Apprendre à programmer en C pour les Nuls

Solutions des exercices

Cahier 3/3 : Parties IV et V (chapitres 18 à 25)

PARTIE IV

Chapitre 18

ex1801

```
#include <stdio.h>

int main()
{
    char   c = 'c';
    int    i = 123;
    float  f = 98.6;
    double d = 6.022E23;

    printf("char  \t%u\n", sizeof(c));   // L10
    printf("int   \t%u\n", sizeof(i));
    printf("float \t%u\n", sizeof(f));
    printf("double\t%u\n", sizeof(d));
    return(0);
}
```

Remarques

Ce code source s'applique à Windows. Sous MacOS ou Linux, il faut remplacer le formateur `%u` par `%ld` car sur ces variantes d'Unix, `sizeof` renvoie une valeur de type entier long.

Il n'est pas obligatoire d'initialiser les variables (lignes 5 à 8) avant de les faire traiter par l'opérateur `sizeof`.

Le formateur `%u` est indispensable pour bien afficher l'entier non signé `unsigned int` que renvoie `sizeof`.

Certains compilateurs acceptent le formateur spécial `%zd` pour afficher une variable de type `size_t` telle que renvoyée par `sizeof`.

ex1802

```
#include <stdio.h>

int main()
{
    char string[] = "Suis-je trop longue pour vous ?";

    printf("La chaine \"%s\" mesure %u.\n", string, sizeof(string));
    return(0);
}
```

Remarque

Comme dans l'exercice précédent, sous Linux ou MacOS, remplacez le formateur `%u` dans `printf()` en ligne 7 par `%ld`, car `sizeof` renvoie un *long int* sur ces systèmes.

ex1803

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

int main()
{
    char string[] = "Suis-je trop longue pour vous ?";

    printf("La chaine \"%s\" mesure %u,\n", string, sizeof(string));
    printf("et strlen renvoie %ld.\n", strlen(string));
    return(0);
}
```

Remarques

Sous MacOS et Linux, remplacez le `%u` en ligne 8 par `%ld`.

N'oubliez pas de faire insérer le fichier `string.h` pour avoir accès à `strlen()`.

Exemple d'affichage résultant :

```
La chaîne "Suis-je trop longue pour vous ?" mesure 35,
et strlen renvoie 34.
```

ex1804

```
#include <stdio.h>

int main()
{
    int array[5];

    printf("Pour ce tableau, sizeof renvoie %u.\n", sizeof(array));
    return(0);
}
```

Remarques

Sous MacOS et Linux, remplacez le `%u` en ligne 7 par `%ld`.

Comme pour les variables simples, il n'est pas obligatoire de peupler le tableau avant de le soumettre à l'opérateur `sizeof`. Le simple fait de déclarer le tableau provoque la réservation d'un espace mémoire pour cinq entiers pendant l'exécution. L'opérateur `sizeof` se contente de renvoyer la quantité d'espace réservé.

Exemple d'affichage résultant :

```
Pour ce tableau, sizeof renvoie 20.
```

Un tableau de cinq entiers `int` occupe 20 octets puisqu'un entier est sur quatre octets.

Si vous déclarez un tableau de 5 valeurs *double* (8 octets), l'espace monte à 40 octets. Changez la ligne de déclaration et relancez pour le vérifier.

ex1805

```
#include <stdio.h>

int main()
{
    struct robot {
        int alive;
        char name[5];
        int xpos;
        int ypos;
```

```
        int strength;
    };

    printf("Taille de la structure robot : %u\n", sizeof(struct robot));
    return(0);
}
```

Remarques

Sous MacOS et plusieurs Unix, remplacez le formateur de `printf()` en ligne 13 par `%ld`.

ex1806

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

int main()
{
    char   c = 'c';
    int    i = 123;
    float  f = 98.6;
    double d = 6.022E23;

    printf("Adresse de 'c' %p\n", &c);
    printf("Adresse de 'i' %p\n", &i);
    printf("Adresse de 'f' %p\n", &f);
    printf("Adresse de 'd' %p\n", &d);
    return(0);
}
```

Remarque

Cet exemple dérive du Listing 18.1 (ex1801).

ex1807

```
#include <stdio.h>

int main()
{
    char hello[] = "Salut!";
    int i = 0;

    while(hello[i])
    {
        printf("%c en %p\n", hello[i], &hello[i]);
        i++;
    }
    return(0);
}
```

Remarque

L'écriture `&hello[x]` désigne l'adresse mémoire d'un élément de tableau, car `&hello` est l'adresse de départ du tableau, comme vous le verrez dans le Chapitre 19.

ex1808

```
#include <stdio.h>

int main()
{
    int montab[5] = { 100, 200, 300, 400, 500 };
    int x;
```

```
    for (x=0; x<5; x++)
        printf("%d at %p\n", montab[x], &montab[x]);
    return(0);
}
```

ex1809

```
#include <stdio.h>

int main()
{
    char lead;
    char *sidekick;

    lead = 'A';          /* Initialise le char */
    sidekick = &lead;    /* Initialise le pointeur - IMPORTANT! */

    printf("Pour la variable 'lead':\n");
    printf("Taille\t\t%u\n",  sizeof(lead));      // L12
    printf("Contenu\t\t%c\n", lead);
    printf("Adresse\t\t%p\n", &lead);
    printf("Pour la variable 'sidekick':\n");
    printf("Contenu\t\t%p\n", sidekick);

    return(0);
}
```

Remarques

Sous MacOS et plusieurs Unix, remplacez le formateur de `printf()` en ligne 12 par `%ld`.

Dans l'affichage, vous ne verrez peut-être pas le préfixe d'adresse `0x` montré dans le livre.

ex1810

```
#include <stdio.h>

int main()
{
    char lead;
    char *sidekick;

    lead = 'A';          /* Initialise le char */
    sidekick = &lead;    /* Initialise le pointeur - IMPORTANT! */

    printf("Pour la variable 'lead':\n");
    printf("Taille\t\t%u\n",  sizeof(lead));      // L12
    printf("Contenu\t\t%c\n", lead);
    printf("Adresse\t\t%p\n", &lead);
    printf("Pour la variable 'sidekick':\n");
    printf("Contenu\t\t%p\n", sidekick);
    printf("Valeur indirecte\t%c\n", *sidekick);

    return(0);
}
```

Remarques

Sous MacOS et plusieurs Unix, remplacez le formateur de `printf()` en ligne 12 par `%ld`.

Comme pour l'opérateur d'adresse `&`, le compilateur sait quand le symbole `*` se trouve utilisé comme préfixe d'un nom de variable et ne le confond pas avec l'opérateur de multiplication. Il devient dans ce contexte l'opérateur unaire de pointeur.

ex1811

```
#include <stdio.h>

int main()
{
    char a,b,c;
    char *p;

    a = 'A'; b = 'B'; c = 'C';       // L08

    printf("Apprenez votre ");
    p = &a;                 // Initialise
    putchar(*p);            //  Utilise
    p = &b;                 // Initialise
    putchar(*p);            //  Utilise
    p = &c;                 // Initialise
    putchar(*p);            //  Utilise
    printf(" \n");

    return(0);
}
```

Remarques

La ligne 8 n'est pas une astuce spéciale : je ne fais que profiter du fait que le compilateur ne tient pas compte des espaces. La brièveté des déclarations ne justifiait pas ici trois lignes distinctes.

ex1812

```
#include <stdio.h>

int main()
{
    int i = 1814;
    int *pi = &i;              // L06

    printf("La valeur de 'i' lue via le pointeur est %d.\n", *pi);
    return(0);
}
```

Remarques

La variable entière `i` est déclarée et initialisée en ligne 5.

L'autre variable entière `pi` est déclarée ensuite et peuplée illico avec l'adresse de `i`. Normalement, le signe `*` sert à déclarer une variable pointeur, et vous ne le mentionnez pas si vous initialisez ce pointeur avec une adresse dans une opération à part. Voici les deux instructions équivalentes à la ligne 6 :

```
int *pi;
pi = &i;          // Pas de signe *
```

La ligne 8 affiche la valeur de `i` sans citer son nom, en passant par le pointeur avec `*pi`.

ex1813

```
#include <stdio.h>

int main()
{
    char a, b, c;
    char *p;
```

```
    p  = &a;
    *p = 'A';
    p  = &b;
    *p = 'B';
    p  = &c;
    *p = 'C';
    printf("Apprenez votre %c%c%c\n", a, b, c);
    return(0);
}
```

Remarques

Voici l'affichage résultant :

```
  Apprenez votre ABC
```

ex1814

```
#include <stdio.h>

int main()
{
    int   age;
    float poids;
    int   *a;
    float *w;

    a = &age;           // Initialise les pointeurs
    w = &poids;

    *a = 55;            // Peuple les variables via
    *w = 217.6;         //  les pointeurs

    printf("Son age est de %d ans.\n", age);
    printf("Et sa masse est de %.1f kilos.\n", poids);

    return(0);
}
```

Remarque

Vous pouvez considérez avoir réussi l'exercice même avec d'autres variables, l'essentiel étant d'avoir créé un entier et un flottant avec leur pointeur associé.

CHAP 19

ex1901

```
#include <stdio.h>

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

    printf("Adresse de 'tablo' = %p\n",&tablo);
    return(0);
}
```

Remarque

Rappelons que l'adresse affichée varie d'une machine à l'autre.

ex1902

```
#include <stdio.h>

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

    printf("Adresse de 'tablo' = %p\n", &tablo);
    printf("Adresse de 'tablo' = %p\n", tablo);
    return(0);
}
```

ex1903

```
#include <stdio.h>

int main()
{
    char c = 'c';
    int i = 123;
    float f = 98.6;
    double d = 6.022E23;

    printf("Adresse de 'c' %p\n", c);
    printf("Adresse de 'i' %p\n", i);
    printf("Adresse de 'f' %p\n", f);
    printf("Adresse de 'd' %p\n", d);
    return(0);
}
```

Remarques

Le message de compilation n'est qu'un avertissement, et le fichier exécutable est généré, mais l'affichage est très suspect.

Le message laisse comprendre qu'il faut fournir un pointeur quand on spécifier le formateur `%p`.

La mention `type 'void *'` dans le message ne veut pas dire qu'il faut fournir une variable de type `void`. La mention se lit `void *` et rappelle que c'est un pointeur qui est attendu.

ex1904

```
#include <stdio.h>

int main()
{
    int nombres[10];
    int x;
    int *pn;

    pn = nombres;       /* Initialisation de pointeur */

/* Peuplement du tableau */
    for(x=0; x<10; x++)
    {
        *pn=x+1;
        pn++;
    }

/* Affichage du tableau */
    for(x=0; x<10; x++)
```

```
        printf("nombres[%d] = %d\n", x+1, nombres[x]);

    return(0);
}
```

Remarque

Un pointeur est d'abord une variable. Vous pouvez en changer son contenu à volonté en cours d'exécution du code.

ex1905

```
#include <stdio.h>

int main()
{
    int nombres[10];
    int x;
    int *pn;

    pn = nombres;       /* Initialisation de pointeur */

/* Peuplement du tableau */
    for(x=0; x<10; x++)
    {
        *pn=x+1;
        pn++;
    }

    pn = nombres;                   // L18

/* Affichage du tableau */
    for(x=0; x<10; x++)
    {
        printf("nombres[%d] = %d, adresse %p\n", x+1, nombres[x],pn);

        pn++;                       // L25
    }

    return(0);
}
```

Remarques

Pour afficher tour à tour l'adresse de chaque élément, il faut modifier la valeur de `pn`, ce que nous faisons en ligne 18.

