



# Paradigmes de programmation : programmation impérative, Langage C

Marie-Laure Nivet

[nivet\\_m@univ-corse.fr](mailto:nivet_m@univ-corse.fr),

# Objectifs de ce cours

- Maitriser un vocabulaire théorique.
- Voir ou revoir les étapes du processus de compilation.
- Maitriser des concepts : pointeurs, gestion de la mémoire, structure de données, etc.
- Apprendre le langage fondateur de la programmation impérative : le C.

research



DOI:10.1145/3584859

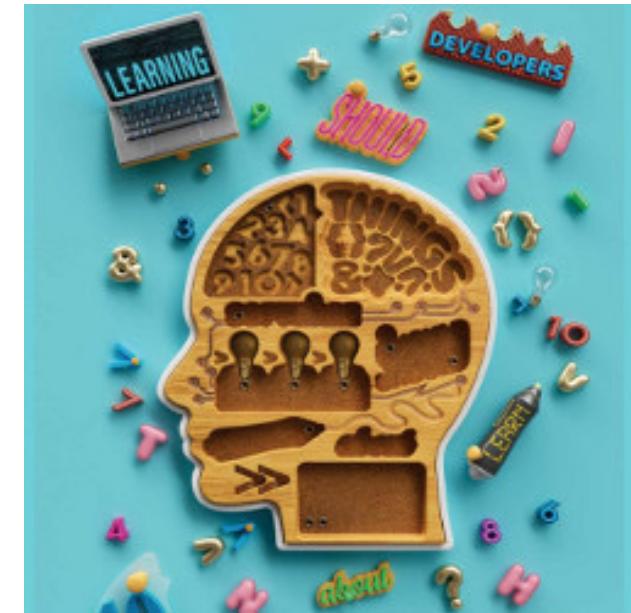
**Understanding how human memory and learning works, the differences between beginners and experts, and practical steps developers can take to improve their learning, training, and recruitment.**

BY NEIL C.C. BROWN, FELIENNE F.J. HERMANS,  
AND LAUREN E. MARGULIEUX

# 10 Things Software Developers Should Learn about Learning

LEARNING IS NECESSARY for software developers. Change is perpetual: New technologies are frequently invented, and old technologies are repeatedly updated. Thus, developers do not learn to program just once—over the course of their careers they will learn many new programming languages and frameworks.

Just because we learn does not mean we understand how we learn. One survey in the U.S. found that the majority of beliefs about memory were contrary to those of scientific consensus: People do not intuitively understand how memory and learning work.<sup>37</sup>



Qu'en avez-vous pensé ?  
Avez-vous appris des choses ?  
Enquête sur l'ENT...

# Points abordés

- Théorie
  - Méthodes de programmation et « bonnes pratiques »
  - Types et abstraction, types simples, composés, pointeurs
  - Structures de données, séquentielles, hiérarchiques et relationnelles
  - Récursivité
    - Les différents types de récursivité, récurrence faible, forte, terminale
- Pratique, Implémentations
  - Structure d'un programme
  - Types primitifs, structures de contrôle
  - Aléatoire
  - Pointeurs, gestion de la mémoire, structure de données non natives
    - Structures séquentielles : Liste, Pile, File, Tableau associatif
    - Structures hiérarchiques : Arbre
    - Structures hiérarchiques et relationnelles : Graphes
  - Structuration du code, outils type make
  - Entrées/sorties, fichiers

De la théorie à la pratique...

De la théorie et de la pratique...

Eminemment sujet à variation !



Rappelez-moi  
pourquoi le C ?

# Caractéristiques de C

- Peu de mots-clés
- Contraintes syntaxiques faibles (cast possible)
- Compilateur permissif (utiliser l'option -Wall pour afficher tous les warnings)
- Plutôt proche de la machine
- Efficace et adapté à la programmation système
- Pas de contrôle à l'exécution (pas de gestion de la mémoire, pas d'exception)
- Le comportement des applications dépend souvent du système sous-jacent

Attention devenir un expert en C est plus difficile (demande des connaissances sur le système) que dans d'autres langages

# Vous vous apprêtez à apprendre (revoir ?) un [nouveau] langage...

Quels sont les points qui vous paraissent importants ? Carte d'identité ?

Les bons réflexes lors d'un tel apprentissage ?

Quelles sont les choses que vous avez « toujours » voulu savoir sur le C ?

# Vos réponses en 2024 ?

# Des questions à se poser

- Impératif ? Déclaratif ? Structuré ?
- Interprété ? Compilé ?
- Typage statique ? Dynamique ?
- Quels sont les types primitifs (pourquoi est-ce intéressant de le savoir ?) ? Peut-on créer des nouveaux types ?
- Peut-on / doit-on gérer la mémoire ? Comment ?
- Comment s'effectue le passage de paramètres dans les fonctions
- Y-a-t-il une notion d'immutabilité ?
- Quelles sont les « bonnes pratiques » ? Règles de nommage, structuration, indentation, organisation des fichiers, du code, de la documentation ?
- Cheat-sheet ?

# Installation d'un environnement de développement C ? C'est fait ?

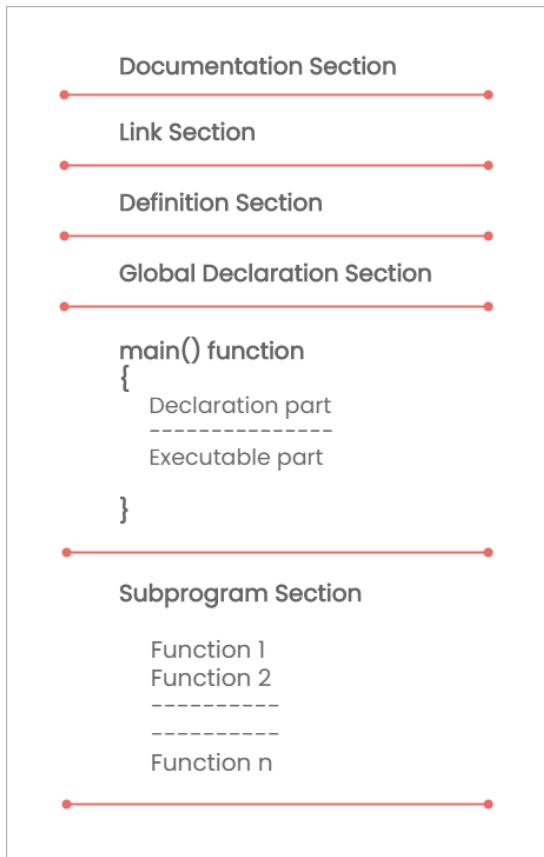
- Compilateur GCC, C Norme Iso
- Editeur
- Si vous êtes sous Windows vous pouvez choisir MinGW-w64

# Structure d'un programme C

Structure à plat

- Ensemble de
    - fonctions déclarées et implémentées
    - variables définies et initialisées
- pouvant être réparties dans plusieurs fichiers
- **Une** fonction main, point d'entrée du programme, première fonction qui s'exécute
  - Pas de structuration du code, les fonctions sont visibles entre elles
  - Les variables peuvent être
    - Globales, visibles par n'importe quelle fonction, déclarées dans le fichier .c
    - Locales, déclarées et initialisées dans une fonction

# Structure d'un programme C



```
/*
 * Documentation section
 * @Project Geometrie en L3
 * @Author Marie-Laure Nivet
 * @Created 20221010
 * @Program Description, Program to compute the area of a circle
 */

//Link section
#include <stdio.h>

//Definition section
#define PI 3.14159
#define carre(r) r*r
#define surface(r) (PI*carre(r))

//Global declaration section

//Function prototype declaration section
void message();

//Main function
int main(int argc, char const *argv[])
{
    /* Declaration part */
    float r;
    float area;
    /* Executable part */
    printf("Entrez le rayon \n");
    scanf("%f",&r);
    /* Computation part */
    area = surface(r);
    printf("La surface du cercle de rayon %f est %f\n",r,area);
    /* Sub functions calling */
    message();
    return 0;
}

//Sub functions
void message(){
    printf("Ceci est une sous-fonction \n");
    printf("Il peut y en avoir plusieurs \n");
}
```

structureProgrammeC.c

# Compiler un programme C

- Compilation simple

```
$gcc structureProgrammeC.c  
$./structureProgrammeC
```

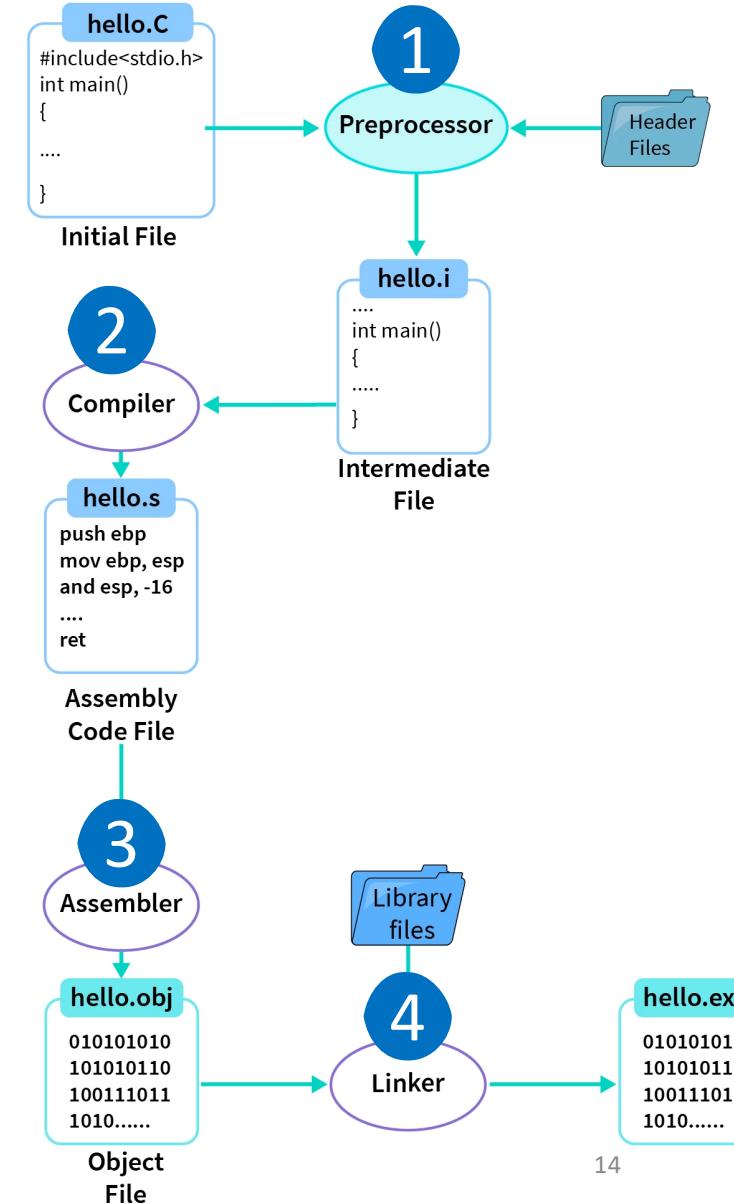
On obtient un exécutable a.out (Windows) ou structureProgrammeC (Linux/MacOS)

- En spécifiant le fichier de sortie

```
$gcc structureProgrammeC.c -o monexecutabledestructureProgramme  
$./monexecutabledestructureProgramme
```

# Les 4 étapes de la compilation C

- Compilation : opération de traduction du code source – langage de haut niveau – en code machine prêt à être exécuté
- Les étapes :
  1. Appel du pré-processeur :  
prise en compte de toutes les directives de pré-processing (#)  
- include  
- define  
et suppression des commentaires
  2. Compilation : traduction par le compilateur du langage source en langage d'assemblage correspondant au jeu d'instructions du processeur
  3. Assemblage : obtention à partir du code en assembleur, du programme binaire équivalent, c'est le module objet.
  4. Edition de liens : obtenir le programme binaire (code machine) final à partir de tous les modules objets et libraires compilés et assemblés séparément.



# Pré-processeur

- Modification du texte du fichier source
  - Commentaires ôtés
- Les directives commençant par # lui sont adressées
- Inclusion d'autres fichiers sources : #include
  - Recopie le contenu d'un fichier dans le fichier courant
  - Si #include <...> recherche dans l'include paths standard
  - Si #include "..." recherche dans le répertoire courant
- Compilation conditionnelle
  - #if, #ifdef, #ifndef, #else, #elif #endif

# Pré-processeur

- Définition de macros (fonction) ou de symboles (constantes) :

```
#define
```

```
#define cube (r) r*r*r  
#define somme (a,b) a+b  
#define volume (r) (4*PI*cube(r))/3
```

```
#define PI 3.14159
```

- Remplacement du nom de la macros (ex : cube (r) ) ou du symbole (PI) par le code ( $r * r * r$ ) ou le littéral (3.14) correspondant dans le corps du code avant compilation

Attention : la valeur assignée via un `define` peut être écrasée et modifiée par un `define` ultérieur.

# Exemple après pre-processeur

```
//Commentaires pour le lecteur du code  
  
int main(int argc, char const *argv[])  
{  
    int variable = 0;  
    return 0;  
}  
simplemainforcompilerexplained.c
```

```
//Commentaires pour le lecteur du code  
#define VALEUR 50  
int main(int argc, char const *argv[])  
{  
    int variable = VALEUR;  
    return 0;  
}  
simplemainwithmacrosforcompilerexplained.c
```

Arrêt après le préprocesseur

```
$ gcc -E simplemainforcompilerexplained.c  
$ cat simpleaprespreprocess.e  
# 1 "simplemainforcompilerexplained.c"  
# 1 "<built-in>" 1  
# 1 "<built-in>" 3  
# 400 "<built-in>" 3  
# 1 "<command line>" 1  
# 1 "<built-in>" 2  
# 1 "simplemainforcompilerexplained.c" 2
```

Spécification du fichier de sortie

```
$ gcc -E simplemainforcompilerexplained.c -o simpleaprespreprocess.e
```

Arrêt de la compilation après le préprocesseur  
gcc -E fichier.c

```
int main(int argc, char const *argv[])  
{  
    int variable = 0;  
    return 0;  
}
```

```
$ gcc -E simplemainwithmacrosforcompilerexplained.c -o simpleaprespreprocess.e  
(base) udc-1-1:simpleC nivet_m$ cat simpleaprespreprocess.e  
# 1 "simplemainwithmacrosforcompilerexplained.c"  
# 1 "<built-in>" 1  
# 1 "<built-in>" 3  
# 400 "<built-in>" 3  
# 1 "<command line>" 1  
# 1 "<built-in>" 2  
# 1 "simplemainwithmacrosforcompilerexplained.c" 2
```

```
int main(int argc, char const *argv[])  
{  
    int variable = 50;  
    return 0;  
}
```

Faites le sur votre propre machine avec votre compilateur, en ligne de commande

# Exemple après pre-processeur

```
//Commentaires pour le lecteur du code
#include <stdio.h>

int main(int argc, char const *argv[])
{
    int variable = 0;
    printf("La valeur de la variable est %d\n",variable);
    return 0;
}
```

simplemainwithincludeforcompilerexplained.c

Regardez ce que fait le pré-processeur dans le cas des include...

Arrêt après le preprocesseur

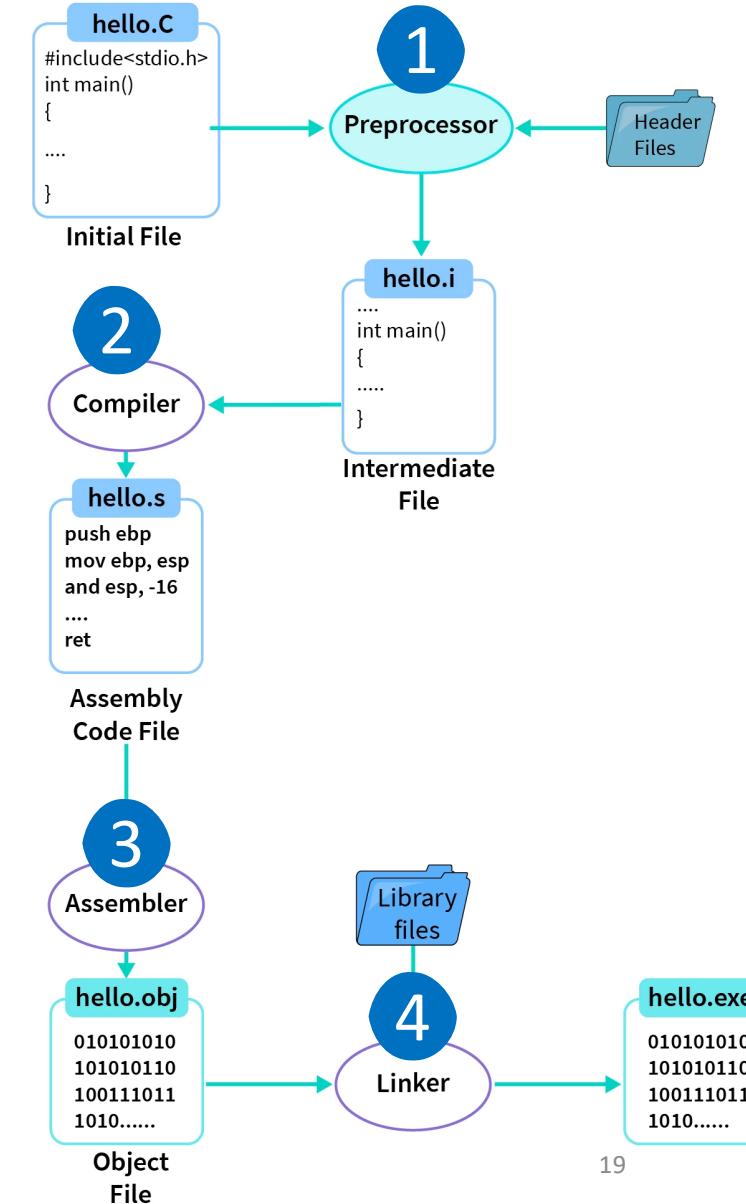
```
$ gcc -E simplemainwithincludeforcompilerexplained.c -o simpleaprespreprocess.e
```

Spécification du fichier de sortie

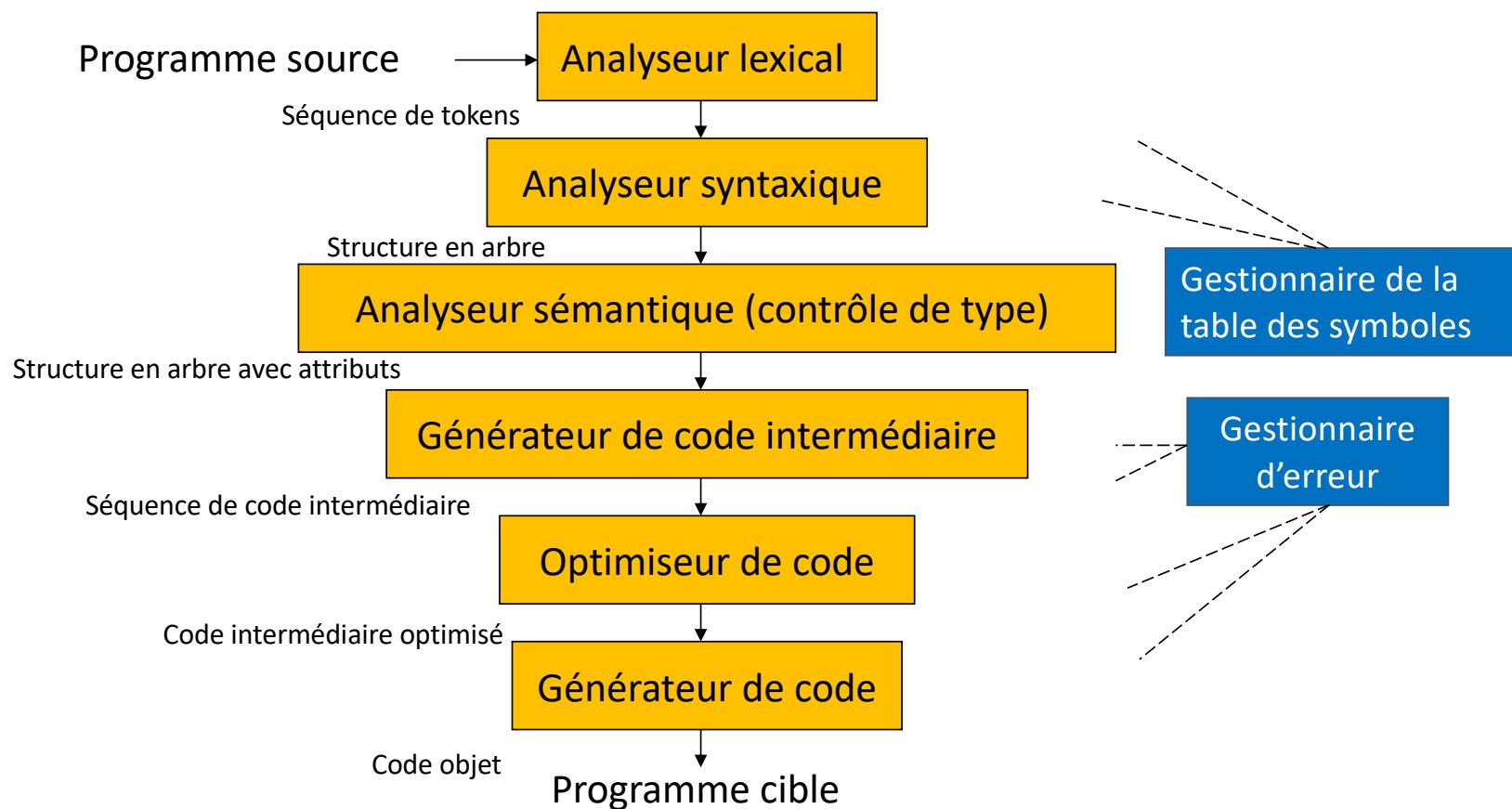
Faites le sur votre propre machine avec votre compilateur, en ligne de commande

# Les 4 étapes de la compilation C

- Compilation : opération de traduction du code source – langage de haut niveau – en code machine prêt à être exécuté
- Les étapes :
  1. Appel du pré-processeur :  
prise en compte de toutes les directives de pré-processing (#)  
- include  
- define  
et suppression des commentaires
  2. Compilation : traduction par le compilateur du langage source en langage d'assemblage correspondant au jeu d'instructions du processeur
  3. Assemblage : obtention à partir du code en assembleur, du programme binaire équivalent, c'est le module objet.
  4. Edition de liens : obtenir le programme binaire (code machine) final à partir de tous les modules objets et libraires compilés et assemblés séparément.

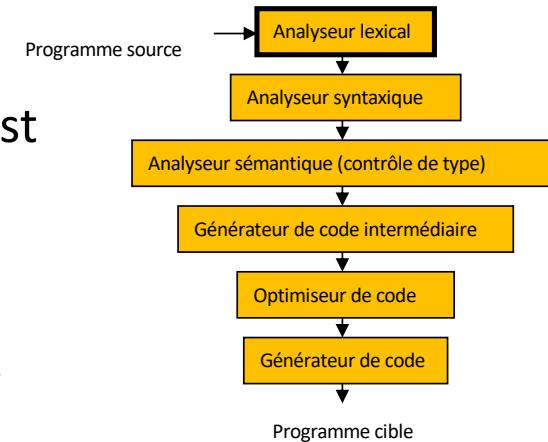


# Comprendre et voir les différentes étapes de la compilation



# Analyseur lexical (scanner)

- Transforme une suite de caractères en une suite d'entités de plus haut niveau
  - Lus de gauche à droite
  - Groupés en unités lexicales lexèmes (*token* : suite de caractères ayant une signification collective)
  - Caractères superflus supprimés
- But de l'analyse
  - Générer des unités lexicales
  - Déterminer si chaque unité lexicale est un mot du vocabulaire
- Résultat
  - Erreur générée si mot non autorisé



# Exemple : analyse lexicale

```
result := somme + test/20
```

Identificateur result

Symbole d' affectation

Identificateur somme

Symbole d'addition

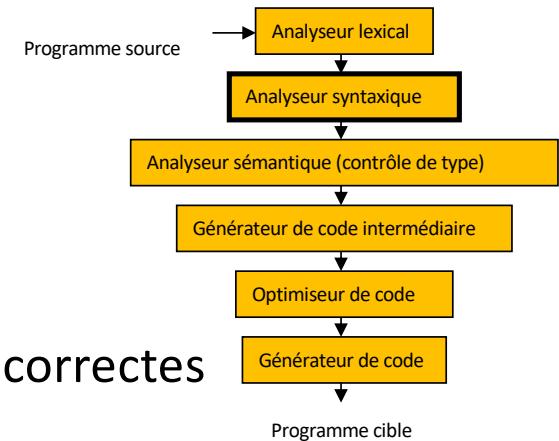
Identificateur test

Symbole de division

Nombre 20

# Analyseur syntaxique (parser)

- But de l'analyse
  - Regrouper les unités lexicales en structures grammaticales correctes
  - Déterminer si la syntaxe (grammaire) est correcte
- La grammaire d'un langage est définie par un ensemble de règles (récursives ou non)
- Résultat
  - Construction à partir des tokens d'un arbre syntaxique
  - Erreur générée si structure non valide c'est à dire ne respecte pas la grammaire

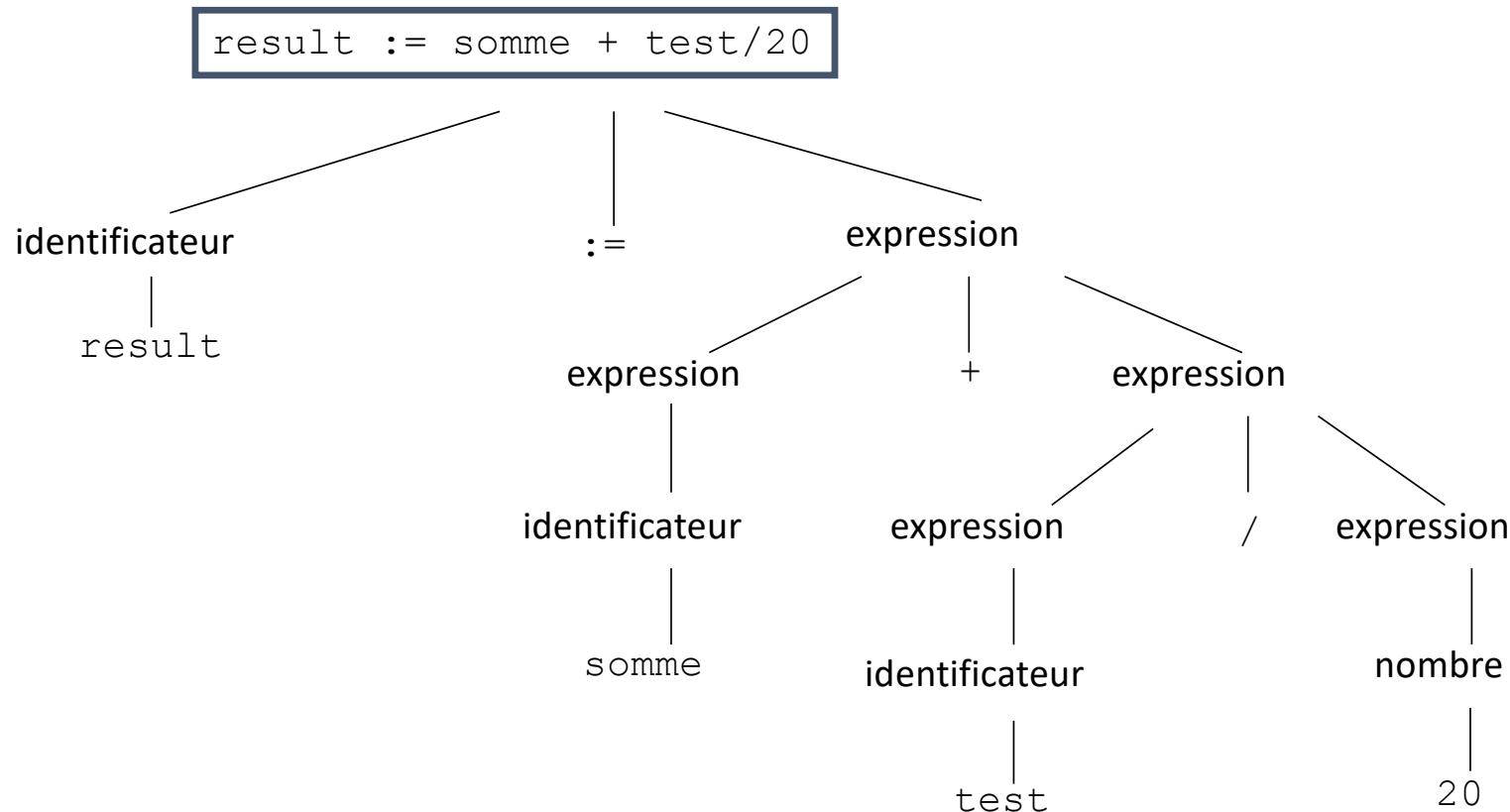


# Analyseur syntaxique (parser)

## Comment ?

- Construction d'arbres d'analyse
- Deux types d'analyses
  - Descendante (racine vers feuilles) ou descendante récursive ou prédictive
  - Ascendante (feuilles vers racine)

# Exemple : analyse syntaxique, Arbre d'analyse ou de dérivation

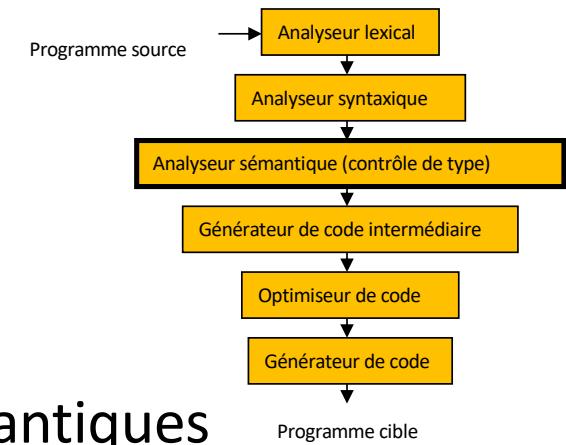


# Table des symboles

- Structure de données
- Stockage d'informations concernant les identificateurs du programme source
  - Type, emplacement mémoire, portée, visibilité, nombre et types des paramètres d'une fonction
- Le remplissage de cette table a lieu lors des phases d'analyse

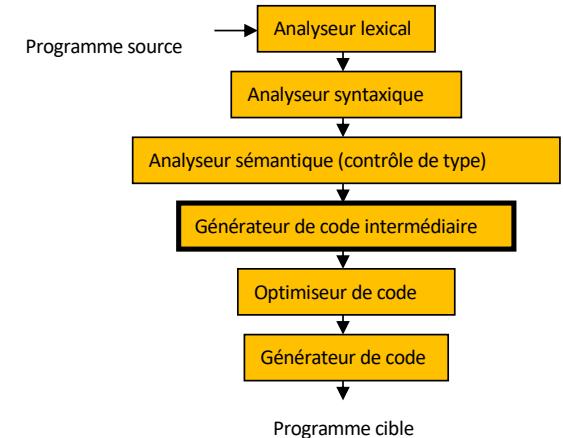
# Analyseur sémantique

- Déterminer si le programme contient des erreurs sémantiques statiques
  - Contrôle de type des identificateurs
  - Portée des identificateurs
  - Unicité des identificateurs
  - Flot d'exécution
    - exemple : utilisation d'un break alors qu'il n'y a pas de boucle englobante



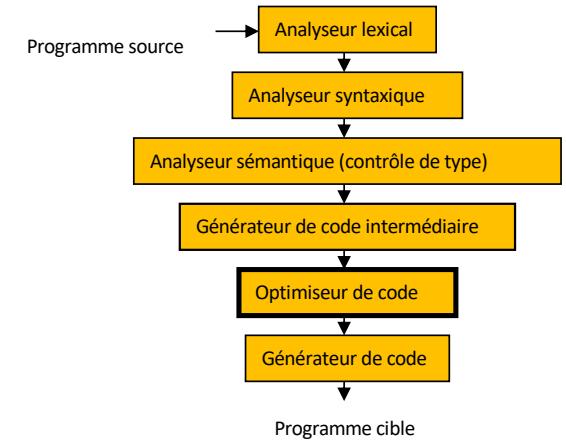
# Générateur de code intermédiaire

- Le code intermédiaire doit être
  - Indépendante de la machine cible
  - Décrire en détail les opérations à effectuer
- C'est un code exécutable d'une machine abstraite
- Exemple de code intermédiaire
  - AST : Abstract Syntax Tree
  - TAC : Three-Adress Code (Code à trois adresses)
    - Au plus un opérateur, trois opérandes, une affectation



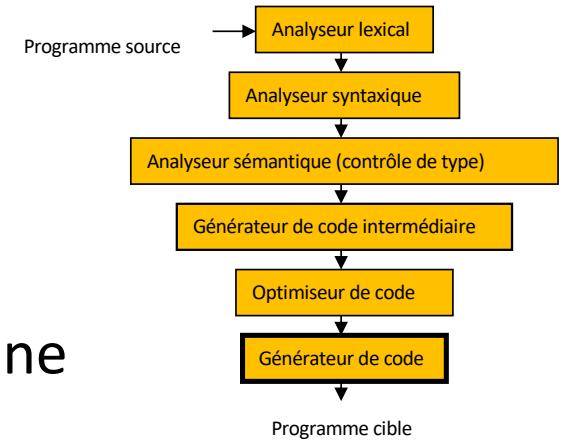
# Optimisation de code

- Amélioration du code intermédiaire suivant
  - Le temps : réduire la durée d'exécution
  - La mémoire : occupation à minimiser
- Les deux étant incompatible il faut faire des compromis...



