

Langage C avancé : Séance 1

Marc TAJCHMAN

@-mail : marc.tajchman@cea.fr

CEA - DES/ISAS/DM2S/STMF/LMES

Les supports de ce cours sont disponibles sur Github

Ouvrir la page web : https://github.com/tajchman/MACS1_LC

Objectifs du cours

- Rappels de notions de base de programmation en langage C
- Quelques notions avancées : pointeurs, structures, mémoire dynamique, etc.
- Maîtrise des outils de développement
- Gestion du code source
- Test du code, corrections des erreurs
- Conseil pour du code maintenable, robuste, évolutif

- On suppose que vous savez ce qu'est un répertoire, un fichier;
- Que vous pouvez ouvrir un terminal, dans lequel vous pouvez taper des commandes;
- Que vous pouvez créer, détruire un répertoire, et positionner un terminal dans un répertoire;
- Que vous savez utiliser un éditeur de texte, pour créer et écrire un fichier ou modifier un fichier existant.

N'hésitez pas à poser des questions si vous n'avez pas l'habitude d'effectuer les opérations ci-dessus !!

Ce support de cours et plusieurs exemples vus dans cette séance sont disponibles dans la page internet:

https://github.com/tajchman/MACS1_LC/Seance_1

Avant chaque séance du cours, les supports seront ajoutés dans

https://github.com/tajchman/MACS1_LC

Langage de programmation C

Un langage de programmation :

- Aide le programmeur à structurer les **données** et décrire la suite des **opérations** qui manipulent ces données et génèrent des **résultats**
- Traduit la vision (haut niveau) du programmeur vers la vision (bas niveau) que la machine peut comprendre
- Il existe beaucoup de langages de programmation (~ 1000 actuellement)

Caractéristiques principales:

- typés ou non (les données peuvent changer ou non de type),
- compilés ou interprétés,
- impératifs, fonctionnels, ...
- (orienté-)objets, ...

C est un langage
impératif, typé, compilé

Code source C

Le code source est un texte écrit dans un langage de programmation (par exemple C), lisible par un humain

```
#include <stdio.h>

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

Fichier [ex1.c](#) dans
[exemples/exemple1](#)

Dans le code source, on décrit l'organisation des données et des résultats et les opérations à effectuer.

C est un langage impératif

C langage impératif

Les instructions sont exécutées dans un ordre décidé par le programmeur et modifient des zones mémoires réservées par le programmeur (accessibles par des variables).
Elles sont regroupées dans un ensemble de fonctions, exemple :

```
int F1(double y)
{
    return (int) y*1.5;
}

int F2(double x)
{
    double a = x*3.4;
    int b = F1(a) + F1(a*2.5);
    return b;
}
```

```
int main()
{
    double u = 2.4;
    int v = F2(u);
    return 0;
}
```

Une (et une seule) de ces fonctions doit avoir le nom « main », l'exécution commencera par la 1^{ère} instruction dans « main »

C est un langage typé

Types en C

C a 4 types de base permettant de définir des valeurs et variables simples :
char (caractère), int, float, double

par exemple : `double u;`
 `u = 2.4;`

u est une **variable** de type **double** définie par le programmeur
2.4 est une **valeur** de type **double**

et des variantes : unsigned int, long, wchar_t, ...
(différents intervalles de valeurs possibles, avec ou sans signe)

Une variable occupe une zone mémoire, une valeur n'occupe pas de place en mémoire, mais peut être rangée dans une variable

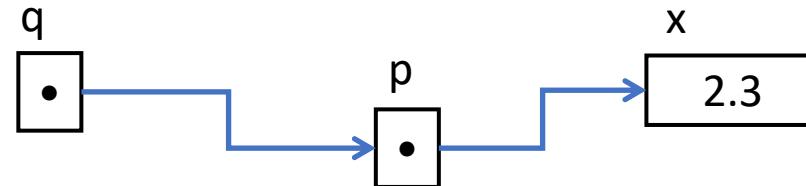
Types en C (2)

D'autres types sont définis à partir des types précédents

- Pointeur qui contient l'adresse d'une zone mémoire, exemples :

```
double x = 2.3;
double *p = &x;
double **q = &p;
char *s;
```

x est une variable de type double, p un pointeur sur un double, q un pointeur sur un pointeur sur double, s un pointeur sur caractère



```
void * p;
```

Type spécial de pointeur quand on ne connaît pas le type de la zone mémoire pointée par p

Types en C (3)

➤ Vecteurs (ensemble de valeurs de même type)

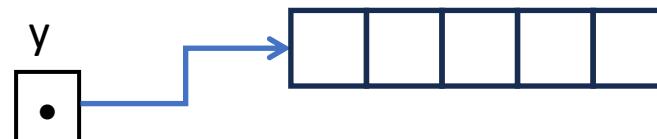
Plusieurs façons de définir des vecteurs:

```
double x[4];
```

```
int n = 5;  
int k[n];
```

```
double *y;  
y = (double *) malloc(n * sizeof(double));  
free(y);
```

x est un vecteur statique de 4 doubles,
k un vecteur statique de 5 entiers,
y un vecteur dynamique de 5 doubles



Types en C (4)

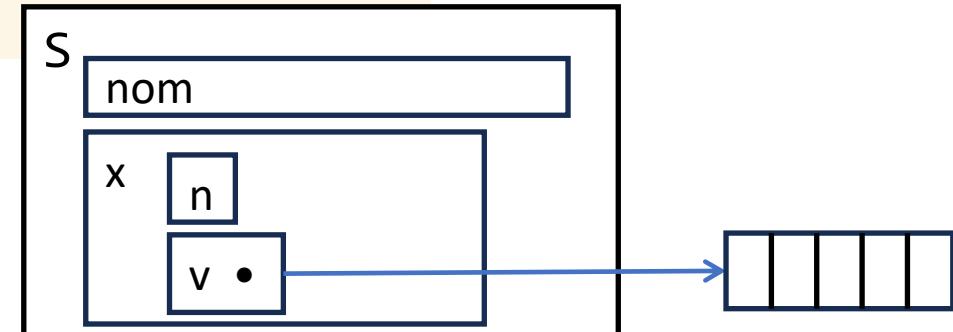
- Structure : agrégation d'un nombre fixe de composantes nommées et de type différents ou non

Les composantes d'une structure peuvent être des types simples, ou des pointeurs ou des vecteurs ou d'autres structures, exemple:

```
struct V {           struct S z;
    int n;
    double * v;
};

struct S {
    char nom[10];
    struct V x;
};
```

```
    strcpy(z.nom, "exemple");
    z.x.n = 5;
    z.x.v = (double *) malloc(z.x.n *
                               sizeof(double));
    z.x.v[2] = 3.2;
```



Types en C (5)

Un octet est une zone mémoire utilisable de la plus petite taille (8 bits, chaque bit peut prendre la valeurs 0 ou 1). Un octet peut être vu comme contenant un entier entre 0 et 255.

Pour tous les types, une fonction standard : **sizeof**, calcule la taille (en nombre d'octets) occupée directement en mémoire par une variable de ce type

Exemples: voir exemple1/source3.c

`sizeof(int)` : taille d'un entier

`sizeof(double)` : taille d'un double

`sizeof(double *)` : taille d'un pointeur de double

`sizeof(double [10])` : taille de 10 doubles

`sizeof(struct V)` : taille \geq somme des tailles d'un entier
et d'un pointeur de double

Types en C (6)

Toutes les valeurs utilisées dans le code source C doivent avoir un type défini et ne peuvent pas changer de type au cours de l'exécution

Toutes les valeurs utilisées dans le code source doivent être représentées par des types définis ci-dessus.

C est un langage compilé

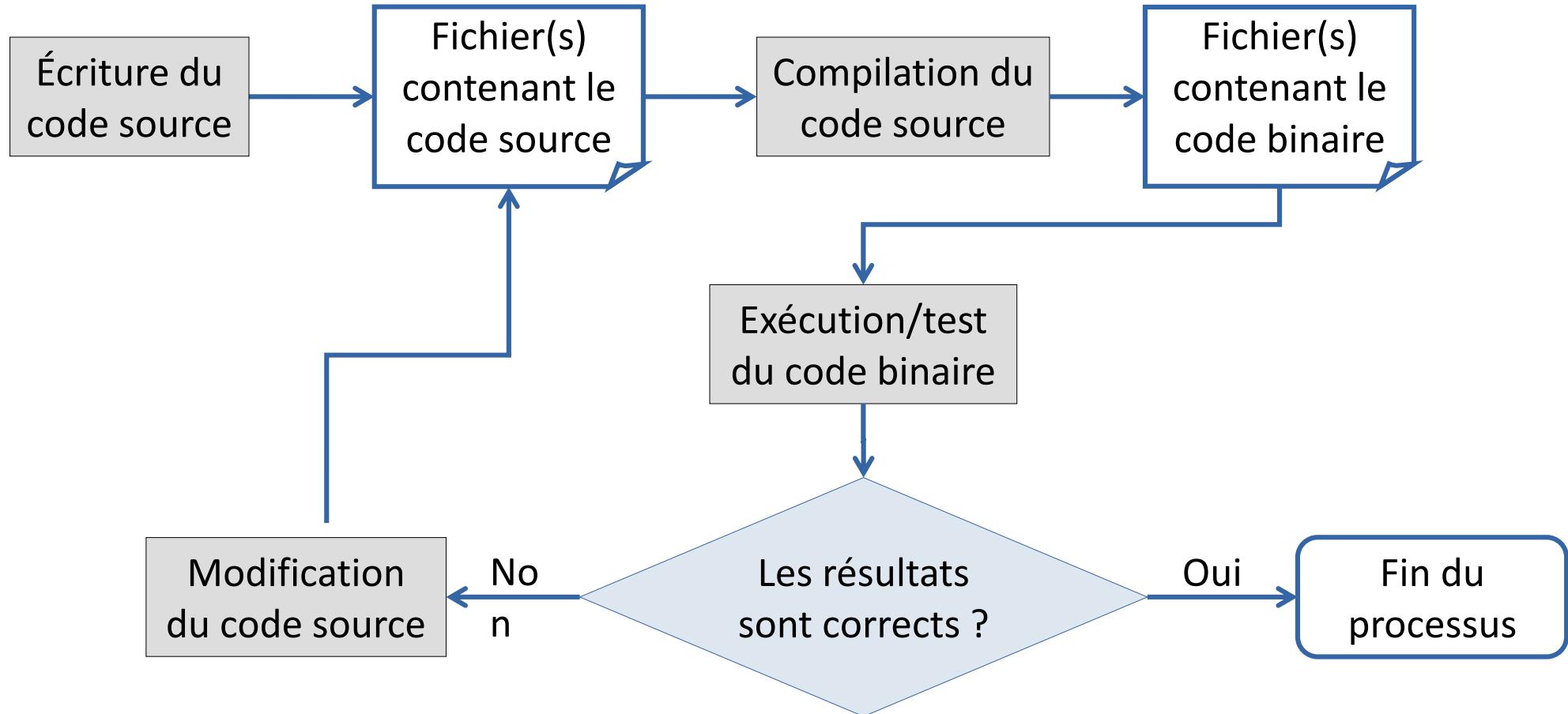
Compilateur

Pour traduire le **code source** (écrit par le programmeur humain) en **code binaire** : instructions bas niveau (compréhensibles par l'ordinateur), un logiciel appelé compilateur est utilisé.

Il existe plusieurs compilateurs : gcc, clang, icx (Intel), xlc (IBM), aocc (AMD), cl.exe (Microsoft), etc.

Leur disponibilité dépend du système qui gère l'ordinateur (Linux, Windows, MacOS, ...) que vous utilisez.

Processus pour aboutir à un code binaire



Compilation, exécution d'un code

Ouvrir un terminal dans le répertoire exemples/exemple1.

Dans ce répertoire, il y a un fichier ex1.c contenant un exemple de code source C

- Pour compiler, tapez la ligne (exemple avec gcc) :

```
gcc ex1.c -o ex1
```

qui génère l'exécutable ex1.

ex1 contient les instructions bas niveau pour produire le résultat des instructions haut niveau contenues dans ex1.c.

- Pour exécuter, tapez la ligne :

```
./ex1
```

En interne la compilation se fait en plusieurs étapes

La compilation, exemple : `gcc ex1.c -o ex1` est faite en interne en plusieurs étapes :

1. Pré-traitement : **preprocessing**

```
gcc -E ex1.c -o ex1b.c
```

2. Génération d'un code source intermédiaire en langage **assembleur**

```
gcc -S ex1b.c -o ex1b.s
```

3. Génération d'instructions binaires correspondant au code source

```
gcc -c ex1b.s -o ex1b.o
```

4. Un fichier exécutable (code binaire) est créé avec le(s) fichier(s) créés par (3) et des instructions binaires système : **édition de liens**

```
gcc ex1b.o -o ex1
```

Première étape : preprocessing

Dans la 1ère étape, un **préprocesseur** génère à partir du code source, un **second code source** où les lignes qui commencent par # sont traitées.

Par exemple :

```
#include "toto.h" /* ligne remplacée par le contenu du fichier toto.h */
#define N 10 /* remplace N par 10 dans toute la suite
           du fichier, */
#ifndef condition
...      /* ne compile les lignes entre ifdef et endif */
#endif /* que si "condition" à la valeur vraie */
```

Pour voir le résultat de cette phase, tapez (pour gcc) : `gcc -E ex1.c`

Exemple de traitement du code par le pré-processeur

Codes source ex2.c
(plusieurs versions)

```
#define N 10
int main() {
    int i=N ;
}
```

gcc -E ex2.c

```
int main() {
    int i=N ;
}
```

gcc -E -DN=10 ex2.c

```
#ifndef N
#define N 10
#endif
int main() {
    int i=N ;
}
```

gcc -E ex2.c

gcc -E -DN=10 ex2.c

Code produit par le pré-
processeur

```
int main() {
    int i=10 ;
}
```

Seconde étape : le code source écrit par le programmeur est traduit en langage assembleur

main:

.LFB0:

```
.cfi_startproc
endbr64
pushq %rbp
.cfi_offset 16 %rax
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_offset 6
subq $16, %rsp
movl %edi, -4(%rbp)
movq %rsi, -16(%rbp)
leaq .LC0(%rip),
movq %rax, %rdi
call puts@PLT
movl $0, %eax
leave
.cfi_offset 7, 8
ret
.cfi_endproc
```

Pour voir le code assembleur, tapez `gcc -S ex1.c` et examiner le fichier `ex1.s`

Troisième étape : l'assembleur (code intermédiaire) est traduit en liste de hexadécimaux (entiers en base 16)

Certains hexadécimaux représentent des instructions bas niveau, d'autres des valeurs numériques ou des lettres, etc

...

0003010	4347	3a43	2820	6255	6e75	7574	3120	2e33
0003020	2e33	2d30	7536	7562	746e	3275	327e	2e34
0003030	3430	2029	3331	332e	302e	0000	732e	7368
0003040	7274	6174	0062	692e	746e	7265	0070	6e2e
0003050	746f	2e65	6e67	2e75	7270	706f	7265	7974
0003060	2e00	6f6e	6574	672e	756e	622e	6975	646c
0003070	692d	0064	6e2e	746f	2e65	4241	2d49	6174
0003080	0067	672e	756e	682e	7361	0068	642e	6e79
0003090	7973	006d	642e	6e79	7473	0072	672e	756e
00030a0	762e	7265	6973	6e6f	2e00	6e67	2e75	6576
00030b0	7272	6660	5556	0073	7222	6655	2e61	7964

Pour produire ces valeurs binaires à partir du code assembleur, tapez :

`gcc -c ex1.s -o ex1.o` OU `gcc -c ex1.c -o ex1.o`

Pour les afficher à l'écran, tapez :

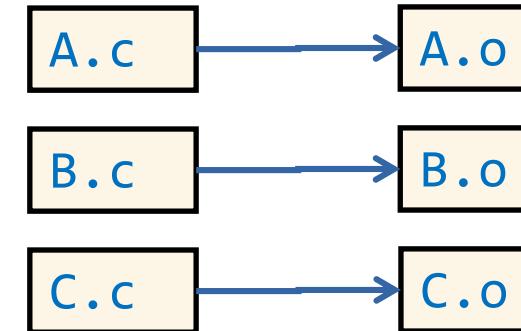
`od -c ex1.o`

Si le code source est contenu dans plusieurs fichiers

Le compilateur traduit en instructions bas niveau chaque fichier source C **séparément** :

Exemple :

dans le répertoire exemple2, les fichiers A.c, B.c et C.c contiennent, chacun, une partie du code source



Pour voir le résultat de cette phase, tapez (pour gcc) :

`gcc -c A.c -o A.o`

`gcc -c B.c -o B.o` **attention : A.o, B.o et C.o**

`gcc -c C.c -o C.o` **ne sont pas exécutables
(ils sont incomplets)**

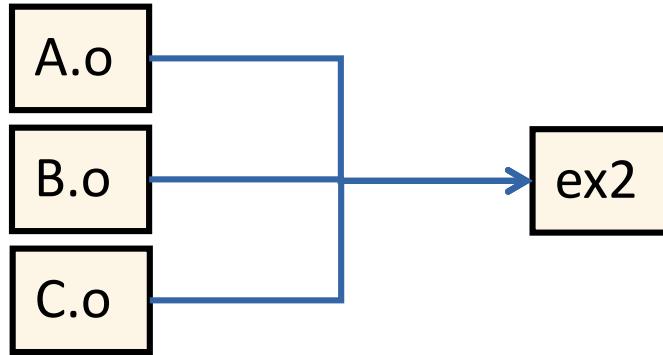
Quatrième étape : Edition de liens (link)

Dans cette étape, l'éditeur de liens :

- Examine les fichiers binaires générés à l'étape précédente
- Vérifie que, si une fonction/variable globale est utilisée dans un des fichiers binaire, cette fonction/variable globale est définie quelque part dans les différents fichiers binaires
- Vérifie que toutes les fonctions sont définies une seule fois
- Vérifie qu'il existe une et une seule fonction « int main(...) » (indique où l'exécution doit commencer)
- Supprime (éventuellement) les fonctions définies mais non utilisées
- Rassemble les fichiers binaires dans un seul fichier qui peut être exécuté

A la fin de cette étape, si le compilateur n'a pas trouvé d'erreur, il crée un fichier binaire prêt à être exécuté sur l'ordinateur.

Quatrième étape : Édition de liens (2)



Pour la compilation complète,
tapez (cas de gcc) :

```
gcc -c A.c -o A.o  
gcc -c B.c -o B.o  
gcc -c C.c -o C.o  
gcc A.o B.o C.o -o ex2
```

On peut faire toutes les étapes en une seule commande :

```
gcc A.c B.c C.c -o ex2
```

C'est plus rapide écrire mais c'est déconseillé si le code source de grande taille et réparti dans de nombreux fichiers (voir séance sur les outils type make, cmake, ...)

Variables

Pour s'exécuter un code doit réserver une zone mémoire qui va contenir les données utilisées et résultats produits.

L'utilisateur doit donc utiliser des variables qui sont des emplacements mémoire de taille suffisante pour contenir des valeurs modifiables dont le type est défini dès la réservation.

Exemples :

```
int i ; /* déclaration d'une variable entière  
         (réservation d'un espace mémoire assez grand  
         pour un entier) */  
  
double x ; /* déclaration d'une variable décimale */
```

```
i = 3 ; /* modification de la valeur dans la variable i */  
x = 4.4 ; /* modification de la valeur dans la variable x */
```

Variables

Toutes les variables utilisées dans un code source doivent avoir un type défini et non modifiable, et on peut seulement les modifier avec des valeurs du même type.

La seule dérogation est s'il existe une conversion implicite entre le type d'une valeur et le type de la variable qui doit contenir cette valeur.
Il y a des conversions de type dangereuses ou non.

Si une variable est utilisée mais non déclarée, le compilateur suppose qu'elle est de type entier.

(voir conseils de bonne pratique, 2 pages plus loin)

Variables (2)

Exemples avec les variables définies précédemment :

```
x = 4 ; /* x contiendra le double 4.0 */
```

Il y a une conversion implicite entier → double. Pas de problèmes, une variable double est capable de représenter un entier sans perte de précision.

```
i = 3.5 ; /* i contiendra l'entier 3 */
```

Il y a une conversion implicite double → entier. La partie décimale de 3.5 (0.5) est perdue. Les compilateurs acceptent cette instruction, certains préviennent le programmeur.

Si c'est vraiment ce que veut le programmeur, il est préférable d'écrire

```
i = (int) 3.5 ;
```

(on utilise une conversion explicite avant de ranger la valeur dans la variable)

Variables : bonnes pratiques

Même si un compilateur accepte de ne pas l'écrire dans le code source:

- Définir explicitement le type de toutes les variables utilisées
- Rechercher les conversions implicites entre variables de type différents et y mettre une conversion explicite

Pour compiler, surtout dans la phase de développement d'un code source, utiliser les options de vérification du compilateur et tenir compte des messages d'avertissement (warnings)

- Pour gcc et clang, utiliser les options `-Wall -Werror -Wextra -pedantic`
- Pour les autres compilateurs, regarder leur documentation

Langage C avancé : Séance 2

Vecteurs

Chaines de caractères

Entrées/sorties sur clavier/écran

Marc TAJCHMAN

@-mail : marc.tajchman@cea.fr

CEA - DES/ISAS/DM2S/STMF/LDEI

Remarques générales que j'ai oublié de mentionner dans la première séance:

Dans le code source C:

- toutes les instructions et les définitions de variables se terminent par un séparateur « ; »
- La partie de texte qui commence par /* et qui se termine par */ est ignorée par le compilateur (c'est un « commentaire »)

➤ Certaines notions sont utilisées et présentées dans cette séance (mémoire dynamique, représentation en mémoire, ...), ce sera développé plus tard.

Code source minimal

Code C minimal

- Ouvrir un éditeur de texte
- Entrer les 4 lignes ci-dessous dans l’éditeur de texte et sauver dans un fichier **ex1.c**

```
int main()
{
    return 0;
}
```

- Enregistrer dans un fichier **ex1.c** (retenir dans quel répertoire se trouve **ex1.c**)

Commentaires sur le code source minimal

- Le code source minimal contient une fonction « main » qui sera utilisée comme point de départ de l'exécution du code binaire.
- Cette fonction n'utilise pas de données, ne contient aucune instruction et génère un seul résultat entier (égal à 0)
- Cet entier est en général utilisé comme indication si l'exécution s'est bien passée (0 si exécution réussie, autre valeur en cas de problème)

Chaque fois que vous commencerez un nouveau code C, il faudra taper ce code source minimal et le compléter

Compilation et exécution du code source minimal

- Ouvrir un terminal de commandes
- Se placer dans le répertoire qui contient le fichier « ex1.c » (dans lequel se trouve le code source minimal).
- Compiler en tapant la commande :

```
gcc ex1.c -o ex1
```

qui génère le fichier binaire « ex1 »

- Exécuter le fichier binaire « ex1 » en tapant la commande :

```
./ex1
```

(rien n'est affiché à l'écran, c'est normal, le code minimal ne contient aucune instruction)

Les vecteurs

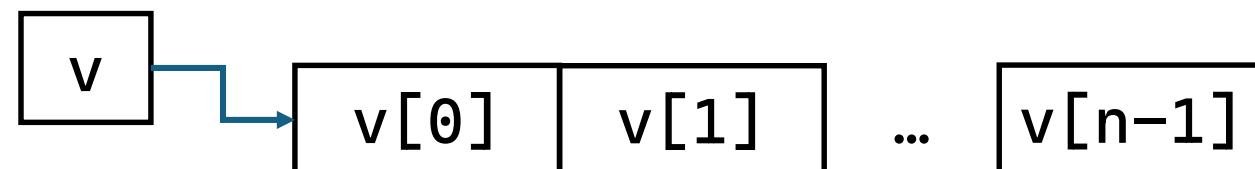
Qu'est ce qu'un vecteur ?

- Un vecteur est un ensemble de n éléments de même type (int, double ou autre type C)
- Pour utiliser un vecteur de type T (T = int, double ou autre type C), on passe par un pointeur :

T * v;

(explicitement ou implicitement)

- En mémoire :



Composantes d'un vecteur

Les composantes d'un vecteur v de taille n sont accessibles sous la forme:

```
v[i] = ... /* écriture (modification) */  
x = ... v[j] ... /* lecture (utilisation) */
```

où i et j sont des entiers entre 0 et $n-1$

Toute utilisation de $v[i]$ où

- i est négatif ou
- i est égal ou supérieur à n

est une erreur (on verra dans une autre séance des outils pour détecter ce type d'erreur)

Plusieurs façons de définir un vecteur (1)

Vecteur de taille définie dans le code source

```
double v[3];
```

La taille de `v` est constante (ici 3), mais chaque composante peut être modifiée:

```
v[2] = v[1] + 1.4;
```

Attention, à la déclaration, les composantes d'un vecteur n'ont pas de valeur prédefinie (en particulier, pas 0)

Il y a un pointeur (fixe) invisible pour l'utilisateur vers le début de la zone mémoire occupée par le vecteur.

Plusieurs façons de définir un vecteur (2)

Vecteur de taille fixe et initialisé à la définition (la taille du vecteur est celle du nombre de valeurs initiales)

```
double v[] = {1.2, 3.4};
```

La taille de v est constante (ici 2), mais chaque composante peut être modifiée:

```
v[0] = v[1] + 1.4;
```

Plusieurs façons de définir un vecteur (3)

Vecteur de taille fixe mais dont la taille n'est pas connue dans le code source (seulement à l'exécution)

```
int n = ...  
...  
double v[n];
```

Cette façon de définir un vecteur (vla : variable length array) est déconseillée : cela ne marche pas avec tous les compilateurs et/ou pour des tailles de vecteur (n) assez grandes.

(voir la séance où on parlera des détails de la gestion mémoire)

Utiliser plutôt l'allocation dynamique de la mémoire (page suivante).

Plusieurs façons de définir un vecteur (4)

Vecteur dont la taille peut être spécifiée à l'exécution et dont la mémoire est gérée dynamiquement.

Pour utiliser un vecteur de taille n et de type entier (par exemple):

- On déclare d'abord un pointeur sur entier:

```
int * v;
```

- On réserve (alloue) la mémoire suffisante pour n entiers (v désigne le début de la mémoire réservé par le système):

```
v = (int *) malloc(n * sizeof(int));
```

- Le vecteur peut être utilisé (n'oubliez pas de l'initialiser)

```
v[0] = 1;  
v[1] = v[0] + 3;
```

Plusieurs façons de définir un vecteur (4)

- Quand le vecteur n'est plus utilisé, on signale au système que sa mémoire est disponible (on libère la mémoire)

```
free(v);
```

- Le pointeur est à nouveau disponible pour un autre vecteur

```
v = (int *) malloc(m * sizeof(int));
```

- ...

Il n'y a pas de restriction sur la taille du vecteur (sauf la taille totale disponible dans le système).

Si malloc ne parvient pas à réservé la mémoire, il a comme résultat un pointeur null (NULL), pour plus de sécurité, ajouter un test sur le pointeur:

```
if (v == NULL)  
    exit(-1); /* ou autre traitement d'erreur */
```

Exercice 1 : Echange de vecteurs

Echange du contenu de 2 vecteurs

- Définir et initialiser 2 vecteurs d'entiers de taille n=100 :
 $v = (v_i = i, i=0, \dots, n)$ et $w = (w_i = i*i, i=0, \dots, n)$
- Echanger le contenu de v et de w
 - en utilisant des vecteurs définis comme à la page 10
 - en utilisant des vecteurs définis avec de la mémoire dynamique (pages 13 et 14), proposer 2 façons de faire

Rappel: pour échanger deux valeurs de type entier (ou tout autre type C), on utilise généralement une variable supplémentaire du même type (temp dans l'exemple):

```
temp = a;  
a = b;  
b = temp;
```

Pour afficher le contenu d'un vecteur d'entiers

Si `v` est un vecteur d'entiers de taille `n`, les lignes de code source suivantes affichent à l'écran le vecteur:

```
printf("v = \n");
for (i=0; i<n; i++)
    printf("v[%3d]: %5d\n", i, v[i]);
printf("\n");
```

Il faut ajouter la ligne `#include <stdio.h>` au début du fichier et ne pas oublier de déclarer l'entier `i`.

Les chaînes de caractères

Les chaines de caractères

- Ce sont des vecteurs de caractères
- Comme pour les nombres (entiers, décimaux), on peut définir des variables/constantes de type caractère ou chaines de caractères
- On utilise des pointeurs sur caractère pour les utiliser

Exemples:

```
int a = 1;  
double x = 1.2e3;  
  
char c1 = 'z';  
char c2 = '1';  
  
char * ch = "azerty";
```

a, x, c1, c2, ch sont des variables
1, 1.2e3 (=1200.0), 'z', '1' et
"azerty" sont des constantes
(remarquer les guillemets simples
pour les constantes caractères et les
guillemets doubles pour les chaines)

Taille (ou longueur) des chaines de caractères

Les chaines de caractères sont des vecteurs de caractères, pour les utiliser, il faut donc connaître leur taille.

1^{ère} possibilité : définir aussi un entier qui contient la longueur de la chaînes

Pas pratique : on utilise souvent beaucoup de chaînes de caractères, et il est facile et dangereux de se tromper en associant à une chaîne, un entier qui contient la longueur d'une autre chaîne : **non retenu dans le langage C**

2^{ème} possibilité : ajouter, en fin de chaîne, un caractère spécial '\0' qui permet de savoir quand la chaîne se termine

Taille (ou longueur) des chaines de caractères (2)

Pour connaitre la longueur d'une chaine, C propose une fonction `strlen` qui fournit cette information en cherchant la position de \0

```
char * s = "azertyuiop";
int n = strlen(s);

/* n contient la valeur 10 */
```

Le résultat de `strlen` est un entier positif ou nul (cas d'une « chaine vide » `char *s = ""`)

Représentation mémoire d'une chaîne de caractères

Une chaîne de n caractères est donc représentée en mémoire par une zone mémoire de $n+1$ emplacements

Par exemple:

"azerty" (chaîne de 6 caractères) est représentée en mémoire par une zone mémoire de taille $7 \times$ (taille de 1 caractère)

a	z	e	r	t	y	\0
---	---	---	---	---	---	----

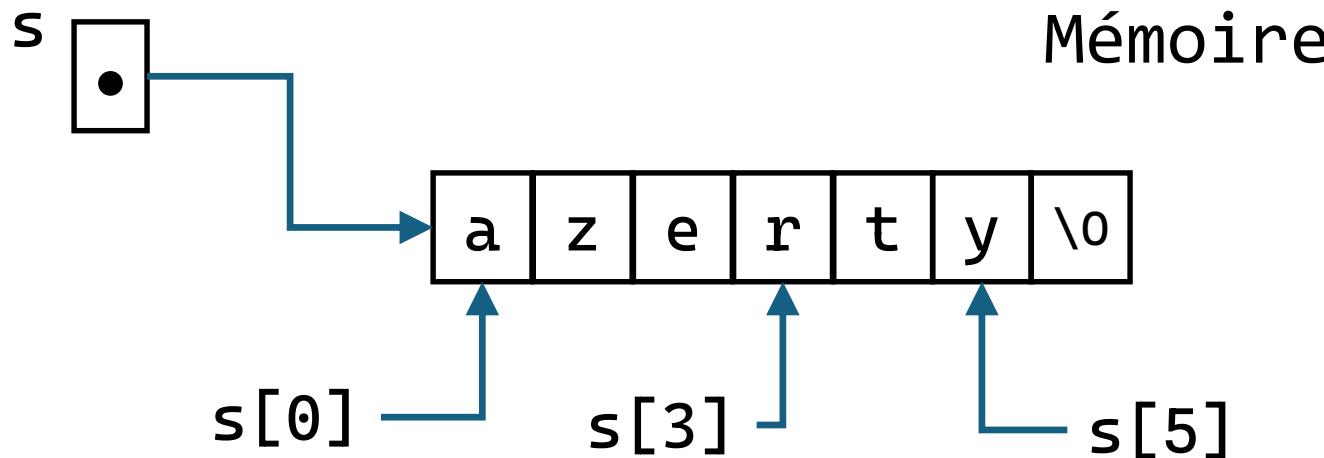
Le caractère en 7^{ème} position à un code spécial (0), il n'est pas possible de l'afficher à l'écran ou de le rentrer au clavier
On appelle parfois ce type de valeur, une sentinelle.

Accès aux caractères dans une chaîne de caractères

Exemple:

```
char * s = "azerty";
```

Code Source



s[i] (*i*=0, ..., *strlen(s)* -1) désigne le *i*^{ème} caractère de la chaîne (si *i* prend une valeur en dehors de l'intervalle [0, *strlen(s)* -1], c'est une erreur)

Plusieurs façons de définir une chaîne de caractères (1)

Chaîne de caractères **constante** (la valeur est fixée à la définition et ne change pas dans le code)

```
char * s = "azertyuiop";
```

Une tentative de modifier le contenu de s est acceptée par le compilateur, mais l'exécution s'arrêtera sur une erreur:

```
s[3] = 'X';
```

Donc, on conseille (fortement) de définir ce type de chaîne comme suit:

```
const char * s = "azertyuiop";
```

Le compilateur refusera toute modification de s.

Plusieurs façons de définir une chaîne de caractères (2)

Seconde façon de définir une chaîne avec une valeur connue

```
char s[] = "azerty";
```

Dans ce cas, le système

- calcule la longueur (6) de la valeur de type chaîne (azerty)
- réserve un espace mémoire de longueur 7 caractères
- recopie la valeur dans la nouvelle zone mémoire

Dans ce cas, une instruction telle que:

```
char * s[3] = 'X';
```

ne pose pas de problème (s n'est pas constante).

Plusieurs façons de définir une chaîne de caractères (3)

Troisième façon de définir une chaîne avec une **valeur non connue mais dont la taille maximale est connue**:

```
char * s;
int n;
n = ... /* par ex. valeur entrée au clavier */

/* réservation de la mémoire de s */
s = (char *) malloc((n+1) * sizeof(char));

/* utilisation de s */

/* libération de la mémoire de s */
free(s)
```

Plus souple, et permet de définir de très grandes chaînes de caractères, mais il faut gérer soi-même la mémoire utilisée par `s` (à l'aide des fonctions `malloc` et `free`)

Plusieurs façons de définir une chaîne de caractères (4)

Variante si une chaîne est initialisée avec une **valeur initiale connue** et la chaîne est **modifiable dans la suite** :

```
#include <string.h>
#include <stdlib.h>
#define CH "azerty"

int main()
{
    char *s = (char *)
        malloc((strlen(CH)+1)*sizeof(char));
    strcpy(s, CH); /* copie dans s */

    free(s);
    return 0;
}
```

La fonction malloc réserve une zone mémoire suffisante pour s (7 caractères). La fonction strcpy est utilisée pour initialiser la chaîne s.

Plusieurs façons de définir une chaîne de caractères (5)

L'exemple précédent peut-être simplifié en:

```
#include <string.h>
#include <stdlib.h>
#define CH "azerty"

int main()
{
    char *s = strdup(CH);

    free(s);
    return 0;
}
```

Plus simple, mais la fonction strdup est une fonction qui n'est pas disponible avec tous les compilateurs et il faut tout de même penser à libérer la mémoire avec free

Exercice 2: Affichage à l'écran

Le but est d'afficher à l'écran un texte suivi d'un entier

Il faudra :

1. Ajouter au code source minimal une variable de type chaîne de caractères et lui donner une valeur
2. Ajouter au code source minimal une variable de type réel simple précision (float) et lui donner une valeur
3. Appeler la fonction C standard qui affiche à l'écran ces informations

Chaine de caractères

- Pour utiliser un texte dans le code source, on utilise une chaine de caractères, en tapant ces 2 lignes dans le code source :
 - une ligne pour définir le pointeur « message » vers un (des) caractère(s) et
 - une ligne pour connecter le pointeur à la chaine fixe de caractères "bonjour")

```
char * message;  
message = "bonjour";
```

- On peut aussi écrire la même chose en une seule ligne :

```
char * message = "bonjour";
```

Réel de type float

- Pour utiliser un float dans le code source, on utilise une variable (de type float), en tapant ces 2 lignes dans le code source :
 - une ligne pour définir une variable de type float et
 - une ligne pour ranger une valeur dans cette variable

```
float x;  
x = 34.4;
```

- On peut aussi écrire la même chose en une seule ligne :

```
float x = 34.4;
```

Affichage de valeurs à l'écran

- Pour afficher des informations à l'écran, il faut
 - Inclure le fichier système `stdio.h` qui définit les fonctions nécessaires
 - Appeler la fonction `printf` (voir plus de précisions à la page suivante)
- Le code source devient:

```
#include <stdio.h>
int main()
{
    char * message = "bonjour x = ";
    float x = 34.4;
    printf("%s %f", message, x);
    return 0;
}
```

La fonction printf

printf est une fonction système de C pour afficher des informations à l'écran, elle a un nombre variable de paramètres (au moins 1) :

- Le premier paramètre (appelé format) est une chaîne de caractères qui indique le format d'affichage (nombre et type de valeurs à afficher)
- Dans le format, « % » suivi d'un entier éventuel et d'un caractère indique le type de valeur à afficher et la façon de traduire la valeur en texte (largeur, nombre de décimale, etc...)
- Le(s) paramètre(s) suivant(s) sont les valeurs à afficher, chaque paramètre doit correspondre à un des « % », en respectant l'ordre et les types

La fonction printf (2)

Exemple:

```
printf("%s n = %3d", message, N);
```

Paramètre de type chaîne appelé « format »
%s : affichage d'une chaîne de caractères
%3d : affichage d'un entier sur 3 positions cadre à droite si l'entier est entre -99 et 999, sur >3 positions sinon

Autres caractères du format affichés tels quels

Valeur/variable de type entier, compatible avec %s

Valeur/variable de type « chaîne de caractères », compatible avec %3d

Référence sur printf

La spécification complète de printf et de son paramètre format est disponible dans la norme C (payante)

<https://www.iso.org/fr/standard/82075.html>

Mais on pourra trouver une information assez complète dans les manuels C ou dans la page

<https://fr.cppreference.com/w/c/io/fprintf>

Exercice

1. Créer un fichier avec le code source de la page 23
2. Le compiler et l'exécuter
3. Modifier la valeur de x à 34.5 dans le code source
4. Recompile et exécuter à nouveau ?
5. Essayer d'expliquer ce qui est affiché dans les 2 cas.

Exercice 3: Lecture d'informations
depuis le clavier

Le but est d'utiliser des valeurs entrées par l'utilisateur au clavier

Il faudra :

1. Définir des variables qui vont contenir les informations (pas nécessaire de les initialiser)
2. Appeler la fonction C standard **scanf** qui attend que l'utilisateur entre des informations et les range dans les variables ci-dessus

Variables

- Le code source doit définir des variables de types correspondant aux types des informations introduites par l'utilisateur au clavier et de **tailles suffisantes**
- Par exemple

```
int N;  
char *prenom = (char *) malloc(10*sizeof(char));
```

Si l'utilisateur peut rentrer au clavier, un entier et une chaîne de caractères de taille maximale 9 (penser au caractère « sentinelle »)

Lecture des valeurs depuis le clavier

- Pour lire des informations depuis le clavier, il faut
 - Inclure le fichier système **stdio.h** qui définit les fonctions nécessaires
 - Appeler la fonction **scanf** (voir plus de précisions à la page suivante)

Pour lire une variable, il faut mettre en paramètre un pointeur sur cette variable, sauf si la variable est déjà un pointeur (par exemple, chaîne de caractères), voir exemple page suivante

Exemple de lecture avec la fonction scanf

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

int main()
{
    char *prenom = (char *) malloc(10*sizeof(char));
    float x;

    printf("Entrez un nombre décimal suivi de votre prénom > ");
    scanf("%f%9s", &x, prenom);

    printf("%s, vous avez entré %f\n", prenom, x);
    free(prenom);
    return 0;
}
```

Référence sur scanf

La spécification complète de scanf est disponible dans la norme C (payante)

<https://www.iso.org/fr/standard/82075.html>

Mais on pourra trouver une information assez complète dans les manuels C ou dans la page

<https://fr.cppreference.com/w/c/io/fscanf>

Exercice

1. Créer un fichier avec le code source de la page 32
2. Le compiler
3. Exécuter le code binaire, rentrer un nombre suivi d'un mot (un prénom), faire plusieurs essais avec
12.3Jean

12.3 Jean

12.34 Jean

Langage C avancé : Séance 3

Pointeurs

Fonctions

Variables locales/globales

Marc TAJCHMAN

@-mail : marc.tajchman@cea.fr

CEA - DES/ISAS/DM2S/STMF/LDEI

Pointeurs

Définitions : zone mémoire, adresse

La mémoire de travail de l'ordinateur est constituée d'une suite d'unités de mémoire (octets ou bytes en anglais) numérotées (ou indicées) de 0 à N (où N est peut être très grand, actuellement souvent $> 10^{10}$)

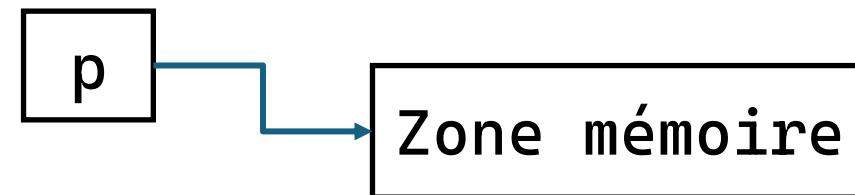
Une zone mémoire est un sous-ensemble d'unités de mémoire consécutives.

