Perfectionnement à la programmation en C

Livret d'exercices

L2 Informatique 2021-2022

Table des matières

Chapitre 1

Bases de la programmation

Exercice 1.1. (Structures conditionnelles)

1. Rappeler quelles sont les structures conditionnelles et expliquer leur utilité.
2. Expliquer dans quelles circonstances il est préférable d'utiliser une structure en `if` plutôt qu'une structure en `switch`. Donner des exemples pour illustrer le propos.
3. Rappeler comment fonctionne l'opérateur conditionnel ternaire `C?X:Y` et énoncer sa principale différence vis-à-vis des structures conditionnelles.

Exercice 1.2. (Structures de boucle)

1. Rappeler quelles sont les structures de boucle et expliquer leur utilité.
2. Expliquer dans quelles circonstances il est préférable d'utiliser une boucle `for` plutôt qu'une boucle `while` et réciproquement. Donner des exemples pour illustrer le propos.

Exercice 1.3. (Portée lexicale)

1. Rappeler ce qu'est un *bloc*.
2. Rappeler ce qu'est la *portée lexicale* d'une variable.
3. Déterminer la portée lexicale de chaque variable déclarée dans les suite d'instructions ci-dessous.

(a)

```
{
    int a, b, c;

    a = 10;
    c = 8;
    {
        char c;
        c = 'a';
    }
    {
```

```
        int c;
        c = 16;
    }
    a += 1;
    b = 5;
}
```

(b)
```
int x;
{
    int y;
    float z;
    printf("%d", x);
    {
        float x;
        x = 3.4;
    }
    {
        char y;
        y = 'F';
    }
}
```

(c)
```
int a, b;

a = 0;
{
    int a;
    float b;
    a = 3;
    {
        b = 10;
        {
            int b;
            b = 3;
            a += 1;
        }
    }
}
```

Exercice 1.4. (Conversions)

1. Convertir en binaire les valeurs suivantes exprimées en base dix. Les résultats sont à donner sur 16 bits.

   (a) 0 ; (b) 1 ; (c) 10 ; (d) 2224 ; (e) 3200 ; (f) −0 ; (g) −1 ; (h) −10 ; (i) −16 ; (j) −32 ; (k) −2224 ; (l) −3200.

2. Convertir en hexadécimal les valeurs suivantes exprimées en base deux. Les résultats sont à donner sur huit chiffres hexadécimaux.

   (a) `0011101010000001` ; (b) `1` ; (c) `11111111001001011111` ; (d) `00010000`.

3. Convertir en binaire les valeurs suivantes exprimées en hexadécimal. Les résultats sont à donner sur 16 bits.

   (a) `1010` ; (b) `ABC` ; (c) `F0B` ; (d) `123A`.

Exercice 1.5. (Suite de Syracuse)

La *suite de Syracuse* est une suite $\left(s_i^{(n)}\right)_{i\geqslant 0}$ d'entiers dépendant d'un paramètre n définie de la manière suivante :

$$s_i^{(n)} := \begin{cases} n & \text{si } i = 0, \\ \dfrac{1}{2}\, s_{i-1}^{(n)} & \text{si } s_{i-1}^{(n)} \text{ est pair}, \\ 3\, s_{i-1}^{(n)} + 1 & \text{sinon.} \end{cases}$$

Par exemple, la suite de Syracuse avec $n = 12$ comme paramètre commence par

$$s_0^{(12)} = 12, 6, 3, 10, 5, 16, 8, 4, 2, 1, s_{10}^{(12)} = 4, 2, 1, 4, 2, 1.$$

Une conjecture célèbre énonce que pour tout entier $n \geqslant 1$, il existe un entier $i \geqslant 0$ tel que $S_i = 1$. Le statut de cette assertion demeure encore inconnu aujourd'hui (2020).

1. Écrire un programme qui demande à l'utilisateur d'entrer au clavier un entier `n` et qui affiche les éléments de la suite $\left(s_i^{(n)}\right)_{i\geqslant 0}$ et s'arrête dès qu'un terme est égal à 1.
2. Expliquer si le programme précédent est un algorithme.

Chapitre 2

Expressions et effets secondaires

Exercice 2.1. (Instructions à effets secondaires)

1. Rappeler ce qu'est une *expression à effet secondaire*.
2. Déterminer si les instructions suivantes sont à effet secondaire :

(a)
```
2 + (8 * 2);
```

(b)
```
printf("Bonjour\n");
```

(c)
```
int a;
```

(d)
```
int a = 2;
```

(e)
```
a = 2 + (8 * 2);
```

(f)
```
a == 2 + (8 * 2);
```

(g)
```
a * 2 + (8 * 2);
```

(h)
```
if (a == 17) {a;}
```

(i)
```
if (--a == 16) {a;}
```

(j)
```
a++;
```

(k)
```
a + 1;
```

(l)
```
while (1) a + 1;
```

(m)
```
while (1) a += 1;
```

(n)
```
return 1;
```

(o)
```
return a;
```

(p)
```
return a + 1;
```

(q)
```
p = malloc(64);
```

(r)
```
malloc(64);
```

Exercice 2.2. (Fonctions à effets secondaires)

