LangageC_1-2

December 3, 2024

*Ceci est le classeur numéro **2** de la matière Langage C du **1er semestre**, à réaliser dans le temps de la deuxième séance de Langage C.*

Ce deuxième classeur contient les points suivants : - Sous-programmes - Pointeurs - Tableaux

Il se termine avec quelques petits exercices de mise en pratique. (Ces exercices ne sont pas à rendre mais vous pouvez bien sûr les envoyer à votre encadrant.e de TP pour qu'iel y jette un oeil).

0.1 Les sous-programmes

Le langage C permet de définir des *sous-programmes*. Contrairement à des langages comme ADA et Fortran, le C ne **fait pas** la différence entre une *procédure* et une *fonction*.

0.1.1 Déclarations et implantations

Un sous-programme est déclaré avec : * Un type de retour * Un nom (mêmes règles que pour les variables) * Une liste de paramètres, un paramètre étant composé d'un type et d'un identifiant

Cela donne l'instruction suivante :

```
<type> <nom>(<type1> <param1>, <type2> <param2> [...]);
```

L'ensemble de ces éléments forment la *signature* du sous-programme.

Le type de retour d'un sous-programme peut être le type spécial `void`, qui signifie "aucune valeur retournée". On pourrait donc considérer qu'un sous-programme qui retourne `void` est une procédure (mais l'inverse n'est pas nécessairement vrai). À noter aussi qu'un sous-programme peut tout à fait ne définir aucun paramètre, auquel cas on laisse les parenthèses avec rien entre.

[1]:
```c
int mon_sous_programme(char a, float b); // Sous-programme qui prend 2␣
↪paramètres (un caractère et un flottant)
                                         // et retourne un entier
float pi(); // Sous-programme sans paramètre, qui retourne un flottant
void ecrire(int i); // Sous-programme avec un paramètre entier, qui ne retourne␣
↪rien
void mystere(); // Sous-programme qui ne prend rien en entrée et ne donne rien␣
↪en sortie

int main() {} // (obligé d'avoir un symbole main pour Jupyter)
```

Une fois déclaré, on donne *l'implantation* d'un sous-programme en rappelant sa signature, et en remplaçant le point-virgule `;` par un bloc d'instructions :

[2]:
```c
//%cflags: -I .
#include "affichage.h"

// Déclarations
int mon_sous_programme(char a, float b);
float pi();
void ecrire(int i);
void mystere();

// Implantations
int mon_sous_programme(char a, float b) {
    int x = (int) a + (int) b;
    return x; // Retour de la fonction
}

float pi() {
    return 3.1415;
}

void ecrire(int i) {
    afficher_entier(i);
}

void mystere() {
    puts("Des choses mystérieuses se passent");
}

int main() {}
```

Il faut savoir que le nom d'une fonction est *unique*, et est associé à une signature précise (cela n'est pas le cas en Java ou en C++ par exemple, qui autorisent la *surcharge*). Si l'implantation et la signature ne concordent pas sur le type de retour et les arguments, cela créera une erreur à la compilation.

[3]:
```c
//%cflags: -I .
#include "affichage.h"

float pi();

double pi() { // Mauvais type de retour
    return 3.1415;
}

int ecrire(int x);

int ecrire(char x) { // Mauvais type pour l'argument
    afficher_carac(x);
```

```c
}

void deux(char a, char b);

void deux(char a) { // Mauvais nombre d'arguments
    afficher_carac(a);
}
```

```
/tmp/tmpxqne9qgl.c:6:8: error: conflicting types for 'pi'
    6 | double pi() { // Mauvais type de retour
      |        ^~
/tmp/tmpxqne9qgl.c:4:7: note: previous declaration of 'pi' was here
    4 | float pi();
      |       ^~
/tmp/tmpxqne9qgl.c:12:5: error: conflicting types for 'ecrire'
   12 | int ecrire(char x) { // Mauvais type pour l'argument
      |     ^~~~~~
/tmp/tmpxqne9qgl.c:10:5: note: previous declaration of 'ecrire' was here
   10 | int ecrire(int x);
      |     ^~~~~~
/tmp/tmpxqne9qgl.c:18:6: error: conflicting types for 'deux'
   18 | void deux(char a) { // Mauvais nombre d'arguments
      |      ^~~~
/tmp/tmpxqne9qgl.c:16:6: note: previous declaration of 'deux' was here
   16 | void deux(char a, char b);
      |      ^~~~
[C kernel] GCC exited with code 1, the executable will not be executed
```

Comme pour les variables, le C ne permet pas de *déclaration en avant*. À chaque ligne d'un fichier, les sous-programmes accessibles sont ceux qui ont été déclarés avant. Si la fonction existe mais est déclarée *après*, cela donne une erreur (parfois…).

À noter par ailleurs qu'une implémentation seule vaut aussi déclaration (on pourrait appeler ça une "déclaration-implantation"). Il n'est donc pas généralement nécessaire de déclarer une fonction avant de l'implanter, ce qui permet de faire de la récursivité sans problème. La déclaration devient nécessaire lorsque l'on fait des modules ou lorsque l'on écrit des fonctions mutuellement récursives.