L'adresse de la zone mémoire est l'indice de la 1^{ère} unité mémoire de la zone mémoire, c'est un **nombre entier**.

C définit un type d'entier spécifique, **size_t**, capable de représenter correctement une adresse.

Pointeurs

- On a déjà vu la notion de pointeurs pour manipuler des vecteurs et des chaînes de caractères
- Ce ne sont pas les seules utilisations possibles des pointeurs
- Définition générale : un pointeur est une variable, cette variable occupe une zone mémoire et sa valeur est l'adresse d'une autre zone mémoire
- On peut représenter un pointeur p comme ci-dessous :



Pointeurs générique et typé

- Un pointeur peut contenir l'adresse d'une zone mémoire dont on ne connaît pas la structure : on parle d'un pointeur générique

```
void * v;
```

- Un pointeur peut contenir l'adresse d'une zone mémoire qui ne peut contenir que des valeurs d'un **type** choisi par le programmeur (int, double, char, struct, pointeur, ...)

```
int * p;  
double * q;
```

Copies entre pointeurs générique et typé

- Il est possible de copier la valeur d'un pointeur typé ou d'un pointeur générique dans un autre pointeur générique :

```
int * p = ...;  
void * q = p;
```

- Par contre, on ne peut pas copier la valeur d'un pointeur générique dans un pointeur typé, sauf si on change le type du pointeur générique ([et le compilateur ne fait aucune vérification](#)):

```
void * q = ...;  
double * p = q; /* erreur */  
double * r = (double *) q;
```

Copies entre pointeurs générique et typé

- De même, on ne peut pas copier la valeur d'un pointeur typé dans un pointeur d'un type différent:

```
int * q = ...;
double * p = q; /* erreur */
```

Il est toujours possible de forcer la copie en changeant le type d'un des pointeurs, mais c'est dangereux (et rarement une bonne idée)

```
int * q = ...;
double * p = (double *) q;
```

Exemples d'initialisation d'un pointeur générique

1. Un pointeur générique peut être initialisé avec une adresse spéciale (NULL) qui signale qu'il ne pointe vers aucune adresse utilisable:

```
void * v;  
v = NULL;
```

ou

```
void * v = NULL;
```

On pourra écrire, pour tester si un pointeur est utilisable:

```
if (v == NULL)  
    printf("erreur v n'est pas utilisable\n");
```

Exemples d'initialisation d'un pointeur générique (2)

2. Un pointeur générique peut être initialisé avec l'adresse d'une zone mémoire réservée par malloc

```
void * v;  
v = malloc(n);
```

ou void * v = malloc(n);

Ici n représente la taille de la zone mémoire en octets.

La fonction malloc donne comme résultat un pointeur générique (void *) qui est copié directement dans v.

Exemples d'initialisation d'un pointeur générique (3)

Le résultat de `malloc` est `NULL` si la zone mémoire disponible n'est pas suffisante (n est trop grand) ou si $n \leq 0$.

Une règle de bonne programmation est de vérifier que ce n'est pas le cas:

```
if (v == NULL)
    printf("erreur la mémoire demandée n'a pas pu"
           "être réservée\n");
```

Exemples d'initialisation d'un pointeur générique (4)

3. Un pointeur générique peut être initialisé avec l'adresse d'une variable existante (de type quelconque) :

```
int k;  
void * v;  
v = &k;
```

ou

```
int k;  
void * v = &k;
```

&k représente l'adresse de la variable k.

On ne doit pas tester si le pointeur est NULL puisque toute variable existante a forcément une adresse valable.

Exemples d'initialisation d'un pointeur typé

1. Un pointeur typé peut être initialisé avec une valeur spéciale (NULL) qui signale qu'il ne pointe vers aucune adresse:

```
double * d;  
d = NULL;
```

ou

```
double * d = NULL;
```

On pourra donc tester si un pointeur typé est utilisable ou non dans la suite du code:

```
if (d == NULL)  
    printf("erreur d n'est pas utilisable\n");
```

Exemples d'initialisation d'un pointeur typé (2)

2. Un pointeur typé peut être initialisé avec l'adresse d'une zone mémoire réservée par malloc:

```
int * i;  
i = (int *) malloc(n * sizeof(int));
```

ou

```
int * i = (int *) malloc(n * sizeof(int));
```

Dans ce cas, le pointeur i contient l'adresse d'une suite de n éléments du type choisi (ici int).

Il faut donc fournir à malloc la taille cette zone mémoire: n x la taille d'une valeur du type choisi (en bleu dans les exemples ci-dessus).

Exemples d'initialisation d'un pointeur typé (3)

La fonction `malloc` a pour résultat un pointeur générique `void *` et il faut changer son type pour initialiser le pointeur typé (ici `int *`):

```
int * i;  
i = (int *) malloc(n * sizeof(int));
```

C'est un des seuls changements de type de pointeurs qu'il est « raisonnable » de s'autoriser.

Ne pas oublier de tester si le pointeur initialisé par `malloc` n'est pas `NULL`.

Exemples d'initialisation d'un pointeur typé (4)

3. Un pointeur typé peut être initialisé avec l'adresse d'une variable existante du même type :

```
int k;  
int * v;  
v = &k;
```

ou

```
int k;  
int * v = &k;
```

&k représente l'adresse de la variable k.

4. On peut aussi définir et initialiser une zone mémoire de plusieurs valeurs du même type, et en même temps, définir un pointeur vers le début de cette zone:

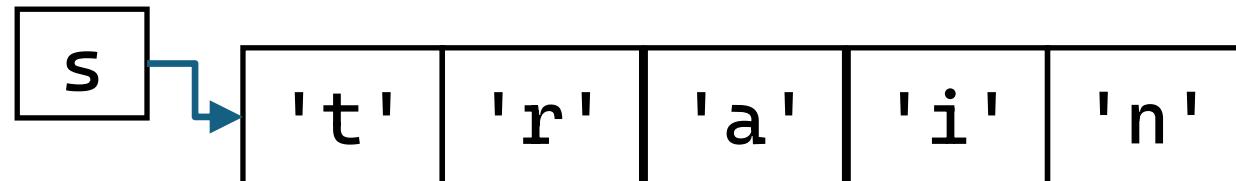
```
double * v = {1.2, 3.4, 4.5};
```

Remarque: traitement particulier des caractères

Notez la différence entre :

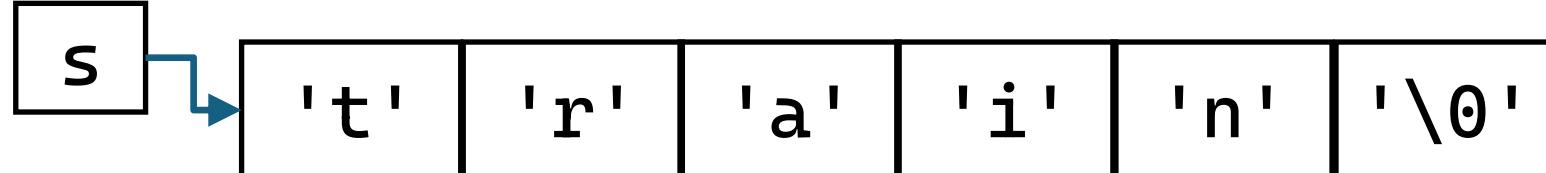
- Un vecteur de caractères (une suite de caractères quelconques)

```
char * s = {'t', 'r', 'a', 'i', 'n'};
```



- Une chaîne de caractères (une suite de caractères dont le dernier est le caractère spécial '\0')

```
char * s = "train";
```



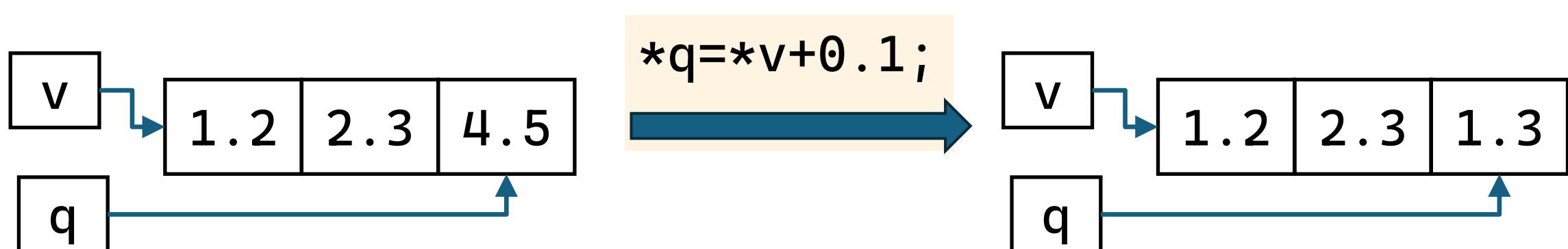
Accès à la valeur pointée par le pointeur

- Pour utiliser ou modifier la valeur contenue dans la mémoire à l'adresse contenue dans le pointeur, on utilise l'opérateur d'indirection *.

Par exemple

```
double v[] = {1.2, 2.3, 4.5};  
double *q = v + 2;  
*q = *v + 0.1;
```

L'effet est le même que si on avait écrit
 $v[2] = v[0] + 0.1$



Caractère * dans le code source

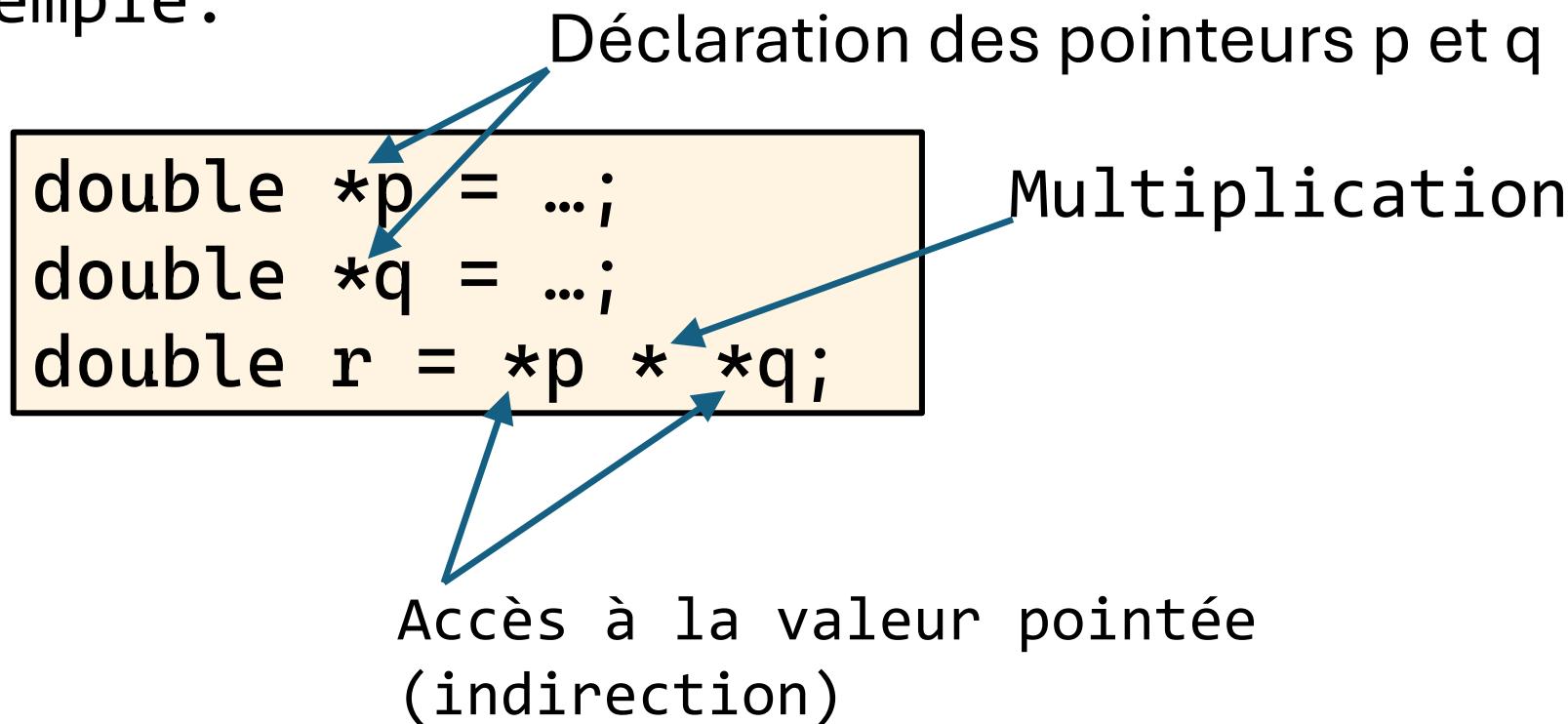
Noter que, en C, le caractère * a plusieurs significations, notamment :

- Opérateur de multiplication entre des nombres
- Bornes d'un commentaire /* ... */
- Définition d'un pointeur (int *p = ...)
- Accès à la valeur pointée par un pointeur ou indirection (*p = 3)

Un caractère * dans une instruction à l'une des significations ci-dessus en fonction du contexte.

Caractère * dans le code source (2)

Par exemple:

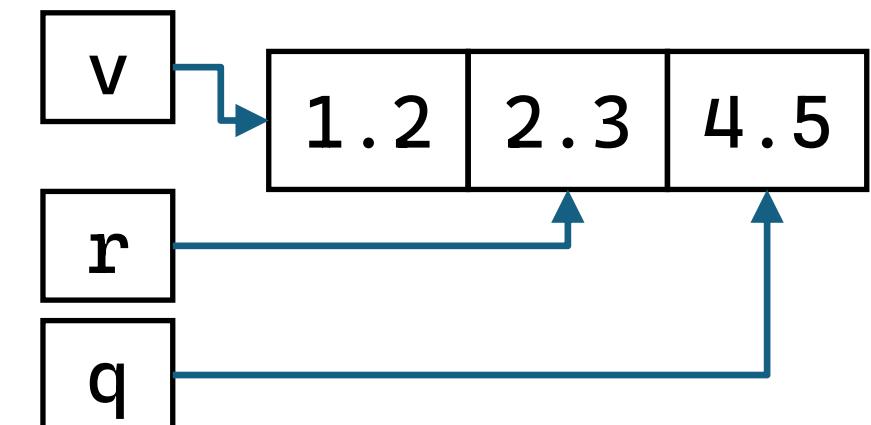


En général, le compilateur déduit la signification du contexte. Mais il peut arriver, c'est rare, que l'instruction soit ambiguë. Dans ce cas, il faut l'aider, en particulier en ajoutant des (): double r = (*p)*(*q);

Opérations avec des pointeurs

- Un pointeur contient une adresse qui est un type particulier d'entier. On peut donc imaginer de faire des opérations arithmétiques ou autres sur les pointeurs.
- Seules certaines opérations ont un sens:
 - Addition d'un entier positif à un pointeur, le résultat est un pointeur, on dit qu'on décale le pointeur vers une adresse supérieure.
 - Soustraction d'un entier positif à une adresse, on dit qu'on décale un pointeur vers une adresse inférieure

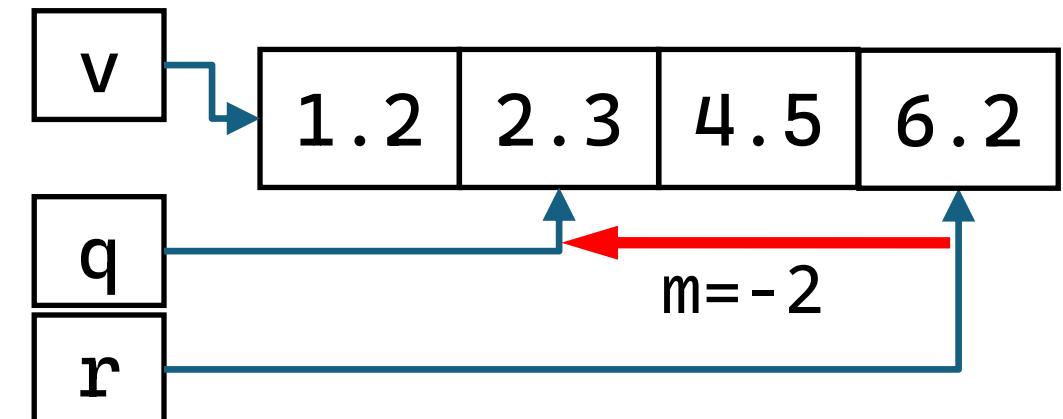
```
double v[] = {1.2, 2.3, 4.5};  
double *q = v + 2;  
Double *r = q - 1;
```



Opérations avec des pointeurs (2)

□ Différence entre 2 pointeurs

```
double v[] = {1.2, 2.3, 4.5, 6.2};  
double *q = &v[1]; /* ou q = v+1 */  
double *r = &v[3]; /* ou r = v+3 */  
int m = r - q;
```



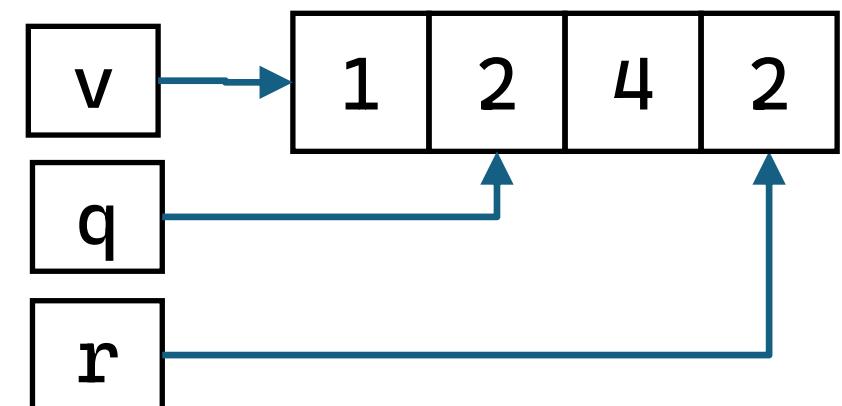
- Par contre, les opérations d'addition de 2 pointeurs, de multiplication et de division entre des pointeurs et des entiers n'ont pas de sens

Opérations avec des pointeurs (3)

□ Comparaison de deux pointeurs

```
int v[] = {1, 2, 4, 2};  
int *q = &v[1];  
int *r = &v[3];  
r == q; /* est faux */  
*r == *q ; /* est vrai */  
r > q; /* est vrai */
```

La comparaison de pointeurs consiste à comparer les adresses contenues dans les pointeurs



Affichage des pointeurs / adresses

Un pointeur contient une adresse (un entier de type `size_t`).

La fonction `printf` peut l'afficher comme un entier, mais utilise aussi un format spécifique (`%p`) :

```
double x ;  
double *p = &x ; /* p contient l'adresse de x */  
printf("adresse de x = %ld\n", p) ;  
printf("adresse de x = %p\n", p) ;
```

Ces lignes compilent mais le compilateur affiche des messages d'avertissement.

Ajouter ce qui manque pour ne plus avoir de messages du compilateur

Opérations avec des pointeurs (4)

Il vaut mieux ne pas comparer des pointeurs sur des types différents

Certains compilateurs l'acceptent (et affichent peut-être un message d'avertissement), d'autres non

Exercice: somme des composantes d'un vecteur

Le code ci-dessous initialise un vecteur de n doubles et calcule la somme des ses composantes.

Modifiez-le en utilisant uniquement des pointeurs pour accéder aux composantes du vecteur.

```
for (i=0; i<n; i++)
    v[i] = sin(i*1.0);

somme = v[0];
for (i=1; i<n; i++)
    somme += v[i];
```

Fonctions Variables locales et globales

Fonction

Une fonction possède :

- un nom,
- un ensemble de paramètres (valeurs reçues du contexte extérieur de la fonction, en spécifiant leur type)
- zéro ou un seul résultat (une unique valeur),
- un corps (un ensemble d'instructions à exécuter quand on utilise la fonction)

Une fonction peut définir des variables supplémentaires (variables locales explicites) qui n'existeront que pendant l'exécution de cette fonction.

La fonction définit pour chaque paramètre, une copie locale (variable local implicite) et travaille avec la copie locale (détruite à la fin de la fonction).

Exemple de fonction

On peut mettre les instructions qui calculent la somme des composantes d'un vecteur (exercice précédent) dans une fonction :

```
double calculeSomme(double *w, int m)
{
    double s = w[0];
    int i ;
    for (i=1; i<m; i++)
        s += w[i];
    return s ;
}
```

Type du résultat

Nom

Liste des paramètres
(types et noms internes)

Variables locales explicites (s et i)

Corps

Déclaration et définition d'une fonction

- La définition d'une fonction dans le code source est l'écriture complète de la fonction : nom, paramètres, type du résultat et corps
- La déclaration d'une fonction est l'ensemble des informations sans le corps de la fonction : nom, paramètres, type de retour

La déclaration d'une fonction est souvent appelée le **prototype de la fonction** ou la **signature de la fonction**.

Exemple de fonction (2)

L'utilisation de cette fonction, par exemple dans la fonction main (programme principal), s'écrit

```
int main()
{
    int n = ... ;
    double * v = ... ;
    double somme ;
    somme = calculSomme(v, n) ;
}
```

Le nombre et les types de paramètres doit correspondre à la définition de la fonction page précédente.

La type de résultat de la fonction doit être compatible (identique ou convertible) avec la variable où on range le résultat

Exemple de fonction (3)

Si on ne s'intéresse pas au résultat de la fonction, on ne le range pas dans une variable:

```
calculSomme(v, n) ;
```

Certains compilateur affichent un avertissement, dans ce cas, on peut écrire:

```
(void) calculSomme(v, n) ;
```

Si une fonction ne renvoie pas de résultat, on indique dans la définition que le « résultat » est de type void:

```
void calculSomme(double *v, int n)
{
    ...
}
```

Opérations effectuées à l'exécution de la fonction

1. Un premier ensemble de variables locales sont automatiquement créées qui contiennent une copie des valeurs passées à la fonction (paramètres de la fonction).
2. Les autres variables locales (explicitement définies dans la fonction) sont créées.
3. Le corps de la fonction est exécuté.
4. Si la fonction a un résultat, la dernière instruction exécutée doit être « return valeur » qui range cette valeur dans le résultat de la fonction.
5. Les variables locales sont détruites

Fonctions ayant une forme particulière

Certaines fonctions ont un mode d'exécution particulier :

- Fonctions qui ne prennent pas de paramètres (qui n'ont pas besoin de données) pour s'exécuter
- Fonctions qui ne produisent pas de résultat
- Fonctions qui modifient des variables globales ou des valeurs indirectement à travers les paramètres

Fonctions ayant une forme particulière (2)

Fonctions qui ne prennent pas de paramètres (qui n'ont pas besoin de données du contexte) pour s'exécuter.

La fonction « Question », ci-dessous, demande à l'utilisateur de rentrer un nombre entre 1 et 10 :

```
int Question()
{
    int r;
    do {
        printf("Entrer un entier entre 1 et 10 ");
        scanf("%d", &r);
    }
    while ((r > 0) && (r < 11));
    return r;
}
```

Fonctions ayant une forme particulière (3)

Fonctions qui ne produisent pas de résultat.

Par exemple, la fonction ci-dessous affiche un vecteur:

```
void AfficheVecteur(double *v, int n, char *nom)
{
    int i;
    printf("%s\n", nom);
    for (i=0; i<n; i++)
        printf("%3d: %g\n", i, v[i]);
    printf ("\n");
}
```

Dans ce cas, il n'y a pas d'instruction « return valeur » (ou alors « return; »)

Fonctions ayant une forme particulière (4)

Fonctions qui produisent, en plus de la valeur de retour, d'autres résultats.

Les cas les plus courants sont:

- Une fonction modifie une variable globale
- Une fonction reçoit un pointeur en paramètre et modifie la valeur pointée par le pointeur

Ce sont des cas particuliers de ce qu'on appelle en informatique un « effet de bord » (side effect en anglais)

Fonctions ayant une forme particulière (5)

Exemple d'une fonction qui modifie une variable globale:

```
int nAppels; /* variable globale */

void f(double x) {
    nAppels += 1;
    printf("x = %g\n" , x);
}

int main() {
    nAppels = 0;
    f(1.0); f(2.0);
    printf("La fonction f a été appelée %d fois\n", nAppels);
    return 0;
}
```

Fonctions ayant une forme particulière (6)

La fonction ci-dessous modifie les composantes d'un vecteur passé en paramètre:

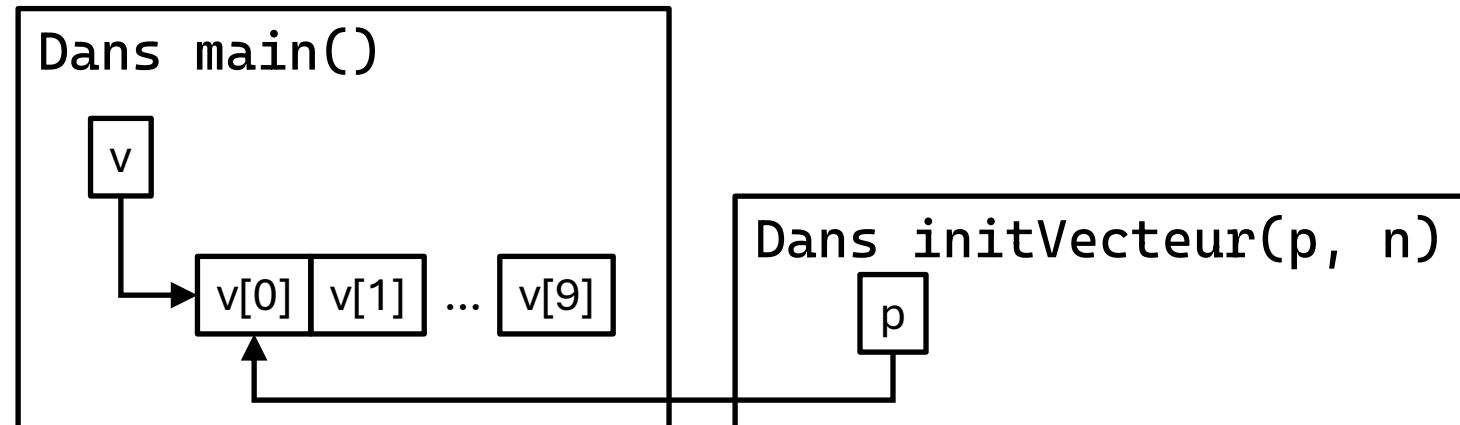
```
void initVecteur(double * p, int n) {
    int i;
    for (i=0; i<n; i++) p[i] = 0.0;
}

int main() {
    double *v = (double *) malloc(10*sizeof(10));
    initVecteur(v, 10);
    free(v);
    return 0;
}
```

Fonctions ayant une forme particulière (7)

Quand on rentre dans la fonction initVecteur:

- La fonction crée un pointeur local p et y copie l'adresse du vecteur v passé en paramètre dans le programme principal
- La fonction utilise le pointeur p pour utiliser ou modifier les composantes du vecteur
- Le pointeur local p est détruit automatiquement à la fin de la fonction



Un code C est constitué :

- d'un ensemble de fonctions, chaque fonction définit des variables qui n'existent que dans cette fonction, appelées variables locales
- de variables définies en dehors des fonctions, appelées variables globales ; ces variables existent pendant toute la durée de l'exécution

Quand le compilateur compile un code C :

- Le compilateur lit le code source de la première ligne à la dernière, une et une seule fois
- Quand une variable ou une fonction sont utilisées, il faut que cette variable soit déclarées/définies avant leur utilisation

Exercice :

```
#include <stdio.h>

void g(int b) {
    a = a * b;
}

int a;

void f(int c) {
    a = a + c;
}
```

```
int main() {
    a = 0;
    f(2); g(5);
    return 0;
}
```

Le code contient 3 fonctions (main, f, g), une variable globale (a) et deux variables locales implicites (b et c, paramètres dans f et g).

Il y a une erreur dans ce code,
corrigez cette erreur.

Langage C avancé : Séance 4

Représentation mémoire des types de base

Opérations arithmétiques sur ordinateur

Marc TAJCHMAN

@-mail : marc.tajchman@cea.fr

CEA - DES/ISAS/DM2S/STMF/LDEI

Objectifs de la séance

On examinera ici

- Comment les différents types de données sont représentés en mémoire
- Dans quelles parties de la mémoire sont rangées les fonctions, variables locales, variables globales
- Des contraintes sur le choix par le système des adresses mémoire
- Quand on appelle une fonction, l'organisation mémoire quand on y rentre, à l'intérieur et quand on sort de la fonction

Représentation mémoire des types simples

Nombre binaire (bit) et octet (byte)

Un bit est une donnée pouvant prendre une des valeurs 0 ou 1 (ou vrai/faux, ou haut/bas, etc.).

L'octet est l'unité mémoire la plus petite manipulée (lue, modifiée) par le système. Cette zone mémoire est capable de contenir une suite de 8 bits.

Un octet peut donc représenter $256 = 2^8$ suites différentes de 8 valeurs 0 ou 1.

La mémoire de travail d'un ordinateur est constituée d'un grand nombre d'unités mémoire de ce type.

Octet

On prendra ici l'exemple de l'octet :

1	0	1	1	0	0	1	1
---	---	---	---	---	---	---	---

La valeur de cet octet peut être vue comme

➤ un entier entre 0 et 255 (somme de puissances de 2), il y a deux possibilités:

- Little-endian (puissances de 2 croissantes)

$$1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 0 \times 2^5 + 1 \times 2^6 + 1 \times 2^7 = 205$$

- Big-endian (puissances de 2 décroissantes)

$$1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 179$$

Octet (2)

Toujours avec l'exemple 

- un entier entre -128 et +127 (somme de puissances de 2),
un des bits indique si l'entier est positif ou négatif:

- Little-endian

dernier bit = 1 : entier négatif, 0 : entier positif

$$1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 0 \times 2^5 + 1 \times 2^6, \text{ 1:signe-} = -77$$

- Big-endian

premier bit = 1 : entier négatif, 0 : entier positif

$$1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + 0 \times 2^4 + 0 \times 2^5 + 1 \times 2^6 + 1 \times 2^7 = -51$$

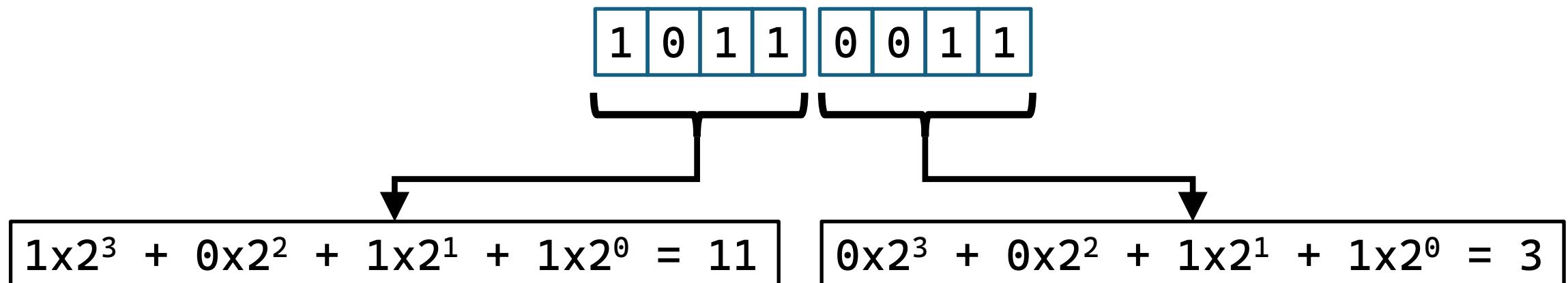
Le système le plus courant est big-endian

Le système le plus courant est big-endian, dans la suite on utilisera uniquement ce système.

Octet (3)

Toujours avec l'exemple  1 0 1 1 0 0 1 1

- Deux entiers entre 0 et 15 (chacun regroupe 4 bits) :



La valeur de l'octet est $2^4 \times$ la valeur du premier entier + $2^0 \times$ la valeur du second entier :

$$2^4 \times 11 + 3 = 179$$

On utilise plutôt la notation hexadécimale (base 16) :

$$10 = A, 11 = B, 12 = C, 13 = D, 14 = E, 15 = F$$

L'entier dans l'octet est noté B3.

Type C pour définir un octet

Il n'existe pas de type C spécifique pour définir un octet, les possibilités sont:

- Utiliser les types « `char` » ou « `unsigned char` ».

En général, les valeurs de ce type occupent une zone mémoire de 1 octet (mais il y a des exceptions, par exemple les DSP - processeurs de signaux numériques: son, etc.)

- Utiliser les types « `int8_t` » ou « `uint8_t` ».

Ces types ne sont pas toujours définis (il faut un compilateur C assez récent, qui suit la norme C99).

Il faut inclure `stdint.h`

Opérations sur un octet

Un octet peut être vu comme un entier (entre 0 et 255, ou entre -128 et 127).

On peut donc effectuer des opération arithmétiques (+, -, *, /) ou de comparaison sur un octet

Attention, le résultat doit se trouver dans l'intervalle 0 ... 255 (cas sans signe) ou -128 ... 127 (cas avec signe), sinon le résultat sera différent de celui attendu.

Opérations sur un octet (2)

Exemple 1 (avec uint8_t):

```
uint8_t n = 70;
uint8_t m = 90;
uint8_t r = n+m;
printf("n = %u m = %u n+m = %u\n", n, m, r);
```

Le résultat est bien celui attendu (160)

Exemple 2 (avec int8_t):

```
int8_t n = 70;
int8_t m = 20;
int8_t r = n+m;
printf("n = %d m = %d n+m = %d\n", n, m, r);
```

Le résultat est bien celui attendu (90)

Opérations sur un octet (3)

Exemple 3:

```
uint8_t n = 255u;
uint8_t m = 1u;
uint8_t p = 2u*m;
uint8_t r1 = n + m;
uint8_t r2 = n + p;

printf("n = %u, m = %u, 2*m = %u\n", n, m, p);
printf("n+m = %u, n+2*m = %u\n", r1, r2);
```

On n'obtient pas les mêmes résultats qu'avec des entiers mathématiques.

Remarque: C utilise «u» pour spécifier des valeurs entières positives et %u pour les afficher.

Opérations sur un octet (4)

Exemple de dépassement de valeur limite

Variable n

```
uint8_t n = 255u;
```

Variable m

```
uint8_t m = 1u;
```

Résultat : 0

Zone mémoire

$$(n = 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0)$$

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

+

0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---

=

1	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---

Le bit supplémentaire
est jeté

Représentation d'un caractère

On vient de voir qu'un caractère est enregistré en mémoire comme un octet, donc une valeur entière.

Il y a trois types caractères:
(en fait, 2 différents, char est identique à un des autres - dépend du compilateur)

char
signed char
unsigned char

Ces types conviennent pour les caractères dans la table de caractères de base (lettres majuscules et minuscules, chiffres, caractères de ponctuation).

Examiner et exécuter le code dans le fichier `char1.c`

Représentation d'un caractère (2)

Le type `char` est indiqué quand on utilise des chaînes de caractères (des variables contenant du texte)

L'un des autres types est utilisé quand on utilise des caractères comme des octets (bytes).

Si on veut garantir la portabilité^(*) du code, plutôt que `signed char` (resp. `unsigned char`), il vaut mieux utiliser les types `int8_t` (resp. `uint8_t`) (à condition qu'ils existent)

[Examiner et exécuter le code dans le fichier `char1.c`](#)

(*) le code fournit les mêmes résultats avec des compilateurs différents et/ou sur des machines différentes

Table des caractères de base

La table ASCII ci-contre est celle qui est utilisée pour représenter le jeu minimal de caractères (127) et appliquer les fonctions `strlen`, `printf`, ...

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	'
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	\$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(72	48	110	H	104	68	150	h
9	9	11		41	29	51)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	:	91	5B	133	[123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	
29	1D	35		61	3D	75	=	93	5D	135]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	-
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Tables étendues de caractères

De nombreuses tables de caractères existent pour pouvoir utiliser:

- des caractères accentués (é, É, à, ...)
- des caractères d'autres alphabets : grec, japonais, arabe, ... (β, 力, ...)

