

# Explication Approfondie : Question 9

## ⌚ Analysons le code étape par étape

```
int arr[] = {5, 10, 15};  
int *p = arr;  
printf("%d %d", *p++, *p);
```

---

## 🔍 Comprendre `*p++` VS `(*p)++`

### Le point clé : Précédence des opérateurs

En C/C++, l'opérateur **postfix ++** a une **précédence plus élevée** que l'opérateur **prefix**

Précédence décroissante :

1. `++` (postfix) ← Plus haute
2. `*` (déréférencement) ← Plus basse

Cela signifie que `*p++` est interprété comme `*(p++)` et non pas `(*p)++`.

---

## 📋 Exécution pas à pas

### État initial :

Copy  
`arr[] = [5, 10, 15]`  
↑  
p pointe à `arr[0]`

### Premier argument : `*p++`

`*p++`

### Étape 1 : Évaluer `p++`

- Cet opérateur postfix retourne la **valeur actuelle** de p
- MAIS incrémente p pour la prochaine utilisation
- Donc : retourne arr (l'adresse originale)

### Étape 2 : Appliquer \* au résultat

- `*(arr) = *p` avant l'incrémantion = `arr[0] = 5`

### Après cette expression :

```
p = &arr[1] ← p a été incrémenté !
```

### Deuxième argument : \*p

\*p

Maintenant p pointe à arr[1]

- $*p = arr[1] = 10$
- 



## Visualisation temporelle

```
Initial : p → arr[0]

printf("%d %d", *p++, *p);
↓
Évalue *p++ :
- Retourne arr[0] = 5
- p devient &arr[1]

Évalue *p :
- p pointe maintenant à arr[1]
- Retourne 10

Résultat : Affiche "5 10"
```

---



## Comparaison avec (\*p)++

Si on avait écrit  $(*p)++$  à la place :

```
int arr[] = {5, 10, 15};
int *p = arr;
printf("%d %d", (*p)++, *p);
```

### Exécution :

#### Premier argument : (\*p)++

- $*p = 5$
- Postfix ++ incrémente la **valeur pointée**
- Retourne 5 (ancienne valeur)
- **arr[0] devient 6**

#### Deuxième argument : \*p

- $*p = 6$  (la valeur a changé !)

## Résultat : Affiche "5 6"

---



## Tableau comparatif

Expression	Effet	Valeur renvoyée	État de p après
<code>*p++</code>	Incrémente <b>le pointeur</b>	Valeur avant incrémantation du pointeur	p pointe au suivant
<code>(*p)++</code>	Incrémente <b>la valeur pointée</b>	Valeur avant incrémantation de la valeur	p inchangé
<code>*++p</code>	Incrémente <b>le pointeur</b> d'abord	Valeur après incrémantation du pointeur	p pointe au suivant
<code>++*p</code>	Incrémente <b>la valeur pointée</b> d'abord	Valeur après incrémantation	p inchangé

---



## Code d'expérimentation

Voici un code pour tester tous les cas :

```
#include <stdio.h>

int main() {
    int arr[] = {5, 10, 15};

    // Test 1 : *p++
    int *p1 = arr;
    printf("Test 1 (*p++) : %d %d\n", *p1++, *p1);
    // Affiche : 5 10

    // Test 2 : (*p)++
    int *p2 = arr;
    printf("Test 2 ((*p)++) : %d %d\n", (*p2)++, *p2);
    // Affiche : 5 6

    // Test 3 : *++p
    int *p3 = arr;
    printf("Test 3 (*++p) : %d\n", *++p3);
```

```

// Affiche : 10

// Test 4 : ++*p
int *p4 = arr;
printf("Test 4 (++*p) : %d\n", ++*p4);
// Affiche : 6

return 0;
}

```

---

## Pièges courants

### Piège 1 : Oublier la précédence

```

int *p = arr;
int val = *p++; // ✓ Correct : * (p++) – le pointeur avance

```

### Piège 2 : Confusion avec (\*p)++

```

int *p = arr;
int val = (*p)++; // X Modifie la valeur pointée, pas le pointeur !

```

### Piège 3 : Utilisation en boucle

```

// ✓ Bon : boucle typique sur tableau
for (int *p = arr; p < arr + 3; p++) {
    printf("%d ", *p);
}

// X Mauvais : mélange *p++ avec d'autres opérations
while (*p++) { // Modifie le pointeur ET accède à la valeur
    // Attention au comportement !
}

```

---

## Mémorisation

**La règle d'or :**  $*p++ = * (p++) \rightarrow$  **le pointeur avance**, on retourne l'ancienne valeur