Apprendre à programmer en C pour les Nuls

Solutions des exercices

Cahier 1/3 : Parties I & II (chapitres 1 à 10)

Version anglaise sur http://www.c-for-dummies.com/begc4d/Exercices

Chapitre 01

ex0101

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

int main()
{
    printf("Hello world!\n");
    return 0;
}
```

Remarques

Rien à ajouter, puisque ce bloc de code est celui généré par Code::Blocks.

Il me plairait que l'utilisateur puisse personnaliser ce bloc, pour pouvoir débuter un projet soit sans aucune ligne, soit avec les lignes minimales dans `main.c`.

N.d.T. : Ce programme étant une amorce générée par le logiciel Code::Blocks, le message reste en anglais.

Chapitre 02

ex0201

```
#include <stdio.h>

int main()
{
    puts("Bienvenue aux humains.");
    return 0;
}
```

Remarque

Cet extrait est un peu plus long que le traditionnel premier programme d'autres livres. J'ai notamment personnalisé le message de bienvenue.

Chapitre 03

ex0301

```
#include <stdio.h>

main ()
{
    printf("4 fois 5 font %d\n",4*5);
    return(0);
}
```

Remarque

Ceci est le résultat final des multiples opérations d'édition proposées au cours du chapitre.

Chapitre 04

ex0401

```
#include <stdio.h>

int main()
{
  puts("Ne viens pas m'importuner maintenant. J'ai du travail.");
  return(0);
}
```

Remarque

Il s'agit une fois de plus du classique programme de bienvenue, sauf que j'utilise la fonction *puts()* au lieu de ***printf()***.

ex0402

```
#include <stdio.h>

int main()
{
  puts("J'adore afficher du texte !");
  return(0);
}
```

Remarques

Par le passé, à l'époque où les ordinateurs ne fonctionnaient qu'en mode texte avec une ligne de commande, j'avais écrit un programme qui affichait le message "Qui est là ?" Ce programme se nommait *Knock-Knock*.

En termes pratiques, sachez que vous n'êtes pas forcé de créer un nouveau projet pour chaque variante. Lorsque je vous invite par exemple dans l'Exercice 4.2 à modifier l'Exercice 4.1, vous pouvez faire la modification dans le projet ex0401 au lieu de créer un projet vide ex4020 puis y copier le source du précédent. Très rarement, je vous demande de repartir d'un exercice fait dans un chapitre antérieur. Si vous ne trouvez pas ce nom de projet, cherchez le numéro de projet précédent.

ex0403

```
#include <stdio.h>

int main()
{
    puts("Une souris verte,");
    puts("qui courait dans l'herbe.");
    return(0);
}
```

Remarque

La fonction *puts()* se contente d'afficher du texte.

ex0404

```
#include <stdio.h>

int main()
{
```

```
    puts("Une souris verte,");
    puts("qui courait dans l'herbe.");
    puts("Je l'attrape par la queue, ");
    puts("Je la montre à ces messieurs.");
    puts("Ces messieurs me disent");
    return(0);
}
```

ex0405

```
#include <stdio.h>

int main()
{
    puts("Le mot de passe secret est :");
/*  puts("Spatula."); */
    return(0);
}
```

Remarques

En général, pour le débogage, je neutralise une instruction suspecte en la transformant en commentaires. Soit elle ne fonctionne pas, soit elle me gêne pour étudier un autre partie du code.

Code::Blocks va éventuellement ajouter une indentation d'office pour aboutir à ceci :

```
#include <stdio.h>

int main()
{
    puts("Le mot de passe secret est :");
    /*  puts("Spatula."); */
    return(0);
}
```

C'est selon moi moins lisible qu'avant. Prenez l'habitude de neutraliser vos instructions en posant le marqueur de début de commentaires en colonne 1.

ex0406

```
#include <stdio.h>

int main()
{
    puts("Le mot de passe secret est :");
    puts("Spatula.");
    return(0);
}
```

Remarques

Rappelons que vous n'êtes pas forcé de créer un nouveau projet pour chaque étape d'une série de retouche du même code.

Vous n'avez pas oublié d'enlever aussi le couple de fin de commentaires `*/`, n'est-ce pas ?

Vous profiterez souvent de la possibilité de neutraliser une ligne par mise en commentaires, surtout pendant l mise au point.

ex0407

```
#include <stdio.h>

int main()
{
```

```
//  puts("Le mot de passe secret est :");
    puts("Spatula.");
    return(0);
}
```

Remarques

Comme dans l'Exercice 4.5, posez les signe `//` en tout début de ligne pour plus de lisibilité.

Dans mon code, j'utilise les marqueurs appariés `/*` et `*/` pour les blocs et le couple `//` pour une ligne isolée. Ainsi, quand je vois la séquence `//`, je sais que c'est une unique ligne neutralisée.

ex0408

```
#include <stdio.h>

int main()
{
    puts("Ce programme fait BOUM !)
    return(0);
}
```

Remarque

Vous avez trouvé l'erreur ?

ex0409

```
#include <stdio.h>

int main()
{
    puts("Ce programme fait BOUM !")
    return(0);
}
```

Remarque

En ajoutant le guillemet fermant, nous réduisons déjà le nombre d'erreurs émises par le compilateur. Voici à quoi se résume chez moi le prochain essai de compilation :

```
ex0409.c:6: error: expected ';' before 'return'
```

Les choses deviennent bien plus faciles à corriger : il manque un signe ; avant `return`.

ex0410

```
#include <stdio.h>

int main()
{
    puts("Ce programme fait BOUM !");
    return(0);
}
```

Remarque

Faire deux exercices pour corriger une double bourde peut sembler superflu, mais c'est souvent de cette façon que vous allez progresser, pas à pas. Vous corrigez un problème, vous observez le résultat, vous corrigez le suivant, etc.

ex0411

```
#include <stdio.h>

int main()
{
    printf("Etranger dans son propre pays.");
```

```
    return(0);
}
```

Remarque

Cet exemple ressemble au code initial standard de Code::Blocks avec un appel à ***`printf()`***.

ex0412

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

int main()
{
    printf("Une souris verte,");
    printf("qui courait dans l'herbe.");
    printf("Je l'attrape par la queue, ");
    printf("Je la montre à ces messieurs.");
    printf("Ces messieurs me disent");
    return(0);
}
```

Remarque

Voyez aussi la solution de l'exercice 4.4.

ex0413

```
#include <stdio.h>

int main()
{
    printf("Une souris verte,\n");
    printf("qui courait dans l'herbe.\n");
    printf("Je l'attrape par la queue, \n");
    printf("Je la montre à ces messieurs.\n");
    printf("Ces messieurs me disent\n");
    return(0);
}
```

Remarque

Pour ajouter une ligne vide après chaque strophe, redoublez le métacaractère : `\n\n`.

ex0414

```
#include <stdio.h>

int main()
{
    printf("\"Bonjour,\" dit le corbeau, \"que vous me semblez joli !\"\n");
    return(0);
}
```

Remarques

Il est nécessaire de désactiver l'effet normal du guillemet lorsqu'il doit être littéral et non servir à marquer la fin de la chaîne, et ceci par le signe antibarre \.

Vous n'avez pas oublié d'ajouter le marqueur de saut de ligne à la fin ?

ex0415

```
#include <stdio.h>

int main()
{
```

```
    puts("\"Bonjour,\" dit le corbeau, \"que vous me semblez joli !\"");
    return(0);
}
```

Remarques

Comme avec ***`printf()`***, il faut désactiver les guillemets à faire paraître dans la chaîne.

Le saut de ligne `\n` n'est plus requis puisque `puts()` en ajoute un d'office.

ex0416

```
#include <stdio.h>

int main()
{
    writeln("Ceci est une erreur pour Monsieur le lieur.");
    return(0);
}
```

Remarque

Au lieu de faire une faute de frappe comme dans le livre (`pritf`), nous proposons ici une autre bourde consistant à se tromper de langage. La fonction `writeln()` n'existe pas en langage C, mais elle est très utilisée en langage Pascal. Elle a le même effet que la fonction C nommée `puts()`.