1. Rappeler ce qu'est une *fonction à effet secondaire*.
2. Déterminer si les fonctions suivantes sont à effet secondaire :

(a)
```
int addition_1(int a, int b) {
    return a + b;

}
```

(b)

```c
int addition_2(int a, int b) {
    int res;

    res = a + b;
    return res;
}
```

(c)

```c
void addition_3(int a, int b,
        int *res) {

    *res = a + b;
}
```

(d)

```c
int somme(int *tab, int n) {
    int i, res;

    res = 0;
    for (i = 0; i < n; i++)
        res += tab[i];
    return res;
}
```

(e)

```c
void afficher(int *tab, int n) {
    int i;

    for (i = 0; i < n; i++)
        printf("%d␣", tab[i]);
}
```

(f)

```c
void echanger(int *x, int *y) {
    int tmp;


    tmp = *x;
    *x = *y;
    *y = tmp;
}
```

(g)

```c
int nb_appels = 0;

int fct_1(int n) {
    nb_appels++;
    return n + 1;
}
```

(h)

```c
int nb_appels = 0;

int fct_2(int n) {
    if (nb_appels == 0)
        return 0;
    else
        return n + 1;
}
```

(i)

```c
int remplacer(char *tab, char a,
        char c) {
    int i, nb;
    nb = 0;
    i = 0;
    while (tab[i] != '\0') {
        if (tab[i] == a) {
            tab[i] = c;
            nb += 1;
        }
        i += 1;
    }
    return nb;
}
```

Exercice 2.3. (R-values et L-values)

1. Rappeler ce qu'est une *R-value*.

2. Rappeler ce qu'est une *L-value*.

3. On suppose que `num` est une variable de `int`, `ptr` est un pointeur sur une variable de type `int`, `fct_1` est une fonction paramétrée par un entier et de type de retour `int` et `fct_2` est une fonction paramétrée par un pointeur sur un entier et de type de retour `int`.

   Dans les expressions suivantes, pour chaque sous-expression qui les constitue, déterminer s'il s'agit de R-values et/ou de L-values :

   (a)
   ```
   23
   ```

   (b)
   ```
   num
   ```

   (c)
   ```
   ptr
   ```

   (d)
   ```
   *ptr
   ```

   (e)
   ```
   num = num + 1;
   ```

   (f)
   ```
   *ptr = num;
   ```

   (g)
   ```
   ptr = ptr + 2;
   ```

   (h)
   ```
   *(ptr + 2)
   ```

   (i)
   ```
   *(ptr + 2) = 3;
   ```

   (j)
   ```
   &num
   ```

   (k)
   ```
   ptr = &num;
   ```

   (l)
   ```
   *&ptr
   ```

   (m)
   ```
   fct_1(num)
   ```

   (n)
   ```
   fct_2(&num)
   ```

   (o)
   ```
   fct_2(ptr)
   ```

Chapitre 3

Fonctions et pile d'appel

Exercice 3.1. (Déclaration de fonctions)

1. Identifier et donner les différentes parties (identificateur, signature, type de retour, instructions) de la fonction suivante :

```c
float aire(int a, int b) {
    return (.0 + a * b) / 2;
}
```

2. **Déclarer** une fonction testant la primalité d'un entier.

3. Déclarer une fonction paramétrée par une chaîne de caractères et deux caractères. Cette fonction remplace les occurrences du 1er caractère par le 2e dans la chaîne de caractères.

4. Déclarer une fonction qui joue une note dans le terminal. La fonction accepte comme arguments la fréquence en Hz de la note à jouer ainsi que sa durée en ms.

Exercice 3.2. (Programme douteux)

```c
#include <stdio.h>

int etrange(int *n, int m) {
    *n += m;
    return *n + 1;
}

int main() {
    int n;
    n = 0;
    n = etrange(&n, etrange(&n, 10));
    printf("%d\n", n);
    return 0;
}
```

Expliquer ce qu'affiche le programme ci-contre et en quoi il n'est pas recommandable.

Indication : tenter de suivre l'exécution du programme pas à pas en présentant l'évolution de la valeur de la variable `n` *en fonction du temps.*

Donner tous les possibilités d'évolution. Isoler l'instruction qui pose problème et tenter de dégager une règle générale qui fait que toute instruction similaire provoque un problème.

Exercice 3.3. (Pile et fonctions)

1. Schématiser l'état de la pile à chaque instant de l'exécution de l'instruction

```
afficher(7, 2);
```

avec les définitions suivantes :

```
void afficher(int larg, int haut) {
    int i;
    for (i = 1 ; i <= haut ; ++i) {
        afficher_ligne(larg);
        printf("\n");
    }
}
```

```


void afficher_ligne(int larg) {
    int i;
    for (i = 1 ; i <= larg ; ++i)
        printf("*");
}
```

2. Dessiner l'arbre des appels récursifs puis schématiser l'état de la pile à chaque instant de l'exécution de l'instruction

```
tribo(5);
```

avec la définitions suivante :

```
int tribo(int n) {
    if (n <= 2)
        return n;
    return tribo(n - 1) + tribo(n - 2) + tribo(n - 3);
}
```

3. Dessiner l'arbre des appels récursifs puis schématiser l'état de la pile à chaque instant de l'exécution de l'instruction

```
flip(5);
```

avec les définitions suivantes :

```
void flip(int n) {
    printf("flip␣%d\n", n);
    if (n >= 1)
        flop(n - 1);
}
```

```
void flop(int n) {
    printf("flop␣%d\n", n);
    if (n >= 1)
        flip(n - 1);
}
```

Chapitre 4

Entrées et sorties

Exercice 4.1. (Boucles et affichage de motifs)

Écrire, en utilisant judicieusement des boucles `for`, `while` ou encore `do while`, les fonctions suivantes.

