu'elle renvoie, signification des valeurs spéciales...
 - Le **main()** renvoie 0 si OK et une autre valeur en cas d'erreur. C'est récupéré par le shell.

Appel d'une fonction

- On appelle une fonction en indiquant son nom avec, entre (), une éventuelle liste d'arguments.
- En C, les arguments sont **toujours** passés par valeur.
- On recopie la valeur des arguments effectifs (ceux de l'appel) dans les arguments formels (variables locales à la fonction)
- Une fonction ne peut pas modifier une variable qui lui est passée en paramètre (car c'est sa valeur qui est passée à la fonction).

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

```
int f(int x) {  
    x++;  
    return x;  
}
```

```
int main() {  
    int i = 10;  
    printf("%d\n", i);  
    f(i);  
    printf("%d\n", i); /* quelle valeur ? */  
    return 0;  
}
```

- Lors d'un appel de fonction, le compilateur vérifie le nombre d'arguments et que les types des arguments effectifs sont bien compatibles avec les types des arguments formels.
- Il procède alors aux éventuelles conversions de type nécessaire pour ranger la valeur des arguments effectifs dans les arguments formels.

```
#include <stdio.h>

double f(double x) {
    return x+1;
}

int main() {
    double dd;
    dd = f(10);
    printf("%f\n", dd);
    return 0;
}
```

- Quel est l'affichage ?

Prototype de fonctions

- Que se passe-t-il si on intervertit les deux fonctions `f()` et `main()` ? Quel est l'affichage ?

```
#include <stdio.h>

int main() {
    double dd;
    dd = f(10);
    printf("%f\n", dd);
    return 0;
}

double f(double x) {
    return x+1;
}
```

Il y a une erreur car le compilateur voit d'abord l'appel à `f()` et construit alors une déclaration implicite pour `f()` :

`int f(int)`

Une fonction inconnue renvoie un `int` (c'est la règle) et elle a un paramètre de type `int`.

Lorsque la vraie déclaration de `f()` arrive, il y a un conflit de type car elle renvoie un `double`.

Prototype de fonctions

- Que se passe-t-il si on intervertit les deux fonctions et que `f()` est modifiée pour renvoyer un `int` ?

```
#include <stdio.h>

int main() {
    double dd;
    dd = f(10);
    printf("%f\n", dd);
    return 0;
}

int f(double x) {
    return x+1;
}
```

Cela compile (avec un warning) mais le résultat est faux !

Le compilateur a construit la déclaration implicite pour `f()` :

`int f(int)`

Le paramètre est donc passé sans conversion de type vers le type `double`.

[En fait, les arguments entiers et flottants sont passés par des registres différents. La fonction récupère donc son argument dans un registre différent de celui où il est passé.]

Prototype de fonctions

- Si, coup de chance, les types des arguments correspondent, tout se passe bien :

```
#include <stdio.h>

int main() {
    double dd;
    dd = f(10.0);
    printf("%f\n", dd);
    return 0;
}

int f(double x) {
    return x+1;
}
```

```
#include <stdio.h>

int main() {
    double dd;
    dd = f(10);
    printf("%f\n", dd);
    return 0;
}

int f(int x) {
    return x+1;
}
```

VI. Fonctions

- Lorsqu'il rencontre un appel de fonction, le compilateur vérifie sa conformité *s'il connaît la définition de la fonction*.
- Sinon, il construit une définition implicite de la fonction :
 - il suppose que la fonction renvoie un entier ;
 - il suppose que les arguments formels correspondent exactement aux types des arguments effectifs (et ne prévoit donc aucune conversion de type).
- Dans le meilleur des cas, quand le compilateur voit la définition de la fonction, il indique une « redéfinition illégale » si elle diffère trop de l'implicite :
 - type de retour différent de `int` ;
 - types des arguments trop différents.

- Dans le pire des cas, le compilateur ne dit rien. Si les arguments correspondent, tant mieux, mais s'ils diffèrent (types compatibles mais pas identiques)...
- Il est en fait important de déclarer chaque fonction par un prototype en tête de son code. C'est l'annonce la fonction, définie plus loin ou dans un autre fichier :
 - type du résultat
 - nombre et types des arguments.

- Pour ce faire, on reproduit l'entête de la définition de la fonction suivi de ';'. Le nom des arguments formels n'est pas obligatoire.

- Exemples:

```
double f(double dd);
```

```
void truc(int i, double f, char T[100]);
```

- qui s'écrivent aussi :

```
double f(double);
```

```
void truc(int, double, char []);
```

→ Cela permet au compilateur de vérifier les appels

Prototype de fonctions

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

```
/* Prototype de f */
```

```
double f(double);
```

```
int main() {  
    double dd;  
    dd = f(10);  
    printf("%f\n", dd);  
    return 0;  
}
```

```
double f(double x) {  
    return x+1;  
}
```

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

```
/* Prototype de f */
```

```
int f(double);
```

```
int main() {  
    double dd;  
    dd = f(10);  
    printf("%f\n", dd);  
    return 0;  
}
```

```
int f(double x) {  
    return x+1;  
}
```

Visibilité des identificateurs

- Une fonction est normalement visible de tout le programme :
 - aussi bien de tout le code situé dans le même fichier que des autres fichiers sources inclus dans le même exécutable.
- Une fonction préfixée par **static** ne sera visible que depuis son fichier.
 - limite les risques de collision de noms.

VI. Fonctions

- Une variable peut être :
 - globale : définie en dehors des fonctions, elle est visible de tout le fichier source.
 - locale : définie dans une fonction, elle n'est pas accessible en dehors et est recréée à chaque appel de la fonction.
 - **static** : définie dans une fonction (et accessible que depuis cette fonction), elle est initialisée à une valeur de départ à la compilation ; elle existe et garde sa valeur entre deux appels.
 - mise dans un registre pour un accès plus rapide (mot-clé **register**). Les compilateurs classiques ignorent cette information qui est plutôt valable pour les développements sur microcontrôleur.

```
#include <stdio.h>

/* Avec une variable globale */
int varGlob = 0;

void f(void) {
    printf("Appel de f : %d\n", ++varGlob);
}

int main() {
    int i;
    for (i = 0; i < 10; i++)
        f();
    return 0;
}
```

- Quel est l'affichage ?

```
#include <stdio.h>

/* Avec une variable locale */
void f(void) {
    int cumul = 0;
    printf("Appel de f : %d\n", ++cumul);
}

int main() {
    int i;
    for (i = 0; i < 10; i++)
        f();
    return 0;
}
```

- Quel est l'affichage ?

```
#include <stdio.h>

/* Avec une variable locale static */
void f(void) {
    static int cumul = 0;
    printf("Appel de f : %d\n", ++cumul);
}

int main() {
    int i;
    for (i = 0; i < 10; i++)
        f();
    return 0;
}
```

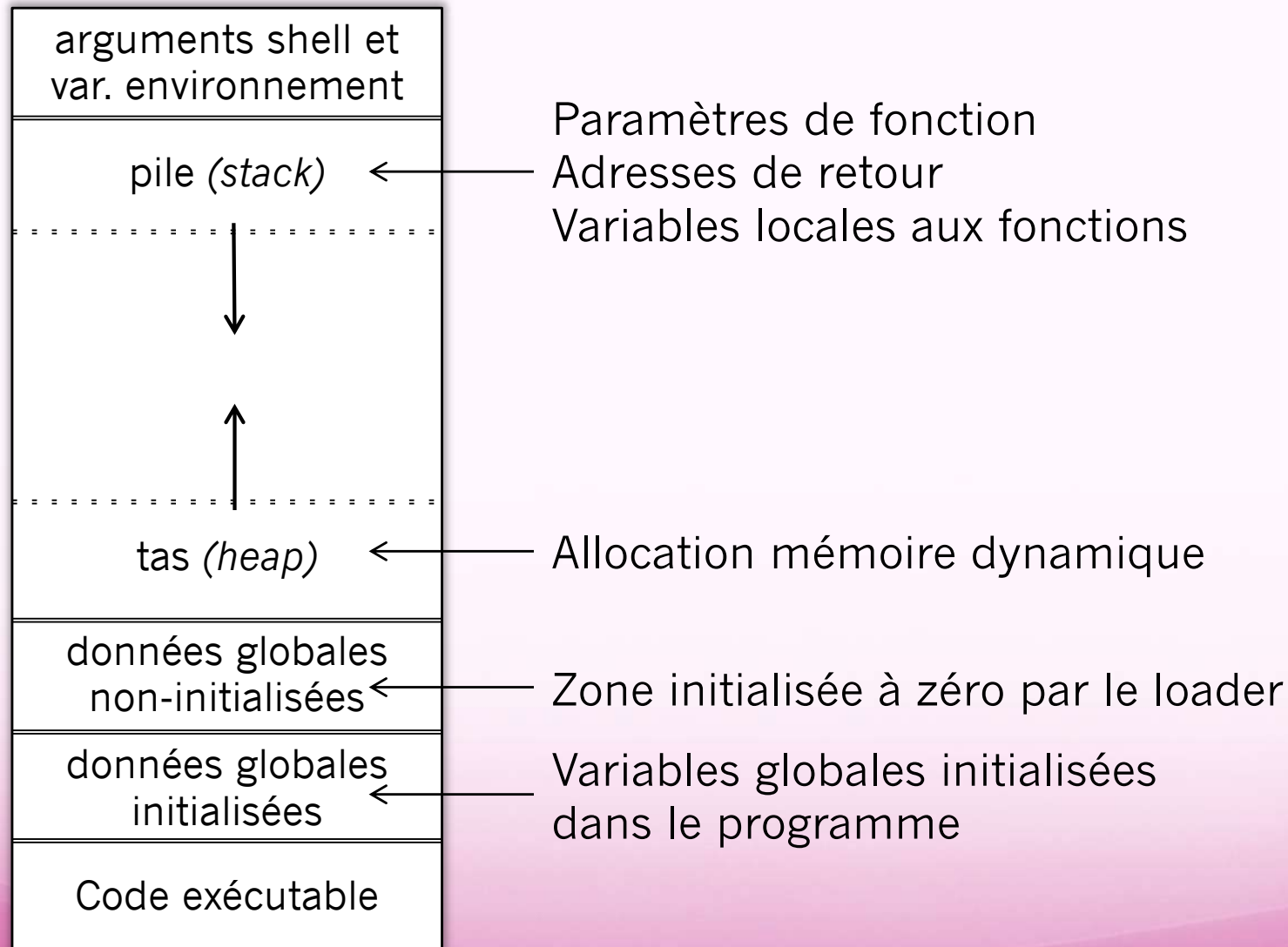
- Quel est l'affichage ?

