TRAVAUX PRATIQUES II

Fonctions C

Utilisez l'éditeur de texte de votre choix ! Être à l'aise avec ses outils de travail, c'est important.

A Types de base

A.1 Types entiers

Les types entiers signés au programme sont :

Définition 1 : entiers signés

- `int` : le principal type d'entiers signés, à utiliser par défaut. Sa taille dépend de l'architecture de la machine sur laquelle le code est compilé. Il est cependant garanti qu'il fait au moins 2 octets.
- `int8_t`, `int32_t` et `int64_t` : types d'entiers signés dont la taille est fixée à respectivement 1, 4 et 8 octets. Leur taille est donc indépendante de l'architecture de la machine sur laquelle le code est compilé. Ils ne sont accessibles qu'en incluant `stdint.h`.

Les opérateurs d'arithmétiques sur ces types d'entiers et leurs priorités sont ceux de `int` présentées auTP I. De même pour les opérateurs de comparaison.

Pour afficher avec `printf` les types d'entiers signés vous pouvez utiliser `%d`, sauf pour `int_64t` qui necéssite `%ld` (le `l` signifie **l**ong).

1. Rappeler la définition de chacune de ces opérations arithmétique et leurs priorités.

Et pour les entiers non-signés :

Définition 2 : entiers non-signés

- `unsigned int` : le principal type d'entiers non-signés, à utiliser par défaut en cas de besoin d'entiers non-signés. Il est abrégeable en `unsigned`. Sa taille dépend de l'architecture de la machine sur laquelle le code est compilé. Il est cependant garanti qu'il fait au moins 2 octets.
- `uint8_t`, `uint32_t` et `uint64_t` : types d'entiers non-signés dont la taille est fixée à respectivement 1, 4 et 8 octets. Leur taille est donc indépendante de l'architecture de la machine sur laquelle le code est compilé. Ils ne sont accessibles qu'en incluant `stdint.h`.
- `size_t` : type d'entiers non-signés dont la taille est assez grande pour que le nombre total d'octets de la mémoire vive soit représentable dans ce type. Sa taille dépend donc de l'architecture de la machine sur laquelle le code est compilé. Il est cependant garanti qu'il fait au moins 2 octets.

Les opérateurs d'arithmétique sur ces types d'entiers et leurs priorités sont les mêmes que pour les entiers signés. Idem pour les opérateurs de comparaison.

Pour afficher avec `printf` les types d'entiers non-signés vous pouvez utiliser `%u`, sauf pour `uint_64t` qui necéssite `%lu`.

Si vous avez découvert C par vous même, vous connaissez probablement d'autres types d'entiers : ceux-ci sont hors-programme. En MP2I/MPI, dès lors que les deux types `int` ne suffisent plus, on cherchera à contrôler précisement la taille des entiers utilisés en utilisant les types de `stdint.h` présentés ci-dessus.

A.1.i : Caractères

C a un type pour manipuler les caractères ASCII étendu : le type `char`. Les versions récentes de C peuvent même manipuler des caractères unicode, comme nous le verrons dans un prochain TP ; cela n'est cependant pas possible aux simples `char`.

Définition 3

Un caractère ascii en C est écrit entre guillements simples : `char nom = 'C'`

C'est en fait un type d'entiers ! Les caractères sont encodés par leur code ASCII étendu, de manière non-signée (d'un point de vue représentation mémoire, `uint8_t` et `char` sont donc identiques). Cela a une conséquence intéressante :

Proposition 1

Les opérations applicables aux types entiers sont également applicables à `char` .

En particulier, `+1` (resp. `-1`) permet d'accéder au caractère ascii précédent (resp. suivant). La comparaison permet de comparer les codes ascii : `'a' < 'e'` est vrai.

