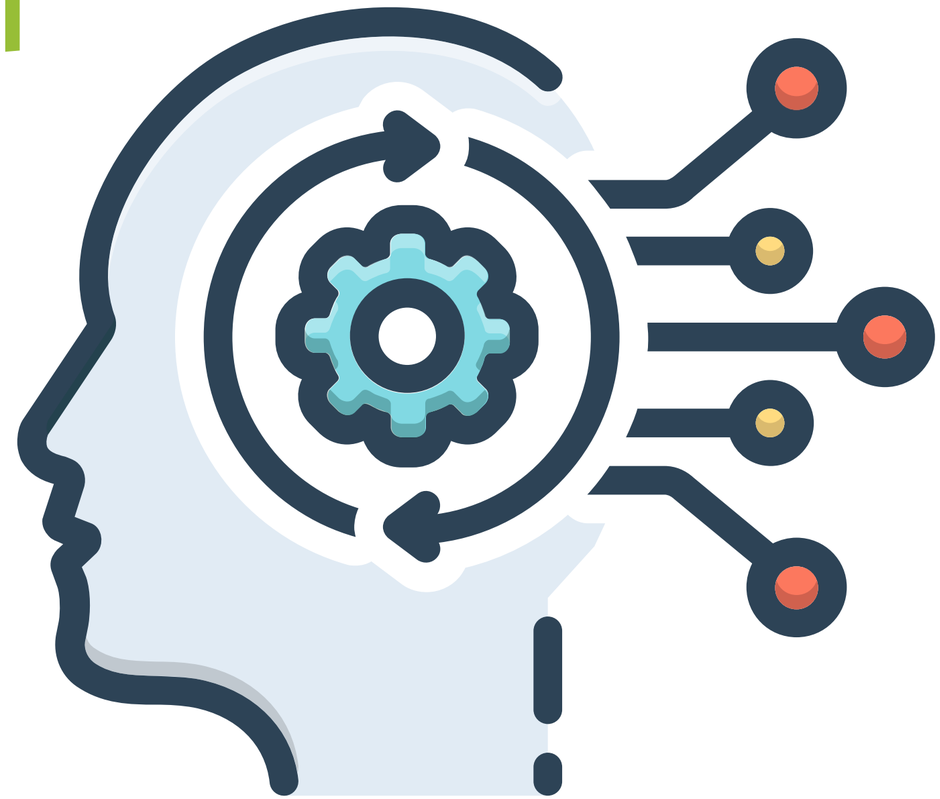
DR B. DIOP
babacar.diop@ussein.edu.sn

Algorithmes et Programmation II

Pointeurs



INFO – AGROTIC

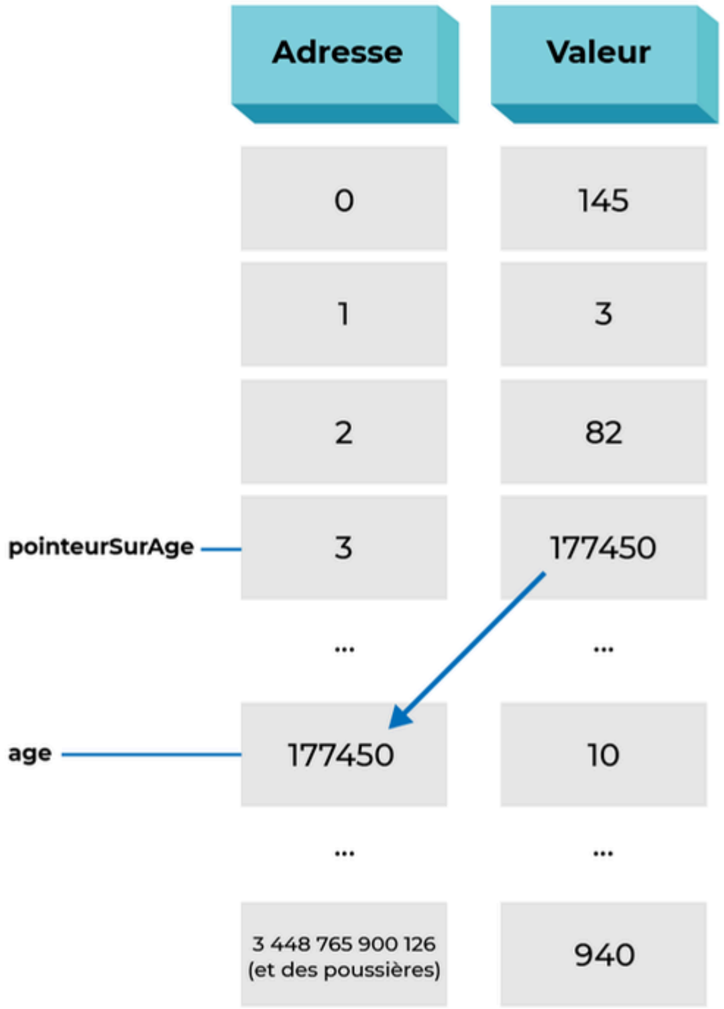
Last update: oct. 2025

Pointeurs et références

Les pointeurs en programmation

- Les pointeurs sont des variables contenant l'adresse mémoire d'une donnée.
- Permettent un accès direct et efficace à la mémoire.

- Cette capacité "bas niveau" améliore les performances (copier une adresse coûte moins cher que copier une grosse structure), mais elle expose aussi à des erreurs critiques (déréférencement invalide, fuites, accès hors limites).
    -