# Production de code

- Code cible en langage assembleur ou machine
- Le code engendré peut
  - Être exécutable
  - Nécessiter une édition de liens avec des librairies
- Phase dépendant de la machine sur laquelle tournera le programme



# Exemple, assembleur généré

Arrêt de la compilation après génération de l'assembleur  
Obtenu via `gcc -S simplemain.c`

```
int main(int argc, char const *argv[])
{
    int variable;
    variable=3;
    return 0;
}
```

simplemain.c

Faites le sur votre propre machine avec votre compilateur, en ligne de commande

Instruction du processeur  
Étiquette (:) nom symbolique @ mémoire  
Directive donnée à l'assembleur (.)

```
$ gcc -S simplemain.c
$ cat simplemain.s
    .section      __TEXT,__text,regular,pure_instructions
    .build_version macos, 12, 0      sdk_version 12, 3
    .globl  _main
function main
    .p2align   2
main:                                ; @main
    .cfi_startproc
; %bb.0:
    sub    sp, sp, #32
    .cfi_def_cfa_offset 32
    mov    x8, x0
    mov    w0, #0
    str    wzr, [sp, #28]
    str    w8, [sp, #24]
    str    x1, [sp, #16]
    mov    w8, #3
    str    w8, [sp, #12]
    add    sp, sp, #32
    ret
    .cfi_endproc
; -- End function
.subsections_via_symbols
```

# Exemple

test\_Assembleur.c

```
int t[5];

int somme () {
    int s = 0;
    int i;
    for (i=0;i<10;i++)
        s = s + t[i];
    return s;
}
```

Faites le sur votre propre machine avec votre compilateur, en ligne de commande

Arrêt de la compilation après génération de l'assembleur  
Obtenu via `gcc -S test_Assembleur.c`

test\_Assembleur.s

```
.file      "test_Assembleur.c"
.text
.globl _somme
.def      _somme; .scl      2;
.type     32;       .endef

_somme:
    pushl   %ebp
    movl   %esp, %ebp
    subl   $8, %esp
    movl   $0, -4(%ebp)
    movl   $0, -8(%ebp)

L2:
    cmpl   $9, -8(%ebp)
    jg     L3
    movl   -8(%ebp), %eax
    movl   _t(%eax,4), %edx
    leal   -4(%ebp), %eax
    addl   %edx, (%eax)
    leal   -8(%ebp), %eax
    incl   (%eax)
    jmp    L2

L3:
    movl   -4(%ebp), %eax
    leave
    ret
    .comm   _t, 32      # 20
```

si  $i < 10$  (\$9) on va en L3

Jump to L2, fin de boucle

# Édition de liens

test\_Assembleur.c

```
#include <stdio.h>

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

Il faut utiliser l'éditeur de liens pour faire le liens entre le symbole `_printf` et l'adresse mémoire à laquelle on peut trouver son code

Arrêt de la compilation après génération de l'assembleur  
Obtenu via `gcc -S test_Assembleur.c`

test\_Assembleur.s

```
.file
"test_Assembleur_edition_lien.c"
.def    _main; .scl    2;
.type   32;      .endef
.section .rdata, "dr"

LC0:
.ascii "Bonjour \12\0"
.text
.globl _main
.def    _main; .scl    2;
.type   32;      .endef
_main:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    andl $-16, %esp
...
...
    sall    $4, %eax
    movl    %eax, -4(%ebp)
    movl    -4(%ebp), %eax
    call    __alloca
    call    __main
    movl    $LC0, (%esp)
    call    printf
    leave
    ret
.def    _printf; .scl    3;
.type   32;      .endef
```

# L'éditeur de lien

- L'assembleur génère le code objet (.o)
  - call \_printf
  - \_printf est un symbole utilisé mais non résolu
- L'éditeur de liens prend tous les fichiers objets (.o) et fabrique l'exécutable (binaire)
  - Il résout les références symboliques entre les fichiers

You can observe what happens by using the verbose option  
gcc **-v** test\_Assembleur.c

# Editeur de liens avec des librairies externes

- Si vous faites appel à des librairies externes

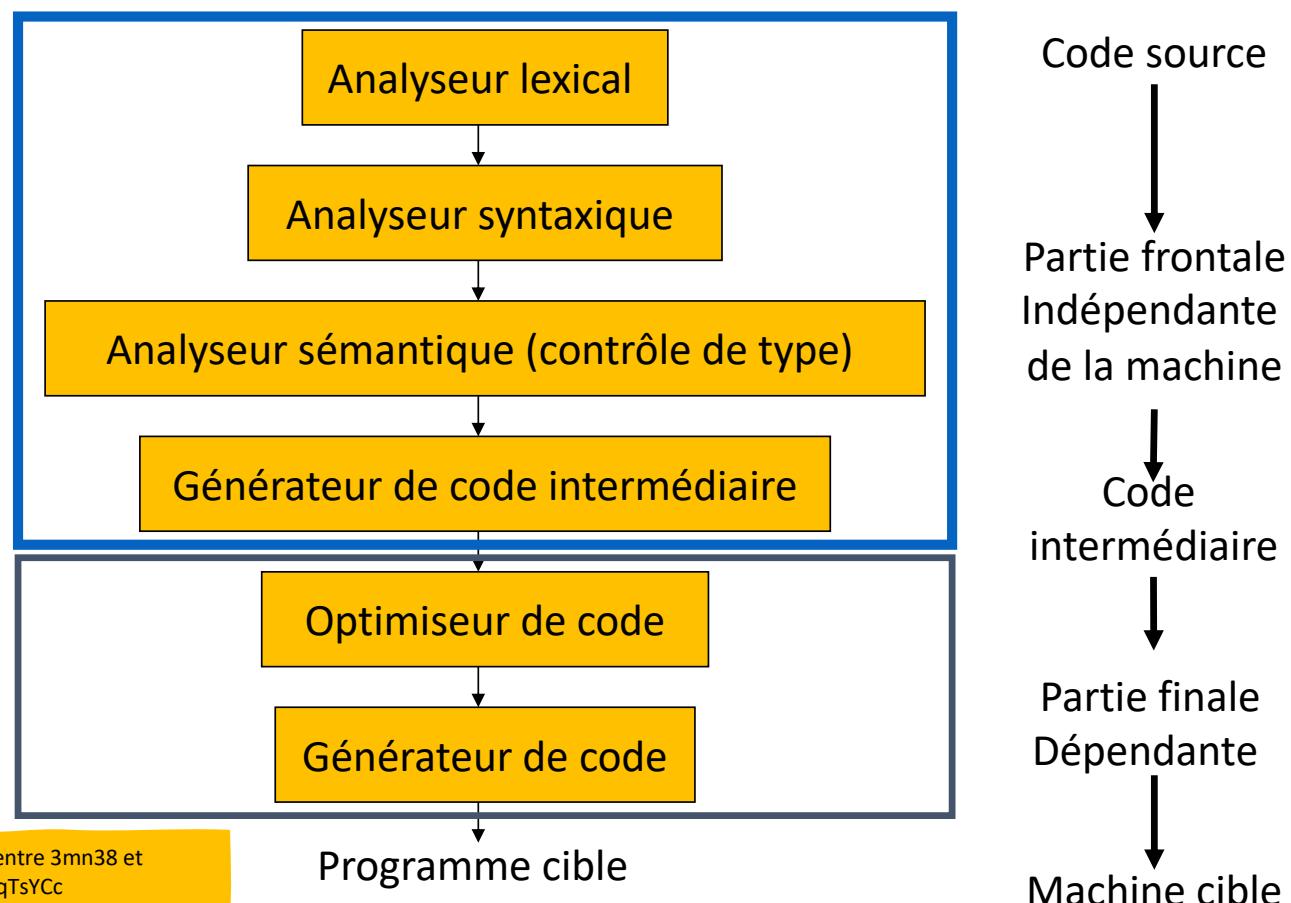
- Exemple mode console :

- ncurses (Linux/Macos/Windows Cygwin)

```
gcc -lncurses jeuModeConsole_0.c -o jeu
```

- windows et conio (Windows)

# Parties frontale (*front-end*) et finale (*back-end*)



How do computer read code ? (Frame of essence) entre 3mn38 et 9mn10 <https://www.youtube.com/watch?v=QXjU9qTsYCc>

# La compilation C

- Le programme compilé dépend du SE et du matériel (architecture processeurs)
- La compilation peut optimiser le programme et favoriser :
  - La vitesse d'exécution
  - L'empreinte mémoire du binaire, etc...

Exercice :

- Les différentes phases de la compilation
- Les directives de compilation gcc

# Pour réviser un peu...

Connaissez-vous les différentes étapes de ce qu'on appelle la compilation C, leur succession et leur rôle respectif ?

Comment s'adresse-t-on au pré-processeur ?

Qu'est-ce qu'un fichier objet (.o) ?

Qu'est-ce que l'éditeur de lien ?

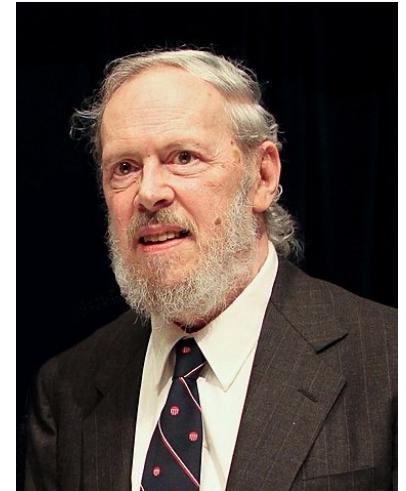
# Historique et filiation de C

Des origines jusqu'à aujourd'hui

# Dennis Ritchie (1941-2011)

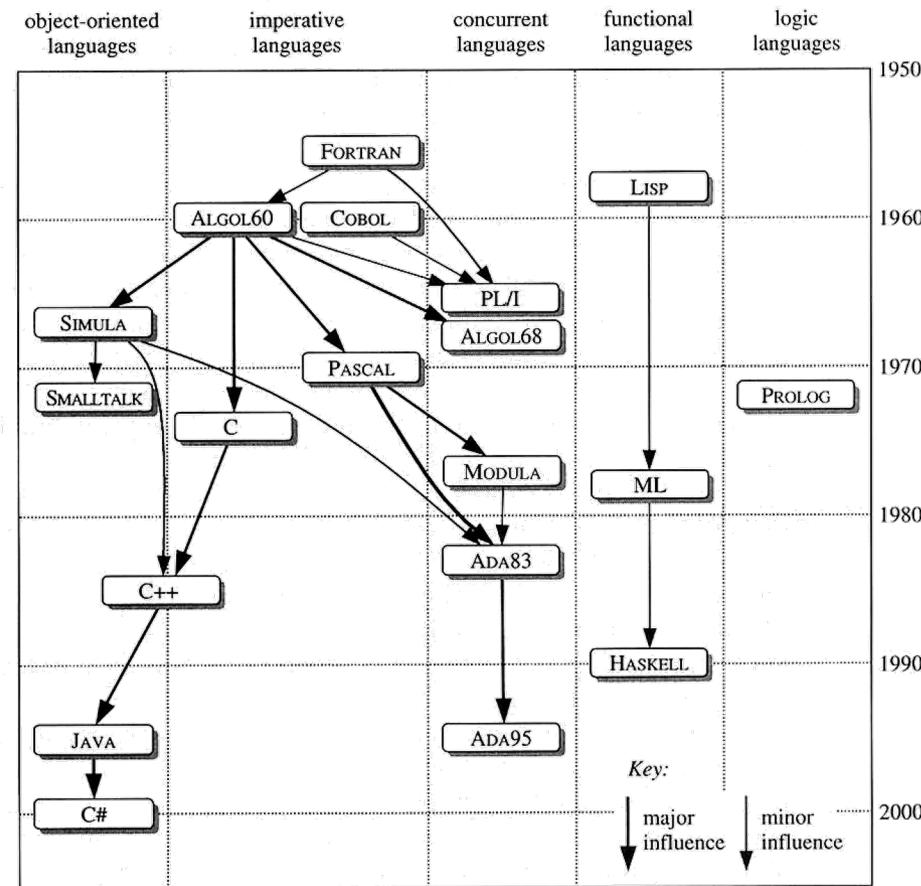
Informaticien américain, inventeur du langage C et développeur avec Ken Thompson du système d'exploitation Unix, pour lequel les deux scientifiques reçoivent en 1983 le prix Turing et d'autres récompenses prestigieuses. Pionnier de l'informatique moderne, il passe sa carrière comme employé aux Bell Labs jusqu'à sa retraite en 2007, terminant à la tête du département de recherche sur les logiciels système.

Il est l'auteur avec Brian Kernighan d'un livre référence et célèbre sur le **langage C, qu'il a inventé en 1972**, The C programming language (1978).



Au départ C avait été uniquement  
pensé pour concevoir un OS : Unix

# Les langages ayant précédé C



Pour un historique plus complet  
<http://www.levenez.com/lang>

Source :  
 « Programming Language Design Concepts »,  
 David A. Watt, Edition Wiley

Figure 1.1 Dates and ancestry of major programming languages.

# Bonnes pratiques

Discipline de programmation en C mais pas que

- Spécifications
- Code design guide lines, etc.
- Du code, plus propre, plus lisible, plus...

# Pourquoi suivre/s'imposer de bonnes pratiques ?

En premier lieu pour vous faciliter la vie !!!!!!

Pas pour : le prof/votre boss/votre collègue/je ne sais qui...

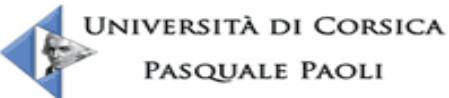


[http://imgs.xkcd.com/comics/future\\_self.png](http://imgs.xkcd.com/comics/future_self.png)

# Conseils/obligations ? (1)

- Règles de nommage : lisibilité, constance (définition d'une norme, respect d'un standard), pertinence des noms
- Structuration : aération du code, indentation
- Organisation : modularité du code (fichiers/fonctions/procédures/blocs)
  - Soyez cohérent : regroupez des choses ayant un lien dans le même fichier
  - Limitez les dépendances : si vous changez une ligne de code dans le fichier X cela ne doit pas impacter les autres fichiers
- Privilégier les solutions simples et intuitives
- Chercher les conventions et s'il y en a les respecter

Ce n'est pas parce qu'on peut le faire qu'on doit le faire !



## Curiosité :

Il existe un concours annuel « *The International Obfuscated C Code Contest* » récompensant le code C le plus obscur.

Cf : <https://www.ioccc.org/>

## Simulateur de vol gagnant en 1998

# Conseils/obligations ? (2)

- Commentaires indispensables
  - Faciliter la lecture du code
  - Indiquer des choix de conceptions/résolutions de pb
  - Indiquer les spécifications
- Programmation défensive : augmenter la robustesse du code
  - Codes d'erreur en C

# Conseils/obligations ? (3)

Cours :  
- Qualité logicielle

- DRY : Don't Repeat Yourself !
  - Encapsulez votre code dans une fonction au lieu de le copier/coller partout
- Une fonction = une opération, une fonctionnalité, évitez les usines à gaz
- Validation et tests de code
  - Différences entre résultats attendus (spécifications) et effectifs...
  - Pensez les tests en amont...
  - Test dynamiques / Tests statiques
- Maintenance
  - La vie du code **commence** à l'écriture
  - Suppression du superflu, optimisation, ré-écriture, choix d'un meilleur algorithme, révision des commentaires, etc.

Dynamiques : exécution du code  
Statiques : lecture du code

# A faire...

Proposez par binôme une liste de bonnes pratiques que vous vous **engagez** à respecter dans le cadre de ce module...

Rechercher/proposer une Cheat Sheet C

Déposez votre « manifeste » sous la forme d'un fichier PDF (nom\_prenom1\_nom\_prenom2\_manifesteC.pdf) dans la zone travaux sur l'ENT

Vous veillerez à bien citez vos sources en étayant votre texte par la citation de références précises en Français et en Anglais.

# Attribution des sujets de séminaire

1 - Les différents modes de gestion de la mémoire, garbage collector, avantages et inconvénients des différentes méthodes. Vous pouvez regarder par exemple la gestion de la mémoire dans le langage RUST :

Mehdi – Anthony – Abdou-Salam

2 - Informatique Quantique où en est-on ? Quel langage pour l'informatique quantique ?  
L'algorithme quantique :

Jacques – Frédéric – Stéphane

3 - Les assistants de codage à base d'IAS génératives, pour qui, pour quoi ? Quels avantages ? Quelles limites ?  
Quelles précautions prendre ?

Clémentine – Matthieu - Eva

4 - Quelles caractéristiques sont nécessaires pour obtenir un langage sécurisé ? Gestion de la mémoire, exceptions, typage ? Quoi d'autre encore ?

Badre – Nicolas – Dorian

5 – L'utilisation des expressions régulières en programmation, principes, usages.

Ilyas – Ahmed - Akran

6 – Utilisation des lambda expressions dans les langages impératifs – ou plus généralement des concepts de prog. Fonctionnelle – pour quoi faire et pourquoi sont-elles introduites ?

Asmae – Oumaima – Mohamed-Amine

FEATURE COMPUTING

# A CRITICAL LOOK AT AI-GENERATED SOFTWARE

Coding with ChatGPT, GitHub Copilot, and other AI tools is both irresistible and dangerous

Article à lire pour la prochaine fois...

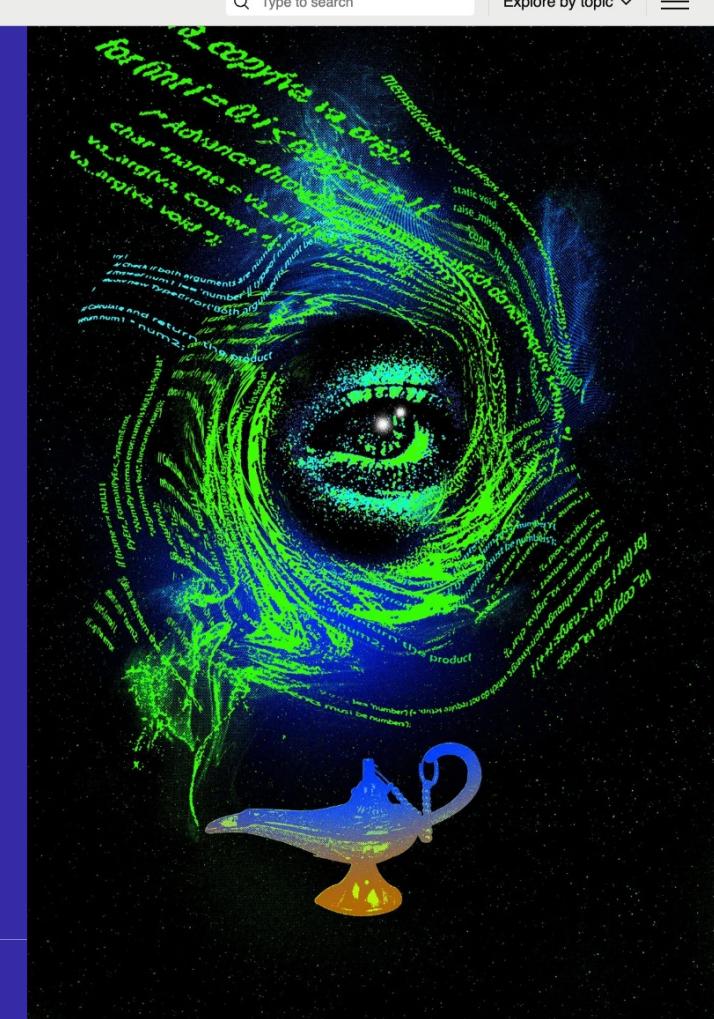
BY JAIDEEP VAIDYA HAFIZ ASIF

11 JUN 2023 | 11 MIN READ |



DANIEL ZENDER

<https://spectrum.ieee.org/ai-software>



# Spécifications algorithmiques

Savoir et **dire** ce qu'on va faire

Pourquoi on le fait

Quand on le fait et enfin

Comment on le fait !

# Spécifications algorithmiques

- Décrire le plus formellement possible le problème qui est censé être résolu par l'algorithme
  1. Les entrées qui caractérisent une instance – un exemple – du problème à résoudre
  2. Une précondition – prédicat – sur les entrées, si elles ne sont pas vérifiées le problème n'a pas lieu d'être
  3. Les sorties qui décrivent une solution au problème
  4. Une postcondition – prédicat – qui doit être vérifié après l'appel de l'algorithme. Si elle n'est pas vérifiée la solution n'est pas correcte

Ces quatre éléments définissent les **spécifications de l'algorithme** [1].

Ils doivent obligatoirement apparaître dans le code sous la forme de commentaires.

[1] V. Barra, *Informatique MP2I et MPI : CPGE 1re et 2e années - Nouveaux programmes - ScholarVox Université*, Ellipses. 2021. Consulté le: 6 septembre 2022. [En ligne]. Disponible sur: <http://univ.scholarvox.com.udcpp.idm.oclc.org/catalog/book/docid/88915779>

# Exemple de spécifications

Problème : « Factoriser dans  $\mathbb{R}$  le trinôme du second degré  $ax^2 + bx + c = 0$  et  $b^2 - 4ac \geq 0$  »

1. Les entrées sont  $a, b$  et  $c$  réels
2. La précondition est  $\{a \neq 0, b^2 - 4ac \geq 0\}$
3. La sortie est  $x_1, x_2 \in \mathbb{R}$ , tel que  $x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$
4. La postcondition est  $ax_1^2 + bx_1 + c = ax_2^2 + bx_2 + c = 0$

Ces spécifications devront être conservées dans le code et apparaître sous la forme de commentaires

# Code résultant

Vous veillerez à toujours inclure les spécifications de vos fonctions/procédures dans vos codes.  
On peut bien sûr s'aider des IA génératives

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

typedef struct
{
    double x1,x2;
}double_pair;

/*
@inputs: double a,b,c polynom parameters
@outputs: double_pair roots;
@precondition: a != 0 and b*b - 4 ac >= 0
@postcondition: ax1*x1 + bx1 + c = ax2*x2 + bx2 + c = 0
@description: compute if there exists the roots of the second order quadratic equation
@author:Nivet ML
@date: 2022-11-01
@version: v1
*/
double_pair compute_roots(double a, double b, double c){
    double_delta = b*b - 4*a*c;
    double_pair result;
    if (a!=0){
        if (delta>=0){
            double square_delta = sqrt(delta);
            result.x1 = (-b - square_delta)/(2*a);
            result.x2 = (-b + square_delta)/(2*a);
        }else{printf("Error, precondition isn't respected, 'a' must not be equal to 0");}
    }
    return result;
}
```

# Abstractions et paramétrisation

# Variables

- Qu'est-ce qu'une variable ?
  - Espace mémoire réservé et accessible permettant de stocker et utiliser des valeurs
  - Abstraction sur la mémoire

# Comment parler des variables ?

Portée :

Zone de code dans un programme  
dans laquelle il est possible de  
faire référence à la variable

Durée de vie :

Temps pendant lequel une  
variable est présente en mémoire  
vive

En C la portée est statique,  
elle dépend uniquement du  
code source à la compilation

# Nature d'une variable en C ?

- Globale
  - Portée étendue à l'ensemble du programme, durée de vie identique à celle du programme
  - Déclarée en dehors de toute fonction
- Locale
  - Portée limitée à la fonction dans laquelle elle est déclarée, durée de vie identique à celle de la fonction (créée à chaque appel, détruite à la fin de l'exécution de la fonction)
- Locale statique :
  - Sa durée de vie dépasse sa portée

```
void f (int x)
{
    static int compteur = 0;
    ++compteur;
    printf("%d-e appel de la fonction f\n", compteur);
    // Traitements de la fonction
    ...
}
```

Compte le nombre de fois où la fonction f est invoquée. Variable locale à f – inconnue en dehors de f – initialisée uniquement lors du premier appel à f – garde sa valeur d'un appel à l'autre – et détruite à la fin de l'exécution du programme.

# Transparence référentielle ?

Nous reviendrons sur ce concept en fonctionnel

- Def : Toute variable, tout appel à une fonction peut être remplacé par sa valeur sans changer le comportement du programme
- En C – en général dans le paradigme impératif – pas de transparence référentielle

```
int i = 0;

int g(int n) {
    i += n;
    return i;
}
```

Evaluez :

```
int appell1 = g(1) + g(1);
int appell2 = 2*g(1);
```

- La présence de variables globales – à proscrire le plus possible ! – rend la transparence référentielle impossible.
- Les calculs dépendent de l'état du système.

# Exercice portée

- Soit le code ci-contre donnez pour chacune des variables la portée de celle-ci (ligne X à ligne Y)

```
1 #include "stdio.h"
2 float v = 50;
3
4 float f(float x)
5 {
6     float i = 2.*x+v;
7     return i;
8 }
9 int main(void)
10 {
11     float val = 4.;
12     float res = f(val);
13     printf("%f",res);
14     return 1;
15 }
```

Variable	Lignes de portée
v	
x	
i	
val	
res	
f	

# Abstraction

Différence entre expression et commande ?

- Nommage d'expression / Définition de constantes

```
#define TAILLE (elt*10)  
...  
TAILLE ...
```

- Nommage de commande

```
void f (void) {  
    g1 = e1;  
    g2 = e2;  
}  
f();
```

- Nommage de type

```
typedef
```

# Constantes

- Dans le code avec le modificateur `const`
  - Vérification du compilateur qui assure la non-modification de la variable
- Pré-processeur `#define`
  - Dans le corps du programme, de préférence au début
  - Dans un fichier d'entête
    - Toutes les références à la constante seront remplacées par la valeur dans le code au moment de la compilation
    - Attention, le dernier `#define` gagne (Warning)
- Constantes énumération, avec `enum`

Le plus lisible et le plus sécurisé demeure l'utilisation du `const`

```
1 #include <stdio.h>
2
3 #define TEST 0;
4 #define AUTRE_CONSTANTE 5;
5 #define LANGAGE_DEV "C";
6
7 int main()
8 {
9     const float UNE_AUTRE = 5.5;
10
11     enum DIRECTION {NORD=100, SUD, EST, OUEST}; // Si on ne donne pas de valeur de départ NORD sera à 0
12     printf("Nord : %d\n", NORD); // affiche 100
13     printf("Sud : %d\n", SUD); // affiche 101
14     printf("Est : %d\n", EST); // affiche 102
15     return 0;
16 }
```

# Types et abstractions de types en C

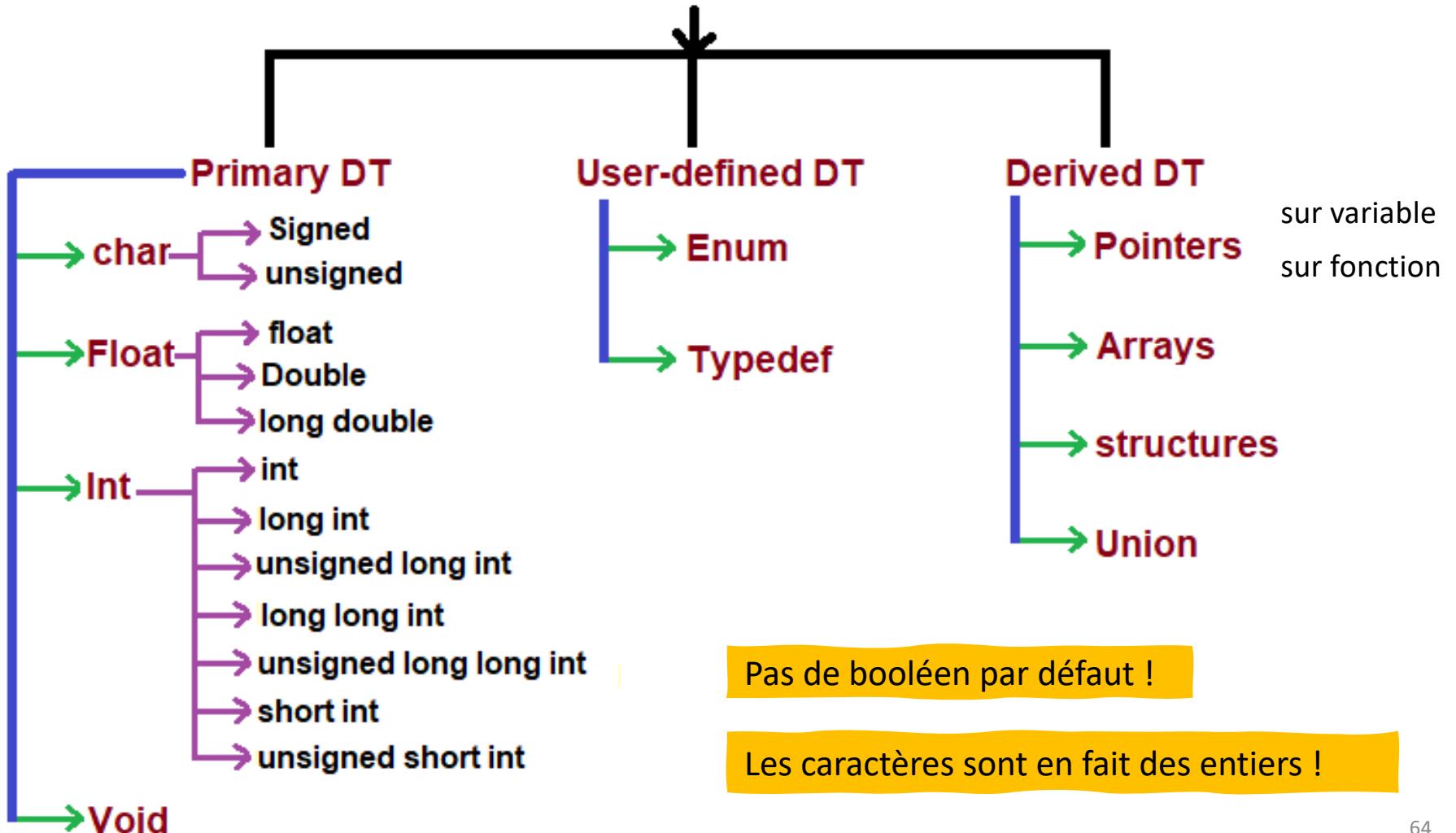
Types abstraits de données

Built-in types ? Types primitifs ? Création de nouveaux types ?

Types composés ?

## DT - Data type

# Data Types in C



# Les types simples/types primitifs (1) : domaine de valeur. Les caractères.

- Pas de véritable type caractère en C.
- Même chose que le type entier.
- char est juste un type entier codé sur 1 Octet.
- Comme tous les autres types entiers, il peut être signé ou non signé

<b>char</b>	1 byte	-128 to 127	%c
<b>signed char</b>	1 byte	-128 to 127	%c
<b>unsigned char</b>	1 byte	0 to 255	%c

```
char a,b;  
a = 'A';  
b = 'B';  
printf("%d\n", a); // 65  
printf("%c\n", a); // A  
printf("%d\n", a+b); // 131
```

# Les types simples/types primitifs (2) : domaine de valeur. Les booléens.

Introduit dans C99

- Des booléens, prenant leur valeur dans {true, false}

```
//Entête à inclure
#include <stdbool.h>
//Permet d'utiliser bool au lieu de _Bool
```

```
#define bool unsigned int
#define true 1
#define false 0
#define __bool_true_false_are_defined 1
```

stdbool.h

- Attention : toutes les affectations à \_Bool/bool qui ne sont pas 0 (faux) sont stockées comme 1 (vrai)

# Types primitifs (3)

## Entiers et Char

- Les tailles en octets allouées ne sont pas les mêmes sur tous les systèmes en particulier sur les systèmes embarqués...

Tableau 3.2 : contraintes imposées à la représentation des entiers

Nature de l'entier	Contrainte imposée par la norme	Remarque
Non signé	Codage binaire pur	Un entier non signé de taille donnée est donc totalement portable.
Signé	Un entier positif doit avoir la même représentation que l'entier non signé de même valeur.	La plupart des implémentations utilisent, pour les nombres négatifs, la représentation dite du « complément à deux ».

« Complément à deux » :

- on exprime la valeur en base 2 ;
- tous les bits à gauche du premier bit de droite à 1 sont « inversés » : 1 devient 0 et 0 devient 1 ;

4 = 0000 0100

-4 = 1111 1100

Type	Explication courte	Formateurs
<code>char</code>	Plus petite unité adressable d'une machine, elle peut contenir les caractères de base. C'est une valeur entière qui peut être signée ou non.	<code>%C</code>
<code>signed char</code>	Type char signé, capable de représenter au moins les nombres [-127 ; +127].	<code>%C</code>
<code>unsigned char</code>	Type char non-signé, capable de représenter au moins les nombres [0 ; 255].	<code>%C</code>
<code>short</code> <code>short int</code> <code>signed short</code> <code>signed short int</code>	Type entier minimum, court, entier et signé, capable de représenter au moins les nombres [-32 767 ; +32 767].	<code>%i</code>
<code>unsigned short</code> <code>unsigned short int</code>	Type entier minimum, court, idem que le type entier standard non signé.	<code>%hu</code>
<code>int</code> <code>signed</code> <code>signed int</code>	Type entier standard, signé, capable de représenter au moins les nombres [-32 767 ; +32 767].	<code>%i</code> ou <code>%d</code>
<code>unsigned</code> <code>unsigned int</code>	Idem que le type entier standard, mais non signé, capable de représenter au moins les nombres [0 ; 65 535].	<code>%u</code>
<code>long</code> <code>long int</code> <code>signed long</code> <code>signed long int</code>	Type entier long, entier et signé, capable de représenter au moins les nombres [-2 147 483 647 ; +2 147 483 647].	<code>%li</code> ou <code>%ld</code>
<code>unsigned long</code> <code>unsigned long int</code>	Idem type entier long mais non signé, capable de représenter au moins les nombres [0 ; 4 294 967 295].	<code>%lu</code>
<code>long long</code> <code>long long int</code> <code>signed long long</code> <code>signed long long int</code>	Type entier long long, entier et signé, capable de représenter au moins les nombres [-9 223 372 036 854 775 807 ; +9 223 372 036 854 775 807].	<code>%lli</code>
<code>unsigned long long</code> <code>unsigned long long int</code>	Idem type entier long long mais non signé, capable de représenter au moins les nombres [0 ; +18 446 744 073 709 551 615].	<code>%llu</code>

[https://fr.wikipedia.org/wiki/Types\\_de\\_donnée\\_du\\_langage\\_C](https://fr.wikipedia.org/wiki/Types_de_donnée_du_langage_C)

```
int i1,i2;
i1 = 4; i2 = -5;
printf("%d\n",i1+i2);// -1
// Division entre entiers
printf("%f\n",i1/i2);// 0.00
```

# Types primitifs (4)

## Entiers

- Les tailles en octets allouées ne sont pas les mêmes sur tous les systèmes en particulier sur les systèmes embarqués...
- Si vous voulez des tailles fixes utilisez

```
#include <stdint.h>
```

Specific integral type limits					
Specifier	Signing	Bits	Bytes	Minimum Value	Maximum Value
int8_t	Signed	8	1	$-2^7$ which equals -128	$2^7 - 1$ which is equal to 127
uint8_t	Unsigned	8	1	0	$2^8 - 1$ which equals 255
int16_t	Signed	16	2	$-2^{15}$ which equals -32,768	$2^{15} - 1$ which equals 32,767
uint16_t	Unsigned	16	2	0	$2^{16} - 1$ which equals 65,535
int32_t	Signed	32	4	$-2^{31}$ which equals -2,147,483,648	$2^{31} - 1$ which equals 2,147,483,647
uint32_t	Unsigned	32	4	0	$2^{32} - 1$ which equals 4,294,967,295
int64_t	Signed	64	8	$-2^{63}$ which equals -9,223,372,036,854,775,808	$2^{63} - 1$ which equals 9,223,372,036,854,775,807
uint64_t	Unsigned	64	8	0	$2^{64} - 1$ which equals 18,446,744,073,709,551,615

The limits of these types are defined with macros with the following formats:

- `INTN_MAX` is the maximum value ( $2^N - 1$ ) of the signed version of `intN_t`.
- `INTN_MIN` is the minimum value ( $-2^N - 1$ ) of the signed version of `intN_t`.
- `UINTN_MAX` is the maximum value ( $2^N - 1$ ) of the unsigned version of `uintN_t`.