A.2 Opérations entre opérandes de types entiers distincts

Les opérateurs sur les entiers peuvent être utilisés sur des opérandes de types d'entiers distincts. Par exemple, un programmeur insouciant peut tester une égalité entre un `unsigned` et un `int` . Dans ce cas, pour évaluer le résultat de l'opération, C va convertir l'un des opérandes dans le type de l'autre... ce qui peut entrainer des dépassements de capacité [1] ou d'autres comportements non-attendus. **On veillera toujours à ce que les opérandes d'un même opérateur soient de même type.**

Dans les rares cas où il y aura réellement besoin d'enfreindre cette règle, on fera la conversion soi-même, en amont de l'opération afin de la contrôler [2].

A.3 Booléens

Les booléens sont un ajout récent de C : ils ont été ajouté dans le standard de 1999 [3]. Il faut donc spécifier à `gcc` que l'on utilise un standard de C : on utilise pour cela l'option `--std=c17` pour sélectionner le dernier standard en date, de 2017. La ligne de compilation ressemble donc à :

```
gcc -o executable fichier0.c fichier 1.c ... fichiern.c --std=c17
```

De plus, il faut commencer son code C en incluant la librairie `stdbool.h` .

Définition 4

Le type des booléens en C est `bool` .
Un booléen ne peut prendre que deux valeurs : `true` (Vrai) ou `false` (Faux). On peut aussi assigner à une variable booléenne le résultat d'un opérateur de comparaison (qu'il concerne des booléens ou non), qui sera évalué soit à `true` soit à `false`.
Les booléens sont munis des opérateurs de comparaison suivants : `!` le NON logique, `&&` le ET logique, `||` le OU logique ; ainsi que `==` l'évaluation logique d'égalité et `!=` l'évaluation logique d'inégalité.

Ainsi, `bool sup = !(5 >= 3)` est évalué à `false` .
Une variable booléenne a pour taille 1 octet.

2. Un commentaire sur la taille des booléens ?

Proposition 2

L'ordre de priorité des opérations de comparaison sur les booléens est : `||` ≤ `&&` ≤ `==`, `!=` ≤ `!`

On évitera d'essayer d'afficher un booléen avec `printf`.

3. À quoi s'évalue `(1 <= 3) && ! true || (42+17 / 3 == 10 + 157 % (6*20))` ? Implémentez ce calcul pour vérifier

4. Ré-écrivez votre code `bissextile.c` du TP I en utilisant ces opérateurs de comparaisons. l'objectif est de n'utiliser plus qu'un seul `if` .

Théorème 1

C évaluera toujours l'opérande de gauche de `&&` (resp `||`) avant celui de droite. Si l'opérande de gauche s'évalue à `false` (resp. `true`), l'opérateur de droite n'est même pas évalué et le résultat de l'opération vaut `false` (resp. `true`).
On parle d'**évaluation paresseuse**.
On dit aussi que `false` (resp. `true`) est absorbant pour `&&` (resp. `||`).

1. On en parle bientôt en cours.
2. En réalité, les conversions faites par C ne sont pas mystérieuses mais bel et bien très strictement codifiées par la norme de C. Toutefois, ces détails-ci sont hors-programme - et il est plus simple de relire un code avec conversions explicites que de se souvenir des règles de conversion.
3. Toute remarque sur votre date de naissance sera sanctionnée proportionnellement au coup de vieux qu'elle me causera.

5. Considérez le code ci-dessous : pouvez-vous le simplifier ?

C
```c
unsigned x = 0;
... // du code qui modifie x
if (x >= 0 || un_test_complique(x)) {
    ...
}
```

A.4 Opérateur `sizeof`

Tous les types sont munis de l'opérateur `sizeof` :

Définition 5

L'opérateur `sizeof` calcule la taille en nombre d'octets d'un objet. Son résultat est donc de type `size_t`. Sa syntaxe est la suivante :

`sizeof identifiant`

Si l'identifiant est celui d'un type (comme `int`), le résultat sera la taille du type. Si c'est un identifiant de variable, le résultat sera la taille de la variable.