Chapitre 05

ex0501

```
#include <stdio.h>

int main()
{
    printf("La valeur %d est un entier.\n", 986);
    printf("La valeur %f est un flottant.\n", 98.6);
    return(0);
}
```

Remarques

Deux éléments de la chaîne ne sont pas affichés littéralement dans les appels de fonction : le formateur numérique et le métacaractère de saut de ligne.

Tous les formateurs commencent par le préfixe `%` et les séquences d'échappement commencent par le signe antibarre `\` (*backslash*).

Les formateurs sont aussi appelés *conteneurs*.

Pour afficher littéralement le signe %, vous le redoublez : `%%`. Même les pros l'oublient parfois. Pour afficher `50%`, vous écrivez ceci :

```
printf("50%%");
```

ou bien :

```
printf("%d%%",50);
```

De même, pour rendre littéral le signe \, vous le redoublez : `\\`.

N.d.T. : Dans le livre imprimé, certains auront détecté que la valeur **869** dans l'affichage résultant doit se lire **986**. Heureusement que les machines ne modifient pas nos bulletins de salaire comme bon leur semble.

ex0502

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

```
int main()
{
    printf("%d\n", 127);
    printf("%f\n", 3.1415926535);
    printf("%d\n", 122013);
    printf("%f\n", 0.00008);
    return(0);
}
```

Remarques

Le formateur `\n` (saut de ligne) n'est pas obligatoire, mais il rend la sortie plus lisible.

Pour voir comment le compilateur s'en sort avec une erreur de formateur numérique, changez un des formateurs `%d` en `%f` ou vice versa, et recompilez. Observez les erreurs qui en résultent.

Une occasion de gagner un bonus Pour les nuls : compactez le code pour tout afficher avec un seul appel à `printf()`, mais sur des lignes successives. (Je ne reproduit pas le code source, car vous savez que vous avez réussi en regardant le résultat.)

ex0503

```
#include <stdio.h>

int main()
{
    printf("%d\n", 127);
    printf("%1.2f\n", 3.1415926535);
    printf("%d\n", 122013);
    printf("%1.1f\n", 0.00008);
    return(0);
}
```

Remarques

Voici ce que je vois s'afficher :

```
127
3.14
122013
0.0
```

La valeur de PI est affichée avec un signe dans la partie entière et deux dans la partie fractionnaire et la quatrième avec un chiffre avant et un après le point décimal. Qu'est devenue notre valeur .00008 ? Elle a été arrondie.

Retouchez le dernier appel de fonction pour qu'il stipule `%1.4f` puis recompilez/exécutez :

```
127
3.14
122013
0.0001
```

Le "8" n'apparaît pas, mais il a fait son effet sur l'arrondi.

Retouchez le deuxième appel pour remplacer `%1.2f par %1.3f`, recompilez et exécutez :

```
127
3.142
122013
0.0001
```

La valeur de PI a été arrondie à 3.142.

ex0504

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

```c
int main()
{
    puts("Valeurs initiales : 8 et 2");
    printf("Addition : %d\n",8+2);
    printf("Soustraction : %d\n",8-2);
    printf("Multiplication : %d\n",8*2);
    printf("Division : %d\n",8/2);
    return(0);
}
```

Remarque

De telles valeurs immédiates sont parfois utiles, mais la puissance d'un langage de programmation vient notamment de sa capacité à gérer des variables.

ex0505

```c
#include <stdio.h>

int main()
{
    puts("Valeurs initiales : 8 et 2");
    printf("Addition : %d\n",8+2);
    printf("Soustraction : %d\n",8-2);
    printf("Multiplication : %d\n",8*2);
    printf("Division : %d\n",8/2);
    printf("Addition fractionnaire : %f\n", 456.98 + 213.4);
    return(0);
}
```

Remarques

Le texte de légende de la nouvelle ligne est libre, mais vous devez mentionner le formateur `%f`, puisque les deux valeurs sont flottantes et donc le résultat aussi.

Si vous avez oublié de limiter l'affichage à deux chiffres après la virgule, corriger et recompilez.

En guise de variante, retouchez le code pour afficher ceci :

`456.98 + 213.40 = 670.38`

Servez-vous de la dernière ligne qui vient d'être ajoutée. Indiquez des formateurs et non des valeurs immédiates dans la chaîne. Voici ma solution :

```c
#include <stdio.h>

int main()
{
    puts("Valeurs initiales : 8 et 2");
    printf("Addition : %d\n",8+2);
    printf("Soustraction : %d\n",8-2);
    printf("Multiplication : %d\n",8*2);
    printf("Division : %d\n",8/2);
    printf("%.2f + %.2f = %.2f\n", 456.98, 213.4, 456.98 + 213.4);
    return(0);
}
```

ex0506

```c
#include <stdio.h>

int main()
{
    puts("Valeurs initiales : 8 et 2");
    printf("Addition : %d\n",8+2);
    printf("Soustraction : %d\n",8-2);
```

```
    printf("Multiplication : %d\n",8*2);
    printf("Division : %d\n",8/2);
    printf("Addition fractionnaire : %f\n", 456.98 + 213.4);
    printf("%.2f + %.2f = %.2f\n", 456.98, 213.4, 456.98 + 213.4);
    printf("Multiplication complexe : %d\n", 8*14*25);
    return(0);
}
```

Remarques

Modifiez le code pour afficher ceci :

`8 * 14 * 25 = 2800`

Voyez si le fait de changer l'ordre des valeurs change le résultat.

ex0507

```
#include

int main()
{
    printf("La solution est %d\n", 0+50*1-60-60*0+10);
    return(0);
}
```

Remarque

Pour un humain qui lit de gauche à droite, le résultat est 10 puisque ce qu'il y a à sa gauche finit par être multiplié par zéro et 0 + 10 = 10. L'ordinateur voit les choses autrement.

ex0508

```
#include <stdio.h>

int main()
{
    printf("Le total vaut %d\n", 16+17);
    return(0);
}
```

Remarque

Les valeurs immédiates 16 et 17 sont considérées par le compilateur comme de type `int` parce qu'elles n'ont pas de partie décimale.

ex0509

```
#include <stdio.h>

int main()
{
    printf("Le total vaut %d\n", 16.0+17);
    return(0);
}
```

Remarques

Le compilateur considère la valeur 16.0 comme un flottant à cause de la partie décimale, mais la valeur 17 comme un entier.

Voici ce qui est apparu sur ma machine :

`Le total vaut 1606416648`

Ce n'est pas une valeur vraiment aléatoire comme serait la lecture d'une position mémoire au hasard. C'est le résultat de la tentative de la fonction ***printf()*** d'afficher au format d'un entier (à cause du formateur `%d`) une valeur qui est devenue flottante.

ex0510

```
#include <stdio.h>

int main()
{
    printf("Le total vaut %.1f\n", 16.0+17.0);
    return(0);
}
```

Remarque

Du fait que les deux valeurs n'ont pas de partie fractionnaire (`.0`), vous auriez pu spécifier le formateur `%.0f` pour ne faire afficher que la partie entière.

ex0511

```
#include

int main()
{
    printf("La valeur est %d\n", 3);
    printf("et %d est bien la valeur\n", 3);
    printf("Ce n'est pas %d\n", 3+1);
    printf("ni %d non plus.\n", 3-1);
    printf("Non, la valeur reste %d\n", 3);
    return(0);
}
```

ex0512

```
#include

int main()
{
    printf("La valeur est %d\n", 5);
    printf("et %d est bien la valeur\n", 5);
    printf("Ce n'est pas %d\n", 5+1);
    printf("ni %d non plus.\n", 5-1);
    printf("Non, la valeur reste %d\n", 5);
    return(0);
}
```

ex0513

```
#include <stdio.h>

#define VALEUR 3

int main()
{
    printf("La valeur est %d\n", VALEUR);
    printf("et %d est bien la valeur\n", VALEUR);
    printf("Ce n'est pas %d\n", VALEUR+1);
    printf("ni %d non plus.\n", VALEUR-1);
    printf("Non, la valeur reste %d\n", VALEUR);
    return(0);
}
```

Remarque

Si vous faites une erreur dans le nom d'une constante, elle ne sera bien sûr pas reconnue, mais le compilateur repère assez facilement ce genre de bourde.

ex0514

```
#include <stdio.h>

#define OCTO 8
#define DUO  2

int main()
{
    puts("Valeurs initiales : 8 et 2");
    printf("Addition : %d\n", OCTO+DUO);
    printf("Soustraction : %d\n", OCTO-DUO);
    printf("Multiplication : %d\n", OCTO*DUO);
    printf("Division : %d\n", OCTO/DUO);
    return(0);
}
```

Remarques

Vous n'êtes pas obligé de mentionner les constantes `OCTO` et `DUO` dans la première instruction. Il suffit de rappeler les valeurs comme dans l'Exercice 5.4.

