Lire un appel indirect en C

Fonction, champ de structure et contexte

BUT2 Informatique

24 septembre 2026

Cette capsule complète le CM3. Elle sert à préparer la lecture du port fourni au TP3. Elle n'exige pas de concevoir ce port ni d'écrire un adaptateur POSIX. L'exemple complet se trouve dans `exemples-cm/cm3.c`.

1 Une variable peut contenir une adresse de fonction

```c
int addition(int a, int b) { return a + b; }
int difference(int a, int b) { return a - b; }
/* Dans main : */
int (*operation)(int, int) = addition;
int x = operation(7, 3);
operation = difference;
int y = operation(7, 3);
```

`x` vaut 10, `y` vaut 4. En lisant depuis le nom, `operation` est un pointeur vers une fonction prenant deux entiers et renvoyant un entier. Les parenthèses autour de `*operation` sont nécessaires : sans elles, on déclare une fonction qui renvoie un pointeur.

2 Une structure peut porter cette adresse

```c
typedef struct {
    void *ctx;
    int (*envoyer)(void *ctx, const char *message);
} notification_t;
int avertir(const notification_t *n) {
    return n->envoyer(n->ctx, "Nouvelle echeance");
}
```

1. `n` est l'adresse d'une structure valide.
2. `n->envoyer` lit le champ qui désigne une fonction compatible.
3. `n->ctx` lit une adresse de données à transmettre à cette fonction.
4. Les parenthèses finales réalisent l'appel avec ce contexte et le message.

Le contrat annonce 0 en cas de succès et -1 en cas d'échec. Le programme choisit l'implémentation en construisant la structure avant l'appel.

3 Le contexte transporte un état propre au test

```c
typedef struct { int echec; int appels; } simulation_t;
int envoyer_simule(void *ctx, const char *message) {
    simulation_t *s = ctx;
    (void)message;
    s->appels++;
    return s->echec ? -1 : 0;
}
/* Dans main : */
simulation_t s = { .echec = 1, .appels = 0 };
notification_t n = { .ctx = &s, .envoyer = envoyer_simule };
int code = avertir(&n);
```

Après l'appel, `code` vaut -1 et `s.appels` vaut 1. Aucun message n'est envoyé. Le test a contrôlé une réponse et peut observer le nombre d'appels.

Le montage doit garantir que `ctx` pointe vers le bon type et que l'objet existe encore pendant les appels. `void *` ne vérifie pas cette promesse. La structure n'emporte pas automatiquement une copie de l'état.

4 Exercice de lecture

Avec le montage précédent, affecter `s.echec = 0`, puis appeler à nouveau `avertir(&n)`. Prédire le résultat et le compteur. La fonction appelée a-t-elle changé ?

Correction : retour 0 et compteur 2. La fonction reste `envoyer_simule`. On a changé une donnée. Affecter une autre fonction compatible à `n.envoyer` changerait le comportement choisi derrière le même appel.

5 Quatre vérifications avant de modifier

— La signature affectée correspond au type du champ. On affecte une adresse de fonction, pas le résultat d'un appel.
— Le pointeur de fonction est initialisé avant l'appel.
— Le contexte a le type attendu par l'implémentation et une durée de vie suffisante.
— Les valeurs de retour conservent la même signification pour toutes les implémentations.

Une variable globale éviterait de passer le contexte, mais partagerait l'état entre les tests ou les instances. Ici, chaque test construit son propre état.

✓✓ Point d'arrêt — Manipulation courte avant le TP3

Compiler `exemples-cm/` avec `make test`. Dans une copie de `cm3.c`, changer la valeur initiale de `echec`, prédire les assertions qui échoueront, puis exécuter. Expliquer ensuite la différence entre modifier l'état et remplacer la fonction. Aucun nouveau système de fichiers n'est à écrire.