Wikibooks C Programming/stdint.h

[https://en.wikibooks.org/wiki/C\\_Programming/stdint.h](https://en.wikibooks.org/wiki/C_Programming/stdint.h)

# Types primitifs (5)

## Réels

- Les tailles en octets allouées ne sont pas les mêmes sur tous les systèmes
- Problématique de la précision...

```
[>>> 999999999999999999999999999999999999999999999999999999999999999.0
1e+29
[>>> 999999999999999999999999999999999999999999999999999999999999999.0 + 0.000000000000000000000000000000000000000000000000000000000000001
1e+29
[>>> 999999999999999999999999999999999999999999999999999999999999999.0 + 0.000000000000000000000000000000000000000000000000000000000000002
1e+29
[>>> 999999999999999999999999999999999999999999999999999999999999999.0 + 0.000000000000000000000000000000000000000000000000000000000000003
1e+29
[>>> 999999999999999999999999999999999999999999999999999999999999999.0 + 0.000000000045000000000000000000000000000000000000000000000000003
1e+29
```

<code>float</code>	Type flottant. Simple précision (4 octets, ou 32 bits) sur quasiment tous les systèmes.	%f %F %g %G %e %E %a %A
<code>double</code>	Type flottant. Double précision (8 octets, ou 64 bits) sur quasiment tous les systèmes.	%lf %lf %lg %lg %le %le %la %la
<code>long double</code>	Type flottant. En pratique, selon le système, de la double précision à la quadruple précision.	%Lf %Lf %Lg %Lg %Le %Le %La %La

[https://fr.wikipedia.org/wiki/Types\\_de\\_donnée\\_du\\_langage\\_C](https://fr.wikipedia.org/wiki/Types_de_donnée_du_langage_C)

L'opérateur `sizeof()` retourne la taille en octet de la variable qui lui est passée

```
float f1,j;
i = 4.; j = -5.;
printf("%f\n",i+j);// -1.000000
printf("%f\n",i/j);// 0.800000
```

## 2.2 Codage des nombres

Un `int` est codé sur 32 bits, en base 2 (signe codé par complémentation). Donc `x=19` est codé par le mot sur 32 bits :

000000000000000000000000000000010011

On peut jongler avec la représentation binaire, décalage à gauche :  $19 << 1$  ( $\dots 100110 = 38$ ), à droite  $19 >> 1$  ( $\dots 1001 = 9$ ), masquage (`&`, `|`) etc.

Pour les types `float` et `double`, la différence avec les nombres idéaux (les réels dans ce cas), est encore pire, d'une certaine façon : Il s'agit d'un codage en précision finie : la *mantisse* est codée en base 2, l'*exposant* également, et le tout sur un nombre fini de bits (cf. norme IEEE 754).



Attention, à cause de tout cela, et des erreurs d'arrondi dues au nombre fini de bits utilisés pour le codage des nombres, il n'y a pas associativité de l'addition, multiplication etc.

Considérons le programme suivant :

```
float x, y;
x = 1.0f;
y = x+0.00000001f;
```

Alors, `x` et `y` ont la même valeur !

Un grand classique (Kahan-Muller) de programme qui donne un résultat surprenant est le suivant :

```
float x0, x1, x2;
x0=11/2;
x1=61/11;
for (i=1; i<=N; i++) {
    x2=111 - (1130-3000/x0)/x1;
    x0=x1;
    x1=x2;
}
```

Une exécution produit la suite : 5.5902 ; 5.6333 ; 5.6721 ; 5.6675 ; 4.9412 ; -10.5622 ; 160.5037 ; 102.1900 ; 100.1251 ; 100.0073 ; 100.0004 ; 100.0000 ; 100.0000 ; ... (stable) alors que la vraie limite dans les réels est 6 !

Ou un autre exemple, beaucoup plus simple.

```
float x0, x1;
x0=7/10;
```

Extrait de « Principes des langages de programmation », Eric Goubaud, 2014,  
<https://www.enseignement.polytechnique.fr/informatique/INF321/poly2013.pdf>

# Petit point codage des nombres réels

14

CHAPITRE 2. PROGRAMMATION IMPÉRATIVE

```
for (i=1; i<=N; i++) {
    x1 = 11*x0 - 7;
    x0 = x1;
}
```

Ce programme produit la suite 0.7000 ; 0.7000 ; 0.7000 ; 0.6997 ; 0.6972 ; 0.6693 ; 0.3621 ; -3.0169 ; -40.1857 ; -449.0428 ; -4946.4712 ; -54418.1836 ; -598607.0000 ; -6584684.0000 ; diverge vers  $-\infty$ , alors que dans les réels c'est la suite constante égale à 0.7 !

# Les pointeurs

Qu'est-ce qu'un pointeur ? Quelles sont ses utilisations ? Quels sont les opérateurs associés ? Qu'est-ce que la valeur NULL ? Qu'est-ce qu'un void \* ? Qu'est-ce qu'une allocation en programmation ? Qu'est-ce qu'une allocation dynamique ? Qu'est-ce qu'une variable statique ? Qu'est-ce qu'une variable dynamique ? Que signifie libérer la mémoire ? Comment et pourquoi la libérer ? Qu'est-ce qu'une adresse mémoire non valide ? Quelles sont les erreurs classiques provoquées par des adresses mémoire non valide ?

# Démo... Test... Expérimentations

- Ouvrez votre IDE/Editeur
- Créez un nouveau fichier pointeurstests.c
- Déclarez un nouveau main
- Amusez-vous !

## Les opérateurs/instructions à connaître

```
type *pt_sur_un_type = NULL; //Déclaration d'un pointeur sur...
&variable //pour récupérer l'adresse mémoire de variable
*pt_sur_un_type // pour atteindre l'objet pointé en lecture ou en écriture
```

# Expérimentation, les pointeurs

```
#include <stdio.h>

int main(int argc, char const *argv[]){
    int *pt_int; // i pointeur sur entier
    int un_int = 5;
    printf("Valeur du pointeur pt_int à la décalration %p\n", pt_int);
    //A la déclaration en C un pointeur il est recommandé de l'assigner à NULL
    pt_int = NULL;
    printf("Valeur du pointeur pt_int après assignation a NULL %p\n", pt_int);
    pt_int = &un_int; //Récupération de l'adresse mémoire de un_int
    printf("Vérification du fait que la val de pt_int %p et l'adresse de un_int %p sont les mêmes\n", pt_int, &un_int);
    printf("Vérification du fait que la val un_int %i et la valeur de l'objet pointé par pt_int %i sont les mêmes\n", un_int, *pt_int);
    //Modification de la valeur de un_int en passant par le pointeur
    *pt_int = 2;
    printf("Vérification du fait que la val de un_int %i et donc de l'objet pointé par pt_int %i ont bien été modifiés\n", un_int, *pt_int);
    return 0;
}
```

A sa création en C il est recommandé de mettre le pointeur à la valeur NULL.

# Définition : pointeur (i)

- Un pointeur est une variable qui contient l'adresse mémoire codée en binaire (Notée en Hexa) d'un autre objet
  - Le nombre d'octets k à réservé pour un pointeur dépend de la machine et du “modèle” de mémoire choisi : taille du “mot machine”
  - En général 32 bits sur proc 32 bits, 64 bits sur proc 64...
- Un pointeur a toujours la même taille sur un même système, quel que soit le type de la variable pointée

```
sizeof(void*) = sizeof(int*) =  
sizeof(double*) = sizeof(char*)
```

## Définition : pointeur (ii)

- Permet de faire référence à n'importe quel octet de la mémoire
- Permet d'accéder à la valeur d'une variable ou d'un objet présent à cette adresse
- L'objet dont l'adresse est repérée par un pointeur s'appelle **objet pointé**  
c'est souvent un objet créé dynamiquement
  - Si p est le pointeur qui le désigne, cet objet est accessible par \*p
- Ecriture niveau algorithmique :
  - On définit un pointeur par ^
  - On utilise @ ou & pour obtenir l'adresse d'une variable existante

# Question pointeur

- La déclaration d'un pointeur implique-t-elle de connaître le type des objets qu'il peut pointer ?

- Non : même taille pour le pointeur donc a priori pas besoin de connaître le type de l'objet pointé

- Oui : il faut savoir le nombre de cases mémoires qu'il faut considérer pour récupérer l'objet pointé

# Exercice pointeurs sur machine 32 bits

```
int i=3;  
int* pi;  
pi=&i;
```

```
char c='c' ;  
char* pc;  
pc=&c;
```

Objet	Adresse mémoire	valeur
i	4831836000	
pi		
c		
pc		

Remplissez ce tableau

# Solution

```
int i=3;  
int* pi;  
pi=&i;
```

```
char c='c' ;  
char* pc;  
pc=&c;
```

Objet	Adresse mémoire	valeur
i	4831836000	3
pi	4831836004	4831836000
c	4831836008	'c'
pc	4831836009	4831836008

[https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS\\_C/chapitre3.html](https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS_C/chapitre3.html)

# Mémoire RAM

```
1 #include <stdio.h>
2
3 char C = 'a';
4 int A = 256;
5 int B = 129;
```

Ecrivez le code permettant d'obtenir cet affichage.

Adresse de c = 0x16d68f22f  
Code ASCII du caractère a = 97  
Adresse de a = 0x16d68f228  
Adresse de b = 0x16d68f224

00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000

Octet 3   Octet 2   Octet 1   Octet 0

Adresse n°116  
Adresse n°112  
Adresse n°108  
Adresse n°104  
Adresse n°100

Exemple sur 32 bits

00000000	00000000	00000000	<b>10000001</b>
00000000	00000000	00000000	00000000
00000000	00000000	00000000	00000000
00000000	00000000	00000001	<b>00000000</b>
00000000	00000000	00000000	01100001

Octet 3   Octet 2   Octet 1   Octet 0

Adresse n°116 : B  
Adresse n°112  
Adresse n°108  
Adresse n°104 : A  
Adresse n°100 : C

Chaque nouvel objet commence toujours sur une frontière de mot. Seuls les objets de types char (1 octet) ou éventuellement short (2 octets) peuvent posséder des adresses intermédiaires d'octets.

# Exercice

```
1 #include <stdio.h>
2
3 char C = 'a';
4 int A = 256;
5 int B = 129;
```

Ecrivez le code permettant d'obtenir cet affichage.

Adresse de c = 0x16d68f22f  
Code ASCII du caractère a = 97  
Adresse de a = 0x16d68f228  
Adresse de b = 0x16d68f224

```
printf("Adresse de c = %p\n", &c);
printf("Code ASCII du caractère %c = %i\n", c, c);
printf("Adresse de a = %p\n", &a);
printf("Adresse de b = %p\n", &b);
return 0;
```

# Résumé type primitif : les pointeurs en C

- Obtenir et afficher l'adresse mémoire d'une variable via l'opérateur de référence '&'

```
char a,b;
a = 'A';
b = 'B';
printf("%d\n",a); //65
printf("%p\n",&a); //l'adresse de a 0x16dc6f23f
```

- Le type pointeur sur, récupérer l'objet pointé, l'opérateur d'indirection '\*'

```
char a = 'A';
char *pt_c = NULL; //pt_c est un pointeur sur un char
pt_c = &a; //pt_c contient l'adresse de a
printf("%c\n",*pt_c); //'A'
printf("%p\n",pt_c); //l'adresse de a
*pt_c = 'B';
printf("%c\n",*pt_c); //le contenu pointé par pt_c
printf("%c\n",a); //a a également été modifié
```

A  
0xb3cb22f  
B  
B

# Pointeurs et allocation dynamique de la mémoire

Testez ce code, ou mieux devinez et expliquer ce qui va être affiché

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

int main(int argc, char const *argv[]){
    int *pt_int=NULL;
    *pt_int = 10;
    printf("%p\n",pt_int);
    printf("%d\n",*pt_int);
    return 0;
}
```

On essaie d'écrire dans une zone non allouée de la mémoire on a une « Segmentation fault »

# Allocation dynamique de la mémoire : malloc

## Libération de la mémoire : free

- Il faut affecter l'espace mémoire nécessaire à l'accueil d'une valeur via les fonctions d'allocation dynamique

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

Il faut **caster** la valeur de retour pour correspondre à ce qui est attendu  
En cas d'échec NULL est retourné.

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

int main(int argc, char const *argv[]){
    int *pt_int=NULL;
    //Allocation dynamique d'un int
    pt_int = (int *) malloc(sizeof(int));
    printf("%p\n",pt_int);
    //Essai d'affectation après allocation
    *pt_int = 20;
    printf("%i\n",*pt_int);
    free(pt_int);
    pt_int=NULL;
    return 0;
}
```

Quand vous écrivez une allocation vous **DEVEZ** écrire la **libération** de la mémoire allouée.



# Fonctions d'allocation dynamique de mémoire

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

Pour memory allocation

Alloue un bloc de mémoire de taille totale ‘taille’ octets sans modifier les valeurs dans la zone de mémoire.

```
void* calloc(size_t elementCount, size_t elementSize);
```

Pour contiguous allocation

Alloue un bloc de mémoire d'un nombre elementCount éléments consécutifs chacun de taille elementSize octets et met tous les octets à 0.

```
void* realloc( void * pointer, size_t memorySize );
```

Pour re-allocation

Réallocation d'un bloc de mémoire dans le tas. Si l'espace libre suivant le bloc à réallouer pointé par ‘pointer’ est suffisamment grand, le bloc est simplement agrandi. Si l'espace libre n'est pas suffisant, un nouveau bloc de mémoire sera alloué, le contenu de la zone d'origine recopié dans la nouvelle zone et le bloc mémoire d'origine sera libéré automatiquement.

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

Pour la libération de l'espace mémoire alloué

# Résumé fonctions d'allocation/libération mémoire

## Malloc()

```
int* ptr = ( int* ) malloc ( 5* sizeof ( int ));
```

ptr =  A large 20 bytes memory block is dynamically allocated to ptr

← 20 bytes of memory →

DG

## Calloc()

```
int* ptr = ( int* ) calloc ( 5, sizeof ( int ));
```

ptr =  5 blocks of 4 bytes each is dynamically allocated to ptr

← 4b →  
← 20 bytes of memory →

DG

## Realloc()

```
int* ptr = ( int* ) malloc ( 5* sizeof ( int ));
```

ptr =  A large 20 bytes memory block is dynamically allocated to ptr

← 20 bytes of memory →

```
ptr = realloc ( ptr, 10* sizeof( int ));
```

ptr =  The size of ptr is changed from 20 bytes to 40 bytes dynamically

← 40 bytes of memory →

DG

## Free()

```
int* ptr = ( int* ) calloc ( 5, sizeof ( int ));
```

ptr =  5 blocks of 4 bytes each is dynamically allocated to ptr

← 4b →  
← 20 bytes of memory →

operation on ptr

```
free( ptr )
```

 The memory of ptr is released

DG

Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc()

<https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/>

## Dates importantes - Séminaires

- Choix argumenté (deux phrases par sujet) par trinôme/binôme de 3 sujets par ordre de préférence avant **partie 24 Janvier avant 18h00**
- Attribution des sujets aux groupes : **Vendredi 31 Janvier**
- Première proposition de plan **Lundi 17 Février**
  - Définition des objectifs
  - Description en une ou deux phrases de chaque partie et sous-parties —> Plan détaillé
  - Répartition des tâches au sein du binôme et rattachement aux parties
  - Construction de la bibliographie et référencement dans les différentes parties du plan
- Présentations orales le **3 Avril**
- Rapports à rendre **le 26 Mars**

# Exécution mémoire (RAM) d'un programme C

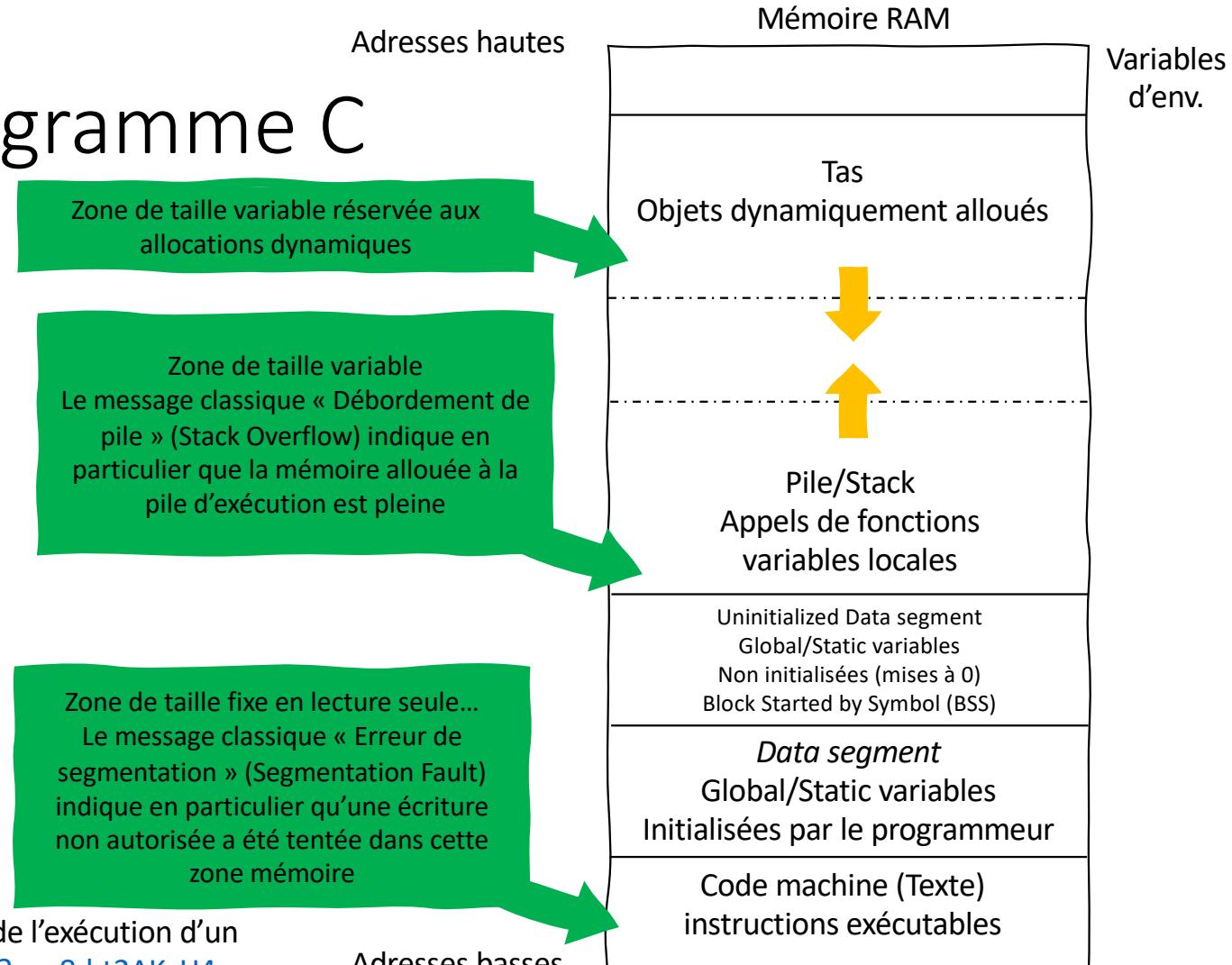
## La pile et le tas !

- Pile (Last In First Out) : espace mémoire de travail (Frame) pour un thread d'exécution
  - A l'appel d'une fonction, réservation d'un bloc en sommet de pile pour paramètre de la fonction, les variables locales et l'adresse de retour
  - Lors du retour le bloc est libéré (mise à jour pointeur sommet de pile)
- Tas : mémoire réservée à l'allocation dynamique
  - Réserveration/allocation d'un nouvel espace à chaque appel aux fonctions malloc(), calloc(), ou realloc() en C
  - Désallocation free() obligatoire

# Exécution d'un programme C

La taille de la pile d'appel (d'exécution) dépend de plein de facteurs le langage de programmation, l'architecture de la machine, machine architecture, quantité de mémoire disponible, etc. Si le programme essaye d'utiliser plus d'espace que ce qui est disponible dans la pile (ce qui est un buffer overflow) la pile est en débordement...

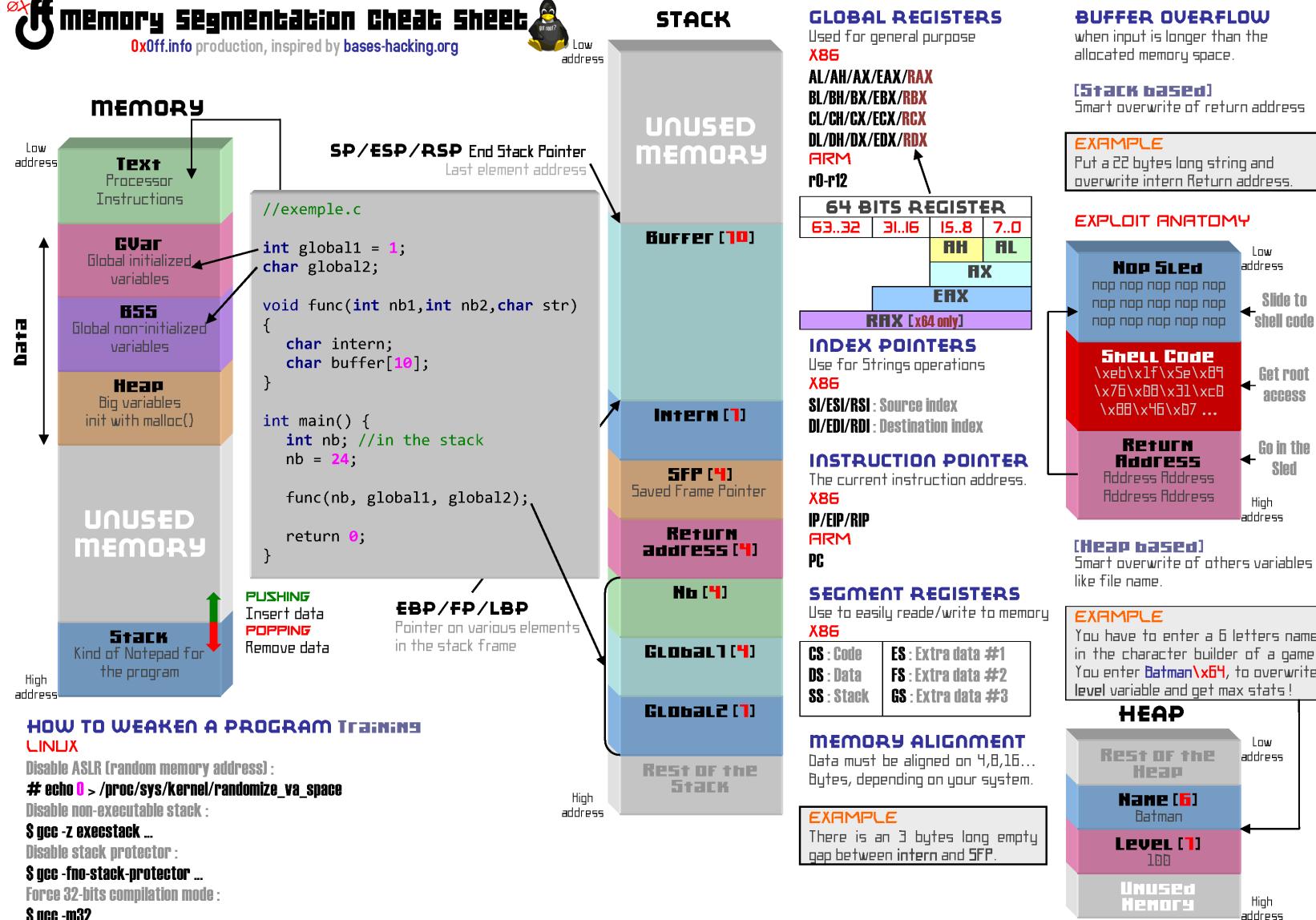
Une explication des concepts de pile et de tas lors de l'exécution d'un programme en C <https://www.youtube.com/watch?v=8ht2AKyH4>





# memory Segmentation cheat sheet

0xOff.info production, inspired by bases-hacking.org



# Les erreurs communes dans la gestion de la mémoire en C

Cf. Exercice TD identification  
erreur gestion de mémoire

1. Fuites mémoire (*memory leaks*) si on oublie de libérer via free la mémoire allouée  
Peut prendre plus ou moins de temps...
2. Mémoire insuffisante pour réussir l'allocation, malloc retourne NULL  
Toujours vérifier que malloc retourne un pointeur != NULL
3. Non initialisation de la mémoire allouée via malloc  
Si vous voulez une initialisation à 0 utilisez calloc (plus lent)
4. Non allocation mémoire préalablement à une tentative d'accès en lecture ou écriture  
SEGMENTATION FAULT
5. Non libération de la mémoire allouée  
**Dès que** vous écrivez malloc ou calloc écrivez le free correspondant
6. Pointeurs pendant (*dangling pointer*), tentative d'accès à de la mémoire déjà libérée  
via un pointeur qui ne pointe plus sur de la mémoire allouée
7. Désallocation mémoire multiple  
Toujours valuer un pointeur libéré à NULL car free ne fait rien sur un pointeur NULL

# Les erreurs/maladresses communes dans la gestion de la mémoire en C

Cf. Exercice TD identification  
erreur gestion de mémoire

8. Tentative de libération de mémoire non allouée dynamiquement  
SEGMENTATION FAULT
9. Utiliser une allocation dynamique (plus lente) de tableau, alors qu'une allocation statique (int tab[TAILLE]) aurait suffi  
C'est le cas si vous n'avez pas besoin du tableau déclaré en dehors de votre fonction
10. Utiliser sizeof sur un tableau alloué dynamiquement pour déterminer sa taille  
Fonctionne avec un tableau statique, mais pas dynamique, dans ce cas c'est la taille du pointeur qui est renvoyée
11. Toujours utiliser le pointeur ayant servi à l'allocation dynamique pour la libération de la mémoire  
Travailler sur une copie de ce pointeur afin de ne jamais perdre l'adresse originale de début du bloc alloué.

# Pointeur générique void\*

- Pointeur de type indéfini, permettant de manier l'adresse mémoire d'un objet dont on ignore le type.
- Toutes les opérations d'accès en lecture ou écriture nécessiteront un cast explicite

```
int un_int = 49;
void* pt = &un_int;
printf("un_int contient %d",*((int*)pt));
//printf("Essayez sans cast cela fait une erreur à la compilation ! %d", *pt);
return 0;
```

Nous reviendrons sur le sujet...

# Exercice

- Déclarez deux entiers et trois pointeurs sur void et échangez le contenu des deux entiers initialisés respectivement à 10 et 20 de façon qu'à la fin de l'exécution les valeurs aient été échangées.
- Vous n'avez le droit de manipuler les entiers que via les pointeurs et jamais directement.

# Solution

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

int main(int argc, char const *argv[]){
    int a = 10;
    int b = 20;
    void * pt_a = &a;
    void * pt_b = &b;
    void * pt_temp = NULL;
    printf("Les deux entiers au départ sont %d et %d\n",a,b);
    //Permutation utilisant les pointeurs
    pt_temp = (int*)malloc(sizeof(int));
    if pt_temp == NULL return 1;
    *(int*)pt_temp = *(int*)pt_a;
    *(int*)pt_a = *(int*)pt_b;
    *(int*)pt_b = *(int*)pt_temp;
    free(pt_temp);
    printf("Les deux entiers après permutation sont %d et %d\n",a,b);
    return 0;
}
```

# Pointeurs et constantes

- Pointeur variable sur un objet constant

```
const int *pt_sur_const;  
int const *pt_sur_const_int;
```

- Pointeur constant sur un objet variable

```
int *const pt_const_sur_int;
```

- Pointeur constant sur un objet constant

```
const int * const pt_const_sur_int_const;
```

# Pointeur variable sur objet constant

```
const int *pt_sur_const;
int const *pt_sur_const_int;
```

- L'objet pointé devient constant donc non modifiable par le pointeur même s'il n'est pas constant à l'origine.
- Le pointeur peut changer de valeur et pointer sur un autre objet.

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

int main(){
    const int val = 100; //objet constant
    int v2 = 0; //objet non constant
    const int *p = &val; //pointeur de const int, non constant
    *p += 10; // erreur, objet pointé supposé constant
    p = &v2; // p non constant, peut varier

    *p += 10; // erreur, objet pointé supposé constant

    printf("val : %d et *p : %d\n", val, *p);

    return 0;
}
```

- Pointeur variable sur un objet constant

```
const int *pt_sur_const;
int const *pt_sur_const_int;
```

- Pointeur constant sur un objet variable

```
int *const pt_const_sur_int;
```

- Pointeur constant sur un objet constant

```
const int * const pt_const_sur_int const;
```

Pour chaque instruction ci-contre déterminez s'il y a erreur ou non et justifiez votre réponse.

# Variation... Que se passe-t-il ici ?

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

int main(){
    const int val = 100;      //objet constant
    int *p = &val;          //pointeur sur int ordinaire
    *p += 10;               // pas d'erreur, pourtant objet pointé supposé constant

    printf("val : %d et *p : %d\n", val, *p);

    return 0;
}
```

- Pointeur variable sur un objet constant  
`const int *pt_sur_const;  
int const *pt_sur_const_int;`
- Pointeur constant sur un objet variable  
`int *const pt_const_sur_int;`
- Pointeur constant sur un objet constant  
`const int * const pt_const_sur_int_const;`

Le type du pointeur prime sur celui de l'objet pointé !

# Pointeur constant sur objet variable ou non

```
int value0=10;
int value1=60;
const int valueconst=100;

int *const pt_const_sur_int=&value0; //pointeur constant sur int
int *const pt_const_sur_int_const=&valueconst; //pointeur constant sur un int
                                                //constant
pt_const_sur_int = &value1; //erreur pt_const_sur_int constant

*pt_const_sur_int_const += 10; //OK objet pointé devient modifiable, même
                             //si constant à l'origine

printf("val : %d et *p : %d\n", valueconst, *pt_const_sur_int);
```

- Pointeur variable sur un objet constant

```
const int *pt_sur_const;
int const *pt_sur_const_int;
```

- Pointeur constant sur un objet variable

```
int *const pt_const_sur_int;
```

- Pointeur constant sur un objet constant

```
const int * const pt_const_sur_int_const;
```

Le type du pointeur prime sur celui de l'objet pointé et détermine le fonctionnement du pointeur !

# Pointeur constant sur objet constant

```
int value0=10;
int value1=60;
const int * const pt_const_sur_const_int=&value0;
pt_const_sur_const_int=&value1; //Erreur car pointeur constant

*pt_const_sur_const_int += 10; //Erreur pointeur sur un const int

printf("val : %d et *p : %d\n", value1, *pt_const_sur_const_int);
```

Ici L'objet pointé, qu'il soit constant ou pas, et pointeur sont considérés constants et ne peuvent changer de valeur après l'affectation de déclaration.

- Pointeur variable sur un objet constant

```
const int *pt_sur_const;
int const *pt_sur_const_int;
```

- Pointeur constant sur un objet variable

```
int *const pt_const_sur_int;
```

- Pointeur constant sur un objet constant

```
const int * const pt_const_sur_int_const;
```

# Opérations et opérateurs sur les pointeurs

- Les pointeurs ont pour valeur des adresses mémoires
- Plusieurs opérations sont possibles sur les **pointeurs de même type**
  - L'addition avec un entier
  - La soustraction avec un entier
  - La différence entre deux pointeurs
  - La comparaison entre pointeurs via opérateurs ==, !=, <, >, etc.

<https://www.geeksforgeeks.org/pointer-arithmetics-in-c-with-examples/>

# Arithmétique des pointeurs, addition, soustraction d'un entier

Les pointeurs sont des adresses mémoire, les opérations autorisées sur les adresses mémoire sont :

- L'addition (ou la soustraction) d'un entier à un pointeur : `pointeur + unentier` (ou `pointeur - unentier`).
  - Le résultat de `pointeur + unentier` est une nouvelle adresse, décalée de `unentier` cases.
  - *Attention* : La taille d'une "case" (en nombre d'octets) dépend du type du pointeur, c'est-à-dire de la taille mémoire de l'objet pointé.

Ainsi :

- Si `pointeur` est de type `int *`, alors `pointeur + N` décale la case mémoire de `N` cases, et ajoutera donc  $(N*4)$  octets à la valeur de `pointeur`, puisqu'un `int` occupe 4 octets en mémoire;
- Si `pointeur` est de type `char *`, alors `pointeur + N` décale la case mémoire de `N` cases, et ajoutera donc  $(N*1)$  octets à la valeur de `pointeur`, puisqu'un `char` occupe 1 octet;
- Si `pointeur` est de type `double *`, alors `pointeur + N` décale la case mémoire de `N` cases, et ajoutera donc  $(N*8)$  octets à la valeur de `pointeur`, puisqu'un `double` occupe 8 octets;

# Arithmétique des pointeurs

## Différence entre deux pointeurs

- Possible qu'entre pointeurs du même type.
- La valeur renournée sera égale au nombre de cases (conformément à la taille mémoire des objets pointés) existants entre les deux pointeurs.

# Comparaison entre pointeurs

- Il n'est possible que de comparer des pointeurs sur le même type.
- On peut alors savoir si :
  - Les pointeurs pointent sur la même adresse mémoire ==
  - Les pointeurs pointent sur des adresses mémoire différentes !=
  - L'un des pointeurs pointe « plus loin » en mémoire que l'autre >
  - L'un des pointeurs pointe « avant » en mémoire <

# Les fonctions

Mode de passage d'arguments ?

Comment renvoyer plusieurs résultats ?

Nombre variable d'arguments ? Fonctions variadiques...

Fonctions ou Macros ?

# Qu'est-ce qu'une fonction ?

- Unité de traitement fondamentale du C
  - Permet la **factorisation** et donc la **généralisation** de code qui se répète
  - Consiste en la **paramétrisation** d'un bloc de code
  - **Abstraction** d'instructions et/ou de commandes (masquage des détails du code)
- Bloc d'instructions doté d'un nom et :
  - D'un mécanisme d'entrée de valeurs : les paramètres
  - D'un mécanisme de sortie de valeur : **la** valeur de retour

# En C passage d'arguments par valeur

- Cette règle ne souffre aucune exception !

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

void modif(int a, int b){
    a = 10;
    b = 15;
    printf("Dans la fonction modif a=%d, b=%d\n",a,b);
}

int main(int argc, char const *argv[]){
    int a=0, b=0;
    modif(a,b);
    printf("Dans le main, après appel à modif a=%d, b=%d\n",a,b);
    return 0;
}
```

Qu'est-ce qui est affiché à l'exécution ?