[4]:
```c
//%cflags: -I .
#include "affichage.h"

// Déclaration de pi() avant son utilisation
//float pi();  // Si on décommente le code compile

float fois_pi(float x) {
    return x * pi();
}

float pi() {
```

```c
    return 3.1415;
}

// "Déclaration-implantation" de fact, dont le bloc utilise fact (récursion)
int fact(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * fact(n - 1);
    }
}

int main() {
    afficher_flottant(fois_pi(2.0));
    afficher_entier(fact(5));
    return 0;
}
```

```
/tmp/tmpxcmfzcn9.c: In function 'fois_pi':
/tmp/tmpxcmfzcn9.c:8:16: warning: implicit declaration of function 'pi'
[-Wimplicit-function-declaration]
    8 |     return x * pi();
      |                ^~
/tmp/tmpxcmfzcn9.c: At top level:
/tmp/tmpxcmfzcn9.c:11:7: error: conflicting types for 'pi'
   11 | float pi() {
      |       ^~
/tmp/tmpxcmfzcn9.c:8:16: note: previous implicit declaration of 'pi' was here
    8 |     return x * pi();
      |                ^~
[C kernel] GCC exited with code 1, the executable will not be executed
```

Attention : avoir une déclaration de fonction sans implantation conduit à une erreur à l'édition des liens, du type *undefined reference to xxx*. C'est une erreur difficile à traquer, car elle advient *après* la compilation (le C autorise à appeler des fonctions qui ne sont déclarées nul part) ; soyez donc très vigilants et assurez-vous que toutes vos fonctions sont bien implémentées.

0.1.2 Appel de sous-programme

Une fois déclaré/implanté, un sous-programme peut être *appelé*. Pour ce faire, on donne son nom et la liste des arguments correspondant à chaque paramètre du sous-programme :

```
<sous-programme>(<argument 1>, <argument 2>);
```

Un argument doit être une expression valide. On peut donc y mettre des littéraux, des opérations, etc. À noter que si le sous-programme n'a pas de paramètres, on doit quand même mettre les parenthèses (dont l'intérieur est alors vide).

Si le sous-programme retourne une valeur, son appel compte comme une expression valide, et peut donc être stocké dans une variable, combiné dans une expression, et même utilisé comme argument

d'un autre appel de fonction !

[5]:
```c
//%cflags: -I .
#include "affichage.h"

int mon_sous_programme(char a, float b);
float pi();
void ecrire(int i);
void mystere();

int mon_sous_programme(char a, float b) {
    int x = (int) a + (int) b;
    return x;
}

float pi() {
    return 3.1415;
}

void ecrire(int i) {
    afficher_entier(i);
}

void mystere() {
    puts("Des choses mystérieuses se passent");
}

int main() {
    float x = 3.0;
    int y = 27;

    // Appels de sous-programmes
    mon_sous_programme('a', x);
    ecrire((int) x + y);

    // Appels de sous-programmes sans arguments
    pi();
    mystere();

    // Appels de sous-programmes comme expressions
    int z = mon_sous_programme('z', pi() * x);
    ecrire(z);
}
```

```
i = 30
Des choses mystérieuses se passent
i = 131
```

Note : l'instruction `return` L'instruction spéciale `return` interrompt un sous-programme, et *retourne* la valeur associée (si le sous-programme le permet). Concrètement, lors d'un appel de sous-programme, si l'instruction `return xxx` est atteinte, le sous-programme s'arrête et son appel est comme "remplacé" par la valeur de `xxx`.

On peut donner un `return` sans valeur (dans un sous-programme qui ne retourne rien), auquel cas aucune valeur n'est transmise mais le sous-programme est interrompu.

[6]:
```c
//%cflags: -I .
#include "affichage.h"

void test() {
    afficher("avant");
    return;
    afficher("après"); // Code jamais atteint
}

int compare_0(float f) {
    if (f > 0.0) {
        return 1;
    } else if (f < 0.0) {
        return -1;
    } else {
        return 0;
    }
}

int main() {
    test();
    afficher_entier(compare_0(37.9));
    afficher_entier(compare_0(-1.7e-4));
    afficher_entier(compare_0((float) 0));
}
```

```
avantcompare_0(37.9) = 1
compare_0(-1.7e-4) = -1
compare_0((float) 0) = 0
```

Vous aurez peut-être remarqué que le sous-programme ***`main`*** *(qui est effectivement un sous-programme (presque) comme les autres) ne présente pas d'instruction* ***`return`****. Il se trouve que c'est tout à fait admis par la plupart des compilateurs, et correspond implicitement à faire* ***`return 0`****.*

En pratique cependant, il ne faut ***jamais, jamais*** *faire ça. De manière général, si c'est implicite, c'est une mauvaise idée.*

Dans la suite, nous mettrons le ***`return 0`*** *à la fin du main.*

0.1.3 Programmation par contrats

Un sous-programme doit exhiber un contrat, que l'on donne généralement au niveau de la déclaration/spécification (et que l'on rappelle parfois au niveau de l'implantation, même si c'est bien la déclaration que l'utilisateur va consulter).