En effet, nous ne les utilisons pas dans la Ligne 8, parce qu'il faudrait modifier l'instruction pour afficher les valeurs des constantes. Pour le compilateur, tout ce qui entre guillemets, sauf les formateurs et échappements, est du texte statique. L'instruction suivante :

```
puts("Valeurs initiales : OCTO et DUO");
```

va afficher :

```
Valeurs initiales : OCTO et DUO
```

Les constantes ne sont pas détectées par le compilateur lorsque leur nom est placé dans une chaîne littérale.

Chapitre 06

ex0601

```
#include <stdio.h>

int main()
{
    int x;

    x = 5;
    printf("La valeur de la variable x est %d.\n", x);
    return(0);
}
```

Remarques

J'ai l'habitude de prévoir une ligne vide après la dernière déclaration de variable.

La syntaxe d'affectation de valeur à une variable est stricte : la valeur source doit être à droite du signe et le nom de variable (la `destination`) à gauche. L'écriture suivante n'a aucun sens en C :

```
5 = x;
```

Le c dans la chaîne de `printf()` est la lettre x de l'alphabet, pas le nom de la variable, puisqu'elle est entre les guillemets de la chaîne.

Le choix du nom x pour la variable provient bien sûr du monde des mathématiques : x est l'inconnue (belle ?). N.d.T. : ceci dit, les noms de variables sur une seule lettre ne sont pas les plus pratiques à rechercher.

ex0602

```
#include <stdio.h>

int main()
{
    char c;
    int i;
    float f;
    double d;

    c = 'a';
    i = 1;
    f = 19.0;
    d = 20000.009;

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

Remarques

Rappelons que pour spécifier un caractère isolé, vous le délimitez par des apostrophes, pas des guillemets.

Pour faire afficher l'apostrophe, vous devez désactiver son effet normal par le préfixe antibarre `\'`. En tant que caractère isolé à afficher, cela donne `'\''`.

Tout caractère non disponible au clavier peut s'exprimer par son code ASCII (ou UTF). Indiquez la valeur hexa entre apostrophes après le préfixe `\x`. Pour spécifier le caractère de contrôle de valeur 11 (donc 0B en hexa), vous écrivez `'\x0B'`

ex0603

```
#include <stdio.h>

int main()
{
    char c;
    int i;
    float f;
    double d;

    c = 'a';
    i = 1;
    f = 19.0;
    d = 20000.009;

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

Remarques

L'instruction d'affichage semble indigeste. Mais il suffit de repérer les quatre formateurs (car quatre variables) séparés par des séquences d'échappement `\n`.

En général, je n'écris pas de chaînes aussi peu lisibles. Il est bien plus simple de la fractionner en quatre appels successifs à `printf()` comme nous l'avons fait dans ex0602.

ex0604

```
#include <stdio.h>

int main()
{
    char c;
    int i;
    float f;
    double d;

    c = 'a';
    i = 1;
    f = 19.8;
    d = 20000.009;

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

Remarques

Cet exercice a pour but de vous rappeler comment afficher des valeurs flottantes en contrôlant la précision comme dans `%.2f` au lieu de la lettre isolée `%f`. Ici, nous ne contrôlons pas les détails du format ; l'affichage aura la précision que l'ordinateur peut produire, mais cela risque de perturber l'utilisateur.

ex0605

```
#include <stdio.h>

int main()
{
    int blorf;

    blorf = 22;

    printf("La valeur de blorf est %d.\n", blorf);
    printf("La valeur de blorf + 16 est %d.\n", blorf+16);
    printf("La valeur de blorf puissance deux est %d.\n", blorf*blorf);
    return(0);
}
```

Remarques

Les deux derniers affichages montrent comment produire une valeur littérale à afficher.

Nous puisons la valeur de départ dans une variable, mais le résultat est de type immédiat. Il n'est pas restocké dans une variable. En effet :

* En C, une équation peut comporter un nom de variable numérique à la place d'une valeur, et c'est souvent le cas.

* L'utilisation de la variable `blorf` ne change pas sa valeur, puisque nous ne faisons que la lire. Vous le prouvez en créant une quatrième ligne d'affichage identique à la première pour voir quelle est la valeur de `blorf` après les calculs.

ex0606

```
#include <stdio.h>

#define GLORKUS 16

int main()
```

```
{
    int blorf;

    blorf = 22;

    printf("La valeur de blorf est %d.\n", blorf);
    printf("La valeur de blorf + 16 est %d.\n", blorf+GLORKUS);
    printf("La valeur de blorf puissance deux est %d.\n", blorf*blorf);
    return(0);
}
```

Remarques

La solution suivante est encore meilleure :

```
#include <stdio.h>

#define GLORKUS 16

int main()
{
    int blorf;

    blorf = 22;

    printf("La valeur de blorf est %d.\n", blorf);
    printf("La valeur de blorf + %d est %d.\n", GLORKUS, blorf+GLORKUS);
    printf("La valeur de blorf puissance deux est %d.\n", blorf*blorf);
    return(0);
}
```

Nous affichons la valeur de `GLORKUS` dans le deuxième appel à `printf()` avec le formateur `%d`. Cette technique permet de modifier la valeur de la constante sans devoir retoucher en plusieurs points du code source, au risque d'en oublier.

ex0607

```
#include <stdio.h>

int main()
{
    unsigned int ono;

    ono = -10;
    printf("La valeur de ono est %u.\n", ono);
    return(0);
}
```

Remarques

C'est avec l'opérateur unaire `-` que nous affectons une valeur négative dans la variable. Ce n'est pas la valeur négative -10, mais la valeur 10 à laquelle nous appliquons l'opérateur de signe négatif en préfixe.

En remplaçant le formateur `%u` par `%d`, l'affichage deviendra correct (-10). Comme pour les flottants, la valeur négative peut être interprétée faussement en tant que valeur strictement positive (non signée, *unsigned*), mais elle vaudra alors 4294967286. Le formateur a donc une grande importance.

Ne confondez pas ce formateur avec les besoins de certains fonctions qui attendent une valeur entière non signée ou en renvoient une. Dans ces cas, si vous ne précisez `unsigned int`, le compilateur va se plaindre à juste titre.

ex0608

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

```
int main()
{
    int ananias, azarias, misael;

    ananias = 701;
    azarias = 709;
    misael  = 719;
    printf("Ananias est %d\nAzarias est %d\nMisael est %d\n", ananias, azarias,
    misael);
    return(0);
}
```

Remarques

La ligne de l'instruction d'affichage a été saisie sans discontinuer, mais un saut de ligne apparaît sans doute dans ce que vous voyez ici.

Voici le même programme, mais en contrôlant la fin de cette ligne avec l'opérateur `\` de poursuite d'instruction :

```
#include <stdio.h>

int main()
{
    int ananias, azarias, misael;

    ananias = 701;
    azarias = 709;
    misael  = 719;
    printf("Ananias est %d\nAzarias est %d\nMisael est %d\n",\
        ananias, azarias, misael);
    return(0);
}
```

Le caractère antibarre `\` (backslash) est ignoré par le compilateur tout comme le saut de ligne qui le suit. Rappelons qu'à l'intérieur d'une chaîne, ce signe sert de préfixe pour neutraliser l'effet normal d'un métacaractère.

Une solution encore plus lisible est celle-ci :