N'oubliez pas d'incrément la variable `pn` dans la boucle (ligne 25).

ex1906

```
#include <stdio.h>

int main()
{
    int nombres[10];
    int x;
    int *pn;

    pn = nombres;       /* Initialisation de pointeur */

/* Peuplement du tableau */
    for(x=0; x<10; x++)
    {
```

```
        *pn=x+1;
        pn++;
    }

    pn = nombres;

/* Affichage du tableau */
    for(x=0; x<10; x++)
    {
        printf("nombres[%d] = %d, adresse %p\n", x+1, *pn, pn);  // L23
        pn++;
    }

    return(0);
}
```

Remarque

La seule retouche est le remplacement de la notation tableau `nombres[x]` par la notation pointeur `*pn` en ligne 23.

ex1907

```
#include <stdio.h>

int main()
{
    char alphabet[27];
    int x;
    char *pa;

    pa = alphabet;      /* Initialisation de pointeur */

/* Peuplement du tableau */
    for(x=0; x<26; x++)
    {
        *pa=x+'A';
        pa++;
    }

    pa = alphabet;

/* Affichage du tableau */
    for(x=0; x<26; x++)
    {
        putchar(*pa);
        pa++;
    }
    putchar('\n');

    return(0);
}
```

Remarques

Voici une autre technique pour balayer le tableau des lettres A à Z :

```
  for(x='A';x<='Z';x++)
```

Vous pouvez alors avec `*pa=x;` dans la boucle écrire chaque élément.

Le code ne manipule pas de lettres, mais des codes ASCII de lettres. Ainsi, le `'A'` devient la valeur 65. Voyez l'Annexe A pour la liste des codes ASCII.

ex1908

```
#include <stdio.h>

int main()
{
    char alphabet[27];
    int x;
    char *pa;

    pa = alphabet;      /* Initialisation de pointeur */

/* Peuplement du tableau */
    for(x=0; x<26; x++)
    {
        *pa++ = x+'A';            // L14
    }

    pa = alphabet;

/* Affichage du tableau */
    for(x=0; x<26; x++)
    {
        putchar(*pa);
        pa++;
    }
    putchar('\n');

    return(0);
}
```

Remarque

Voyons l'expression `*pa++` en ligne 14. En théorie, l'opérateur `++` est plus local à la variable que `*`. Les deux `++` et `*` sont de même niveau de priorité, mais ils sont évalués de droite à gauche. Ici, l'opérateur d'incrémentation `++` est en postfixe ; la valeur actuelle de `pa` sert d'adresse de copie avant d'être modifiée. En inversant l'ordre des opérations (`*++pa`), l'affichage oublierait la lettre Z en ajouterait un caractère bizarre en premier (lecture hors tableau !).

ex1909

```
#include <stdio.h>

int main()
{
    char alphabet[27];
    int x;
    char *pa;

    pa = alphabet;      /* Initialisation de pointeur */

/* Peuplement du tableau */
    for(x=0;x<27;x++)
    {
        *pa++ = x+'A';
    }

    pa = alphabet;

/* Affichage du tableau */
    for(x=0; x<26; x++)
    {
```

```
        putchar(*pa++);
    }
    putchar('\n');

    return(0);
}
```

Remarque

La ligne 22 regroupe les deux instructions suivantes :

```
  putchar(*pa);
  pa++;
```

Nous avons imbriqué la seconde dans l'appel à `putchar()`. N'est-ce pas élégant ?

ex1910

```
#include <stdio.h>

int main()
{
    char alpha = 'A';
    int x;
    char *pa;

    pa = &alpha;        /* Initialisation de pointeur */

    for(x=0; x<26; x++)
        putchar((*pa)++);          // L12
    putchar('\n');

    return(0);
}
```

Remarques

La délicate notation `(*pa)++` illustre la puissance des pointeurs. La même variable en gère une autre à distance tout en pouvant changer de cible. (même si ici l'adresse stockée dans `pa` ne change pas.)

Il faut ajouter des parenthèses dans `(*pa)++`. Sans parenthèses, `*pa++` récupère la valeur stockée à l'adresse montrée par `pa`, puis cette adresse est incrémentée. Avec les parenthèses dans `(*pa)++` , nous incrémentons la valeur stockée à l'adresse trouvée dans `pa`, et c'est tout.

N.d.T. : Tout le monde suit toujours ?

Voici un bon exemple du défi intellectuel que vous propose la notation par pointeur en langage C.

ex1911

```
#include <stdio.h>

int main()
{
    float temps[5] = { 58.7, 62.8, 65.0, 63.3, 63.2 };

    printf("Mardi, il fera    %.1f\n", *(temps+1));
    printf("Vendredi, il fera %.1f\n", *(temps+4));
    return(0);
}
```

Remarques

Voici la solution littérale : nous avons remplacé la notation tableau par une notation pointeur sans changer le nom de variable, e qui peut vous intriguer. Cela fonctionne pourtant. Voyez plutôt cette approche plus orthodoxe qui a besoin d'une variable pointeur. Elle est plus ergonomique :

```c
#include <stdio.h>

int main()
{
    float temps[5] = { 58.7, 62.8, 65.0, 63.3, 63.2 };
    float *t;

    t = temps;
    printf("Mardi, il fera    %.1f\n",     *(t+1));
    printf("Vendredi, il fera %.1f\n",     *(t+4));
    return(0);
}
```

Nous déclarons le pointeur de travail en ligne 6.

En ligne 8, nous le faisons pointer sur le tableau.

Les deux instructions d'affichage contiennent plusieurs espaces "cosmétiques" pour bien montrer le travail du pointeur.

En ligne 10, `t+1` désigne le deuxième élément du tableau, le premier étant à `t+0` (ou bien `t` directement). Il faut des parenthèses (revoir le Tableau 19.2) pour viser l'élément de tableau. L'opérateur `*` en dehors des parenthèses sert à lire la valeur trouvée dans cet élément. Même principe en ligne suivante pour le cinquième élément `*(t+4)`.

ex1912

```c
#include <stdio.h>

int main()
{
    char sample[] = "D'ou me viendra le secours ?\n";
    char *ps = sample;

    while(*ps != '\0')
    {
        putchar(*ps);
        ps++;
    }
    return(0);
}
```

Remarques

Contrairement au Listing 19.6 du livre, nous n'avons pas besoin d'une variable `index` puisque le pointeur `ps` peut jouer ce rôle.

En ligne 6, le pointeur `ps` est déclaré puis orienté sur la variable `sample`.

La boucle `while` balaye la chaîne comme dans le Listing 19.6. Si nous avions utilisé la variable `index`, la boucle serait celle-ci :

```c
while(*(ps+index) != '\0')
{
    putchar(*(ps+index));
    index++;
}
```

Je trouve que ma version (la première) est plus lisible.

Revoyez le Tableau 19.2 du livre pour réviser l'effet de `*(a+n)`, mais c'est inutile pour la version n'utilisant que le pointeur.

Les deux instructions de la boucle auraient pu être regroupées ainsi :

```c
  putchar(*ps++);
```

Un bonus pour vous si vous aviez écrit de cette façon.

Cette écriture compacte `putchar(*ps++);` est réalisable avec la notation tableau, mais le résultat est très laid.

ex1913

```
#include <stdio.h>

int main()
{
    char sample[] = "D'ou me viendra le secours ?\n";
    char *ps = sample;

    while(*ps)
        putchar(*ps++);
    return(0);
}
```

Remarque

Vous constatez que le regroupement des deux instructions m'a permis d'enlever les accolades du bloc conditionnel `while`.

ex1914

```
#include <stdio.h>

int main()
{
    char sample[] = "D'ou me viendra le secours ?\n";
    char *ps = sample;

    while(putchar(*ps++))
        ;
    return(0);
}
```

Remarques

Cette boucle `while` (ligne 8) suffit à afficher toute une chaîne via un pointeur.

Rappelons la boucle de l'exercice précédent :

```
while(*ps)
    putchar(*ps++);
```

Quelle est la différence entre l'élément dans la condition du `while` et celui en argument de `putchar()` ? Il s'agit dans les deux cas d'un caractère sélectionné d'une chaîne. Vous pourriez aussi écrie ceci :

```
while(putchar(*ps))
    ps++;
```

J'ai dit dans le livre que *putchar()* renvoyait le caractère affiché, et donc le zéro terminal `\0` en fin de chaîne, ce qui suffit à sortir de la boucle.

L'unique instruction `ps++` ne sert qu'à incrémenter le pointeur, ce qui peut aussi être fait dans la condition de boucle. Nous avons parlé plus haut des priorités relatives de `++` et de `*`, et du fait que la lecture à l'adresse actuelle était faite avant de changer l'adresse. Voilà pourquoi, pas à pas, nous retrouvons une boucle vide, tout le traitement étant fait dans sa condition. Il ne reste qu'un point-virgule isolé :

```
;
```

ex1915

```
#include <stdio.h>

int main()
{
    char *sample = "D'ou me viendra le secours ?\n";

    while(putchar(*sample++))     // L07
        ;
    return(0);
}
```

Remarque

L'incrémentation de `sample` en ligne 7 fait évoluer sa valeur. La position de départ qui avait été stockée dans le pointeur an début est perdue. Il n'est plus possible ensuite d'accéder à la chaîne via `sample`. (En théorie, c'est possible, mais au prix d'acrobaties avec le pointeur qui vous pousseront à ne pas accéder à la chaîne via un pointeur dans cette situation.)

ex1916

```
#include <stdio.h>

int main()
{
    char *sample = "D'ou me viendra le secours ?\n";
    char *sauvePtr;

    sauvePtr = sample;

    while(putchar(*sample++))
        ;
    sample = sauvePtr;
    puts(sample);
    return(0);
}
```

Remarques

Pour faire une copie de sécurité d'un pointeur, il faut un autre pointeur (déclare en ligne 6).

La ligne 8 écrit dans `sauvePtr` le contenu (une adresse) trouvé dans `sample`. L'opérateur `&` est inutile ici, puisque c'est la valeur que nous désirons et les deux éléments sont de type pointeur.

Si vous aviez écrit `sauvePtr=&sample`, vous auriez récupéré l'adresse d'une variable qui contient l'adresse d'une variable qui contient une valeur (tout va bien ?).

Nous affichons la chaîne en lignes 10 et 11 grâce au pointeur à usage unique `sample`. Vous pourriez aussi bien utiliser `sauvePtr` et c'est une pratique répandue. Le pointeur initial est gardé inchangé, et ion travaille sur des copies jetables (`sauvePtr` ici).

La ligne 12 réinitialise `sample` sur le début de chaîne.

Enfin, en ligne 13, nous prouvons que le pointeur a été restauré en affichant la chaîne via `sample`.

Un exemple d'affichage :

```
D'ou me viendra le secours ?
 D'ou me viendra le secours ?
```

Si la seconde ligne est en retrait de un, c'est que la boucle a aussi affiché le zéro terminal, mais tous les systèmes ne l'affichent pas en tant qu'espace. Sous MacOS, les deux lignes restent alignées. Enfin, la fonction `puts()` ajoute un saut de ligne final.

ex1917

```
#include <stdio.h>

int main()
{
    char *fruits[] = {
        "melon",
        "banane",
        "poire",
        "pomme",
        "noix",
        "raisin",
        "myrtille"
    };
    int x;

    for(x=0; x<7; x++)
        puts(fruits[x]);

    return(0);
}
```

Remarque

Aucun pointeur dans cet exemple initial. Nous initialisons un tableau de chaînes et nous cantonnons à la notation tableau.

ex1918

```
#include <stdio.h>

int main()
{
    char *fruits[] = {
        "melon",
        "banane",
        "poire",
        "pomme",
        "noix",
        "raisin",
        "myrtille"
    };
    int x;

    for(x=0; x<7; x++)
        puts(*(fruits+x));          // L17

    return(0);
}
```

Remarques

La ligne 17 (`puts()`) est correcte parce que le tableau `fruits` contient des pointeurs. Chaque élément est une adresse : celle du début d'une chaîne. Ainsi, `fruits+0` désigne la première chaîne et `*(fruits+0)` son adresse. Cette variable est donc utilisable comme une chaîne partout dans le code.

