Programmation en C et structures de données

Guillaume Revy

guillaume.revy@univ-perp.fr

Université de Perpignan Via Domitia

Prérequis et organisation du cours

- Prérequis : bonne connaissance en algorithmique
  - Algorithmique 1 et 2 de L1
- Organisation :
  - 15h de CM = 10 séances de 1.5h
  - 45h de TP : 30 séances 1.5h
- Évaluation :
  - 50% contrôle continu : 2 épreuves sur machine (entre 3h et 4h)
  - 50% examen terminal : 1 épreuve écrite (2h)

Introduction (1/2)

- 1972 : invention du langage C par Dennis Ritchie et Ken Thompson
  - laboratoires Bell
  - développement d'un système UNIX portable
- 1983 → 1989 : normalisation
  - American National Standard Institute
  - C ANSI
  - depuis, plusieurs versions du standard ont été adoptées (C99, C11, C14, ...)
- Deux références :
  - *The C programming language*, Brian Kernighan et Dennis Ritchie (1978)
  - *Programmation avancée en C*, Sébastien Varrette et Nicolas Bernard (2007)

Introduction (2/2)

- Il existe plusieurs types de langages
  - langage machine : bas niveau (instructions spécifiques à un processeurs)
  - langage assembleur : instructions de base
  - langage de haut niveau
    - impératif : suite d'instructions exécutées de manière séquentielle
    - orienté objet (C++, Java, ... par exemple)
    - déclaratif : fonctionnel (Caml) ou logique (Prolog, Lisp, ...)
- Les langages de haut niveau sont soit interprétés soit compilés
- Le langage C est un langage impératif et compilé
  - contrairement au langage Python (cf. L1), qui est interprété

Que va-t-on voir dans ce cours ?

Les éléments de base du C

Les éléments de base du C

Les éléments de base du C

Un premier exemple

- On part d'un fichier source (extension .c) : texte (hello.c).

```c
/* Role: mon premier programme C
   Auteur: ... */

#include <stdio.h>  // inclusion de l'entete standard des E/S

int main()          // programme principal
{                   // debut des instructions
  printf("Hello world ! \n");
  return 0;
}                   // fin des instructions
```

- Ensuite, on utilise un compilateur (ici, `gcc` ou `clang`), pour transformer le fichier source en un binaire (a.out), exécutable sur la machine cible.

```
$> gcc hello.c
$> ./a.out
Hello world !
```

```
$> clang hello.c
$> ./a.out
Hello world !
```

Structure d'un programme C

```c
/* Role: mon premier programme C
   Auteur: ... */

#include <stdio.h>  // inclusion de l'entete standard des E/S

int main()          // programme principal
{                   // debut des instructions
  printf("Hello world ! \n");
  return 0;
}                   // fin des instructions
```

- En-tête en commentaire, décrivant le fichier source
- Inclusion des fichiers d'en-tête : directive `#include ...`
- Déclaration des fonctions, des constantes et des macros
- Définition des fonctions et notamment de la fonction principale `main`
- Remarques importantes :
  - commentaires matérialisés par les symboles `/* ... */` (bloc) ou `//` (ligne)
  - instructions délimitées par un point-virgule `;`
  - blocs d'instructions matérialisés par des accolades `{}`
  - affichage avec l'instruction `printf(...)`
  - retour du programme grâce à l'instruction `return ...`

Les éléments de base du C

Qu'est-ce qu'une variable ?

- Rappel de L1 : Une variable est une donnée qu'un programme peut manipuler. Toute variable possède un type, un identificateur (nom), et une valeur correspondant à son type.
- En langage C, toute variable doit être déclarée avant sa première utilisation.
- Pour déclarer une variable, on doit indiquer son type et son identificateur.
- Principaux types du langage C :
  - entier : `short` (16 bits), `int` (32 bits), `long int` (64 bits)
    → 0, 17, −208, ...
  - caractère : `char` (8 bits)
    → 'a', 'A', ';', ...
  - réel : `float` (32 bits), `double` (64 bits), `long double` (80 bits)
    → 2.0, 0.3, −1.17, ...
  - modificateur de représentation (entier et caractère) : `unsigned` (non signé → positif)

Déclaration et initialisation d'une variable

- En langage C, toute variable doit être déclarée avant sa première utilisation.
- Pour déclarer une variable, on doit indiquer son type et son identificateur.

```
int var1;              // declaration d'une variable 'var1'
                       //   de type 'int' (entier 32 bits)
short var2, var3;      // declaration de deux variables 'var2' et 'var3'
                       //   de type 'short' (entier 16 bits)
```

- Dans ce cas, les trois variables `var1`, `var2` et `var3` n'ont pas de valeur initiale.
- Pour initialiser une variable, c'est-à-dire, lui donner une valeur, à la déclaration, il faut utiliser la syntaxe suivante.

```
int var4 = 17;         // declaration d'une variable 'var4'
                       //   de type 'int' (entier 32 bits)
                       //   et initialisation de 'var4' avec la valeur 17
```

- Dans ce cas, la variable `var4` a pour valeur 17.

Affectation d'une valeur à une variable

- Rappel de L1 : L'affectation est une opération qui permet de donner ou d'affecter une valeur à une variable. La valeur de la variable sera alors modifiée.
- En langage C, le symbole d'affectation est le symbole '='.

```
int var5;          // declaration d'une variable 'var5'
                   //   de type 'int' (entier 32 bits)
var5 = 17;         // affectation de la valeur 17 a la variable 'var5'
```

Affectation d'une valeur à une variable

- Rappel de L1 : L'affectation est une opération qui permet de donner ou d'affecter une valeur à une variable. La valeur de la variable sera alors modifiée.

- En langage C, le symbole d'affectation est le symbole '='.

```
int var5;          // declaration d'une variable 'var5'
                   //   de type 'int' (entier 32 bits)
var5 = 17;         // affectation de la valeur 17 a la variable 'var5'
```

- L'élément de droite du symbole d'affectation peut être :
  - le nom d'une variable : la valeur de cette variable sera affectée,
  - une expression : cette expression sera évaluée, et son résultat sera affecté.

Les éléments de base du C

Fonction d'affichage

- En langage C, l'affichage sur la sortie standard se fait avec la fonction `printf`.
- Son utilisation est le suivante

```
printf(expression, var1, var2, ...)
```

où

  - `expression` est une chaîne de caractères contenant, notamment, l'emplacement des variables,
  - et `var1`, `var2`, ... est la liste des variables à afficher.

- Remarques importantes :
  - l'emplacement d'une variable est matérialisé par le caractère `%` suivi d'un ou plusieurs caractères indicant le type de la variable attendue.
  - `int` / `short` → `%d`
    `unsigned int` / `unsigned short` → `%u`
    `float` → `%f`
    `double` → `%lf`
    `long double` → `%Lf`
    `long int` → `%ld`
    `unsigned long int` → `%lu`
    `char` → `%c` / `char[]` → `%s`
    `float` / `double` / `long double` → `%e`

Exemple d'affichage

```c
#include <stdio.h>

int main()
{
  int p = 10;
  unsigned int m = 4;

  printf("J'ai achete %d pommes, et j'en ai deja mangees %u. \n", p, m);

  return 0;
}
```

- Dans ce programme :
  - `%d` est remplacé par la valeur de la variable `p`, c'est-à-dire, la valeur entière 10,
  - et `%u` est remplacé par celle de la variable `m`, c'est-à-dire, la valeur entière 4.

```
$> gcc printf.c
$> ./a.out
J'ai achete 10 pommes, et j'en ai deja mangees 4.
```

Fonction de lecture

- En langage C, la lecture de données sur l'entrée standard (lecture au clavier) se fait avec la fonction `scanf`.
- Son utilisation est le suivante

```
scanf(formats, &var1, &var2, ...)
```

où

  - `formats` est chaîne de caractères contenant, notamment, les formats des variables à lire, matérialisés par des caractères `%` (de la même manière que pour l'affichage),
  - et `var1`, `var2`, ... est la liste des variables dont la valeur doit être lue au clavier.

- Remarques importantes :
  - la fonction `scanf` renvoie le nombre d'éléments lus (et dont la conversion vers le type souhaité a pu être effectuée), et `EOF` si une erreur est survenue,
  - le `&` devant le nom des variables n'est pas une erreur, ceci permet d'indiquer que la fonction modifiera la valeur de la variable.

Exemple de lecture

```c
#include <stdio.h>

int main()
{
  int i;
  float f;

  printf("Lecture d'un entier et d'un flottant \n");
  scanf("%d", &i);
  printf("La valeur i=%d \n", i);

  scanf("%d %f", &i, &f);
  printf("Les valeurs i=%d et f=%f\n", i, f);

  return 0;
}
```

```
$> gcc scanf.c
$> ./a.out
Lecture d'un entier et d'un flottant
3
La valeur i=3
3 4.5
Les valeurs i=3 et f=4.500000
```

Exercice

Écrire un programme C qui lit deux entiers au clavier, et qui calcule puis affiche la somme de ces deux entiers sur la sortie standard.

Exercice

Écrire un programme C qui lit deux entiers au clavier, et qui calcule puis affiche la somme de ces deux entiers sur la sortie standard.

```c
#include <stdio.h>

int main()
{
  int a, b, somme = 0;            /* declaration de trois entiers 'a', 'b' et */
                                  /* 'somme', et initialisation de 'somme' a la */
                                  /* valeur 0 */

  printf("Entrez deux valeurs entieres: ");
  scanf("%d %d", &a, &b);         /* lecture de 'a' et 'b' */

  somme = a + b;                  /* calcul de la somme de 'a' et 'b' */
  printf("Somme= %d\n", somme);   /* affichage de la valeur calculee */

  return 0;
}
```

Exercice

Écrire un programme C qui lit deux entiers au clavier, et qui calcule puis affiche la somme de ces deux entiers sur la sortie standard.

```c
#include <stdio.h>

int main()
{
  int a, b, somme = 0;            /* declaration de trois entiers 'a', 'b' et */
                                  /* 'somme', et initialisation de 'somme' a la */
                                  /* valeur 0 */

  printf("Entrez deux valeurs entieres: ");
  scanf("%d %d", &a, &b);         /* lecture de 'a' et 'b' */

  somme = a + b;                  /* calcul de la somme de 'a' et 'b' */
  printf("Somme= %d\n", somme);   /* affichage de la valeur calculee */

  return 0;
}
```

```
$> gcc exercice1.c
$> ./a.out
Entrez deux valeurs entieres: 2 3
Somme= 5
```

Les éléments de base du C

Opérations arithmétiques

- En langage C, comme dans les autres langages, il est possible d'effectuer les opérations arithmétiques de base : $+$, $-$, $\times$ ($*$) et $/$.
- Le langage C fournit également l'opérateur modulo, $\%$, qui retourne le reste de la division euclidienne (division entière) : $17 \% 3 = 2$ car $17 = 3 \times 5 + 2$.

Opérations arithmétiques

- En langage C, comme dans les autres langages, il est possible d'effectuer les opérations arithmétiques de base : $+$, $-$, $\times$ ($*$) et $/$.
- Le langage C fournit également l'opérateur modulo, `%`, qui retourne le reste de la division euclidienne (division entière) : 17 `%` 3 = 2 car $17 = 3 \times 5 + 2$.

- Priorité des opérateurs : La priorité des opérateurs est la même que sur les réels, sauf pour le modulo.
  - ordre de priorité : $\times$ et $/$ $\rightarrow$ modulo (`%`) $\rightarrow$ $+$ et $-$
  - `2 % a * 5 + b` $\rightarrow$ `(2 % (a * 5)) + b`

Opérations arithmétiques

- En langage C, comme dans les autres langages, il est possible d'effectuer les opérations arithmétiques de base : $+$, $-$, $\times$ ($*$) et $/$.
- Le langage C fournit également l'opérateur modulo, `%`, qui retourne le reste de la division euclidienne (division entière) : 17 `%` 3 = 2 car $17 = 3 \times 5 + 2$.

- Priorité des opérateurs : La priorité des opérateurs est la même que sur les réels, sauf pour le modulo.
  - ordre de priorité : $\times$ et $/$ $\rightarrow$ modulo (`%`) $\rightarrow$ $+$ et $-$
  - `2 % a * 5 + b` $\rightarrow$ `(2 % (a * 5)) + b`

- Ordre d'évaluation : Sauf si l'on utilise des parenthèses, les opérations d'une même expression sont évaluées de gauche à droite.
  - `a + b + c + d` $\rightarrow$ `((a + b) + c) + d`

Opérations arithmétiques

- En langage C, comme dans les autres langages, il est possible d'effectuer les opérations arithmétiques de base : $+$, $-$, $\times$ ($*$) et $/$.
- Le langage C fournit également l'opérateur modulo, `%`, qui retourne le reste de la division euclidienne (division entière) : 17 `%` 3 = 2 car $17 = 3 \times 5 + 2$.

- Priorité des opérateurs : La priorité des opérateurs est la même que sur les réels, sauf pour le modulo.
  - ordre de priorité : $\times$ et $/$ $\rightarrow$ modulo (`%`) $\rightarrow$ $+$ et $-$
  - `2 % a * 5 + b` $\rightarrow$ `(2 % (a * 5)) + b`

- Ordre d'évaluation : Sauf si l'on utilise des parenthèses, les opérations d'une même expression sont évaluées de gauche à droite.
  - `a + b + c + d` $\rightarrow$ `((a + b) + c) + d`

- Conversion implicite de type : Si une opération arithmétique fait intervenir deux opérandes de types différents (un entier et un réel, par exemple) :
  - les opérandes sont implicitement convertis vers le type le plus complexe,
  - l'opération est ensuite effectuée.

Exemple de calculs arithmétiques

```c
#include <stdio.h>

int main()
{
  float celsius, fahrenheit;

  printf("Entrez la temperature en C: ");
  scanf("%f", &celsius);

  fahrenheit = 9/5 * celsius + 32;

  printf(" ... ce qui donne %f C = %f F \n", celsius, fahrenheit);

  return 0;
}
```

Exemple de calculs arithmétiques

```c
#include <stdio.h>

int main()
{
  float celsius, fahrenheit;

  printf("Entrez la temperature en C: ");
  scanf("%f", &celsius);

  fahrenheit = 9/5 * celsius + 32;

  printf(" ... ce qui donne %f C = %f F \n", celsius, fahrenheit);

  return 0;
}
```

```
$> gcc operation.c
$> ./a.out
Entrez la temperature en C: 17
... ce qui donne 17.000000 C = 49.000000 F
```

- ... alors que $17°C \approx 62.6°F$!!

Exemple de calculs arithmétiques ... corrigé

```c
#include <stdio.h>

int main()
{
  float celsius, fahrenheit;

  printf("Entrez la temperature en C: ");
  scanf("%f", &celsius);

  fahrenheit = 9.0/5 * celsius + 32; /* ou bien : ((float)9)/5 */

  printf(" ... ce qui donne %f C = %f F \n", celsius, fahrenheit);

  return 0;
}
```

Exemple de calculs arithmétiques ... corrigé

```c
#include <stdio.h>

int main()
{
  float celsius, fahrenheit;

  printf("Entrez la temperature en C: ");
  scanf("%f", &celsius);

  fahrenheit = 9.0/5 * celsius + 32; /* ou bien : ((float)9)/5 */

  printf(" ... ce qui donne %f C = %f F \n", celsius, fahrenheit);

  return 0;
}
```

```
$> gcc operation-corrige.c
$> ./a.out
Entrez la temperature en C: 17
... ce qui donne 17.000000 C = 62.599998 F
```

- ... faites donc attention au type des opérandes ! !

Fonctions mathématiques

- Les fonctions mathématiques telles que $\sqrt{\cdot}$, $\cos(\cdot)$, $\sin(\cdot)$, $\exp(\cdot)$, $\log(\cdot)$, ... et les constantes (π, e, ...) sont définies et accessibles depuis une librairie séparée.
- Pour les utiliser, il faut ajouter :
  - `#include <math.h>` en début du fichier source,
  - `-lm` sur la ligne de compilation.

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

int main()
{
  float f = 2., r;
  r = sqrtf(f); // sqrtf(...) -> racine carree de type 'float'

  printf("sqrt(%f) = %f \n", f, r);
  printf("valeur de pi = %f \n", M_PI); // M_PI : constante PI

  return 0;
}
```

```
$> gcc fonction-sqrt.c -lm
$> ./a.out
sqrt(2.) = 1.414214
valeur de pi = 3.141593
```

Exercice

Écrire un programme C qui lit un réel (flottant) au clavier, représentant un angle, et qui calcule puis affiche le sinus et le cosinus de cet angle sur la sortie standard.

Exercice

Écrire un programme C qui lit un réel (flottant) au clavier, représentant un angle, et qui calcule puis affiche le sinus et le cosinus de cet angle sur la sortie standard.

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

int main()
{
  float a, c, s;                    /* declaration de l'angle 'a', */
                                    /* du cosinus 'c' et du sinus 's */

  printf("Entrez l'angle 'a': ");
  scanf("%f", &a);                  /* lecture de l'angle 'a' */

  c = cosf(a);                      /* calcul du cosinus */
  s = sinf(a);                      /* calcul du sinus */
  printf("cos= %f et sin= %f \n", c, s);  /* affichage des valeurs calculees */

  return 0;
}
```

Exercice

Écrire un programme C qui lit un réel (flottant) au clavier, représentant un angle, et qui calcule puis affiche le sinus et le cosinus de cet angle sur la sortie standard.

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

int main()
{
  float a, c, s;                    /* declaration de l'angle 'a', */
                                    /* du cosinus 'c' et du sinus 's */

  printf("Entrez l'angle 'a': ");
  scanf("%f", &a);                  /* lecture de l'angle 'a' */

  c = cosf(a);                      /* calcul du cosinus */
  s = sinf(a);                      /* calcul du sinus */
  printf("cos= %f et sin= %f \n", c, s);  /* affichage des valeurs calculees */

  return 0;
}
```

```
$> gcc exercice2.c -lm
$> ./a.out
Entrez l'angle 'a': 3.14
cos= -0.999999 et sin= 0.001593
```

Exercice

Écrire un programme C qui lit un réel (flottant) au clavier, représentant le rayon d'un disque, et qui calcule puis affiche l'aire de ce disque sur la sortie standard.

Exercice

Écrire un programme C qui lit un réel (flottant) au clavier, représentant le rayon d'un disque, et qui calcule puis affiche l'aire de ce disque sur la sortie standard.

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

int main()
{
  float r, a;                    /* declaration du rayon 'r' et */
                                 /*   de l'aire 'a' */

  printf("Entrez le rayon du disque: ");
  scanf("%f", &r);               /* lecture du rayon 'r' */

  a = M_PI * (r * r);            /* calcul de l'aire 'a' */
  printf("Aire= %f \n", a);      /* affichage de la valeur calculee */

  return 0;
}
```

Exercice

Écrire un programme C qui lit un réel (flottant) au clavier, représentant le rayon d'un disque, et qui calcule puis affiche l'aire de ce disque sur la sortie standard.

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

int main()
{
  float r, a;                    /* declaration du rayon 'r' et */
                                 /*   de l'aire 'a' */

  printf("Entrez le rayon du disque: ");
  scanf("%f", &r);               /* lecture du rayon 'r' */

  a = M_PI * (r * r);            /* calcul de l'aire 'a' */
  printf("Aire= %f \n", a);      /* affichage de la valeur calculee */

  return 0;
}
```

```
$> gcc exercice3.c -lm
$> ./a.out
Entrez le rayon du disque: 2.5
Aire du disque= 19.634954
```

Que va-t-on voir dans ce cours ?

(1) Les éléments de base du C

(2) Les structures de contrôle

(3) Les tableaux et les structures

(4) Les pointeurs et l'allocation dynamique

(5) Les fonctions

(6) Les autres types dérivés

(7) Les fichiers

(8) Quelques éléments avancés

(9) Complément 1 : Les listes chaînées

(10) Complément 2 : Les piles et les files

Les structures de contrôle

Les structures de contrôle

1. Qu'est qu'une condition ?

2. Structures conditionnelles

3. Structures itératives

4. Remarques importantes

Les structures de contrôle

Qu'est ce qu'une condition ?

- Rappel de L1 : Une condition est une comparaison, composée de trois éléments :
  1. une première variable ou constante,
  2. un opérateur de comparaison,
  3. et une deuxième variable ou constante.
- Opérateurs de comparaison : <, <=, >, >=, !=, ==.
- Remarque importante : le langage C ne fournit pas de type booléen, on utilisera alors des valeurs entières
  - valeur nulle ($= 0$) : condition fausse,
  - valeur non nulle ($\neq 0$) : condition vraie.

Qu'est ce qu'une condition ?

- Rappel de L1 : Une condition est une comparaison, composée de trois éléments :
  1. une première variable ou constante,
  2. un opérateur de comparaison,
  3. et une deuxième variable ou constante.
- Opérateurs de comparaison : <, <=, >, >=, !=, ==.
- Remarque importante : le langage C ne fournit pas de type booléen, on utilisera alors des valeurs entières
  - valeur nulle (= 0) : condition fausse,
  - valeur non nulle (≠ 0) : condition vraie.

```c
#include <stdio.h>

int main()
{
  int a = 6;
  printf("Valeur [a == 5] => %d \n", (a == 5));
  printf("Valeur [a == 6] => %d \n", (a == 6));
}
```

```
$> gcc def-condition.c
$> ./a.out
Valeur [a == 5] => 0
Valeur [a == 6] => 1
```

Condition composée

- Rappel de L1 : Certains problèmes exigent parfois de formuler des conditions qui ne peuvent pas être exprimées sous une forme simple : on a donc besoin de les composer, en les reliant par des opérateurs logiques.
- Opérateurs logiques : ET (`&&`), OU (`||`), NON (`!`)
  - le langage C ne fournit pas d'opérateur XOR (ou exclusif)

`C1`	`C2`	`C1 && C2`	`C1 || C2`	`!C2`
faux	faux	faux	faux	vrai
faux	vrai	faux	vrai	faux
vrai	faux	faux	vrai	–
vrai	vrai	vrai	vrai	–

Condition composée

- Rappel de L1 : Certains problèmes exigent parfois de formuler des conditions qui ne peuvent pas être exprimées sous une forme simple : on a donc besoin de les composer, en les reliant par des opérateurs logiques.
- Opérateurs logiques : ET (`&&`), OU (`||`), NON (`!`)
  - le langage C ne fournit pas d'opérateur XOR (ou exclusif)

`C1`	`C2`	`C1 && C2`	`C1 || C2`	`!C2`
faux	faux	faux	faux	vrai
faux	vrai	faux	vrai	faux
vrai	faux	faux	vrai	–
vrai	vrai	vrai	vrai	–

- Remarque importante : L'évaluation d'une condition composée s'effectue de gauche à droite. Dès lors que le résultat de cette condition est connu (faux pour le ET, vrai pour le OU), l'évaluation est interrompue. On parle d'évaluation paresseuse ou en court-circuit.

Les structures de contrôle

Structures conditionnelles en C

- Condition si - alors

```
if(condition)
  {
    // suite d'instructions
  }
```

  - dans ce cas, la suite d'instructions sera exécutée si la condition est vraie,
  - l'indentation du bloc `{...}` n'a pas d'impact sur le flot d'exécution.

- Condition si - alors - sinon

```
if(condition)
  {
    // suite d'instructions (1)
  }
else
  {
    // suite d'instructions (2)
  }
```

  - dans ce cas, la suite d'instructions (1) sera exécutée si la condition est vraie,
  - sinon, la condition est fausse, et la suite d'instructions (2) sera alors exécutée.

Exemple de conditionnelle

```c
#include <stdio.h>

int main()
{
  int a;

  printf("Entrez la valeur de 'a': ");
  scanf("%d", &a);

  if(a < 0)
    {
      printf("La variable est negative. \n");
    }
  else
    {
      printf("La variable est positive ou nulle. \n");
    }

  return 0;
}
```

Exemple de conditionnelle

```c
#include <stdio.h>

int main()
{
  int a;

  printf("Entrez la valeur de 'a': ");
  scanf("%d", &a);

  if(a < 0)
    {
      printf("La variable est negative. \n");
    }
  else
    {
      printf("La variable est positive ou nulle. \n");
    }

  return 0;
}
```

```
$> gcc condition.c
$> ./a.out
Entrez la valeur de 'a': 5
La variable est positive ou nulle.
```

```
$> gcc condition.c
$> ./a.out
Entrez la valeur de 'a': -6
La variable est negative.
```

Autre exemple de conditionnelle

```c
#include <stdio.h>

int main()
{
  int test;

  printf("Entrez une valeur pour 'test' : ");
  scanf("%d", &test);

  if(test) {
    printf("Test est != 0. \n");
  }else{
    printf("Test est == 0. \n");
  }
}
```

Autre exemple de conditionnelle

```c
#include <stdio.h>

int main()
{
  int test;

  printf("Entrez une valeur pour 'test' : ");
  scanf("%d", &test);

  if(test) {
    printf("Test est != 0. \n");
  }else{
    printf("Test est == 0. \n");
  }
}
```

```
$> gcc condition-2.c
$> ./a.out
Entrez une valeur pour 'test' : 0
Test est == 0.
```

```
$> gcc condition-2.c
$> ./a.out
Entrez une valeur pour 'test' : 5
Test est != 0.
```

Exercice

*Écrire un programme C qui lit deux notes réelles au clavier, puis qui calcule la moyenne pondérée (*40% – 60%*), et indique si la moyenne est strictement inférieure à* 10*.*

Exercice

Écrire un programme C qui lit deux notes réelles au clavier, puis qui calcule la moyenne pondérée (40% – 60%), et indique si la moyenne est strictement inférieure à 10.

```c
#include <stdio.h>

int main()
{
  float examen, controle, moyenne;

  printf("Entrez la note au controle continu et a l'examen : ");
  scanf("%f %f", &controle, &examen);

  moyenne = controle * 0.4 + examen * 0.6;

  if(moyenne < 10) {
    printf("L'etudiant n'a pas la moyenne [%f] \n", moyenne);
  } else {
    printf("L'etudiant a la moyenne [%f] \n", moyenne);
  }

  return 0;
}
```

Exercice

Écrire un programme C qui lit deux notes réelles au clavier, puis qui calcule la moyenne pondérée (40% – 60%), et indique si la moyenne est strictement inférieure à 10.

```c
#include <stdio.h>

int main()
{
  float examen, controle, moyenne;

  printf("Entrez la note au controle continu et a l'examen : ");
  scanf("%f %f", &controle, &examen);

  moyenne = controle * 0.4 + examen * 0.6;

  if(moyenne < 10) {
    printf("L'etudiant n'a pas la moyenne [%f] \n", moyenne);
  } else {
    printf("L'etudiant a la moyenne [%f] \n", moyenne);
  }

  return 0;
}
```

```
$> gcc exercice1.c
$> ./a.out
Entrez la note au controle continu et a l'examen : 4 18
L'etudiant a la moyenne [12.400000]
```

Remarques sur les structures conditionnelles

- Bien sûr, le langage C autorise d'imbriquer les structures conditionnelles.

```
if(condition1)
{
  if(condition2){
    // si condition1 et condition2 sont vraies
  }
  // si condition1 est vraie, quelque soit la valeur de condition2
}
```

Remarques sur les structures conditionnelles

- Bien sûr, le langage C autorise d'imbriquer les structures conditionnelles.

```
if(condition1)
{
  if(condition2){
    // si condition1 et condition2 sont vraies
  }
  // si condition1 est vraie, quelque soit la valeur de condition2
}
```

- Si la suite d'instructions ne contient qu'une seule instruction, les délimiteurs de bloc { . . . } peuvent être omis.

```
if(condition1)
  // instruction a executer si condition1 est vraie
// instructions a executer quelque soit la valeur de condition1
```

Remarques sur les structures conditionnelles

- Bien sûr, le langage C autorise d'imbriquer les structures conditionnelles.

```
if(condition1)
{
  if(condition2){
    // si condition1 et condition2 sont vraies
  }
  // si condition1 est vraie, quelque soit la valeur de condition2
}
```

- Si la suite d'instructions ne contient qu'une seule instruction, les délimiteurs de bloc `{...}` peuvent être omis.

```
if(condition1)
  // instruction a executer si condition1 est vraie
// instructions a executer quelque soit la valeur de condition1
```

- En langage C, quand les délimiteurs de blocs `{...}` sont omis, un `else` fait référence au dernier `if` rencontré.

```
if(condition1)
  if(condition2)
    // condition1 et condition2 sont vraies
  else
    // condition1 est vraie et condition2 est fausse
```

Les branchements

- Lorsqu'une cascade de conditions porte sur une seule et même variable entière et que les tests réalisés sont des tests d'égalité à des valeurs, il est alors possible d'utiliser des branchements, avec la structure `switch-case`.

```
if(var == valeur1) {
  // instruction1
} else
  if (var == valeur2) {
    // instruction2
  } else if (var == valeur3 ||
             var == valeur4) {
    // instruction3
  } else {
    // instruction4
  }
```

Les branchements

- Lorsqu'une cascade de conditions porte sur une seule et même variable entière et que les tests réalisés sont des tests d'égalité à des valeurs, il est alors possible d'utiliser des branchements, avec la structure `switch-case`.

```
if(var == valeur1) {
  // instruction1
} else
  if (var == valeur2) {
    // instruction2
  } else if (var == valeur3 ||
             var == valeur4) {
    // instruction3
  } else {
    // instruction4
  }
```

```
switch(var) {
  case valeur1:
    // instruction1
    break;
  case valeur2:
    // instruction2;
    break;
  case valeur3:
  case valeur4:
    // instruction3
    break;
  default:
    // instruction4
    break;
}
```

Les branchements

- Lorsqu'une cascade de conditions porte sur une seule et même variable entière et que les tests réalisés sont des tests d'égalité à des valeurs, il est alors possible d'utiliser des branchements, avec la structure `switch-case`.

```
if(var == valeur1) {
  // instruction1
} else
  if (var == valeur2) {
    // instruction2
  } else if (var == valeur3 ||
             var == valeur4) {
    // instruction3
  } else {
    // instruction4
  }
```

```
switch(var) {
  case valeur1:
    // instruction1
    break;
  case valeur2:
    // instruction2;
    break;
  case valeur3:
  case valeur4:
    // instruction3
    break;
  default:
    // instruction4
    break;
}
```

- L'instruction `break` doit être ajoutée à la fin de chaque cas, sinon les cas suivants seront également traîtés.

Exemple de branchement

```c
#include <stdio.h>

int main()
{
  int a;
  printf("Entrez une valeur pour 'a' = ");
  scanf("%d", &a);

  switch(a) {
  case 1:
    printf("valeur de 'a' = '1' \n");
  case 2:
    printf("valeur de 'a' = '2' \n");
    break;
  case 3:
  case 4:
    printf("valeur de 'a' = '3' ou '4' \n");
    break;
  default:
    printf("valeur de 'a' inconnue \n");
    break;
  }
  return 0;
}
```

```
$> gcc branchement.c
$> ./a.out
Entrez une valeur pour 'a' = 1
valeur de 'a' = '1'
valeur de 'a' = '2'

$> ./a.out
Entrez une valeur pour 'a' = 2
valeur de 'a' = '2'

$> ./a.out
Entrez une valeur pour 'a' = 3
valeur de 'a' = '3' ou '4'

$> ./a.out
Entrez une valeur pour 'a' = 4
valeur de 'a' = '3' ou '4'

$> ./a.out
Entrez une valeur pour 'a' = 5
valeur de 'a' inconnue
```

Les structures de contrôle

Boucle `tant que` en C

```
while(condition)
{
  // suite d'instructions
}
```

- dans ce cas, la suite d'instructions sera exécutée tant que la condition est vraie,
- la suite d'instructions doit modifier (à un certain moment) la valeur de vérité de la condition ... sinon, il y aura une boucle infinie,
- si dès l'initialisation, la condition est fausse, la suite d'instructions ne sera jamais exécutée.

Boucle `tant que` en C

```
while(condition)
{
  // suite d'instructions
}
```

- dans ce cas, la suite d'instructions sera exécutée tant que la condition est vraie,
- la suite d'instructions doit modifier (à un certain moment) la valeur de vérité de la condition ... sinon, il y aura une boucle infinie,
- si dès l'initialisation, la condition est fausse, la suite d'instructions ne sera jamais exécutée.

- Si la suite d'instructions ne contient qu'une seule instruction, les délimiteurs de bloc `{...}` peuvent être omis.

```
while(condition)
  // instruction a executer
```

Exemple de boucle `tant que`

```c
#include <stdio.h>

int main()
{
  int a, val;
  printf("Entrez la valeur de 'a' = ");
  scanf("%d", &a);

  val = a;
  while(val >= 0 && val < 10) {
    printf("%d ", val);
    val = val + 1;          // ou | val += 1
                            //    | val ++
  }
  printf("\n");

  return 0;
}
```

Exemple de boucle `tant que`

```c
#include <stdio.h>

int main()
{
  int a, val;
  printf("Entrez la valeur de 'a' = ");
  scanf("%d", &a);

  val = a;
  while(val >= 0 && val < 10) {
    printf("%d ", val);
    val = val + 1;          // ou | val += 1
                            //    | val ++
  }
  printf("\n");

  return 0;
}
```

```
$> gcc tantque.c
$> ./a.out
Entrez la valeur de 'a': 5
5 6 7 8 9
```

```
$> gcc tantque.c
$> ./a.out
Entrez la valeur de 'a': -4
```

Exercice

Écrire un programme C qui lit deux entiers a et b au clavier (a < b), puis affiche les valeurs paires entre a et b.

Exercice

Écrire un programme C qui lit deux entiers a et b au clavier (a < b), puis affiche les valeurs paires entre a et b.

```c
int main() {
  int a, b, val;
  printf("Entrez la valeur de 'a' et 'b' = ");
  scanf("%d %d", &a, &b);

  if(a >= b)
    printf("a [%d] et b [%d] ne respectent pas a < b ... \n", a, b);
  else {
    val = a;
    while(val <= b) {
      if(val % 2 == 0)
        printf("%d ", val);
      val ++;
    }
    printf("\n");
  }

  return 0;
}
```

Exercice

Écrire un programme C qui lit deux entiers a et b au clavier (a < b), puis affiche les valeurs paires entre a et b.

```c
int main() {
  int a, b, val;
  printf("Entrez la valeur de 'a' et 'b' = ");
  scanf("%d %d", &a, &b);

  if(a >= b)
    printf("a [%d] et b [%d] ne respectent pas a < b ... \n", a, b);
  else {
    val = a;
    while(val <= b) {
      if(val % 2 == 0)
        printf("%d ", val);
      val ++;
    }
    printf("\n");
  }

  return 0;
}
```

```
$> gcc exercice2.c
$> ./a.out
Entrez la valeur de 'a' et 'b' = 5 10
6 8 10
```

Boucle `pour` en C

```
for(initialisation; condition; incrementation)
{
  // suite d'instructions
}
```

- `initialisation` : initialisation des variables de la condition
- `condition` : condition de continuation de la boucle
- `incrementation` : modification des variables de la condition

- dans ce cas, la suite d'instructions est exécutée tant que la condition est vraie,
- si dès l'initialisation, la condition est fausse, la suite d'instructions ne sera jamais exécutée,
- par contre, la suite d'instructions n'a pas à modifier la valeur de vérité de la condition.

Boucle `pour` en C

```
for(initialisation; condition; incrementation)
{
  // suite d'instructions
}
```

- `initialisation` : initialisation des variables de la condition
- `condition` : condition de continuation de la boucle
- `incrementation` : modification des variables de la condition

- dans ce cas, la suite d'instructions est exécutée tant que la condition est vraie,
- si dès l'initialisation, la condition est fausse, la suite d'instructions ne sera jamais exécutée,
- par contre, la suite d'instructions n'a pas à modifier la valeur de vérité de la condition.

- Si la suite d'instructions ne contient qu'une seule instruction, les délimiteurs de bloc { . . . } peuvent être omis.

```
for(initialisation; condition; incrementation)
  // instruction a executer
```

Exemple de boucle `pour`

```c
#include <stdio.h>

int main()
{
  int a, val;
  printf("Entrez la valeur de 'a' = ");
  scanf("%d", &a);

  for(val = a; val >= 0 && val < 10; val ++) {
    printf("%d ", val);
  }
  printf("\n");

  return 0;
}
```

Exemple de boucle `pour`

```c
#include <stdio.h>

int main()
{
  int a, val;
  printf("Entrez la valeur de 'a' = ");
  scanf("%d", &a);

  for(val = a; val >= 0 && val < 10; val ++) {
    printf("%d ", val);
  }
  printf("\n");

  return 0;
}
```

```
$> gcc pour.c
$> ./a.out
Entrez la valeur de 'a': 5
5 6 7 8 9
```

```
$> gcc pour.c
$> ./a.out
Entrez la valeur de 'a': -4
```

Exercice

Écrire un programme C qui lit deux entiers a et b au clavier (a < b), puis affiche les valeurs paires entre a et b.

Exercice

Écrire un programme C qui lit deux entiers a et b au clavier (a < b), puis affiche les valeurs paires entre a et b.

```c
int main() {
  int a, b, val;
  printf("Entrez la valeur de 'a' et 'b' = ");
  scanf("%d %d", &a, &b);

  if(a >= b)
    printf("a [%d] et b [%d] ne respectent pas a < b ... \n", a, b);
  else {
    for(val = a; val <= b; val ++) {
      if(val % 2 == 0)
        printf("%d ", val);
    }
    printf("\n");
  }

  return 0;
}
```

Exercice

Écrire un programme C qui lit deux entiers a et b au clavier (a < b), puis affiche les valeurs paires entre a et b.

```c
int main() {
  int a, b, val;
  printf("Entrez la valeur de 'a' et 'b' = ");
  scanf("%d %d", &a, &b);

  if(a >= b)
    printf("a [%d] et b [%d] ne respectent pas a < b ... \n", a, b);
  else {
    for(val = a; val <= b; val ++) {
      if(val % 2 == 0)
        printf("%d ", val);
    }
    printf("\n");
  }

  return 0;
}
```

```
$> gcc exercice3.c
$> ./a.out
Entrez la valeur de 'a' et 'b' = 5 10
6 8 10
```

Boucle répéter en C

```
do
{
  // suite d'instructions
}while(condition);
```

- dans ce cas, la suite d'instructions sera exécutée tant que la condition est vraie,
- la suite d'instructions doit modifier (à un certain moment) la valeur de vérité de la condition ... sinon, il y aura une boucle infinie,
- la suite d'instructions est exécutée au moins une fois ... la condition est vérifiée ensuite.

Boucle `répéter` en C

```
do
{
  // suite d'instructions
}while(condition);
```

- dans ce cas, la suite d'instructions sera exécutée tant que la condition est vraie,
- la suite d'instructions doit modifier (à un certain moment) la valeur de vérité de la condition ... sinon, il y aura une boucle infinie,
- la suite d'instructions est exécutée au moins une fois ... la condition est vérifiée ensuite.

- Si la suite d'instructions ne contient qu'une seule instruction, les délimiteurs de bloc `{...}` peuvent être omis.

```
do
  // instruction a executer
while(condition);
```

Exemple de boucle répéter

```c
#include <stdio.h>

int main()
{
  int a, val;
  printf("Entrez la valeur de 'a' = ");
  scanf("%d", &a);

  val = a;
  do{
    printf("%d ", val);
    val = val + 1;          // ou | val += 1
                            //    | val ++
  }while(val >= 0 && val < 10);
  printf("\n");

  return 0;
}
```

Exemple de boucle répéter

```c
#include <stdio.h>

int main()
{
  int a, val;
  printf("Entrez la valeur de 'a' = ");
  scanf("%d", &a);

  val = a;
  do{
    printf("%d ", val);
    val = val + 1;          // ou | val += 1
                            //    | val ++
  }while(val >= 0 && val < 10);
  printf("\n");

  return 0;
}
```

```
$> gcc repeter.c
$> ./a.out
Entrez la valeur de 'a': 5
5 6 7 8 9
```

```
$> gcc repeter.c
$> ./a.out
Entrez la valeur de 'a': -4
-4
```

Exercice

Écrire un programme qui permet de saisir un choix utilisateur et d'afficher ce choix, tant que l'utilisateur ne veut pas quitter.

Exercice

Écrire un programme qui permet de saisir un choix utilisateur et d'afficher ce choix, tant que l'utilisateur ne veut pas quitter.

