PG108
UNIX - Programmation C

1A - Électronique

2023-2024

Rémi Giraud
remi.giraud@enseirb-matmeca.fr

Objectifs

- Apprendre à utiliser un ordinateur en tant que **développeur**
  - Maîtrise du terminal, du système de fichier, etc
  - Maîtrise des commandes UNIX (navigation, scripts, etc)
- Appréhender les bases de l'**algorithmique**
  - Concepts, complexité calculatoire, etc
- Développer des bases solides en **programmation en langage C**
  - De la création, la compilation, jusqu'à l'exécution
  - Apprendre à rechercher l'information par vous-même

Sources

Certains slides sont tirés des supports de :

- Floréal Morandat : https://www.labri.fr/perso/fmoranda/pg101/
- Yannick Bornat : https://yannick-bornat.enseirb-matmeca.fr/

Organisation du module

Cours (CM)	Introduction Base d'algorithmique Programmation en langage C	8x1h20
TD + TP	Tutoriel + TP UNIX Rappels et exercices guidés (C)	6x4h20
TP	Exercices + Projet	6x3h
Examen	Écrit individuel (sur papier)	2h

Note finale : Examen (x0.5) + Contrôle continu et Projet (x0.5)

Règles

Assiduité !

Coder seul

Utiliser un vrai éditeur de code (emacs, atom, VS code, Codium, ...) :
#1 coloration syntaxique, #2 indentation, #3 complétion

Réfléchir en utilisant papier et stylo

Pour vous aider

Poly de cours (A4 et slides) :
https://remi-giraud.enseirb-matmeca.fr/teaching/

Plateforme d'exercices en ligne pour s'entraîner :
https://thor.enseirb-matmeca.fr:4443/

Doc outils et dépôt des codes sur thor :
https://thor.enseirb-matmeca.fr/ruby/projects

Question sur une fonction ? → page de manuel (ex : `man 3 printf`)

Comment coder en C sur votre machine ?

Linux :

- Généralement tout est déjà compris. Pour Ubuntu, installer le paquet build-essential :
  `sudo apt-get install build-essential`

MacOS X :

- Pareil pour MacOSX qui dispose de plus ou moins tout de base. Activer le support en ligne de commande : `xcode-select --install`

Windows : (Choisir une des solutions)

- *TDM-gcc* : suite de compilation basée sur le projet GNU gcc :
  https://jmeubank.github.io/tdm-gcc/
- *QT Creator* : IDE (Environnement de développement intégré) qui embarque un compilateur (*clang*) : https://www.qt.io/download
- Installer une distribution Linux sans dual boot (depuis Windows 10) :
  https://ubuntu.com/tutorials/ubuntu-on-windows#1-overview
- Installer une machine virtuelle Linux :
  https://thor.enseirb-matmeca.fr/ruby/docs/teaching/vmlinux
- Installer un dual boot : (à vos risques et périls)
  https://www.malekal.com/installer-ubuntu-20-04-dual-boot-windows-10/
  Pour les trois dernières méthodes se référer ensuite à la section Linux.

En ligne :

- https://replit.com/languages/c

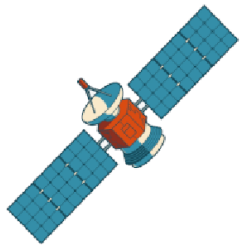
Source : https://www.labri.fr/perso/fmoranda/cathome/

Pourquoi de l'informatique en Électronique ?

- En tant qu'outil :
  - Modélisation / Conception / Réalisation / Test / Caractérisation : TOUT se fait sur ordinateur (station de travail / instrumentation)
  - Le fer à souder n'intervient que pour certains prototypes / petites séries (généralement en labo)

- Pour faire fonctionner ça :




Pourquoi UNIX ?

- Grosse majorité des systèmes d'exploitation UNIX-like
- Toutes les archis matérielles font tourner au moins un système UNIX-like
  - Même environnement / mêmes outils sur ces systèmes
  - Gestion des fichiers identique
- Pas d'interfaces graphiques ni de souris (vite moins efficaces)
  - Moins d'utilisation de ressources matérielles / de bande passante
- On travaille dans un terminal (ou console, ou shell)
  - Meilleure maîtrise de l'exécution des programmes
  - Grande liste de commandes de base (navigation, copie, droits, etc)
- En détails :
  - Les données du système sont dans des fichiers organisés en dossiers
  - Exécution du programme en tapant son nom au clavier

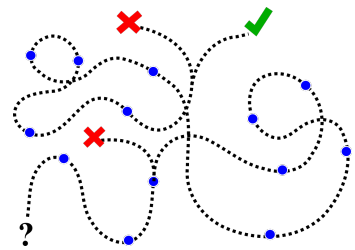
    Exécution associée au dossier/terminal en cours

    Ajout d'options possible à l'exécution du programme

TP : Utilisation UNIX

- Introduction :
  - UNIX, commandes de base
  - Réseau / ressources informatiques de l'ENSEIRB-MATMECA (serveur, connexion à distance)

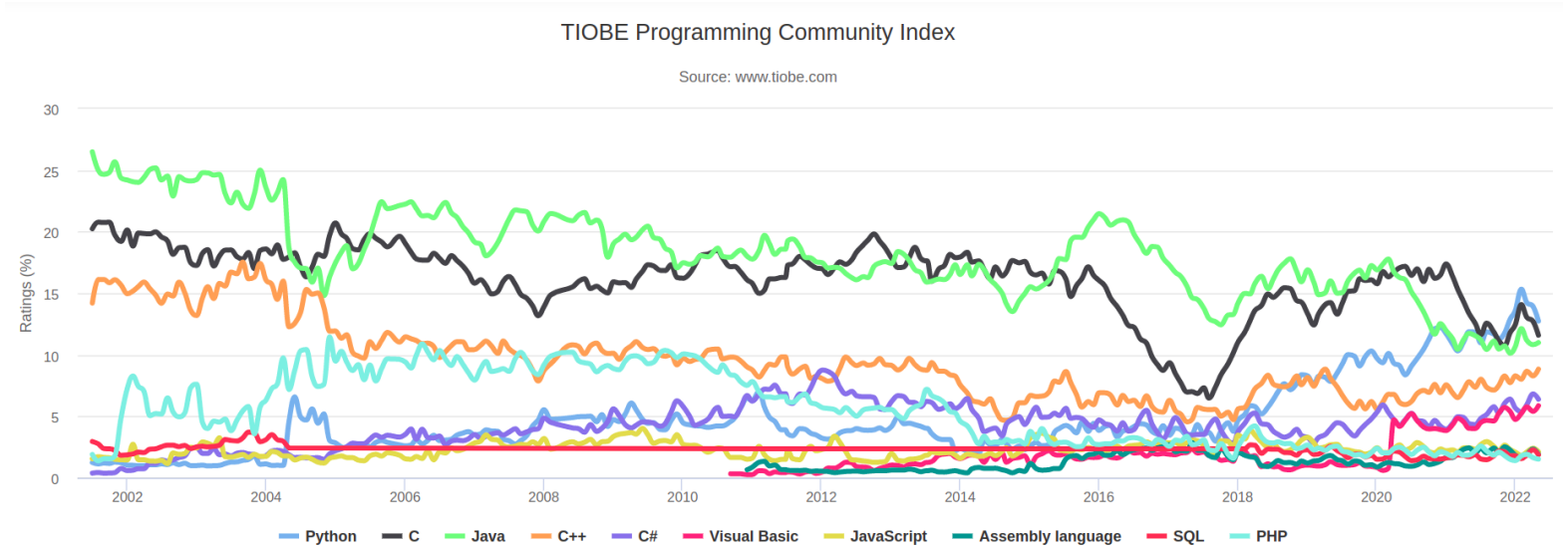
- Jeu de piste par étapes :
  - Chaque étape est associée à un concept à acquérir
  - Chaque étape se valide en exécutant un programme
  - La validation d'une étape débloque la suivante



- TP 1 & 2. Perfectionnement des notions :
  - Arborescence, droits d'accès, variables, scripts

Pourquoi le C ?

- Un des plus utilisés (en informatique pure)



- Privilégié pour sa proximité avec le système (matériel)
- Portable (dans une certaine mesure)
- Quasiment la norme pour les microcontrôleurs, calculateurs matériel, etc
- Développement un peu plus pénible, mais grande efficacité
- Contient de grands principes de programmation généralisables

L'algorithmique c'est quoi ?

– **Algorithme :** séquence d'opérations de calcul élémentaires, organisée selon des règles précises dans le but de résoudre un problème donné.

– **Structures de données :** moyen de stocker et organiser des données pour faciliter l'accès à ces données et leur modification.

- Traduire des concepts en un ensemble de valeurs numériques
  – Définir les règles pour que ces valeurs numériques se comportent comme les concepts d'origine à partir de règles de base
- Écrire ces règles dans un langage non équivoque
  – Rien ne doit être implicite
  – Rien de doit être laissé à l'initiative de l'exécutant (machine ou humain)
- Des compétences à développer au travers de la programmation :
  – Se familiariser avec des problèmes classiques et leur solutions
  – Savoir trouver et comparer les performances de solutions
  – S'entraîner à écrire des algorithmes et choisir les structures de données.

Exemple d'algorithme

Boire son café

- 1. Prenez une dosette de café.
- 2. Mettez-la dans la machine à café.
- 3. Vérifiez si la machine à café est allumée.
  Si ce n'est pas le cas, allumez la machine.
- 4. Vérifiez si le filtre à eau est suffisamment rempli.
  - Si ce n'est pas le cas, ajoutez de l'eau.
- 5. Placez une tasse à café sous le distributeur de café.
- 6. Appuyez sur le bouton café.
- 7. Le café est en train d'être servi.
  - Attendez que la machine indique que le café est prêt.
- 8. Si le café est prêt - Sortez la tasse à café de la machine à café.

Programme

- Un algorithme écrit dans un langage informatique donné
  - En informatique : les opérations sont des actions qui peuvent par ex. :
    - Se répéter
    - Ne pas se produire
- En informatique, des valeurs abstraites (entiers) sont utilisées pour représenter des grandeurs “qui parlent” à l’utilisateur
  - Date, lieu, couleur, nom, ...

La programmation c’est donc

- Comprendre l’algorithme à développer
- Le traduire en opérations élémentaires manipulant des valeurs abstraites
- L’écrire dans la syntaxe du langage choisi

Plan du cours

Le langage C

- Un peu d'histoire :
  - Langage mis au point en 1972 par Kenneth Thomson et Dennis Ritchie
  - Créé pour implémenter le système UNIX de manière portable
- Pourquoi apprendre le C ?
  - Langage "haut niveau" (mais le plus bas), très exigeant en termes de gestion de la mémoire par exemple
  - Très utilisé dans le développement d'applications scientifiques et de développement de systèmes (ex : systèmes embarqués, traitement des signaux et images)
  - Beaucoup de langages dérivés : C++, Objective-C, C#
- Caractéristiques :
  - **Impératif** : suite d'opérations exécutées successivement
  - **Compilé** : utilisation d'un outil (compilateur) pour générer un exécutable, fonctionnel uniquement sur le système où il a été compilé.
  - Typage des variables : explicite, statique (vérification à la compilation), faible (conversions possibles)
  - Différentes version (C89, C99, C11), introduisant quelques simplifications

Bonnes pratiques de codage (cf. 1.4)

```
int function(int k, int t, int v[]) {
int h = 0;
for (int i=0; i<k; i++)
if (v[i]==t) h++;
return h; }
```

Que fait ce code ?

Bonnes pratiques de codage (cf. 1.4)

- Indentez le code *[tab]*

```
int function(int k, int t, int v[]) {
int h = 0;
for (int i=0; i<k; i++)
if (v[i]==t) h++;
return h; }
```

Que fait ce code ?

Bonnes pratiques de codage (cf. 1.4)

- **Indentez le code** *[tab]*

```
int function(int k, int t, int v[]) {
␣ int h = 0;
␣ for (int i=0; i<k; i++)
␣ ␣ if (v[i]==t)
␣ ␣ ␣ h++;
␣ return h;
}
```

Bonnes pratiques de codage (cf. 1.4)

- **Indentez le code** *[tab]*
- Commentez le code `/* Commentaires */` ou `//` (C99)

```
int function(int k, int t, int v[]) {
   int h = 0;
   for (int i=0; i<k; i++)
      if (v[i]==t)
         h++;
   return h;
}
```

Bonnes pratiques de codage (cf. 1.4)

- **Indentez le code** *[tab]*
- **Commentez le code** `/* Commentaires */` ou `//` (C99)

```c
/* Return the number of occurrences
   of val in tab of length len */
int function(int k, int t, int v[]) {
  int h = 0;
  for (int i=0; i<k; i++)
    if (v[i]==t)
      h++;
  return h;
}
```

Bonnes pratiques de codage (cf. 1.4)

- **Indentez le code** *[tab]*
- **Commentez le code** `/* Commentaires */` ou `//` (C99)
- Nommez explicitement vos variables

```
/* Return the number of occurrences
   of val in tab of length len */
int function(int k, int t, int v[]) {
  int h = 0;
  for (int i=0; i<k; i++)
    if (v[i]==t)
      h++;
  return h;
}
```

Bonnes pratiques de codage (cf. 1.4)

- **Indentez le code** *[tab]*
- **Commentez le code** `/* Commentaires */` ou `//` (C99)
- **Nommez explicitement vos variables**

```c
/* Return the number of occurrences
   of val in tab of length len */
int count_occurr(int len, int val, int tab[]) {
  int occ = 0;
  for (int i=0; i<len; i++)
    if (tab[i]==val)
      occ++;
  return occ;
}
```

Bonnes pratiques de codage (cf. 1.4)

- **Indentez le code** *[tab]*
- **Commentez le code** `/* Commentaires */` ou `//` (C99)
- **Nommez explicitement vos variables**
- Profitez de la coloration syntaxique de l'éditeur (.c)

```c
/* Return the number of occurrences
   of val in tab of length len */
int count_occurr(int len, int val, int tab[]) {
  int occ = 0;
  for (int i=0; i<len; i++)
    if (tab[i]==val)
      occ++;
  return occ;
}
```

Bonnes pratiques de codage (cf. 1.4)

- **Indentez le code** *[tab]*
- **Commentez le code** `/* Commentaires */` ou `//` (C99)
- **Nommez explicitement vos variables**
- **Profitez de la coloration syntaxique de l'éditeur (.c)**

```c
/* Return the number of occurrences
   of val in tab of length len */
int count_occurr(int len, int val, int tab[]) {
  int occ = 0;
  for (int i=0; i<len; i++)
    if (tab[i]==val)
      occ++;
  return occ;
}
```

Bonnes pratiques de codage (cf. 1.4)

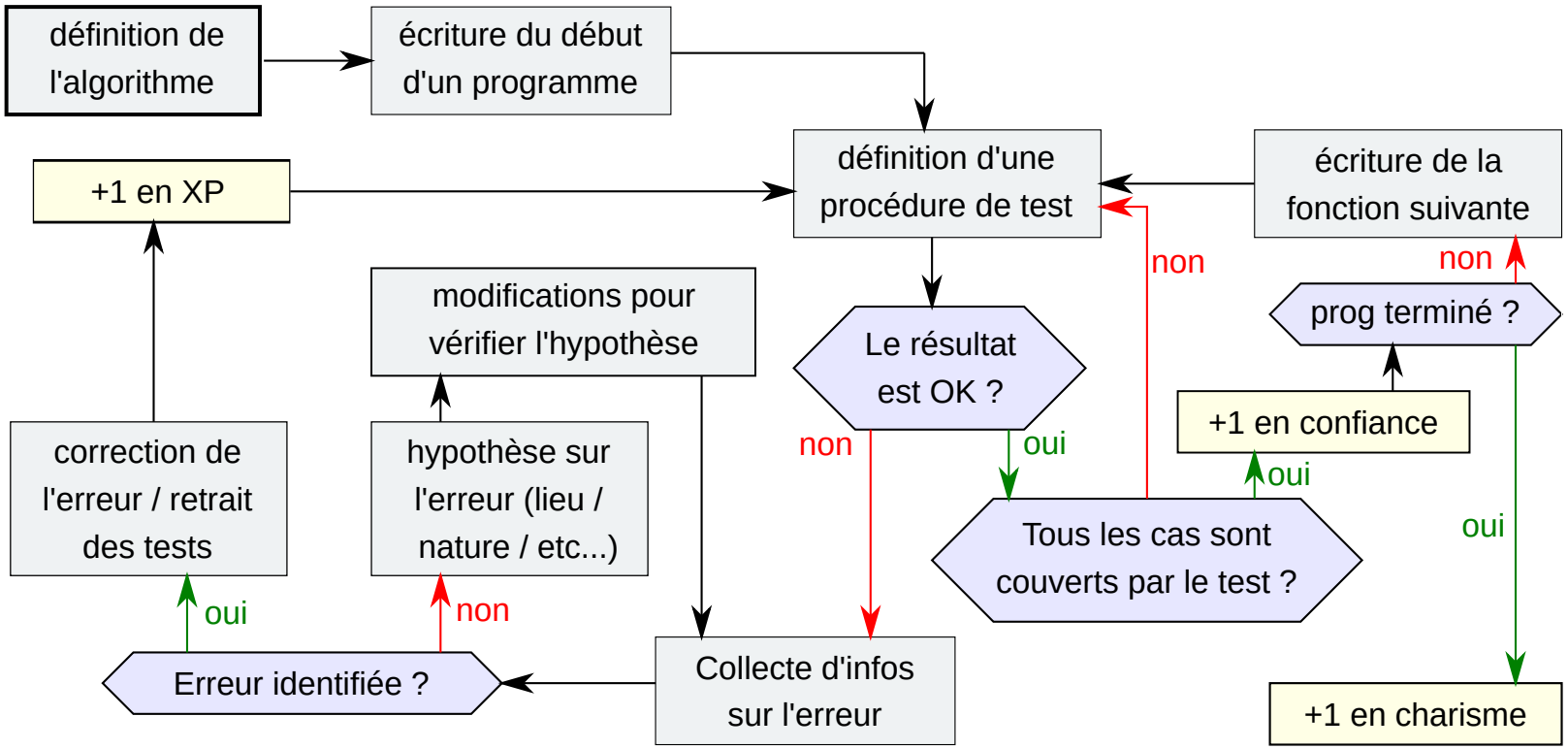
- **Indentez le code** *[tab]*
- **Commentez le code** `/* Commentaires */` ou `//` (C99)
- **Nommez explicitement vos variables**
- **Profitez de la coloration syntaxique de l'éditeur (.c)**
- **Testez en compilant régulièrement**

Attention :

- Écrire en C nécessite de respecter la syntaxe :
  - `;` à la fin des instructions, existence des variables, etc.
  - Indentation et espaces PAS pris en compte
  - Minuscules et majuscules prises en compte

    `printf` $\neq$ `PRINTF`
- Un code qui compile est syntaxiquement correct mais pas forcément bon

Bonnes pratiques de codage (cf. 1.4)



Exemple de code

hello.c

```c
#include <stdio.h>
/* This program prints Hello World */
int main(){
 printf("Hello World\n");
 return 0;
}
```

Exemple de code

hello.c

```c
#include <stdio.h>
/* This program prints Hello World */
int main(){
 printf("Hello World\n");
 return 0;
}
```

Exercice : Premier programme

- Écrivez ce programme à l'aide d'un éditeur de texte approprié
- Compilez ce programme en tapant dans le terminal :
  `gcc hello.c`
- Exécutez ce programme avec la commande suivante :
  `./a.out`

Concept de compilation (cf. 2.1)

Un programme, c'est du texte (humainement compréhensible)

– Le processeur ne comprend pas le texte ("code source", "code", "source")

– Le processeur ne comprend que son propre jeu d'instructions en binaire

– Comment exécuter le programme ?

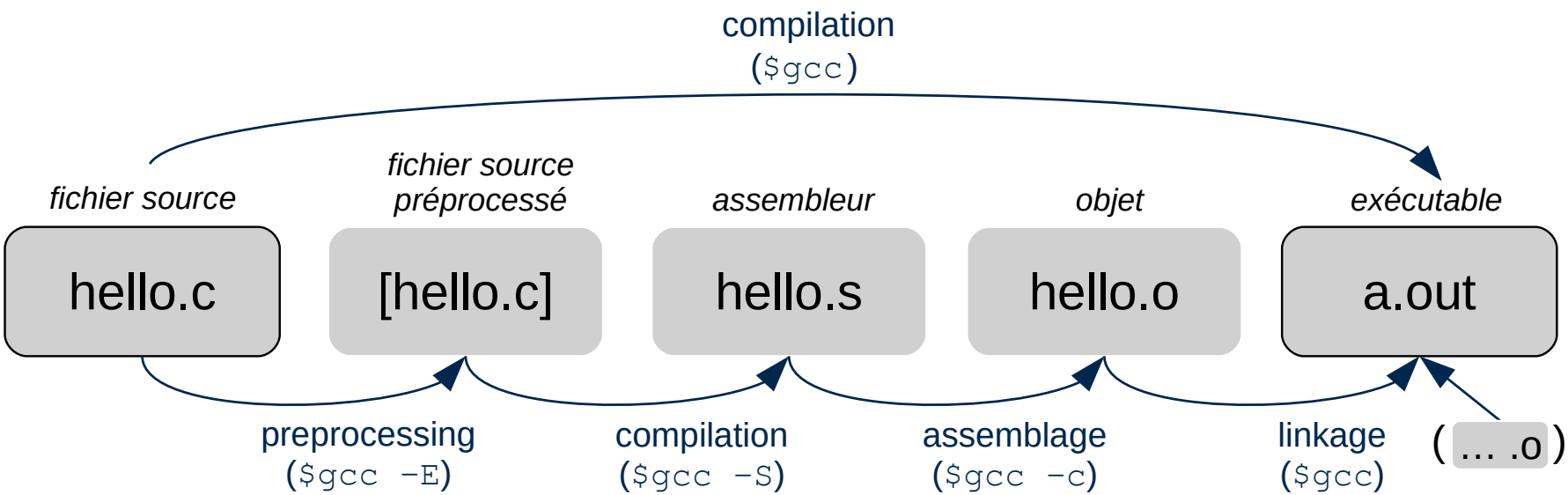
- **Solution 1** : usage d'un interpréteur (ex. : python, Matlab)
  – Un autre programme analyse le code et effectue les opérations
  – Programmes faciles à écrire, mais exécution lente

- **Solution 2** : usage d'un compilateur (ex. : C)
  – Un programme traduit le code source en code binaire que le processeur comprend directement
  – Programme rapide, mais doit être recompilé pour chaque nouvelle architecture

Compilation C (cf. 2.1)

Compilateur C : Traduire le code source (.c) en un exécutable pour la machine
On utilise généralement gcc (`man gcc`)

Chaîne de compilation :



Compilation (cf. 2.1)

Exercice : Préprocessing

- Tapez la commande `gcc -E hello.c`
  (quand cette option est passée au compilateur, ce dernier s'arrête après la phase de preprocessing).
- Trouvez une façon d'écrire automatiquement ce nouveau code source dans un fichier *new_hello.c*).
  Comparez la taille du fichier d'origine avec celle du fichier produit.
  Que constatez-vous ?
- Le fichier produit est-il dépendant du processeur ?
- Le fichier produit est-il dépendant du système d'exploitation ?

Compilation (cf. 2.1)

Exercice : Préprocessing

- Tapez la commande `gcc -E hello.c`
  (quand cette option est passée au compilateur, ce dernier s'arrête après la phase de preprocessing).
- Trouvez une façon d'écrire automatiquement ce nouveau code source dans un fichier *new_hello.c*).
  Comparez la taille du fichier d'origine avec celle du fichier produit.
  Que constatez-vous ?
- Le fichier produit est-il dépendant du processeur ?
- Le fichier produit est-il dépendant du système d'exploitation ?

Exercice : Compilation

- Tapez la commande `gcc -S hello.c`
- Regardez le code assembleur produit (`ls` pour voir quel fichier a été généré).
  Connaissant le rôle du programme d'origine, que reconnaissez-vous dans le code assembleur ?

Compilation (cf. 2.1)

Exercice : Assemblage

- Tapez la commande `gcc -c hello.s` (quand cette option est passée compilateur, ce dernier s'arrête après la phase d'assemblage).
- Regardez le code objet produit, contenu dans le fichier *hello.o*.
  Que constatez-vous ?
- A quoi sert la commande `readelf` ?

Exercice : Édition de liens

- Tapez la commande `gcc hello.o` et exécutez le programme correspondant.
- Comparez avec le code objet de l'étape précédente.
- Exécutez le code (`./a.out`).
- Trouvez comment renommer l'exécutable en sortie lors de la compilation (`man gcc`).

Compilation (cf. 2.2)

Commande de compilation générale

```
gcc [options] fichier.c
```

Commande de compilation minimale

```
gcc fichier.c
```

crée un exécutable nommé par défaut `a.out`

Commande de compilation recommandée

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

`-Wall` = avertissements suppl.
`-o nom` = exé. nommé `nom`

Exécution

```
./hello [argument#1 argument#2 ...]
```

Attention :

- Sauvegarder fréquemment le fichier source (.c)
- Recompiler avant exécution pour regénérer l'exécutable
- Lire les avertissements et erreurs (pour la plupart compréhensibles !)
  Peuvent dépendre du compilateur utilisé
- Attention au nom de l'exécutable

La trilogie

- **Valeur / Variable / Type**

  Toute valeur ou variable a un type
  ex. : `42` de type `int`, `"abc"` de type `const char*`

  Portée des variables :
  De la déclaration → jusqu'à l'accolade fermante

- **Expression**

  Produit une valeur, possède un type
  ex. : `42, 6*7, foo(2, x-1)`

- **Instruction**

  Ne produit pas de valeur
  ex. : structures de contrôle (`if, for, return`)

Une expression est : (cf. 3.1)

- évaluable : elle produit un résultat qui possède une valeur
- typée : ce résultat possède un type

Une expression est : (cf. 3.1)

- évaluable : elle produit un résultat qui possède une valeur
- typée : ce résultat possède un type

Expressions de base :

- **identificateurs** (noms des variables, fonctions, mots-clefs de type, ...)

Liste des mots-clefs du langage C

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

Une expression est : (cf. 3.1)

- évaluable : elle produit un résultat qui possède une valeur
- typée : ce résultat possède un type

Expressions de base :

- **identificateurs** (noms des variables, fonctions, mots-clefs de type, ...)

Liste des mots-clefs du langage C (types de base)

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

Une expression est : (cf. 3.1)

- évaluable : elle produit un résultat qui possède une valeur
- typée : ce résultat possède un type

Expressions de base :

- **identificateurs** (noms des variables, fonctions, mots-clefs de type, ...)

```
int fonction() {
    char a  = 'c';
    char* s = "coucou";
    if(s[0]==a)
      printf("Ok\n");
    return 0;
}
```

Une expression est : (cf. 3.1)

- évaluable : elle produit un résultat qui possède une valeur
- typée : ce résultat possède un type

Expressions de base :

- **identificateurs** (noms des variables, fonctions, mots-clefs de type, ...)

```
int fonction() {
    char a  = 'c';
    char* s = "coucou";
    if(s[0]==a)
      printf("Ok\n");
    return 0;
}
```

Une expression est : (cf. 3.1)

- évaluable : elle produit un résultat qui possède une valeur
- typée : ce résultat possède un type

Expressions de base :

- **identificateurs** (noms des variables, fonctions, mots-clefs de type, ...)
- **constantes** numériques (5, 17.2, 'a', ...)

```
int fonction() {
    char a  = 'c';
    char* s = "coucou";
    if(s[0]==a)
      printf("Ok\n");
    return 0;
}
```

Une expression est : (cf. 3.1)

- évaluable : elle produit un résultat qui possède une valeur
- typée : ce résultat possède un type

Expressions de base :

- **identificateurs** (noms des variables, fonctions, mots-clefs de type, ...)
- **constantes** numériques (5, 17.2, 'a', ...)
- **chaînes de caractères constantes** (déclarées entre " ")

```
int fonction() {
    char a  = 'c';
    char* s = "coucou";
    if(s[0]==a)
      printf("Ok\n");
    return 0;
}
```

Une expression est : (cf. 3.1)

- évaluable : elle produit un résultat qui possède une valeur
- typée : ce résultat possède un type

Expressions de base :

- **identificateurs** (noms des variables, fonctions, mots-clefs de type, ...)
- **constantes** numériques (5, 17.2, 'a', ...)
- **chaînes de caractères constantes** (déclarées entre " ")
- **expressions parenthésées**

```
int fonction() {
    char a  = 'c';
    char* s = "coucou";
    if(s[0]==a)
      printf("Ok\n");
    return 0;
}
```

Expressions composées élémentaires :

- Déclaration de variable : *type nom* (ne doit pas débuter par un chiffre)
  ex : `int x` (déclaration d'une variable `x` de type `int`)
- Affectation de variable : *nom = valeur*
  ex : `x = 0`
  ex : `int y = 2` (déclaration + affectation simultanées)
- Appel de fonctions : *nom ()*
  ex : `my_function()`

Expressions composées élémentaires :

- Déclaration de variable : *type nom* (ne doit pas débuter par un chiffre)
  ex : `int x` (déclaration d'une variable `x` de type `int`)
- Affectation de variable : *nom = valeur*
  ex : `x = 0`
  ex : `int y = 2` (déclaration + affectation simultanées)
- Appel de fonctions : *nom ()*
  ex : `my_function()`

Expressions composées complexes : (cf. 3.2)

- Opérateurs de comparaison et logiques :

  de comparaison : `<` , `>` , `<=` , `>=` , `==` (égalité), `!=` (différence)

  logiques : `||` (OU), `&&` (ET), `!` (négation)

  ex : val. diff. de 0, et inf. ou égale à 100 :

Expressions composées élémentaires :

- Déclaration de variable : *type nom* (ne doit pas débuter par un chiffre)
  ex : `int x` (déclaration d'une variable `x` de type `int`)
- Affectation de variable : *nom = valeur*
  ex : `x = 0`
  ex : `int y = 2` (déclaration + affectation simultanées)
- Appel de fonctions : *nom ()*
  ex : `my_function()`

Expressions composées complexes : (cf. 3.2)

- Opérateurs de comparaison et logiques :

  de comparaison : `<` , `>` , `<=` , `>=` , `==` (égalité), `!=` (différence)

  logiques : `||` (OU), `&&` (ET), `!` (négation)

  ex : val. diff. de 0, et inf. ou égale à 100 : `((x != 0) && (x <= 100))`

Expressions composées élémentaires :

- Déclaration de variable : *type nom* (ne doit pas débuter par un chiffre)
  ex : `int x` (déclaration d'une variable `x` de type `int`)
- Affectation de variable : *nom = valeur*
  ex : `x = 0`
  ex : `int y = 2` (déclaration + affectation simultanées)
- Appel de fonctions : *nom ()*
  ex : `my_function()`

Expressions composées complexes : (cf. 3.2)

- Opérateurs de comparaison et logiques :

  de comparaison : `<` , `>` , `<=` , `>=` , `==` (égalité), `!=` (différence)

  logiques : `||` (OU), `&&` (ET), `!` (négation)

  ex : val. diff. de 0, et inf. ou égale à 100 : `((x != 0) && (x <= 100))`

- Opérateurs d'affectation combinée : `-=` , `+=` , `/=` , `*=` , etc.
  ex : `x = x / 2` équivalent à `x /= 2`

Les instructions (cf. 3.3)

- Instructions simples : expressions + caractère de terminaison `;`
  ex : `int a = 5;` ex : `;` (instruction vide)

Les instructions (cf. 3.3)

- Instructions simples : expressions + caractère de terminaison `;`
  ex : `int a = 5;` ex : `;` (instruction vide)
- Instructions groupées (blocs) - délimitées par des accolades `{ }`
  ex : `int a = 5;`
  `{ int a = 4; }` Affecte la visibilité/portée des variables

Les instructions (cf. 3.3)

- Instructions simples : expressions + caractère de terminaison `;`
  ex : `int a = 5;` ex : `;` (instruction vide)
- Instructions groupées (blocs) - délimitées par des accolades `{ }`
  ex : `int a = 5;`
  `{ int a = 4; }` Affecte la visibilité/portée des variables
- Instructions étiquetées, de saut, ... (cf. 5.3.2, 6)

Les instructions (cf. 3.3)

- Instructions simples : expressions + caractère de terminaison `;`
  ex : `int a = 5;` ex : `;` (instruction vide)
- Instructions groupées (blocs) - délimitées par des accolades `{ }`
  ex : `int a = 5;`
  `{ int a = 4; }` Affecte la visibilité/portée des variables
- Instructions étiquetées, de saut, ... (cf. 5.3.2, 6)
- Instructions de branchement (structures conditionnelles) (cf. 5.1)
  - `switch (expression)` ...
  - `if (expression) instruction [else instruction]`

  ex :
```
if (a!=0)
  printf("a non nul\n");
else {
  printf("a nul\n");
}
```

Les instructions (cf. 3.3)

- Instructions simples : expressions + caractère de terminaison `;`
  ex : `int a = 5;` ex : `;` (instruction vide)
- Instructions groupées (blocs) - délimitées par des accolades `{ }`
  ex : `int a = 5;`
  `{ int a = 4; }` Affecte la visibilité/portée des variables
- Instructions étiquetées, de saut, ... (cf. 5.3.2, 6)
- Instructions de branchement (structures conditionnelles) (cf. 5.1)
  - `switch (expression)` ...
  - `if (expression) instruction [else instruction]`

  ex :
```
if (a!=0)
  printf("a non nul\n");
else {
  printf("a nul\n");
}
```

Attention :

- Sans blocs, `if/(else)` considèrent la première instruction suivante
  Pour éviter les erreurs, on mettra d'abord tout le temps les accolades

Les instructions (cf. 3.3)

Comment est géré le Vrai / Faux ?