# Implications du passage d'arguments par valeur

- La valeur de l'expression passée en paramètre est copiée dans une variable locale
- Les modifications internes à la fonction portant sur les variables paramètres ne sont pas répercutées à l'extérieur de la fonction puisque on travaille sur des copies locales

# Exercice Swap, cas d'école...

Ecrivez une procédure swap\_int qui prend en paramètre deux pointeurs sur des entiers et qui échange le contenu des deux variables de telle sorte que la valeur de la variable 1 se retrouve dans la variable 2 et vis-versa.

Vous invoquerez cette fonction depuis un main et afficherez les contenus des deux variables avant et après appel à la procédure swap.

# Solution

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

void swap(int* pt_a, int* pt_b){
    int temp = *pt_a;
    *pt_a = *pt_b;
    *pt_b=temp;
}

int main(int argc, char const *argv[]){
    int a = 10;
    int b = 20;
    printf("Les deux entiers au départ sont %d et %d\n",a, b);
    //Permutation via appel à Swap
    swap(&a,&b);
    printf("Les deux entiers après permutation sont %d et %d\n",a, b);
    return 0;
}
```

# Passage par référence et pointeurs

- Si on passe en paramètre à une fonction l'adresse mémoire d'une variable il lui sera possible le lire et écrire dans cet espace mémoire et ainsi de modifier le contenu de cette variable
- L'adresse mémoire d'un objet est une **référence**
- Les variables qui prennent comme valeur des références sont des **pointeurs**
- Ce qui est passé en paramètre c'est **la valeur de l'adresse**

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

int main(int argc, char const *argv[]){
    int intlu = 0;
    scanf("%d", &intlu); //On passe la référence à intlu, cad son adresse
    printf("La valeur lue est %d", intlu);
}
```

# Fonction empoisonnement...

Ecrivez une procédure `empoisonner` prenant deux paramètres entiers, permettant de soumettre un personnage – représenté par un nombre entier de points de vie – à un poison – représenté également par un entier. Lorsque le poison agit, il fait perdre des points de vie au personnage (point de vie du personnage - points de nocivité du poison), et perd également de la nocivité (décrémentation des points de nocivité du poison de 1 à chaque empoisonnement).

Quand le poison est totalement épuisé, il retrouve par magie de la force, sans toutefois jamais dépasser 20 !

Proposez un code pour la fonction empoisonnement

Avant empoisonnement Faustus a 35 points de vie et la cigue est au niveau 15  
Après empoisonnement Faustus a 20 points de vie et la cigue est au niveau 14

# Un solution

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

void empoisonnement(int *personnage, int *poison);

int main(){
    srand((unsigned)time(NULL));
    int faustus = 35;
    int cigue = rand()%20;
    printf("Avant empoisonnement Faustus a %d points de vie et la cigue est au niveau %d\n",faustus,cigue);
    empoisonnement(&faustus,&cigue);
    printf("Après empoisonnement Faustus a %d points de vie et la cigue est au niveau %d\n",faustus,
    cigue);
    return 0;
}

void empoisonnement(int *personnage, int *poison){
    *personnage -= *poison;
    if (*poison>0)
        (*poison)--;
    else (*poison) = rand()%20;
}
```

Avant empoisonnement Faustus a 35 points de vie et la cigue est au niveau 15  
Après empoisonnement Faustus a 20 points de vie et la cigue est au niveau 14

Nous reviendrons dessus en Fonctionnel

# Les fonctions récursives

Simple – multiple – croisée – terminale

# Les fonctions récursives, récurivité simple

- Fonction qui s'appelle elle-même.
- Exemple basique

## Exemple

L'exemple trivial d'algorithme récursif est le calcul de la factorielle d'un entier.

### Algorithme 3.1 : Calcul de la factorielle de $n \in \mathbb{N}$

Fonction **factorielle**( $n$ )

Entrées :  $n \in \mathbb{N}$ .

Sorties :  $n!$ .

début

  si  $n = 0$  alors  
    ▷ Condition d'arrêt  
    retourner 0

  sinon  
    ▷ Appel récursif  
    retourner  $n * \text{factorielle}(n - 1)$

Barra, V., 2021. Informatique MP2I et MPI : CPGE 1re et 2e années - Nouveaux programmes - ScholarVox Université, Ellipses. ed.  
<http://univ.scholarvox.com.udcpp.idm.oclc.org/catalog/book/docid/88915779>



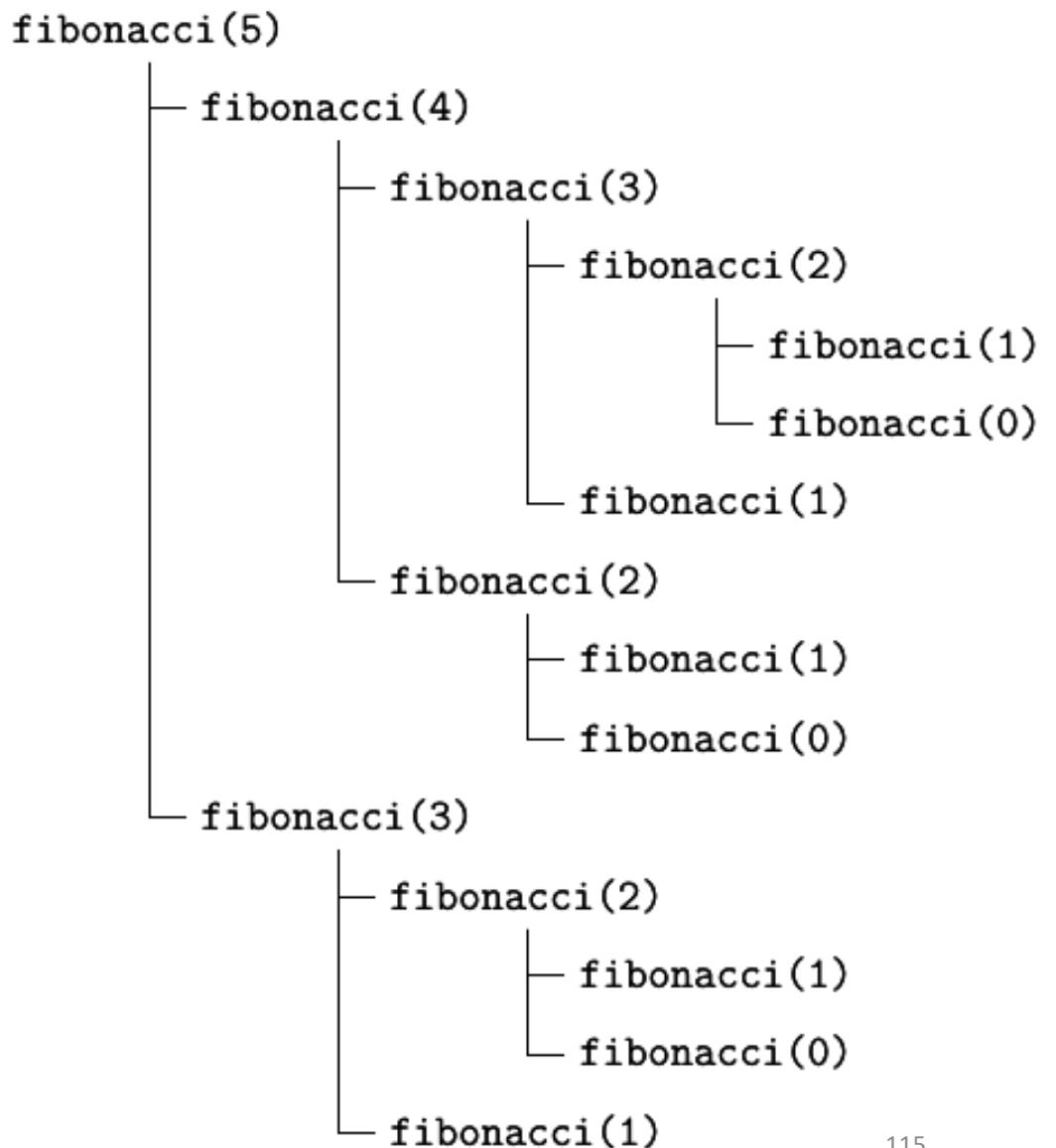
```
unsigned long factoriel (int n)
{
    if (n < 0) {
        exit (EXIT_FAILURE);
    }
    else if (n == 1 || n == 0) {
        return 1L;
    }

    return n * factoriel (n - 1);
}
```

C

# Récursivité multiple

```
def fibonacci(n) :  
    if n <= 1 :  
        u = 1  
    else :  
        u = fibonacci(n-1) + fibonacci(n-2)  
    return u
```



# Récursivité croisée

- Deux fonctions (algorithmes) qui sont mutuellement récursifs.
- Ici Pair appelle Impair et vice versa.

## Exemple



Pour tester la parité d'un entier  $n$ , on utilise les deux fonctions  $\text{Pair}(n)$  et  $\text{Impair}(n)$ .

### Fonction $\text{Pair}(n)$

Entrées :  $n$ .

Sorties : Vrai si  $n$  est pair.

début

```
    si  $n = 0$  alors
        retourner Vrai
    sinon
        retourner
            Impair( $n - 1$ )
```

### Fonction $\text{Impair}(n)$

Entrées :  $n$ .

Sorties : Vrai si  $n$  est pair.

début

```
    si  $n = 0$  alors
        retourner Faux
    sinon
        retourner
            Pair( $n - 1$ )
```

$\text{Pair}(3)$  appelle  $\text{Impair}(2)$ , qui appelle  $\text{Pair}(1)$ , qui appelle  $\text{Impair}(0)$  qui renvoie Faux.

# Récursivité terminale

- Utilisation d'un accumulateur pour stocker les résultats intermédiaires, le calcul est effectué au fur et à mesure, aucun calcul n'est effectué à la remontée de l'appel récursif
- Facilement transformable en version itérative

## Exemple

On peut calculer la factorielle en utilisant une récursion terminale ( $n! = \text{Fact}(n, 1)$ ).

**Fonction `Fact`(*n,temp*)**

**Entrées :** *n,temp* l'accumulateur des produits.

**Sorties :** *n!*.

**début**

**si** *n* ≤ 1 **alors**

**retourner** *temp*

**retourner** *Fact*(*n* − 1, *n,temp*)



# Récursivité terminale

```
unsigned long factoriel_terminal (int n, unsigned long acc){  
    if (n < 0) {  
        exit (EXIT_FAILURE);  
    }  
    if (n == 0) {  
        return acc;  
    }  
    return factoriel_terminal (n - 1, n * acc); }
```

C

// A l'appel il faut donner à acc la valeur de l'accumulateur correspondant à la condition d'arrêt, ici 1L, car 0! = 1

# Récursivité imbriquée

- L'appel récursif contient lui-même un appel récursif

## Exemple

La fonction d'Akermann est donnée par

$$A(m, n) = \begin{cases} n + 1 & \text{si } m = 0 \\ A(m - 1, 1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{si } m > 0 \text{ et } n > 0 \end{cases}$$

L'algorithme de calcul correspondant est un algorithme de récursivité imbriquée. La ressource nécessaire pour le calcul est très importante, même pour de petites valeurs de  $m$  et  $n$ . Par exemple, le calcul de  $A(3, 2) = 29$  nécessite 541 appels à  $A$ . C'est une des raisons qui fait que l'on utilise cette fonction pour tester des langages de programmation.

Notons enfin que  $A(m, n)$  croît très vite, et que par exemple  $A(5, 1) > 10^{80}$ .



# Types composés

Nouveaux types constitués de regroupement de types simples

Ensemble organisé de données

Homogènes : tableaux

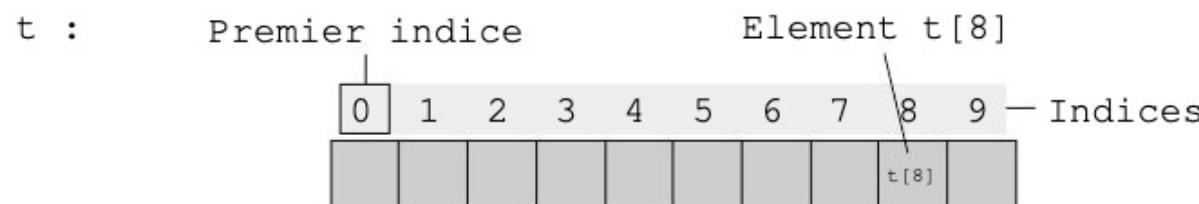
Hétérogènes : enregistrements

# Types composés, homogènes : tableaux

- Collection dont tous les éléments sont de même type.
- Chaque élément possède une adresse permettant un accès direct.
- Les tableaux sont mutables
- En C : tableau statique, de longueur fixe.
- Cases mémoires adjacentes

Possibilité d'allouer dynamiquement des tableaux et de modifier ultérieurement leur taille...

En C, une chaîne de caractères est un tableau de caractères terminé par \0.



Si débordement du tableau - écriture en dehors des bornes dans une zone mémoire non réservée - plantage à l'exécution... **Testez le !**

# Tableaux statiques C

```
#include <stdio.h>

int main(int argc, char const *argv[]){
    int tab1[5] = {234,45,23,15,4};
    int tab2[10]={0,1,2}; //Le reste est rempli de 0
    int tab3[] = {140,10,0,1,22}; //prend la taille 5
    for (int i=0; i<10; i++) printf("%i ",tab2[i]);
    return 0;
}
```

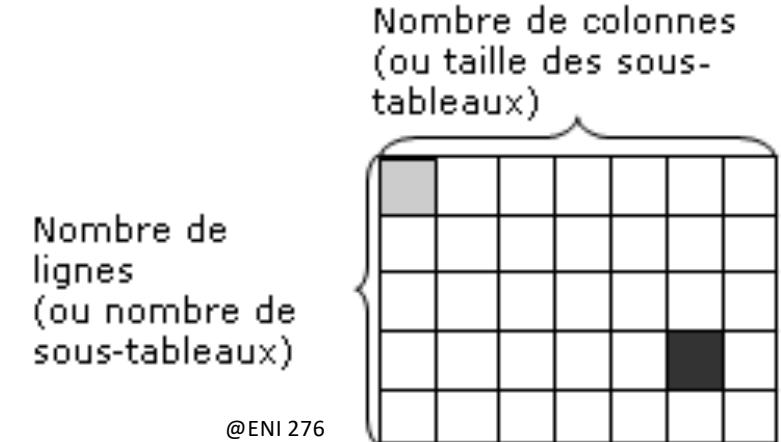
# Tableaux à plusieurs dimensions

- Il faut spécifier pour chaque dimension sa taille avec l'opérateur []
- Exemple de déclaration de matrice, tableau à deux dimensions

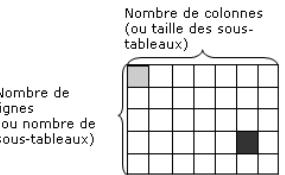
```
#include <stdio.h>
#define NB_LIGNES 5
#define NB_COLONNES 7

int main(int argc, char const *argv[]){
    float matrice[NB_LIGNES][NB_COLONNES]={0}; //Attention si
    vous n'initialisez pas à 0
    //il y aura potentiellement n'importe quoi dans les
    cases...
    matrice[0][0]=10; //En gris
    matrice[3][5]=50; //En noir
    for (int ligne = 0;ligne < NB_LIGNES;ligne++){
        for (int colonne = 0; colonne < NB_COLONNES; colonne++){
            printf("%.0f \t",matrice[ligne][colonne]);
        }
        printf("\n");
    }
    return 0;
}
```

C'est en fait un tableau à une dimension dont chaque élément est un tableau le parcours des indices de droite (colonnes) est plus rapide (adjacence), que celui des indices de gauche (lignes) saut en mémoire...



# Tableaux à plusieurs dimensions

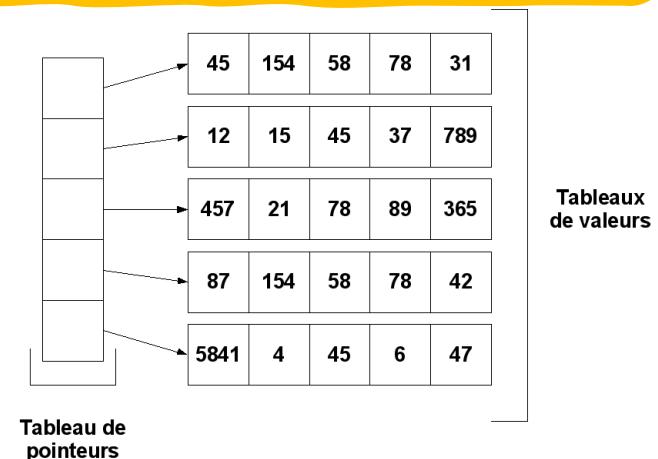


- Il faut spécifier pour chaque dimension sa taille avec l'opérateur []
- Exemple de déclaration de matrice, tableau à deux dimensions

```
#include <stdio.h>
#define NB_LIGNES 5
#define NB_COLONNES 7

int main(int argc, char const *argv[]){
    float matrice[NB_LIGNES][NB_COLONNES]={0}; //Attention si
    vous n'initialisez pas à 0
    //il y aura potentiellement n'importe quoi dans les
    cases...
    matrice[0][0]=10; //En gris
    matrice[3][5]=50; //En noir
    for (int ligne = 0;ligne < NB_LIGNES;ligne++){
        for (int colonne = 0; colonne < NB_COLONNES; colonne++){
            printf("%.0f \t",matrice[ligne][colonne]);
        }
        printf("\n");
    }
    return 0;
}
```

C'est en fait un tableau à une dimension dont chaque élément est un tableau le parcours des indices de droite (colonnes) est plus rapide (adjacence), que celui des indices de gauche (lignes) saut en mémoire...



# Tableaux à N dimensions

- On peut avoir des tableaux à un nombre quelconque de dimensions

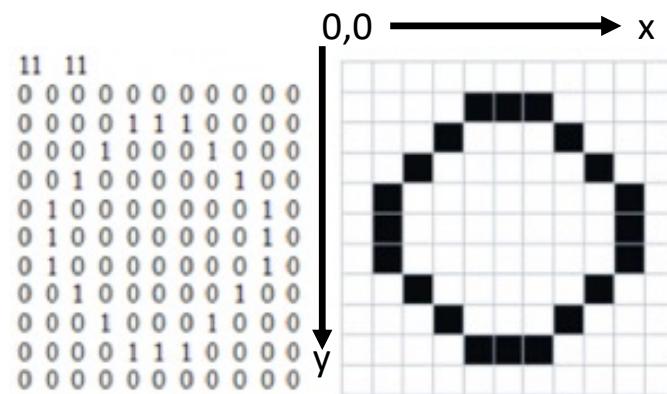
Exemple : char tata[3][3][3];

tata[0] [0] [0]  
tata[0] [1] [0]  
tata[0] [2] [0]  
tata[1] [0] [0]  
tata[1] [1] [0]  
tata[1] [2] [0]  
tata[2] [0] [0]  
tata[2] [1] [0]  
tata[2] [2] [0]

tata[0] [0] [1]  
tata[0] [1] [1]  
tata[0] [2] [1]  
tata[1] [0] [1]  
tata[1] [1] [1]  
tata[1] [2] [1]  
tata[2] [0] [1]  
tata[2] [1] [1]  
tata[2] [2] [1]

tata[0] [0] [2]  
tata[0] [1] [2]  
tata[0] [2] [2]  
tata[1] [0] [2]  
tata[1] [1] [2]  
tata[1] [2] [2]  
tata[2] [0] [2]  
tata[2] [1] [2]  
tata[2] [2] [2]

# Tableaux à 2 dimensions : Images Bitmap



# A quoi peuvent servir des tableaux à N dimensions ?

- Imaginons que nous devons établir la liste des noms des voyageurs assis dans les trains de plusieurs gares sans perdre les informations qui permettent de localiser chacun des voyageurs. Admettons que nous ayons 4 gares, un maximum de 100 trains par gare, de 25 wagons par train, et de 60 places par wagon. Une structure de données peut être :

```
char stock_noms [4] [100] [25] [60] [100];
```

Je prends mon train dans la gare 3, le train n°33, le wagon n°17, la place n°42. L'affectation n'est pas possible avec les chaînes de caractères et le nom est stocké par copie avec la fonction strcpy\_s() :

```
strcpy_s(stock_noms[3][33][17][42], 100, "Frédéric Drouillon");
```

Plus propre et lisible...

```
enum voyageurs {  
    NORD,  
    EST,  
    LYON,  
    MONTPARNASSE,  
    NB_GARE,  
    NB_TRAIN=100,  
    NB_WAGON=25,  
    NB_PLACE=60,  
    TAILLE_NOM=100 } ;  
char  
stock_noms [NB_GARE] [NB_TRAIN]  
[NB_WAGON] [NB_PLACE]  
[TAILLE_NOM] ;
```

Encore plus propre et  
lisible les structures...

# Les chaines de caractères : tableaux de caractères

- Suite de caractères stockée dans un tableau de caractères et terminée par '\0'.
- Attention le '\0' est important toutes les fonctions string se basent dessus pour tester la fin de la chaîne.

```
char chaine[] = {'e','s','s','a','i','\0'};
char chaine_nonfinie[] = {'e','s','s','a','i'};
char tab_char[25] = "Bonjour\n";
printf("La taille du tableau est : %i\n", (int)sizeof(tab_char));
printf("La taille de la chaîne \"Bonjour\n\" stockée dans le tableau est : %i\n", (int)sizeof("Bonjour\n"));
printf("La chaîne finie contient %s\n", chaine);
printf("La chaîne non finie contient %s\n", chaine);
```

# Exercices

Evidemment tout cela existe déjà dans la librairie <string.c>  
\$man string pour voir la liste des fonctions...

- Ecrivez une fonction `get_string_length` qui prend en paramètre un tableau de caractères et retourne sa longeur.
- Ecrivez une fonction `reverse_string` qui prend en paramètre un tableau de caractères et renverse ce tableau en place.
- Ecrivez une fonction `copy_string` qui prend en paramètre deux tableaux de caractères et recopie le premier dans le deuxième.

```
int main(int argc, char const *argv[]){
    char chaine[] = {'e', 's', 's', 'a', 'i', '\0'};
    char chaine_nonfinie[] = {'e', 's', 's', 'a', 'i'};
    char tab_char[25] = "Bonjour\n";
    char tab_copie[25];
    printf("La taille du tableau est : %i\n", (int)sizeof(tab_char));
    printf("La taille de la chaine \"Bonjour\\n\" stockée dans le tableau est : %i\n",
    (int)sizeof("Bonjour\n"));
    printf("La chaine finie contient %s\n", chaine);
    printf("La chaine non finie contient %s\n", chaine_nonfinie);
    char texte[] = "Chaine de test !";
    printf("Le nombre de caracteres constituant la chaine %s est %d\n", texte, get_string_length(texte));
    printf("Le nombre de caracteres constituant la chaine %s est %lu\n", texte, strlen(texte));
    printf("La chaine avant renversement est %s\n", texte);
    reverse_string(texte);
    printf("Et apres renversement en place %s\n", texte);
    copy_string(tab_char, tab_copie);
    printf("La chaine apres copie est %s\n", tab_copie);
    return 0;
}
```

```
La taille du tableau est : 25
La taille de la chaine "Bonjour\n" stockée dans le
tableau est : 9
La chaine finie contient essai
La chaine non finie contient essai
Le nombre de caractères constituant la chaine Chaine
de test ! est 16
Le nombre de caractères constituant la chaine Chaine
de test ! est 16
La chaine avant renversement est Chaine de test !
Et après renversement en place ! tset ed eniahC
La chaine après copie est Bonjour
```

# Une solution...

malibstring.h

```
#ifndef __MALIBSTRING_H_
#define __MALIBSTRING_H_

int get_string_length(char chaine[]);
void reverse_string(char chaine[]);
int copy_string(char chaine_dep[], char chaine_dest[]);

#include "malibstring.h" #endif

int get_string_length(char chaine[]){
    int i = 0;
    while (chaine[i] != 0){i++;}
    return i;
}

void reverse_string(char chaine[]){
    char temp;
    int len = get_string_length(chaine);
    // We keep the last caracter 0 in place
    for (int i = 0; i < len / 2; i++){
        temp = chaine[i];
        chaine[i] = chaine[len - 1 - i];
        chaine[len - 1 - i] = temp;
    }
}

int copy_string(char chaine_dep[], char chaine_dest[]){
    int i = 0;
    while (chaine_dep[i] != 0){
        chaine_dest[i] = chaine_dep[i];
        i++;
    }
    chaine_dest[i] = '\0'; // Last character is NULL
    return 0;
}
```

avecdesstrings\_main.c

```
#include <stdio.h>
#include <string.h>
#include "malibstring.h"

int main(int argc, char const *argv[])
{
    char chaine[] = {'e', 's', 's', 'a', 'i', '\0'};
    char chaine_nonfinie[] = {'e', 's', 's', 'a', 'i'};
    char tab_char[25] = "Bonjour\n";
    char tab_copie[25];
    printf("La taille du tableau est : %i\n",
    (int)sizeof(tab_char));
    printf("La taille de la chaine \"Bonjour\\n\" stockée dans le
    tableau est : %i\n", (int)sizeof("Bonjour\n"));
    printf("La chaine finie contient %s\n", chaine);
    printf("La chaine non finie contient %s\n", chaine_nonfinie);
    char texte[] = "Chaine de test !";

    printf("Le nombre de caracteres constituant la chaine %s est
    %d\n", texte, get_string_length(texte));
    printf("Le nombre de caracteres constituant la chaine %s est
    %lu\n", texte, strlen(texte));

    printf("La chaine avant renversement est %s\n", texte);
    reverse_string(texte);
    printf("Et apres renversement en place %s\n", texte);

    copy_string(tab_char, tab_copie);
    printf("La chaine apres copie est %s\n", tab_copie);
    return 0;
}
```

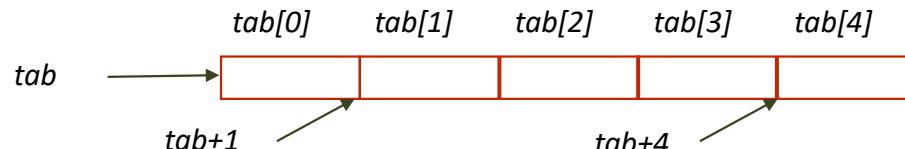
malibstring.c

# Les pointeurs en C et les tableaux

- Accéder aux éléments d'un tableau à partir de l'adresse du premier élément, opérateur crochet « [] »
- C'est de l'arithmétique des pointeurs camouflée

Si on a :

```
int tab[5]
```



$tab$  est l'adresse du premier élément du tableau, adresse non modifiable car gérée dans la pile.

Pour accéder à l'élément à l'indice  $i$  du tableau :

\* $(tab+i)$  ou  $tab[i]$

# Allocation dynamique d'un tableau

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

int main(int argc, char const *argv[])
{
    int *pt_tabint = NULL;
    //Allocation dynamique d'un tableau de 20 int
    pt_tabint = (int *) malloc(20*sizeof(int));
    //If faut vérifier que la mémoire a pu être allouée
    if (pt_tabint==NULL) printf("Allocation impossible\n");
    else{
        //Attention la mémoire est pleine de 0 on ne sait pas quoi !
        //Il faut donc initialiser ces espaces mémoire
        //On peut également utiliser calloc pour initialiser à 0
        for(int i=0; i<20; i++){
            *(pt_tabint+i)=i;
        }
        //Parcours en utilisant les []
        for(int i=0; i<20; i++) {
            printf("%d\t",pt_tabint[i]);
        }
        printf("\n");
    }
    free(pt_tabint);
    return 0;
}
```

# Comment choisir ?

## Tableaux statiques ou tableaux dynamiques ?

- Tableaux statiques, déclarés dans la pile, cela implique qu'il n'est accessible que dans la procédure ou fonction dans laquelle il est déclaré
- Tableaux dynamique, créé via un appel à `malloc` ou `calloc` et placés dans le tas. Potentiellement accessibles car persistant depuis partout en passant l'adresse mémoire du début du tableau.

L'ancêtre de la programmation orientée objet...

# Types composés hétérogènes : enregistrements/structures

Qu'est-ce qu'une structure ? Comment la définir ? Quelle est l'utilité du `typedef` ?  
Comment accéder aux champs d'une structure ? Comment faire l'initialisation ?  
Comment copier une structure ?

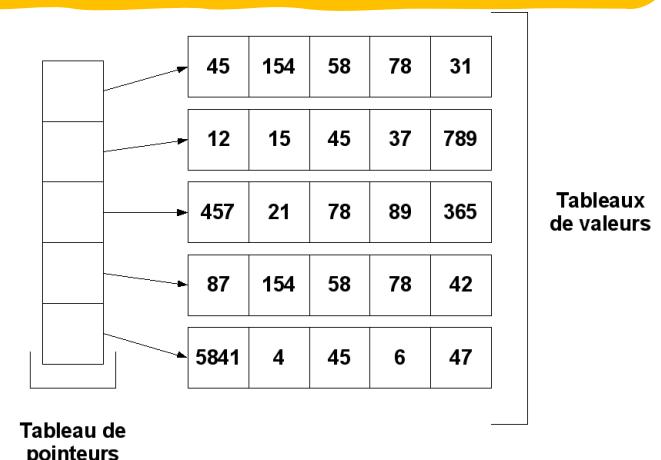
# Tableaux à plusieurs dimensions avec manipulation via pointeurs

- Pour chaque dimension du tableau il faut un tableau de pointeurs
- Exemple de déclaration de matrice, via pointeurs

```
#include <stdio.h>
#include <stdlib.h>
#define NB_LIGNES 5
#define NB_COLONNES 7

int main()
{
    float **matrice=NULL; //pointeur sur un pointeur
    matrice = (float **)malloc(NB_LIGNES * sizeof(float *));
    for (int i = 0; i < NB_LIGNES; i++)
        matrice[i] = (float *)malloc(NB_COLONNES * sizeof(float));
    ...
    for (int i = 0; i < NB_LIGNES; i++) free(matrice[i]);
    free(matrice);
    ...
}
```

C'est en fait un tableau à une dimension dont chaque élément est un tableau le parcours des indices de droite (colonnes) est plus rapide (adjacence), que celui des indices de gauche (lignes) saut en mémoire...



# Types composés hétérogènes : enregistrements/structures

- Constitués de types hétérogènes
- Les membres de l'enregistrement sont des champs

```
struct entite {           // nommer son type de structure
    float x,y;
    float dx,dy;
    int color;
    char lettre;
};

struct entite e1, e2;
```

struct entite est un nom de type, pas une variable

Les définitions/déclaration de struct sont généralement placées dans le '.h'

# Déclaration de nouveau type : typedef

- Les champs d'une structure sont accédés par l'opérateur ‘.’

```
#include <stdio.h>

typedef int MonTypeIntAMoi;

typedef struct{
    int x, y;
    int color;
}Pixel;

int main(){
    MonTypeIntAMoi test = 10;
    printf("La variable test vaut %d\n", test);
    Pixel p;
    p.x = 10;
    p.y = 20;
    p.color = 255;
    return 0;
}
```

A placer dans le '.h'

# Structures imbriquées

```
typedef struct rgb{      // valeur de rouge, de vert et de bleu
    int r,g,b;          // pour avoir une couleur
} Rgb;

typedef struct coord{
    int x, y;           // pour stocker une position en 2D
} Coord;

typedef struct rectangle{
    Coord hg;           // coord du coin haut gauche
    Coord bd;           // coord du coin bas droite
    Rgb color;          // couleur du rect
} Rectangle;

Rectangle r1;
r1.hg.x = 0;
r1.hg.y = 0;
r1.bd.x = 100;
r1.bd.y = 50;
r1.color.r = 255;
r1.color.g = r1.color.b = 0;
```

# Initialisation de structures à la déclaration

```
typedef struct {
    float x, y;
}Coord;
typedef struct {
    const int tx;
    const int ty;
    Coord coord;
    int px, py;
    char lettre;
}Personnage;

#define P_DEFAULT == { 50,100,1.5f,2.5f,rand()%100,rand()%100,'A' };

Personnage p1 = { 50,100,1.5f,2.5f,rand()%100,rand()%100,'A' }; //initialisation de tous les champs
Personnage p2 = { 50,100,{1.5f,2.5f},rand()%100,rand()%100,'A' }; //l'init de Coord est mise entre {} pour plus de lisibilité
Personnage p3 = { 50,100 }; //les autres champs sont mis à zéro
Personnage p4 = { 50,100,{1.5f}, rand()%100,rand()%100,'A' }; //Coord.y est mis à zéro
Personnage p5 = { 0 }; //Tout est mis à zéro
Personnage p6 = { .tx =50, .lettre ='b', .coord.x=50.0f}; //les autres champs sont mis à zéro
Personnage p7 = P_DEFAULT; //P_DEFAULT sera remplacé à la compilation
```

# Copie des structures

La copie de structure à structure est possible

```
typedef struct {  
    float x, y;  
}Coord;
```

```
Coord pt1, pt2;  
pt1.x = 10;  
pt1.y = 20;
```

```
pt2 = pt1; //les valeurs des champs de pt1 sont re-copiés dans pt2
```

# Fonctions d'initialisation de structures

- Comme la copie des structures est autorisée elles peuvent être des valeurs de retour de fonction

```
typedef struct coord{  
    int x, y;  
}Coord;  
  
Coord ConstructPoint1()  
{  
    Coord t0;  
    t0.x = rand()%1024;  
    t0.y = rand()%768;  
    return t0;  
}  
  
#include <stdio.h>  
#include <stdlib.h>  
// définir ici structure et fonctions  
  
int main()  
{  
    Coord pt1, pt2;  
  
    pt1 = ConstructPoint1();  
    pt2 = ConstructPoint2(40, 40);  
    printf("pt1.x=%d, pt1.y=%d\n"  
          "pt2.x=%d, pt2.y=%d\n", pt1.x, pt1.y, pt2.x, pt2.y);  
    return 0;  
}
```

# Structures en paramètres de fonction, passage par valeur

- Passage par valeur comme n'importe quel autre type de variable.
- Attention ! La structure passée est copiée dans le paramètre de la fonction.