Il existe une table [ASCII étendue](#) (voir page suivante), qui contient 128 caractère supplémentaires (dont les lettre accentuée en français).

Table ASCII étendue

Cette table utilise les 256 valeurs positives d'un octet:

ASCII control characters			ASCII printable characters			Extended ASCII characters										
00	NULL	(Null character)	32	space	64	@	96	'	128	ç	160	á	192	ł	224	ó
01	SOH	(Start of Header)	33	!	65	A	97	a	129	ü	161	í	193	ł	225	þ
02	STX	(Start of Text)	34	"	66	B	98	b	130	é	162	ó	194	ł	226	ö
03	ETX	(End of Text)	35	#	67	C	99	c	131	â	163	ú	195	ł	227	ő
04	EOT	(End of Trans.)	36	\$	68	D	100	d	132	ä	164	ñ	196	—	228	õ
05	ENQ	(Enquiry)	37	%	69	E	101	e	133	à	165	Ñ	197	ł	229	ö
06	ACK	(Acknowledgement)	38	&	70	F	102	f	134	å	166	ª	198	ã	230	µ
07	BEL	(Bell)	39	'	71	G	103	g	135	ç	167	º	199	Ã	231	þ
08	BS	(Backspace)	40	(72	H	104	h	136	ê	168	ξ	200	Ł	232	þ
09	HT	(Horizontal Tab)	41)	73	I	105	i	137	ë	169	®	201	Ł	233	ú
10	LF	(Line feed)	42	*	74	J	106	j	138	è	170	¬	202	Ł	234	ú
11	VT	(Vertical Tab)	43	+	75	K	107	k	139	ï	171	½	203	Ł	235	ú
12	FF	(Form feed)	44	,	76	L	108	l	140	î	172	¼	204	Ł	236	ý
13	CR	(Carriage return)	45	-	77	M	109	m	141	ì	173	í	205	=	237	Ý
14	SO	(Shift Out)	46	.	78	N	110	n	142	Ä	174	«	206	†	238	—
15	SI	(Shift In)	47	/	79	O	111	o	143	Å	175	»	207	¤	239	‘
16	DLE	(Data link escape)	48	0	80	P	112	p	144	É	176	⋮	208	ð	240	Ξ
17	DC1	(Device control 1)	49	1	81	Q	113	q	145	æ	177	⋮	209	đ	241	±
18	DC2	(Device control 2)	50	2	82	R	114	r	146	Æ	178	⋮	210	È	242	≡
19	DC3	(Device control 3)	51	3	83	S	115	s	147	ô	179	⋮	211	È	243	¾
20	DC4	(Device control 4)	52	4	84	T	116	t	148	ö	180	⋮	212	È	244	¶
21	NAK	(Negative acknowl.)	53	5	85	U	117	u	149	ò	181	À	213	í	245	§
22	SYN	(Synchronous idle)	54	6	86	V	118	v	150	û	182	À	214	í	246	÷
23	ETB	(End of trans. block)	55	7	87	W	119	w	151	ù	183	À	215	í	247	.
24	CAN	(Cancel)	56	8	88	X	120	x	152	ÿ	184	⌚	216	í	248	°
25	EM	(End of medium)	57	9	89	Y	121	y	153	Ö	185	⋮	217	í	249	..
26	SUB	(Substitute)	58	:	90	Z	122	z	154	Ü	186	⋮	218	Γ	250	·
27	ESC	(Escape)	59	;	91	[123	{	155	ø	187	⋮	219	█	251	¹
28	FS	(File separator)	60	<	92	\	124		156	£	188	⋮	220	█	252	³
29	GS	(Group separator)	61	=	93]	125	}	157	Ø	189	¢	221	—	253	²
30	RS	(Record separator)	62	>	94	^	126	~	158	×	190	¥	222	—	254	■
31	US	(Unit separator)	63	?	95	-			159	f	191	ł	223	█	255	nnbsp
127	DEL	(Delete)							160	ñ	192	ł	224	ñ		

Tables étendues de caractères (2)

Une des tables les plus complètes est la table **Unicode** (+ 1 million de codes disponibles, ~15% attribués à des caractères actuellement).

Voir, par exemple: <https://symb1.cc/fr/unicode-table/>

Par exemple, code(A) = 65 (identique code ASCII),
code(β) = 03B2, (code hexadécimal)
code(力) = 30AB

Codage étendu

Plusieurs codages ont été proposés pour représenter des caractères de tables étendues.

Le codage **UTF8** est utilisé dans beaucoup de pages internet.

Ce codage utilise un nombre variable d'octets pour représenter un caractère par un code entier fourni par la table Unicode.

Il faut appeler des fonctions C spécifiques, non standard, pour l'utiliser (voir, par exemple, **libunistring** <https://www.gnu.org/software/libunistring/>).

Dans ce cours, on n'approfondira pas ce point.

Types entiers

C définit de multiples types d'entiers. En fonction de l'ensemble des valeurs que l'on veut représenter, on choisira:

`short`
`int`
`long`
`long long`

Chacun de ces types à une version « signed » (exemple `signed long`) et une version « unsigned » pour des nombres positifs (exemple `unsigned int`)

Ce qui donne 12 types (en fait 8 types différents), la version `signed` est prise par défaut.

Types d'entiers (2)

La taille et la plage de valeurs d'entiers (short, int, long long long) ne sont pas normalisées en C, chaque machine/compilateur peut choisir celles qu'il propose.

La seule certitude c'est que la taille est croissante.

Examiner et exécuter le fichier `entier1.c` qui affiche les tailles des types d'entier et leur intervalle de valeurs sur la machine que vous utilisez.

Nouveaux types d'entier

Si on utilise d'un compilateur assez récent (qui suit la norme C99(*)), on dispose aussi des types entiers suivants:

`int8_t`
`int16_t`
`int32_t`
`int64_t`

avec les versions sans signe

`uint8_t`
`uint16_t`
`uint32_t`
`uint64_t`

Pour les utiliser, il faut inclure le fichier `stdint.h`

Vérifiez que ces types sont disponibles sur la machine et le compilateur que vous utilisez avec le fichier `entier2.c` (qui affiche aussi la taille et l'intervalle de valeurs de ces types)

Nouveaux types d'entier (2)

On a la garantie que ces types d'entier occupent exactement:

- 1 octet pour `uint8_t` et `int8_t`
- 2 octets pour `uint16_t` et `int16_t`
- 4 octets pour `uint32_t` et `int32_t`
- 8 octets pour `uint64_t` et `int64_t`

quelque soient la machine et le compilateur utilisé

Types d'entiers supplémentaires

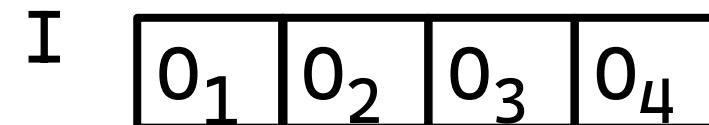
- Le type entier `size_t` suffisamment long pour pouvoir contenir une adresse mémoire.
- Les types `char` et `unsigned char` qui peuvent contenir un caractère et codés dans quasiment tous les cas sur 1 octet
- Le type `bool` qui peut représenter les valeurs "vrai" et "faux"

Traditionnellement, les compilateurs C considèrent que 0 signifie faux et tout autre valeur entière signifie vrai, ils utilisent un entier comme "type virtuel" `bool`.
Il faut inclure le fichier `stdbool.h`.

Représentation des entiers non signés

Une variable entière non signée (positive ou nulle) de n'importe quel type entier, est représentée par un nombre fixe d'octets.

Par exemple, pour variable entière non signée, codée sur 4 octets :



La valeur de l'entier (rappel, on ne considère ici que le rangement big-endian) est

$$\begin{aligned}\text{Valeur}(I) &= \text{Valeur}(0_1) 256^3 \\ &+ \text{Valeur}(0_2) 256^2 \\ &+ \text{Valeur}(0_3) 256^1 \\ &+ \text{Valeur}(0_4) 256^0\end{aligned}$$

Représentation des entiers non signés (2)

La valeur maximale de l'entier non signé, codé sur 4 octets, est 256^4 (= 4294967296) et la valeur minimale 0

Représentation des entiers signés

Une variable entière signée (positive, négative ou nulle) de n'importe quel type entier, est représentée par un nombre fixe d'octets.

Par exemple, pour une variable entière signée, codée sur 2 octets (ou 16 bits) :

I

1	1	0	1	0	1	1	0	1	1	0	0	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Le premier bit indique le signe (négatif : 1, positif 0)

Les autres bits donnent la valeur de l'entier

Représentation des entiers signés (2)

La valeur maximale d'un entier signé est la moitié de la valeur maximale d'un entier non signé de même taille.

Il faut en effet attribuer une partie des contenus possibles aux valeurs négatives.

Opérations sur les entiers

Les opérations possibles sur les entiers de même type sont les opérations arithmétiques

$$c=a+b, c=a-b, c=a*b, c=a/b$$

avec a, b, c des entiers du même type.

Ces opérations donnent un **résultat exact**, sauf:

- Si le résultat est trop grand (ou trop petit) pour pouvoir le ranger dans l'entier qui reçoit le résultat.
- Dans le cas de la division, si a n'est pas un multiple de b , le résultat sera un entier de même type proche du (mais pas égal au) résultat exact (qui n'est pas entier)

Débordement d'entier

Si a et b sont deux entiers du même type, la valeur de l'une des opérations $a+b$, $a-b$ ou $a * b$ pourrait ne pas entrer dans l'intervalle des valeurs possibles de ce type entier. On parle de **débordement (overflow)**.

Exemple: si a et b et c sont de type entier signé codé sur 2 octets (short ou `int16_t`),
 $a = 200$, $b = -300$, $a * b = -60000$
n'est pas représentable avec c .

En général, le système ne prévient pas si un débordement a eu lieu

Débordement d'entier (2)

Repérer des débordements de ce type est compliqué et peut ralentir le calcul.

En général, on essaie de le faire sur des données fournies par l'utilisateur ou lues dans des fichiers, etc.

- Un test possible pour repérer cela est d'écrire:

```
int16_t a = 200, b = -300, c;

if (a >= INT16_MAX/b || a <= INT16_MIN/b)
    printf("erreur ...");
else
    c = a*b;
```

Débordement d'entier (3)

- ou passer, temporairement si c'est possible, dans un type entier plus grand:

```
int16_t a=200, b=-300, c;
int32_t temp = ((int32_t) a) * ((int32_t) a);

if (temp > (int32_t) INT16_MAX ||
    temp < (int32_t) INT16_MIN))
    printf("erreur ...");
else:
    c = (int16_t) temp;
```

Remarque : les conversions `int16_t <> int32_t` sont inutiles (effectuées sans pertes implicitement par le compilateur), sauf la dernière (certains compilateurs affichent un message)

Division d'entiers

Une division d'un entier par un autre donne comme résultat un entier, qui peut ne pas être exact.

Exemples: si x et y sont deux entiers (signés):

```
x = 10; y = 4; x/y == 2
x = -10; y = 4; x/y == -2
x = 2; y = 3; x/y == 0
x = -5; y = -2; x/y == 2
x = -6; y = -2; x/y == 3
```

L'algorithme de division entre entiers calcule l'entier le plus grand entre la division de la valeur absolue de x par celle de y (le résultat est positif)

Puis, change le signe de ce résultat si x a un signe différent de celui de y .

Division d'entiers (2)

Ce n'est pas considéré comme une erreur, puisque le résultat mathématique n'est pas entier.

Si on veut avoir le résultat exact (en tout cas plus précis), il faut convertir les deux entiers (au moins un des entiers) en un des formats réels fournis par C. Dans ce cas, le résultat est lui aussi réel.

```
int x = ... ,y = ... ;  
double r;  
  
r = ((double) x)/((double) y);  
r = x/((double) y);  
r = (1.0*x)/y;
```

Ces trois versions donnent un résultat identique (on force la conversion au moins partielle int -> double)

Types réels

C fournit 3 types de nombres réels (avec partie entière et décimale) : **float**, **double**, **long double**. Tous ces types sont signés (il n'existe pas de type réel positif).

Les capacités de ces types varient (ce n'est pas normalisé), mais la règle est :

- taille(long double)
 - > taille(double)
 - > taille(float)
- précision(long double)
 - > précision(double)
 - > précision(float)
- intervalle de valeurs(long double)
 - > intervalle de valeurs(double)
 - > intervalle de valeurs(float)

Types réels (2)

Les types réels ne sont pas spécifiés dans la norme du langage, mais il existe la norme IEEE754 (IEEE : Institute of Electrical and Electronics Engineers) qui définit un format standard pour les différents types reel.

Presque tous les constructeurs de matériel informatique et compilateurs respectent cette norme.

IEEE754 définit non seulement les types de données mais aussi comment faire des opérations arithmétiques, arrondir les résultats, etc.

D'autres langages que C utilisent ces types.

Dans ce qui suit, on ne parlera que des types définis dans cette norme.

Types réels (3)

Exécuter le code C dans le fichier reel1.c pour afficher les caractéristiques de type réels sur la machine que vous utilisez.

Remarques sur ce qui est affiché:

- Min et max sont les valeurs minimale et maximale (en valeur absolue) différentes de zéro
- Epsilon est la valeur positive minimale telle que
- $1.0 + \text{epsilon} <> \text{epsilon}$

Réels normalisés

Dans la mémoire, un nombre réel est enregistré avec un format en base 2 normalisé

$$R = \pm 0.X_0X_1X_2\dots 2^{\pm Y_0Y_1\dots}$$

Où $X_0, X_1, \dots, Y_0, Y_1, \dots$ sont des nombres binaires (0 ou 1), mais où **X_0 doit être égal à 1** (c'est la normalisation, l'exposant est calculé en conséquence)

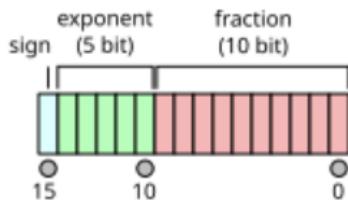
X_0, X_1, \dots est appelé la mantisse (ou fraction en anglais), les chiffres binaires significatifs et Y_0, Y_1, \dots est l'exposant

$$\begin{aligned} \text{Exemple : } 2.75 &= 2 + 0.5 + 0.25 = 1*2^1 + 1*2^{-1} + 1*2^{-2} \\ &= (2^{-1} + 1*2^{-3} + 1*2^{-4})*2^2 \\ &= 0.1011 \ 2^{10} \ (\text{normalisé base 2}) \end{aligned}$$

Types réels IEEE754 – correspondance C

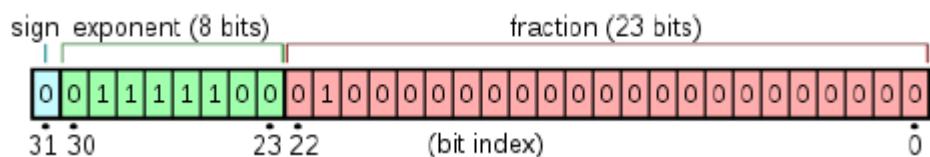
La norme IEEE754 définit 5 types de réels:

- Demi-précision (2 octets, précision ~4 décimales)



Utilisé pour des usages qui n'ont pas besoin de beaucoup de précision (rendus sur carte graphique, machine learning)

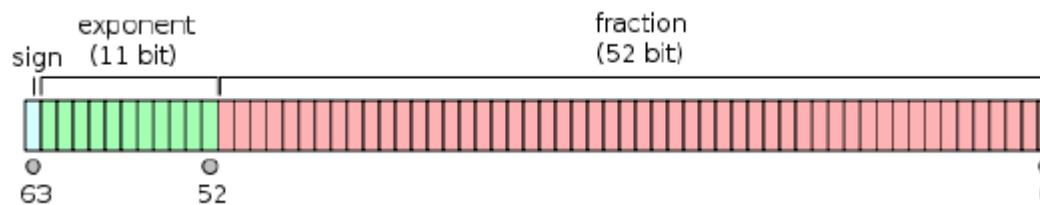
- Simple précision (4 octets, précision ~7 décimales)



Correspond souvent au type C float

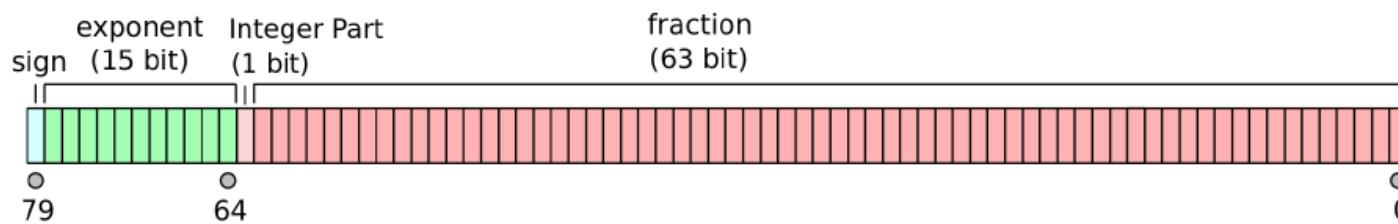
Types réels IEEE754 – correspondance C

- Double précision (8 octets, précision ~16 décimales)



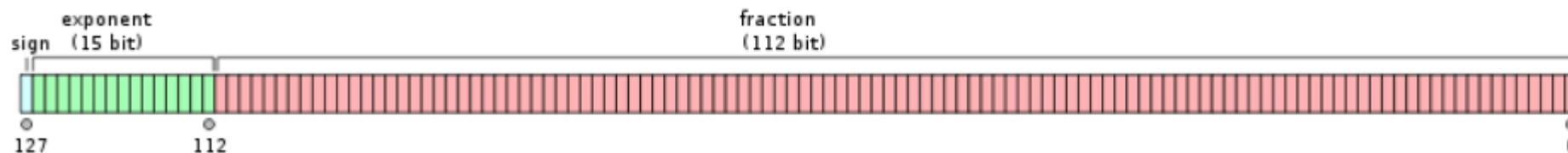
Correspond souvent au type C
double

- Précision étendue (10 octets, précision ~18 décimales)



Utilisé en interne dans
les processeurs pour
améliorer la double
précision

- Quadruple précision (16 octets, précision ~34 décimales)



Pas toujours disponible, si oui,
correspond au type long double

Ces types IEEE754 ont été conçus pour obtenir le meilleur compromis entre

- La meilleure précision
- Le moins de place mémoire
- La vitesse des opérations arithmétiques

- Les réels C ne peuvent pas représenter exactement tous les réels mathématiques, on parle d'erreur de troncature (on ne garde qu'un nombre fini de chiffres derrière la virgule)
 - Exemple 1 : 0.1 n'est pas représentable exactement dans la mémoire de l'ordinateur (écrire 0.1 comme somme finie de puissance de 2 n'est pas possible)
 - Exemple 2 : 0.625 est représentable exactement :
$$0.625 = 2^{-1} + 2^{-3}$$

- Les opérations arithmétiques introduisent des erreurs d'arrondi, parce que le résultat exact à besoin de plus de décimale ou qu'on soustrait des nombres proches

Par exemple: supposons qu'on calcule en base 10, avec des nombres à 3 chiffres significatifs

$$x = 0.423 \cdot 10^2 \quad (= 42.3)$$

$$y = 1.21 \cdot 10^0 \quad (= 1.21)$$

$$x + y = (0.423 + 0.0121) \times 10^2$$

$$= 0.435 \quad (\text{avec erreur d'arrondi, sur machine})$$

$$\neq 0.4351 \quad (\text{valeur exacte qui n'est pas représentable})$$

Arithmétique avec les réels C

➤ L'ordre des opérations peut être important

Par exemple, si a , b et c sont des réels mathématiques,

$$a + (b + c) \equiv (a + b) + c \equiv (a + c) + b$$

Si a , b , c sont des réels C:

$$a + (b + c) \equiv? (a + b) + c \equiv? (a + c) + b$$



Pas toujours, dépend de la
valeur de a , b , et c

Exécuter plusieurs fois le code `reels2.c` (qui combine de plusieurs façons 3 nombres pris au hasard)

Langage C avancé : Séance 5

Représentation mémoire : vecteurs et structures

Position en mémoire des données

Organisation de la mémoire utilisée par le code

Marc TAJCHMAN

@-mail : marc.tajchman@cea.fr

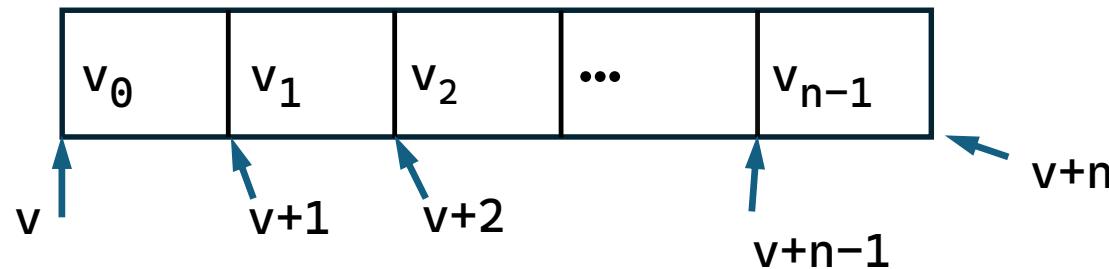
CEA - DES/ISAS/DM2S/STMF/LDEI

Représentation mémoire (suite)

Représentation des vecteurs

Un vecteur est un ensemble de n données **du même type** (composantes), on accède à une composante par son indice (un entier de 0 à $n-1$).

Les composantes d'un vecteur sont les unes à côté des autres.
Une variable v de type vecteur est un pointeur contenant l'adresse du début de la première composante.



Pour accéder à une composante d'indice i ($0 \leq i < n$), on utilise le pointeur $v + i$ (qui contient l'adresse du début de la composante) :

$$*(v + i) = 3.5 \text{ ou } v[i] = 3.5$$

Représentation des vecteurs

L'accès aux composantes du vecteur se fait en général très souvent et dans un ordre quelconque a priori, il faut donc que l'accès soit le plus rapide possible.

Ceci explique pourquoi, **les composantes d'un vecteur doivent être du même type**: l'adresse de la composante i du vecteur v est alors l'adresse du début du vecteur + $i \times$ le nombre d'octets occupés par une valeur du type des composantes

Il est inutile (et déconseillé) de calculer soi-même l'adresse d'une composante, il suffit d'utiliser l'une des 2 expressions, par exemple pour accéder à la composante d'indice 3 du vecteur de doubles v :

```
v[3] = 1.5;  
*(v + 3) = 1.5;
```

Structures

Une structure est constituée d'un ensemble **fixe** de composantes de types différents ou non.

Au contraire des types de base, une structure est un type défini par le programmeur.

Le mot clef `struct` doit être utilisé pour définir un type structure et pour déclarer des variables de ce type.

Exemple de structure :

```
struct S {  
    int n;  
    double x;  
};
```

```
struct S v;
```

On peut aussi définir un type utilisateur (un alias) :

```
struct S {  
    int n;  
    double x;  
};  
typedef struct S Vecteur;  
Vecteur v;
```

Structures

Quand on définit une structure, une zone mémoire suffisante est réservée pour contenir toutes les composantes.

La taille de cette zone est donnée par

- `sizeof(a)` si `a` est une variable de type structure
- `sizeof(struct S)` si `S` est un type structure
- `sizeof(T)` si `T` est un type structure (défini par `typedef`)

L'accès au composantes se fait par la notation :

`nom_variable.nom_composante`

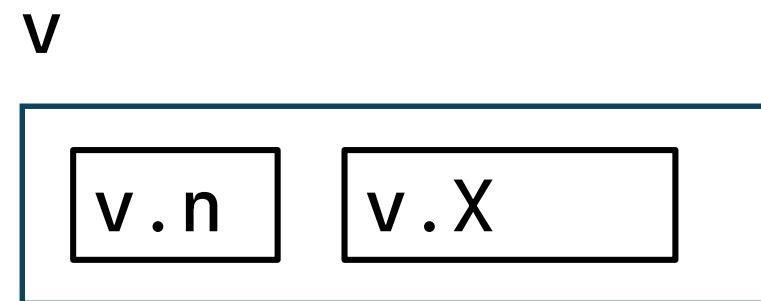
```
struct S {  
    int n;  
    double X;  
};
```

```
struct S v;  
v.n = 3;  
v.X = 1.5;
```

Représentation mémoire d'une structure

Les composantes d'une structure sont placées dans une zone mémoire de taille suffisante (il peut y avoir des espaces vides entre les composantes: plus de précisions dans la suite)

```
struct S {  
    int n;  
    double X;  
};  
struct S v;
```



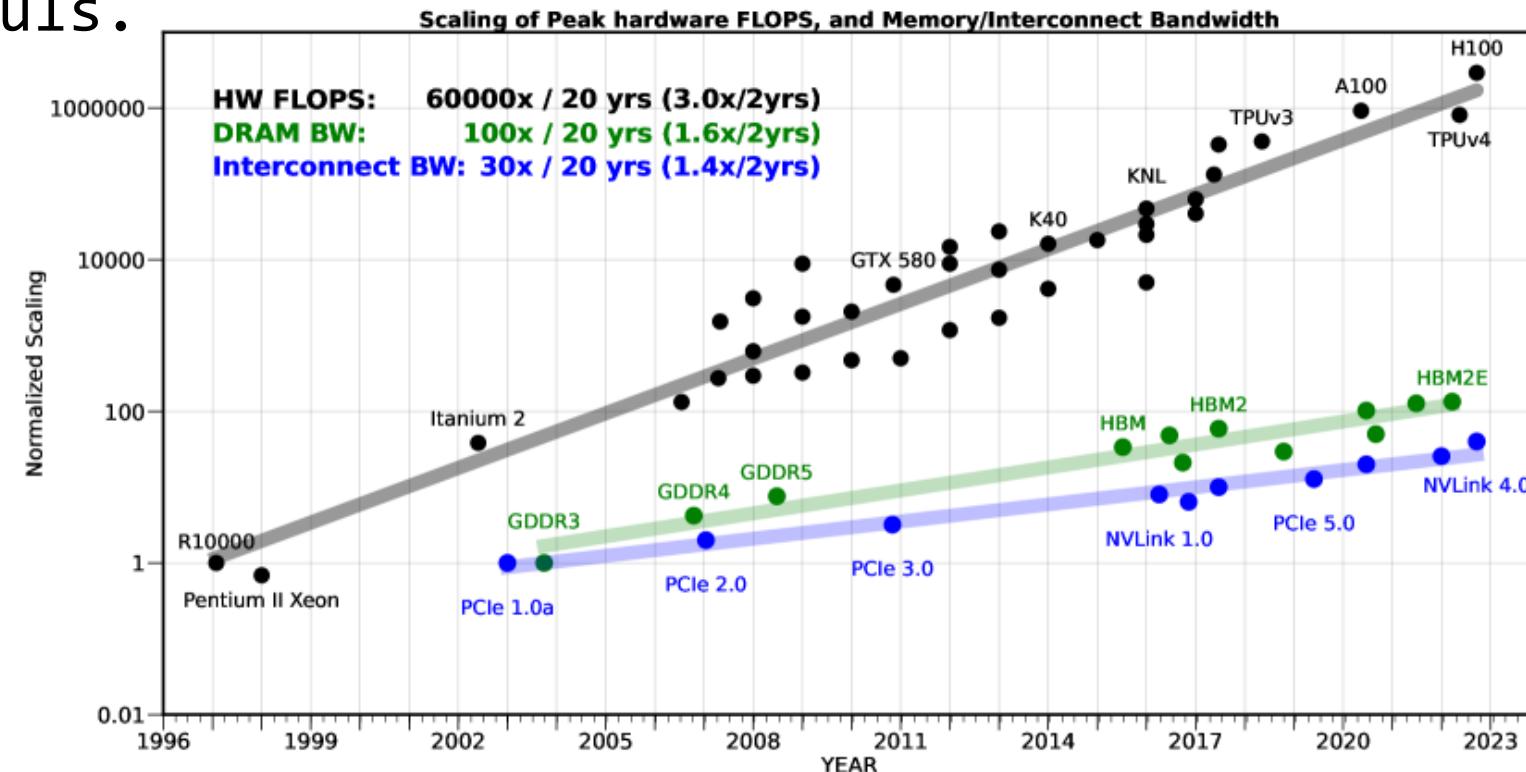
Contraintes sur les adresses mémoires

Transfert de données entre la mémoire et le processeur

Un ordinateur comporte de la mémoire connectée à un (ou plusieurs) processeur(s).

Ces dernières années, la vitesse des processeurs a augmenté beaucoup plus vite que celle de la mémoire.

On dit parfois que les processeurs passent plus de temps à attendre la réception des données ou l'envoi des résultats, qu'à effectuer des calculs.



Transfert de données entre la mémoire et le processeur

Pour essayer d'améliorer la situation, à part d'améliorer le matériel, on essaie de diminuer le nombre de transferts par rapport au nombre de calculs:

- On a plusieurs niveaux de mémoire intermédiaires entre la mémoire de travail et le processeur (de plus en plus rapide quand on est « s'approche » du processeur), **on parle de mémoire cache**

Dans ce cas, le processeur ne communique pas directement avec La mémoire de travail, mais avec une mémoire plus petite mais plus rapide : une mémoire cache

- On fait des transferts par **bloc d'octets** (appelés **lignes de cache**) plutôt que par octets individuels

Souvent, on fait des suites d'instructions de calcul qui utilisent des données proches, par exemple

for (i=0; i<n; i++) v[i] = ...

Transfert de données entre la mémoire et le processeur

Si on exécute une instruction qui a besoin de données très proches de celles utilisées par l'instruction précédente, les données ont peut-être déjà été transférées, ce qui accélère le traitement de l'instruction

Voir fichier *MemoireCache.pdf*

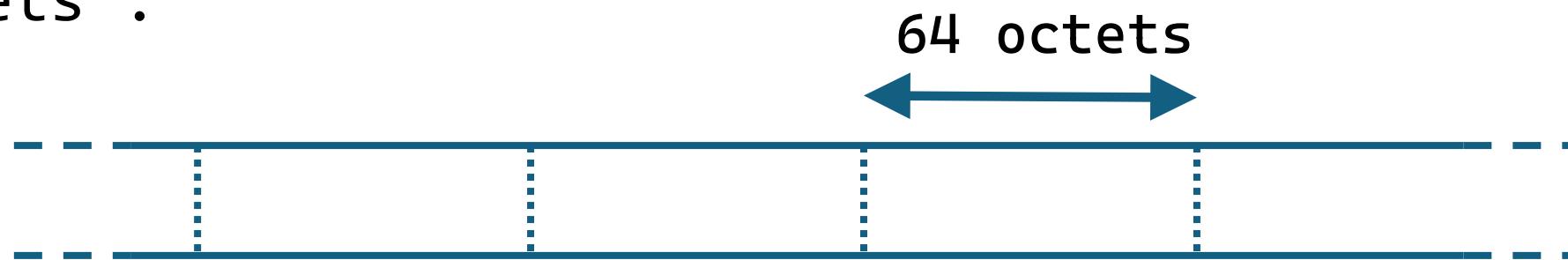
Transferts par bloc

Le processeur utilise la mémoire cache, il faut donc faire des transferts mémoire de travail - mémoire cache, dans les 2 sens.

Les transferts mémoire \leftrightarrow mémoire cache se font par blocs de la taille d'une ligne de cache.

Supposons qu'une ligne de cache a une taille de 64 octets (c'est en général le cas).

Le système considère la mémoire comme l'union de blocs de 64 octets :



Choix par le système des adresses mémoire

Quand on définit une variable, le système réserve une zone mémoire en essayant

- quand le processeur a besoin de la donnée, d'effectuer le moins possible de transferts de blocs de données
- si la variable est de type structure ou de type vecteur, que l'accès à chaque composante soit optimisé de la même façon

Chaque compilateur décide quel est le critère à appliquer pour choisir une adresse

Exemple : vecteur

Considérons une chaîne de 100 caractères.

Sans contraintes, il y a plusieurs choix pour la position de la chaîne, entre autres :

Placement 1



Placement 2



Placement 3



← 101 octets →

Pour que le processeur utilise la variable, il faudra :
3 transferts de blocs de 64 octets (placement 1), 2
transferts (placement 2) ou 2 transferts (placement 3)

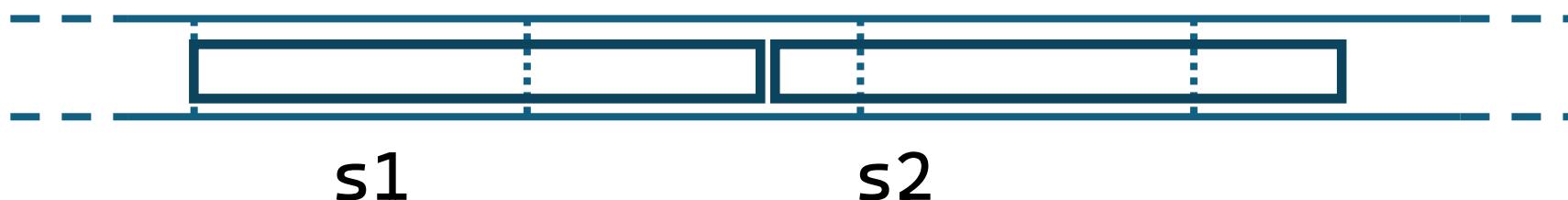
Exemple: vecteur

Donc le compilateur ne choisira pas un placement de type 1.

En fonction des variables précédemment déclarées, il pourra choisir un placement 2 ou 3, ou tout autre placement qui n'a besoin que de 2 transferts.

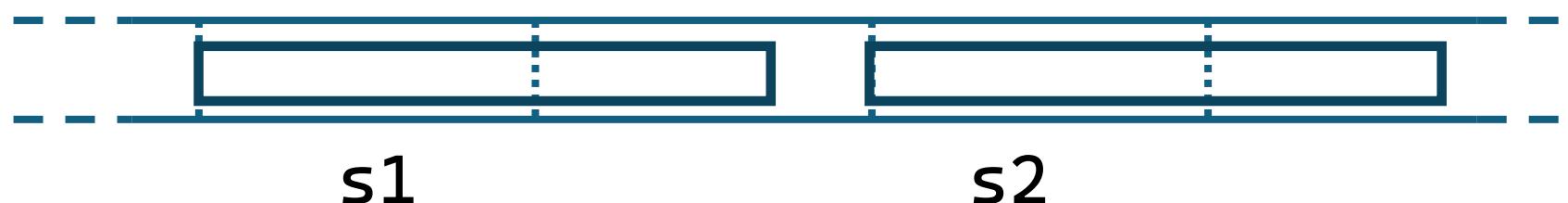
Exemple: vecteurs

Le code utilise 2 chaines s_1 et s_2 de 100 caractères, si les 2 chaines sont mises côté l'une de l'autre:



Le transfert de s_1 se fera en 2 blocs et le transfert de s_2 en 3 blocs.

Donc les compilateurs écartent s_1 et s_2 :



Pour que les transferts de s_1 et s_2 nécessitent chacun 2 blocs.

Exemple : structure

Une structure peut avoir des composantes de différents type et taille.

De plus, certaines instructions travailleront avec une partie seulement des composantes.

```
struct S {  
    int n;  
    double X;  
};  
struct S v;  
  
v.X = 1.4;
```

Il faut donc que chaque composante de la structure demande le moins de transferts possibles

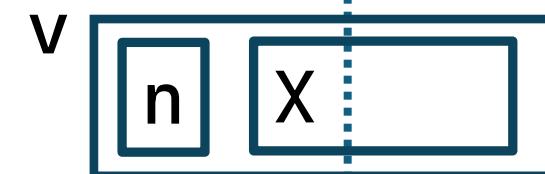
Exemple : structure

```
struct S {  
    int n;  
    double X;  
};  
struct S v;
```

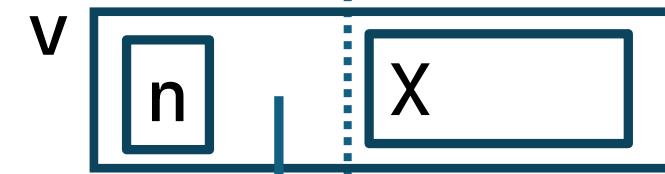
v.X = 1.4;

Parmi les possibilités:

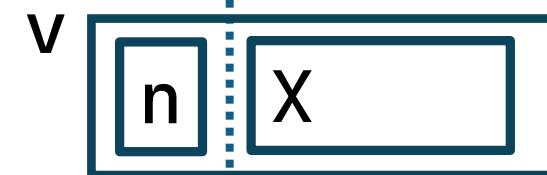
Placement 1



Placement 2



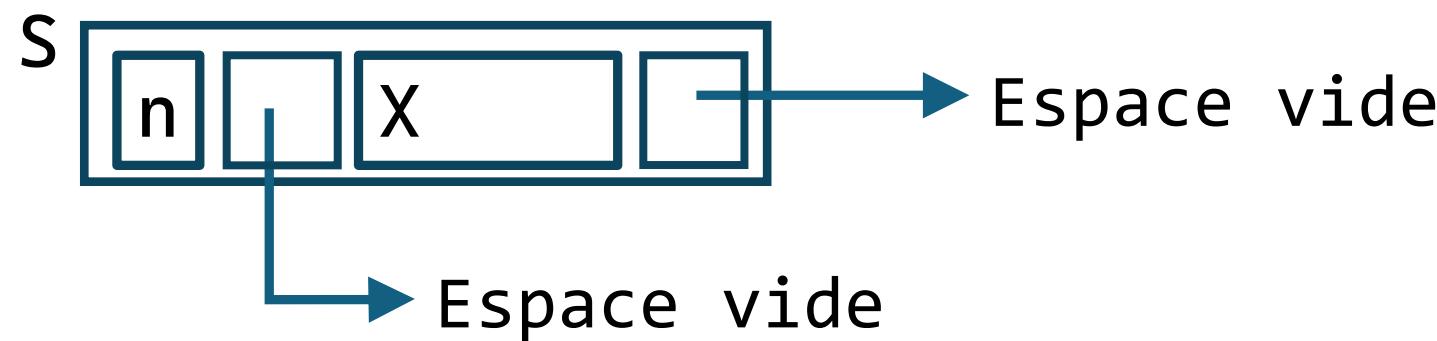
Placement 3



Espace vide

Placement des structures

Pour pouvoir s'adapter à tous les cas (structures, structures de vecteurs, vecteurs de structures), la plupart des compilateurs introduisent, **si c'est utile**, des espaces mémoire non utilisés à l'intérieur et à la fin des structures.