Gestion de la mémoire

Pourquoi gérer la mémoire en C ?

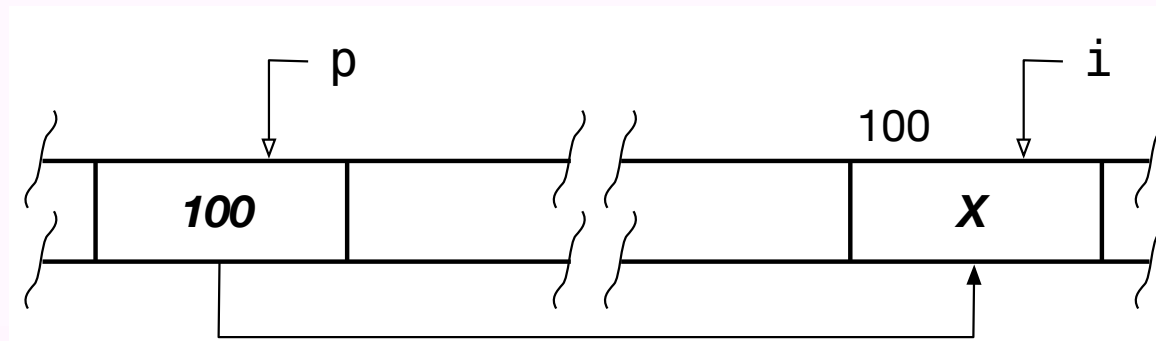
- Pour court-circuiter le passage d'arguments par valeur.
- Pour gérer dynamiquement la mémoire lors de l'ajout ou le retrait des structures de données.

Plan de la mémoire



Pointeurs

- C'est une variable contenant une adresse mémoire, par exemple celle d'une autre variable.



- **i** est une variable située à l'adresse 100 et contenant la valeur **X** ;
- **p** est un pointeur sur **i**, contenant son adresse.

Intérêts des pointeurs

- Modifier la valeur originale d'une variable passée en argument d'une fonction.
- Allouer dynamiquement de la mémoire.
- Gérer des structures récursives.
- Accéder autrement à des tableaux.
- Accéder au matériel.
- Passer une fonction en argument.

VII. Pointeurs

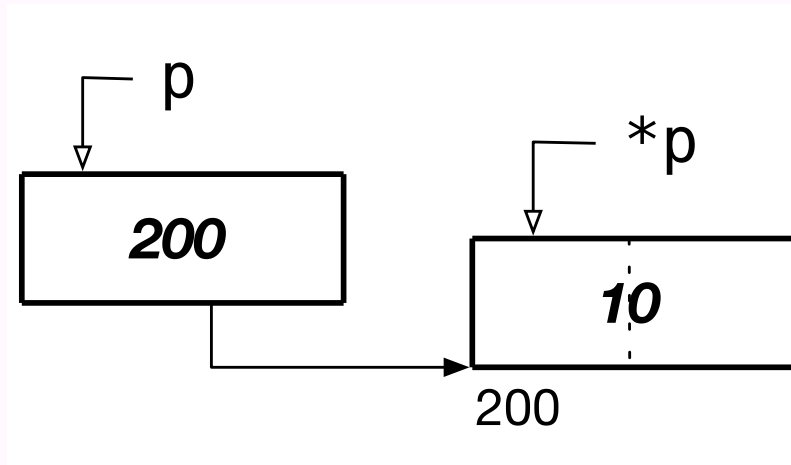
- Tout pointeur est associé à un type, celui de la case pointée. Cela permet au compilateur de savoir comment récupérer les données pointées (combien d'octets à considérer).
- Déclaration : on fait précéder l'identificateur du pointeur par *. Exemple :

```
int *p;           /* pointeur sur int */  
char *q, *s;      /* deux pointeurs sur char */  
long l, *pl;      /* un long et un pointeur sur long */
```

- L'identificateur sert à désigner le pointeur mais aussi à le déréférencer en le préfixant par *.

VII. Pointeurs

```
short *p;  
p = 200;  
*p = 10;
```



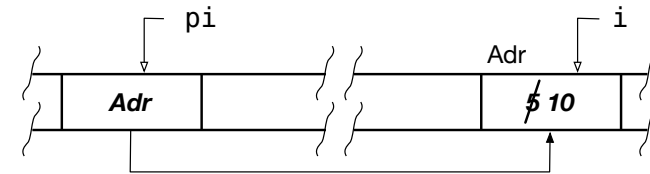
- Déclarer un pointeur permet de réserver une zone mémoire où stocker une adresse (avant sur 32 bits, maintenant très souvent sur 64).
- Il n'y a pas d'initialisation de sa valeur ! Donc pas d'adresse légitime dans le pointeur. Le déréférencer conduit à accéder à une case mémoire inconnue, peut-être interdite (plantage), peut-être liée à une autre variable (conduisant à une mauvaise exécution).

VII. Pointeurs

- Initialisation d'un pointeur:
 - En prenant l'adresse d'une variable avec l'opérateur & :

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

```
int main() {  
    int i = 5;  
    int *pi;      /* ou int *pi = &i; */  
    pi = &i;  
    *pi = 10;     /* modifie i !      */  
    printf("%d - %d\n", i, *pi); /* affichage des valeurs */  
    printf("%p - %p\n", &i, &pi); /* affichage des adresses */  
    return 0;  
}
```



- Avec un autre pointeur

```
char t[10], *p, *q;  
p = &t[0];    /* pointe sur le 1er élément */  
q = p + 3;    /* pointe sur t[3] */
```
- Avec une adresse matérielle
- Avec l'adresse d'une zone mémoire allouée

Exercice

- Écrire un programme déclarant trois variables initialisées (`char/int/double`) ainsi qu'un pointeur sur chacune de ces variables. Afficher la valeur de la variable, son adresse (avec `%p` dans le `printf`) ainsi que l'adresse du pointeur correspondant en n'utilisant que la variable pointeur.

Modification d'argument

- Un pointeur passé en argument ne peut pas être modifié (ce n'est qu'une adresse) mais la case pointée si :
→ on peut modifier une variable extérieure à une fonction à l'intérieur de celle-ci.

```
#include <stdio.h>

void fct(int i) { i = 10; }

void fct_2(int *ptr) { *ptr = 10; }

int main() {
    int i = 5;
    printf("%d\n", i);    /* affiche ? */
    fct(i);
    printf("%d\n", i);    /* affiche ? */
    fct_2(&i);
    printf("%d\n", i);    /* affiche ? */
    return 0;
}
```

Exercice

- Écrire un programme déclarant deux variables `int` dans la fonction `main()`. Écrire une fonction permettant l'échange de valeur entre les deux variables. Vérifier l'échange en affichant la valeur des variables locales à `main()` avant et après l'appel.

Autres usages

- Renvoi de plusieurs valeurs :
 - Une fonction peut renvoyer une valeur et modifier un argument via son adresse pour « renvoyer » une valeur supplémentaire.
 - Une fonction peut prendre ou renvoyer un pointeur plutôt qu'une variable complète pour éviter de devoir la recopier en entrée ou sortie de la fonction.

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

long fct(long *R1, long *R2) {
    *R1 = random();
    *R2 = random();
    return random();
}

int main() {
    long N1, N2;
    long N3 = fct(&N1, &N2);
    printf("%ld, %ld, %ld\n", N1, N2, N3);
}
```

VII. Pointeurs

- Un pointeur de type `void *` ne connaît pas le type de l'objet pointé. Utilisé par des fonctions dont l'adresse de retour peut être utilisée par différents types de pointeurs.
 - On ne peut pas le dérérérencer
 - On peut mettre sa valeur dans un pointeur du bon type.

```
#include <stdio.h>

int main() {
    int i = 10;
    void *q = &i;
    int *p = q;
    *p = 5;
    /* *q = 5; provoquerait une erreur */
    printf("%d\n", i);
    return 0;
}
```

VII. Pointeurs

- Un pointeur égal à `NULL` ne pointe pas sur une adresse licite (son déréférencement provoque un bug).
 - permet de détecter les bugs plus rapidement

```
#include <stdio.h>

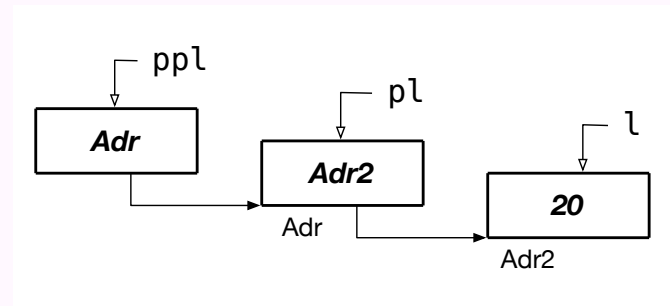
int main() {
    int *p;
    int *q = NULL;
    printf("%p %x\n", p, *p); /* affiche des valeurs */
    printf("%p %x\n", q, *q); /* provoque un plantage */
    return 0;
}

/* un warning de compilation prévient quand même
   de l'utilisation de p sans initialisation. */
```

VII. Pointeurs

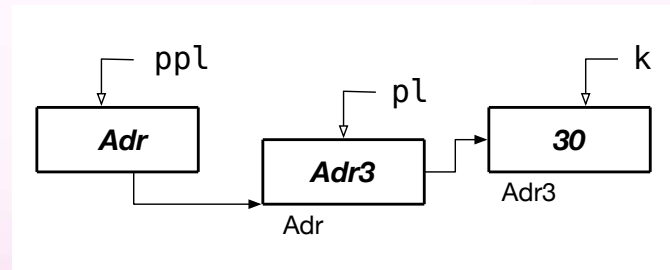
- Un pointeur est une variable, on peut pointer dessus. On a alors un pointeur de pointeur.