- Il n'existe pas de vrai booléen en C → Annexe : Type booléen
  → **Toute valeur non nulle est considérée logiquement vraie**
- Un test logique renvoie l'entier 1 si vrai, 0 sinon
  ex : `int a = 3 > 2; // a vaut 1`
  `int b = 3 < 2; // b vaut 0`
- Toute expression ou variable est directement évaluable
  ex : `if (3 > 2)` {...} (test OK)
  `if (a)` {...} (test OK)
  `if (b)` {...} (test KO)
  `if (-2)` {...} (test OK, attention aux valeurs négatives vraies)

Pour aller plus loin

- Le `if` teste si la valeur de l'expression ou de la variable est différent de 0
  On a donc équivalence : `if (a!=0)` ≡ `if (a)`
  Et également : `if (a==0)` ≡ `if (!a)`

- Instructions d'itération (structures répétitives) (cf. 5.2)
  - `for (`*`[expression]`* `;` *`[expression]`* `;` *`[expression]`*`)` {...}
  - `while (`*`expression`*`) {`*`instructions`*`}`
    (exécute `instructions` tant que `expression` est vrai)

```c
#include<stdio.h>
/*This program computes a product*/
int main(){
  int a = 5, b = 3;
  int res = 0;
  while(b > 0) {
    res = res + a;
    b = b - 1;
  }
  printf("Résultat: %d\n", res);
  return 0;
}
```

Attention :

- Boucles infinies : s'assurer que l'expression devient fausse à un moment
- Erreur courante : oublier les { }

→ [Ctrl+c] pour arrêter l'exécution

Exercice : Boucle : Modifiez ce code pour créer un programme qui calcule la somme de N premiers entiers et affiche le résultat.

Pour aller plus loin

- do {*instructions*} while (*expression*);
  (exécute *instructions* une fois puis tant que *expression* est vrai)

La fonction `printf` (cf. 4.1)

- Affiche une chaîne de caractères (guillemets) : `printf("bonjour\n");`
- Pour aller à la ligne : `'\n'`
- On peut inclure dans la chaîne des évaluations d'expressions, de variables, données en arguments et positionnées dans la chaîne par un code `%<code>`

Syntaxe :

```
printf("texte [et codes]", [arg1], [arg2], ...);
```

Exemples :

- Valeur numérique :
```
printf("la valeur est %d", 42);
```
- Plusieurs valeurs numériques : (arguments pris dans l'ordre)
```
int a = 10;
printf("les valeurs sont %d et %d\n", a, 2021*2);
```
- Chaîne de caractères :
```
printf("Mon nom est %s\n", "superman");
```
- Valeur numérique et texte :
```
printf("Monsieur %s a %d ans\n", "superman", 42);
```

Types - Affichages avec `printf` (cf. 4.1)

	Type	Taille (LP64)	code `printf`
Entiers	`char` `unsigned char`	1 octet	`%c` (caractère ASCII) `%hhi` (décimal) `%c` (caractère ASCII) `%hhu` (décimal)
	`short (int)` `unsigned short (int)`	2 octets	`%hd` `%hu`
	`int` `unsigned (int)`	4 octets	`%d` `%u`
	`long (int)` `unsigned long (int)`	8 octets	`%ld` `%lu`
Flottants	`float`	4 octets	`%f` (décimal) , `%e` (scientifique)
	`double`	8 octets	`%lf` , `%lg`
	`long double`	8 octets	`%Lf` , `%Lg`

Pour les chaînes de caractères (type `char*`) : `%s`

Pas de vrai type booléen sur 1 bit → Annexe : Type booléen

Types(cf. 4.1)

Attention :

- Les tailles dépendent de la plateforme (connaissables avec `sizeof`)
  ex. :
  ```
  sizeof(int); //Renvoie généralement 4
  int a;
  sizeof(a); //Renvoie généralement 4
  ```
- Attention au code d'affichage utilisé
  ```
  float a = 0.5;
  printf("%d (int) - %f (float)\n", a, a); //Affiche : 0 - 0.5
  ```
- Pour comparer des flottants :
  `if (a == b)` → `if (fabs(a - b) < seuil)`
- Ne pas confondre un caractère `char` désigné avec des single quotes ' ' et une chaîne de caractères `char*` déclarées avec des guillemets " " :
  ```
  char c = 'A';
  char* str = "hello";
  ```
- Le type `void` ... que l'on verra plus tard

Représentation des entiers

- **Entiers non signés** :

1 octet = 8 bits =

0	0	0	0	1	1	0	1
7	6	5	4	3	2	1	0

$= 2^3 + 2^2 + 2^0 = 13$

Propriétés : Possibilité de représenter des entiers entre $[0\,;\,2^n\text{-}1]$ sur n bits
Opérations arithmétiques bit à bit

Représentation des entiers

- **Entiers non signés** :

1 octet = 8 bits =

0	0	0	0	1	1	0	1
7	6	5	4	3	2	1	0

$= 2^3 + 2^2 + 2^0 = 13$

Propriétés : Possibilité de représenter des entiers entre [0 ; 2^n-1] sur n bits
Opérations arithmétiques bit à bit

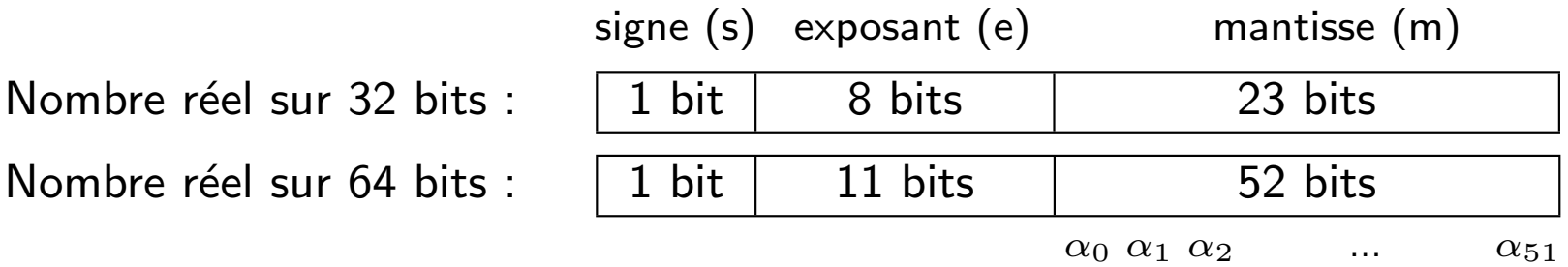
- **Entiers signés** :

Complément à 2 : miroir sur tous les bits et + 1 (X + (-X) = 0)

Représentation binaire								Non signé	Signé
0	0	0	0	1	1	0	1	13	13
1	1	1	1	0	0	1	0	242	-14
1	1	1	1	0	0	1	1	243	-13

Propriétés : Représentation des entiers signés entre [-2^{n-1} ; 2^{n-1} -1] sur n bits
Opérations arithmétiques identiques bit à bit

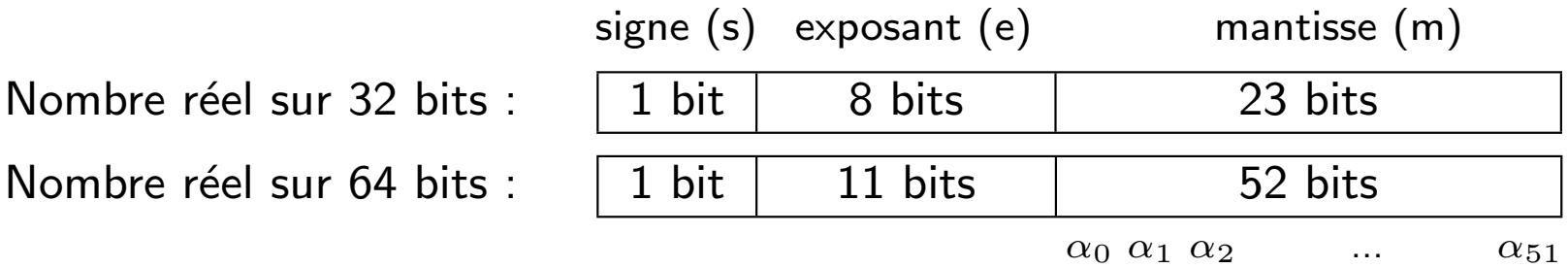
Représentation des flottants



$$\text{Valeur} = (-1)^{s} * m * 2^{(e - (2^{n-1} - 1))}$$

- signe (s) : bit à 0 ou 1 (signe resp. 1 ou -1)
- exposant (e) : entier non signé $\in [0 ; 2^n - 1]$
  n=8 donc $2^{n-1} - 1 = 127$ (32 bits)
  n=11 donc $2^{n-1} - 1 = 1023$ (64 bits)
- mantisse (m) : entier non signé $\sum_i \alpha_i 2^{-i}$ avec $\alpha_0 = 1$

Représentation des flottants



$$\text{Valeur} = (-1)^s * m * 2^{(e - (2^{n-1} - 1))}$$

- signe (s) : bit à 0 ou 1 (signe resp. 1 ou -1)
- exposant (e) : entier non signé $\in [0 ; 2^n - 1]$
  n=8 donc $2^{n-1} - 1 = 127$ (32 bits)
  n=11 donc $2^{n-1} - 1 = 1023$ (64 bits)
- mantisse (m) : entier non signé $\sum_i \alpha_i 2^{-i}$ avec $\alpha_0 = 1$

Cas particuliers :

0 : $m = 0$; $e = 0$ ∞ : $m = 0$; $e = 111...1$ NaN : $m \neq 0$; $e = 111...1$

Représentation des flottants

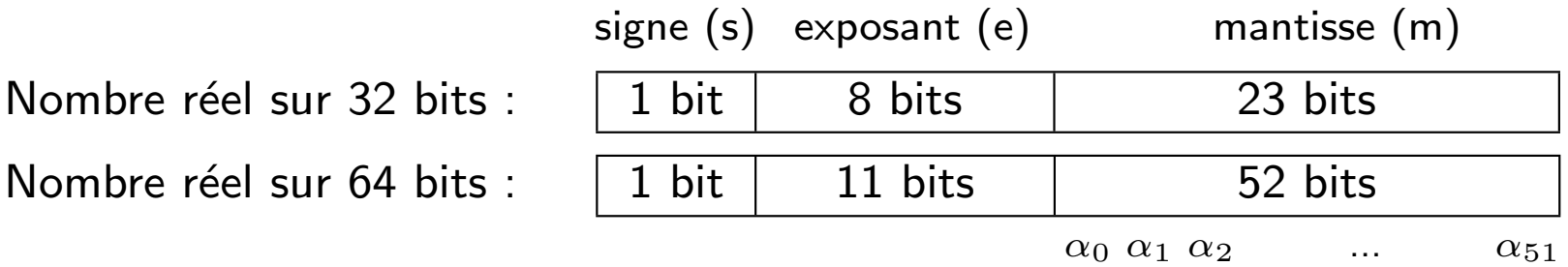
	signe (s)	exposant (e)	mantisse (m)
Nombre réel sur 32 bits :	1 bit	8 bits	23 bits
Nombre réel sur 64 bits :	1 bit	11 bits	52 bits
		α_0 α_1 α_2	... α_{51}

$$\text{Valeur} = (-1)^s * m * 2^{(e - (2^{n-1} - 1))}$$

- signe (s) : bit à 0 ou 1 (signe resp. 1 ou -1)
- exposant (e) : entier non signé $\in [0 ; 2^n - 1]$
  n=8 donc $2^{n-1} - 1 = 127$ (32 bits)
  n=11 donc $2^{n-1} - 1 = 1023$ (64 bits)
- mantisse (m) : entier non signé $\sum_i \alpha_i 2^{-i}$ avec $\alpha_0 = 1$

Exemple (32 bits) : $-18,625 = ???$

Représentation des flottants



$$\text{Valeur} = (-1)^{s} * m * 2^{(e - (2^{n-1} - 1))}$$

- signe (s) : bit à 0 ou 1 (signe resp. 1 ou -1)
- exposant (e) : entier non signé $\in [0 ; 2^n - 1]$
  n=8 donc $2^{n-1} - 1 = 127$ (32 bits)
  n=11 donc $2^{n-1} - 1 = 1023$ (64 bits)
- mantisse (m) : entier non signé $\sum_i \alpha_i 2^{-i}$ avec $\alpha_0 = 1$

Exemple (32 bits) : $-18,625 = |1|$
$s = 1$;

Représentation des flottants

	signe (s)	exposant (e)	mantisse (m)
Nombre réel sur 32 bits :	1 bit	8 bits	23 bits
Nombre réel sur 64 bits :	1 bit	11 bits	52 bits
		α_0 α_1 α_2	... α_{51}

$$\text{Valeur} = (-1)^s * m * 2^{(e - (2^{n-1} - 1))}$$

- signe (s) : bit à 0 ou 1 (signe resp. 1 ou -1)
- exposant (e) : entier non signé $\in [0 ; 2^n - 1]$
  n=8 donc $2^{n-1} - 1 = 127$ (32 bits)
  n=11 donc $2^{n-1} - 1 = 1023$ (64 bits)
- mantisse (m) : entier non signé $\sum_i \alpha_i 2^{-i}$ avec $\alpha_0 = 1$

Exemple (32 bits) : $-18,625 = |1|$

$s = 1$;

$$-18,625 = -(16 + 2 + 0.5 + 0.125) = -(2^4 + 2^1 + 2^{-1} + 2^{-3})$$
$$= -(1 + 2^{-3} + 2^{-5} + 2^{-7}) * 2^4$$

Représentation des flottants

	signe (s)	exposant (e)	mantisse (m)
Nombre réel sur 32 bits :	1 bit	8 bits	23 bits
Nombre réel sur 64 bits :	1 bit	11 bits	52 bits
		α_0 α_1 α_2	... α_{51}

$$\text{Valeur} = (-1)^s * m * 2^{(e - (2^{n-1} - 1))}$$

- signe (s) : bit à 0 ou 1 (signe resp. 1 ou -1)
- exposant (e) : entier non signé $\in [0 ; 2^n - 1]$
  n=8 donc $2^{n-1} - 1 = 127$ (32 bits)
  n=11 donc $2^{n-1} - 1 = 1023$ (64 bits)
- mantisse (m) : entier non signé $\sum_i \alpha_i 2^{-i}$ avec $\alpha_0 = 1$

Exemple (32 bits) : $-18,625 = |1|10000011|$
s = 1 ; 4 = e - 127 $\rightarrow$ e = 131 (10000011) ;

$$-18,625 = -(16 + 2 + 0.5 + 0.125) = -(2^4 + 2^1 + 2^{-1} + 2^{-3})$$
$$= -(1 + 2^{-3} + 2^{-5} + 2^{-7}) * 2^4$$

Représentation des flottants

	signe (s)	exposant (e)	mantisse (m)
Nombre réel sur 32 bits :	1 bit	8 bits	23 bits
Nombre réel sur 64 bits :	1 bit	11 bits	52 bits
	α_0	α_1 α_2 ...	α_{51}

$$\text{Valeur} = (-1)^s * m * 2^{(e - (2^{n-1} - 1))}$$

- signe (s) : bit à 0 ou 1 (signe resp. 1 ou -1)
- exposant (e) : entier non signé $\in [0 ; 2^n - 1]$
  n=8 donc $2^{n-1} - 1 = 127$ (32 bits)
  n=11 donc $2^{n-1} - 1 = 1023$ (64 bits)
- mantisse (m) : entier non signé $\sum_i \alpha_i 2^{-i}$ avec $\alpha_0 = 1$

Exemple (32 bits) : $-18,625 = |1|10000011|(1)00101010000000000000|$
$s = 1$; $4 = e - 127 \rightarrow e = 131$ (10000011) ; $m = (1)00101010...0$;

$$-18,625 = -(16 + 2 + 0.5 + 0.125) = -(2^4 + 2^1 + 2^{-1} + 2^{-3})$$
$$= -(1 + 2^{-3} + 2^{-5} + 2^{-7}) * 2^4$$

Représentation des flottants

	signe (s)	exposant (e)	mantisse (m)
Nombre réel sur 32 bits :	1 bit	8 bits	23 bits
Nombre réel sur 64 bits :	1 bit	11 bits	52 bits
	α_0	α_1 α_2 ...	α_{51}

$$\text{Valeur} = (-1)^s * m * 2^{(e - (2^{n-1} - 1))}$$

- signe (s) : bit à 0 ou 1 (signe resp. 1 ou -1)
- exposant (e) : entier non signé $\in [0 ; 2^n - 1]$
  n=8 donc $2^{n-1} - 1 = 127$ (32 bits)
  n=11 donc $2^{n-1} - 1 = 1023$ (64 bits)
- mantisse (m) : entier non signé $\sum_i \alpha_i 2^{-i}$ avec $\alpha_0 = 1$

Exemple : $0,1$ en base 2

$$\begin{aligned} 0,1_{10} &= b_1 2^0 + b_1 2^{-1} + b_2 2^{-2} + \ldots \\ &= 0\times 1 + 0\times 0,5 + 0\times 0,25 + 0\times 0,125 + 1\times 0,00625 \\ &\quad + 1\times 0,03125 + 0\times \ldots \\ &= 0,0\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\ \ldots \end{aligned}$$

Représentation des flottants

	signe (s)	exposant (e)	mantisse (m)
Nombre réel sur 32 bits :	1 bit	8 bits	23 bits
Nombre réel sur 64 bits :	1 bit	11 bits	52 bits
			$\alpha_0\ \alpha_1\ \alpha_2 \quad \ldots \quad \alpha_{51}$

$$\text{Valeur} = (-1)^s * m * 2^{(e - (2^{n-1} - 1))}$$

- signe (s) : bit à 0 ou 1 (signe resp. 1 ou -1)
- exposant (e) : entier non signé $\in [0\,;\, 2^n - 1]$
  n=8 donc $2^{n-1} - 1 = 127$ (32 bits)
  n=11 donc $2^{n-1} - 1 = 1023$ (64 bits)
- mantisse (m) : entier non signé $\sum_i \alpha_i 2^{-i}$ avec $\alpha_0 = 1$

Exemple : $0,1$ en base 2 n'a pas de représentation finie...

$$\begin{aligned} 0,1_{10} &= b_1 2^0 + b_1 2^{-1} + b_2 2^{-2} + \ldots \\ &= 0\times 1 + 0\times 0,5 + 0\times 0,25 + 0\times 0,125 + 1\times 0,00625 \\ &\quad + 1\times 0,03125 + 0\times \ldots \\ &= 0,0\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,1\ \ldots \end{aligned}$$

Conversion de types (cf. 4.1.3)

Toute expression est typée. Conversion en cas de types hétérogènes.

- Conversion automatique : règles implicites

  Le calcul se fait dans le type de la variable de plus haute précision

  - D'abord, si une opérande de type `long double`, alors l'autre convertie en `long double` ;
  - Ensuite, si une opérande de type `double`, alors l'autre convertie en `double` ;
  - Ensuite, si une opérande de type `float`, alors l'autre convertie en `float` ;
  - Ensuite, convertir les types `char` et `short` en `int` ;
  - Enfin, si une opérandes de type `long`, alors l'autre convertie en `long`

```
ex. : int   a = 7;      // a de type int
      int   b = 2;      // b de type int
      float w = a/b;    // w vaut 3 (opération entière)
      float x = 7/2;    // x vaut 3 (7, 2 de type int)
      float y = 7/2.;   // y vaut 3.5 (2. de type double)
```

- Conversion explicite : *cast*

```
ex. : float z = (float)a/b;   // z vaut 3.5 (a → type float)
```

Exercice : Overflow sur les entiers

Définition : Un overflow se produit sur des entiers quand on cherche à représenter un nombre trop grand en valeur absolue pour le codage choisi.

- 1. Écrire un programme qui déclare un `char` et l'initialise à 1, suivi d'une boucle qui teste si cet entier est positif et dans laquelle on affiche et incrémente cet entier. L'overflow se produit inéluctablement et on finit par obtenir une valeur négative.

  Modifier légèrement le code pour déterminer l'intervalle de valeurs représentables à l'aide d'un `char`. On affichera les bornes de cet intervalle.
- 2. Faire de même avec un `short`, `int`, et un `unsigned char`. Pour ce dernier, on trouvera un overflow quand la séquence de valeur revient à 0.
- 3. Écrire un code qui fait le calcul suivant et affiche toutes les valeurs intermédiaires de x :

```
char x=120;
x = x +120;
x = x-75;
x = x-87;
```

- 4. Pourquoi le résultat est-il juste ?

Exercice : Overflow/underflow sur les flottants

- 1. Écrire un programme qui calcule les éléments de la suite $s_n = 2 * s_{n-1} + 1$ et $s_0 = 1$ pour n entre 1 et 128. On utilisera des variables de type `float`. On affichera pour chaque valeur de n la valeur de s_n.
- 2. Pour quelle valeur de n se produit un overflow ? Pourquoi ?
- 3. Tous les éléments de la suite sont normalement impairs. Est-ce le cas et à partir de quel n les éléments ne le sont plus ? Pourquoi ?
- 4. Changer dans le code précédent le type des éléments en double. Trouver quelle est l'exposant maximum représentable et la taille de la mantisse.
- 5. Faites le calcul suivant :

```
float a[]={1, 5, 104, 1020, 20531, 6543023, 6940381931,
           94932010553101};
float s=0;
for (int i=0; i<8; i++) {
  s = s + a[i]-a[7-i];
}
```

Quel devrait être le résultat de ce calcul ? Combien obtenez-vous ?

Premiers pas en C

Résumé :

- Commande de compilation recommandée :
  `gcc -Wall -o hello hello.c` `./hello` (exécution)
- On doit ajouter un `;` à la fin de chaque instruction :
  `int y;`//décla. `y = 3;`//affect. `int x = 2;`//décla.+affect.
- La taille d'une variable dépend du système :
  `int a; sizeof(a); sizeof(int);` //Renvoie **généralement** 4
- Domaine de valeurs signé / non-signé
  `unsigned type` == $[0, 2^{\text{sizeof(type)*8}}\text{-}1]$
  `(signed) type` == $[\text{-}2^{\text{sizeof(type)*8-1}}, 2^{\text{sizeof(type)*8-1}}\text{-}1]$
- Utiliser des fonctions dans des librairies standards : `#include<>`
  Ex. : `#include<stdio.h>` //qui contient notamment printf
- Pour afficher du texte, des variables :
  Syntaxe : `int printf("texte", [arg1], [arg2], ...);`
  Ex. : `printf("%s a %d ans\n", "Superman", 42);`

Premiers pas en C

Résumé :

- Toute valeur non nulle est considérée logiquement vraie :

```
int x = -2;
if (x!=0) {...}  //est OK
if (x) {...} //équivalent (on évalue directement la variable)
```

- Sans {}, `if/(else)` ne considèrent que l'instruction suivante
  Si plus d'une instruction dans un `if/(else)`, on met des accolades, car les espaces et les tabulations ne sont PAS pris en compte en C :

```
int function_1(int x){
 if(x<0)
   printf("x négatif\n");
   x = -x;
 return x;
}
```

```
int function_2(int x){
 if(x<0) {
   printf("x négatif\n");
 }
 x = -x;
 return x;
}
```

  Sans accolades, la fonction_1 est donc équivalente à la fonction_2 et change le signe de `x` quelle que soit sa valeur d'entrée

Fonctions

Fonction `main` obligatoire = point d'entrée du programme

Pour faciliter la lisibilité, la maintenance et réduire la longueur des codes en permettant de réutiliser des fonctionnalités, on utilise d'autres fonctions

Conseils :

- Envisagez l'écriture d'une fonction dès qu'un copier/coller est nécessaire
- Préférez les fonctions courtes et génériques
- Réduisez à l'essentiel le nombre de paramètres
- Donnez des noms explicites aux fonctions (avec commentaires)
- Faites attention à la visibilité des variables

Attention :

- Il est impossible d'écrire des instructions en dehors du corps d'une fonction, à l'exception de la déclaration de variables globales (cf. 4.3.2) ou de prototypes de fonctions (cf. 6.1)

Déclaration de fonctions (cf. 6.1)

- Syntaxe : *type-de-retour nom([type1 param1], [...])* *prototype* { */* instructions */* } *corps*

ex : `int main() { ... }` `float pow_f(float a, int power) {...}`

Déclaration de fonctions (cf. 6.1)

- Syntaxe : *type-de-retour nom([type1 param1], [...])* *prototype* { */* instructions */* } *corps*

  ex : `int main() { ... }` `float pow_f(float a, int power) {...}`

- Visibilité : Pour que le compilateur puisse analyser une fonction, elle doit avoir été déclarée auparavant dans le programme (ex. gauche)
  Possibilité de simplement déclarer au préalable son prototype (ex. droite)

```c
#include<stdio.h>
int ma_fonction(int argument) {
  int var_locale = 0;
  var_locale += 2*argument + 33;
  return var_locale;
}
int main() {
  int truc = 47;
  truc = ma_fonction(truc);
  printf("Résultat : %d\n", truc);
  return 0;
}
```

```c
#include<stdio.h>
int ma_fonction(int argument);
int main() {
  int truc = 47;
  truc = ma_fonction(truc);
  printf("Résultat : %d\n", truc);
  return 0;
}
int ma_fonction(int argument) {
  int var_locale = 0;
  var_locale += 2*argument + 33;
  return var_locale;
}
```

Déclaration de fonctions (cf. 6.1)

- Syntaxe : *type-de-retour nom([type1 param1], [...])* *prototype* { */* instructions */* } *corps*

  ex : `int main() { ... }` `float pow_f(float a, int power) {...}`

- Visibilité : Pour que le compilateur puisse analyser une fonction, elle doit avoir été déclarée auparavant dans le programme (ex. gauche)
  Possibilité de simplement déclarer au préalable son prototype (ex. droite)

```c
#include<stdio.h>
int ma_fonction(int argument) {
  int var_locale = 0;
  var_locale += 2*argument + 33;
  return var_locale;
}
int main() {
  int truc = 47;
  truc = ma_fonction(truc);
  printf("Résultat : %d\n", truc);
  return 0;
}
```

```c
#include<stdio.h>
int ma_fonction(int argument);
int main() {
  int truc = 47;
  truc = ma_fonction(truc);
  printf("Résultat : %d\n", truc);
  return 0;
}
int ma_fonction(int argument) {
  int var_locale = 0;
  var_locale += 2*argument + 33;
  return var_locale;
}
```

Pour aller plus loin

Visibilité restreinte au fichier : mot-clef `static` (cf. 6.2.2)

Exercice : Ma première fonction

- Écrire un programme qui contient un `main` ET une fonction `polynome` qui reçoit un réel de type `double` et qui retourne un réel de type `double`.

  Son prototype est le suivant :

  ```
  double polynome(double x);
  ```

  Cette fonction est appelée depuis le `main` et retourne la valeur du polynôme $P(x)$:
  $P(x) = x^3 + 3 * x^2 + x + 1$, x étant le réel fourni à la fonction.

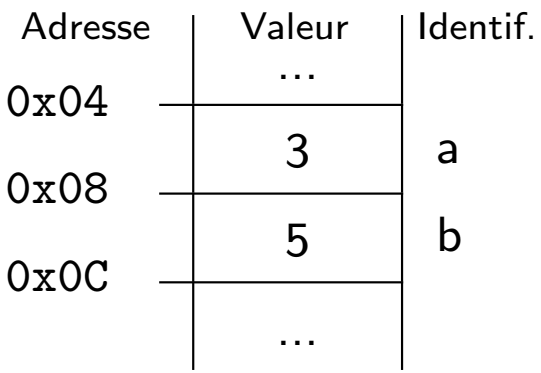
  La valeur récupérée dans le `main` est ensuite affichée par un `printf`.