```c
#include <stdio.h>

int main()
{
  int choix;

  do {
    printf("---------- \n");
    printf("1. Choix 1 \n");
    printf("2. Choix 2 (quitter) \n");
    printf("Entrez votre choix: ");
    scanf("%d", &choix);

    if(choix != 1 && choix != 2)
      printf("--> choix invalide \n");
    else
      printf("--> choix %d \n", choix);
  } while (choix != 2);
  printf("--> quitter \n");
  return 0;
}
```

```
$> gcc exercice4.c
$> ./a.out
----------
1. Choix 1
2. Choix 2 (quitter)
Entrez votre choix: 1
--> choix 1
----------
1. Choix 1
2. Choix 2 (quitter)
Entrez votre choix: 3
--> choix invalide
----------
1. Choix 1
2. Choix 2 (quitter)
Entrez votre choix: 2
--> choix 2
--> quitter
```

Les structures de contrôle

Quelques problèmes classiques

```c
int main()
{
  float a;

  for(a = 0; a != 1.; a += 0.1)
    printf("-> %f \n", a);

  return 0;
}
```

Quelques problèmes classiques

```c
int main()
{
  float a;

  for(a = 0; a != 1.; a += 0.1)
    printf("-> %f \n", a);

  return 0;
}
```

... du fait des erreurs d'arrondi, la variable `a` ne sera jamais égale à 1

... il faut remplacer `a != 1` par `a < 1`

```c
int main()
{
  unsigned short a;       // a \in [0, 65535]

  for(a = 0; a <= 65535; a ++)
    printf("-> %u \n", a);

  return 0;
}
```

Quelques problèmes classiques

```c
int main()
{
  float a;

  for(a = 0; a != 1.; a += 0.1)
    printf("-> %f \n", a);

  return 0;
}
```

... du fait des erreurs d'arrondi, la variable `a` ne sera jamais égale à 1

... il faut remplacer `a != 1` par `a < 1`

```c
int main()
{
  unsigned short a;       // a \in [0, 65535]

  for(a = 0; a <= 65535; a ++)
    printf("-> %u \n", a);

  return 0;
}
```

... lorsque la variable `a` sera égale à 65535, `a + 1` sera égale à 0

Nouveaux opérateurs

- On a vu dans cette section que le langage C fournit trois opérateurs permettant l'incrémentation d'une variable :
  - `a += b` équivalent à `a = a + b`,
  - `a++` et `++a` équivalent à `a = a + 1` (avec `a` une variable entière).
- En fait, il fournit également `-=`, `*=`, `/=`, $\cdots$, `a--` et `--a`.
- Attention tout de même à l'utilisation de `a++` et `++a`, car l'ordre d'évaluation n'est pas toujours le même, notamment lorsqu'il est utilisé en tant qu'opérande d'une autre opération.

```
int a = 17, b;

b = (a++) - 1;
printf("--> b = %d \n", b);
```

```
$> gcc incrementation1.c
$> ./a.out
--> b = 16
```

Nouveaux opérateurs

- On a vu dans cette section que le langage C fournit trois opérateurs permettant l'incrémentation d'une variable :
  - `a += b` équivalent à `a = a + b`,
  - `a++` et `++a` équivalent à `a = a + 1` (avec `a` une variable entière).
- En fait, il fournit également `-=`, `*=`, `/=`, $\cdots$, `a--` et `--a`.
- Attention tout de même à l'utilisation de `a++` et `++a`, car l'ordre d'évaluation n'est pas toujours le même, notamment lorsqu'il est utilisé en tant qu'opérande d'une autre opération.

```c
int a = 17, b;

b = (a++) - 1;
printf("--> b = %d \n", b);
```

```c
int a = 17, b;

b = (++a) - 1;
printf("--> b = %d \n", b);
```

```
$> gcc incrementation1.c
$> ./a.out
--> b = 16
```

```
$> gcc incrementation2.c
$> ./a.out
--> b = 17
```

Que va-t-on voir dans ce cours ?

(1) Les éléments de base du C

(2) Les structures de contrôle

(3) Les tableaux et les structures

(4) Les pointeurs et l'allocation dynamique

(5) Les fonctions

(6) Les autres types dérivés

(7) Les fichiers

(8) Quelques éléments avancés

(9) Complément 1 : Les listes chaînées

(10) Complément 2 : Les piles et les files

Les tableaux et les structures

Les tableaux et les structures

1. Tableaux uni-dimensionnels

2. Tableaux multi-dimensionnels

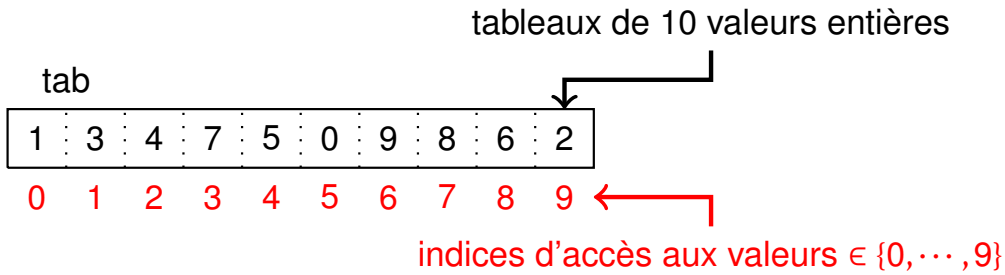
3. Tableaux particuliers : chaînes de caractères

4. Structures

Les tableaux et les structures

Qu'est-ce qu'un tableau uni-dimensionnel ?

- Rappel de L1 : Un tableau 1D est une structure de données unidimensionelle qui permet de stocker un ensemble de valeurs de même type en leur associant un nom commun. L'accès à chacune de ces valeurs se fait par un indice.



- Les éléments d'un tableau sont placés de manière contiguë en mémoire, et sa taille est connue à sa création.
- Ici le nom du tableau est **tab**.
- Les indices du tableau vont de 0 à taille du tableau -1 $(= 9)$.
- L'accès à l'élément d'indice 1 du tableau **tab** se fait de la manière suivante : **tab**[1].

Déclaration d'un tableau uni-dimensionnel

- En langage C, la déclaration d'un tableau 1D se fait de la manière suivante

```
type nom[taille]
```

où

- `type` est le type des éléments du tableau (entier, flottant, ...),
- `nom` est le nom du tableau,
- et `taille` est la taille du tableau, c'est-à-dire, le nombre d'éléments qu'il contient.

```
int tab1[100];          // declaration d'un tableau 'tab1'
                        //   de 100 elements de type 'int' (entier 32 bits)
float tab2[17];         // declaration d'un tableau 'tab2'
                        //   de 17 elements de type float (flottant 32 bits)
```

Déclaration d'un tableau uni-dimensionnel

- En langage C, la déclaration d'un tableau 1D se fait de la manière suivante

  `type nom[taille]`

  où
  - `type` est le type des éléments du tableau (entier, flottant, ...),
  - `nom` est le nom du tableau,
  - et `taille` est la taille du tableau, c'est-à-dire, le nombre d'éléments qu'il contient.

```
int tab1[100];          // declaration d'un tableau 'tab1'
                        //   de 100 elements de type 'int' (entier 32 bits)
float tab2[17];         // declaration d'un tableau 'tab2'
                        //   de 17 elements de type float (flottant 32 bits)
```

- La déclaration d'un tableau de cette manière réserve automatiquement l'espace mémoire nécessaire sur la pile.
- L'accès en lecture/écriture aux éléments du tableau se fait en temps constant.

Initialisation d'un tableau uni-dimensionnel

- L'affectation des éléments d'un tableau se fait élément par élément.

```
int i, tab1[10];
for(i = 0; i < 10; i ++)
  tab1[i] = i;
```

Initialisation d'un tableau uni-dimensionnel

- L'affectation des éléments d'un tableau se fait élément par élément.

```
int i, tab1[10];
for(i = 0; i < 10; i ++)
  tab1[i] = i;
```

- On peut également initialiser les éléments du tableau directement à la déclaration.

```
int tab2[4] = {1, 2, 3, 4};
int tab3[] = {1, 2, 3};
```

  - dans ce cas, la taille de `tab3` est automatiquement déterminée et fixée à 3.

Exemple de tableau 1D

```c
#include <stdio.h>

int main()
{
  int i, tab1[4];
  int tab2[] = {0, 1, 0, 1};

  for(i = 0; i < 4; i++)
    tab1[i] = i;

  for(i = 0; i < 4; i++)
    printf("tab1[%d] = %d et tab2[%d] = %d \n", i, tab1[i], i, tab2[i]);

  return 0;
}
```

```
$> gcc tableau1D.c
$> ./a.out
tab1[0] = 0 et tab2[0] = 0
tab1[1] = 1 et tab2[1] = 1
tab1[2] = 2 et tab2[2] = 0
tab1[3] = 3 et tab2[3] = 1
```

Exercice

Écrire un programme C qui lit et stocke les températures de chaque mois d'une année à Perpignan, et qui calcule et affiche la plus grande et plus petite températures.

Exercice

Écrire un programme C qui lit et stocke les températures de chaque mois d'une année à Perpignan, et qui calcule et affiche la plus grande et plus petite températures.

```c
#include <stdio.h>

int main()
{
  float temperatures[12], min, max;
  int i;

  // ... lecture des donnees
  printf("Entrez la temperature de chaque mois. \n");
  for(i = 0; i < 12; i++) {
    printf("> mois [%d] : ", i);
    scanf("%f", &temperatures[i]);
  }

  // ... calcule et affichage de la temperature min/max
  min = temperatures[0];
  max = temperatures[0];
  for(i = 1; i < 12; i++) {
    min = ((temperatures[i] < min) ? temperatures[i] : min);
    max = ((temperatures[i] > max) ? temperatures[i] : max);
  }
  printf("Temperature ... [%f] ... [%f] \n", min, max);

  return 0;
}
```

... et donc

```
$> gcc exercice1.c
$> ./a.out
Entrez la temperature de chaque mois.
> mois [0] : 2
> mois [1] : 3
> mois [2] : 5
> mois [3] : 6
> mois [4] : 12
> mois [5] : 15
> mois [6] : 19
> mois [7] : 23
> mois [8] : 32
> mois [9] : 28
> mois [10] : 25
> mois [11] : 21
Temperature ... [2.000000] ... [32.000000]
```

Erreur classique : "array index out of bounds"

```c
#include <stdio.h>

int main()
{
  int tab[4];
  tab[4] = 6;

  return 0;
}
```

```
$> clang erreur1.c
test.c:7:3: warning: array index 4 is past the end of the array
      (which contains 4 elements) [-Warray-bounds]
  tab[4] = 6;
  ^   ~
test.c:5:3: note: array 'tab' declared here
  int tab[4];
  ^
1 warning generated.
```

- Remarques :
  - le même résultat doit être obtenu avec le compilateur `gcc`,
  - cette erreur n'est pas toujours détectable à la compilation.

Une autre erreur classique

```c
#include <stdio.h>

int main()
{
  int tab[10] = {17, 1, 53, 8, 9, 3, 7, 1, 2, 45};
  int elt, i;

  // ...
  printf("Rechercher l'element = ");
  scanf("%d", &elt);
  // ...
  i = 0;
  while(tab[i] != elt && i < 10)
    i ++;
  if(i == 10)
    printf("L'element '%d' n'appartient pas au tableau. \n", elt);
  else
    printf("L'element '%d' appartient au tableau en case [%d]. \n", elt, i);

  return 0;
}
```

Une autre erreur classique

```c
#include <stdio.h>

int main()
{
  int tab[10] = {17, 1, 53, 8, 9, 3, 7, 1, 2, 45};
  int elt, i;

  // ...
  printf("Rechercher l'element = ");
  scanf("%d", &elt);
  // ...
  i = 0;
  while(tab[i] != elt && i < 10)
    i ++;
  if(i == 10)
    printf("L'element '%d' n'appartient pas au tableau. \n", elt);
  else
    printf("L'element '%d' appartient au tableau en case [%d]. \n", elt, i);

  return 0;
}
```

... l'ordre du test doit être inversé, sinon, cela peut poser problème à la dernière itération (`i == 10`)

Tableau à taille variable

- Le langage C ANSI interdit le déclaration d'un tableau à taille variable, c'est-à-dire, un tableau dont la taille n'est pas connue à la compilation.

```
int n;
// ... affectation/lecture de la variable n
int tab[n];
```

```
$> gcc test.c -ansi -pedantic
test.c:10:10: warning: variable length arrays are a C99 feature [-Wvla-extension]
  int tab[n];
          ^
test.c:10:7: warning: ISO C90 forbids mixing declarations and code
             [-Wdeclaration-after-statement]
  int tab[n];
      ^
2 warnings generated.
```

- Par contre, les nouveaux standards du langage C ($\geq$ C99) autorisent l'utilisation de ce type de tableaux :
  - la taille *n* doit être connue avant la création du tableau.

Exemple

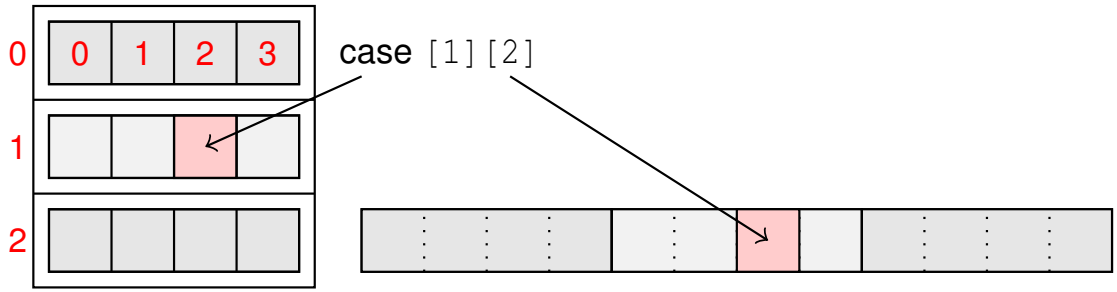
```c
#include <stdio.h>

int main()
{
  int i, n;
  printf("Entrez le nombre d'elements = ");
  scanf("%d", &n);
  // ...
  int tab[n];
  for(i = 0; i < n; i++)
    tab[i] = i;
  // ...
  for(i = 0; i < n; i++)
    printf("%d ", tab[i]);
  printf(" \n");
  return 0;
}
```

```
$> gcc tableauVLA.c
$> ./a.out
Entrez le nombre d'elements = 4
0 1 2 3
```

Les tableaux et les structures

Tableaux 2D

- Rappel de L1 : Un tableau 2D (à deux dimensions) est en fait un tableau de tableaux, c'est-à-dire, un tableau où chaque case contient un tableau.



- En langage C, la déclaration d'un tableau 2D se fait de la manière suivante.

```
type nom[ligne][colonne]
```

- L'accès à la case de la ligne d'indice 1 et de la colonne d'indice 2 se fait de la manière suivante : **tab**[1][2].
- On aurait pu utiliser un tableau 1D de taille 12, où les 4 premières cases correspondent à la première ligne, ... Mais l'accès aurait été moins direct.

Exemple de tableau 2D

```c
#include <stdio.h>

int main()
{
  float A[4][4], X[4], R[4];
  int i, j;
  // ... initialisation de A et X
  for(i = 0; i < 4; i++) {
    R[i] = 0.f;
    for(j = 0; j < 4; j++) {
      R[i] += A[i][j] * X[j];
    }
  }
  // ... manipulation du resultat R
  return 0;
}
```

- Si la première dimension peut être déterminée, elle peut être omise dans la déclaration ... mais ce n'est pas vrai pour les autres
  - `A[4][4]` peut être remplacé par `A[][4]`,
  - `A[4][]` est incorrect.
- Avant de finir ... bien sûr, on peut étendre cela à *n* dimensions,
  - pour cela, il faut ajouter `[dim]` pour chacune des dimensions supplémentaires.

Tableaux et occupation mémoire

```
int tab1[2] = {0, 1};
int tab2[2][2] = {{0, 1}, {2, 3}};
```

0x1···8		
0x1···7		
0x1···6		
0x1···5		
0x1···4		
0x1···3		
0x1···2		
0x1···1		
0x1···0		

Tableaux et occupation mémoire

```
int tab1[2] = {0, 1};
int tab2[2][2] = {{0, 1}, {2, 3}};
```

0x1···8		
0x1···7	tab2[1][1]	3
0x1···6	tab2[1][0]	2
0x1···5	tab2[0][1]	1
0x1···4	tab2[0][0]	0
0x1···3	tab2	···
0x1···2	tab1[1]	1
0x1···1	tab1[0]	0
0x1···0	tab1	···

- Remarque : on verra plus tard dans le cours ce à quoi correspondent précisément les valeurs de tab1 et tab2.

Les tableaux et les structures

Les chaînes de caractères en C

- Le langage C ne propose pas de type "chaîne de caractères" : une chaîne de caractères est en fait un tableau de caractères (`char`).

- Remarques sur les chaînes de caractères :
  - la fin de la chaîne est modélisée par un caractère spécial '\0' (backslash zero),
  - pour stocker une chaîne de caractères, il faut un tableau qui comporte au moins une case de plus que le nombre de caractères à stocker.

Les chaînes de caractères en C

- Le langage C ne propose pas de type "chaîne de caractères" : une chaîne de caractères est en fait un tableau de caractères (`char`).

- Remarques sur les chaînes de caractères :
  - la fin de la chaîne est modélisée par un caractère spécial '\0' (backslash zero),
  - pour stocker une chaîne de caractères, il faut un tableau qui comporte au moins une case de plus que le nombre de caractères à stocker.

- En langage C, son utilisation est la suivante.

```
char str[100] = "Vive la programmation"; // declaration d'une chaine de
                                         //   caracteres d'au plus 99
                                         //   caracteres
printf("--> %s \n", str);
```

Exemple de chaînes de caractères

```c
#include <stdio.h>

int main()
{
  char str1[256] = "Vive le cours de programmation C";
  char str2[256] = "... on a hate d'etre \0 a la semaine prochaine";
  char str3[256];
  // ...
  printf("Entrez une chaine de caracteres: ");
  scanf("%s", str3);                                  // /!\ pas de symbole &
  // ...
  printf("str1: '%s' \n", str1);
  printf("str2: '%s' \n", str2);
  printf("str3: '%s' \n", str3);

  return 0;
}
```

Exemple de chaînes de caractères

```c
#include <stdio.h>

int main()
{
  char str1[256] = "Vive le cours de programmation C";
  char str2[256] = "... on a hate d'etre \0 a la semaine prochaine";
  char str3[256];
  // ...
  printf("Entrez une chaine de caracteres: ");
  scanf("%s", str3);                                  // /!\ pas de symbole &
  // ...
  printf("str1: '%s' \n", str1);
  printf("str2: '%s' \n", str2);
  printf("str3: '%s' \n", str3);

  return 0;
}
```

```
$> gcc chaines.c
$> ./a.out
Entrez une chaine de caracteres: argh ...
str1: 'Vive le cours de programmation C'
str2: '... on a hate d'etre '
str3: 'argh'
```

... attention, ici, il n'y a pas de symbole & dans l'appel à la fonction `scanf`

Exercice

Écrire un programme C qui lit une chaîne de caractères au clavier, et qui compte et affiche le nombre de caractères lus.

Exercice

Écrire un programme C qui lit une chaîne de caractères au clavier, et qui compte et affiche le nombre de caractères lus.

```c
#include <stdio.h>

int main()
{
  char str[256];
  int nbCar = 0;
  // ...
  printf("Lire une chaine de caracteres: ");
  scanf("%s", str);
  // ... on n'a pas besoin de parcourir les 256 caracteres
  while(nbCar < 256 && str[nbCar] != '\0')
    nbCar ++;
  // ...
  printf("La chaine '%s' contient %d caracteres. \n", str, nbCar);

  return 0;
}
```

Exercice

Écrire un programme C qui lit une chaîne de caractères au clavier, et qui compte et affiche le nombre de caractères lus.

```c
#include <stdio.h>

int main()
{
  char str[256];
  int nbCar = 0;
  // ...
  printf("Lire une chaine de caracteres: ");
  scanf("%s", str);
  // ... on n'a pas besoin de parcourir les 256 caracteres
  while(nbCar < 256 && str[nbCar] != '\0')
    nbCar ++;
  // ...
  printf("La chaine '%s' contient %d caracteres. \n", str, nbCar);

  return 0;
}
```

```
$> gcc exercice2.c
$> ./a.out
Lire une chaine de caracteres: abcdefghijklmnopqestuvwxyz
La chaine 'abcdefghijklmnopqestuvwxyz' contient 26 caracteres.
```

Exercice

Écrire un programme C qui lit une chaîne de caractères au clavier, et qui détermine et affiche si cette phrase est un palindrome.

Exercice

Écrire un programme C qui lit une chaîne de caractères au clavier, et qui détermine et affiche si cette phrase est un palindrome.

```c
#include <stdio.h>

int main()
{
  char str[256];
  int nbCar = 0, i, j, palindrome = 1;
  // ...
  printf("Lire une chaine de caracteres: ");
  scanf("%s", str);
  // ... on n'a pas besoin de parcourir les 256 caracteres
  while(nbCar < 256 && str[nbCar] != '\0')
    nbCar ++;
  // ...
  i = 0;
  j = nbCar - 1;
  while(i < j && palindrome == 1)
    if(str[i ++] != str[j --])
      palindrome = 0;
  // ...
  if(palindrome)
    printf("La phrase 'str' est un palindrome. \n");
  else
    printf("La phrase 'str' n'est pas un palindrome. \n");

  return 0;
}
```

Les tableaux et les structures

Qu'est-ce qu'une structure ?

- En langage C, une structure permet de définir un type composé, c'est-à-dire, de regrouper un ensemble d'éléments au sein d'une même entité, n'occupant pas nécessairement des zones contigües en mémoire :
  - les champs d'une structure peuvent être de tout type, notamment des tableaux.
- La définition d'une structure suit la syntaxe suivante.

```
struct nom
{
  type1 champ1;
  type2 champ2;
  ...
};
```

Qu'est-ce qu'une structure ?

- En langage C, une structure permet de définir un type composé, c'est-à-dire, de regrouper un ensemble d'éléments au sein d'une même entité, n'occupant pas nécessairement des zones contigües en mémoire :
  - les champs d'une structure peuvent être de tout type, notamment des tableaux.
- La définition d'une structure suit la syntaxe suivante.

```
struct nom
{
  type1 champ1;
  type2 champ2;
  ...
};
```

- Ensuite, l'utilisation d'une structure se fait de la manière suivante.

```
  struct s1 var1;                    // declaration
  var1.champ1 = ...;                 // acces
  struct s2 var2 = {..., ... ,...};  // initialisation
```

- L'accès aux éléments de la structure se fait par l'opérateur '.'.
- On peut bien évidemment manipuler des tableaux de structures.

Exemple d'utilisation d'une structure

```c
struct temps
{
  short heure;
  short minute;
  short seconde;
};

int
main()
{
  struct temps horaire1, horaire2 = {11, 02, 35};
  // ...
  horaire1.heure = 12;
  horaire1.minute = 17;
  horaire1.seconde = 35;
  // ...
  printf("Horaire (1) : %2d:%2d:%2d \n", horaire1.heure,
                 horaire1.minute, horaire1.seconde);
  printf("Horaire (2) : %2d:%2d:%2d \n", horaire2.heure,
                 horaire2.minute, horaire2.seconde);
  // ...
  return 0;
}
```

Exemple d'utilisation d'une structure

```c
struct temps
{
  short heure;
  short minute;
  short seconde;
};

int
main()
{
  struct temps horaire1, horaire2 = {11, 02, 35};
  // ...
  horaire1.heure = 12;
  horaire1.minute = 17;
  horaire1.seconde = 35;
  // ...
  printf("Horaire (1) : %2d:%2d:%2d \n", horaire1.heure,
                 horaire1.minute, horaire1.seconde);
  printf("Horaire (2) : %2d:%2d:%2d \n", horaire2.heure,
                 horaire2.minute, horaire2.seconde);
  // ...
  return 0;
}
```

```
$> gcc structure.c
$> ./a.out
Horaire (1) : 12:17:35
Horaire (2) : 11: 2:35
```

Exercice

Définir une structure qui permet de représenter les coordonnées d'un nombre complexe, et écrire programme principale pour illuster l'utilisation de cette structure.

Exercice

Définir une structure qui permet de représenter les coordonnées d'un nombre complexe, et écrire programme principale pour illuster l'utilisation de cette structure.

```c
#include <stdio.h>

struct complexe
{
  float i, r;
};

int main()
{
  struct complexe pt;
  pt.i = 1.7f;
  pt.r = 1.34f;
  // ...
  printf("Complexe = %f + %f * i \n", pt.r, pt.i);
  // ...
  return 0;
}
```

Exercice

Définir une structure qui permet de représenter les coordonnées d'un nombre complexe, et écrire programme principale pour illuster l'utilisation de cette structure.

```c
#include <stdio.h>

struct complexe
{
  float i, r;
};

int main()
{
  struct complexe pt;
  pt.i = 1.7f;
  pt.r = 1.34f;
  // ...
  printf("Complexe = %f + %f * i \n", pt.r, pt.i);
  // ...
  return 0;
}
```

```
$> gcc exercice4.c
$> ./a.out
Complexe = 1.340000 + 1.700000 * i
```

Remarques sur les structures

- Une structure fonctionne comme un type simple concernant l'affectation : elle se fait alors champ par champ.

```c
struct temps { short heure, minute, seconde; };

void afficher_horaire(struct temps h)
{
  printf("Horaire : %2d:%2d:%2d \n", h.heure,
         h.minute, h.seconde);
}

int
main()
{
  struct temps horaire1 = {11, 02, 35}, horaire2 = horaire1;
  // ...
  afficher_horaire(horaire1);                // Horaire : 11: 2:35
  afficher_horaire(horaire2);                // Horaire : 11: 2:35
  // ...
  horaire2.heure = 12;
  printf("Et apres modification ... \n");
  afficher_horaire(horaire1);                // Horaire : 11: 2:35
  afficher_horaire(horaire2);                // Horaire : 12: 2:35
  return 0;
}
```

- Par contre, on ne peut pas comparer deux structures entre elles.

Structures et occupation mémoire

- L'ordre des éléments dans une structure impacte sa taille en mémoire.

```c
struct foo1
{
  char a, b, c, d;
  float d1, d2, d3;
};

int main()
{
  struct foo1 tmp1;
  // ...
  printf("'foo1' = '%d' octets. \n",
         sizeof(tmp1));
  // ...
  return 0;
}
```

```c
struct foo2
{
  char a;
  float d1;
  char b;
  float d2;
  char c;
  float d3;
  char d;
};

int main()
{
  struct foo2 tmp2;
  // ...
  printf("'foo2' = '%d' octets. \n",
         sizeof(tmp2));
  // ...
  return 0;
}
```

```
$> gcc structure-taille-memoire.c
'foo1' = '16' octets.
'foo2' = '28' octets.
```

Comment ça marche, précisément ?

- Pour des raisons de performance, les processeurs sont reliés à la mémoire vive par un bus de données de 32 ou 64 bits (granularité d'adressage = 8 bits) :
  - accès direct à une case entière de la mémoire de 32 ou 64 bits.
- En mémoire, une donnée de taille X octets doit donc être alignée sur une adresse divisible par X :
  - par exemple : l'adresse d'un entier 32 bits (= 4 octets) doit être divisible par 4.

```
struct foo1 // 16 octets
{
  char a, b, c, d;
  float d1, d2, d3;
};

struct foo2 // 28 octets
{
  char a;
  float d1;
  char b;
  float d2;
  char c;
  float d3;
  char d;
};
```

0x1···20				
0x1···1C				
0x1···18				
0x1···14				
0x1···10				
0x1···0C				
0x1···08				
0x1···04				
0x1···00				

Comment ça marche, précisément ?

- Pour des raisons de performance, les processeurs sont reliés à la mémoire vive par un bus de données de 32 ou 64 bits (granularité d'adressage = 8 bits) :
  - accès direct à une case entière de la mémoire de 32 ou 64 bits.
- En mémoire, une donnée de taille X octets doit donc être alignée sur une adresse divisible par X :
  - par exemple : l'adresse d'un entier 32 bits (= 4 octets) doit être divisible par 4.

```
struct foo1 // 16 octets
{
  char a, b, c, d;
  float d1, d2, d3;
};

struct foo2 // 28 octets
{
  char a;
  float d1;
  char b;
  float d2;
  char c;
  float d3;
  char d;
};
```

0x1···20				
0x1···1C				
0x1···18				
0x1···14				
0x1···10				
0x1···0C				
0x1···08				
0x1···04				
0x1···00	a			

Comment ça marche, précisément ?

- Pour des raisons de performance, les processeurs sont reliés à la mémoire vive par un bus de données de 32 ou 64 bits (granularité d'adressage = 8 bits) :
  - accès direct à une case entière de la mémoire de 32 ou 64 bits.
- En mémoire, une donnée de taille X octets doit donc être alignée sur une adresse divisible par X :
  - par exemple : l'adresse d'un entier 32 bits (= 4 octets) doit être divisible par 4.

```
struct foo1 // 16 octets
{
  char a, b, c, d;
  float d1, d2, d3;
};

struct foo2 // 28 octets
{
  char a;
  float d1;
  char b;
  float d2;
  char c;
  float d3;
  char d;
};
```

0x1···20				
0x1···1C				
0x1···18				
0x1···14				
0x1···10				
0x1···0C				
0x1···08				
0x1···04				
0x1···00	a	b		

Comment ça marche, précisément ?

- Pour des raisons de performance, les processeurs sont reliés à la mémoire vive par un bus de données de 32 ou 64 bits (granularité d'adressage = 8 bits) :
  - accès direct à une case entière de la mémoire de 32 ou 64 bits.
- En mémoire, une donnée de taille X octets doit donc être alignée sur une adresse divisible par X :
  - par exemple : l'adresse d'un entier 32 bits (= 4 octets) doit être divisible par 4.

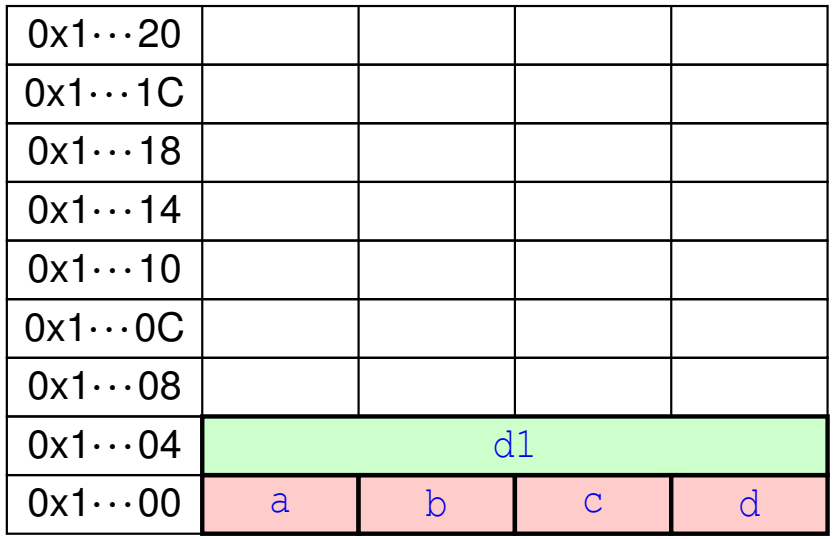
```
struct foo1 // 16 octets
{
  char a, b, c, d;
  float d1, d2, d3;
};

struct foo2 // 28 octets
{
  char a;
  float d1;
  char b;
  float d2;
  char c;
  float d3;
  char d;
};
```

0x1···20				
0x1···1C				
0x1···18				
0x1···14				
0x1···10				
0x1···0C				
0x1···08				
0x1···04				
0x1···00	a	b	c	d

Comment ça marche, précisément ?

- Pour des raisons de performance, les processeurs sont reliés à la mémoire vive par un bus de données de 32 ou 64 bits (granularité d'adressage = 8 bits) :
  - accès direct à une case entière de la mémoire de 32 ou 64 bits.
- En mémoire, une donnée de taille X octets doit donc être alignée sur une adresse divisible par X :
  - par exemple : l'adresse d'un entier 32 bits (= 4 octets) doit être divisible par 4.

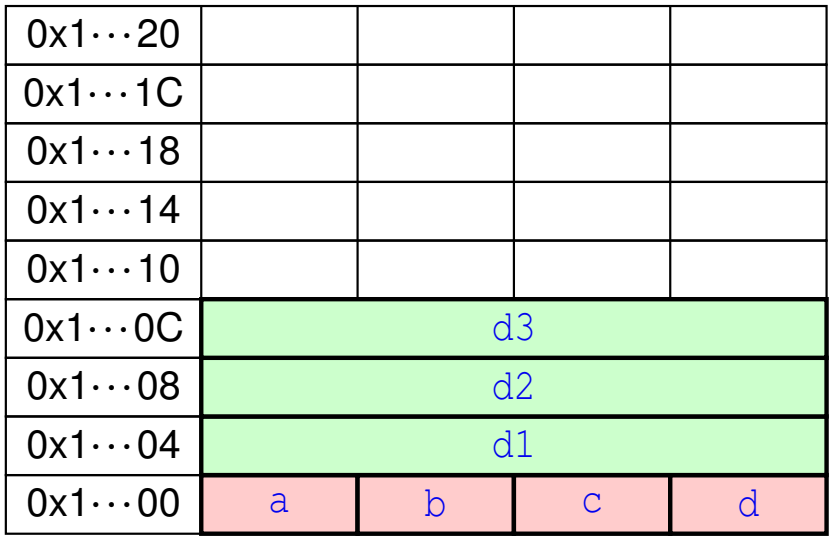
```
struct foo1 // 16 octets
{
  char a, b, c, d;
  float d1, d2, d3;
};

struct foo2 // 28 octets
{
  char a;
  float d1;
  char b;
  float d2;
  char c;
  float d3;
  char d;
};
```



Comment ça marche, précisément ?

- Pour des raisons de performance, les processeurs sont reliés à la mémoire vive par un bus de données de 32 ou 64 bits (granularité d'adressage = 8 bits) :
  - accès direct à une case entière de la mémoire de 32 ou 64 bits.
- En mémoire, une donnée de taille X octets doit donc être alignée sur une adresse divisible par X :
  - par exemple : l'adresse d'un entier 32 bits (= 4 octets) doit être divisible par 4.

```
struct foo1 // 16 octets
{
  char a, b, c, d;
  float d1, d2, d3;
};

struct foo2 // 28 octets
{
  char a;
  float d1;
  char b;
  float d2;
  char c;
  float d3;
  char d;
};
```



Comment ça marche, précisément ?

- Pour des raisons de performance, les processeurs sont reliés à la mémoire vive par un bus de données de 32 ou 64 bits (granularité d'adressage = 8 bits) :
  - accès direct à une case entière de la mémoire de 32 ou 64 bits.
- En mémoire, une donnée de taille X octets doit donc être alignée sur une adresse divisible par X :
  - par exemple : l'adresse d'un entier 32 bits (= 4 octets) doit être divisible par 4.

```
struct foo1 // 16 octets
{
  char a, b, c, d;
  float d1, d2, d3;
};

struct foo2 // 28 octets
{
  char a;
  float d1;
  char b;
  float d2;
  char c;
  float d3;
  char d;
};
```

0x1···20				
0x1···1C				
0x1···18				
0x1···14				
0x1···10				
0x1···0C				
0x1···08				
0x1···04				
0x1···00				

Comment ça marche, précisément ?

- Pour des raisons de performance, les processeurs sont reliés à la mémoire vive par un bus de données de 32 ou 64 bits (granularité d'adressage = 8 bits) :
  - accès direct à une case entière de la mémoire de 32 ou 64 bits.
- En mémoire, une donnée de taille X octets doit donc être alignée sur une adresse divisible par X :
  - par exemple : l'adresse d'un entier 32 bits (= 4 octets) doit être divisible par 4.

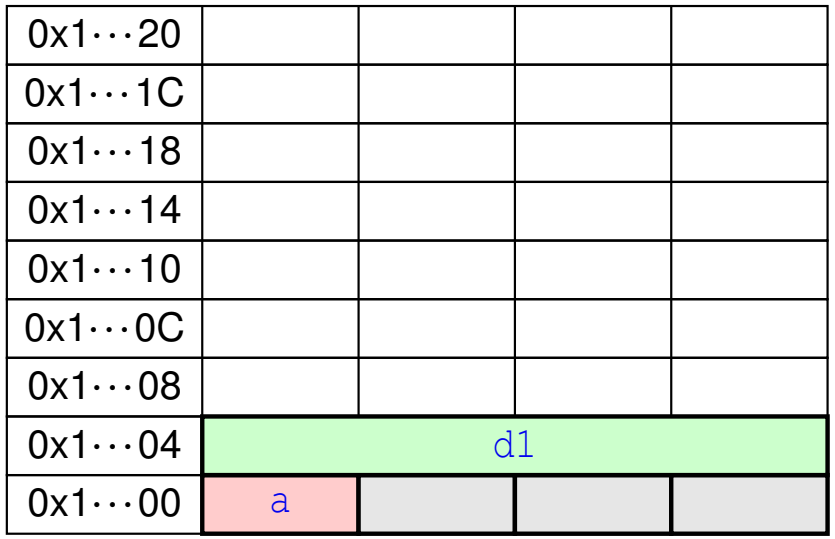
```
struct foo1 // 16 octets
{
  char a, b, c, d;
  float d1, d2, d3;
};

struct foo2 // 28 octets
{
  char a;
  float d1;
  char b;
  float d2;
  char c;
  float d3;
  char d;
};
```

0x1···20				
0x1···1C				
0x1···18				
0x1···14				
0x1···10				
0x1···0C				
0x1···08				
0x1···04				
0x1···00	a			

Comment ça marche, précisément ?

- Pour des raisons de performance, les processeurs sont reliés à la mémoire vive par un bus de données de 32 ou 64 bits (granularité d'adressage = 8 bits) :
  - accès direct à une case entière de la mémoire de 32 ou 64 bits.
- En mémoire, une donnée de taille X octets doit donc être alignée sur une adresse divisible par X :
  - par exemple : l'adresse d'un entier 32 bits (= 4 octets) doit être divisible par 4.

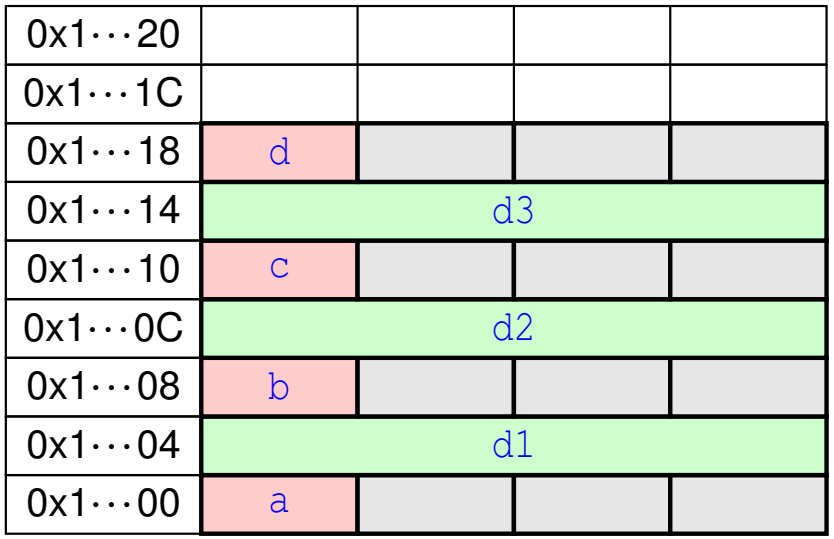
```
struct foo1 // 16 octets
{
  char a, b, c, d;
  float d1, d2, d3;
};

struct foo2 // 28 octets
{
  char a;
  float d1;
  char b;
  float d2;
  char c;
  float d3;
  char d;
};
```



Comment ça marche, précisément ?

- Pour des raisons de performance, les processeurs sont reliés à la mémoire vive par un bus de données de 32 ou 64 bits (granularité d'adressage = 8 bits) :
  - accès direct à une case entière de la mémoire de 32 ou 64 bits.
- En mémoire, une donnée de taille X octets doit donc être alignée sur une adresse divisible par X :
  - par exemple : l'adresse d'un entier 32 bits (= 4 octets) doit être divisible par 4.