```
long l;  
long *p1 = &l;  
long **ppl = &p1;  
**ppl = 20; /* modifie l */
```



- On peut modifier **p1** via son pointeur **ppl** :

```
long k;  
*ppl = &k; /* modifie p1 */  
*p1 = 30;  
/* nouvelle case pointée */
```



Pointeurs et tableaux

- L'id. tableau = pointeur constant sur 1^{er} élément.

```
short T[10] = {0};
short *p = T;      /* ou short *p = &T[0] */
```

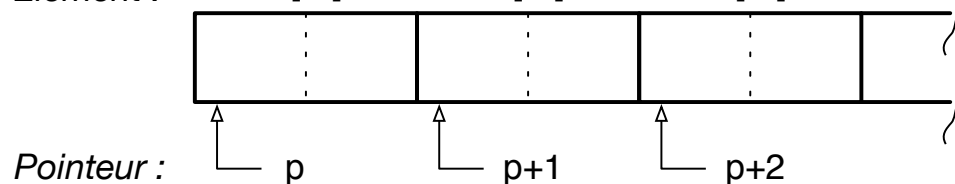
- Accéder au i^e élément revient à additionner i au pointeur (addition de i éléments et pas i octets).

```
for (i = 0; i < 10; i++) {
    *(p+i) = i;
    printf("%d ", T[i]); }
```

```
short T[10]; /* 2 octets par élément */
short *p = T; /* ou &T[0] */
```

Adresse : A A+1 A+2 A+3 A+4 A+5

Élément : T[0] T[1] T[2]



- Différences pointeur – tableau

```
int T[5];
```

- déclare un tableau et réserve la place mémoire de stockage
- la variable pointeur T n'est pas modifiable.

```
int *p;
```

- déclare un pointeur modifiable
- mais pour l'instant **p** ne pointe nulle part et ne peut pas encore être déréférencé.

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

```
int main() {  
    int T[5] = {0};  
    int X[5] = {0};  
    X = T; /* erreur */  
    return 0;  
}
```

Tableau en argument

- Lorsqu'on passe un tableau en argument, on passe l'adresse du premier élément du tableau, c'est-à-dire un pointeur.
- `void f(char T[]);` \Leftrightarrow `void f(char *T);`
- On ne connaît pas, dans la fonction la taille du tableau ! Il faut la passer en paramètre.

Tableau en argument

```
#include <stdio.h>

void f(int *T) {    /*  <=>  int T[]      */
    *T = 42;        /*  <=>  T[0] = 42      */
}

int main() {
    int X[5] = {0};
    printf("%d\n", X[1]); /* affiche 0 */
    f(X+1); /* correspond à l'adr. de X[1] */
    printf("%d\n", X[1]); /* affiche 42 */
    return 0;
}
```

Tableau en argument

```
#include <stdio.h>

char *strcpy_TAB(char dest[], char src[]) {
    int i = 0;
    while (src[i] != '\0') {
        dest[i] = src[i];
        i++;
    }
    dest[i] = '\0';
    return dest;
}

char *strcpy_PTR(char *dest, char *src) {
    char *p = dest;
    while ((*dest++) = *(src++)) != '\0'
        ;
    return p;
}

int main() {
    char s[] = "Hello!";
    char XX[7];
    char YY[7];

    printf("XX: %s\n", strcpy_TAB(XX, s));
    printf("YY: %s\n", strcpy_PTR(YY, s));
    printf("s : %s\n", s);
    return 0;
}
```

- Que se passe-t-il si XX et YY ne sont pas assez longs ?

```
char s[] = "Hello!";
char XX[4];
char YY[4];
```

Renvoi d'un tableau

- De la même façon, le renvoi d'un tableau n'est que le renvoi de son adresse, ce qui est une erreur si le tableau est déclaré en variable locale :

```
int *f(void) {  
    int T[10];  
    ...  
    return T; /* erreur en vue */  
}
```

- T est une variable locale qui n'a plus d'existence après exécution de la fonction ; son adresse est inutilisable.

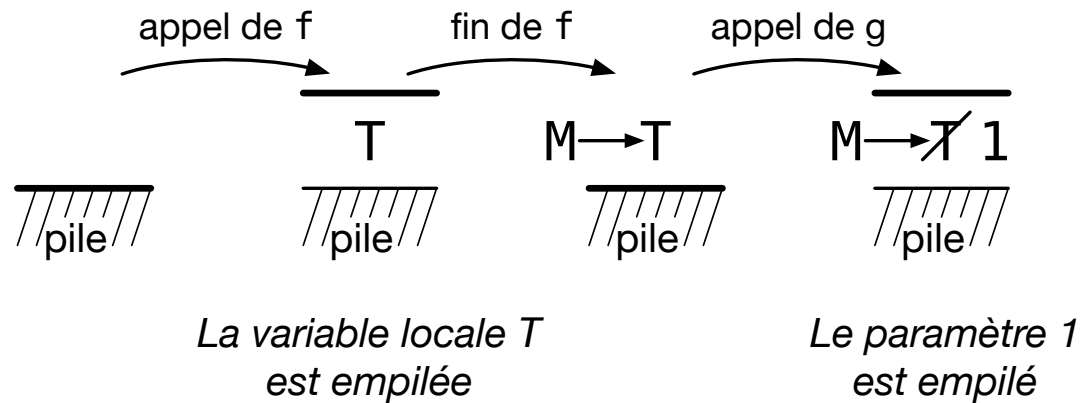
Renvoi d'un tableau

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

```
int g(int j) {
    return j+1;
}
```

```
int *f(void) {
    int T[1];
    T[0] = 42;
    return T;
}
```

```
int main() {
    int *M;
    M = f();
    printf("%d\n", M[0]);
    g(1);
    printf("%d\n", M[0]); /* que s'est-il passé ? */
    g(2);
    printf("%d\n", M[0]); /* de nouveau ? */
    return 0;
}
```



Exercice

- Déclarer dans la fonction `main()` un tableau d'entiers rempli de valeurs positives ou nulles.
- Passer ce tableau à une fonction qui prend un pointeur en argument et affiche tous les éléments du tableau. Comment connaître la taille du tableau ?

Allocation dynamique

- Permet de réserver une zone mémoire dont la taille est connue à l'exécution.
- On y accède uniquement par son adresse, mise dans un pointeur.

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

```
...
```

```
void *malloc(size_t size);
```

→ alloue une zone de **size** octets et renvoie son adresse.

- Si on veut réserver une suite d'éléments (comme un tableau), il faut réserver une zone dont la taille est le nombre d'éléments multiplié par la taille d'un élément.
→ `sizeof(type)` donne la taille d'un type.
- On y accède ensuite comme à un tableau : si `ptr` contient l'adresse, `*ptr` est le premier élément, `*(ptr+1)` le deuxième, etc.

[écriture classique : `*ptr++` permet d'accéder à un élément et de décaler le pointeur sur le deuxième.]

Exemple d'allocation d'une zone pour stocker 3 entiers
= allocation dynamique d'un tableau de 3 `int`.

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

int main() {
    int i;
    /* équivalent à un tableau de 3 int */
    int *T = malloc(3*sizeof(int));

    for (i = 0; i < 3; i++)
        T[i] = i;
    for (i = 0; i < 3; i++)
        printf("%d\n", T[i]);
    return 0;
}
```

Exercices

- Écrire un programme qui demande à l'utilisateur combien de nombres il veut entrer. Le programme doit ensuite :
 - créer une zone mémoire pour stocker ces nombres ;
 - lire en boucle tous les nombres et les stocker ;
 - réafficher tous les nombres et leur somme.

Libération de la mémoire

- Il n'y a pas de mécanisme de ramasse-miettes pour libérer les zones mémoires allouées sur lesquelles plus aucun pointeur ne pointe.

→ Elles vont s'accumuler durant toute la vie du programme.

→ Il faut libérer à la main les zones qui ne sont plus nécessaires :

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

→ Libère la zone pointée par l'adresse passée en argument.
Elle doit être valable !

VII. Pointeurs

Exemple d'allocation et de renvoi, dans une fonction, d'un tableau de taille décidée à l'exécution.