Appel de fonctions (cf. 6.4)

Lors de l'appel, les paramètres sont évalués comme expressions puis recopiés dans une mémoire propre au contexte de la fonction (*passage par valeur*)

⚠ La visibilité d'une variable est limitée au bloc d'instructions dans lequel elle est déclarée, au sein de la même fonction

```
1  #include<stdio.h>
2  void swap_val(int a, int b) {
3    int c = a;
4    a = b;
5    b = c;
6    printf("swap: a=%d, b=%d\n", a, b);
7  }
8  int main() {
9    int a = 3;
10   int b = 5;
11   swap_val(a,b);
12   printf("main: a=%d, b=%d\n", a, b);
13   return 0;
14 }
```



- déclaration de `a` et `b` (l10-l11)

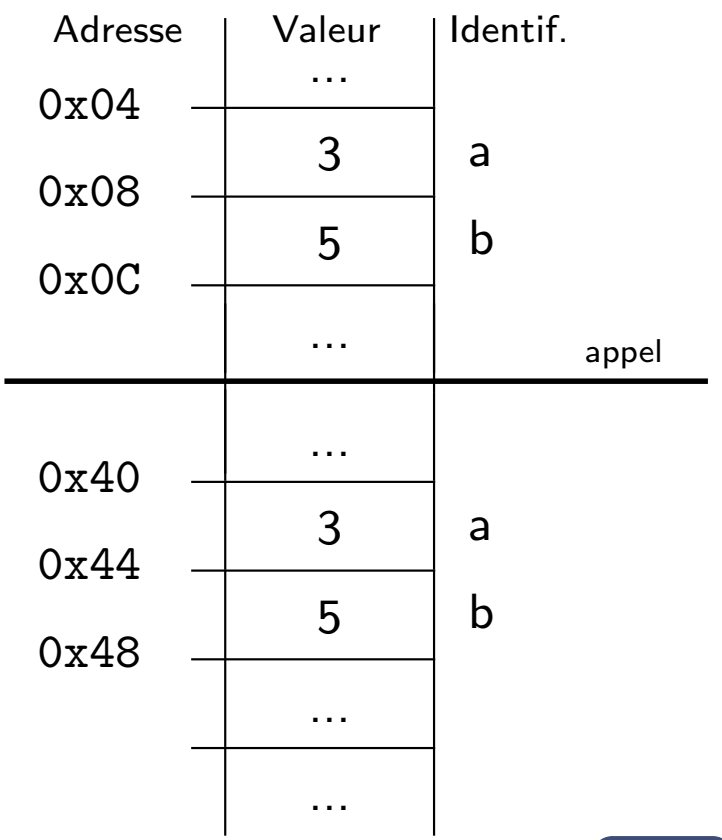
Appel de fonctions (cf. 6.4)

Lors de l'appel, les paramètres sont évalués comme expressions puis recopiés dans une mémoire propre au contexte de la fonction (*passage par valeur*)

⚠ La visibilité d'une variable est limitée au bloc d'instructions dans lequel elle est déclarée, au sein de la même fonction

```
1  #include<stdio.h>
2  void swap_val(int a, int b) {
3    int c = a;
4    a = b;
5    b = c;
6    printf("swap: a=%d, b=%d\n", a, b);
7  }
8  int main() {
9    int a = 3;
10   int b = 5;
11   swap_val(a,b);
12   printf("main: a=%d, b=%d\n", a, b);
13   return 0;
14 }
```

- déclaration de `a` et `b` (l10-l11)
- appel de `swap_val` (l12)



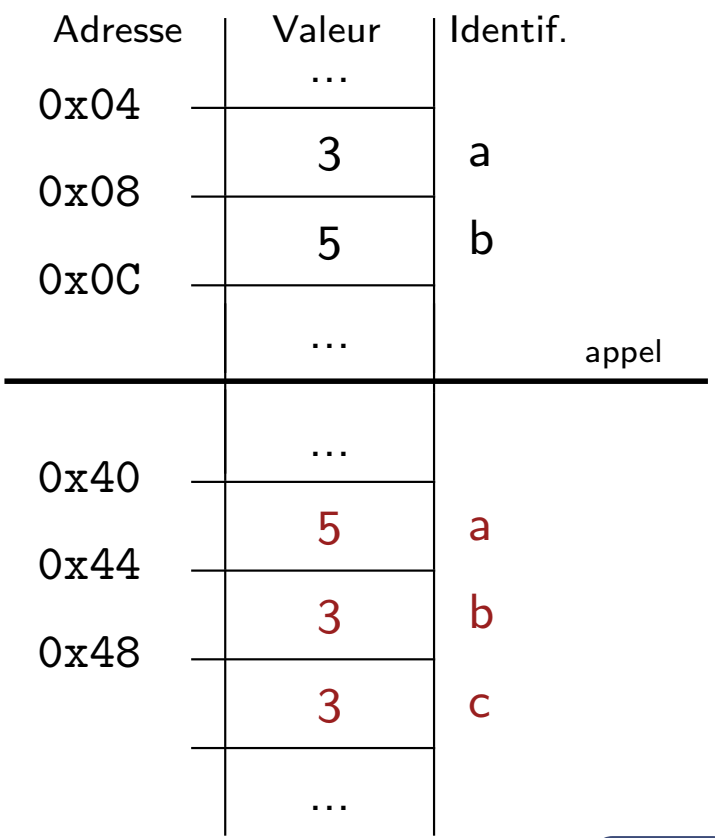
Appel de fonctions (cf. 6.4)

Lors de l'appel, les paramètres sont évalués comme expressions puis recopiés dans une mémoire propre au contexte de la fonction (*passage par valeur*)

⚠ La visibilité d'une variable est limitée au bloc d'instructions dans lequel elle est déclarée, au sein de la même fonction

```
1  #include<stdio.h>
2  void swap_val(int a, int b) {
3    int c = a;
4    a = b;
5    b = c;
6    printf("swap: a=%d, b=%d\n", a, b);
7  }
8  int main() {
9    int a = 3;
10   int b = 5;
11   swap_val(a,b);
12   printf("main: a=%d, b=%d\n", a, b);
13   return 0;
14 }
```

- déclaration de `a` et `b` (l10-l11)
- appel de `swap_val` (l12)
- exécution de `swap_val` (l3-l6)



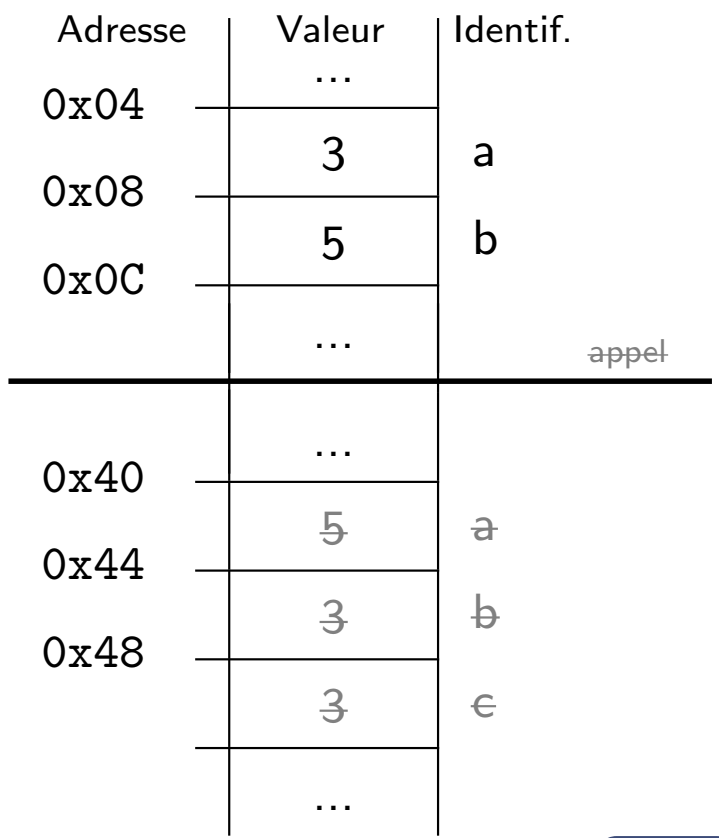
Appel de fonctions (cf. 6.4)

Lors de l'appel, les paramètres sont évalués comme expressions puis recopiés dans une mémoire propre au contexte de la fonction (*passage par valeur*)

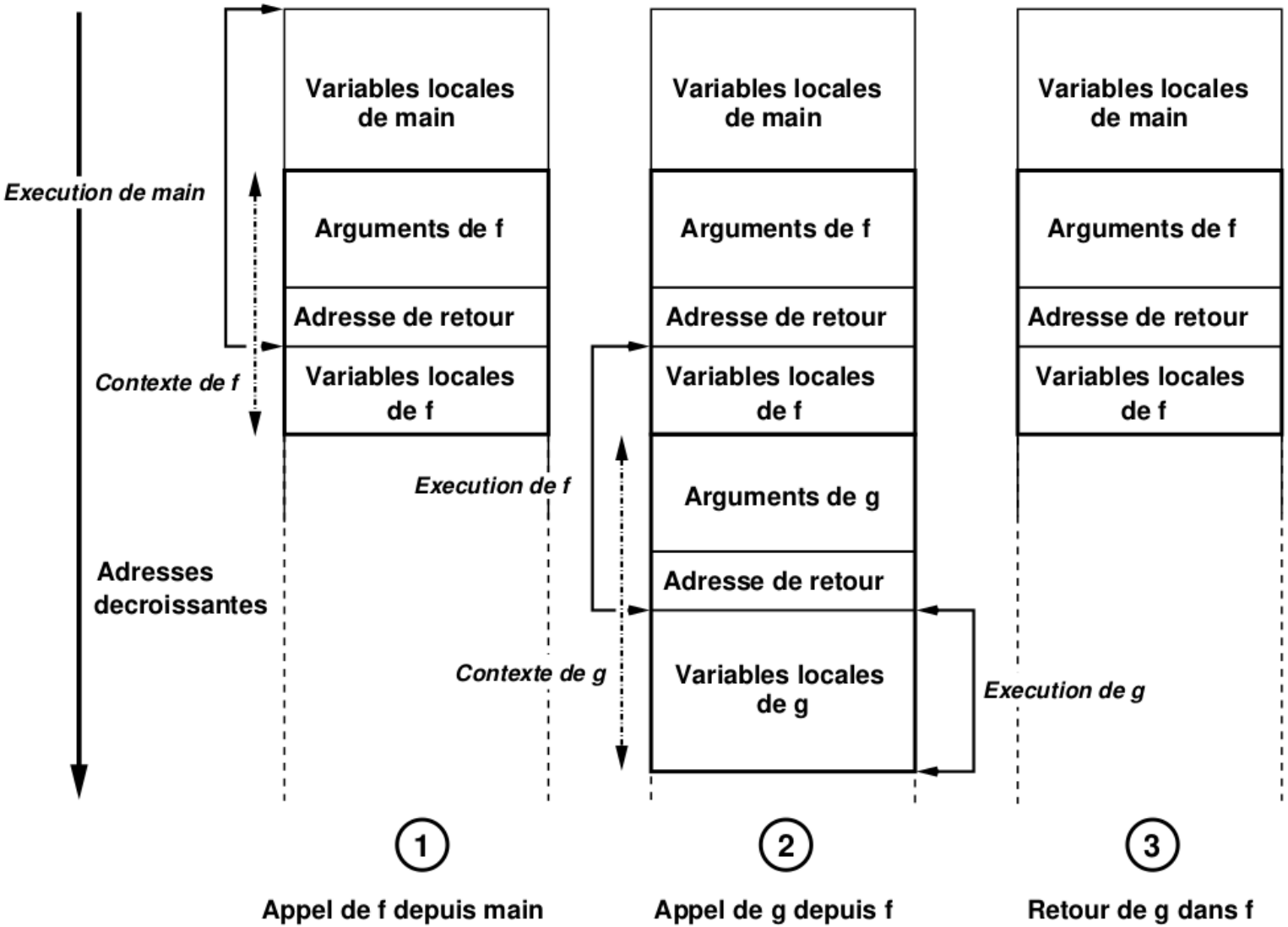
La visibilité d'une variable est limitée au bloc d'instructions dans lequel elle est déclarée, au sein de la même fonction

```
1  #include<stdio.h>
2  void swap_val(int a, int b) {
3    int c = a;
4    a = b;
5    b = c;
6    printf("swap: a=%d, b=%d\n", a, b);
7  }
8  int main() {
9    int a = 3;
10   int b = 5;
11   swap_val(a,b);
12   printf("main: a=%d, b=%d\n", a, b);
13   return 0;
14 }
```

- déclaration de `a` et `b` (l10-l11)
- appel de `swap_val` (l12)
- exécution de `swap_val` (l3-l6)
- sortie de `swap_val` (l7)



Exemple d'exécution d'un programme (cf. 4.4.4)

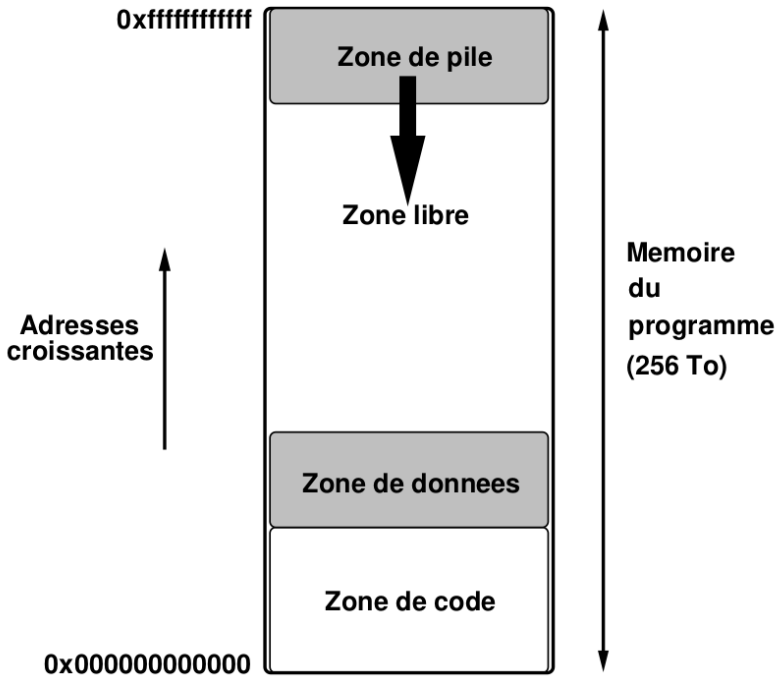


Zones de mémoire d'un programme (cf. 4.4.4)

- La mémoire virtuelle d'un programme est organisée en différentes zones.
- La zone de stockage des variables dépend de leur classe de stockage.
  - Les variables globales sont toujours stockées dans la **zone de données** (data) du programme (qu'elles soient de classe `static` ou non).
  - Les variables locales `static` sont stockées dans la **zone de données** et par défaut dans la **pile** du programme :

Dans la **pile**, une variable est :

- automatiquement allouée au moment de la déclaration
- automatiquement libérée quand le programme sort du bloc d'instructions où cette variable est déclarée.



Fonctions récursives (cf. 6.5.1)

Une fonction peut s'appeler elle-même et permettre une écriture plus condensée

```c
#include <stdio.h>

int loop_overflow (int arg) {
  printf ("Arg val : %i\n", arg);
  return loop_overflow(--arg);
}

int loop_no_overflow (int arg) {
  printf ("Arg val : %i\n ", arg);
  if (arg)
    return loop_no_overflow(--arg);
  return -1;
}

int main () {
  int var = 10;
  loop_no_overflow (var);
  loop_overflow(var);
  return 0;
}
```

Ne pas oublier une condition d'arrêt pour éviter une récursion infinie (qui surchargerait la pile)

Fonctions récursives (cf. 6.5.1)

Une fonction peut s'appeler elle-même et permettre une écriture plus condensée

```c
#include <stdio.h>

int loop_overflow (int arg) {
  printf ("Arg val : %i\n", arg);
  return loop_overflow(--arg);
}

int loop_no_overflow (int arg) {
  printf ("Arg val : %i\n ", arg);
  if (arg)
    return loop_no_overflow(--arg);
  return -1;
}

int main () {
  int var = 10;
  loop_no_overflow (var);
  loop_overflow(var);
  return 0;
}
```

Ne pas oublier une condition d'arrêt pour éviter une récursion infinie (qui surchargerait la pile)

Exercice : Récursif : Écrivez une fonction qui calcule les N premiers termes de la suite de Fibonacci : $F(0) = 0$, $F(1) = 1$, $F(n) = F(n-1) + F(n-2)$

Retour de fonctions (cf. 6.3)

- Si la fonction a un type de retour, elle doit retourner une valeur de ce type (On peut choisir de l'ignorer)

```c
int toto(int a, int b) {
  return a + b;
}

int main() {
  int c = toto(1,2);
  toto(3,4);
  return 0;
}
```

- Si la fonction ne retourne rien, est elle de type `void`
(On parle alors de *procédure*)

```c
#include<stdio.h>

void affichage() {
  printf("Coucou\n");
}

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

En pratique, très peu de "vraies" procédures
Code de retour (ex : -1, 0, 1) utilisés pour faire remonter d'éventuelles erreurs

Question : Pourquoi la fonction main renvoie-t-elle un résultat alors qu'il n'est pas possible de le récupérer dans le reste du programme ?

Retour de fonctions (cf. 6.3)

Le retour (ou résultat) d'une fonction est limité aux types :

- `void` : la fonction ne renvoie pas de résultat, c'est donc une procédure
- n'importe quel type de base de C : entier ou flottant, signé ou non-signé
- un *pointeur* (cf. 8), de n'importe quel type
- les types définis par l'utilisateur (avec `typedef` ou `struct` (cf. 9.1.1))

→ Limité à UNE seule variable typée

Retour de fonctions (cf. 6.3)

Le retour (ou résultat) d'une fonction est limité aux types :

- `void` : la fonction ne renvoie pas de résultat, c'est donc une procédure
- n'importe quel type de base de C : entier ou flottant, signé ou non-signé
- un *pointeur* (cf. 8), de n'importe quel type
- les types définis par l'utilisateur (avec `typedef` ou `struct` (cf. 9.1.1))

→ Limité à UNE seule variable typée

Comment faire pour modifier ou renvoyer plusieurs variables par une fonction ?

→ Utilisation des **pointeurs** (*passage par variable*) : (cf. 8.4)

On donne en paramètres les adresses dans la mémoire des variables de la fonction appelante pour directement modifier leur valeur.

C'est ce qui est fait pour les tableaux qui sont définis par un pointeur sur le premier élément dans la mémoire (cf. 8.3.3)

Déclaration des tableaux

- Tableaux : Réservation de n **cases contiguës** dans la mémoire
  Besoin de définir la taille, *i.e.*, le nombre de cases n dès la déclaration pour réserver l'espace mémoire
- Type : `type nom[]` ou `type* nom`
- Déclaration : `type nom[taille];`

```
✓ int tab[4]; //Tableau de taille 4 (contenu non contrôlé)
∼ int tab[n]; //!!! Interdit en PG108 !!!
```

- Déclaration et initialisation : `type nom[<taille>] = {valeurs};`

```
✓ int tab[4] = {};          //Tableau de taille 4 rempli de 0
✓ int tab[4] = {4,5,5,-2}; //Tableau de taille 4 prérempli
✓ int tab[]  = {4,5,5,-2}; //Tableau de taille 4 prérempli
✓ int tab[6] = {3,4,5};     //Tableau de taille 6 prérempli (reste à 0)
✓ char tab[] = {17,23,-6}; //Tableau de 3 entiers char
∼ int tab[2] = {3,2,1};     //!!! × Seules deux cases réservées × !!!
× int tab[n] = {};          //!!! Erreur de compilation !!!
```

Déclaration des tableaux

- Accès : `tab[i]` pour la i-ème case avec `i`$\in [0, n-1]$

Attention :

- Indices d'un tableau de 0 à $n-1$
- Pas de copie directe possible
- Pas de retour (au sens de la copie) possible
- Aucune fonction sûre pour déterminer la taille d'un tableau (que l'on donnera avec le tableau en paramètre)

Pour aller plus loin : seul cas où la taille est accessible

```c
#include<stdio.h>

long int array_size(int array[]) {
  return (sizeof(array)/sizeof(array[0]));
}

int main() {
  int tab[10];
  printf("tab possède %lu elements\n", sizeof(tab)/sizeof(tab[0]));
  printf("tab possède %lu elements\n", array_size(tab));
  return 0;
}
```

Exemple de Tableaux d'entiers

c2_tab.c

```c
#include<stdio.h>

int main() {
  int tab[5] = {2,3,15,-3,3};
  int len = 5;
  //Display tab elements
  int i = 0;
  while(i<len) {
    printf("Case %d : %d\n", i, tab[i]);
    i = i + 1;
  }
  return 0;
}
```

Retour aux opérateurs d'affectation combinée

Expressions composées complexes :

- Opérateurs d'affectation combinée : `-=` , `+=` , `/=` , `*=` , `etc.`

  ex : `x = x / 2` équivalent à `x /= 2`
  ex : `x = x + 1` équivalent à `x += 1` et `++x` mais pas `x++`

  `++` et `--` pré ou post in/dé-crémentation unitaire avec priorités élevées

Quel sera l'affichage du code suivant ?

```c
#include<stdio.h>

int main() {
  int a = 0;
  ++a;
  printf("Valeur de la variable a : %d\n", a);
  a++;
  printf("Valeur de la variable a : %d\n", a);
  printf("Valeur de la variable a : %d\n", a++);
  printf("Valeur de la variable a : %d\n", ++a);
  return 0;
}
```

Priorités des opérations

L'ordre d'évaluation d'une expression composée de multiples opérateurs, dépend de leurs priorités respectives

Priorité forte ↓ Priorité faible

Opérateur(s)	Associativité	Commentaire
`()`	de gauche à droite	Forçage de la priorité

`()`
`[] -> .`
`++ --`
	de gauche à droite	Appel de fonction
		
		Opérateurs suffixés

`& * + - ~ !`
`sizeof`
`++ --`
`(`*type*`)`
	de droite à gauche	Opérateurs unaires
		
		Opérateurs préfixés
		Conversion de type (*cast*)

`* / %`	de gauche à droite	Opérateurs arithmétiques binaires
`+ -`	de gauche à droite	Opérateurs arithmétiques binaires
`<< >>`	de gauche à droite	Décalage de bits
`< <= > >=`	de gauche à droite	Opérateurs de comparaison
`== !=`	de gauche à droite	Opérateurs de comparaison
`&`	de gauche à droite	ET bit-à-bit
`^`	de gauche à droite	OU exclusif bit-à-bit
`|`	de gauche à droite	OU bit-à-bit
`&&`	de gauche à droite	ET logique
`||`	de gauche à droite	OU logique
`?:`	de gauche à droite	Opérateur ternaire
`= += -= *= /= %=` `&= |= ^= <<= =>>`	**de droite à gauche**	Opérateurs d'affectation
`,`	de gauche à droite	Succession (boucle `for`)

Boucles for (cf. 5.2.2)

- Syntaxe : `for ( [express_1] ; [express_2] ; [express_3] ) { ... }`
- Exécution : Le programme commence par exécuter *epress_1* (initialisation).
  Un cycle démarre alors : le programme évalue *express_2*.
  Si le résultat est "vrai" (non-nul), alors le corps de la boucle `for` puis l'*express_3* sont exécutés.
  Ce cycle continue jusqu'à ce qu'*express_2* soit évaluée à "faux" (nul).

```c
#include<stdio.h>
int main() {
  int i; //décla
  int j = 10; //décla + init
  for(i=0; i<=j; i++, j--) {
    printf("i:%d, j:%d\n", i, j);
  }
  printf("i:%d, j:%d\n", i, j);
  return 0;
}
```

```c
#include<stdio.h>

int main() {

  for(int i=0, j=10; i<=j; i++, j--) {
    printf("i:%d, j:%d\n", i, j);
  }

  return 0;
}
```

Depuis C99, *express_1* peut déclarer et initialiser des variables (ex. à droite).
Pour utiliser cette version, il faut parfois l'option de compilation : `-std=c99`

⚠ Leur portée est limitée au bloc de boucle ! Ne pas faire les deux !

Exercice : Manipulation de Tableaux d'entiers

Modifiez le code de `c2_tab.c` en utilisant des boucles `for` pour créer des fonctions qui doivent :

- 1. Afficher les éléments d'un tableau d'entiers donné en paramètres
- 2. Calculer le maximum d'un tableau d'entiers donné en paramètres
- 3. Tester si un entier appartient à un tableau (on renvoie 1), ou non (0).

c2_tab.c

```c
#include<stdio.h>

int main() {
  int tab[5] = {2,3,15,-3,3};
  int len = 5;
  //Display tab elements
  int i = 0;
  while(i<len) {
    printf("Case %d : %d\n", i, tab[i]);
    i = i + 1;
  }
  return 0;
}
```

Exercice : Manipulation de Tableaux d'entiers

Modifiez le code de `c2_tab.c` en utilisant des boucles `for` pour créer des fonctions qui doivent :

- 1. Afficher les éléments d'un tableau d'entiers donné en paramètres
- 2. Calculer le maximum d'un tableau d'entiers donné en paramètres
- 3. Tester si un entier appartient à un tableau (on renvoie 1), ou non (0).

Quelle est la complexité de cette fonction ?

c2_tab.c

```c
#include<stdio.h>

int main() {
  int tab[5] = {2,3,15,-3,3};
  int len = 5;
  //Display tab elements
  int i = 0;
  while(i<len) {
    printf("Case %d : %d\n", i, tab[i]);
    i = i + 1;
  }
  return 0;
}
```

Exercice : Manipulation de Tableaux d'entiers

Modifiez le code de `c2_tab.c` en utilisant des boucles `for` pour créer des fonctions qui doivent :

- 1. Afficher les éléments d'un tableau d'entiers donné en paramètres
- 2. Calculer le maximum d'un tableau d'entiers donné en paramètres
- 3. Tester si un entier appartient à un tableau (on renvoie 1), ou non (0).

Quelle est la complexité de cette fonction ? $\mathcal{O}(n)$

c2_tab.c

```c
#include<stdio.h>

int main() {
  int tab[5] = {2,3,15,-3,3};
  int len = 5;
  //Display tab elements
  int i = 0;
  while(i<len) {
    printf("Case %d : %d\n", i, tab[i]);
    i = i + 1;
  }
  return 0;
}
```

Opérateur ternaire

Pouvant être utilisé dans une expression comme alternative à `if/else` :

- Syntaxe : `express_1 ? express_2 : express_3`
- Fonctionnement : `express_1` est évaluée complètement
  - Si le résultat est "vrai", alors la valeur d'`express_2` est renvoyée
  - Sinon c'est la valeur de l'exécution d'`express_3` qui est renvoyée.
- Exemples :
  - `printf( x > 0 ? "coucou" : "hello" );`

    Équivalent à :
    ```
    if (x > 0)
      printf("coucou");
    else
      printf("hello");
    ```
  - `printf("Vous avez %d carte%c \n", n, (n==1) ? ' ' : 's');`

Exercice : Opérateur ternaire

- Modifier le programme pour calculer le maximum de deux variables en utilisant l'opérateur ternaire.

Exercice : Triangle de Pascal

Le triangle de Pascal est constitué des coefficients binomiaux :

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
...

Une façon aisée de calculer ce triangle est de constater que chaque élément i est la somme des éléments i et i-1 de la ligne précédente.
Pour simuler un tel affichage, on prendra un tableau d'entiers 1D suffisamment grand qu'on mettra à jour et affichera itérativement.

- Déclarer dans le `main`, un tableau suffisamment grand, initialisé à 0 partout sauf pour sa première valeur qui vaudra 1.
- Écrire la fonction `affiche_pascal` qui reçoit un tableau d'entiers ainsi que sa taille, et qui affiche sur une ligne les coefficients non nuls de ce tableau.
- Écrire la fonction `maj_pascal` qui reçoit un tableau d'entiers ainsi que sa taille. Ce tableau contient les éléments du rang n où n est strictement inférieur à la taille du tableau (les cases non utilisées contiennent des 0). Cette fonction doit mettre à jour ce tableau pour qu'il contienne les éléments du rang n+1.

Manipulation de Tableaux d'entiers - Tri par insertion

Tri par insertion d'un tableau de valeurs *tab* de taille *taille* :

```
1:  function InsertSort(tab, taille)
2:      i ← 1
3:      while i < taille do
4:          e ← tab[i]
5:          j ← i
6:          while j > 0 and tab[j − 1] > e do
7:              tab[j] ← tab[j − 1]
8:              j ← j − 1
9:          tab[j] ← e
10:         i ← i + 1
```

Manipulation de Tableaux d'entiers - Tri par insertion

Tri par insertion d'un tableau de valeurs *tab* de taille *taille* :

```
1:  function InsertSort(tab, taille)
2:      i ← 1
3:      while i < taille do
4:          e ← tab[i]
5:          j ← i
6:          while j > 0 and tab[j − 1] > e do
7:              tab[j] ← tab[j − 1]
8:              j ← j − 1
9:          tab[j] ← e
10:         i ← i + 1
```

Quelle est la complexité de cette fonction ?

Manipulation de Tableaux d'entiers - Tri par insertion

Tri par insertion d'un tableau de valeurs *tab* de taille *taille* :

```
1:  function InsertSort(tab, taille)
2:      i ← 1
3:      while i < taille do
4:          e ← tab[i]
5:          j ← i
6:          while j > 0 and tab[j − 1] > e do
7:              tab[j] ← tab[j − 1]
8:              j ← j − 1
9:          tab[j] ← e
10:         i ← i + 1
```

Quelle est la complexité de cette fonction ? $\mathcal{O}((n^2))$

Manipulation de Tableaux d'entiers - Recherche dichotomique

Definition

La recherche dichotomique, ou recherche par dichotomie (en anglais : *binary search*), est un algorithme de recherche pour trouver la position d'un élément dans un tableau **trié**.

Le principe

L'objectif est trouver la position d'un élément dans un tableau trié.

- On trouve l'élément m avec la position la plus centrale du tableau (si le tableau est vide on s'arrête) ;
- On compare la valeur de l'élément recherché avec l'élément m ;
- Si elle est plus petite, on recommence dans le sous-tableau de gauche, sinon dans le sous-tableau de droite.

L'algorithme de la dichotomie itérative

```
1:  function DICHOTOMIE(tab, taille, e)
2:      min ← 0
3:      max ← taille − 1
4:      while min < max do
5:          mid ← (min + max)/2
6:          if tab[mid] < e then
7:              min ← mid + 1
8:          else
9:              max ← mid
10:     if tab[min] == e then
11:         return min
12:     else
13:         return -1 //nonTrouvé
```

L'algorithme de la dichotomie itérative

```
1:  function DICHOTOMIE(tab, taille, e)
2:      min ← 0
3:      max ← taille − 1
4:      while min < max do
5:          mid ← (min + max)/2
6:          if tab[mid] < e then
7:              min ← mid + 1
8:          else
9:              max ← mid
10:     if tab[min] == e then
11:         return min
12:     else
13:         return -1 //nonTrouvé
```

Quelle est la complexité de cette fonction ?

L'algorithme de la dichotomie itérative

```
1:  function DICHOTOMIE(tab, taille, e)
2:      min ← 0
3:      max ← taille − 1
4:      while min < max do
5:          mid ← (min + max)/2
6:          if tab[mid] < e then
7:              min ← mid + 1
8:          else
9:              max ← mid
10:     if tab[min] == e then
11:         return min
12:     else
13:         return -1 //nonTrouvé
```

Quelle est la complexité de cette fonction ? $\mathcal{O}(log_2(n))$

Variable aléatoire

La fonction `rand` (`#include<stdlib.h>`) retourne un entier pseudo-aléatoire :

```
int a = rand(); //entre 0 et la constante RAND_MAX
int b = rand()%10; //entier entre 0 et 9
```

L'entier renvoyé varie-t-il à chaque exécution ?

La fonction `srand` (`#include<stdlib.h>`) permet de fixer la graine de la séquence pseudo-aléatoire générée par `rand()`. Exemple d'utilisation pour une séquence aléatoire avec `time` (`#include<time.h>`) :

```
int graine = time(NULL); //Retourne le temps en seconde depuis
                         00:00:00 UTC, January 1, 1970
srand(graine); //la graine est fixée selon l'heure actuelle
int a = rand(); //entier entre 0 et RAND_MAX
```

L'entier renvoyé varie-t-il à chaque exécution ?

Exercice : Tri

- Générer dans le *main*, un tableau d'entiers aléatoires de taille N=10.
- Implémenter l'algorithme de tri par insertion.
- Implémenter l'algorithme de recherche dichotomique.

Mesure du temps d'exécution

La fonction `clock()` (`#include<time.h>`) renvoie le temps processeur consommé depuis le dernier appel :

```
clock_t start = clock();
/* Instructions */
clock_t stop = clock();
double total_t = ((double)stop - start) / CLOCKS_PER_SEC;
printf(``Total time taken by CPU: %f\n'', total_t);
```

Exercice : Tri

- Mesurer le temps de calcul moyen d'un tri de tableau de taille N=100000 par insertion suivi de R=10000 recherches dichotomique.
- Comparer avec le temps de calcul pour R=10000 recherches purement itérative (parcours linéaire), sans tri de tableau.
- Faites varier N et R. Conclure.

Chaînes de caractères

Le type *chaînes de caractères* n'existe pas en C, seul le type `char` existe.

Et il existe une **correspondance automatique** entre les entiers $\in [\![0, 127]\!]$ et une table de caractères (la table ASCII).

$\rightarrow$ On stockera donc une suite de caractères dans un tableau (de `char`)

- Définition :
  Une chaîne de caractères est un tableau de `char` avec pour dernier élément le caractère de fin de chaîne `'\0'` ($\equiv$ `0`)

Chaînes de caractères

Le type *chaînes de caractères* n'existe pas en C, seul le type `char` existe.

Et il existe une **correspondance automatique** entre les entiers $\in [\![0, 127]\!]$ et une table de caractères (la table ASCII).

$\rightarrow$ On stockera donc une suite de caractères dans un tableau (de `char`)

- Définition :

  Une chaîne de caractères est un tableau de `char` avec pour dernier élément le caractère de fin de chaîne `'\0'` ($\equiv 0$)

- Déclaration :

```
char str1[] = {'t','o','t','o'};
char str2[] = {'t','o',116,111,'\0'};
char str3[] = {116,111,'t','o',0};
char str4[] = "toto";
```

Chaînes de caractères

Le type *chaînes de caractères* n'existe pas en C, seul le type `char` existe.

Et il existe une **correspondance automatique** entre les entiers $\in [\![0, 127]\!]$ et une table de caractères (la table ASCII).

$\rightarrow$ On stockera donc une suite de caractères dans un tableau (de `char`)

- Définition :

  Une chaîne de caractères est un tableau de `char` avec pour dernier élément le caractère de fin de chaîne `'\0'` ($\equiv 0$)

- Déclaration :

```
~ char str1[] = {'t','o','t','o'};       //Tableau de 4 caractères
✓ char str2[] = {'t','o',116,111,'\0'}; //Chaîne de caractères (5 octets)
✓ char str3[] = {116,111,'t','o',0};    //Même chaîne que str2 (5 octets)
✓ char str4[] = "toto";                 //Même chaîne que str2 (5 octets)
```

Chaînes de caractères

Le type *chaînes de caractères* n'existe pas en C, seul le type `char` existe.

Et il existe une **correspondance automatique** entre les entiers $\in [\![0, 127]\!]$ et une table de caractères (la table ASCII).

$\rightarrow$ On stockera donc une suite de caractères dans un tableau (de `char`)

- Définition :
  Une chaîne de caractères est un tableau de `char` avec pour dernier élément le caractère de fin de chaîne `'\0'` ($\equiv 0$)
- Déclaration :

```
~ char str1[] = {'t','o','t','o'};       //Tableau de 4 caractères
✓ char str2[] = {'t','o',116,111,'\0'}; //Chaîne de caractères (5 octets)
✓ char str3[] = {116,111,'t','o',0};    //Même chaîne que str2 (5 octets)
✓ char str4[] = "toto";                 //Même chaîne que str2 (5 octets)

~ char* str5 = "toto";                  //Chaîne de caractères constante!
× str5[2] = 's';                        //× Erreur de segmentation ×
```

Chaînes de caractères

Le type *chaînes de caractères* n'existe pas en C, seul le type `char` existe.

Et il existe une **correspondance automatique** entre les entiers $\in [\![0, 127]\!]$ et une table de caractères (la table ASCII).

$\rightarrow$ On stockera donc une suite de caractères dans un tableau (de `char`)

- Définition :
  Une chaîne de caractères est un tableau de `char` avec pour dernier élément le caractère de fin de chaîne `'\0'` ($\equiv$ `0`)
- Déclaration :

```
~ char str1[] = {'t','o','t','o'};        //Tableau de 4 caractères
✓ char str2[] = {'t','o',116,111,'\0'}; //Chaîne de caractères (5 octets)
✓ char str3[] = {116,111,'t','o',0};     //Même chaîne que str2 (5 octets)
✓ char str4[] = "toto";                  //Même chaîne que str2 (5 octets)

~ char* str5 = "toto";                   //Chaîne de caractères constante!
× str5[2] = 's';                         //× Erreur de segmentation ×
```

- Affichage avec `printf` :
  `char` : `%c` : caractère (ex : `'C'`) `%hhi` : valeur numérique (ex : `67`)
  `char*` : `%s` : chaîne de caractères (ex : `"hello"`)
  `printf` lit alors `char` par `char` avant de tomber sur `'\0'`

Chaînes de caractères

- Longueur d'une chaîne : Puisque les chaînes ont toutes pour dernier élément `'\0'`, on peut utiliser `size_t strlen(const char* s)` pour calculer leur taille (contrairement aux tableaux d'entiers classiques).
- D'autres fonctions comme `strcpy` ou `strcmp` (`#include<string.h>`).

Pour aller plus loin

- Le type `size_t` : Type entier, non signé, suffisamment grand pour contenir la valeur en octets, de la taille du plus grand tableau possible. Généralement sur 8 octets ($\approx$ `unsigned long int`).
- Le mot-clef `const` : signale au compilateur que l'élément ne doit pas être modifié pendant l'exécution du programme.