```
#include <stdio.h>

int main()
{
    int ananias, azarias, misael;

    ananias = 701;
    azarias = 709;
    misael  = 719;

    printf("Ananias est %d\nAzarias est %d\nMisael est %d\n",
        ananias,
        azarias,
        misael);
    return(0);
}
```

Ici, nous n'avons plus besoin d'antibarre. les sauts de lignes et espaces d'alignement sont ignorés, comme des espaces. Chaque variable est sur une ligne distincte. Cette manière est très utile dans les instructions `printf()` complexes, avec beaucoup de formateurs.

N'oubliez pas pour autant la parenthèse fermante et le point-virgule.

Nous aurions pu écrire trois appels à `printf()` sur trois lignes. Il y a souvent plusieurs solutions de syntaxe en langage C.

ex0609

```
#include <stdio.h>

int main()
{
    int start = 0;

    printf("La valeur initiale est %d.\n", start);
    return(0);
}
```

ex0610

```
#include <stdio.h>

int main()
{
    int ananias = 701;
    int azarias = 709;
    int misael  = 719;

    printf("Ananias est %d\nAzarias est %d\nMisael est %d\n",\
            ananias, azarias, misael);
    return(0);
}
```

Remarques

Vous pouvez même regrouper les trois déclarations et initialisations sur la même ligne en séparant les variables par des virgules :

```
#include <stdio.h>

int main()
{
    int ananias=701, azarias=709, misael=719;

    printf("Ananias est %d\nAzarias est %d\nMisael est %d\n",\
            ananias, azarias, misael);
    return(0);
}
```

La lisibilité commence à être mise à mal. Je pense que le second code de l'exemple ex0608 est le plus lisible.

ex0611

```
#include <stdio.h>

int main()
{
    int npremier;

    npremier = 701;
    printf("Ananias vaut %d\n", npremier);
    npremier = 709;
    printf("Azarias vaut %d\n", npremier);
    npremier = 719;
    printf("Misael vaut %d\n", npremier);
    return(0);
```

```
}
```

Remarques

En général en C, plutôt que d'affecter une nouvelle valeur littérale à la même variable, c'est le fruit d'un calcul qui est affecté comme nouvelle valeur de variable.

ex0612

```
#include <stdio.h>

int main()
{
    int a,b,c;

    a = 5;
    b = 7;
    c = a + b;
    printf("Variable c=%d\n", c);
    return(0);
}
```

Remarque

La valeur mystère qui s'affiche est bien sûr 12.

ex0613

```
#include <stdio.h>

int main()
{
    float flot1, flot2, flores;

    flot1  = 15,5;
    flot2  = 3.2;
    flores = flot1 / flot2;
    printf("Variable flores=%f\n", flores);
    return(0);
}
```

Remarques

En partant du code de ex0612, il suffit de quelques retouches.

Vous pouvez choisir d'autres valeurs pour les deux premières variables, du moment qu'elles soient flottantes.

Les calculs sont toujours en membres droits des instructions en langage C. Ceci est incorrect et n'a aucun sens :

```
a / b = c;
```

N'oubliez pas que pour qu'une valeur soit détectée comme flottante, il faut indiquer une partie fractionnaire et au moins un chiffre avant ! Pour indiquer la valeur .5, il faut écrire `0.5`. De même, pour écrire la valeur flottante 15, il faut écrire `15.0`.

Chapitre 07

ex0701

```
#include <stdio.h>

int main()
```

```
{
    int c;

    printf("Je vais recevoir un caractere: ");
    c = getchar();
    printf("J'ai obtenu le caractere '%c'.\n", c);
    return(0);
}
```

Remarque

Le caractère saisi au clavier et lu par **`getchar()`** est stocké en tant que valeur numérique de son code ASCII (les codes ASCII sont fournis dans l'Annexe A du livre).

ex0702

```
#include <stdio.h>

int main()
{
    int c;

    printf("Je vais recevoir un caractere: ");
    c = getchar();
    printf("J'ai obtenu le caractere '%d'.\n", c);  // L09
    return(0);
}
```

Remarque

Pour en savoir plus sur la relation entre caractère affiché et code ASCII, vous pouvez modifier la ligne 9 sur ce modèle :

```
J'ai obtenu le caractere 'M', de code ASCII 77.
```

Voici ma solution :

```
#include <stdio.h>

int main()
{
    int c;

    printf("Je vais recevoir un caractere: ");
    c = getchar();
    printf("J'ai obtenu le caractere '%c', de code ASCII %d.\n", c, c);
    return(0);
}
```

ex0703

```
#include <stdio.h>

int main()
{
    int c;

    printf("Je vais recevoir un caractere: ");
    c = getc(stdin);
    printf("J'ai obtenu le caractere '%c'.\n", c);
    return(0);
}
```

Remarques

`getchar()` est une macro, mais les programmeurs parlent d'une fonction C.

La définition de `getchar()` se trouve dans le fichier d'en-tête `stdio.h`. La voici après un léger nettoyage :

```
#define getchar()   getc(stdin)
```

C'est bien une sorte de constante. Toutes les occurrences de `getchar()` sont remplacées par `getc(stdin)`.

Rappelons que `stdin` est une constante désignant l'entrée standard, bien qu'elle ne suive pas la convention consistant à écrire les constantes en lettres majuscules.

La sortie standard s'écrit `stdout`. et la sortie pour les messages d'erreur est `stderr`. Elle ne peut pas être redirigée comme ses deux collègues afin que les erreurs ne puissent pas être rendues invisibles. Les affichages seront toujours envoyés vers la sortie standard qui est l'écran ou la console système, quels que soient les efforts de redirection stipulés sur la ligne de commande.

ex0704

```
#include <stdio.h>

int main()
{
    int a,b,c;

    printf("J'attend trois lettres : ");
    a = getchar();
    b = getchar();
    c = getchar();
    printf("Les trois lettres sont '%c', '%c' et '%c'\n",a,b,c);
    return(0);
}
```

Remarque

Rappelons que *getchar()* n'est pas interactive ; aucune fonction de lecture de flux en entrée ne permet de réagir à une saisie suivie d'une validation.

ex0705

```
#include <stdio.h>

int main()
{
    int ch;

    printf("Frappez Entree : ");
    getchar();
    ch = 'H';
    putchar(ch);
    ch = 'i';
    putchar(ch);
    putchar('!');
    return(0);
}
```

Remarques

Comme sa collègue `getchar()`, `putchar()` est une macro définie dans `stdio.h` à peu près comme ceci :

```
#define putchar(x)   putc(x, stdout)
```

Il s'agit donc en fait de la fonction standard de sortie `putc()` à laquelle on demande d'envoyer une valeur de type `int` vers la sortie standard `stdout`.

La macro *putchar()* est très utilisée pour afficher des caractères isolés.

Le caractère apparaît à la position du curseur ou à la suite des données présentes si vous redirigez vers un fichier. Il n'est pas suivi d'un saut de ligne.

ex0706

```
#include <stdio.h>

int main()
{
    int ch;

    printf("Frappez Entree : ");
    getchar();
    ch = 'H';
    putchar(ch);
    ch = 'i';
    putchar(ch);
    putchar('!');
    putchar('\n');
    return(0);
}
```

Remarques

Le métacaractère de saut de ligne est `\n`. Cette séquence d'échappement doit être délimitée par des apostrophes car elle correspond à un code numérique de caractère isolé.

Vous auriez pu affecter ce symbole à la `ch` puis le faire "afficher" par `putchar()`.

ex0707

```
#include <stdio.h>

int main()
{
    char a, b, c, d;

    a = 'W';
    b = a + 24;
    c = b + 8;
    d = '\n';
    printf("%c%c%c%c", a, b, c, d);
    return(0);
}
```

Remarques

Les opérations mathématiques sur des variables `char` modifient la valeur numérique qu'elle contiennent, valeur qui est en fait de type entier `int` et correspond à un code ASCII qui détermine l'aspect visuel du caractère à afficher quand on indique le formateur `%c` dans `printf()` ou `putchar()`.

Pour le compilateur, une constante caractère comme `'W'` est une valeur immédiate. Vous êtes autorisé à écrire `'W'` pour plus de confort, mais c'est une valeur numérique.

Rappelons que le nombre de formateurs de caractères cités dans `printf()` doit coïncider avec le nombre de variables.

ex0708

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

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

    a = 'W';
    b = 'o';
    c = 'w';
    d = '\n';
    printf("%c%c%c%c", a, b, c, d);
    return(0);
}
```

Remarque

Pour savoir quels caractères spécifier pour les variables `b` et `c`, il faut avoir réussi l'exercice précédent pour les avoir vus au moins une fois.

ex0709

```
#include <stdio.h>

int main()
{
    char a, b, c, d;

    a = 'W';
    b = 'o';
    c = 'w';
    d = '\n';
    putchar(a);
    putchar(b);
    putchar(c);
    putchar(d);
    return(0);
}
```

Remarques

Il faut un appel à `putchar()` par variable.