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

double *f(int n) {
    int i;
    double *T = malloc(n * sizeof(double)); /* tableau de n double */
    for (i = 0; i < n; i++)
        T[i] = i/10.0;
    return T;
}

int main() {
    int i, n;
    double *M;
    srand(time(NULL));
    n = (random()%10)+1; /* tirer un nombre aléatoire entre 1 et 10 */
    M = f(n);
    for (i = 0; i < n; i++)
        printf("%f\n", M[i]);
    free(M);
    return 0;
}
```

VII. Pointeurs

Exemple de copie d'une chaîne avec allocation de la place nécessaire pour la destination.

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

char *copie(char *str) {
    /* strdup dans la bibliothèque */
    char *p = malloc(strlen(str) + 1); /* +1 pour le '\0' final */
    char *tmp = p;
    while ((*p++ = *str++) != '\0')
        ;
    return tmp;
}

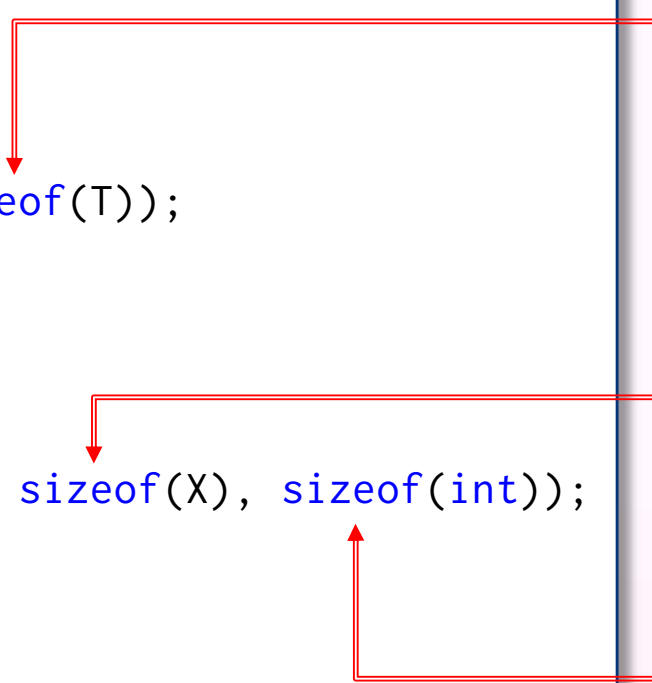
int main() {
    char *src = "Hello world!";
    char *dest;
    dest = copie(src);
    puts(dest);
    return 0;
}
```

Attention : `sizeof()` ne permet pas d'avoir la taille d'un tableau passé en argument !

```
#include <stdio.h>

void f(int T[]) {
    printf("%lu\n", sizeof(T));
}

int main() {
    int X[5] = {0};
    printf("%lu %lu\n", sizeof(X), sizeof(int));
    f(X);
    return 0;
}
```



T N'EST PAS un tableau mais un simple pointeur sur son premier élément. Sa taille de stockage est celle d'un pointeur soit 8 octets.

Le compilateur connaît le tableau X (il est localement défini) et donc sa taille : 20 octets. Taille d'un int : 4 octets.

*Un warning est généré par le compilateur qui indique bien que `sizeof()` va renvoyer la taille d'un `int *` et pas du `int []` passé en paramètre.*

*On peut mettre `int *` à la place : plus de warning mais le fonctionnement est le même !*

On retrouve bien les 5 éléments du tableau.

Exercice

- On souhaite stocker des chaînes de caractères entrées par l'utilisateur.
- Écrire un programme qui
 - contient un tableau de pointeurs ;
 - recopie en boucle des chaînes de caractères entrées par l'utilisateur et stocke leur adresse dans le tableau ;
 - les réaffiche puis libère les zones allouées lorsque l'utilisateur a fini.

Structures

- Elles permettent de regrouper des variables dans une structure manipulable en bloc.
- Exemples :
 - coordonnées géométriques : x, y (et z)
 - adresse : numéro, rue, code postal, ville...
 - personne : nom, prénom, adresse...

- Déclaration :

```
struct nom {  
    type_1 id_1;  
    type_2 id_2;  
    ...  
    type_n id_n;  
} var_1, var_2...;
```

nom du nouveau type

identificateurs et types des différents champs

éventuelles déclarations de variables de ce type structure

- Exemple

```
struct point {  
    double x;  
    double y;  
} p1;  
struct point p2;
```

- Initialisation

```
struct point p3 = {2, 3.5};
```

- Copie

```
struct point p4;  
p4 = p3;  
→ recopie de tous les champs.
```

- Accès aux champs :

```
p4.x = 2.5;  
double ordo = p3.y;
```

- Simplification du type :

```
typedef struct point Point;  
Point p5;
```

Exemple d'utilisation d'une structure point

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

typedef struct point {
    double x;
    double y;
} Point;

/* Prototypes */
Point creePoint(double, double);
void affichePoint(Point);
double distance(Point, Point);

/*****/

Point creePoint(double x, double y) {
    Point P = {x, y};
    return P;
}

void affichePoint(Point p) {
    printf("(%.f;%.f)\n", p.x, p.y);
}
```

```
double distance(Point p1, Point p2) {
    double deltaX = p1.x-p2.x;
    double deltaY = p1.y-p2.y;
    return sqrt(deltaX*deltaX + deltaY*deltaY);
}

int main() {
    Point P[3];
    int i;

    P[0] = creePoint(0,0);
    P[1] = creePoint(1,1);
    P[2] = creePoint(3,3);

    for (i = 0; i < 3; i++)
        affichePoint(P[i]);

    printf("Distance 0-1 : %.f\n", distance(P[0], P[1]));
    printf("Distance 1-2 : %.f\n", distance(P[1], P[2]));
    printf("Distance 0-2 : %.f\n", distance(P[0], P[2]));
    return 0;
}
```

Exercice

- Écrire et tester les fonctions suivantes :

`Point creePoint(double x, double y);` qui renvoie un point

`void affichePoint(Point p);` qui affiche les coordonnées du point passé en paramètre

`double distance(Point p1, Point p2);` qui renvoie la distance entre deux points

- Définir une structure `Segment` composée de deux points.

- Écrire et tester les fonctions suivantes :

`Segment creeSegment(Point A, Point B);` qui renvoie un segment

`void afficheSegment(Segment s);` qui affiche les coordonnées du segment passé en paramètre ainsi que sa longueur.

Structures et pointeurs

- On peut avoir un pointeur dans une structure :

```
struct point {  
    double x;  
    double y;  
    char *nom;  
};  
typedef struct point Point;  
Point origine = {0, 0, "origine"};  
Point A;  
A.x = A.y = 1;  
A.nom = "unite";
```

VIII. Structures

Exemple d'utilisation d'une structure point.

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

typedef struct point {
    double x;
    double y;
    char *nom;
} Point;

void affiche(Point P) {
    printf("%s: (%f;%f)\n", P.nom, P.x, P.y);
}

Point dup(Point P) {
    Point NvP = P;
    /* On alloue la place pour un nouveau nom */
    NvP.nom = malloc(strlen(P.nom) + 3);
    /* On recopie l'ancien nom */
    strcpy(NvP.nom, P.nom);
    /* On ajoute un suffixe */
    strcat(NvP.nom, "_2");
    return NvP;
}
```

```
int main() {
    Point O = {0, 0, "origine"};
    Point A, B;
    A.x = A.y = 1;
    A.nom = "unite";
    affiche(O);
    affiche(A);
    B = dup(A);
    affiche(B);
    return 0;
}
```

Affichage final :

```
origine: (0.000000;0.000000)
unite: (1.000000;1.000000)
unite_2: (1.000000;1.000000)
```


VIII. Structures

- On peut allouer une structure :

```
Point *p = malloc(sizeof(Point));
```

- puis accéder aux champs :

```
(*p).x = 2.5;
```

```
(*p).y = 3.3;
```

- mais en fait on utilise l'opérateur -> :

```
p->x = 2.5;
```

```
p->y = 3.3;
```

```
p->nom = "Un point";
```

- On peut passer une structure à une fonction et renvoyer une structure (recopie de valeurs) ou travailler avec des pointeurs (pas de recopie, d'où gain de temps – mais il faut allouer sur le tas pour renvoyer une structure).

VIII. Structures

Exemple d'utilisation d'un pointeur sur une structure point.

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

typedef struct point {
    double x;
    double y;
    char *nom;
} Point;

void affiche(Point *P) {
    printf("%s: (%f;%f)\n", P->nom, P->x, P->y);
}

Point *dup(Point *P) {
    Point *NvP = malloc(sizeof(Point));
    NvP->x = P->x;
    NvP->y = P->y;
    /* On alloue la place pour un nouveau nom */
    NvP->nom = malloc(strlen(P->nom) + 3);
    /* On recopie l'ancien nom */
    strcpy(NvP->nom, P->nom);
    /* On ajoute un suffixe */
    strcat(NvP->nom, "_2");
    return NvP;
}
```

```
int main() {
    Point O = {0, 0, "origine"};
    Point A = {1, 1, "unite"};
    Point *B;
    affiche(&O);
    affiche(&A);
    B = dup(&A);
    affiche(B);
    return 0;
}
```

Affichage final :

```
origine: (0.000000;0.000000)
unite: (1.000000;1.000000)
unite_2: (1.000000;1.000000)
```

VIII. Structures

Attention, ne jamais faire de supposition sur la taille d'une structure en fonction de la taille de ses types internes : des contraintes d'alignement peuvent amener le compilateur à ajouter des octets.

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

```
struct s1 {  
    char c1;  
    char c2;  
    char c3;  
    int i1;  
    int i2;  
    int i3;  
};
```

