Base VIII

Programmation séquentielle en C, 2020-2021

Orestis Malaspinas (A401), ISC, HEPIA

2020-01-06

Inspirés des slides de F. Glück

Tests unitaires

- Compilation != bon fonctionnement!
- Toujours tester vos programmes.
- Tester les fonctionnalités une par une $\Rightarrow$ **tests unitaires**.
- Plus le code est modulaire, plus il est simple à tester.
- Plus le code est testé, moins il aura contiendra de bugs.

Testing shows the presence, not the absence of bugs. E. W. Dijkstra.

Assertions (1/3)

```c
#include <assert.h>
void assert(int expression);
```

Qu'est-ce donc?

- Macro permettant de tester une condition lors de l'exécution d'un programme:
  - Si `expression == 0` (condition fausse), `assert()` affiche un message d'erreur sur `stderr` et termine l'exécution du programme.
  - Sinon l'exécution se poursuit normalement.

À quoi ça sert?

- Permet de réaliser des tests unitaires.
- Permet de tester des conditions catastrophiques d'un programme.
- **Ne permet pas** de gérer les erreurs.

Assertions (2/3)

Exemple

```c
#include <assert.h>
#include <stdlib.h>
void copy(int *src, int *dst, int n) {
    // identique à assert(src != NULL)
    assert(src); assert(dst);
    assert(n >= 0);

    for (int i = 0; i < n; ++i) {
        dst[i] = src[i];
    }
}
void fill(int *dst, int n, int val) {
    assert(dst &&
        "problem with allocated mem");
    assert(n >= 0 &&
        "n is the size of dst");

    for (int i = 0; i < n; ++i) {
        dst[i] = val;
    }
}
```

```c
int main(int argc, char **argv) {
    int size = 10;
    int *src = malloc(size *
                      sizeof(int));
    fill(src, size, 0);
    int *dst = malloc(size *
                      sizeof(int));
    copy(src, dst, size);

    return EXIT_SUCCESS;
}
```

Assertions (3/3)

Cas typiques d'utilisation

- Vérification de la validité des pointeurs (typiquement `!= NULL`).
- Vérification du domaine des indices (dépassement de tableau).

Bug vs erreur de *runtime*

- Les assertions sont là pour détecter les bugs (erreurs d'implémentation).
- Les assertions ne sont pas là pour gérer les problèmes externes au programme (allocation mémoire qui échoue, mauvais paramètre d'entrée passé par l'utilisateur, ...).

Généricité en C

Problématique

- En C on doit écrire chaque algorithme/structures de données pour des types précis (arbres, tri, ...).
- Duplication du code pour chaque type possible et imaginable.

Solution: `void *`

- En général, un pointeur connaît son **adresse** et le **type** des données sur lesquelles il pointe.

```
int *a = malloc(sizeof(*a));
int *b = malloc(sizeof(int));
```

- Un `void *` le connaît **que** son adresse, au programmeur de pas faire n'importe quoi.
- Vous avez déjà utilisé des fonctions utilisant des `void *`

```
void *malloc(size_t size);
void free(void *);
```

Attention danger

- Ne permet pas au compilateur de vérifier les types.
- Les données pointées n'ayant pas de type, il faut déréférencer avec précaution:

```
int a = 2;
void *b = &a; //jusqu'ici tout va bien
double c = *b; // argl!
```

- À la programmeuse de faire attention à ce qu'elle fait.

Cas d'utilisation (1/3)

Que fait cette fonction?

```
void *foo(void *tab, int n_items, int s_items,
          bool (*bar)(void *, void *)) {
    if (n_items <= 0 || s_items <= 0 || NULL == tab) {
        return NULL;
    }
    void *elem = tab;
    for (int i = 1; i < n_items; ++i) {
        // void pointer arithmetics is illegal in C
        // (gcc is ok though)
        void *tmp_elem = (void *)((char *)tab + i*s_items);

        if (bar(elem, tmp_elem)) {
            elem = tmp_elem;
        }
    }
    return elem;
}
```

Cas d'utilisation (2/3)

Avec un tableau de `int`

```
bool cmp_int(void *a, void *b) {
    return (*(int *)a < *(int *)b);
}

int main() {
    int tab[] = {-1, 2, 10, 3, 8};
    int *a = foo(tab, 5, sizeof(int), cmp_int);
    printf("a = %d\n", *a);
}
```

Cas d'utilisation (3/3)

Avec un tableau de `double`

```
bool cmp_dbl(void *a, void *b) {
    return (*(double *)a < *(double *)b);
}

int main() {
    double tab[] = {-1.2, 2.1, 10.5, 3.6, 18.1};
    double *a = foo(tab, 5, sizeof(double), cmp_dbl);
    printf("a = %f\n", *a);
}
```