1. ```c
   void afficher_drapeau(int n);
   ```

   qui produit la sortie suivante (donnée ici dans le cas n = 4) :

   ```
   ----
   *---
   **--
   ***-
   ****
   ```

2. ```c
   void afficher_damier(int n);
   ```

   qui produit la sortie suivante (donnée ici dans le cas n = 4) :

   ```
   *-*-
   -*-*
   *-*-
   -*-*
   ```

3. ```c
   void afficher_triangle(int n);
   ```

   qui produit la sortie suivante (donnée ici dans le cas n = 6) :

   ```
   *
   **
   ****
   *******
   **********
   ***************
   ```
   ```

Exercice 4.2. (Écriture formatée)

Écrire un programme `Occurrences.c` qui accepte en paramètre des entiers en base dix (en nombre arbitraire mais au moins un) et affiche, ligne par ligne pour chaque nombre entré, son nombre d'occurrences de manière justifiée. Par exemple, la commande

```
Occurrences 211 2 2 1 211 1 1 1 44 211 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1
```

affiche

```
211  3
  2 16
  1  6
 44  1
```

Ceci indique qu'il y a entre autres seize occurrences d'arguments égaux à 2. Respecter la mise en page et l'alignement de l'exemple.

Exercice 4.3. (Miroir)

Écrire un programme qui lit des chaînes de caractères de longueur au plus 4 (la chaîne lue est tronquée le cas échéant) et qui les affiche de la plus récente à la plus ancienne une fois la chaîne `"fin"` saisie. Par exemple, sur l'entrée de

```
Bien le bonjour camarade ! fin,
```

le programme affiche

```
! cama bonj le Bien.
```

Exercice 4.4. (Fichier de configuration)

Écrire un programme qui lit un fichier de configuration (dont le nom est passé en argument au programme) qui renseigne sur une largeur, une hauteur et un caractère au format

```
largeur=L
hauteur=H
caractere=C
```

où `L` et `H` sont des entiers positifs non nuls, et `C` est un caractère affichage. Le programme affiche ensuite sur la sortie standard un rectangle à `L` colonnes et `H` lignes, fait de caractères `C`. Si le fichier de configuration n'est pas au bon format, le programme doit afficher un message d'erreur explicite renseignant sur la première erreur rencontrée dans le fichier de configuration.

Chapitre 5

Gestion d'erreurs et assertions

Exercice 5.1. (Déclarations et pré-assertions)

Pour chaque tâche décrite, proposer une **déclaration de fonction** y répondant, ainsi qu'une **liste de pré-assertions** adéquate. Il est important de s'interroger sur les entrées de chaque problème ainsi que sur sa (ou ses) sortie(s). Pour obtenir des pré-assertions pertinentes, il est nécessaire de s'interroger sur les entrées qui posent problème.

1. Le calcul du quotient de deux entiers ;
2. le calcul du reste de la division euclidienne de deux entiers a et b, compris entre 0 et $b - 1$ (*Attention : l'opérateur modulo % du C peut renvoyer un reste négatif.*) ;
3. le calcul du n^{e} nombre de Fibonacci ;
4. le calcul de l'image en x d'un polynôme du second degré $a_0 + a_1 x + a_2 x^2$;
5. l'affichage d'une lettre a minuscule suivie de sa version en majuscule ;
6. l'affichage d'un rectangle composé de n lignes et de m colonnes d'étoiles ;
7. l'échange des valeurs de deux entiers ;
8. le calcul du nombre de cases contenant des entiers positifs dans un tableau ;
9. le calcul de la moyenne des valeurs contenues dans un tableau de flottants ;
10. le calcul du maximum des valeurs contenues dans un tableau d'entiers ;
11. le calcul de la distance d'un point du plan par rapport à l'origine ;
12. l'initialisation de chaque case d'un tableau à deux dimensions par une valeur a ;
13. le calcul du nombre de voyelles dans une chaîne de caractères contenant des chiffres, des espaces ou des lettres ;
14. la création d'une chaîne de caractères obtenue en remplaçant toutes les majuscules par des minuscules d'une chaîne de caractères donnée en entrée ;

15. la création d'une chaîne de caractères obtenue en remplaçant tout caractère alphabétique c par un caractère alphabétique c' d'une chaîne de caractères donnée en entrée.

Exercice 5.2. (Ajout de mécanismes de gestion d'erreurs)

Réécrire les fonctions suivantes de sorte à les munir d'un mécanisme de gestion d'erreurs. Pour chaque nouvelle fonction ainsi écrite, rédiger une documentation appropriée.