Pour rappel, un contrat se compose : - du nom de la fonction - d'une description de ce que fait la fonction - des paramètres avec leur utilité - de la valeur de retour (le cas échéant) - des pré et post conditions - des cas d'erreur (le cas échéant)

On propose le patron suivant :

```
/**
 * fact
 * Cette fonction calcule la factorielle d'un nombre entier long.
 *
 * Paramètres :
 *  n     entier dont on veut la factorielle
 *
 * Retour : n!, défini par 0! = 1 et pour tout n > 0, n! = n * (n - 1)!
 *
 * Pré-conditions :
 *   - n >= 0
 *   - n! < valeur maximale d'un entier long
 *
 * Post-conditions :
 *   - résultat > 0
 *   - si n > 1 alors résultat > 1
 *
 * Cas d'erreur : aucun
 */
```

Petit exercice : somme des entiers de 1 à n Spécifier (dans un contrat) puis écrire le sous-programme qui, pour un entier n retourne la somme des entiers de 1 à n (dont l'algorithme a été implanté dans le classeur précédent).

[13]:
```
//%cflags: -I .
#include "affichage.h"

/**
somme
Cette fonction calcule la somme des entiers de 1 à un entier choisi

Paramètres :
n    l'entier jusq'où on veut sommer

Retour : s la somme des entiers de 1 à n

Pré-conditions :
```

```
    - n > 0

Post-conditions :
    - s > 0

Cas d'erreur : aucun
**/

int somme(int n) {
   int s=0;
   for (int i=1;i<=n;i++) {
       s+=i;
   }
return s;
}


int main() {
    afficher_entier(somme(5));
    afficher_entier(somme(10));

    afficher_entier(somme(21));
    afficher_entier(somme(140));

    return 0;
}
```

```
somme(5) = 15
somme(10) = 55
somme(21) = 231
somme(140) = 9870
```

0.2 Les pointeurs

Matériellement, une variable prend la forme d'un ensemble contigu d'emplacements mémoire, associés à des *adresses*. On peut accéder à l'adresse d'une variable `x` (ou plutôt du premier emplacement qu'elle occupe) à l'aide de l'opérateur unaire spéciale `&x`, appelé "référencement". Le résultat de cette opération est appelée *pointeur sur la variable* `x`.

Cette adresse peut être manipulée comme n'importe quelle autre expression du langage (on dit qu'un pointeur est un *citoyen de première classe*), et en particulier être stockée dans une autre variable, passée en paramètre d'une fonction, etc. Si la variable `x` est de type `<type>`, alors un pointeur sur `x` est de type `<type>*` (notez l'étoile).

Lorsqu'on a une adresse/un pointeur `p`, on peut accéder à l'emplacement mémoire associé à cette adresse en utilisant l'opérateur unaire spéciale `*p`, appelé "déréférencement". À noter que `*p` se comporte comme une variable. Notamment, on peut s'en servir dans une expression, et on peut s'en servir **à gauche d'une affectation** (auquel cas, ce qui est affecté est bien l'emplacement pointé par `p` !).

De manière duale au référencement, si le type du pointeur `p` est `<type>*` alors le type de la valeur pointée est `<type>` (et on remarque que l'on ne peut pas déréférencer une variable qui n'est pas un pointeur).

[16]:
```c
//%cflags: -I .
#include "affichage.h"

int main() {
    int a = 777;
    char b = '$';
    float c = 0.01;

    // Référencement
    int* p_a = &a;
    char* p_b = &b;
    float* p_c = &c;

    // Déréférencement
    int a2 = *p_a;
    afficher_entier(a2);

    afficher_carac(*p_b); // p_b est une variable comme les autres, on peut␣
 ↪l'utiliser dans des expressions !

    afficher_flottant((*p_c) * 100);

    // Affectation d'emplacement déréférencé
    *p_b = '+'; // Légal
    afficher_carac(*p_b);
    afficher_carac(b); // p_b pointe sur b == l'emplacement *p_b et b sont les␣
 ↪mêmes
                       // (si on modifie l'un ça modifie l'autre)

    // Référencement imbriqués
    int** p_p_a = &p_a; // p_a étant une variable, on peut pointer dessus...
    int*** p_p_p_a = &p_p_a; // ...et ainsi de suite (notez le type qui gagne␣
 ↪des étoiles à chaque fois)

    afficher_entier(***p_p_p_a); // On peut tout déréférencer d'un coup
    // (en fait * est associatif, ceci est équivalent à *(*(*p_p_p_a)))

    // Déréférencement interdit
    // afficher_entier(*a); // ERREUR: type incompatible !

    return 0;
}
```

```
a2 = 777
```

```
*p_b = $
(*p_c) * 100 = 1.000000
*p_b = +
b = +
***p_p_p_a = 777
```

L'intérêt des pointeurs est que l'on a accès à *l'emplacement* de la variable, et pas seulement à la variable. Cela signifie qu'une mise à jour du contenu de l'emplacement sera "répercutée" entre tous les pointeurs. En quelques sortes, c'est comme si on avait "partagé" l'accès à une variable, en lecture et en écriture !