```
const char c = 'A'   //La variable c ne peut être modifiée
```

Les syntaxes suivantes sont (presque) équivalentes :

```
const char str[] = {'l','o','l','\0'}; //Aucun modif. possible
const char* str = "lol";               //Aucun modif. possible
char* str = "lol";                     //Aucun modif. possible
```

Dans le dernier cas, ni erreur ni warning ne sont fournis par le compilateur si on tente de modifier la chaîne

Chaînes de caractères

-  `char` **ne veut pas dire caractères !**

  Un `char` est un petit entier sur 8 bits.

```c
#include<stdio.h>
int main(){
  char c = 12, d = 24;  //char tous les deux
  printf("%hhi\n", c+d); //36
  return 0;
}
```

- Lien avec la table ASCII :

  Si on est dans un contexte où on manipule des caractères textuels, ces derniers n'interviennent qu'en constantes ou par le `printf`.

  Correspondance automatique entre entiers $\in [\![0, 127]\!]$ et caractères.

  Un caractère peut donc être traité numériquement comme une constante $\in [\![0, 127]\!]$. Par exemple le caractère 'C' correspond à la valeur 67.

```c
#include<stdio.h>
int main(){
  char c = 67, d = 'C';
  printf("%hhi, %c, %d\n", c, c, c==d); //67 C 1
  return 0;
}
```

Exercice : Table ASCII

- 1) Observer les correspondances au sein de la table ASCII (`man ascii`)
- 2) Écrire une fonction qui affiche ligne par ligne les correspondances entre les caractères affichables de la table ASCII et leur valeur entière décimale.
- 3) Écrire la fonction `my_isalpha` qui teste si un caractère est une lettre de l'alphabet (cette fonction existe sous le nom `isalpha` dans `ctype.h`).
- 4) Écrire la fonction `my_tolower` qui transforme une lettre de l'alphabet en minuscule (cette fonction existe sous le nom `tolower` dans `ctype.h`)

Interactions avec le programme

Il existe trois façons d'interagir entre un programme et son utilisateur/ utilisatrice :

- par l'utilisation d'un mode interactif avec les fonctions `scanf` et `printf`
- par l'utilisation de variables d'environnement via les fonctions `setenv` et `getenv`
- par l'utilisation de la *ligne de commande* du programme :

```
gcc -Wall -o prog prog.c
```

```
./prog arg1 arg2 arg3
```

Gestion des arguments de la Ligne de commande

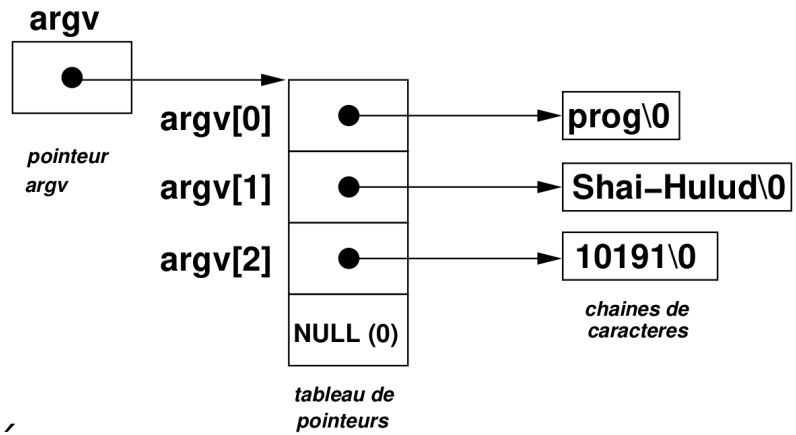
Prototype complet : `int main(int argc, char *argv[])`

Un programme possède `argc` ≥ 1 arguments
Stockés comme chaînes de caractères : `argv[0], ... , argv[argc-1]`

Exemple :

Lancement du programme `prog`
avec comme arguments :
"Shai-Hulud" et "10191"

```
./prog Shai-Hulud 10191
```



- `argv[0]` : nom du programme appelé
- `argv[argc]` : pointeur qui a pour valeur `NULL` (cf. 8) qui a pour valeur 0

Exercice : Ligne de commande n°1

- Écrire un programme qui affiche tous les paramètres donnés en arguments depuis la ligne de commande

Conversion des arguments numériques

Impossible d'utiliser directement des nombres depuis la ligne de commande.
On récupère une chaîne de caractères, par ex. `"123"`≡`{'1','2','3','\0'}`, qui n'est pas un entier de valeur numérique `123`.

Conversion, à l'aide de l'une des fonctions suivantes (`#include<stdlib.h>`) :

- `atoi` (alpha to integer) : convertit une chaîne en entier `int`
- `atol` (alpha to long integer) : convertit une chaîne en entier `long int`
- `atof` (alpha to float) : convertit une chaîne en nombre flottant (`double`)

Exercice : Ligne de commande n°2

- Écrire un programme qui affiche la somme des entiers qui lui sont passés en ligne de commande.
- Écrire une fonction qui teste si une année est bissextile. La fonction renvoie 1 si c'est le cas, 0 sinon, et l'année sera donnée en ligne de commande.
  (Une année est bissextile si elle est divisible par 4 mais pas par 100, ou si elle est divisible par 400)

Opérateur modulo % :

```
int c = 11 % 2; //1
```

Exercice : Calculatrice

Écrire un programme de calculatrice simple.

Ce programme reçoit trois arguments : `a`, `op` et `b` :

- `a` et `b` sont des réels quelconques.
- `op` est un caractère (`+`, `-`, `x`, ou `/`) qui permet de choisir l'opération.
  (La multiplication est représentée par la lettre x car l'étoile est le joker qui signifie "tous les fichiers dans le dossier courant")

Ce n'est pas la peine de faire de test pour vérifier si l'utilisateur rentre une valeur qui ne correspond pas à un réel pour `a` ou `b`. Par contre, dans tous les autres cas, les erreurs doivent êtres gérées en affichant "Erreur", puis éventuellement la raison de l'erreur.

Attention : Pour que le programme reçoive bien trois arguments différents, il faudra placer un espace dans la ligne de commande entre les valeurs numériques et l'opération comme suit :

```
./calculatrice 42 + 201
```

Exercice : Ligne de commande n°3

- 1) Écrire une fonction `my_strlen` qui calcule la longueur d'une chaîne de caractères, que l'on continuera à récupérer en ligne de commande (Rappel : une chaîne de caractères se termine nécessairement par `'\0'`)
- 2) Écrire une fonction `chaine_miroir` qui inverse une chaîne de caractères passée en argument (ex : *aabc* devient *cbaa*). Proposer une solution pour que votre fonction n'écrase pas la chaîne d'entrée.
- 3) Écrire une fonction `my_atoi` qui convertit une chaîne de caractères ne contenant que des chiffres en sa valeur numérique. Par exemple, la chaîne "1234" devra être convertie en un entier valant 1234.
- Bonus : Écrire une fonction testant si une chaîne de caractères est un palindrome (se lisant dans les deux sens comme "radar" ou "ressasser"). L'améliorer pour qu'elle fonctionne avec des phrases, et des mots ne contenant pas que des lettres. (On pourra réutiliser les fonctions `my_isalpha` et `my_tolower` écrites précédemment).
  Ex : "L'ami naturel ? Le rut animal."

Opérateurs bit-à-bit (bitwise operators)

- `&` (ET) , `|` (OU) , `^` (OU exclusif) bit-à-bit
  Attention à ne pas confondre avec les opérateurs logiques (&& ||)

Exemple : `unsigned char`									Non signé	Signé
`a`	0	0	0	0	1	1	0	1	13	13
`b`	0	0	0	0	0	1	1	0	6	6
`a&b`	0	0	0	0	0	1	0	0	4	4
`a|b`	0	0	0	0	1	1	1	1	15	15
`a^b`	0	0	0	0	1	0	1	1	11	11

Opérateurs bit-à-bit (bitwise operators)

- `&` (ET) , `|` (OU) , `^` (OU exclusif) bit-à-bit
  Attention à ne pas confondre avec les opérateurs logiques (&& ||)
- `~` complément à un (inversion de bits)

Exemple : `unsigned char`									Non signé	Signé
`a`	0	0	0	0	1	1	0	1	13	13
`b`	0	0	0	0	0	1	1	0	6	6
`a&b`	0	0	0	0	0	1	0	0	4	4
`a|b`	0	0	0	0	1	1	1	1	15	15
`a^b`	0	0	0	0	1	0	1	1	11	11
`~a`	1	1	1	1	0	0	1	0	242	-14

Opérateurs bit-à-bit (bitwise operators)

- `&` (ET) , `|` (OU) , `^` (OU exclusif) bit-à-bit
  Attention à ne pas confondre avec les opérateurs logiques (&& ||)
- `~` complément à un (inversion de bits)
- `var<<n` décalage de `var` de `n` bits vers la gauche, `var>>n` vers la droite

Exemple : `unsigned char`									Non signé	Signé
`a`	0	0	0	0	1	1	0	1	13	13
`b`	0	0	0	0	0	1	1	0	6	6
`a&b`	0	0	0	0	0	1	0	0	4	4
`a|b`	0	0	0	0	1	1	1	1	15	15
`a^b`	0	0	0	0	1	0	1	1	11	11
`~a`	1	1	1	1	0	0	1	0	242	-14
`a>>1`	0	0	0	0	0	1	1	0	6	6
`a<<3`	0	1	1	0	1	0	0	0	104	104
`3>>a`	0	0	0	0	0	0	0	0	0	0
`c`	1	1	1	1	1	1	1	1	255	-1
`c>>1`	0	1	1	1	1	1	1	1	127	127
`(char) c>>1`	1	1	1	1	1	1	1	1	255	-1

Quels seront les affichages du code suivant ?

```c
#include<stdio.h>

int main() {
  char a = 1;
  char b = 2;
  printf("Valeur ET bitwise : %hhi\n", a & b);
  printf("Valeur OU bitwise : %hhi\n", a | b);
  printf("Valeur ET logique : %hhi\n", a && b);
  printf("Valeur OU logique : %hhi\n", a || b);
  printf("Valeur OU exclusif bitwise : %hhi\n", a ^ b);
  b = 4;
  printf("Valeur bitwise : %hhi\n", b<<2);
  printf("Valeur bitwise : %hhi\n", b>>2);
  printf("Valeur bitwise : %hhi\n", 2>>b);
  a = 255;
  printf("Valeur complément à un bitwise : %hhi\n", ~a);
  return 0;
}
```

Exercice : Opérateurs bits à bits

- Écrire une fonction qui compte le nombre de bits positionnés à un dans un entier (on pourra éventuellement utiliser `sizeof` pour déterminer le nombre de bits de l'entier)

Tableaux, chaînes de caractères

Résumé :

- Fonction `main` = seul point d'entrée du programme
  Obligatoire pour générer un exécutable
- Fonction : `type fonction(type1 arg1, ...) {...};`
- Passage des arguments par valeur : `type val = fonction(arg1);`
  - Les arguments sont évalués et écrits dans la mémoire de la fonction
  - Toute ces données sont perdues quand on sort de la fonction
  - On ne peut renvoyer avec `return` qu'une seule variable de type :
    `int`, `float`, `char`, ..., pointeur, struct, ...
- Ligne de commande : `int main(int argc, char* argv[])`
  `argc` = nombre de paramètres ($\geq$1)
  `argv` = tableau de chaînes de caractères des paramètres
  `argv[0]` = chaîne de caractères : nom du programme
  `argv[argc-1]` = chaîne de caractères : dernier paramètre
  `atoi(argv[1])` = conversion en entier du premier paramètre

```
./program arg1 arg2
```

Tableaux, chaînes de caractères

Résumé :

- **Tableau** = cases contiguës dans la mémoire
  - Déclaration statique : `type tab[size];` size cases [0, size-1]
  - Déclaration et initialisation : `type tab[] = {1, 10, -2};`
    Le nombre de cases suffisant (ici 3) est réservé
  - Pas de fonction pour connaître la taille d'un tableau !
    On donnera toujours la taille du tableau en paramètres
  - Parcours :
```
int i;
for(i=0; i<size; i++) //Attention aux ;
  printf("tab[%d] = %d\n", i, tab[i]);
```
  - Le standard C99 (option de compilation `-std=c99`) permet :
```
for(int i=0; i<size; i++) //Portée de i = boucle for
```

Tableaux, chaînes de caractères

Résumé :

- **Chaîne de caractères** : tableau de `char` avec `'\0'` $\equiv$ `0` à la fin
  - Cas général, `char` = petit entier sur un octet ! (`%hhi`)
  - Table ASCII : correspondance entre entiers $\in$ [0, 127] et caractères
  - `char` = caractère ASCII à l'affichage (`%c`, `%s`) ou l'affectation

    Ex. : `char str[] = "hello"; //'\0' ajouté automatiquement`
    Ex. : `char str[] = {'h', 101, 'l', 108, 'o', '\0'};`

    Dans la mémoire `str` :

104	101	108	108	111	0

  - `printf("val.:%hhi=%hhi=%hhi, car.:%c=%c=%c, str:%s\n", 'h', 104, str[0], 'h', 104, str[0], str);`

    Affiche : `val.:104=104=104, car.:h=h=h, str:hello`

  - Fonctions de manipulation de chaînes dans `#include<string.h>`

    `strlen` pour calculer la taille (`strlen(str)` == 5)

    Attention au `'\0'` qui n'est pas compté, mais réservé

Architecture système (cf. 8.1.1)

- Les programmes sont divisés en unités élémentaires, les instructions, qui sont exécutées par le **processeur**.
- Ces instructions et données sont stockées dans la mémoire (physique) de la machine (la **RAM**) et sont transférées dans les registres du processeur à l'aide d'un élément matériel appelé **bus** (ou bus mémoire).
- Ces éléments travaillent sur des objets ayant une taille minimale que l'on appelle des **mots-mémoire** (*memory word*).
  Il s'agit de la plus petite quantité quantité d'information pouvant être traitée par la machine. (Généralement 8 octets - système 64 bits)
- La mémoire physique est organisée en éléments contigus que l'on appelle **cases** ou cellules (*memory cells*) qui possèdent une **adresse** unique.
  Il s'agit de la plus petite unité de mémoire identifiable par le processeur. (Généralement 8 bits)

Format hexadécimal 1-9a-f (base 16) utilisée souvent pour les adresses :

ex. :
```
int a = 0x2c; /*= 0010 1100 (binaire) = 44 (décimal)*/
printf(``a = %x (hexa)\n'',a);
```

Pointeurs

Définition :

Les pointeurs sont des **variables qui stockent des adresses** dans la mémoire
Un pointeur vers n'importe quel type est sur 8 octets (`%lu`, `%p` (hexa))
(sur une architecture standard 64 bits)
Ils nous permettront notamment de modifier des variables au sein d'une autre fonction (*passage par variable*)

Syntaxe :

- Déclaration de pointeur : `type* nom_ptr`
- Récupération de pointeur : `&nom_var`
- Déréférencement (accès à l'adresse contenue dans le pointeur) : `*nom_ptr`

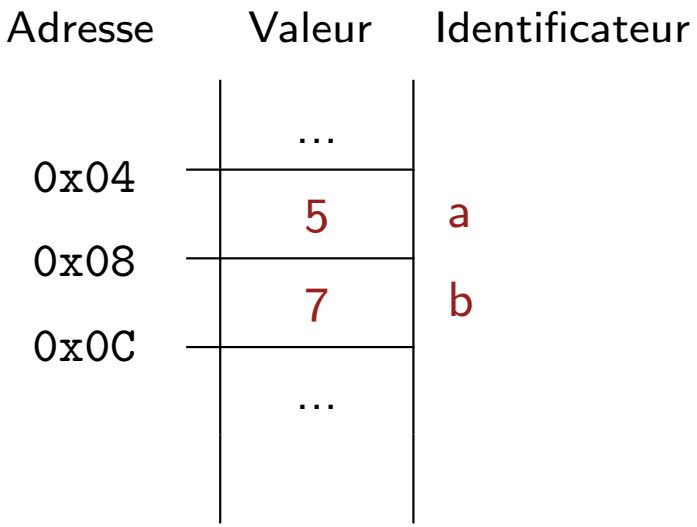
Exemple :

```
int* ptr = NULL;       // ptr est un pointeur (*) vers une variable
                       // de type int, et est initialisé à NULL (ou 0)
int a;                 // a est un entier de type int
ptr = &a;              // ptr contient l'adresse de la variable a
*ptr = 5;              // On met 5 à l'adresse contenue dans ptr
printf(``a=%d\n'',a); // a vaut à présent 5
```

Dans la mémoire

Exemple d'illustration d'écriture dans la mémoire physique :

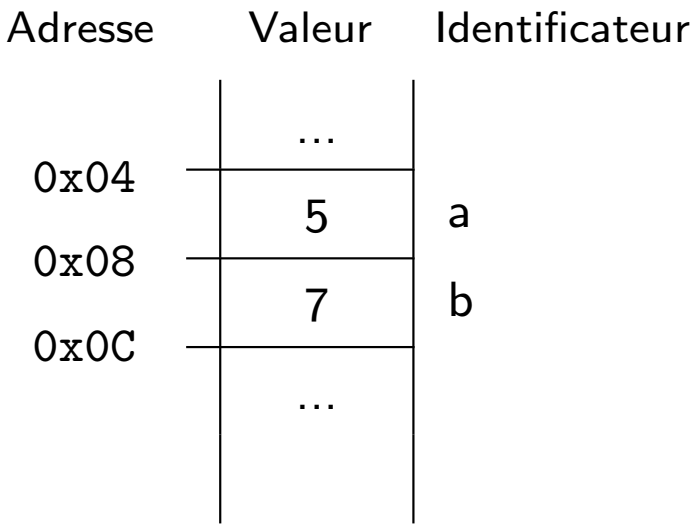
```
int a = 5;
int b = 7;
```



Dans la mémoire

Exemple d'illustration d'écriture dans la mémoire physique :

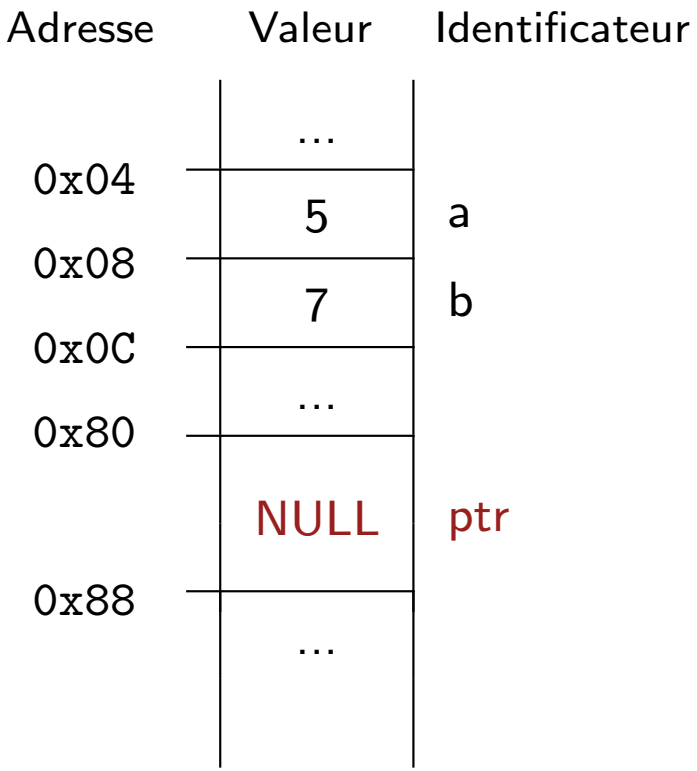
```
int a = 5;
int b = 7;
...
```



Dans la mémoire

Exemple d'illustration d'écriture dans la mémoire physique :

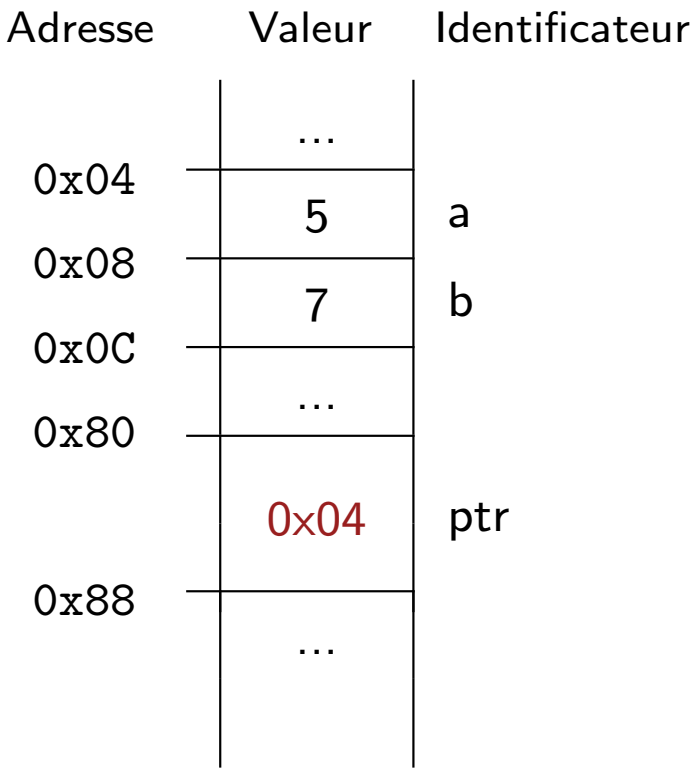
```
int a = 5;
int b = 7;
...
int* ptr = NULL;
```



Dans la mémoire

Exemple d'illustration d'écriture dans la mémoire physique :

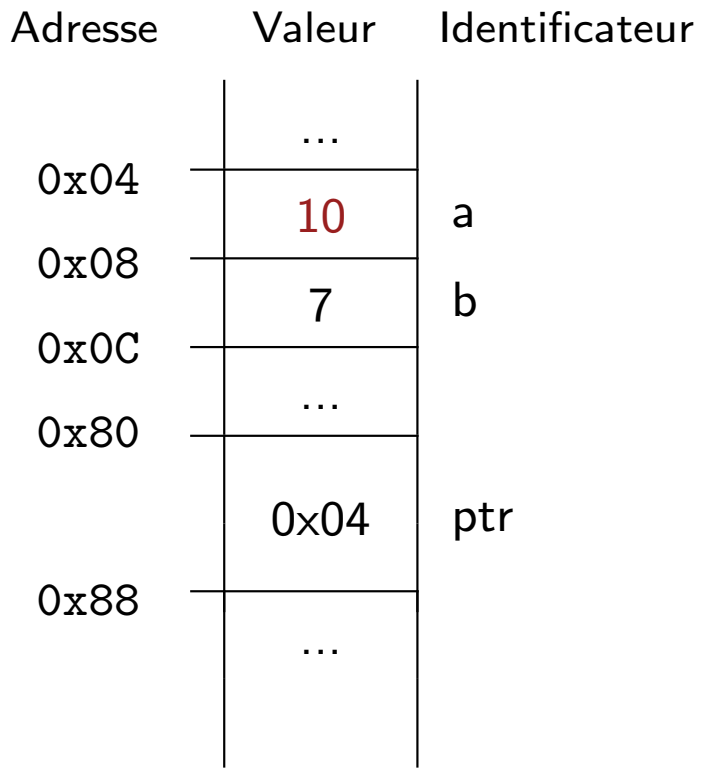
```
int a = 5;
int b = 7;
...
int* ptr = NULL;
ptr = &a;
```



Dans la mémoire

Exemple d'illustration d'écriture dans la mémoire physique :

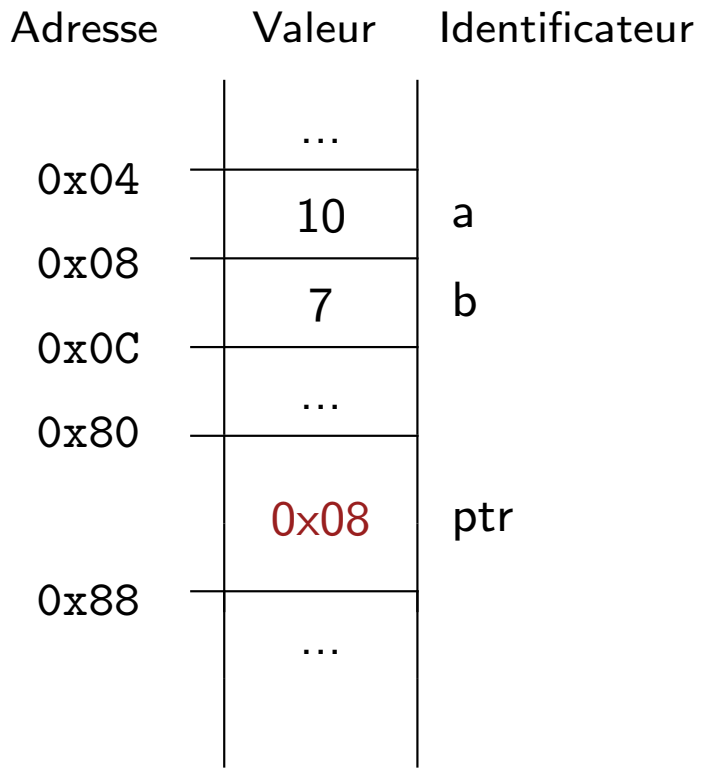
```
int a = 5;
int b = 7;
...
int* ptr = NULL;
ptr = &a;
*ptr = 10;
```



Dans la mémoire

Exemple d'illustration d'écriture dans la mémoire physique :

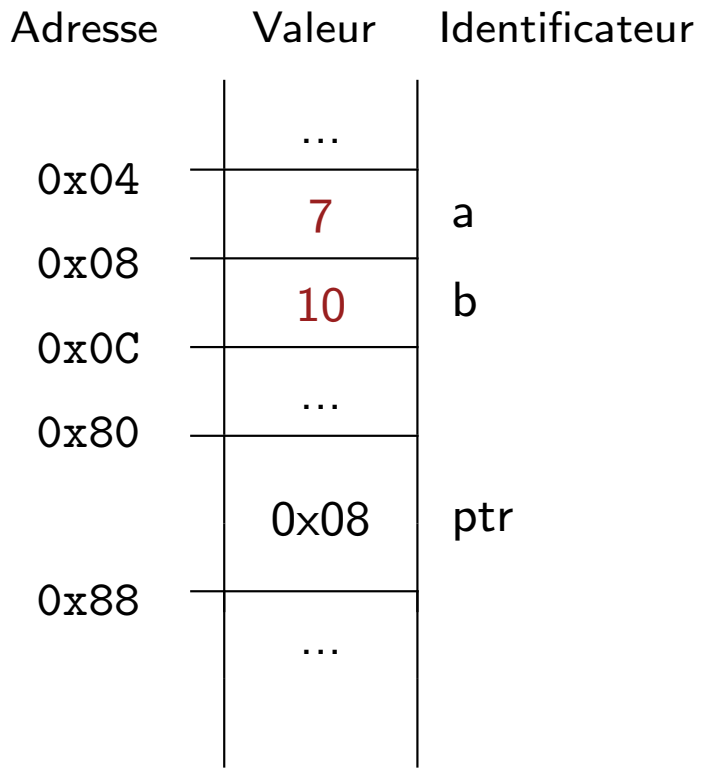
```
int a = 5;
int b = 7;
...
int* ptr = NULL;
ptr = &a;
*ptr = 10;
ptr = &b;
```



Dans la mémoire

Exemple d'illustration d'écriture dans la mémoire physique :

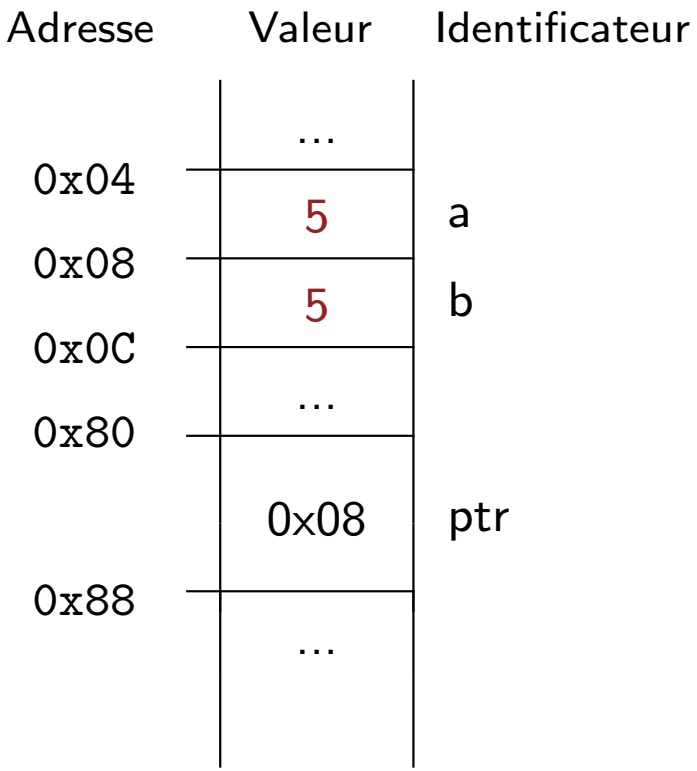
```
int a = 5;
int b = 7;
...
int* ptr = NULL;
ptr = &a;
*ptr = 10;
ptr = &b;
swap_var(ptr,&a);
```



Dans la mémoire

Exemple d'illustration d'écriture dans la mémoire physique :