```
struct foo1 // 16 octets
{
  char a, b, c, d;
  float d1, d2, d3;
};

struct foo2 // 28 octets
{
  char a;
  float d1;
  char b;
  float d2;
  char c;
  float d3;
  char d;
};
```



En fait, votre compilateur vous donne ces informations

```
$> gcc structure-taille-memoire.c -Wpadded
structure-taille-memoire.c:12:9: warning: padding struct 'struct foo2' with 3
     bytes to align 'd1' [-Wpadded]
  float d1;
        ^
structure-taille-memoire.c:14:9: warning: padding struct 'struct foo2' with 3
     bytes to align 'd2' [-Wpadded]
  float d2;
        ^
structure-taille-memoire.c:16:9: warning: padding struct 'struct foo2' with 3
     bytes to align 'd3' [-Wpadded]
  float d3;
        ^
structure-taille-memoire.c:9:8: warning: padding size of 'struct foo2' with 3
     bytes to alignment boundary [-Wpadded]
struct foo2
```

Que va-t-on voir dans ce cours ?

(1) Les éléments de base du C

(2) Les structures de contrôle

(3) Les tableaux et les structures

(4) Les pointeurs et l'allocation dynamique

(5) Les fonctions

(6) Les autres types dérivés

(7) Les fichiers

(8) Quelques éléments avancés

(9) Complément 1 : Les listes chaînées

(10) Complément 2 : Les piles et les files

Les pointeurs et l'allocation dynamique

Les pointeurs et l'allocation dynamique

Les pointeurs et l'allocation dynamique

Qu'est-ce-qu'un pointeur ?

- Définition : un pointeur est une variable dont la valeur est une adresse mémoire.
- Pourquoi les pointeurs ont-il été inventés ?
  - tableaux de taille variable et allocation dynamique,
  - modification d'une variable en paramètre d'une fonction,
  - référence à une structure de données (taille importante),
  - simplicité de l'arithmétique des pointeurs.
- La déclaration d'un pointeur se fait en utilisant le symbole `*`.
- On peut affecter une adresse (celle d'une variable, par exemple) à un pointeur.
- Ensuite, l'accès à la valeur pointée se fait via l'opérateur de déréférencement `*`, et l'accès à l'adresse d'une variable se fait via l'opérateur d'indirection `&`.

```
int a = 17;
int *ptr = &a;    // declaration d'un pointeur sur un entier
                  // dont la valeur est l'adresse de la variable 'a'

printf("--> %d %d \n", a, *ptr); // --> 17 17
*ptr = 26;
printf("--> %d %d \n", a, *ptr); // --> 26 26
```

Que se passe-t-il réellement ?

```
int a = 17;
int *ptr = &a;

printf("--> %d %d \n", a, *ptr);
*ptr = 26;
printf("--> %d %d \n", a, *ptr);
```

0x1···8		
0x1···7		
0x1···6		
0x1···5		
0x1···4		
0x1···3		
0x1···2		
0x1···1		
0x1···0		

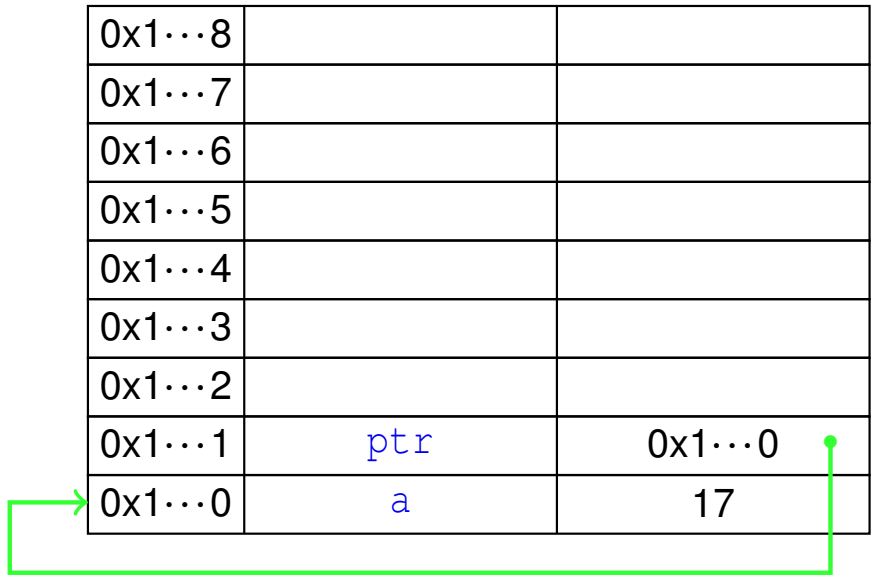
Que se passe-t-il réellement ?

```
int a = 17;
int *ptr = &a;

printf("--> %d %d \n", a, *ptr);
*ptr = 26;
printf("--> %d %d \n", a, *ptr);
```

0x1···8		
0x1···7		
0x1···6		
0x1···5		
0x1···4		
0x1···3		
0x1···2		
0x1···1		
0x1···0	a	17

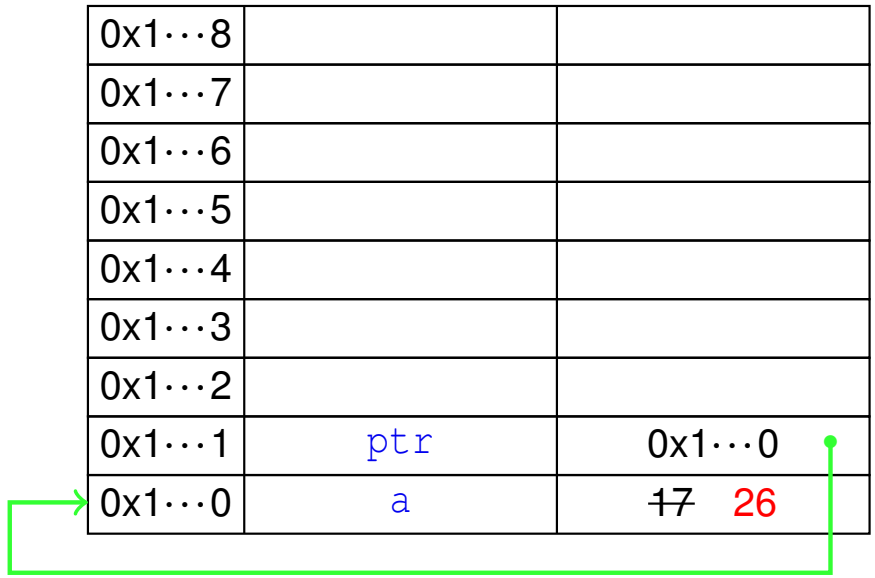
Que se passe-t-il réellement ?

```
int a = 17;
int *ptr = &a;

printf("--> %d %d \n", a, *ptr);
*ptr = 26;
printf("--> %d %d \n", a, *ptr);
```



Que se passe-t-il réellement ?

```
int a = 17;
int *ptr = &a;

printf("--> %d %d \n", a, *ptr);
*ptr = 26;
printf("--> %d %d \n", a, *ptr);
```



Lien entre tableau et pointeur

- Les tableaux 1D sont en réalité repérés par des pointeurs.

```
int tab[17]
```

- Dans ce cas
  - tab n'est pas une variable,
  - tab n'est pas de type tableau,
  - tab ne désigne pas tout le tableau.
- En fait, tab est un pointeur sur entier, sur le premier élément du tableau.
- Sa valeur est l'adresse du premier élément dans le tableau.
- Et l'accès à l'élément d'indice *i* de tab peut se faire en utilisant ce pointeur.

```
int tab[4] = {0, 1, 2, 3};
int *ptr = &(tab[0]);
// ...
printf("--> tab[0] = %d \n", *ptr);       // --> tab[0] = 0
printf("--> tab[2] = %d \n", *(ptr+2));   // --> tab[2] = 2
// ...
*(ptr+1) = 17;
// ...
printf("--> tab[1] = %d \n", tab[1]);     // --> tab[1] = 17
```

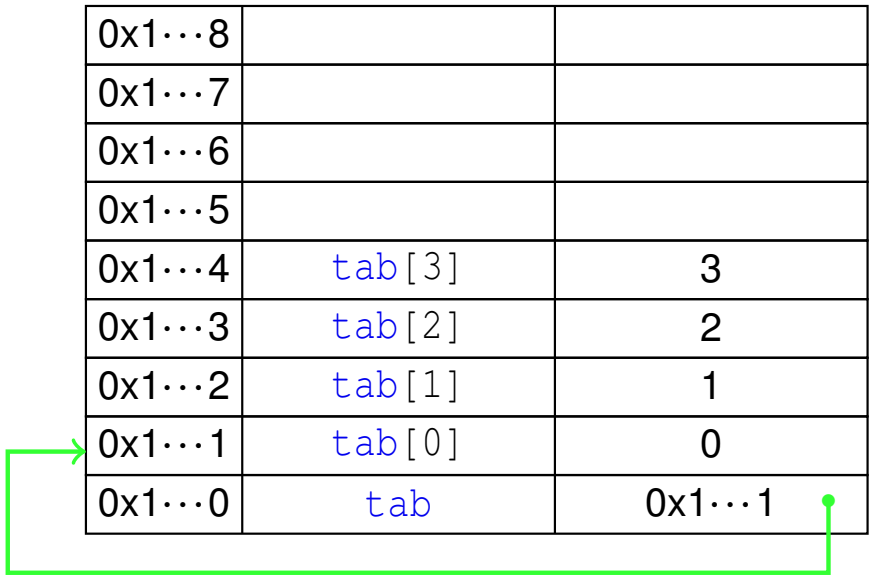
Que se passe-t-il maintenant sur les tableaux ?

```c
int tab[4] = {0, 1, 2, 3};
int *ptr = &(tab[0]);
// ...
printf("--> tab[0] = %d \n", *ptr);
printf("--> tab[2] = %d \n", *(ptr+2));
// ...
*(ptr+1) = 17;
// ...
printf("--> tab[1] = %d \n", tab[1]);
```

0x1···8		
0x1···7		
0x1···6		
0x1···5		
0x1···4		
0x1···3		
0x1···2		
0x1···1		
0x1···0		

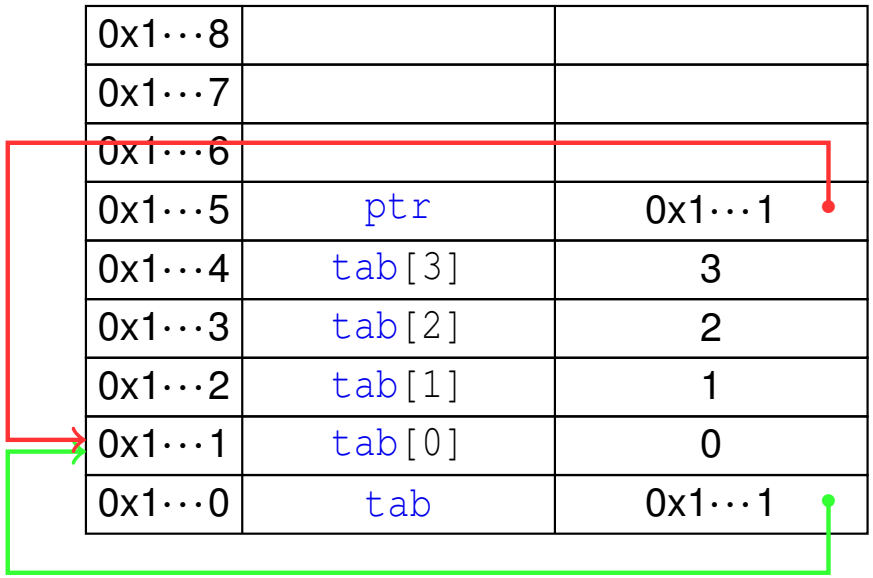
Que se passe-t-il maintenant sur les tableaux ?

```c
int tab[4] = {0, 1, 2, 3};
int *ptr = &(tab[0]);
// ...
printf("--> tab[0] = %d \n", *ptr);
printf("--> tab[2] = %d \n", *(ptr+2));
// ...
*(ptr+1) = 17;
// ...
printf("--> tab[1] = %d \n", tab[1]);
```



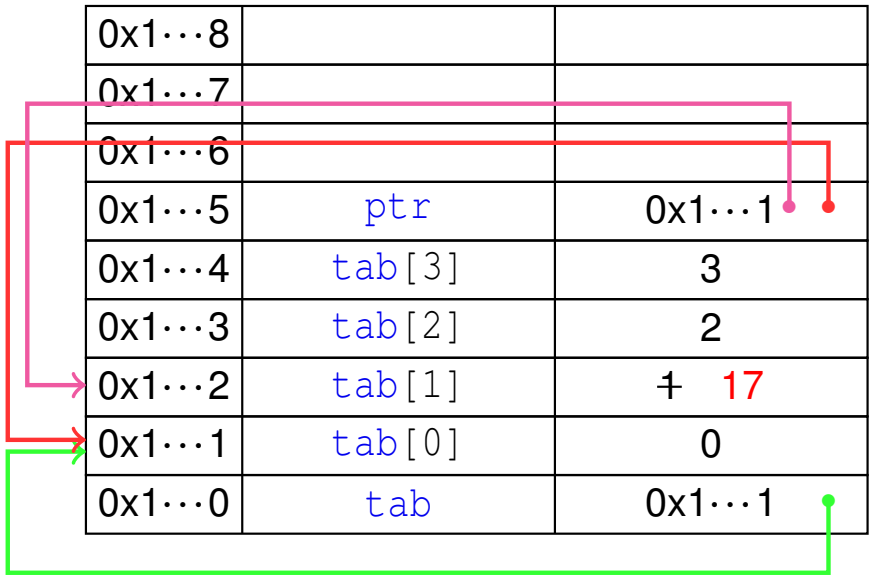
Que se passe-t-il maintenant sur les tableaux ?

```
int tab[4] = {0, 1, 2, 3};
int *ptr = &(tab[0]);
// ...
printf("--> tab[0] = %d \n", *ptr);
printf("--> tab[2] = %d \n", *(ptr+2));
// ...
*(ptr+1) = 17;
// ...
printf("--> tab[1] = %d \n", tab[1]);
```



Que se passe-t-il maintenant sur les tableaux ?

```
int tab[4] = {0, 1, 2, 3};
int *ptr = &(tab[0]);
// ...
printf("--> tab[0] = %d \n", *ptr);
printf("--> tab[2] = %d \n", *(ptr+2));
// ...
*(ptr+1) = 17;
// ...
printf("--> tab[1] = %d \n", tab[1]);
```



Arithmétique des pointeurs

- On peut appliquer des opérations arithmétiques sur les pointeurs.
- Mais ces opérations n'ont de sens qu'avec des pointeurs repérant des éléments d'un même tableau.

- Addition/soustraction : pointeur ± entier → pointeur

```
int tab[4], *p, *q, i, j, k;
// ...
p = &tab[i];          // -> adresse du i-eme element de tab
q = p + j;            // -> pointeur vers le (i+j)-eme element de tab
q = p - j;            // -> pointeur vers le (i-j)-eme element de tab
```

- Différence : pointeur – pointeur → entier

```
k = q - p;            // -> nb d'elements entier entre p et q = j
```

- Comparaison : pointeur vs. pointeur → vrai/faux

Exercice

Écrire un programme C qui lit et stocke les températures de chaque mois d'une année à Perpignan, et qui calcule et affiche la plus grande et plus petite températures, en utilisant les pointeurs.

Exercice

Écrire un programme C qui lit et stocke les températures de chaque mois d'une année à Perpignan, et qui calcule et affiche la plus grande et plus petite températures, en utilisant les pointeurs.

```c
#include <stdio.h>

int main()
{
  float temperatures[12], min, max, *ptr = &temperatures[0];
  int i;

  // ... lecture des donnees
  printf("Entrez la temperature de chaque mois. \n");
  for(i = 0; i < 12; i++) {
    printf("> mois [%d] : ", i);
    scanf("%f", (ptr+i));  // &temperatures[i] <=> &(*(ptr + i))
  }                        //                   <=> (ptr + i)

  // ... calcule et affichage de la temperature min/max
  min = *ptr;  // temperatures[0]
  max = *ptr;  // temperatures[0]
  for(i = 1; i < 12; i++) {
    min = ((*(ptr+i) /* temperatures[i] */ < min) ? *(ptr+i) : min);
    max = ((*(ptr+i) /* temperatures[i] */ > max) ? *(ptr+i) : max);
  }
  printf("Temperature ... [%f] ... [%f] \n", min, max);
  return 0;
}
```

Et pour les tableaux multidimensionnels ?

- Les tableaux 2D, et plus généralement les tableaux multidimensionnels, peuvent également être repérés par un pointeur sur le premier élément.

```
int tab[2][2] = {{0, 1}, {2, 3}};
int *ptr = &(tab[0][0]);
int **ptr_ptr = &(tab[0]);                    // /!\ Argh !
// ...
printf("--> tab[0][0] = %d \n", *ptr);       // --> tab[0][0] = 0
printf("--> tab[0][1] = %d \n", *(ptr+1));   // --> tab[0][1] = 1
// ...
printf("--> tab[1][0] = %d \n", *(ptr+2));   // --> tab[1][0] = 2
```

- Dans ce cas, le parcours du tableau est plus compliqué, car on n'a plus accès à l'opérateur [i][j]

- Attention, un tableau 2D ne peut pas être repéré par un pointeur de pointeurs.

```
$> clang test.c
test.c:5:10: warning: incompatible pointer types initializing 'int **' with an
    expression of type 'int (*)[2]' [-Wincompatible-pointer-types]
  int ** ptr_ptr = &(tab[0]);
         ^         ~~~~~~~~~
```

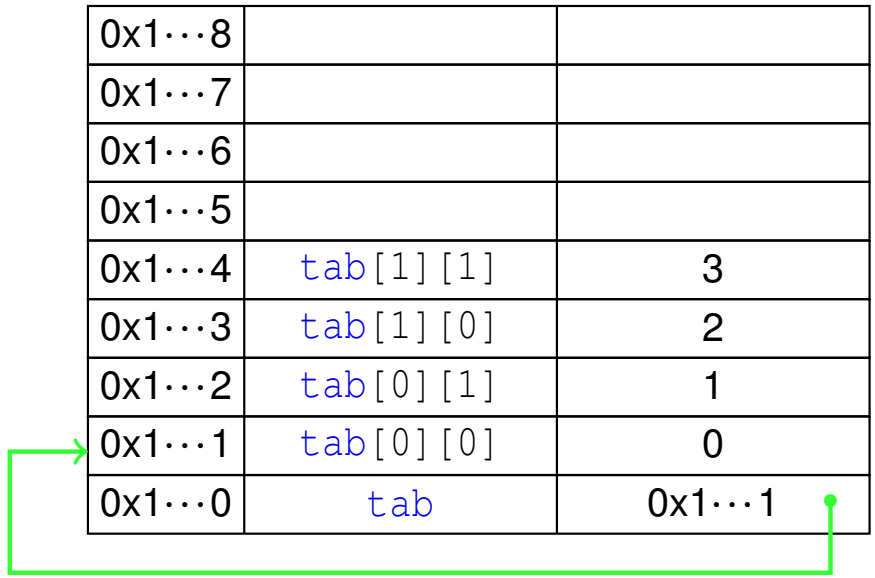
Et dans ce cas, que se passe-t-il ?

```
int tab[2][2] = {{0, 1}, {2, 3}};
int *ptr = &(tab[0][0]);

*(ptr + 2) = 17;
```

0x1···8		
0x1···7		
0x1···6		
0x1···5		
0x1···4		
0x1···3		
0x1···2		
0x1···1		
0x1···0		

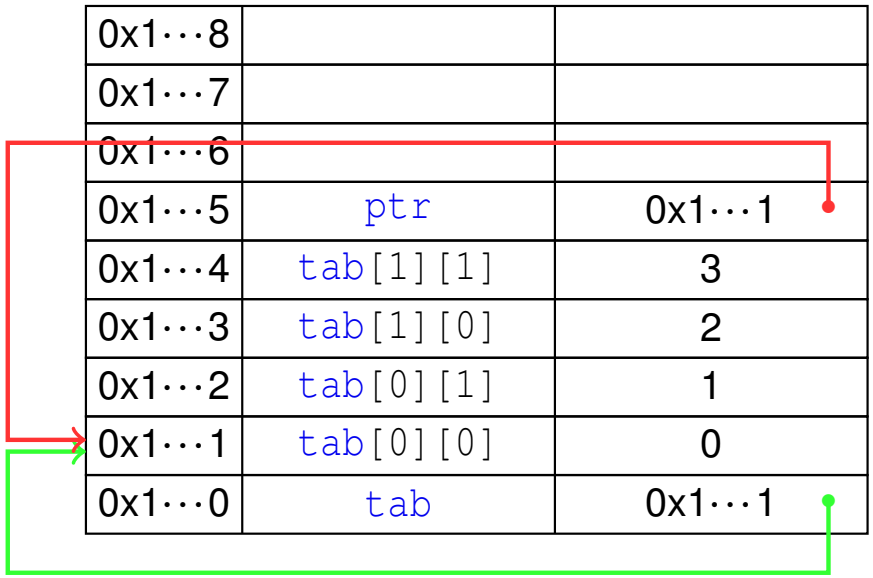
Et dans ce cas, que se passe-t-il ?

```
int tab[2][2] = {{0, 1}, {2, 3}};
int *ptr = &(tab[0][0]);

*(ptr + 2) = 17;
```



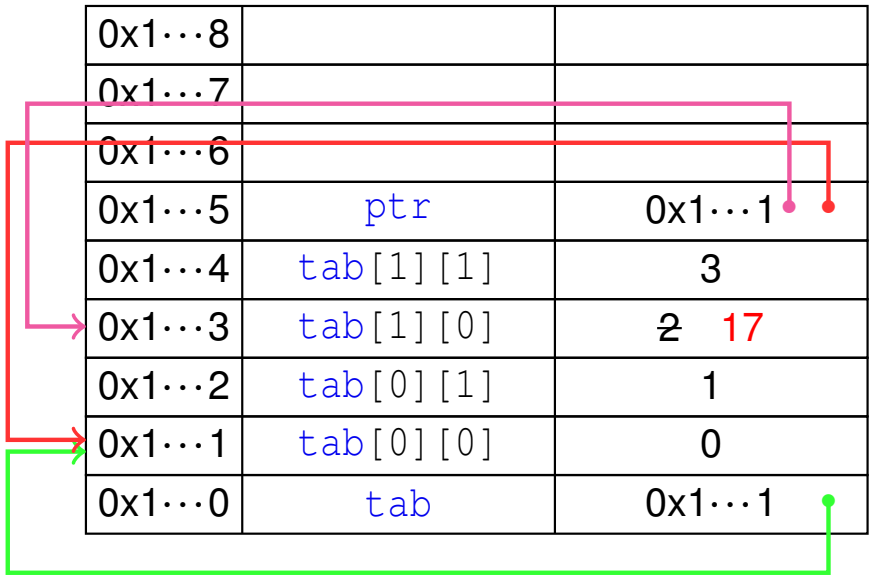
Et dans ce cas, que se passe-t-il ?

```
int tab[2][2] = {{0, 1}, {2, 3}};
int *ptr = &(tab[0][0]);

*(ptr + 2) = 17;
```



Et dans ce cas, que se passe-t-il ?

```
int tab[2][2] = {{0, 1}, {2, 3}};
int *ptr = &(tab[0][0]);

*(ptr + 2) = 17;
```



Les pointeurs et l'allocation dynamique

Tout d'abord, l'organisation de la mémoire

- La mémoire d'un ordinateur est une succession d'octets (8 bits), organisés les uns à la suite des autres et directement accessibles par une adresse.
- En langage C, la mémoire pour stocker des variables est organisée en deux catégories :
  - la pile (stack),
  - le tas (heap).

Tout d'abord, l'organisation de la mémoire

- La mémoire d'un ordinateur est une succession d'octets (8 bits), organisés les uns à la suite des autres et directement accessibles par une adresse.
- En langage C, la mémoire pour stocker des variables est organisée en deux catégories :
  - la pile (stack),
  - le tas (heap).

- La pile est un espace mémoire :
  - où sont stockés les variables locales des fonctions et les paramètres d'appel,
  - réservé au stockage des variables allouées et désallouées automatiquement.

- Le tas est un espace mémoire :
  - utilisé lors de l'allocation dynamique de mémoire durant l'exécution d'un programme,
  - où la mémoire allouée doit être désallouée explicitement.

Qu'est-ce que l'allocation dynamique ?

- Comme n'importe quelle variable, avant de manipuler un pointeur, il faut l'initialiser. Sinon, sa valeur est égale à `NULL`.
- On peut initialiser un pointeur en lui affectant une valeur, notamment l'adresse d'un espace mémoire existant : il *pointe* alors vers cet espace mémoire.

```
int a = 17;
int *ptr = &a; // OK car un espace memoire existe pour la variable a
```

Qu'est-ce que l'allocation dynamique ?

- Comme n'importe quelle variable, avant de manipuler un pointeur, il faut l'initialiser. Sinon, sa valeur est égale à `NULL`.
- On peut initialiser un pointeur en lui affectant une valeur, notamment l'adresse d'un espace mémoire existant : il *pointe* alors vers cet espace mémoire.

```
int a = 17;
int *ptr = &a; // OK car un espace memoire existe pour la variable a
```

- On peut également directement modifier la valeur pointée par un pointeur, c'est-à-dire, affecter une valeur à `*ptr`.

```
int a = 17;
int *ptr = NULL;
// ...
*ptr = a;       // Argh, aucun espace existe pour stocker *ptr
```

- Dans ce cas, il faut tout d'abord réserver pour `*ptr` un espace mémoire de taille suffisante : l'adresse de cet espace mémoire sera la valeur de `ptr`.
- Ce processus est appelé allocation dynamique.

Allocation et désallocation

- L'allocation dynamique se fait en utilisant la fonction C `malloc`.

```
void * malloc(size_t taille_octets)
```

- La fonction `malloc` renvoie un pointeur générique (`void*`), qui devra être convertit explicitement (*casté*) vers le type approprié.
- Si la mémoire n'est pas suffisante, la fonction `malloc` renvoie `NULL`.

Allocation et désallocation

- L'allocation dynamique se fait en utilisant la fonction C `malloc`.

```
void * malloc(size_t taille_octets)
```

- La fonction `malloc` renvoie un pointeur générique (`void*`), qui devra être convertit explicitement (*casté*) vers le type approprié.
- Si la mémoire n'est pas suffisante, la fonction `malloc` renvoie `NULL`.
- La mémoire allouée dynamiquement avec `malloc` se situe sur le tas.
- Toute zone mémoire allouée avec `malloc` devra ensuite être désallouée avec la fonction C `free`.

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

Allocation et désallocation

- L'allocation dynamique se fait en utilisant la fonction C `malloc`.

```
void * malloc(size_t taille_octets)
```

- La fonction `malloc` renvoie un pointeur générique (`void*`), qui devra être convertit explicitement (*casté*) vers le type approprié.
- Si la mémoire n'est pas suffisante, la fonction `malloc` renvoie `NULL`.
- La mémoire allouée dynamiquement avec `malloc` se situe sur le tas.
- Toute zone mémoire allouée avec `malloc` devra ensuite être désallouée avec la fonction C `free`.

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

```
int a = 17;
int *ptr = NULL;
ptr = (int*)malloc(sizeof(int));
*ptr = a;       // OK car un espace memoire a ete alloue pour *ptr
free(ptr);
```

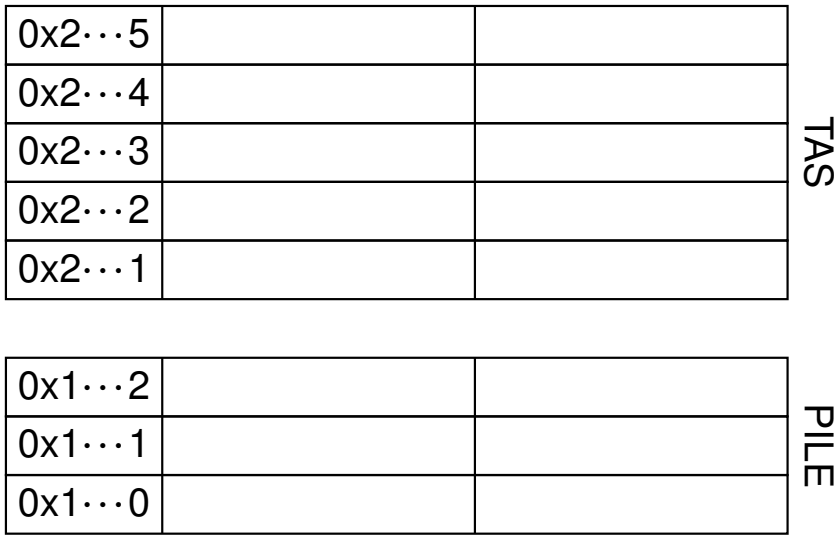
Exemple d'allocation dynamique pour un tableau 1D

```c
#include <stdlib.h>   // pour malloc/free

int main()
{
  int * tab = NULL;
  int i, n;
  // ...
  printf("Entrez la taille du tableau= ");
  scanf("%d", &n);
  // ...
  tab = (int*)malloc(n*sizeof(int));
  if(tab != NULL) {
    // ...
    for(i = 0; i < n; i++) {
      scanf("%d", &tab[i]);
      printf("tab[%d] = %d \n", i, *(tab + i));
    }
    // ...
    free(tab);
  }
  return 0;
}
```

- Remarques :
  - il faut tester si le retour de `malloc` n'est pas `NULL`,
  - la fonction `sizeof(X)` retourne la taille en octet d'un élément `X` ou de type `X`.

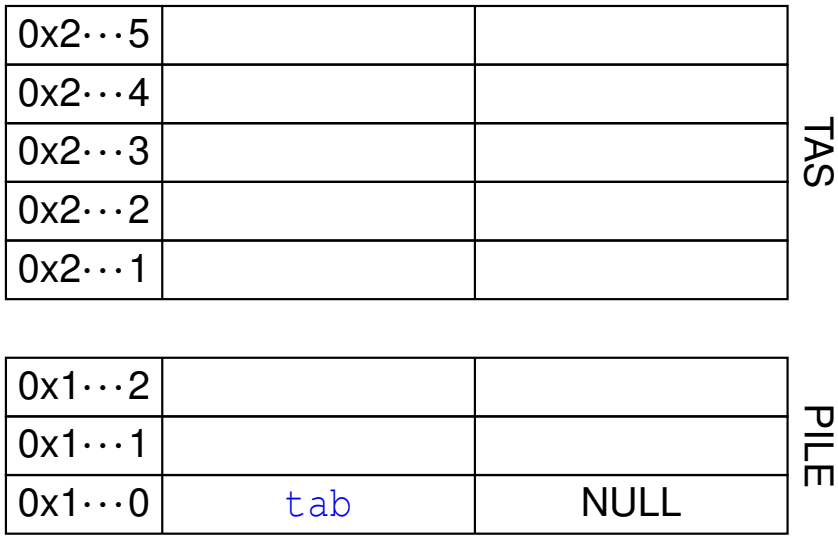
Que se passe-t-il dans ce cas ?

```c
int main()
{
  int * tab = NULL;
  int i, n;
  // ...
  tab = (int*)malloc(n*sizeof(int));
  if(tab != NULL) {
    // ...
    free(tab);
  }
  return 0;
}
```



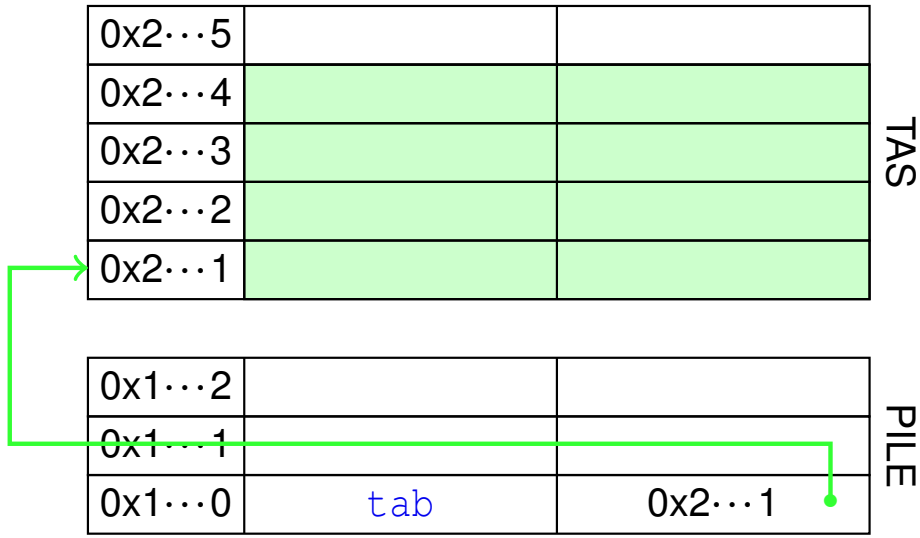
Que se passe-t-il dans ce cas ?

```c
int main()
{
  int * tab = NULL;
  int i, n;
  // ...
  tab = (int*)malloc(n*sizeof(int));
  if(tab != NULL) {
    // ...
    free(tab);
  }
  return 0;
}
```



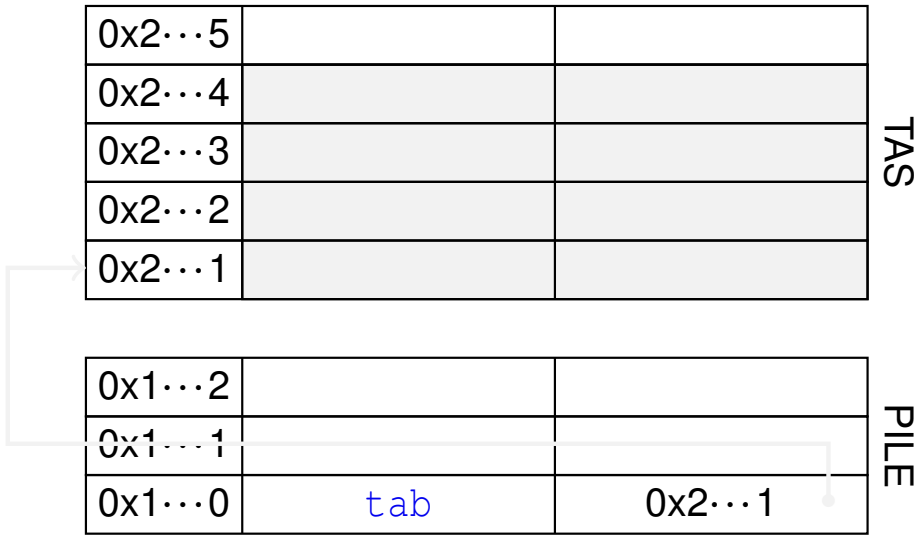
Que se passe-t-il dans ce cas ?

```c
int main()
{
  int * tab = NULL;
  int i, n;
  // ...
  tab = (int*)malloc(n*sizeof(int));
  if(tab != NULL) {
    // ...
    free(tab);
  }
  return 0;
}
```



Que se passe-t-il dans ce cas ?

```c
int main()
{
  int * tab = NULL;
  int i, n;
  // ...
  tab = (int*)malloc(n*sizeof(int));
  if(tab != NULL) {
    // ...
    free(tab);
  }
  return 0;
}
```



Exemple d'allocation dynamique pour un tableau 2D

```c
int main()
{
  int ** matrice = NULL;
  int i, j, n;
  // ...
  printf("Entrez la taille de la matrice tableau= ");
  scanf("%d", &n);
  // ...
  matrice = (int**)malloc(n*sizeof(int*));
  if(matrice != NULL) {
    i = 0;
    do{
      matrice[i] = (int*)malloc(n*sizeof(int));
      i ++;
    } while (i < n && matrice[i-1] != NULL);
    // ...
    if(i == n) {
      // ... manipulation de 'matrice'
    }
    // ...
    for(j = 0; j < i; j++)
      free(matrice[j]);
    free(matrice);
  }
  return 0;
}
```

- Remarques :
  - les lignes du tableau peuvent ne pas être stockées de manière contigüe en mémoire,
  - la libération mémoire se fait dans le sens inverse de l'allocation.

Et ici, que se passe-t-il dans ce cas ?

```c
int main()
{
  int ** matrice = NULL;
  int i, j, n;
  // ...
  matrice = (int**)malloc(n*sizeof(int*));
  if(matrice != NULL) {
    i = 0;
    do{
      matrice[i] = (int*)malloc(n*sizeof(int));
      i ++;
    } while (i < n && matrice[i-1] != NULL);
    // ...
    for(j = 0; j < i; j++)
      free(matrice[j]);
    free(matrice);
  }
  return 0;
}
```

0x2···9		
0x2···8		
0x2···7		
0x2···6		
0x2···5		
0x2···4		
0x2···3		
0x2···2		
0x2···1		

0x1···2		
0x1···1		
0x1···0		

Et ici, que se passe-t-il dans ce cas ?

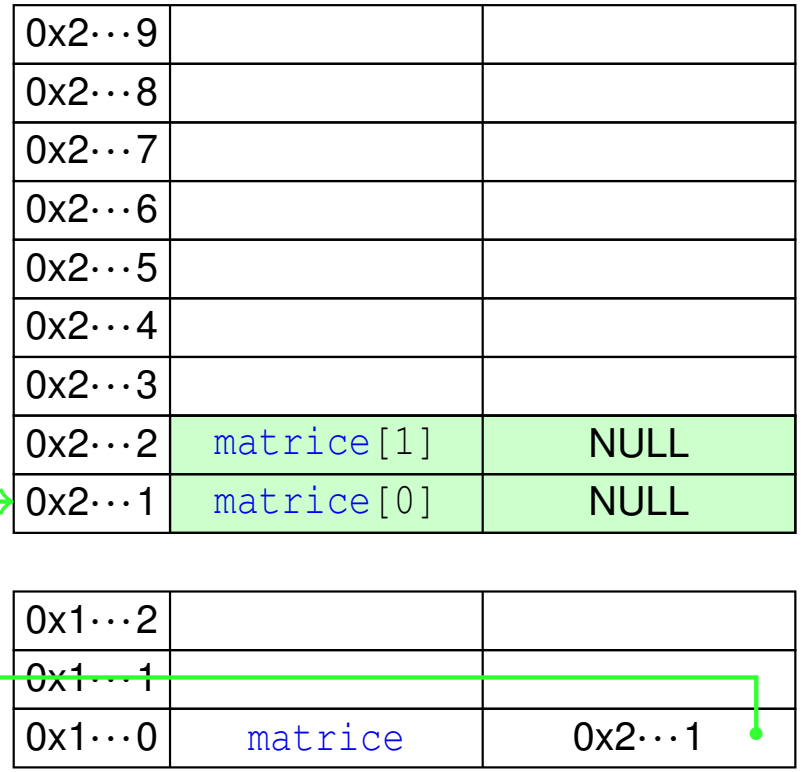
```c
int main()
{
  int ** matrice = NULL;
  int i, j, n;
  // ...
  matrice = (int**)malloc(n*sizeof(int*));
  if(matrice != NULL) {
    i = 0;
    do{
      matrice[i] = (int*)malloc(n*sizeof(int));
      i ++;
    } while (i < n && matrice[i-1] != NULL);
    // ...
    for(j = 0; j < i; j++)
      free(matrice[j]);
    free(matrice);
  }
  return 0;
}
```

0x2···9		
0x2···8		
0x2···7		
0x2···6		
0x2···5		
0x2···4		
0x2···3		
0x2···2		
0x2···1		

0x1···2		
0x1···1		
0x1···0	matrice	NULL

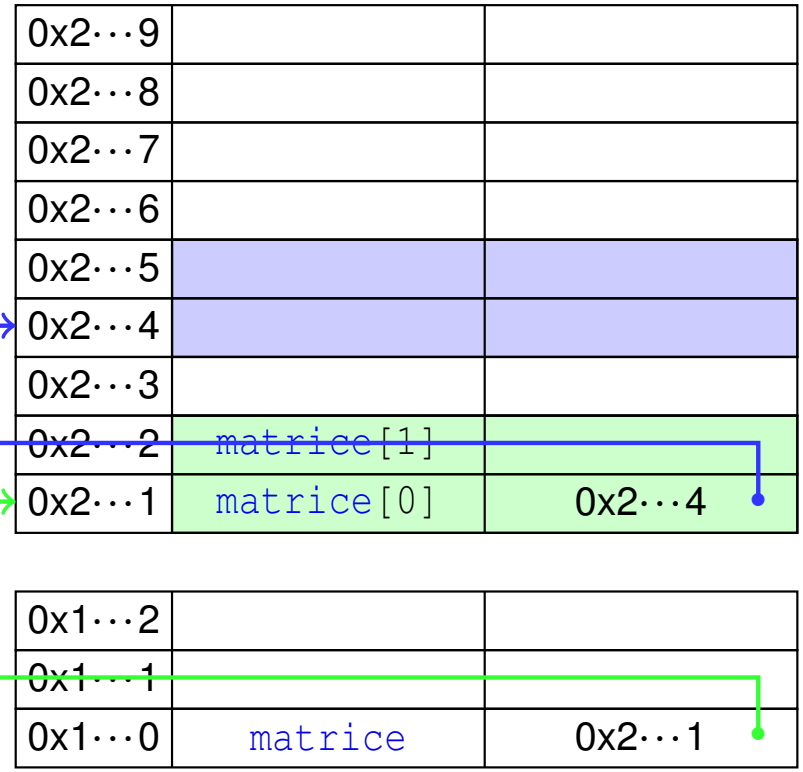
Et ici, que se passe-t-il dans ce cas ?