[17]:
```c
//%cflags: -I .
#include "affichage.h"

int main() {
    int a = 5;
    int* p_a1 = &a; // Un premier pointeur sur a
    int* p_a2 = &a; // Un second pointeur sur a

    afficher_entier(a);
    afficher_entier(*p_a1);
    afficher_entier(*p_a2);

    // On modifie la variable directement
    a = 6;
    afficher_entier(a);
    afficher_entier(*p_a1);
    afficher_entier(*p_a2);

    // On modifie la variable en utilisant le pointeur
    *p_a1 = 10;
    afficher_entier(a);
    afficher_entier(*p_a1);
    afficher_entier(*p_a2);

    return 0;
}
```

```
a = 5
*p_a1 = 5
*p_a2 = 5
a = 6
*p_a1 = 6
*p_a2 = 6
a = 10
*p_a1 = 10
*p_a2 = 10
```

Il existe une valeur spéciale pour les pointeurs, qui correspond à dire "ne pointe sur rien". Cette

valeur est `NULL`. C'est à cette valeur qu'on initialise un pointeur qui n'est pas encore utilisé, et c'est aussi souvent le pointeur qu'on retourne lorsqu'une fonction s'est mal passée.

En interne, NULL est une constante qui vaut 0. En fait, peu importe l'OS, peu importe le processeur, la mémoire ou autre, l'adresse 0 **n'est jamais une adresse valide**. C'est pour cela qu'on s'autorise à l'utiliser pour indiquer un pointeur qui ne pointe sur rien.

0.2.1 Pointeurs comme paramètres

Un intérêt majeur des pointeurs vient du fait que, en C, contrairement à ADA, **il n'y a pas de direction pour les paramètres** (in/out). Formellement, quand on appelle une fonction, la *valeur* de chaque argument est *copiée* puis donnée au bloc de code correspondant. Autrement dit, tous les paramètres sont en *in* : toute modification apportée à un paramètre **n'est pas répercutée en dehors de la fonction**.

[18]:
```c
//%cflags: -I .
#include "affichage.h"

void ma_fonction(int a) {
    a = a + 1; // Je modifie a
    afficher_entier(a);
}

int main() {
    int x = 21;
    afficher_entier(x);

    ma_fonction(x);
    afficher_entier(x); // x n'a pas changé, même si on fait +1 dans la fonction

    // Heureusement que c'est le cas, sinon quel est le sens de cette␣
 ↪instruction ?
    ma_fonction(5);
    afficher_entier(5);

    return 0;
}
```

```
x = 21
a = 22
x = 21
a = 6
5 = 5
```

Mais si on utilise un *pointeur* comme paramètre, certes on ne peut pas changer le pointeur lui-même, mais on a accès à l'emplacement sur lequel il pointe, et on peut mettre à jour le contenu de cet emplacement !

[19]:
```c
//%cflags: -I .
#include "affichage.h"

void ma_fonction(int* p_a) { // Pointeur
    *p_a = *p_a + 1; // Je modifie **le contenu de l'emplacement pointé** (en␣
 ↪utilisant ce même contenu)
    afficher_entier(*p_a);
}

int main() {
    int x = 21;
    afficher_entier(x);

    ma_fonction(&x); // Je passe la référence
    afficher_entier(x); // x a changé !!

    // ma_fonction attend un pointeur, l'instruction suivante cause une erreur␣
 ↪de type
    // ma_fonction(x);

    // Cette instruction n'est plus possible
    // ma_fonction(5);
    // ma_fonction(&5); // Ça non plus ça ne marche pas : & ne peut être␣
 ↪utilisé que sur une variable

    return 0;
}
```

```
x = 21
*p_a = 22
x = 22
```

Pour résumer : si on veut modifier un paramètre de façon à ce que la mise à jour soit répercutée en dehors de la fonction (ce qui est analogue à un paramètre en mode *in-out*), on doit utiliser des pointeurs.

Lorsqu'un paramètre est donné sous forme de pointeur, on parle de *passage par référence*. Sinon, on parle de *passage par valeur*.

Petit exercice : la division euclidienne Spécifier et écrire le sous programme qui, pour deux entiers a et b, calcule le quotient et le reste de la division euclidienne de a par b (dont l'algorithme a été écrit dans le classeur précédent).

Le quotient sera communiqué à l'appelant par *la valeur de retour* du sous-programme, tandis que le reste sera communiqué *à l'aide d'un pointeur*. Si le pointeur en paramètre est `NULL`, alors on ne l'affecte pas (on considèrera que l'appelant n'a pas besoin du reste).

[42]:
```c
//%cflags: -I .
#include "affichage.h"

/**
division
Cette fonction calcule le quotient et le reste de la division euclidienne de␣
 ↪deux entiers

Paramètres :
a    le dividende
b    le diviseur
p_r  le pointeur associé au reste

Retour :
q    le quotient

Pré-conditions :
    - a >= 0 et 0 < b <= a

Post-conditions :
    - b * q + p_r* = a
    - 0 <= p_r* < b

Cas d'erreur : aucun
**/
int division(int a, int b, int* p_r) {
    int r = a;
    int q = 0;
    while (r>=b) {
        ++q;
        r -= b;
    }
    if (p_r == NULL) {}
    else {
        *p_r = r;
    }
    return(q);
}

int main() {
    int q, r;

    q = division(1287, 50, &r); // Je récupère le reste
    afficher_entier(q);
    afficher_entier(r);

    q = division(97813, 100, NULL); // Je m'en fiche du reste
```

```
    afficher_entier(q);

    return 0;
}
```

```
q = 25
r = 37
q = 978
```

0.3 Les tableaux

Un tableau est une succession contiguë d'emplacements mémoire qui permet de stocker plusieurs éléments d'un même type. En C, les tableaux ne sont pas vraiment des "types" à proprement parler (en fait, il s'agit de pointeurs, le pointeur sur la première case du tableau). Néanmoins, le langage présente quelques syntaxes spécifiques pour les manipuler.