Dès que vous voyez plusieurs variables simples manipulées de la même manière (affectation puis transfert à une fonction), ayez le réflexe de constituer un tableau regroupant ces variables (voyez le Chapitre 12 pour les tableaux).

ex0710

```
#include <stdio.h>

int main()
{
    char prompt[] = "Frappez Entree pour exploser :";    // L05

    printf("%s", prompt);
    getchar();
    return(0);
}
```

Remarques

Les crochets droits en ligne 5 sont bien vides. Dans certaines polices, cela ressemble à un carré. En langage C, il n'y a pas de signe carré ; il s'agit bien de `[` et `]` à la suite.

En C, un tableau est souvent déclaré avec une dimension :

```
char prompt[24] = "Frappez Entree pour exploser :";
```

La valeur entre crochets est le nombre maximal d'éléments (de caractères de la chaîne). Si vous ne précisez pas la longueur, le compilateur compte pour vous la longueur de la chaîne littérale, ce qui est le cas ici.

Souvent, on laisse le compilateur s'occuper de ce dimensionnement pour les chaînes statiques comme ci-dessus.

Tout ce qui se trouve entre les guillemets en ligne 5 est stocké dans le tableau de type `char` nommé `prompt`.

Il n'existe pas de type officiel **string** pour les chaînes en C, mais vous pouvez désigner `prompt` comme une "variable chaîne". Vous serez compris de tous.

Le formateur `%s` est pourtant le formateur destiné aux chaînes de caractères.

ex0711

```
#include <stdio.h>

int main()
{
    char prompt[] = "Programme pour exploser le Monde\nFrappez Entree pour lancer
    l'explosion :";

    printf("%s", prompt);
    getchar();
    return(0);
}
```

Remarques

Le métacaractère de saut de ligne `\n` peut être inséré n'importe où dans une chaîne.

Nous n'avons pas ajouté de saut de ligne en fin de chaîne `prompt` pour que l'affichage respecte les attentes habituelles des utilisateurs : le curseur doit clignoter à la suite du message et pas sur la ligne suivante.

Nous affichons deux lignes, mais un seul formateur `%s` suffit dans `printf()` car le saut de ligne est un caractère, presque comme un autre.

ex0712

```
#include <stdio.h>

int main()
{
    char prenom[15];          // L05

    printf("Veuillez indiquer votre petit nom : ");
    scanf("%s", prenom);
    printf("Ravi de vous saluer, %s.\n", prenom);
    return(0);
}
```

Remarques

Nous déclarons la chaîne en limitant la longueur à 15 caractères (ligne 5), dont 14 utiles. Cette approche n'est pas très robuste, parce que rien n'empêche l'utilisatrice "Charlotte-Dominique" de saisir son prénom ; les caractères au delà du quinzième vont être stockés à la suite en mémoire en écrasant les données qui se trouvaient à cet endroit, ce qui va créer un sérieux problème au programme.

N.d.T. : De plus, il n'y aura plus de place en dernière position pour le zéro terminal de fin de chaîne et le programme va planter quand vous voudrez relire la variable.

La fonction *scanf()* s'arrête de lire lorsqu'elle détecte un espace. Si vous indiquez `Charles Emile`, seul `Charles` sera lu.

ex0713

```
#include <stdio.h>

int main()
{
    char prenom[15];
    char nomfami[15];

    printf("Veuillez indiquer votre petit nom : ");
    scanf("%s", prenom);
    printf("Et votre nom de famille : ");
    scanf("%s", nomfami);
    printf("Ravi de vous saluer, %s %s.\n", prenom, nomfami);
    return(0);
}
```

Remarques

J'ai choisi une longueur maximale de 15 (14 utiles) pour les deux tableaux `prenom` et `nomfami` mais vous pouvez voir plus large, par exemple pour tenir compte des longs patronymes malgaches.

Le compilateur ajoute d'office le zéro terminal `\0` en fin de chaîne. N'oubliez pas d'ajouter un à votre longueur maximale de chaîne.

Comme pour l'exercice précédent, si vous indiquez un nom composé sans tirets comme `Van Beethoven`, `scanf()` ne va garder que `Van`. Il faut plutôt utiliser `fgets()`, ce que nous montrons dans le livre.

ex0714

```
#include <stdio.h>

int main()
{
    int fav;

    printf("Saisissez votre chiffre favori : ");
    scanf("%d", &fav);
    printf("%d est mon chiffre favori aussi !\n", fav);
    return(0);
}
```

ex0715

```
#include <stdio.h>

int main()
{
    float fav;                // L05

    printf("Saisissez votre chiffre favori : ");
    scanf("%f", &fav);
    printf("%f est mon chiffre favori aussi !\n", fav);
    return(0);
}
```

Remarques

Pour pouvoir faire saisir une valeur numérique flottante, trois retouches sont à prévoir : dans la déclaration de `fav` en ligne 5 et au niveau du formateur qui devient `%f` dans `scanf()` et `printf()`.

Même si vous saisissez une valeur entière, `scanf()` va stocker un flottant. En saisie, ce n'est pas gênant, mais n'oubliez pas d'écrire une valeur flottante en ajoutant `.0` si nécessaire si vous indiquez une valeur littérale dans le code source.

ex0716

```
#include <stdio.h>

int main()
{
    char personne[10];

    printf("Qui etes-vous ? ");
    fgets(personne, 10, stdin);
    printf("Heureux de vous rencontrer, %s.\n", personne);
    return(0);
}
```

Remarques

La capacité utile du tableau `personne` est de 9 caractère, le dixième servant au zéro terminal `\0` (valeur zéro).

La fonction `fgets()` sait gérer cette limite en ne lisant que 9 puis en ajoutant le zéro terminal.

Indiquez toujours `stdin` avec `fgets()` pour lire depuis l'entrée standard qui est le clavier.

ex0717

```
#include <stdio.h>

#define INPUT_TAILLE 3

int main()
{
    char personne[INPUT_TAILLE];

    printf("Qui etes-vous ? ");
    fgets(personne, INPUT_TAILLE, stdin);
    printf("Heureux de vous rencontrer, %s.\n", personne);
    return(0);
}
```

Remarques

Cet exercice montre comment une constante (`INPUT_TAILLE`) limite une saisie, ici à deux caractères plus le `\0`. Testez-le pour le vérifier.

Avec cette constante, vous redimensionnez la chaîne aisément. Pour que ce programme devienne utile, amenez la valeur de `INPUT_TAILLE` à 30 ou 40.

ex0718

```
#include <stdio.h>

int main()
{
    char prenom[15];
    char nomfami[15];
```

```
    printf("Veuillez indiquer votre petit nom : ");
    fgets(prenom, 15, stdin);
    printf("Et votre nom de famille : ");
    fgets(nomfami, 15, stdin);
    printf("Ravi de vous saluer, %s %s.\n", prenom, nomfami);
    return(0);
}
```

Remarques

Cet exercice illustre un petit souci d'utilisation de `fgets()` pour saisir du texte : la frappe de la touche Entrée est stockée dans la chaîne. Voici une exécution du programme :

```
Veuillez indiquer votre petit nom : Danny
Et votre nom de famille : Gookin
Ravi de vous saluer, Danny
 Gookin
.
```

La touche Entrée de fin de saisie du prénom est reproduite en sortie, et le nom est sur la ligne suivante, de même que le point final. Une première solution consiste à lire les deux chaînes sous forme d'une seule, comme dans le prochain exemple.

ex0718bis (deux chaînes en une)

```
#include <stdio.h>

int main()
{
    char personne[30];

    printf("Indiquez le prenom puis le nom : ");
    fgets(personne, 30, stdin);
    printf("Ravi de vous saluer, %s.\n",personne);
    return(0);
}
```

Hélas, l'affichage comporte toujours le saut de ligne provoqué par le code la touche Entrée de fin de saisie :

```
Indiquez le prenom puis le nom : Annie Kotin
Ravi de vous saluer, Annie Kotin
.
```

Vous voyez le point isolé en troisième ligne.