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

void modif(Coord pt)
{
    // ici définitions struct et fonction
    pt.x++;
    if (pt.x>1000)        int main()
    {
        pt.x=0;
        Coord p={ 10,20};
        printf("p.x=%d, p.y=%d\n", p.x, p.y);
        modif(p);
        printf("p.x=%d, p.y=%d\n", p.x, p.y);
        return 0;
    }
}
```

Qu'est-ce qui est  
affiché ?

p.x=10, p.y=20  
p.x=10, p.y=20

# Structures en paramètres de fonction, passage par valeur

- Passage par valeur comme n'importe quel autre type de variable.
- Attention ! La structure passée est copiée dans le paramètre de la fonction, si vous effectuez des modifications sur ses champs et que ces changements soient répercutés à l'extérieur de la fonction vous devez retourner la structure en résultat.

```
Coord modif(Coord pt)
{
    pt.x++;
    if (pt.x>1000)
        pt.x=0;
    return pt;
}

int main()
{
    Coord p={10,20};;
    printf("p.x=%d, p.y=%d\n", p.x, p.y);
    p=modif(p);
    printf("p.x=%d, p.y=%d\n", p.x, p.y);
    return 0;
}
```

Qu'est-ce qui est affiché ?

p.x=10, p.y=20  
p.x=11, p.y=20

# Occupation mémoire des enregistrements de structures

- Les différents éléments d'un enregistrement struct sont stockés dans des cases mémoires adjacentes/contiguës.

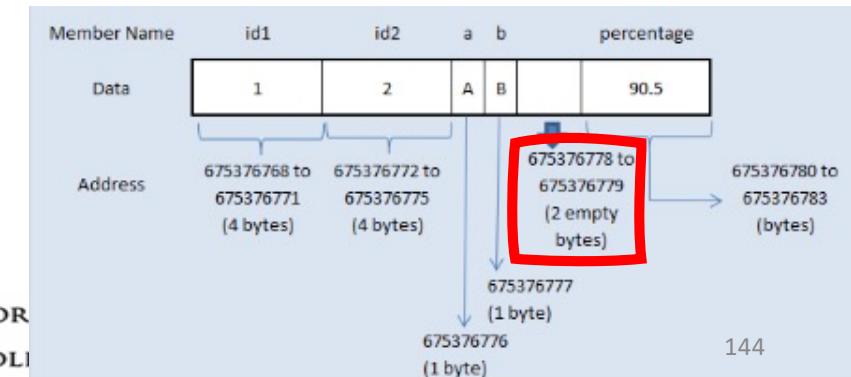
```
1 #include <stdio.h>
2 #include <string.h>
3
4 struct student
5 {
6     int id1;
7     int id2;
8     char a;
9     char b;
10    float percentage;
11 };
12
13 int main()
14 {
15     int i;
16     struct student record1 = {1, 2, 'A', 'B', 90.5};
17
18     printf("size of structure in bytes : %d\n",
19             sizeof(record1));
20
21     printf("\nAddress of id1      = %u", &record1.id1 );
22     printf("\nAddress of id2      = %u", &record1.id2 );
23     printf("\nAddress of a         = %u", &record1.a );
24     printf("\nAddress of b         = %u", &record1.b );
25     printf("\nAddress of percentage = %u", &record1.percentage);
26
27     return 0;
28 }
```

OUTPUT:

```
size of structure in bytes : 16
Address of id1 = 675376768
Address of id2 = 675376772
Address of a = 675376776
Address of b = 675376777
Address of percentage = 675376780
```

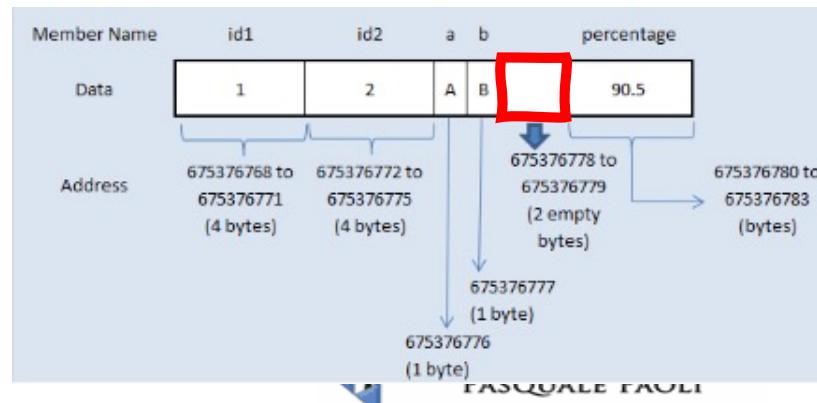
Compilateur 32 bit  
=  
mots de 4 Octets

Datatype	Memory allocation in C (32 bit compiler)		
	From Address	To Address	Total bytes
int id1	675376768	675376771	4
int id2	675376772	675376775	4
char a	675376776		1
char b	675376777		1
	Addresses 675376778 and 675376779 are left empty. (Do you know why? Please refer below, the next topic C - Structure padding)		2
float percentage	675376780	675376783	4



# Occupation mémoire des enregistrements de structure

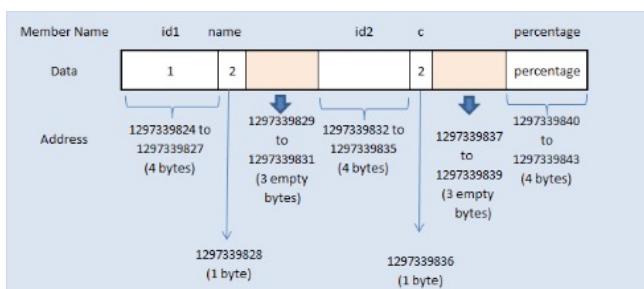
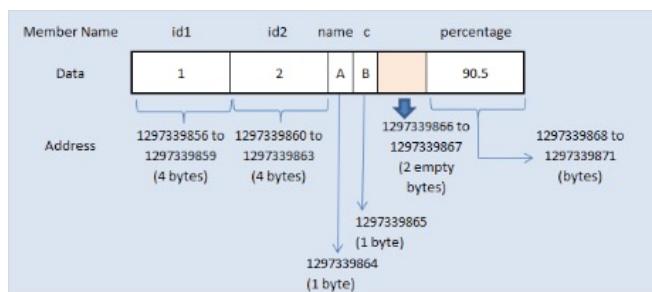
- L'occupation mémoire d'une struct donnée dépend de la taille des mots pris en charge par le processeur, 32 bits ou 64 bits et de l'enchainement des éléments.
- Le processeur peut lire un mot (32 ou 64 bits) à la fois en mémoire.
- Les données sont organisées pour profiter de cette possibilité quitte à inclure des espaces vides entre les données de la struct



Si l'optimisation de l'espace mémoire est importante on peut forcer l'optimisation avec une directive au préprocesseur C

# Exercice :

Déterminer la taille en mémoire, à l'exécution des deux structures a et b ci-contre.



```

1 #include <stdio.h>
2 #include <string.h>
3
4 /* Below structure1 and structure2 are same.
   They differ only in member's alignment */
5
6 struct structure1
7 {
8     int id1;
9     int id2;
10    char name;
11    char c;
12    float percentage;
13};
14
15 struct structure2
16 {
17     int id1;
18     char name;
19     int id2;
20     char c;
21     float percentage;
22};
23
24 int main()
25 {
26     struct structure1 a;
27     struct structure2 b;
28
29     printf("size of structure1 in bytes : %d\n",
30           sizeof(a));
31     printf( "\n Address of id1      = %u", &a.id1 );
32     printf( "\n Address of id2      = %u", &a.id2 );
33     printf( "\n Address of name      = %u", &a.name );
34     printf( "\n Address of c          = %u", &a.c );
35     printf( "\n Address of percentage = %u",
36             &a.percentage );
37
38
39     printf(" \n\nsize of structure2 in bytes : %d\n",
40           sizeof(b));
41     printf( "\n Address of id1      = %u", &b.id1 );
42     printf( "\n Address of name      = %u", &b.name );
43     printf( "\n Address of id2      = %u", &b.id2 );
44     printf( "\n Address of c          = %u", &b.c );
45     printf( "\n Address of percentage = %u",
46             &b.percentage );
47
48     getchar();
49 }

```

OUTPUT:

```

size of structure1 in bytes : 16
Address of id1 = 1297339856
Address of id2 = 1297339860
Address of name = 1297339864
Address of c = 1297339865
Address of percentage = 1297339868
size of structure2 in bytes : 20
Address of id1 = 1297339824
Address of name = 1297339828
Address of id2 = 1297339832
Address of c = 1297339836
Address of percentage = 1297339840

```

# Optimisation de l'occupation mémoire

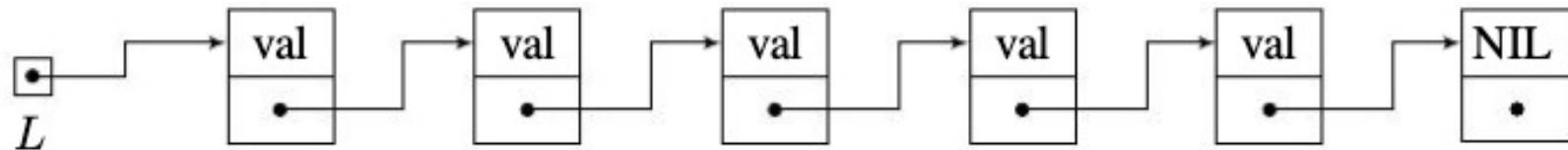
Si l'optimisation de l'espace mémoire est importante - en embarqué généralement - on peut forcer l'optimisation avec une directive au préprocesseur C  
#pragma pack(1)

```
1 #include <stdio.h>
2 #include <string.h>
3
4 /* Below structure1 and structure2 are same.
   They differ only in member's alignment */
5
6 #pragma pack(1)
7 struct structure1
8 {
9     int id1;
10    int id2;
11    char name;
12    char c;
13    float percentage;
14 };
15
16 struct structure2
17 {
18     int id1;
19     char name;
20     int id2;
21     char c;
22     float percentage;
23 };
24
25 int main()
26 {
27     struct structure1 a;
28     struct structure2 b;
29
30     printf("size of structure1 in bytes : %d\n",
31            sizeof(a));
32     printf( "\n Address of id1      = %u", &a.id1 );
33     printf( "\n Address of id2      = %u", &a.id2 );
34     printf( "\n Address of name      = %u", &a.name );
35     printf( "\n Address of c          = %u", &a.c );
36     printf( "\n Address of percentage = %u",
37            &a.percentage );
38
39     printf(" \n\nsize of structure2 in bytes : %d\n",
40            sizeof(b));
41     printf( "\n Address of id1      = %u", &b.id1 );
42     printf( "\n Address of name      = %u", &b.name );
43     printf( "\n Address of id2      = %u", &b.id2 );
44     printf( "\n Address of c          = %u", &b.c );
45     printf( "\n Address of percentage = %u",
46            &b.percentage );
47     getchar();
48     return 0;
49 }
50 }
```

OUTPUT:

```
size of structure1 in bytes : 14
Address of id1 = 3438103088
Address of id2 = 3438103092
Address of name = 3438103096
Address of c = 3438103097
Address of percentage = 3438103098
size of structure2 in bytes : 14
Address of id1 = 3438103072
Address of name = 3438103076
Address of id2 = 3438103077
Address of c = 3438103081
Address of percentage = 3438103082
```

# Structures récursives : exemple d'une liste chainée d'entiers



```
typedef struct liste
{
    int val;
    struct liste *p_suiv;
} Liste_int;
```

On parle de structure récursive car pour définir la struct liste on utilise un appel à struct liste

De quel type sera L , la tête de liste ?

`Liste_int * L = NULL;`

# Les pointeurs en C et les structures

- On peut manipuler les structures via des pointeurs
- Accéder aux champs d'une structure à partir de son adresse mémoire, opérateur flèche « -> »

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

struct madate{
    int jour;
    int mois;
    int annee;};
int main(){
    struct madate aujourd'hui, *ptr_date;
    aujourd'hui.jour = 13;
    aujourd'hui.mois = 2;
    aujourd'hui.annee = 2022;
    ptr_date = &aujourd'hui;
    (*ptr_date).annee = 2023;
    ptr_date->annee = 2024;
    printf("La date est %d/%d/%d\n", ptr_date->jour, ptr_date->mois, ptr_date->annee);
    printf("La date est donc %d/%d/%d\n", (*ptr_date).jour, (*ptr_date).mois, (*ptr_date).annee);
}
```

Retour sur les fonctions...  
Comment faire pour qu'une fonction renvoie  
plusieurs résultats ?

# Comment faire pour qu'une fonction renvoie plusieurs résultats ?

- Utiliser des paramètres d'entrée/sortie
  - Transformer la fonction en procédure
  - Utiliser le passage de paramètres par références
- Définir une structure permettant l'agrégation des résultats devant être renvoyés par la fonction

# Exercice

- Ecrire deux versions d'une fonction/prodédure nommée `get_max_and_min` prenant à minima en paramètre un tableau de double, ainsi que le nombre d'éléments dans le tableau et retournant le min et le max de ce tableau.
- Utilisez les deux méthodes citées
  - Procédure avec variables d'entrée/sortie, min et max ajoutés aux paramètres de `get_max_and_min_1`
  - Création d'une structure ad hoc nommée `double_pair` pour le stockage et le renvoie du min et du max par la fonction `get_max_and_min_2`

# Solution

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

void get_max_and_min_1(double *values, int length, double *min, double *max){
    assert(length>0);
    *max = values[0];
    *min = values[0];
    for(int i=1;i<length;i++){
        if (values[i]> *max){*max=values[i];}
        if (values[i]< *min){*min=values[i];}
    }
}

typedef struct {
double min, max;
} double_pair;

double_pair get_max_and_min_2(double *values, int length){
    assert(length>0);
    double_pair result;
    result.max = values[0];
    result.min = values[0];
    for(int i=1;i<length;i++){
        if (values[i]> result.max){result.max=values[i];}
        if (values[i]< result.min){result.min=values[i];}
    }
    return result;
}

int main(int argc, char const *argv[]){
    double values[5] = {10.5, 13.3, -45.5, 12, 2.5};
    double mymin = 0, mymax=0;
    get_max_and_min_1(values, 5, &mymin, &mymax);
    printf("Max : %lf et Min : %lf \n", mymax, mymin);
    double_pair minmax = get_max_and_min_2(values,5);
    printf("Max : %lf et Min : %lf \n", minmax.max, minmax.min);
    return 0;
}
```



# Fonction à arguments variables en nombre, fonctions « variatiques »

- Liste variable de paramètres de même type
- Une telle fonction doit comporter au moins un paramètre fixe qui précède la liste des paramètres variables

va_list	est le type pour la liste de récupération.
va_start(va_list liste, DPF)	est la fonction d'initialisation de la liste.
va_arg(va_list liste, type param)	est la fonction pour la récupération de chaque élément de la liste.
va_end(va_list liste)	est la fonction qui libère la mémoire de la liste constituée de façon dynamique (l'allocation dynamique est abordée au chapitre Les pointeurs).

# Exemple de code fonction variatique

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

int somme(int nbparam,...){
    va_list liste_param;
    int result = 0;
    int i_param;
    va_start(liste_param,nbparam);
    for (i_param = 0; i_param<nbparam; i_param++){
        result+= va_arg(liste_param,int);
    }
    va_end(liste_param);
    return result;
}

int main(int argc, char const *argv[]){
    printf("%d\n", somme(1,1));
    printf("%d\n", somme(3,1,2,3));
    printf("%d\n", somme(5,1,2,3,4,5));
    return 0;
}
```

# Structures de données

Quesaco ? Aperçu général.

Types et abstraction, types simples, composés, pointeurs

Structures de données séquentielles ou linéaires

Structures de données hiérarchiques et relationnelles

Préalable...



Oui ! Nous allons  
ré-inventer la  
roue !!!!

# Qu'est-ce qu'une structure de données ?

Est-ce important d'y réfléchir ?

Si oui, pourquoi ?

Si non, pourquoi ?



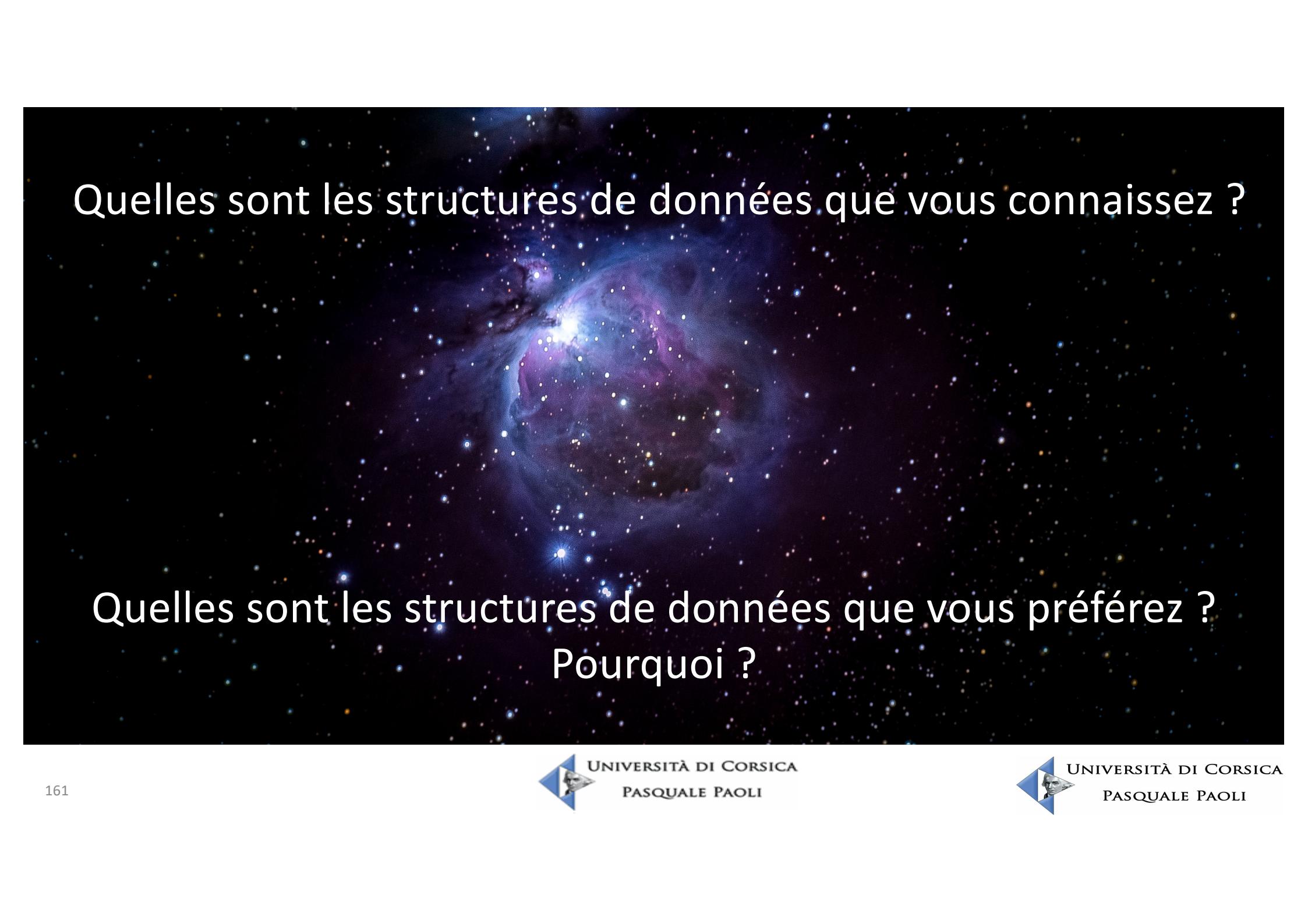


On vous pose un  
problème  
d'algorithmique,  
sur une échelle  
de 1 à 10,  
à combien estimez-  
vous  
votre capacité à choisir  
**la bonne structure de**  
**données ?**



Quels sont les critères  
qui vont guider votre  
choix de structure de  
données ?





Quelles sont les structures de données que vous connaissez ?

Quelles sont les structures de données que vous préférez ?  
Pourquoi ?

# Les enjeux du choix d'une bonne structure de données

- Représentation adéquate des données
- Accès rapide et aisée aux données
- Efficacité en temps des opérations nécessaires sur la structure
- Empreinte faible en mémoire

Du choix de la structure adéquate va souvent dépendre l'efficacité de l'algorithme

# Les structures de données

## *Data Structure...*

*I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important.  
Linus Torvalds, 2006)*

D'autres citations <https://medium.com/webdevops/data-structures-548cbea9c520>

# Organiser les données pour les gérer efficacement

Combinaison d'objets simples en un objet plus complexe, permettant d'attribuer à cet ensemble composite un nom unique

Relations entre les données au sein de la structure

Types de données de base ou scalaires ou types primitifs tels que les entiers, les caractères, les réels



# Organiser les données pour les gérer efficacement

Combinaison d'objets simples en un objet plus complexe, permettant d'attribuer à cet ensemble composite un nom unique

Organiser

Manipuler

Stocker



# Une structure de données



Regroupement  
d'éléments



Moyens, opérations  
qui permettent d'y  
accéder et de la  
manipuler

**TAD**  
Type Abstrait de  
Données



# Fonction de ce qu'elles contiennent

Homogènes

Tous les objets stockés  
sont du même type

Ex : Tableau d'entiers

```
int tab[10];  
int mat[10][5];
```

Hétérogènes

Les données stockées  
sont de différents  
types

Ex : Un tuple  
(5, "Machin",True)



# Fonction de leur structuration

Séquentielles ou Linéaires

Organisation linéaire,  
Notion de successeur

Parcourable en un seul passage

On peut aller du début à la fin en un seul coup

Chaque élément possède un index

Non linéaires

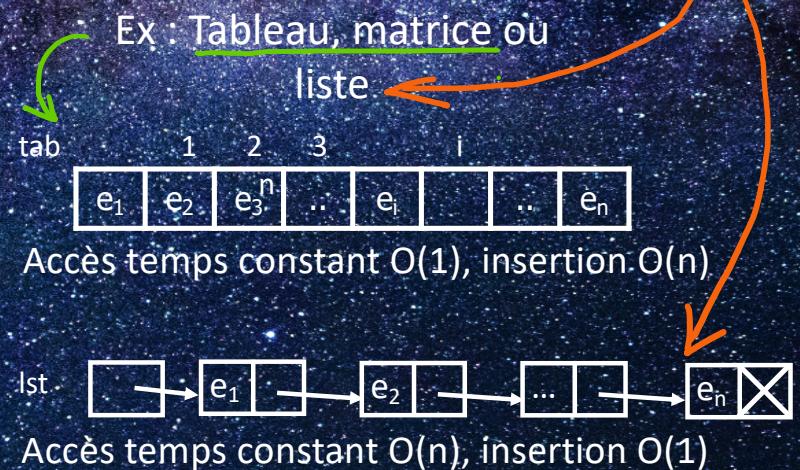
Notion d'ordre non  
chronologique, relation  
hiérarchique entre les  
éléments

On ne peut pas parcourir toutes  
les données de façon séquentielle  
Relation de parent/enfants, voisins



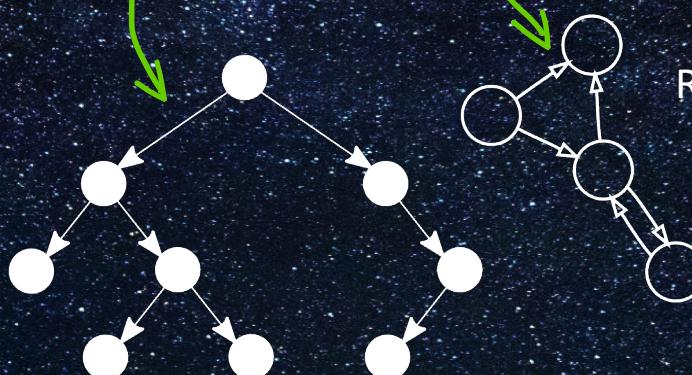
# Fonction de leur structuration

## Séquentielles ou Linéaires



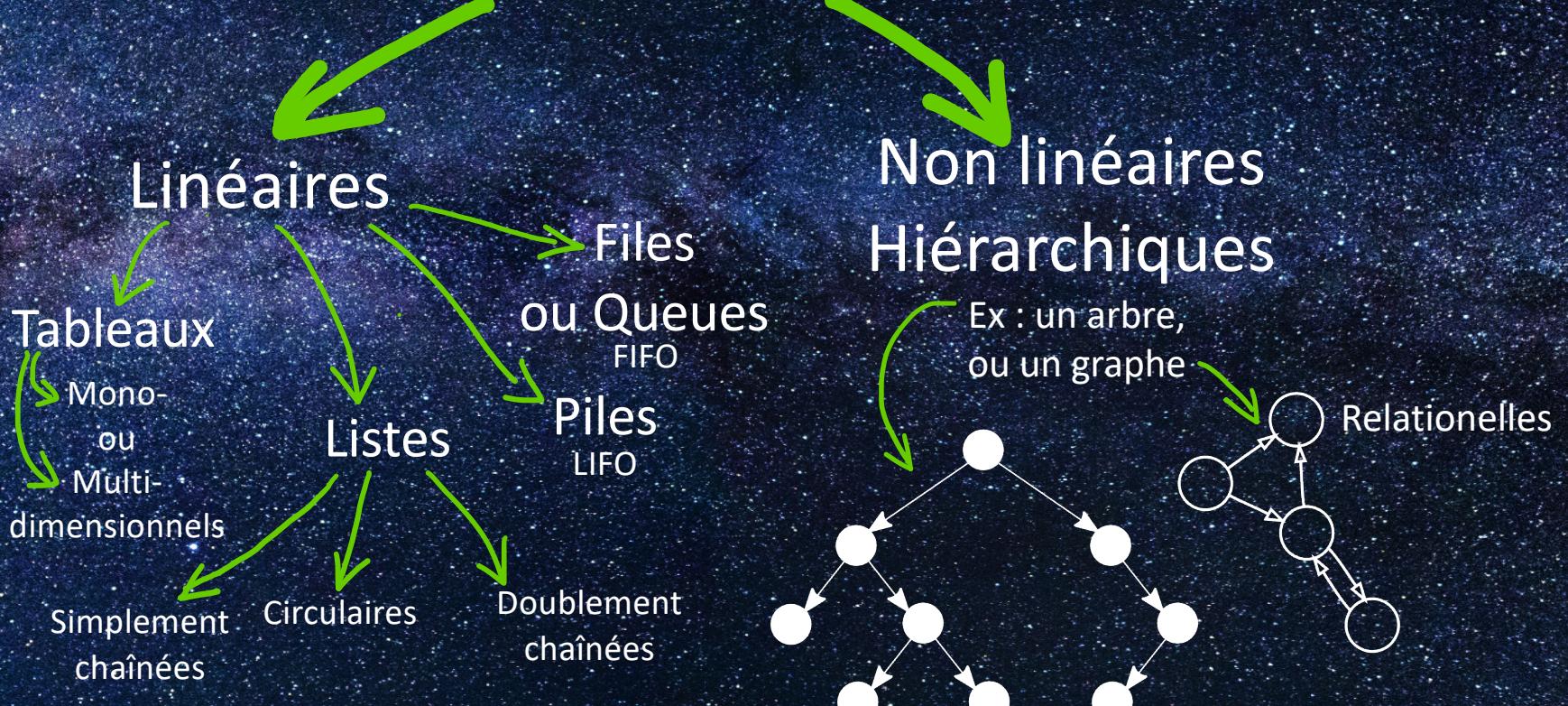
## Non linéaires Hiérarchiques

Ex : un arbre,  
ou un graphe



Relationnelles

# Des structures de données organisées différemment...



# Fonction de leur gestion mémoire

Statique



Taille figée, déterminée  
à l'avance



Emplacement mémoire  
statique

Dynamique



Taille évolutive



Emplacement mémoire  
dynamique

# Des propriétés spécifiques

Ordonnées ou non !

- Chronologiquement
- Suivant un ordre spécifique

Accès indexé  
ou non !

Enumérables  
ou non !

Acceptant les  
doublons  
ou non !

Taille dynamique  
ou non !

Acceptant les types  
primitifs  
ou non !

Types homogènes  
ou non !

Des **questions** spécifiques



UNIVERSITÀ DI CORSICA  
PASQUALE PAOLI



UNIVERSITÀ DI CORSICA  
PASQUALE PAOLI

On choisit une structure pour un comportement

TAD<sup>Interface</sup>  
Contrat API

Plusieurs implémentations



Performances  $O(\dots)$  en temps, en espace...

Ex : comportement de Liste

Implémentation tableau



Implémentation  
liste simplement chainée



Des performances différentes pour l'insertion ou la recherche

Les structures de  
données  
ou *Data  
Structure...*

# Rapide aperçu des structures de données tableaux associatifs

Ou dictionnaires, ou tables de hashage, ou Hashtable

C'est quoi ? A quoi ça sert ? Comment ça marche ? Comment l'implémenter en C ?

# Les tableaux associatifs ?

## Association entre une clé et une valeur

- **Type abstrait de données** / structure de données séquentielle qui stocke une collection de couples clé/valeur non ordonnés.
  - Les données (valeurs) sont indiquées mais pas nécessairement par des entiers, par des valeurs quelconques : les clés
    - Généralisation du concept de tableau :
      - Tableau : association entre index successifs entiers et valeurs
      - Table associative : association entre une clé d'un type quelconque à une valeur au plus d'un type quelconque
      - Maps : clé unique ; dictionnaires : clé non nécessairement unique
  - Les opérations fournies sont :
    - L'ajout/modification d'un couple clé-valeur
    - La suppression
    - La recherche/récupération
- Un annuaire est un tableau associatif  
Un dictionnaire aussi
- Utilisé par exemple pour gérer :
  - la table des symboles des compilateurs
  - la table d'allocation des fichiers (File Allocation Table FAT)
  - les tables de routage IP

# Différences par rapport aux tableaux Intérêts des tableaux associatifs ?

Tableau indexé

Index	Valeur
0	('Paul', 12.5)
1	('Jean', 8)
...	...
N	('Etienne',17)

Tableau associatif

Clé	Valeur
'Etienne'	17
'Jean'	8
...	
'Paul'	12.5

*Bucket*

*Bucket array*

L'idée est que l'index auquel on va positionner le couple Clé-valeur dépend d'un calcul simple effectué sur la clé...

# Tables associatives (TAD) - dictionnaires – maps

## Implémentation

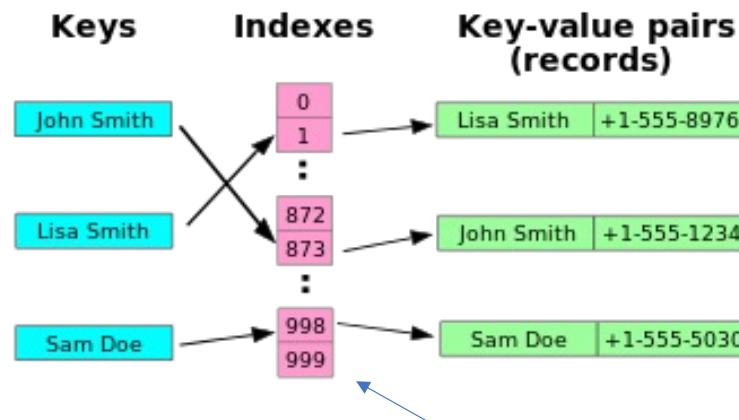
- Type de données associant (« mappant »/*mapping*) une valeur à une clé
- Chaque clé est associée à une seule et unique valeur
  - Variante possible pour les multimap : plusieurs valeurs possibles pour une même clé
- Une façon d'implanter une table associative est de créer une liste chaînée de paires clé-valeurs, ou liste d'association
  - Inefficace (pourquoi ?) mais simple !
- On peut aussi utiliser
  - une table de hachage
  - un arbre binaire équilibré

# Les opérations minimales sur les tableaux associatifs

- `create_table(t)`
- `search(t, k)` : renvoie la valeur  $v$  associée à  $k$  présente dans la table  $t$  ou `NULL` si la clé n'est pas dans la table
- `insert(t, k, v)` : association d'une valeur à une clé et insertion dans la table  $t$
- `delete(t, k)` : retire le couple clé valeur associé à la clé  $k$  dans la table  $t$
- `detete_table(t)` : libère la table

# Tables associatives vs Tables de Hachage

- Une table de hachage est une implémentation d'une table associative où on utilise une fonction de hachage pour obtenir à partir de la valeur de la clé, l'indice auquel est stocké le couple clé-valeur.



Bonnes performances en termes d'insertion, de recherche et de retrait, proche de  $O(1)$  et toujours  $< O(n)$ .  
Cout de la fonction de hachage

Fonction de hachage sur la clé : renvoie l'adresse (l'indice) de la clé dans la table en deux étapes -> calcul du hash et utilisation d'une fonction de répartition.