On peut déclarer un tableau d'éléments de type `<type>` et de taille `<taille>` avec l'instruction suivante :

```
<type> mon_tableau[<taille>];
```

À noter que la taille d'un tableau défini ainsi **est nécessairement constante** (nous verrons au deuxième semestre comment faire des tableaux de taille dynamique).

On peut initialiser un tableau (= donner des valeurs pour chaque emplacement) avec une liste de valeurs entre accolades, mais seulement au moment de la déclaration (on ne peut pas utiliser ça dans des affectations seules) :

```
<type> mon_tableau[<taille>] = { <valeur 1>, <valeur 2>, ... };
```

Il est possible de donner moins de valeurs que la taille donnée, auquel cas les premières cases sont initialisées avec les valeurs, et les autres ne sont juste pas initialisées (impossible de donner plus de valeurs qu'attendues, par contre).

Si on donne une initialisation au tableau, on peut omettre la taille, auquel cas la taille du tableau correspond au nombre de valeurs utilisées dans l'initialisation.

On peut accéder aux emplacements du tableau avec le nom de la variable et l'indice de l'emplacement entre crochets (attention les indices commencent à 0). Il s'agit bien *d'emplacement*, que l'on peut donc utiliser dans des expressions ou à gauche d'affectations.

[43]:
```
//%cflags: -I .
#include "affichage.h"

int main() {
    char tableau_car[3] = { 'a', 'b', 'c' }; // Tableau de 3 caractères␣
 ↪initialisé
    afficher_carac(tableau_car[1]);
    tableau_car[1] = 'z'; // L'emplacement marche comme une variable en lecture␣
 ↪et en écriture
    afficher_carac(tableau_car[1]);
```

```
    int tableau_entiers[10]; // Tableau d'entiers de taille 10 (non-initialisé)
    for (int i = 0; i < 10; i++) {
        tableau_entiers[i] = i * 2; // On peut utiliser une expression comme␣
 ↪indice !
    }
    afficher_entier(tableau_entiers[3] + tableau_entiers[6]);

    float tableau_f[] = { 1.5, 2.5 }; // Tableau de deux flottants
    afficher_flottant(tableau_f[0] + tableau_f[1]);

    return 0;
}
```

```
tableau_car[1] = b
tableau_car[1] = z
tableau_entiers[3] + tableau_entiers[6] = 18
tableau_f[0] + tableau_f[1] = 4.000000
```

Attention, la validité d'un indice n'est pas vérifiée (ni à la compilation ni à l'exécution). Accéder à une case du tableau en dehors de l'intervalle défini est donc *dangereux* (ça *pourrait* planter, ou pas ; à voir laquelle de ces deux situations est pire).

0.3.1 Tableaux comme paramètres

Le C permet de passer des tableaux en paramètre de fonctions. On renseigne le type tableau du paramètre comme si on déclarait un tableau (`<type> xxx[]`). À noter qu'on ne donne pas de taille, car le compilateur ne fait aucune vérification sur la compatibilité des tableaux passés en paramètre (contrairement à Ada).

Avoir un tableau en paramètre est absolument équivalent à avoir un pointeur en paramètre (qui pointe sur une succession d'emplacements). En particulier, **les tableaux en paramètre sont in-out par défaut**.

[46]:
```
//%cflags: -I .
#include "affichage.h"

void somme(int a[]) {
    a[2] = a[0] + a[1];
}

void somme2(int* a) { // Totalement équivalent
    a[2] = a[0] + a[1];
}

int main() {
    int tab[3] = { 3, 5, 0 };
    somme(tab);
    afficher_entier(tab[2]);
```

```
    tab[0] = 27;
    tab[1] = 661;
    somme2(tab);
    afficher_entier(tab[2]);

    return 0;
}
```

```
tab[2] = 8
tab[2] = 688
```

On peut empêcher la direction *out* des tableaux (et même des pointeurs, en fait) en utilisant le mot-clef `const`. Cela demande au compilateur de vérifier que les emplacements marqués n'apparaissent jamais à gauche d'une affectation.