```
int a = 5;
int b = 7;
...
int* ptr = NULL;
ptr = &a;
*ptr = 10;
ptr = &b;
swap_var(ptr,&a);
b = 5;
a = *ptr;
```



Pointeurs

```c
#include<stdio.h>
void swap_val(int a, int b) {
  int c = a;
  a = b;
  b = c;
  printf("swap: a=%d, b=%d\n", a, b);
}
int main() {
  int a = 3;
  int b = 5;
  swap_val(a,b);
  printf("main: a=%d, b=%d\n", a, b);
  return 0;
}
```

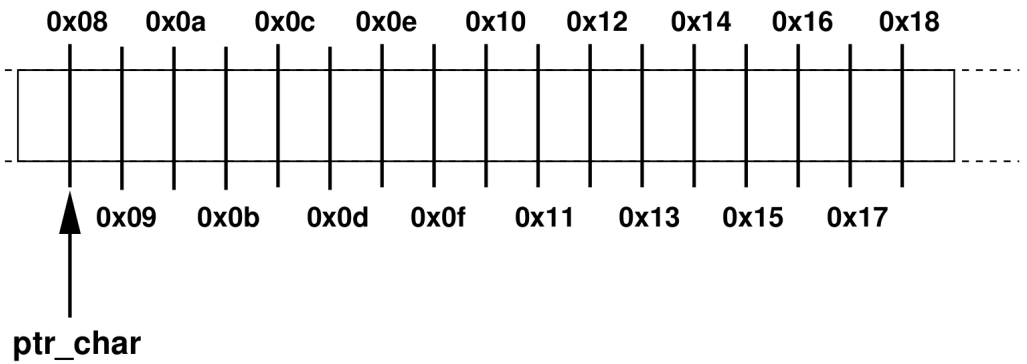
Exercice : Swap

- Modifier la fonction `swap_val()` en `swap_var()` où les modifications seront appliquées aux variables `a` et `b` du main (cf. 8.4.2)
- En utilisant `swap_var`, écrire la fonction `order` qui s'assure que ses deux paramètres entiers sont, à la fin de l'exécution, dans l'ordre croissant.

Typage des pointeurs (cf. 8.1.3)

Le typage du pointeur va influencer le comportement des opérations qui vont porter sur lui et modifie la "vision" qu'il possède de l'organisation de la mémoire, lui permettant d'accéder, ou non, à certaines adresses.

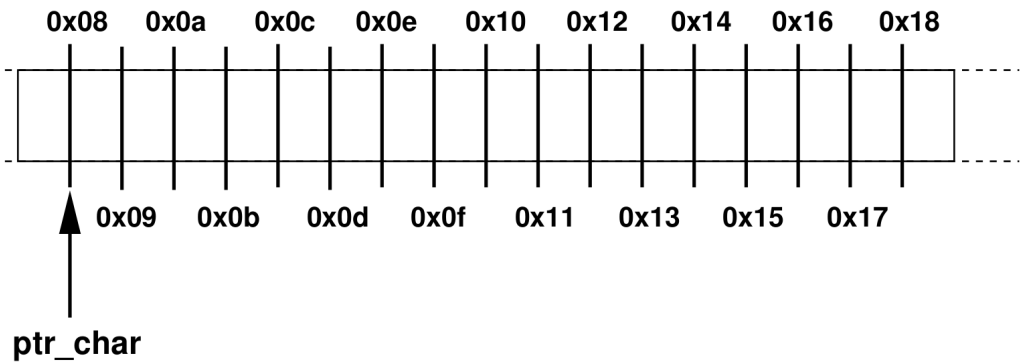
- Exemple `char` : `char* ptr_char = 0x08;`



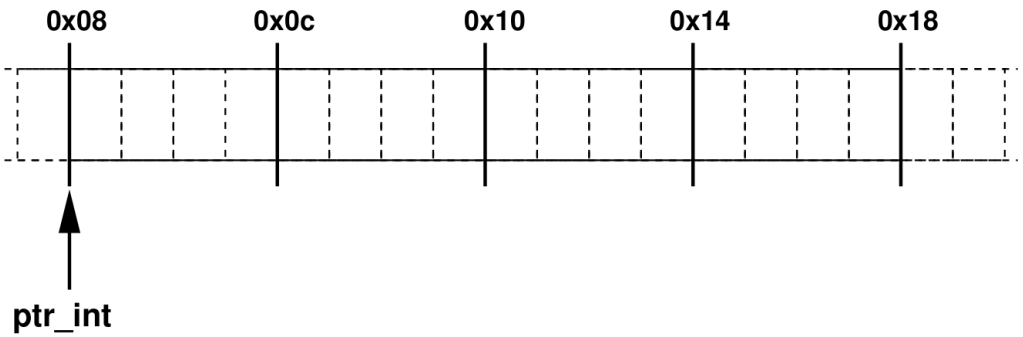
Typage des pointeurs (cf. 8.1.3)

Le typage du pointeur va influencer le comportement des opérations qui vont porter sur lui et modifie la "vision" qu'il possède de l'organisation de la mémoire, lui permettant d'accéder, ou non, à certaines adresses.

- Exemple `char` : `char* ptr_char = 0x08;`



- Exemple `int` : `int* ptr_int = 0x08;`



Typage des pointeurs (cf. 8.1.3)

- Un pointeur véhicule donc deux informations pour manipuler la mémoire :
  1. **une information d'adresse** : cette information est véhiculée par la valeur du pointeur ;
  2. **une information de taille** des données accessibles : cette information est véhiculée par le type du pointeur.

Pour aller plus loin

- Il existe une constante spécifique aux pointeurs : `NULL`
  Il s'agit d'une constante qui a pour valeur 0 (adresse 0x000000000000) et qui sert notamment à l'initialisation des pointeurs.
  Tout accès à cette adresse se solde automatiquement par un échec de l'exécution du programme (*Segmentation Fault*).
- A ne pas confondre avec les pointeurs de type `void`, qui sont appelés **pointeurs génériques**. Une variable de ce type indique juste que l'on va utiliser un pointeur sans en connaître le type exact au moment de sa déclaration. Ce typage interviendra plus tard dans le programme à l'aide de l'opérateur de conversion (ou *cast*). Un abus de langage consiste à dire que ces pointeurs ne "pointent sur rien".

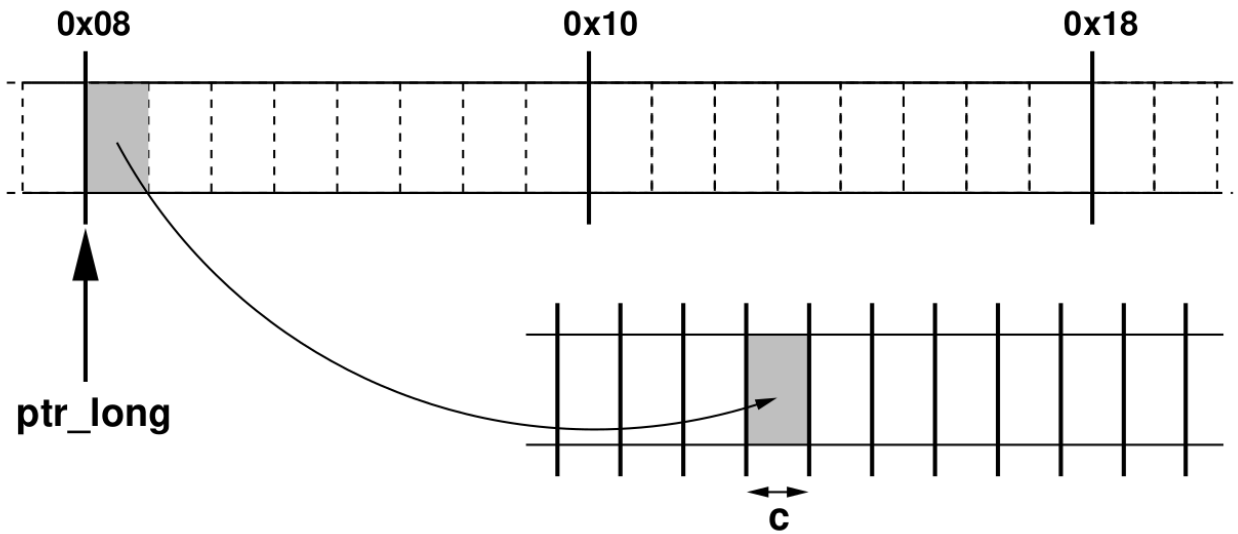
Conversion de pointeurs (cf. 8.1.1)

- Plusieurs pointeurs peuvent pointer à la même adresse et éventuellement être de types différents. On peut alors notamment **convertir** un pointeur en un pointeur d'un autre type avec l'opérateur de cast `(type)` :

```
char* ptr_char = 0x08;
int* ptr_int = (int*)ptr_char;
long int* ptr_long = (long int*)ptr_char;
```

- Conversion et déréférencement :

```
long int* ptr_long = 0x08;
char c = *((char*)ptr_long);
```



Des variables comme les autres (cf. 8.1.5)

Tout ce que nous avons vu concernant les variables s'applique aux pointeurs :

- Il est possible de créer des tableaux de pointeurs ;
- Les pointeurs peuvent être passés en tant qu'arguments à des fonctions ;
- Les pointeurs sont un type de retour licite dans les fonctions ;
- Les pointeurs peuvent être déclarés en tant que champ de structure (cf. 9) ;
- Un pointeur peut pointer sur un pointeur...

En revanche, les pointeurs obéissent à des règles d'arithmétique spéciales et tous les opérateurs ne sont pas utilisables ...

Attention :

Le C permet d'accéder en théorie à l'intégralité de la mémoire. En pratique, des zones sont inaccessibles en lecture et/ou écriture et nécessitent d'être allouées. Un accès à une zone encore non allouée se soldera par un échec à l'exécution. Les erreurs les plus fréquentes que vous allez commettre sont des accès illicites à des zones mémoires non autorisées suite à un calcul d'adresse erroné.

Arithmétique des pointeurs

- Addition pointeur / entier : (cf. 8.2.1)

  Considérons un pointeur : `type* ptr;` (`type` $= \{$`int`, `char`, ...$\}$)

  Le type du pointeur détermine automatiquement le comportement de l'addition de sorte que `ptr` ne puisse évoluer que par multiples de `type` :

  ex : `ptr + 1` pointe vers l'élément juste après celui pointé par `ptr`

  $$\texttt{(ptr + 1)}_{\texttt{type}} \equiv \texttt{(type*)ptr + (1)}_{\texttt{type}} \equiv \texttt{(char*)ptr + sizeof(type)}$$

Arithmétique des pointeurs

- Addition pointeur / entier : (cf. 8.2.1)

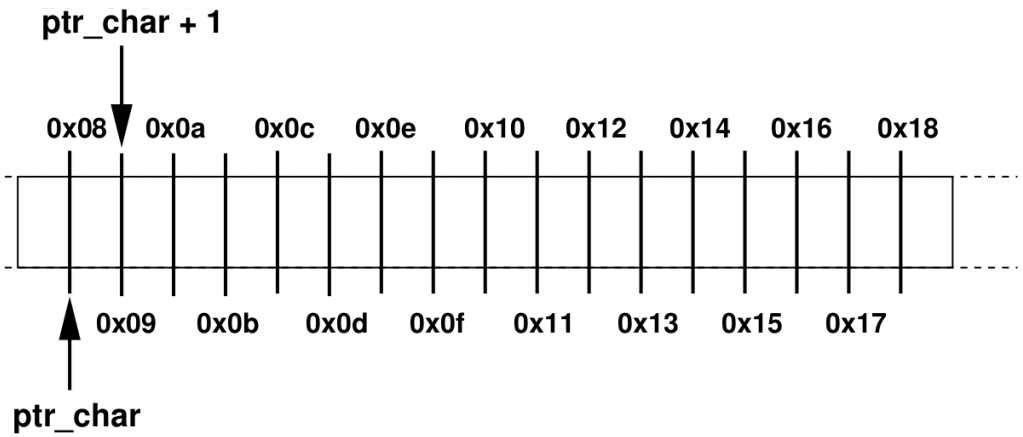
  Considérons un pointeur : `type* ptr;` (`type` = {`int`, `char`, ...})

  Le type du pointeur détermine automatiquement le comportement de l'addition de sorte que `ptr` ne puisse évoluer que par multiples de `type` :

  ex : `ptr + 1` pointe vers l'élément juste après celui pointé par `ptr`

  $$(\texttt{ptr + 1})_{\texttt{type}} \equiv (\texttt{type*})\texttt{ptr} + (\texttt{1})_{\texttt{type}} \equiv (\texttt{char*})\texttt{ptr} + \texttt{sizeof(type)}$$

  ex : `char* ptr_char = 0x08;`



Arithmétique des pointeurs

- Addition pointeur / entier : (cf. 8.2.1)

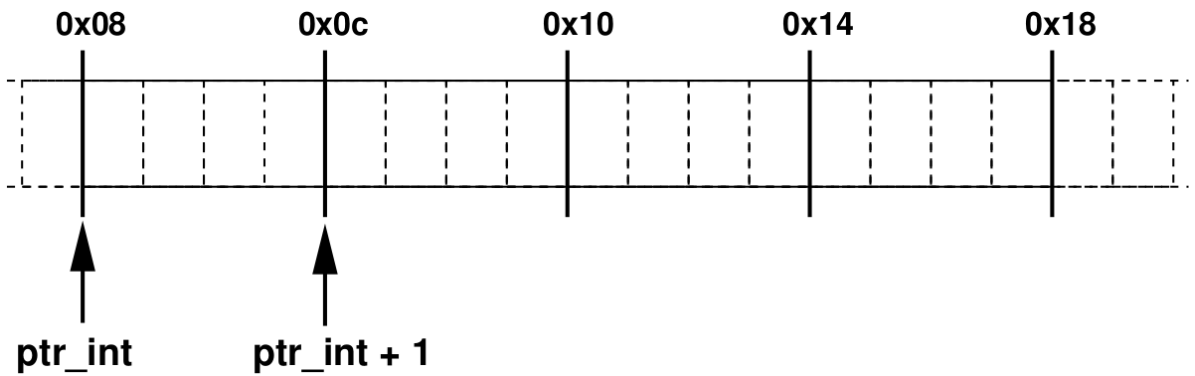
  Considérons un pointeur : `type* ptr;` (`type` = {`int`, `char`, ...})

  Le type du pointeur détermine automatiquement le comportement de l'addition de sorte que `ptr` ne puisse évoluer que par multiples de `type` :

  ex : `ptr + 1` pointe vers l'élément juste après celui pointé par `ptr`

  $$(\texttt{ptr + 1})_{\texttt{type}} \equiv (\texttt{type*})\texttt{ptr} + (1)_{\texttt{type}} \equiv (\texttt{char*})\texttt{ptr} + \texttt{sizeof(type)}$$

  ex :
  ```
  char* ptr_char = 0x08;
  int* ptr_int = (int*)ptr_char;
  ```



Arithmétique des pointeurs

- Addition pointeur / entier : (cf. 8.2.1)

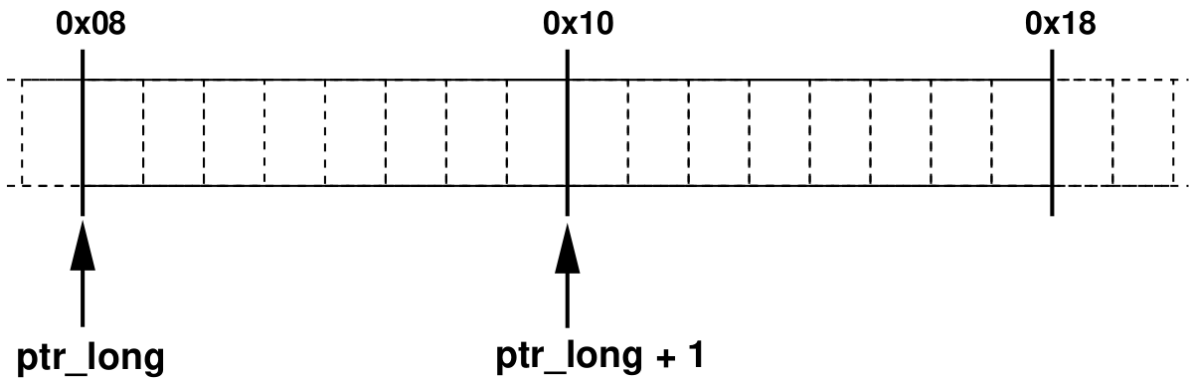
  Considérons un pointeur : `type* ptr;` (`type` = {`int`, `char`, ...})

  Le type du pointeur détermine automatiquement le comportement de l'addition de sorte que `ptr` ne puisse évoluer que par multiples de `type` :

  ex : `ptr + 1` pointe vers l'élément juste après celui pointé par `ptr`

  $$(\texttt{ptr + 1})_{\texttt{type}} \equiv (\texttt{type*})\texttt{ptr} + (1)_{\texttt{type}} \equiv (\texttt{char*})\texttt{ptr} + \texttt{sizeof(type)}$$

  ex :
  ```
  char* ptr_char = 0x08;
  int* ptr_int = (int*)ptr_char;
  long int* ptr_long = (long int*)ptr_char;
  ```



Arithmétique des pointeurs

- Addition pointeur / entier : (cf. 8.2.1)

  Considérons un pointeur : `type* ptr;` (`type` = {`int`, `char`, ...})

  Le type du pointeur détermine automatiquement le comportement de l'addition de sorte que `ptr` ne puisse évoluer que par multiples de `type` :

  ex : `ptr + 1` pointe vers l'élément juste après celui pointé par `ptr`

  $$\texttt{(ptr + 1)}_{\texttt{type}} \equiv \texttt{(type*)ptr + (1)}_{\texttt{type}} \equiv \texttt{(char*)ptr + sizeof(type)}$$

  ex :
  ```
  char* ptr_char = 0x08;
  int* ptr_int = (int*)ptr_char;
  long int* ptr_long = (long int*)ptr_char;
  ```

Généralisation :

$$\texttt{(ptr + i)}_{\texttt{type}} \equiv \texttt{(char *)ptr + i*sizeof(type)}$$

Arithmétique des pointeurs

Pour aller plus loin

En particulier les opérations suivantes **ne sont pas permises** ou aboutissent au moins à un avertissement :

- Assigner un pointeur d'un type sans le convertir (`cast`) à un pointeur d'un autre type (exception : les pointeurs génériques (`void*`)) ;
- Additionner deux pointeurs y compris de même type ;
- Additionner un nombre flottant à un pointeur ;
- Utiliser la multiplication/division avec un pointeur ;
- Utiliser les opérateurs de décalage >> et << sur un pointeur.

Tableaux et pointeurs

Les pointeurs et les tableaux sont en réalité liés et on peut accéder aux éléments d'un tableau par la notation tableau [ ] ou la notation pointeur :

En effet, considérons un tableau de type `type` :

```
type tab[SIZE_TAB];
```

En réalité `tab` est un pointeur contenant l'adresse de la première case mémoire :

```
type* ptr = &tab[0]
```

`tab` ≡ `ptr`

Alors pour accéder à `tab[i]`, une variable de type `type`, on effectue un décalage de `i*sizeof(type)` octets par rapport à l'élément 0 de `tab` :

`tab[i]` ≡ `*(tab + i)` (notation pointeur)

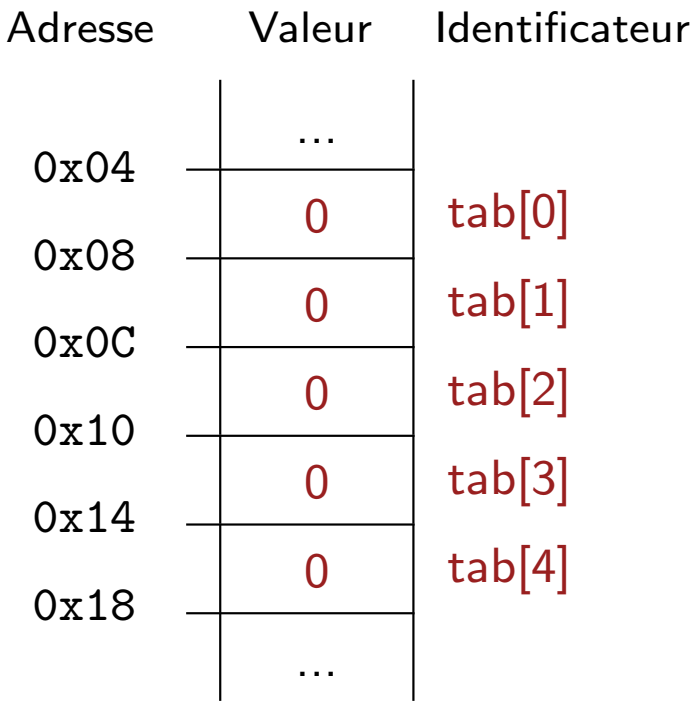
Pour aller plus loin

La notation pointeur est plus compacte et permet par exemple la pre/post in/de-crémentation (ex. : `*tab++`) mais peut complexifier la lecture des codes.
→ À utiliser avec parcimonie

Dans la mémoire

Exemple d'illustration d'écriture dans la mémoire physique :

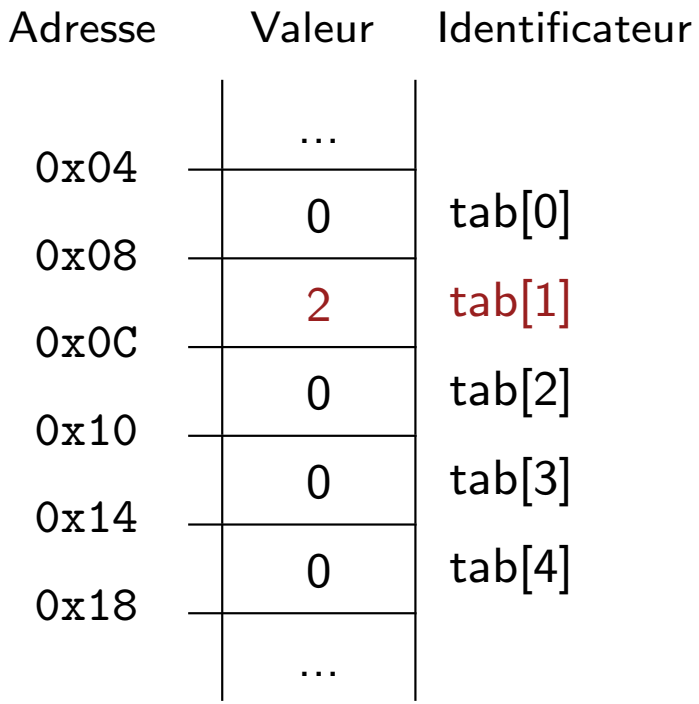
```
int tab[5] = {};
```



Dans la mémoire

Exemple d'illustration d'écriture dans la mémoire physique :

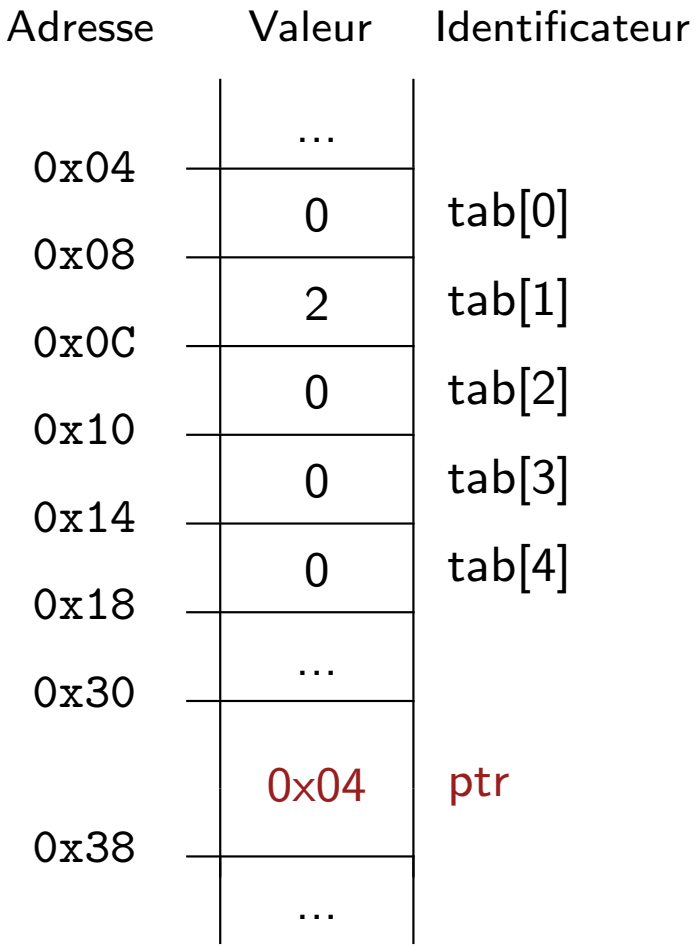
```
int tab[5] = {};
tab[1] = 2;
...
```



Dans la mémoire

Exemple d'illustration d'écriture dans la mémoire physique :

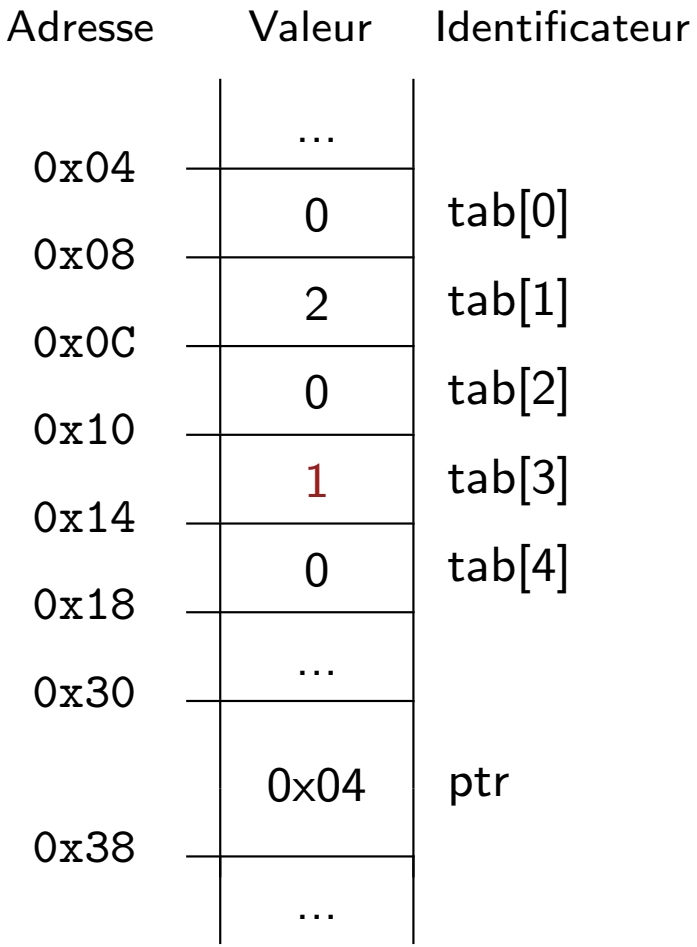
```
int tab[5] = {};
tab[1] = 2;
...
int* ptr = tab;
```



Dans la mémoire

Exemple d'illustration d'écriture dans la mémoire physique :

```
int tab[5] = {};
tab[1] = 2;
...
int* ptr = tab;
*(ptr+3) = 1;
```



Tableaux et pointeurs

- ⚠ Retourner un tableau, revient en réalité à retourner le pointeur vers la première case.
  "Retourner un tableau" n'effectue donc pas de copie du tableau.

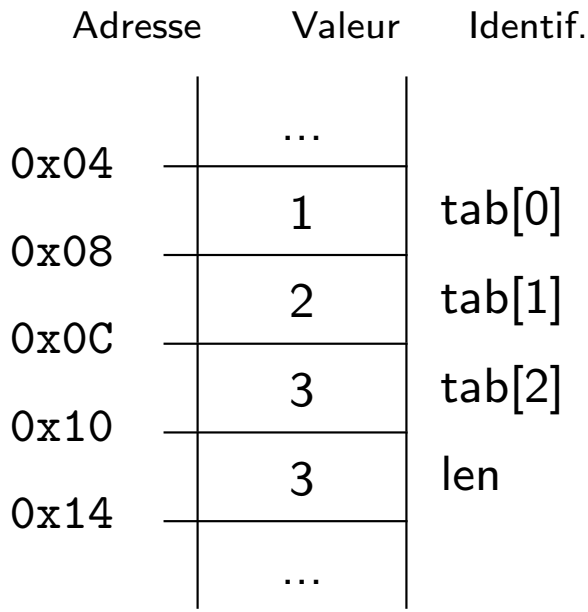
- ⚠ Équivalence entre `type* nom` et `type nom[]` : (pour `type` $\neq$ `void*`)
  ex : `int f(int len, int* tab);` $\equiv$ `int f(int len, int tab[]);`
  On peut néanmoins donner une indication sur le type d'objet manipulé en précisant `[]` pour un tableau (souvent d'entiers), et `*` dans le cas d'un pointeur vers une unique variable ou une chaîne de caractères (`char*`).
  Seule différence à l'initialisation :
  ex : `char s[] = "toto";` $\neq$ `char* s = "toto"; //s constante`

Pour aller plus loin

- ⚠ En réalité le pointeur `tab` est déclaré comme une constante :
  `type tab[10];` (tab $\equiv$ `const type*`)
  Donc toute modification de la valeur de `tab` est interdite.
  En pratique `tab` est souvent donné en argument (`type tab[]`), donc sa valeur est recopiée lors de l'appel et devient alors modifiable.

Tableaux et appels de fonction