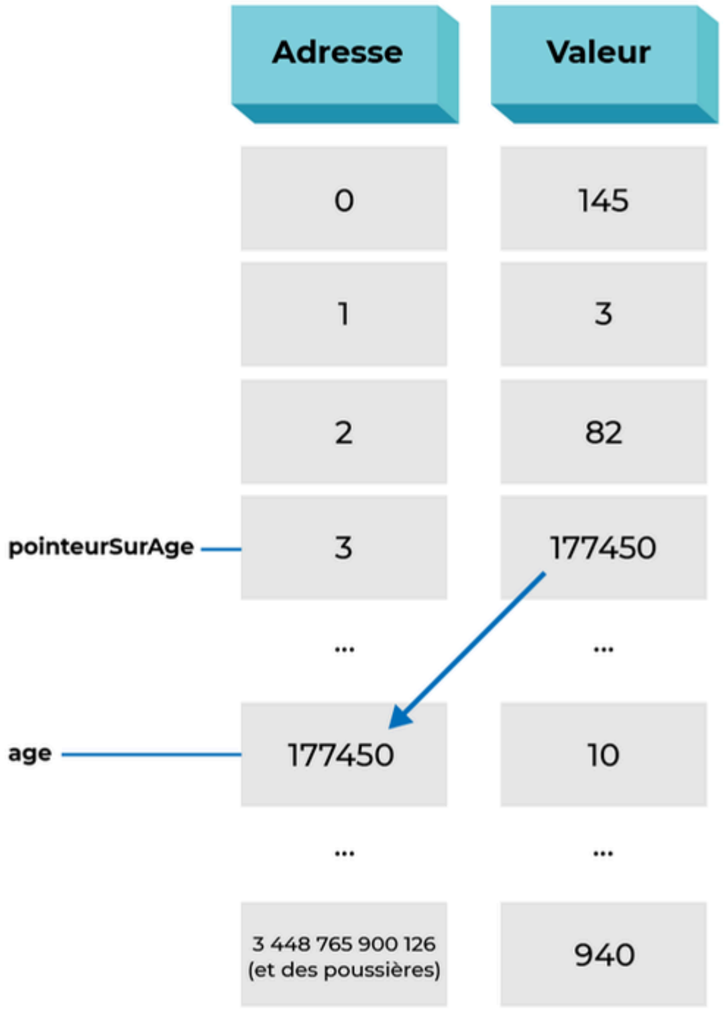


Pointeurs et références

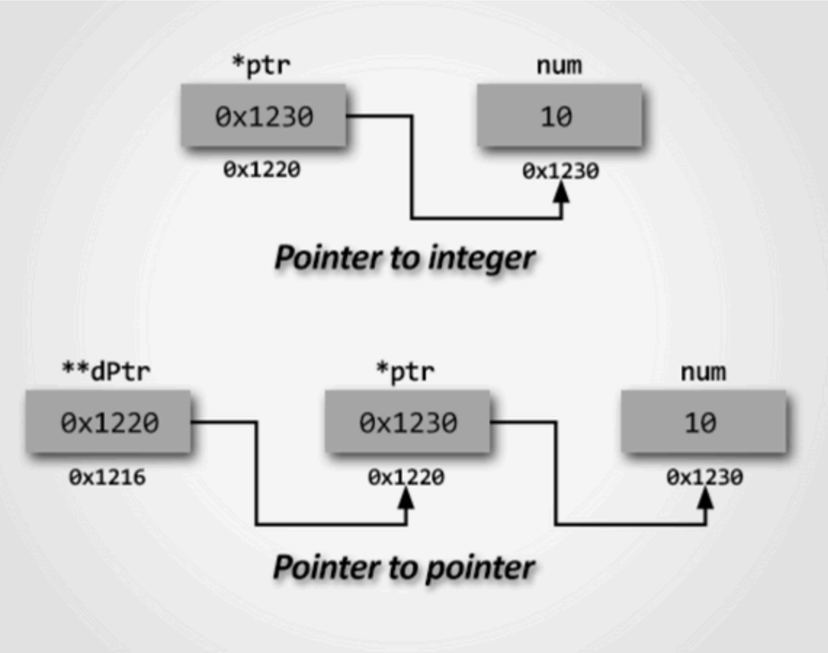
Les références en programmation

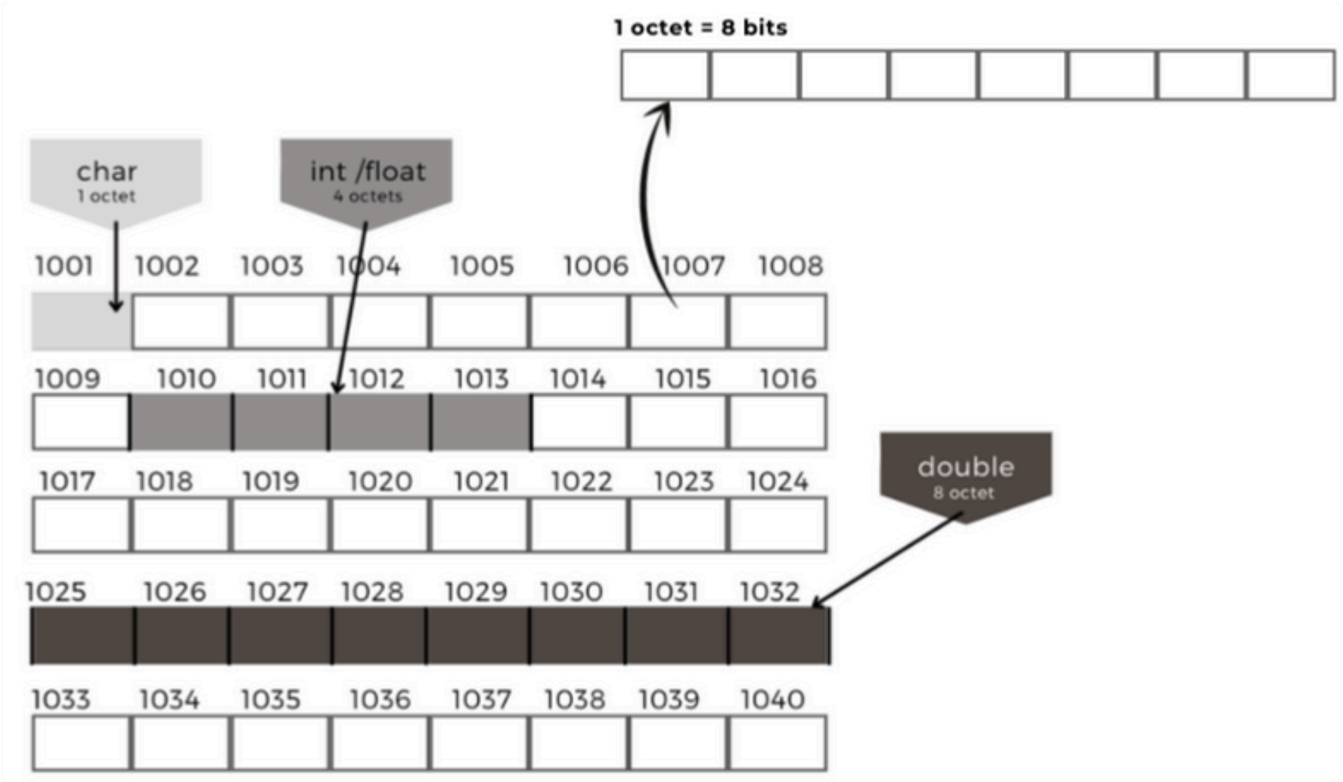
- En revanche, les références (alias de variable) sont généralement plus sûres : elles ne peuvent souvent pas être nulles et évitent les manipulations d'adresse explicites.
  - Exemples: C, C++

