

## Solution exercice 2.15

```
#include <assert.h>
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>

int* initialiser_1(size_t taille, int valeur);
int* initialiser_2(size_t taille, int valeur);
void afficher(const int* ptr, size_t taille);

int main(void) {
    {
        const size_t TAILLE = 3;
        int* p = initialiser_1(TAILLE, 1);
        afficher(p, TAILLE); // Affiche [1, 1, 1]
        free(p);
    }

    {
        const size_t TAILLE = 5;
        int* p = initialiser_2(TAILLE, 2);
        afficher(p, TAILLE); // Affiche [2, 2, 2, 2, 2]
        free(p);
    }

    return EXIT_SUCCESS;
}

int* initialiser_1(size_t taille, int valeur) {
    assert(taille > 0);
    int* p = (int*) calloc(taille, sizeof(int));
    if (p)
        for (int* tmp = p; tmp < p + taille; *tmp++ = valeur);
    return p;
}

int* initialiser_2(size_t taille, int valeur) {
    assert(taille > 0);
    int* p = (int*) calloc(taille, sizeof(int));
    if (p) {
        const int* const FIN = p + taille;
        for (; p < FIN; *p++ = valeur);
        p -= taille;
    }
    return p;
}

void afficher(const int* ptr, size_t taille) {
    assert(ptr != NULL);
    printf("[");
    for (size_t i = 0; i < taille; ++i) {
        if (i > 0)
            printf("%s", ", ");
        printf("%d", ptr[i]); // ou *(ptr + i)
    }
    printf("]\n");
}
```

## Solution exercice 2.16

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

int* inverse(const int* debut, const int* fin);
void afficher(const int tab[], size_t taille);
void test(const int tab[], size_t taille);

int main(void) {
    {
        const int TAB[] = {1};
        test(TAB, sizeof(TAB) / sizeof(int));
    }

    {
        const int TAB[] = {1, 2};
        test(TAB, sizeof(TAB) / sizeof(int));
    }

    {
        const int TAB[] = {1, 2, 3};
        test(TAB, sizeof(TAB) / sizeof(int));
    }

    {
        const int TAB[] = {1, 2, 3};
        test(TAB, 0);
    }

    return EXIT_SUCCESS;
}

int* inverse(const int* debut, const int* fin) {
    assert(debut != NULL);
    assert(fin != NULL);
    assert(fin - debut + 1 > 0); // pour garantir que (cf ci-dessous) TAILLE > 0
    const size_t TAILLE = (size_t) (fin - debut + 1);
    int* ptr = (int*) calloc(TAILLE, sizeof(int));
    if (ptr)
        for (size_t i = 0; i < TAILLE; ++i)
            ptr[TAILLE - 1 - i] = debut[i]; // ou ptr[i] = *fin--;
    return ptr;
}

void afficher(const int tab[], size_t taille) {
    assert(tab != NULL);
    printf("[");
    for (size_t i = 0; i < taille; ++i) {
        if (i > 0)
            printf("%s", ", ");
        printf("%d", tab[i]);
    }
    printf("]\n");
}
```

```
void test(const int tab[], size_t taille) {
    printf("Avant inversion : \n");
    afficher(tab, taille);
    int* ptr = inverse(tab, tab + taille - 1);
    printf("Après inversion : \n");
    afficher(ptr, taille);
    free(ptr);
}

// Avant inversion :
// [1]
// Après inversion :
// [1]
// Avant inversion :
// [1, 2]
// Après inversion :
// [2, 1]
// Avant inversion :
// [1, 2, 3]
// Après inversion :
// [3, 2, 1]
// Avant inversion :
// []
// Assertion failed: fin - debut + 1 > 0, file Ex2-16.c, line 37
```

## Solution exercice 2.18

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void afficher(const int* adr, size_t n);

int main(void) {

    int tab1[] = {1, 2, 3};
    const size_t SIZE = sizeof(tab1) / sizeof(int);

    afficher(tab1, SIZE);
    int* tab2 = (int*) calloc(SIZE, sizeof(int));
    if (tab2) {
        memcpy(tab2, tab1, sizeof(tab1));
        afficher(tab2, SIZE);
        free(tab2);
    }

    return EXIT_SUCCESS;
}

void afficher(const int* adr, size_t n) {
    assert(adr != NULL);
    printf("[");
    for (size_t i = 0; i < n; ++i) {
        if (i > 0)
            printf(", ");
        printf("%d", *(adr + i));
    }
    printf("]\n");
}
```

## Solution exercice 2.19

```
memset(tab, 0, SIZE * sizeof(int));
```

ou

```
memset(tab, 0, sizeof(tab));
```

## Solution exercice 2.20

memmove est tres tres utile.

