

Les pointeurs de fonctions

Comment on écrit une interface dans un langage qui n'en a pas

Ressource R3.03 — BUT2 Informatique

September 11, 2026

Pourquoi cette note

Au TP2, vous allez écrire une *interface* en C. Or le C n'a ni *interface*, ni *class*, ni *implements*. Il a autre chose — les pointeurs de fonctions — et cette note montre comment on passe de l'un à l'autre, en partant de l'exemple le plus simple possible.

À retenir dès maintenant : **les principes de conception ne sont pas des propriétés du langage**. L'inversion des dépendances s'écrit en C, en Java ou en Python ; seule la syntaxe change.

1 Une variable qui contient une fonction

Vous savez qu'une variable peut contenir un entier, ou l'adresse d'un entier. Elle peut aussi contenir l'adresse d'une fonction.

```
1 #include <stdio.h>
2
3 int addition(int a, int b) { return a + b; }
4 int soustraction(int a, int b) { return a - b; }
5
6 int main(void)
7 {
8     int (*operation)(int, int); /* une variable qui contient une fonction */
9
10    operation = addition;
11    printf("%d\n", operation(7, 3)); /* affiche 10 */
12
13    operation = soustraction;
14    printf("%d\n", operation(7, 3)); /* affiche 4 */
15
16    return 0;
17 }
```

C'est tout. La variable `operation` désigne tantôt une fonction, tantôt une autre, et `operation(7, 3)` appelle celle qu'elle désigne *au moment de l'exécution*.

✓ À faire : Comment lire la déclaration

```
int (*operation)(int, int);
```

On lit en partant du nom, vers l'extérieur :

operation est un pointeur (*)
vers une fonction prenant (int, int)
et renvoyant int.

Les parenthèses autour de `*operation` sont indispensables. Sans elles, `int *operation(int, int)` déclare tout autre chose : une *fonction* qui renvoie un `int *`. C'est l'erreur la plus fréquente, et le compilateur ne vous aidera pas à la voir.

Un typedef rend la suite lisible

```
1 typedef int (*operation_t)(int, int);
2
3 operation_t operation = addition; /* beaucoup plus clair */
```

2 Regrouper plusieurs fonctions : la structure devient un contrat

Une fonction seule ne fait pas une interface. Une interface, c'est un ensemble d'opérations qui vont ensemble. En C, on les met dans une struct :

```
1 typedef struct journal {
2     void (*ecrire)(const char *msg);
3 } journal_t;
```

On tient là un contrat : « quoi que tu sois, tu sais écrire un message ». Qui écrit, et où, n'est pas dit — et c'est exactement le but.

3 Le chaînon manquant : void *ctx

Le contrat ci-dessus a un défaut. Si une implémentation a besoin de **se souvenir de quelque chose** — un fichier ouvert, un compteur, une liste de messages — elle n'a nulle part où le ranger. Elle devrait recourir à une variable globale, et deux instances ne pourraient plus coexister.

On ajoute donc un champ qui transporte l'état de l'implémentation, et on le passe en premier argument de chaque opération :

```
1 typedef struct journal {
2     void *ctx; /* l'état, propre a chacun */
3     void (*ecrire)(void *ctx, const char *msg);
4 } journal_t;
```

C'est le this des langages à objets

En Java ou en C++, quand vous écrivez `objet.methode(x)`, le compilateur passe en secret un premier argument : l'objet lui-même. Python ne le cache même pas, il l'appelle `self`.

En C, rien n'est caché : **c'est vous qui écrivez le `this`**, et il s'appelle `ctx`.

4 Un exemple complet, en 70 lignes

Voici, en miniature, **toute l'architecture de votre projet** : un contrat, deux implémentations, un métier qui ne connaît que le contrat, et un test qui devient possible grâce à la seconde implémentation.

Le port — le contrat

```
1 typedef struct journal {
2     void *ctx;                                /* l'état de l'implémentation */
3     void (*ecrire)(void *ctx, const char *msg);
4 } journal_t;
```

Le métier — il ignore où part le message

```
1 static void saluer(const journal_t *j, const char *nom)
2 {
3     char message[64];
4     snprintf(message, sizeof message, "Bonjour %s !", nom);
5     j->ecrire(j->ctx, message);
6 }
```

Relisez cette fonction : elle ne contient ni `printf`, ni `fopen`, ni rien qui touche le monde extérieur. Elle est donc testable.