- De nombreux langages modernes n'exposent pas de pointeurs explicites et utilisent un ramasse-miettes (GC) pour gérer la mémoire de façon automatique
  - Exemples: (Java, Python, Go, etc.)



Pointeurs et références





Pointeurs vs références dans les langages

Langage	Pointeurs explicites	Références / alias	Gestion mémoire	Avantages / Inconvénients
C	Oui Type T*	Non	Manuel (malloc/free)	Contrôle total et haute performance Risques élevés segfault, fuite de mémoire
C++	Oui Type T* + Pointeurs intelligents	Oui T&, alias sécurisé	Manuel + Destructors /RAII)	Ajout des références (alias) sûres Toujours initialisées, non null Smart pointers pour la sécurité Complexité et gestion manuelle conservées
Java	Non Pas de */& Tout est objet	Oui Références d'objets	Automatique (GC: Gargage collector)	Accès mémoire sécurisé GC, pas de pointeur explicite Moins de contrôle bas-niveau Surcharge du GC
Python	Non Noms liés à des objets	Oui Tout est objet)	Automatique (réf. comptées + GC)	Très haut niveau : pas de manipulation d'adresse directe, code plus simple. **Inconvénient : performances inférieures et impossibilité de gérer explicitement la mémoire.**

Pointeurs en C : déclaration

Déclaration d'un pointeur

- Le C est un langage système fondé sur les pointeurs.
- Une variable occupe un emplacement mémoire fixe.
- Un pointeur est une variable qui contient l'adresse d'une autre variable.

Exemple-

Un pointeur se déclare comme suit :

Type * nomVariblePointeur;
Type: int, char, float, struct, ...

```c
int x = 10;
int *p = &x;  // p contient l'adresse de x
printf("x = %d, *p = %d, adresse p = %p\n", x, *p, p);
```

- *p → déréférencement (accès à la valeur pointée)
- &x → adresse de la variable x

Pointeurs en C : initialisation

Initialisation d'un pointeur

Un pointeur nécessite d'être initialisé après sa déclaration.

Attention

- Si vous ajoutez un nombre à un pointeur; vous ne modifiez pas sa valeur mais vous faites pointer ce dernier au bloc ayant la référence adresse + nombre.