En un sens, cet exemple est une version basée pointeurs de **ex1917**. La notation `fruits[x]` est remplacée par `*(fruits+x)`, en conformité avec le Tableau 19.2.

ex1919

```
#include <stdio.h>

int main()
{
    char *fruits[] = {
        "melon",
        "banane",
        "poire",
        "pomme",
        "noix",
        "raisin",
        "myrtille"
    };
    int x;

    for(x=0; x<7; x++)
    {
        putchar(**(fruits+x));
        putchar('\n');
    }

    return(0);
}
```

Remarques

Voici ce qui s'affiche :

```
m
b
p
p
n
r
m
```

L'initiale de chaque fruit ! En effet, la notation `**(fruits+x)` désigne un seul caractère, pas une chaîne. Souvenez-vous que `*(fruits+x)` est l'adresse de la chaîne ; le second indirecteur * lit le contenu de cette adresse (la première case mémoire de la chaîne).

ex1920

```
#include <stdio.h>

int main()
{
    char *fruits[] = {
        "melon",
        "banane",
        "poire",
        "pomme",
        "noix",
        "raisin",
        "myrtille"
    };
    int x,a;

    for(x=0; x<7; x++)
    {
        a=0;
```

```
        while(putchar(*(*(fruits+x)+a++)))      // L19
            ;
        putchar('\n');
    }

    return(0);
}
```

Remarques

J'ai réussi à regrouper tout le traitement de la boucle en ligne 19 dans sa condition. C'est peu-être un peu trop compact. Développons un peu :

```
while(putchar(*(*(fruits+x)+a)))
    a++;
```

J'ai ressorti l'incrémentation du pointeur `a`. Il nous en reste une belle bête `*(*(fruits+x)+a)`. Nous pouvons déporter l'appel, mais ce n'est pas plus digeste, car la condition doit rester :

```
while(*(*(fruits+x)+a))
{
    putchar(*(*(fruits+x)+a));
    a++;
}
```

Aucun souci si vous avez pas trouvé par vous-même, car cet exercice était difficile. D'ailleurs, vous en rencontrerez rarement dans les programmes. Il est plus lisible d'utiliser `printf()` pour afficher des chaînes.

ex1921

```
#include <stdio.h>

int main()
{
    char *fruits[] = {
        "abricot",
        "banane",
        "ananas",
        "pomme",
        "kaki",
        "raisin",
        "myrtille"
    };
    char *temp;
    int a,b,x;

    for(a=0; a<6; a++)
        for(b=a+1; b<7; b++)
            if(*(fruits+a) > *(fruits+b))
            {
                temp = *(fruits+a);
                *(fruits+a) = *(fruits+b);
                *(fruits+b) = temp;
            }

    for(x=0; x<7; x++)
        puts(fruits[x]);

    return(0);
}
```

Remarque

Est-ce de cette manière que l'on réussit à comparer des chaînes en C ?

ex1922

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

int main()
{
    char *fruits[] = {
        "abricot",
        "banane",
        "ananas",
        "pomme",
        "kaki",
        "raisin",
        "myrtille"
    };
    char *temp;
    int a,b,x;

    for(a=0; a<6; a++)
        for(b=a+1; b<7; b++)
            if(strcmp(*(fruits+a), *(fruits+b)) > 0 )      // L20
            {
                temp = *(fruits+a);
                *(fruits+a) = *(fruits+b);
                *(fruits+b) = temp;
            }

    for(x=0; x<7; x++)
        puts(fruits[x]);

    return(0);
}
```

Remarques

Pour exploiter `strcmp()`, il faut insérer le fichier `string.h` (ligne 2).

La comparaison a lieu en ligne 20. Si `strcmp()` renvoie une valeur supérieure à 0, c'est que la première chaîne contient des codes ASCII supérieurs à la seconde et doit donc passer après elle. Elles sont alors permutées. (Pour un tri inverse, remplacez `> 0` par `< 0`.)

Pour permuter les chaînes, on échange leurs adresses en lignes 22 à 24 (comme dans le tri à bulles vu plus haut dans le livre). La magie des pointeurs fait qu'il n'est pas nécessaire de permuter les caractères un à un, seulement les adresses de début.

ex1923

```
#include <stdio.h>

void remiser(float *a);

int main()
{
    float prixbase = 42.99;

    printf("L'article coute $%.2f\n", prixbase);
    remiser(&prixbase);
    printf("Apres remise, cela donne  $%.2f\n", prixbase);
    return(0);
```

```
}

void remiser(float *a)
{
    *a = *a * 0.90;             // L17
}
```

Remarques

La ligne 17 peut étonner, mais c'est parce qu'elle utilise le signe `*` pour l'opérateur d'adresse et pour la multiplication. (C'est assez fréquent en C.) Nous aurions même pu écrire :

```
*a *= 0.90;
```

Et en enlevant des espaces, cela devient franchement baroque :

```
*a*=0.90;
```

On croirait un nouvel opérateur `*a*`. En fait, il y a bien deux opérations : `*a` et `*=`.

ex1924

```
#include <stdio.h>

void remiser(float *a);

int main()
{
    float prixbase = 42.99;
    float *p;

    p = &prixbase;
    printf("L'article coute $%.2f\n", prixbase);
    remiser(p);
    printf("Apres remise, cela donne  $%.2f\n", prixbase);
    return(0);
}

void remiser(float *a)
{
    *a = *a * 0.90;
}
```

Remarques

L'ajout de la variable `p` ne modifie pas le principe du programme. La petite différence est que dorénavant c'est `p` qui est transmise à la fonction et non l'adresse de `prixbase`. Cela me permet de montrer qu'il est possible de transmettre un pointeur en entrée d'une fonction au lieu de l'adresse avec `&` d'une variable normale pointée.

De ce fait, quand une fonction attend une valeur pointeur en entrée, comprenez que c'est une adresse. Regardez le prototype suivant :

```
time_t time(time_t *tloc);
```

Dans cette déclaration, `time_t *tloc` concerne une adresse mémoire. L'argument que vous transmettez à *time()* est soit une variable pointeur de type `time_t` , soit l'adresse d'une variable normale de type `time_t` avec `&` en préfixe. Vous pouvez éventuellement indiquer la constante spéciale `NULL`, car le compilateur va la considérer comme une valeur pointeur (contenant une "non-adresse").

ex1925

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

```
void creerTab(int *a);
void montrerTab(int *a);

int main()
{
    int r[10];
    int *pr;

    pr = r;
    creerTab(pr);
    montrerTab(pr);

    return(0);
}

void creerTab(int *a)
{
    int x,r;

    srand((unsigned)time(NULL));    // L24
    for(x=0; x<10; x++)
    {
        r = rand();                 // L27
        r%=10;
        *a = r;
        a++;
    }
}

void montrerTab(int *a)
{
    int x;

    for(x=0;x<10;x++)
        printf("%d\n", *a++);
}
```

Remarques

N.d.T. : Vous aurez corrigé `sjhow()` en `show()`. Nous profitons de cette remarque pour franciser les noms des deux fonctions en `creerTab()` et `montrerTab()`.

Il faut inclure les fichiers d'en-tête `stdlib.h` et `time.h` pour pouvoir appeler `srand()` et `rand()` rspectivement dans les lignes 24 et 27.

Le modulo `r%=10` en ligne 28 contraint les valeurs stockées dans le tableau à la plage 0 à 9.

Nous n'utilisons pas ici la notation tableau/pointeur du fait que la variable `a` progresse une unité à la fois dans les deux boucles.

Voici un exemple d'affichage, mais chaque exécution sera différente :

```
8
1
2
0
5
9
0
4
6
6
```

ex1926

```
#include <stdio.h>

char *strinverser(char *input);

int main()
{
    char maChaine[64];

    printf("Saisissez du texte : ");
    fgets(maChaine, 62, stdin);
    puts(strinverser(maChaine));

    return(0);
}

char *strinverser(char *entrante)
{
    static char sortante[64];
    char *i,*o;

    i=entrante; o=sortante;

    while(*i++ != '\n')
        ;
    i--;

    while(i >= entrante)
        *o++ = *i--;
    *o = '\0';

    return(sortante);
}
```

Remarque

Vous vérifiez que la déclaration statique de la variable en ligne 18 n 'est pas un effet de manche en enlevant le mot clé static puis en recompilant. Vous verrez ce message :

```
Warning: function returns address of local variable
```

Le programme va se compiler et s'exécuter, mais il est à la merci d'un changement du contenu de l'emplacement mémoire de la chaîne. Pourquoi se créer des soucis superflus ? Utilisez `static` quand c'est nécessaire pour que la valeur soit préservée en sortie de fonction.

CHAP 20

ex2001

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

int main()
{
    int *age;
```

```
    age = (int *)malloc(sizeof(int)*1);
    if(age == NULL)
    {
        puts("Allocation mémoire impossible !");
        exit(1);
    }
    printf("Quel est votre age ? ");
    scanf("%d", age);
    printf("Vous avez %d ans.\n", *age);
    return(0);
}
```

Remarques

Dans `sizeof(int)*1` en ligne 8, l'indication de taille est superflue, pour l'instant. On aurait pu simplifier ainsi :

```
  age = (int *)malloc(sizeof(int));
```

Par défaut, le code réserve l'espace d'une valeur du type `int`, sans mention de `*1`.

Mais pourquoi cette précision ?

Cela permet d'être prêt à réserver plus d'une variable. Si j'ai besoin de cinq entiers, j'écris ceci :

```
  (int *)malloc(sizeof(int)*5);
```

C'est la syntaxe normale. En indiquant `*1`, vous prenez un bon réflexe et restez cohérent dans l'écriture.

ex2002

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

int main()
{
    float *temperature;
    char c;

    temperature = (float *)malloc(sizeof(float)*1);
    if(temperature == NULL)                          // L11
    {
        puts("Allocation mémoire impossible !");
        exit(1);
    }
    printf("Quelle est la temperature ? ");
    scanf("%f", temperature);                        // L17
    getchar();
    printf("En Celsius ou en Fahrenheit (C/F )? ");
    c = toupper(getchar());
    if(c == 'F')
        *temperature = (*temperature+459.67)*(5.0/9.0);  // L22
    else
        *temperature += 273.15;
    printf("Il fait %.1f Kelvin.\n", *temperature);

    return(0);
}
```

Remarques

La ligne 7 déclare un pointeur `float` qui est implanté en mémoire par un appel à `malloc()` en ligne 10. Vous voyez le transtypage dans `malloc()` ? L'argument profite de `sizeof` pour déterminer l'espace mémoire à réserver pour un `float`.

Les lignes 11 à 15 gèrent une éventuelle erreur mémoire.

Nous récupérons la saisie de `temperature` par appel à `scanf()` en ligne 17. L'opérateur `&` est inutile puisque `temperature` est un pointeur.

En ligne 18, nous avalons le saut de ligne ramené par l'appel à `scanf()` en ligne 17. Sans cette précaution, le `getchar()` en ligne 20 lirait le code de Entrée, ce qui ferait croire au test conditionnel en ligne 21 que la saisie a été faite en degrés Celsius. Cette portion du code permet de saisir dans les deux unités.

Les conversions de température sont faites en ligne 22 ou 24, selon l'unité choisie.

Nous affichons le résultat en Kelvin en ligne 25 avec la notation indirecte `*temperature`.

ex2003

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

int main()
{
    int *entrio;
    int x;

    entrio = (int *)malloc(sizeof(int)*3);
    if(entrio == NULL)
    {
        puts("Allocation mémoire impossible !");
        exit(1);
    }

    *entrio = 100;                              // L16
    *(entrio+1) = 200;
    *(entrio+2) = 300;

    for(x=0;x<3;x++)                            // L20
        printf("%d: %d\n", x+1, *(entrio+x));

    return(0);
}
```

Remarques

Nous demandons de l'espace mémoire pour trois variables `int` en ligne 9 en profitant de `sizeof(int)` qui renvoie la taille d'un `int` sur la machine concernée. L'espace est associé au pointeur nommé `entrio`.