Padding

La technique d'insérer des espaces mémoire inutilisés vus dans les pages précédentes est appelée **padding**

Elle a un inconvénient et un avantage:

- Avantage : la vitesse d'accès aux données dans la mémoire de travail est (nettement) meilleure
- Inconvénient : les espaces mémoire inutilisés occupent de la place qui n'est plus disponible pour les données

Tous les systèmes actuels utilisent cette technique
(l'avantage est plus important)

Padding

On peut essayer d'aider le compilateur à utiliser le moins possible du padding.

La règle souvent suivie est d'ordonner les déclarations de variable et les composantes de structure : on déclare les variables/composantes par taille décroissante.

Par exemple, au lieu de :

```
struct S {  
    int n;  
    char c;  
    double X;  
};
```

On écrira:

```
struct S {  
    double X;  
    int n;  
    char c;  
};
```

Organisation de la mémoire utilisée par un code

Zones mémoire utilisées par un code

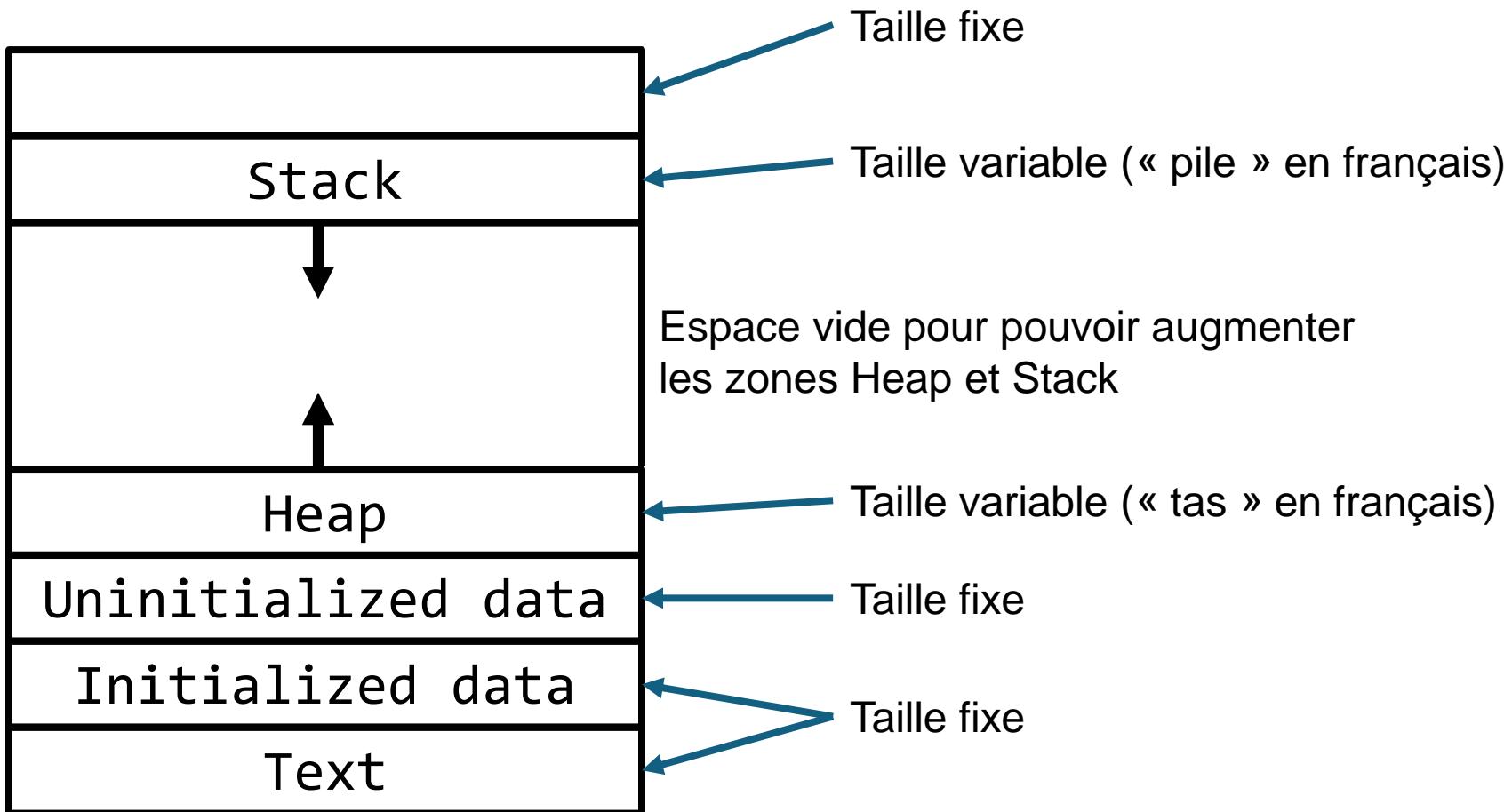
Cette partie n'est pas spécifique au langage C.

Les codes écrits en d'autres langages, en général compilés (C++, java, etc.) suivent la même organisation de la mémoire

La mémoire utilisée est répartie en plusieurs zones (on parle souvent de segments de mémoire)

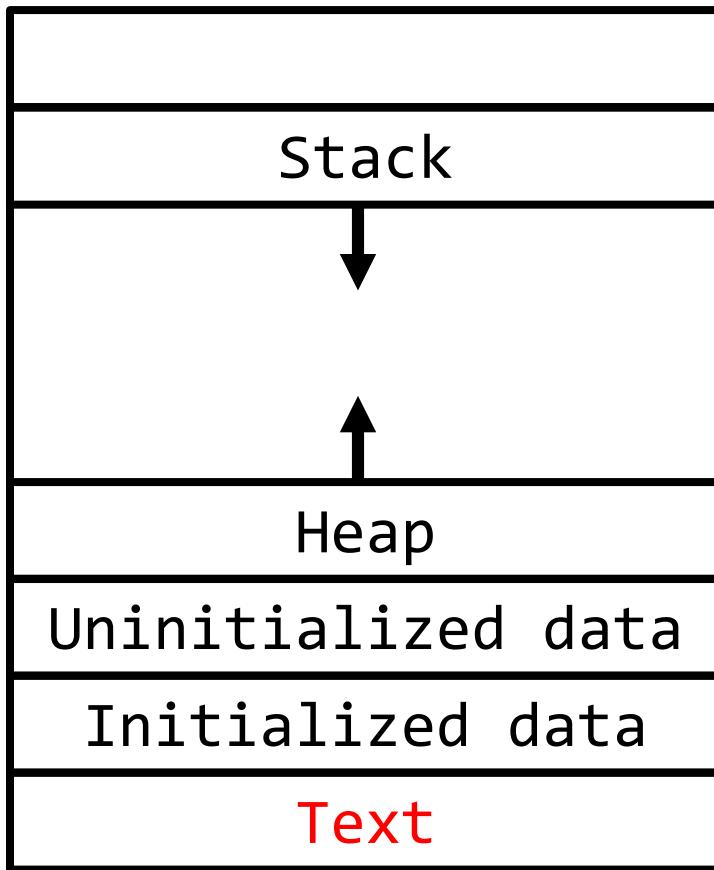
Zones mémoire utilisées par un code

La mémoire utilisée par un code C, en général, est constituée de 6 parties :



Zones mémoire utilisées par un code

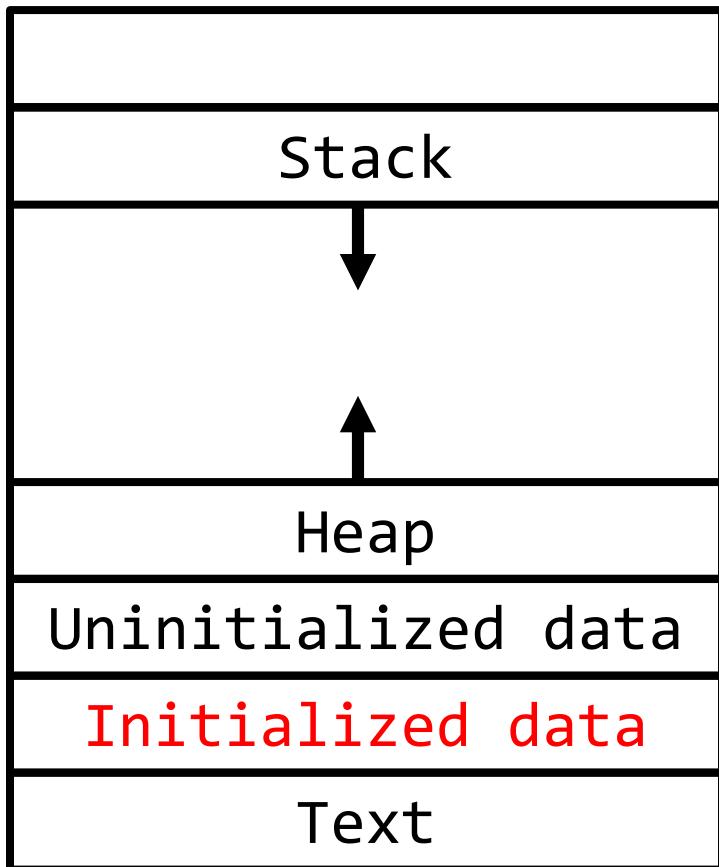
La mémoire utilisée par un code, en général, est constituée de 6 parties :



code compilé (copie du binaire contenu dans le fichier créé par le compilateur)

Zones mémoire utilisées par un code

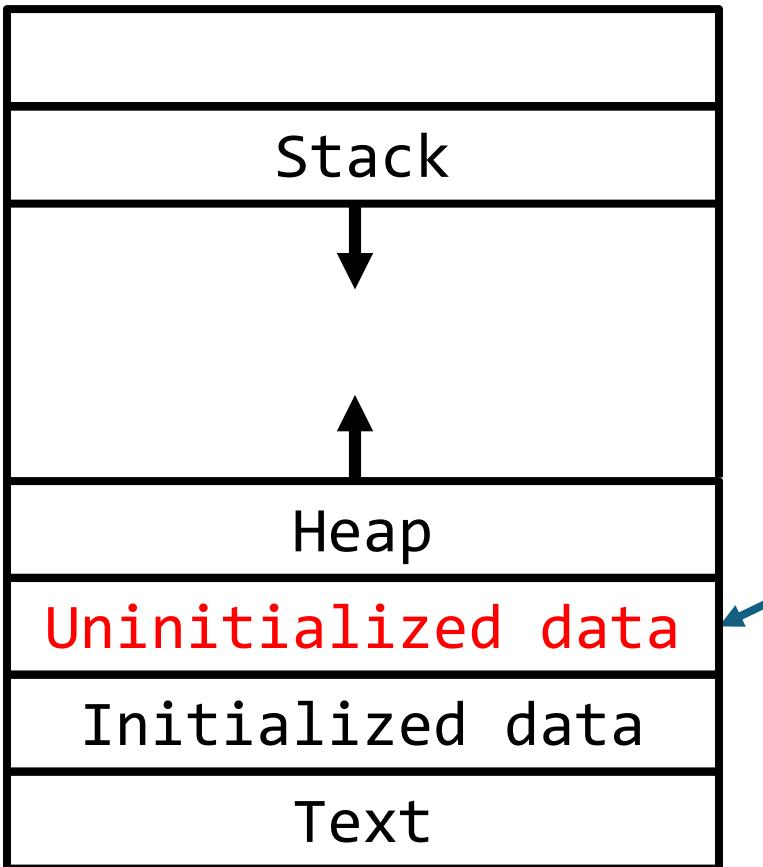
La mémoire utilisée par un code, en général, est constituée de 6 parties :



variables globales et locales statiques
dans les fonctions (initialisées),
contenues dans le fichier binaire

Zones mémoire utilisées par un code

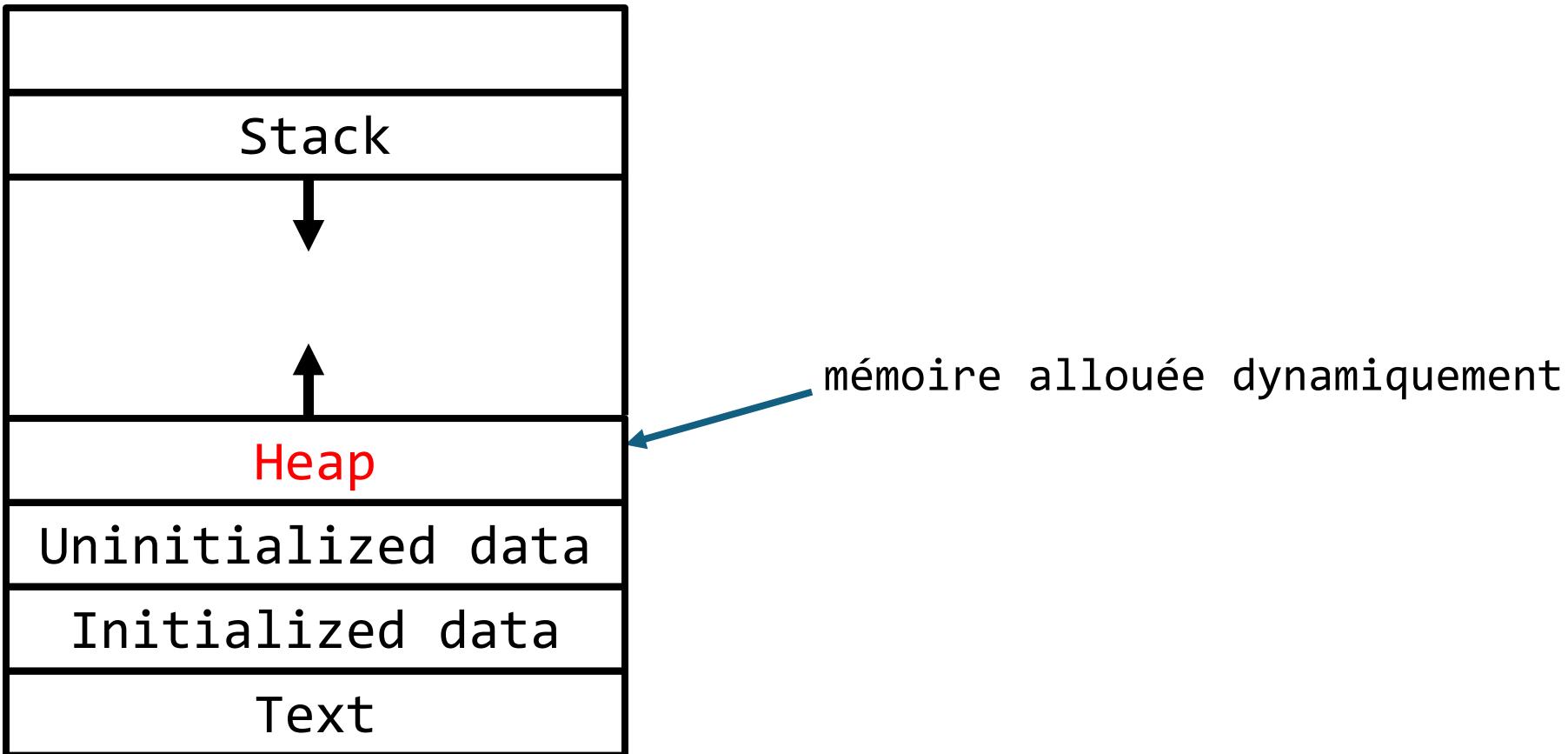
La mémoire utilisée par un code, en général, est constituée de 6 parties :



variables globales et locales statiques
dans les fonctions (non initialisées),
initialisation à 0 au démarrage de
l'exécution

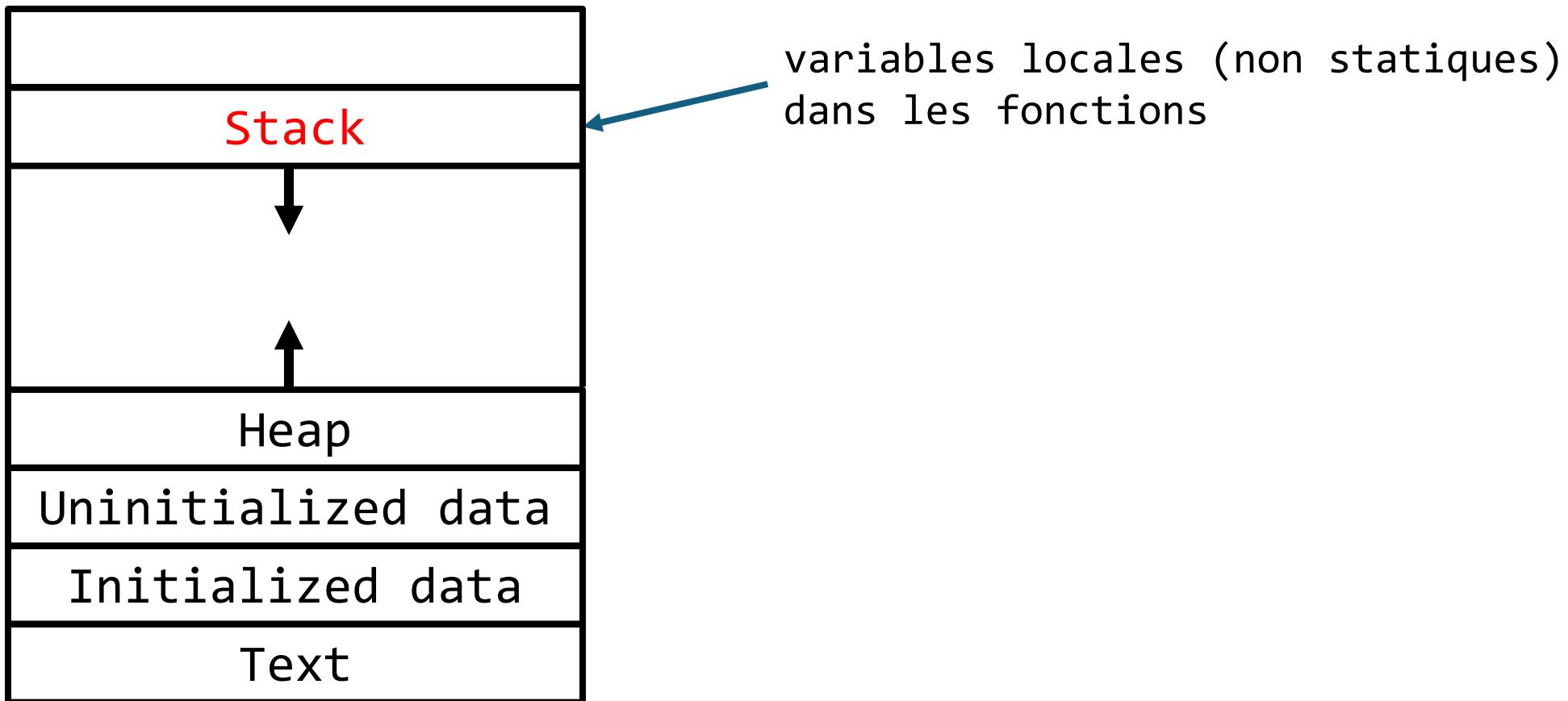
Zones mémoire utilisées par un code

La mémoire utilisée par un code, en général, est constituée de 6 parties :



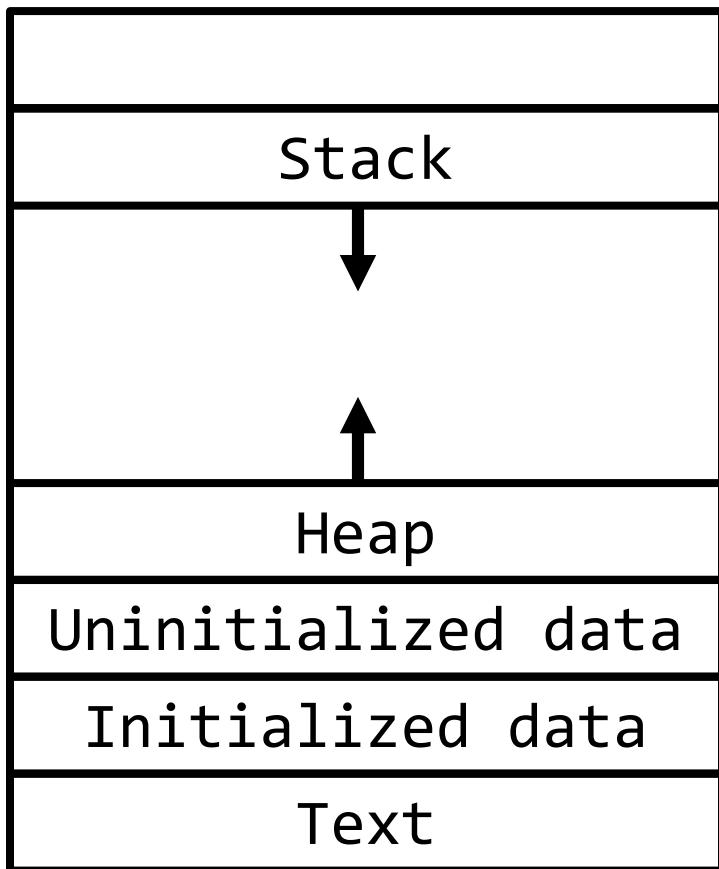
Zones mémoire utilisées par un code

La mémoire utilisée par un code, en général, est constituée de 6 parties :



Zones mémoire utilisées par un code

La mémoire utilisée par un code, en général, est constituée de 6 parties :



Arguments de la ligne de commande, valeur de retour du code (un entier) mise dans la variable d'environnement \$? (linux) et autres variables d'environnement

Zones mémoire « Text »

« Text » contient les instructions binaires produites par le compilateur.

Cette zone mémoire est copiée du fichier binaire dans la mémoire de travail de l'ordinateur.

Son contenu est fixe pendant l'exécution du code (autrement dit le code ne change pas pendant l'exécution)

Zones mémoire « Initialized data »

Cette zone mémoire est copiée du fichier binaire dans la mémoire de travail de l'ordinateur.

Elle contient (une partie) des **variables globales** et (une partie) des **variables statiques locales**.

Leur valeur initiale est contenue dans le fichier binaire mais peut changer au cours de l'exécution.

Zone « Initialized data » : Variable globale initialisée

v est une variable entière **globale** dont la valeur initiale (contenue dans le fichier) est 5 mais qui peut être modifiée pendant l'exécution:

```
int v = 5;

int main()
{
    v = v + 1;
    return 0;
}
```

Zone « Initialized data » : Variable locale statique initialisée

`v` est une **variable locale statique** entière dont la valeur initiale est contenue dans le fichier binaire.

Elle peut être modifiée seulement dans la fonction `run` et garde sa valeur entre 2 appels de la fonction `run`.

```
int run()
{
    static int v = 1
    v = v + 1;
    return v;
}
```

```
int main() {
    int w;
    w = run();
    w = run();
    print("w = %d\n", w)
    return 0;
}
```

Question: quelle est la valeur affichée par le programme principal ?

Zone mémoire « Uninitialized data »

Cette zone est de taille constante.

La différence avec la zone précédente est que le fichier binaire ne contient pas la valeur initiale des variables de cette zone.

Ces variables sont initialisées à 0 au début de l'exécution (avant de commencer le programme principal).

Pendant l'exécution, le code peut modifier leur valeur.

Zone mémoire « Stack »

Cette zone est de taille variable et est utilisée pour les données locales des fonctions y compris celle de la fonction main (le programme principal).

Registre d'instruction

Quand on exécute un code C, par exemple:

```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}
```

```
int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a)
    return 0;
}
```

L'exécution du code démarre en créant un **registre d'instruction**, pointeur positionné à l'adresse du **début du code de la fonction main** dans la zone mémoire « text ».

Chaque fois qu'une instruction du code est exécutée, le système met l'adresse du binaire de l'instruction suivante dans le registre.

L'instruction pointée par le registre est appelée « **instruction courante** ».

Evolution de la « stack » pendant l'exécution

```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}
int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a)
    return 0;
}
```

Stack

main:
:



Résultat de la fonction
main (variable locale
entière sans nom et
sans valeur)

Evolution de la « stack » pendant l'exécution

```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}
int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a)
    return 0;
}
```



Stack

main:

:

a:



b:



Evolution de la « stack » pendant l'exécution

```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}
int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a)
    return 0;
}
```



Stack

main:

:

a:

1.5

b:

Evolution de la « stack » pendant l'exécution

```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}
int main()
{
    double a, b;
    a = 1.5;
b = f(2, a);
    return 0;
}
```



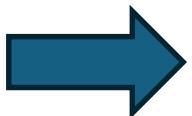
Stack

main:
:
a:
b:
f:



Pointeur :
adresse de l'instruction
courante (qui appelle f)

Evolution de la « stack » pendant l'exécution



```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}
int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a)
    return 0;
}
```

Stack

main:

:

a:

1.5

b:

f:

:

i:

2

x:

1.5

Résultat de la fonction f
(variable locale entière
sans nom et sans valeur)

Evolution de la « stack » pendant l'exécution

```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}
int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a)
    return 0;
}
```

Stack

main:

:

a:

1.5

b:

f:

:

i:

2

x:

1.5

y:

3.0

Evolution de la « stack » pendant l'exécution

```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}
int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a);
    return 0;
}
```



Stack

main:

:

a:

1.5

b:

f:

:

i:

2

x:

1.5

y:

3.0

j:

3

Evolution de la « stack » pendant l'exécution

```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}
int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a);
    return 0;
}
```

Stack

main:

:

a:

1.5

b:

f:

:

9.0

i:

2

x:

1.5

y:

3.0

j:

3

Evolution de la « stack » pendant l'exécution

```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}

int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a);
    return 0;
}
```

Stack

main:

:

a:

1.5

b:

f:

:

9.0

Les variables locales de f sont détruites ainsi que les copies locales des paramètres de f

Evolution de la « stack » pendant l'exécution

```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}

int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a);
    return 0;
}
```



Stack

main:

:

a:

1.5

b:

9.0

Le résultat de f est copié dans b

Le reste des données de f sont détruites:

- variable résultat
- adresse de retour

Evolution de la « stack » pendant l'exécution

```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}

int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a);
    return 0;
}
```

Stack

main:

:

a:

b:

0

1.5

9.0

Le résultat de f est copié dans b

Le reste des données de f sont détruites:

- variable résultat
- adresse de retour

Evolution de la « stack » pendant l'exécution

```
int f(int i, double x)
{
    double y = 2*x;
    int j = i+1;
    return y*j;
}

int main()
{
    double a, b;
    a = 1.5;
    b = f(2, a);
    return 0;
}
```



Stack

main:

:

0

La variables locales de
main sont détruites

Zone mémoire « Heap »

Cette zone est de taille variable et contient les zones mémoire allouées par malloc.

Evolution de « Heap » pendant l'exécution

On considère le code suivant qui utilise de la mémoire dynamique :

```
int main()
{
    double *v1, *v2, *v3;
    v1 = (double *) malloc(10*sizeof(double));
    v2 = (double *) malloc(10*sizeof(double));

    /* ... */

    free(v1);
    v3 = (double *) malloc(50*sizeof(double));
    free(v2);
    free(v3);
    return 0;
}
```

Evolution de « Heap » pendant l'exécution

Text (code)

```
int main()
{
    double *v1, *v2, *v3;
    v1 = (double *) malloc(10*sizeof(double));
    v2 = (double *) malloc(10*sizeof(double));

    /* ... */

    free(v1);
    v3 = (double *) malloc(50*sizeof(double));
    free(v2);
    free(v3);
    return 0;
}
```

Stack

v1:	
v2:	
v3:	

Heap



Evolution de « Heap » pendant l'exécution

Text (code)

```
int main()
{
    double *v1, *v2, *v3;
    v1 = (double *) malloc(10*sizeof(double));
    v2 = (double *) malloc(10*sizeof(double));

    /* ... */

    free(v1);
    v3 = (double *) malloc(50*sizeof(double));
    free(v2);
    free(v3);
    return 0;
}
```

Stack

```
v1: [ ]  
v2: [ ]  
v3: [ ]
```

Heap



Evolution de « Heap » pendant l'exécution

Text (code)

```
int main()
{
    double *v1, *v2, *v3;
    v1 = (double *) malloc(10*sizeof(double));
    v2 = (double *) malloc(10*sizeof(double));

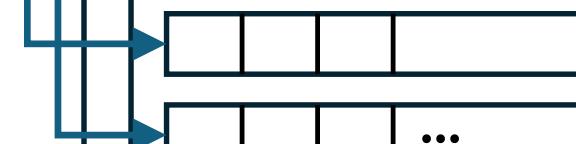
    /* ... */

    free(v1);
    v3 = (double *) malloc(50*sizeof(double));
    free(v2);
    free(v3);
    return 0;
}
```

Stack

```
v1: [ ] ---+
v2: [ ] ---+
v3: [ ] ---+
```

Heap



Evolution de « Heap » pendant l'exécution

Text (code)

```
int main()
{
    double *v1, *v2, *v3;
    v1 = (double *) malloc(10*sizeof(double));
    v2 = (double *) malloc(10*sizeof(double));

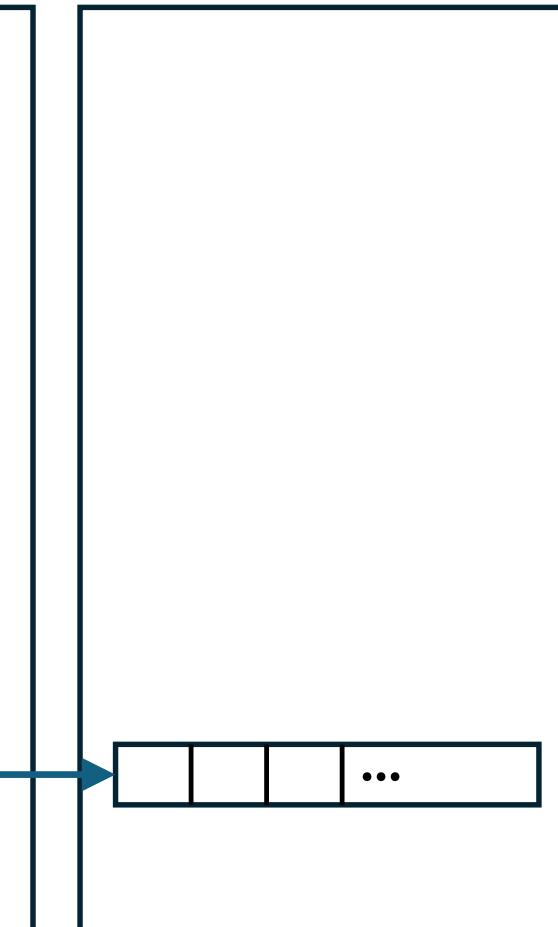
    /* ... */

    free(v1);
    v3 = (double *) malloc(50*sizeof(double));
    free(v2);
    free(v3);
    return 0;
}
```

Stack

v1:
v2:
v3:

Heap



Evolution de « Heap » pendant l'exécution

Text (code)

```
int main()
{
    double *v1, *v2, *v3;
    v1 = (double *) malloc(10*sizeof(double));
    v2 = (double *) malloc(10*sizeof(double));

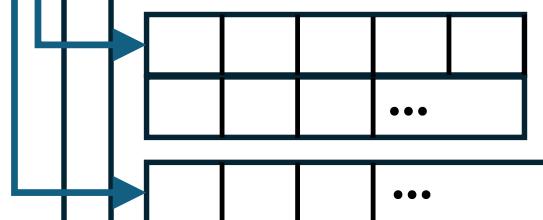
    /* ... */

    free(v1);
    v3 = (double *) malloc(50*sizeof(double));
    free(v2);
    free(v3);
    return 0;
}
```

Stack

v1:
v2:
v3:

Heap



« Trou » dans la Heap

Evolution de « Heap » pendant l'exécution

Text (code)

```
int main()
{
    double *v1, *v2, *v3;
    v1 = (double *) malloc(10*sizeof(double));
    v2 = (double *) malloc(10*sizeof(double));

    /* ... */

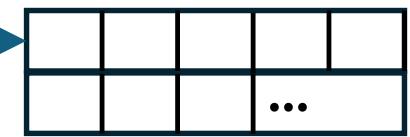
    free(v1);
    v3 = (double *) malloc(50*sizeof(double));
    free(v2);
    free(v3);
    return 0;
}
```



Stack

v1:
v2:
v3:

Heap



« Trou »
dans la Heap

Evolution de « Heap » pendant l'exécution

Text (code)

```
int main()
{
    double *v1, *v2, *v3;
    v1 = (double *) malloc(10*sizeof(double));
    v2 = (double *) malloc(10*sizeof(double));

    /* ... */

    free(v1);
    v3 = (double *) malloc(50*sizeof(double));
    free(v2);
    free(v3);
    return 0;
}
```



Stack

v1:
v2:
v3:

Heap



Morcellement de la « Heap »

On voit que au cours de l'exécution, quand des zones mémoire dynamiques sont libérées, cela peut aboutir à avoir de la mémoire disponible en plusieurs parties disjointes.

On appelle ce phénomène la **fragmentation de la mémoire**.

Tant qu'il reste une zone disponible et en un seul morceau, assez grande pour les allocations mémoire, cela ne pose pas de problème.

Sinon cela empêche de traiter des problèmes de grande taille même si la mémoire totale est théoriquement suffisante.

Pour éviter, si possible cette situation, il faut ordonner les allocations mémoire.

Morcellement de la « Heap »

Par exemple, le bloc d'instructions à gauche peut donner de la fragmentation de la mémoire, et celui de droite non

```
v1 = (int *) malloc(N*sizeof(int));
v2 = (int *) malloc(N*sizeof(int));

free(v1);

/* Ici, la mémoire est fragmentée */

free(v2);
```

```
v1 = (int *) malloc(N*sizeof(int));
v2 = (int *) malloc(N*sizeof(int));

free(v2);

/* Ici, pas de fragmentation */

free(v1);
```

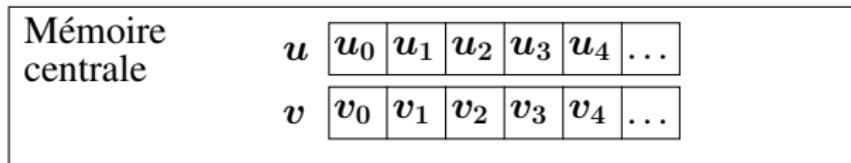
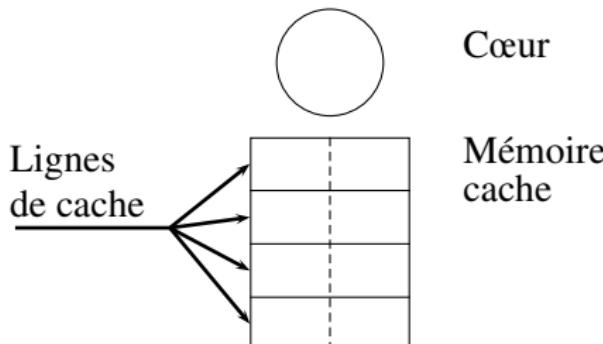
Une bonne utilisation de la mémoire est très importante pour l'optimisation de code.