[47]:
```
//%cflags: -I .
#include "affichage.h"

int somme(const int a[]) {
    //a[2] += a[0] + a[1]; // Illégal ! a apparaît à gauche d'une affectation !
    return a[0] + a[1] + a[2];
}

int somme2(const int* a) {
    //a[2] += a[0] + a[1]; // Illégal ! a apparaît à gauche d'une affectation !
    return a[0] + a[1] + a[2];
}

int main() {
    int tab[3] = { 3, 5, 7 };
    int r = somme(tab);
    afficher_entier(tab[2]);
    afficher_entier(r);

    tab[0] = 27;
    tab[1] = 661;
    r = somme2(tab);
    afficher_entier(tab[2]);
    afficher_entier(r);

    return 0;
}
```

```
tab[2] = 7
r = 15
tab[2] = 7
r = 695
```

À noter qu'en C la taille d'un tableau n'est pas stockée par défaut. Soit on connaît la taille à la compilation (parce qu'on a déclaré le tableau nous-même), soit on est obligé de manipuler la taille en plus (en variable ou en paramètre).

[48]:
```c
//%cflags: -I .
#include "affichage.h"

/**
 * afficher_tab
 * Affiche le tableau d'entiers de taille donnée.
 *
 * Paramètres :
 *   tab      tableau à afficher
 *   taille   taille du tableau
 *
 * Pré-conditions :
 *   - tab != NULL
 *   - tab contient au moins taille cases
 */
void afficher_tab(int* tab, int taille) {
    for (int i = 0; i < taille; i++) { // Je croise les doigts que␣
 ↪l'utilisateur ne m'aie pas menti
        afficher_entier(tab[taille - i - 1]);
    }
    nouvelle_ligne();
}

int main() {
    int mon_tab[5] = { 1, 3, 7, 11, 2 };

    afficher_tab(mon_tab, 5); // Honnête
    afficher_tab(mon_tab, 3); // Malhonnête mais ça respecte la pré-condition
    afficher_tab(mon_tab, 10); // Potentiellement dangereux

    return 0;
}
```

```
tab[taille - i - 1] = 2
tab[taille - i - 1] = 11
tab[taille - i - 1] = 7
tab[taille - i - 1] = 3
tab[taille - i - 1] = 1

tab[taille - i - 1] = 7
tab[taille - i - 1] = 3
tab[taille - i - 1] = 1

tab[taille - i - 1] = 32764
```

```
tab[taille - i - 1] = 1185508016
tab[taille - i - 1] = 0
tab[taille - i - 1] = 4196880
tab[taille - i - 1] = 0
tab[taille - i - 1] = 2
tab[taille - i - 1] = 11
tab[taille - i - 1] = 7
tab[taille - i - 1] = 3
tab[taille - i - 1] = 1
```

0.3.2 Les chaînes de caractères

En C, les chaînes de caractère sont en fait des *tableaux de caractères*. Comme le C ne stock pas la longueur des tableaux, les chaînes de caractère contiennent toujours une case en plus qui contient un 0 (`'\0'`) et qui délimite sa fin. Le *contrat* des fonctions manipulant des chaînes de caractères suppose toujours que la chaîne se trouve entre son début et le caractère 0 (il ne faut donc surtout pas le retirer où on risque de parcourir un tableau plus loin que sa taille). Cela signifie également que tout ce qui se trouve après le premier 0 est systématiquement ignoré.

Comparé aux autres tableaux, les chaînes de caractères bénéficient de sucre syntaxique pour l'initialisation : on peut écrire `"hello"` directement au lieu d'écrire `{ 'h', 'e', 'l', 'l', 'o', '\0' }` (ce qui serait la façon "correcte" de faire).

[49]:
```c
//%cflags: -I .
#include "affichage.h"

int main() {
    char test[] = "ma super chaîne de caractères"; // Longueur et caractère de␣
 ↪terminaison implicite
    afficher_chaine(test);

    char test2[100] = "chaîne pas trop longue"; // Chaîne plus courte que␣
 ↪taille déclarée (légal)
    afficher_chaine(test2);                    // (à noter que la taille du␣
 ↪tableau est bien 100)

    char test3[] = "cette chaine s'arrête ici\0et tout le reste est ignoré";
    afficher_chaine(test3);

    char test4[] = { 'c', '\'', 'e', 's', 't', ' ', 'l', 'o', 'n', 'g', ' ', '!
 ↪', '\0' };
    afficher_chaine(test4);

    return 0;
}
```

```
test = ma super chaîne de caractères
```

```
test2 = chaîne pas trop longue
test3 = cette chaine s'arrête ici
test4 = c'est long !
```

0.4 Exercices

Mettons en application ce que l'on a vu dans ce classeur.

Note : dans la suite, nous utilisons un petit module de tests unitaires développé en interne (`libt`). Ce module ajoute des instructions spéciales : - `INITIALIZE_TESTS();` instruction à exécuter avant tout autre du module (au début du `main` en général) - `BEGIN_SECTION("ma-section") ... END_SECTION()` délimite une *section* (= une collection de tests) nommée "ma-section" (compte comme un bloc) - `BEGIN_TESTI("mon-test") ... END_TEST` délimite un test unitaire (= un bout de code avec un oracle) nommé "mon-test" (compte comme un bloc) - `REPORT_TO_STDOUT` affiche le rapport des tests sur la sortie standard (sinon le moteur n'affiche rien !)