# Description générale

- Concept des tables de Hachage
  - Correspond aux collections indexées par des clés, groupe de couples clé-valeur rangés dans un tableau  $t$  de taille  $n$
  - Il ne peut y avoir deux fois la même clé dans la table
  - Une clé correspond à une et une seule valeur
  - En fonction de la clé on obtient – via la fonction de hachage et la fonction de compression – un indice permettant de ranger – et de retrouver – le couple clé-valeur dans le tableau de façon efficace

Le but est de répartir le plus uniformément possible les couples clé-valeur dans le tableau  $t$  plutôt que de les ajouter les unes à la suite des autres

# Fonction de hachage

- La fonction de hachage est composée de deux fonctions :
  - Le code de hachage  $h_1 : U \rightarrow \mathbb{N}$ , où  $U$  représente l'ensemble des valeurs des clés, donnant pour chaque clé  $k$ , un *hash code*  $h_1(k)$ .
  - Une fonction de compression déterministe,  $h_c : \mathbb{N} \rightarrow \llbracket 0, n - 1 \rrbracket$  où  $n$  est la taille de la table de hachage
- La fonction de hachage calcule un indice qui représente l'indice de la case du tableau  $h(k) = h_c \circ h_1(k)$

## Exemple

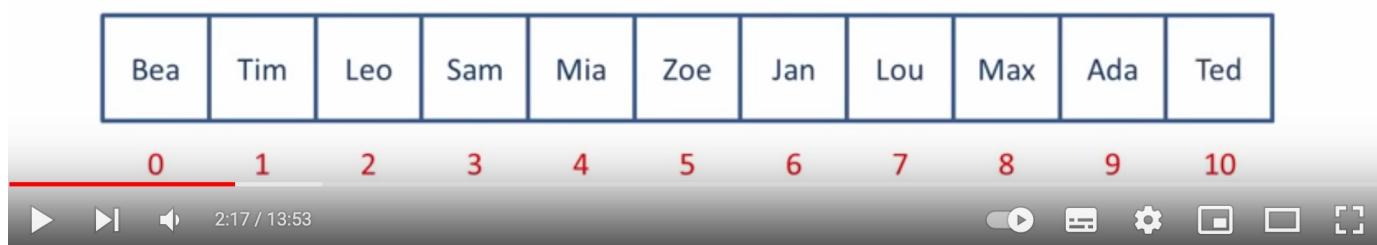
- Exemples de codes de hachage : interprétation des bits de la clé comme un entier, interprétation de l'adresse mémoire de la clé comme un entier, addition des codes ASCII des caractères de la chaîne de caractères,...
- Exemples de fonctions de compression : division :  $h_1(k) \% n$ , MAD :  $(ah_1(k) + b) \% n$  tels que  $a, b \in \mathbb{N}$ ,  $a \% n \neq 0$ ...

Cf. [1]

Multiplication – Addition - Division

# Illustration, explication des tables de hachage

Mia	M	77	i	105	a	97	279	4
Tim	T	84	i	105	m	109	298	1
Bea	B	66	e	101	a	97	264	0
Zoe	Z	90	o	111	e	101	302	5
Jan	J	74	a	97	n	110	281	6
Ada	A	65	d	100	a	97	262	9
Leo	L	76	e	101	o	111	288	2
Sam	S	83	a	97	m	109	289	3
Lou	L	76	o	111	u	117	304	7
Max	M	77	a	97	x	120	294	8
Ted	T	84	e	101	d	100	285	10



# Conditions sur le hachage

On souhaite que pour toutes clés  $k_1, k_2 \in U$  :

1.  $k_1 = k_2 \Rightarrow h(k_1) = h(k_2)$ ,
2.  $k_1 \neq k_2 \Rightarrow h(k_1) \neq h(k_2)$ ,
3. si  $k_1$  et  $k_2$  sont « semblables », les indices  $h(k_1)$  et  $h(k_2)$  doivent être éloignés.

Une fonction de hachage doit être simple à calculer en  $O(1)$ , permettant un accès optimal aux données.

La définition d'une bonne fonction de Hachage n'est pas un pb simple !

La première condition est assez simple à mettre en place. En revanche, la seconde (injectivité) est plus délicate dès que  $|U| > n$ .

# Exercice – Exemple

- On a un tableau de 10 cases et on souhaite insérer les clés suivantes (on se moque des valeurs) : 4, 42, 7 et 0
- La fonction de hachage est composée de :
  - Partie hachage :  $h_h(k) = 3*k + 14$
  - Partie compression :  $h(k) = h_h(k) \bmod 10 = h_c(h_h(k)) = \text{indice du tableau}$

Calculez  $h_h$  et  $h_c$  pour chaque clé et placez les clés dans la table.

Clé	$h_h$	$h_c(h_h)$	indices	0	1	2	3	4	5	6	7	8	9
0			clés										
4													
7													
42													



# Exercice – Exemple

- On a un tableau de 10 cases et on souhaite insérer les clés suivantes (on se moque des valeurs) : 4, 42, 7 et 0
- La fonction de hachage est composée de :
  - Partie hachage :  $h_h(k) = 3*k + 14$
  - Partie compression :  $h(k) = h_h(k) \bmod 10 = h_c(h_h(k))$  = indice du tableau

Calculez  $h_h$  et  $h_c$  pour chaque clé et placez les clés dans la table.

Clé	$h_h$	$h_c(h_h)$
0	14	4
4	26	6
7	35	5
42	140	0

indices	0	1	2	3	4	5	6	7	8	9
clés	42				0	7	4			

# Mesure du remplissage de la table

## Le facteur de charge ou *load factor*

$$\text{Facteur de charge} = \frac{\text{Nombre de clés}}{\text{Nombre de places total}}$$

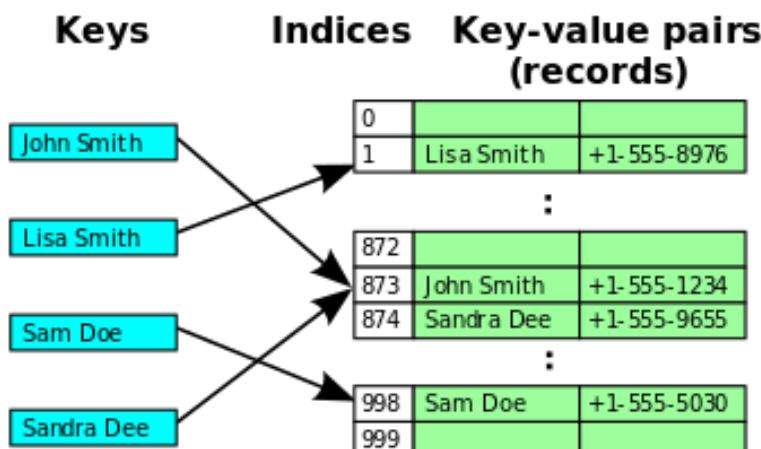
# Quid des collisions ?

- Quand la table se remplit il y a de grande chance d'y avoir des collisions -> deux clés différentes qui ont le même indice après hachage

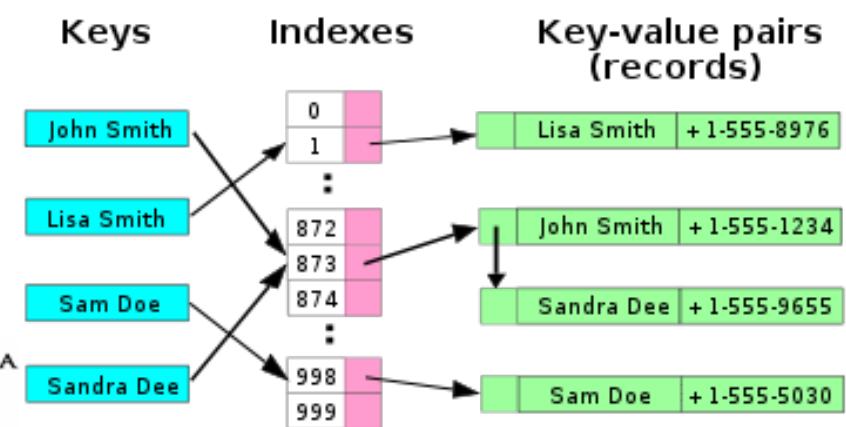
$$h(k) = h(k') \text{ mais } k \neq k'$$

# Gestion des collisions adressage ouvert vs chaînage

- Adressage ouvert : en cas de collision les buckets (couples clés-valeur) sont stockées ailleurs dans le corps de la table via une méthode de sondage
  - Sondage linéaire : première case libre à côté de la collision
  - Sondage quadratique (c'est-à-dire qui est du second degré, élevé au carré)
  - Double hachage
- Chainage : en cas de collision les buckets sont stockées dans des listes chaînées, on a, au final, une table de listes



@Wikipedia  
<< Adressage ouvert  
Chainage >>



# Gestion des collisions par sondage linéaire (le plus simple)

Si collision on regarde si la case suivante est libre, si oui on y place la clé

- Exemple avec la clé 22 :  $h_h(22) = 80$ ,

$$h(22) = h_h(22) \bmod 10 = 0$$

indices	0	1	2	3	4	5	6	7	8	9
clés	42	22			0	7	4			

Pour découvrir d'autres méthodes de gestion de collisions, cf. [Jean-Philippe Collette](#), Site developper.com, Tables de hachage - dont est extrait l'exemple ci-dessus - , 7 Juillet 2011  
<https://jipe.developpez.com/articles/algo/table-hachage/>

# Probabilité d'absence de collision

Pour ceux que cela amuse...  
et seulement pour ceux-là ! ;-)

- Pour une table avec  $N$  clés et  $M$  places potentielles, la probabilité qu'il n'y ait pas de collision est  $P(N, M)$

$$P(N, M) = \frac{M}{M} \left( \frac{M-1}{M} \right) \left( \frac{M-2}{M} \right) \dots \left( \frac{M-(N-1)}{M} \right) = 1 \left( 1 - \frac{1}{M} \right) \left( 1 - \frac{2}{M} \right) \dots \left( 1 - \frac{N-1}{M} \right) = \prod_{i=0}^{N-1} \left( 1 - \frac{i}{M} \right)$$

- On doit se rappeler (ಠ... ou savoir ಠ) que pour tout  $x$  réel,  $1 - x \leq e^{-x}$ , donc on peut majorer  $P(N, M)$  et écrire :  $P(N, M) \leq \prod_{i=0}^{N-1} e^{-\frac{i}{M}}$

- Or (ಠ)  $e^x * e^y = e^{x+y}$ , donc  $\prod_{i=0}^{N-1} e^{-\frac{i}{M}} = e^{-\frac{\sum_{i=0}^{N-1} i}{M}}$

- Or (ಠ)  $\sum_{i=0}^{N-1} i = \frac{N(N-1)}{2}$ , donc  $P(N, M) \leq e^{-\frac{N(N-1)}{2M}}$  et

$$1 - P(N, M) \geq 1 - e^{-\frac{N(N-1)}{2M}}$$

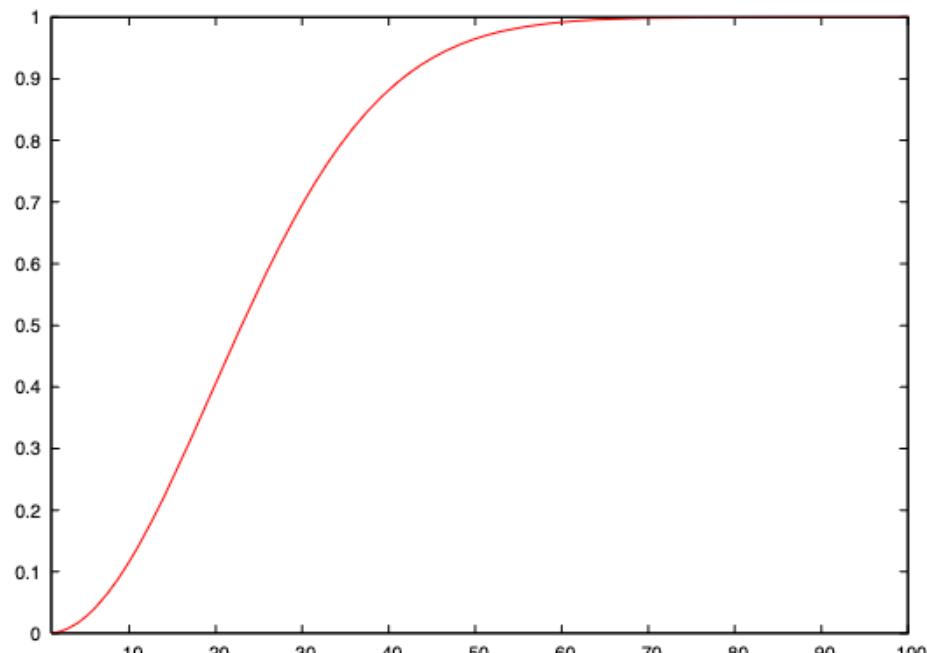
Suite arithmétique !

Proba qu'il y ait au moins une collision

PASQUALE PAOLI

# Probabilité de présence d'une collision

$1 - P(n, m)$  en fonction de  $n$  pour  $m = 365$



Cf. <https://www.fil.univ-lille.fr/~oussous/portail/API2/2008-2009/Cours/Tables/tables.4on1.pdf>

# Pour résumer...

- L'intérêt d'utiliser un tableau associatif est d'avoir de bonnes performances à l'ajout et à la recherche d'une paire clé-valeur dans le tableau
- On a un accès en moyenne en  $O(1)$  (coût de la fonction de Hachage) quelle que soit la taille du tableau
- Attention cependant, au pire des cas – collisions – on peut arriver à  $O(n)$

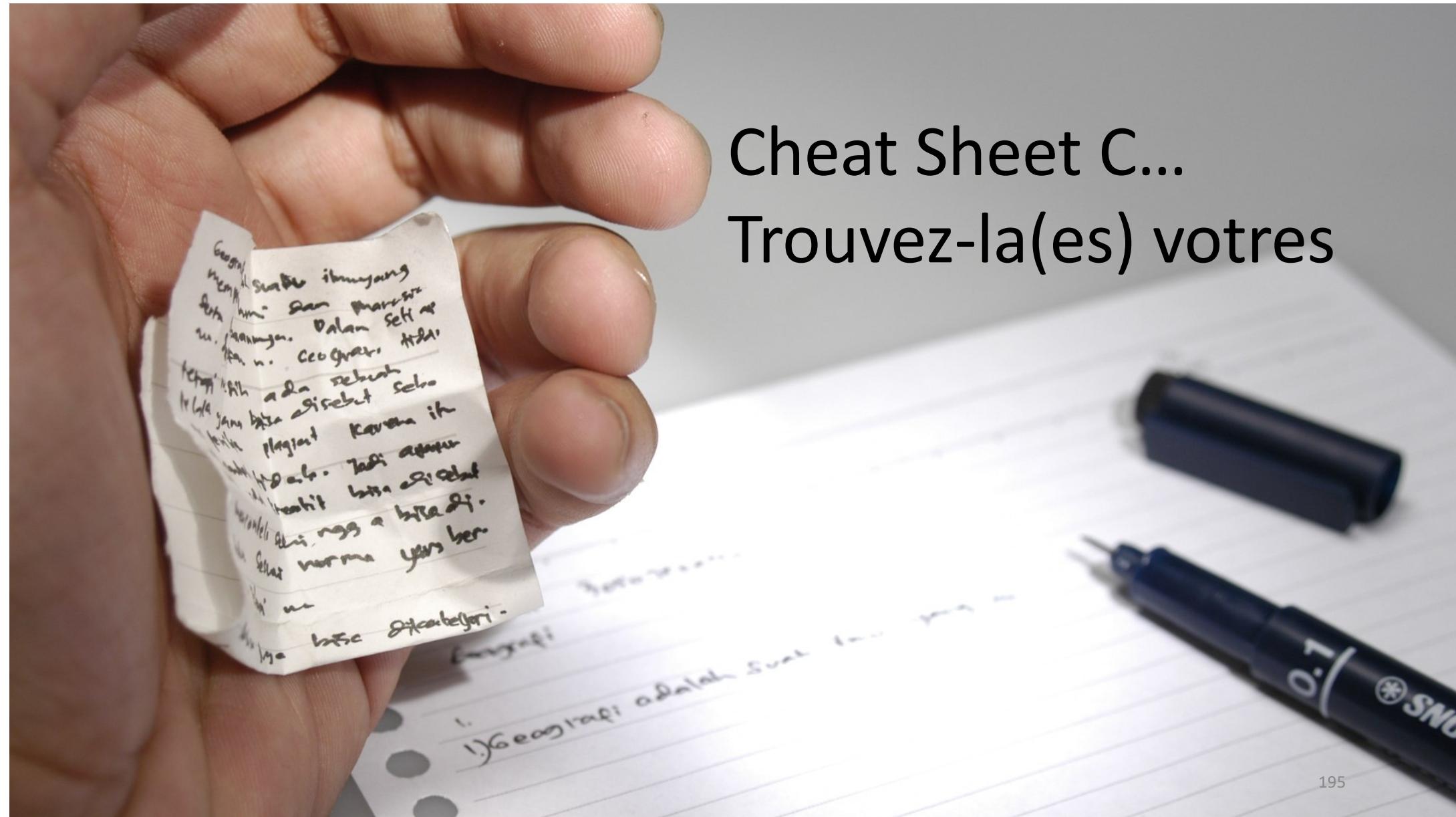
Nous ferons une implémentation en TD/TP

# Pratique

Langage C

Aléatoire

# Cheat Sheet C... Trouvez-la(es) votres



# Compilation/debugage et exécution

- Via IDE : (VS code, Codeblocks, ?)
- En ligne de commande : gcc/gdb
- L'outil make
- Compilateur fonction du SE :
  - Windows (like Linux) : Cygwin / MinGw-w64
  - Linux : gcc de base
  - MacOS : gcc (clang)

# Pas très haut niveau... donc

- Manque de portabilité sur pas mal d'aspects
  - Codage de l'information
  - Aspects graphiques

# Compiler

Avec l'IDE, ok, mais...

Important de savoir le faire en ligne de commande via gcc

# Le compilateur GCC

Options	Description
gcc -E	Only invokes the preprocessor without compiling or using the linker.
gcc -g	Enables debug information, needed before gdb
gcc -c	Compiles source files to object files without linking to any other object files.
gcc -I dir	Includes the directories of header files
gcc -l lib	link the code with the library files
gcc file.c -o output_file	Build the output generated to output file
gcc -Wall -pedantic	Enables all warning messages during the compilation.

Some interesting Gcc command line options <https://renenyffenegger.ch/notes/development/languages/C-C-plus-plus/GCC/options/index>  
Option controlling the King of Output, GNU gcc onlinedocs, <https://gcc.gnu.org/onlinedocs/gcc/Overall-Options.html#Overall-Options>

# Programme monolithique

- Option 1 : Tout est dans le même fichier on compile donc tout d'un coup

```
$gcc -Wall -pedantic testbaseentier.c
```

Si le fichier est gros  
c'est peu pratique ! Et peu  
modulaire...

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

void swap_int(int *v1, int *v2);

int main(int argc, char const *argv[]){
    int a = 10;
    int b = 15;
    printf("Avant appel à swap_int le contenu de a est %d et
celui de b est %d\n", a, b);
    swap_int(&a,&b);
    assert(a==15 && b==10);
    printf("Après appel à swap_int le contenu de a est %d et
celui de b est %d\n", a, b);
    return 0;
}

void swap_int(int *v1, int *v2){
    int temp = *v1;
    *v1 = *v2;
    *v2 = temp;
}
```

testbase.c

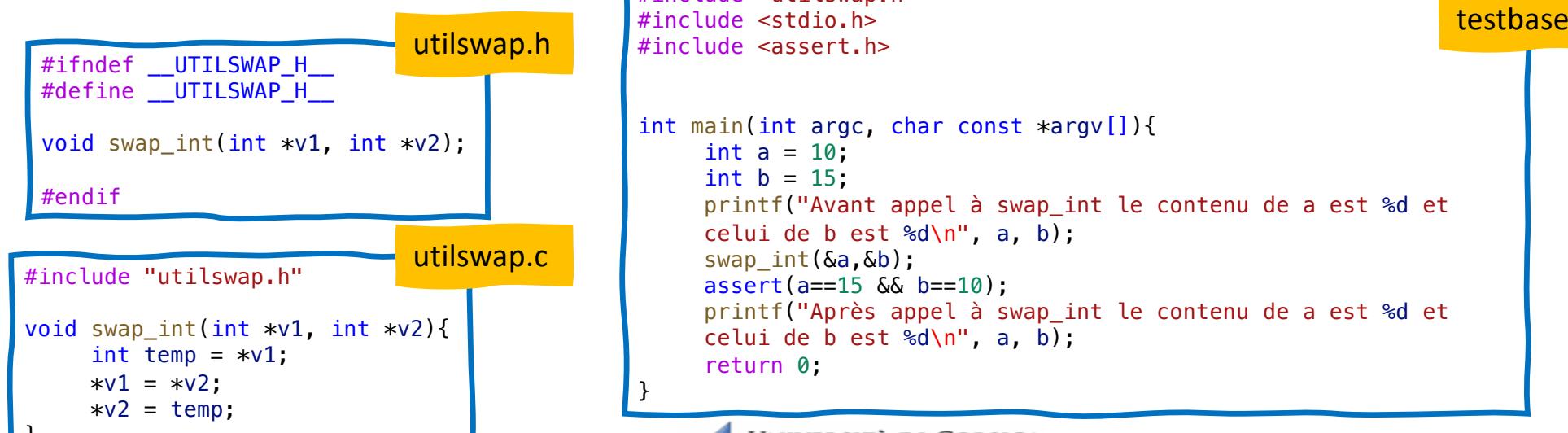
# Compiler un programme C fait de plusieurs fichiers

- Option : Tout compiler d'un coup

```
$gcc -Wall -pedantic testbase.c utilswap.c
```

S'il y a beaucoup de fichiers cela peut être long...

Compiling C programs with multiple files, Jacob Sorber  
[https://www.youtube.com/watch?v=2YfM-HxQd\\_8](https://www.youtube.com/watch?v=2YfM-HxQd_8)



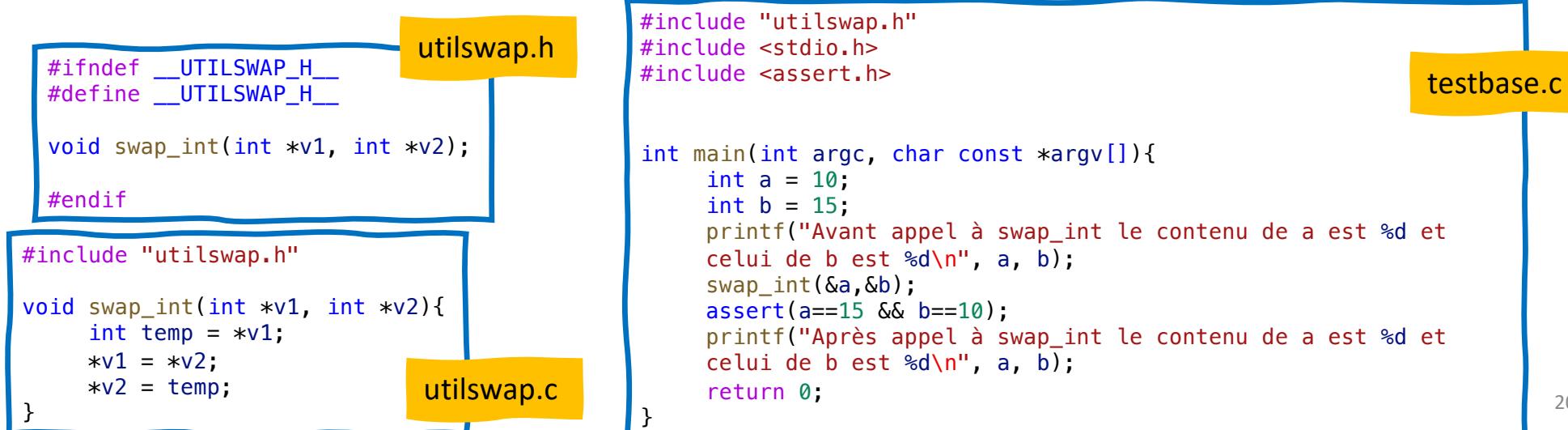
# Compiler un programme C fait de plusieurs fichiers

Compiling C programs with multiple files, Jacob Sorber  
[https://www.youtube.com/watch?v=2YfM-HxQd\\_8](https://www.youtube.com/watch?v=2YfM-HxQd_8)

- Option 2 : compiler les fichiers un à un (générer les .o) puis les linker à la fin

```
$gcc -Wall -pedantic -c utilswap.c
$gcc -Wall -pedantic -c testbase.c
$gcc -Wall -pedantic utilswap.c testbase.o -o testbase
```

On ne recompile que ce qui a été changé



# Compiler un programme C fait de plusieurs fichiers

- Option 3 : Utiliser un outil de build de code type Make

Compiling C programs with multiple files, Jacob Sorber  
[https://www.youtube.com/watch?v=2YfM-HxQd\\_8](https://www.youtube.com/watch?v=2YfM-HxQd_8)

# Un outil de build de code : make

make et les makefiles,

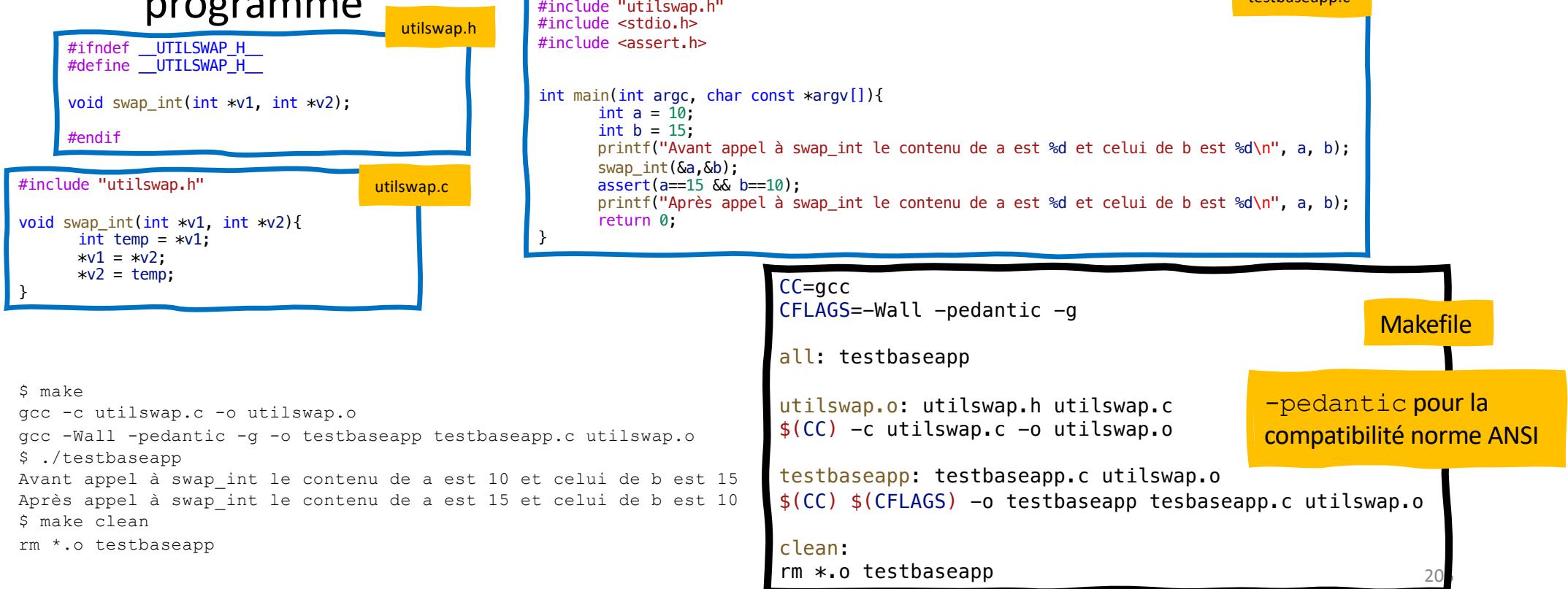
Ou bien...

ant, maven, rake, ...

Learn Make in 60 secondes, Jacob Sorber,  
<https://www.youtube.com/watch?v=a8mPKBxQ9No>  
Makefile-Related Videos, Jacob Sorber,  
<https://www.youtube.com/playlist?list=PL9IEJIKnBJjEPxenuhKU7J5smY4XfNvg>

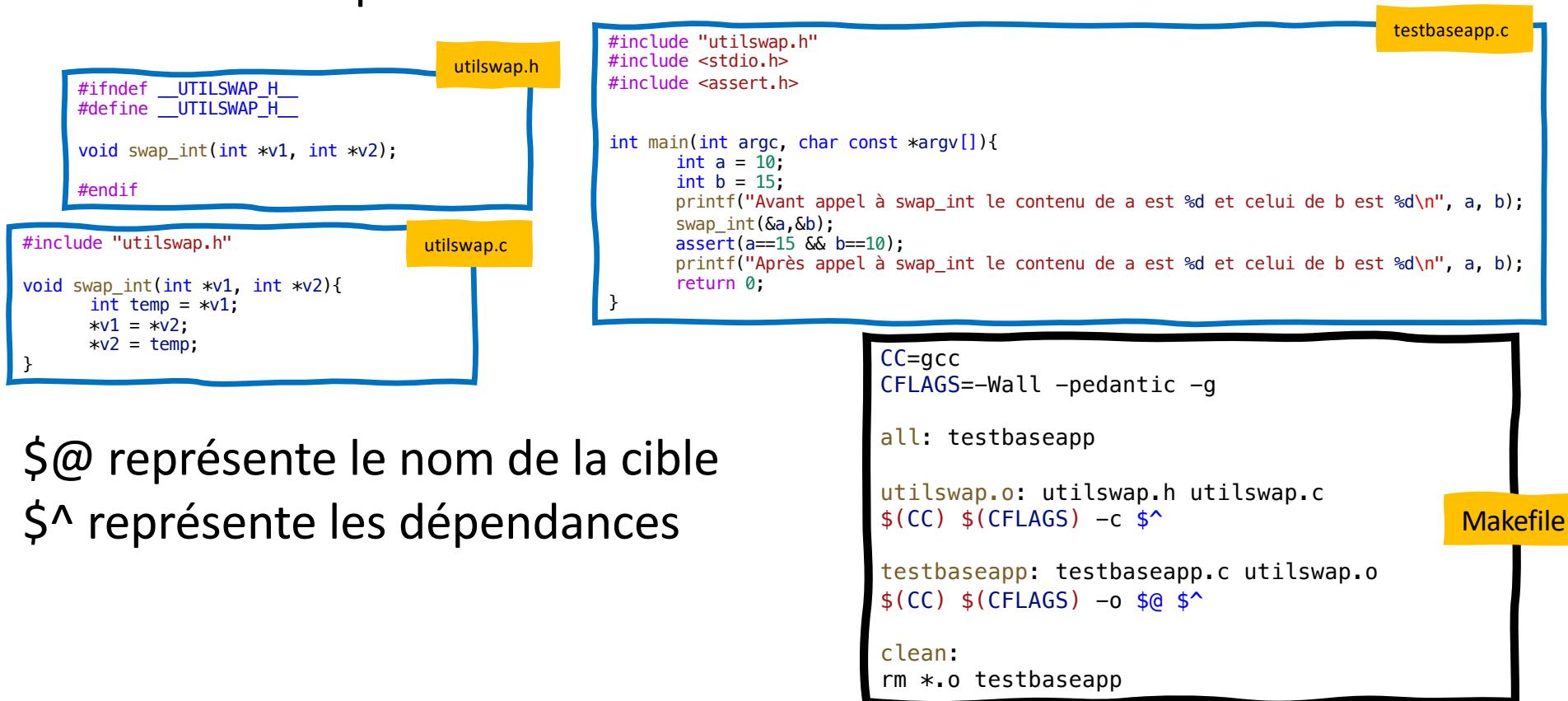
# Le makefile un fichier nommé Makefile

- Il décrit les règles dictant la manière de faire le « build » de votre programme



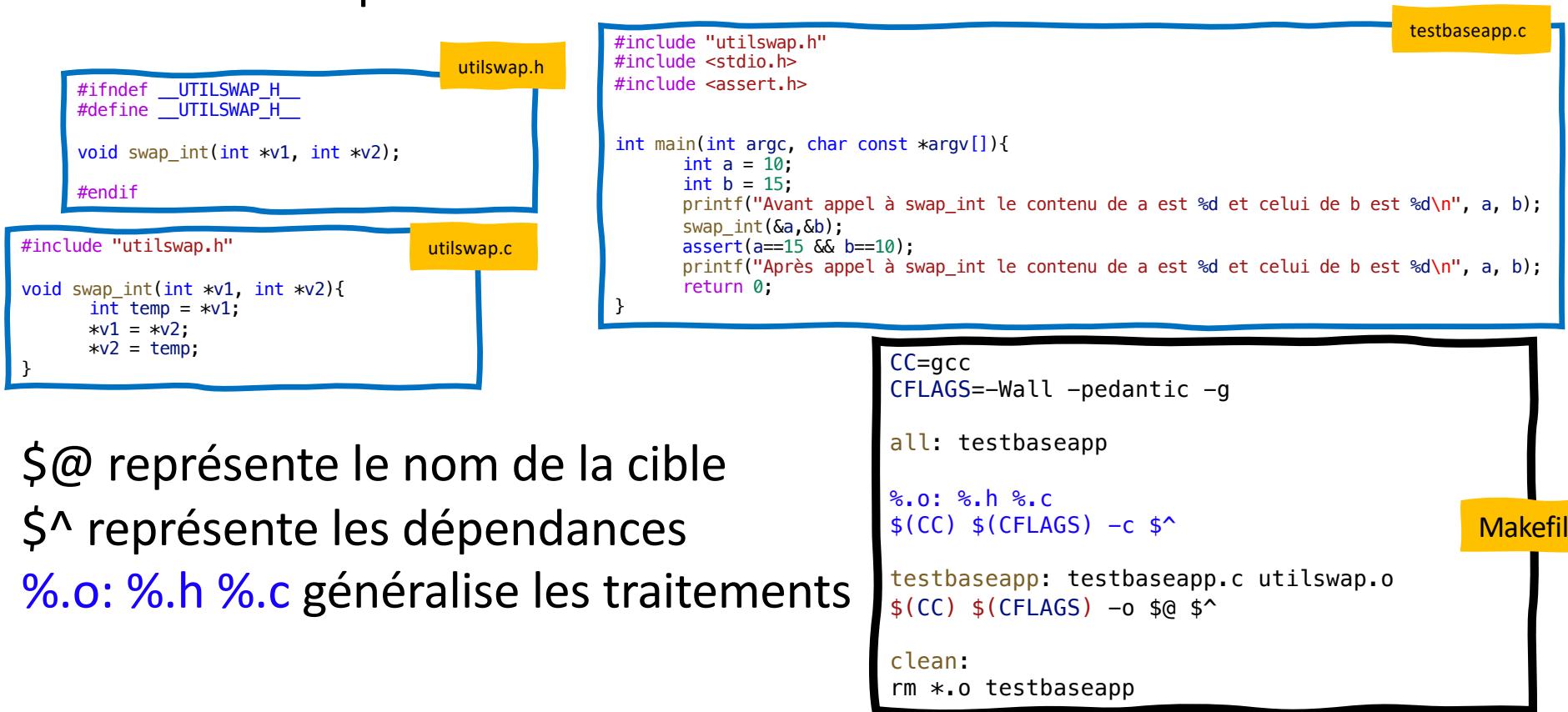
# Make, variables automatiques (1)

- Utilisées pour faciliter la maintenance de vos Makefile



# Make, variables automatiques (2)

- Utilisées pour faciliter la maintenance de vos Makefile



\$@ représente le nom de la cible

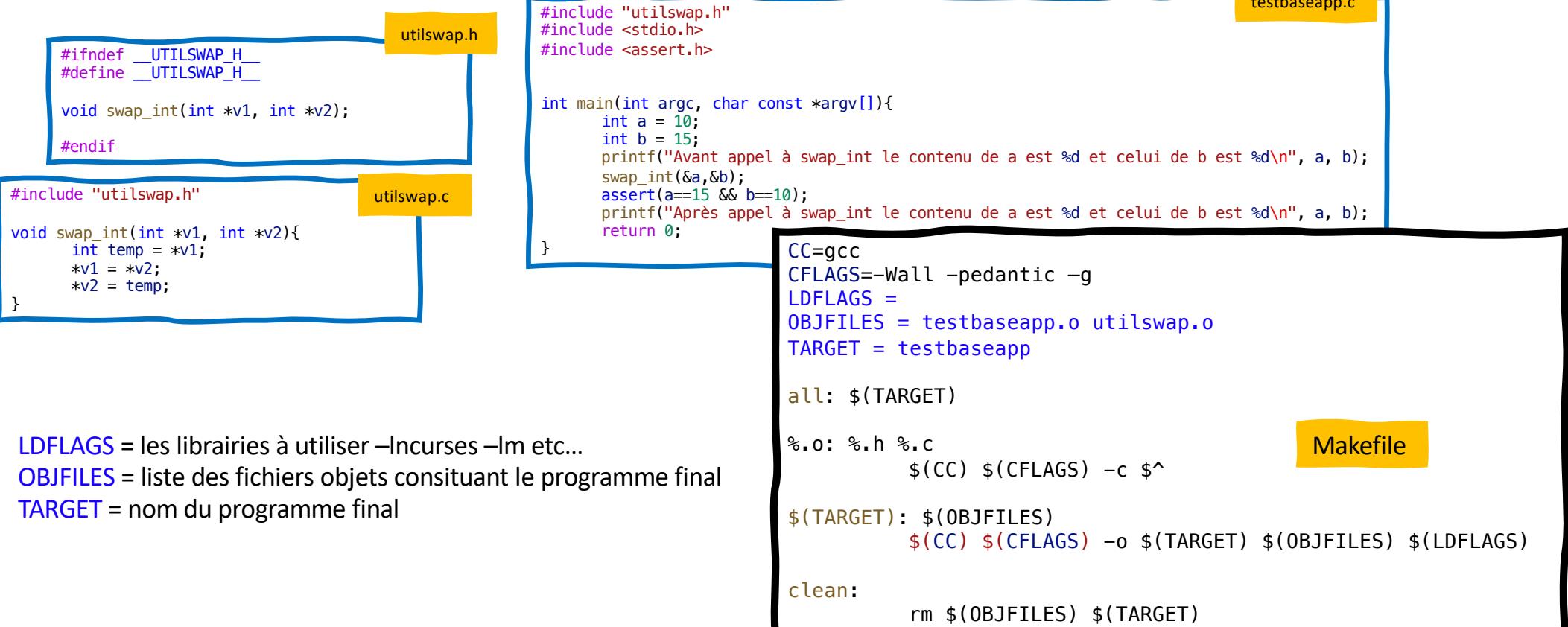
\$^ représente les dépendances

%.o: %.h %.c généralise les traitements

# Make, variables automatiques (3)

Trouvez-vous une bonne Cheat Sheet MakeFile...

- Utilisées pour faciliter la maintenance de vos Makefile



LDLIBS = les librairies à utiliser –lcurSES –lm etc...

OBJFILES = liste des fichiers objets constituant le programme final

TARGET = nom du programme final

# « Débuguer »

Certes via votre IDE, mais aussi en ligne de commandes via  
l'utilitaire gdb (The Gnu project Debugger), llDb

## What Languages does GDB Support?

GDB supports the following languages (in alphabetical order):

- Ada
- Assembly
- C
- C++
- D
- Fortran
- Go
- Objective-C
- OpenCL
- Modula-2
- Pascal
- Rust

La page officielle : <https://www.sourceware.org/gdb/>

GDB en 60 secondes (Jacob Sorber) : <https://www.youtube.com/watch?v=mfmXcbiRsOE>

# Un exemple simple

Cherchez l'erreur !  
En débugant  
factorial.c

Enter the number : 3

The factorial number of 3 is 61831800