Exemple : si u et v sont des vecteurs de taille $n > 4$, on veut calculer la boucle :

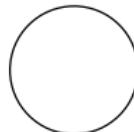
$$\begin{aligned} v_0 &= u_0 \\ v_n &= u_n \\ \text{for}(i = 1; i < n - 1; i++) \\ v_i &= (u_{i-1} + 2 * u_i + u_{i+1})/4; \end{aligned} \tag{1}$$

Supposons un système **idéalisé** par:

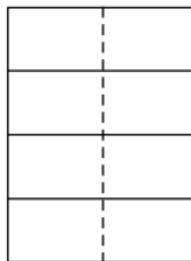
- ▶ un processeur (qui contient un seul cœur),
- ▶ un seul niveau de mémoire cache de taille 8 nombres réels (réparti en 4 lignes de cache de taille 2 nombres réels),
- ▶ la mémoire centrale



1. Avant d'exécuter $v_1 = (u_0 + 2 * u_1 + u_2) / 4$:



Cœur

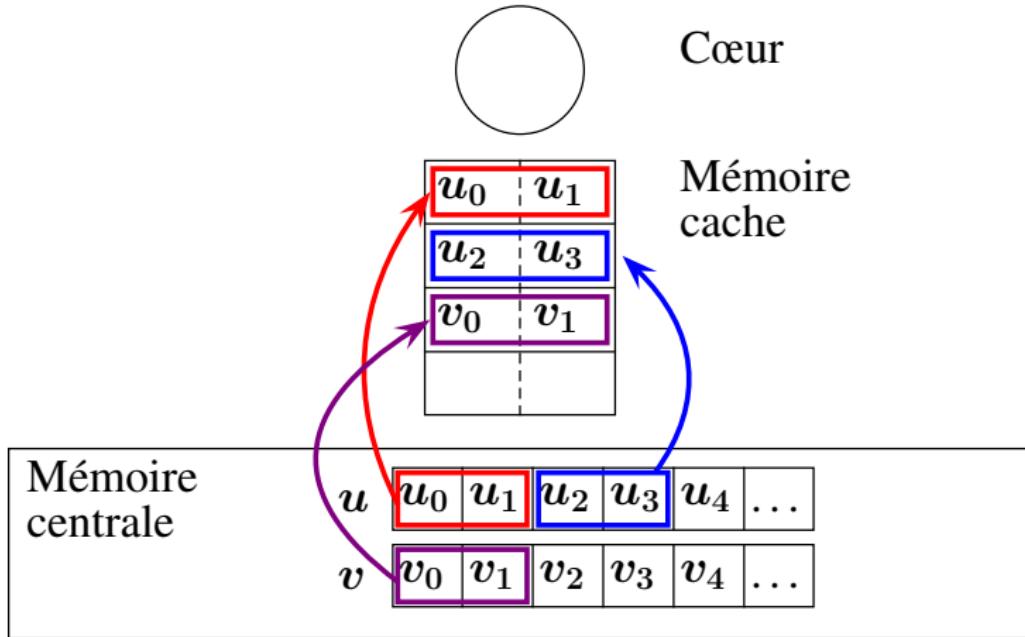


Mémoire
cache

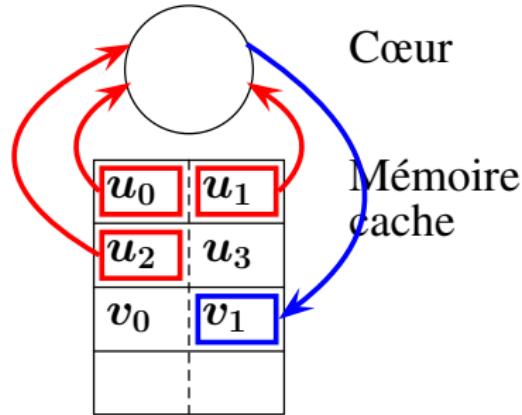
Mémoire
centrale

u	u_0	u_1	u_2	u_3	u_4	\dots
v	v_0	v_1	v_2	v_3	v_4	\dots

2. Les blocs contenant u_0 , u_1 , u_2 et v_1 (3 blocs) sont copiés dans la mémoire cache :

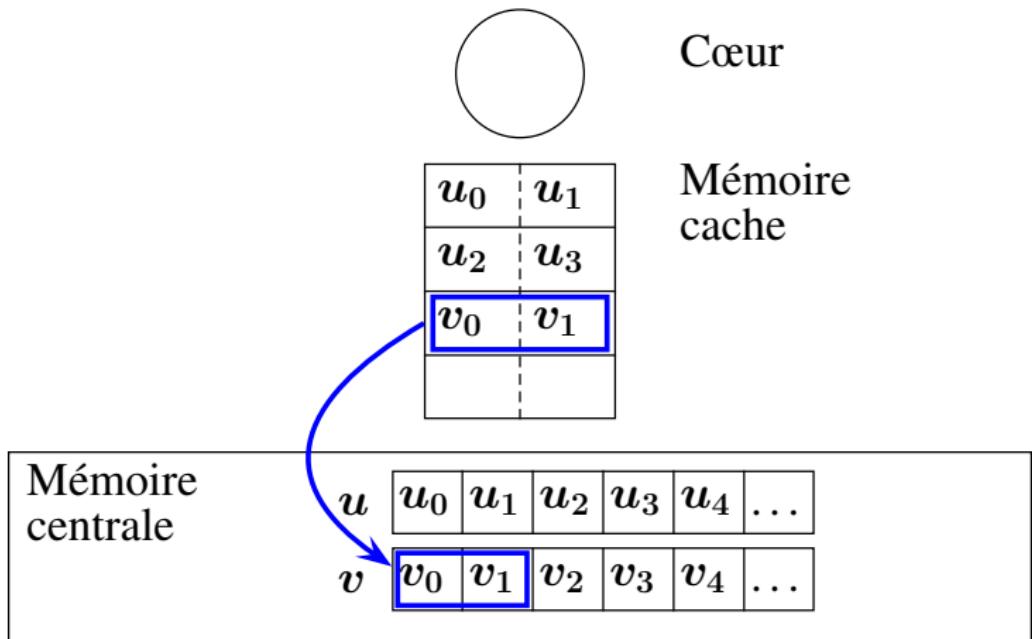


3. Le cœur utilise les copies de u_0 , u_1 , u_2 de la mémoire cache, calcule l'expression et place le résultat dans la mémoire cache :

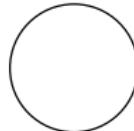


Mémoire centrale	u	u_0	u_1	u_2	u_3	u_4	\dots
	v	v_0	v_1	v_2	v_3	v_4	\dots

4. Le bloc contenant le résultat est recopié dans la mémoire centrale :



5. Le calcul de l'instruction suivante $v_2 = (u_1 + 2*u_2 + u_3)/4$ peut commencer:



Cœur

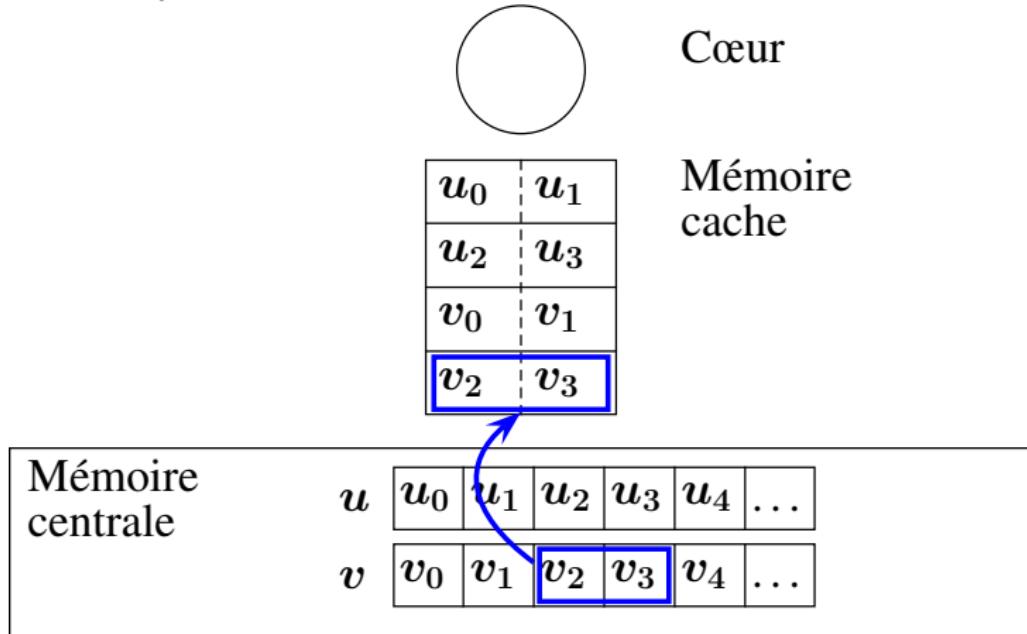
u_0	u_1
u_2	u_3
v_0	v_1

Mémoire cache

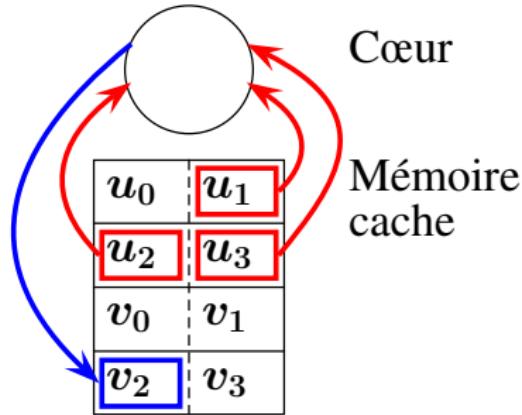
Mémoire centrale

u	u_0	u_1	u_2	u_3	u_4	\dots
v	v_0	v_1	v_2	v_3	v_4	\dots

6. Les composantes de u nécessaires sont déjà dans la mémoire cache, seul le bloc contenant la composante v_2 doit être copié dans la mémoire cache :

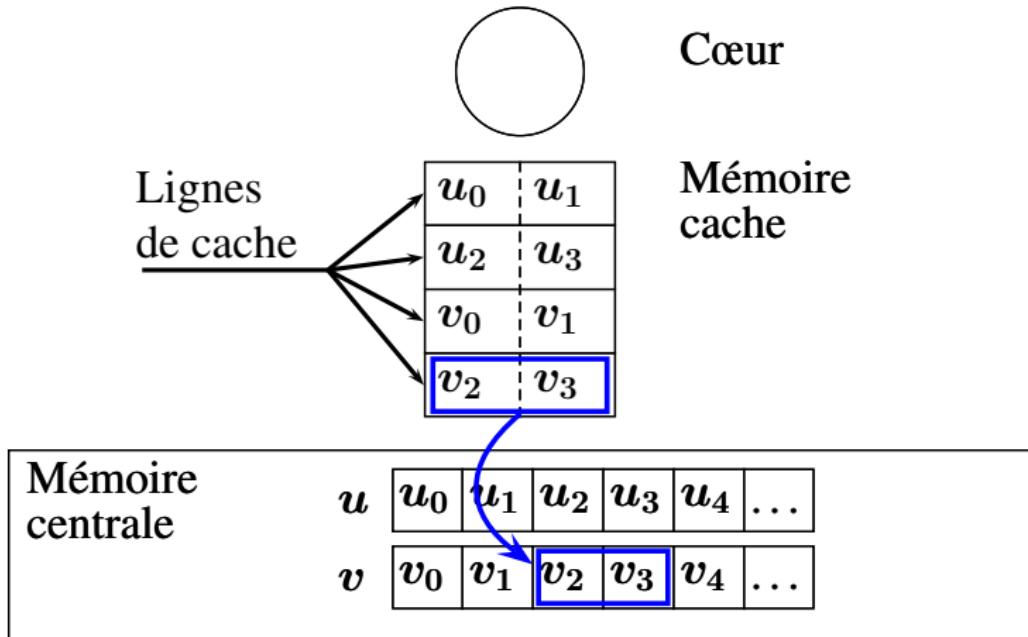


7. Le cœur utilise les copies de u_1 , u_2 , u_3 de la mémoire cache, calcule l'expression et place le résultat dans la mémoire cache :



Mémoire centrale	u	u_0	u_1	u_2	u_3	u_4	\dots
	v	v_0	v_1	v_2	v_3	v_4	\dots

8. Le bloc contenant le résultat est recopié dans la mémoire centrale :



En résumé :

- ▶ La première instruction $v_1 = (u_0 + 2 * u_1 + u_2) / 4$ utilise
 - ▶ 4 transferts (lents) mémoire centrale - mémoire cache
 - ▶ 4 transferts (rapides) mémoire cache - cœur
- ▶ La deuxième instruction $v_2 = (u_1 + 2 * u_2 + u_3) / 4$ utilise
 - ▶ 2 transferts (lents) mémoire centrale - mémoire cache
 - ▶ 4 transferts (rapides) mémoire cache - cœur

Attention : cet exemple est (très) simplifié : en général la mémoire cache est de taille plus grande que dans l'exemple et, de plus, il y a plusieurs types de mémoire cache dans un ordinateur

Dans les processeurs actuels, les lignes de cache ont une taille de quelques dizaines d'octets (entre 32 et 128 octets).

- Langage C avancé : Séance 6
 - Outils d'aide à la compilation
 - Types de données utilisateurs : vecteurs et matrices
-

Marc TAJCHMAN
@-mail : marc.tajchman@cea.fr

CEA - DES/ISAS/DM2S/STMF/LDEI

Outils d'aide à la compilation

Exemple de code source réparti dans plusieurs répertoires et/ou fichiers

Comme exemple, on utilisera le code source réparti dans les différents fichiers du répertoire **Exemple1_Make**:

main.c: contient le programme principal qui appelle les fonctions f et g et inclut les fichiers A.h et B.h

A.c: contient le code source de la fonction f et inclut le fichier A.h

B.c: contient le code source de la fonction g et inclut le fichier B.h

A.h: contient le prototype de la fonction f (sert à garantir que la fonction f est utilisée correctement)

B.h: contient le prototype de la fonction g (sert à garantir que la fonction g est utilisée correctement)

Exemple de code source réparti dans plusieurs répertoires et/ou fichiers

On compile les fichiers de code source, pour produire un exécutable exec1 (on a le choix du nom de l'exécutable)

- soit en une étape:

```
gcc main.c A.c B.c -o exec1
```

- soit en plusieurs étapes:

```
gcc -c main.c          # crée le fichier main.o
gcc -c A.c             # crée le fichier A.o
gcc -c B.c             # crée le fichier B.o
gcc main.o A.o B.o -o exec1 # crée l'exécutable exec1
```

Exemple de code source réparti dans plusieurs répertoires et/ou fichiers

Dès que le nombre de fichiers est important, on préfère procéder en plusieurs étapes pour éviter de recompiler le code source complet, si on ne modifie qu'une partie des fichiers.

Par exemple, si on modifie seulement l'intérieur de la fonction f (dans le fichier A.c):

```
gcc -c main.c          (inutile)
gcc -c A.c
gcc -c B.c          (inutile)
gcc main.o A.o B.o -o exec1
```

Dépendances entre les fichiers

Dans cet exemple, on a donc les règles suivantes:

- exec1 dépend de main.o, A.o et B.o
- main.o dépend directement de main.c et, indirectement, de A.h et B.h (inclus dans main.c)
- A.o dépend directement de A.c et, indirectement, de A.h (inclus dans A.c)
- B.o dépend directement de B.c et, indirectement, de B.h (inclus dans B.c)

Pour chaque règle, on devra effectuer une ou plusieurs commandes pour mettre à jour le résultat de la règle (souvent appelée « cible » ou target en anglais)

Dépendances entre les fichiers

Par exemple, si A.c a été modifié:

- Il faut recréer A.o, qui dépend de A.c, avec la commande
`gcc -c A.c`
- Il faut recréer exec1 qui dépend de A.o, qui vient de changer, avec la commande
 - `gcc main.o A.o B.o -o exec1`

Dépendances entre les fichiers

Par exemple, si B.h a été modifié:

- Il faut recréer B.o, qui dépend de B.c (et B.c inclut B.h), avec la commande

```
gcc -c B.c
```

- Il faut recréer main.o, qui dépend de main.c (et main.c inclut B.h), avec la commande

```
gcc -c main.c
```

- Il faut recréer exec1 qui dépend de main.o et B.o, qui viennent de changer, avec la commande

```
gcc main.o A.o B.o -o exec1
```

Util Make

Outil d'aide à la compilation : make

Très rapidement, pour recompiler « intelligemment » le code source en fonction des modifications, on a besoin d'utiliser un outil. Cet outil déclenche les compilations nécessaires (et elles seules) pour mettre à jour les binaires en fonction des modifications du code source.

L'outil utilisé ici est la commande `make` (il en existe d'autres: par exemple `nmake` sous windows)

On crée un fichier dont le nom est `Makefile` et où sont décrites les règles de construction, la commande `make` utilise ce fichier. Si on donne un autre nom au fichier de règles, il faut taper

```
make -f nom_du_fichier_de_règles
```

Syntaxe du fichier Makefile

Le fichier Makefile contient une liste de règles de la forme (chaque commande est précédée d'une tabulation):

Cible: Liste_de_dépendances

 Commande1

 Commande2

...

Make détermine qu'il faut exécuter commande1, commande2,... si au moins une dépendance est plus récente que la cible

Quand la cible d'une règle ne correspond à aucun fichier et que les commandes ne créent pas la cible, la règle est toujours exécutée

Syntaxe « make » (2)

Quand on tape « make », l'outil examine les dépendances de la première cible.

Si une dépendance n'est pas un fichier existant ou est plus récente que la cible, make cherche dans le fichier Makefile si une autre règle dont la cible est la dépendance.

Si aucune règle n'est disponible, make affiche un message d'erreur

Quand on tape « make nom_de_cible », l'outil examine les dépendances de la cible « nom_de_cible »

Syntaxe « make »

Dans le code source pris comme exemple, le fichier Makefile peut s'écrire :

```
exec1: main.o A.o B.o  
        gcc main.o A.o B.o -o exec1  
  
main.o: main.c A.h B.h  
        gcc -c main.c  
  
A.o: A.c A.h  
        gcc -c A.c  
  
B.o: B.c B.h  
        gcc -c B.c
```

Voir le répertoire
Exemple1_Make

Fonctionnalités supplémentaires

Plusieurs fonctionnalités permettent de simplifier ou d'écrire un makefile plus générique

- Variables utilisateur
- Commandes
- Variables spéciales
- Modèles de règle

Variables utilisateur

Si un mot est utilisé plusieurs fois dans plusieurs règles et/ou plusieurs commandes, on peut définir une variable (exemple : la variable Code contient le nom de l'exécutable construit par make) :

```
Code = exec1
```

```
 ${Code}: main.o A.o B.o
```

```
     gcc main.o A.o B.o -o ${Code}
```

```
clean:
```

```
     rm -f ${Code} *.o
```

Fonctions standards dans un makefile

Make propose des fonctions utilitaires, deux exemples

wildcard : crée une liste de tous les fichiers suivant un modèle

exemple : liste des fichiers C dans le répertoire du fichier Makefile

```
sources = $(wildcard *.c)
```

patsubst : à partir d'une liste de mots, crée une seconde liste où les mots de la première liste sont transformés avec une règle de substitution

exemple : on crée une liste de fichiers binaires (terminés par .o) à partir de la liste dans sources :

```
binaires = $(patsubst %.o, %.c, ${sources})
```

Fonctions standards dans un makefile (2)

On peut ensuite compiler un code qui contiendra tous les binaires correspondant aux fichiers sources C :

```
Exec1 : ${binaires}  
        gcc ${binaires} -o Exec1
```

Variables spéciales

Ces variables sont automatiquement définies pour chaque règle et doivent être utilisée seulement dans les commandes associées à cette règle, elles ne sont pas modifiables.

`$@` contient la cible d'une règle

`$<` contient l'ensemble des dépendances

$\* contient la première dépendance

Exemples

représentent

Exec1 : main.o A.o B.o

gcc \$< -o \$@

Exec1 : main.o A.o B.o

```
gcc main.o A.o B.o -o Exec1
```

A.o : A.c A.h

```
gcc -c $^ -o $@
```

A.o : A.c A.h

```
gcc -c A.c -o A.o
```

Modèles de règle

Plusieurs règles peuvent être remplacées par un modèle de règle si les cibles et dépendances ont une forme similaire

Exemples :

```
%.pdf: %.tex
```

```
pdflatex @<
```

La règle s'applique pour tous les noms de fichier se terminant en .tex et construisent le fichier dont le nom se termine en .pdf

```
%.o : %.c
```

```
gcc -c $^ -o $@
```

Dans le répertoire Exemple1_Make, il y a plusieurs versions de fichiers Makefile équivalents pour compiler un code.

Dans le répertoire Exemple2_Make, se trouve un Makefile qui compile un code dont les sources se trouvent dans plusieurs répertoires.

Pour plus de fonctionnalités de make, voir le manuel de référence (de la version GNU de make) :

<https://www.gnu.org/software/make/manual/make.pdf>

ou

https://www.gnu.org/software/make/manual/html_node/index.html

Outils de plus haut niveau

L’écriture des fichiers Makefile n’est pas simple dans le cas de codes source complexes.

D’autre part, dans les exemples précédents, on a mis « en dur » dans les Makefiles des informations système, par exemple le nom du compilateur.

Des outils supplémentaires ont été proposés pour utiliser le système de compilation sur plusieurs machines, systèmes et avec des compilateurs différents.

Si on se trouve dans un environnement machine / système / compilateur spécifique, ces outils génèrent des fichiers Makefile adaptés.

Outils autoools : automake, autoconf, ...

Outils autotools

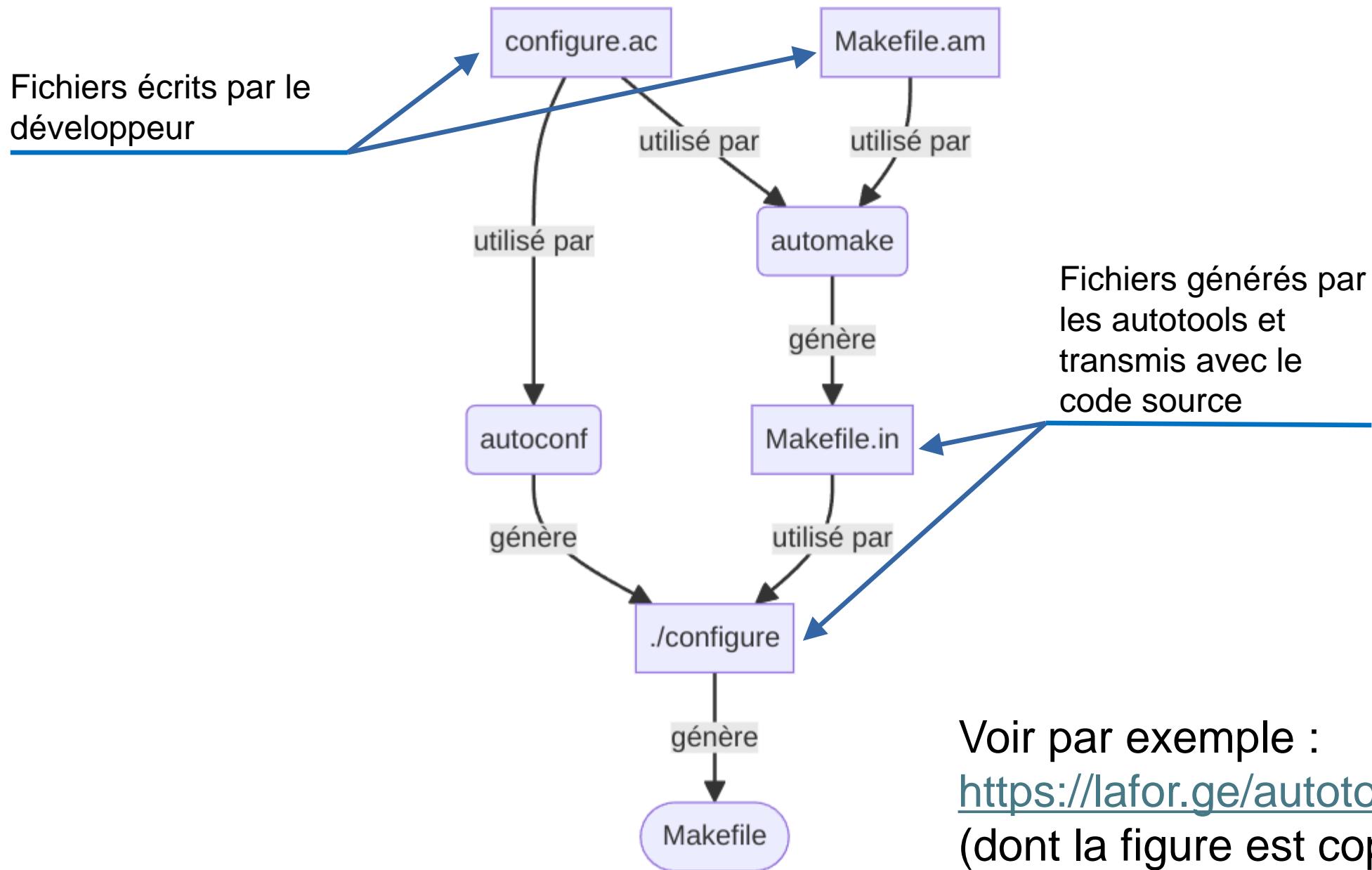
La famille d'outils m4, autoconf, automake, ... ont été conçus pour générer les Makefiles et des outils de configuration.

Le processus complet se fait en 2 parties:

- *une première partie à faire par ceux/celles qui écrivent le code source*
- *une seconde partie à faire par ceux/celles qui compilent le code source et installent le code compilé.*

Un fichier principal de configuration « configure » est créé par les autotools, il teste le compilateur et les autres outils pour savoir s'ils fonctionnent, quelles sont leur options, ...

Outils autotools



Exemple autotools

Dans le répertoire Exemple3_autotools se trouvent le code source et les fichiers Makefile.am et configure.ac (et des fichiers qui décrivent le code, les auteurs, etc.) :

```
aclocal  
automake --add-missing  
autoconf
```

Le code source + les fichiers/répertoires créés par les autotools sont transmis à ceux/celles qui installent utilisent le code et tapent les commandes :

```
./configure --prefix <chemin d'installation>  
make  
make install
```

Outils autotools

Les autotools sont utilisés par un grand nombre de codes C (et C++) existants. Il est donc utile de connaître ces outils.

Pour plus d'information, les manuels de ces outils sont disponibles :

<https://www.gnu.org/software/autoconf/manual/autoconf.pdf>

https://www.gnu.org/savannah-checkouts/gnu/autoconf/manual/autoconf-2.72/html_node/index.html

On conseille cependant d'utiliser l'outil présenté aux pages suivantes, plus récent et plus puissant.

Util CMake

CMake

cmake est un outil qui génère des procédures de configuration et de compilation.

Pour utiliser cmake, il faut écrire un fichier CMakeLists.txt dans tous les répertoires qui contiennent des fichiers sources.

cmake est un « vrai » langage de programmation avec :

- des boucles (foreach ... endforeach),
- des instructions conditionnelles (if ... else ... endif)
- des variables définies par cmake ou par l'utilisateur,
- des fonctions,
- des générateurs (pour créer des makefiles ou d'autres systèmes de compilation, par exemple des projets visual studio (sous Windows))

Cmake Exemple 1

Dans le répertoire Exemple4_Cmake, le premier exemple concerne un code compilé code1 à partir d'un seul fichier C : main.c dans le sous- répertoire sources.

Créer un fichier CMakeLists.txt dans le répertoire Exemple4_Cmake/sources (qui contient main.c) avec:

```
cmake_minimum_required(VERSION 3.10)
project(Exemple1 C)
add_executable(code1 main.c)
```

Dans le répertoire Exemple4_Cmake, taper les commandes :

```
cmake -S sources -B builds
cmake --builds builds
```

Cmake Exemple 1

Examiner le contenu des répertoires sources et builds après avoir tapé les commandes cmake.

- **Le répertoire sources n'est pas modifié** (aucun autre fichier que CmakeLists.txt et le(s) fichier(s) source).
- Le répertoire builds contient le fichier code1 (code compilé) et aussi tous les fichiers intermédiaires.

L'utilisateur du code a besoin du fichier code1 mais pas des fichiers intermédiaires.

On va légèrement modifier le fichier CmakeLists.txt et le commandes de compilation pour mettre le code compilé final à un autre endroit que les fichiers intermédiaires

L'utilisateur du code a besoin du fichier code1 mais pas des fichiers intermédiaires.

On va légèrement modifier le fichier CmakeLists.txt et le commandes de compilation pour mettre le code compilé final à un autre endroit que les fichiers intermédiaires

Cmake Exemple 1

Ajouter la ligne qui commence par install ci-dessous dans le fichier CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(Exemple1 C)
add_executable(code1 main.c)
install(TARGET code1 DESTINATION bin)
```

Supprimer le répertoire builds et taper les commandes :

```
cmake -S sources -B builds -DCMAKE_INSTALL_PREFIX=distrib
cmake --build builds
cmake --install builds
```

Cmake Exemple 1

Examiner le contenu des répertoires sources, builds et distrib après avoir tapé les commandes cmake.

On a une séparation claire des fichiers:

- le répertoire sources contient les fichiers source non modifiés + les fichiers CMakeLists.txt
 - on peut le supprimer sans risquer de perdre des fichiers source ou le code final compilé et on peut le régénérer facilement*
- le répertoire builds contient les fichiers intermédiaires produits par la compilation
- le répertoire distrib contient le code final compilé
 - on peut transmettre ce répertoire aux utilisateurs du code, il contient tout ce qui est nécessaire pour utiliser le code*

Cmake Exemple 2

Dans le répertoire `Exemple5_Cmake`, le second exemple concerne un code compilé `code2` (résolution de système linéaire en utilisant des structures vecteurs et matrices) à partir de plusieurs fichiers dans des répertoires différents.

De plus, il y a des fichiers qui ne contiennent pas de code source C qu'il faut installer en même temps que le code (fichiers d'exemples)

Cmake Exemple 2

Les différents répertoires contiennent chacuns un fichier CMakeLists.txt, dont un est le fichier cmake principal:

sources:

 CMakeLists.txt (fichier principal)

types:

 CMakeLists.txt (fichier secondaire)

data:

 CMakeLists.txt (fichier secondaire)

Le fichier cmake principal appelle les fichiers secondaires qui définissent des cibles. Ces cibles sont utilisées comme dépendances dans le fichier principal.

[Examiner les différents fichiers CMakeLists.txt](#)

Structures vecteur et matrice

Rangement des coefficients de vecteurs et matrices

La mémoire des ordinateurs est organisée comme une structure à une dimension d'octets.

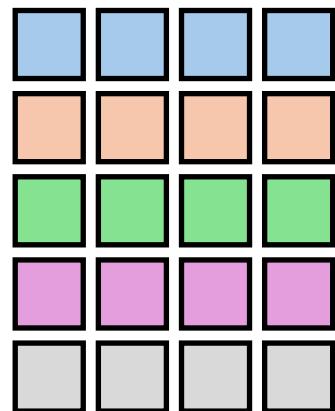
Il y a donc une façon « naturelle » de ranger les coefficients des vecteurs en mémoire : dans l'ordre des indices croissants.

Pour les matrices, par contre, plusieurs choix sont possibles.

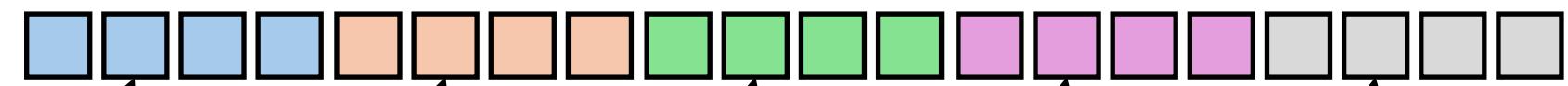
Rangement des coefficients de matrices à n lignes et m colonnes

Première façon, lignes par lignes

Structure
mathématique A



Structure informatique A

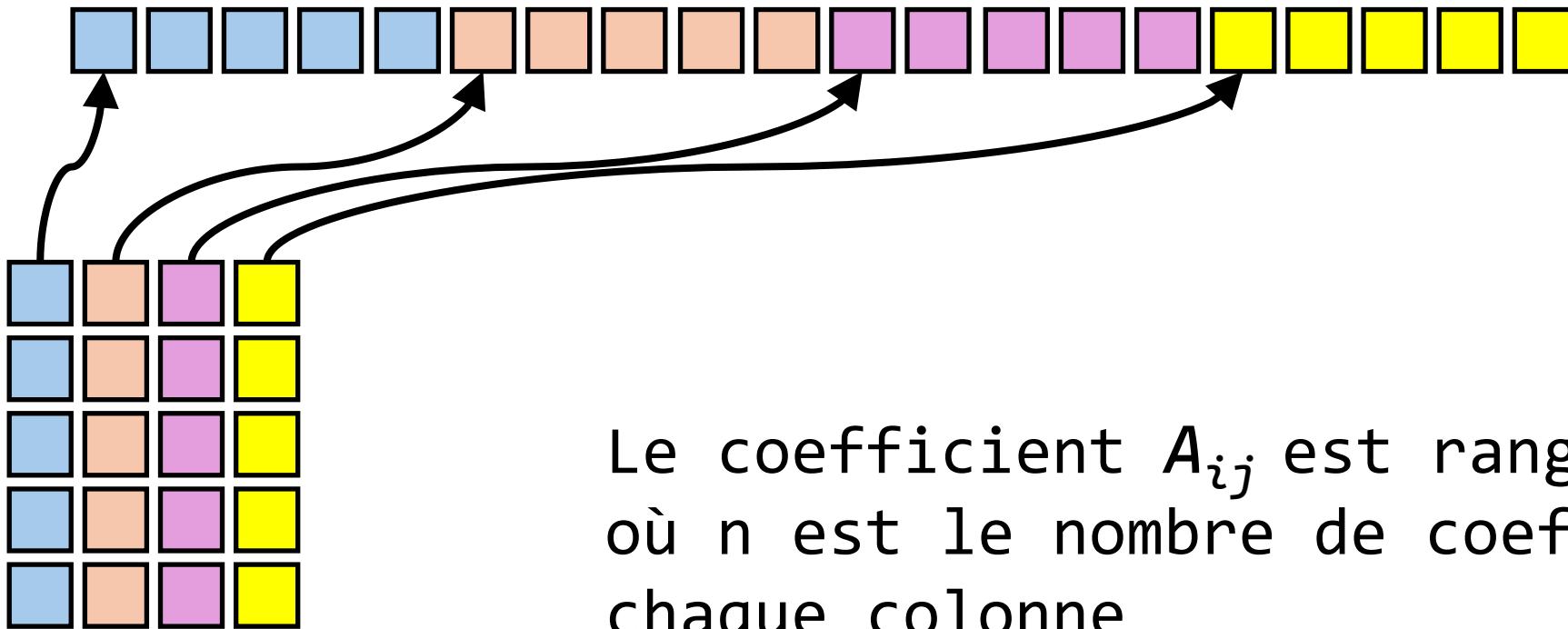


Le coefficient A_{ij} est rangé en $A[i*m+j]$
où m est le nombre de coefficients dans
chaque ligne

Rangement des coefficients de matrices à n lignes et m colonnes (2)

Deuxième façon, colonnes par colonnes

Structure informatique A



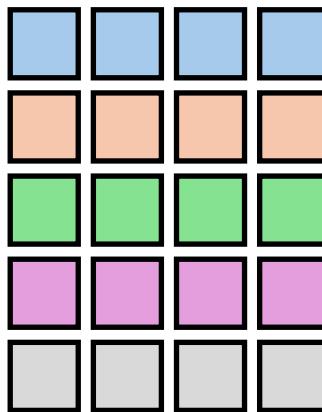
Le coefficient A_{ij} est rangé en $A[i+j*n]$
où n est le nombre de coefficients dans
chaque colonne

Structure
mathématique A

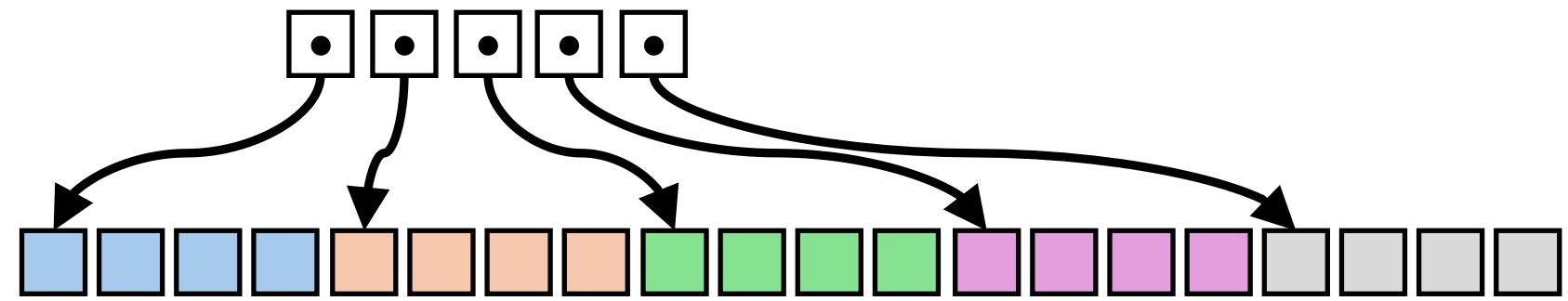
Rangement des coefficients de matrices à n lignes et m colonnes (3)

Troisième façon, ligne par ligne en passant par un vecteur de pointeurs

Structure informatique A



Structure
mathématique A



Le coefficient A_{ij} est rangé en $A[i][j]$
(remarquer que la taille de A n'intervient pas dans la formule)

Rangement des coefficients de matrices à n lignes et m colonnes (4)

Le choix du rangement des coefficients en mémoire dépend de plusieurs facteurs, entre autres:

- le langage de programmation
- le sens de parcours des coefficients par l'algorithme choisi par l'utilisateur

Par exemple, si A est une matrice, u et v deux vecteurs,
 ${}^T u A v$ peut être calculé de deux façons:

- $({}^T u A) v$ se fait plus rapidement si A est rangée colonne par colonne
- ${}^T u (A v)$ se fait plus rapidement si A est rangée ligne par ligne

Vecteurs, matrices

On a vu qu'un vecteur et une matrice sont caractérisées par le type et le nombre de leurs composantes. En C, il n'y pas de type vecteur et matrice en tant que tels (on doit utiliser des pointeurs et mettre les informations de taille à part dans des variables entières)

Par exemple, pour écrire une fonction qui calcule le produit matrice vecteur, on doit fournir :

- Une matrice de taille $n \times m$ (matrice de départ)
- Un vecteur de taille p (vecteur de départ)
- Un autre vecteur de taille q (qui va recevoir le résultat du calcul)

Avec les contraintes que $p = m$ et $q = n$ (sinon le calcul du produit n'a pas de sens)

Vecteurs, matrices

Il est possible de définir la fonction produit matrice vecteur de la façon ci-dessus:

```
void ProduitMatriceVecteur(double * M, int nM, int mM,  
                           double * V, int nV,  
                           double * W, int nW)  
{ ... }
```

Et appeler la fonction comme ceci pour calculer $y = Ax$:

```
ProduitMatriceVecteur(A, n, m, x, p, y, q);
```

L'utilisateur doit faire très attention de transmettre les bonnes valeurs en arguments, en particulier les tailles.

Vecteurs, matrices

Une solution est de définir des types vecteur et matrice (c'est le choix de la plupart des langages modernes)
En C, on peut utiliser des structures pour cela.

Exercice:

Dans le répertoire `SystemeLineaire/types` sont définies des structures vides `Vecteur` et `Matrice` dans les fichiers `matrice.h` et `vecteur.h`, les compléter pour y ranger leurs informations de taille et leurs composantes.

Exercice:

Dans les fichiers vecteur.c et matrice.c, les fonctions alloueMatrice et alloueVecteur doivent réserver de la mémoire pour des matrices et vecteurs connaissant leurs tailles. Elles sont actuellement vides, les compléter.

De même pour les fonctions libereVecteur et libereMatrice qui rendent au système, la mémoire utilisée par un vecteur ou une matrice.

Exercice:

Dans le répertoire `SystemeLineaire/data` se trouve deux fichiers qui contiennent chacun une matrice, après une première ligne qui donne la taille de la matrice et le nombre de coefficients non nuls, sous la forme d'une liste de triples:

 numéro de ligne,

 numéro de colonne,

 valeur du coefficient

Le fichier `matrice.c` définit la fonction `lectureMatrice` qui lit une matrice depuis un fichier. Cette fonction est incomplète, la compléter

Fonctions utilitaires

Pour rendre la lecture du code plus claire, on définit souvent des fonctions qui accèdent au coefficient A_{ij} . Par exemple pour le stockage ligne par ligne :

Fonction qui récupère une valeur depuis un coefficient

```
inline double get(Matrice M, int i, int j)
{ return M.coef[i*M.m + j]; }
```

Fonction qui range une valeur dans un coefficient

```
inline set(Matrice M, int i, int j, double v)
{ M.coef[i*M.m + j] = v; }
```

Macros

On peut utiliser des macros du préprocesseur, toujours dans le cas du stockage ligne par ligne :

```
#define Coef(M,i,j) M.coef[(i)*M.m + j];
```

(ne pas oublier de mettre des parenthèses autour de i, sinon Coef(A,k+1,l) sera interprété comme A.coef[k + 1*A.m + l] qui est différent de A.coef[(k + 1)*A.m + l])

Langage C avancé : Séance 7

Types de données utilisateurs : vecteurs (suite), listes

Complexité

Marc TAJCHMAN

@-mail : marc.tajchman@cea.fr

CEA - DES/ISAS/DM2S/STMF/LDEI

Types Union

Union

Le but est d'utiliser des variables qui peuvent contenir des valeurs de type différents.

On ne peut pas définir un « type universel ».

Par contre, on peut définir un **pointeur universel** : `void *`

C propose de définir un type qui peut représenter un ensemble fini de types existants, grâce au mot-clef `union`:

```
union _nombre
{
    int n;
    float f;
    double d;
};
typedef union _nombre nombre;
```

Les variables de type `nombre` peuvent contenir une valeur entière, réelle simple précision ou réelle double précision.

Mais **un seul** de ces 3 types (et pas de chaîne de caractère, pas de pointeur, ...)

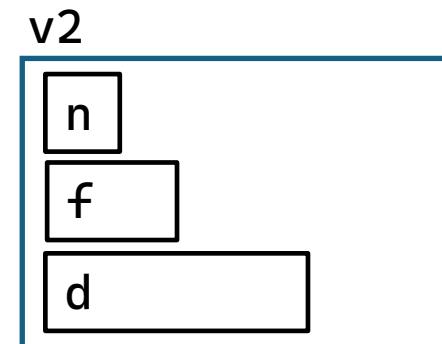
Différence entre union et struct

Les 2 types utilisateurs ont une déclaration et une syntaxe très proches :

```
union _nbre1 {  
    int n;  
    float f;  
    double d;  
};  
typedef union _nbre1 Nombre1;  
Nombre1 v1;
```

```
struct _nbre2 {  
    int n;  
    float f;  
    double d;  
};  
typedef struct _nbre2 Nombre2;  
Nombre2 v2;
```

Mais les variables de type struct et union sont organisées différemment en mémoire:



Différence entre union et struct (2)

Une variable de type Nombre1, défini en utilisant **union**:

- Occupe une zone mémoire dont la taille est celle du plus grande sous-type (ici `sizeof(double)`);
- Contient un `int` ou un `float` ou un `double`.

Une variable de type Nombre2, défini en utilisant **struct**:

- Occupe une zone mémoire dont la taille est la somme des tailles de tous les sous-type (ici `sizeof(double) + sizeof(float) + sizeof(int)`);
- Contient un `int` et un `float` et un `double`

Exemple d'utilisation

```
union _nbre {  
    int n;  
    float f;  
    double d;  
    const char *s;  
};  
typedef union _nbre Nombre;  
Nombre v;  
  
v.n = 1;  
v.f = 1.4; /* ecrase v.n */  
  
printf("v.f = %g\n", v.f);  
printf("v.n = %d\n", v.n);
```

```
void affiche(Nombre x)  
{  
    printf("%d", x.n); // ???  
    printf("%f", x.f); // ???  
    printf("%g", x.d); // ???  
}
```

Dans la fonction `affiche`, on ne sait pas ce qui est contenu dans le paramètre `x`.

Donc il faut passer une autre information en paramètre : le type exact contenu dans `x`

En pratique

En pratique, on met l'union dans un type struct avec un indicateur de type:

```
#define Entier 1
#define Float 2
#define Double 3
#define Chaine 4
struct _nombre {
    int type;
    union _valeur {
        int n;
        float f;
        double d;
        const char *s;
    } v;
};
typedef struct _nombre nombre;
```

Exemple d'une fonction qui affiche une variable de ce type (tester en compilant union2.c) :

```
void affiche(nombre x) {
    if (x.type == Entier)
        printf("%d", x.v.n);
    else if (x.type == Float)
        printf("%f", x.v.f);
    else if (x.type == Double)
        printf("%lf", x.v.d);
    else if (x.type == Chaine)
        printf("%s", x.v.s);
    printf("\n");
}
```

En pratique (2)

On peut (un peu) simplifier en utilisant une union anonyme dans struct :

```
#define Entier 1
#define Float 2
#define Double 3
#define Chaine 4
struct _nombre {
    int type;
    union {
        int n;
        float f;
        double d;
        const char *s;
    };
};

typedef struct _nombre nombre;
```

Exemple d'une fonction qui affiche une variable de ce type (tester en compilant union2.c) :

```
void affiche(nombre x) {
    if (x.type == Entier)
        printf("%d", x.n);
    else if (x.type == Float)
        printf("%f", x.f);
    else if (x.type == Double)
        printf("%lf", x.d);
    else if (x.type == Chaine)
        printf("%s", x.s);
    printf("\n");
}
```

Type vecteur (suite et fin)

Types de données définis par l'utilisateur

Les types C standards ne sont pas toujours suffisants (on a vu l'exemple des vecteurs « pointeur C » où la taille du vecteur n'accompagne pas les composantes).

On a donc défini un type utilisateur « vecteur » avec **struct** (qui définit l'organisation des données dans le type utilisateur) et le mot-clef **typedef** (qui facilite l'écriture du type utilisateur) :

```
struct _vecteur {  
    int n;  
    double * c;  
    const char *nom;  
};  
typedef struct _vecteur vecteur;
```

Ici les composantes sont de type réel (double) mais peuvent être de n'importe quel type standard ou utilisateur
On a ajouté un pointeur sur le nom du vecteur dans la struct (peut être utile)

Utilisation de variables de type utilisateur

```
Vecteur V;  
V.Nom = "V";  
V.n = 10;  
V.c = (double *) malloc(V.n*sizeof(double));  
  
for (i=0; i<V.n; i++)  
    V.c[i] = 2.5*i;  
  
free(V.c);
```

Définition d'une variable de type vecteur et réservation de la mémoire interne dynamique éventuelle

Utilisation de la variable V

Libération de la mémoire réservée à la définition de la variable V

Examiner et compiler le code dans le répertoire vecteurs/v1

La syntaxe des types utilisateurs en C n'est pas très élégante, d'autres langages de programmation (C++, python, ...) permettent de définir des types utilisateurs avec une syntaxe plus « naturelle »

Fonctions utilitaires pour un type utilisateur

On définit souvent des fonctions qui:

- préparent une variable de type utilisateur (constructeur)
- nettoient la mémoire utilisée par une variable de type utilisateur (destructeur)
- font des opérations « standards » avec des variables de type utilisateur

Ces fonctions sont intéressantes parce que le code est plus facile à lire, mais aussi parce que cela diminue la possibilité d'erreurs.

Par contre, il faut les écrire soigneusement.

Exemple de fonctions utilitaires pour le type vecteur (version 1)

```
Vecteur construitVecteur(int n, const char *s)
{
    Vecteur v;
    v.nom = s;
    v.n = n;
    v.c = (double *) malloc(sizeof(double) *n);
    return v;
}
```

Fonction
« constructeur »

```
void detruitVecteur(Vecteur v)
{
    v.n = 0; v.nom = "";
    free(v.c);
    v.c = NULL;
}
```

Fonction
« destructeur »

Examiner les fichiers dans le répertoire vecteurs/v2. Le type vecteur est défini dans le fichier vecteur.h et les fonctions utilitaires dans le fichier vecteur.c

Exemple de fonctions utilitaires pour le type vecteur (version 1)

La compilation dans le répertoire vecteurs/v2 produit 2 fichiers exécutables: install/exec2a et install/exec2b

Exécuter install/exec2a et install/exec2b.

L'exécution de install/exec2b produit des erreurs.
Essayer de les expliquer.

De plus, cette version des constructeur/destructeur a aussi des désavantages.

Quels sont-ils ?

Dans le fichier main2b.c

```
Vecteur V = construitVecteur(10);
Vecteur W = construitVecteur(10);

...
W = V;
W.c[5] = -56.0;

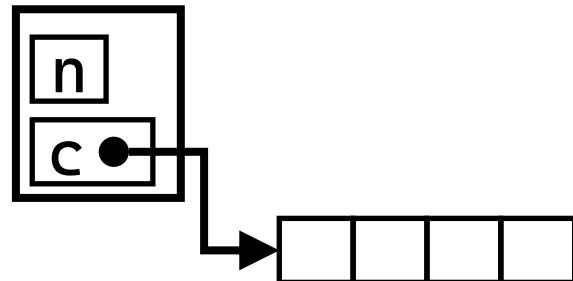
...
detruitVecteur(V);
detruitVecteur(W);
```

Erreurs dans le fichier main2b.c

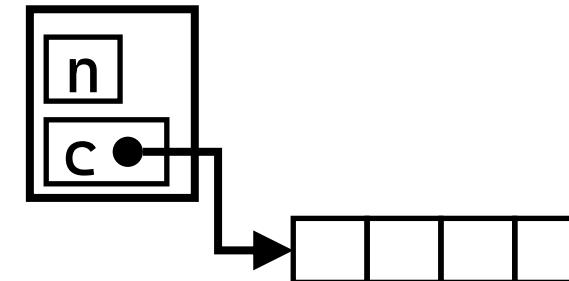
L'instruction $W = V$, où V et W sont des structures, copie les composantes de la structure V dans W .

Mais cette copie est une copie de surface (shallow copy).

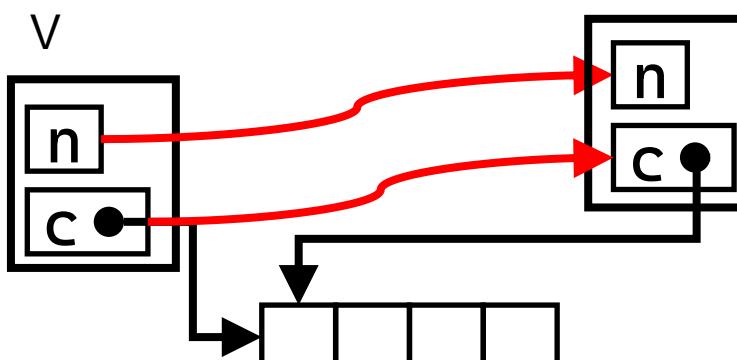
Avant $W = V$: V



W



$W = V$ (copie de V dans W)



Zone mémoire réservée pour V par le code, avec 2 pointeurs qui contiennent son adresse ($V.c$ et $W.c$)



Zone mémoire réservée pour W par le code mais inutilisable (« décrochée » de W , plus aucun pointeur ne contient son adresse)

Erreurs dans le fichier main2b.c

```
W.c[2] = -56.0;
```

Modifie à la fois W.c[2] et V.c[2] puisque W.c et V.c contiennent la même adresse. D'où l'erreur dans les valeurs affichées.

```
detruitVecteur(V);
```

Rend au système la mémoire réservée pour V

```
detruitVecteur(W);
```

W contient un pointeur vers la mémoire réservée par V (à cause de W = V). Donc la mémoire réservée par V est rendue 2 x au système.

Ce qui provoque une erreur d'exécution et l'exécution du code s'arrête brutalement (dans les meilleurs des cas).

Désavantages dans l'écriture des fonctions utilitaires (1)

```
Vecteur construitVecteur(int n, const char *s)
{
    Vecteur v;
    v.n = n; v.nom = s;
    v.c = (double *) malloc(sizeof(double) *n);
    return v;
}
```

Dans cette fonction, on crée 2 variables temporaires de type Vecteur:

- la variable locale v dans la fonction construitVecteur et
- la valeur de retour de cette fonction qui est une copie de v
- ensuite cette valeur de retour est copiée dans la variable vecteur du programme principal

Cela ne pose pas de problème parce qu'on alloue qu'une seule zone mémoire qui sera libérée une seule fois (dans detruitVecteur).

Mais on crée des variables temporaires et on fait des copies inutiles

Désavantages dans l'écriture des fonctions utilitaires (2)

```
void detruitVecteur(Vecteur vl)
{
    vl.n = 0; v.nom = "";
    free(vl.c);
    vl.c = NULL;
}
```

Dans cette fonction, on crée une copie locale `vl` de type `Vecteur`:

- le paramètre local `vl` dans la fonction `detruitVecteur`
- cette variable locale rend la zone mémoire qui contient ses coefficients au système (`free`)
- le contenu de la variable locale `vl` est mis à zéro

Cela ne pose pas de problème parce que la mémoire réservée par le vecteur du programme principal, est rendue au système par la variable locale.

Mais quand on sort de la fonction, le vecteur `v` a un pointeur qui n'a pas changé et désigne une zone mémoire qui ne lui est pas affectée. Potentiellement, cela peut provoquer des erreurs plus tard.

Exemple de fonctions utilitaires pour le type vecteur (version 2)

```
void construitVecteur(Vecteur * v, int n)
{
    v->n = n;
    v->c = (double *) malloc(sizeof(double) *n);
}
```

Fonction
« constructeur »

```
void detruitVecteur(Vecteur *v)
{
    v->n = 0;
    free(v->c);
    v->c = NULL;
}
```

Fonction
« destructeur »

Dans cette version, c'est un pointeur sur vecteur qui est passé en paramètre. On a des copies de pointeurs, ce qui coute beaucoup moins de temps que des copies de vecteurs. De plus on met bien à zéro le vecteur passé en paramètre à `detruitVecteur`.

Exemple de fonctions utilitaires pour le type vecteur (version 2)

A la place de $W = V$, il faut utiliser une fonction utilitaire de copie dont le code est écrit ci-dessous

```
void copieVecteurs(Vecteur * w, const Vecteur *v)
{
    if (w->n != v->n) {
        detruitVecteur(w);
        construitVecteur(w, v->n);
    }
    memcpy(w->c, v->c, v->n * sizeof(double));
}
```

Fonction
« copie »

Ce type de fonction de copie effectue une copie complète (y compris de la mémoire dynamique).

On parle aussi de « **deep copy** ».

Dans le fichier main2c.c

```
Vecteur V, W;  
construitVecteur(&V, 10);  
construitVecteur(&W, 10);  
...  
copieVecteur(&W, &V);  
W->c[5] = -56.0;  
...  
detruitVecteur(&V);  
detruitVecteur(&W);
```

Cette version ne contient pas les erreurs de la version précédente mais utilise beaucoup de pointeurs.

Evolution du temps de calcul quand la taille des structures ou du problème varie.

On parle parfois de l'évaluation de la complexité du code : notation $O(\dots)$.

Notation O(1)

C'est une indication de l'ordre de grandeur du temps mis pour réaliser une opération ou plusieurs.

Exemple 1: `v.c[k] = 1.2` où v est une variable de type vecteur, k est un entier entre 0 et n (n est la taille du vecteur).

On peut décomposer cette instruction en plusieurs parties:

```
double *p = v.c;      // pointeur sur la première composante
p = p + k;            // décalage de p de k * sizeof(double) octets
*p = 1.2;             // on rentre la valeur dans la
                      // composante pointée par p
```

Le temps d'exécution de l'instruction est la somme des temps de chacune des parties (création/initialisation/destruction d'un pointeur, décalage du pointeur : 1 addition et 1 multiplication d'entier, copie d'une valeur double).

Il ne dépend pas de n, la taille du vecteur.
On dit que l'instruction est en **O(1)**.

Notation O (2)

2^{ème} exemple: recherche d'un minimum dans un vecteur

Deux algorithmes, parmi d'autres :

```
if (v.n > 0) {  
    vmin = v.c[0];  
    for (i=1; i<v.n; i++)  
        if (v.c[i] < vmin) vmin = v.c[i];  
}
```

```
#include <float.h>  
  
vmin = DBL_MAX;  
for (i=0; i<v.n; i++)  
    if (v.c[i] < vmin) vmin = v.c[i];
```

Chaque instruction, prise séparément, est en $O(1)$, mais l'ensemble de chaque algorithme contient $C_1 n + C_0$ instructions simples (C_1 et C_0 sont des constantes) où n est la taille du vecteur.

Pour n grand, le nombre d'instructions simples (et donc le temps de calcul) est (à peu près) proportionnel à n .
On dit que l'algorithme est en $O(n)$.

Notation O (3)

3^{ème} exemple: multiplication de matrices $w = u * v$, où u , v , w sont des matrices, u de taille $(N \times M)$, v de taille $(M \times P)$ et w de taille $(N \times P)$ (pour que la multiplication ait un sens).

Pour chaque coefficient de w : $w_{ij} = \sum_{k=0}^{k=M-1} u_{ik}v_{kj}$ pour $0 \leq i < N$, $0 \leq j < P$

Un algorithme, parmi d'autres, est:

```
for (i=0; i < N; i++)
    for (j=0; j < P; j++) {
        double s = 0.0;
        for (k=0; k < M; k++)
            s += u.c[i*M + k] * v.c[k*P + j];
        w.c[i*P + j] = s;
    }
```

Le nombre d'instructions simples est $C_3n^3 + C_2n^2 + C_1n + C_0$. Quand n est grand, le terme le plus important est proportionnel à n^3 .

On dit que l'algorithme est en $O(N \times M \times P)$ si les matrices sont rectangulaires ou $O(n^3)$ si les matrices sont carrées de taille $n \times n$.

Notation O (4)

En fonction de l'algorithme considéré, l'exposant k dans $O(n^k)$ peut être évalué en comptant :

- Toutes les instructions
- Les opérations arithmétiques sur des entiers et des réels
- Les opérations seulement entre des entiers
- Les opérations seulement entre des réels
- Les lectures et/ou les écritures en mémoire de travail
- Les lectures et/ou les écritures dans des fichiers
- La taille de la zone mémoire nécessaire

Chaque fois que la complexité d'un code est fournie, il faut aussi préciser la méthode de calcul.

Opérations bien adaptées et moins bien adaptées sur des vecteurs

Accès aux composantes d'un vecteur dans un ordre quelconque

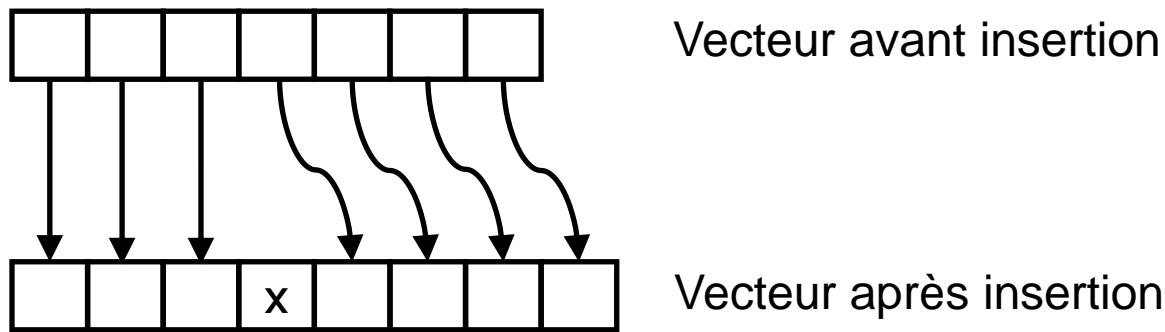
L'utilisation ou la modification d'une composante d'indice donné d'un vecteur est efficace (c'est une opération en $O(1)$)

Faire cette opération k fois dans un ordre quelconque, a une complexité d'ordre $O(k)$. L'opération ne demande pas de zone mémoire supplémentaire.

Un vecteur est bien adapté à ce type d'algorithme.

Insertion d'une nouvelle composante dans un vecteur

Par contre l'insertion d'une nouvelle composante est moins adaptée.
Principe (insertion de la valeur x à la 4^{ème} position) :



Augmenter la taille d'un vecteur n'est pas immédiat parce que le vecteur peut être coincé entre des zones mémoire déjà utilisées par d'autres variables.

Donc il faut:

- Demander au système une nouvelle zone mémoire de taille $n+1$
- Recopier les composantes existantes à la bonne place dans la nouvelle zone
- Mettre la valeur de la nouvelle composante dans la nouvelle zone
- Rendre au système l'ancienne zone mémoire des composantes

Insertion d'une nouvelle composante dans un vecteur

Un exemple d'algorithme pour insérer une valeur à la $k^{\text{ème}}$ position:

```
void insere(Vecteur *v, int k, double x) {
    /* v : vecteur de taille n */
    int i, n_old = v->n, n_new = n_old+1;
    double *c_old = v->c;
    v->c = NULL;
    construitVecteur(v, n_new, v->name);

    for (i=0; i<k; i++)
        v->c[i] = c_old[i];
    for(i=k; i<n_old; i++)
        v->c[i+1] = c_old[i];
    v->c[k] = x;
}
```

Construit un vecteur avec $n+1$ composantes mais garde une copie des composantes existantes (complexité $O(1)$, mais $O(n)$ en mémoire supplémentaire)

Recopie les anciennes composantes en gardant une place libre pour la valeur à insérer (complexité $O(n)$)

Remarque: on peut remplacer les boucles par deux appels à `memcpy` (qui prennent moins de temps mais qui sont aussi en $O(n)$)

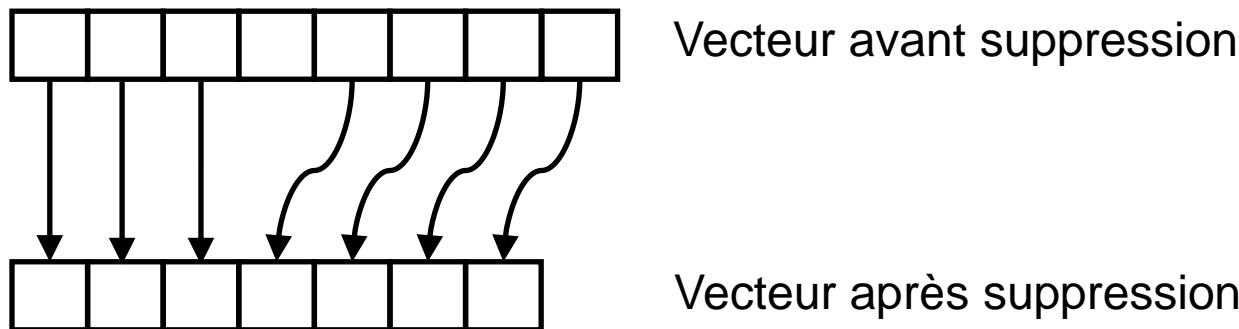
Faire k insertions demande $O(n+k)$ mémoires supplémentaires et a une complexité en $O(nk)$.

Il manque une instruction pour que l'algorithme soit « propre », laquelle ?

Suppression d'une composante dans un vecteur

La situation plus simple pour les suppressions.

Principe (suppression de la 4^{ème} composante) :



On n'a pas la nécessité de changer la taille du vecteur, puisque cette taille diminue (et donc pas besoin de mémoire supplémentaire). On aura des (petites) zones mémoires inutilisées (c'est un inconvénient mineur).

La complexité en nombre d'opérations est en $O(n)$ pour déplacer les composantes après la composante supprimée.

Suppression d'une composante dans un vecteur

Un exemple d'algorithme pour supprimer la k^{ème} composante:

```
void supprime(Vecteur *v, int k)
{
    int i;
    for(i=k; i<v->n-1; i++)
        v->c[i] = v->c[i+1];
    v->n -= 1;
}
```

Déplace les composantes, après la composante à supprimer, d'une position vers la gauche (complexité O(n))

Capacité des vecteurs

Pour pouvoir faire plus simplement à la fois des insertions et suppressions dans un vecteur, on distingue les notions de taille (taille courante) et de capacité (taille maximale) dans la structure vecteur:

```
struct _vecteur
{
    const char *name;
    int n;
    int n_max;
    double *c;
};
typedef struct _vecteur Vecteur;
```

Capacité des vecteurs (2)

Avec des règles (exemple) :

- Si le vecteur est presque plein ($n > 0.75 * n_{\max}$), on augmente la capacité (par ex. $n_{\max} \rightarrow n_{\max} * 2$)
- Si le vecteur utilise peu de composantes par rapport à la capacité (par ex., $n < n_{\max}/2$) on diminue la capacité (par ex. $n_{\max} \rightarrow n_{\max}/2$).

Ces règles sont un exemple, ceux qui développent les structures vecteur font en général beaucoup de tests pour trouver les règles les plus optimales pour les algorithmes utilisés (en temps calcul et en occupation mémoire).

Ne pas oublier de réallouer les composantes quand on change la capacité !!

Capacité des vecteurs (3)

Le répertoire vecteurs/capacite contient une définition de vecteur avec les indicateur de taille et de capacité.

Examiner les fichiers sources, les compiler et les exécuter.

Types liste

Définitions

Une liste permet de représenter un ensemble d'éléments, avec un ordre donné entre les éléments et un (ou des) sens de parcours entre les éléments bien défini. On appelle parfois les éléments des « nœuds ».

Pour utiliser une liste, on doit pouvoir au moins :

- Créer une liste vide
- Insérer un nouvel élément dans une liste
- Supprimer un élément existant dans une liste
- Accéder au début de la liste (si elle est non vide)
- Passer d'un élément au suivant dans la liste
- Tester si on se trouve à la fin de la liste

En fonction des besoins, il faudra peut-être pouvoir :

- Accéder au dernier élément de la liste
- Passer d'un élément au précédent dans la liste
- Tester si on se trouve en début de liste

Comparaison des listes et vecteurs

- Un vecteur a pour principale fonctionnalité l'accès facile et rapide à chacune de ses composantes, dans un ordre quelconque (avec un indice entier).
Par contre, les opérations d'insertion, suppression, déplacement de composantes sont moins « naturelles » et peuvent avoir une complexité non négligeable.
- Une liste d'éléments est conçue pour rendre rapide les opérations d'insertion, suppression, déplacement dans la liste.
Par contre, il y a peu de façons naturelle de se déplacer dans la liste (une ou deux en général)

Types liste chainée

Type nœud

La première étape est de définir un type pour représenter un nœud de la liste :

- il contiendra l'information utile (la valeur entière),
- il permettra d'accéder au nœud suivant pendant un parcours de la liste

```
struct _node
{
    int value;
    struct _node *next;
};
typedef struct _node Node;
```

Remarquer qu'on utilise la structure `_node` à un endroit où cette structure n'est pas complètement définie, cela ne pose pas de problème parce qu'on utilise seulement un pointeur sur `_node` (voir l'exemple `lists/list1/main.c`). C'est un exemple de définition récursive : une structure contient un pointeur sur une structure de même type.

Dans la structure `_node`, l'information utile est dans « `value` », « `next` » est un pointeur qui permet de passer au nœud suivant dans la liste.

Création/destruction de nœud

Il sera peut-être utile de définir des fonctions pour créer/détruire des nœuds :

```
Node* creeNode(int v) {
    Node* newNode = (Node*) malloc(sizeof(Node));
    newNode->value = data;
    newNode->next = NULL;
    return newNode;
}
```

```
void detruitNode(Node ** n) {
    if (*n) {
        free(*n);
        *n = NULL;
    }
}
```

Type liste en C

On veut un type liste qui puisse représenter une collection de plusieurs nœuds (y compris 0 nœuds).

Le plus simple est de définir une liste comme un pointeur sur un nœud:

```
typedef Node * List;
```

```
List L = NULL;  
  
Node * n1 = (Node *) malloc(sizeof(Node));  
n1->value = 1; n1->next = NULL;  
  
L = n1;  
  
Node * n2 = (Node *) malloc(sizeof(Node));  
n2->value = 2; n2->next = NULL;  
  
Node * start = L;  
start->next = n2;
```

Liste vide

Nouveau nœud

La liste contient un nœud

2^{ème} nouveau nœud

Ajout du 2^{ème} nœud à la suite du 1^{er} dans la liste

Insertion d'un nœud dans une liste

L'insertion consiste en une manipulation de pointeurs.

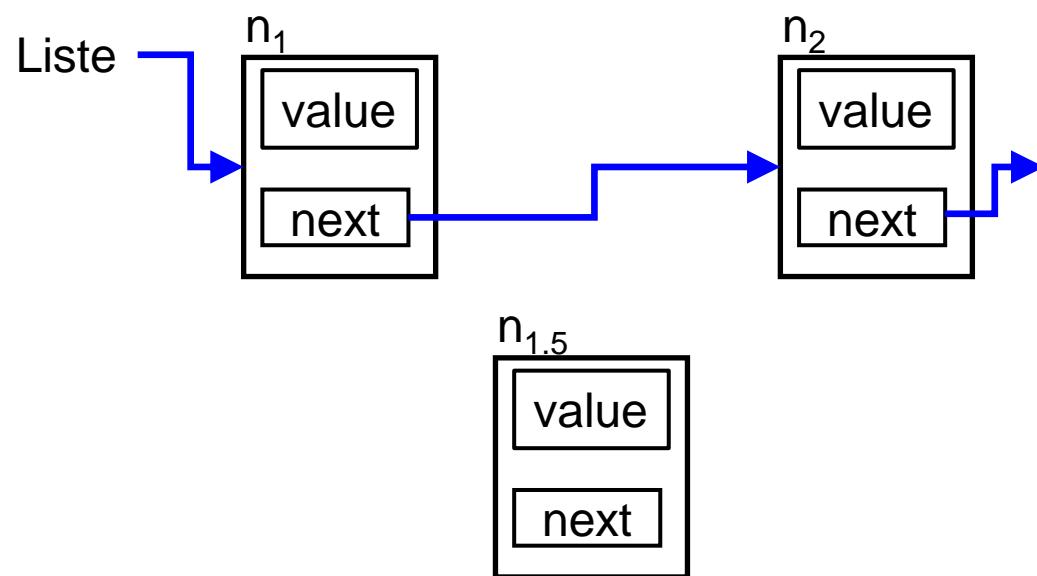
Il faut considérer tous les cas possibles:

- Insertion d'un nœud à l'intérieur d'une liste non vide
- Insertion d'un nœud en début de liste
- Insertion d'un nœud en fin de liste
- Insertion dans une liste vide

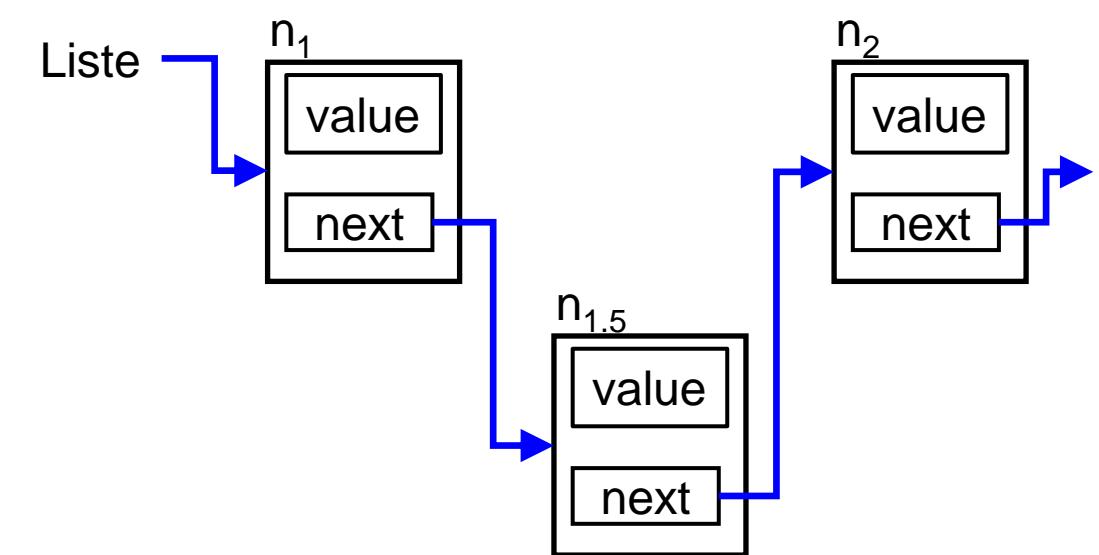
Insertion dans une liste

On considère dans un premier temps l'insertion d'un nouveau nœud $n_{1.5}$ entre 2 nœuds n_1 et n_2 qui se suivent :

Avant insertion :



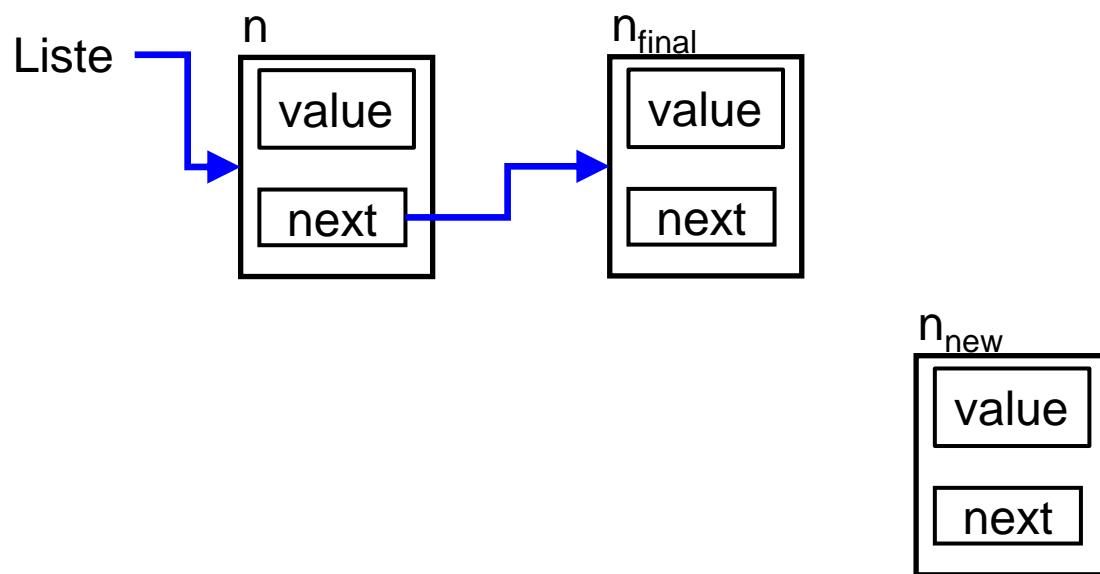
Après insertion :



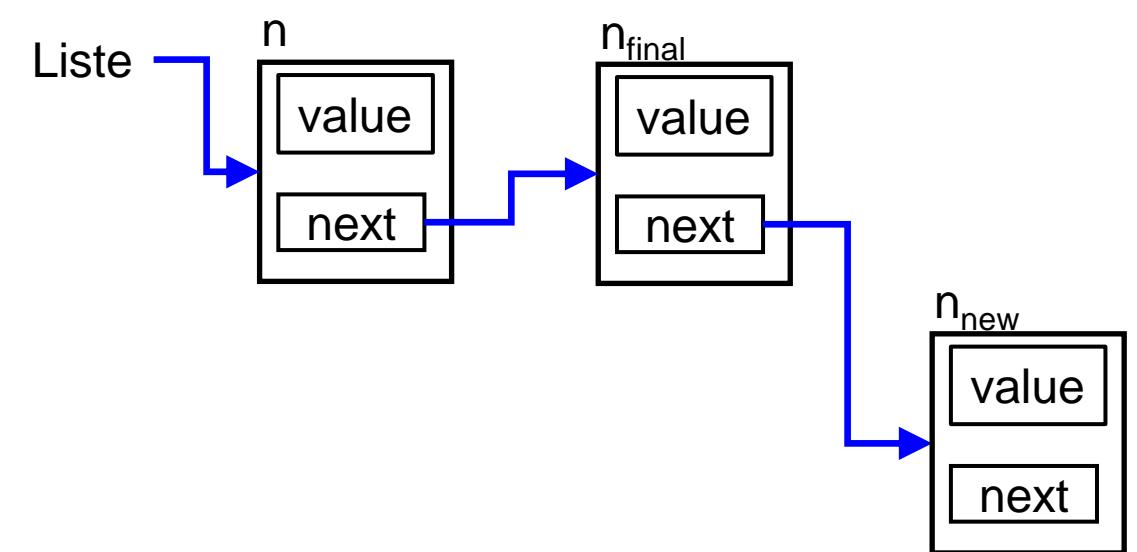
Insertion dans une liste (2)

On considère ensuite l'insertion d'un nouveau nœud n_{new} après le dernier nœud n_{final} d'une liste :

Avant insertion :



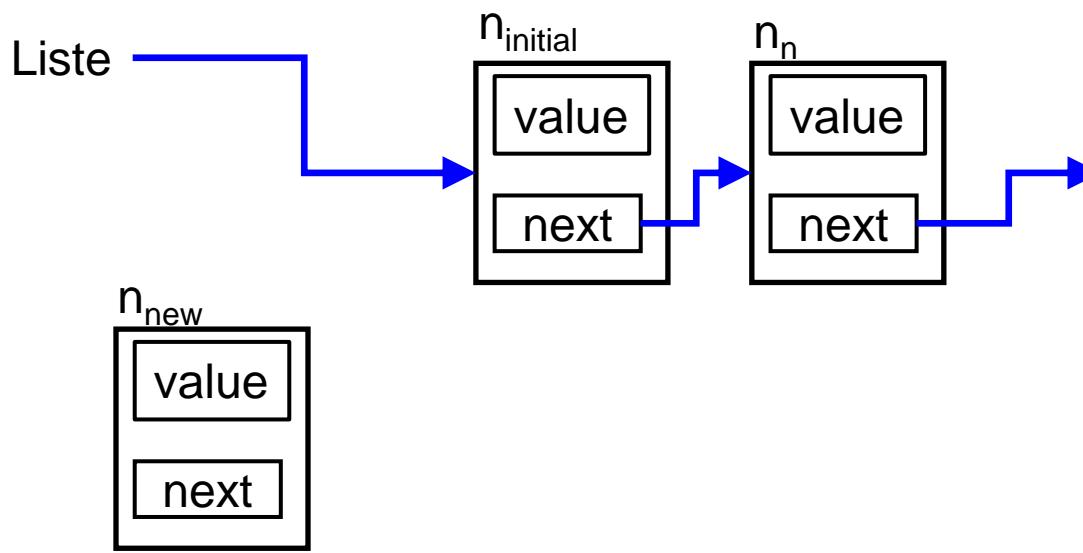
Après insertion :



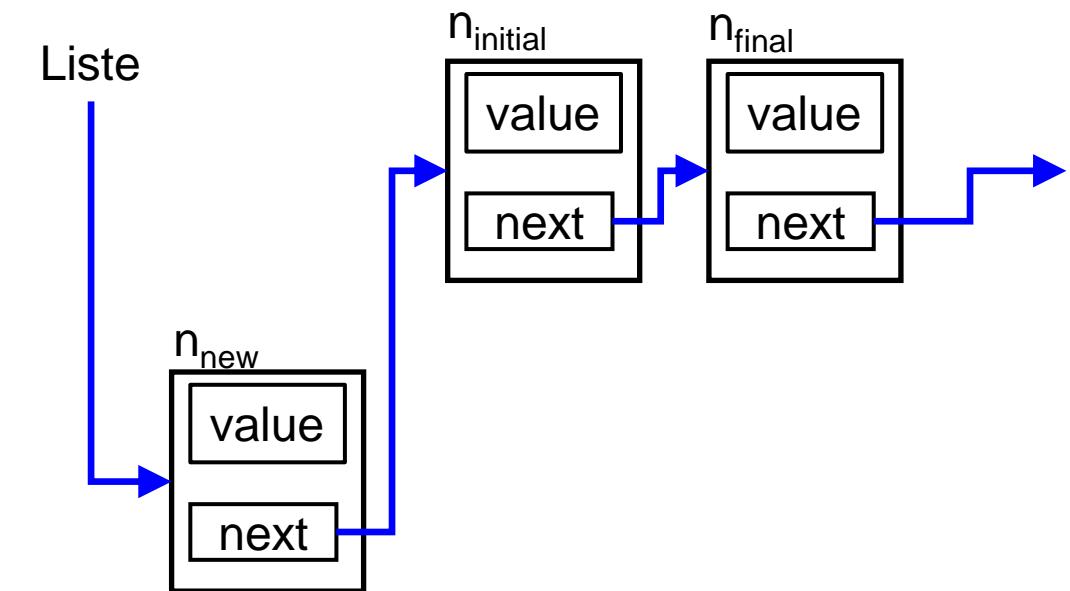
Insertion dans une liste (3)

On considère enfin l'insertion d'un nouveau nœud n_{new} au début d'une liste non vide (premier nœud n_{initial}) :

Avant insertion :



Après insertion :



Insertion dans une liste (4)

Il faudra considérer le cas de l'insertion d'un nœud dans une liste vide (c'est simple mais il ne faut pas oublier de le coder).

La plupart du temps, on le fait en testant si la liste est vide au début de l'insertion.

Insertion dans une liste (5)

Le plus simple (pour l'utilisateur du type liste) est de fournir une ou plusieurs fonctions d'insertion, par exemple:

(exemple inspiré des listes standards C++)