```c
int main()
{
  int ** matrice = NULL;
  int i, j, n;
  // ...
  matrice = (int**)malloc(n*sizeof(int*));
  if(matrice != NULL) {
    i = 0;
    do{
      matrice[i] = (int*)malloc(n*sizeof(int));
      i ++;
    } while (i < n && matrice[i-1] != NULL);
    // ...
    for(j = 0; j < i; j++)
      free(matrice[j]);
    free(matrice);
  }
  return 0;
}
```



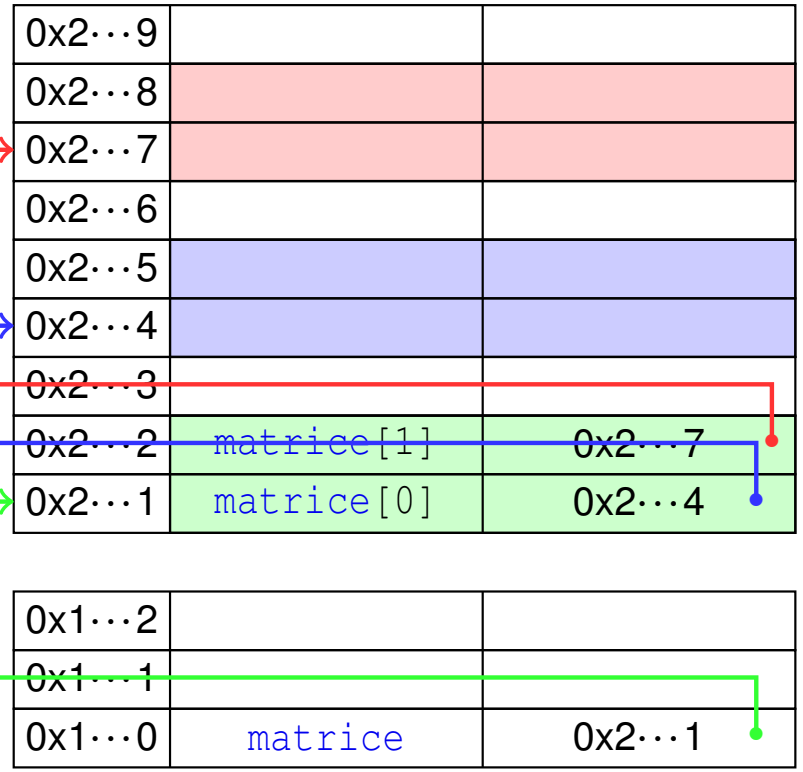
Et ici, que se passe-t-il dans ce cas ?

```c
int main()
{
  int ** matrice = NULL;
  int i, j, n;
  // ...
  matrice = (int**)malloc(n*sizeof(int*));
  if(matrice != NULL) {
    i = 0;
    do{
      matrice[i] = (int*)malloc(n*sizeof(int));
      i ++;
    } while (i < n && matrice[i-1] != NULL);
    // ...
    for(j = 0; j < i; j++)
      free(matrice[j]);
    free(matrice);
  }
  return 0;
}
```



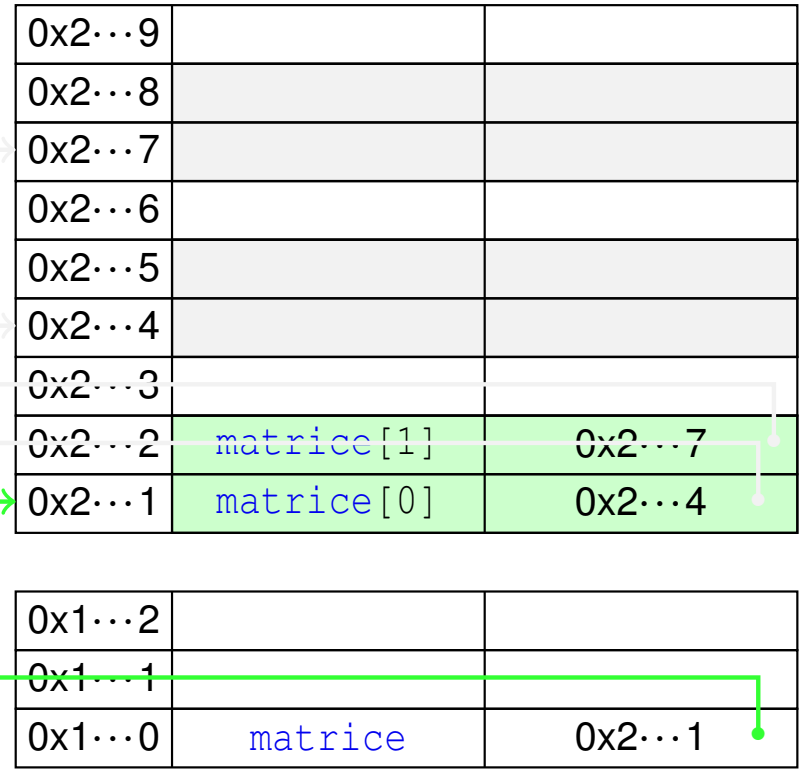
Et ici, que se passe-t-il dans ce cas ?

```c
int main()
{
  int ** matrice = NULL;
  int i, j, n;
  // ...
  matrice = (int**)malloc(n*sizeof(int*));
  if(matrice != NULL) {
    i = 0;
    do{
      matrice[i] = (int*)malloc(n*sizeof(int));
      i ++;
    } while (i < n && matrice[i-1] != NULL);
    // ...
    for(j = 0; j < i; j++)
      free(matrice[j]);
    free(matrice);
  }
  return 0;
}
```



Et ici, que se passe-t-il dans ce cas ?

```c
int main()
{
  int ** matrice = NULL;
  int i, j, n;
  // ...
  matrice = (int**)malloc(n*sizeof(int*));
  if(matrice != NULL) {
    i = 0;
    do{
      matrice[i] = (int*)malloc(n*sizeof(int));
      i ++;
    } while (i < n && matrice[i-1] != NULL);
    // ...
    for(j = 0; j < i; j++)
      free(matrice[j]);
    free(matrice);
  }
  return 0;
}
```



Et ici, que se passe-t-il dans ce cas ?

```c
int main()
{
  int ** matrice = NULL;
  int i, j, n;
  // ...
  matrice = (int**)malloc(n*sizeof(int*));
  if(matrice != NULL) {
    i = 0;
    do{
      matrice[i] = (int*)malloc(n*sizeof(int));
      i ++;
    } while (i < n && matrice[i-1] != NULL);
    // ...
    for(j = 0; j < i; j++)
      free(matrice[j]);
    free(matrice);
  }
  return 0;
}
```

0x2···9		
0x2···8		
0x2···7		
0x2···6		
0x2···5		
0x2···4		
0x2···3		
0x2···2	matrice[1]	0x2···7
0x2···1	matrice[0]	0x2···4

0x1···2		
0x1···1		
0x1···0	matrice	0x2···1

Exemple de l'allocation d'une structure

```
struct temps { short heure, minute, seconde; };

void afficher_horaire(struct temps h)
{
  // ... meme function que slide 82
}

int
main()
{
  struct temps * horaire;
  horaire = (struct temps*)malloc(sizeof(struct temps));
  (*horaire).heure = 11;
  (*horaire).minute = 2;
  (*horaire).seconde = 35;
  // ...
  afficher_horaire(*horaire);
  // ...
  free(horaire);
  return 0;
}
```

Exemple de l'allocation d'une structure

```c
struct temps { short heure, minute, seconde; };

void afficher_horaire(struct temps h)
{
  // ... meme function que slide 82
}

int
main()
{
  struct temps * horaire;
  horaire = (struct temps*)malloc(sizeof(struct temps));
  (*horaire).heure = 11;
  (*horaire).minute = 2;
  (*horaire).seconde = 35;
  // ...
  afficher_horaire(*horaire);
  // ...
  free(horaire);
  return 0;
}
```

```
$> gcc structure-allocation.c
$> ./a.out
Horaire : 11: 2:35
```

Attention à l'amalgame

⚠ L'utilisation des pointeurs n'implique en aucun cas l'utilisation de l'allocation dynamique.

```
int a = 17;
int *ptr = &a;    // declaration d'un pointeur sur un entier
                  // dont la valeur est l'adresse de la variable 'a'

printf("--> %d %d \n", a, *ptr); // --> 17 17
*ptr = 26;
printf("--> %d %d \n", a, *ptr); // --> 26 26
```

Que va-t-on voir dans ce cours ?

(1) Les éléments de base du C

(2) Les structures de contrôle

(3) Les tableaux et les structures

(4) Les pointeurs et l'allocation dynamique

(5) Les fonctions

(6) Les autres types dérivés

(7) Les fichiers

(8) Quelques éléments avancés

(9) Complément 1 : Les listes chaînées

(10) Complément 2 : Les piles et les files

Les fonctions

Les fonctions

Les fonctions

Retour sur ce qu'est une fonction

- Rappel de L1 : Une fonction est une action globale utilisée dans un programme principal ou une fonction, et pouvant faire appel à plusieurs actions pour se réaliser. Une fonction admet en entrée un ou plusieurs paramètres et restitue (renvoie) au programme appelant un ou plusieurs résultats.
- Une fonction est donc définit par :
  - un type de retour, ou vide (`void`) si elle ne retourne aucune valeur,
  - un identificateur (ou nom),
  - une liste de paramètres (type et nom),
  - et une suite d'instructions.

```
type_retour nom_fonction (type1 param1, type2 param2, ...)
{
  instruction1
  ...
}
```

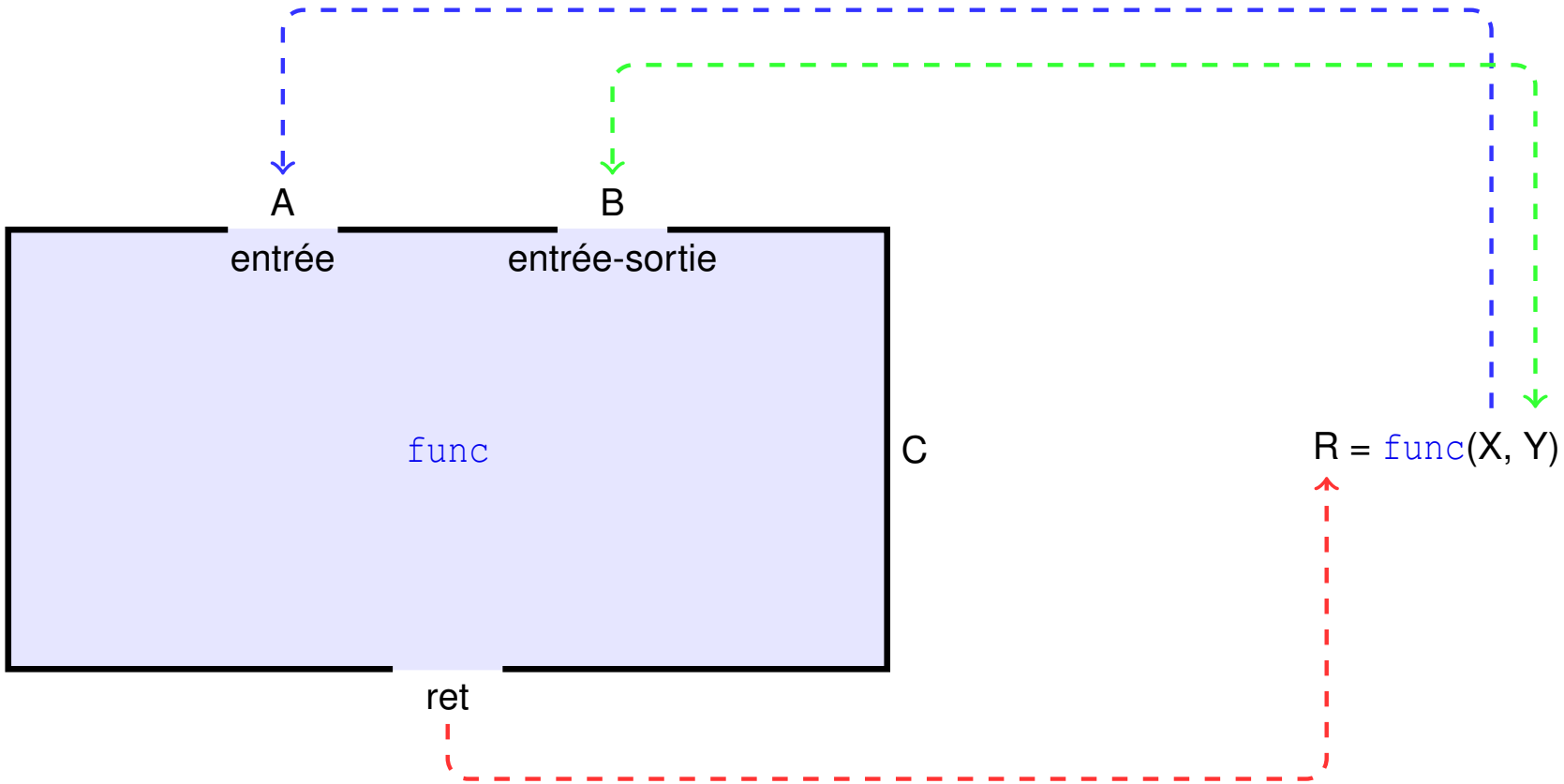
Retour sur ce qu'est une fonction

- Rappel de L1 : Une fonction est une action globale utilisée dans un programme principal ou une fonction, et pouvant faire appel à plusieurs actions pour se réaliser. Une fonction admet en entrée un ou plusieurs paramètres et restitue (renvoie) au programme appelant un ou plusieurs résultats.
- Une fonction est donc définit par :
  - un type de retour, ou vide (`void`) si elle ne retourne aucune valeur,
  - un identificateur (ou nom),
  - une liste de paramètres (type et nom),
  - et une suite d'instructions.

```
type_retour nom_fonction (type1 param1, type2 param2, ...)
{
  instruction1
  ...
}
```

- La définition d'une fonction peut se faire :
  - avant la fonction `main`,
  - après la fonction `main`, mais dans ce cas, elle doit être déclarée avant le `main`.

Vue schématique d'une fonction en C



Exemple de définition d'une fonction

```c
#include <stdio.h>

int fonction1(int);

int fonction2(int b)
{
  printf(" ... fonction2(%d) \n", b);
  return 2*b;
}

int main()
{
  // ...
  return 0;
}

int fonction1(int a)
{
  printf(" ... fonction1(%d) \n", a);
  return 2*a;
}
```

Les fonctions

Exemple de fonction sans paramètre et sans retour

```c
void foo(); // declaration de la fonction

int main()
{
  foo();    // appel a la fonction
  return 0;
}

void foo() // definition de la fonction
{
  int i, j, n = 4;
  for(i = 0; i < n; i ++) {
    for(j = 0; j < n; j ++)
      printf("*");
    printf(" \n");
  }
}
```

- L'appel à une fonction se fait en indiquant le nom de cette fonction, suivi de la valeur des paramètres (si nécessaire).

Exemple de fonction sans paramètre et sans retour

```c
void foo(); // declaration de la fonction

int main()
{
  foo();    // appel a la fonction
  return 0;
}

void foo() // definition de la fonction
{
  int i, j, n = 4;
  for(i = 0; i < n; i ++) {
    for(j = 0; j < n; j ++)
      printf("*");
    printf(" \n");
  }
}
```

- L'appel à une fonction se fait en indiquant le nom de cette fonction, suivi de la valeur des paramètres (si nécessaire).

```
$> gcc fonction.c
$> ./a.out
****
****
****
****
```

Exemple de fonction sans paramètre

```c
int mul_2x2();

int main()
{
  int i = mul_2x2(); // affectation de la
                     //   valeur renvoyee
  printf("mul_2x2() -> %d \n", i);

  return 0;
}

int mul_2x2()
{
  int r = 2 * 2;
  return r; // retour de la fonction
}
```

- Le retour d'une fonction se fait grâce à l'instruction `return`.
- Cette valeur est de même type que le type de retour de la fonction.
- Cette valeur est renvoyée à la fonction appelante.
- La valeur renvoyée pourra ensuite être affectée à une variable.

Exemple de fonction sans paramètre

```c
int mul_2x2();

int main()
{
  int i = mul_2x2(); // affectation de la
                     //   valeur renvoyee
  printf("mul_2x2() -> %d \n", i);

  return 0;
}

int mul_2x2()
{
  int r = 2 * 2;
  return r; // retour de la fonction
}
```

- Le retour d'une fonction se fait grâce à l'instruction `return`.
- Cette valeur est de même type que le type de retour de la fonction.
- Cette valeur est renvoyée à la fonction appelante.
- La valeur renvoyée pourra ensuite être affectée à une variable.

```
$> gcc fonction-retour.c
$> ./a.out
mul_2x2() -> 4
```

Les fonctions

Passage des paramètres par copie

- En langage C, lors de l'appel à une fonction, les paramètres (arguments) sont passés par copie, ou par valeur :
  - lors de l'exécution, les paramètres dans la fonction sont une copie en mémoire, sur la pile, des paramètres de la fonction appelante,
  - par conséquent toute modification de la valeur d'un paramètre dans une fonction ne modifie pas la valeur de ce paramètre dans la fonction appelante,
  - les copies temporaires des paramètres sont détruites à la sortie de la fonction.

```c
int mul(int x, int y)
{
  int res = x * y;
  x = 19;
  return res;
}
```

```c
int main()
{
  int a = 17, b = 18;
  int r = mul(a, b);
  // ... a = 17, b = 18 et r = 306
  return 0;
}
```

Passage des paramètres par copie

- En langage C, lors de l'appel à une fonction, les paramètres (arguments) sont passés par copie, ou par valeur :
  - lors de l'exécution, les paramètres dans la fonction sont une copie en mémoire, sur la pile, des paramètres de la fonction appelante,
  - par conséquent toute modification de la valeur d'un paramètre dans une fonction ne modifie pas la valeur de ce paramètre dans la fonction appelante,
  - les copies temporaires des paramètres sont détruites à la sortie de la fonction.

```c
int mul(int x, int y)
{
  int res = x * y;
  x = 19;
  return res;
}
```

```c
int main()
{
  int a = 17, b = 18;
  int r = mul(a, b);
  // ... a = 17, b = 18 et r = 306
  return 0;
}
```

- Dans cet exemple
  - `x` est une copie de `a`,
  - et `y` est une copie de `b`.

Que se passe-t-il dans cet exemple ?

```
int mul(int x, int y)
{
  int res = x * y;
  x = 19;
  return res;
}

int main()
{
  int a = 17, b = 18;
  int r = mul(a, b);
  // ... a = 17, b = 18 et r = 306
  return 0;
}
```

0x1···F		
0x1···E		
0x1···D		
0x1···C		
0x1···B		
0x1···A		
0x1···9		
0x1···8		
0x1···7		
0x1···6		
0x1···5		
0x1···4		
0x1···3		
0x1···2		
0x1···1		
0x1···0		

Que se passe-t-il dans cet exemple ?

```
int mul(int x, int y)
{
  int res = x * y;
  x = 19;
  return res;
}

int main()
{
  int a = 17, b = 18;
  int r = mul(a, b);
  // ... a = 17, b = 18 et r = 306
  return 0;
}
```

0x1···F		
0x1···E		
0x1···D		
0x1···C		
0x1···B		
0x1···A		
0x1···9		
0x1···8		
0x1···7		
0x1···6		
0x1···5		
0x1···4		
0x1···3		
0x1···2		
0x1···1	b	18
0x1···0	a	17

Que se passe-t-il dans cet exemple ?

```
int mul(int x, int y)
{
  int res = x * y;
  x = 19;
  return res;
}

int main()
{
  int a = 17, b = 18;
  int r = mul(a, b);
  // ... a = 17, b = 18 et r = 306
  return 0;
}
```

0x1···F		
0x1···E		
0x1···D		
0x1···C		
0x1···B		
0x1···A		
0x1···9		
0x1···8		
0x1···7		
0x1···6		
0x1···5	y	18
0x1···4	x	17
0x1···3		
0x1···2	r	...
0x1···1	b	18
0x1···0	a	17

Que se passe-t-il dans cet exemple ?

```
int mul(int x, int y)
{
  int res = x * y;
  x = 19;
  return res;
}

int main()
{
  int a = 17, b = 18;
  int r = mul(a, b);
  // ... a = 17, b = 18 et r = 306
  return 0;
}
```

0x1···F		
0x1···E		
0x1···D		
0x1···C		
0x1···B		
0x1···A		
0x1···9		
0x1···8		
0x1···7		
0x1···6	res	306
0x1···5	y	18
0x1···4	x	17
0x1···3		
0x1···2	r	...
0x1···1	b	18
0x1···0	a	17

Que se passe-t-il dans cet exemple ?

```c
int mul(int x, int y)
{
  int res = x * y;
  x = 19;
  return res;
}

int main()
{
  int a = 17, b = 18;
  int r = mul(a, b);
  // ... a = 17, b = 18 et r = 306
  return 0;
}
```

0x1···F		
0x1···E		
0x1···D		
0x1···C		
0x1···B		
0x1···A		
0x1···9		
0x1···8		
0x1···7		
0x1···6	res	306
0x1···5	y	18
0x1···4	x	~~17~~ 19
0x1···3		
0x1···2	r	...
0x1···1	b	18
0x1···0	a	17

Que se passe-t-il dans cet exemple ?

```
int mul(int x, int y)
{
  int res = x * y;
  x = 19;
  return res;
}

int main()
{
  int a = 17, b = 18;
  int r = mul(a, b);
  // ... a = 17, b = 18 et r = 306
  return 0;
}
```

0x1···F		
0x1···E		
0x1···D		
0x1···C		
0x1···B		
0x1···A		
0x1···9		
0x1···8		
0x1···7		
0x1···6	res	306
0x1···5	y	18
0x1···4	x	~~17~~ 19
0x1···3		
0x1···2	r	306
0x1···1	b	18
0x1···0	a	17

Que se passe-t-il dans cet exemple ?

```
int mul(int x, int y)
{
  int res = x * y;
  x = 19;
  return res;
}

int main()
{
  int a = 17, b = 18;
  int r = mul(a, b);
  // ... a = 17, b = 18 et r = 306
  return 0;
}
```

0x1···F		
0x1···E		
0x1···D		
0x1···C		
0x1···B		
0x1···A		
0x1···9		
0x1···8		
0x1···7		
0x1···6	res	306
0x1···5	y	18
0x1···4	x	~~17~~ 19
0x1···3		
0x1···2	r	306
0x1···1	b	18
0x1···0	a	17

Exemple de passage de paramètre par copie

```c
float puissance(float x, int n)
{
  int i = 0;
  float res = 1.f;

  for(i = 0; i < n; i++)
    res = res * x;
  x = res;
  printf("--> %f \n", x);
  return res;
}

int main(){
  float f, r;
  int n;

  printf("Entrez f et n, pour calculer f^n = ");
  scanf("%f %d", &f, &n);
  r = puissance(f, n);
  printf("--> %f %f\n", r, f);
  return 0;
}
```

Exemple de passage de paramètre par copie

```c
float puissance(float x, int n)
{
  int i = 0;
  float res = 1.f;

  for(i = 0; i < n; i++)
    res = res * x;
  x = res;
  printf("--> %f \n", x);
  return res;
}

int main(){
  float f, r;
  int n;

  printf("Entrez f et n, pour calculer f^n = ");
  scanf("%f %d", &f, &n);
  r = puissance(f, n);
  printf("--> %f %f\n", r, f);
  return 0;
}
```

```
$> gcc param-copie.c
$> ./a.out
Entrez f et n, pour calculer f^n = 5 2
--> 25.000000
--> 25.000000 5.000000
```

Exercice

Écrire une fonction qui prend en paramètre deux valeurs entières a et b, puis qui calcule et retourne le pgcd de a et b.

Exercice

Écrire une fonction qui prend en paramètre deux valeurs entières a et b, puis qui calcule et retourne le pgcd de a et b.

```c
int pgcd(int a, int b)
{
  int c;
  do{
    // ... echanger a et b, si a < b
    if(a < b) {
      a = a + b; b = a - b; a = a - b;
    }
    // ... calculer la difference entre a et b, puis affecter c a la variable a
    c = a - b;
    a = c;
  } while(c != 0);
  return b;
}

int main()
{
  int x, y, r;

  printf("Entrez les entiers a et b= ");
  scanf("%d %d", &x, &y);
  r = pgcd(x, y);
  printf("pgcd(%d, %d) = %d \n", x, y, r);

  return 0;
}
```

Les fonctions

Pourquoi passer un paramètre par adresse ?

- Lorsque l'on veut qu'une fonction puisse modifier un paramètre, il faut le passer par adresse, ou plus particulièrement il faut passer la valeur de son adresse :
  - lors de l'exécution, ce paramètre dans la fonction est une copie en mémoire, sur la pile, d'une adresse, plus particulièrement de l'adresse de la variable dans la fonction appelante,
  - les deux pointeurs (adresses) pointent vers la même zone mémoire,
  - par conséquent toute modification de la valeur à cette adresse modifiera également la valeur à la même adresse dans la fonction appelante,
  - les copies temporaires des paramètres sont également détruites à la sortie de la fonction.

```c
void mul(int x, int y, int *z)
{
  *z = x * y;
}
```

```c
int main()
{
  int a = 17, b = 18, r;
  mul(a, b, &r);
  // ... a = 17, b = 18 et r = 306
  return 0;
}
```

Pourquoi passer un paramètre par adresse ?

- Lorsque l'on veut qu'une fonction puisse modifier un paramètre, il faut le passer par adresse, ou plus particulièrement il faut passer la valeur de son adresse :
  - lors de l'exécution, ce paramètre dans la fonction est une copie en mémoire, sur la pile, d'une adresse, plus particulièrement de l'adresse de la variable dans la fonction appelante,
  - les deux pointeurs (adresses) pointent vers la même zone mémoire,
  - par conséquent toute modification de la valeur à cette adresse modifiera également la valeur à la même adresse dans la fonction appelante,
  - les copies temporaires des paramètres sont également détruites à la sortie de la fonction.

```c
void mul(int x, int y, int *z)
{
  *z = x * y;
}
```

```c
int main()
{
  int a = 17, b = 18, r;
  mul(a, b, &r);
  // ... a = 17, b = 18 et r = 306
  return 0;
}
```

- Dans cet exemple
  - `z` est une copie de l'adresse de `r`,
  - et `z` pointe au même endroit en mémoire que l'adresse de `r`.

Pourquoi passer un paramètre par adresse ?

- Lorsque l'on veut qu'une fonction puisse modifier un paramètre, il faut le passer par adresse, ou plus particulièrement il faut passer la valeur de son adresse :
  - lors de l'exécution, ce paramètre dans la fonction est une copie en mémoire, sur la pile, d'une adresse, plus particulièrement de l'adresse de la variable dans la fonction appelante,
  - les deux pointeurs (adresses) pointent vers la même zone mémoire,
  - par conséquent toute modification de la valeur à cette adresse modifiera également la valeur à la même adresse dans la fonction appelante,
  - les copies temporaires des paramètres sont également détruites à la sortie de la fonction.

```c
void mul(int x, int y, int *z)
{
  *z = x * y;
}
```

```c
int main()
{
  int a = 17, b = 18, r;
  mul(a, b, &r);
  // ... a = 17, b = 18 et r = 306
  return 0;
}
```

- Dans cet exemple
  - `z` est une copie de l'adresse de `r`,
  - et `z` pointe au même endroit en mémoire que l'adresse de `r`.
- C'est comme cela que fonctionne la fonction de `scanf`.

Maintenant, que se passe-t-il dans cet exemple ?

```c
void mul(int x, int y, int *z)
{
  *z = x * y;
}

int main()
{
  int a = 17, b = 18, r;
  mul(a, b, &r);
  // ... a = 17, b = 18 et r = 306
  return 0;
}
```

0x1···F		
0x1···E		
0x1···D		
0x1···C		
0x1···B		
0x1···A		
0x1···9		
0x1···8		
0x1···7		
0x1···6		
0x1···5		
0x1···4		
0x1···3		
0x1···2		
0x1···1		
0x1···0		

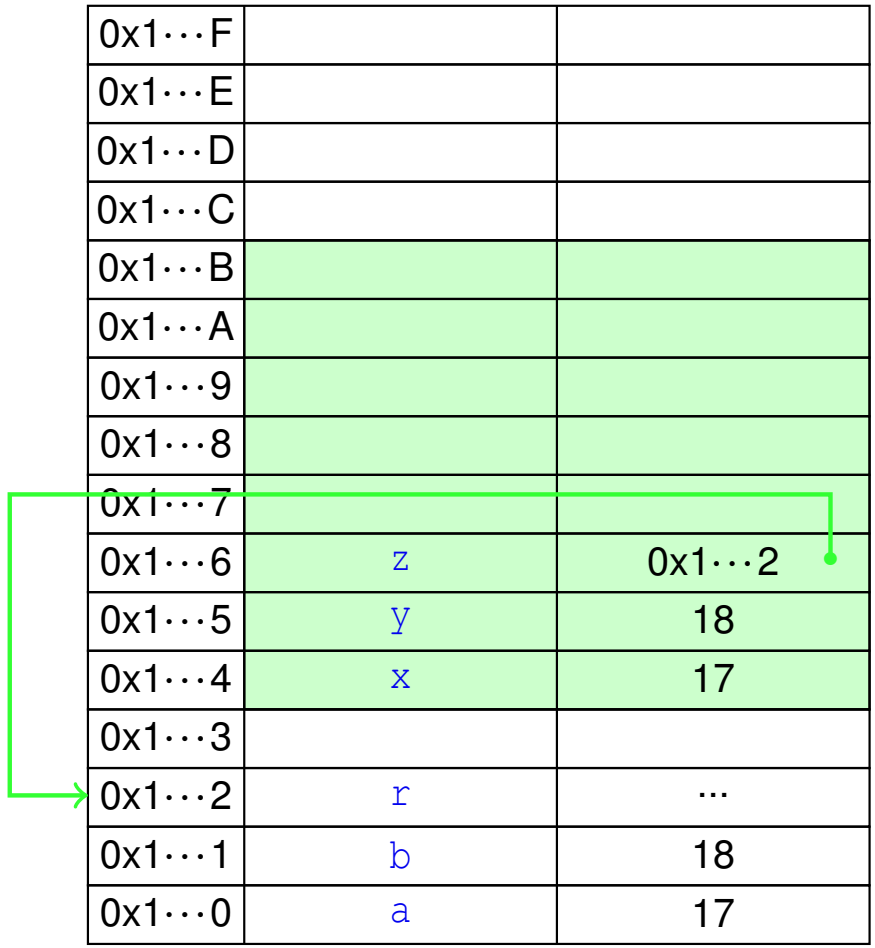
Maintenant, que se passe-t-il dans cet exemple ?

```c
void mul(int x, int y, int *z)
{
  *z = x * y;
}

int main()
{
  int a = 17, b = 18, r;
  mul(a, b, &r);
  // ... a = 17, b = 18 et r = 306
  return 0;
}
```

0x1···F		
0x1···E		
0x1···D		
0x1···C		
0x1···B		
0x1···A		
0x1···9		
0x1···8		
0x1···7		
0x1···6		
0x1···5		
0x1···4		
0x1···3		
0x1···2	r	...
0x1···1	b	18
0x1···0	a	17

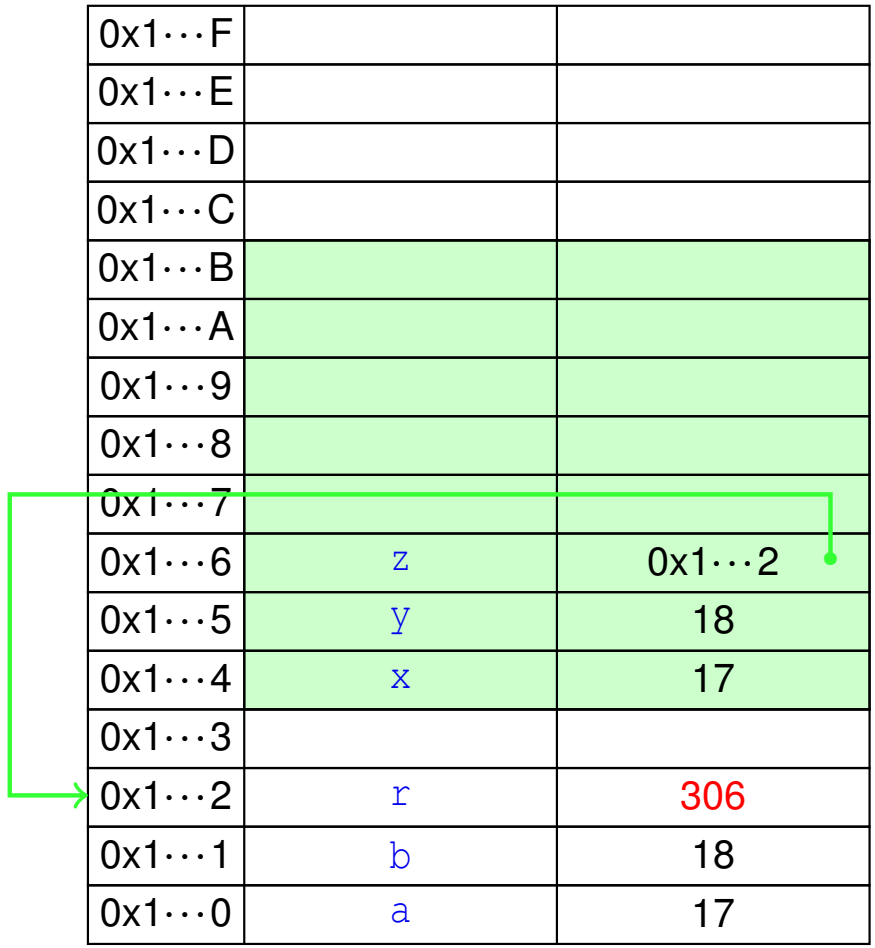
Maintenant, que se passe-t-il dans cet exemple ?

```c
void mul(int x, int y, int *z)
{
  *z = x * y;
}

int main()
{
  int a = 17, b = 18, r;
  mul(a, b, &r);
  // ... a = 17, b = 18 et r = 306
  return 0;
}
```



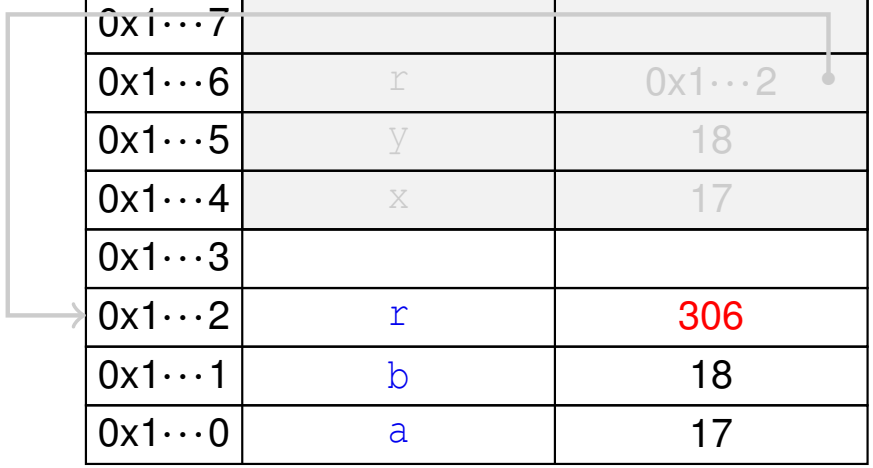
Maintenant, que se passe-t-il dans cet exemple ?

```c
void mul(int x, int y, int *z)
{
  *z = x * y;
}

int main()
{
  int a = 17, b = 18, r;
  mul(a, b, &r);
  // ... a = 17, b = 18 et r = 306
  return 0;
}
```



Maintenant, que se passe-t-il dans cet exemple ?

```c
void mul(int x, int y, int *z)
{
  *z = x * y;
}

int main()
{
  int a = 17, b = 18, r;
  mul(a, b, &r);
  // ... a = 17, b = 18 et r = 306
  return 0;
}
```



Exemple de passage de paramètre par adresse

```
void puissance(float *x, int n, float *res)
{
  int i = 0;
  *res = 1.f;

  for(i = 0; i < n; i++)
    *res = *res * *x;
  *x = *res;
  printf("--> %f \n", *x);
}

int main(){
  float f, r;
  int n;

  printf("Entrez f et n, pour calculer f^n = ");
  scanf("%f %d", &f, &n);
  puissance(&f, n, &r);
  printf("--> %f %f\n", r, f);
  return 0;
}
```

Exemple de passage de paramètre par adresse

```c
void puissance(float *x, int n, float *res)
{
  int i = 0;
  *res = 1.f;

  for(i = 0; i < n; i++)
    *res = *res * *x;
  *x = *res;
  printf("--> %f \n", *x);
}

int main(){
  float f, r;
  int n;

  printf("Entrez f et n, pour calculer f^n = ");
  scanf("%f %d", &f, &n);
  puissance(&f, n, &r);
  printf("--> %f %f\n", r, f);
  return 0;
}
```

```
$> gcc param-adresse.c
$> ./a.out
Entrez f et n, pour calculer f^n = 5 2
--> 25.000000
--> 25.000000 25.000000
```

Exercice

Écrire une fonction qui prend en paramètre deux entiers a et b, puis calcule et renvoie la somme et le produit de a et b.

Exercice

Écrire une fonction qui prend en paramètre deux entiers a et b, puis calcule et renvoie la somme et le produit de a et b.

```c
int somme_produit(int a, int b, int * prod)
{
  int somme = a + b;
  *prod = a * b;

  return somme;
}

int main()
{
  int x, y, s, p;

  printf("Entrez les entiers a et b= ");
  scanf("%d %d", &x, &y);

  s = somme_produit(x, y, &p);
  printf("%d + %d = %d et %d x %d = %d \n", x, y, s, x, y, p);

  return 0;
}
```

Les fonctions

Tableau 1D en paramètre d'une fonction

- Pour passer un tableau 1D en paramètre d'une fonction, on passe à la fonction un pointeur vers un élément, typiquement le premier élément du tableau.
- En langage C, la déclaration de telles fonctions utilise la syntaxe suivante.

```c
void foo1(int[3]);
void foo2(int[]);
void foo3(int*);

int main(){
  int tab[3] = {0,1,2};
  // ...
  foo1(&(tab[0]));
  foo2(tab);
  foo3(tab);
  return 0;
}
```

Tableau 1D en paramètre d'une fonction

- Pour passer un tableau 1D en paramètre d'une fonction, on passe à la fonction un pointeur vers un élément, typiquement le premier élément du tableau.
- En langage C, la déclaration de telles fonctions utilise la syntaxe suivante.