Le pointeur `entrio` n'est pas un tableau, mais vous l'utilisez comme un tableau avec des indices vers les trois valeurs en lignes 16, 17 et 18.

Revoyez le Tableau 19.2 pour la syntaxe des pointeurs et tableaux.

La boucle *for* en ligne 20 fait trois tours, pour afficher les trois valeurs. L'écriture `x+1` ne sert qu'à rendre plus logique l'affichage des numéros de variables. La notation `*(entrio+x)` permet de lire une valeur entière en mémoire, comme le ferait `tablo[x]`.

Exemple d'affichage résultant :

```
1: 100
```

```
  2: 200
  3: 300
```

Rappellons qu'il n'y a pas qu'une bonne réponse. Vous avez réussi si vous avez utilisé un pointeur, appelé `malloc()` et affiché les trois valeurs 100, 200 et 300 !

ex2004

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

int main()
{
    char *entrante;

    entrante = (char *)malloc(sizeof(char)*1024);
    if(entrante == NULL)
    {
        puts("Allocation impossible ! Banzai !");
        exit(1);
    }
    puts("Saisissez quelque chose d'un peu long :");
    fgets(entrante, 1023, stdin);
    puts("Vous avez saisi :");
    printf("\"%s\"\n", entrante);

    return(0);
}
```

ex2005

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

int main()
{
    char *entrante, *entrante2;
    char *i1, *i2;

    entrante  = (char *)malloc(sizeof(char)*1024);
    entrante2 = (char *)malloc(sizeof(char)*1024);
    if(entrante == NULL || entrante2 == NULL)
    {
        puts("Allocation impossible ! Banzai !");
        exit(1);
    }

    puts("Saisissez quelque chose d'un peu long :");
    fgets(entrante, 1023, stdin);

    puts("Copie du contenu du tampon...");
    i1 = entrante; i2 = entrante2;
    while(*i1 != '\n')                          // L22
        *i2++=*i1++;                            // L23

    printf("Texte original :\n\"%s\"\n", entrante);
    printf("Texte copie :\n\"%s\"\n", entrante2);

    return(0);
}
```

Remarques

En ligne 7 sont déclarés deux pointeurs `char` `i1` et `i2` qui vont servir à copier la chaîne. Cela me pemet de préserver l'adresse de départ de `entrante` et `entrante2`.

La copie se déroule à partir de la ligne 21 avec l'initialisation de `i1` et `i2` (deux instructions sur une ligne).

La boucle en ligne 22 copie le texte et définit la condition de sortie : la détection dans `*i1` du saut de ligne.

La ligne 23 est un régal de compacité très apprécié des programmeurs C. Rappelons que l'opérateur `*` est plus intime par rapport à la variable que celui d'incrémentation. Le caractère est donc copié de `i1` vers `i2`, puis les deux pointeurs (donc les adresses) sont incrémentés.

Cette ligne 23 peut s'écrire de façon plus aérée :

```
while(*i1 != '\n')
{
    *i2=*i1;
    i1++;
    i2++;
}
```

Les deux approches sont valables. Vous pouvez même avoir décidé de faire la copie autrement.

Exemple d'affichage résultant (saisie en gras) :

```
Saisissez quelque chose d'un peu long :
Combien de nuits faut-il attendre avant le jour?
Copie du contenu du tampon...
Texte original :
"Combien de nuits faut-il attendre avant le jour?
"
Texte copie :
"Combien de nuits faut-il attendre avant le jour?"
```

Vous constatez que cette approche permet d'éviter de copier le saut de ligne parasite ajouté par `fgets()`.

ex2006

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

int main()
{
    char *entrante, *entrante2;
    char *i1, *i2;

    entrante  = (char *)malloc(sizeof(char)*1024);
    entrante2 = (char *)malloc(sizeof(char)*1024);
    if(entrante == NULL || entrante2 == NULL)
    {
        puts("Allocation impossible ! Banzai !");
        exit(1);
    }

    puts("Saisissez quelque chose d'un peu long :");
    fgets(entrante, 1023, stdin);

    puts("Copie du contenu du tampon...");
```

```
    i1 = entrante; i2 = entrante2;
    while(*i1 != '\n')
    {
        switch(*i1)
        {
            case 'a':
            case 'A':
            case 'e':
            case 'E':
            case 'i':
            case 'I':
            case 'o':
            case 'O':
            case 'u':
            case 'U':
                *i2++='@';      // L36
                i1++;
                break;
            default:
                *i2++=*i1++;
        }
    }

    printf("Texte original :\n\"%s\"\n", entrante);
    printf("Texte copie :\n\"%s\"\n", entrante2);

    return(0);
}
```

Remarques

Le noyau du programme est la structure de test `switch/case`. Le but est d'intercepter toutes les voyelles dans les branches `case`. Chaque voyelle est remplacée par l'arobase `@` en l'insérant dans `i2` et le pointeur `i1` est incrémenté pour passer à la lettre suivante (lignes 36 et 37). La branche par défaut est la même que dans la solution **ex2005**.

Exemple d'affichage résultant (saisie en gras) :

```
Saisissez quelque chose d'un peu long :
Combien de nuits faut-il attendre avant le jour?
Copie du contenu du tampon...
Texte original :
"Combien de nuits faut-il attendre avant le jour?
"
Texte copie :
"C@mb@@n d@ n@@ts f@@t-@l @tt@ndr@ @v@nt l@ j@@r?"
```

ex2007

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

int main()
{
    char *entrante;
    int longue;

    entrante = (char *)malloc(sizeof(char)*1024);
    if(entrante == NULL)
    {
        puts("Allocation impossible ! Banzai !");
```

```
        exit(1);
    }
    puts("Saisissez quelque chose d'un peu long :");
    fgets(entrante, 1023, stdin);
    longue = strlen(entrante);
    if(realloc(entrante, sizeof(char)*(longue+1)) == NULL)
    {
        puts("Allocation impossible ! Banzai !");
        exit(1);
    }
    puts("Reallocation correcte.");
    puts("Vous avez saisi :");
    printf("\"%s\"\n", entrante);

    return(0);
}
```

ex2008

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

int main()
{
    int *age;

    age = (int *)malloc(sizeof(int)*1);
    if(age==NULL)
    {
        puts("Plus de memoire ?");
        exit(1);
    }
    printf("Votre age en ans ? ");
    scanf("%d", age);
    *age *= 365;
    printf("Vous avez vu plus de %d jours !\n", *age);
    free(age);

    return(0);
}
```

Remarques

Dans les premiers temps de l'informatique, on utilisait une combinaison de `malloc()` et de `free()` pour connaître la quantité de mémoire restante et en libérer une partie. Avec les énormes espaces mémoire (RAM) des machines actuelles et le mécanisme de mémoire virtuelle, cette pratique n'a plus cours.

Vous pouvez néanmoins interroger la machine pour savoir combien il reste de mémoire disponible en faisant un appel à l'interface API du système d'exploitation concerné. C'est une opération dont la description dépasse l'objectif de ce livre, d'autant qu'elle est différente selon le système. Sachez que c'est possible et sans utiliser ni `malloc()`, ni `free()`.

ex2009

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

int main()
{
```

```
  struct actions {
    char  symbole[5];
    int   quantite;
    float cours;
  };
  struct actions *invest;

/* Creation de la structure en memoire */
  invest=(struct actions *)malloc(sizeof(struct actions));   // L15
  if(invest == NULL)
  {
    puts("Erreur malloc()");
    exit(1);
  }

/* Affectation des valeurs */
  strcpy(invest->symbole, "GOOG");            // L23
  invest->quantite = 100;
  invest->cours = 801.19;

/* Affichage */
  puts("Portefeuille d'actions");
  printf("Symbole\t  Qte\tCours\tValeur\n");
  printf("%-6s\t%5d\t%.2f\t%.2f\n",\          // L31
      invest->symbole,
      invest->quantite,
      invest->cours,
      invest->quantite*invest->cours);
  return(0);
}
```

Remarques

Nous ne créons pas une variable structure en début de source (ligne 12, mais un pointeur nommé `invest` ciblant une structure de type `actions`.

En ligne 15, nous réservons de l'espace pour une structure `actions` en stockant l'adresse de début dans `invest`.

Dans les lignes 23 à 25, nous peuplons la structure avec l'opérateur `->` et pas l'opérateur point `.` car `invest` est un pointeur.

L'affichage des valeurs des lignes 30 à 34 utilise aussi l'opérateur `->` pour lire les valeurs en mémoire.

Vous pouvez référencer une variable pointeur sur stucture avec la notation point, mais cela désigne l'adresse et pas la valeur lue à cette adresse. Ainsi, `invest.symbole` est une adresse, alors que `invest->symbole` est la valeur à cette adresse.

ex2010

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

int main()
{
  struct actions {
    char symbole[5];
    int quantite;
    float cours;
  struct actions *asuiv;
  };
  struct actions *aprems;
```

```
  struct actions *acour;
  struct actions *anouv;

/* Creation structure */
  aprems = (struct actions *)malloc(sizeof(struct actions));
  if(aprems==NULL)
  {
    puts("Erreur malloc()");
    exit(1);
  }

/* Remplissage */
  acour = aprems;
  strcpy(acour->symbole,"GOOG");
  acour->quantite = 100;
  acour->cours = 801.19;
  acour->asuiv = NULL;

  anouv = (struct actions *)malloc(sizeof(struct actions));
  if(anouv==NULL)
  {
    puts("Autre erreur malloc()");
    exit(1);
  }
  acour->asuiv = anouv;
  acour = anouv;
  strcpy(acour->symbole,"MSFT");
  acour->quantite = 100;
  acour->cours = 28.77;
  acour->asuiv = NULL;

/* Affichage */
  puts("Portefeuille");
  printf("Symbole\tQte\tCours\tValeur\n");
  acour = aprems;
  printf("%-6s\t%5d\t%.2f\t%.2f\n",\
      acour->symbole,
      acour->quantite,
      acour->cours,
      acour->quantite*acour->cours);
  acour = acour->asuiv;
  printf("%-6s\t%5d\t%.2f\t%.2f\n",\
      acour->symbole,
      acour->quantite,
      acour->cours,
      acour->quantite*acour->cours);
  return(0);
}
```

ex2011 (Listing 20.7)

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

#define MEMBRES 5

struct actions {
  char symbole[5];
  int quantite;
```

```
  float cours;
  struct actions *asuiv;
  };
struct actions *aprems;
struct actions *acour;
struct actions *anouv;

struct actions *creer_struc(void);
void remplir_struc(struct actions *a,int c);
void montrer_struc(struct actions *a);

int main()
{
  int x;
  for(x=0;x<MEMBRES;x++)
  {
    if(x==0)
    {
      aprems=creer_struc();
      acour=aprems;
    }
    else
    {
      anouv = creer_struc();
      acour->asuiv = anouv;
      acour = anouv;
    }
    remplir_struc(acour,x+1);
  }
  acour->asuiv=NULL;

/* Affichage */
  puts("Portefeuille");
  printf("Symbole\t   Qte\tCours\tValeur\n");
  acour = aprems;
  while(acour)
  {
    montrer_struc(acour);
    acour=acour->asuiv;
  }
  return(0);
}

struct actions *creer_struc(void)
{
  struct actions *a;

  a=(struct actions *)malloc(sizeof(struct actions));
  if(a==NULL)
  {
    puts("Erreur malloc()");
    exit(1);
  }
  return(a);
}

void remplir_struc(struct actions *a,int c)
{
  printf("Membre #%d/%d:\n",c,MEMBRES);
  printf("Symbole: ");
  scanf("%s", a->symbole);
```

```
  printf("Nombre d'actions : ");
  scanf("%d", &a->quantite);
  printf("Cours : ");
  scanf("%f", &a->cours);
}

void montrer_struc(struct actions *a)
{
  printf("%-6s\t%5d\t%.2f\t%.2f\n",\
    a->symbole,
    a->quantite,
    a->cours,
    a->quantite*a->cours);
}
```