```
void push_front(List *L, int value);
```

```
void push_back(List *L, int value);
```

```
void insert_after(Node *N, int value);
```

Ajout en début de liste

Ajout en fin de liste

Ajout à l'intérieur d'une liste
(après le nœud passé en paramètre)

(voir le code dans le répertoire list/list_insertion)

Suppression d'un nœud d'une liste

Comme pour l'insertion, la suppression consiste en une manipulation de pointeurs.

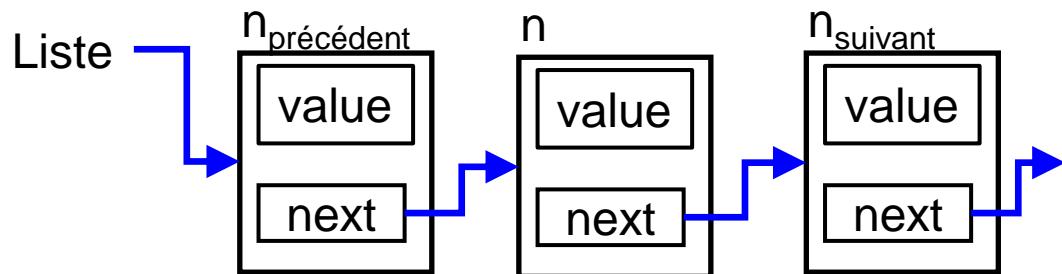
Il faut considérer tous les cas possibles:

- Suppression d'un nœud à l'intérieur d'une liste
- Suppression d'un nœud en début de liste
- Suppression d'un nœud en fin de liste
- Suppression du seul nœud d'une liste

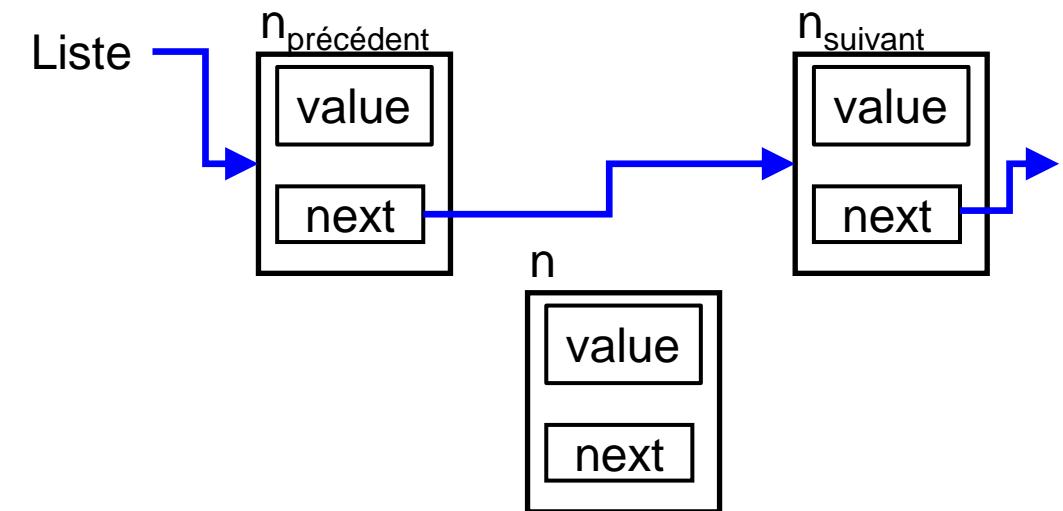
Suppression d'un nœud d'une liste (2)

On considère dans un premier temps la suppression d'un nœud n qui n'est ni le premier ni le dernier nœud d'une liste:

Avant suppression :



Après suppression :

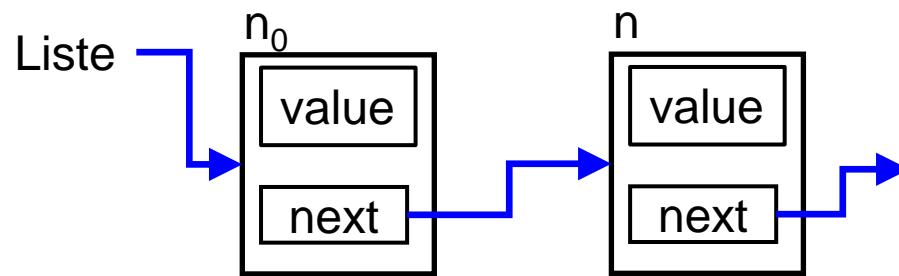


A noter qu'après avoir enlevé le nœud de la liste, il faut bien penser à mettre à zéro son pointeur next pour éviter de rentrer dans la liste par un nœud qui ne lui appartient plus.

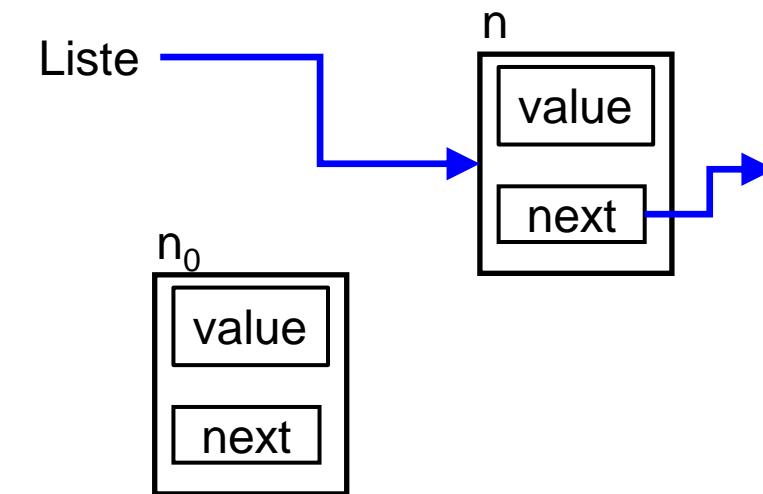
Suppression d'un nœud d'une liste (3)

On considère ensuite la suppression du premier nœud n_0 d'une liste :

Avant suppression :



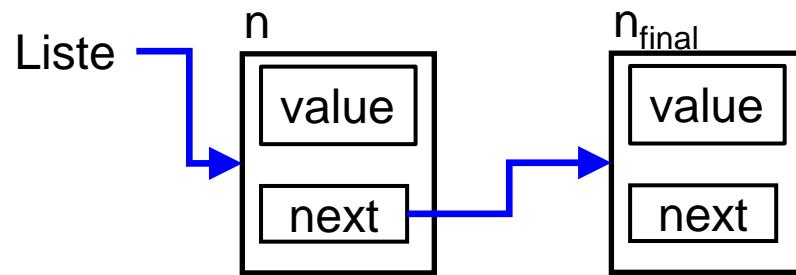
Après suppression :



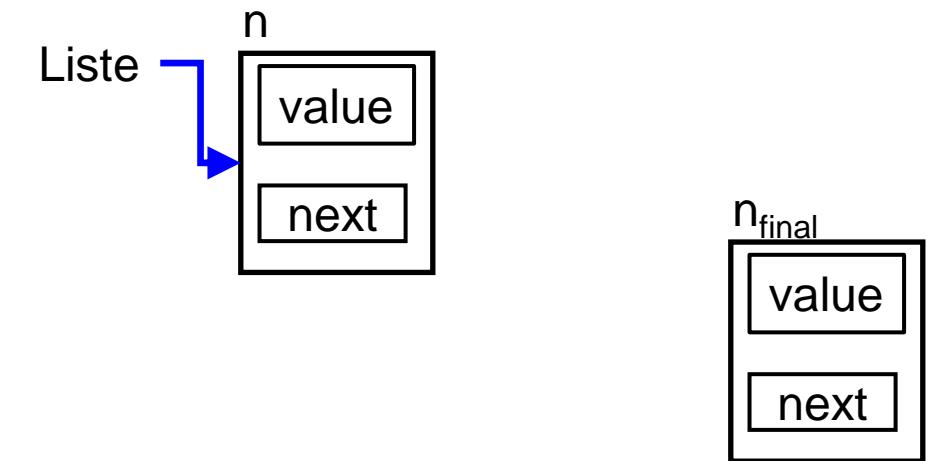
Suppression d'un nœud d'une liste (4)

On considère ensuite la suppression du dernier nœud n_{final} d'une liste :

Avant suppression :



Après suppression :



Suppression dans une liste (5)

Le plus simple (pour l'utilisateur du type liste) est de fournir une ou plusieurs fonctions de suppression, par exemple:

(exemple inspiré des listes standards C++)

```
int pop_front(List *L);
```

Suppression en début de liste (et retourne la valeur du premier nœud)

```
void pop_back(List *L);
```

Suppression en fin de liste (et retourne la valeur du dernier nœud)

```
void pop(List *L, k);
```

Suppression du $k^{\text{ème}}$ à l'intérieur d'une liste (et retourne la valeur du nœud)

(voir le code dans le répertoire list/list_suppression)

Types liste doublement chainée

Liste doublement chainée

Le type liste doublement chainée est une extension du type liste vu précédemment.

Chaque nœud possède les pointeurs pour pouvoir accéder au nœud précédent et au nœud suivant (s'ils existent)

```
struct _node
{
    int value;
    struct _node *next;
    struct _node *previous;
};
typedef struct _node Node;
```

Création/destruction de nœud

Il sera peut-être utile de définir des fonctions pour créer/détruire des nœuds :

```
Node* creeNode(int v) {
    Node* newNode = (Node*) malloc(sizeof(Node));
    newNode→value = data;
    newNode→next = NULL;
    newNode→previous = NULL;
    return newNode;
}
```

```
void detruitNode(Node ** n) {
    if (*n) {
        free(*n);
        *n = NULL;
    }
}
```

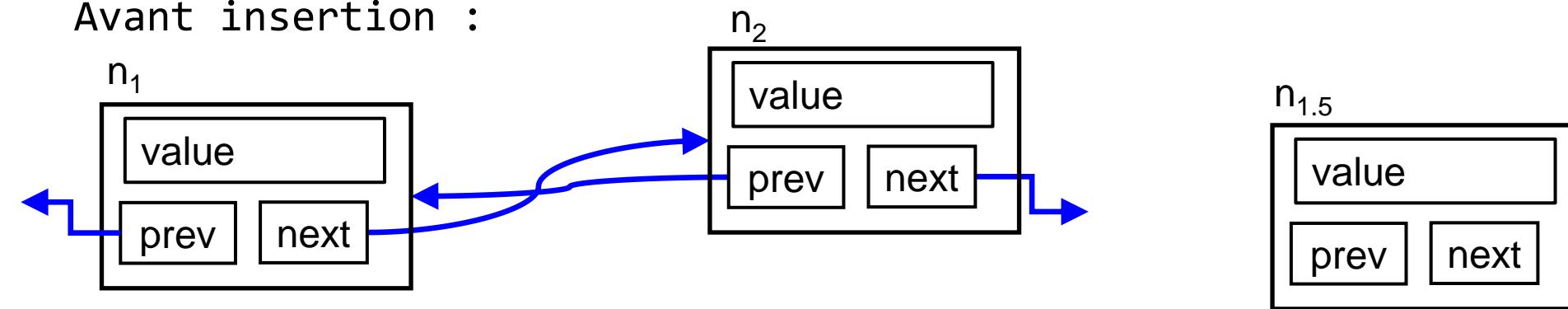
Opération d'insertion/suppression de nœuds dans une liste doublement chainée

Toutes les opérations vues pour les listes (simplement) chainées peuvent être adaptées aux listes doublement chainées

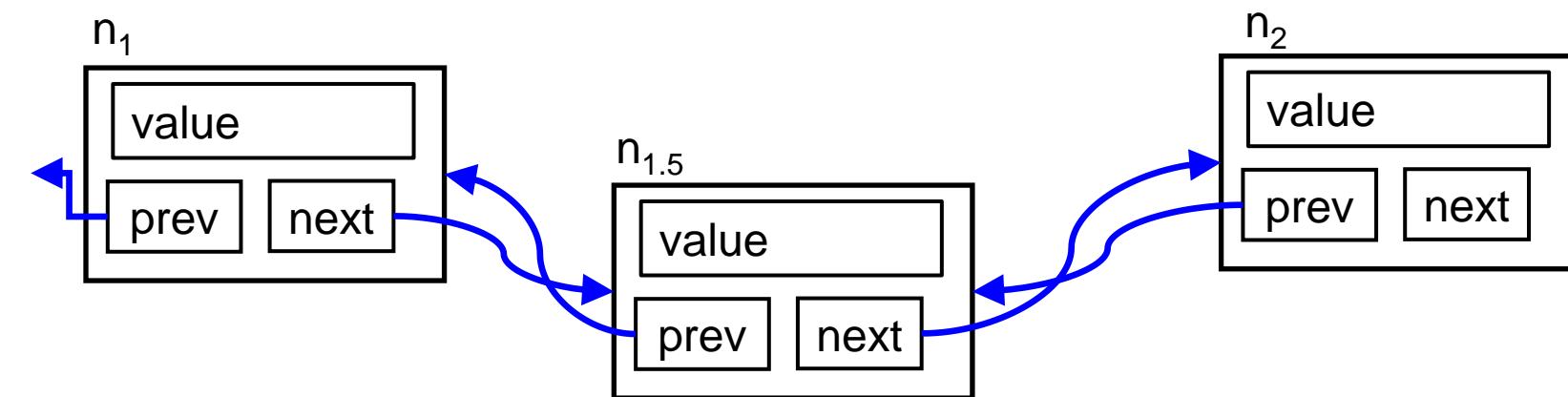
Insertion dans une liste doublement chainée

On considère l'insertion d'un nouveau nœud $n_{1.5}$ entre 2 nœuds n_1 et n_2 qui se suivent :

Avant insertion :



Après insertion :



Insertion dans une liste doublement chainée

Les fonctions réalisant l'insertion et la suppression des nœuds dans une liste doublement chainée sont laissées à titre d'exercice.

Langage C avancé : Séance 8

Outils de recherche et de correction d'erreurs

Marc TAJCHMAN

@-mail : marc.tajchman@cea.fr

CEA - DES/ISAS/DM2S/STMF/LDEI

Erreurs dans un code C (1)

Un code source peut contenir des erreurs qui provoquent des messages d'avertissement ou erreurs de compilation, des arrêts prématurés de l'exécution du code binaire, ou une exécution complète mais qui fournit des résultats faux.

On a déjà vu que le compilateur propose des options qui affichent des messages d'avertissement sur du code dont la syntaxe est correcte mais qui risquent des arrêts prématurés, de produire d'autres résultats que ceux attendus: -Wall -Wextra ...
Ces options repèrent certaines des erreurs les plus habituelles mais ne garantissent pas l'absence d'erreurs.

Il est, de toute façon, fortement conseillé d'utiliser ces options.

Erreurs et messages d'avertissement de compilation dans un code C (1)

- Les erreurs de compilation sont dues à des erreurs de syntaxe C, des fonctions ou des variables globales utilisées mais non définies, des fonctions ou des variables globales définies plusieurs fois, etc.
Ce sont les plus faciles à détecter et à corriger: le compilateur refuse de générer un code binaire en affichant des messages qui donnent en général assez d'information pour corriger ce type d'erreurs.

Les messages d'avertissemens et d'erreurs sont différents suivant les compilateurs, la version du langage C, etc.

Pour les codes destinés à être diffusés, il peut être intéressant de compiler le code source avec plusieurs compilateurs et sur plusieurs systèmes différents.

Erreurs et messages d'avertissement de compilation dans un code C (2)

- Dans le sous-répertoire erreurs_compilation/err1, compiler en tapant :

```
gcc main.c -o code
```

Corriger les erreurs éventuelles jusqu'à ce que le code compile.
Exécutez le code binaire.

- Recompilez avec les options de vérification, en tapant :

```
gcc -Wall -Wextra main.c -o code
```

Modifier le code source pour tenir compte des messages d'avertissement.

Exécutez le code binaire

Erreurs et messages d'avertissement de compilation dans un code C (3)

- Premier enseignement : il peut y avoir plusieurs façons de corriger les erreurs

Exemple :

```
double f(int n) {  
    ...  
}  
  
int main() {  
    double x = f(10, 3.4);  
}
```



```
double f(int n, double x) {  
    ...  
}  
  
int main() {  
    double x = f(10, 3.4);  
}
```

```
double f(int n) {  
    ...  
}  
  
int main() {  
    double x = f(10);  
}
```

Les 2 modifications donnent une syntaxe correctes.

Pour choisir, il faut comprendre ce qu'est censé faire le code

Erreurs et messages d'avertissement de compilation dans un code C (4)

- Deuxième enseignement : avoir une syntaxe correcte ne suffit pas, le code peut se compiler (éventuellement avec des messages d'avertissement) mais s'exécuter en fournissant des résultats faux ou en s'arrêtant sur une erreur.

Donc, il faut utiliser toutes les options vérifications à la compilation et tenir compte des messages d'avertissement.

On verra souvent qu'il faut aussi faire des vérifications à l'exécution (voir plus loin).

Erreurs dans l'appel des fonctions ou dans l'utilisation des variables globales

- Dans le sous-répertoire erreurs_compilation/err2, compiler en tapant :

```
gcc main.c f.c -o code
```

Le compilateur refuse de créer un binaire exécutable parce que la variable globale `m` est définie 2 fois. Modifier `f.c` en ajoutant « `extern` » devant la définition de `m` dans `f.c`:

```
double m;
```



```
extern double m;
```

Le message du compilateur signifie que la variable globale `m` ne peut pas être définie séparément dans plusieurs fichiers, il faut la définir dans un seul fichier et utiliser « `extern` » dans les autres.

Erreurs dans l'appel des fonctions ou l'utilisation des variables globales

- Recompile et exécuter le code en tapant ./code
Le code s'exécute mais affiche des valeurs incohérentes, sur ma machine:

```
> f: x = 0 m = 2.47033e-323
> f: r = 0
> main: f(10) = 9
```

- Le problème vient de ce fait le compilateur quand on tape:

```
gcc main.c f.c -o code
```

- Compilation séparée de main.c (fichier caché main.o)
- Compilation séparée de f.c (fichier caché f.o)
- Rassemblement des fichiers main.o et f.o dans l'exécutable code

Le compilateur vérifie que toutes les fonctions et les variables globales utilisées sont définies une et une seule fois, **mais pas que les utilisations des fonctions ou variables globales sont conformes à leur définition.**

Exploration des fichiers binaires

Pour voir ce qui se passe, recompiler les fichiers en plusieurs étapes:

```
gcc -c main.c  
gcc -c f.c  
gcc main.o f.o -o code
```

Et utiliser la commande `objdump` qui montre le contenu des fichiers compilés (l'exécutable final ou les fichiers objets .o):

```
objdump -t <nom de fichier>
```

On peut aussi utiliser la commande `nm` qui affiche les mêmes informations sous une forme différente:

```
nm <nom de fichier>
```

Exploration des fichiers binaires

```
objdump -t main.o
```

```
...
0000000000000000 g    0 .bss    0000000000000004 m
0000000000000000 g    F .text    0000000000000049 main
0000000000000000          *UND*    0000000000000000 f
0000000000000000          *UND*    0000000000000000 printf
...
```

```
objdump -t f.o
```

```
...
0000000000000000 g    F .text    0000000000000074 f
0000000000000000          *UND*    0000000000000000 m
0000000000000000          *UND*    0000000000000000 printf
...
```

UND : signifie que le symbole (fonction, variable globale) est utilisé dans le fichier .o mais sa définition est à l'extérieur.

Exploration des fichiers binaires

```
objdump -t code
```

(extraits)

```
0000000000001192 g F .text 0000000000000074 f
0000000000000000 F *UND* 0000000000000000 printf@GLIBC_2.2.5
000000000004014 g O .bss 0000000000000004 m
000000000001149 g F .text 0000000000000049 main
```

Tous les symboles sont bien définis (f, m, main), c.-à-d. le compilateur les a trouvés dans un des .o.

Seule la fonction printf est non définie dans le binaire mais l'exécutable utilise la fonction printf du système.

Utilisez la commande ldd pour voir ce que l'exécutable utilise dans la machine:

```
ldd code
```

```
linux-vdso.so.1 (0x00007fff391f7000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ff50749f000)
/lib64/ld-linux-x86-64.so.2 (0x00007ff5076cc000)
```

Exploration des fichiers binaires

L'information qui nous intéresse ici est que les symboles (fonctions, variables globales) sont enregistrés dans les .o avec leur nom, mais pas avec leur type (variables globales), pas avec leur type de retour ou leurs types et nombre de paramètres (fonctions).

Ce qui fait que le compilateur ne voit pas qu'il y des différences entre les symboles de même nom mais dans des .o différents.

Ceci est particulier au C !!

Correction de l'erreur

Pour corriger ce type d'erreur, on procède généralement comme suit, pour chaque variable globale:

- On met la déclaration de la variable globale dans un fichier d'entête (se terminant souvent par .h) avec le mot clef « `extern` »
- On remplace la déclaration de cette variable globale dans les .c par l'inclusion du fichier d'entête ci-dessus
- Dans un des fichiers .c (le « propriétaire » de la variable globale), on définit la variable globale (sans le mot clef « `extern` ») - C accepte de déclarer une variable globale 2 fois (avec et sans « `extern` »)

On peut réutiliser le même fichier d'entête pour plusieurs variables globales.

Correction de l'erreur

Pour chaque fonction:

- On met le prototype de la fonction dans un fichier d'entête (se terminant souvent par .h) avec ou sans le mot clef « `extern` » (qui est ajouté par défaut)
- On remplace le prototype de cette fonction dans les .c par l'inclusion du fichier d'entête ci-dessus
- Dans un des fichiers .c (le « propriétaire » de la fonction), on définit la fonction (avec le code interne de la fonction)

On peut réutiliser le même fichier d'entête pour les prototypes de plusieurs fonctions.

Cela permet au compilateur C de contrôler la cohérence des fonctions et variables globales.

Voir le répertoire `symboles_globaux` pour un exemple incomplet à compléter.

Un code avec une syntaxe correcte qui compile peut contenir des erreurs

Quand on exécute un code binaire C (produit de la compilation d'un code source), il peut se produire des erreurs à l'exécution ou des résultats faux.

Dans ce cas, il y a plusieurs moyens pour les repérer et les corriger:

➤ Réviser le code source

En général, valable pour un petit code et si on a une bonne idée du problème

➤ Utiliser printf pour afficher des variables au cours du calcul

➤ Utiliser la fonction assert pour contrôler des conditions au cours du calcul

➤ Utiliser un outil de débug

Il s'agit outils qui permettent d'exécuter un code instruction par instruction et d'afficher des variables du code pendant l'exécution.

➤ Utiliser un outil de test de la mémoire dynamique

Si on soupçonne un problème avec l'utilisation de la mémoire dynamique.

Afficher des valeurs intermédiaires pendant le calcul

Afficher des valeurs intermédiaires avec printf

Quand on soupçonne que des variables prennent des valeurs non correctes pendant le calcul, on peut afficher ces valeurs en plusieurs endroits du code source.

Avantages:

- Ce moyen est souvent celui qui reste disponible, si aucun autre outil de recherche d'erreur ne l'est.
- Si on a une idée précise de la variable et de l'endroit qui pose problème, c'est un moyen de contrôle simple.

Inconvénients:

- C'est un processus qui peut-être long, surtout si un code source est de grande taille et qu'on n'a pas d'idée précise où se situe le problème.
- Ajouter des instructions au code (même des printf), le modifie et peut changer son comportement.

Dans certains cas, un code peut s'arrêter sur une erreur sans printf et s'exécuter jusqu'à la fin avec des printf

Exemple

Dans le répertoire racine, un fichier C (main.c) contient une fonction qui calcule la racine carrée d'un nombre par une méthode de type Newton.

Le programme principal demande à l'utilisateur de rentrer un nombre et affiche la racine carrée calculée de ce nombre.

Compiler et exécuter plusieurs fois ce code en rentrant des valeurs quelconques positives, nulles et négatives (2, 3, 0, -1, -6).
Quel est le comportement du code ?

Afficher les valeurs successives de r dans la fonction racine.

Utilisation de la fonction assert

Tester des conditions dans le code source avec assert

C propose une fonction « assert » qui teste une condition. Si la condition est vraie, l'exécution continue, sinon l'exécution s'arrête en affichant un message d'erreur. Ne pas oublier d'inclure le fichier assert.h. Voir le répertoire tests/racine2.

Il est possible de désactiver les asserts en compilant le code avec l'option NDEBUG (no debug):

```
gcc -DNDEBUG main.c -o racine
```

Dans ce cas, les instructions asserts seront ignorées.

Si la variable qui peut poser problème, est une variable initialisée par l'utilisateur, à la place de assert (ou en plus de assert), il faut plutôt ajouter un test explicite:

```
scanf("%lg", &x);
if (x < 0) { printf("entrer un nombre positif ou nul"); exit(-1); }
```

Conseils avec les instructions « assert »

En général, on procède comme suit :

- pendant la phase d'écriture du code, on compile en mode debug et on active les asserts et les tests sur le données d'entrée

```
gcc -g main.c -o racine
```

- quand le code a été testé et avant de le fournir aux utilisateurs, on compile en mode optimisé et on désactive les assert mais on garde les tests sur les données d'entrée

```
gcc -O2 -DNDEBUG main.c -o racine
```

La fonction assert est plutôt destinée aux développeurs de code et pas aux utilisateurs.

Préconditions avec les instructions « assert »

Dans chaque fonction du code source, on ajoute des "assert" au début de la fonction pour contrôler la validité des paramètres de cette fonction (si cela à un sens)

Exemple: produit de matrices A, B avec résultat la matrice C (le type matrice utilisé est celui vu à la séance précédente)

```
void produit(const Matrice *A, const Matrice *B, Matrice *C) {  
  
    assert(A.n() == C.n());  
    assert(B.m() == C.m());  
    assert(A.m() == B.n());  
  
    for (int i=0; i<C.n(); i++)  
        for (int j=0; j<C.m(); j++)  
            for (int k=0; k<A.m(); k++)  
                C.c[i*C.m() + j] += A.c[i*A.m() + k]*B.c[k*B.m() + j];  
}
```

Postconditions avec les instructions « assert »

Dans chaque fonction du code source qui produit des résultats, on ajoute des "assert" en fin de la fonction pour contrôler la validité des résultats de cette fonction (si cela à un sens)

Exemple: calcul par une formule de Taylor de $\sin(x)$

```
double sinus(double x) {  
    double s = x - x*x*x/6 + ...;  
    assert(s>-1 && s<1);  
}
```

L'utilisation de pré- et post-conditions est parfois appelée "programmation par contrat" ou "programmation défensive"

Utilisation d'un outil de debug : exemple avec gdb

Utilisation d'un outil de debug

Les outils logiciels de debug permettent :

- d'exécuter instruction par instruction un code binaire,
- d'afficher la valeur d'une variable du code à un instant
- d'exécuter le code jusqu'à une instruction donnée
- d'exécuter le code tant qu'une condition est vraie
- d'exécuter le code tant qu'une variable ne change pas de valeur
- ...

C'est donc un outil très puissant pour examiner ce qui se passe pendant l'exécution d'un code.

Il existe plusieurs outils de debug (on dit aussi debugger): gdb (linux), lldb (linux), codeview (intégré à visual studio dans windows), ...

Gdb est souvent à la base des outils de débug disponibles dans les environnements de développement (codeblocks, visual code, eclipse,...)

Debugger gdb

On utilisera ici gdb en ligne de commande (c'est le debugger le plus utilisé et il est parfois le seul disponible)

Pour utiliser pleinement les fonctionnalités d'un debugger, il faut compiler « en mode debug » et « non optimisé »:

À la place de
l'une des commandes

```
gcc main.c -o code  
gcc -O2 main.c -o code  
gcc -g -O2 main.c -o code
```

Il faut utiliser
ou mieux:

```
gcc -g main.c -o code  
gcc -g -O0 main.c -o code
```

Si on utilise cmake, il faut ajouter l'option CMAKE_BUILD_TYPE=Debug à l'étape de configuration:

```
cmake -S srcDir -B buildDir -DCMAKE_BUILD_TYPE=Debug
```

Debugger gdb

On utilisera ici gdb en ligne de commande (c'est le debugger le plus utilisé et il est parfois le seul disponible)

Pour utiliser pleinement les fonctionnalités d'un debugger, il faut compiler « en mode debug » et « non optimisé »:

À la place de
l'une des commandes

```
gcc main.c -o code  
gcc -O2 main.c -o code  
gcc -g -O2 main.c -o code
```

Il faut utiliser
ou mieux:

```
gcc -g main.c -o code  
gcc -g -O0 main.c -o code
```

Si on utilise cmake, il faut ajouter l'option CMAKE_BUILD_TYPE=Debug à l'étape de configuration:

```
cmake -S srcDir -B buildDir -DCMAKE_BUILD_TYPE=Debug
```

Debugger gdb : utilisation (1)

Pour exécuter le code binaire « code » sous le contrôle de gdb, il faut taper :

```
gdb code
```

Pour arrêter gdb, taper ctrl-d (et répondre y pour confirmer).

Si le message suivant (ou similaire) s'affiche :

```
This GDB supports auto-downloading debuginfo from the following URLs:  
<https://debuginfod.ubuntu.com>  
Enable debuginfod for this session? (y or [n])
```

Quitter gdb et tapez la commande (dans le terminal) :

```
echo "set debuginfod enabled on" >> $HOME/.gdbinit
```

Debugger gdb : utilisation (2)

Quand on lance gdb (avec le nom du code binaire exécutable), gdb se met en attente d'une commande.

Les principales commandes sont:

- `ctrl-x a`: permet de voir le code source en même temps qu'on tape des commandes de gdb (`ctrl-x a` à nouveau pour sortir)
- `run` : lance l'exécution (jusqu'à la fin ou une erreur d'exécution ou un point d'arrêt (voir ci-dessous))
- `quit` : sort de gdb (confirmer en tapant `y`)
- `b xxx.c:n` (break) : met un point d'arrêt au début d'une instruction à la ligne numéro n du fichier xxx.c
 - Gdb affiche un numéro de point d'arrêt qui permettra de le supprimer, de le désactiver, de le réactiver, ...
- `p v` : affiche la valeur de la variable v

Debugger gdb : utilisation (3)

- `del k` : supprime le point d'arrêt numéro k
- `disable k` : si le point d'arrêt numéro k est actif, le désactiver
- `enable k` : si le point d'arrêt numéro k est inactif, le réactiver
- `cont` (continue) : si l'exécution est arrêtée à un point d'arrêt, continuer jusqu'à la fin ou au point d'arrêt suivant
- `n` (next) : exécuter une seule instruction (si l'instruction est un appel d'une fonction, « sauter au-dessus de l'appel))
- `s` (step) : exécuter une seule instruction (si l'instruction est un appel d'une fonction, gdb se met au début du code de la fonction)
- `w x` (watch) : où x est le nom d'une variable simple (pas une structure), si on tape cont ou run à la suite de la commande watch, l'exécution se poursuit tant que la variable x ne change pas de valeur

Debugger gdb : utilisation (4)

- **b xxx.c:n if « condition »** : pose un point d'arrêt conditionnel à la ligne n du fichier xxx.c (l'exécution s'arrêtera au point d'arrêt seulement si la condition est vraie)

Exemple: extrait du fichier « toto.c »:

```
21: for (i=0; i<1000; i++)  
22:     a[i] = 2*b[i]-b[i-1]-b[i+1];
```

Si on tape : **b toto.c:22 if (i==500)**

l'exécution s'arrêtera à l'itération numéro 501 de la boucle sur i

- **bt** : affiche la pile d'appels (la liste des lignes par où est passé le code pour arriver à la ligne courante)

Debugger gdb : utilisation (5)

Il existe (beaucoup) d'autres commandes, voir, par exemple, le manuel de gdb (<https://sourceware.org/gdb/current/onlinedocs/gdb.html>)

Pour se rappeler les commandes de gdb, le mieux est de s'exercer sur des codes :

Recompiler le code dans le répertoire sqrt en mode debug et utiliser gdb pour exécuter le code, mettre des points d'arrêts, afficher des variables, etc.

Comprendre les erreurs d'utilisation de la mémoire dynamique :
l'outil valgrind

Erreurs d'exécution : utilisation de la mémoire dynamique

Un grand nombre d'erreurs d'exécution sont dues à une mauvaise utilisation de la mémoire dynamique:

- Utilisation d'une mémoire dynamique non (ou pas encore) réservée
- Utilisation d'un pointeur en dehors de la mémoire dynamique réservée
- Utilisation d'une mémoire dynamique après qu'elle ait été rendue au système.

L'outil valgrind

valgrind est un logiciel conçu à l'origine, pour détecter et aider à corriger des problèmes de gestion de la mémoire dynamique pour l'environnement KDE dans linux.
Il s'est avéré un des outils les plus efficaces pour cela.

En quelques mots:

- **valgrind** crée une machine virtuelle dans laquelle il exécute le code.
- Dans cette machine virtuelle, **valgrind** peut suivre toutes les opérations sur la mémoire dynamique (réservation, libération, modification) et repérer ce qui est problématique.
- Une conséquence, mais qui est acceptable, compte-tenu de l'efficacité de l'outil, est que le temps d'exécution est (considérablement) allongé.

L'outil valgrind

Valgrind est un logiciel conçu à l'origine, pour détecter et aider à corriger des problèmes de gestion de la mémoire dynamique pour l'environnement KDE dans linux.

Il s'est avéré un des outils les plus efficaces pour cela.

En quelques mots:

- valgrind crée une machine virtuelle dans laquelle il exécute le code.
- Dans cette machine virtuelle, valgrind peut suivre toutes les opérations sur la mémoire dynamique (réservation, libération, modification) et repérer ce qui est problématique.
- Une conséquence, mais qui est acceptable, compte-tenu de l'efficacité des vérifications, est que le temps d'exécution est (considérablement) allongé.

Utilisation de valgrind

Utiliser valgrind est simple, il suffit de taper valgrind suivi du nom du code binaire:

```
valgrind code
```

valgrind affiche à l'écran, en plus des affichages normaux du code, tout ce qu'il a noté dans l'utilisation de la mémoire dynamique.

Compte-tenu du (potentiellement) grand nombre de message affiché, il est préférable de rediriger les affichages écran dans un fichier qu'on examinera à la fin de l'exécution :

```
valgrind code >& log
```

(crée un fichier de nom log qui contient les messages de valgrind et les affichages du code)

Exemple d'utilisation de valgrind

Se mettre dans le répertoire `memoire_dynamique/exemple2`, compiler en mode debug et exécuter la commande

```
valgrind ./code >& log
```

On examinera en séance le fichier log avec les messages valgrind.

Comprendre les erreurs d'utilisation de la mémoire dynamique :
l'outil sanitizer

L'outil sanitizer

Plusieurs outils ont été développés chez google, puis intégrés dans les compilateurs (d'abord clang, puis gcc et icx (le compilateur Intel))

C'est un fichier binaire (une librairie) à ajouter à la compilation. Il contient des fonctions qui interceptent toutes les utilisations de la mémoire dynamique.

Pour compiler le code, il faut ajouter une option `-fsanitize=...` et, éventuellement, une librairie (exemples pour gcc et clang) :

```
gcc -g -fsanitize=address main.c -static-libasan -o code
```

```
clang -fsanitize=memory -fno-omit-frame-pointer -g main.c -o code
```

Exemple d'utilisation des sanitizers

Le répertoire `mémoire_dynamique/exemple2` contient plusieurs versions avec différents types d'erreurs possibles.

Un Makefile compile chaque fichier séparément pour générer des codes, avec et sans les options sanitizer.

- Utiliser valgrind avec les exécutables compilés sans les options sanitizer
- Exécuter directement, les exécutables générés avec les options sanitizer

Interpréter les sorties.

La librairie sanitizer est peut-être un futur outil standard.

Cet outil n'est pas encore aussi efficace que valgrind mais est beaucoup plus rapide.

Par contre, il faut recompiler tous les fichiers sources.

Dans la pratique

Aucun des outils présenté ici n'est capable de trouver toutes les erreurs possibles.

Il faut donc, en général, les combiner

Cas des vecteurs de taille fixe – débordement de tableau

Compiler et exécuter le code contenu dans le répertoire « vecteurs-statiques ».

Expliquer l'erreur dans ce qui est affiché.

Dans la séance, on utilisera gdb pour repérer la cause de l'erreur.

Ce genre d'erreur (débordement de tableau) est souvent utilisé par les virus informatiques qui parviennent à écraser des zones mémoires normalement inaccessibles dans un code et à modifier le comportement du code.

Il est donc important de corriger ce type d'erreur.

Malheureusement, les outils de debug arrivent difficilement à trouver la cause de l'erreur (dans l'exemple, on a repéré la zone écrasée en affichant toutes les variables en plusieurs points du code).

Langage C avancé : Séance 9

Arguments fournis à l'exécution

Fonctions récursives

Pointeurs de fonction

Librairies statiques et dynamiques

Marc TAJCHMAN

@-mail : marc.tajchman@cea.fr

CEA - DES/ISAS/DM2S/STMF/LDEI

Arguments fournis à l'exécution

Il est possible dans un code C, de donner la possibilité à l'utilisateur du binaire de fournir des valeurs à l'exécution.

Pour cela, la fonction principale (`main`) peut s'écrire de 2 façons différentes:

```
int main()
{
    ...
}
```

```
int main(int argc, char **argv)
{
    ...
}
```

Si l'utilisateur lance l'exécutable compilé « code » en tapant:

./code a b c

où a, b, c sont les **paramètres du code** (des mots composés avec des lettres et des chiffres mais sans espaces dans ces mots)

Arguments fournis à l'exécution

Arguments fournis à l'exécution (2)

Dans le cas où le code source de la fonction principale commence par :

```
int main()
```

les paramètres a, b, c seront ignorés

Dans le cas où le code source de la fonction principale commence par :