```
#include <stdio.h>

int factorial(int n){
    int result;
    for (int i=1;i<=n;i++){
        result *= i;
    }
    return result;
}

int main(int argc, char const *argv[])
{
    int num;
    printf("Enter the number : ");
    scanf("%d",&num);
    printf("The factorial number of %d is %d\n",num,factorial(num));
    return 0;
}
```

# Les points importants pour débuguer

- Il faut compiler en incluant les informations nécessaires au débug -g

```
>gcc -g -o factorial factorial.c
```

- Lancer le debuger (**gdb ou lldb**) sur l'exécutable

```
>gdb factorial           >lldb factorial
```

- Positionner un point d'arrêt dans le programme à la ligne voulue

```
(gdb) break 10
```

```
(lldb) b main
```

- Lancer le debugage

```
(gdb) run
```

```
(lldb) run
```

- Ecrire le contenu des variables

```
(gdb) print num
```

```
(lldb) p num
```

Une gdb Cheat sheet complète :

<https://users.ece.utexas.edu/~adnan/gdb-refcard.pdf>

- c, continue : continue jusqu'au prochain point d'arrêt
- n, next : exécute la prochaine instruction sans rentrer dedans
- s, step : idem next mais entre dans la fonction et l'exécute ligne par ligne
- p, print
- ENTER réitère la commande précédente

# Explorer la mémoire pendant l'exécution via gdb

```
#include <stdio.h>
#include <stdint.h> //Pour utiliser des types entiers à taille fixée

typedef struct
{
    int8_t hours;
    uint32_t micros;
    uint16_t seconds;
} timestuff_t;

int main(int argc, char const *argv[])
{
    timestuff_t t= {.hours=6, .micros=0x12345678, .seconds = 0xDEAD};
    printf("%lu", sizeof(t));
    return 0;
}
```

How to examine memory in GDB ?, Jacob Sorber,  
[https://www.youtube.com/watch?v=A\\_pV61xFty8](https://www.youtube.com/watch?v=A_pV61xFty8)

Cheat Sheet GDB,  
<https://users.ece.utexas.edu/~adnan/gdb-refcard.pdf> 212

# Explorer la mémoire pendant l'exécution via gdb

```
$ gcc -g -o testmemory testmemorystructgdb.c
$ lldb testmemory
(lldb) target create "testmemory"
Current executable set to 'testmemory' (arm64).
(lldb) b main
Breakpoint 1: where = testmemory`main + 40 at testmemorystructgdb.c:13:17, address =
0x0000000100003f64
(lldb) run
Process 45769 launched: 'testmemory' (arm64)
Process 45769 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
  frame #0: 0x0000000100003f64 testmemory`main(argc=1, argv=0x000000016fdff428) at
testmemorystructgdb.c:13:17
10
11     int main(int argc, char const *argv[])
12 {
-> 13         timestamp_t t= {.hours=6, .micros=0x12345678, .seconds = 0xDEAD};
14         printf("%lu",sizeof(t));
15         return 0;
16     }
Target 0: (testmemory) stopped.
(lldb)
```

Combien d'octets devrait faire la structure t ?  
Combien en fait elle réellement ?  
Testez avec GDB en regardant le contenu de la mémoire...

How to examine memory in GDB ?, Jacob Sorber,  
[https://www.youtube.com/watch?v=A\\_pV61xFty8](https://www.youtube.com/watch?v=A_pV61xFty8)

x [/Nuf] expr	examine memory at address <i>expr</i> ; optional format spec follows slash
N	count of how many units to display
u	unit size; one of
b	individual bytes
h	halfwords (two bytes)
w	words (four bytes)
g	giant words (eight bytes)
f	printing format. Any <code>print</code> format, or
s	null-terminated string
i	machine instructions

## Cheat Sheet GDB,

<https://users.ece.utexas.edu/~adnan/gdb-refcard.pdf>

```
//File testmemorystructgdb.c
#include <stdio.h>
#include <stdint.h> //Pour utiliser des types entiers à taille fixée

typedef struct
{
    int8_t hours;
    uint32_t micros;
    uint16_t seconds;
} timestamp_t;

int main(int argc, char const *argv[])
{
    timestamp_t t= {.hours=6, .micros=0x12345678, .seconds =
0xDEAD};
    printf("%lu",sizeof(t));
    return 0;
}
```

# Explorer la mémoire pendant l'exécution via gdb

```
$11db testmemory
(lldb) target create "testmemory"
Current executable set to '/testmemory' (arm64).
(lldb) b 14
Breakpoint 1: where = testmemory`main + 64 at testmemorystructgdb.c:14:5, address =
0x0000000100003f7c
(lldb) run
Process 46008 launched: '/testmemory' (arm64)
Process 46008 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
  frame #0: 0x0000000100003f7c testmemory`main(argc=1, argv=0x000000016fdff428) at
testmemorystructgdb.c:14:5
11    int main(int argc, char const *argv[])
12    {
13        timestamp_t t = {.hours=6, .micros=0x12345678, .seconds = 0xDEAD};
-> 14        printf("%lu",sizeof(t));
15        return 0;
16    }
17

Target 0: (testmemory) stopped.
(lldb) p sizeof(t)
(unsigned long) $0 = 12
(lldb) p t
(timestamp_t) $1 = (hours = '\x06', micros = 305419896, seconds = 57005)
(lldb) p &t
(timestamp_t *) $2 = 0x000000016fdff290
(lldb) x &t
0x16fdff290: 06 00 00 00 78 56 34 12 ad de 00 00 01 80 60 29  ....xV4.....
0x16fdff2a0: 28 f4 df 6f 01 00 00 00 01 00 00 00 00 00 00 00
(lldb) x/12 &t
0x16fdff290: 0x00000006 0x12345678 0x0000dead 0x29608001
0x16fdff2a0: 0x6fdff428 0x00000001 0x00000001 0x00000000
0x16fdff2b0: 0x6fdff400 0x00000001 0x0001108c 0x00000001
(lldb) x/12xb &t
0x16fdff290: 0x00000006 0x12345678 0x0000dead 0x29608001
0x16fdff2a0: 0x6fdff428 0x00000001 0x00000001 0x00000000
0x16fdff2b0: 0x6fdff400 0x00000001 0x0001108c 0x00000001
(lldb) x/12xb &t
0x16fdff290: 0x06 0x00 0x00 0x00 0x78 0x56 0x34 0x12
0x16fdff298: 0xad 0xde 0x00 0x00
(lldb) x/3xw &t
0x16fdff290: 0x00000006 0x12345678 0x0000dead
(lldb) quit
```

x [ /Nuf ] expr	examine memory at address <i>expr</i> ; optional format spec follows slash
N	count of how many units to display
u	unit size; one of
b	individual bytes
h	halfwords (two bytes)
w	words (four bytes)
g	giant words (eight bytes)
f	printing format. Any print format, or
s	null-terminated string
i	machine instructions

## Cheat Sheet GDB,

<https://users.ece.utexas.edu/~adnan/gdb-refcard.pdf>

```
#include <stdio.h>
#include <stdint.h> //Pour utiliser des types entiers à taille fixée

typedef struct
{
    int8_t hours;
    uint32_t micros;
    uint16_t seconds;
} timestamp_t;

int main(int argc, char const *argv[])
{
    timestamp_t t= {.hours=6, .micros=0x12345678, .seconds = 0xDEAD};
    printf("%lu",sizeof(t));
    return 0;
}
```

La structure devrait faire 7 octets... Elle en fait 12...

How to examine memory in GDB ?, Jacob Sorber,  
[https://www.youtube.com/watch?v=A\\_pv61xFty8](https://www.youtube.com/watch?v=A_pv61xFty8)

# Créer ses propres librairies C

Pourquoi ? Comment ?

# Fichiers d'entête \*.h

## Différence entre déclaration et définition

- Déclaration – dans le '.h' on ne donne que l'entêtes des fonctions/procédure
- Définition – dans le '.c' - on donne le code

# Pourquoi écrire ses propres librairies ?

- Partager du code avec d'autres
- Réutiliser plus facilement et plus proprement votre propre code dans différents projets

# Les différentes étapes

- Identifier les fonctionnalités et le code que vous souhaitez partager
- Créer un header file correspondant à ce code (déclaration des éventuels types) et ces fonctionnalités (déclaration des en-têtes de fonctions/procédures)
- Ecrire le code C correspondant à la définition des fonctions/procédures déclarées dans le header.
- Compiler le code de votre librairie et fournir à vos potentiels usagers le header ainsi que le code compilé de votre librairie.

# L'aléatoire

Quelle fonction utiliser pour avoir un nombre aléatoire ?

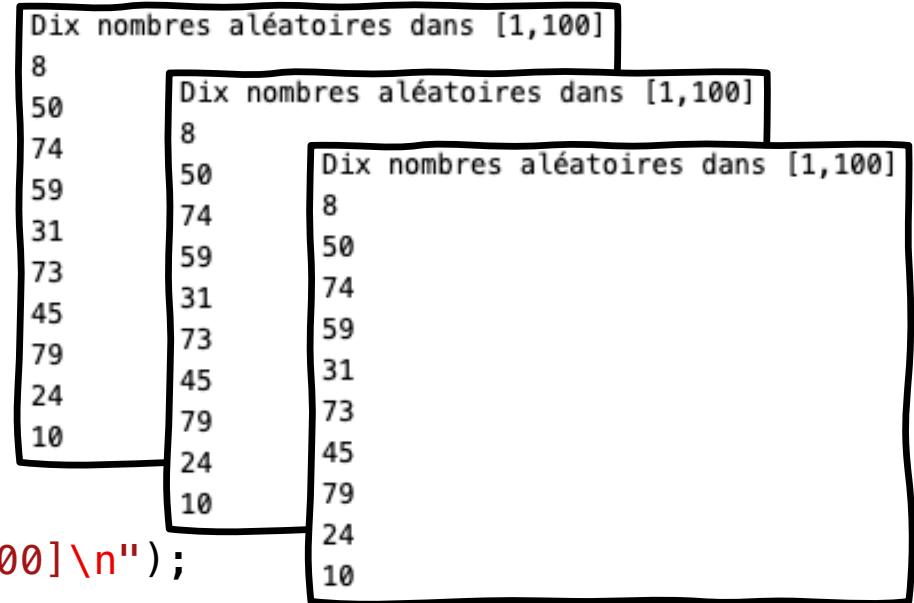
Comment obtenir des suites de nombres toujours différentes d'une exécution à l'autre ?

Obtenir un nombre aléatoire dans une fourchette de valeur donnée ?

# Pseudo-aléatoire en C

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

int main(){
    int i, n;
    printf("Dix nombres aléatoires dans [1,100]\n");
    for (i = 1; i <= 10; i++){
        n = rand() % 100 + 1;
        printf("%d\n", n);
    }
    return 0;
}
```



Testez-le plusieurs fois...  
Que constatez vous ?  
Que proposez vous ?

# Aléatoire ou non aléatoire ?

- L'aléatoire n'existe pas pour un ordinateur car c'est une machine déterministe
  - Elle exécute les instructions qu'on lui donne de façon logique et ordonnée
- Seul l'observation d'un système naturel pourrait générer du « vrai » aléatoire
  - Quantité humidité dans l'air
- On se contente du Pseudo-Aléatoire
  - Résultat complexe à anticiper donnant l'illusion de l'aléatoire

Pseudorandom number generators (Khan Academy), Created by Brit Cruise

<https://www.khanacademy.org/computing/computer-science/cryptography/crypt/v/random-vs-pseudorandom-number-generators>

# Pseudo-aléatoire

- Construction d'une suite de nombres à partir d'une « graine » (seed) et d'un algorithme de calcul
  1. On prend un nombre graine
  2. On lui applique un calcul (par exemple  $(x + 3,14159)^8$ )
  3. On obtient un nombre difficilement prévisible, pseudo-aléatoire
  4. On ré-applique le même calcul sur le résultat obtenu
- On obtient une suite peu prévisible mais reproductible via la même graine et le même calcul

Utiliser `srand(seed)` oui, mais comment choisir la graine ?

En C :

- `rand()` retourne un nombre entre 0 et `RAND_MAX (>32767)`
- `srand(unsigned int seed);` initialise le générateur de nombre

# Pseudo-aléatoire en C

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

int main(){
    int i, n;
    //Initialisation du générateur de nombre pseudo-aléatoires
    srand(time(NULL));
    printf("Dix nombres aléatoires dans [1,100]\n");
    for (i = 1; i <= 10; i++){
        n = rand() % 100 + 1;
        printf("%d\n", n);
    }
    return 0;
}
```

Dix nombres aléatoires dans [1,100]	Dix nombres aléatoires dans [1,100]	Dix nombres aléatoires dans [1,100]
75	41	81
61	88	27
82	40	48
4	34	99
34	62	17
32	34	97
94	92	52
41	19	32
100	83	1
94	52	31

# Entrées/sorties utilisateur

## printf chaînes de formatage

- printf prend en paramètre une chaîne de formatage ainsi que la variable contenant la valeur à afficher

- **%d** est le format pour afficher une valeur entière.
- **%i** est un autre format pour afficher un entier, une variable de type int.
- **%ld** est le format pour afficher un long.
- **%f** est le format pour afficher un float.
- **%lf**, **%g**, **%e** sont des formats pour afficher un double.
- **%x** est le format pour afficher une valeur hexadécimale.
- **%c** est le format pour afficher un caractère.
- **%s** est le format pour afficher une chaîne de caractères.
- **%p** est le format pour afficher une adresse mémoire (en fait une valeur hexadécimale comme pour **%x**).

```
printf("test=% .2f\n", 1.234567f); // affiche : 1.23
printf("[%4d] [%-4d]\n", 10, 20); // [ 10] [20 ] '-' left
printf("[%5.2lf] [%-5.2s]\n", 1.567, "Arthur");
// [ 1.57] [Ar ] '-' left 5 car en tout, 2 nb car affichés
```

Les formats de printf **%f**, pour des valeurs **float** et **%lf**, pour des valeurs **double**, pratiquent des arrondis lorsque tous les chiffres après la virgule ne sont pas affichés. Ainsi, 1.567 avec le format "% .2f" devient pour l'affichage uniquement 1.57.

Exercice : jouez avec ces chaînes de formatage pour vous les approprier...

# Entrées/sorties utilisateur

## printf/scanf(chaineFormatage,adressesDesVariables)

- scanf prend en paramètre une chaîne de formatage ainsi que l'adresse de la variable devant recevoir la valeur saisie
  - Rappelez-vous que le passage de paramètre par défaut en C est un passage par valeur

```
#include <stdio.h>
int main() {
    int rec;
    //lecture d'un nombre, validation touche entrée
    scanf("%d",&rec);
    return 0;
}
```

```
#include <stdio.h>
int main() {
    int rec1,rec2;
    float rec3;
    //les différents nombres doivent être séparés par des espaces
    scanf("%d%d%f",&rec1,&rec2,&rec3);
    return 0;
}
```

```
#include <stdio.h>
int main() {
    char c1,c2,c3,c4;
    printf("Entrez quatre lettres : ");
    //Ici ne pas séparer les caractères par des espaces ou
    //l'espace sera considéré comme un des caractères
    scanf("%c%c%c%c", &c1, &c2, &c3, &c4);
    printf("vous avez entre : %c, %c, %c, %c\n", c1, c2, c3, c4);
    return 0; }
```

# Entrées/sorties simples, préférez

scanf\_s(chaineFormatage, adressesDesVariables, nbOctets)

- scanf\_s fonctionne comme scanf sauf pour les caractères et chaînes de caractères où il faut en plus spécifier la taille des espaces de stockage en octets

```
#include <stdio.h>
int main() {
    int rec;
    //Idem que scanf
    scanf_s("%d", &rec);
    return 0;
}
```

Spécifique  
Microsoft  
Windows

```
#include <stdio.h>
int main() {
    char c=0;
    printf("Entrez une lettre : ");
    scanf_s("%c", &c, 1);
    printf("vous avez entre : %c\n", c);
    return 0;
}
```

```
#include <stdio.h>
int main() {
    char a,b,c;
    printf("Entrez trois lettres : ");
    scanf_s("%c%c%c", &a, 1,&b,1,&c,1);
    printf("vous avez saisi : %c, %c, %c", a, b, c);
    return 0;
}
```

```
#include <stdio.h>
int main(){
    char mot[80];
    printf("entrez un mot : ");
    scanf_s("%s", mot,80);
    printf("vous avez entre le mot : %s\n", mot);
    return 0; }
```

# Entrées, récupération des erreurs

- `scanf_s` et `scanf` retournent le nombre de saisies correctes ayant eu lieu sachant qu'à la première erreur les suivantes ne sont pas effectuées.

```
#include <stdio.h>
int main() {
    int rec1, rec2;
    printf("Entrez deux valeurs entières : \n");
    int nbInt = scanf_s("%d%d", &rec1, &rec2);
    printf( "Il y a %d valeurs saisis : %d et %d\n", nbInt, rec1, rec2);
    return 0;
}
```

# Eviter les erreurs d'entrée, réinitialisation du buffer

- Lors de plusieurs appels successifs de `scanf_s` ou `scanf` le comportement est parfois incompréhensible à des caractères stockés dans le buffer d'entrée

```
#include <stdio.h>
int main() {
    char c1;
    char c2;
    char c3;
    scanf("%c", &c1);
    scanf("%c", &c2);
    scanf("%c", &c3);
    printf("vous avez entre : %c, %c, %c\n", c1, c2,
c3);
    return 0;
}
```

A tester...

```
#include <stdio.h>
int main() {
    char c1;
    char c2;
    char c3;
    scanf("%c", &c1);
    rewind(stdin);
    scanf("%c", &c2);
    rewind(stdin);
    scanf("%c", &c3);
    rewind(stdin);
    printf("vous avez entre : %c, %c, %c\n", c1, c2,
c3);
    return 0;
}
```

# Exercices Jeux mode console

- Deux possibilités selon votre OS
  - Windows : librairies conio.h et windows.h
  - Linux/MacOS : ncurses.h
- Principes équivalents
  - Lectures d'événements non blocants dans une boucle infinie
    - W : \_kbhit() et \_getch()
    - L/MacOS : préparation du terminal en non blocant avec {initscr(), endwin(), getch(), printw(), and refresh()}

Récupérez le code jeuModeConsole\_0.c sur l'ENT,  
testez-le et explorez en les possibilités

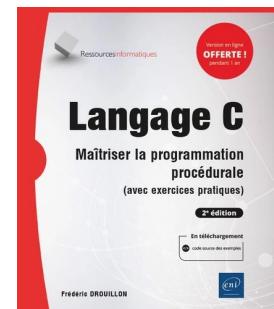
# Ressources

- [1] V. Barra, *Informatique MP2I et MPI : CPGE 1re et 2e années - Nouveaux programmes* - ScholarVox Université, Ellipses. 2021.



Consulté le: 6 septembre 2022. [En ligne]. Disponible sur:  
<http://univ.scholarvox.com.udcpp.idm.oclc.org/catalog/book/docid/88915779>

- [2] *Langage C. Maîtriser la programmation procédurale (avec exercices pratiques)* 2e édition - Frédéric Drouillon. Consulté le: 18 septembre 2022. [En ligne]. Disponible sur: <http://www.eni-training.com/portal/client/mediabook/home>



# Annexes

# TAD

# Types abstraits de données

Présentation et utilisation

# Plan du cours d'algorithmique

- La récursivité
- Les variables dynamiques, les pointeurs
- Les types abstraits de données
- Les listes
- Les piles
- Les files
- Les arbres

# Définition TAD

- Un type de donnée n'est pas seulement un ensemble de valeurs, mais aussi un ensemble d'opérations
- Cette notion permet d'analyser un grand nombre d'erreur

# Principe des TAD

- Décrire précisément les opérations
- Décrire les propriétés de ces opérations
- Sans donner explicitement l'ensemble des valeurs
- Sans spécifier la représentation choisie

On ne s'occupe ni de la machine ni du langage



# Pourquoi des TAD ?

Abstraction

- On se concentre sur l'essentiel
  - Qu'est-ce que c'est ?
  - Qu'est-ce que ça fait ?
  - Comment cela (inter)agit ?
- Les détails de représentation sont laissés pour plus tard, on reste loin de la machine
  - A l'implémentation on choisit le plus efficace, selon différents critères

On dit ce qu'il faut faire sans dire comment le faire

# Utilité des TAD

- L'utilisateur peut écrire ces algorithmes en utilisant les opérations qu'il a demandé
- L'implémenteur doit seulement assurer que les opérations respectent le contrat passé, qu'elles possèdent bien les propriétés attendues.
- L'implémenteur peut choisir le langage, la machine qu'il veut

# Plan du cours sur les TAD

- Première approche
  - Exemple du TAD Booléen
- Hiérarchie des TAD
- Format de spécification d'un TAD
- Un exemple complet
  - Le TAD Tableaux

# Première approche : spécification du type booléen

- *Type booléen volet spécification*
- *Fonctions d'accès :*
  - *Constantes :*
    - vrai :  $\rightarrow$  booléen
    - faux :  $\rightarrow$  booléen
  - *Opérations :*
    - $\neg_$  : booléen  $\rightarrow$  booléen
    - $_ \wedge _$  : booléen  $\times$  booléen  $\rightarrow$  booléen
    - $_ \vee _$  : booléen  $\times$  booléen  $\rightarrow$  booléen
- *Sémantique ou axiomes (Précise les propriétés des opérations)*
  - $\forall a, b, \text{booléen} : 1. \neg \text{vrai} = \text{faux} \quad 2. \neg \neg a = a$
  - $3. \text{vrai} \wedge a = a \quad 4. \text{faux} \wedge a = \text{faux}$
  - $5. a \vee b = \neg (\neg a \wedge \neg b)$

# Première approche : booléen(2)

- On peut déduire des propriétés à partir des propriétés donnée
  - $\neg \text{faux} = \text{vrai}$  {1 puis 2}
  - $\text{faux} \vee b = b$  {5, 6, 3}
  - $\text{vrai} \vee a = \text{vrai}$  {5,1,4,6}
- Le T.A.D. booléen est autonome, il ne fait référence à aucun autre T.A.D, ce n'est pas souvent le cas

# Notion de hiérarchie de TAD

- Construire un T.A.D à partir de types prédéfinis
- Exemple : T.A.D Tableau de réel à une dimension
  - Bornes du tableau : Type entier
  - Éléments du tableau : Type réel
- On a pourtant pas besoin de ces deux types de la même façon...
  - Propriétés sur les entiers, mais pas sur les réels
  - Les éléments du tableau peuvent être de n'importe quel type

# Spécification d'un T.A.D.

- Type nom\_du\_type
- Paramètres, Utilise
- Fonctions d'accès
  - Constantes
    - Invariants qui caractérisent les données du type abstrait
  - Opérations de construction
    - Permettent de produire des données du type abstrait à partir d'autres données
  - Test de type
    - Permet de déterminer si un objet appartient au type abstrait
  - Opérateurs
    - Permettent de manipuler les données du type abstrait
  - Opérateurs de Test
- Axiomes : décrivent les propriétés des opérations sous forme d'équivalences entre termes

# Spécification d'un T.A.D.

- On précise si le type fait appel à d'autres types
- Pour chacune des opérations on donne la signature
  - *Les types et le nombre de ses paramètres*
  - *Le type du résultat*

Opération : type1 x ... x typeN → type\_resultat

vrai : → booléen

Et : booléen x booléen → booléen

# Concevoir un T.A.D, exemple des tableaux

- Partir d'un cahier des charges
  - Opération de création de tableau
  - Quelle quantité d'information
  - Intervalle des numéros [b\_inf, b\_sup]
  - Quels types on utilise ?
  - Comment retrouver un élément ?
  - Comment insérer un élément ?
  - En paramètre : le type des éléments



On obtient le T.A.D. Tableau suivant

# Un T.A.D. tableau

Type tableau volet spécification

Paramètre élément

Utilise entier booléen

Opérations de construction

tab : entier x entier → tableau

Opération d'accès et de modification

ième : tableau x entier → élément

affecter : tableau x entier x élément → tableau

bornesup : tableau → entier

borneinf : tableau → entier

valué? : tableau x entier → booléen

# Un T.A.D. tableau

## Type tableau volet sémantique, axiomes

- ième ( $\text{tab}(d,f),i$ ) :
  - indéfini
- ième ( $\text{affecter}(t,k,e),i$ ) :
  - Si  $i > \text{bornesup}(t)$  ou  $i < \text{borneinf}(t)$   
alors indéfini
  - Sinon      Si  $i = k$  alors  $e$
  - Sinon ième( $t,i$ )
  - Finsi
  - Finsi
- $\text{bornesup}(\text{tab}(i,j))$ 
  - $j$
- $\text{bornesup}(\text{affecter}(t,k,e))$ 
  - $\text{bornesup}(t)$

# Un T.A.D. tableau

Type tableau volet spécification

Paramètre élément

Utilise entier, booléen

Opérations de construction

tab : entier x entier → tableau

Opération d'accès et de modification

ième : tableau x entier → élément

affecter : tableau x entier x élément → tableau

bornesup : tableau → entier

borneinf : tableau → entier

valué? : tableau x entier → booléen

Préconditions

ième(t,i) : valué?(t,i)

affecter(t,i,e) : bornesup(t) ≥ i ≥ borneinf(t)

# Un T.A.D. tableau

## Type tableau volet sémantique axiomes

- ième ( $\text{affecter}(t,i,e),j$ ) :
  - Si  $j=i$  alors  $e$   
Sinon ième( $t,j$ )  
Finsi
- borneinf( $t$ ) :
  - Si  $t=\text{tab}(i,j)$  alors  $i$   
Sinon soit  $t = \text{affecter}(t',k,e);$   
 $\text{borneinf}(t');$   
Finsi
- bornesup( $t$ ) :
  - Si  $t=\text{tab}(i,j)$  alors  $j$   
Sinon soit  $t = \text{affecter}(t',k,e);$   
 $\text{bornesup}(t');$   
Finsi
- valué?( $t,i$ ) :
  - Si  $t=\text{tab}(d,f)$  alors faux  
Sinon soit  $t=\text{affecter}(t',k,e);$   
Si  $k=i$  alors vrai  
Sinon  $\text{valué?}(t',i)$   
Finsi

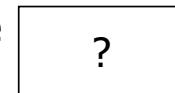
## Modèle abstrait de stockage

- La mémoire est vue comme une collection de cellules de stockage (case mémoire ou storage cells), chacune ayant une adresse unique
- Chaque cellule est dans un état donné
  - Allouée ou non alloué (libre)
- Chaque cellule allouée possède à tout moment un contenu
  - Une valeur ou indéfini

Cellule allouée et  
valuée



Cellule allouée  
non valuée



Cellule non allouée

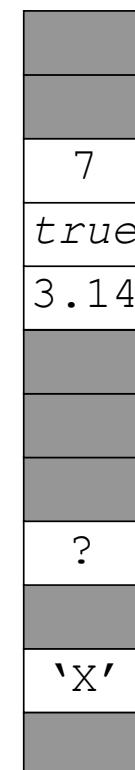
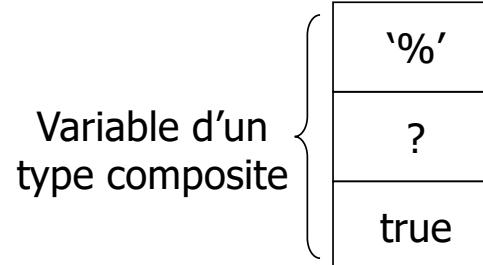


# Modèle abstrait de la variable

- Une variables est un conteneur consistant en une ou plusieurs cellule de stockage
- Une variable d'un type
  - primitif occupe une seule cellule
  - composite occupe plusieurs cellules contiguës

Variable d'un type primitif

7



# Variables statiques/dynamiques

- Quel est votre définition de ces deux termes ?
- Qu'est ce qui différencie une variable statique d'une variable dynamique ?
- Un pointeur sur un int est-il une variable statique ou une variable dynamique ?