Adaptateur 1 — le vrai : le terminal

```
1 static void terminal_ecrire(void *ctx, const char *msg)
2 {
3     (void) ctx;                                /* pas d'état : rien à transporter */
4     printf("%s\n", msg);
5 }
6
7 static journal_t journal_terminal(void)
8 {
9     journal_t j = { .ctx = NULL, .ecrire = terminal_ecrire };
10    return j;
11 }
```

Adaptateur 2 — la doublure : la mémoire

```
1 typedef struct {
2     char lignes[8][64];
3     int nb;
4 } memoire_t;
5
6 static void memoire_ecrire(void *ctx, const char *msg)
7 {
8     memoire_t *m = ctx;                        /* ICI ctx sert : c'est notre état */
9     if (m->nb < 8)
10        snprintf(m->lignes[m->nb++], 64, "%s", msg);
11 }
12
13 static journal_t journal_memoire(memoire_t *m)
14 {
15     journal_t j = { .ctx = m, .ecrire = memoire_ecrire };
16    return j;
17 }
```

Le test — impossible sans la doublure

```

1 static void test_saluer(void)
2 {
3     memoire_t m = { .nb = 0 };          /* Arrange */
4     journal_t j = journal_memoire(&m);
5
6     saluer(&j, "Karim");                 /* Act */
7
8     if (m.nb == 1 && strcmp(m.lignes[0], "Bonjour Karim !") == 0)
9         printf(" [OK] le message est correct\n");    /* Assert */
10    else
11        printf(" [ECHEC] reçu : \"%s\"\n", m.lignes[0]);
12 }

```

✓ À faire : Ce que ce test prouve, et que rien d'autre ne prouverait

Le métier a produit **exactement** « Bonjour Karim ! ». Sans la doublure, il faudrait rediriger la sortie standard vers un fichier, la relire, et espérer que rien d'autre n'y écrive. Avec la doublure : trois lignes, aucune entrée-sortie, quelques microsecondes.

C'est *précisément* ce que vous ferez pour tester qu'un fichier inchangé n'est pas recopié.

5 Et dans le projet ?

La correspondance est terme à terme :

La miniature	Le projet	Rôle
journal_t	fs_port_t	le contrat : les opérations dont le métier a besoin
saluer()	creer_instantane()	le métier, qui ne connaît que le contrat
terminal_ecrire	fs_posix.c	l'adaptateur réel, seul à faire des appels système
memoire_ecrire	la doublure de test	une arborescence en mémoire, au lieu du disque
ctx = NULL	ctx = NULL	l'adaptateur réel n'a pas d'état
ctx = &memoire	ctx = la fausse arborescence	la doublure, elle, en a un

Une seule différence d'échelle : votre port aura **cinq** opérations au lieu d'une, parce que votre scénario nominal en réclame cinq.

Le vocabulaire, pour le dire juste

Port	la structure de pointeurs de fonctions — le contrat.
Adaptateur	une implémentation de ce contrat, tournée vers une technologie précise (POSIX, la mémoire, un service distant).
Injection	le fait de <i>passer</i> le port en paramètre au métier, au lieu qu'il aille le chercher lui-même.
Polymorphisme	deux adaptateurs différents derrière le même appel <code>j->ecrire(...)</code> . En C, c'est un simple déréférencement de pointeur.

6 Quatre pièges

Piège	Ce qui se passe
<code>int *f(int)</code> au lieu de <code>int (*f)(int)</code>	vous déclarez une fonction, pas un pointeur. Le message d'erreur sera très loin de la cause.
appeler un pointeur non initialisé	<code>j->ecrire</code> vaut n'importe quoi : plantage immédiat, ou pire. Initialisez toujours la structure entièrement.
oublier <code>(void)ctx</code> ;	<code>-Wextra</code> lève <code>unused parameter</code> sur chaque opération qui ignore <code>ctx</code> . Cinq avertissements prévisibles noient les vrais.
mettre un état dans une variable globale	deux instances ne peuvent plus coexister, et deux tests s'influencent mutuellement. C'est exactement ce que <code>ctx</code> évite.

7 Ce n'est pas une astuce de cours

Vous croiserez ce motif partout dans le monde Unix :

<code>qsort(3)</code>	reçoit en dernier argument un pointeur vers votre fonction de comparaison
<code>struct file_operations</code>	le noyau Linux : chaque pilote remplit la même structure de pointeurs (<code>read</code> , <code>write</code> , <code>open...</code>)
<code>FILE *</code>	la libc : un fichier, un tube ou un tampon mémoire, derrière les mêmes <code>fread</code> / <code>fwrite</code>
<code>signal(2)</code> , <code>atexit(3)</code>	vous confiez au système une fonction qu'il appellera plus tard

Un pilote Linux, c'est une structure de pointeurs de fonctions remplie par son auteur. Vous êtes en train d'écrire la même chose, en plus petit.

✓ À faire : À essayer avant le TP2, dix minutes

Recopiez l'exemple des 70 lignes dans un fichier, compilez-le avec `cc -std=c11 -Wall -Wextra -Wpedantic`, exécutez-le. Puis :

1. retirez le `(void)ctx;` et observez l'avertissement ;
2. ajoutez une troisième implémentation qui écrit dans un fichier ;
3. écrivez un second test vérifiant que `saluer` n'écrit **qu'une** ligne.

Si ces trois points vous sont clairs, le TP2 ne vous posera aucune difficulté de langage : il ne vous restera que la question d'analyse — *quelles opérations mettre dans le port ?*