Attention. Contrairement à ce que l'on peut croire, ce n'est pas une fonction, mais bien un opérateur du langage C au même titre que `+` ou `=`. Dans le cadre des types au programme, cet opérateur a un comportement intéressant :

Proposition 3

Le compilateur est capable de calculer tous les `sizeof` à la compilation ; on dit qu'il les calcule *statiquement*. Il les remplace dans le programme créé par des constantes littérales. Par exemple, `sizeof uint32_t` sera remplacé par le compilateur par `4`.

A.5 Type d'absence

Parfois, on aura besoin de parler de l'absence (principalement de l'absence d'arguments ou de renvoi d'une fonction). Un type existe pour cela : le type `void`. C'est un type qui n'a aucun habitant, et qui sert uniquement à dénommer une absence.

B Tableaux

En C, les tableaux sont de longueur fixe et donnée à la déclaration, et ne peuvent contenir qu'un seul type de donnée lui aussi indiqué à la déclaration.

Définition 6

Pour déclarer un tableau d'objets de type `<type>` et contenant `N` éléments (**où `N` est une constante littérale entière** c'est à dire un entier écrit en tous chiffres), on utilise la syntaxe :

`<type> identifiant_du_tableau[N] ;`

On peut aussi initialiser le tableau en spécifiant ainsi la valeur initiale de chacune de ses `N` cases :

`<type> identifiant_du_tableau[N] = {`v_0`, `v_1`, …, `v_{N-1}` } ;`

⚠ Cette syntaxe `{`v_0`, `v_1`, …, `v_{N-1}` }` ne marche **que** lors de la délcraation d'un tableau. Ce que vous voyez entre ces acolades n'est *pas* un tableau, c'est une façon d'initialisser lors de sa déclaration un tableau.

Adressons quelques remarques que vous pourriez avoir :

— *« Mais enfin, dans <langage autrefois favori> on peut mettre plein de variables de types différents dans un tableau ! »*
C n'est pas aussi permissif. D'ailleurs, la quasi-totalité des langages ne le permettent pas. Votre langage autrefois préféré est sans doute Python, où les "tableaux" ne sont pas des tableaux. Sortez cette vision de votre esprit. À partir de maintenant, tableau égale données de même type. Nous verrons pendant l'année comment, à partir des tableaux, créer des structures de données plus complexes.

— *« Mais enfin, dans <langage autrefois favori> on peut changer la longueur des tableaux au fur et à mesure ! »*
Cf point précédent : ce ne sont pas des tableaux mais des structures plus avancées, possiblement construites à partir de tableaux. On étudiera ces constructions.

— *« Mais enfin, <site d'apprentissage du C> a dit que l'on peut faire des tableaux dont la longueur n'est pas une constante littérale mais une variable ! »*

En effet, il l'a dit. Il l'a dit.

On parle de tableaux de longueur variable (*Variable Length Array*, ou *VLA*). Ceux-ci ont été introduit dans C99 et ne sont plus obligatoirement supportés par les compilateurs depusi C11. S'ils ont l'air plus pratiques, ils créent du code très lent et lourd, modifient d'autres parties du fonctionnement de C (par exemple, `sizeof` ne peut pas être calculé à la compilation pour les VLA) et mennent trop facilement à des erreurs de débordement mémoire (ce qui n'est pas acceptable dans un langage utilisé par la plupart de programmeurs de circuits embarqués). Logiquement, les VLA de C seront donc proscrites en MP2I/MPI. Proscrites. PROSCRITES. PRO-SCRITES. **Je ne veux aucune VLA dans vos codes C**, zéro, niet, nada.

— *« Mais enfin, on peut utiliser les initialisateurs <de telle ou telle autre façon> ! »*

Oui. C'est hors-programme pour simplifier. Vous aurez déjà assez à maitriser comme ça.

Définition 7

Un tableau à `N` cases est indicé de `0` à `N-1` . Pour accéder à la `i`-ème valeur d'un tableau nommé `T`, on utilise l'opérateur d'indice `[]` . `T[i]` est le contenu de la case d'indice i case du tableau, c'est à dire la `(i+1)`-ème valeur.