```
struct s2 {  
    char c1;  
    int i1;  
    char c2;  
    int i2;  
    char c3;  
    int i3;  
};
```

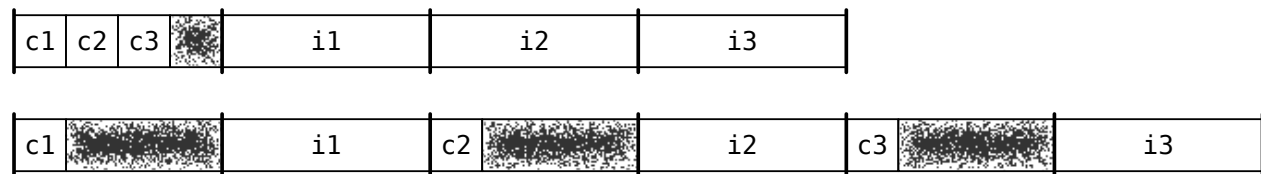
```
int main() {  
    printf("%lu\n", sizeof(struct s1));  
    printf("%lu\n", sizeof(struct s2));  
    return 0;  
}
```

Affichage final (peut varier suivant le compilateur et la machine cible car dépend des contraintes d'alignement liées au CPU :

16

24

- Toujours utiliser `sizeof()` pour réserver la place ;
- Ordonner peut permettre d'économiser la mémoire.

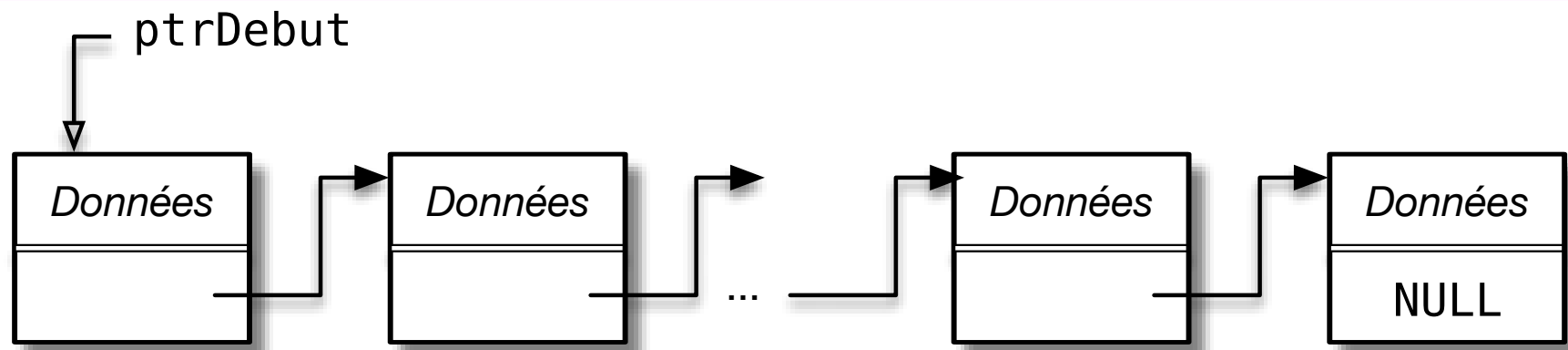


Contrainte : un `int` doit être à une adresse multiple de 4

Structures de données

- Comment organiser une collection de données ?
 - tableau
 - liste chaînée (simple ou double)
 - pile
 - file
 - arbre
 - table de hachage

Ex : une liste chaînée



- Variantes :
 - cellules spéciales en début/fin
 - pointeur sur la dernière cellule
 - liste doublement chaînée...

IX. Structures de données

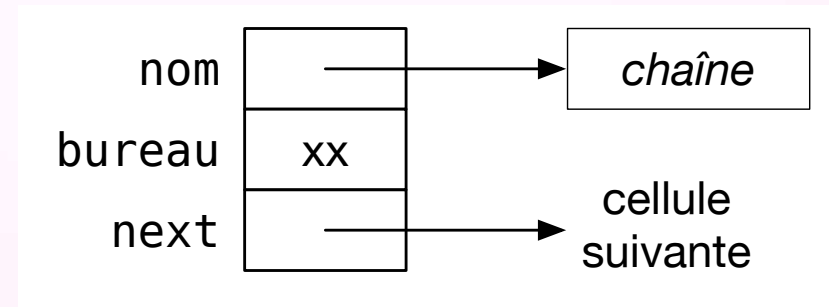
- Exemple : implémentation d'un répertoire sous la forme d'une liste chaînée.

a) Définition d'une cellule de la liste :

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

struct cellule {
    char *nom;
    int bureau;
    struct cellule *next;
};

typedef struct cellule cellule;
```



b) Écriture d'une fonction `nouveau()` :

- qui récupère du clavier les infos d'une personne (`strcpy()` ou `strdup()` permet de recopier une chaîne) ;
- et qui renvoie un pointeur sur la nouvelle cellule correspondante allouée sur le tas.

```
cellule *nouveau(void) {
    cellule *p = malloc(sizeof(cellule));
    /* On suppose qu'un nom a une taille limite */
    char buf[1024];

    printf("Nom? ");
    fgets(buf, 1024, stdin);
    /* On enlève le '\n' final */
    if ((strlen(buf) > 0) && (buf[strlen(buf)-1] == '\n'))
        buf[strlen(buf) - 1] = '\0';
    /* Il faut recopier car le tableau est local */
    p->nom = strdup(buf);

    printf("bureau? ");
    scanf("%d", &p->bureau);
    while (getchar() != '\n')
        ; /* Pour vider le buffer d'entrée */
    p->next = NULL;
    return p;
}
```

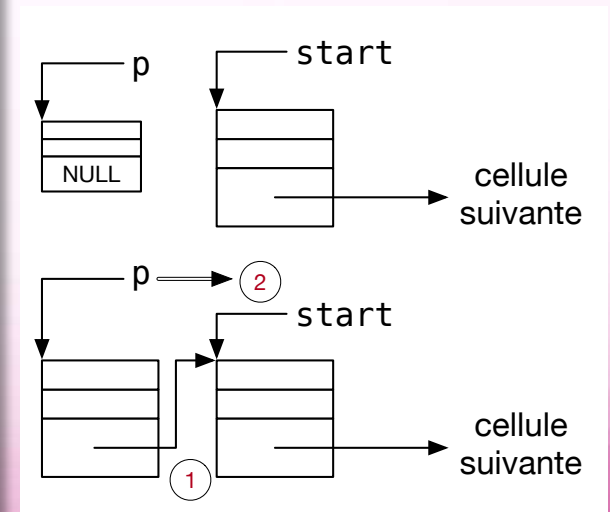
IX. Structures de données

c) Écriture d'une fonction `ajoutTete()` :

- qui ajoute une cellule, dont l'adresse est passée en paramètre, en tête de liste (cette dernière est aussi passée en paramètre) ;
- et qui renvoie le nouveau pointeur de tête.

d) Écriture d'une fonction `affiche()` qui affiche le contenu du répertoire.

```
cellule *ajoutTete(cellule *p, cellule *start) {  
    p->next = start; /* 1 - On suppose p correct */  
    return p;        /* 2 */  
}  
  
void affiche(cellule *current) {  
    printf("Affichage repertoire\n");  
    while (current != NULL) {  
        printf(" --> %s %d\n", current->nom, current->bureau);  
        current = current->next;  
    }  
}
```



e) Tester ces fonctions avec un programme dans lequel on entrera 3 personnes, on les ajoutera au répertoire et affichera ce dernier.

```
int main() {  
    int i;  
    cellule *debut = NULL;  
  
    for (i = 0; i < 3; i++)  
        debut = ajoutTete(nouveau(), debut);  
  
    affiche(debut);  
    return 0;  
}
```

Liste chaînée triée

a) Écrire une fonction `recherche()` :

- qui renvoie un pointeur sur la cellule correspondant à un nom passé en paramètre (et du début de la liste), ou `NULL` si le nom n'est pas dans le répertoire.

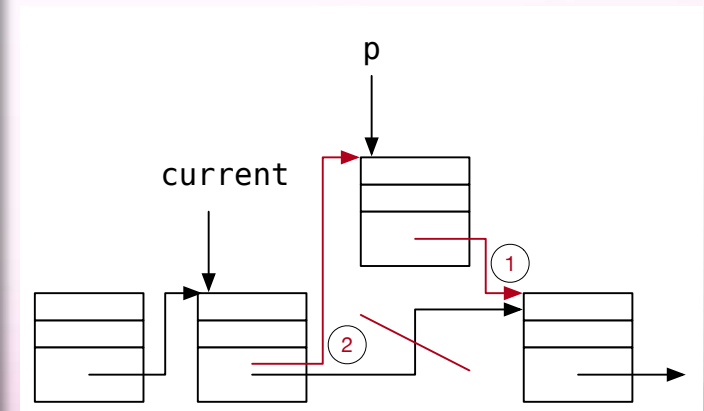
```
cellule *recherche(char *str, cellule *current) {  
    while (current != NULL)  
        if (strcmp(current->nom, str) == 0)  
            break;  
        else  
            current = current->next;  
    return current;  
}
```

IX. Structures de données

b) Écrire une fonction `ajout()` prenant l'adresse d'une cellule et la tête de liste en paramètre :

- qui vérifie que le nom n'est pas déjà présent dans la liste ;
- qui insère la nouvelle cellule dans la liste au bon endroit de telle sorte qu'elle soit toujours triée par nom ;
- et renvoie le (éventuel) nouveau pointeur de tête.

```
cellule *ajout(cellule *p, cellule *start) {  
    cellule *current = start;  
    if (p && recherche(p->nom, start) == NULL) {  
        if (start == NULL) // liste vide  
            return p;      // p est le nouveau pointeur de tête  
        if (strcmp(start->nom, p->nom) > 0) { // ajout en tête  
            p->next = start;  
            return p;  
        }  
        while (current->next &&  
            strcmp(current->next->nom, p->nom) < 0)  
            current = current->next;  
        p->next = current->next; /* 1 */  
        current->next = p;      /* 2 */  
    }  
    return start;  
}
```



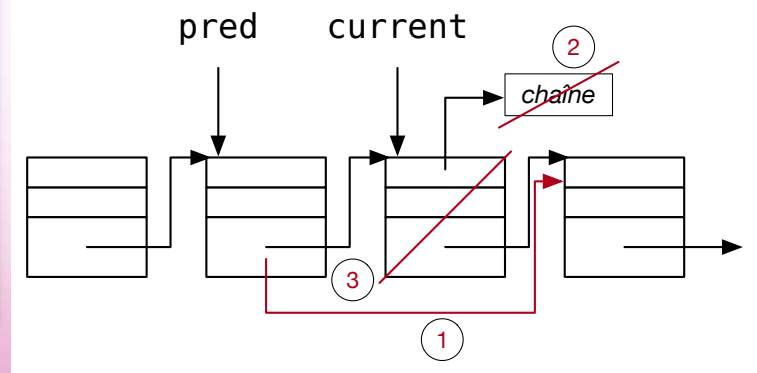
IX. Structures de données

c) Écrire une fonction `supprimer()` :

- qui retire de la liste une cellule correspondant à un nom passé en paramètre (avec la tête de liste) ;
- et renvoie le nouveau pointeur de tête ;
- comment faire pour savoir s'il y a eu succès ou échec ?

```
cellule *supprimer(char *str, cellule *start) {  
    cellule *current = start;  
    cellule *pred;  
  
    if (current && strcmp(current->nom, str) == 0) {  
        // On supprime la première cellule  
        start = current->next;  
        free(current->nom);  
        free(current);  
        return start;  
    }  
    while (current && strcmp(current->nom, str) != 0) {  
        // On va sur la bonne cellule  
        pred = current;  
        current = current->next;  
    }  
}
```

```
if (current != NULL) {  
    // On a trouvé la bonne cellule  
    /* 1 */ pred->next = current->next;  
    /* 2 */ free(current->nom);  
    /* 3 */ free(current);  
}  
return start;  
}
```



- Tester les fonctions précédentes avec un programme qui :
 - permet d'entrer plusieurs personnes
 - affiche le répertoire
 - permet de supprimer une personne
 - affiche le nouveau répertoire.