ex2012 (Listing 20.8)

```
/* Programme interactif */
/* Dan Gookin, VF par O.E. */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

struct stypik {
  int maval;
  struct stypik *asuiv;
};
struct stypik *aprems;
struct stypik *acour;
struct stypik *anouv;

int menu(void);
void ajouter(void);
void montrer(void);
void supprimer(void);
struct stypik *creer(void);

/* La fonction main() ne se charge que de la saisie.
   Le reste est dans les fonctions. */
int main()
{
  int choixmenu = '\0'; /* Initialise la boucle while */
  aprems=NULL;

  while(choixmenu!='Q')
  {
    choixmenu=menu();
    switch (choixmenu)
    {
      case 'M':
        montrer();
        break;
      case 'A':
        ajouter();
        break;
      case 'S':
        supprimer();
        break;
      case 'Q':
        break;
```

```
    default:
      break;
    }
  }
  return(0);
}

/* Affiche le menu et collecte le choix */
int menu(void)
{
  int ch;

  printf("M)ontrer, A)jouter, S)upprimer, Q)uitter: ");
  ch=getchar();
  while(getchar() != '\n') /* Ignore la saisie superflue */
    ;
  return(toupper(ch));
}

/* Ajoute un membre en fin de liste */
void ajouter(void)
{
  if(aprems == NULL) /* Cas unique de aprems */
  {
    aprems = creer();
    acour  = aprems;
  }
  else /* Cherche le dernier */
  {
    acour = aprems;
    while(acour->asuiv) /* Dernier == NULL */
      acour = acour->asuiv;
    anouv = creer();
    acour->asuiv = anouv; /* Actualisation lien */
    acour = anouv;
  }
  printf("Indiquez une valeur numerique : ");
  scanf("%d", &acour->maval);
  acour->asuiv = NULL;
  while(getchar() != '\n') /* Ignore la saisie superflue */
    ;
}

/* Affiche tous les enregs de la liste */
void montrer(void)
{
  int nbrenreg = 1;
  if(aprems == NULL) /* Liste vide */
  {
    puts("Rien a afficher");
    return;
  }
  puts("Affichage complet :");
  acour = aprems;
  while(acour) /* Dernier == NULL */
  {
    printf("Enregistrement %d: %d\n", nbrenreg, acour->maval);
    acour = acour->asuiv;
    nbrenreg++;
  }
}
```

```
/* Supprime un enreg de la liste */
void supprimer(void)
{
  struct stypik *eprec; /* Sauve l'enreg d'avant */
  int r,c;

  if(aprems == NULL) /* Teste si liste vide */
  {
    puts("Aucun enregistrement !");
    return;
  }
  puts("Choisissez quel enreg. supprimer :");
  montrer();
  printf("Enregistrement : ");
  scanf("%d",&r);
  while(getchar() != '\n') /* Ignore la saisie superflue */
    ;
  c=1;
  acour = aprems;
  eprec = NULL; /* Pas de precedent du 1er */
  while(c != r)
  {
    if(acour == NULL) /* Teste si 'r' dans la plage */
    {
      puts("Enreg introuvable");
      return;
    }
    eprec = acour;
    acour = acour->asuiv;
    c++;
  }
  if(eprec == NULL)       /* Cas unique du 1er enreg. */
    aprems = acour->asuiv;
  else                    /* Raccorde precedent et suivant*/
    eprec->asuiv = acour->asuiv;
  printf("L'enregistrement %d n'existe plus.\n",r);
  free(acour);            /* Restitue memoire */
}

/* Construit une structure vide et renvoie son adresse */
struct stypik *creer(void)
{
  struct stypik *a;
  a = (struct stypik *)malloc(sizeof(struct stypik));
  if(a == NULL)
  {
    puts("Erreur malloc()");
    exit(1);
  }
  return(a);
}
```

CHAP 21

ex2101 (Listing 21.1)

```
#include <stdio.h>
#include <time.h>
```

```
int main()
{
    time_t tictoc;

    time(&tictoc);
    printf("Il est exactement %ld\n", tictoc);
    return(0);
}
```

Remarques

Exemple d'affichage résultant :

```
Il est exactement 1393344630
```

La valeur de type `time_t` renvoyée varie bien sûr sans cesse. Ici, il s'agissait du 25 février 2014 à 17h12.

ex2102

```
#include <stdio.h>
#include <time.h>

int main()
{
    time_t tictoc;

    tictoc = time(NULL);
    printf("Il est exactement %ld\n", tictoc);
    return(0);
}
```

Remarque

Vous pouvez exploiter la valeur renvoyée par *time()*. de plusieurs façons. Puisque la fonction doit recevoir un pointeur, autant travailler directement sur la valeur de type `time_t` renvoyée, en fournissant `NULL` entre parenthèses. Vous pouvez aussi fournir l'adresse d'une valeur de type `time_t`. Veuillez cependant à ne pas confondre types, valeurs et adresses.

ex2103

```
#include <stdio.h>
#include <time.h>

int main()
{
    time_t tictoc;

    time(&tictoc);
    printf("Il est exactement %ld\n", tictoc);
    time(&tictoc);
    printf("Il est exactement %ld\n", tictoc);
    return(0);
}
```

Remarques

Exemple d'exécution :

```
Il est exactement 1397557524
Il est exactement 1397557524
```

Mon ordinateur est trop performant ! Voici une vingtaine d'années, les deux chiffres étaient franchement différents.

Vous pouvez ajouter un appel à `getchar()` pour provoquer une pause afin de rendre la seconde valeur différente.

ex2104

```
#include <stdio.h>
#include <time.h>

int main()
{
    time_t tictoc;
    int x;

    for(x=0; x<20; x++)
    {
        time(&tictoc);
        printf("%2d:%ld\n", x+1, tictoc);
    }
    return(0);
}
```

Remarques

Pour voir changer la valeur, et donc faire durer le programme plus d'une seconde, il faut demander au moins 20000 tours de boucle sur un Intel Core i7 (à peu près). Cela ne signifie pas que ce nombre de tours consomme une seconde d'exécution. Loin de là, car l'essentiel du temps passé l'est dans l'affichage des données dans la fenêtre de terminal.

ex2105 (Listing 21.2)

```
#include <stdio.h>
#include <time.h>

int main()
{
    time_t tictoc;

    time(&tictoc);
    printf("Il est exactement %s\n", ctime(&tictoc));   // L09
    return(0);
}
```

Remarques

Comme sa collègue `time()`, la fonction `ctime()` attend en entrée l'adresse d'une variable de type `time_t` (ligne 9).

ex2106 (Listing 21.3)

```
#include <stdio.h>
#include <time.h>

int main()
{
    time_t tictoc;
    struct tm *present;

    time(&tictoc);
    present = localtime(&tictoc);
    printf("Nous sommes le %d/%d/%d\n",
        present->tm_mday,
        present->tm_mon,
        present->tm_year);
    return(0);
```

```
}
```

ex2107

```
#include <stdio.h>
#include <time.h>

int main()
{
    time_t tictoc;
    struct tm *present;

    time(&tictoc);
    present = localtime(&tictoc);
    printf("Nous sommes le %d/%d/%d\n",
        present->tm_mday,
        present->tm_mon+1,
        present->tm_year+1900);             // L14
    return(0);
}
```

Remarque

La fonction `localtime()` peuple la structure de type `tm` avec les valeurs des composantes de temps, mais celles du mois et de l'année doivent être corrigées. Le mois de janvier vaut zéro ; il faut donc ajouter un au membre `tm_mon`. L'année commençant à 1900, il faut aussi ajuster le membre `tm_year`.

ex2108

```
#include <stdio.h>
#include <time.h>

int main()
{
    time_t tictoc;
    struct tm *present;

    time(&tictoc);
    present = localtime(&tictoc);
    printf("A present, il est %d:%d:%d\n",
        present->tm_hour,
        present->tm_min,
        present->tm_sec);
    return(0);
}
```

Remarques

N.d.T. : j'ai retouché le message pour qu'il colle avec une heure plutôt qu'une date.

Exemple d'affichage résultant :

```
A present, il est 22:21:18
```

Nous sommes au format européen sur 24 heures.

L'absence de zéro initial peut donner un affichage étonnant :

```
A present, il est 8:3:9
```

Sauriez-vous améliorer l'affichage en faisant ajouter des zéros préfixes quand c'est nécessaire ?

ex2109

```
#include <stdio.h>
#include <time.h>

int main()
{
    time_t tictoc;
    struct tm *present;
    int heure;
    char *am = "a.m.";
    char *pm = "p.m.";
    char *m;

    time(&tictoc);
    present = localtime(&tictoc);
    heure = present->tm_hour;
    if(heure == 12)                    // L16
    {
        m=pm;
    }
    else if(heure > 12)                // L20
    {
        heure-=12;
        m=pm;
    }
    else
    {
        m=am;
    }
    printf("A present, il est %d:%02d:%02d %s\n",    // L29
        heure,
        present->tm_min,
        present->tm_sec,
        m);
    return(0);
}
```

Remarques

Il y a plusieurs manières de répondre à cet exercice. Le code ci-dessus en est une.

Voyons d'abord comment j'ai formaté l'affichage des minutes et secondes en ligne 29. Le formateur `%02d` fait afficher un entier sur deux chiffres avec un zéro éventuel. C'est ce que je demandais de faire pour **ex2108**.

J'ai choisi de préparer deux chaînes pour les mentions AM et PM (lignes 9 et 10). Le pointeur `char` nommé `m` reçoit la bonne chaîne plus loin.

Il faut savoir si l'heure est supérieur à 11 (les heures 00 à 11 sont avant midi, donc AM). Mais il y a un problème à gérer au format 12 heures : 12:00 le midi est le début d'aprtès-midi (donc PM, ce que nous traitons par un bloc `if / else if / else`.

La ligne 16 gère l'heure de midi.

La branche `else if` gère les heures après midi en retranchant 12 à `heure`.

La branche `else` (ligne 25) ajoute le suffixe AM pour les heures du matin.

Nous aurions pu exploiter directement la variable `present->tm_hour` dans les conditions, mais comme il faut pouvoir modifier la valeur plus loin, une variable de travail est plus pratique (`heure`).

Si vous voulez, vous pouvez peaufiner l'exemple en faisant en sorte que minuit soit affiché en 12:00:00 au lieu de 00:00:00. Bon courage !

ex2110

```
#include <stdio.h>
#include <time.h>

int main()
{
    time_t tictoc;
    struct tm *present;
    char *joursem[] = {                          // L08
        "Dimanche",
        "Lundi",
        "Mardi",
        "Mercredi",
        "Jeudi",
        "Vendredi",
        "Samedi"
    };

    time(&tictoc);
    present = localtime(&tictoc);
    printf("Nous sommes le %s %d/%d/%d\n",
        *(joursem+present->tm_wday),           // L21
        present->tm_mday,
        present->tm_mon,
        present->tm_year+1900);
    return(0);
}
```

Remarques

Le tableau de pointeurs sur les noms des jours est défini en ligne 8. Notez que l'ordre des chaînes est important ! L'élément prédéfini `tm_wday` numérote à partir de zéro pour le dimanche (début de semaine anglo-saxonne).