```c
void foo1(int[3]);
void foo2(int[]);
void foo3(int*);

int main(){
  int tab[3] = {0,1,2};
  // ...
  foo1(&(tab[0]));
  foo2(tab);
  foo3(tab);
  return 0;
}
```

- Remarques
  - lors de l'appel à `foo1`, aucune vérification n'est faite pour savoir si tab est effectivement un tableau de 3 éléments,
  - une bonne manière de programmer est d'utiliser la déclaration 2 ou 3 (`foo2` ou `foo3`), et de passer également la taille du tableau en paramètre.

Exemple pour tableau 1D

```c
void print_tab(int[], int);

int main()
{
  int tab[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  // ...
  print_tab(tab, 10);
  print_tab(&(tab[1]), 9);
  return 0;
}

void print_tab(int tab[], int n)
{
  int i;
  for(i = 0; i < n; i++) {
    if(i == 0)
      printf("{");
    printf("%d", tab[i]);
    if(i != n-1)
      printf(", ");
    else
      printf("} \n");
  }
}
```

Exemple pour tableau 1D

```c
void print_tab(int[], int);

int main()
{
  int tab[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  // ...
  print_tab(tab, 10);
  print_tab(&(tab[1]), 9);
  return 0;
}

void print_tab(int tab[], int n)
{
  int i;
  for(i = 0; i < n; i++) {
    if(i == 0)
      printf("{");
    printf("%d", tab[i]);
    if(i != n-1)
      printf(", ");
    else
      printf("} \n");
  }
}
```

```
$> gcc param-tab.c
$> ./a.out
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

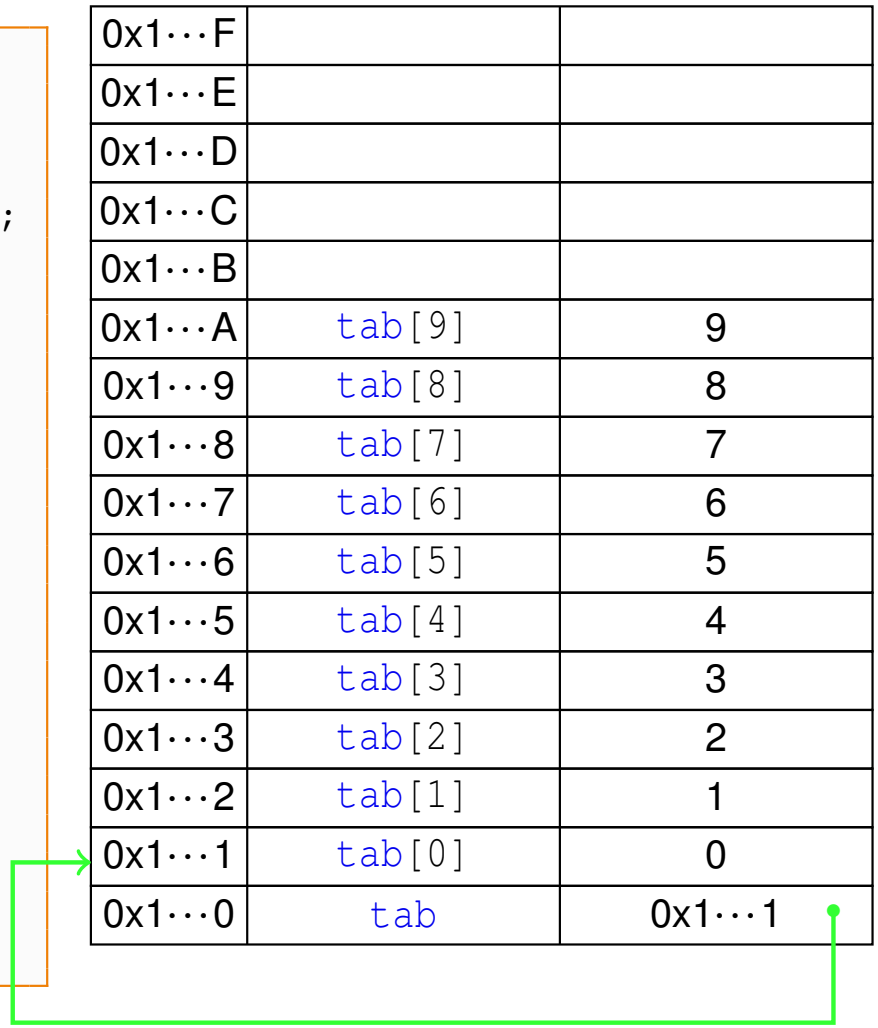
Et comment ça fonctionne avec un tableau ?

```c
void print_tab(int[], int);

int main()
{
  int tab[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  // ...
  print_tab(tab, 10);
  print_tab(&(tab[1]), 9);
  return 0;
}

void print_tab(int tab[], int n)
{
  int i;
  for(i = 0; i < n; i++) {
    if(i == 0)
      printf("{");
    printf("%d", tab[i]);
    if(i != n-1)
      printf(", ");
    else
      printf("} \n");
  }
}
```

0x1···F		
0x1···E		
0x1···D		
0x1···C		
0x1···B		
0x1···A		
0x1···9		
0x1···8		
0x1···7		
0x1···6		
0x1···5		
0x1···4		
0x1···3		
0x1···2		
0x1···1		
0x1···0		

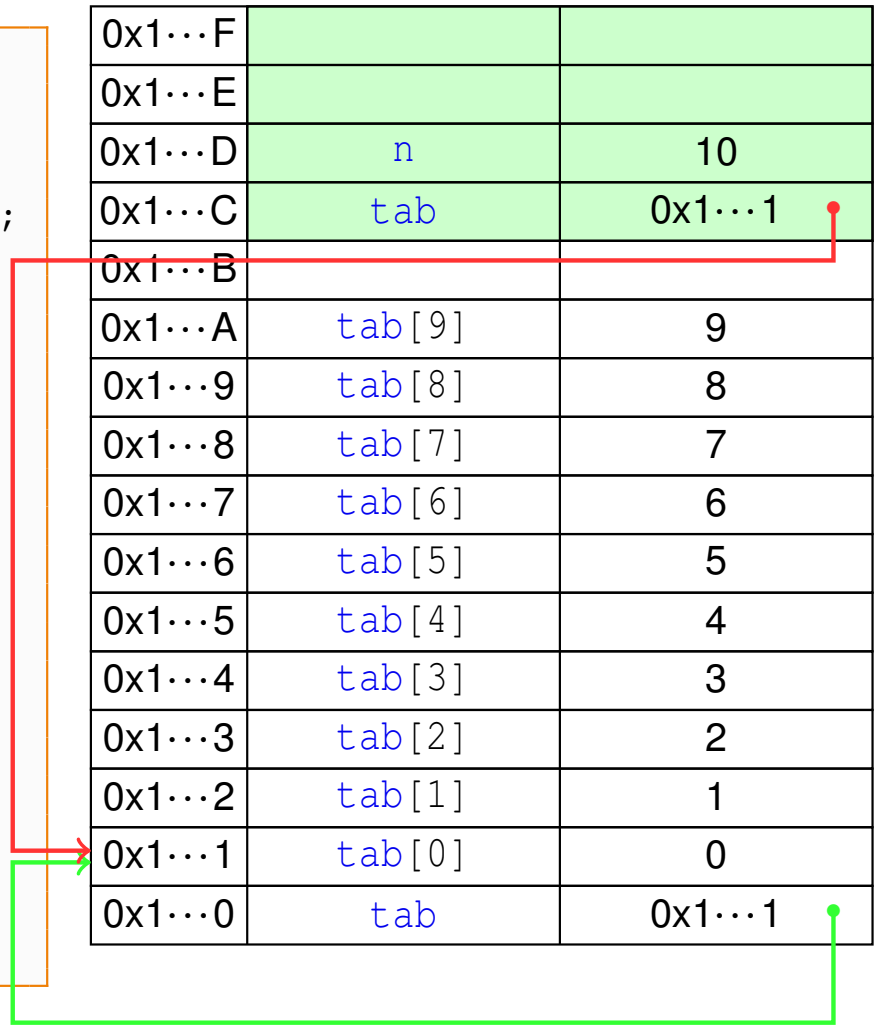
Et comment ça fonctionne avec un tableau ?

```c
void print_tab(int[], int);

int main()
{
  int tab[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  // ...
  print_tab(tab, 10);
  print_tab(&(tab[1]), 9);
  return 0;
}

void print_tab(int tab[], int n)
{
  int i;
  for(i = 0; i < n; i++) {
    if(i == 0)
      printf("{");
    printf("%d", tab[i]);
    if(i != n-1)
      printf(", ");
    else
      printf("} \n");
  }
}
```



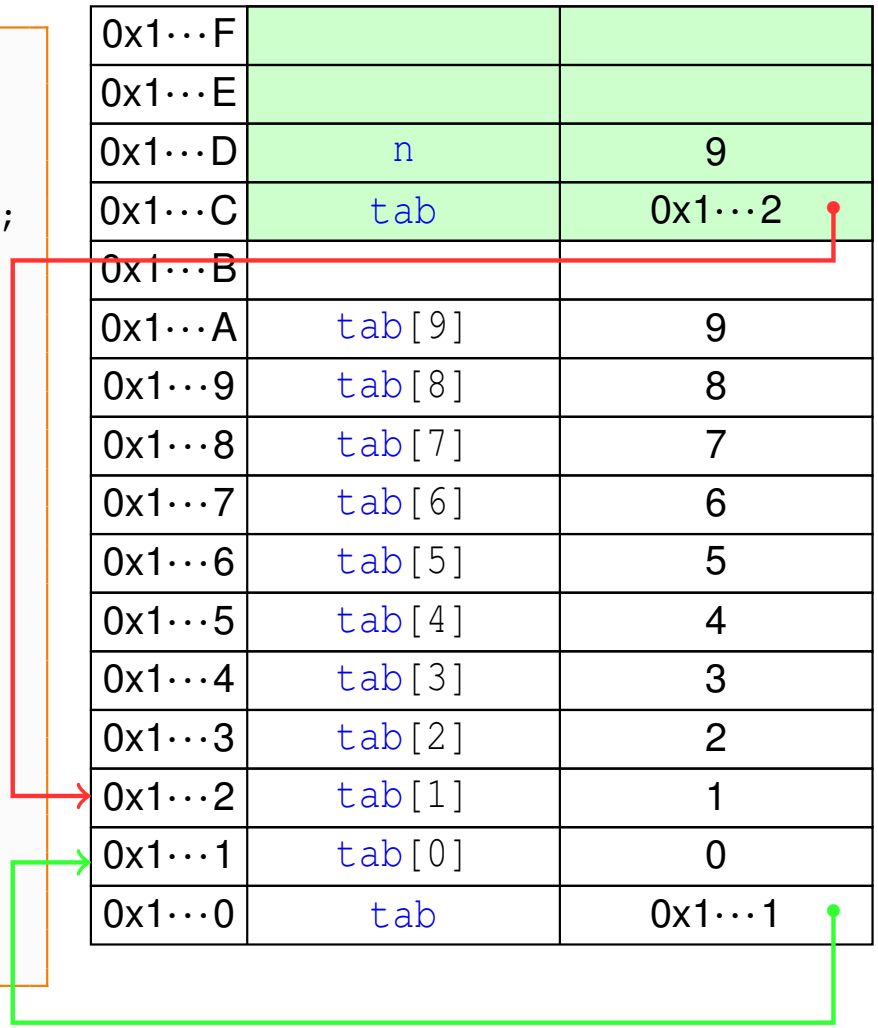
Et comment ça fonctionne avec un tableau ?

```c
void print_tab(int[], int);

int main()
{
  int tab[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  // ...
  print_tab(tab, 10);
  print_tab(&(tab[1]), 9);
  return 0;
}

void print_tab(int tab[], int n)
{
  int i;
  for(i = 0; i < n; i++) {
    if(i == 0)
      printf("{");
    printf("%d", tab[i]);
    if(i != n-1)
      printf(", ");
    else
      printf("} \n");
  }
}
```



Et comment ça fonctionne avec un tableau ?

```c
void print_tab(int[], int);

int main()
{
  int tab[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  // ...
  print_tab(tab, 10);
  print_tab(&(tab[1]), 9);
  return 0;
}

void print_tab(int tab[], int n)
{
  int i;
  for(i = 0; i < n; i++) {
    if(i == 0)
      printf("{");
    printf("%d", tab[i]);
    if(i != n-1)
      printf(", ");
    else
      printf("} \n");
  }
}
```



Exemple pour tableau 1D alloué dynamiquement

```c
void print_tab(int[], int);

int main()
{
  int i, *tab = (int*)malloc(10*sizeof(int));
  for(i = 0; i < 10; i++)
    tab[i] = i;
  // ...
  print_tab(tab, 10);
  print_tab(tab+1, 9);
  free(tab);
  return 0;
}

void print_tab(int tab[], int n)
{
  // ... meme fonction que l'exemple precedent
}
```

Exemple pour tableau 1D alloué dynamiquement

```
void print_tab(int[], int);

int main()
{
  int i, *tab = (int*)malloc(10*sizeof(int));
  for(i = 0; i < 10; i++)
    tab[i] = i;
  // ...
  print_tab(tab, 10);
  print_tab(tab+1, 9);
  free(tab);
  return 0;
}

void print_tab(int tab[], int n)
{
  // ... meme fonction que l'exemple precedent
}
```

```
$> gcc param-tab-ptr.c
$> ./a.out
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

- Dans ce cas, on peut passer directement le pointeur retourné par `malloc`.

Exercice

Écrire une fonction qui prend en paramètre un tableau d'entier de taille n, et qui détermine puis retourne le nombre d'éléments nuls.

Exercice

Écrire une fonction qui prend en paramètre un tableau d'entier de taille n, et qui détermine puis retourne le nombre d'éléments nuls.

```c
int nb_elements_nuls(int tab[], int n)
{
  int i, res = 0;
  for(i = 0; i < n; i++)
    if(tab[i] == 0)
      res ++;
  return res;
}

int main()
{
  int tab[] = {0, 1, 0, 0, 0, 1, 1, 1, 0};
  int res = 0;

  res = nb_elements_nuls(tab, sizeof(tab)/sizeof(int));
  printf("Le tableau 'tab' contient '%d' elements nuls. \n", res);

  return 0;
}
```

Exercice

Écrire une fonction qui prend en paramètre un tableau d'entier de taille n, et qui détermine puis retourne le nombre d'éléments nuls.

```c
int nb_elements_nuls(int tab[], int n)
{
  int i, res = 0;
  for(i = 0; i < n; i++)
    if(tab[i] == 0)
      res ++;
  return res;
}

int main()
{
  int tab[] = {0, 1, 0, 0, 0, 1, 1, 1, 0};
  int res = 0;

  res = nb_elements_nuls(tab, sizeof(tab)/sizeof(int));
  printf("Le tableau 'tab' contient '%d' elements nuls. \n", res);

  return 0;
}
```

```
$> gcc exercice3.c
$> ./a.out
Le tableau 'tab' contient '5' elements nuls.
```

Exercice

Écrire une fonction qui prend en paramètre une chaîne de caractères, et qui détermine puis retourne sa longueur.

Exercice

Écrire une fonction qui prend en paramètre une chaîne de caractères, et qui détermine puis retourne sa longueur.

```c
int longueur(char str[])
{
  int l = 0;
  while(str[l] != '\0')
    l ++;
  return l;
}

int main()
{
  char str[256];
  int l = -1;
  printf("Entrez une chaine de caracteres = ");
  scanf("%s", str);

  l = longueur(str);
  printf("Votre chaine est de longueur '%d' \n", l);

  return 0;
}
```

Exercice

Écrire une fonction qui prend en paramètre une chaîne de caractères, et qui détermine puis retourne sa longueur.

```c
int longueur(char str[])
{
  int l = 0;
  while(str[l] != '\0')
    l ++;
  return l;
}

int main()
{
  char str[256];
  int l = -1;
  printf("Entrez une chaine de caracteres = ");
  scanf("%s", str);

  l = longueur(str);
  printf("Votre chaine est de longueur '%d' \n", l);

  return 0;
}
```

```
$> gcc exercice4.c
$> ./a.out
Entrez une chaine de caracteres = bonjour
Votre chaine est de longueur '7'
```

Et pour les tableaux multidimensionnels ?

- Pour passer un tableau multidimensionnel, par exemple 2D, en paramètre d'une fonction, on peut passer un pointeur vers le premier élément, d'indice [0][0] :
  - la déclaration de la fonction utilise la même syntaxe que pour un tableau 1D.
- Une autre manière de faire consiste à utiliser la syntaxe suivante.

```
void foo1(int[3][3]);
void foo2(int[][3], int line);

int main(){
  int tab[3][3] = {{0,1,2},{0,1,2},{0,1,2}};
  // ...
  foo1(tab);
  foo2(tab, 3);
  return 0;
}
```

Et pour les tableaux multidimensionnels ?

- Pour passer un tableau multidimensionnel, par exemple 2D, en paramètre d'une fonction, on peut passer un pointeur vers le premier élément, d'indice [0][0] :
  - la déclaration de la fonction utilise la même syntaxe que pour un tableau 1D.
- Une autre manière de faire consiste à utiliser la syntaxe suivante.

```
void foo1(int[3][3]);
void foo2(int[][3], int line);

int main(){
  int tab[3][3] = {{0,1,2},{0,1,2},{0,1,2}};
  // ...
  foo1(tab);
  foo2(tab, 3);
  return 0;
}
```

- Remarques
  - lors de l'appel à `foo1`, aucune vérification n'est faite pour savoir si tab est effectivement un tableau de 3 éléments,
  - seule la première dimension peut être omise dans la déclaration 2 (`foo2`), et il faut alors passer la première dimension du tableau en paramètre.

Exemple pour tableau 2D

```c
void matmul(float A[][2], float X[2], float *R)
{
  int i, j;
  for(i = 0; i < 2; i++) {
    R[i] = 0.f;
    for(j = 0; j < 2; j++) {
      R[i] += A[i][j] * X[j];
    }
  }
}

int main()
{
  float A[2][2] = {{1.f,1.f},{1.f,0.f}}, X[2] = {1.f,2.f}, R[2];
  // ...
  matmul(A, X, R);
  // ...
  printf("R = {%f, %f} \n", R[0], R[1]);
  return 0;
}
```

Exemple pour tableau 2D

```c
void matmul(float A[][2], float X[2], float *R)
{
  int i, j;
  for(i = 0; i < 2; i++) {
    R[i] = 0.f;
    for(j = 0; j < 2; j++) {
      R[i] += A[i][j] * X[j];
    }
  }
}

int main()
{
  float A[2][2] = {{1.f,1.f},{1.f,0.f}}, X[2] = {1.f,2.f}, R[2];
  // ...
  matmul(A, X, R);
  // ...
  printf("R = {%f, %f} \n", R[0], R[1]);
  return 0;
}
```

```
$> gcc param-tab2d.c
$> ./a.out
R = {3.000000, 1.000000}
```

Exemple pour tableau 2D alloué dynamiquement

```
void matmul(float **A, float X[2], float *R)
{ // ... meme fonction que dans l'exemple precedent
}

// void matmul_errone(float A[2][2], float X[2], float *R) { }

int main()
{
  float **A, X[2] = {1.f,2.f}, R[2];
  A = (float**)malloc(2*sizeof(float*));
  A[0] = (float*)malloc(2*sizeof(float));
  A[1] = (float*)malloc(2*sizeof(float));
  A[0][0] = 1.f; A[0][1] = 1.f; A[1][0] = 1.f; A[1][1] = 0.f;
  // ...
  matmul(A, X, R);
  // matmul_errone(A, X, R);
  free(A[1]); free(A[0]); free(A);
  // ...
}
```

Exemple pour tableau 2D alloué dynamiquement

```
void matmul(float **A, float X[2], float *R)
{ // ... meme fonction que dans l'exemple precedent
}

// void matmul_errone(float A[2][2], float X[2], float *R) { }

int main()
{
  float **A, X[2] = {1.f,2.f}, R[2];
  A = (float**)malloc(2*sizeof(float*));
  A[0] = (float*)malloc(2*sizeof(float));
  A[1] = (float*)malloc(2*sizeof(float));
  A[0][0] = 1.f; A[0][1] = 1.f; A[1][0] = 1.f; A[1][1] = 0.f;
  // ...
  matmul(A, X, R);
  // matmul_errone(A, X, R);
  free(A[1]); free(A[0]); free(A);
  // ...
}
```

```
$> gcc param-tab2d-ptr.c
$> ./a.out
R = {3.000000, 1.000000}
```

Exemple pour tableau 2D alloué dynamiquement

```c
void matmul(float **A, float X[2], float *R)
{ // ... meme fonction que dans l'exemple precedent
}

// void matmul_errone(float A[2][2], float X[2], float *R) { }

int main()
{
  float **A, X[2] = {1.f,2.f}, R[2];
  A = (float**)malloc(2*sizeof(float*));
  A[0] = (float*)malloc(2*sizeof(float));
  A[1] = (float*)malloc(2*sizeof(float));
  A[0][0] = 1.f; A[0][1] = 1.f; A[1][0] = 1.f; A[1][1] = 0.f;
  // ...
  matmul(A, X, R);
  // matmul_errone(A, X, R);
  free(A[1]); free(A[0]); free(A);
  // ...
}
```

```
$> gcc param-tab2d-ptr.c
$> ./a.out
R = {3.000000, 1.000000}
```

```
$> clang param-tab2d-ptr.c
param-tab2d-ptr.c:27:10: warning: incompatible pointer types passing 'float **' to
      parameter of type 'float (*)[2]' [-Wincompatible-pointer-types]
  matmul_errone(A, X, R);
                ^
```

Attention tout de même !

- En utilisant les pointeurs pour manipuler les tableaux, on perd la notion de taille de tableau.

```c
int nb_elements_nuls(int tab[], int n)
{
  int i, res = 0;
  for(i = 0; i < n; i++)
    if(tab[i] == 0)
      res ++;
  return res;
}

int main()
{
  int tab[] = {0, 1, 0, 0, 0, 1, 1, 1, 0};
  int * ptr = &(tab[0]), res = 0;

  res = nb_elements_nuls(ptr, sizeof(ptr)/sizeof(int));
  printf("Le tableau pointe par 'ptr' contient '%d' elements nuls. \n", res);

  return 0;
}
```

Attention tout de même !

- En utilisant les pointeurs pour manipuler les tableaux, on perd la notion de taille de tableau.

```c
int nb_elements_nuls(int tab[], int n)
{
  int i, res = 0;
  for(i = 0; i < n; i++)
    if(tab[i] == 0)
      res ++;
  return res;
}

int main()
{
  int tab[] = {0, 1, 0, 0, 0, 1, 1, 1, 0};
  int * ptr = &(tab[0]), res = 0;

  res = nb_elements_nuls(ptr, sizeof(ptr)/sizeof(int));
  printf("Le tableau pointe par 'ptr' contient '%d' elements nuls. \n", res);

  return 0;
}
```

```
$> gcc exercice3-ptr.c
$> ./a.out
Le tableau pointe par 'ptr' contient '1' elements nuls.
```

... car, ici, `sizeof(tab)` $\neq$ `sizeof(ptr)`

Les fonctions

Passage d'une structure en paramètre d'une fonction

- Le langage C autorise de passer une structure en paramètre d'une fonction, soit par copie, soit en passant un pointeur sur cette structure (par adresse).

- Passage par copie :
  - lors de l'exécution, la structure paramètre dans la fonction est une copie en mémoire, sur la pile et champ par champ, de la structure paramètre de la fonction appelante,
  - par conséquent toute modification de la valeur d'un champ de cette structure dans la fonction ne modifie pas la valeur de ce champ dans la structure paramètre de la fonction appelante.

- Passage par adresse :
  - lors de l'exécution, le pointeur sur la structure dans la fonction est une copie en mémoire, sur la pile, du pointeur sur la structure de la fonction appelante,
  - les deux pointeurs (adresses) pointent alors vers la même structure en mémoire,
  - par conséquent toute modification de la valeur d'un champ de cette structure dans la fonction modifiera la valeur de ce champ dans la structure dans la fonction appelante.

Exemple de structure en paramètre de fonction

```c
struct temps { short heure, minute, seconde; };

void afficher_horaire(struct temps h)
{
  printf("Horaire : %2d:%2d:%2d \n", h.heure,
         h.minute, h.seconde);
  h.heure = 0;
  h.minute = 0;
  h.seconde = 0;
}

int
main()
{
  struct temps horaire = {11, 02, 35};
  // ...
  afficher_horaire(horaire);
  // ...
  printf("Et apres modification : %2d:%2d:%2d \n", horaire.heure,
         horaire.minute, horaire.seconde);
  return 0;
}
```

Exemple de structure en paramètre de fonction

```c
struct temps { short heure, minute, seconde; };

void afficher_horaire(struct temps h)
{
  printf("Horaire : %2d:%2d:%2d \n", h.heure,
         h.minute, h.seconde);
  h.heure = 0;
  h.minute = 0;
  h.seconde = 0;
}

int
main()
{
  struct temps horaire = {11, 02, 35};
  // ...
  afficher_horaire(horaire);
  // ...
  printf("Et apres modification : %2d:%2d:%2d \n", horaire.heure,
         horaire.minute, horaire.seconde);
  return 0;
}
```

```
$> gcc structure-param.c
$> ./a.out
Horaire : 11: 2:35
Et apres modification : 11: 2:35
```

Exemple de pointeur de structure en paramètre de fonction

```
void afficher_horaire(struct temps *h)
{
  printf("Horaire : %2d:%2d:%2d \n", (*h).heure,
         (*h).minute, (*h).seconde);
  (*h).heure = 0;                    // h->heure
  (*h).minute = 0;                   // h->minute
  (*h).seconde = 0;                  // h->seconde
}

int
main()
{
  struct temps horaire = {11, 02, 35};
  // ...
  afficher_horaire(&horaire);
  printf("Et apres modification : %2d:%2d:%2d \n", horaire.heure,
         horaire.minute, horaire.seconde);
  return 0;
}
```

Exemple de pointeur de structure en paramètre de fonction

```
void afficher_horaire(struct temps *h)
{
  printf("Horaire : %2d:%2d:%2d \n", (*h).heure,
         (*h).minute, (*h).seconde);
  (*h).heure = 0;                    // h->heure
  (*h).minute = 0;                   // h->minute
  (*h).seconde = 0;                  // h->seconde
}

int
main()
{
  struct temps horaire = {11, 02, 35};
  // ...
  afficher_horaire(&horaire);
  printf("Et apres modification : %2d:%2d:%2d \n", horaire.heure,
         horaire.minute, horaire.seconde);
  return 0;
}
```

```
$> gcc structure-param-ptr.c
$> ./a.out
Horaire : 11: 2:35
Et apres modification :  0: 0: 0
```

- Accéder aux champs d'une structure manipulée à travers un pointeur peut se faire par l'opérateur '->' : `(*h).heure` peut s'écrire `h->heure`.

Les fonctions

Qu'est-ce que la portée d'une variable

- Selon l'endroit où une variable est déclarée, elle pourra ne pas être visible de n'importe quelle zone du programme : c'est ce qu'on appelle la portée (ou visibilité) d'une variable.

Qu'est-ce que la portée d'une variable

- Selon l'endroit où une variable est déclarée, elle pourra ne pas être visible de n'importe quelle zone du programme : c'est ce qu'on appelle la portée (ou visibilité) d'une variable.

- Variables globales : déclarée en dehors de tout bloc d'instructions :
  - elle sera visible de n'importe où dans le programme (portée spatiale),
  - et uniquement à partir de sa déclaration (portée temporelle).

- Variables locales : déclarée à l'intérieure d'un bloc d'instructions :
  - elle sera visible dans ce bloc d'instruction et ses sous-blocs (portée spatiale),
  - et uniquement à partir de sa déclaration (portée temporelle).

Exemple de portées de variables

```c
int var = 17;                                   // 'var' est une variable globale

int main()
{
  int a = 24, b = 18, i;                        // 'a', 'b' et 'i' sont 3 variables
                                                //   locales a la fonction

  for(i = 0; i < 10; i++) {
    int a = var * i;                            // 'a' est une variable locale au
                                                //   corps de boucle
    // ... attention 'y' n'est toujours pas accessible
    //
    int y = a;                                  // ici 'a' correspond a la variable 'a'
                                                //   declaree dans le bloc d'instructions
                                                //   courant
    printf("[a, y, b, i] = [%d, %d, %d, %d] \n", a, y, b, i);
  }
  // ... attention 'y' n'est plus accessible
  // ... attention 'var2' n'est toujours pas accessible
  printf("a = %d ? \n", a);
  return 0;
}

int var2 = 19;                                  // 'var2' est une variable globale
                                                //   accessible uniquement a partir de sa
void foo() {                                    //   declaration
  printf("var2 = %d \n", var2);
}
```

Que va-t-on voir dans ce cours ?

Les autres types dérivés

Les autres types dérivés

1. Unions

2. Champs de bits

3. Enumérations

4. Redéfinition de type

Les autres types dérivés

Qu'est-ce qu'une union ?

- En langage C, une union est une structure de données qui permet de rassembler un ensemble de valeurs de types différents susceptibles d'occuper la même zone mémoire.
- La déclaration d'une union se fait de la manière suivante.

```
union nom_union
{
  type1 champ1;
  type2 champ2;
  ...
};
```

- Remarques sur les unions :
  - si les membres d'une union ne sont pas tous de la même taille, la place réservée en mémoire correspond à la place mémoire du plus grand de ces membres,
  - l'accès aux membres d'une union se fait de la même manière que dans le cas d'une structure.

Exemple d'utilisation d'une union

```c
#include <stdio.h>

union ascii
{
  int  encodage;
  char caractere;
};

int main()
{
  union ascii tmp;
  // ...
  tmp.encodage = 65;
  printf("union(encodage) = %d \n", tmp.encodage);
  // ...
  printf("union(caractere) = %c \n", tmp.caractere);
  // ...
  return 0;
}
```

Exemple d'utilisation d'une union

```c
#include <stdio.h>

union ascii
{
  int  encodage;
  char caractere;
};

int main()
{
  union ascii tmp;
  // ...
  tmp.encodage = 65;
  printf("union(encodage) = %d \n", tmp.encodage);
  // ...
  printf("union(caractere) = %c \n", tmp.caractere);
  // ...
  return 0;
}
```

```
$> gcc union.c
$> ./a.out
union(encodage) = 65
union(caractere) = A
```

- Remarque : ici les deux champs 'encodage' et 'caractere' utilisent le même espace mémoire : modifier l'un modifie l'autre automatiquement.

Les autres types dérivés

Qu'est-ce qu'un champ de bits ?

- En langage C, un champ de bits est une structure pour laquelle on spécifie la longueur des champs en nombre de bits, pour les champs de type entier (c'est-à-dire, `int` ou `unsigned int`) uniquement.
- La déclaration d'un champ de bits se fait de la manière suivante.

```
struct champ_bits
{
  type [champ] : nb_bits;
  ...
};
```

- Le langage C autorise que l'on ne nomme pas un membre d'un champ de bits.
- Contraintes à respecter :
  - la taille d'un champ de bits doit être inférieure au nombre de bits d'un entier,
  - un champ de bits n'a pas d'adresse : on ne peut donc pas lui appliquer l'opérateur '`&`'.

Exemple d'utilisation d'un champ de bits

```
struct nombre
{
  unsigned int signe : 1;            // {0, 1}
  unsigned int valeur_absolue : 31; // {0, ... , 2^(31)-1}
};

void print_nombre(struct nombre x)
{
  if(x.signe == 0)       printf("+");
  else if(x.signe == 1) printf("-");
  else                   printf("?");
  printf("%u \n", x.valeur_absolue);
}

int main() {
  struct nombre nb1 = {0, 1255445}, nb2 = nb1;     nb2.signe = 1;
  // ...
  printf("nb1= "); print_nombre(nb1);
  printf("nb2= "); print_nombre(nb2);
  // ...
  nb1.signe += 2;  printf("nb1= ");   print_nombre(nb1);
  // ...
  return 0;
}
```

Exemple d'utilisation d'un champ de bits

```
struct nombre
{
  unsigned int signe : 1;            // {0, 1}
  unsigned int valeur_absolue : 31; // {0, ... , 2^(31)-1}
};

void print_nombre(struct nombre x)
{
  if(x.signe == 0)       printf("+");
  else if(x.signe == 1) printf("-");
  else                   printf("?");
  printf("%u \n", x.valeur_absolue);
}

int main() {
  struct nombre nb1 = {0, 1255445}, nb2 = nb1;     nb2.signe = 1;
  // ...
  printf("nb1= "); print_nombre(nb1);
  printf("nb2= "); print_nombre(nb2);
  // ...
  nb1.signe += 2;  printf("nb1= ");   print_nombre(nb1);
  // ...
  return 0;
}
```

```
$> gcc champ-bits.c
$> ./a.out
nb1= +1255445
nb2= -1255445
nb1= +1255445
```

Exercice

Écrire un programme qui permet de découper un entier 32 bits en 4 octets (8 bits).

Exercice

Écrire un programme qui permet de découper un entier 32 bits en 4 octets (8 bits).

```
union decoupage
{
  int i;
  struct octets
  {
    int octet1 : 8;
    int octet2 : 8;
    int octet3 : 8;
    int octet4 : 8;
  } o;
};

int main() {
  union decoupage nb;
  nb.i = -1593770189;       // 10100001|00000000|11111111|00110011
                            //      -95        0       -1       51
  // ...
  printf("nombre = %d|%d|%d|%d \n", nb.o.octet1, nb.o.octet2,
                                    nb.o.octet3, nb.o.octet4);
  // ...
  return 0;
}
```

Exercice

Écrire un programme qui permet de découper un entier 32 bits en 4 octets (8 bits).

```
union decoupage
{
  int i;
  struct octets
  {
    int octet1 : 8;
    int octet2 : 8;
    int octet3 : 8;
    int octet4 : 8;
  } o;
};

int main() {
  union decoupage nb;
  nb.i = -1593770189;       // 10100001|00000000|11111111|00110011
                            //      -95        0       -1       51
  // ...
  printf("nombre = %d|%d|%d|%d \n", nb.o.octet1, nb.o.octet2,
                                    nb.o.octet3, nb.o.octet4);
  // ...
  return 0;
}
```

```
$> gcc exercice1.c
$> ./a.out
nombre = 51|-1|0|-95
```

Les autres types dérivés

Qu'est-ce qu'une énumération ?

- En langage C, une énumération permet de définir un type pour des variables ne pouvant prendre qu'un nombre fini de valeurs.
- La définition d'une énumération (ou type énuméré) suit la syntaxe suivante.

```
enum nom_type {val_1, val_2, val_3, ..., val_n};
```

- Les valeurs représentant `val1`, `val2`, ... ne sont rien d'autres que des valeurs entières allant de 0 à $n-1$.

Qu'est-ce qu'une énumération ?

- En langage C, une énumération permet de définir un type pour des variables ne pouvant prendre qu'un nombre fini de valeurs.
- La définition d'une énumération (ou type énuméré) suit la syntaxe suivante.

```
enum nom_type {val_1, val_2, val_3, ..., val_n};
```

- Les valeurs représentant `val1`, `val2`, ... ne sont rien d'autres que des valeurs entières allant de 0 à $n-1$.

- On peut modifier la valeur entière représentant chaque valeur de la liste, en utilisant la syntaxe suivante :

```
enum nom_type {val_1 = 1, val_2 = 4, val_3, ..., val_n};
```

  - dans ce cas, `val3` vaut 5.

Exemple d'utilisation d'une énumération

```
enum booleen {false, true};
enum tmp {val1 = 17, val2};

void test(enum booleen val)
{
  if(val == true)        printf("--> vrai \n");
  else if(val == false) printf("--> faux \n");
  else                   printf("--> inconnu \n");
}

int
main()
{
  enum booleen param = true;
  test(param);
  // ...
  param += 1;
  test(param);
  // ...
  printf("\n--> tmp(val2) = %d \n", val2);
  return 0;
}
```

Exemple d'utilisation d'une énumération

```c
enum booleen {false, true};
enum tmp {val1 = 17, val2};

void test(enum booleen val)
{
  if(val == true)        printf("--> vrai \n");
  else if(val == false) printf("--> faux \n");
  else                   printf("--> inconnu \n");
}

int
main()
{
  enum booleen param = true;
  test(param);
  // ...
  param += 1;
  test(param);
  // ...
  printf("\n--> tmp(val2) = %d \n", val2);
  return 0;
}
```

```
$> gcc enumeration.c
$> ./a.out
--> vrai
--> inconnu

--> tmp(val2) = 18
```

Les autres types dérivés

Comment redéfinir un type ?

- En langage C, il est possible de redéfinir un type, c'est-à-dire, de lui donner un nouveau nom :
  - ceci permet l'alléger l'écriture de certains programmes.
- La définition d'un nouveau type se fait en utilisant l'instruction `typedef` de la manière suivante.

```
typedef type nouveau_nom;
```

- Ensuite, `nouveau_nom` peut être utilisé en lieu et place de `type`.

```
#include <stdio.h>

typedef unsigned long int ulint;

int main(){
  ulint a = 17;                  // unsigned long int a = 17;
  printf("-> a = %lu \n", a);    // -> a = 17
  return 0;
}
```

Exemple d'utilisation de la redéfinition de type

```c
#include <stdio.h>

struct complexe
{
  float i, r;
};
typedef struct complexe complexe_t;

int main()
{
  complexe_t pt;
  pt.i = 1.7f;
  pt.r = 1.34f;
  // ...
  printf("Complexe = %f + %f * i \n", pt.r, pt.i);
  // ...
  return 0;
}
```

Exemple d'utilisation de la redéfinition de type

```c
#include <stdio.h>

struct complexe
{
  float i, r;
};
typedef struct complexe complexe_t;

int main()
{
  complexe_t pt;
  pt.i = 1.7f;
  pt.r = 1.34f;
  // ...
  printf("Complexe = %f + %f * i \n", pt.r, pt.i);
  // ...
  return 0;
}
```

```
$> gcc typedef.c
$> ./a.out
Complexe = 1.340000 + 1.700000 * i
```

Exemple d'utilisation de la redéfinition compacte de type

```c
#include <stdio.h>

typedef struct /* complexe */
{
  float i, r;
} complexe_t;

int main()
{
  complexe_t pt;
  pt.i = 1.7f;
  pt.r = 1.34f;
  // ...
  printf("Complexe = %f + %f * i \n", pt.r, pt.i);
  // ...
  return 0;
}
```

Exemple d'utilisation de la redéfinition compacte de type

```c
#include <stdio.h>

typedef struct /* complexe */
{
  float i, r;
} complexe_t;

int main()
{
  complexe_t pt;
  pt.i = 1.7f;
  pt.r = 1.34f;
  // ...
  printf("Complexe = %f + %f * i \n", pt.r, pt.i);
  // ...
  return 0;
}
```

```
$> gcc typedef-compact.c
$> ./a.out
Complexe = 1.340000 + 1.700000 * i
```

Que va-t-on voir dans ce cours ?

(1) Les éléments de base du C

(2) Les structures de contrôle

(3) Les tableaux et les structures

(4) Les pointeurs et l'allocation dynamique

(5) Les fonctions

(6) Les autres types dérivés

(7) Les fichiers

(8) Quelques éléments avancés

(9) Complément 1 : Les listes chaînées

(10) Complément 2 : Les piles et les files

Les fichiers

Les fichiers

1. Manipulation d'un fichier

2. Fichiers binaires

3. Positionnement dans un fichier

Les fichiers

Pourquoi et comment utliser un fichier ?

- Un programme peut avoir besoin de manipuler un grand nombre de données : cela se fait généralement au travers l'utilisation de fichiers :
  - pour lire de données : textes, images, sons, ...
  - pour écrire des données : sauvegardes de résultats, ...

- En langage C, la manipulation de fichiers se fait via un pointeur particulier, sur un objet de type `FILE`

```
FILE* fichier
```

  - ici la variable `fichier` contiendra l'adresse en mémoire du début du fichier.

Ouverture et fermeture d'un fichier

- En langage C, l'ouverture d'un fichier se fait avec la fonction `fopen`

```
FILE* fichier = fopen(char* nom, char* mode)
```

où

  - `nom` est le chemin relatif vers le fichier (à partir du répertoire de l'exécutable) ou le chemin absolu vers le fichier,
  - et `mode` est le mode d'ouverture du fichier.

mode	traitement	signification
`"r"`	lecture	si le fichier existe, il est ouvert
`"w"`	écriture	si le fichier existe, il est écrasé, sinon, il est créé
`"a"`	écriture en fin	si le fichier existe, l'écriture se fait en fin, sinon, il est créé

Ouverture et fermeture d'un fichier

- En langage C, l'ouverture d'un fichier se fait avec la fonction `fopen`

```
FILE* fichier = fopen(char* nom, char* mode)
```

où

  - `nom` est le chemin relatif vers le fichier (à partir du répertoire de l'exécutable) ou le chemin absolu vers le fichier,
  - et `mode` est le mode d'ouverture du fichier.

mode	traitement	signification
`"r"`	lecture	si le fichier existe, il est ouvert
`"w"`	écriture	si le fichier existe, il est écrasé, sinon, il est créé
`"a"`	écriture en fin	si le fichier existe, l'écriture se fait en fin, sinon, il est créé

- La fonction `fopen` renvoie NULL si le fichier n'a pas pu être ouvert.
- Tout fichier dont l'ouverture a réussi doit être fermé avec la fonction `fclose`.