Nous verrons plus loin comment éliminer le caractère `\n` superflu dans une chaîne lue par `fgets()`.

Chapitre 08

ex0801

```
#include <stdio.h>

int main()
{
    int a,b;

    a = 6;
    b = a - 2;

    if( a > b)
```

```
    {
        printf("%d est plus grand que %d\n", a, b);
    }
    return(0);
}
```

Remarque

Nous sommes ici dans un cas d'école avec des valeurs fixes, mais dans la réalité, ce que vous allez comparer proviendra en général d'une saisie utilisateur ou d'un calcul.

ex0802

```
#include <stdio.h>

int main()
{
    int a,b;

    a = 6;
    b = a + 2;

    if( a > b)
    {
        printf("%d est plus grand que %d\n", a, b);
    }
    return(0);
}
```

Remarques

Vous n'avez pas fait d'erreur. Le programme n'affiche rien, mais devinez-vous pourquoi avant de lire la suite ?

Observez bien le code source !

Rien ne s'affiche, car la valeur de la variable `a` n'est pas supérieure à celle de la variable `b`. L'instruction entre accolades n'est donc pas exécutée puisque le test `if` est faux. Lisez la suite du livre pour en savoir plus.

ex0803

```
#include <stdio.h>

int main()
{
    int a,b;

    a = 6;
    b = a - 2;

    if( a > b)
        printf("%d est plus grand que %d\n", a, b);
    return(0);
}
```

Remarques

Une convention bien partagée est d'indenter l'instruction unique d'un bloc conditionnel `if` en la rejetant sur une nouvelle ligne, mais ceci est autorisé :

```
if( a > b) printf("%d est plus grand que %d\n",a,b);
```

Pour plus de robustesse du code source, même lorsqu'il n'y a qu'une instruction, prévoyez les accolades même lorsque vous savez qu'elles sont facultatives. Vous pourrez les omettre quand vous aurez quelques centaines d'heures de programmation.

ex0804

```
#include <stdio.h>

int main()
{
    int premier, second;

    printf("Indiquez la valeur de premier : ");
    scanf("%d", &premier);
    printf("Indiquez la valeur de second : ");
    scanf("%d", &second);

    puts("Evaluation en cours...");
    if(premier < second)
    {
        printf("%d est plus petit que %d\n", premier, second);
    }
    if(premier > second)
    {
        printf("%d est plus grand que %d\n", premier, second);
    }
    return(0);
}
```

ex0805

```
#include <stdio.h>

int main()
{
    int premier, second;

    printf("Indiquez la valeur de premier : ");
    scanf("%d", &premier);
    printf("Indiquez la valeur de second : ");
    scanf("%d", &second);

    puts("Evaluation en cours...");
    if(premier < second)
    {
        printf("%d est plus petit que %d\n", premier, second);
    }
    if(premier > second)
    {
        printf("%d est plus grand que %d\n", premier, second);
    }
    if(premier == second)
    {
        printf("%d et %d sont identiques\n", premier, second);
    }
    return(0);
}
```

Remarques

L'opérateur de comparaison correspond au double signe égal.

Quand vous voyez `==`, il faut lire "est égal à".

L'opérateur `==` ne peut pas être utilisé pour comparer des chaînes de caractères.

ex0806

```
#include <stdio.h>

#define SECRET 17

int main()
{
    int devinessai;

    printf("Tentez de deviner le chiffre secret : ");
    scanf("%d", &devinessai);
    if(devinessai == SECRET)
    {
        puts("Bravo !");
        return(0);
    }
    if(devinessai != SECRET)
    {
        puts("Non, ce n'est pas cela !");
        return(1);
    }
}
```

Remarques

En théorie, un `return` final après la sortie du second test est inutile, car les deux conditions testées couvrent tous les cas possibles (elles sont absolues). Mais tenez compte de ce qui suit.

Le compilateur peut émettre un avertissement pour dire que la fonction `main()` étant de type `int`, elle doit renvoyer une valeur de ce type (un code de fin). C'est le genre de message que vous apprendrez à ignorer, mais seulement lorsque vous saurez sans hésiter quand l'instruction `return` n'est pas requise.

En général, vous pouvez placer un `return` n'importe où dans une fonction, mais restez logique. Un test avec `if` peut confirmer que la fonction n'a plus rien à faire et en forcer la fin par un `return`. Si la condition n'est pas satisfaite, le traitement continue avec la suite de la fonction. Dans ce cas précis, un `return` final reste obligatoire, surtout dans `main()` pour renvoyer un code de fin au système d'exploitation.

ex0807

```
#include <stdio.h>

int main()
{
    int a;

    a = 5;

    if(a =- 3)                          // L09
    {
        printf("%d egale %d\n", a, -3);
    }
    return(0);
}
```

Remarques

En langage C, une opération d'affectation de valeur renvoie toujours la valeur TRUE (heureusement qu'une copie mémoire réussit toujours !). Le test `if` et donc toujours satisfait quelle que soit la valeur de `a`. Ici, l'égalité est vérifiée, mais parce que nous avons modifié la variable sans le vouloir.

Modifiez maintenant le test comme ceci :

```
if(a = 5)
```

Recompilez et exécutez. Vous voyez s'afficher `5 egale -3` parce que le test a réussi pour la mauvaise raison (affectation toujours vraie), pas parce que la variable `a` vaut 5.

Le but de cet exercice n'est pas d'utiliser des affectations dans des tests, mais au contraire de vous rappeler qu'il ne faut jamais oublier que la comparaison correspond à deux signes égal successifs.

Il est possible que le compilateur se plaigne en voyant une affectation là où il ne devrait trouver qu'un opérateur de comparaison (`a=-3` ou `a=5`). Un message peut vous demander d'ajouter des parenthèses autour d'une affectation si elle doit servir de valeur vraie. C'est l'indice d'une confusion entre simple et double signe égal.

Pour être exact, une affectation renvoie la valeur FALSE, faux (mais elle réussit tout de même !) dans un cas unique :

```
if(a = 0)
```

En effet, vous demandez d'écrire dans la variable la valeur zéro, et zéro vaut FALSE. Méfiez-vous, car des confusions d'opérateurs peuvent de ce fait rester non détectées.

ex0808

```
#include <stdio.h>

int main()
{
    int a;

    a = 5;

    if(a == -3)
    {
        printf("%d egale %d\n", a, -3);
    }
    return(0);
}
```

Remarques

Le programme est maintenant correct, et il n'affiche rien pour une bonne raison : les valeurs sont différentes.

Sauriez-vous ajouter un caractère au code pour que le test réussisse et que le message s'affiche ? Normalement, cela suppose aussi de modifier le message de `printf()`, mais ce n'est pas le sujet ici.

ex0809

```
#include <stdio.h>

int main()
{
    int a,b;

    a = 5;
    b = -3;
```

```
    if(a == b);                          // 10
        printf("%d egale %d\n", a, b);
    return(0);
}
```

Remarques

Le point-virgule en fin de ligne 10 est sans doute une erreur, vue la ligne 11 indentée. Le compilateur ne va rien trouver de suspect ; selon lui, tout va bien.

Ce genre d'erreur est vicieuse, car difficile à repérer.

ex0810

```
#include <stdio.h>

int main()
{
    int a,b;

    a = 6;
    b = a - 2;

    if( a > b)
    {
        printf("%d est plus grand que %d\n", a, b);
    }
    else
    {
        printf("%d n'est PAS plus grand que %d\n", a, b);  // L16
    }
    return(0);
}
```

Remarques

La ligne 16 ne dit pas "est inférieur à" parce que la valeur peut aussi être égale à l'autre.

Les accolades ne sont pas obligatoires si le `if` ne gère qu'une instruction ; de même pour un bloc `else`.