Le jour est affiché en ligne 21 via la notation pointeur `*(p+x)`, similaire à `array[x]`. Voici d'ailleurs l'accès équivalent avec un tableau :

```
joursem[present->tm_wday],
```

Vous n'avez pas oublié de corriger l'année (ligne 24).

Pour le mois (ligne 23) aucune correction n'est nécessaire car `tm_mon` commence à zéro pour janvier, ce qui nous convient pour le premier élément d'indice zéro.

Exemple d'affichage résultant :

```
Nous sommes le Mardi 25/2/2014
```

ex2111 (Listing 21.4)

```
#include <stdio.h>
#include <time.h>

int main()
{
    time_t present, avant;
    float  delai = 0.0;

    time(&avant);
    puts("Commencer");
    while(delai < 1)
```

```
    {
        time(&present);
        delai = difftime(present, avant);
        printf("%f\r", delai);
    }
    puts("\nStopper");
    return(0);
}
```

Remarques

Exemple d'affichage résultant :

```
Commencer
1.000000
Stopper
```

Sur ma machine, la valeur n'affiche que de secondes entières. La boucle fait donc une pause pendant une seconde.

Modifiez le code en indiquant une valeur supérieure à 1 en ligne 11.

Vous disposez aussi de la fonction *difftime()* pour réaliser des calculs temporels, mais prenez alors le temps de bien comprendre le format temporel natif de votre système.

N.d.T. : notez l'utilisation astucieuse du métacaractère `\r` pour revenir au début de la même ligne.

CHAP 22

ex2201 (Listing 22.1)

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

int main()
{
    FILE *fh;

    fh = fopen("hello.txt", "w");
    if(fh == NULL)
    {
        puts("Ouverture du fichier impossible !");
        exit(1);
    }
    fprintf(fh,"Je laisse une trace.\n");
    fclose(fh);
    return(0);
}
```

Remarques

Le fichier créé porte le nom `hello.txt`. Vous le trouverez dans la racine relative du projet (là d'où partent les sous-dossiers `bin` et `obj`).

L'élément nommé `FILE` est une définition de type (*typedef*) qui se trouve dans le fichier `stdio.h`. `FILE` C'est une structure pointeur utilisée en permanence pour accéder à un fichier. Considérez l'élément comme un identifiant d'accès à un fichier (un *handle*).

Appelez toujours `fclose()` pour refermer le fichier après usage. Bien sûr, le système sait faire le ménage à la place de votre programme si ce dernier s'est arrêté inopinément. Mais si vous ne fermez

pas votre fichier, il est possible que des données n'y aient pas encore été récrites en fin d'exécution ; elles seront donc perdues.

ex2202 (Listing 22.2)

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

int main()
{
    FILE *fh;
    int ch;

    fh = fopen("hello.txt", "r");
    if(fh == NULL)
    {
        puts("Ouverture du fichier impossible !");
        exit(1);
    }
    while((ch=fgetc(fh)) != EOF)
        putchar(ch);
    fclose(fh);
    return(0);
}
```

Remarque

Si vous avez créé un nouveau projet pour cet exercice, il ne trouvera pas le fichier `hello.txt` qu'il doit lire, car il se trouve dans le dossier du projet précédent (voir les remarques de **ex2201**).

ex2203

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

int main()
{
    FILE *fh;

    fh = fopen("hello.txt", "w");
    if(fh == NULL)
    {
        puts("Ouverture du fichier impossible !");
        exit(1);
    }
    fprintf(fh,"Je laisse une trace.\n");
    fputs("C'est mon programme qui produit ce contenu.\n", fh);
    fclose(fh);
    return(0);
}
```

ex2204

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

int main()
{
    FILE *fh;
    char buffer[64];
```

```
    fh = fopen("hello.txt", "r");
    if(fh == NULL)
    {
        puts("Ouverture du fichier impossible !");
        exit(1);
    }
    while(fgets(buffer, 64, fh))
        printf("%s", buffer);
    fclose(fh);
    return(0);
}
```

Remarques

Comme indiqué dans le livre, la lecture d'une chaîne se termine par le zéro terminal `\0`. Mais la lecture de ce caractère ne provoque pas la fin de la saisie, seulement celle de la chaîne en cours. Autrement dit, lors de la lecture d'un fichier, le signe `\0` ne marque pas la fin du fichier.

Dans l'appel à `fgets()`, le tampon récepteur `buffer` n'est pas aussi contraignant pour la lecture depuis un fichier que pour la lecture d'une chaîne au clavier. La limite et les craintes de débordement disparaissent. En effet, `fgets()` va alimenter le tampon à son rythme. Le texte qui reste à copier dans le tampon reste sagement dans le flux (*stream*) qu'un autre appel à `fgets()` aille le chercher. Si la rupture tombe au milieu d'une phrase, les deux moitiés sont raccordées automatiquement en sortie.

Pour la taille de bloc (deuxième paramètre de `fgets()`), il est conseillé de travailler avec un multiple de 2 pour coïncider avec la taille unitaire des opérations fichiers sur tous les ordinateurs, ce qu'on appelle les nombres sacrés de l'informatique. Pour de petits fichiers comme `hello.txt`, j'ai choisi la valeur 64. Pour de grands fichiers, les valeurs 512 ou 1024 sont appropriés.

Les nombres sacrés de l'informatique sont le puissances de 2 : 4, 8, 16, 32, 64, 128, 256, 512, etc.

N.d.T. : Vous savez d'où proviennent les noms Nintendo 64 et Win32.

ex2205 (Listing 22.4)

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

int main()
{
    FILE *fh;

    fh=fopen("hello.txt", "a");
    if(fh == NULL)
    {
        puts("Ouverture du fichier impossible !");
        exit(1);
    }
    fprintf(fh,"Texte apparu plus tard\n");
    fclose(fh);
    return(0);
}
```

Remarques

Pour résoudre le problème du fichier `hello.txt` introuvable, vous pouvez le copier depuis le sous-dossier dans lequel il a été généré ou bien copier l'exécutable de cet exercice dans ce sous-dossier.

ex2206 (Listing 22.5)

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

int main()
```

```
{
    FILE *handle;
    int scoremax;

    handle = fopen("scores.dat", "w");
    if(!handle)
    {
        puts("Erreur fichier!");
        exit(1);
    }
    printf("Indiquez votre meilleur score : ");
    scanf("%d", &scoremax);
    fprintf(handle, "%d", scoremax);
    fclose(handle);
    puts("Score sauf");
    return(0);
}
```

ex2207

```
#include <stdio.h>
#include &lg
stdlib.h>

int main()
{
    FILE *handle;
    int scoremax;

    handle = fopen("scores.dat","w");
    if(!handle)
    {
        puts("Erreur fichier!");
        exit(1);
    }
    printf("Indiquez votre meilleur score : ");
    scanf("%d", &scoremax);
    fwrite(&scoremax, sizeof(int), 1, handle);
    fclose(handle);
    puts("Score sauf");
    return(0);
}
```

Remarques

La fonction `fwrite()` stocke dans un fichier la valeur de type `int` qu'elle trouve dans la variable `scoremax`. La valeur est écrite comme un entier, donc quatre octets (sur ma machine).

Si le programme ne trouve pas le fichier nommé `scores.dat`, copiez-le dans le dossier du programme exécutable de l'exercice. Le chemin d'accès est du style `../BegC4D/ex2207/bin/Release/`

ex2208

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

int main()
{
    FILE *handle;
    int scoremax[5];
    int x;
```

```
    handle = fopen("scores.dat", "w");
    if(!handle)
    {
        puts("Erreur fichier!");
        exit(1);
    }
    for(x=0; x<5; x++)                          // L16
    {
        printf("Meilleur score #%d? ", x+1);
        scanf("%d", &scoremax[x]);
    }
    fwrite(&scoremax, sizeof(int), 5, handle);
    fclose(handle);
    puts("Scores saufs");
    return(0);
}
```

Remarques

Le code demande à l'utilisateur de saisir les cinq meilleurs scores dans la boucle en ligne 16 et les stocke dans le tableau `scoremax` en ligne 19. Nous référençons un élément de tableau : l'opérateur d'indirection `&` est donc indispensable.

En ligne 21, nous écrivons les scores dans le fichier `scores.dat` en une fois.

Une autre approche consiste à écrire chaque valeur dès qu'elle est saisie. Il suffit pour cela d'ajouter l'instruction suivante dans la boucle juste après l'appel à `scanf()` :

```
fwrite(&scoremax[x], sizeof(int), 1, handle);
```

Mais je voulais vous montrer comment écrire d'un coup dans un fichier tout un tableau de valeurs avec un seul appel à `fwrite()`.

ex2209 (Listing 22.6)

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

int main()
{
    FILE *handle;
    int scoremax[5];
    int x;

    handle = fopen("scores.dat","r");
    if(!handle)
    {
        puts("Erreur fichier!");
        exit(1);
    }
    fread(scoremax,sizeof(int),5,handle);
    fclose(handle);
    for(x=0;x<5;x++)
        printf("Meilleur score #%d: %d\n",x+1,scoremax[x]);
    return(0);
}
```

Remarques

Cet exemple relit les cinq valeurs de type `int` sans se soucier du fait qu'elles ont été écrites en une fois ou une par une.

L'opérateur `&` n'est pas nécessaire dans l'appel à `fread()` en ligne 16 parce que `scoremax` est un tableau. Pour des variables simples (atomiques), il faut ajouter `&` en préfixe pour obtenir son adresse.

La fonction `fread()` sait lire les valeurs une par une. Voici à quoi ressemblerait l'appel dans une boucle :

```
fread(&scoremax[x], sizeof(int), 1, handle);
```

L'opérateur `&` est requis puisque nous manipulons des éléments de tableau.

ex2210 (Listing 22.7)

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

int main()
{
    char nomfic[255];
    FILE *dumpme;
    int x,c;

    printf("Nom du fichier : ");
    scanf("%s", nomfic);
    dumpme = fopen(nomfic, "r");
    if(!dumpme)
    {
        printf("Ouverture impossible de '%s'\n", nomfic);
        exit(1);
    }
    x=0;
    while( (c = fgetc(dumpme)) != EOF)
    {
        printf("%02X ",c);
        x++;
        if(!(x%16))
            putchar('\n');
    }
    putchar('\n');
    fclose(dumpme);
    return(0);
}
```

Remarques

Exemple d'affichage à partir du fichier `scores.dat` :

```
Nom du fichier : scores.dat
10 27 00 00 84 03 00 00 20 03 00 00 EF 02 00 00
06 00 00 00
```

ex2211

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

int main(int argc, char *argv[])
{
    char nomfic[255];
    FILE *dumpme;
    int x,c;

    if(argc > 1)
    {
        strcpy(nomfic,argv[1]);
```

```
    }
    else
    {
        printf("Nom du fichier : ");
        scanf("%s",nomfic);
    }
    dumpme = fopen(nomfic, "r");
    if(!dumpme)
    {
        printf("Ouverture impossible de '%s'\n", nomfic);
        exit(1);
    }
    x=0;
    while( (c = fgetc(dumpme)) != EOF)
    {
        printf("%02X ", c);
        x++;
        if(!(x%16))
            putchar('\n');
    }
    putchar('\n');
    fclose(dumpme);
    return(0);
}
```

Remarques

Dans ce programme, nous attendons les paramètres de ligne de commande, ce que vous constatez dans la déclaration de la fonction `main()` qui présente les arguments (ligne 5).

Le bloc conditionnel `if/else` à partir de la ligne 11 sert à vérifier si un nom de fichier a été fourni au lancement du programme. S'il n'y en a pas, nous demandons d'en saisir un. La suite est similaire au code de **ex2210**.