```
int fclose(FILE* fichier)
```

Écriture dans un fichier

- En langage C, l'écriture dans un fichier se fait en utilisant la fonction `fprintf`.
- Son utilisation est très similaire à la fonction `printf`

```
fprintf(file, expression, var1, var2, ...)
```

où

  - `file` est le fichier dans lequel on souhaite écrire des données,
  - `expression` est une chaîne de caractères contenant l'emplacement des variables,
  - et `var1`, `var2`, ... est la liste des variables à écrire.

Écriture dans un fichier

- En langage C, l'écriture dans un fichier se fait en utilisant la fonction `fprintf`.
- Son utilisation est très similaire à la fonction `printf`

```
fprintf(file, expression, var1, var2, ...)
```

où

- `file` est le fichier dans lequel on souhaite écrire des données,
- `expression` est une chaîne de caractères contenant l'emplacement des variables,
- et `var1`, `var2`, ... est la liste des variables à écrire.

```c
int main()
{
  int a = 17;
  FILE* fic = fopen("tmp.txt", "w"); // ouverture du fichier 'tmp.txt' en ecriture
  if(fic != NULL) {
    fprintf(fic, "%d", a);           // ecriture de '17' dans le fichier 'tmp.txt'
    fclose(fic);                     // fermeture du fichier 'tmp.txt'
  }
  return 0;
}
```

Lecture dans un fichier

- En langage C, la lecture dans un fichier se fait en utilisant la fonction `fscanf`.
- Son utilisation est très similaire à la fonction `scanf`

```
fscanf(file, formats, &var1, &var2, ...)
```

où

  - `file` est le fichier dans lequel on souhiate lire les données,
  - `formats` est chaîne de caractères contenant les formats des variables à lire,
  - et `var1`, `var2`, ... est la liste des variables dont la valeur doit être lue dans le fichier.

Lecture dans un fichier

- En langage C, la lecture dans un fichier se fait en utilisant la fonction `fscanf`.
- Son utilisation est très similaire à la fonction `scanf`

```
fscanf(file, formats, &var1, &var2, ...)
```

où

- `file` est le fichier dans lequel on souhiate lire les données,
- `formats` est chaîne de caractères contenant les formats des variables à lire,
- et `var1`, `var2`, ... est la liste des variables dont la valeur doit être lue dans le fichier.

```c
int main()
{
  int a;
  FILE* fic = fopen("tmp.txt", "r"); // ouverture du fichier 'tmp.txt' en lecture
  if(fic != NULL) {
    fscanf(fic, "%d", &a);           // lecture de 'a' dans le fichier 'tmp.txt'
    fclose(fic);                     // fermeture du fichier 'tmp.txt'
  }
  return 0;
}
```

Exemple de manipulation de fichier

```c
int
main()
{
  double m[2][2], tmp;
  FILE* fichier;
  int i = 0, j;
  // ...                                                    // matrice.txt
  fichier = fopen("matrice.txt", "r");                      //
  if(fichier != NULL) {                                     // 1.000000000
    while(fscanf(fichier, "%lf", &tmp) != EOF) {            // 1.500000000
      m[i/2][i%2] = tmp;                                    // 2.000000000
      i ++;                                                 // 2.500000000
    }
    fclose(fichier);
    // ...
    for(i = 0; i < 2; i++) {
      for(j = 0; j < 2; j++)
        printf("%1.10lf ", m[i][j]);
      printf("\n");
    }
  }
  return 0;
}
```

Exemple de manipulation de fichier

```c
int
main()
{
  double m[2][2], tmp;
  FILE* fichier;
  int i = 0, j;
  // ...                                                    // matrice.txt
  fichier = fopen("matrice.txt", "r");                      //
  if(fichier != NULL) {                                     // 1.000000000
    while(fscanf(fichier, "%lf", &tmp) != EOF) {            // 1.500000000
      m[i/2][i%2] = tmp;                                    // 2.000000000
      i ++;                                                 // 2.500000000
    }
    fclose(fichier);
    // ...
    for(i = 0; i < 2; i++) {
      for(j = 0; j < 2; j++)
        printf("%1.10lf ", m[i][j]);
      printf("\n");
    }
  }
  return 0;
}
```

```
$> gcc fichier.c
$> ./a.out
1.0000000000 1.5000000000
2.0000000000 2.5000000000
```

Les fichiers

Intérêt des fichiers binaires

- Jusqu'à maintenant, les fichiers ont été manipulés au format texte :
  - les données sont enregistrées au format ASCII,
  - on peut gaspiller de la mémoire,
  - on doit connaitre le format des données pour pouvoir les relire.

- Lorsqu'on a des données numériques, il peut être plus intéressant de les stocker sous forme binaire, c'est-à-dire, de recopier directement le contenu de la zone mémoire leur correspondant :

3.141592741012573242187 5

$$\begin{cases} 4 \text{ octets} & \text{au format } \texttt{float}, \\ 24 \text{ octets} & \text{au format ASCII (1 par caractère).} \end{cases}$$

Intérêt des fichiers binaires

- Jusqu'à maintenant, les fichiers ont été manipulés au format texte :
  - les données sont enregistrées au format ASCII,
  - on peut gaspiller de la mémoire,
  - on doit connaitre le format des données pour pouvoir les relire.

- Lorsqu'on a des données numériques, il peut être plus intéressant de les stocker sous forme binaire, c'est-à-dire, de recopier directement le contenu de la zone mémoire leur correspondant :

$$3.141592741012573242187 5 \begin{cases} 4 \text{ octets} & \text{au format } \texttt{float}, \\ 24 \text{ octets} & \text{au format ASCII (1 par caractère).} \end{cases}$$

- En langage C, l'ouverture et la fermeture d'un fichier binaire se font de la même manière qu'un fichier texte :
  - `"rb"` : lecture binaire,
  - `"wb"` : écriture binaire,
  - `"ab"` : écriture binaire en fin de fichier.

Écriture dans un fichier binaire

- En langage C, l'écriture dans un fichier binaire se fait avec la fonction `fwrite`.
- Elle permet d'écrire un bloc de données en un seul appel

```
fwrite(src, taille_elt, nb_elt, fichier)
```

où

  - `src` est un pointeur sur les données à écrire,
  - `taille_elt` est la taille d'une donnée,
  - `nb_elt` est le nombre de données à écrire,
  - et `fichier` est le fichier dans lequel on souhaite écrire des données.

- La fonction `fwrite` renvoie le nombre d'éléments effectivement écrits.

Écriture dans un fichier binaire

- En langage C, l'écriture dans un fichier binaire se fait avec la fonction `fwrite`.
- Elle permet d'écrire un bloc de données en un seul appel

```
fwrite(src, taille_elt, nb_elt, fichier)
```

où

  - `src` est un pointeur sur les données à écrire,
  - `taille_elt` est la taille d'une donnée,
  - `nb_elt` est le nombre de données à écrire,
  - et `fichier` est le fichier dans lequel on souhaite écrire des données.

- La fonction `fwrite` renvoie le nombre d'éléments effectivement écrits.

```c
int main()
{
  int a[4] = {17, 18, 19, 20}, ret = 0;
  FILE* fic = fopen("tmp.bin", "wb"); // ouverture du fichier binaire 'tmp.bin' en
  if(fic != NULL) {                   //    ecriture
    ret = fwrite(a, sizeof(int), 4, fic);    // ecriture des donnees dans 'tmp.bin
    printf("Nombre d'elements : %d \n", ret); // --> Nombre d'elements : 4
    fclose(fic);                              // fermeture du fichier 'tmp.bin'
  }
  return 0;
}
```

Lecture dans un fichier binaire

- En langage C, la lecture dans un fichier binaire se fait avec la fonction `fread`.
- Elle permet de lire un bloc de données en un seul appel

```
fread(dst, taille_elt, nb_elt, fichier)
```

où

  - `dst` est l'adresse de la mémoire où l'on souhaite stocker les données,
  - `taille_elt` est la taille d'une donnée,
  - `nb_elt` est le nombre de données à lire,
  - et `fichier` est le fichier duquel on souhaite lire des données.

- La fonction `fread` renvoie le nombre d'éléments effectivement lus.

Lecture dans un fichier binaire

- En langage C, la lecture dans un fichier binaire se fait avec la fonction `fread`.
- Elle permet de lire un bloc de données en un seul appel

```
fread(dst, taille_elt, nb_elt, fichier)
```

où

  - `dst` est l'adresse de la mémoire où l'on souhaite stocker les données,
  - `taille_elt` est la taille d'une donnée,
  - `nb_elt` est le nombre de données à lire,
  - et `fichier` est le fichier duquel on souhaite lire des données.

- La fonction `fread` renvoie le nombre d'éléments effectivement lus.

```c
int main()
{
  int a[4], ret = 0, i;
  FILE* fic = fopen("tmp.bin", "rb"); // ouverture du fichier binaire 'tmp.bin en
  if(fic != NULL) {                   //     lecture
    ret = fread(a, sizeof(int), 4, fic);    // lecture des donnees dans 'tmp.bin'
    printf("Nombre d'elements : %d \n", ret); // --> Nombre d'elements : 4
    for(i = 0; i < ret; i++)
      printf("%d ", a[i]);                    // 17 18 19 20
    fclose(fic);                              // fermeture du fichier 'tmp.bin
  }
  return 0;
}
```

Les fichiers

Accès en mode direct

- Les fonctions d'E/S précédentes permettent d'accéder aux fichiers en mode séquentiel : les données sont lues/écrites les unes à la suite des autres.
- Le mode direct permet de se positionner à un certain endroit dans le fichier, et d'accéder aux données présentes à cet endroit.

Accès en mode direct

- Les fonctions d'E/S précédentes permettent d'accéder aux fichiers en mode séquentiel : les données sont lues/écrites les unes à la suite des autres.
- Le mode direct permet de se positionner à un certain endroit dans le fichier, et d'accéder aux données présentes à cet endroit.

- Pour se positionner dans un fichier, on utilise la fonction `fseek`

```
int fseek(FILE* fichier, long offset, int position)
```

où

  - `fichier` est le fichier que l'on manipule,
  - `offset` est la nouvelle position dans le fichier,
  - et `position` est une constante $\in$ {`SEEK_SET`, `SEEK_CUR`, `SEEK_END`} indiquant l'origine du déplacement (début, position courante ou fin, respectivement).

Exemple d'utilisation du mode direct

```c
int main()
{
  int a[4] = {17, 18, 19, 20}, b[2], i, ret = 0;
  // ...
  FILE* fic = fopen("tmp.bin", "wb");
  if(fic != NULL) {
    ret = fwrite(a, sizeof(int), 4, fic);
    fclose(fic);
  }
  // ...
  fic = fopen("tmp.bin", "rb");
  if(fic != NULL) {
    fseek(fic, -2*sizeof(int), SEEK_END);  // deplacement de 2*4 octets,
                                           //    en partant de la fin du
                                           //    fichier
    ret = fread(b, sizeof(int), 2, fic);
    printf("Nombre d'elements lus : %d \n", ret);
    for(i = 0; i < ret; i++)
      printf("%d ", b[i]);
    printf(" \n");
    fclose(fic);
  }
  return 0;
}
```

Exemple d'utilisation du mode direct

```c
int main()
{
  int a[4] = {17, 18, 19, 20}, b[2], i, ret = 0;
  // ...
  FILE* fic = fopen("tmp.bin", "wb");
  if(fic != NULL) {
    ret = fwrite(a, sizeof(int), 4, fic);
    fclose(fic);
  }
  // ...
  fic = fopen("tmp.bin", "rb");
  if(fic != NULL) {
    fseek(fic, -2*sizeof(int), SEEK_END);  // deplacement de 2*4 octets,
                                           //    en partant de la fin du
                                           //    fichier
    ret = fread(b, sizeof(int), 2, fic);
    printf("Nombre d'elements lus : %d \n", ret);
    for(i = 0; i < ret; i++)
      printf("%d ", b[i]);
    printf(" \n");
    fclose(fic);
  }
  return 0;
}
```

```
$> gcc positionnement.c
$> ./a.out
Nombre d'elements lus : 2
19 20
```

Que va-t-on voir dans ce cours ?

(1) Les éléments de base du C

(2) Les structures de contrôle

(3) Les tableaux et les structures

(4) Les pointeurs et l'allocation dynamique

(5) Les fonctions

(6) Les autres types dérivés

(7) Les fichiers

(8) Quelques éléments avancés

(9) Complément 1 : Les listes chaînées

(10) Complément 2 : Les piles et les files

Quelques éléments avancés

Quelques éléments avancés

Quelques éléments avancés

Qu'est-ce qu'une opération bit à bit ?

- Une opération bit à bit est une opération qui s'applique à chaque bit de ses opérandes.
- Le langage C offre les opérations bit à bit sur les entiers suivantes :
  - `&` (et), `|` (ou), `~` (non), `^` (xor, ou exclusif),
  - `<<` (décalage à gauche), et `>>` (décalage à droite).

```
int
main(void)
{
  unsigned char X = 57, Y = 19;
  print("X =       ", X);        // affichage l'ecriture binaire de X
  print("Y =       ", Y);
  printf("-----------------\n");
  print("X | Y =   ", X | Y);
  print("X & Y =   ", X & Y);
  print("~X =      ", ~X);
  print("X ^ Y =   ", X ^ Y);
  printf("-----------------\n");
  print("X << 2 = ", X << 2); // les 2 bits de poids fort sont perdus
  print("X >> 2 = ", X >> 2); // en arithmetique non signee, les 2 bits de poids
                              // forts sont remplaces par des '0', et les 2 bits
                              // de poids faible sont perdus
  printf("-----------------\n");
}
```

Exemple 1/3 : et, ou, non et ou exclusif

```c
int
main(void)
{
  unsigned char X = 57, Y = 19;
  print("X =       ", X);        // affichage l'ecriture binaire de X
  print("Y =       ", Y);
  printf("-----------------\n");
  print("X | Y =   ", X | Y);
  print("X & Y =   ", X & Y);
  print("~X =      ", ~X);
  print("X ^ Y =   ", X ^ Y);
  return 0;
}
```

```
$> gcc operation-bit-a-bit.c
$> ./a.out
X =       00111001
Y =       00010011
-----------------
X | Y =   00111011
X & Y =   00010001
~X =      11000110
x ^ Y =   00101010
```

Exemple 2/3 : décalage en arithmétique non signée

```c
int
main(void)
{
  unsigned char X = 57, Y = 19;
  print("X =      ", X);        // affichage l'ecriture binaire de X
  print("Y =      ", Y);
  printf("-----------------\n");
  print("X << 2 = ", X << 2); // les 2 bits de poids fort sont perdus
  print("X >> 2 = ", X >> 2); // en arithmetique non signee, les 2 bits de poids
                              // forts sont remplaces par des '0', et les 2 bits
                              // de poids faible sont perdus

  return 0;
}
```

```
$> gcc operation-bit-a-bit.c
$> ./a.out
X =      00111001
Y =      00010011
-----------------
X << 2 = 11100100
X >> 2 = 00001110
```

Exemple 3/3 : décalage en arithmétique signée

```c
int
main(void)
{
  char P = 57, N = -19;
  print("P (+)  = ", P);
  print("N (-)  = ", N);
  printf("-----------------\n");
  print("P >> 2 = ", P >> 2); // en arithmetique signee, les 2 bits de poids
  print("N >> 2 = ", N >> 2); // forts sont remplaces par des bits de signes :
                             //   on parle d'extension de signe
  return 0;
}
```

```
$> gcc operation-bit-a-bit.c
$> ./a.out
P (+)  = 00111001
N (-)  = 11101101
-----------------
P >> 2 = 00001110
N >> 2 = 11111011
```

Exercice

Écrire une fonction qui affiche dans un terminal l'écriture binaire d'un entier 8 bits.

Exercice

Écrire une fonction qui affiche dans un terminal l'écriture binaire d'un entier 8 bits.

```c
void
print_int_8b(char I)
{
  int i;
  for(i = 0; i < 8; i++)
    printf("%d", (I >> (7 - i)) & 1);
  printf("\n");
}
```

Quelques éléments avancés

Qu'est-ce qu'un pointeur de fonctions ?

Un pointeur de fonctions est un pointeur dont la valeur est l'adresse du code exécutable d'une fonction. Son utilisation se fait de la manière suivant.

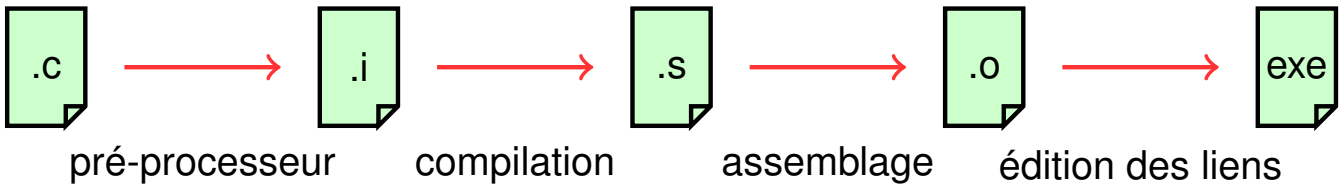
```c
int max(int a, int b) { return (a > b ? a : b); }

int min(int a, int b) { return (a < b ? a : b); }

int
main(void)
{
  int (*ptr_fct)(int, int);
  ptr_fct = &max;
  printf("max(%d, %d) = %d\n", 3, 4, (*ptr_fct)(3, 4));
  ptr_fct = &min;
  printf("min(%d, %d) = %d\n", 3, 4, (*ptr_fct)(3, 4));

  return 0;
}
```

- Remarques importantes :
  - le pointeur `ptr_fct` pointe vers une fonction qui prend deux `int` en paramètre, et retourne un `int`,
  - lors de l'appel, le pointeur est déférérencé : `*ptr_fct` doit être mis en parenthèse.

Quelques éléments avancés

Comment marche la compilation ?

- La compilation (avec `gcc` ou `clang`) se décompose en 4 étapes principales :



  - le pré-processeur,
  - la compilation proprement dite,
  - l'assemblage,
  - et l'édition des liens.

Focus sur le pré-processeur

- Le pré-processeur (ou pré-traitement) permet de transformer le code source C initial en un autre code source C dans lequel :
  - les directives de pré-traitement ont été supprimées,
  - les fichiers d'entête ont été inclus,
  - les macros ont été expansés,
  - et les constantes ont été propagées.

- Une directive de pré-traitement commence par le caractère `#`, comme par exemple :
  - l'inclusion des fichiers d'en-tête : `#include <stdio.h>`
  - la déclaration de constantes et des macros : `#define PI 3.14`

- Le résultat du pré-traitement peut être obtenu avec l'option de compilation `-E`.

Exemple de définition de constantes et de macros

```c
// definition de la constante PI de valeur 3.14
#define PI 3.14

// definition de la macro MIN(a, b) qui determine le min de deux elements
#define MIN(a, b) ((a)<(b)?(a):(b))

int
main(void)
{
  float x = PI, y = CST;           // CST n'est pas defini dans le fichier
  printf("x= %f, y= %f\n", x, y); //  ... elle pourra l'etre sur la ligne
  printf("min= %d\n", MIN(1, 2)); //  de commande en utilisant l'option -D

  return 0;
}
```

```
$> gcc -E constante-macro.c -DCST=37
...
int
main()
{
  float x = 3.14, y = 37;
  printf("x= %f, y= %f\n", x, y);
  printf("min= %d\n", ((1)<(2)?(1):(2)));

  return 0;
}
```

```
$> gcc constante-macro.c -DCST=37
$> ./a.out
x= 3.140000, y= 37.000000
min= 1
```

Et maintenant, la compilation séparée

- L'intérêt de la compilation séparée est de rendre :
  - la programmation est modulaire, donc plus compréhensible,
  - le code plus lisible en le séparant en plusieurs fichiers,
  - la maintenance plus facile car seuls les fichiers sources modifiés seront recompilés.

- La compilation séparée pourra être automatiser avec des outils comme `make`.

Exemple de compilation séparée

```c
#ifndef __FOO_H__
#define __FOO_H__

void foo(int);

#endif // __FOO_H__
```

```c
#include <stdio.h>

void foo(int);

int
main(void)
{
  foo(17);
  return 0;
}

void
foo(int a)
{
  printf("... %d\n", a);
}
```

```c
#include <stdio.h>
// dans le repertoire courant
//   --> #include "..."
#include "foo.h"

void
foo(int a)
{
  printf("... %d\n", a);
}
```

```c
#include <stdio.h>
#include "foo.h"

int
main(void)
{
  foo(17);
  return 0;
}
```

Et comment on compile cela ?

```c
#ifndef __FOO_H__
#define __FOO_H__

void foo(int);

#endif // __FOO_H__
```

```c
#include <stdio.h>
// dans le repertoire courant
//   --> #include "..."
#include "foo.h"

void
foo(int a)
{
  printf("... %d\n", a);
}
```

```c
#include <stdio.h>
#include "foo.h"

int
main(void)
{
  foo(17);
  return 0;
}
```

```
$> ls
foo.c  foo.h  main-foo.c
$> gcc -c foo.c
$> ls
foo.c  foo.h  foo.o  main-foo.c
$> gcc -c main-foo.c
$> ls
foo.c  foo.h  foo.o  main-foo.c
main-foo.o
$> gcc -o exe foo.o main-foo.o
$> ./exe
... 17
```

- Chaque fichier source est compilé en utilisant l'option `-c`.
- Ensuite, on effectue l'édition de liens pour produire l'exécutable exe

Quelques éléments avancés

Format d'un fichier makefile

- L'intérêt d'un fichier makefile est :
  - qu'il assure la compialtion séparée,
  - qu'iil utilise des macro-commandes et des variables,
  - qu'il permet de ne recompiler que le code modifié,
  - et qu'il permet d'utiliser des commandes shell.

Format d'un fichier makefile

- L'intérêt d'un fichier makefile est :
  - qu'il assure la compialtion séparée,
  - qu'iil utilise des macro-commandes et des variables,
  - qu'il permet de ne recompiler que le code modifié,
  - et qu'il permet d'utiliser des commandes shell.

- Un fichier makefile (nommé généralement `makefile` ou `Makefile`) est composé d'un ensemble de règles, chacune ayant la forme suivante.

```
regle1: dependance1 dependance2 <...>
<tabulation> commande1
```

- Ici
  - `regle1` : généralement le nom du fichier que l'on veut créer avec la commande `commande1`,
  - `dependance1`, `dependance2`, ... : nom des fichiers ou des règles utilisés pour générer le fichier `regle1`,
  - `commande1` : commande (de compilation) pour générer `regle1`.

Exemple de fichier makefile

```
# Definition de deux variables CC et EXE
CC=gcc
EXE=exe

# Regle par defaut
default: $(EXE)

# Ensemble de regles
foo.o: foo.c foo.h
	$(CC) -c foo.c

main-foo.o: main-foo.c foo.h
	$(CC) -c main-foo.c

$(EXE): foo.o main-foo.o
	$(CC) -o $(EXE) foo.o main-foo.o

clean:
	rm -f *~ *.o $(EXE)
```

```
$> make foo.o
gcc -c foo.c
$> make exe
gcc -c main-foo.c
gcc -o exe foo.o main-foo.o
$> ./exe
... 17
```

```
$> make clean
rm -f *~ *.o exe
$> make
gcc -c foo.c
gcc -c main-foo.c
gcc -o exe foo.o main-foo.o
```

Quelques éléments avancés

Méthode avec la macro `assert`

- La macro `assert` prend une expression en paramètre : si cette expression est fausse (ou égale à 0), le programme est interrompu, avec un message indiquant l'expression en argument de la macro, la fonction, le fichier et le numéro de ligne.

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

int
main(void)
{
  int tab[10], i;
  for(i = 0; i <= 10; i++){
    assert(i < 10 && "Depassement de tableau");
    tab[i] = i;
  }
  return 0;
}
```

```
$> gcc assert.c
$> ./a.out
Assertion failed: (i < 10 && "Depassement de tableau"), function main,
     file assert.c, line 9.
Abort trap: 6
```

L'outil GDB

- L'outil GDB est un débogeur interactif. Il permet de :
  - interrompre et reprendre l'exécution d'un programme,
  - suivre l'exécution d'un programme pas à pas,
  - poser des points d'arrêts,
  - inspecter le contenu des variables,
  - connaître la ligne exacte et le contenu des variables lors d'une erreur de segmentation, par exemple.

- Pour pouvoir être inspecter avec GDB, le programme doit être compilé avec l'option de compilation `-g` :
  - lancer GDB : `gdb program`
  - inspecter le programme : `run arg1 arg2 ...`

Exemple d'utilisation de GDB (1/4)

```c
int
ret(int n, int i)
{
  if((n % i) == 0)
    return i;
  else
    return 0;
}

int
main(void)
{
  int n, i;
  int somme = 0;

  printf("Lire la valeur n= ");
  scanf("%d", &n);

  i = n;
  while(i >= 0)
    somme += ret(n, i --);

  // ...
  return 0;
}
```

- On compile le fichier source et on l'inspecte avec GDB : `run`

```
$> gcc -g exemple-gdb.c
$> gdb ./a.out
...
Reading symbols from /home/grevy/a.out...done.
(gdb) run
Starting program: /home/grevy/a.out
Lire la valeur n= 6

Program received signal SIGFPE, Arithmetic
    exception.
0x0000000000400585 in ret (n=6, i=0) at exemple
    -gdb.c:7
7         if((n % i) == 0)
```

Exemple d'utilisation de GDB (2/4)

```c
int
ret(int n, int i)
{
  if((n % i) == 0)
    return i;
  else
    return 0;
}

int
main(void)
{
  int n, i;
  int somme = 0;

  printf("Lire la valeur n= ");
  scanf("%d", &n);

  i = n;
  while(i >= 0)
    somme += ret(n, i --);

  // ...
  return 0;
}
```

- Maintenant, on exécute notre programme pas à pas : `break`, `next` et `print`

```
...
(gdb) break exemple-gdb.c:24
Breakpoint 1 at 0x4005d6: file exemple-gdb.c,
     line 24.
(gdb) run
Starting program: /home/grevy/a.out
Lire la valeur n= 6

Breakpoint 1, main () at exemple-gdb.c:24
24            somme += ret(n, i --);
(gdb) next
23          while(i >= 0)
(gdb) print somme
$1 = 6
(gdb) next

Breakpoint 1, main () at exemple-gdb.c:24
24            somme += ret(n, i --);
(gdb) next
23          while(i >= 0)
(gdb) print i
$2 = 4
(gdb) ...
```

Exemple d'utilisation de GDB (3/4)

```
int
ret(int n, int i)
{
  if((n % i) == 0)
    return i;
  else
    return 0;
}

int
main(void)
{
  int n, i;
  int somme = 0;

  printf("Lire la valeur n= ");
  scanf("%d", &n);

  i = n;
  while(i >= 0)
    somme += ret(n, i --);

  // ...
  return 0;
}
```

- Et on descend dans les fonctions appelées : `step`

```
...
(gdb) next

Breakpoint 1, main () at exemple-gdb.c:24
24          somme += ret(n, i --);
(gdb) next
23        while(i >= 0)
(gdb) print i
$2 = 3
(gdb) step

Breakpoint 1, main () at exemple-gdb.c:24
24          somme += ret(n, i --);
(gdb) step
ret (n=6, i=3) at exemple-gdb.c:7
7         if((n % i) == 0)...
(gdb) step
10          return i;
(gdb) step
11        }
(gdb) step
main () at exemple-gdb.c:23
23        while(i >= 0)
(gdb) ...
```

Exemple d'utilisation de GDB (4/4)

- On peut même afficher plein d'informations `list` et `backtrace`

```
int
ret(int n, int i)
{
  if((n % i) == 0)
    return i;
  else
    return 0;
}

int
main(void)
{
  int n, i;
  int somme = 0;

  printf("Lire la valeur n= ");
  scanf("%d", &n);

  i = n;
  while(i >= 0)
    somme += ret(n, i --);

  // ...
  return 0;
}
```

```
...
(gdb) list
18
19          printf("Lire la valeur n= ");
20          scanf("%d", &n);
21
22          i = n;
23          while(i >= 0)
24            somme += ret(n, i --);
25
26          if(n == somme)
27            printf("Le nombre n=%d est parfait
      .\n", n);
(gdb) step

Breakpoint 1, main () at exemple-gdb.c:24
24            somme += ret(n, i --);
(gdb) step
ret (n=6, i=2) at exemple-gdb.c:7
7           if((n % i) == 0)
(gdb) backtrace
#0  ret (n=6, i=2) at exemple-gdb.c:7
#1  0x00000000004005eb in main () at exemple-
     gdb.c:24
(gdb) ...
```

L'outil Valgrind

- L'outil Valgrind permet est un outil de débogage et de profilage de codes. Il est composé d'un ensemble de modules, dont `memcheck`.
- Le module `memcheck` permet de vérifier les points suivants :
  - les valeurs et pointeurs utilisés sont initialisés,
  - les zones mémoire auxquelles on accède ne sont pas non allouées ou déjà libérées,
  - les zones mémoire qu'on libère n'ont pas déjà été libérées,
  - la mémoire allouée a effectivement été libérée, ...
- Pour pouvoir être inspecter avec Valgrind, le programme doit être compilé avec l'option de compilation `-g` et en niveau d'optimisation 0 (option `-O0`).

Exemple d'utilisation de Valgrind

```c
int
main(void)
{
  int i;
  char * tab = (char*)malloc(10*sizeof(char));

  for(i = 0; i <= 10 ; i++)
    tab[i] = (char)i;

  return 0;
}
```

```
$> gcc -g -O0 exemple-valgrind.c
$> valgrind --tool=memcheck a.out
...
==17362== Invalid write of size 1
==17362==    at 0x40055F: main (exemple-valgrind.c:9)
==17362==  Address 0x51f404a is 0 bytes after a block of size 10 alloc'd
==17362==    at 0x4C27BFD: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64
    -linux.so)
==17362==    by 0x400541: main (exemple-valgrind.c:7)
...
==17362== HEAP SUMMARY:
==17362==     in use at exit: 10 bytes in 1 blocks
==17362==   total heap usage: 1 allocs, 0 frees, 10 bytes allocated
==17362==
==17362== LEAK SUMMARY:
==17362==    definitely lost: 10 bytes in 1 blocks
...
```

Quelques éléments avancés

Arguments d'entrée

- Un programme C peut prendre un ensemble de valeurs en paramètre, passées sur la ligne de commande en forme de chaînes caractères.

- Ces paramètres sont récupérés dans le programme de la manière suivante.

```c
int
main(int argc, char **argv)
{
  // ...
}
```

  - `argc` : nombre de paramètres (incluant le nom de l'exécutable),
  - `argv` : tableau de chaînes de caractères, contenant la liste des paramètres.

Exemple de passage de paramètres

```c
int
main(int argc, char **argv)
{
  int i;
  for(i = 0; i < argc; i++)
    printf("argv[%d] = '%s'\n", i, argv[i]);

  return 0;
}
```

```
$> gcc exemple-arg.c
$> ./a.out
argv[0] = './a.out'
$> ./a.out 1 2 3
argv[0] = './a.out'
argv[1] = '1'
argv[2] = '2'
argv[3] = '3'
```

Dépassement de tampons (1/2)

```
int
main(int argc, char **argv)
{
  int ret = 0;
  char buffer[6];

  if (argc == 2)
    strcpy(buffer, argv[1]);

  printf("buffer = %s\n",buffer);
  printf("i      = %d\n",ret);

  if(ret == 0)
    printf("==> Oh !! Le resultat est normal...\n");
  else
    printf("==> Argh !! Le resultat est bizarre...\n");

  return 0;
}
```

Dépassement de tampons (1/2)

```c
int
main(int argc, char **argv)
{
  int ret = 0;
  char buffer[6];

  if (argc == 2)
    strcpy(buffer, argv[1]);

  printf("buffer = %s\n",buffer);
  printf("i      = %d\n",ret);

  if(ret == 0)
    printf("==> Oh !! Le resultat est normal...\n");
  else
    printf("==> Argh !! Le resultat est bizarre...\n");

  return 0;
}
```

```
$> gcc -O0 -g -fno-stack-protector depassement-tampon.c
$> ./a.out exemple1
buffer = exemple1
i      = 0
==> Oh !! Le resultat est normal...
```

Dépassement de tampons (2/2)

```c
int
main(int argc, char **argv)
{
  int ret = 0;
  char buffer[6];

  if (argc == 2)
    strcpy(buffer, argv[1]);

  printf("buffer = %s\n",buffer);
  printf("i      = %d\n",ret);

  if(ret == 0)
    printf("==> Oh !! Le resultat est normal...\n");
  else
    printf("==> Argh !! Le resultat est bizarre...\n");

  return 0;
}
```

```
$> gcc -O0 -g -fno-stack-protector depassement-tampon.c
$> ./a.out exemple1....E
buffer = exemple1....E
i      = 69
==> Argh !! Le resultat est bizarre...
```

Exécuter une fonction jamais appelée (1/2)

```
int
main (int argc, char **argv)
{
  if (argc == 2){
    printf ("Adresse de code_inutile           = %p\n", fct_inutile);
    printf ("Adresse de fonction_jamais_appelee = %p\n", fct_jamais_appelee);

    fct_inutile (argv[1]);
    printf ("  -> argv[1] copie : %s\n",argv[1]);
  }

  return 0;
}

void
fct_inutile (const char *s)
{
  if (s){
    char text[20];
    strcpy (text, s);
  }
}

void
fct_jamais_appelee (void)
{
  printf ("Argh !! Pourquoi le programme arrive ici !!\n");
}
```

Exécuter une fonction jamais appelée

```
$> gcc -O0 -g -fno-stack-protector fonction-jamais-appelee.c
$> ./a.out aaaabbbbccccddddeeee
Adresse de code_inutile             = 0x40063e
Adresse de fonction_jamais_appelee = 0x400667
  -> argv[1] copie : aaaabbbbccccddddeeee
```

Exécuter une fonction jamais appelée

```
$> gcc -O0 -g -fno-stack-protector fonction-jamais-appelee.c
$> ./a.out aaaabbbbccccddddeeee
Adresse de code_inutile             = 0x40063e
Adresse de fonction_jamais_appelee = 0x400667
  -> argv[1] copie : aaaabbbbccccddddeeee
```

```
$> gcc -O0 -g -fno-stack-protector fonction-jamais-appelee.c
$> ./a.out aaaabbbbccccddddeeee00000000000000000000$(echo -ne '\x53\x06\x40')
Adresse de code_inutile             = 0x40062a
Adresse de fonction_jamais_appelee = 0x400653
Argh !! Pourquoi le programme arrive ici !!
Erreur de segmentation (core dumped)
```

Que va-t-on voir dans ce cours ?

Complément 1 : Les listes chaînées

Complément 1 : Les listes chaînées

1. Liste chaînée simple

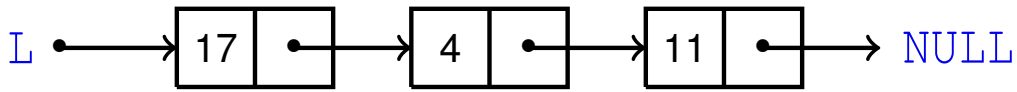
2. Liste chaînée circulaire

3. Liste doublement chaînée

Complément 1 : Les listes chaînées

Qu'est-ce qu'une liste chaînée ?

- Définition : Une liste chaînée est une structure de données qui contient un ensemble ordonné et de taille arbitraire d'éléments de même type.
- Plus particulièrement, elle est constituée d'un ensemble de cellules, modélisées par des structures, contenant les champs suivants :
  - une ou plusieurs données, comme dans n'importe quelle structure,
  - et un pointeur `next` vers la cellule suivante dans la liste.



- Remarques
  - on manipule la liste par un pointeur `L` sur la première cellule,
  - on accède à un élément (cellule) de la lise en parcourant les cellules une à une en partant de la cellule pointée par `L`, et en suivant les pointeurs `next`,
  - et le pointeur `next` de la dernière cellule pointe vers `NULL`.

Déclaration d'une liste chaînée

- Pour pouvoir créer et utiliser une liste chaînée, il faut déclarer une nouvelle structure de données, qui représentera une cellule.
  - elle peut contenir plusieurs données, de types différents

```
struct list
{
  int value;                    // donnee contenu dans la liste
  struct list *next;            // pointeur vers la cellule suivante dans la liste
};
```

Déclaration d'une liste chaînée

- Pour pouvoir créer et utiliser une liste chaînée, il faut déclarer une nouvelle structure de données, qui représentera une cellule.
  - elle peut contenir plusieurs données, de types différents

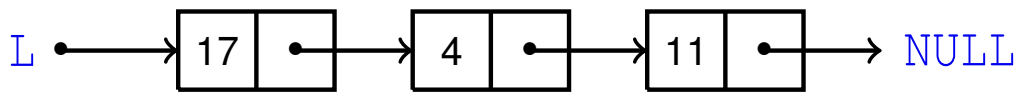
```
struct list
{
  int value;                    // donnee contenu dans la liste
  struct list *next;            // pointeur vers la cellule suivante dans la liste
};
```

- Pour utiliser une liste chaînée de ce type, il faut déclarer un pointeur vers la première cellule de la liste.
  - à sa création, la liste est vide (taille = 0)

```
int
main(void)
{
  struct list *L = NULL;        // creation d'une liste vide
  // ...
}
```

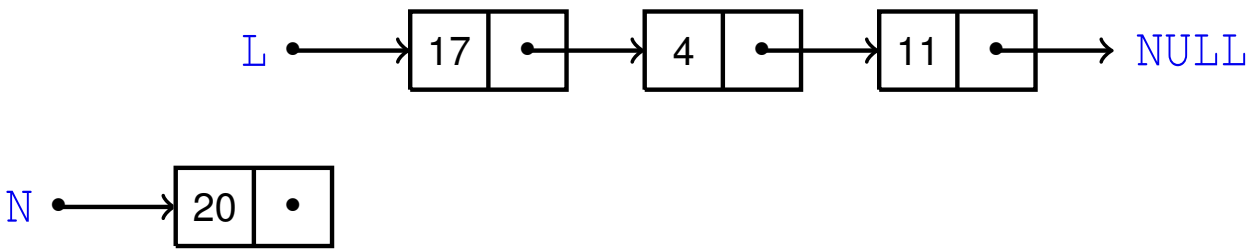
Insertion d'un élément en tête de liste

- Pour insérer un élément en tête de liste, il faut :



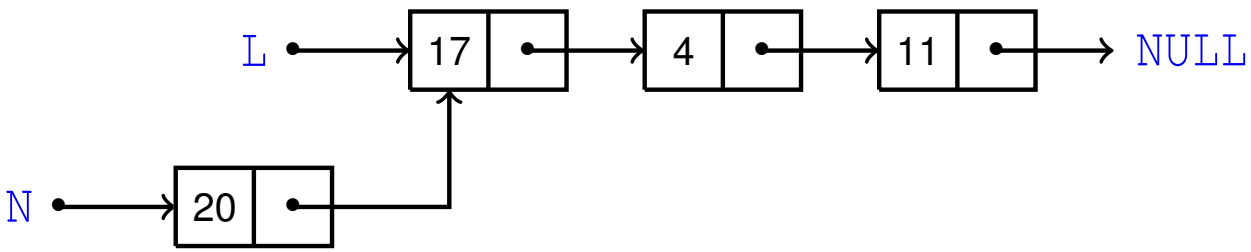
Insertion d'un élément en tête de liste

- Pour insérer un élément en tête de liste, il faut :
  - créer une nouvelle cellule pour stocker cet élément (ici 20), pointée par N,



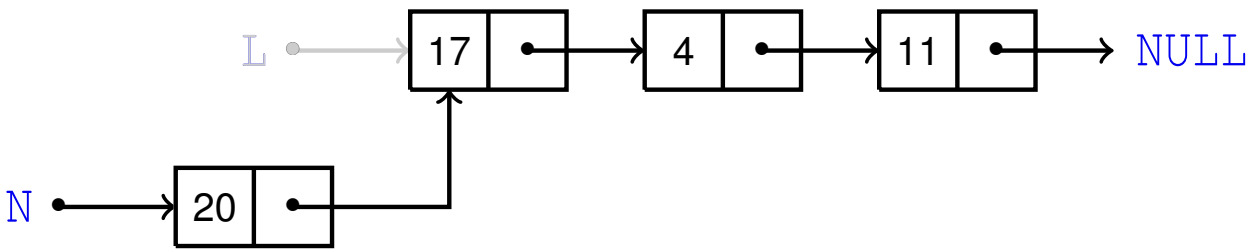
Insertion d'un élément en tête de liste

- Pour insérer un élément en tête de liste, il faut :
  - créer une nouvelle cellule pour stocker cet élément (ici 20), pointée par N,
  - et faire pointer cette nouvelle cellule vers la première cellule de la liste.