Chaque sous-bloc conditionnel est autonome : si un bloc contrôle plusieurs instructions, il faut des accolades.

ex0811

```
#include <stdio.h>

int main()
{
    int a,b;

    a = 6;
    printf("Fournissez une valeur entière : ");
    scanf("%d", &b);                              // L09
    if( a > b)
    {
        printf("%d est plus grand que %d\n", a, b);
    }
    else
    {
        printf("%d n'est PAS plus grand que %d\n", a, b);
    }
```

```
    return(0);
}
```

Remarque

Vous connaissez les fonctions `printf()` et `scanf()`. Vous vous souvenez notamment que `scanf()` doit recevoir en entrée **l'adresse** de la variable réceptrice en ajoutant le préfixe `&` avant le nom de variable `b` (ligne 9).

ex0812

```
#include <stdio.h>

#define SECRET 17

int main()
{
    int devinessai;

    printf("Tentez de deviner le chiffre secret : ");
    scanf("%d", &devinessai);
    if(devinessai == SECRET)
    {
        puts("Bravo !");
        return(0);
    }
    else
    {
        puts("Non, ce n'est pas cela !");
        return(1);
    }
}
```

Remarques

Nous repartons ici de **ex0806** (Listing 8.3). La solution rapide dont je parle dans le livre concerne la ligne 16 : remplacez `if(devinessai!=SECRET)` par un simple `else`.

Le résultat fonctionnel est strictement le même, mais la structure à deux branches `if-else` permet de gérer la condition de façon absolue (tous les cas sont prévus). Deux blocs `if` successifs, donc indépendants, forment deux évaluations. Sans entrer dans les détails techniques du gain en performances, il vous sera utile de savoir qu'à un même résultat utile peuvent correspondre des approche différentes.

ex0813

```
#include <stdio.h>

int main()
{
    int premier,second;

    printf("Indiquez la valeur de premier : ");
    scanf("%d",&premier);
    printf("Indiquez la valeur de second : ");
    scanf("%d",&second);

    puts("Evaluation en cours...");
    if(premier < second)
    {
        printf("%d est plus petit que %d\n", premier, second);
    }
```

```
    else if(premier > second)
    {
        printf("%d est plus grand que %d\n", premier, second);
    }
    else
    {
        printf("%d et %d sont identiques\n", premier, second);
    }
    return(0);
}
```

Remarque

La condition qu'intercepte la nouvelle branche `else` est l'égalité `==`. Vous auriez pu ajouter une branche `else if`, mais c'est inutile, d'autant que cela vous aurait obligé à ajouter une branche finale `else` sans rien d'autre à tester.

ex0814

```
#include <stdio.h>

int main()
{
    int coordonnee;

    printf("Coordonnees de la cible : ");
    scanf("%d", &coordonnee);
    if( coordonnee >= -5 && coordonnee <= 5 )      // L09
    {
        puts("Assez proche !");
    }
    else
    {
        puts("La cible est encore loin !");
    }
    return(0);
}
```

Remarques

Le test de comparaison en ligne 9 est inclusif (bornes -5 et 5 comprises) du fait que nous utilisons les opérateurs `>=` et `<=`. En exclusif, nous aurions écrit :

```
if( coordonnee > -6 && coordonnee < 6 )
```

La condition `else` est satisfaite lorsque `coordonnee` vaut moins de -5 ou plus de 5, soit l'exacte inverse de la ligne 9.

ex0815

```
#include <stdio.h>

int main()
{
    int coordonnee;

    printf("Coordonnees de la cible :: ");
    scanf("%d",&coordonnee);
    if( coordonnee < -5 || coordonnee > 5 )
    {
        puts("Assez proche !");
```

```
    }
    else
    {
        puts("La cible est encore loin !");
    }
    return(0);
}
```

Remarques

Pour le test en ligne 9, le plus simple est de relire le livre : "Modifiez les conditions pour que la variable soit inférieure à -5 ou supérieure à 5.". Voyons cette nouvelle ligne 9 :

`if( coordonnee < -5 || coordonnee > 5 )`

Elle se lit : "Si la valeur de `coordonnee` est inférieure à -5 OU si la valeur de `coordonnee` est supérieure à 5." C'est bien ce qui est demandé.

Vos comparaisons seront correctes tant que vous ne vous méprenez pas sur le sens exact des opérateurs `<`, `||` et `>`.

ex0816

```
#include <stdio.h>

int main()
{
    char yorn;

    printf("Voulez-vous continuer (Y/N)? ");       // L07
    scanf("%c", &yorn);
    if( yorn == 'Y' || yorn == 'y' )   // Localisable en 'O', 'o'
    {
        puts("On continue...");
    }
    else if( yorn == 'N' || yorn == 'n' )
    {
        puts("D'accord, je sors.");
    }
    else                                    // L17
    {
        puts("Alors, c'est ni oui (Y), ni non (N) ?");
    }
    return(0);
}
```

Remarques

N.d.T. : Vous pouvez bien sûr créer une version française avec détection d'un O au lieu d'un Y pour Oui.

Le secret ici est de bien utiliser deux fois l'opérateur `||` , d'abord pour tester la lettre du oui, puis pour le N du non.

J'ai opté pour une structure `if-else-if-else` pour gérer les trois cas possibles. Bravo si vous y avez pensé aussi.

L'invite de saisie en ligne 7 est à adapter selon vos choix et vos préférences.

Dans la ligne suivante, `scanf()` a été préféré à `getchar()` pour le plaisir, car il me semble que je n'ai pas utilisé assez cette fonction dans le livre. Mais `getchar()` va aussi bien.

Le test en ligne 9 mérite une remarque, car il utilise un OU logique pour intercepter la touche en majuscule comme en minuscule. N'oubliez jamais que l'opérateur de comparaison est la séquence de

deux signes égal. Si vous en oubliez un, le programme va continuer pour toutes les touches (une affectation est toujours vraie).

Le test `else-if` en ligne 13 est le même pour l'autre lettre à détecter (`N` ou `n`).

Enfin, en ligne 17, le bloc `else` gère l'indécision du joueur (ou sa frappe de Y pour un O attendu).

En anglais, la réponse par oui ou non est souvent désignée par **yorn** (Yes OR No).

Vos messages peuvent bien sûr être différents des miens et vous pouvez préférer `printf()` à `puts()`.

Nous n'avons pas besoin des accolades de blocs lorsqu'il n'y a qu'une instruction, mais le résultat est plus lisible.

ex0817

```
#include <stdio.h>

int main()
{
    int code;

    printf("Indiquez le code erreur (1-3): ");
    scanf("%d", &code);

    switch(code)
    {
        case 1:
            puts("Erreur disque, vous n'y pouvez rien.");
            break;
        case 2:
            puts("Format invalide, appelez votre avocat.");
            break;
        case 3:
            puts("Nom de fichier incorrect, punition.");
            break;
        default:
            puts("Haha, ni 1, ni 2, ni 3 ?");
    }
    return(0);
}
```

Remarques

La variable de contrôle de `switch` doit être de type simple (pas de chaîne). Vous pouvez y faire un appel à une fonction qui renvoie une valeur simple, mais pas une expression de comparaison numérique ou logique.

Une instruction `case` peut être muette (sans valeur), mais cela a peu d'intérêt. La valeur peut être un caractère entre apostrophes.

Le mot clé `break` est facultatif.

Le mot clé `default` est le dernier cas. Il peut être vide et n'a pas besoin de `break`.

ex0818

```
#include <stdio.h>

int main()
{
    char code;
```

```
    printf(" Indiquez la lettre d'erreur (A, B, C): ");
    scanf("%c", &code);

    switch(code)
    {
        case 'A':
            puts("Erreur disque, vous n'y pouvez rien.");
            break;
        case 'B':
            puts("Format invalide, appelez votre avocat.");
            break;
        case 'C':
            puts("Nom de fichier incorrect, punition.");
            break;
        default:
            puts("Haha, ni A, ni B, ni C ?");       // L22
    }
    return(0);
}
```

Remarques

En ligne 8, `getchar()` peut remplacer `scanf()`, car l'effet est le même pour saisir un caractère isolé.

Les caractères doivent être entre apostrophes. (Entre guillemets, ce seraient des chaînes d'une lettre.)

