CRO TD7 : pointeur de fonction et type `void *`

TD sur machine (2h), 6 octobre 2025

1 Petit retour sur les déclarations de type

On rappelle que pour comprendre une déclaration, il faut savoir la parenthéser correctement. Par exemple pour la déclaration `int *q[3]`

- `[]` plus prioritaire que `*`, donc :
  `int *q[3]` ⇔ `int (*(q[3]))`
- l'expression `(*(q[3]))` est de type `int`
- l'expression `q[3]` est de type *pointeur vers un* `int`
- l'expression (i.e. la variable) `q` est de type *tableau de pointeur vers un* `int`

ou, en partant de l'intérieur :

- `int (*(`**`q[3]`**`)) // q une tableau de`
- `int `**`(*`**`(q[3])`**`)` `// q une tableau de pointeur`
- **int** `(*(q[3])) // q une tableau de pointeur vers un int`

QUESTION 1 ▶ quel est le type de `foo` : `char(* foo)[5];`

QUESTION 2 ▶ Dans les déclaration de type, les parenthèses *postfixes* indique que ce qui précède les parenthèse est une fonction. par exemple, dans la déclaration de ; `int fonct1(int a)`

- l'expression `fonct1(int a)` est de type `int`
- l'expression (i.e. le symbole) `fonct1` est de type fonction qui prend un `int` et renvoie un `int`

Quel est le type de la variable `addition` dans la déclaration suivantes, est ce que c'est le même type que la variable `fonc1` ci-dessus ?
`int addition(int a, int b);`

QUESTION 3 ▶ Il existe un type de pointeur particulier qui est le type `void *`. Ce type de pointeur est appelé *pointeur générique*. Les pointeurs sont tous stockés sur 64 bits (ou 32 bits il y quelques années), ils sont donc transtypables : un pointeur vers un entier peut être transtypé en un pointeur sur un char, le premier octet de l'entier, c'est simplement l'arthmétique de pointeur qui sera modifié suivant le type de pointeur.
Ce type de pointeur permet d'adresser la mémoire comme tout autre pointeur. L'intérêt du type `void *` est de pouvoir unifier tous les types de pointeurs. En effet, on peut convertir tous les pointeurs vers le type `void *` et réciproquement.

QUESTION 4 ▶ Si on a une déclaration de `tmp` comme ca : `void *temp` :

1. Peux-t'on écrire `tmp++;` dans le code de la fonction ?
2. Peux-t'on écrire :

```
1  void *temp=0;
2  int *p;
3
4  p = (int *)temp;
5  p++;
```

Introduction aux Pointeurs de Fonctions

En langage C, les pointeurs de fonctions permettent de stocker l'adresse d'une fonction dans une variable. Cela permet d'appeler des fonctions de manière dynamique, offrant ainsi une grande flexibilité, notamment pour l'implémentation de structures de données et le développement de bibliothèques modulaires. Les pointeurs de fonctions sont souvent utilisés dans les tables de fonctions, les rappels (callbacks) et les applications où il est nécessaire de manipuler des fonctions de manière générique.

Déclaration et Utilisation Pour déclarer un pointeur de fonction, il faut spécifier le type de retour et les paramètres de la fonction. La syntaxe générale est la suivante :

```
// Declaration d un pointeur vers une fonction
retour (*nom_pointeur)(type1, type2, ...);
```

où `retour` est le type de retour de la fonction pointée, et `type1, type2, ...` sont les types des paramètres.