Le préprocesseur

- Précède la phase de compilation
- Ajoute, supprime ou remplace des portions de code source sans tenir compte de la syntaxe du langage.
 - Suppression des commentaires
 - Expansion des macros
 - Compilation conditionnelle
 - Modularité

X. Le préprocesseur

- Macros : définit un nom qui sera remplacé par du texte quelconque dans le code source.

```
#define NOM corps
```

- Ex:

```
#define PI 3.1415  
...  
double circ = 2 * PI * rayon;
```

- Ex:

```
#define TAILLE 20  
...  
short tab[TAILLE];  
...  
while (i < TAILLE) {...}
```

X. Le préprocesseur

- Attention, pas ';' à la fin d'une macro :

```
#define TAILLE 10;
```

```
int nbr = TAILLE; // ok, deux ';' à la suite
```

```
int tab[TAILLE]; // erreur, devient tab[10;]
```

→ Et en plus c'est cette dernière ligne que le compilateur indique comme erronée.

- Compilation conditionnelle : permet d'inclure une portion de code en fonction de la valeur ou de l'existence d'une macro.

```
#ifdef NOM
```

```
    code
```

```
[#else
```

```
    code]
```

```
#endif
```


X. Le préprocesseur

→ Si la macro *NOM* est définie (par `#define NOM...`), la portion de code entre le `#ifdef` et le `#else` (ou le `#endif`) est gardée pour le compilateur, si la macro est inconnue, c'est l'inverse.

```
#define MACRO
#ifdef MACRO
    printf("Oui\n");
#else
    printf("Non\n");
#endif
```

→ Oui

```
// #define MACRO
#ifdef MACRO
    printf("Oui\n");
#else
    printf("Non\n");
#endif
```

→ Non

- Ex : inclure des informations de débogage

```
#define DEBUG      /* ou // #define DEBUG */

#include <stdio.h>

int fct(void) {
    #ifdef DEBUG
        printf("-- entree dans fct().\n");
    #endif

    int i = 0;
    i++;

    #ifdef DEBUG
        printf("-- valeur de i : %d\n", i);
    #endif

    return i;
}

int main() {
    printf("Hello world! %d\n", fct());
    return 0;
}
```

- Ex : adaptation du code source à plusieurs architectures entraînant des instructions spécifiques (entrées/sorties graphiques, API système...)

```
#define UNIX  
// #define WINDOWS    /* ou l'inverse... */
```

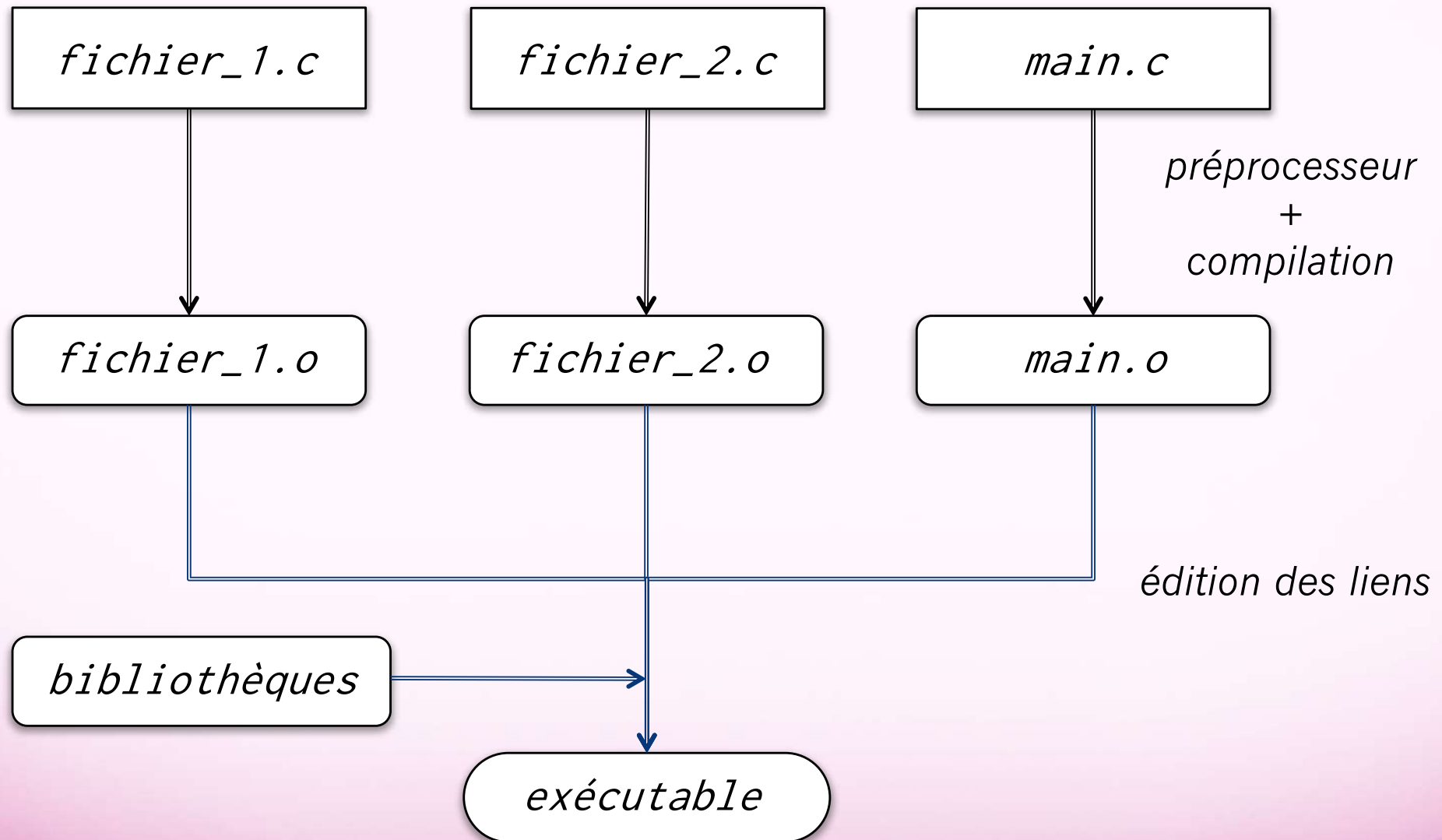
```
#ifdef UNIX  
    // code spécifique à Unix  
#endif
```

```
#ifdef WINDOWS  
    // code spécifique à Windows  
#endif
```

X. Le préprocesseur

- Modularité : découpage du programme en plusieurs fichiers sources
 - développement à plusieurs facilité
 - découpage logique : maintenance et évolution plus faciles
 - isoler certaines parties du code (code non-portable, lien avec le matériel)
 - limiter les interactions entre fonctions : moins de bugs
 - réutilisation d'une partie du code pour d'autres projets.

X. Le préprocesseur



X. Le préprocesseur

- Compilation :
 - traduction des instructions en langage machine et génération de fichiers objets
 - les références non-résolues sont laissées en blanc
 - référence à une variable globale définie dans un autre fichier source
 - appel d'une fonction située (définie) dans un autre fichier source
 - Appel d'une fonction de bibliothèque externe
 - Édition des liens :
 - on regroupe tous les fichiers objets, ainsi que les bibliothèques pour générer un exécutable
- Notion de projet (IDE) qui regroupe tous les fichiers

X. Le préprocesseur

- Pour que l'exécutable soit correct, il doit y avoir exactement UNE fonction `main()` dans l'ensemble du code. C'est le point d'entrée du programme.

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

- les paramètres correspondent à des arguments passés par le shell (ou l'IDE) au lancement du programme :
 - `argc` : nombre d'arguments (il y en aura au moins 1 car le nom de la commande est le premier argument)
 - `argv` : tableau de chaînes, chacune correspondant aux arguments successifs (ex : `argv[0]` est le nom de la commande)
- Convention : `main()` renvoie 0 en cas de succès

X. Le préprocesseur

- Exemple de récupération des arguments.

```
#include <stdio.h>

int main(int argc, char *argv[]) { // ou char **argv
    int i = 0;

    while (i < argc)
        printf("%s\n", argv[i++]);

    if (argc > 2)
        printf("Trop d'args.\n");

    return 0;
}
```

Affichage

```
$/arg 1 2 3 4
./arg
1
2
3
4
Trop d'args.
```

```
$/arg 1
./arg
1
```

```
$/arg 1 2
./arg
1
2
Trop d'args.
```


- Problèmes :
 - comment vérifier les appels à une fonction extérieure ?
 - Comment connaître le type d'une variable globale extérieure ?
- Fichier d'entête : permet d'annoncer aux autres fichiers sources les ressources utilisables
 - prototypes de fonctions
 - déclarations de nouveaux types (**typedef**, structures)
 - annonces de variables externes