Attention à ne jamais essayer d'accéder à un indice hors du tableau. Cela va au mieux renvoyer n'importe quoi, au pire faire crasher votre programme. Démonstration :

C : out-of-range.c

```c
#include <stdio.h>

void main() {
        int T[5] = {0,1,2,3,4} ;
        for (int i = 0; i <= 5; i = i+1) {
                printf("%d␣", T[i]);
        }
        printf("\n");
}
```

6. **a.** Le code `out_of_range.c` essaye d'accéder à une case mémoire qui n'appartient pas au tableau. Quelle(s) instruction(s) en est responsable(s) ?
   b. Sans corriger le bug, recopier le code et compilez. Lancez plusieurs fois le programme, que constatez-vous ?
7. Écrire un programme qui demande 5 entiers `int` à l'utilisateur et les stocke dans tableau, puis affiche les éléments du tableau qui sont strictement supérieurs à leur éventuel élément précédent et éventuel élément suivant. Testez-le.

Terminons cette section par une astuce : quand on a besoin de stocker dans un tableau des valeurs dont on ignore à l'avance la quantité, on peut créer un tableau plus grand que nécessaire et n'en utiliser qu'une partie. Par exemple, si on doit stocker au plus 31 valeurs, on crée un tableau à 31 cases quitte à ne pas en utiliser certaines.

Voici un exemple où l'on crée un tableau de taille 31 mais où l'on ne stocke finalement que 20 valeurs :

C

```c
int T[31];

/* ... du code qui fait des choses ...*/

int l = 20; // le nombres de cases réellement occupées après le code qui fait des choses
for (int i = 0; i < l; i = i+1) {
   printf("%d", i);
} // avec cette boucle on a parcouru et affiché tout le tableau
  // car les cases d'indices >= 20 ne sont pas occupées
```

C Fonctions

Définition 8

La syntaxe pour déclarer une fonction nommée `f` à `N` arguments en C est la suivante :

```
<type_de_sortie> f(<type_0> argument_0 , ... , <type_(N-1)> argument_(N-1) ) {
    ...
    instructions
}
```

L'instruction de renvoi est `return` . Si la fonction ne doit rien renvoyer, il suffit de ne pas lui donner d'instruction `return` ou n'utiliser que des `return;` (renvoi vide) : dans ces deux cas, il faut déclarer `void` comme type de sortie à la fonction.
Si une fonction ne prend pas d'arguments, il est de bon ton de lui donner `void` en argument plutôt que de laisser les parenthèses vides.

Exemple :

C : ppcm-naif.c

```c
#include <stdio.h>

/** Renvoie le plus petit multiple commun de p et q
 */
int ppcm_naif(int p, int q) {
        int candidat = p;
        if (candidat < q) {
                candidat = q;
        }
        while ( !(candidat % p == 0 && candidat % q == 0) ) {
                candidat = candidat +1;
        }
        return candidat;
}

void main(void) { // NB : ce prototype de fonction main n'est pas acceptable, cf suite du TP !
        int n = 0, m = 0;
        scanf("%d %d", &n, &m);
        int resultat = ppcm_naif(n,m);
        printf("ppcm(%d,%d) = %d\n", n, m, resultat);
        return;
}
```

Une syntaxe particulière est à noter : celle des tableaux. Pour spécifier un tableau d'entier `int` en argument, il faut écrire `int identifiant[]` dans la déclaration de la fonction.

C : appel-tableau.c

```c
#include <stdio.h>

/** Affiche le contenu d'un tableau de 5 int
 */
void affiche_tableau(int T[5]) {
        for (int i = 0; i < 5; i = i+1) {
                printf("%d\n", T[i]);
        }
        return;
}

void main(void) { // NB : ce prototype de fonction main n'est pas acceptable, cf suite du TP !
        int T[5] = {-3, 5, -7, 82, 63};
        printf("Voici les éléments du tableau :\n");
        affiche_tableau(T);
        return;
}
```

Théorème 2

Les appels des arguments des fonctions C sont toujours des appels par valeur.
Toutefois, le *contenu* d'un tableau se comporte *comme si* le tableau était appelé par référence.