Les oracles (= ce qui détermine si un test est validé ou non) sont les suivant : - `FAIL(msg)` échoue ave les message donné - `ASSERT(cond)` échoue si et seulement si la condition `cond` s'évalue à faux - `ASSERT_EQ(a, b)` échoue si et seulement si `a` et `b` sont *différents* - `ASSERT_EQ_F(a, b, eps)` échoue si et seulement si les *flottants* `a` et `b` sont distant de plus de `eps` - `ASSERT_EQ_S(a, b)` échoue si et seulement si les *chaînes de caractère* `a` et `b` sont différentes

Il y a d'autres fonctionnalités, mais tout ceci est suffisant pour le moment !

[51]:
```c
//%cflags: -I .
//%ldflags: -L. -lt
#include "affichage.h"
#include "tests.h"

int main() {
INITIALIZE_TESTS();

BEGIN_SECTION("ma-section_1")
    int r = 1; // r locale à la section

    BEGIN_TESTI("test-1")
    int x = r * 2 + 1; // x locale au test
    ASSERT_EQ(x, 3);
    END_TEST

    BEGIN_TESTI("test-2")
    int x = r + r + 1;
    int y = x * x;
    ASSERT(x == 3); // On peut faire autant d'ASSERT qu'on veut
    ASSERT(y == 9);
    END_TEST

    BEGIN_TESTI("test-3")
    FAIL("test incomplet")
```

```
    END_TEST

    REPORT_TO_STDOUT

END_SECTION()

    return 0;
}
```

```
[Section ma-section_1 - RESULTS]
( 3/ 3) test-3 FAILURE (/tmp/tmpxb55h54t.c:25)
        test incomplet
TOTAL: #tests: 3, #signaled: 0, #timedout: 0, #failures: 1 (success ratio:
66.67%)
```

0.4.1 1. Le PGCD, le retour

Compléter le code suivant en y intégrant un sous-programme qui calcule le PGCD. Vous devez déterminer vous-même la signature du sous-programme, l'implantation du sous-programme sera l'algorithme que vous avez écrit au classeur 1, exercice 2. Attention à bien respecter le contrat donné.

[ ]:
```
//%cflags: -I .
//%ldflags: -L. -lt
#include "affichage.h"
#include "tests.h"

/**
 * pgcd
 * Calcule le PGCD de deux entiers relatifs.
 *
 * Paramètres :
 *   a    premier entier
 *   b    second entier
 *
 * Retour :
 *   d nombre entier positif tel que d est le plus grand nombre qui divise a et␣
 ↪b
 *
 * Pré-conditions :
 *   - a et b non simultanément nuls
 *   - a et b peuvent être négatifs
 *
 * Erreurs : aucune
 */
int pgcd(int a, int b){
```

```
    // Vérifier que a est bien positif et en prendre l'opposé le cas échéant
    if (a<0) {
        a = -a;
    } else {}

    // Idem avec b
    if (b<0) {
        b = -b;
    } else {}

    // Calcul du PGCD
    while (a!=b) {
        if (a>b) {
            a -= b;
        } else {
            b -= a;
        }
    }
    return(a);
}

int main() {
    INITIALIZE_TESTS();

    BEGIN_SECTION("pgcd");
        BEGIN_TESTI("pgcd-nominal-1");
        int r = pgcd(24, 16);
        ASSERT_EQ(r, 8);
        END_TEST;

        BEGIN_TESTI("pgcd-nominal-2");
        int r = pgcd(22022, 770);
        ASSERT_EQ(r, 154);
        END_TEST;

        BEGIN_TESTI("pgcd-neg-1");
        int r = pgcd(-15457, 4459);
        ASSERT_EQ(r, 13);
        END_TEST;

        BEGIN_TESTI("pgcd-neg-2");
        int r = pgcd(45167, -31093);
        ASSERT_EQ(r, 31);
        END_TEST;

        BEGIN_TESTI("pgcd-neg-3");
        int r = pgcd(-124579, -104377);
```

```
        ASSERT_EQ(r, 3367);
        END_TEST;

        BEGIN_TESTI("pgcd-0-1");
        int r = pgcd(771, 0);
        ASSERT_EQ(r, 771);
        END_TEST;

        BEGIN_TESTI("pgcd-0-2");
        int r = pgcd(0, -131);
        ASSERT_EQ(r, 131);
        END_TEST;

        REPORT_TO_STDOUT;
    END_SECTION()
}
```

0.4.2 2. Moyenne d'un tableau

Implanter une fonction qui réalise la moyenne des nombre réels donnés sous forme de tableau.