X. Le préprocesseur

```
#include <stdio.h>
int vg;

void f1(int i) {
    printf("%d\n", i);
}

double f2(double f) {
    printf("%f\n", f);
    return vg*f*1.3333333;
}
```

fichier.c

- Résultat de la compilation

gcc -o exe main.c fichier.c -Wall

```
main.c:4:5: error: use of undeclared identifier 'vg'
    vg = 2;
    ^
main.c:5:5: warning: implicit declaration of function 'f1'
    is invalid in C99 [-Wimplicit-function-declaration]
    f1(42);
    ^
main.c:6:18: warning: implicit declaration of function 'f2'
    is invalid in C99 [-Wimplicit-function-declaration]
    printf("%f", f2(2));
                  ^
main.c:6:18: warning: format specifies type 'double'
    but the argument has type 'int' [-Wformat]
    printf("%f", f2(2));
                ~~~ ^~~~~~
                %d
```

```
#include <stdio.h>

int main() {
    vg = 2;
    f1(42);
    printf("%f\n", f2(2));
}
```

main.c

variable inconnue

pas de vérification
des appels de fonction

X. Le préprocesseur

```
#include <stdio.h>
#include "fichier.h"
int vg;

void f1(int i) {
    printf("%d\n", i);
}

double f2(double f) {
    printf("%f\n", f);
    return vg*f*1.333333;
}
```

fichier.c

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

int main() {
    vg = 2;
    f1(42);
    printf("%f\n", f2(2));
}
```

main.c

```
/* Fichier d'entête avec
   prototypes et variables externes */
```

```
extern int vg;

void f1(int);
double f2(double);
```

fichier.h

`#include <fichier>`

→ inclut un fichier localisé dans un répertoire système (exemple : un fichier d'entête pour une bibliothèque).

`#include "fichier"`

→ inclut un fichier pris dans le répertoire courant (exemple : un fichier d'entête utilisateur).

X. Le préprocesseur

- Le regroupement de tous les fichiers sources se fait
 - soit dans un projet avec un IDE
 - soit en ligne de commande :

```
gcc -o fich_exe main.c fichier.c
```

- On peut aussi le faire en plusieurs fois :

```
gcc -c main.c (compilation seule)
```

```
gcc -c fichier.c (compilation seule)
```

```
gcc -o fich_exe main.o fichier.o (édition des liens)
```

- On peut ajouter des bibliothèques si nécessaire :

(ici la biblio. mathématique — `#include <math.h>` aura été mis dans le code source pour utiliser les fonctions)

```
gcc -o fich_exe main.c fichier.c -lm
```

X. Le préprocesseur

- Exemples d'IDE

- Code::Blocks (codeblocks.org)

Ne fonctionne pas vraiment sous macOS

- NetBeans (netbeans.apache.org)

*C/C++ pas encore intégré dans la dernière version ;
plugins de la version 8.2 à activer puis installer le C*

- Xcode (App Store)

Que sous macOS ; lourd à utiliser

- Visual Studio Code (code.visualstudio.com)

Pratique à utiliser. Sous macOS, voir www.cs.auckland.ac.nz/~paul/C/Mac/

- Compilateur en ligne (onlinegdb.com)

➤ Notion de projet regroupant les fichiers sources pour générer un exécutable.

➤ On peut exécuter directement depuis l'IDE (fenêtre console) ou depuis le terminal.

Entrées/sorties

- Toutes les entrées/sorties se font sous la forme de flots de données
 - fichiers
 - accès réseaux
 - communications inter-processus
 - accès périphériques...
- On peut utiliser les appels systèmes (open/read/write/close)
- On peut utiliser les appels de bibliothèque (fopen/fread/fwrite/fgets/fprintf.../fclose)

Gestion de fichiers

- Fichier : une suite d'octets sans structure appelée flot.
 - flot caractère (fichier texte) : composé de caractères ASCII imprimables, organisé en lignes (terminées par le caractère local de fin-de-ligne).
 - On peut utiliser des fonctions travaillant sur des lignes (ex : fgets).
 - flot binaire : composé d'octets bruts, copie de la représentation mémoire.
- Dans le cas d'un flot caractère, les fins-de-ligne locales (`\r`, `\n` ou `\r\n`) sont converties en `\n` pour le code.

- Ouverture de fichier

`FILE *fopen(char *nom, char *mode);`

- Le mode d'ouverture peut être

"r" : opérations de lecture ; le fichier doit exister

"w" : opération d'écriture ; le fichier est créé s'il n'existe pas ou effacé s'il existe

"a" : opération d'ajout ; le fichier est créé s'il n'existe pas, sinon on écrit à la fin du fichier

- Le type renvoyé est une structure opaque gérée par la bibliothèque d'entrées/sorties et contenant toutes les informations nécessaires aux opérations. `fopen()` renvoie un pointeur sur une telle structure, pointeur qui sera passé en argument à toutes les autres fonctions d'E/S.

- Trois flots prédéfinis sont ouverts par le système au lancement du programme :

`FILE *stdin` : flot standard d'entrée (clavier)

`FILE *stdout` : flot standard de sortie (écran)

`FILE *stderr` : flot standard de sortie pour les messages d'erreur (écran)

- Fermeture de fichier

`int fclose(FILE *f);`

→ Ferme le flot indiqué et vide les buffers de sortie

→ Se fait automatiquement à la terminaison normale du programme (mais pas en cas de plantage)

- Lecture d'une ligne

`char *fgets(char *buff, int n, FILE *f);`

→ Lit à partir du flot `f` une ligne de caractères de longueur maximale `n-1` et la stocke dans `buff` (tableau de caractères ou zone mémoire allouée).

→ Renvoie `NULL` en cas d'erreur.

→ `fgets(buff, n, stdin)` permet de lire au clavier, comme `gets()`, MAIS avec une limite de taille, ce qui évite un débordement de buffer.

- `fgets()` lit jusqu'à ce que :
 - `n-1` caractères soient lus, ou
 - la fin de ligne est atteinte, ou
 - la fin de fichier est atteinte.

XI. Les entrées/sorties

- Écriture d'une ligne

`char *fputs(char *str, FILE *f);`

→ Écrit dans le flot `f` une chaîne stockée dans `str`.

- Lecture/écriture formatées

`int fscanf(FILE *f, char *format, adr1, adr2, ...)`

`int sscanf(char *src, char *format, adr1, adr2, ...)`

→ Reconnaît les éléments de la chaîne format à partir d'un fichier ou d'une autre chaîne.

`int fprintf(FILE *f, char *format, expr1, expr2, ...)`

`int sprintf(char *dest, char *format, expr1, expr2, ...)`

→ Écrit dans un fichier ou une chaîne le format incluant les expressions.

XI. Les entrées/sorties

- Vidage des buffers

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

- Force la bibliothèque à transmettre les données bufferisées au système pour une sortie immédiate.
- Cela permet d'être sûr que les données dont le programme a demandé l'écriture (écran ou disque) seront immédiatement envoyées en sortie.

```
#include <stdio.h>

int main() {
    FILE *f = fopen("Dauphine.txt", "w");
    fputs("DAUPHINE\n", f);
    int *p = 0;
    *p = 0; // Provoque un plantage
    fclose(f);
    return 0;
}
```

⇒ Rien dans le fichier !

```
#include <stdio.h>

int main() {
    FILE *f = fopen("Dauphine.txt", "w");
    fputs("DAUPHINE\n", f); fflush(f);
    int *p = 0;
    *p = 0; // Provoque un plantage
    fclose(f);
    return 0;
}
```

⇒ Écriture dans le fichier !

- Recopie d'un fichier texte dans un autre

```
#include <stdio.h>
#define TAILLE 256

int main() {
    FILE *fIN  = NULL;
    FILE *fOUT = NULL;
    char buff[TAILLE];
    if (!(fIN = fopen("fichier.txt", "r")))
        printf("Ouverture du fichier impossible\n");
    else if (!(fOUT = fopen("fichier_BAK.txt", "w")))
        printf("Erreur de fichier de sortie\n");
    else
        while (fgets(buff, TAILLE, fIN))
            fputs(buff, fOUT);
    fclose(fIN);
    fclose(fOUT);
    return 0;
}
```

- Certains fichiers ne peuvent pas être lus en mode texte : exécutables, fichiers compressés, copies mémoire... On travaille alors sur des suites d'octets (en ajoutant **"b"** au mode d'ouverture).

```
int fread (void *dest, size_t taille, size_t nbr, FILE *f);
```

→ lit sur le flot **f** un maximum de **nbr** éléments de taille indiquée et les stocke dans le buffer **dest** (qui doit être de taille suffisante)

→ la fonction renvoie le nombre d'éléments effectivement lus.

```
int fwrite(void *src, size_t taille, size_t nbr, FILE *f);
```

→ écrit sur le flot **f** **nbr** éléments de taille indiquée se trouvant en mémoire à partir de l'adresse **src**

- Exemples

- Écriture d'un entier dans le fichier binaire :

```
int i = 42;  
fwrite(&i, sizeof(int), 1, file);
```

- Écriture d'un tableau :

```
double T[10];  
fwrite(T, sizeof(double), 10, file);
```

- Attention, les écritures binaires ne sont pas portables (taille de type, ordre d'écriture en mémoire...)

- Recopie d'un fichier binaire dans un autre

```
#include <stdio.h>
#define NOMBRE 1024

int main() {
    FILE *fIN  = NULL;
    FILE *fOUT = NULL;
    char buff[NOMBRE];
    int nbr;
    if (!(fIN = fopen("essai", "rb")))
        printf("Ouverture du fichier impossible\n");
    else if (!(fOUT = fopen("sortie", "wb")))
        printf("Erreur de fichier de sortie\n");
    else
        while ((nbr = fread(buff, 1, NOMBRE, fIN)))
            fwrite(buff, 1, nbr, fOUT);
    fclose(fIN);
    fclose(fOUT);
    return 0;
}
```


- Positionnement dans un fichier

```
int fseek(FILE *f, long offset, int where);
```

→ L'indicateur de position courante dans le flot `f` est fixé à `offset` octets à partir de la position `where`

- `where` peut valoir :

`SEEK_SET` : déplacement compté à partir du début

`SEEK_CUR` : déplacement compté à partir de la pos. courante

`SEEK_END` : déplacement compté à partir de la fin

- Ex: `fseek(f, 0, SEEK_SET)` permet de revenir au début du fichier.
- Il est difficile de compter les octets dans un fichier texte en raison des différentes fins-de-ligne possibles.

Tableaux multidimensionnels

- Les tableaux multidimensionnels sont des tableaux de tableaux formels

- Déclaration

```
type id[dim_1][dim_2]...[dim_n];
```

- Exemples

```
unsigned long T[10][10];  
char *P[5][2];  
struct cellule C[100][50];
```

XII. C divers

- Initialisation

On peut mettre les différentes lignes entre {...}

```
/* tab. à 3 lignes de 4 colonnes */  
short T[3][4] = { {1,2,3,4},  
                  {5,6,7,8},  
                  {9,10,11,12}  
                };
```

T ↓	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

Les lignes incomplètes sont complétées par des 0 :

```
/* tab. à 3 lignes de 4 colonnes */  
short T[3][4] = { {1,2,3},  
                  {5,6},  
                  {9,10,11}  
                };
```

T ↓	0	1	2	3
0	1	2	3	0
1	5	6	0	0
2	9	10	11	0

Dans ces deux cas, le nombre de lignes est facultatif car le compilateur peut le retrouver

```
Short T[][4] = ...;
```

XII. C divers

On peut également supprimer les accolades internes des lignes. Le tableau est alors rempli ligne par ligne de manière linéaire (seuls les derniers éléments peuvent être initialisés à 0).