1.
```c
void doubler() {
    int entree;

    scanf("%d", &entree);
    printf("%d\n", 2 * entree);
}
```

2.
```c
void repeter_affichage(char *chaine,
        int nombre) {
    int i;

    assert(chaine != NULL);
    assert(nombre >= 0);

    for (i = 0 ; i < nombre ; ++i)
        printf("%s\n", chaine);
}
```

3.
```c
int premiere_position(char *chaine,
        char lettre) {
    int i;

    assert(chaine != NULL);

    i = 0;
    while (chaine[i] != '\0') {
        if (chaine[i] == lettre)
            return i;
        i += 1;
    }
    return -1;
}
```

4.
```c
int *allouer_tab_int(int taille) {
    assert(taille >= 1);

    return (int *)
        malloc(sizeof(int) * taille);
}
```

5.
```c
void copier_pointeur(char *source,
        char **cible) {
    assert(source != NULL);
    assert(cible != NULL);

    *cible = (char *)
        malloc(sizeof(char));

    **cible = *source;
}
```

Exercice 5.3. (Écriture de fonctions sûres)

Pour chaque fonction ci-dessous, la définir, la munir de pré-assertions, imaginer les cas d'erreur possibles et mettre en place un mécanisme de gestion d'erreurs.

1. `calculer_moyenne` paramétrée par un tableau de notes comprises entre `0` et `20`, calculant la moyenne des notes ;
2. `lettre_frequente` paramétrée par une chaîne de caractères, calculant sa lettre strictement plus fréquente ;
3. `dessiner_etoiles` paramétrée par un tableau d'entiers et dessinant, pour chaque entier a du tableau lu de la droite vers la gauche, une nouvelle ligne de a occurrences de `'*'` sur la sortie standard ;
4. `lire_points_isobarycentre` qui lit sur l'entrée standard deux points du plan cartésien au format `X Y` et affiche sur la sortie standard leur isobarycentre.

Exercice 5.4. (Utilisation de fonctions à gestion d'erreurs)

1. Écrire une fonction `est_nom` paramétrée par une chaîne de caractères qui teste si celle-ci est constituée uniquement de caractères alphabétiques.
2. Écrire une fonction `demander_nom` qui lit une chaîne de caractères sur l'entrée standard. Celle-ci doit gérer les erreurs qui peuvent survenir (une chaîne de caractère qui n'est pas un nom est entrée ou bien une erreur provient de la fonction de lecture sur l'entrée standard).
3. Utiliser la fonction `demander_nom` dans un programme complet pour demander un nom à un utilisateur et l'afficher sur la sortie standard. Lors de l'exécution, le programme demande un nom tant que l'utilisateur ne rentre pas un nom valide.

Exercice 5.5. (Sécurisation des fonctions précédentes)

Reprendre toutes les fonctions écrites dans les exercices précédents et les munir d'un mécanisme de gestion d'erreurs lorsque cela est approprié. On n'oubliera pas de les munir également de pré-assertions adéquates.

Chapitre 6

Modularisation

Exercice 6.1. (Projets et graphes d'inclusions)

On considère un projet constitué de cinq modules `A`, `B`, `C`, `D` et `E`. Ces modules sont utilisés dans un fichier `Main.c` qui contient la fonction `main`. Dans ce projet figurent les inclusions suivantes :

- `A.h` inclut `B.h`, `C.h` et `E.h`;
- `A.c` inclut `D.h`;
- `B.h` inclut `C.h`;
- `D.h` inclut `A.h`;
- `E.c` inclut `D.h`;
- `Main.c` inclut `A.h` et `E.h`.

1. Donner le nombre de fichiers qui constituent le projet.
2. Donner la liste des commandes qui permettent de compiler le projet.
3. Dire si l'ordre des commandes de compilation est important et justifier pourquoi.
4. Tracer le graphe d'inclusions (étendues) du projet.
5. Expliquer, graphe d'inclusions (étendues) à l'appui, si le projet est bien structuré. Mettre en évidence les éventuels problèmes qu'il contient.
6. Supposons (uniquement pour cette question) que `B.h` inclut `D.h`. Expliquer, graphe d'inclusions (étendues) à l'appui, si le projet est bien structuré. Mettre en évidence les éventuels problèmes qu'il contient.
7. Supposons (uniquement pour cette question) que `B.c` inclut `D.h`. Expliquer, graphe d'inclusions (étendues) à l'appui, si le projet est bien structuré. Mettre en évidence les éventuels problèmes qu'il contient.
8. Supposons (uniquement pour cette question) que `B.h` inclut `D.h`. On suppose également que dans l'en-tête du module `B` est déclaré un type `B_type` et que dans l'en-tête du module `A` est déclarée une fonction de prototype `void a_fct(B_type x)`. Expliquer pourquoi la commande `gcc -c B.c` provoque une erreur. Expliquer pourquoi la commande `gcc -c A.c` produit bien un fichier objet.

9. Écrire un `Makefile` complet pour compiler le projet.

Exercice 6.2. (Découpage d'un projet en modules : le Gomoku)

Le but de cet exercice est de réaliser l'analyse d'un petit projet en le découpant en modules et en écrivant ses fichiers d'en-tête. On ne donnera pas l'implantation des modules.