Insertion d'un élément en tête de liste

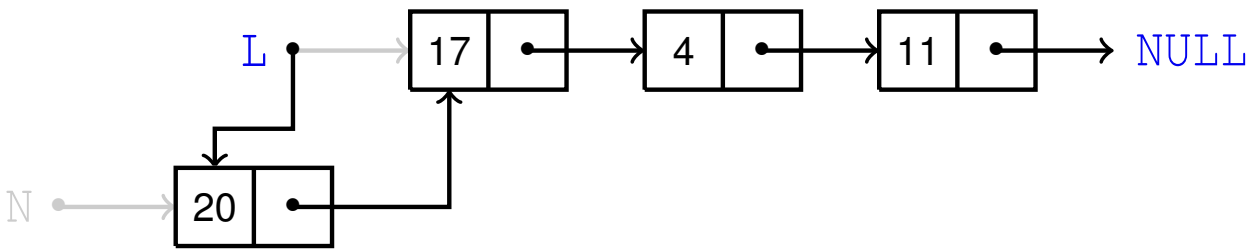
- Pour insérer un élément en tête de liste, il faut :
  - créer une nouvelle cellule pour stocker cet élément (ici 20), pointée par `N`,
  - et faire pointer cette nouvelle cellule vers la première cellule de la liste.



- La tête de la liste est maintenant pointée par `N`.

Insertion d'un élément en tête de liste

- Pour insérer un élément en tête de liste, il faut :
  - créer une nouvelle cellule pour stocker cet élément (ici 20), pointée par `N`,
  - et faire pointer cette nouvelle cellule vers la première cellule de la liste.



- La tête de la liste est maintenant pointée par `N`.
- On peut aussi faire pointer l'ancienne tête de liste `L` sur le nouvel élément.

Exercice

Écrire une fonction qui prend en paramètre une liste chaînée et une valeur (entier), qui insère cette valeur en tête de la liste, et qui retourne la liste modifiée.

Exercice

Écrire une fonction qui prend en paramètre une liste chaînée et une valeur (entier), qui insère cette valeur en tête de la liste, et qui retourne la liste modifiée.

```c
struct list
{
  int value;                // donnee contenu dans la liste
  struct list *next;        // pointeur vers la cellule suivante dans la liste
};

struct list*
insertionEnTete(struct list* L, int V)
{
  struct list *N = NULL;
  N = (struct list*) malloc(sizeof(struct list));
  N->value = V;
  N->next = L;
  return N;
}
int
main(void)
{
  struct list *L = NULL;    // creation d'une liste vide
  // ...
  L = insertionEnTete(L, 11);
  L = insertionEnTete(L, 4);
  affichageCirculaire(L);
  affichageCirculaire(L);
  // ...
```

Mais que fait cette fonction ?

```
void
insertionEnTete(struct list* L2, int V)
{
  struct list *N = NULL;
  N = (struct list*) malloc \
                (sizeof(struct list));

  N->value = V;
  N->next = L2;
  L2 = N;
}

int
main(void)
{
  struct list *L = NULL;
  // ...
  insertionEnTete(L, 11);
  // ...
  return 0;
}
```

0x2···9		
0x2···8		
0x2···7		
0x2···6		
0x2···5		
0x2···4		

0x1···5		
0x1···4		
0x1···3		
0x1···2		
0x1···1		
0x1···0		

Mais que fait cette fonction ?

```
void
insertionEnTete(struct list* L2, int V)
{
  struct list *N = NULL;
  N = (struct list*) malloc \
                (sizeof(struct list));

  N->value = V;
  N->next = L2;
  L2 = N;
}

int
main(void)
{
  struct list *L = NULL;
  // ...
  insertionEnTete(L, 11);
  // ...
  return 0;
}
```

0x2···9		
0x2···8		
0x2···7		
0x2···6		
0x2···5		
0x2···4		

0x1···5		
0x1···4		
0x1···3		
0x1···2		
0x1···1		
0x1···0	L	NULL

Mais que fait cette fonction ?

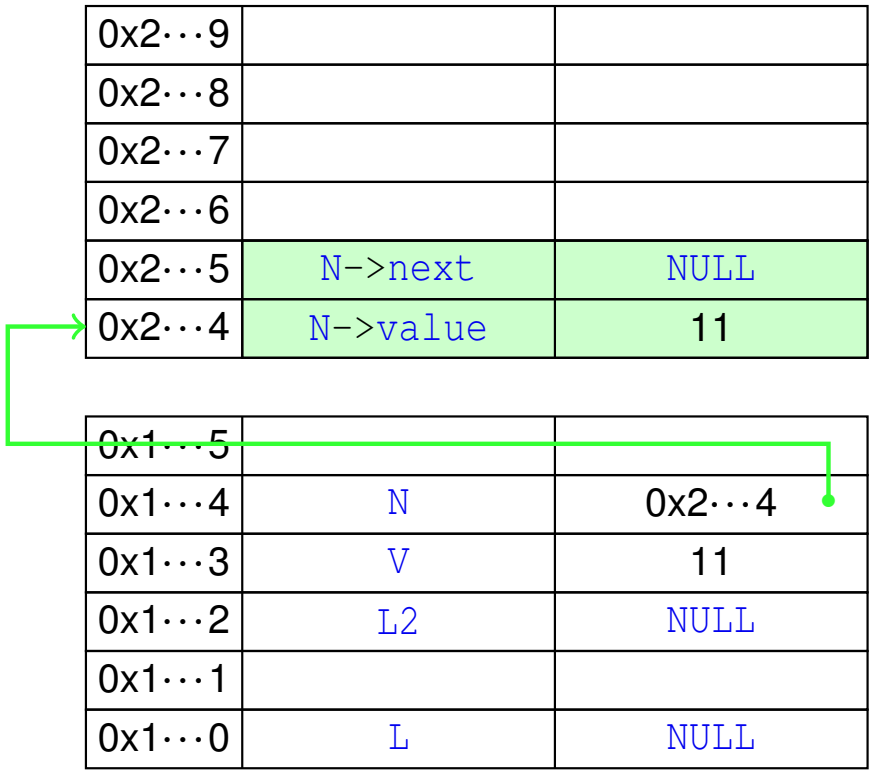
```
void
insertionEnTete(struct list* L2, int V)
{
  struct list *N = NULL;
  N = (struct list*) malloc \
                (sizeof(struct list));

  N->value = V;
  N->next = L2;
  L2 = N;
}

int
main(void)
{
  struct list *L = NULL;
  // ...
  insertionEnTete(L, 11);
  // ...
  return 0;
}
```

0x2···9		
0x2···8		
0x2···7		
0x2···6		
0x2···5		
0x2···4		

0x1···5		
0x1···4	N	NULL
0x1···3	V	11
0x1···2	L2	NULL
0x1···1		
0x1···0	L	NULL

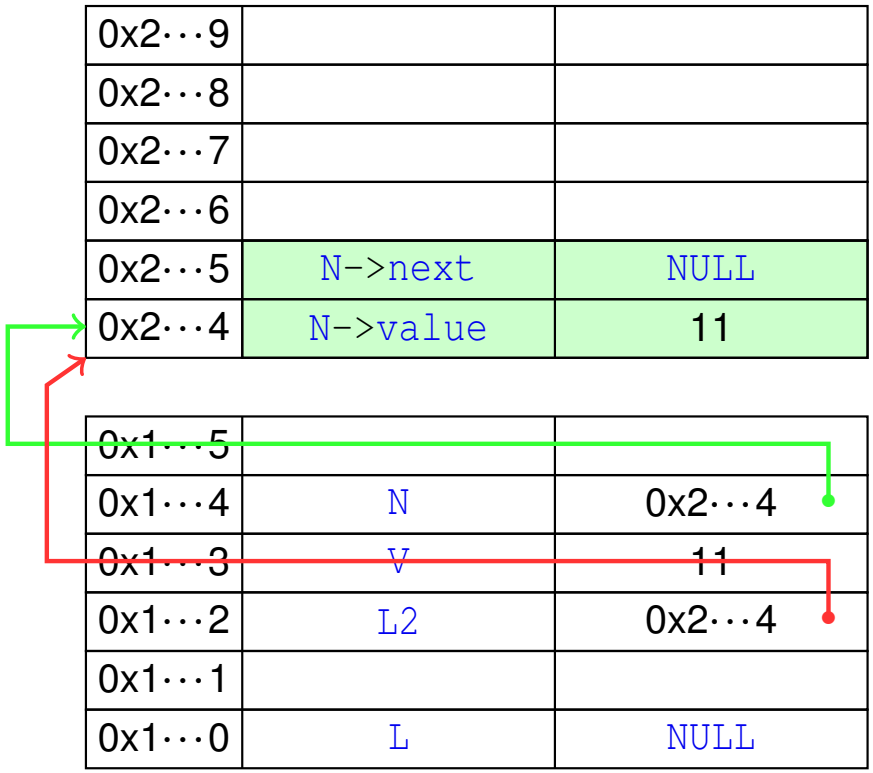
Mais que fait cette fonction ?

```
void
insertionEnTete(struct list* L2, int V)
{
  struct list *N = NULL;
  N = (struct list*) malloc \
                (sizeof(struct list));

  N->value = V;
  N->next = L2;
  L2 = N;
}

int
main(void)
{
  struct list *L = NULL;
  // ...
  insertionEnTete(L, 11);
  // ...
  return 0;
}
```



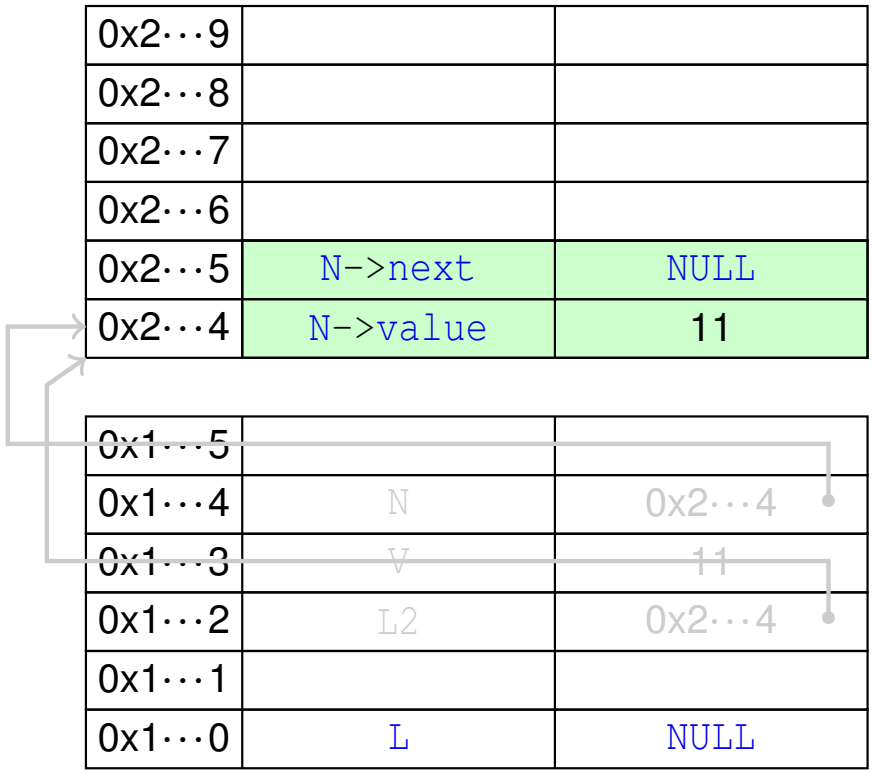
Mais que fait cette fonction ?

```
void
insertionEnTete(struct list* L2, int V)
{
  struct list *N = NULL;
  N = (struct list*) malloc \
                (sizeof(struct list));

  N->value = V;
  N->next = L2;
  L2 = N;
}

int
main(void)
{
  struct list *L = NULL;
  // ...
  insertionEnTete(L, 11);
  // ...
  return 0;
}
```



Mais que fait cette fonction ?

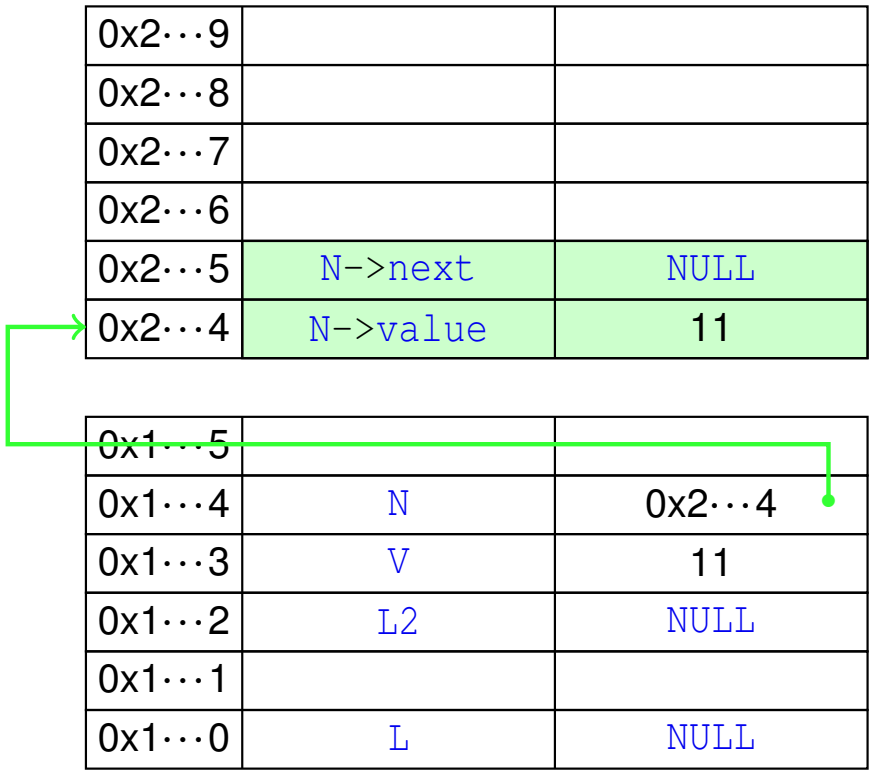
```
void
insertionEnTete(struct list* L2, int V)
{
  struct list *N = NULL;
  N = (struct list*) malloc \
                (sizeof(struct list));

  N->value = V;
  N->next = L2;
  L2 = N;
}

int
main(void)
{
  struct list *L = NULL;
  // ...
  insertionEnTete(L, 11);
  // ...
  return 0;
}
```



- La variable `L2` est une variable locale (de type "pointeur sur liste") à la fonction : modifier sa valeur ne modifie pas la valeur de `L` dans la fonction appelante.

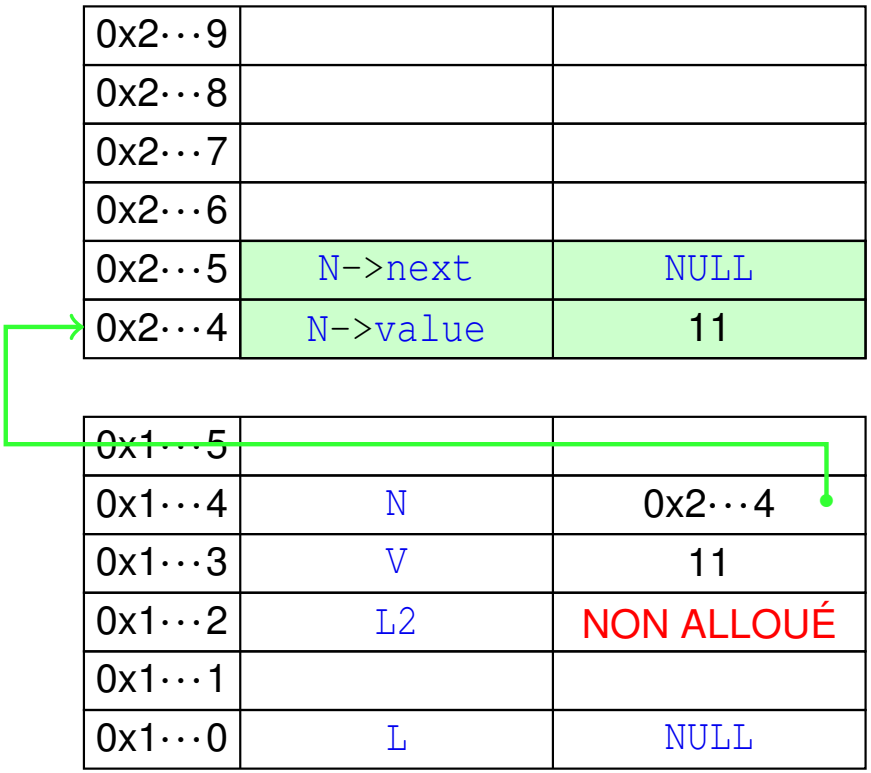
Et celle-ci ?

```
void
insertionEnTete(struct list* L2, int V)
{
  struct list *N = NULL;
  N = (struct list*) malloc \
                 (sizeof(struct list));

  N->value = V;
  N->next = L2;
  *L2 = *N;
}

int
main(void)
{
  struct list *L = NULL;
  // ...
  insertionEnTete(L, 11);
  // ...
  return 0;
}
```



Et celle-ci ?

```
void
insertionEnTete(struct list* L2, int V)
{
  struct list *N = NULL;
  N = (struct list*) malloc \
                 (sizeof(struct list));

  N->value = V;
  N->next = L2;
  *L2 = *N;
}

int
main(void)
{
  struct list *L = NULL;
  // ...
  insertionEnTete(L, 11);
  // ...
  return 0;
}
```



- La zone mémoire pointée par `L2` n'a jamais été allouée !

Comment modifier une liste passée en paramètre ?

```
void
insertionEnTete(struct list** L2, int V)
{
  struct list *N = NULL;
  N = (struct list*) malloc \
                 (sizeof(struct list));

  N->value = V;
  N->next = *L2;
  *L2 = N;
}

int
main(void)
{
  struct list *L = NULL;
  // ...
  insertionEnTete(&L, 11);
  insertionEnTete(&L, 4);
  insertionEnTete(&L, 17);
  insertionEnTete(&L, 20);
  // ...
  affichage(L);
  // ...
  return 0;
}
```

0x2···9		
0x2···8		
0x2···7		
0x2···6		
0x2···5		
0x2···4		

0x1···5		
0x1···4		
0x1···3		
0x1···2		
0x1···1		
0x1···0		

Comment modifier une liste passée en paramètre ?

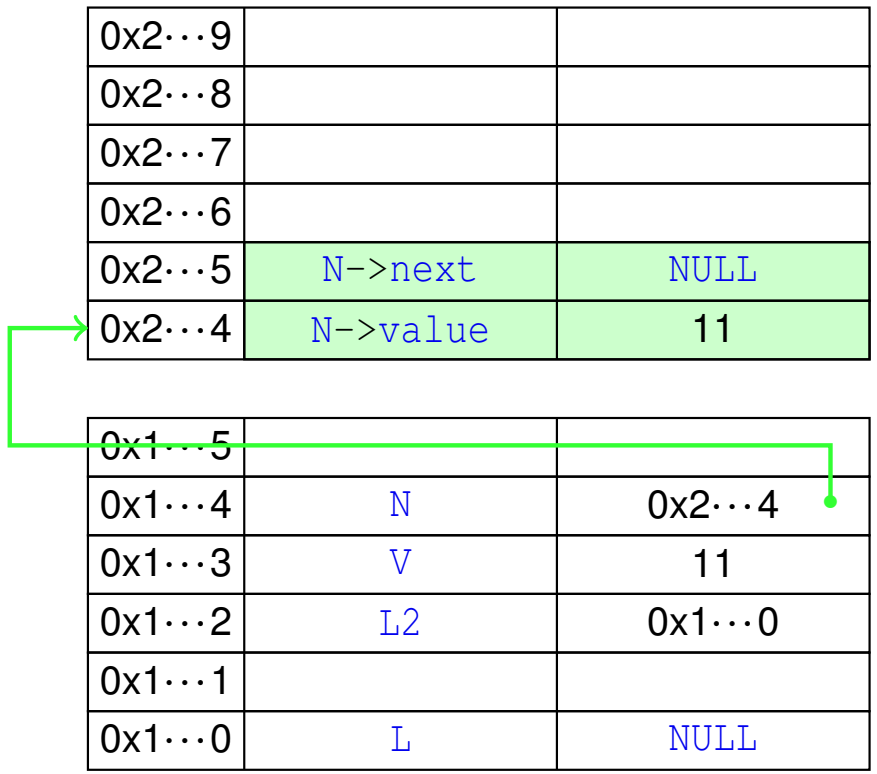
```
void
insertionEnTete(struct list** L2, int V)
{
  struct list *N = NULL;
  N = (struct list*) malloc \
                 (sizeof(struct list));

  N->value = V;
  N->next = *L2;
  *L2 = N;
}

int
main(void)
{
  struct list *L = NULL;
  // ...
  insertionEnTete(&L, 11);
  insertionEnTete(&L, 4);
  insertionEnTete(&L, 17);
  insertionEnTete(&L, 20);
  // ...
  affichage(L);
  // ...
  return 0;
}
```

0x2···9		
0x2···8		
0x2···7		
0x2···6		
0x2···5		
0x2···4		

0x1···5		
0x1···4		
0x1···3		
0x1···2		
0x1···1		
0x1···0	L	NULL

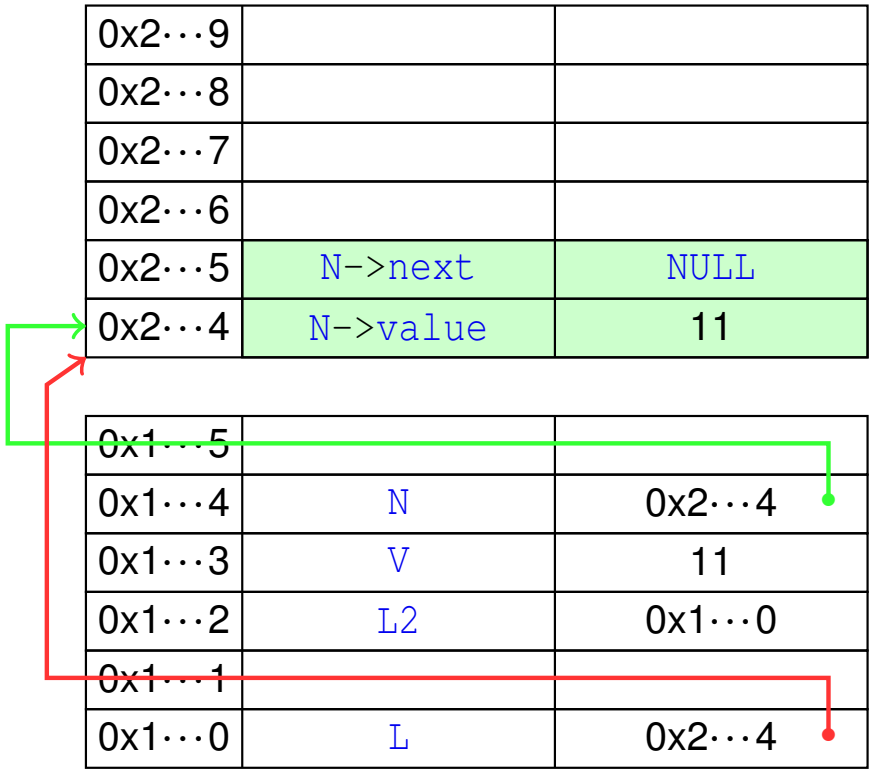
Comment modifier une liste passée en paramètre ?

```c
void
insertionEnTete(struct list** L2, int V)
{
  struct list *N = NULL;
  N = (struct list*) malloc \
                 (sizeof(struct list));

  N->value = V;
  N->next = *L2;
  *L2 = N;
}

int
main(void)
{
  struct list *L = NULL;
  // ...
  insertionEnTete(&L, 11);
  insertionEnTete(&L, 4);
  insertionEnTete(&L, 17);
  insertionEnTete(&L, 20);
  // ...
  affichage(L);
  // ...
  return 0;
}
```



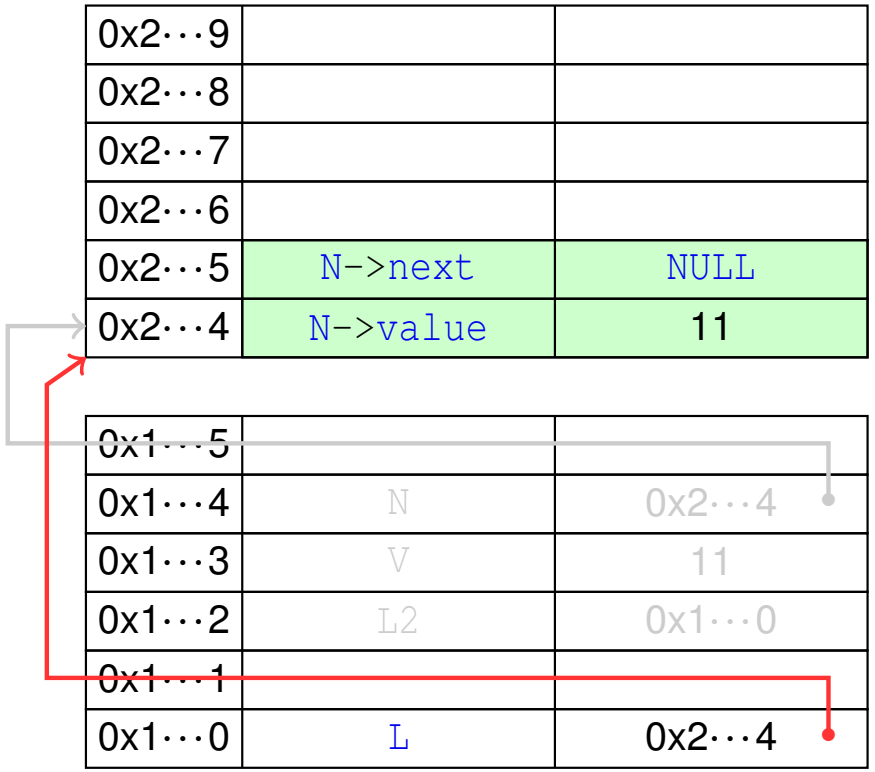
Comment modifier une liste passée en paramètre ?

```c
void
insertionEnTete(struct list** L2, int V)
{
  struct list *N = NULL;
  N = (struct list*) malloc \
                 (sizeof(struct list));

  N->value = V;
  N->next = *L2;
  *L2 = N;
}

int
main(void)
{
  struct list *L = NULL;
  // ...
  insertionEnTete(&L, 11);
  insertionEnTete(&L, 4);
  insertionEnTete(&L, 17);
  insertionEnTete(&L, 20);
  // ...
  affichage(L);
  // ...
  return 0;
}
```



Comment modifier une liste passée en paramètre ?

```c
void
insertionEnTete(struct list** L2, int V)
{
  struct list *N = NULL;
  N = (struct list*) malloc \
                 (sizeof(struct list));

  N->value = V;
  N->next = *L2;
  *L2 = N;
}

int
main(void)
{
  struct list *L = NULL;
  // ...
  insertionEnTete(&L, 11);
  insertionEnTete(&L, 4);
  insertionEnTete(&L, 17);
  insertionEnTete(&L, 20);
  // ...
  affichage(L);
  // ...
  return 0;
}
```



Parcours d'une liste chaînée

- Le parcours d'une liste chaînée permet :
  - d'afficher le contenu d'une liste,
  - de rechercher un élément,
  - de libérer la mémoire, ...

Parcours d'une liste chaînée

- Le parcours d'une liste chaînée permet :
  - d'afficher le contenu d'une liste,
  - de rechercher un élément,
  - de libérer la mémoire, ...

Exercice

Écrire une fonction qui prend en paramètre une liste chaînée, et qui affiche le contenu de cette liste.

Parcours d'une liste chaînée

- Le parcours d'une liste chaînée permet :
  - d'afficher le contenu d'une liste,
  - de rechercher un élément,
  - de libérer la mémoire, ...

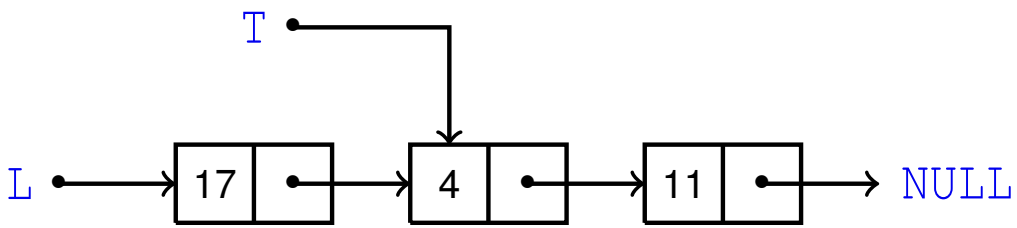
Exercice

Écrire une fonction qui prend en paramètre une liste chaînée, et qui affiche le contenu de cette liste.

```
void
affichage(struct list* L)
{
  struct list *N = L;
  while(N != NULL) {
    printf("%d -> ", N->value);
    N = N->next;
  }
  printf("NULL\n");
}
```

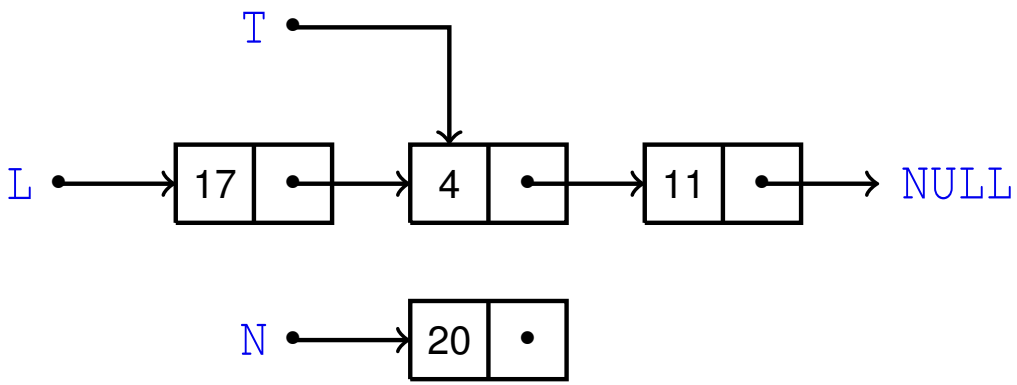
Insertion d'un élément après un élément de la liste

- Pour insérer un élément après un élément pointé par `T`, il faut :



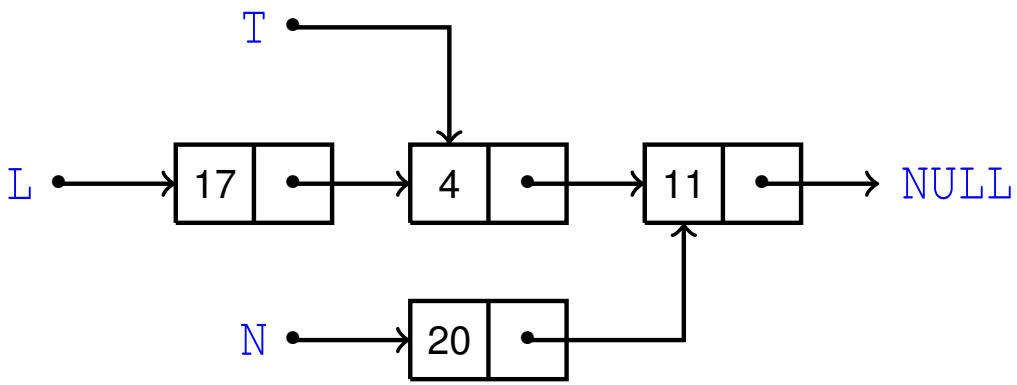
Insertion d'un élément après un élément de la liste

- Pour insérer un élément après un élément pointé par `T`, il faut :
  - créer une nouvelle cellule pour stocker cet élément (ici 20), pointée par `N`,



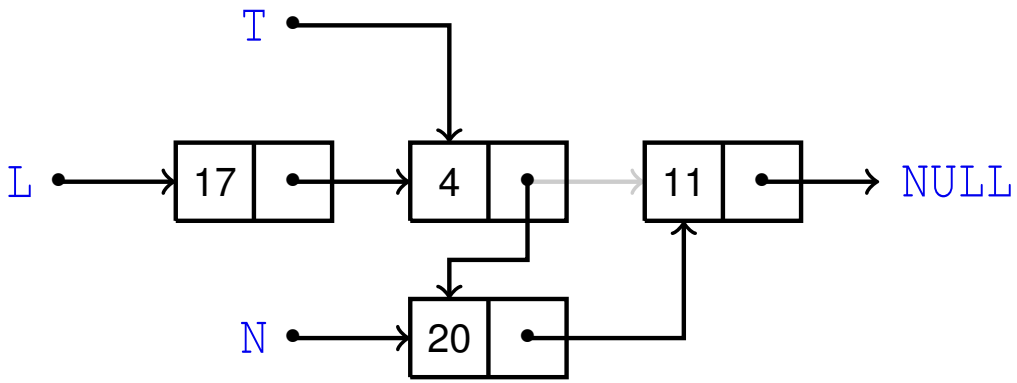
Insertion d'un élément après un élément de la liste

- Pour insérer un élément après un élément pointé par `T`, il faut :
  - créer une nouvelle cellule pour stocker cet élément (ici 20), pointée par `N`,
  - faire pointer cette nouvelle cellule vers le successeur de la cellule pointée par `T`, c'est-à-dire, vers la cellule pointée par `T->next`,



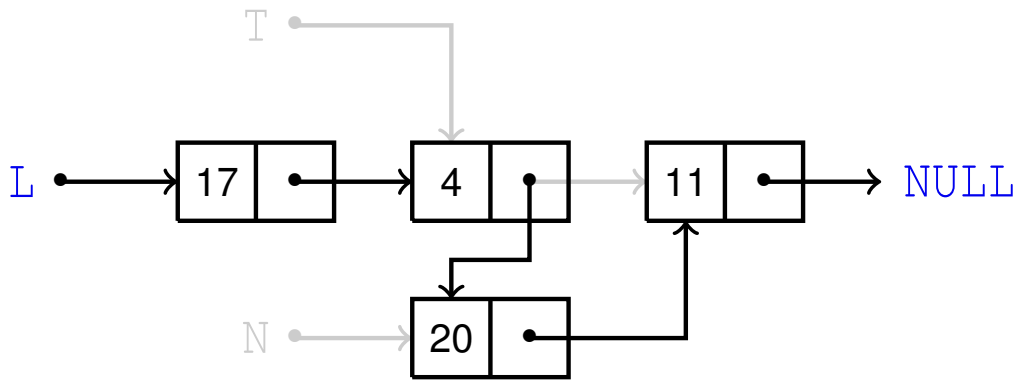
Insertion d'un élément après un élément de la liste

- Pour insérer un élément après un élément pointé par `T`, il faut :
  - créer une nouvelle cellule pour stocker cet élément (ici 20), pointée par `N`,
  - faire pointer cette nouvelle cellule vers le successeur de la cellule pointée par `T`, c'est-à-dire, vers la cellule pointée par `T->next`,
  - et faire pointer la cellule pointée par `T` vers cette nouvelle cellule.



Insertion d'un élément après un élément de la liste

- Pour insérer un élément après un élément pointé par `T`, il faut :
  - créer une nouvelle cellule pour stocker cet élément (ici 20), pointée par `N`,
  - faire pointer cette nouvelle cellule vers le successeur de la cellule pointée par `T`, c'est-à-dire, vers la cellule pointée par `T->next`,
  - et faire pointer la cellule pointée par `T` vers cette nouvelle cellule.



- La tête de la liste est toujours pointée par `L`.
- L'insertion en fin de liste est un cas particulier, où `T` pointe vers le dernier élément de la liste.

Exercice

Écrire une fonction qui prend en paramètre une liste chaînée, une cellule et une valeur (entier), qui insère cette valeur après cette cellule dans la liste, et qui retourne la liste modifiée.

Exercice

Écrire une fonction qui prend en paramètre une liste chaînée, une cellule et une valeur (entier), qui insère cette valeur après cette cellule dans la liste, et qui retourne la liste modifiée.

```
struct list*
insertionApresElement(struct list* L, struct list* T, int V)
{
  struct list *N = NULL;

  if(T != NULL) {
    N = (struct list*) malloc(sizeof(struct list));
    N->value = V;
    N->next = NULL;

    if(L != NULL) {
      N->next = T->next;
      T->next = N;
      return L;
    } else
      return N;
  }
  return L;
}
```

Exercice

Écrire une fonction qui prend en paramètre une liste chaînée et une valeur (entier), qui insère cette valeur en queue de la liste, et qui retourne la liste modifiée.