```
1  #include<stdio.h>
2
3  void inverse_tab(int l, int t[]) {
4    int tmp = 0;
5    for (int i=0; i<l/2; i++) {
6      tmp = t[i];
7      t[i] = t[l-i-1];
8      t[l-i-1] = tmp;
9    }
10 }
11
12 int main() {
13   int tab[] = {1,2,3};
14   int len = 3;
15   inverse_tab(len, tab);
16   return 0;
17 }
```

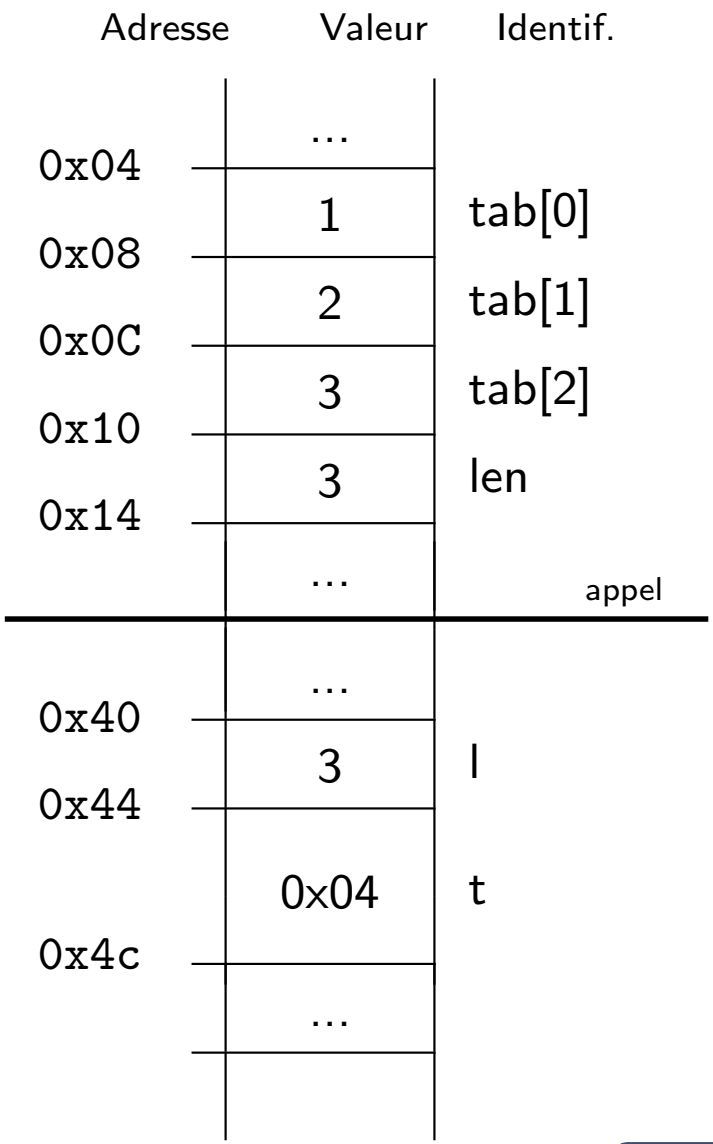


- déclaration du tableau (l14-l15)

Tableaux et appels de fonction

```
1  #include<stdio.h>
2
3  void inverse_tab(int l, int t[]) {
4    int tmp = 0;
5    for (int i=0; i<l/2; i++) {
6      tmp = t[i];
7      t[i] = t[l-i-1];
8      t[l-i-1] = tmp;
9    }
10 }
11
12 int main() {
13   int tab[] = {1,2,3};
14   int len = 3;
15   inverse_tab(len, tab);
16   return 0;
17 }
```

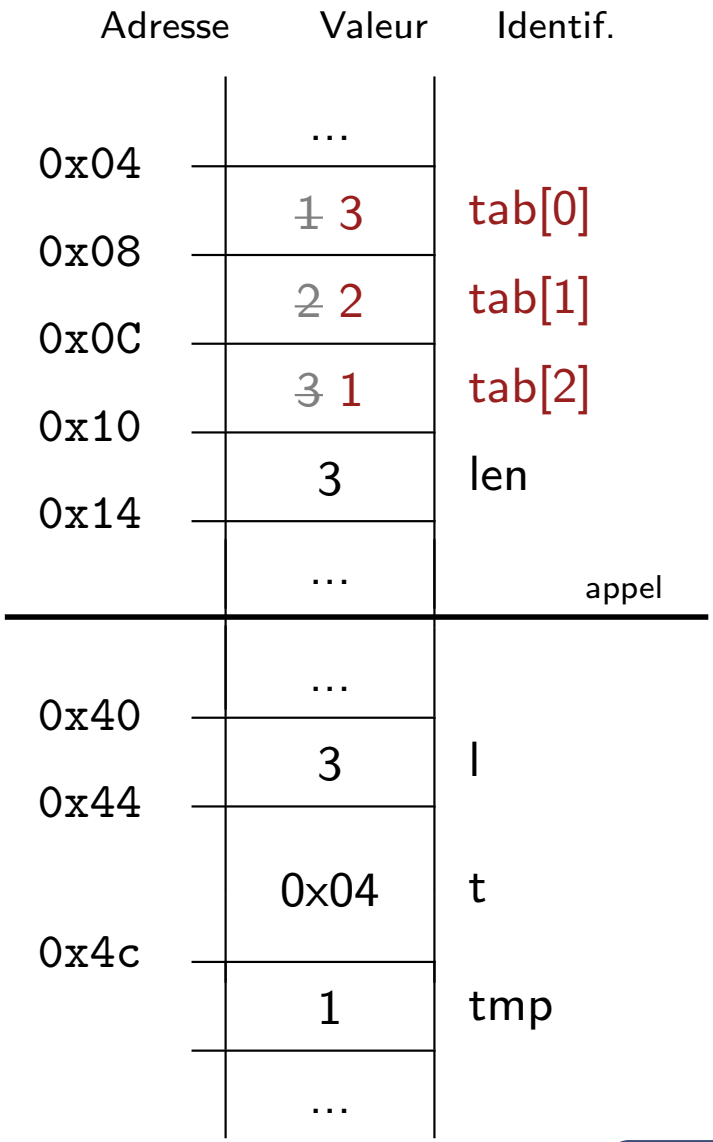
- déclaration du tableau (l14-l15)
- appel de `inverse_tab` (l16)



Tableaux et appels de fonction

```
1  #include<stdio.h>
2
3  void inverse_tab(int l, int t[]) {
4    int tmp = 0;
5    for (int i=0; i<l/2; i++) {
6      tmp = t[i];
7      t[i] = t[l-i-1];
8      t[l-i-1] = tmp;
9    }
10 }
11
12 int main() {
13   int tab[] = {1,2,3};
14   int len = 3;
15   inverse_tab(len, tab);
16   return 0;
17 }
```

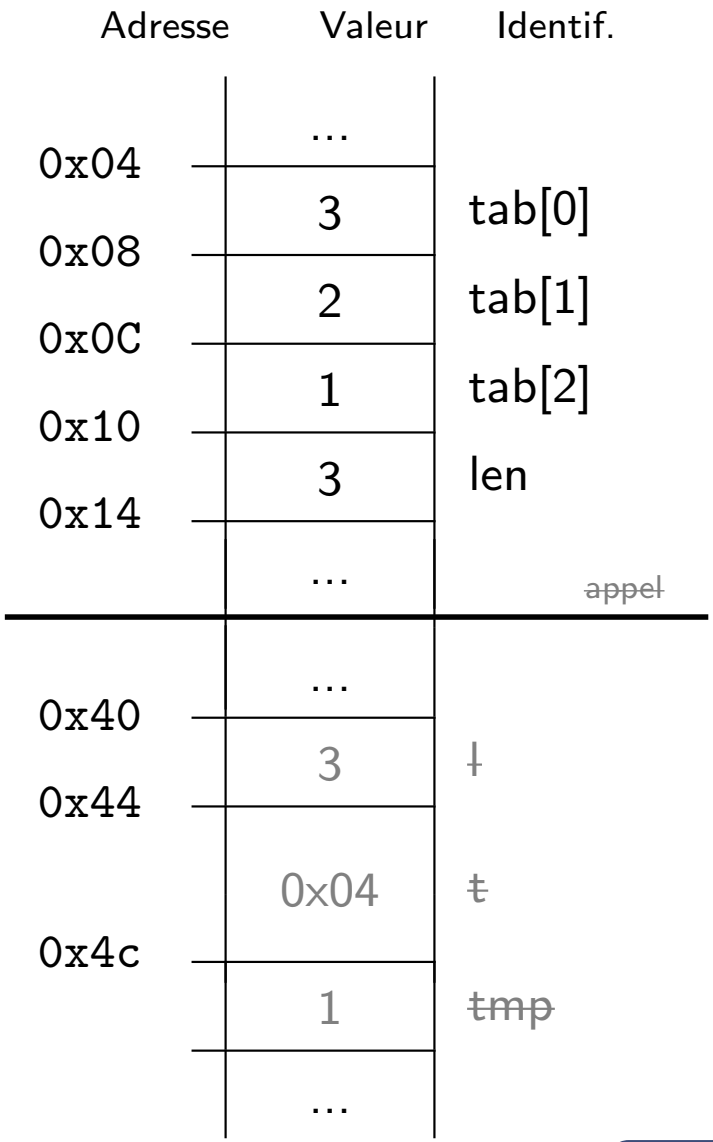
- déclaration du tableau (l14-l15)
- appel de `inverse_tab` (l16)
- exécution de `inverse_tab` (l4-l9)



Tableaux et appels de fonction

```
1  #include<stdio.h>
2
3  void inverse_tab(int l, int t[]) {
4    int tmp = 0;
5    for (int i=0; i<l/2; i++) {
6      tmp = t[i];
7      t[i] = t[l-i-1];
8      t[l-i-1] = tmp;
9    }
10 }
11
12 int main() {
13   int tab[] = {1,2,3};
14   int len = 3;
15   inverse_tab(len, tab);
16   return 0;
17 }
```

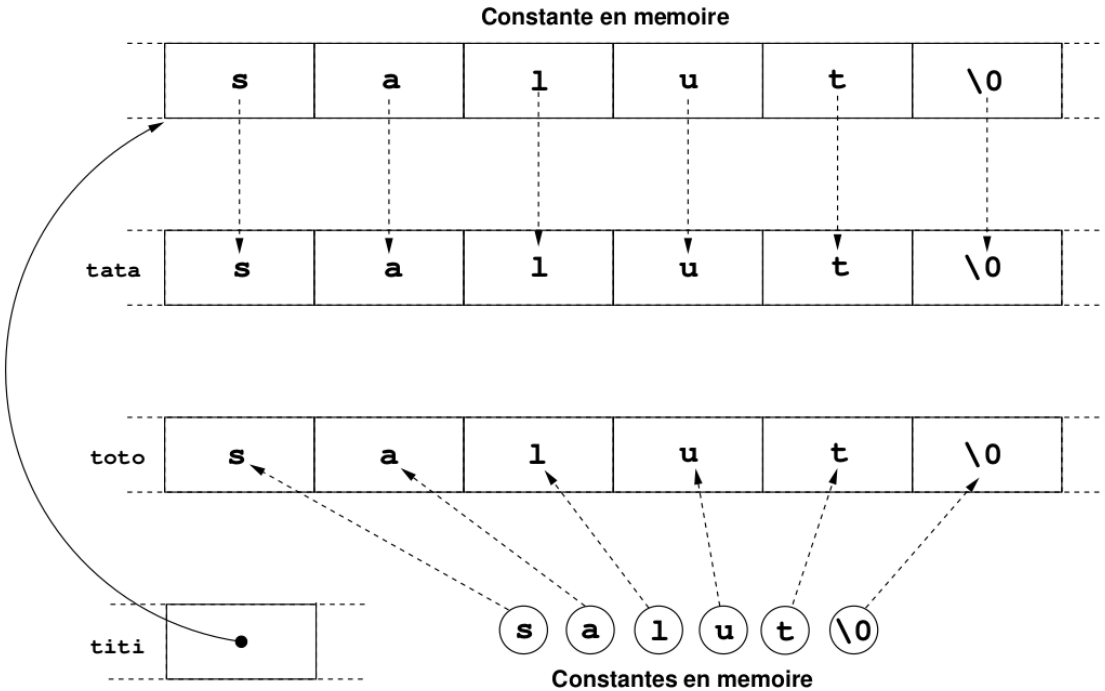
- déclaration du tableau (l14-l15)
- appel de `inverse_tab` (l16)
- exécution de `inverse_tab` (l4-l9)
- sortie de `inverse_tab` (l10)



Chaînes de caractères

- Chaînes de caractères = tableau de caractères avec un zéro terminal '\0'
- `char* truc;` est simplement un pointeur sur des octets en mémoire
- Déclaration :

```
char toto[] = {'s','a','l','u','t','\0'}; //Chaîne de caractères (6 octets)
char tata[] = ''salut'';                  //Chaîne de caractères (6 octets)
char* titi = ''salut'';                   //Chaîne de caractères constante!
```



Exercice : Tableaux d'entiers / Pointeurs

- Reprendre le code de parcours d'un tableau d'entiers (`c2_tab.c`) et afficher l'adresse de chaque case mémoire.

- On veut écrire une fonction qui retourne en même le minimum et le maximum d'un tableau.
  Écrire la fonction :
  ```
  void min_max(int l, int t[], int* min, int* max)
  ```
  telle que après l'appel min (resp. max) représentent le minimum (resp. le maximum) du tableau.

  Attention, si `min` ou `max` sont nuls (`NULL`), on ne doit pas faire la recherche associée ni tenter de les déréférencer.

- Écrire une fonction qui calcule la somme de deux tableaux d'entiers de même taille, membre à membre.
  On pourra choisir d'écrire cette somme dans un des tableaux... ou trouver une façon de modifier un tableau contenant le résultat.

Exercice : Chaînes de caractères / Pointeurs

Écrire les fonctions suivantes :

- `char* my_stpcpy(char* dst, const char* src)`, qui copie la chaîne source dans la destination. Cette fonction retourne un pointeur sur le 0 terminal de la destination (variante de `strcpy` très utile).
  Quel doit être le prérequis sur `dst` pour pouvoir accueillir la chaîne `src` ?
- `char* my_strcat(char* dst, const char* src)`, qui concatène la chaîne `src` à la chaîne `dst` et retourne un pointeur sur `dst`.
  Attention à la taille de `dst`.
- `int my_strcmp(const char* s1, const char* s2)`, qui compare les chaînes `s1` et `s2` et retourne un nombre négatif si `s1` est avant `s2` dans l'ordre lexicographique, positif si `s1` est après `s2`, zéro s'il s'agit des mêmes chaînes.

Reprendre les fonctions (y compris `my_strlen`) pour les écrire avec la notation pointeur uniquement type : `*str++`

Pointeurs de fonction (cf. 8.4.3)

Pour aller plus loin

Mécanisme puissant pour gagner en abstraction dans les programmes :

- `type_retour_t (* nom_fonction)(liste_des_arguments)`
  Cette déclaration signifie : `nom_fonction` est un pointeur sur une fonction qui prend comme arguments `liste_des_arguments` et qui retourne un résultat de type `type_retour_t`.

  ex. :
  ```
  int (* super_fonction)(int arg1, int* arg2, void* arg3);
  ```
  ou (juste les types des arguments)
  ```
  int (* super_fonction)(int, int*, void*);
  ```

- Le nom d'une fonction est également "un pointeur sur cette fonction" :
  `nom_fonction` ≡ `&nom_fonction`
  et pour déréférencer un pointeur de fonction (par exemple pour effectuer un appel avec des arguments) :
  `nom_fonction(args ...)` ≡ `*nom_fonction(args ...)`

Pointeurs de fonction (cf. 8.4.3)

Pour aller plus loin

Donc en pratique ....

```c
#include<stdio.h>

int f(int i, int j) {
  return i+j;
}

int (*pf)(int, int); // Déclare un pointeur de fonction

int main(int argc, char* argv[]) {
  int l = 3, m = 5;
  pf = &f;
  printf("Leur somme est de : %u\n", pf(l,m));
  return 0;
}
```

Pointeurs de fonction (cf. 8.4.3)

Pour aller plus loin

Très utile pour l'appel d'une fonction parmi une liste de fonctions disponibles. Possibilité de faire un tableau de pointeurs de fonctions et d'appeler la fonction dont on connaît l'indice de son pointeur dans le tableau :

```c
#include<stdio.h>

int somme(int i, int j){
  return i+j;
}
int multiplication(int i, int j){
  return i*j;
}
int quotient(int i, int j){
  return i/j;
}
int modulo(int i, int j){
  return i%j;
}
```

```c
typedef int (*fptr)(int, int);
fptr ftab[4];

int main(int argc, char* argv[]) {
  int i = 31, j = 4, n = 2;
  ftab[0]=&somme; // Initialise le tableau de
  ftab[1]=&multiplication; // ptr de fonctions
  ftab[2]=&quotient;
  ftab[3]=&modulo;
  if (n < 4)
    printf("Résultat : %u\n", ftab[n](i,j));
  else
    printf("Mauvais numéro de fonction\n");
  return 0;
}
```

Pointeurs génériques

De type `void*` :

- `void* ptr;` représente un pointeur quelconque, compatible avec tous les types de pointeurs (en lecture comme en écriture)
- Pas possible d'utiliser l'opérateur `*` ou de faire d'arithmétique sur ce type (on ne peut pas ni lire ni écrire dans le néant).

Permet de passer des arguments génériques aux fonctions :

```c
#include<stdio.h>

void toto(void* v){
  char* c = (char*) v;
  c[0] = 65; //code de 'A'
}

int main(int argc, char* argv[]) {
  char str[] = "salut";
  toto(str);
  printf("%s\n", str);
  return 0;
}
```

Exercice : Pointeurs génériques

Écrire les fonctions génériques (consulter le manuel pour en savoir plus) :

- `void* my_memcpy(void* dst, const void* src, size_t len)`, qui copie la zone pointée par `src` dans `dst` sur `len` octets. Elle retourne le pointeur `dst`.
- `void* my_memset(void* src, int c, size_t len)`. La fonction `memset` écrit `len` octets de la valeur c (convertie vers un `unsigned char`) vers la zone de mémoire `src`. Cette fonction retourne son premier argument.
- `int my_memcmp(void* src, void* dst, size_t len)`, qui compare les zones mémoires `src` et `dst` sur une longueur maximale `len`. (voir `strcmp`).
- Bonus : `void swab(const void* src, void* dest, size_t nbytes)`. Cette fonction copie n octets de `src` vers `dest` en permutant les octets pairs et les impairs.

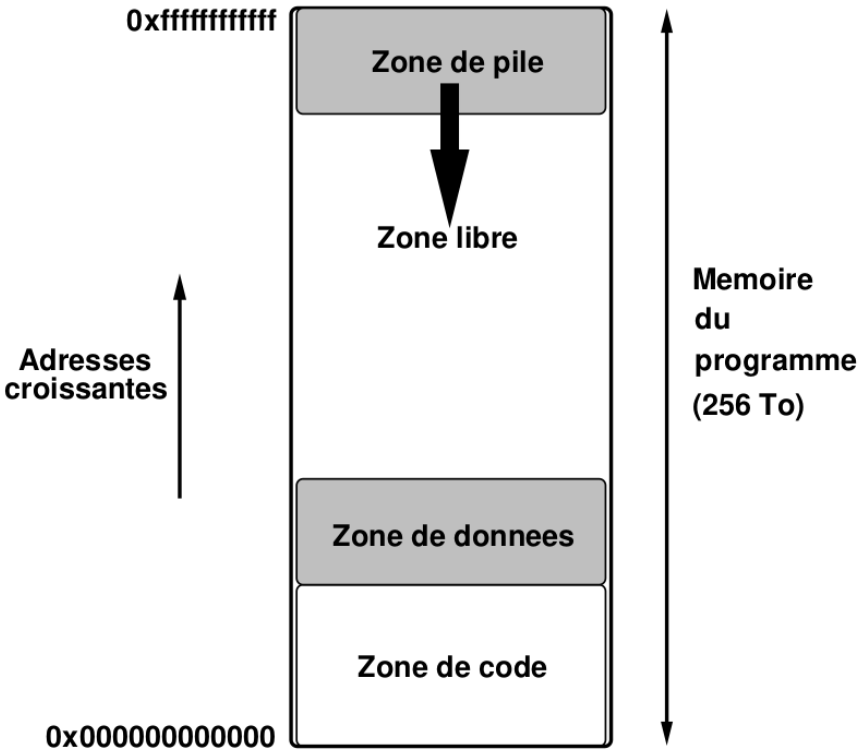
Allocation statique de mémoire

Jusqu'à présent :

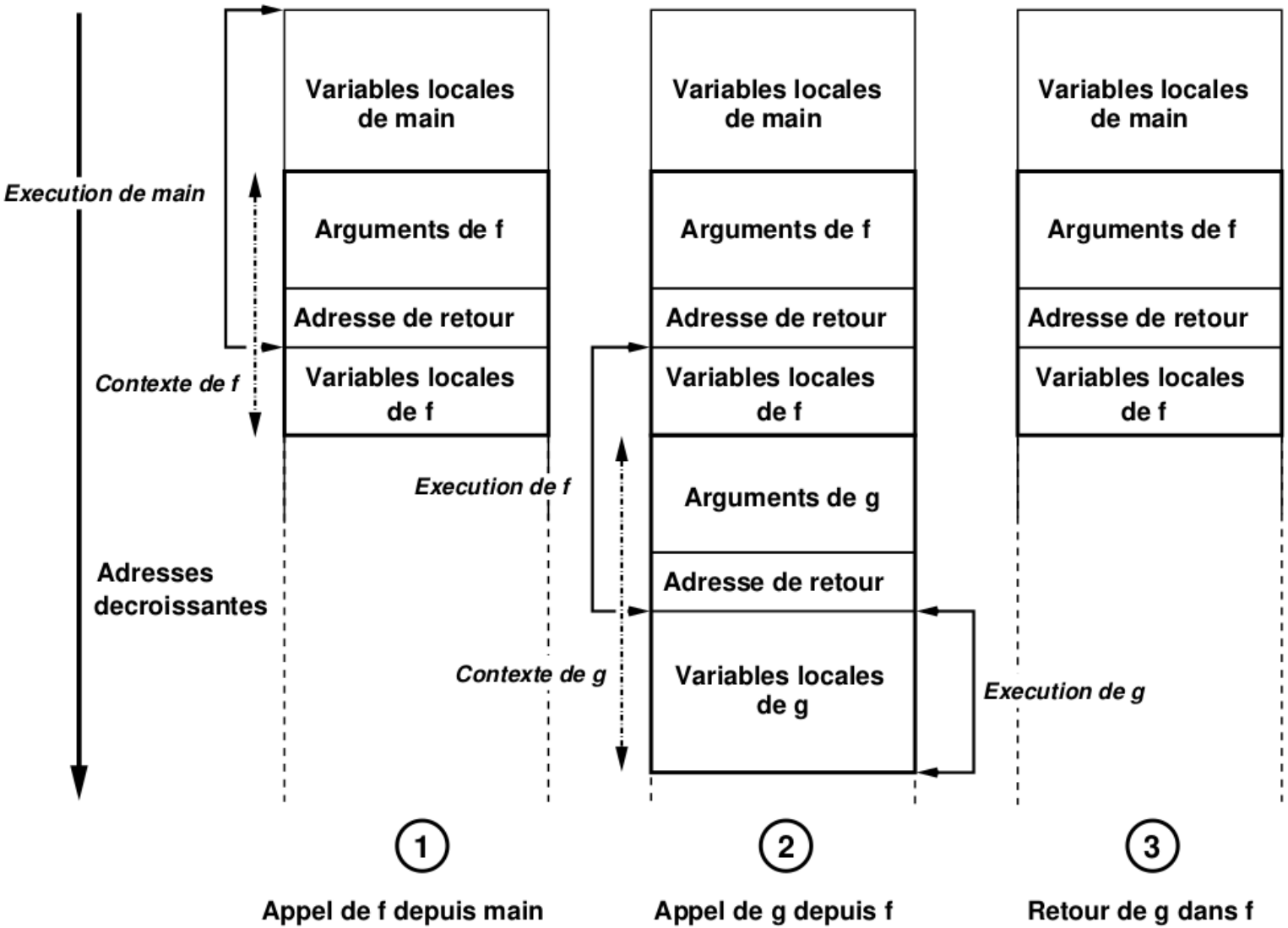
- Contexte **statique** où la taille des données est connue à la compilation.
- Les variables possèdent une taille bien déterminée (critère de compilation).

Dans la mémoire :

Les variables locales sont par défaut stockées dans la zone de **pile** (stack) (de taille limitée) et sont :
- automatiquement allouée au moment de la déclaration de la variable
- automatiquement libérée quand le programme sort du bloc d'instructions où cette variable est déclarée.



Exemple d'exécution d'un programme (cf. 4.4.4)



Exemple d'exécution d'un programme

Que se passe-t-il dans ce programme ?

```c
#include<stdio.h>

int* sum_tab(int tab1[], int tab2[]) {
  int tab[4];
  for(int i=0; i<4; i++)
    tab[i] = tab1[i] + tab2[i];
  return tab;
}

int main(int argc, char* argv[]) {
  int tab1[4] = {1,3,4,5};
  int tab2[4] = {1,-3,2,8};
  int* tab = sum_tab(tab1, tab2);
  printf("Adresse de tab %p\n", tab);
  return 0;
}
```

Exemple d'exécution d'un programme

Que se passe-t-il dans ce programme ?

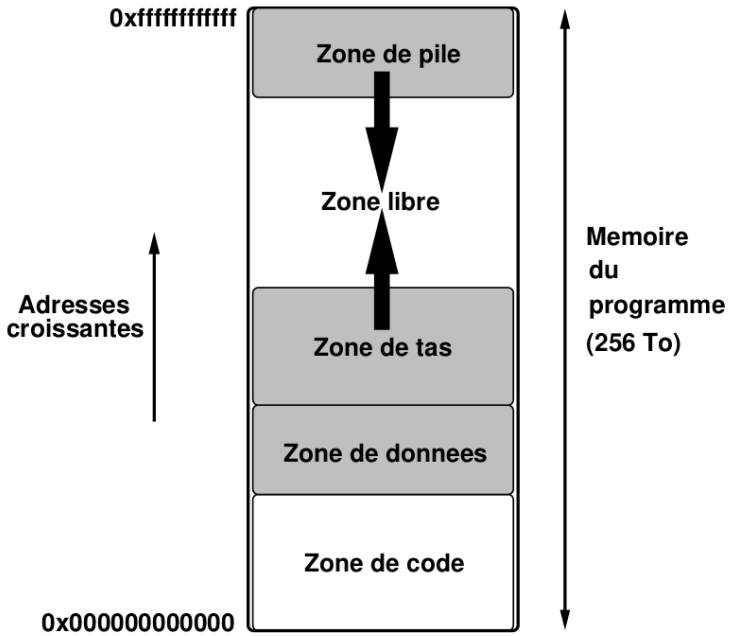
```c
#include<stdio.h>

int* sum_tab(int tab1[], int tab2[]) {
  int tab[4];
  for(int i=0; i<4; i++)
    tab[i] = tab1[i] + tab2[i];
  return tab;
}

int main(int argc, char* argv[]) {
  int tab1[4] = {1,3,4,5};
  int tab2[4] = {1,-3,2,8};
  int* tab = sum_tab(tab1, tab2);
  printf("Adresse de tab %p\n", tab);
  return 0;
}
```

Que faire lorsque la taille n'est connue qu'en cours d'exécution du programme ?
Comment déclarer une variable dans une fonction et renvoyer son adresse ?

Allocation dynamique de mémoire (cf. 8.5)

→ Grâce au **tas** (heap) : zone dans laquelle peuvent être allouées des données pendant l'exécution du programme.

- Fonctions de manipulation dans `stdlib.h`
- Les pointeurs vont pouvoir servir pour référencer des zones mémoire allouées dynamiquement dans le tas (allocation dynamique de tas).
- Les variables sont libérées uniquement manuellement.
- Si le tas augmente, les adresses ↑. Si la pile augmente, les adresses ↓.



Les deux principales fonctions d'allocation dynamiques sont (`man nom_fct`) :

- `void* malloc(size_t size);`
- `void free(void* ptr);`

`malloc` (Memory ALLOCation) permet de réserver la mémoire dans le tas.
Prend en paramètre le nombre d'octets à réserver.
`free` libère ensuite cet espace mémoire, qui devient donc réutilisable, lorsque l'on n'utilise plus cette variable.

ex :
```
int* tab = (int*) malloc(nbr_ele*sizeof(int));
...
free(tab);
```

 Toujours libérer la mémoire (et qu'une seule fois) avec `free` pour éviter les problèmes de mémoire.

Pour aller plus loin

Il existe des variantes de `malloc` :

- `void* calloc(size_t nbr_ele, size_t size);`
  (initialisation et mise à zéro. Attention calloc prend deux arguments)
- `void* realloc(void* ptr, size_t size);`
  (permet d'agrandir l'espace mémoire)

Exercice : Allocation dynamique

- Écrire la fonction `int* alloc_vec(int len, int val)` qui alloue dynamiquement la mémoire (`malloc`) pour un vecteur de taille `len`, puis qui l'initialise à la valeur `val`.
- Écrire la fonction `void free_vec(int* vec)`, qui libère la mémoire (`free`) occupée par le vecteur.
- Reprendre le code de la fonction `sum_tab` qui calculait la somme de deux tableaux terme à terme pour qu'elle retourne un pointeur vers le tableau alloué dynamiquement dans la fonction.

Pointeurs n°1 et Tableaux

Résumé :

- **Pointeur** = adresse d'une variable dans la mémoire
  - Forcément de type entier positif (affichage avec `%p` (hexa))
  - Pointeur vers n'importe quel type sur 8 octets (archi 64 bits)
  - Syntaxe :

```
int* ptr = NULL; //ptr est un pointeur (*) vers une
  //variable int, et est initialisé à NULL (≡ 0)
//*ptr = 5; //Segfault ! L'adresse 0 n'est pas valide
int a, b; //a et b entiers de type int
ptr = &a; //ptr récupère l'adresse de la variable a
*ptr = 5; //On met 5 à l'adresse contenue dans ptr
printf("a=%d\n",a); //a vaut à présent 5
```

- Passage par variable : Modifier des variables depuis une fonction
  Ex. : `void swap(int* ptr_a, int* ptr_b){...}`
  À l'appel `swap(&a, &b);`, les adresses `&a` et `&b`, sont évaluées, écrites dans la mémoire de la fonction puis perdues à la sortie
  Mais les modifications affectent `a` et `b` dans la fonction appelante

Pointeurs n°1 et Tableaux

Résumé :

- **Pointeurs et tableaux :**
  - Arithmétique pointeurs considère le type pointé : `type* ptr`
    `ptr+=1;` //pointeur a été incrémenté de sizeof(type)
    "On se déplace dans la mémoire par paquet de sizeof(type)"
  - L'identificateur d'un tableau est un pointeur vers la première case :
    `type tab[SIZE_TAB];` //statique, perdu en sortie de fonction
    `type* ptr = &tab[0];` //adresse de la première case
    `tab` ≡ `ptr` //tab est de type (type*)
    `tab[i]` ≡ `*(tab + i)` //(notation pointeur)
    `return tab;` //renvoie le pointeur (type*) invalide car statique
  - Fonction de calcul de longueur de chaîne de caractères :

```
int strlen(char* str) { ≡ int strlen(char str[])
  char* str_cpy = str;
  while(*str++){}; ≡ while(*str++ != 0)
  return str-strcpy-1;
}
```

Pointeurs n°1 et Tableaux

Résumé :

- **Pointeurs génériques** : Permet de gérer des types différents
  - On donne tout type de pointeur à une fonction, qui le récupère en argument sous le type `void*` (ne contenant que l'information d'adresse)
  - Cast du `void* ptr_input` : `char* ptr = (char*) ptr_input;`
    On lit alors la mémoire octet par octet depuis `ptr`
  - Fonctions génériques de comparaison (`memcmp`), copie (`memcpy`), ...
- **Allocation dynamique** : Mémoire désallouée que manuellement
  - Permet de récupérer des tableaux déclarés dans une fonction
  - Nécessaire si la taille du tableau n'est pas connue à la compilation
  - Syntaxe : `type* tab = (type*) malloc(sizeof(type)*size);`
    `//Tableau de size éléments type (on compte en octets)`
  - Ne pas oublier de libérer la mémoire : `free(tab);`

Interactions - Le retour

- Utilisation de la fonction `int scanf(const char* format, ...)` :

  Permet la saisie de données durant l'exécution du programme
  Format identique à `printf` (mais pas tous les types disponibles)
  Le `%s` ne lit jamais d'espace (s'arrête au premier rencontré)

Exercice : scanf

Compléter ce programme pour lire un entier, un flottant, et une chaîne de caractères (sans espace) en une seule commande :

```c
#include<stdio.h>

int main(int argc, char* argv[]) {
  int a;
  float f;
  char s[50];

  printf("Paramètres à entrer :\n");
  //Appel à scanf (à compléter)

  printf("%d paramètres lus : a=%d, f=%f, s=%s\n", r, a, f, s);
  return 0;
}
```

Interactions - Le retour

- Utilisation de la fonction `fgets` :

  Permet la saisie de texte comprenant des espaces
  La conversion s'effectue a posteriori

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

int main(int argc, char* argv[]) {
  char buffer[51] = {}; // penser au 0 à la fin
  fgets(buffer, 50, stdin); // stdin = entrée standard

  printf("J'ai lu : %s, (%lu)\n", buffer, strlen(buffer));
  printf("comme un entier ? %d\n", atoi(buffer));
  printf("comme un double ? %lf\n", atof(buffer));

  return 0;
}
```

Exercice : Interactions utilisateur

- Écrire une fonction qui demande à l'utilisateur la taille du tableau d'entiers qu'il souhaite initialiser.
  Allouer dynamiquement ce tableau, puis permette à l'utilisateur de le remplir valeur par valeur. (`scanf`)

```
Quelle taille fait le tableau ? : 10
Valeur 1 ? : 3
Valeur 2 ? : 51
...
```

  Retourner le pointeur vers la première case du tableau

- Que se passe-t-il si on tape dans le terminal plusieurs valeurs séparées par des espaces ?