Exemple de Base Dans l'exemple suivant, nous créons un pointeur de fonction pour une fonction qui prend deux entiers en paramètre et retourne un entier.

```
// Fonction addition
int addition(int a, int b) {
    return a + b;
}

// Declaration du pointeur de fonction
int (*operation)(int, int);

// Affectation de l'adresse de la fonction
operation = addition;

// Utilisation du pointeur de fonction
int resultat = operation(5, 3); // Appelle addition(5, 3)
printf("Resultat: %d\n", resultat); // Affiche 8
```

Mais si vous êtes très attentifs, vous noterez que, lors de l'affectation de la fonction `operation`, on aurait du écrire : `operation = &addition;` puisque `operation` est *un pointeur sur une fonction*. Est ce que ce code est juste ?
Oui, on peut écrire les deux, la raison en est la suivante : en C, une fonction est presque toujours automatiquement castée en pointeur de fonction (et inversement). Pour les fonctions et les tableaux qui ne sont pas des L-values (on les appelle quelquefois des *labels*) le compilateur identifie `a` et `&a` (sauf dans certains cas comme l'utilisation dans la fonction `sizeof` par exemple, ou le passage par référence).
`foncPtr=&fonct1` ⇔ `foncPtr=fonct1`
`(*foncPtr)(10);` ⇔ `(foncPtr)(10)`
Tout comme pour les tableaux :`tab` ⇔ `&tab`

Les pointeurs de fonctions en C offrent une flexibilité importante pour :

- implémenter des fonctions *callback*, par exemple pour des gestionnaires d'événements,
- organiser du code modulaire et réutilisable, et dynamiquement modifiable

2 Tri par ordre croissant ou décroissant

On suppose qu'on a une fonction de comparaison :
`int (*comparaison)(int, int))`
qui compare deux entiers. le code suivant effectue une tri par permutation sur un tableau `tableau` de taille `taille` avec la fonction `comparaison`.

```
1  //tri par permuation avec fcomp comme fonction de comparaison
2  for (i=0;i<taille;i++)
3    {
4      min=tableau[i];
5      for (j=i+1;j<taille;j++)
6        if (comparaison(tableau[i],tableau[j]))
7          {
8            min = tableau[j];
9            tableau[j]=tableau[i];
10            tableau[i]=min;
11         }
12   }
```

QUESTION 5 ▶ Écrivez un programme C qui implémente ce tri par permutation dans une fonction `tri` :
`tri(int tableau[], int taille, int (*comparaison)(int, int))`
On écrira ensuite deux fonctions `croissant` et `décroissant` qui permettront lorsque l'on les passera en argument de la fonction `tri` de trier de manière croissante ou décroissante. Testez vos deux ordres de tri.

3 Fonction `qsort`

Tiens ! mais cette fonction de tri existe dans la librairie standard C ! Elle s'appelle `qsort` (mais elle n'a pas exactement le meme prototype que notre fonction `tri`).

QUESTION 6 ▶ Tapez `man qsort`, comprenez vous le fonctionnement de la fonction `qsort`.

QUESTION 7 ▶ Écrivez un programme `main.c` qui tri un tableau d'entier en utilisant la fonction `qsort`.
Hint : pour cela vous devrez écrire une fonction `compareInt` qui a le prototype demandé. pour récupérer les valeurs des deux arguments passés à la fonction `compareInt`, il faudra faire un transtypage qui ressemble à : `int firstInt = * (const int *) first;`

QUESTION 8 ▶ Ajouter maintenant une fonction pour trier les tableaux de chaînes de caractère (on pourra utiliser la fonction `strcmp` de la bibliothèque `string`).

4 Pointeur de pointeur de fonction

Imaginons que l'on veuille changer dynamiquement le comportement d'une fonction. Par exemple avec une fonction `changeOrdre` qui modifie le tri de votre première question (passe de croissant à décroissant ou l'inverse). Comment faire ?
Il faut que votre fonction prenne en paramètre un *pointeur sur un pointeur de fonction* (puisqu'on fonction ne peut pas modifier son paramètre).
On vous propose d'essayer avec ce prototype pour la fonction `changeOrdre` :

```
1    changeOrdre(int (**fcomp1)(int, int), int (*fcomp2)(int, int))
```

QUESTION 9 ▶ Modifiez le programme de la section 2 pour que votre programme demande à l'utilisateur si il veut un ordre croissant ou décroissant.