Si un nom de fichier avait été indiqué sur la ligne de commande en premier argument (deuxième si on compte le nom du programme exécutable), la chaîne du nom est dans `argv[1]`. Elle est copiée dans le tampon `nomfic` en ligne 13.

ex2212 (Listing 22.8)

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

int main()
{
    struct filmo {
        char acteur[32];
        int  annee;
        char titre[32];
    };
    struct filmo bond;
    FILE *a007;

    strcpy(bond.acteur, "Sean Connery");
    bond.annee = 1962;
    strcpy(bond.titre, "Dr. No");

    a007 = fopen("bond.db", "w");
    if(!a007)
    {
        puts("SPECTRE gagne !");
```

```
        exit(1);
    }
    fwrite(&bond, sizeof(struct filmo), 1, a007);
    fclose(a007);
    puts("Enregistrement ecrit");

    return(0);
}
```

Remarque

La fonction `fwrite()` en ligne 25 transmet l'adresse de la variable `bond` grâce à l'opérateur `&`. Vous constatez que l'opérateur `sizeof` obtient la valeur adéquate directement depuis la structure `filmo`, pas depuis la variable `bond`. Citez toujours la structure par une variable qui en dérive avec `sizeof`.

ex2213

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

int main()
{
    struct filmo {
        char acteur[32];
        int  annee;
        char titre[32];
    };
    struct filmo bond2;
    struct filmo bond3;
    FILE *a007;

    strcpy(bond2.acteur, "Roger Moore");
    bond2.annee = 1973;
    strcpy(bond2.titre, "Vivre et laisser mourir");
    strcpy(bond3.acteur, "Pierce Brosnan");
    bond3.annee = 1995;
    strcpy(bond3.titre, "GoldenEye");

    a007 = fopen("bond.db", "a");
    if(!a007)
    {
        puts("SPECTRE gagne !");
        exit(1);
    }
    fwrite(&bond2, sizeof(struct filmo), 1, a007);   // L29
    fwrite(&bond3, sizeof(struct filmo), 1, a007);
    fclose(a007);
    puts("Enregistrements ecrits");

    return(0);
}
```

Remarques

En ligne 23, la fonction `fopen()` stipule le mode d'ouverture `"a"` pour ajouter des données au fichier `bond.db` existant.

Ici, nous n'écrivons pas un tableau. C'est pourquoi il faut deux appels à `fwrite()` (lignes 29 et 30) pour stocker les deux structures dans le fichier. Un tableau de structures aurait été écrit en une opération.

ex2214 (Listing 22.9)

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

int main()
{
    struct filmo {
        char acteur[32];
        int  annee;
        char titre[32];
    };
    struct filmo bond;
    FILE *a007;

    a007 = fopen("bond.db", "r");
    if(!a007)
    {
        puts("SPECTRE gagne !");
        exit(1);
    }
    while(fread(&bond, sizeof(struct filmo), 1, a007))
        printf("%s\t%d\t%s\n",
            bond.acteur,
            bond.annee,
            bond.titre);
    fclose(a007);

    return(0);
}
```

Remarques

Assurez-vous de toujours indiquer la taille de la structure, pas celle de la variable qui en dérive lorsque vous lisez un fichier en accès direct ! Ici, la structure porte le nom `filmo`, alors que la variable se nomme `bond`. De nombreux débutants font l'erreur suivante avec `fread()` (ligne 21) :

```
fread(&bond, sizeof(bond), 1, a007)
```

Cette instruction est incorrecte ! La taille ne peut être connue exactement qu'à partir de la structure, car la variable (son instance) peut avoir une taille inférieure. Dès que vous avez un souci de lecture ou écriture fichier en accès direct, vérifiez ce point en premier.

ex2215 (Listing 22.10)

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

int main()
{
    struct filmo {
        char acteur[32];
        int annee;
        char titre[32];
    };
    struct filmo bond;
    FILE *a007;
    int count = 0;

    a007 = fopen("bond.db", "r");
```

```
    if(!a007)
    {
        puts("SPECTRE gagne !");
        exit(1);
    }
    while(fread(&bond, sizeof(struct filmo), 1, a007))
    {
        printf("%s\t%d\t%s\n",
            bond.acteur,
            bond.annee,
            bond.titre);
        if(ftell(a007) > sizeof(struct filmo))
            rewind(a007);
        count++;
        if(count > 10) break;
    }
    fclose(a007);

    return(0);
}
```

Remarque

Exemple d'affichage (à supposer que le fichier `bond.db` existe) :

```
Sean Connery 1962 Dr. No
Roger Moore 1973 Vivre et laisser mourir
Sean Connery 1962 Dr. No
Roger Moore 1973 Vivre et laisser mourir
Sean Connery 1962 Dr. No
Roger Moore 1973 Vivre et laisser mourir
Sean Connery 1962 Dr. No
Roger Moore 1973 Vivre et laisser mourir
Sean Connery 1962 Dr. No
Roger Moore 1973 Vivre et laisser mourir
Sean Connery 1962 Dr. No
```

ex2216 (Listing 22.11)

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

int main()
{
    struct filmo {
        char acteur[32];
        int  annee;
        char titre[32];
    };
    struct filmo bond;
    FILE *a007;

    a007 = fopen("bond.db", "r");
    if(!a007)
    {
        puts("SPECTRE gagne !");
        exit(1);
    }
    fseek(a007, sizeof(struct filmo)*1, SEEK_SET);
    fread(&bond, sizeof(struct filmo), 1, a007);
```

```
    printf("%s\t%d\t%s\n",
        bond.acteur,
        bond.annee,
        bond.titre);
    fclose(a007);

    return(0);
}
```

Remarques

Exemple d'affichage (à supposer que le fichier `bond.db` existe et contient des données) :

```
  Roger Moore 1973 Vivre et laisser mourir
```

Si vous voulez mieux connaître `fseek()`, repartez de ex2213 pour faire insérer un peu plus d'enregistrements dans `bond.db` puis modifiez le présent exercice pour relire un des enregistrements (pas le premier, trop facile !).

En ligne 21, la mention `*1` est un choix personnel pour rester cohérent. Comme dans **ex2001**, cette mention `*1` me confirme que *fseek()* avance de 1 à la fois. Vous pouvez ne pas l'indiquer, car la valeur 1 est utilisée par défaut.

ex2217 (à partir de ex2012)

```
/* Programme interactif */
/* Dan Gookin, VF par O.E. */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

struct stypik {
  int maval;
  struct stypik *asuiv;
};
struct stypik *aprems;
struct stypik *acour;
struct stypik *anouv;
char *nomfic="liste.txt";            // NOUVEAU

int menu(void);
void ajouter(void);
void montrer(void);
void supprimer(void);
struct stypik *creer(void);
void ficlire(void);                  // NOUVEAU
void ficecrire(void);                // NOUVEAU

/* La fonction main() ne se charge que de la saisie.
   Le reste est dans les fonctions. */
int main()
{
  int choixmenu = '\0'; /* Initialise la boucle while */
  aprems=NULL;

  ficlire();                         /* NOUVEAU (lecture fichier) */

  while(choixmenu!='Q')
  {
    choixmenu=menu();
    switch (choixmenu)
    {
```

```
      case 'M':
        montrer();
        break;
      case 'A':
        ajouter();
        break;
      case 'S':
        supprimer();
        break;
      case 'Q':
        break;
    default:
      break;
    }
  }
  ficecrire();                          /* NOUVEAU */
  return(0);
}

/* Affiche le menu et collecte le choix */
int menu(void)
{
  int ch;

  printf("M)ontrer, A)jouter, S)upprimer, Q)uitter: ");
  ch=getchar();
  while(getchar()!='\n') /* Ignore la saisie superflue */
    ;
  return(toupper(ch));
}

/* Ajoute un membre en fin de liste */
void ajouter(void)
{
  if(aprems==NULL) /* Cas unique de aprems */
  {
    aprems = creer();
    acour = aprems;
  }
  else /* Cherche le dernier */
  {
    acour = aprems;
    while(acour->asuiv) /* Dernier == NULL */
      acour = acour->asuiv;
    anouv = creer();
    acour->asuiv = anouv; /* Actualisation lien */
    acour = anouv;
  }
  printf("Indiquez une valeur numerique : ");
  scanf("%d", &acour->maval);
  acour->asuiv = NULL;
  while(getchar()!='\n') /* Ignore la saisie superflue */
    ;
}

/* Affiche tous les enregs de la liste */
void montrer(void)
{
  int nbrenreg = 1;
  if(aprems == NULL) /* Liste vide */
  {
```

```
    puts("Rien a afficher");
    return;
  }
  puts("Affichage complet :");
  acour = aprems;
  while(acour)               /* Dernier == NULL */
  {
    printf("Enregistrement %d: %d\n",nbrenreg,acour->maval);
    acour = acour->asuiv;
    nbrenreg++;
  }
}

/* Supprime un enreg de la liste */
void supprimer(void)
{
  struct stypik *eprec; /* Sauve l'enreg d'avant */
  int r,c;

  if(aprems==NULL) /* Teste si liste vide */
  {
    puts("Aucun enregistrement !");
    return;
  }
  puts("Choisissez quel enreg. supprimer :");
  montrer();
  printf("Enregistrement : ");
  scanf("%d",&r);
  while(getchar()!='\n') /* Ignore la saisie superflue */
    ;
  c=1;
  acour = aprems;
  eprec = NULL; /* Pas de precedent du 1er */
  while(c!=r)
  {
    if(acour==NULL) /* Teste si 'r' dans la plage */
    {
      puts("Enreg introuvable");
      return;
    }
    eprec = acour;
    acour = acour->asuiv;
    c++;
  }
  if(eprec==NULL) /* Cas unique du 1er enreg. */
    aprems = acour->asuiv;
  else /* Raccorde precedent et suivant*/
    eprec->asuiv = acour->asuiv;
  printf("L'enreg %d n'existe plus.\n",r);
  free(acour); /* Restitue memoire */
}

/* Construit une structure vide et renvoie son adresse */
struct stypik *creer(void)
{
  struct stypik *a;
  a = (struct stypik *)malloc(sizeof(struct stypik));
  if(a==NULL)
  {
    puts("Erreur malloc()");
    exit(1);
```

```
  }
  return(a);
}

// *** NOUVELLES FONCTIONS ***
void ficlire(void)
{
  FILE *f;
  struct stypik charge;

  f = fopen(nomfic, "r");
  if(!f)                          /* Fichier introuvable !*/
    return;
                                  /* Lecture totale */
  while(fread(&charge, sizeof(struct stypik), 1, f))
  {
    if(aprems == NULL)            /* Code provenant de create() */
    {
      aprems = creer();
      acour  = aprems;
    }
    else
    {
      acour = aprems;
      while(acour->asuiv)
        acour = acour->asuiv;
      anouv = creer();
      acour->asuiv = anouv;
      acour = anouv;
    }
    acour->maval = charge.maval;     /* Lecture des valeurs du fichier, */
                                     /* pas des pointeurs ! */
    acour->asuiv = NULL;
  }
  fclose(f);
}

void ficecrire(void)
{
        int nbrenreg = 1;
        FILE *f;

        if(aprems == NULL)              /* Liste vide */
        {
                puts("Rien a sauvegarder !");
                return;
        }
        acour = aprems;
        f = fopen(nomfic, "w");
        if(!f)
        {
                puts("Erreur en ouverture de fichier");
                exit(1);
        }
        while(acour)                    /* Dernier enreg == NULL */
        {
                fwrite(acour, sizeof(struct stypik), 1, f);
                acour = acour->asuiv;
                nbrenreg++;
        }
        nbrenreg--;
```

```
        fclose(f);
        printf("%d enregistrements ecrits\n", nbrenreg);
}
```

CHAP 23

ex2301 (Listing 23.1)

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

int main()
{
    DIR *nomdoss;
    struct dirent *sfic;

    nomdoss=opendir(".");
    if(nomdoss==NULL)
    {
        puts("Lecture du dossier impossible");
        exit(1);
    }
    sfic = readdir(nomdoss);
    printf("Nom du fichier ou dossier '%s'\n", sfic->d_name);
    closedir(nomdoss);
    return(0);
}
```

Remarque

J'ai pensé reproduire toute la définition de la structure standard `dirent` dans le livre, mais elle est vraiment longue. Vous pouvez aller la consulter en ouvrant le fichier `dirent.h`. Sachez que ce fichier en référence et en incorpore plusieurs autres.