8. Coder en C un exemple illustrant la différence entre un entier appelé par valeur et le contenu d'un tableau appelé *comme si* par référence. Le tester.

On verra plus tard dans l'année comment renvoyer plrs valeurs à la fois.

C.1 Cas particulier de `main`

La fonction `main` est particulière. Le flot du programme est le flot du main [4]. A priori, le `main` semble donc ne rien pouvoir renvoyer puisque personne ne l'a appelé… il n'en est rien !

Jusqu'à présent, nous utilisions un `main` renvoyant `void` : c'est fini. Le `main` doit avoir une valeur de retour car celle-ci n'est pas du tout superflue ! Elle est communiquée au terminal qui en conclut si le programme s'est correctement exécuté ou a rencontré une erreur et a dû s'arrêter. Dans la librairie `stdlib.h` (*StanDard LIBrary* [5]) se trouve deux constantes prédéfinies : `EXIT_SUCCESS` et `EXIT_FAILURE` . Un programme C qui s'exécute correctement doit avoir la première renvoyée par son `main` . Si le code détecte une erreur, il doit renvoyer la seconde [6]. **Dorénavant, tous vos `main` devront renvoyer `EXIT_SUCCESS` ou `EXIT_FAILURE` .**

Enfin, la fonction d'entrée peut prendre des arguments. Nous les découvrirons plus tard dans l'année. Pour l'instant, `void` est le seul argument de `main` .

9.
- **a.** Implémentez une fonction `min` (qui n'est pas le `main`) qui prend en argument deux `int` et renvoie leur minimum. Testez-la.
- **b.** Utilisez $\max(x, y) = -\min(-x, -y)$ pour implémenter une fonction `max` . Testez-la.
- **c.** Idem pour des `unsigned` .

D À vos claviers !

Il est temps de pratiquer ! Vous ferez bien attention à : compiler avec les bonnes options et sans avertissement, la valeur de retour de `main` , utiliser les fonctions pour améliorer la lisibilité du code, prévenir les dépassements de capacité, ne pas essayer d'accéder à un indice illégal d'un tableau, et à tester vos fonctions.

Si vous ne le faisiez pas déjà, vous devriez également commencer à créer différents fichiers C pour différentes questions quand cela vous semblera pertinent.

10.
- **a.** Écrire une fonction `int somme(int T[10])` qui prend en argument un tableau de 10 `int` et renvoie la somme de ses éléments.
- **b.** Écrire une fonction `int maxi(int T[10])` qui prend en argument un tableau de 10 `int` et renvoie son élément maximal.

11.
- **a.** Écrire une fonction `void affiche(int T[10])` qui affiche le contenu d'un tableau de 10 `int` .
- **b.** Écrire une fonction `void copie(int dest[10], int src[10])` qui copie le contenu de `src` dans `dest` .

12. La suite de Fibonacci est définie par $u_0 = 0, u_1 = 1$ et $\forall n \in \mathbb{N}, u_{n+2} = u_{n+1} + u_n$.
- **a.** Écrire une fonction prenant en argument n et renvoyant u_n . (Si besoin, on pourra supposer $n < 100$) .
- **b.** Que vaut u_{50} ?

13. On définit le miroir (ou *reverse*) d'un tableau comme étant le même tableau mais dont les éléments respectifs sont ordonnées de droite à gauche au lieu de gauche à droite. Ainsi, $[0; 1; 2]$ est le miroir de $[2; 1; 0]$.
- **a.** Écrire une fonction `void miroir(unsigned resultat[10], unsigned tableau[10])` qui stocke dans `resultat` le miroir de `tableau` .
- **b.** Refaire la question en ajoutant un paramètre $\ell : 1 \leq \ell \leq 10$ est la longueur du tableau réel qui est donc stocké entre les indices 0 et $\ell - 1$ de `tableau` . Dans `resultat` , le miroir devra également être stocké entre ces indices.