## Solution exercice 2.21

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef unsigned short ushort;

void initialiser_1(int* adr, size_t m, size_t n);
void initialiser_2(int* adr, size_t m, size_t n);
void initialiser_3(int* adr, size_t m, size_t n);

void afficher(const int* adr, size_t m, size_t n);

int main(void) {

    int t1[3][3];
    int t2[3][3];
    int t3[3][3];

    int t4[3][4];
    int t5[3][4];
    int t6[3][4];

    initialiser_1((int*) t1, 3, 3); // ou t1[0] ou &t1[0][0]
    initialiser_2((int*) t2, 3, 3);
    initialiser_3((int*) t3, 3, 3);

    initialiser_1((int*) t4, 3, 4);
    initialiser_2((int*) t5, 3, 4);
    initialiser_3((int*) t6, 3, 4);

    afficher((int*) t1, 3, 3);
    afficher((int*) t2, 3, 3);
    afficher((int*) t3, 3, 3);

    afficher((int*) t4, 3, 4);
    afficher((int*) t5, 3, 4);
    afficher((int*) t6, 3, 4);

    return EXIT_SUCCESS;
}

void initialiser_1(int* adr, size_t m, size_t n) { // Approche peu efficace
    assert(adr != NULL);
    assert(m > 0);
    assert(n > 0);
    for (size_t i = 0; i < m; ++i)
        for (size_t j = 0; j < n; ++j)
            *adr++ = i == 0 || i == m - 1 || j == 0 || j == n - 1;
    // Autres variantes possibles :
    // *(adr + i * n + j) = i == 0 || i == m - 1 || j == 0 || j == n - 1;
    // adr[i * n + j] = i == 0 || i == m - 1 || j == 0 || j == n - 1;
}
```

```

void initialiser_2(int* adr, size_t m, size_t n) { // Approche plus efficace
    assert(adr != NULL);
    assert(m > 0);
    assert(n > 0);
    // Mettre tous les éléments à 0
    memset(adr, 0, m * n * sizeof(int));
    // Mettre les "bords" sup et inf à 1
    for (size_t j = 0; j < n; ++j)
        adr[j] = adr[n * (m - 1) + j] = 1;
    // Mettre les "bords" gauche et droit à 1
    for (size_t i = 1; i < n - 1; ++i)
        adr[n * i] = adr[n * i + n - 1] = 1;
}

// L'avantage de l'approche ci-dessous est de pouvoir réutiliser [i][j]
void initialiser_3(int* adr, size_t m, size_t n) {
    assert(adr != NULL);
    assert(m > 0);
    assert(n > 0);
    // tableau de pointeurs sur les lignes de la matrice
    int** ad = (int**) calloc(m, sizeof(int*));
    assert(ad != NULL);
    for (size_t i = 0; i < m; ++i)
        ad[i] = &adr[i * n]; // ou adr + i * n car rappel :
                             // *(adr + i * n) = adr[i * n]
                             // => &adr[i * n] = adr + i * n
    // Mettre tous les éléments à 0
    memset(adr, 0, m * n * sizeof(int));
    // Mettre les "bords" sup et inf à 1
    for (size_t j = 0; j < n; ++j)
        ad[0][j] = ad[m - 1][j] = 1;
    // Mettre les "bords" gauche et droit à 1
    for (size_t i = 1; i < m - 1; ++i)
        ad[i][0] = ad[i][n - 1] = 1;
    // Restituer la mémoire allouée dynamiquement
    free(ad);
}

void afficher(const int* adr, size_t m, size_t n) {
    assert(adr != NULL);
    printf("[");
    for (size_t i = 0; i < m; ++i) {
        if (i > 0)
            printf("%s", ", ");
        printf("[");
        for (size_t j = 0; j < n; ++j) {
            if (j > 0)
                printf("%s", ", ");
            printf("%d", *adr++);
        }
        printf("]");
    }
    printf("\n");
}

// [[1, 1, 1], [1, 0, 1], [1, 1, 1]]
// [[1, 1, 1], [1, 0, 1], [1, 1, 1]]
// [[1, 1, 1], [1, 0, 1], [1, 1, 1]]
// [[1, 1, 1, 1], [1, 0, 0, 1], [1, 1, 1, 1]]
// [[1, 1, 1, 1], [1, 0, 0, 1], [1, 1, 1, 1]]
// [[1, 1, 1, 1], [1, 0, 0, 1], [1, 1, 1, 1]]

```