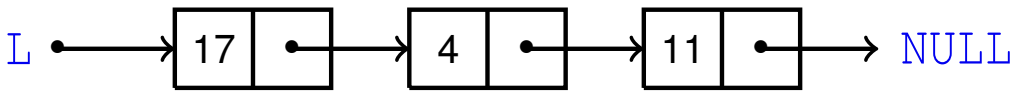
Exercice

Écrire une fonction qui prend en paramètre une liste chaînée et une valeur (entier), qui insère cette valeur en queue de la liste, et qui retourne la liste modifiée.

```
struct list*
insertionEnQueue(struct list* L, int V)
{
  struct list *N = NULL, *T = L;
  N = (struct list*) malloc(sizeof(struct list));
  N->value = V;
  N->next = NULL;

  while(T != NULL && T->next != NULL)
    T = T->next;

  if(L != NULL) {
    N->next = T->next;
    T->next = N;
    return L;
  }else
    return N;
}
```

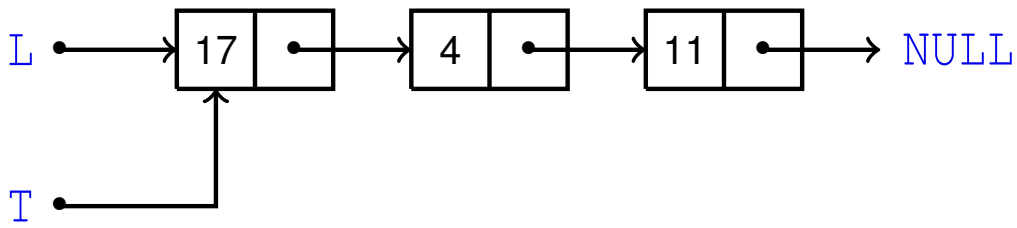
Suppression de l'élément de tête de la liste

- Pour supprimer un élément en tête de liste, il faut :



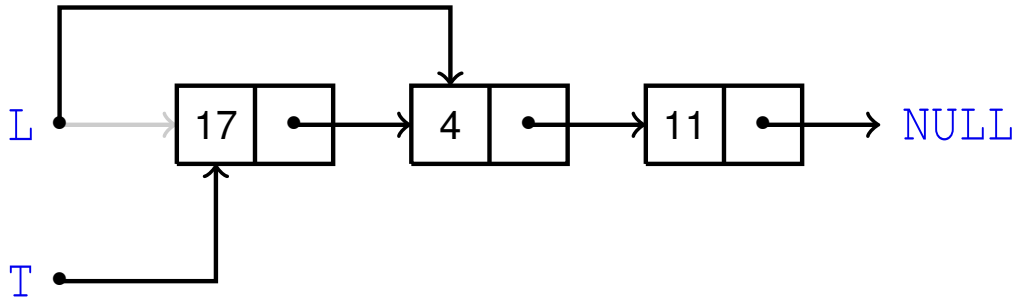
Suppression de l'élément de tête de la liste

- Pour supprimer un élément en tête de liste, il faut :
  - sauvegarder la cellule pointée par `L` (ici, en faisant pointer `T` dessus),



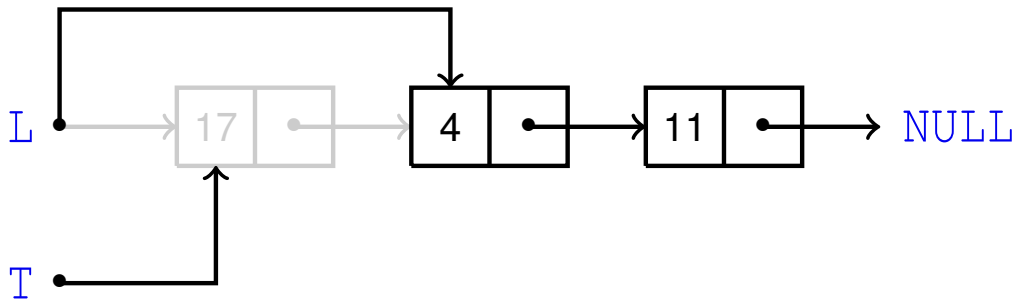
Suppression de l'élément de tête de la liste

- Pour supprimer un élément en tête de liste, il faut :
  - sauvegarder la cellule pointée par `L` (ici, en faisant pointer `T` dessus),
  - faire pointer la tête de liste vers le successeur de la cellule pointée par `L`, c'est-à-dire, vers la cellule pointée par `T->next`,



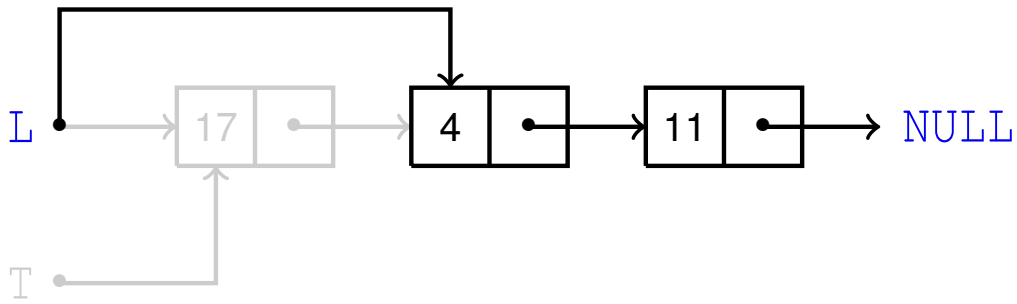
Suppression de l'élément de tête de la liste

- Pour supprimer un élément en tête de liste, il faut :
  - sauvegarder la cellule pointée par `L` (ici, en faisant pointer `T` dessus),
  - faire pointer la tête de liste vers le successeur de la cellule pointée par `L`, c'est-à-dire, vers la cellule pointée par `T->next`,
  - et détruire la cellule pointée par `T`.



Suppression de l'élément de tête de la liste

- Pour supprimer un élément en tête de liste, il faut :
  - sauvegarder la cellule pointée par `L` (ici, en faisant pointer `T` dessus),
  - faire pointer la tête de liste vers le successeur de la cellule pointée par `L`, c'est-à-dire, vers la cellule pointée par `T->next`,
  - et détruire la cellule pointée par `T`.



- La tête de la liste est toujours pointée par `L`.

Exercice

Écrire une fonction qui prend en paramètre une liste chaînée, et qui supprime l'élément de tête de liste.

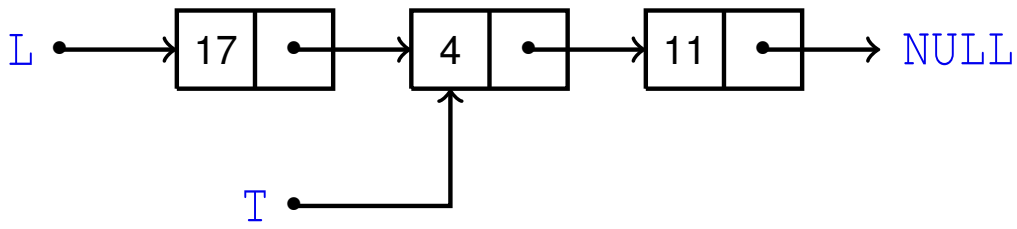
Exercice

Écrire une fonction qui prend en paramètre une liste chaînée, et qui supprime l'élément de tête de liste.

```
struct list*
suppressionEnTete(struct list* L)
{
  struct list *T = L;

  if(T != NULL) {
    L = T->next; // ou L = L->next
    free(T);
  }
  return L;
}
```

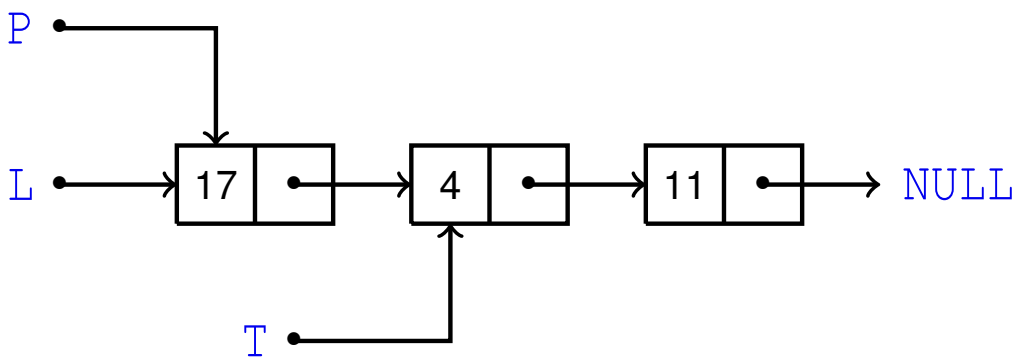
Suppression d'un élément de la liste

- Pour supprimer un élément de la liste pointé par `T`, il faut :



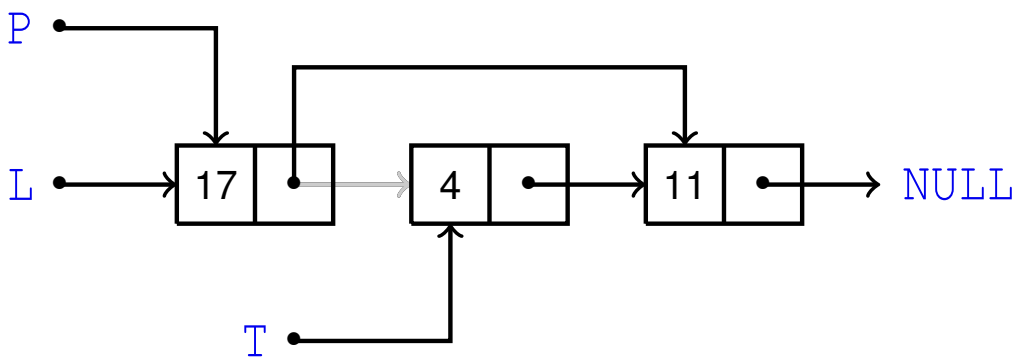
Suppression d'un élément de la liste

- Pour supprimer un élément de la liste pointé par `T`, il faut :
  - rechercher le prédécesseur de la cellule pointée par `T`, et faire pointer `P` dessus,



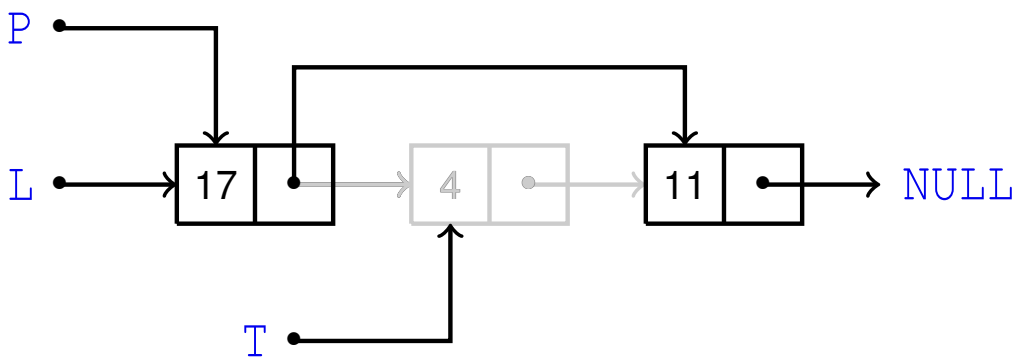
Suppression d'un élément de la liste

- Pour supprimer un élément de la liste pointé par `T`, il faut :
  - rechercher le prédécesseur de la cellule pointée par `T`, et faire pointer `P` dessus,
  - faire pointer la cellule pointée par `P` sur le successeur de la cellule pointée par `T`, c'est-à-dire, vers la cellule pointée par `T->next`,



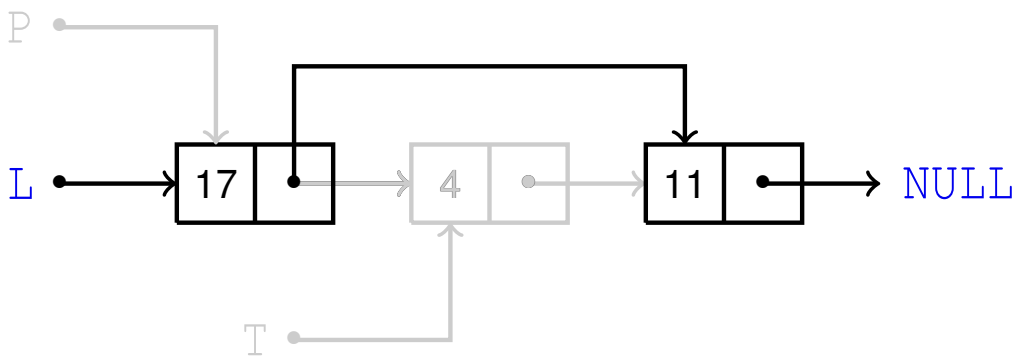
Suppression d'un élément de la liste

- Pour supprimer un élément de la liste pointé par `T`, il faut :
  - rechercher le prédécesseur de la cellule pointée par `T`, et faire pointer `P` dessus,
  - faire pointer la cellule pointée par `P` sur le successeur de la cellule pointée par `T`, c'est-à-dire, vers la cellule pointée par `T->next`,
  - et détruire la cellule pointée par `T`.



Suppression d'un élément de la liste

- Pour supprimer un élément de la liste pointé par `T`, il faut :
  - rechercher le prédécesseur de la cellule pointée par `T`, et faire pointer `P` dessus,
  - faire pointer la cellule pointée par `P` sur le successeur de la cellule pointée par `T`, c'est-à-dire, vers la cellule pointée par `T->next`,
  - et détruire la cellule pointée par `T`.



- La tête de la liste est toujours pointée par `L`.
- La suppression de l'élément en fin de liste est un cas particulier, où `T` pointe vers le dernier élément de la liste.

Exercice

Écrire une fonction qui prend en paramètre une liste chaînée et une cellule, et qui supprime cette cellule de la liste.

Exercice

Écrire une fonction qui prend en paramètre une liste chaînée et une cellule, et qui supprime cette cellule de la liste.

```
struct list*
suppressionElement(struct list* L, struct list *T)
{
  struct list *P = L;

  if(T == L)
    return suppressionEnTete(L);
  else {
    while(P != NULL && P->next != T)
      P = P->next;

    if(P != NULL && T != NULL) {
      P->next = T->next;
      free(T);
    }
    return L;
  }
}
```

Exercice

Écrire une fonction qui prend en paramètre une liste chaînée, et qui supprime le dernier élément de la liste.

Exercice

Écrire une fonction qui prend en paramètre une liste chaînée, et qui supprime le dernier élément de la liste.

```
struct list*
suppressionEnQueue(struct list* L)
{
  struct list *last = NULL;

  while(last != NULL && last->next != NULL)
    last = last->next;

  return suppressionElement(L, last);
}
```

Exercice

Écrire une fonction récursive qui prend en paramètre une liste chaînée, et qui libère la mémoire utilisée.

Exercice

Écrire une fonction récursive qui prend en paramètre une liste chaînée, et qui libère la mémoire utilisée.

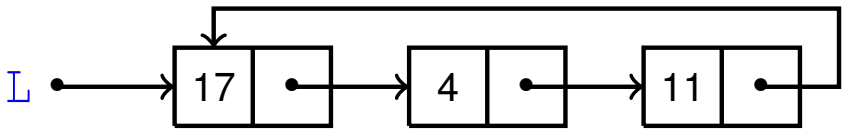
```
struct list*
liberation(struct list* L)
{
  struct list *T = L, *P = NULL;

  while(T != NULL) {
    P = T->next;
    free(T);
    T = P;
  }
  return NULL;
}
```

```
liberation_rec(struct list* L)
{
  struct list *T = L;

  if(L != NULL) {
    T = T->next;
    free(L);
    return liberation_rec(T);
  }
  return NULL;
}
```

Complément 1 : Les listes chaînées

Qu'est-ce qu'une liste chaînée circulaire ?

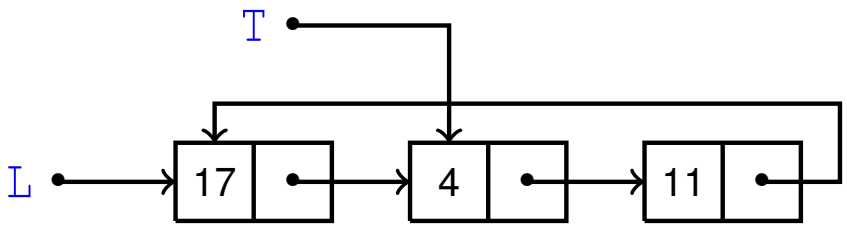
- Définition : Une liste chaînée circulaire est une liste chaînée où le dernier élément ne pointe plus vers `NULL`, mais vers le premier élément de la liste (tête de liste).



- Remarques
  - premier et dernier éléments n'ont plus la même importance,
  - le premier élément peut être n'importe quelle cellule, le dernier étant celui qui le précède dans la liste,
  - il n'est plus nécessaire de différencier les algorithmes en tête, en queue et en milieu de liste.

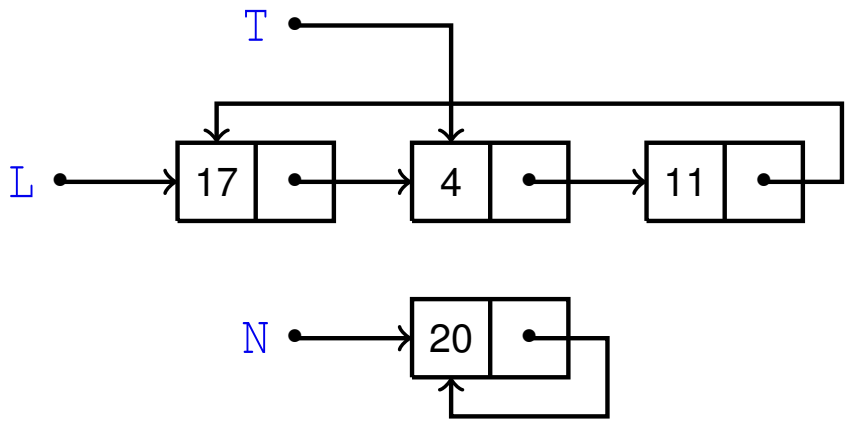
Insertion d'un élément dans une liste circulaire

- Chaque élément a un prédécesseur dans la liste :
  - insertion en tête = insertion après le prédécesseur de la tête de liste,
  - insertion après un autre élément de la liste = insertion après cet élément dans la liste,
  - insertion en queue de liste = insertion après le prédécesseur de la tête de liste.



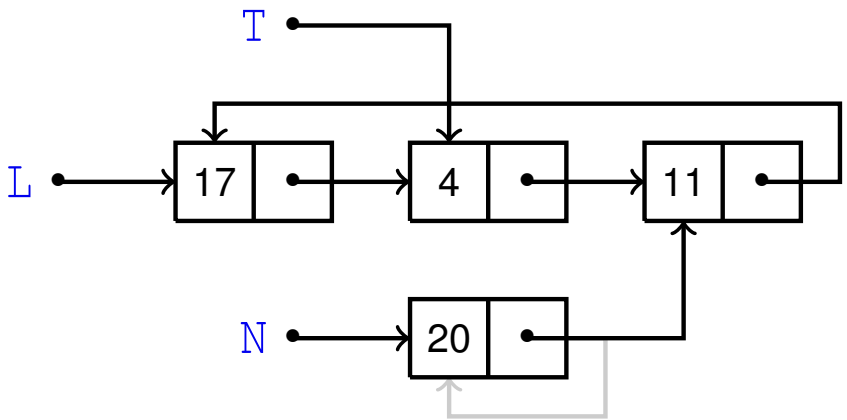
Insertion d'un élément dans une liste circulaire

- Chaque élément a un prédécesseur dans la liste :
  - insertion en tête = insertion après le prédécesseur de la tête de liste,
  - insertion après un autre élément de la liste = insertion après cet élément dans la liste,
  - insertion en queue de liste = insertion après le prédécesseur de la tête de liste.



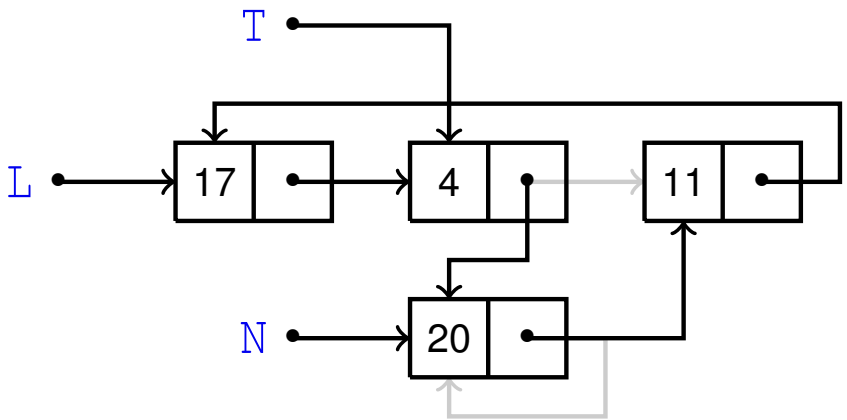
Insertion d'un élément dans une liste circulaire

- Chaque élément a un prédécesseur dans la liste :
  - insertion en tête = insertion après le prédécesseur de la tête de liste,
  - insertion après un autre élément de la liste = insertion après cet élément dans la liste,
  - insertion en queue de liste = insertion après le prédécesseur de la tête de liste.



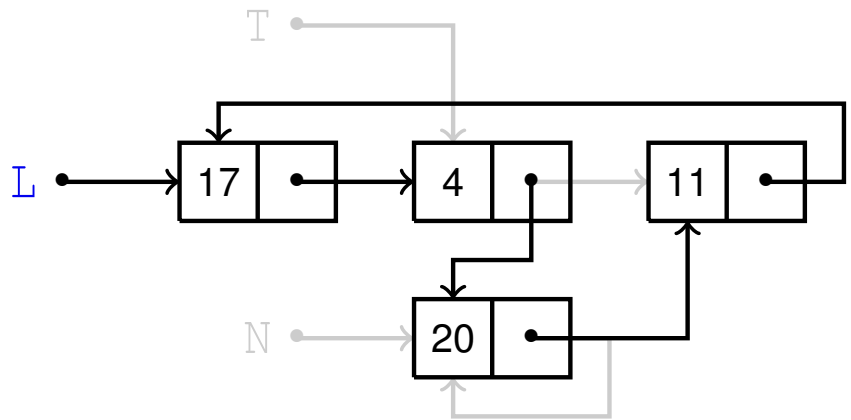
Insertion d'un élément dans une liste circulaire

- Chaque élément a un prédécesseur dans la liste :
  - insertion en tête = insertion après le prédécesseur de la tête de liste,
  - insertion après un autre élément de la liste = insertion après cet élément dans la liste,
  - insertion en queue de liste = insertion après le prédécesseur de la tête de liste.



Insertion d'un élément dans une liste circulaire

- Chaque élément a un prédécesseur dans la liste :
  - insertion en tête = insertion après le prédécesseur de la tête de liste,
  - insertion après un autre élément de la liste = insertion après cet élément dans la liste,
  - insertion en queue de liste = insertion après le prédécesseur de la tête de liste.



- La tête de la liste est ici toujours pointée par `L`, mais elle peut éventuellement pointer vers `N` si on insére l'élément en tête.

Exercice

Écrire une fonction qui prend en paramètre une liste chaînée circulaire, une cellule et une valeur (entier), qui insère cette valeur après cette cellule dans la liste, et qui retourne la liste modifiée.

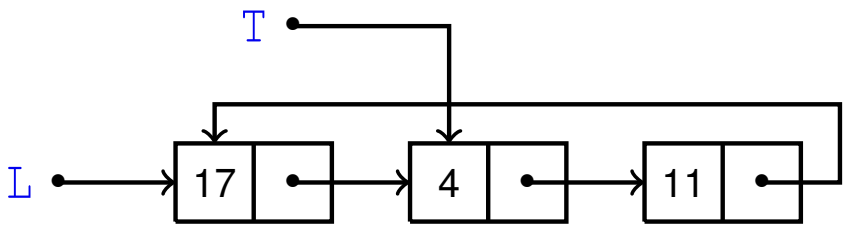
Exercice

Écrire une fonction qui prend en paramètre une liste chaînée circulaire, une cellule et une valeur (entier), qui insère cette valeur après cette cellule dans la liste, et qui retourne la liste modifiée.

```
struct list*
insertionCirculaire(struct list* L, struct list* T, int V)
{
  struct list *N = NULL;
  N = (struct list*) malloc(sizeof(struct list));
  N->value = V;
  N->next = N; // /!\ on fait pointer le dernier element vers le premier

  if(T != NULL) {
    if(L != NULL) {
      N->next = T->next;
      T->next = N;
      return L;
    } else
      return N;
  }
  return N;
}
```

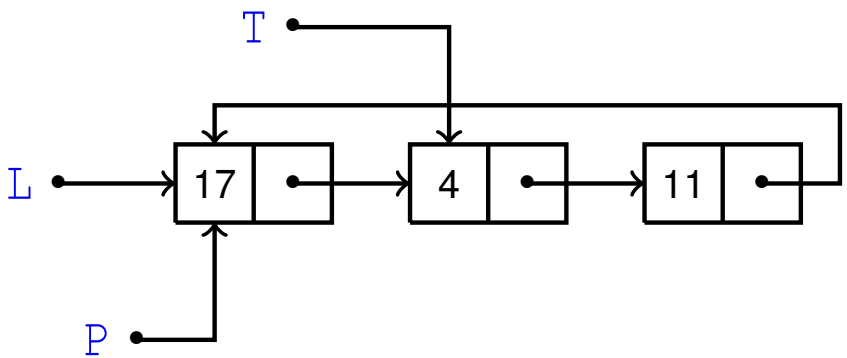
Suppression d'un élément dans une liste circulaire

- Encore une fois, chaque élément a un prédécesseur dans la liste :
  - suppression en tête = suppression de la tête de liste,
  - suppression d'un élément de la liste = suppression de cet élément dans la liste,
  - suppression de la queue de liste = suppression du prédécesseur de la tête de liste.



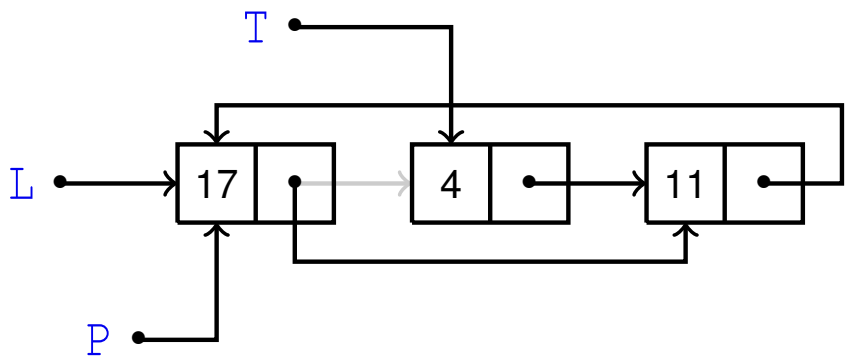
Suppression d'un élément dans une liste circulaire

- Encore une fois, chaque élément a un prédécesseur dans la liste :
  - suppression en tête = suppression de la tête de liste,
  - suppression d'un élément de la liste = suppression de cet élément dans la liste,
  - suppression de la queue de liste = suppression du prédécesseur de la tête de liste.



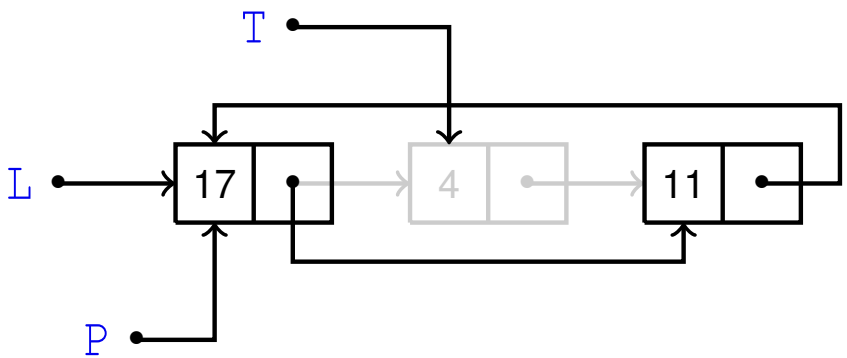
Suppression d'un élément dans une liste circulaire

- Encore une fois, chaque élément a un prédécesseur dans la liste :
  - suppression en tête = suppression de la tête de liste,
  - suppression d'un élément de la liste = suppression de cet élément dans la liste,
  - suppression de la queue de liste = suppression du prédécesseur de la tête de liste.



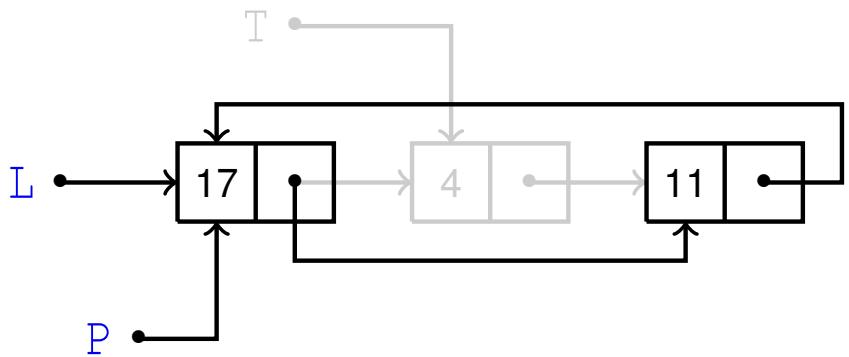
Suppression d'un élément dans une liste circulaire

- Encore une fois, chaque élément a un prédécesseur dans la liste :
  - suppression en tête = suppression de la tête de liste,
  - suppression d'un élément de la liste = suppression de cet élément dans la liste,
  - suppression de la queue de liste = suppression du prédécesseur de la tête de liste.



Suppression d'un élément dans une liste circulaire

- Encore une fois, chaque élément a un prédécesseur dans la liste :
  - suppression en tête = suppression de la tête de liste,
  - suppression d'un élément de la liste = suppression de cet élément dans la liste,
  - suppression de la queue de liste = suppression du prédécesseur de la tête de liste.



- La tête de la liste est ici toujours pointée par `L`, mais elle peut éventuellement pointer vers le successeur de `P` si on supprime l'élément en tête.

Exercice

Écrire une fonction qui prend en paramètre une liste chaînée circulaire et une cellule, et qui supprime cette cellule de la liste.

Exercice

Écrire une fonction qui prend en paramètre une liste chaînée circulaire et une cellule, et qui supprime cette cellule de la liste.

```
suppressionCirculaire(struct list* L, struct list *T)
{
  struct list *P = L, *H = L;

  while(P != NULL && T != NULL && P->next != T)
    P = P->next;

  if(P != NULL && T != NULL) {
    if(P != T) {
      P->next = T->next;
      H = (L == T ? T->next : L);
    } else
      H = NULL;

    free(T);
  }
  return H;
}
```

Exercice

Écrire une fonction qui prend en paramètre une liste chaînée circulaire, et qui affiche le contenu de cette liste.

Exercice

Écrire une fonction qui prend en paramètre une liste chaînée circulaire, et qui affiche le contenu de cette liste.

```
affichageCirculaire(struct list* L)
{
  struct list *N = L;
  if(N != NULL) {
    do {
      printf("%d -> ", N->value);
      N = N->next;
    } while(N != L);
    printf("END\n");
  }
}
```

Exercice

Écrire une fonction qui prend en paramètre une liste chaînée circulaire, et qui libère la mémoire utilisée.

Exercice

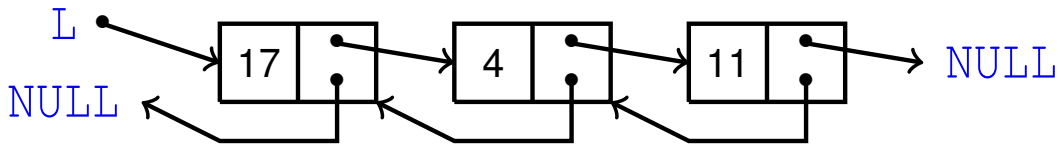
Écrire une fonction qui prend en paramètre une liste chaînée circulaire, et qui libère la mémoire utilisée.

```
liberationCirculaire(struct list* L)
{
  struct list *T = L;

  while(T != NULL)
    T = suppressionCirculaire(T, T);
  return NULL;
}
```

Complément 1 : Les listes chaînées

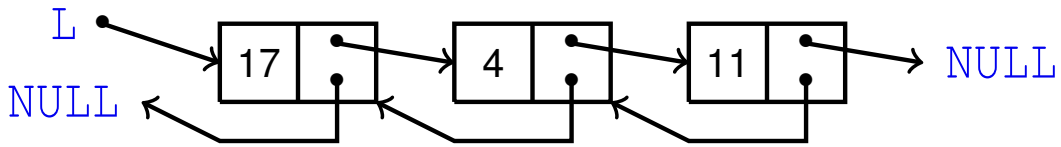
Qu'est-ce qu'une liste doublement chaînée ?

- Définition : Une liste doublement chaînée est une liste chaînée où chaque cellule ne pointe plus uniquement vers son successeur dans la liste, mais également vers son prédécesseur.



Qu'est-ce qu'une liste doublement chaînée ?

- Définition : Une liste doublement chaînée est une liste chaînée où chaque cellule ne pointe plus uniquement vers son successeur dans la liste, mais également vers son prédécesseur.

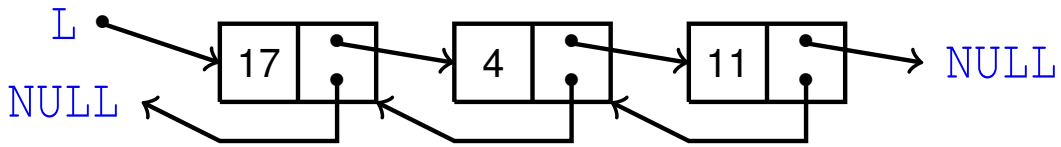


- Comment déclare-t-on une liste doublement chaînée ?

```
struct list_dc
{
  int value;                    // donnee contenu dans la liste
  struct list_dc *prec;         // pointeur vers la cellule precedente dans la liste
  struct list_dc *next;         // pointeur vers la cellule suivante dans la liste
};
```

Qu'est-ce qu'une liste doublement chaînée ?

- Définition : Une liste doublement chaînée est une liste chaînée où chaque cellule ne pointe plus uniquement vers son successeur dans la liste, mais également vers son prédécesseur.



- Comment déclare-t-on une liste doublement chaînée ?

```
struct list_dc
{
  int value;                  // donnee contenu dans la liste
  struct list_dc *prec;       // pointeur vers la cellule precedente dans la liste
  struct list_dc *next;       // pointeur vers la cellule suivante dans la liste
};
```

- Remarques
  - l'accès au prédécesseur d'une cellule est direct,
  - les algorithmes fonctionnent comme pour les listes simplement chaînées, on doit simplement mettre à jour également le pointeur `prec`.

Exercice

Écrire une fonction qui prend en paramètre une liste doublement chaînée, une cellule et une valeur (entier), qui insère cette valeur après cette cellule dans la liste, et qui retourne la liste modifiée.

Exercice

Écrire une fonction qui prend en paramètre une liste doublement chaînée, une cellule et une valeur (entier), qui insère cette valeur après cette cellule dans la liste, et qui retourne la liste modifiée.

```
struct list_dc*
insertionApresElement(struct list_dc* L, struct list_dc* T, int V)
{
  struct list_dc *N = NULL;
  N = (struct list_dc*) malloc(sizeof(struct list_dc));
  N->value = V;
  N->prec = NULL; // /!\ on met a jour le pointeur prec de N
  N->next = NULL;

  if(T != NULL) {
    if(L != NULL) {
      N->prec = T;  // /!\ on met a jour le pointeur prec de N
      N->next = T->next;
      if(T->next != NULL)  // ... si T n'est pas le dernier element
        T->next->prec = N; // /!\ on met a jour le pointeur prec de T->next
      T->next = N;
      return L;
    } else
      return N;
  }
  return N;
}
```

Exercice

Écrire une fonction qui prend en paramètre une liste doublement chaînée et une cellule, qui supprime cette cellule de la liste, et qui retourne la liste modifiée.

Exercice

Écrire une fonction qui prend en paramètre une liste doublement chaînée et une cellule, qui supprime cette cellule de la liste, et qui retourne la liste modifiée.

```
suppressionElement(struct list_dc* L, struct list_dc *T)
{
  struct list_dc *P = NULL;

  if(T == L)
    return suppressionEnTete(L);
  else {
    if(L != NULL && T != NULL) {
      P = T->prec; // /!\ P != NULL
      P->next = T->next;
      if(T->next != NULL)  // ... si T n'est pas le dernier element
        T->next->prec = P; // /!\ on met a jour le pointeur prec de T->next
      free(T);
    }
    return L;
  }
}
```

Que va-t-on voir dans ce cours ?

Complément 2 : Les piles et les files

Complément 2 : Les piles et les files

Complément 2 : Les piles et les files

Qu'est-ce qu'une pile ?

- Une pile est une structure de données dans laquelle on peut ajouter et supprimer des éléments suivant la règle du dernier arrivé, premier sorti : c'est une structure LIFO (Last In First Out).
- Une pile peut être implantée par un tableau ou une liste chaînée.
- Quelque soit le manière utlisée pour l'implanter, les primitives de gestion des piles sont les suivantes :
  - initialisation : création d'une pile vide,
  - estVide : test si la pile est vide,
  - estPleine : test si la pile est pleine,
  - accederSommet : accès au sommet de pile,
  - empiler : ajout (si possible) un élément au sommet de la pile,
  - depiler : suppression (si possible) du sommet de pile,
  - vider : suppression de la totalité des éléments de la pile,
  - detruire : destruction de la pile.

Implantation sous forme d'un tableau

- Déclaration à l'aide d'une structure

```
struct _pile
{
  int nb_element, nb_element_max;
  int *T;
};
typedef struct _pile pile_t;
```

- Intialisation = allocation du tableau

```
pile_t
initialisation(int nb_max)
{
  pile_t P;
  P.nb_element_max = nb_max;
  P.nb_element = 0;
  P.T = (int*)malloc(nb_max*sizeof(int));
  return P;
}
```

Pile vide vs. pleine

- Test si une pile est vide

```
int
estVide(pile_t P)
{
  if(P.nb_element == 0)
    return 1;
  else
    return 0;
}
```

- Test si une pile est pleine

```
int
estPleine(pile_t P)
{
  if(P.nb_element == P.nb_element_max)
    return 1;
  else
    return 0;
}
```

Manipulation des données

- Accès au somment de pile

```
int
accederSommet(pile_t P, int *E)
{
  if(estVide(P))
    return 0;
  else
    *E = P.T[P.nb_element-1];
  return 1;
}
```

- Empiler un élément

```
int
empiler(pile_t P, int E)
{
  if(estPleine(P))
    return 0;
  else {
    P.T[P.nb_element] = E;
    // /!\ allocation eventuelle de
    //     memoire pour chaque
    //     element
    P.nb_element++;
  }
  return 1;
}
```

- Dépiler le sommet de pile

```
int
depiler(pile_t P, int *E)
{
  if(estVide(P))
    return 0;
  else {
    accederSommet(P, E);
    // /!\ liberation eventuelle de la
    //     memoire allouee pour E
    P.nb_element--;
  }
  return 1;
}
```

Destruction d'une pile

- Vider une pile

```
void
vider(pile_t P)
{
  P.nb_element = 0;
  // /!\ liberation eventuelle de la
  //     memoire allouee pour chaque element
}
```

- Destruction d'une pile

```
void
detruire(pile_t P)
{
  if(P.nb_element_max != 0) {
    vider(P);
    free(P.T);
    P.nb_element_max = 0;
  }
}
```

Complément 2 : Les piles et les files

Qu'est-ce qu'une file ?

- Une file est une structure de données dans laquelle on peut ajouter et supprimer des éléments suivant la règle du premier arrivé, premier sorti : c'est une structure FIFO (First In First Out).
- Une file peut également être implantée par un tableau ou une liste chaînée.
- Quelque soit le manière utlisée pour l'implanter, les primitives de gestion des files sont les suivantes :
  - initialisation : création d'une file vide,
  - estVide : test si la file est vide,
  - estPleine : test si la file est pleine,
  - accederTete : accès à la tête de file,
  - enfiler : ajout (si possible) un élément en tête de la file,
  - defiler : suppression (si possible) de la tête de file,
  - vider : suppression de la totalité des éléments de la file,
  - detruire : destruction de la file.

Implantation sous forme d'une liste chaînée

- Déclaration à l'aide d'une structure

```
struct list
{
  int V;
  struct list *N;
};

struct _file
{
  struct list *tete, *queue;
};
typedef struct _file file_t;
```

- Initialisation

```
file_t
initialisation()
{
  file_t F;
  F.tete = NULL;
  F.queue = NULL;
  return F;
}
```

File vide vs. pleine

- Test si une file est vide

```
int
estVide(file_t F)
{
  if(F.tete == NULL)
    return 1;
  else
    return 0;
}
```

- Test si une file est pleine

```
int
estPleine(file_t F)
{
  return 0; // /!\ une liste chainee n'est jamais pleine
}
```

Manipulation des données

- Accès à la tête de file

```
int
accederTete(file_t F, int *E)
{
  if(estVide(F))
    return 0;
  else
    *E = F.tete->V;
  return 1;
}
```

- Défiler la tête de file

```
int
defiler(file_t F, int *E)
{
  if(estVide(F))
    return 0;
  else {
    accederTete(F, E);
    struct list *T = F.tete;
    F.tete = F.tete->N;
    // /!\ liberation eventuelle de la
    //     memoire allouee pour T->V
    free(T);
  }
  return 1;
}
```

- Enfiler un élément

```
int
enfiler(file_t F, int E)
{
  if(estPleine(F))
    return 0;
  else {
    struct list *C = (struct list*) \
           malloc(sizeof(struct list));
    C->V = E;
    // /!\ allocation eventuelle de
    //     memoire pour chaque
    //     element
    C->N = NULL;
    if(estVide(F)) {
      F.tete = C;
      F.queue = C;
    } else {
      // /!\ insertion en queue
      F.queue->N = C;
      F.queue = C;
    }
  }
  return 1;
}
```

Destruction d'une file

- Vider une file

```
void
vider(file_t F)
{
  struct list *p, *q;
  p = F.tete;
  while(p != NULL) {
    q = p;
    p = p->N;
    // /!\ liberation eventuelle de la
    //     memoire allouee pour p->V
    free(q);
    F.tete = NULL;
    F.queue = NULL;
  }
}
```

- Destruction d'une file

```
void
detruire(file_t F)
{
  vider(F);
  // /!\ liberation eventuelle de la memoire
  //     allouee pour la structure sous-jacente
}
```

Questions ?