N.d.T. : Par rapport au livre, j'ai francisé la variable pointeur `file` en `sfic`. Vous distinguez mieux les variables au nom imposé de celles pour lesquelles vous décidez du nom.

ex2302

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

int main()
{
    DIR *nomdoss;
    struct dirent *sfic;

    nomdoss=opendir(".");
    if(nomdoss==NULL)
    {
        puts("Lecture du dossier impossible");
        exit(1);
    }
    while(sfic = readdir(nomdoss))                              // L16
        printf("Nom du fichier ou dossier '%s'\n", sfic->d_name);
    closedir(nomdoss);
    return(0);
}
```

Remarques

La boucle continue à lire une entrée du dossier tant que `readdir()` renvoie un pointeur sur une structure de type `dirent` (ligne 16).

En ligne 16, vous pouvez ignorer le message du compilateur à propos de parenthèses. Il a repéré le signe égal isolé et pense que vous vouliez en écrire deux pour une comparaison (`==`). Pourtant, l'instruction est correcte ici. D'ailleurs, le compilateur émet un avertissement, pas une erreur.

ex2303 (Listing 23.2)

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <time.h>
#include <sys/stat.h>

int main()
{
    DIR *nomdoss;
    struct dirent *sfic;
    struct stat statisfic;

    nomdoss = opendir(".");
    if(nomdoss == NULL)
    {
        puts("Lecture du dossier impossible");
        exit(1);
    }
    while(sfic = readdir(nomdoss))
    {
        stat(sfic->d_name, &statisfic);
        printf("%-14s %5ld %s",
            sfic->d_name,
            (long)statisfic.st_size,            // L24
            ctime(&statisfic.st_mtime));
    }
    closedir(nomdoss);
    return(0);
}
```

Remarques

Le compilateur va se plaindre par un avertissement mais compiler tout de même si en ligne 24 vous oubliez le transtypage de `statisfic.st_size`.

Vous pouvez modifier les largeurs en ligne 22 si l'affichage semble trop serré.

Voyez aussi la solution de ex2302 au sujet de l'avertissement des parenthèses.

ex2304

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <time.h>
#include <sys/stat.h>

int main()
{
    DIR *nomdoss;
    struct dirent *sfic;
    struct stat statisfic;
```

```
    nomdoss = opendir(".");
    if(nomdoss == NULL)
    {
        puts("Lecture du dossier impossible");
        exit(1);
    }
    while(sfic = readdir(nomdoss))
    {
        stat(sfic->d_name, &statisfic);
        if(S_ISDIR(statisfic.st_mode))          // L22
            printf("%-14s %-5s %s",
                sfic->d_name,
                "<DIR>",
                ctime(&statisfic.st_mtime));
        else
            printf("%-14s %5ld %s",
                sfic->d_name,
                (long)statisfic.st_size,
                ctime(&statisfic.st_mtime));
    }
    closedir(nomdoss);
    return(0);
}
```

Remarques

Cette solution n'est que l'une de celles possibles !

J'exploite en ligne 22 la macro `S_ISDIR`. La condition renvoie TRUE si un dossier est trouvé. Je peux donc insérer le texte `<DIR>` dans l'instruction `printf()` en ligne 23 sous forme de valeur littérale. La branche `else` ressemble à celle de **ex2303**.

ex2305 - Version Unix (Listing 23.3)

```
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>

int main()
{
    char dosscour[255];

    getcwd(dosscour,255);
    printf("Dossier courant : %s\n", dosscour);
    mkdir("ultra_tempo",755);                  // Ligne Unix
    puts("Nouveau dossier fait..");
    chdir("ultra_tempo");
    getcwd(dosscour,255);
    printf("Dossier courant : %s\n", dosscour);
    return(0);
}
```

ex2305 - Version Windows

```
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>

int main()
{
    char dosscour[255];

    getcwd(dosscour, 255);
```

```
    printf("Dossier courant : %s\n", dosscour);
    mkdir("ultra_tempo");                       // Ligne Windows
    puts("Nouveau dossier fait.");
    chdir("ultra_tempo");
    getcwd(dosscour,255);
    printf("Dossier courant : %s\n", dosscour);
    return(0);
}
```

Remarque

La seule différence entre les deux versions est en ligne 11.

ex2306 - Version Unix

```
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>

int main()
{
    char dosscour[255];

    getcwd(dosscour, 255);
    printf("Dossier courant : %s\n",dosscour);
    if(mkdir("ultra_tempo", 755) == -1)           //L11
    {
        puts("Erreur de creation du dossier");
        return(1);
    }
    puts("Nouveau dossier fait..");
    if(chdir("ultra_tempo") == -1)
    {
        puts("Erreur de changement de dossier");
        return(1);
    }
    getcwd(dosscour, 255);
    printf("Dossier courant : %s\n",dosscour);
    return(0);
}
```

ex2306 - Version Windows

```
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>

int main()
{
    char dosscour[255];

    getcwd(dosscour, 255);
    printf("Dossier courant : %s\n",dosscour);
    if(mkdir("ultra_tempo") == -1)                  // L11
    {
        puts("Erreur de creation du dossier");
        return(1);
    }
    puts("Nouveau dossier fait..");
    if(chdir("ultra_tempo") == -1)
    {
        puts("Erreur de changement de dossier");
        return(1);
```

```
    }
    getcwd(dosscour,255);
    printf("Dossier courant : %s\n", dosscour);
    return(0);
}
```

Remarques

La seule différence entre les deux versions est en ligne 11.

Les fonctions `chdir()` et `mkdir()` renvoient 0 si succès et -1 sinon.

ex2307 (Listing 23.4)

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

int main()
{
    FILE *test;

    test=fopen("blorfus", "w");
    if(!test)
    {
        puts("Creation du fichier impossible");
        exit(1);
    }
    fclose(test);
    puts("Fichier cree");
    if(rename("blorfus","wambooli") == -1)
    {
        puts("Impossible de renommer");
        exit(1);
    }
    puts("Nom du fichier modifie");
    return(0);
}
```

ex2308 (Listing 23.5)

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

int main()
{
    FILE *ficorig, *ficopie;
    int c;

    ficorig = fopen("main.c", "r");        // L09
    ficopie = fopen("main.bak", "w");
    if( !ficorig || !ficopie)
    {
        puts("Erreur fichier!");
        exit(1);
    }
    while( (c=fgetc(ficorig)) != EOF)
        fputc(c, ficopie);
    puts("Copie de fichier faite");
    return(0);
}
```

Remarques

La ligne 9 suppose la présence d'un fichier nommé `main.c`. Modifiez le paramètre pour qu'elle trouve un petit fichier dans le dossier courant. Vous pouvez aussi modifier le nom du fichier cible en ligne 10.

La condition en ligne 11 se fonde sur un OU logique afin de tester les deux pointeurs fichiers en une fois. En théorie, mieux vaut séparer les actions pour qu'en cas d'erreur, vous sachiez quel est le fichier concerné.

ex2309 (Listing 23.6)

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

int main()
{
    if(unlink("wambooli") == -1)
    {
        puts("Impossible de supprimer ce fichier");
        exit(1);
    }
    puts("Le fichier a disparu.");
    return(0);
}
```

Remarque

Le programme ne fonctionne que si le dossier contient un fichier nommé `wambooli`.

CHAP 24

ex2401 [Module main.c]

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

void second(void);

int main()
{
    printf("Second module, je te souhaite le bonjour !\n");
    second();
    return 0;
}
```

ex2401 [Module alpha.c]

```
#include <stdio.h>

void second(void)
{
    puts("Je suis ton second. Heureux de te voir !");
}
```

ex2402 [Module main.c]

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

void second(void);
```

```
int compteur;

int main()
{
    for(compteur=0; compteur<5; compteur++)
        second();
    return 0;
}
```

ex2402 [Module second.c]

```
#include <stdio.h>

extern int compteur;

void second(void)
{
    printf("%d\n", compteur+1);
}
```

ex2403 [Fichier d'en-tête ex2403.h]

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

/* Prototypes */

void remplir_struc(void);
void montrer_struc(void);

/* Constantes */

/* Variables */

struct sTruc {
    char nom[32];
    int age;
    };

typedef struct sTruc humain;
```

ex2403 [Module main.c]

```
#include "ex2403.h"

humain personne;

int main()
{
    remplir_struc();
    montrer_struc();
    return 0;
}

void remplir_struc(void)
{
    printf("Indiquez votre nom : ");
    fgets(personne.nom, 31, stdin);
    printf("Indiquez votre age : ");
    scanf("%d", &personne.age);
}

void montrer_struc(void)
```

```
{
    printf("Vous vous appelez %s\n", personne.nom);
    printf("Vous avez %d ans.\n", personne.age);
}
```

ex2404 [Fichier d'en-tête ex2403.h]

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

/* Prototypes */

void remplir_struc(void);
void montrer_struc(void);

/* Constantes */

/* Variables */

struct sTruc {
    char nom[32];
    int  age;
    };

typedef struct sTruc humain;
```

ex2404 [Module main.c]

```
#include "ex2403.h"

humain personne;

int main()
{
    remplir_struc();
    montrer_struc();
    return 0;
}
```

ex2404 [Module input.c]

```
#include "ex2403.h"

extern humain personne;

void remplir_struc(void)
{
    printf("Indiquez votre nom : ");
    fgets(personne.nom, 31, stdin);
    printf("Indiquez votre age : ");
    scanf("%d", &personne.age);
}
```

ex2404 [Module output.c]

```
#include "ex2403.h"

extern humain personne;

void montrer_struc(void)
{
    printf("Vous vous appelez %s\n", personne.nom);
    printf("Vous avez %d ans.\n", personne.age);
}
```

Remarque

Voici la commande du compilateur gcc permettant de construire l'exécutable de ce projet multi-modules :

```
gcc  main.c  input.c  output.c  -o ex2404
```

CHAP 25

ex2501 (Listing 25.1)

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

int main()
{
    char loop;

    puts("Affichage de l'alphabet :");
    for(loop='A'; loop<='Z'; loop++);
        putchar(loop);
    return 0;
}
```

Remarque

Lorsque l'exécution échoue, vous voyez un caractère `[` . C'est le signe que la boucle est bien exécutée de `'A'` à `'Z'` mais sans rien afficher. Une fois que la condition de boucle n'est plus satisfaite, le programme passe enfin à l'instruction suivante qui est celle d'affichage. La variable contient à ce moment le caractère dont la valeur suit le Z, et c'est le crochet gauche `[`.

ex2502 (Listing 25.2)

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

int main()
{
    int x;
    int *px;

    px=&x;
    for(x=0; x<10; x++)
        printf("%d\n", *px);
    return 0;
}
```

ex2503 (Listing 25.3)

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

int main()
{
    int e;

    e = rename("blorfus", "fragus");
    if( e != 0 )
    {
        printf("Erreur ! ");
        switch(errno)
```

```
        {
            case EPERM:
                puts("Operation interdite.");
                break;
            case ENOENT:
                puts("Fichier introuvable.");
                break;
            case EACCES:
                puts("Pas de permission.");
                break;
                break;
            case EROFS:
                puts("Fichier en lecture seule.");
                break;
            case ENAMETOOLONG:
                puts("Nom de fichier trop long.");
                break;
            default:
                puts("Trop horrible pour en dire plus.");
        }
    exit(1);
    }
    puts("Le fichier porte le nouveau nom.");
    return 0;
}
```

EOF