Le projet consiste à réaliser un jeu de *Gomoku*[1]. Une partie se joue à deux joueurs sur un plateau de 19 × 19 cases. À chaque tour de jeu, le joueur qui a le trait (Blanc ou Noir) choisit une case du plateau de jeu et y pose un pion de sa couleur. Le premier joueur qui réussit à aligner 5 pions de sa couleur gagne la partie. Si le plateau est rempli de pions sans aucune configuration de gain, la partie est déclarée nulle. Le joueur Noir commence toujours la partie.

Au lancement du programme, il est de possible de :

— commencer une nouvelle partie ;

— charger une partie depuis un fichier ;

— quitter le programme.

Au cours de la partie, il est possible de :

— jouer un coup (par le joueur qui possède le trait) ;

— abandonner la partie et céder ainsi la victoire à son adversaire ;

— sauvegarder la partie en cours (sans quitter le programme, la partie continue) ;

— quitter la partie en cours et revenir au menu de lancement du programme.

Lorsque la partie s'arrête (gain d'un joueur ou partie nulle), le résultat est affiché, suivi du menu de lancement du programme.

Les parties sont sauvegardées dans un fichier selon un format choisi par le programmeur. Ce format doit permettre de représenter toutes les informations d'une partie pour que, à partir d'un fichier de sauvegarde, le programme puisse entièrement restaurer la partie qu'il représente. De plus, l'interaction avec les utilisateurs se fait par le biais de l'entrée et de la sortie standard.

1. Découper ce projet en modules. Ce découpage doit permettre de le faire facilement évoluer. Expliquer comment prendre en compte les variantes suivantes :
   — changement du format des fichiers de sauvegarde ;
   — changement des règles (6 pions alignés pour le gain au lieu de 5, taille du plateau 21 × 21 au lieu de 19 × 19, trois joueurs au lieu de deux) ;

1. En réalité, il s'agit ici d'une variante du Gomoku.

— affichage graphique à la place d'un affichage sur la sortie standard ;

— ajout d'un mode de jeu humain *vs* machine.

Pour chacune des ces quatre évolutions, expliquer leur impact sur l'ensemble des modules du projet.

2. Déterminer les types et les prototypes des fonctions que chaque module doit contenir. En déduire les en-têtes des modules du projet.

Exercice 6.3. (Modélisation d'un projet : polynômes)

On souhaite réaliser un projet qui permet de manipuler des polynômes sur une variable x et à coefficients flottants (`double`). Plus précisément, le programme doit pouvoir :

— lire un polynôme à partir d'un fichier, calculer sa dérivée par rapport à x et afficher le résultat sur la sortie standard ;

— lire un polynôme à partir d'un fichier, calculer sa n^{e} puissance et afficher le résultat sur la sortie standard ;

— lire deux polynômes à partir d'un fichier, calculer leur somme et afficher le résultat sur la sortie standard ;

— lire deux polynômes à partir d'un fichier, calculer leur produit et afficher le résultat sur la sortie standard ;

— lire un polynôme à partir d'un fichier et afficher dans une fenêtre graphique le graphe du polynôme dans un intervalle d'abscisse et d'ordonnée spécifié.

L'utilisateur peut choisir l'une ou l'autre de ces fonctionnalités par le biais d'une option (respectivement `-d`, `-p N`, `-s`, `-m`, `-g`). Le dernier paramètre de l'exécutable est le nom du fichier contenant le(s) polynôme(s) à traiter. Dans le cas où l'option `-g` est choisie, après le nom du fichier figurent les paramètres `X_MIN`, `X_MAX`, `Y_MIN` et `Y_MAX` qui définissent les dimensions du repère du graphique.

Le format des fichiers contenant les polynômes est spécifié par l'exemple suivant : le polynôme $-3 + 2.5x^5 - 4.7x^{11}$ est codé par

```
-3 + 2.5x^5 - 4.7x^11.
```

1. Découper ce projet en modules.
2. Pour chacun des modules, donner les fichiers d'en-tête au complet (directives du pré-processeur, définitions de types, déclarations de fonctions).
3. Dessiner le graphe d'inclusions du projet.
4. Écrire le code de la fonction `main` du projet.
5. Écrire un `Makefile` complet pour compiler le projet.

Chapitre 7

Manipulation de la mémoire

Exercice 7.1. (Opérations sur les pointeurs)

Pour chacune des suites d'instructions suivantes, donner ligne par ligne l'état des variables déclarées. Présenter la solution sous forme de tableau et dessiner, étape par étape, l'état de la mémoire.

(a)
```c
int a, b;
int *p;

a = 10;
p = &a;
b = *p + 2;
*p = *p + 4;
a = *p;
```

(b)
```c
int a, b;
int *p1, *p2;

a = 3;
p2 = &b;
*p2 = a + 1;
p1 = p2;
*p1 = 5;
```

(c)
```c
int tab[10];
int *p1, *p2;

tab[0] = 4;
tab[3] = 2;
p1 = &tab[0];
tab[1] = *p1;
p2 = p1 + 3;
tab[2] = *p2;
```

(d)
```c
int a, b;
int *p1;
int **p2;

p1 = &a;
p2 = &p1;
**p2 = 3;
*p2 = &b;
*p1 = 4;
```

Exercice 7.2. (Tableaux statiques)

On souhaite manipuler une fonction paramétrée par un entier `n` qui renvoie un pointeur sur un tableau de `n` entiers initialisés à `0`.

```c
int *creer_tableau(int n) {
    int tab[n];
    int i;

    for (i = 0 ; i < n ; ++i)
        tab[i] = 0;
    return tab;
}
```

Expliquer pourquoi la fonction ci-contre ne répond pas à cette spécification en donnant deux raisons bien précises. En supposant que cette fonction est acceptée par le compilateur, expliquer aussi ce qu'il se passe lorsqu'elle est appelée (faire des dessins de mémoire).