```
int *pe;
```

# Variable statique

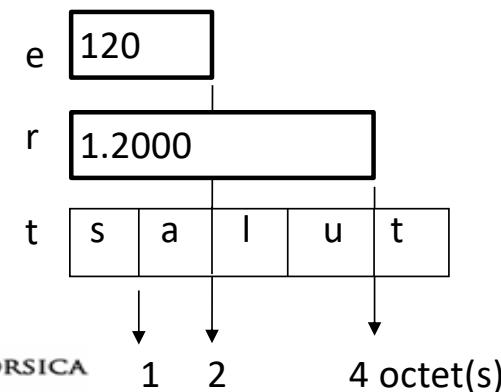
- Un objet créé avant l'exécution possédant
  - Un identificateur
  - Un type (une taille mémoire)
  - Une valeur (éventuelle)
  - Un emplacement mémoire (défini par l'adresse de la case)

```
int_16 e;      float r;
e = 120;       r = 1.2;
char[5] t;
t[0] = 's';   t[3] = 'l';
t[2] ='a';    t[4] = 'u';
t[5] = 't';
```

Que se passe-t-il ?

- Un certain espace, désigné par son adresse en mémoire est associé à l'identificateur e
- Association faite par le compilateur (avant exécution)
- Les occurrences de e dans le prog sont remplacées par l'@ de la variable
- Au moment de l'exécution toutes ces @ sont fixées

Allocation statique



# Exercice

- Donnez la taille mémoire réservée lors de la déclaration des variables ci-dessous en C

```
float a,b,c;
short d[10][20];
char e[] = {"Bonjour !"};
char f[][10] = {"un", "deux", "trois", "quatre"};
```

# Exercice

- Donnez la taille mémoire réservée lors de la déclaration des variables ci-dessous en C

```
float a,b,c;
short d[10][20];
char e[] = {"Bonjour !"};
char f[][10] = {"un", "deux", "trois", "quatre"}
```

```
float a,b,c; //12 octets
short d[10][20]; //10*20*2=400 octets
char e[] = {"Bonjour !"}; //9+1=10 octets
char f[][10] = {"un", "deux", "trois", "quatre"}; //4*10 octets
```

# Variable dynamique

- Un objet créé **pendant** l'exécution d'un algorithme (si besoin est !) possédant
  - Pas d'identificateur
  - Un type
  - Une valeur
  - Un emplacement mémoire (défini par l'adresse de la case, comme tous les objets stockés)
- Cette variable est accessible par le biais d'un **pointeur**

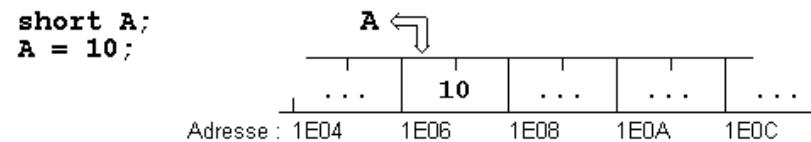
Allocation dynamique

# L'adressage de la mémoire

F.Faber

- Direct

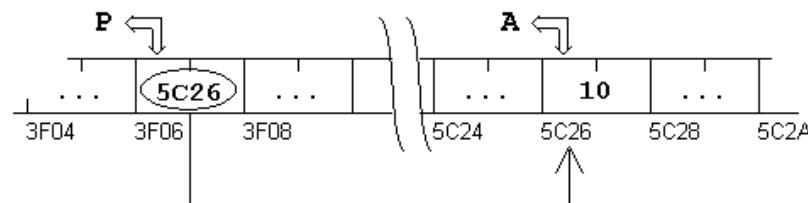
- Accès au contenu d'une variable par le nom de la variable



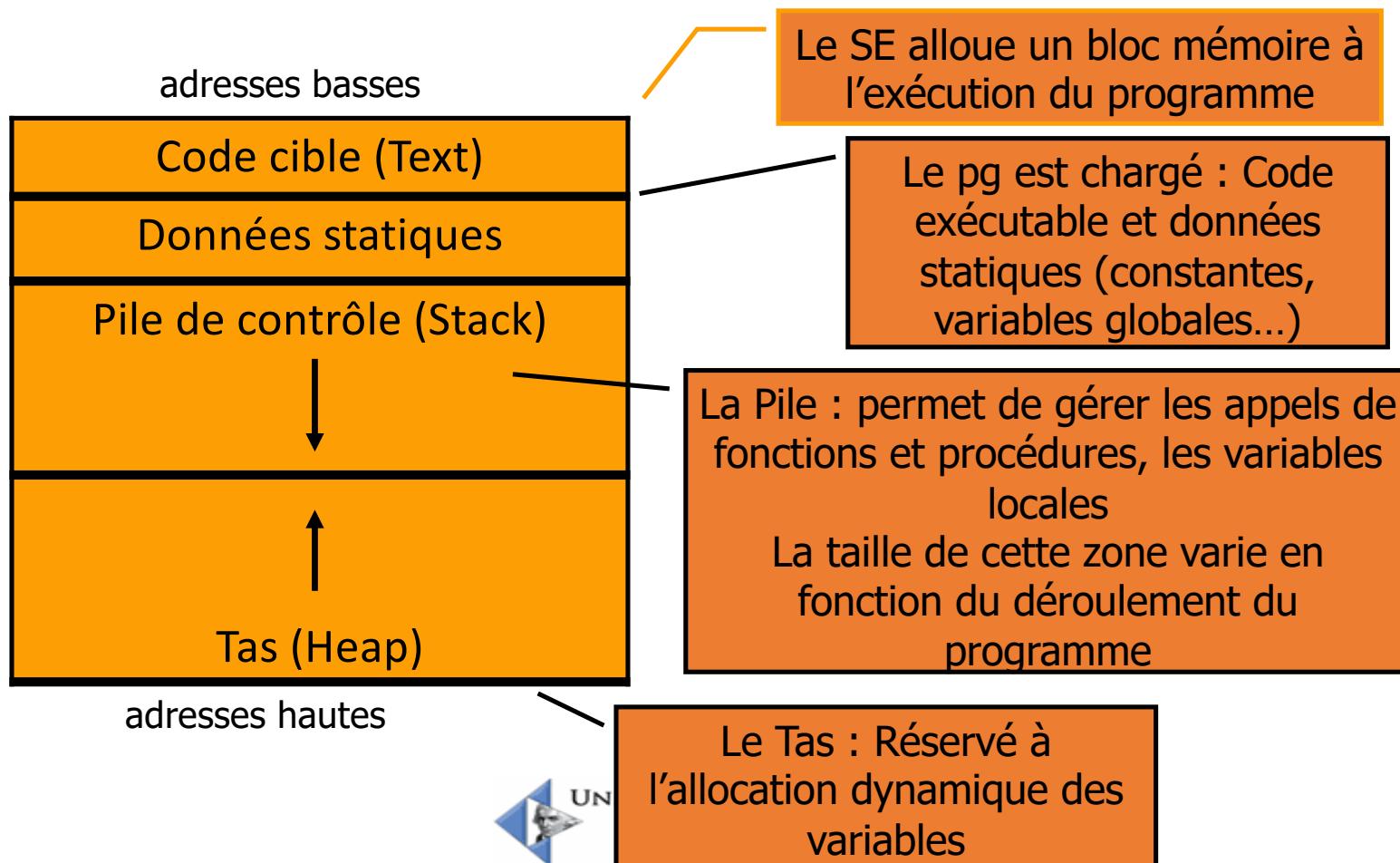
- Indirect

- Accès au contenu d'une variable, en passant par un pointeur qui contient l'adresse de la variable

Où A est une variable d'un type primitif contenant la valeur 10 et P un pointeur qui contient l'adresse de A.



## Rappel : Organisation de la mémoire à l'exécution, structure interne des processus



# Mise à jour partielle ou totale

- Une variable composite peut selon les langages être mise à jour
  - en une seule instruction : mise à jour totale
  - en plusieurs, un composant à chaque fois : mise à jour partielle ou sélective

Mise à jour sélective

C++

```
struct Date{  
    int y,m,d;  
};  
Date today, tomorrow;  
tomorrow = today;  
tomorrow.d = today.d+1;
```

Mise à jour totale : copie du contenu des 3 cellules de today dans les 3 cellules de tomorrow

## Différences entre la sémantique de la copie et de la référence

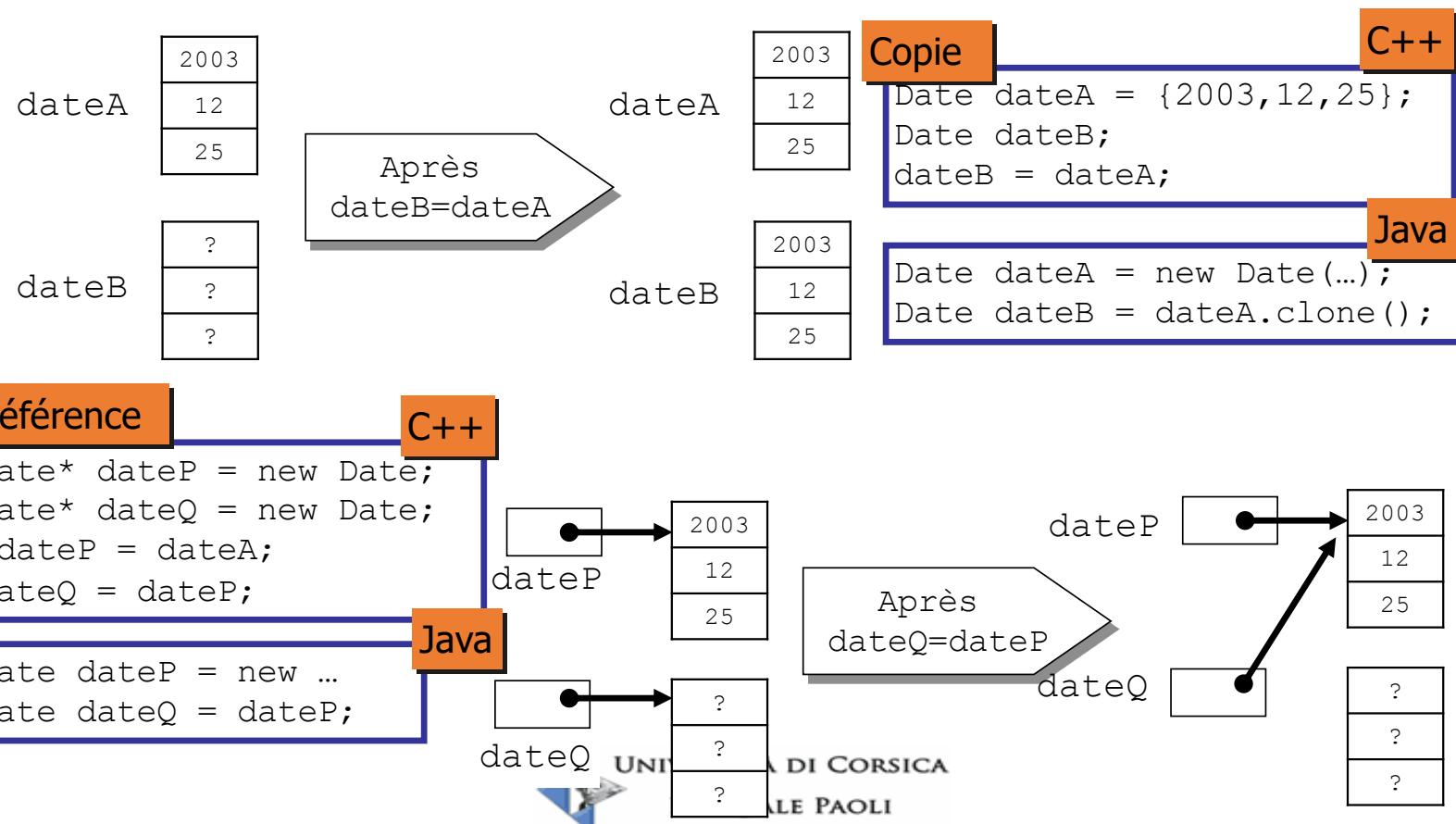
- Que se passe-t-il lors de l'affectation d'une valeur composite à une variable du même type ?
  - Sémantique de la copie : les composants de la valeur composite sont copiés dans les cellules correspondantes de la variable composite
  - Sémantique de la référence : la variable composite contient un pointeur ou référence à la valeur composite

C, C++, ADA : copie  
Possibilité d'utiliser les références en utilisant explicitement les pointeurs; Java : référence

## Sémantique associée au test d'égalité

- Elle doit être consistante avec la sémantique adoptée pour l'affectation
  - On assure ainsi au programmeur qu'immédiatement après l'affectation  $v1 = v2$ ,  $v1$  et  $v2$  sont égales
- Ainsi lors du test d'égalité
  - Sémantique de la copie : teste si les composants des variables composites sont égaux deux à deux (deux espaces mémoire séparés)
  - Sémantique de la référence : teste si la valeur des deux pointeurs référençant les valeurs composites sont égaux (un seul espace mémoire)

## Illustration test d'égalité sémantique de copie/référence



# Portée et durée de vie des variables

- La durée de vie d'une variable est définie comme le temps écoulé entre la création (allocation mémoire) et la destruction (libération de la mémoire) de la variable
- Durée de vie dépendante du type de variable
  - Globale
    - durée de vie du programme
    - créée par une déclaration globale
    - possède un identifiant
  - Locale
    - durée de vie du bloc englobant
    - créée par une déclaration locale dans un bloc, inaccessible à la sortie du bloc
    - possède un identifiant
  - Dynamique (variable du tas)
    - durée de vie arbitraire, au maximum celle de la durée de vie du programme
    - créée n'importe où dans le programme par une instruction ou une expression, peut être ensuite détruite n'importe où.
    - Pas d'identifiant, on y accède via un pointeur
  - Persistante : durée de vie arbitraire, peut dépasser celle de la durée de vie du programme

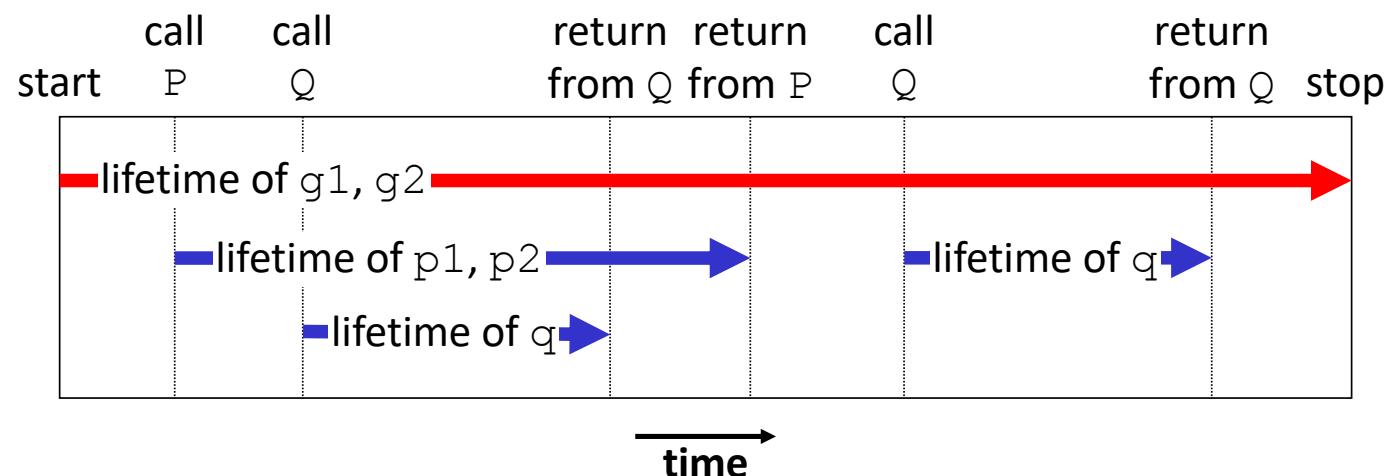
# Exemple variables globales/locales

ADA

```

procedure main is
    g1: Integer; g2: Float;
begin ... P; ... Q; ... Q;
end;
procedure P is
    p1: Float; p2: Integer;
begin ... Q; ... end;
procedure Q is
    q: Integer;
begin ... end;

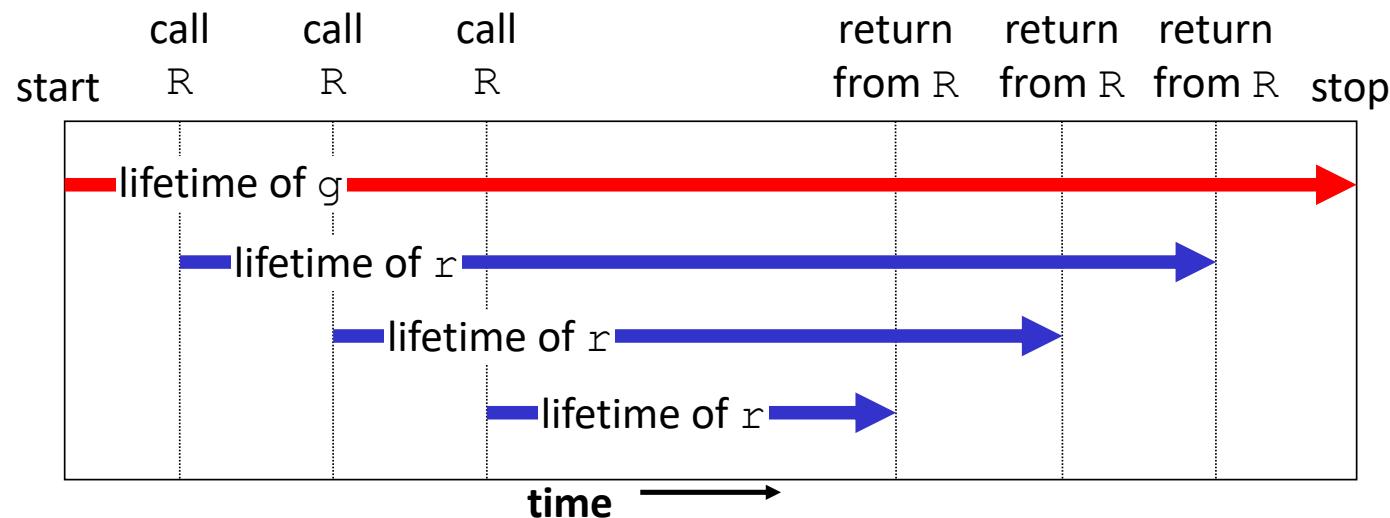
```



# Exemple variables globales/locales dans le cadre de la récursivité

```
void main() {  
    int g;  
    ... R(); ...  
}  
  
void R() {  
    int r;  
    ... R(); ...  
}
```

C++



# Allocation et dé-allocation des variables dynamiques

- Expressions d'allocation

- Réserve, alloue la mémoire nécessaire dans le tas et retourne un pointeur sur cet espace mémoire

Expression `new...`

C++, Java, ADA

fonction `malloc`

C

- Instruction de dés-allocation

- Détruit explicitement une variable dynamique donnée et libère l'espace mémoire qui lui était réservé sur le tas

Expression `delete...`

C++

Aucun moyen de faire de la dé allocation explicite  
→ garbage collector

Java

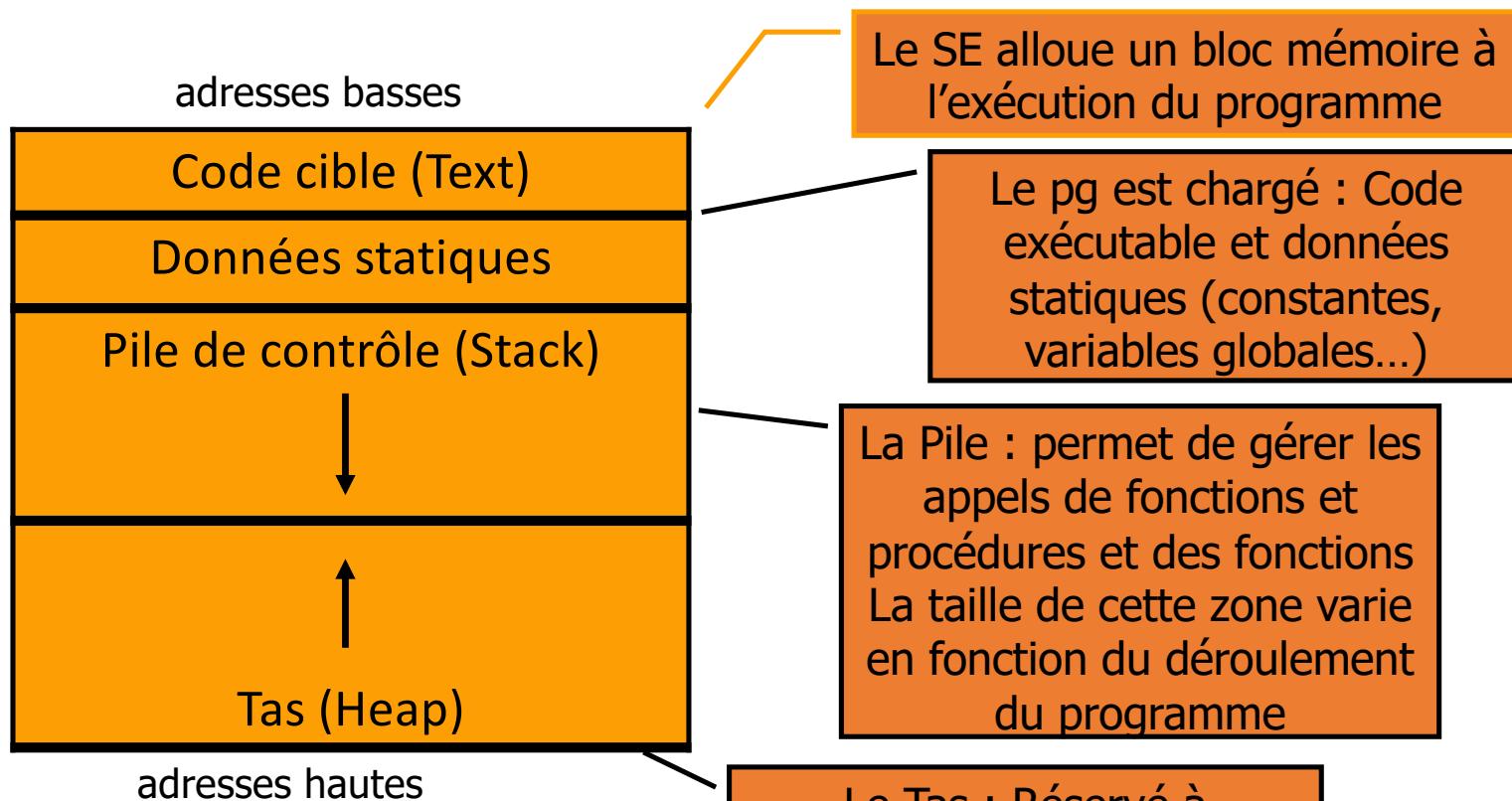
fonction `free...`

C

Possible sous certaine condition (tag du code)

ADA

## Rappel : Organisation de la mémoire à l'exécution, structure interne des processus



Une vidéo pour mieux comprendre  
[https://www.youtube.com/watch?v=\\_8-ht2AKyH4](https://www.youtube.com/watch?v=_8-ht2AKyH4)

# Exemple variables dynamiques

```
procedure main is

type IntNode;
type IntList is access IntNode;
type IntNode is record
    elem: Integer;
    succ: IntList;
end record;

odds, primes: IntList := null;

function cons (h: Integer; t: IntList) return IntList is
begin
    return new IntNode(h,t);
end;

procedure A is
begin
    odds := cons(3, cons(5, cons(7, null)));
    primes := cons(2, odds);
end;

procedure B is
begin
    odds.succ := odds.succ.succ;
end;

begin
... A; ... B; ...
end;
```

Dessinez l'état de la mémoire une fois les procédures A et B exécutées

# Exemple variables dynamiques

```
procedure main is
    type IntNode;
type IntList is access IntNode;
type IntNode is record
    elem: Integer;
    succ: IntList;
end record;

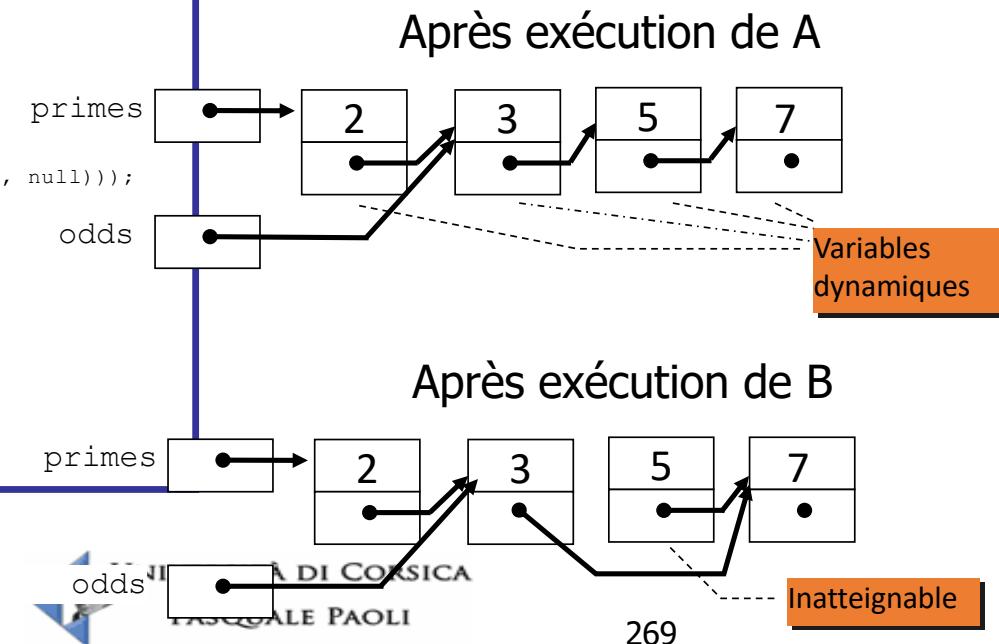
    odds, primes: IntList := null;

    function cons (h: Integer; t: IntList) return IntList
is
begin
    return new IntNode'(h, t);
end;

procedure A is
begin
    odds := cons(3, cons(5, cons(7, null)));
    primes := cons(2, odds);
end;

procedure B is
begin
    odds.succ := odds.succ.succ;
end;

begin
... A; ... B; ...
end;
```



# Exemple variables dynamiques

```
procedure main is

type IntNode;
type IntList is access IntNode;
type IntNode is record
    elem: Integer;
    succ: IntList;
end record;

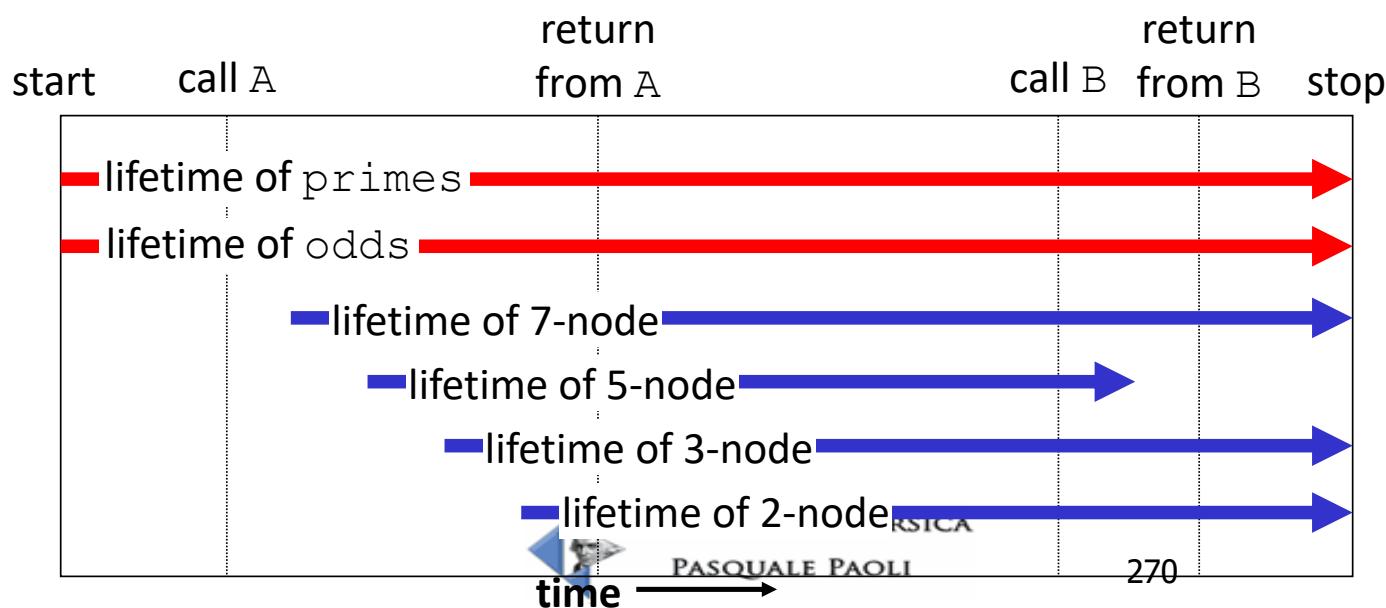
odds, primes: IntList := null;

function cons (h: Integer; t: IntList)
    return IntList is
begin
    return new IntNode(h, t);
end;
```

```
procedure A is
begin
    odds := cons(3, cons(5, cons(7, null)));
    primes := cons(2, odds);
end;

procedure B is
begin
    odds.succ := odds.succ.succ;
end;

begin
... A; ... B; ...
end;
```



## Variables persistantes vs transitoires (persistent vs transient)

- Persistantes : variables dont la durée de vie excède celle de l'exécution du programme
- Transitoires (transient) : variables dont la durée de vie est limitée (liée) à la durée d'exécution du programme
  - Variables globales, locales, dynamiques
- Les fichiers peuvent être vu comme des variables composites
- Deux types
  - Séquentiels : séquence de composants ; lus ou écrits composant par composant de façon séquentielle
  - À accès direct : tableau de composants ; lus ou écrits composant par composant dans un ordre arbitraire en utilisant le numéro de position du composant

# Les pointeurs : définition

- Un pointeur est une référence à une variable donnée (le référent)
- Les pointeurs sont parfois nommés références
- Le pointeur null permet de spécifier qu'il n'y a pas de variable référencée (pas de référent)
- Peut être vu comme l'adresse du référent en mémoire
- Possède un type permettant de connaître le type de la variable référencée

```
int x, y;  
int *ptrx; C  
x=1; y=0;  
ptrx=&x;  
y=*ptrx;
```

```
IntNode* p; C++  
*p  
(*p).elem  
p->elem
```

```
p : access IntNode;  
p.all  
p.all.elem  
p.elem
```

ADA

# Les pointeurs : déréférencement

- Le déréférencement d'un pointeur est l'opération qui consiste à accéder (en lecture ou écriture) à la zone mémoire référencée par le pointeur

```
int x;  
int *ptrx;  
x=1;  
ptrx=&x;  
*ptrx=0; //déréférencement du ptrx, pour affecter 0 à x
```

C

## Des pointeurs... (ça sert à quoi ?)

- Génériques

- Lorsqu'on ne sait pas sur quel type de données le pointeur va pointer...
  - Le type est inconnu et sans importance au moment de la programmation
  - Le type est variable

```
void *pg=NULL;  
int entier = 4; C, C++  
pg=&entier;  
char lettre='c';  
pg=&lettre;
```

Attention !  
On ne peut pas  
dé-référencer  
(\*pg =...) un  
pointeur générique

```
void *pg=NULL;  
int *pi,*pt; C, C++  
int i=6;  
pi=&i;  
pg = pi;  
pt = pg;
```

La conversion  
(transtypage,  
ou cast) est  
implicite

## Des pointeurs... (ça sert à quoi ?)

- Et des tableaux

- Lorsqu'on déclare un tableau, le nom du tableau est considéré comme un pointeur sur le type des éléments contenus dans le tableau

```
double somme(double tab[], ...){ }
```

C

```
double somme(double *tab, ...){ }
```

C

tab est ici l'adresse du  
premier élément du tableau

- Pour voyager dans le tableau  
on utilisera l'arithmétique des  
pointeurs

Qu'est ce que l'arithmétique des pointeurs ?



## Des pointeurs... sur fonctions (ça sert à quoi ?)

- Permet l'appel d'une fonction dont on connaît l'adresse, On doit connaître son type de retour et le type de ses paramètres
  - de transmettre des fonctions en paramètres
  - de constituer des tableaux de fonctions;
  - d'écrire des fonctions retournant des fonctions comme valeur.

```
double (*pf) (double, double);
double max(double a, double b){
    return (a>b)?a:b ;
}
...
pf = &max;
pf = max; // possible
...
double maxDesDeux = *pf(10.5,12.6);
double maxAussi = pf(15.4,12.3);
```

C

Attention à ne pas confondre...  
Ci-dessous fonction qui retourne un pointeur  
sur un double...

double \* pf (double, double);

CA

## Des pointeurs... de pointeurs (ça sert à quoi ?)

Manipuler des tableaux à deux dimensions (ou trois ou...)

Exemple : créer avec un pointeur de pointeur  
une matrice à k lignes et n colonnes

```
main () {  
    int k, n;  
    int **tab;  
    tab = (int**)malloc(k*sizeof(int*));  
    for (i = 0; i < k; i++)  
        tab[i] = (int*)malloc(n * sizeof(int));  
        ....  
    for (i = 0; i < k; i++) free(tab[i]);  
    free(tab);  
}
```

[https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS\\_C/chapitre3.html](https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS_C/chapitre3.html)



## Pointeur pendant (dangling pointer)

Possible en C et C++  
Sécurisé en ADA (code tagué)  
Impossible en Java...

- Pointeur référant (pointant sur) une variable qui a été détruite
- Causes
  - Un pointeur sur une variable dynamique existe toujours même après la variable a été détruite
  - Un pointeur (stocké dans une variable globale) sur une variable locale existe toujours même après qu'on soit sorti de son bloc englobant

# Exemple de pointeur pendant

```
struct Date {int y, m, d;};
Date* dateP;
Date* dateQ;
dateP = (Date*)malloc(sizeof Date);
dateP->y = 2004;
dateP->m = 1;
dateP->d = 1;
dateQ = dateP;
free(dateQ);
dateQ=null;
printf("%d4", dateP->y);
dateP->y = 2005;
```

C

Allocation d'une nouvelle variable

dateQ pointe sur la même variable que dateP

Désallocation de la variable dynamique dateQ et dateP sont maintenant des pointeurs pendants

Erreur la variable pointée n'existe plus...  
Utilisation de pointeur pendant



# Exemple de pointeur pendant

C++

```
int* f() {  
    int fv = 42;  
    return &fv;  
};
```

Variable fv locale à f

Récupération de l'adresse de la variable locale

Destruction de la variable locale fv

```
int* p = f();  
*p = 0;
```

La zone mémoire occupée par la variable locale fv a été libérée.  
La tentative d'affectation échoue et cause une erreur

Exercices sur les pointeurs