```
/* tab. à 3 lignes de 4 colonnes */  
short T[3][4] = { 1,2,3,  
                  5,6,  
                  9,10,11  
                };
```

T	0	1	2	3
0	1	2	3	5
1	6	9	10	11
2	0	0	0	0

Attention, sans indication du nombre de lignes, le compilateur ajuste la tableau à la taille strictement nécessaire :

```
/* tab. à ? lignes de 4 colonnes */  
short T[][4] = { 1,2,3,  
                5,6,  
                9,10,11  
              };
```

T	0	1	2	3
0	1	2	3	5
1	6	9	10	11

Notons que dans ces deux cas, l'indentation et les retours à la ligne sont sans effet et même sans rapport avec le stockage (ce qui peut induire un relecteur inattentif en erreur).

XII. C divers

- Accès

Il suffit de mettre l'indice de chaque dimension entre []

```
short A[5][4], B[5][4], C[5][4];  
/* notez les définitions regroupées */
```

```
...  
for (i = 0; i < 5; i++)  
    for (j = 0; j < 4; j++)  
        C[i][j] = A[i][j] + B[i][j];
```

Attention à ne pas écrire `T[i,j]` au lieu de `T[i][j]`.

Cela provoque simplement un warning de compilation.

→ Car `i,j` est une expression qui vaut `j`, donc `T[i,j]` est équivalent à `T[j]` qui est de type pointeur de tableau, compatible avec un type entier...

XII. C divers

- Stockage

Le stockage mémoire d'un tableau multidimensionnel se fait de manière linéaire, ligne après ligne.

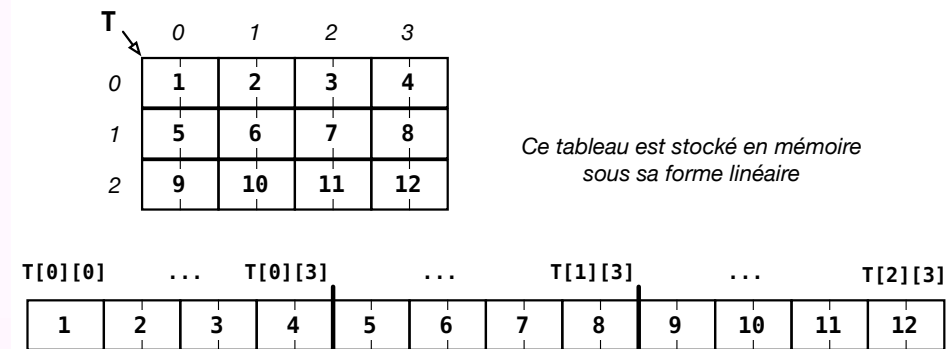
Il vaut donc mieux organiser le parcours d'un tableau dans l'ordre de son stockage (par exemple pour maximiser l'efficacité des accès cache).

```
int A[2000][2000];

int main () {
    int n, i, j;

    for (n = 0; n < 500; n++)
        for (i = 0; i < 2000; i++)
            for (j = 0; j < 2000; j++)
                A[i][j] = 1;

    return 0;
}
```



→ Ce programme s'exécute en ~2,5 s mais si on intervertit les indices **i** et **j** en écrivant **A[j][i] = 1;** le temps d'exécution passe à ~6 s !

XII. C divers

- Tableaux multidimensionnels en argument
 - Comme toujours, c'est l'adresse du premier élément qui est passé à la fonction.

Pb : quelle est l'adresse de $T[i][j]$?

➤ Il s'agit de $i * \text{NbrColonnes} + j$

- Il faut donc que la fonction connaisse le nombre de colonnes pour pouvoir calculer cette adresse lors de l'accès !

```
void fct(short T[][]) {...}    /* erreur de compilation */  
void fct(short T[][4]) {...}   /* ok */
```

- On pourra accéder à $T[i][j]$ dans la fonction.

XII. C divers

→ Et si la taille du tableau n'est pas une constante ?

(ex : un tableau dynamique passé à une fonction)

→ Il faut accéder au tableau de manière linéaire en calculant l'adresse de chaque élément à la main

→ le tableau n'est plus vu comme multidimensionnel

→ la fonction doit connaître le nombre de colonnes qu'il faut donc passer en argument !

```
void fct(int nbrCol, short *T) {  
    /* accès à l'élément T[i][j] */  
    T[i * nbrCol + j] = ... ;  
}
```

→ Et comme d'habitude, il faut aussi passer le nombre de lignes en argument si la fonction a besoin de le connaître (par exemple pour afficher tout le tableau).

XII. C divers

```
#include <stdio.h>
/* Affiche une matrice carrée ligne par ligne */
void affiche(int taille, int *T) {
    int i, j;
    for (i = 0; i < taille; i++) {
        for (j = 0; j < taille; j++)
            printf("%3d ", T[i * taille + j]); /* accès linéaire */
        printf("\n");
    }
}

/* Additionne deux matrices carrées */
void add(int taille, int *A, int *B, int *C) {
    int i, j;
    for (i = 0; i < taille; i++) {
        for (j = 0; j < taille; j++) {
            C[i*taille + j] = A[i*taille + j] + B[i*taille + j];
        }
    }
}

int main() {
    int A[3][3] = {{1, 2, 3}, {4, 5, 6}, { 7, 8, 9}};
    int B[3][3] = {{0, 1,-1}, {1, 0,-1}, {-1, 1, 0}};
    int C[3][3];

    add(3, &A[0][0], &B[0][0], &C[0][0]);
    affiche(3, &C[0][0]);
    return 0;
}
```

Dans les appels de fonctions, on aurait pu utiliser A au lieu de &A[0][0] (les deux ont pour valeur l'adresse de début du tableau) mais cela aurait généré un avertissement de compilation pour l'utilisation de pointeurs de type différents (A est de type pointeur sur un tableau d'entiers et pas pointeur d'entiers).

Pointeurs de fonctions

- Une fonction correspond à du code stocké en mémoire.
- On peut donc pointer sur l'adresse de début de ce code.
- Comme pour les variables, il faut connaître le « type » pointé, c'est-à-dire ici le prototype de la fonction (type de retour, nombre d'arguments et leurs types).

*T (*Id)(type des arguments)*

- ***Id** est une fonction renvoyant un type T et ayant les arguments indiqués, donc **Id** est une variable de type « pointeur sur une fonction de ce type ».

- Exemples

```
void (*p)(void);  
double (*p)(double, char);  
int *(*p)(int *, double *);  
struct s *(*p)(double (*t)(int));
```

- Initialisation

L'identificateur d'une fonction correspond à son adresse, on peut donc s'en servir pour initialiser un pointeur :

```
double F(void) { ... }  
double (*ptrF)(void);  
ptrF = F;
```

- Utilisation

Comme pour tout pointeur, il suffit de le dérérérencer pour accéder à la zone pointée, c'est-à-dire exécuter la fonction : `(*ptrF)()` est l'appel de la fonction pointée par `ptrF`.

- Exemple d'utilisation d'un pointeur de fonction

```
#include <stdio.h>

double F(int i) {
    return 1.5 * i;
}

int main(int argc, char *argv[]) {
    double (*ptr)(int) = F;

    /* appel de la fonction pointée */
    printf("%f\n", (*ptr)(28) );
    return 0;
}

/* Affiche : 42.000000 */
```

- Exemples d'utilisation

- Fonction en argument

- Ex : calculer le zéro d'une fonction par dichotomie ; on passe un pointeur sur la fonction cible en argument d'une fonction générale implémentant la méthode de recherche.
- Ex : la fonction `quicksort()` permet de trier un tableau à partir d'une comparaison de ses éléments ; elle prend comme argument un pointeur sur la fonction de comparaison écrite par le programmeur.

- Tableau de fonctions

- On peut regrouper des pointeurs de fonction dans un tableau afin d'aiguiller plus facilement sur une fonction à exécuter en fonction d'une donnée.

XII. C divers

- Exemple d'utilisation d'un tableau de pointeurs de fonction

```
#include <stdio.h>

int f1(void) { return 10; }
int f2(void) { return 20; }
int f3(void) { return 30; }

int (*T[4])(void) = {NULL, f1, f2, f3};
int main() {
    int res, input;

    while(1) {
        printf("Input ? ");
        scanf("%d", &input);
        if (input == 0)
            break;
        if ((0 < input) && (input < 4))
            /* appel de la fonction pointée par T[input] */
            res = (*T[input])();
        else
            res = 0;
        printf("resultat = %d\n", res);
    }
    return 0;
}
```