Exercice : (Bonus) Jeu du + ou -

Écrire un programme qui demande à l'utilisateur de deviner un nombre pris au hasard, par propositions successives.

- Choisir un nombre au hasard à l'aide de la fonction `rand()` (`stdlib.h`).
- Demander une proposition à l'utilisateur et indiquer si la proposition est supérieure ou inférieure au nombre à deviner. Le programme continue à demander une valeur jusqu'à ce que la proposition de l'utilisateur soit correcte. Il affiche alors le nombre de tentatives.
  Pour rappel, il n'est pas possible de générer des valeurs réellement aléatoires, on utilise plutôt des générateurs pseudo-aléatoires. Souvent, on utilise l'heure et la date pour initialiser le générateur, sinon, la séquence pseudo-aléatoire sera toujours identique. Cette initialisation est effectuée par la fonction `srand()`. La syntaxe est alors la suivante :

```
#include <time.h>
...
srand(time(NULL));
```

- On peut également afficher le temps mis par l'utilisateur pour trouver le résultat (cf. Mesure du temps d'exécution)

Exercice : (Bonus) Jeu du Morpion

Écrire un programme qui permet de jouer au morpion à neuf cases.

Principe du jeu : les deux joueurs inscrivent tour à tour un symbole dans l'une des cases libres de la grille ($\times$ pour le joueur 1, o pour le joueur 2) ; le premier joueur qui réussit à aligner 3 symboles identiques en ligne, colonne ou diagonale est vainqueur, et le match est nul si toutes les cases sont remplies et si aucun joueur n'a gagné. Le programme affichera la grille de la manière suivante :

	1	2	3
1	.	.	o
2	.	$\times$	.
3	.	.	.

Tour à tour, chaque joueur saisira le numéro de ligne suivi du numéro de colonne où il souhaite jouer, puis la grille sera affichée à nouveau en remplaçant le . de la case choisie par le symbole du joueur ($\times$ ou o). Le programme déclarera une victoire d'un joueur ou un match nul si applicable.

Rappel : Pour saisir un nombre dans la variable a (int) : `scanf("%d", &a);`

Variables locales

- **Visibilité/portée :** les endroits dans le code où la variable est utilisable
  Pour une variable locale : à partir de son point de déclaration et jusqu'à la fin du bloc d'instructions (cf. 4.4.5)

  ⚠ Masquage : Une variable est visible dans sa portée, sauf là où elle est masquée par une variable plus locale possédant le même identificateur.

```c
#include<stdio.h>

int main(int argc, char* argv[]) {
  int i = 0, a = 5, b = 0;
  for(int i=0; i<a; i++) {
    b += i;
  }
  //A cet instant, seuls a, b et "un" i sont connus
  printf("i=%d\n", i); //i=0. Le i de la boucle a masqué le premier
  { //Nouveau bloc d'instructions
    int a = b; //Cette nouvelle variable a masque l'ancienne
  }
  printf("a=%d\n", a); //a=5
  return 0;
}
```

Variables locales

- **Durée de vie** : intervalle de temps où une variable existe (cf. 4.4.6)

  ⚠ Par défaut la durée de vie d'une variable locale (automatique) est celle du bloc d'instructions.

  - La durée de vie d'une variable locale statique (`static`) est celle du programme, *c.à.d.*, qu'à chaque exécution du bloc d'instructions où la variable est déclarée, on réutilise la variable de départ.

```c
#include<stdio.h>

int main(int argc, char*argv[]) {
  int i = 0;
  for(i=0; i<10; i++) {
    int x = 0;
    ++x;
    printf("%d\n", x);
  }
  return 0;
}
```

Affiche 1, 1, ..., 1

```c
#include<stdio.h>

int main(int argc, char*argv[]) {
  int i = 0;
  for(i=0; i<10; i++) {
    static int x = 0;
    ++x;
    printf("%d\n", x);
  }
  return 0;
}
```

Affiche 1, 2, ..., 10

Variables locales

- Fonctions et variables statiques

Même variable de départ mais portées différentes pour chaque fonction.
Ex. d'utilisation : compteur d'accès à une fonction.

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

void up(void) {
    static int k = 0;
    k++;
    printf("k=%d\n", k);
}
void down(void) {
    static int k = 0;
    k--;
    printf("k=%d\n", k);
}
int main() {
    int i;
    for (i = 0; i < 3; i++) {up();}
    for (i = 0; i < 2; i++) {down();}
    return 0;
}
```

Affiche : k=1, k=2, k=3, k=-1, k=-2

Variables globales

Mêmes principes :

- **Visibilité :** Dans tout le fichier source (et au delà avec `extern`)
- **Durée de vie :** Durée d'exécution du programme

```c
#include<stdio.h>

int x = 0; //Variable globale

void print x(){
  printf("x dans print x : %d\n", x);
}

int main(int argc, char*argv[]) {
  print x(); //Affiche 0
  x = 2;
  print x(); //Affiche 2
  printf("x : %d\n", x); //Affiche 2
  int x = 3; //Variable locale à la fonction main
  printf("x : %d\n", x); //Affiche 3
  print x(); //Affiche 2
  return 0;
}
```

Variables globales

⚠ Différentes des macros ou constantes de préprocesseur (`#define`) (cf. 10.2.1)

À la compilation le processeur remplace l'identificateur par la valeur :

→ pas de place en mémoire

→ possibilité d'utiliser des syntaxes avec arguments

```c
#define FOO 42
#define BAR (FOO + 3)
#define BAZ FOO + 3

int a = FOO; //a == 42
int b = BAR * 2; //b == 90
int c = BAZ * 2; //c == 48
```

```c
#define MAX(a,b) a>b?a:b

int m = MAX(foo(), bar());
//Nombre d'appels de foo et
      bar ?
int f = foo(), b = bar();
int max = MAX(f, b);
```

```c
#include<stdio.h>
#define SIZE_TAB 10
#define MAX(a,b) a>b?a:b

int max_tab(int tab[]) {
  int max = tab[0];
  for(int i=1; i<SIZE_TAB; i++)
    max = MAX(max, tab[i]);
}

int main(int argc, char*argv[]) {
  int tab[SIZE_TAB] = {1,4,3,-1,5};
  printf("Max : %d\n", max_tab(tab));
  return 0;
}
```

→ Annexe : Remplacer un paramètre par une chaîne de caractères

Instruction conditionnelle - le retour

`switch` : Plusieurs choix possibles selon l'évaluation d'une expression (cf. 5.1.2)

- Syntaxe : `switch(cond) {case(val1): /*instru.*/; break; ...}`
- `cond` seulement de type entier
- Valeurs `val` des branches constantes
- Attention aux `break` ! 

```c
if (choix == 'O' || choix == 'o') {
  printf("Oui\n");
} else if (choix == 'N'
           || choix == 'n') {
    printf("Non merci\n");
} else {
    printf("Pas compris\n");
}
```

```c
switch (choix) {
  case 'O':
  case 'o':
    printf("Oui\n");
    break;
  case 'N':
  case 'n':
    printf("Non merci\n");
    break;
  default:
    printf("Pas compris\n");
    break;
}
```

Énumérations (cf. 4.2.3)

Noms associés à des entiers :

- Auto-numérotation incrémentale depuis 0 ou la dernière initialisation
- Considérés comme des constantes
- Opérations classiques possibles

```c
enum couleur {
  TREFLE, CARREAU, COEUR, PIQUE //0, 1, 2, 3
};

enum valeur {
  DEUX=2, TROIS, QUATRE, CINQ,
  SIX, SEPT, HUIT, NEUF, DIX,
  VALET, DAME, ROI, AS
};

// Une variable
enum couleur atout = PIQUE;

// Une fonction qui prend et retourne
enum valeur choisir_carte(enum couleur demandee);
```

Énumérations

Astuce pour inclure l'information des bornes automatiquement :

- Ajouter un dernier élément dans l'enum
- Ajouter éventuellement un élément qui ait la valeur du premier
- Créer une correspondance avec un tableau de chaînes de caractères

```
1  enum kind {
2    ROCK, POP, JAZZ, HIP_HOP, //0, 1, 2, 3
3    KIND_COUNT, FIRST_KIND=ROCK //4, 0
4  };
5
6  const char* txt[KIND_COUNT] = {
7    "Rock", "Pop", "Jazz", "Hip Hop"
8  };
9
10 const char* kind txt(enum kind k) {
11   if (k >= FIRST_KIND && k < KIND_COUNT)
12     return txt[k];
13   return NULL;
14 }
```

Structures (cf. 9.1)

Nouveaux types pouvant contenir différents types nommés (champs) :

- Syntaxe :
  - Définition : `struct nom_struct {type1 ch1; type2 ch2; ... };`
  - Déclaration d'une variable struct : `struct nom_struct var;`
  - Accès aux champs : `var.ch1;`
- Peuvent être retournées / copiées
- La taille (constante) doit donc être calculable par `sizeof()`
  (par ex. les tableaux doivent avoir une taille connue)

```c
struct point {
  int x;
  int y;
}; //Attention au ;

int main(int argc, char* argv[]) {
  struct point p;
  p.x = 1; p.y = 2;
  struct point p1 = {1, 2};
  struct point p2 = {.y=2, .x=1}; //depuis C99
  struct point p3 = p; //copie des champs de p
  return 0;
}
```

Structures (cf. 9.1)

Le mot-clef `typedef` permet de créer un nouveau nom pour type.

ex. : `typedef int def_type; //def_type représente le même type que int`
`def_type a = 5; //a variable de type int`

Possibilité de l'utiliser pour `struct` :

```c
struct point {
  int x;
  int y;
};
typedef struct point point_t;

int main(int argc, char* argv[]) {
  point_t p;
  p.x = 1; p.y = 2;
  point_t p1 = {1, 2};
  point_t p2 = {.y=2, .x=1}; //depuis C99
  point_t p3 = p; //copie des champs de p
  return 0;
}
```

→ Concrètement, permet de s'abstraire de l'utilisation du mot-clef `struct` lors de la déclaration de variables et du prototypage de fonctions.

Taille d'une structure (cf. 9.1.4)

- Tout comme pour un type de base, il faut toujours utiliser l'opérateur `sizeof` pour calculer la taille d'une structure (ex. : `sizeof(point_t)` ;)
- La taille d'une structure est constante et toujours supérieure ou égale à la somme des tailles des champs qui la composent.

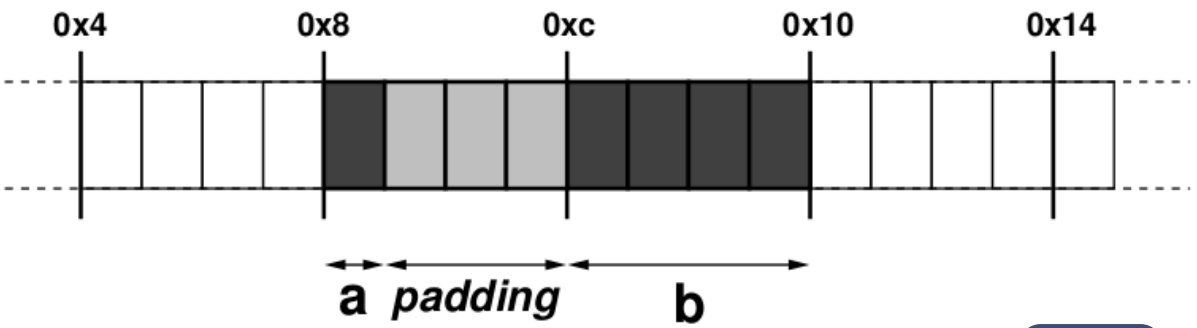
  Principe de l'alignement mémoire : une variable variable de type `type` est alignée si l'expression suivante est vraie :

  ```
  (((size_t) &variable) % sizeof(variable)) == 0
  ```

  - Donc pour les `struct`, alignement de chaque champ (et de la `struct`)
  - La taille (constante) est donc un multiple du plus grand type
  - Possibilité de laisser des octets vides (*padding*)

  Exemple :

```
struct truc {
  char a;
  int b;
};
```



Taille d'une structure (cf. 9.1.4)

Quel sera l'affichage du code suivant ?

```c
#include<stdio.h>

struct machin {
  char a;
  char b;
  int c;
};

struct bidule {
  char a;
  int c;
  char b;
};

int main(int argc, char* argv[]) {
  struct machin x;
  struct bidule y;
  printf("Taille de x : %lu\n", sizeof(x));
  printf("Taille de y : %lu\n", sizeof(y));
  return 0;
}
```

Structures et énumérations

Exemple du jeu de cartes :

```c
enum couleur {
  TREFLE, CARREAU, COEUR, PIQUE
};

enum valeur {
  DEUX=2, TROIS, QUATRE, CINQ,
  SIX, SEPT, HUIT, NEUF, DIX,
  VALET, DAME, ROI, AS
};

// Une variable
enum couleur atout = PIQUE;
```

```c
struct carte {
  enum valeur valeur;
  enum couleur couleur;
};

struct valeur black jack =
    {VALET, PIQUE};

int plus fort(struct carte c1,
              struct carte c2) {
  return c1.valeur > c2.valeur
    || c1.valeur == c2.valeur
    && c1.couleur > c2.couleur;
}
```

Unions (cf. 9.2)

Pour aller plus loin

- Comme une structure où un seul champ peut être utilisé en même temps.
- La taille (`sizeof`) est au moins égale à la taille du plus grand élément
- Souvent utilisé en conjonction avec une structure et une énumération

```c
union some_value {
  int i;
  double d;
};

union some_value v;
v.i = 10;
v.d = 20.3;

// v.i invalide maintenant
printf("d: %lf i:%d\n", v.d, v.i);
```

Structures et énumérations et unions

Pour aller plus loin

Exemple :

```c
struct p_value {
  enum {INT, DOUBLE} type;
  union {
    int i;
    double d;
  } value;
};

struct p_value v = { INT, {.i = 45} };
switch (v.type) {
  case INT:
    printf("%d\n", v.value.i);
    break;
  case DOUBLE:
    printf("%lf\n", v.value.d);
    break;
}
```

Structures et chaînes de caractères

- Penser à réserver la mémoire pour les chaînes de caractères !
  On optera généralement pour la solution `char str_tab[taille]` avec taille souvent définie par une constante de préprocesseur

```c
#include<stdio.h>

#define SIZE 10

struct test {
    int x;
    char str_tab[SIZE]; //ici on a réservé l'espace mémoire
    char* str_ptr; //ici on a juste déclaré un pointeur
};

int main(){
    struct test t; //Ne pas oublier le struct
    scanf("%s", t.str_tab); //OK, l'espace mémoire est réservé
    //scanf("%s", t.str_ptr); //!!!KO segfault, mémoire non réservée
    t.str_ptr = (char *) malloc(SIZE*sizeof(char));
      //t.str_ptr pointe vers une zone mémoire réservée de SIZE char
    scanf("%s", t.str_ptr); //OK, la mémoire est réservée
    return 0;
}
```

Exercice : Structures

- Définir une structure `point` qui contient deux champs `int x; int y;`
- Écrire la fonction `float norme(struct point p1, struct point p2);` qui calcule la distance entre deux `struct point` :

$$\|p1 - p2\|_2 = \sqrt{(p1.x - p2.x)^2 + (p1.y - p2.y)^2}$$

  On utilisera donc la fonction `sqrt` de la bibliothèque `math.h`.
  Pourquoi a-t-on l'erreur *référence indéfinie* juste avec `#include<math.h>` ?
  Que dit la page de *man sqrt* ?
- Définir un tableau de structure `point`, de taille n=5, que l'on initialisera à la main puis avec `scanf`, `fgets` ou par la ligne de commande
- Écrire une fonction qui calcule les paramètres a et b d'un ajustement affine $y = ax + b$ par rapport au nuage de points (x, y) représenté par le tableau :

$$a = \frac{Cov(x,y)}{V(x)} \quad \text{avec} \quad \bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$
$$b = \bar{y} - a\bar{x} \qquad V(x) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$
$$Cov(x,y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

Tableaux multidimensionnels

Il existe 3 différentes façons de manipuler des données multidimensionnelles :

1) Tableaux unidimensionnels ✓
- Le plus simple, le plus courant
- Tableaux de type : `type*`
- On "simule" la multidimensionnalité par un indiçage type `tab[j+X*i]`

2) Tableaux multidimensionnels ∼
- Surcouche des tableaux unidimensionnels
- Tableaux de type : par ex. `type(*)[X]`
- Permettent un indiçage de type `tab[i][j]`
- Peu facilement manipulables (la taille de la 2e dim. fait partie du type)

3) Tableaux de pointeurs ✓
- Intéressants selon les contextes (ex. argv)
- Tableaux de type : par ex. `type**`
- Plus flexibles, tailles potentiellement différentes suivant les dimensions
- Besoin de réserver (et libérer) dynamiquement l'espace mémoire

1) Tableaux unidimensionnels ✓

Syntaxe : ex. `type tab[Y*X];` `tab` est de type `type*`

On a par exemple un tableau de taille `Y*X` pour simuler une matrice 2D $Y \times X$.
Toutes les données sont donc allouées de façon contiguë en mémoire :

0	1	...	X*Y-2	X*Y-1

On peut considérer les données comme étant les lignes de la matrice mises bout à bout :

0	1	...	j	...	X-1
X	X+1		...	...	2*X-1
...			...		...
i*X			i*X+j		(i+1)*X-1
...			...		...
(Y-1)*X			...		X*Y-1

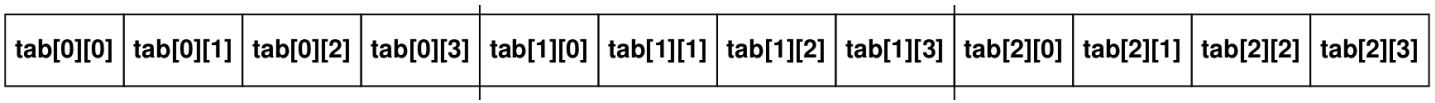
On simule donc la multidimensionnalité par un indiçage à unique coordonnée :
L'information à la position `(i,j)` correspond à l'indice `i*X+j`.

→ Annexe : Parcours des tableaux multidimensionnels

2) Tableaux multidimensionnels ~

Syntaxe : ex. `type tab[Y][X];` `tab` est de type `type(*)[X]`

Ex. : `int tab[3][4];`



Comme pour un un tableau unidimensionnel, toutes les données sont allouées de façon contiguë en mémoire. On a donc une équivalence :

$$\texttt{tab[i][j]} \equiv \texttt{tab[i*X+j]}$$

On pourrait ainsi convertir un pointeur et passer d'une lecture par unique coordonnée à une lecture simplifiée en plusieurs dimensions :

Ex. :
```
int tab_uni[12]; //tableau de 12 int
int (*tab_multi)[4] = tab_uni; //même tableau vu en 3x4
tab_multi[2][1] = 1; //équiv. à tab_uni[2*4+1]
```

⚠ **Peu pratiques à manipuler !** Car `tab` de type `type(*)[X]` et plus `type*`. Pour utiliser les accès `[i][j]`, on doit connaître la taille de la 2e dimension. On ne peut donc pas donner des tableaux de tailles différentes aux fonctions.

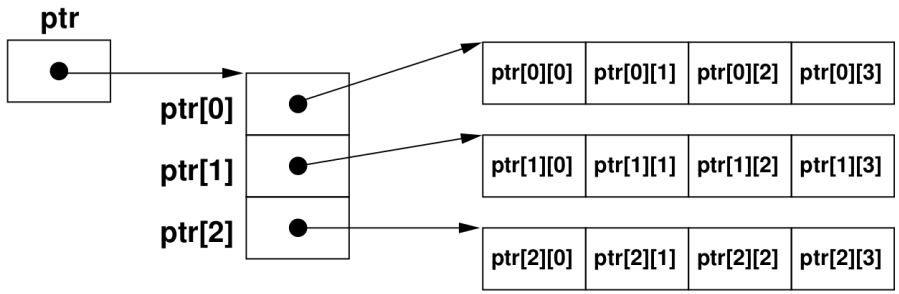
Ex. : `void display_tab_2D(int (*tab)[4], int Y);`

3) Tableaux de pointeurs ✓ (cf. 8.6)

Dans le cas de tableaux de pointeurs (pointeurs de pointeurs), les différentes zones mémoires allouées ne sont pas forcément contiguës.

Par exemple (`Y=3, X=4`) :

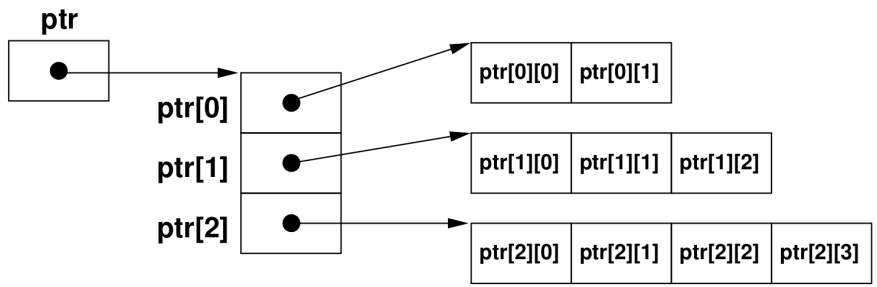
```
int** ptr = NULL;
ptr = malloc(3*sizeof(int*));
for (int i = 0 ; i < 3 ; i++)
  ptr[i] = malloc(4*sizeof(int));
```



Ce qui permet même d'avoir des tableaux de tailles différentes sur les dimensions suivant la première.
Comme pour l'argument `char* argv[]` du `main` qui est un tableau de pointeurs vers des chaînes de caractères de tailles potentiellement différentes.

Par exemple (`Y=3`) :

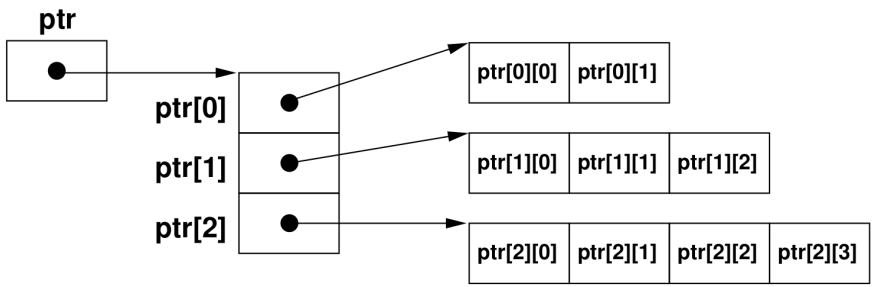
```
int** ptr = NULL;
ptr = malloc(3*sizeof(int*));
ptr[0] = malloc(2*sizeof(int));
ptr[1] = malloc(3*sizeof(int));
ptr[2] = malloc(4*sizeof(int));
```



3) Tableaux de pointeurs ✓ (cf. 8.6)

Pour la libération de la mémoire, ne pas oublier de d'abord libérer tous les sous-tableaux.

```
int** ptr = NULL;
ptr = malloc(3*sizeof(int*));
ptr[0] = malloc(2*sizeof(int));
ptr[1] = malloc(3*sizeof(int));
ptr[2] = malloc(4*sizeof(int));
```




```
int* ptr1 = ptr[1]; //copie de ptr[1]
ptr1[0] = 3; //équivalent à ptr[1][0] = 3;
printf("%d\n", ptr1[0]); //<- 3

// Libération des sous-tableaux
free(ptr[0]); // ptr[0][0], ptr[0][1], ...
free(ptr[1]);
free(ptr[2]);
printf("%d, %d\n", ptr[1][0], ptr1[0]); //<- valeurs non contrôlées
printf("%d\n", ptr[1]==ptr1); // <- 1, Le tableau ptr existe toujours

free(ptr); // Libération du tableau ptr (ptr[0], ptr[1], ...)
printf("%d\n", ptr[1]==ptr1); // <- 0, on ne peut plus lire dans ptr
```

Exercice : Calcul matriciel

Écrire un ensemble de fonctions permettant d'effectuer du calcul matriciel.

- D'abord définir la structure `matrix` qui contient les champs suivants :
  - un entier `lines` pour le nombre de lignes
  - un entier `cols` pour le nombre de colonnes
  - un pointeur vers double qui identifie à quel endroit sont mémorisées les données de la matrice
- Écrire les fonctions suivantes (dans l'ordre, ce sera mieux pour les tester) :
  - `new_matrix` : reçoit deux entiers (nombre de lignes et de colonnes). Cette fonction crée et retourne une matrice ne contenant que des zéros.
  - `get_matrix_value` : reçoit une matrice et deux entiers `col` et `line`. Cette fonction renvoie la valeur (`col`, `line`) de la matrice.
  - `disp_matrix` : affiche le contenu d'une matrice
  - `set_matrix_value` : reçoit une matrice, deux entiers et un double. Cette fonction place le double aux coordonnées définies par les deux entiers.

  À partir de maintenant, on peut tester les fonctions avec des matrices personnalisées construites en début de programme.
  - `add_matrix` : reçoit deux matrices et retourne la somme des deux matrices. Produit une erreur si les dimensions des matrices ne sont pas compatibles
  - `mult_matrix` : reçoit deux matrices et retourne le produit des deux matrices. Produit une erreur si les dimensions des matrices ne sont pas compatibles

Pointeurs et structures (cf. 9.1.5)

Les structures étant des types presque comme les autres, on va être amené à manipuler des pointeurs vers des structures.

- Priorité du `.` supérieure à celle de `*` : `*x.y` $\equiv$ `*(x.y)`
- Syntaxe dédiée pour accéder au champ `y` d'une `struct` pointée par `x` : `(*x).y` $\equiv$ `x->y`

Ex. :

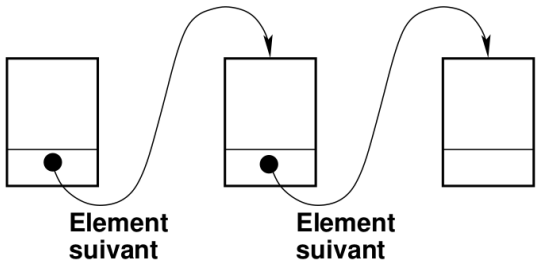
```
struct test {
  int x;
  double* ptr;
};
int main(){
  struct test t;
  struct test* test_ptr = &t;
  (*test_ptr).x = 2;
  test_ptr->x = 2; //équivalents
}
```

Exercice : Transpose matrix

En s'inspirant de l'exercice swap, modifier le code de l'exercice calcul matriciel pour ajouter une fonction qui transpose une matrice.

Listes chaînées (cf. 9.1.5)

- Permet de faire des structures chaînées
- Liste d'éléments de taille variable
- Ordre des éléments facilement modulable




```c
#include<stdio.h>

struct i_list {
  int value;
  struct i_list *next;
};

void append(struct i_list *node,
            struct i_list *other) {
  node->next = other;
}

int main(int argc, char* argv[]) {
  struct i_list node1;
  struct i_list node2;
  node2.value = 5;
  append(&node1, &node2);
  printf("%d\n", node1.next->value); //5
  return 0;
}
```

Exercice : Liste chaînée

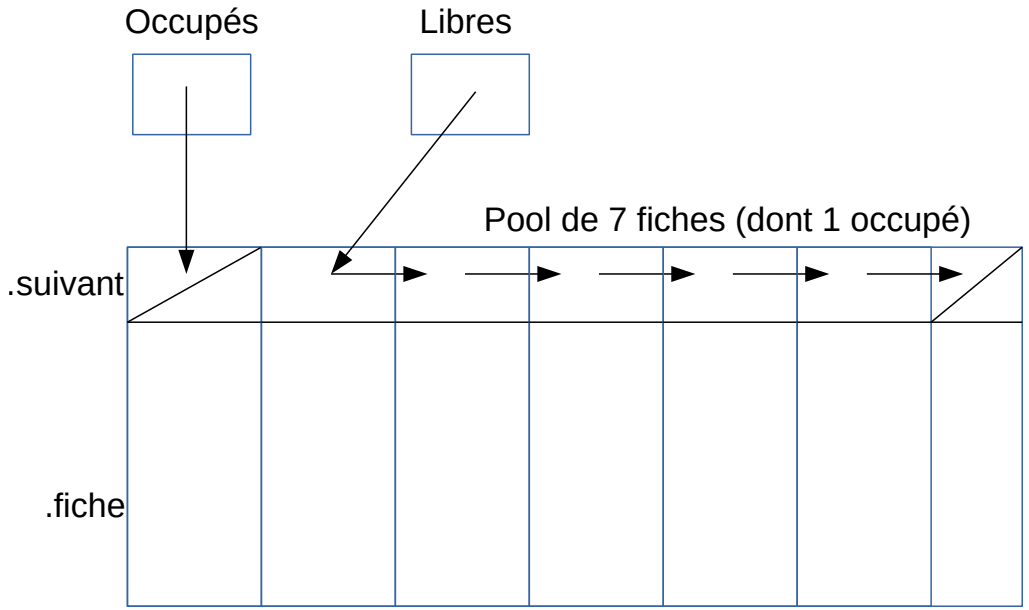
L'objectif de cet exercice est de créer un type "carnet d'adresses" et de programmer un certain nombre de fonctions permettant de le manipuler. Il ne faut pas chercher à réaliser cet exercice parfaitement du premier coup (il sera repris de façon incrémentale). Penser cependant à rajouter des commentaires aux points d'extensions. Dans un premier temps, par exemple, on ne gérera pas les multiples cas d'erreur pouvant survenir : erreurs de saisie (chaîne vide, le numéro de téléphone n'est pas un numéro, le sexe est autre que 'H' ou 'F', ...), chaînes de caractères plus longues que le maximum prévu, ajout d'un contact figurant déjà dans le carnet d'adresse, noms identiques à la casse près, ...

- Écrire une structure contact comprenant les champs *numéro d'identification, nom, prénom, adresse, sexe* et *téléphone*. Existe-t-il une information qui peut être facilement représentée par une énumération ? Écrire les fonctions `contact_afficher` et `contact_saisir` permettant respectivement d'afficher et de saisir les informations sur un contact.

  On pourra simuler des frappes au clavier à l'aide d'un fichier texte : `./a.out < contact.txt`.

Exercice : Liste chaînée

Nous allons créer une structure `carnet` permettant de gérer un ensemble de contacts. On définira un carnet d'adresse comme contenant un tableau de contact (*pool*) de taille `MAX_CONTACTS`, une liste simplement chaînée des contacts utilisables (*available*), c'est à dire, chaque élément est lié au suivant ainsi qu'une seconde liste simplement chaînée contacts utilisés (*used*). Afin de réaliser la chaîne nous rajouterons un champ suivant (*next*) aux contacts. Créer cette structure.



Réf. exercice

Exercice : Liste chaînée

- Initialisation de tous les champs du carnet d'adresses (pointeurs, listes, etc).
- Ajout de contacts au carnet.
- Affichage de l'ensemble des contacts commençant par la chaîne de caractères passée en paramètre.
- Suppression d'un contact (par son numéro d'identification) d'un carnet.
- Modifier la fonction d'ajout d'un contact afin qu'elle insère le contact "à sa place" dans l'ordre alphabétique.
- Modifier la fonction de recherche d'un contact pour qu'elle arrête la recherche plus tôt en cas de recherche infructueuse en exploitant le fait que la liste de contacts est triée.
- Écrire une fonction qui vérifie qu'un carnet d'adresses est trié.
- On aimerait maintenant ajouter une recherche par la ville et/ou le code postal, que faut-il modifier ?
- Bonus : Modifier l'ensemble pour ajouter de nouvelles entrées dans le pool au besoin. Que faut-il modifier au code présent pour imaginer un jour libérer les entrées inutiles ?

Structures et pointeurs n°2

Résumé :

- Interaction utilisateur durant le programme : ex. `scanf`