[ ]:
```
//%cflags: -I .
//%ldflags: -L. -lt
#include "affichage.h"
#include "tests.h"

/**
 * moyenne
 * Calcule la moyenne des nombres réels donnés en paramètre, sous forme de␣
 ↪tableau.
 *
 * Paramètres :
 *   tab     tableau de réels à moyenner
 *   taille  taille du tableau
 *
 * Retour :
 *   m = moyenne des taille éléments du tableau
 *
 * Pré-conditions :
 *   - taille > 0
 *   - taille effective du tableau >= taille
 *   - tab non null
 */
float moyenne(float tab[],int taille) {
    float s = 0;

    for (int i =0;i<taille;i++) {
```

```
        s+=tab[i];
    }
    float moyenne = s/float(taille);
    return moyenne;
}

int main() {
    INITIALIZE_TESTS();

    BEGIN_SECTION("moyenne");
        BEGIN_TESTI("moyenne-nominal-1");
        double tab[] = { 1.0, 2.0, 3.0 };
        double r = moyenne(tab, 3);
        ASSERT_EQ_F(r, 2.0, 1e-6);
        END_TEST;

        BEGIN_TESTI("moyenne-nominal-2");
        double tab[] = { -2.0, 3.0, -5.0, 6.0 };
        double r = moyenne(tab, 4);
        ASSERT_EQ_F(r, 0.5, 1e-6);
        END_TEST;

        BEGIN_TESTI("moyenne-moins-long");
        double tab[] = { 0.1, 0.2, 0.3, 0.4 };
        double r = moyenne(tab, 3);
        ASSERT_EQ_F(r, 0.2, 1e-6);
        END_TEST;

        BEGIN_TESTI("moyenne-1");
        double tab[] = { 5.0 };
        double r = moyenne(tab, 1);
        ASSERT_EQ_F(r, 5.0, 1e-6);
        END_TEST;

        REPORT_TO_STDOUT;
    END_SECTION();

    return 0;
}
```

0.4.3 3. Division euclidienne

Spécifier (donner contrat et signature) et implanter une fonction qui réalise la division euclidienne de deux nombres entiers positifs. Cette fonction doit transmettre à l'appelant le *quotient* ***et*** *le reste*.

Écrivez un jeu de tests unitaires pour tester la fonction écrite.

[26]:
```c
//%cflags: -I .
//%ldflags: -L. -lt
#include "affichage.h"
#include "tests.h"

/** TODO : division euclidienne **/

int main() {
    INITIALIZE_TESTS();

    BEGIN_SECTION("division");
        /** TODO : tests unitaire **/
        BEGIN_TESTI("mon-test");
        FAIL("aucun test écrit");
        END_TEST;

        REPORT_TO_STDOUT;
    END_SECTION();

    return 0;
}
```

```
[Section division - RESULTS]
( 1/ 1) mon-test FAILURE (/tmp/tmpu7hj53y1.c:14)
        aucun test écrit
TOTAL: #tests: 1, #signaled: 1, #timedout: 0, #failures: 0 (success ratio:
0.00%)
```

0.4.4 4. Compteur de caractères (difficile)

Écrire un sous-programme `compter` qui compte le nombre d'occurrences de chaque lettre d'une chaîne de caractère et le communique à l'appelant, en ignorant la casse (majuscules/minuscules). Le résultat est stocké dans un tableau, où la première case correspond au nombre de A, la deuxième au nombre de B, etc.

Par exemple, appeler `compter` sur la chaîne `"AaAbbC"` donnera comme résultat `{ 3, 2, 1, 0, 0, ... }`.

Écrire ensuite un sous-programme `somme` qui réalise la somme *élément par élément* de deux tableaux d'entiers, en stockant le résultat dans le premier. Par exemple, la somme du tableau `{ 1, 2, 3, 4 }` avec le tableau `{ 1, 1, 1, 1 }` donne le tableau `{ 2, 3, 4, 5 }`.

Écrire enfin un sous-programme `compter_accum` qui, étant donné un tableau qui stocke des occurrence de lettres et une chaîne de caractères, met à jour le tableau en ajoutant les occurrences de ladite chaîne de caractère.

[4]:
```c
//%cflags: -I .
#include <stdio.h>
```

```c
#include "affichage.h"

void compter(/** À COMPLÉTER **/) {
    /** À COMPLÉTER **/
}

void somme(/** À COMPLÉTER **/) {
    /** À COMPLÉTER **/
}

void compter_accum(const char* chaine, /** À COMPLÉTER : TABLEAU D'OCCURRENCES
 ↪**/) {
    /** À COMPLÉTER **/
}

int main() {
    int occurrences[26]; // 26 lettres dans l'alphabet
    for (int i = 0; i < 26; i++) {
        occurrences[i] = 0; // On initialise
    }

    // Chaînes à compter (tableau de chaînes de caractères dont la dernière
 ↪case
    // est NULL pour délimiter la fin)
    char* chaines[] = {
        "Une premiere chaine",
        "Une deuxieme chaine",
        "Ici il y a des i à l'infini",
        "Le Z est une lettre rare",
        NULL
    };

    int i = 0;
    while (chaines[i] != NULL) {
        puts(chaines[i]);
        compter_accum(chaines[i], occurrences);
        i++;
    }

    for (int i = 0; i < 26; i++) {
        printf("#%c = %d, ", ((char) i) + 'A', occurrences[i]);
    }

    return 0;
}
```

```
/tmp/tmpky9j201z.c:13:85: error: expected declaration specifiers or ‘…’ before
```

```
‘)’ token
   13 | void compter_accum(const char* chaine, /** À COMPLÉTER : TABLEAU
D'OCCURRENCES **/) {
      |
^
/tmp/tmpky9j201z.c: In function ‘main’:
/tmp/tmpky9j201z.c:36:9: warning: implicit declaration of function
‘compter_accum’ [-Wimplicit-function-declaration]
   36 |         compter_accum(chaines[i], occurrences);
      |         ^~~~~~~~~~~~~
[C kernel] GCC exited with code 1, the executable will not be executed
```