Indice : tenir compte du fait que la taille du tableau statique n'est pas connue à la compilation mais seulement à l'exécution.

Exercice 7.3. (Tableaux dynamiques à une dimension)

1. Écrire une fonction `creer_tab` paramétrée par un entier `n` et qui renvoie un pointeur sur un tableau de `n` entiers initialisés à `0`.
2. Écrire une fonction `detruire_tab` paramétrée par un pointeur `tab` sur un tableau d'entiers et qui libère la place mémoire occupée par `tab`.

Exercice 7.4. (Tableaux dynamiques à deux dimensions)

1. Écrire une fonction `creer_tab_2d` paramétrée par des entiers `n` et `m` et qui renvoie un pointeur sur un tableau à deux dimensions de `n` $\times$ `m` entiers initialisés à `0`.
2. Écrire une fonction `detruire_tab_2d` paramétrée par un entier `n` et un pointeur `tab` sur un tableau à deux dimensions de `n` $\times$ `m` entiers (il n'est pas nécessaire de connaître `m` ici). Cette fonction doit libérer la place mémoire occupée par `tab`.
3. Dessiner l'état de la mémoire étape par étape et de manière très précise lors de l'exécution des instructions

```
int **tab = creer_tab_2d(2, 3);
detruire_tab_2d(tab, 2);
```

4. Expliquer comment représenter et gérer un tableau à deux dimensions par un tableau à une seule dimension.

Exercice 7.5. (Tableaux dynamiques en dents de scie)

1. Écrire une fonction `creer_scie` paramétrée par un entier `n` et qui renvoie un pointeur sur un tableau à deux dimensions de caractères. Pour tout $0 \leqslant i \leqslant n - 1$, la i^e case du tableau à construire contient un tableau à une dimension de i modulo 5 plus un caractères `'*'`.
2. Écrire une fonction `detruire_scie` paramétrée par un entier `n` et un pointeur `scie` sur un tableau à deux dimensions de caractères (pouvant être construit par un appel à `creer_scie`). Cette fonction doit libérer la place mémoire occupée par `scie`.

Exercice 7.6. (Variables simples en mémoire)

1. Rappeler la différence entre la convention *little-endian* et la convention *big-endian* pour l'écriture des données dans la mémoire.
2. Pour chaque déclaration et affectation de variable suivante, dessiner la zone de la mémoire impliquée ainsi que son contenu octet par octet.

(a)
```
unsigned int x;
x = 0;
```

(b)
```
char x;
x = 'a';
```

(c)
```
int x;
x = 2224;
```

(d)
```
int x;
x = -2224;
```

(e)
```
short x;
x = -10;
```

(f)
```
unsigned short x;
x = -10;
```

(g)
```
int *ptr;
ptr = NULL;
```

(h)
```
char tab[4] =
    {21, -1, 'a', 99};
```

(i)
```
unsigned char tab[2] =
    {-1, 1};
```

8.
```
int tab[3] = {21, -1, 'a', 120};
```

9.
```
int tab[3][2];

tab[0][0] = 1;
tab[0][1] = 10;
tab[1][0] = 100;
tab[1][1] = 1000;
tab[2][0] = 10000;
tab[2][1] = 100000;
```

Exercice 7.7. (Lecture de zones mémoire)

Soit la suite d'instructions suivante :

```
unsigned int *ptr_int;
unsigned short *ptr_short;
unsigned char *ptr_char;

unsigned int a;

a = 0xBEA0201F;
```

1. Expliquer ce qu'affichent les instructions

(a)
```
ptr_int = &a;
printf("%x\n", *ptr_int);
```

(b)
```
ptr_short = (unsigned short *) &a;
printf("%x\n", *ptr_short);
```

(c)
```
ptr_char = (unsigned char *) &a;
printf("%x\n", *ptr_char);
```

(d)
```
ptr_short = (unsigned short *) &a;
printf("%x\n", *(ptr_short + 1));
```

(e)
```
ptr_char = (unsigned char *) &a;
printf("%x\n", *(ptr_char + 1));
```

(f)
```
ptr_char = (unsigned char *) &a;
printf("%x\n", *(ptr_char + 2));
```

2. Reprendre la question précédente dans le cas où l'on supprime les quatre occurrences de `unsigned` dans la suite d'instructions.