```
int a; //a de type int, &a de type int*
char s[50]; //s de type char*
scanf("%d", &a); //On écrit dans a un entier depuis le terminal
scanf("%s", s);
```

- Tableaux multidimensionnels :
  - Tableau 1D : `type tab[Y*X];` `tab` est de type `type*`
    Multidimensionnalité par un indiçage du style `tab[j+X*i]`

- Structures : `struct`

```
struct nom_tmp {
  type_1 champ_1;
  type_2 champ_2;
  ...}; //ne pas oublier le ;
struct nom_tmp x; //variable x de type struct nom_tmp
```

- D'autres mots-clefs : `static, switch, enum, union, ...`

Structures et pointeurs n°2

Résumé :

- Alias sur un type : `typedef type alias`

```
typedef struct nom_tmp nom; //nom ≡ struct nom_tmp
nom y; //variable y de type struct nom_tmp
```

- Liste chaînée : Liste d'éléments de taille variable

ex. :
```
struct lst_ch {
  int x;
  struct lst_ch* n; //ptr vers l'élément suivant
};
struct lst_ch l1, l2; l1.x = 2; l2.x = 10;
l1.n = &l2; //l1.n pointe vers l2
l2.n = NULL;
struct lst_ch* l_ptr = &l1; //ptr début de liste
while(l_ptr != NULL) {
  printf("%d\n", l_ptr->x); //x->y ≡ (*x).y
  l_ptr = l_ptr->n; //On passe à l'élément suivant
}
```

Instructions de saut (cf. 5.3)

- Pour les boucles d'itération (`for`, `while`) : (dispensable)
  - `continue;` (passe à l'itération suivante dans le bloc d'instructions)
  - `break;` (termine l'instruction d'itération ou le `switch`)

Exemples :

```c
/*Cherche le premier indice d'une variable
val dans un tableau tab de taille len*/
void find_tab(int tab[], int len, int val) {
  int i;
  for(i=0; i<len; i++) {
    if (tab[i]==val)
      break;
  }
  //Display
  if (i<len)
    printf("Val %d à l'indice %d\n", val, i)
        ;
  else
    printf("Val %d pas retrouvée\n", val);
  return 0;
}
```

```c
/*Fais quelque chose*/
void process_tab(int tab[], int len) {

  for(i=0; i<len; i++) {

  //Ne traite pas les tab[i]==0
  //Sans utiliser de else
  if (!tab[i])
    continue;

  //Instru
  //...
}
```

Instructions de saut (cf. 5.3)

- Saut à une étiquette :

  Une étiquette (*label*) peut être placée devant des instructions pour permettre d'y accéder directement dans le flot d'exécution avec `goto`.
  Mécanisme utilisé par le `switch` avec comme étiquettes `case` et `default`.
  Permet de sortir de plus de niveaux d'imbrication que le `break`, par exemple pour une double boucle.

  Exemple :

```c
/*Fonction qui vérifie que deux tableaux
n'ont aucune valeur en commun*/
void comp_tab(int tab1[], int len1, int tab2[], int
    len2) {
  for(int i=0; i<len1; i++) {
    for(int j=0; j<len2; j++) {
      if (tab1[i]==tab2[j])
        goto end;
    }
  }
  return;
end:
  printf("Test non réussi\n");
}
```

Gestion des fichiers

- Type associé : `FILE*`
- Protocole :
  1. Ouverture : `FILE* fopen(char* nom, char* mode);`
  2. Instructions, travail sur le fichier
  3. Fermeture : `int fclose(FILE* fd);`
- Modes d'ouvertures :

`mode`	**Mode d'ouverture**	**Position**
`"r"`	Lecture	Début
`"w"`	Écriture	Début
`"a"`	Écriture (toujours à la fin)	Fin
`"r+"`	Lecture/écriture. Pas de création	Début
`"w+"`	Lecture/écriture. Le fichier peut être créé	Début

Fichiers textes

- Fichiers déjà ouverts :
  - `stdin` : Entrée standard en lecture. Par défaut le clavier
  - `stdout` : Sortie standard en écriture. Par défaut l'écran
  - `stderr` : Sortie d'erreur en écriture. Par défaut l'écran
- Fonctions de fichiers textes :

  Considèrent le contenu du fichier commes des chaînes de caractères.
  - `int fprintf(FILE* fd, char* format, ...);`
  - `int fscanf(FILE* fd, char* format, ...);`
  - `char* fgets(char* buffer, int size, FILE* fd);` lit une ligne
  - `int fgetc(FILE* fd);` lit un char
  - `int fputc(int c, FILE* fd);` écrit un char.

  `printf("foo");` $\equiv$ `fprintf(stdout, "foo");`
- Rappels redirection :

```
./prog < input.txt > output.txt
```

Manipulations génériques

- Lecture / Écriture :
  - `size_t fread(void* ptr, size_t size, size_t nitems, FILE* stream);`
  - `size_t fwrite(const void* ptr, size_t size, size_t nitems, FILE* stream);`

```c
#define BSIZE 100

void copy_file(FILE* dst, FILE* src) {
  size_t r;
  char buffer[BSIZE];
  while((r = fread(buffer, 1, BSIZE, src)) > 0)
      {
    if(fwrite(buffer, 1, r, dst) <= 0)
      break;
  }
}
```

- Position / déplacement :
  - `long ftell(FILE* fd);`
  - `void rewind(FILE* fd);`
  - `int fseek(FILE* fd, long offset, int whence);`

Gestion des erreurs

- Pas de mécanisme dédié
- Variable globale : `int errno;` (`errno.h`)

  contient le dernier code d'erreur
- `void perror(char* prefix);`

  affiche le préfixe suivi du dernier message d'erreur

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

int main(int argc, char* argv[]) {
  FILE* fd;
  if ((fd = fopen("file.txt", "r")) == NULL)
    goto error;
  //Instru ...
  return 0;
error:
  perror("Error opening file");
  printf("Error code : %d\n", errno);
  return errno;
}
```

Programmation multi-fichiers

Projets plus conséquents → multitude de fichiers `.c` ... Comment les organiser ?

Contraintes :

- Une seule fonction `main`
- Prototype des fonctions accessibles → (fichiers d'en-têtes)
- Visibilité des variables globales → (`extern`, `static`)

Fichiers d'en-têtes

Assurent la visibilité de toutes les variables/fonctions dans les autres fichiers

- Par convention se terminent par `.h` (header)
- Contiennent des déclarations :
  - Prototypes de fonctions définies dans le `.c` (source) correspondant
  - Constantes/Macros
  - Types, structures, etc.
- Inclus par `#include` suivi de :
  - `"./fichier.h"` s'il s'agit d'un fichier perso (répertoire courant)
  - `<fichier.h>` s'il s'agit d'un fichier standard (include path)

Visibilité/portée des variables (cf. 6.2.2)

- Variables locales valables uniquement dans la fonction
- Variables globales, valables tout le programme (attention !)
- Contraintes explicites :
  - `static` sur une variable globale
    La variable n'est plus visible en dehors du fichier
  - `static` sur une variable locale
    La variable garde sa valeur entre les appels
    (par ex. : compteur d'appel à une fonction)
  - `extern` sur une variable globale/fonction
    Il existe une variable/fonction, sa définition est ailleurs
    (par ex. : variable globale dont la valeur est assignée dans le main)

Compilation séparée / multi-fichiers

- `#ifndef`, `#endif` pour éviter les dépendances circulaires
- `#include "name.h"` pour chemin relatif (`<name.h>` librairies standards)

calcul.h

```c
#ifndef CALCUL_H
#define CALCUL_H

#define TAILLE_MAX 42
struct bar;
extern int var;

void foo(); //Fctn. ext.
void baz(struct bar* x);
// ... autres prototypes
#endif
```

calcul.c

```c
#include "calcul.h"

void foo() {
  printf("%d\n", var);
}
```

main.c

```c
#include "calcul.h"

int var = 42;

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

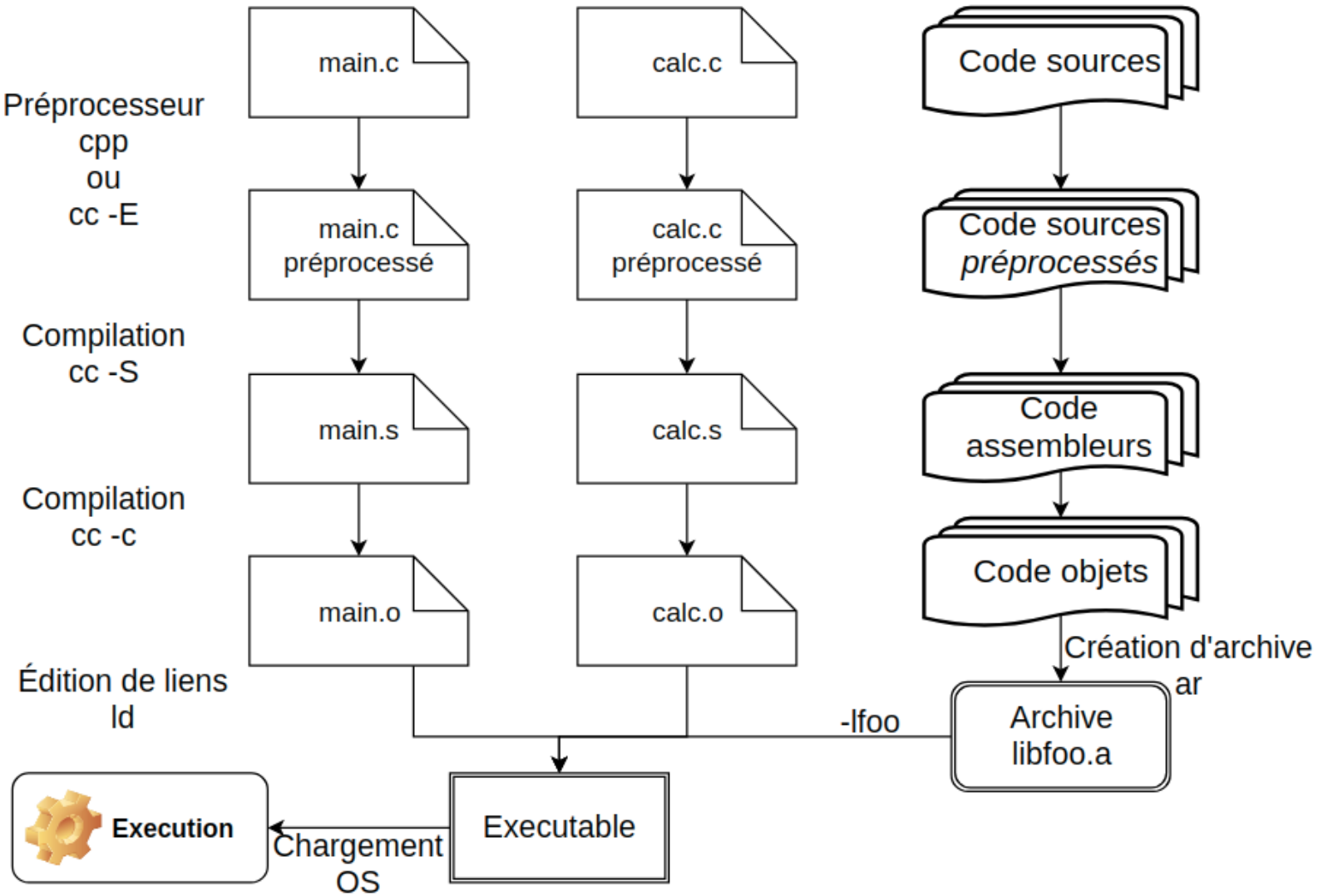
- Compilation séparée :

```
gcc -c main.c                          (génère main.o)
gcc -c calcul.c                        (génère calcul.o)
gcc -o executable main.o calcul.o      (génère l'exécutable)
```

- Compilation directe :

```
gcc -o executable main.c calcul.c      (génère l'exécutable)
```

Compilation séparée / multi-fichiers



Compilation séparée / multi-fichiers

- Solution 1 (manuelle) :
  - ✗ Recompiler toutes les sources à chaque fois, peut être très long !

```
gcc -o exe main.c calcul.c ...
```

  - ✗ On peut aussi ne recompiler que les sources modifiées pour regénérer l'exécutable mais cela nécessite d'être très vigilant

```
gcc -c calcul.c
```

```
gcc -o exe main.o calcul.o
```

- Solution 2 (avec **Make**) : outil d'automatisation de compilation d'un projet
  Devient vite nécessaire dès lors qu'on a plusieurs fichiers sources.

  Principe :
  - Ne compiler automatiquement que les morceaux de codes mis à jour en se basant les dates de création des fichiers
  - Fichier texte dans le dossier du projet : `Make` ou `Makefile`
  - Exécution avec la commande : `make [commande]`

Makefile

Ensemble de règles sous la forme : *produit:source(s)*
commande(s)

Exemple minimal de fichier Makefile dans le cas précédent (calcul.c, main.c) :

```
exe: main.o calcul.o
  gcc -o exe main.o calcul.o #tabulation en début de ligne
  ./exe
main.o: main.c
  gcc -c main.c #tabulation en début de ligne
calcul.o: calcul.c calcul.h
  gcc -c calcul.c #tabulation en début de ligne
```

`make` : Par défaut première règle considérée

On mettra donc généralement la création de l'exécutable en premier

On regarde les sources de la première règle : `main.o` et `calcul.o`.
D'abord `main.o`. La règle 2 indique que `main.o` dépend de `main.c`.
On ne regénère donc `main.o` que si `main.c` a été modifié après la date de création de `main.o` (ou si `main.o` n'existe pas).
Même cheminement pour `calcul.o`.

Si au moins l'une des sources a été mise à jour, on regénère l'exécutable.

Makefile

Pour aller plus loin dans l'automatisation :

`$()` = variable, `$@` = produit, `$^` = sources, `$<` = première source

Exemple de Makefile générique : `make`, `make clean`, `make zip`

```
CC = gcc
CFLAGS = -Wall

.PHONY: clean zip #indique qu'il ne s'agit pas de fichiers

sources= main.c calcul.c #liste de tous les fichiers sources
objets= $(sources:.c=.o) #déduction des noms des binaires

exe: $(objets) #prend tous les binaires
  $(CC) $(CFLAGS) -o $@ $^ # $@=produit(exe), $^=sources

%.o: %.c #règle générale pour chaque .c. $< = première source
  $(CC) $(CFLAGS) -o $@ -c $<

clean: # pour supprimer tous les fichiers binaires
  rm -f exe *.o

zip: # pour archiver le projet
  tar -zcvf prog.tar.gz *.c Makefile
```

Bogues ?

Quand on code, on fait quasiment toujours des erreurs.

Deux types d'erreur :

- Statiques : détectées à la compilation (facilement corrigeables)
- Dynamiques : erreur sémantique, d'accès mémoire → arrêt de l'exécution (plus difficiles à déterminer...)

Parmi les messages d'erreurs classiques :

- *Segmentation fault* : Accès illicite à une zone mémoire
  - Indice incorrect de tableau (ex. : parcours de 0 à len)
  - Pointeur vers une zone non souhaitée (ex. : NULL, chaîne constante)

- *Core dumped*
  Le système va sauvegarder l'état du programme en cours d'exécution dans un fichier appelé *core* (stocké quelque part selon le système)

Débogage par affichage

Première solution : à coups de `printf` à des endroits clés du code

Avantages :

- Localiser facilement la (première) portion de code buguée
- Permet d'afficher les valeurs de variables
- Écriture sur la sortie standard ou dans un fichier de log
- Utiles tout au long de l'écriture du programme

Inconvénients :

- Long à écrire si code très long
- Aller-retours pour afficher de nouvelles variables

→ Méthode de débogage pas à pas, voire automatique : *GDB*

Débogage avec GDB

Deuxième solution : en utilisant l'outil `gdb` (GNU Debugger) qui :

- Permet de fixer des *breakpoints*, d'exécuter le code pas à pas
- Permet d'afficher les valeurs des variables au cours de l'exécution
- Affiche souvent la ligne causant une erreur de segmentation

Exemple :

`gcc -g code.c`	Compilation moins optimisée, pour permettre le débogage
`gdb a.out`	Exécution avec gdb
`break 10`	Crée un breakpoint à la ligne 10
`run`	Lance le programme - qui s'arrête à la ligne 10
`p x`	Affiche la variable x si elle existe avant la ligne 10
`n`	(next) Exécute la ligne suivante
`help`	Pour voir toutes les options existantes
`c`	(continue) Continue jusqu'au breakpoint suivant
`clear`	Efface tous les breakpoints
`run`	Relance le programme entièrement
`quit`	Sort de gdb et retourne au terminal

Limite : Diagnostic difficile des erreurs d'accès mémoire → *Valgrind*

Débogage avec Valgrind ?

Troisième solution : *Valgrind*, outil d'analyse d'accès mémoire

- Permet de détecter les erreurs d'exécution et de profiler le code
- Efficace pour les accès mémoire illicites et les fuites (ex. : oubli de `free`)
- Basé sur un émulateur de langage machine, qui interprète pas à pas et vérifie chaque instruction du programme binaire
- Fonctions disponibles :
  - *memcheck* : vérification mémoire poussée
  - *addrcheck* : version allégée de memcheck, sans le test d'accès aux zones mémoires non initialisées
  - *cachegrind* : simulateur de comportement de la hiérarchie de caches pour l'analyse poussée de performances
  - *massif* : analyse de l'occupation du tas
  - ...

Compilation : `gcc -g code.c`

Exécution : `valgrind --leak-check=full ./a.out`

FIN

Rappels :

- Supports (slides/cours/codes) :
  http://remi-giraud.enseirb-matmeca.fr/teaching/
- Plateforme d'exercices en ligne pour s'entraîner :
  https://thor.enseirb-matmeca.fr:4443/
- Question sur une fonction ? → page de man (ex : `man 3 printf`)

Place aux TP !

remi.giraud@enseirb-matmeca.fr

Pour aller plus loin

- Jouer avec les types :
  - Qualificateurs de type (cf. 4.1.4)
  - Sélection générique (par le type) (cf. 4.1.5)
- Jouer avec les fonctions :
  - Fonctions à nombre variable d'arguments (cf. 6.4.3)
  - Inlining de fonctions (cf. 6.5.2)
- Tableaux multidimensionnels :
  - Disposition en mémoire (cf. 7.4)
  - Pointeurs de pointeurs vs. tableaux multidimensionnels (cf. 8.6)
- Allocation dynamique ++ :
  - Allocation dynamique sur la pile (cf. 8.5.3)
- Préprocesseur :
  - Définition dynamique de constante (cf. 10.1)
  - Macros ++ (cf. 10.2)

Qualificateurs de type (cf. 4.1.4)

Les qualificateurs de type indiquent au compilateur des optimisations (ou non) :

- `const` : une variable qualifiée avec ce mot-clef peut être initialisée au moment de sa déclaration, mais n'est plus modifiable par la suite. Ce qualificateur est souvent employé pour des arguments de fonction afin d'en expliciter un peu plus le rôle.

- `restrict` : utilisable que pour des pointeurs et indique que le pointeur qualifié ne possède pas d'alias, c'est-à-dire qu'il n'existe pas un autre pointeur ayant pour valeur la même adresse.

  ⚠ Ne garantit pas l'absence d'aliasing. Le système et le compilateur ne le vérifient pas.

- `volatile` : indique que la variable est susceptible d'être modifiée par un évènement extérieur au programme et doit être lue en mémoire à chaque accès par le programme. L'utilisation de ce mot-clef permet de désactiver les optimisations (parfois agressives) du compilateur.

Sélection générique (par le type) (cf. 4.1.5)

Permet d'effectuer une sélection d'expression sur la base du type d'une autre expression passée en argument à l'opérateur (depuis C11).
Déterminer le type d'une variable offre des possibilités intéressantes pour produire un code plus générique et gagner en abstraction.
Le nouveau mot-clef introduit est `_Generic` :

```
_Generic(expression-d-affectation, liste-d-associations)
```

La liste des associations génériques prend la forme suivante :

```
identificateur-type 1 : expression-d-affectation,
identificateur-type 2 : expression-d-affectation,
default optionnelle : expression-d-affectation
```

- Tous les identificateurs (noms) de types doivent être distincts et correspondre à des types de base, des types construits par l'utilisateur avec des structures (cf. Chapitre 9) ou des pointeurs sur ces types.
- Une expression d'affection peut être un nom de variable ou de fonction.
- Une seule clause `default` possible, mais pas obligatoire. Dans ce cas, le type de l'expression d'affectation prise en argument par la sélection doit avoir un type compatible avec un type dans la liste d'association.
- L'expression prise en argument n'est pas évaluée, seule l'expression sélectionnée sur la base du type l'est.

Sélection générique (par le type) (cf. 4.1.5)

Exemple permettant d'afficher une chaîne de caractères correspondant au nom et au type de différentes variables :

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

#define typename(X) Generic((X), \
  char : #X "is char\n" , \
  int : #X "is int\n" , \
  unsigned int : #X "is uint\n" , \
  char* : #X "is char ptr\n" , \
  int* : #X "is int ptr\n" , \
  unsigned int* : #X "is uint ptr\n" , \
  void* : #X "is generic ptr\n" , \
  default : #X "is unknown\n" )
```

```c
int main() {
  int truc = 33;
  int * bidule = &truc;
  void * machin = NULL;
  float riri = 1.0;
  char fifi = 'a';
  float * loulou = &riri;
  printf(typename(truc));
  printf(typename(bidule));
  printf(typename(machin));
  printf(typename(riri));
  printf(typename(fifi));
  printf(typename(loulou));
  return 0;
}
```

Type booléen

En C89, le type booléen n'existe pas.
On peut toutefois simuler une variable booléenne avec un `enum` :

```
enum booleen { FAUX, VRAI }; //avec FAUX == 0 et VRAI == 1
```

En revanche une variable de type `enum boolean` est un entier et peut prendre des valeurs différentes de celles de l'enum :

```
enum boolean x = FAUX; //x == 0
enum boolean y = 2; //Association licite, y==2
```

Depuis C99, un type booléen existe : `_Bool`
et une macro `bool` vers ce type, contenu dans `#include<stdbool.h>` :

```
_Bool x = 0;
```

ou

```
#include<stdbool.h>
bool x = 0;
```

Les variables de type `_Bool` ne peuvent prendre que les valeurs 0 et 1 :

```
_Bool y = 2; //y == 1
```

En revanche, elles sont tout de même écrites sur un octet :

```
int s = sizeof(x); //s == 1 (octet)
```

Fonctions à nombre variable d'arguments (cf. 6.4.3)

Il est possible d'écrire des fonctions avec un nombre variable d'arguments avec l'utilisation des macros `va_start`, `va_arg` et `va_end`. `register` Il faut alors indiquer les arguments obligatoires de la fonction puis qu'il y aura une liste avec trois points (...). Par exemple : `int func(int arg, ...);`

Ces macros possèdent les prototypes suivants :

```
void va_start(va_list ap, last);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

- `va_start` doit être appelée en premier et initialise la liste des arguments `ap` qui va être utilisée par les deux autres macros. `last` est le dernier argument de la fonction avant la liste des argument à traiter. C'est le dernier argument dont on connaît a priori le type. Par exemple, `printf` est une fonction à nombre variable d'arguments, mais elle prend au moins un argument qui est une chaîne de caractères puis éventuellement une liste formée par le reste des arguments : `printf(char* str,...)`. Dans ce cas, le dernier argument avant la liste est la chaîne `str` dont le type est connu.
- La macro `va_arg` traite la liste des arguments : chaque nouvel appel retourne une expression dont le type est la valeur qui correspond au prochain argument de la liste. `type` est donc un identificateur (nom) de type.
- La macro `va_end` doit être appelée pour chaque occurrence de `va_start` dans la fonction.

Fonctions à nombre variable d'arguments (cf. 6.4.3)

Exemple d'utilisation :

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

int sum(int num_args, ...) {
  int val = 0;
  va_list ap;

  va_start(ap, num_args);
  for(int i=0; i<num_args; i++) {
    val += va_arg(ap, int);
  }
  va_end(ap);
  return val;
}
```

Inlining de fonctions (cf. 6.5.2)

Il est possible de préfixer le prototype d'un fonction avec le mot-clef `inline` ce qui donne au compilateur des indications d'optimisation pour cette fonction. Ce dernier va essayer de remplacer dans le code un appel de fonction par le corps de la fonction, ce qui évite l'appel justement (et est donc plus rapide).
Le mot-clef `inline` a pour les fonctions un peu le même effet que le mot-clef `register` pour les variables.
De plus, il faut préciser une classe de stockage `static` ou `extern` afin de pouvoir faire de l'inlining.
Le comportement des fonctions `static inline` est le même quel que soit le dialecte de C utilisé, ce qui n'est pas le cas des fonctions `extern inline`.
Il est cependant peu probable que vous ayez à utiliser des fonctions `inline` dans vos codes et si cela devait arriver, elles seraient très probablement de classe `static`.

 Une variable locale de classe `register` est similaire à une variable locale automatique, sauf que ce mot-clef indique qu'elle sera fréquemment accédée. Le compilateur va donc essayer de stocker cette variable dans un registre du processeur afin d'en accélérer les accès. Le mot-clef `register` est utilisable uniquement avec des variables automatiques et des arguments de fonctions.

Disposition en mémoire (cf. 7.4)

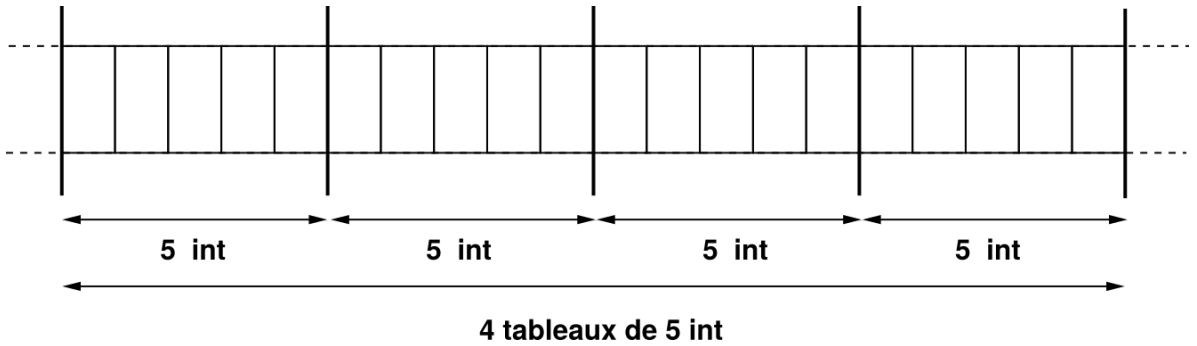
→ Retour aux tableaux

- Optimisation du parcours en mémoire :
  Modèle mémoire de type *Row-Major* : toujours la dimension la plus à droite qui varie le plus vite dans le cas d'un parcours du tableau.

  Le parcours le plus efficace en accès mémoire est donc celui où la boucle sur la dimension `Y` est imbriquée dans celle sur `X` :

```
for(int i = 0 ; i < X ; i++)
  for(int j = 0 ; j < Y ; j++)
    tab[i][j] = 0; //ou tab[i*X+j]
```

  Ex. :
```
int tab[4][5];
int tab[4*5]; // avec représentation ligne par ligne
```



- Équivalence notationnelle :

$$\texttt{tab[i][j]} \equiv \texttt{tab[i*Y + j]}$$

Allocation dynamique sur la pile (cf. 8.5.3)

Il est également possible d'allouer des données dynamiquement sur la pile.
Ce n'est pas forcément recommandé, mais il faut savoir que cela existe. C'est d'ailleurs ce qui se passe en déclarant une taille de tableau dynamiquement.

Par exemple :

```
int n;
/* choses dans le programme */
n = /* calcul de n */
/* autres choses dans le programme */
int tab[n];
/* le programme continue */
```

Cette allocation n'est pas une allocation sur le tas, mais sur la pile.
C'est l'équivalent d'un appel à la fonction `alloca`.
Il n'est pas nécessaire de libérer la mémoire de la pile car c'est automatique. En particulier, il ne faut pas appeler `free` sur des adresses de pile.

Définition dynamique de constante (cf. 10.1)

Il est possible de définir les constantes dans les fichiers sources à l'aide de `#define`, mais cela impose des modifications de fichiers.

L'option `-D` du compilateur qui permet de définir une variable dynamiquement à la compilation :

```
gcc -DDEBUG -o toto toto.c
```

Vous remarquerez l'absence d'espace entre le -D et le nom de la constante.
Également, la constante est uniquement définie mais sans valeur particulière.
Il est possible de définir une valeur à la compilation :

```
gcc -DDEBUG=3 -o toto toto.c
```

Il est alors possible de tester la valeur de cette constante avec la directive `#if` :

```
#if DEBUG == 1
  printf("Debug1 : un message de controle\n");
#elif DEBUG == 2
  printf("Debug2 : un autre message de controle\n");
#endif
```

La directive `#elif` remplace `#else if`.

Macros ++ (cf. 10.2)

→ Retour aux macros

- Remplacer un paramètre par une chaîne de caractères :

  Un nom de paramètre de macro peut être remplacé par une chaîne de caractères correspondante si ce paramètre est précédé du caractère `#` :

  ```
  #define MY_MACRO(x)   printf(#x "= %d\n", x)
  ```

  Si le code du programme “appelle” la macro avec pour paramètre l'expression `y++`, alors cette macro sera remplacée dans le code par :

  ```
  printf("y++""= %d\n", y++)
  ```

  `printf` concatène automatiquement des chaînes de caractères donc :

  ```
  printf("y++ = %d\n", y++)
  ```

Macros ++ (cf. 10.2)

- La directive `##` : Permet de concaténer des paramètres de macros :

  ```
  #define COLLAGE(x, y) x ## y
  ```

  Si le code appelle la macro comme ceci :

  ```
  int truc = COLLAGE(toto, 3);
  ```

  alors le préprocesseur remplace le tout par :

  ```
  int truc = toto3;
  ```

  En supposant qu'une variable `toto3` a été déclarée et est visible ici, il est possible de générer des noms de fonctions suivant certains types de données par exemple.