14. Dans cette question les entiers utilisés doivent pouvoir représenter $[\![-4.10^9; +4.10^9]\!]$.

4. Cela est bien entendu changeable en donnant les bonnes options au compilateur… mais pourquoi chercher compliqué quand on peut rester simple ?
5. Ce n'est effectivement pas un nom de librairie très explicite…
6. En réalité, on peut renvoyer différentes sorties d'erreurs pour indiquer le type d'erreur qui s'est produit… gardons ces formalités pour votre future vie professionnelle, voulez-vous ?

a. Écrire une fonction qui prend en argument un entier c et un tableau de n entiers (où $n \geq 10$), et renvoie un booléen indiquant si c est une des valeurs du tableau.

b. Si vous supposez de plus que les valeurs du tableau sont triées par ordre croissant, pouvez-vous améliorer la fonction précédente ? Pourquoi ?

15. Un nombre parfait est un entier positif égal à la moitié de la somme de ses diviseurs positifs. Écrire un programme qui affiche si un nombre est parfait ou non.

16. Écrire un programme qui prend en argument un entier $n \leq 9$ et affiche les $n+1$ premières lignes du triangle de Pascal [7]. On pourra utiliser deux tableaux : un pour stocker la ligne actuellement calculée du tableau, et un pour stocker la ligne précédente. La fonction `copie` pourra être utile. Par exemple, `Pascal(4)` devra afficher :

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

17. Dans cette question, on représentera l'écriture en base b non-signés sur N bits d'un nombre par un tableau de `int8_t` de longueur N. Les valeurs des cases du tableau sont donc des chiffres de la base b, c'est à dire des entiers de $[\![0; b-1]\!]$. Le chiffre de poids fort est à gauche.

a. Comment serait représenté $^2\overline{10110}$? Et $^{16}\overline{f03a}$?

b. Combien de chiffres hexadécimaux faut-il pour représenter n'importe quel entier non-signé 32-bits (binaire) ?

c. Écrire une fonction `void bin_vers_hexa(int8_t dec_hexa[8], int8_t dec_bin[32])` qui implémente la conversion binaire vers hexadécimal. Après l'exécution de la fonction, `dec_hexa` doit contenir l'écriture hexadécimale de l'entier dont l'écriture binaire est `dec_bin` .

NB : pour tester votre fonction, vous pourrez utiliser `%x` qui lit un entier e l'affichage en hexadécimal. Ainsi, `printf("%x", 9)` affiche 9 mais `printf("%x", 15)` affiche `f` .

E Excursion dans le terminal

Avec le TP 0, vous avez toutes les commandes nécessaires à votre survie dans un terminal. Amusons-nous avec quelques autres.

18. **a.** Que renvoie la commande `whoami` (*qui suis-je* dans la lague de Shakespeare) ?

b. Et la commande `hostname` (*nom de l'hébergeur* dans la lague de Shakespeare) ?

c. Et la commande `date` [8] ?

19. Que fait la commande `ping perdu.com` ? (Rappel : Ctrl+C force un programme qui ne termine pas à s'arrêter.) Que fait l'option `-c` de `ping` ? Utilisez-la.

20. Que fait la commande `top` ? (`q` pour quitter.)

21. Les deux prochaines commandes sont un peu dures à lire pour un oeil non-entrainé, je vous l'accorde.

a. Que fait la commande `lscpu` (*LiSt Central Processing Units*) ?

b. Et `lsmem` (*LiSt MEMory*) ?

22. Continuez Gameshell.

23. Bonus : sur votre machine personnelle, installez `cmatrix` , `nyancat` et `sl` . Frimez avec.

7. Rappel mathématique : pour $n \in \mathbb{N}$ et $0 \leq k \leq n$, $\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} \text{ si } 0 < k < n \\ 1 \text{ sinon} \end{cases}$. Le triangle de Pascal est le "triangle" dont la première ligne est $\binom{0}{0}$, la deuxième $\binom{1}{0}, \binom{1}{1}$, la troisième $\binom{2}{0}, \binom{2}{1}, \binom{2}{2}$, etc.

8. Je pense pouvoir m'épargner la traduction de celui-ci...