Exercice 7.8. (Lecture de zones mémoire et tableaux)

Soit la suite d'instructions suivante :

```
unsigned char tab[8] = {0xAA, 0x10, 0x3B, 0x44, 0x21, 0x45, 0x00, 0x7C};
unsigned char *ptr_char;
unsigned short *ptr_short;
unsigned int *ptr_int;
```

1. Expliquer ce qu'affichent les instructions

(a)
```
ptr_char = tab;
printf("%x\n", *ptr_char);
```

(b)
```
ptr_short = (unsigned short *) tab;
printf("%x\n", *ptr_short);
```

(c)
```
ptr_int = (unsigned int *) tab;
printf("%x\n", *ptr_int);
```

(d)
```
ptr_short = (unsigned short *) tab;
printf("%x\n", *(ptr_short + 1));
```

(e)
```
ptr_char = tab;
printf("%x\n", *(ptr_char + 1));
```

(f)
```
ptr_char = tab;
printf("%x\n", *(ptr_char + 2));
```

2. Reprendre la question précédente dans le cas où l'on supprime les quatre occurrences de `unsigned` dans la suite d'instructions.

Chapitre 8

Pointeurs de fonctions

Exercice 8.1. (Déclarations de pointeurs de fonctions)

1. Déclarer un pointeur `f_1` sur une fonction paramétrée par deux entiers et qui renvoie un caractère.
2. Déclarer un pointeur `f_2` sur une fonction paramétrée par une chaîne de caractères et un flottant et qui renvoie un pointeur sur un entier.
3. Déclarer un pointeur `f_3` sur une fonction paramétrée par un caractère et un pointeur sur une fonction de même prototype que celle de `f_1` et qui renvoie une chaîne de caractères.
4. Déclarer un pointeur `f_4` sur une fonction paramétrée par deux caractères et qui renvoie un pointeur sur une fonction de même prototype que celle de `f_2`.
5. Déclarer un pointeur `f_5` sur une fonction paramétrée par une fonction de même prototype que celle de `f_2` et qui renvoie un pointeur sur une fonction de même prototype que celle de `f_1`.
6. Déclarer un tableau statique `tab_f_1` de `32` pointeurs de fonctions de mêmes signatures que celle de `f_1`.

Exercice 8.2. (Pointeurs de fonction et tableaux)

1. Écrire une fonction `fois_deux` qui renvoie le double de son argument entier.
2. Écrire une fonction `fact` qui renvoie la factorielle de son argument entier.
3. Écrire une fonction

```
void appliquer_tableau(int (*f)(int), int *tab, int n);
```

qui modifie chaque élément du tableau `tab` de taille `n` en son image par la fonction pointée par `f`.

4. En supposant que `tab` est un tableau d'entiers de taille `128`, écrire une suite d'instructions qui modifie chaque élément du tableau en le double de sa factorielle. On suppose que les valeurs du tableau sont telles que chaque calcul peut se faire sans dépassement de capacité.

5. Nous voulons maintenant modifier chaque élément d'un tableau `tab` d'entiers de sorte à remplacer chaque entrée `tab[i]` par son image par une fonction `f_i`. Pour cela, écrire une fonction

```
void appliquer_tableau_2(int (*tab_f[])(int), int *tab, int n);
```

paramétrée par un tableau `tab_f` de `n` fonctions et un tableau d'entiers `tab` de taille `n`.

Exercice 8.3. (Pointeurs de fonction et tris)

Considérons les deux fonctions suivantes :

```
void tri_croissant(int *tab, int n) {
    int i, i_min, j, tmp;
    for (i = 0 ; i < n - 1 ; i++) {
        i_min = i;
        for (j = i ; j < n ; j++) {
            if (tab[j] < tab[i_min])
                i_min = j;
        }
        tmp = tab[i_min];
        tab[i_min] = tab[i];
        tab[i] = tmp;
    }
}
```

```
void tri_decroissant(int *tab, int n) {
    int i, i_max, j, tmp;
    for (i = 0 ; i < n - 1 ; i++) {
        i_max = i;
        for (j = i ; j < n ; j++) {
            if (tab[j] > tab[i_max])
                i_max = j;
        }
        tmp = tab[i_max];
        tab[i_max] = tab[i];
        tab[i] = tmp;
    }
}
```

L'une trie les éléments d'un tableau `tab` de taille `n` dans l'ordre croissant et l'autre, dans l'ordre décroissant.

Ces deux fonctions sont identiques, à l'exception de l'opérateur de comparaison en ligne 6 et de certains noms pour les variables locales. L'utilisation de pointeurs de fonctions permet d'éviter cette redondance de code et d'avoir ainsi une unique fonction qui réalise, selon la manière dont elle est appelée, l'un ou l'autre tri.

1. Écrire une fonction

```
int superieur(int a, int b);
```

qui renvoie `1` si `a` est strictement supérieur à `b`, `0` s'ils sont égaux et `-1` sinon.

2. Écrire une fonction

```
int inferieur(int a, int b);
```

qui renvoie `1` si `a` est strictement inférieur à `b`, `0` s'ils sont égaux et `-1` sinon.

3. Écrire une fonction `tri`, paramétrée par un tableau d'entiers `tab`, sa taille `n` et une fonction `comparer`. Cette fonction modifie `tab` de sorte à le trier selon la comparaison dictée par la fonction `comparer`. Plus précisément, si `a` et `b` sont des éléments de `tab` et que `a` apparaît dans `tab` à gauche de `b`, il faut que la fonction `comparer` appelée avec les arguments `a` et `b` renvoie `-1` (ou `0` pour les répétitions d'éléments).