Exemple-

Considérons le programme ci-contre où on affecte à une variable p1 de type pointeur sur entier l'adresse de l'entier nb. p1 contient l'adresse de nb (6e ligne).

```
1  int nb;
2  int * p1;
3
4  void main (void){
5      nb =10;
6      p1 = &nb; /* p1 contient l'adresse de nb*/
7  }
8
```

Pointeurs en C : variable pointée

Accès à une variable pointée

Après avoir déclaré et initialisé un pointeur, la valeur de la variable pointée est accédée par :

- p1 → contient l'adresse de nb
- *p1 → accès ou modification de la valeur stockée à cette adresse

```c
void main (void){
 int nb;
 int * p1;
    nb =10;
    p1 = &nb; /* p1 contient l'adresse de nb*/
    *p1 = 20; /* nb contient la valeur 20 */
}
```

Pointeurs en C : arithmétique

Arithmétique de pointeurs

- Les pointeurs peuvent être incrémentés pour parcourir la mémoire :
    - Exemple:

```c
int tab[] = {10, 20, 30};
int *ptr = tab;
printf("%d\n", *ptr);     // 10
ptr++;
printf("%d\n", *ptr);     // 20
```

⚠️Une opération très puissante, mais pouvant s’avérer risquée!

Pointeurs en C : structures complexes

Manipulations de structures complexes

- Les pointeurs permettent de manipuler des structures complexes:
  - Tableaux, listes chaînées, piles, files

- Les pointeurs permettent de gérer la mémoire dynamique via
  - malloc, realloc: fonction allouant et réallouant dynamiquement la mémoire
  - free: fonction libérant la mémoire déjà allouée

```c
int *arr = malloc(3 * sizeof(int));  // allocation dynamique
arr[0] = 1; arr[1] = 2; arr[2] = 3;
printf("%d\n", *(arr+1));  // accès par arithmétique de pointeur
free(arr);  // libération manuelle
```

Pointeurs en C : erreurs possibles

Risques et erreurs fréquentes

Dangling pointer:
Pointeur vers une zone libérée

```
free(p); *p = 5; // ❌ erreur
```

Out of bounds:
Dépassement de tableau

```
*(arr + 5) = 10; // ❌ accès illégal
```

Double free:
Appeler ***free*** plus d'une fois

```
free(p); free(p); // 💥 crash
```

Missing free:

`malloc()` sans libération

Pointeurs et arguments

Envoyer un pointeur à une fonction

- En C, les fonctions reçoivent des copies des arguments par défaut.
- Pour modifier directement une variable dans main, il faut envoyer l'adresse de la variable via un pointeur.
- Cela permet de mettre à jour plusieurs variables depuis une fonction.

Pointeurs et arguments

Syntaxe et mécanisme

- ***nombre*** est créé dans main.
- On envoie son adresse (***&nombre***) à la fonction *triple*.
- La fonction reçoit un pointeur (***int *p***) et modifie la valeur pointée avec ****p***.
- Après l'appel, la variable originale (***nombre***) est modifiée.

```c
void triple(int *p) {
    *p *= 3;  // modifie la valeur de la variable pointée
}

int main() {
    int nombre = 5;
    triple(&nombre);   // envoie l'adresse de 'nombre'
    printf("%d", nombre); // 15
}
```

Pointeurs et arguments

Variante avec pointeur dans main

- ***nombre*** est créé dans main.
- Un pointeur (****ptr***) sur la variable **nombre** est créé.
- La fonction reçoit le pointeur et modifie la valeur de ***nombre*** pointée avec ****ptr***.
- Après l'appel, la variable originale (***nombre***) est modifiée.

```c
int main() {
    int nombre = 5;
    int *ptr = &nombre;    // pointeur sur nombre
    triple(ptr);           // envoie le pointeur
    printf("%d", *ptr);    // 15
}
```