```
int main(int argc, char **argv)
```

les paramètres a, b, c seront contenus dans argc et argv

`argv` est un vecteur de chaînes
de caractères

`argc` est le nombre de chaînes
dans argv

Arguments fournis à l'exécution (3)

Dans le cas où on utilise argc et argv, si l'utilisateur du binaire tape

```
./build/code test y 12
```

quand l'exécution entre dans la fonction principale, argc et argv contiendront:

```
argc : 4
argv[0] : "./build/code"
argv[1] : "test"
argv[2] : "y"
argv[3] : "12"
```

argv[0] est la chaîne de caractères qui est la commande : nom du binaire *y compris le chemin d'accès à ce binaire*, les autres éléments de argv sont les paramètres d'exécution.

Arguments fournis à l'exécution (3)

Si l'utilisateur du même binaire, tape

```
./build/code
```

quand l'exécution entre dans la fonction principale, argc et argv contiendront:

```
argc : 1  
argv[0] : "./build/code"
```

argc est toujours au moins égal à 1

Arguments fournis à l'exécution (4)

Le programmeur doit vérifier que les paramètres fournis par l'utilisateur du code binaire sont ceux attendus (certains paramètres peuvent avoir une valeur par défaut si l'utilisateur n'en fournit pas).

Exemple de code de vérification (le code a besoin de 2 paramètres (un entier et un réel), le second a une valeur par défaut (1.5)):

```
int n;
double v;
char * end;
if (argc == 1 || argc > 3) message_erreur(1);
n = strtol(argv[0], &end, 10);
if (end != NULL) message_erreur(2);
if (argc == 3) {
    v = strtod(argv[2], &end);
    if (end != NULL) message_erreur(3);
}
else
    v = 1.5;
```

`message_erreur` est une fonction (à écrire) qui affiche un message d'erreur en fonction de son paramètre

Fonctions récursives

Fonctions récursives

En C, et dans d'autres langages, une fonction peut contenir un appel à elle-même. Rappelons (voir séance 5) que chaque fois qu'on

- rentre dans une fonction, cette fonction **ajoute dans la pile** ses variables locales et des paramètres,
- sort d'une fonction, cette fonction **enlève de la pile** ses variables locales et ses paramètres.

Exemple:

```
int f(int n) {
    int k = f(n-1) * 2;
    return k;
}

int main() {
    int r = f(5);
    printf("r = %d\n", r);
    return 0;
}
```

Ce code a pour but de calculer 2^p
($p = 5$, dans main.c)

Compiler ce code :

```
gcc main.c -o ex1
```

L'exécuter ... il devrait s'arrêter
en affichant un message d'erreur.

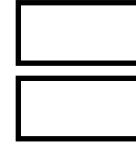
Fonctions récursives

Stack

main:

:

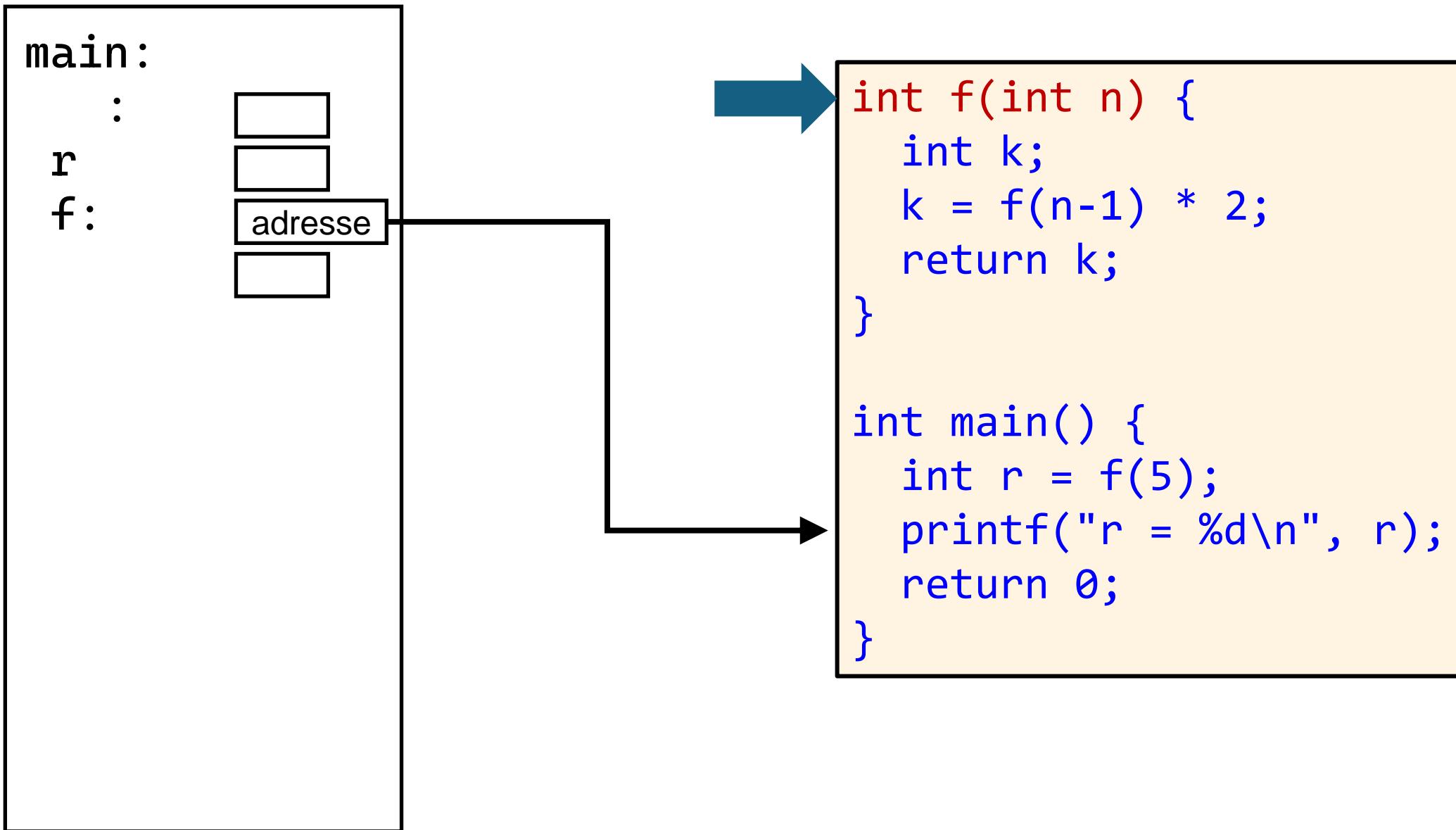
r



```
int f(int n) {  
    int k;  
    k = f(n-1) * 2;  
    return k;  
}  
  
int main() {  
    int r = f(5);  
    printf("r = %d\n", r);  
    return 0;  
}
```

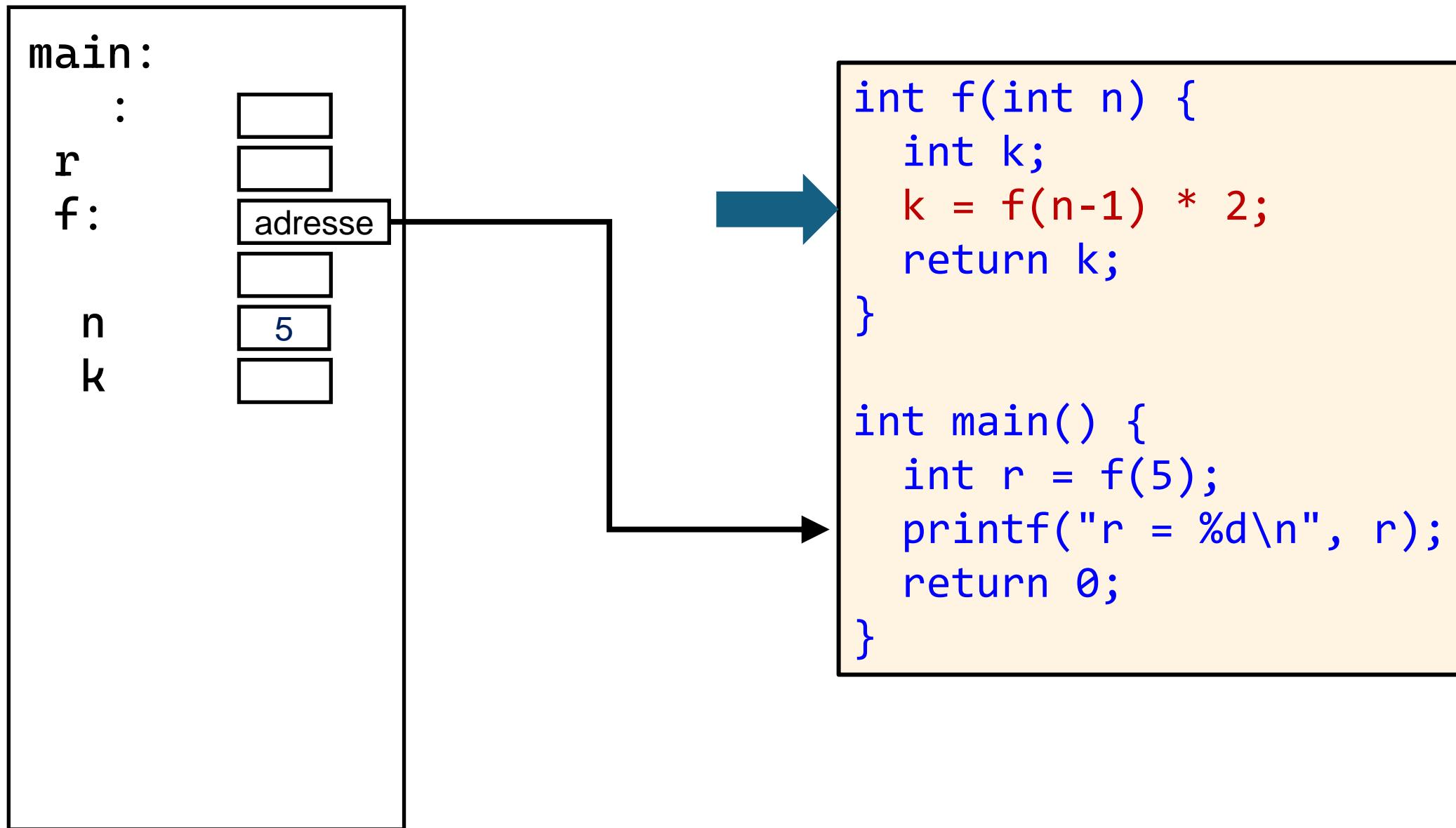
Fonctions récursives

Stack



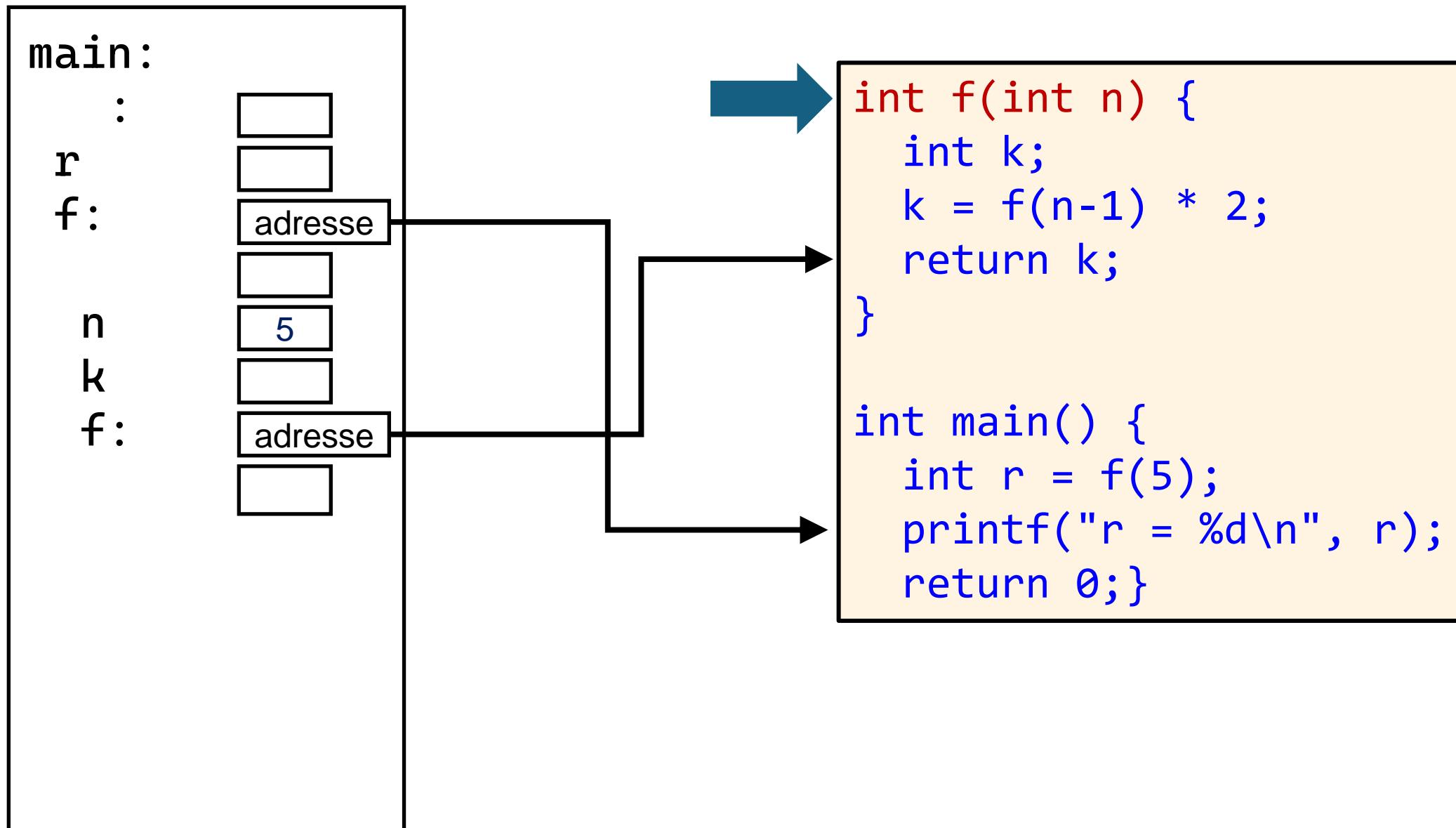
Fonctions récursives

Stack



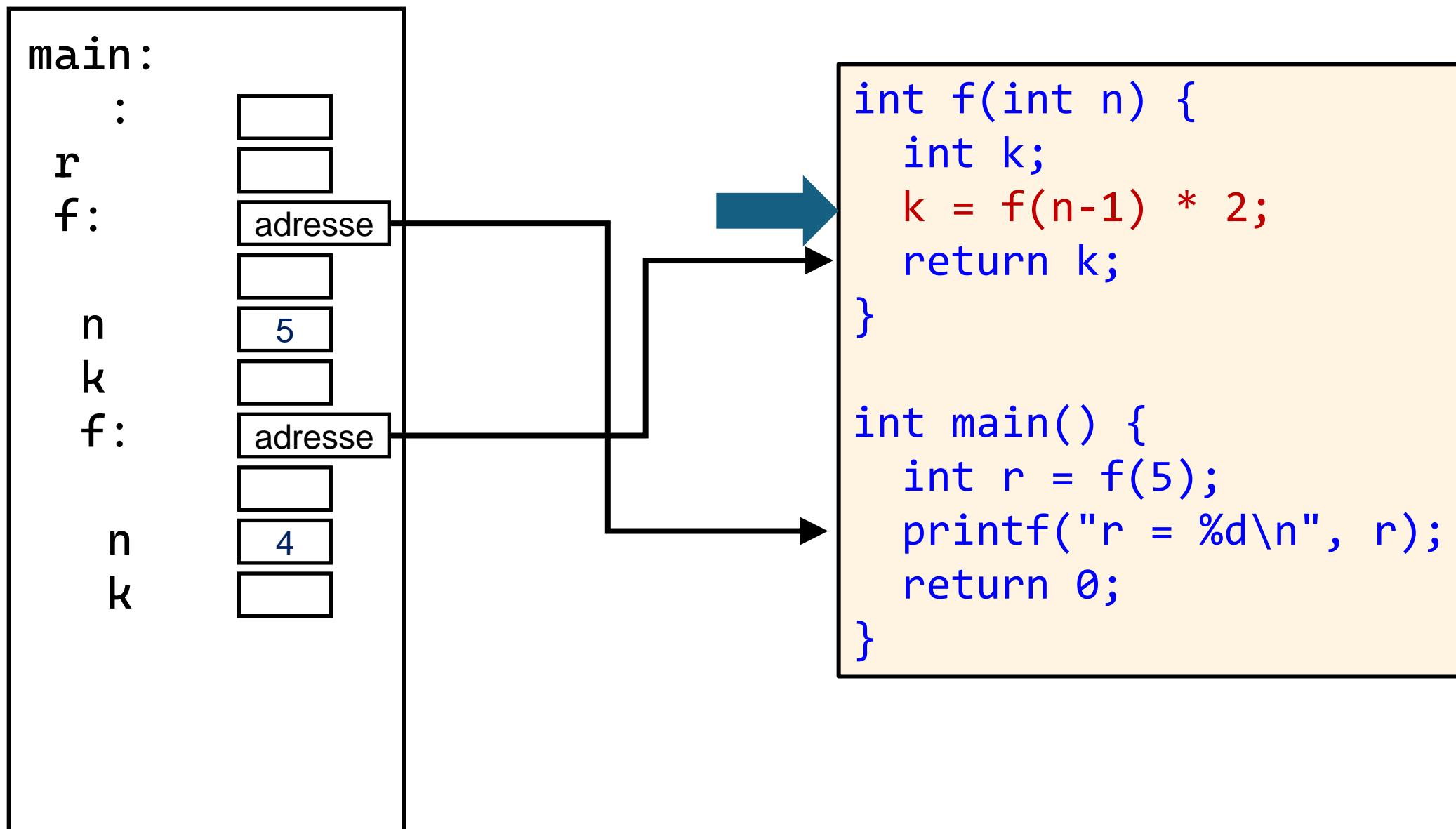
Fonctions récursives

Stack



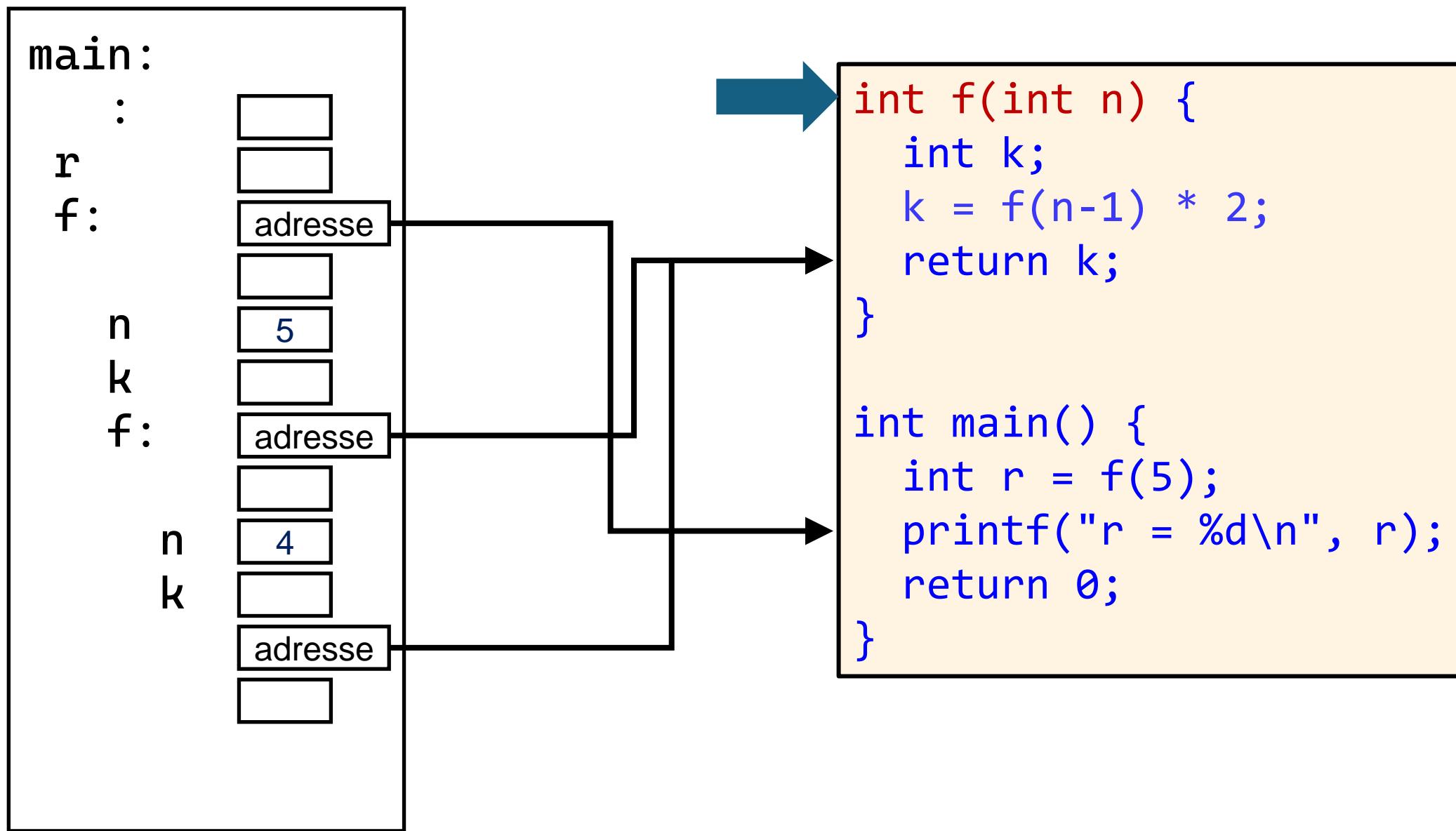
Fonctions récursives

Stack



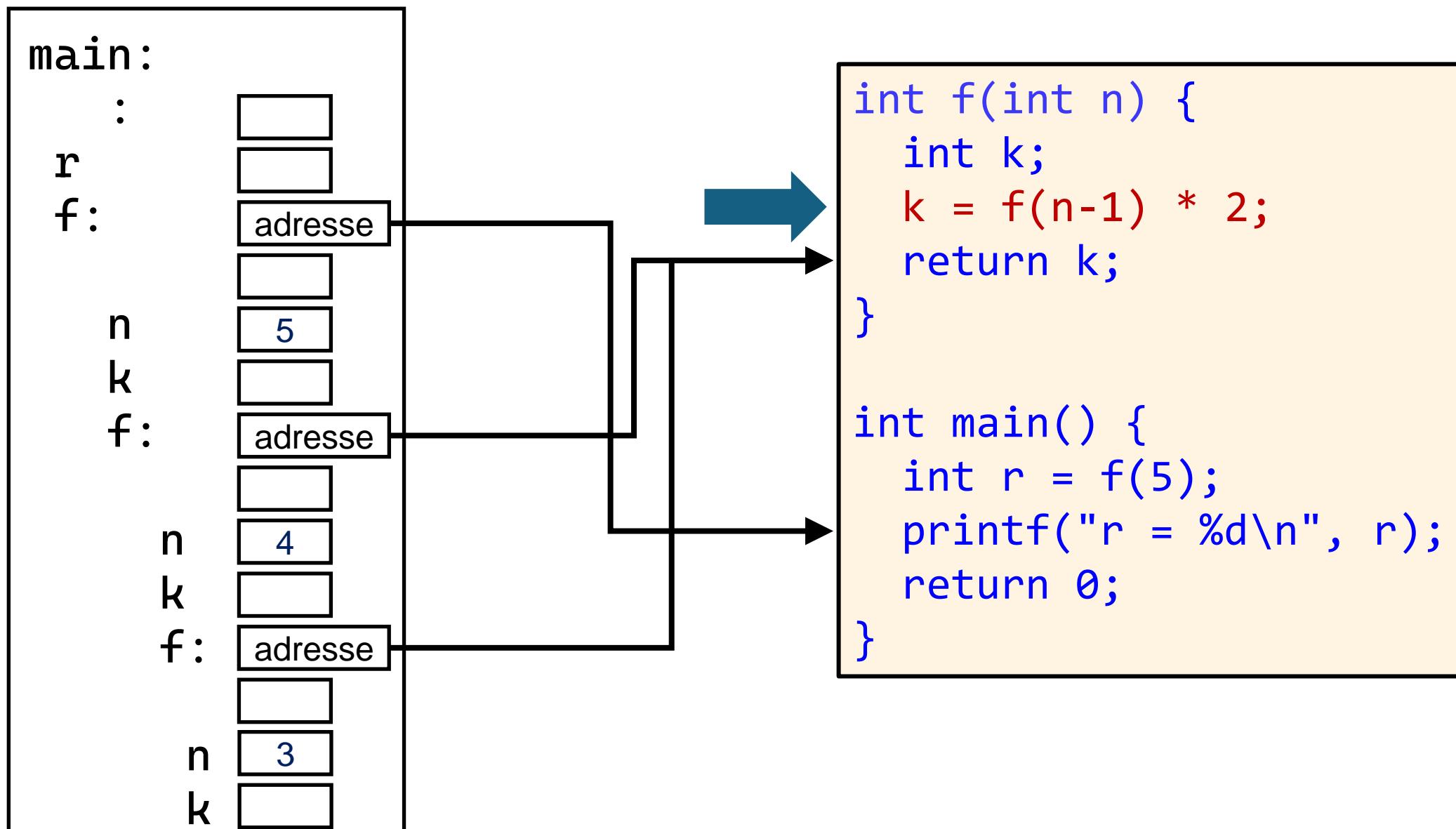
Fonctions récursives

Stack



Fonctions récursives

Stack



Fonctions récursives

A chaque étape, les copies de la fonction f sont mises pause en attendant qu'une nouvelle copie de f s'exécute. Et cela ajoute de nouvelles variables dans la stack (pile).

L'erreur d'exécution se produit quand la stack est pleine. C'est toujours le cas ici, puisque l'algorithme n'a pas de limites sur le nombre de copies de f créées.

```
int f(int n) {
    int k = f(n-1) * 2;
    return k;
}

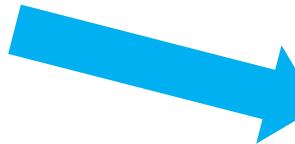
int main() {
    int r = f(5);
    printf("r = %d\n", r);
    return 0;
}
```

Fonctions récursives

Il faut donc toujours mettre une limite à la récursivité (le nombre de fonctions f appelées), par exemple :

```
int f(int n) {  
    int k = f(n-1) * 2;  
    return k;  
}
```

```
int main() {  
    int r = f(5);  
    printf("r = %d\n", r);  
    return 0;  
}
```



```
int f(int n) {  
    int k;  
    if (n > 0) k = f(n-1) * 2;  
    else k = 1;  
    return k;  
}
```

```
int main() {  
    int r = f(5);  
    printf("r = %d\n", r);  
    return 0;  
}
```

Ici la limite, c'est la valeur de n telle que $f(n)$ est explicitement connue ($f(0) = 2^0 = 1$).

Compiler ce code et l'exécuter.

Fonctions récursives (2)

2^{ème} exemple (très connu) : calcul de la suite de Fibonacci f_k ($k=0, \dots$) où

$$f_k = f_{k-1} + f_{k-2} \quad (k \geq 2)$$

$$f_0 = 0$$

$$f_1 = 1$$

La définition de la suite est récursive, on peut essayer de la calculer avec une fonction C récursive.

Examiner le code dans le répertoire `fonctions_recursive/fibonacci`

Compiler le code par `cmake`

```
cmake -S src -B build:
```

```
make -C build
```

et exécuter le code en tapant:

```
./build/fibonacci 40
```

Le temps de calcul des valeurs de $f(n)$ pour $n > 40$ est de plus en plus important. Expliquer pourquoi.

Fonctions récursives (3)

Le code source contient deux autres algorithmes de calcul de la suite de Fibonacci:

- Un algorithme itératif (non récursif)
- Un algorithme récursif mais qui enregistre les valeurs déjà calculées pour ne pas recalculer plusieurs fois le même élément (programmation dynamique)

Pour utiliser ces algorithmes, tapez l'une des 2 commandes :

./build/fibonacci 40 iteratif

./build/fibonacci 40 dynamique

Vérifier que les résultats sont les mêmes avec les 3 algorithmes.

Fonctions récursives (4)

Le temps de calcul des valeurs de $f(n)$ pour $n > 40$ est de plus en plus important. Expliquer pourquoi.

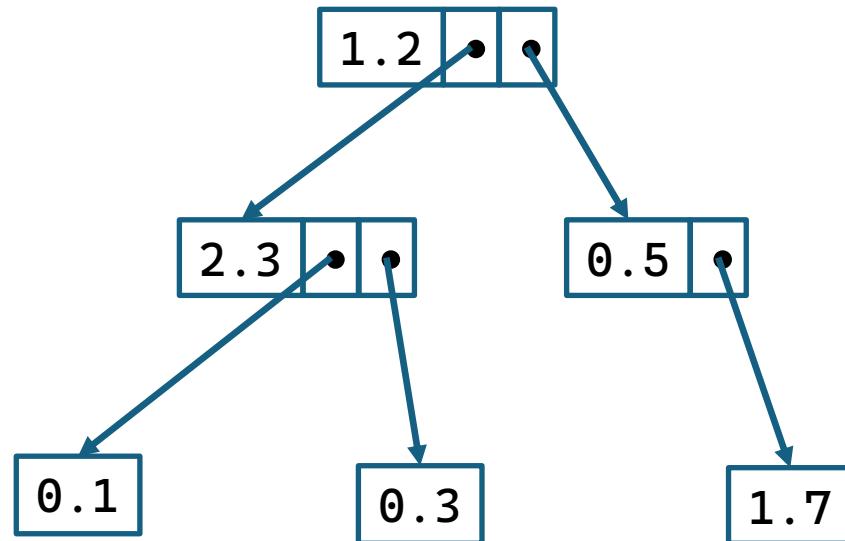
Pour certains algorithmes récursifs, il existe un algorithme non récursif équivalent, qui est souvent plus efficace (plus rapide et utilise moins de mémoire).

Dans d'autres cas, l'algorithme récursif est le seul disponible.

Structure d'arbre

Un arbre est une structure (informatique, plus générale qu'une liste) composée d'un ensemble de nœuds (avec un nœud de départ: nœud racine, des nœuds terminaux : nœuds feuilles et des nœuds intermédiaires). Les nœuds sont reliés par des connexions (sans chemin circulaire)

Exemple (chaque nœud contient un réel) :



Type nœud :

```
typedef struct _noeud
{
    double valeur;
    int nEnfants;
    struct _noeud ** enfants;
} noeud;
```

Type arbre :

```
typedef noeud * arbre;
```

Parcours récursif d'arbre

Parcourir un arbre, c'est accéder au contenu de tous les nœuds. A ma connaissance, les seuls algorithmes qui existent sont des algorithmes récursifs.

Concevoir à titre d'exercice une fonction qui calcule la valeur maximale dans les nœuds.

Pointeurs de fonction

Pointeurs de fonctions

En C, on peut définir et utiliser des pointeurs vers le code (binaire) d'une fonction.

Par exemple, si *f* est une fonction à un paramètre réel qui a un résultat réel:

```
double f(double x) {  
    return x*2.0;  
}
```

Un pointeur *pf* sur *f* est défini et s'utilise comme suit:

```
double (*pf)(double);  
pf = &f; /* ou pf = f */
```

```
x = pf(2.3) + pf(1.2);
```

Pointeurs de fonctions

Faire attention à la syntaxe (place du caractère * et des parenthèses) !!

Quelles sont les significations de 3 lignes ci-dessous ?

```
double pf(double);
```

```
double * pf(double);
```

```
double (*pf)(double);
```

Pointeurs de fonctions (2)

Exemple 1 : évaluation d'une fonction au choix (voir le répertoire pointeur_fonction/exemple1)

```
double (*p) (double);
switch (k) {
    case 1:
        p = &sin;
        break;
    case 2:
        p = &cos;
        break;
    case 3:
        p = &fois2;
        break;
}
x = p(3.14159);
```

Suivant la valeur de l'entier k, p(3.14...) évalue soit une fonction mathématique système sin ou cos, soit une fonction codée par l'utilisateur

Pointeurs de fonctions (3)

Exemple 2 : dérivée approchée

```
double derivee(double (*f) (double),
               double x, double dx)
{
    return (f(x+dx) - f(x-dx))/(2*dx);
}
```

Que l'on peut utiliser comme :

```
x = 1.0;
h = 0.001;
printf("x = %10.5g\n", x);
printf(" sin(x) = %12.7g\n", sin(x));
printf(" dsin/dx(x) = %12.7g\n", derivee(sin, x, h));
```

Pointeurs de fonctions (3)

Examiner le code source dans pointeur_fonction/exemple2 et compiler en tapant :

```
gcc main.c derivee.c -o ex2 -lm
```

Ne pas oublier -lm qui ajoute au code les fonctions mathématiques standards (sin, cos, ...)

Une variante de cet exemple se trouve dans pointeur_fonction/exemple3 où des pointeurs de fonctions sont mis dans une variable de type struct (illustration d'un fonctionnement pré-C++)

Librairies statiques et dynamiques

Librairies

En C (et dans d'autres langages de programmation), les librairies sont un moyen de partager/réutiliser des fonctions compilées entre plusieurs codes.

Un exemple est la librairie mathématique C (contenant sin, cos, tan, exp, ...) qui est utile à beaucoup de codes développés par des programmeurs. Cette librairie est fournie par tous les compilateurs.

Mais un programmeur peut aussi compiler du code source sous forme de librairie s'il estime que ce code source est utile dans plusieurs développements.

Les librairies sont du code binaire incomplet et contiennent:

- des fonctions C (mais pas la fonction principale main)
- des variables globales

Librairies statiques et dynamiques

Il existe deux sortes de librairies : statiques et dynamiques

Les librairies statiques se terminent par .a (linux, macOs), .lib (windows), les librairies dynamiques par .so (linux), .dynlib (macOs), .dll (windows)

- Les librairies statiques sont intégrées dans le code binaire final (les compilateurs font le tri des fonctions et ne gardent que les fonctions réellement utilisées).
- Les librairies dynamiques restent à l'extérieur de code (à la compilation le compilateur vérifie que tout ce qui est nécessaire se trouve dans le code source ou dans les libraires). Chaque fois que l'exécutable est utilisé, la machine charge les librairies dynamiques.

Avantage/inconvénients des librairies statiques

Avantages des librairies statiques :

- le code final contient tout ce qu'il faut pour l'exécuter,
- le code a les mêmes performances que le code qui n'utilise pas de librairies.

Inconvénients des librairies statiques :

- le code est plus volumineux,
- si plusieurs codes utilisent la même librairie, le binaire de la librairie existe en plusieurs copies sur les disques (dans chaque code).

Avantage/inconvénients des librairies dynamiques

Avantages des librairies dynamiques :

- le binaire commun entre plusieurs codes existe dans un seul fichier (xxx.so dans linux, xxx.dll dans windows) et le code exécutable est plus petit,
- Si on fait des corrections dans le code source d'une librairie dynamique (**sans changer le nom des fonctions ou leurs paramètres**), il n'est pas nécessaire de recompiler le code complet.

Inconvénients des librairies dynamiques :

- le code binaire est (un petit peu) moins rapide (parce qu'on passe toujours par des pointeurs de fonction),
- si on diffuse le code, il faut que les librairies dynamiques soient disponibles sur toutes les machines où le code est utilisé,

Exemple d'utilisation des librairies

Dans le répertoire librairies/exemple1 se trouve un exemple de construction

- d'une librairie en version statique contenant des fonctions de manipulation de vecteurs et matrices
- d'une librairie en version dynamique contenant les mêmes fonctions
- d'un code binaire exécutable qui utilise la librairie statique
- d'un code binaire exécutable qui utilise la librairie dynamique

Compilation de la librairie statique et du code qui l'utilise

Tapez les commandes pour compiler la librairie statique alglin.a:

```
gcc -I src/alglin -c src/alglin/matrice.c  
gcc -I src/alglin -c src/alglin/vecteur.c  
gcc -I src/alglin -c src/alglin/prodmat.c  
ar r alglin.a vecteur.o matrice.o
```

Puis les commandes qui créent l'exécutable ex1_static:

```
gcc -I src/alglin -c src/main.c  
gcc main.o alglin.a -o ex1_static
```

Tester en exécutant le code:

```
./ex1_static
```

Compilation de la librairie dynamique et du code qui l'utilise

Tapez les commandes pour compiler la librairie dynamique alglin.so:

```
gcc -I src/alglin -c src/alglin/matrice.c  
gcc -I src/alglin -c src/alglin/vecteur.c  
gcc -I src/alglin -c src/alglin/prodmat.c  
gcc -shared vecteur.o matrice.o prodmat.o -o alglin.so
```

(remarquer que les 3 premières commandes sont identiques au cas statique)

Puis les commandes qui créent l'exécutable ex1_statique:

```
gcc -I src/alglin -c src/main.c  
gcc main.o alglin.so -o ex1_dynamic
```

Tester en exécutant le code:

```
./ex1_dynamic
```

Le code ne devrait pas s'exécuter; voir page suivante)

Compilation de la librairie dynamique et du code qui l'utilise (2)

Si ex1_dynamique ne s'exécute pas, taper la commande:

```
ldd ex1_dynamic
```

Cette commande liste les librairies dynamiques utilisées par ex1_dynamique:

```
linux-vdso.so.1 (0x00007fffa21fb000)
alglin.so => not found
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fbedce69000)
/lib64/ld-linux-x86-64.so.2 (0x00007fbedd096000)
```

Le code ne trouve pas la librairie alglin.so. Pour le lui indiquer, il faut définir la variable d'environnement LD_LIBRARY_PATH:

```
export LD_LIBRARY_PATH=.
```

(à la place de ., on peut mettre le chemin complet vers le fichier alglin.so)

Utilisation de cmake

Dans les répertoires src et src/alglin se trouvent des fichier CMakeLists.txt que cmake utilise pour compiler les librairies et codes. Examinez ces fichiers.

Compiler en tapant les commandes

```
cmake -S src -B build -DCMAKE_INSTALL_PREFIX=install  
make -C build install
```

Qui créent le répertoire install avec les binaires.

Exécuter les 2 versions en tapant:

```
./install/ex1_static  
./install/ex1_dynamic
```

Pour pouvoir exécuter la version dynamique, il faut d'abord modifier la variable LD_LIBRARY_PATH:

```
export LD_LIBRARY_PATH=./install/lib
```

Taille des binaires

Afficher la taille des fichiers binaires, en tapant

```
ls -lR install
```

Et comparer les versions statique et dynamique.