4. En supposant que `tab` est un tableau d'entiers de taille `2047`, écrire une suite d'instructions qui trie `tab` dans l'ordre croissant, affiche ses valeurs, trie `tab` dans l'ordre décroissant et affiche ses valeurs.

Chapitre 9

Généricité

Exercice 9.1. (Pointeurs et données génériques)

Un *pointeur générique* est un pointeur de type `void *`. Une *donnée générique* est une variable adressée par un pointeur générique.

1. On suppose que `a` est une variable de type `int`. Définir un pointeur générique `ptr` qui adresse `a`.
2. Multiplier par trois la valeur de `a` en opérant uniquement sur `ptr`.
3. On suppose maintenant que `ptr` est un pointeur générique et que l'on dispose d'un type

   ```
   typedef struct {
       int x;
       int y;
   } Couple;
   ```

   et d'une variable `b` de ce type. Faire pointer `ptr` vers `b`.
4. Incrémenter le champ `y` de `b` en opérant uniquement sur `ptr`.
5. Écrire une fonction

   ```
   void *twist(void *couple);
   ```

   qui travaille avec des pointeurs génériques voués à être des données de type `Couple`. Cette fonction renvoie un nouveau couple obtenu en permutant les champs `x` et `y` du couple en argument.
6. En supposant que `c` est une variable de type `Couple`, écrire une suite d'instructions appelant la fonction précédente sur `c`.

Exercice 9.2. (Test d'égalité de zones de la mémoire)

1. Écrire une fonction qui permet de tester l'égalité entre deux variables d'un type quelconque. La fonction admet le prototype

```
int sont_egales(int nb_octets, void *var1, void *var2);
```

   et elle renvoie `1` si les `nb_octets` lus à partir des adresses `var1` et `var2` sont égaux deux à deux et `0` sinon.

2. On suppose que `num1` et `num2` sont deux variables de type `short`. Écrire l'appel à la fonction `sont_egales` pour comparer les valeurs de ces variables.

3. On suppose que `res` est une variable de type `int`. Pour chacune des suites d'instructions suivantes, expliquer la valeur de `res` à la fin de leur exécution.

(a)
```
int a;
char b;

a = 3;
b = 3;
res = sont_egales(1, &a, &b);
```

(b)
```
int a;
int b;

a = (1 << 8) + 32;
b = 32;
res = sont_egales(1, &a, &b);
```

(c)
```
int a;
int b;

a = (1 << 8) + 32;
b = 32;
res = sont_egales(2, &a, &b);
```

(d)
```
int a;
int b;

a = 0xAE00BBAA;
b = 0xEA00BBAA;
res = sont_egales(3, &a, &b);
```

4. Expliquer comment utiliser la fonction `sont_egales` pour tester si deux variables d'un type structuré `T` quelconque sont égales. Expliquer ce qu'il se passe si certains champs de `T` sont des pointeurs.

Exercice 9.3. (Tableaux génériques)

Un *tableau générique* est une variable de type `void *`, interprétée comme une concaténation d'octets formant, bloc par bloc, les cases du tableau. Pour manipuler une telle donnée, il est nécessaire de connaître la taille du tableau (nombre de cases) ainsi que le nombre d'octets nécessaires pour représenter un élément du tableau (nombre d'octets par bloc).

1. Calculer le nombre d'octets nécessaires pour représenter un tableau générique de taille 24 voué à contenir des valeurs occupant chacune 8 octets.

2. Représenter graphiquement un tableau générique de taille 4 voué à contenir des valeurs occupant chacune 2 octets.

3. Représenter graphiquement un tableau générique de taille 2 voué à contenir des valeurs occupant chacune 4 octets.
4. Supposons que `tab` est un tableau générique. Supposons de plus que `tab` est utilisé pour contenir des données de type `short`. Donner trois manières d'accéder à la 4^{e} donnée du tableau.
5. Répondre à la même question que la précédente dans le cas où `tab` est utilisé pour contenir des `int`.

Exercice 9.4. (Minimum générique dans un tableau)

L'objectif de cette exercice est d'écrire une fonction générique qui calcule la plus petite valeur contenue dans un tableau générique.

1. Déterminer la signature d'une fonction qui prend deux données génériques en argument et qui renvoie la plus petite des deux. En faire un type.
2. Écrire une telle fonction générique qui calcule la plus petite valeur entre deux entiers.
3. Écrire une telle fonction générique qui calcule la plus petite valeur entre deux chaînes de caractères (il est conseillé d'utiliser la fonction `strcmp` de `string.h`).
4. Déterminer la signature de la fonction `min_tab` qui répond à l'objectif de l'exercice. Elle est paramétrée, entre autres, par un tableau générique et une fonction de calcul de la plus petite valeur entre deux valeurs données.
5. Donner le corps de la fonction `min_tab`.
6. En supposant que `tab` est un tableau d'entiers de taille `64`, écrire une suite d'instructions qui calcule et affiche sa plus petite valeur, en utilisant `min_tab`.
7. En supposant que `tab` est un tableau de `64` chaînes de caractères et toutes de taille `96`, écrire une suite d'instructions qui calcule et affiche sa plus petite valeur, en utilisant `min_tab`.