Vous n'avez pas omis de corriger aussi la ligne 22 ?

ex0819

```
#include <stdio.h>

int main()
{
    char choixMenu;

    puts("Nos formules du jour :");
    puts("A - Petit dejeuner, midi et soir");
    puts("B - Demi-pension, matin et soir");
    puts("C - Repas du soir seul");
    printf("Votre choix : ");
    scanf("%c", &choixMenu);

    printf("Vous avez choisi ");
    switch(choixMenu)
    {
        case 'A':
            printf("Petit dejeuner, ");
        case 'B':
            printf("Repas du midi, ");
        case 'C':
            printf("Repas du soir ");
        default:
            printf("comme formule de restauration.\n");
    }
    return(0);
}
```

Remarques

N.d.T. : Vous aurez remarqué que les messages des cas A et B sont inversés ?

Faites plusieurs essais de réponse.

Vous constatez que le bloc `default` est toujours exécuté.

En l'absence de `break` comme ici, l'exécution passe par tous les cas dans le sens de lecture. Méfiez-vous, c'est rarement volontaire.

ex0820

```
#include <stdio.h>

int main()
{
    char choixMenu;

    puts("Nos formules du jour :");
    puts("A - Petit dejeuner, midi et soir");
    puts("B - Demi-pension, matin et soir");
    puts("C - Repas du soir seul");
    printf("Votre choix : ");
    scanf("%c", &choixMenu);

    printf("Vous avez choisi ");
    switch(choixMenu)
    {
        case 'A':
        case 'a':
            printf("Petit dejeuner, ");
        case 'B':
        case 'b':
            printf("Repas du midi, ");
        case 'C':
        case 'c':
            printf("Repas du soir ");
        default:
            printf("comme formule de restauration.\n");
    }
    return(0);
}
```

Remarques

N.d.T. : Nous faisons bien sûr référence au dernier exercice, donc le 8.19, pas le 8.18.

En empilant deux bloc `case`, nous gérons sans souci les deux casses de lettres (majuscules/minuscules).

Le langage C offre d'autres moyens pour gérer la casse des lettres, notamment des fonctions standard. Le livre les présente plus loin.

ex0821

```
#include <stdio.h>

int main()
{
    int a, b, leplusgrand;

    printf("Indiquez une valeur A: ");
    scanf("%d", &a);
    printf("Indiquez une autre valeur B: ");
    scanf("%d", &b);

    leplusgrand = (a > b) ? a : b;
```

```
    printf("La valeur %d est plus grande.\n", leplusgrand);
    return(0);
}
```

Remarques

Certains programmeurs privilégient l'opérateur `?:` qui rend leur code source plus mystérieux.

Ce n'est que dans de rares cas que cet opérateur est vraiment incontournable. Nous en verrons un exemple dans le Chapitre 17.

Si l'opérateur `?:` vous gêne, vous pouvez souvent le remplacer par un bloc `if-else`. Mais sachez que c'est un élément du langage C dont d'autres langages ont pris soin de s'inspirer. L'étrange opérateur ternaire va vous hanter longtemps !

ex0822

```
#include <stdio.h>

int main()
{
    int a, b, leplusgrand;

    printf("Indiquez une valeur A: ");
    scanf("%d", &a);
    printf("Indiquez une autre valeur B: ");
    scanf("%d", &b);

    if( a > b )
        leplusgrand = a;
    else
        leplusgrand = b;
    printf("La valeur %d  est plus grande.\n", leplusgrand);
    return(0);
}
```

Remarques

Ce code remplit son rôle, mais il lui manque la concision qu'offre l'opérateur `?:`.

Chapitre 09

ex0901

```
#include <stdio.h>

int main()
{
    int x;

    for(x=0; x<10; x=x+1)
    {
        puts("Ne vous l'ais-je pas encore dit ?");
    }
    return(0);
}
```

Remarques

N.d.T. : j'ai pensé que le texte affiché constituait une bonne question à brûle-pourpoint.

ex0902

```
#include <stdio.h>

int main()
{
    int x;

    for(x=0; x<20; x=x+1)
    {
        puts("Ne vous l'ais-je pas encore dit ?");
    }
    return(0);
}
```

Remarque

Même sans encore rien connaître de la syntaxe des boucles `for`, vous devinez que le deuxième membre va contrôler le nombre de tours de boucle.

ex0903

```
#include <stdio.h>

int main()
{
    int cmptr;

    for(cmptr=-5; cmptr<6; cmptr=cmptr+1)
    {
        printf("%d\n", cmptr);
    }
    return(0);
}
```

Remarques

Une fois l'initialisation de la variable locale faite, le premier membre de l'instruction `for` n'est plus utilisé. La valeur de `cmptr` est réglée à -5, puis c'est dans le deuxième membre que la variable `cmptr` est comparée au début de chaque tour de boucle à la valeur littérale 6. Tant qu'elle lui est inférieure, on effectue un nouveau tour de boucle.

En fin de boucle et juste avant de refaire un test, le troisième membre, `cmptr=cmptr+1`. , est exécuté pour faire progresser la variable. Le test est fait juste après.

La plupart des boucles `for` sont écrites comme dans le Listing 9.1 du livre, mais la première valeur du compteur n'est pas obligatoirement égale à zéro.

ex0904

```
#include <stdio.h>

int main()
{
    int cmptr;

    for(cmptr=11; cmptr<=19; cmptr=cmptr+1)
    {
        printf("%d\t", cmptr);        // L09
    }
    putchar('\n');                    // L11
```

```
    return(0);
}
```

Remarques

Pour afficher les valeurs entre 11 et 19, la valeur de `cmptr` a été initialisée à 11.

La valeur de fin 19 set de condition de sortie de boucle `for`. L'opérateur `<=` permet d'indiquer cette valeur 19. Avec l'opérateur plus habituel `<` , il aurait fallu écrire `cmptr<20`.

La partie répétée (le corps de boucle) affiche tour à tour les valeurs de 11 à 19.

En ligne 9, la séquence `\t` symbolise le caractère de tabulation qui sert à séparer les valeurs à l'affichage pour plus de confort.

En ligne 11, nous appelons `putchar()` pour ajouter un saut de ligne final de propreté avec `\n`. Nous aurions pu écrire ceci :

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

Voici ce qui s'est affiché chez moi :

```
11    12    13    14    15    16    17    18    19
```

ex0905

```
#include <stdio.h>

int main()
{
    int duo;

    for(duo=2; duo<=100; duo=duo+2)
    {
        printf("%d\t", duo);
    }
    putchar('\n');
    return(0);
}
```

Remarques

L'augmentation de valeur dans le troisième membre de l'instruction *for* n'est pas limitée à des pas de 1.

N.d.T. : vous pouvez même créer une boucle avec une valeur initiale élevée qui décroit jusqu'à la valeur d'arrêt.

ex0906

```
#include <stdio.h>

int main()
{
    int tre;

    for(tre=3; tre<=100; tre=tre+3)   // L07
    {
        printf("%d\t", tre);
    }
    putchar('\n');
    return(0);
}
```

Remarques

Vous auriez pu vous contenter de retoucher les premier et troisième membre en ligne 7. Je suis allé plus loin en créant une nouvelle variable pour remplacer `duo` par `tre`.

La valeur limite 100 reste la même.

ex0907

```
#include <stdio.h>

int main()
{
    int aRebours;

    for(aRebours=25; aRebours>=0; aRebours=aRebours-1)    // L07
    {
        printf("%d\n",aRebours);
    }
    return(0);
}
```

Remarques

Tout est dans la ligne 7, que vous ayez ou non choisi le nom de variable `aRebours`.

Vous auriez pu écrire `aRebours>-1` comme condition de sortie. La valeur 0 serait affichée en dernier aussi.

ex0908

```
#include <stdio.h>

int main()
{
    char alphabet;

    for(alphabet='A'; alphabet<='Z'; alphabet=alphabet+1)
    {
        printf("%c", alphabet);          // L09
    }
    putchar('\n');
    return(0);
}
```

Remarque

L'affiche n'aurait pas changé en écrivant en ligne 9 `putchar(alphabet)`.

ex0909

```
#include <stdio.h>

int main()
{
    char alphabet;

    for(alphabet='A'; alphabet<='Z'; alphabet=alphabet+1)
    {
        printf("%d ", alphabet);
    }
    putchar('\n');
    return(0);
```

```
}
```

Remarques

Si vous remplacez le formateur `%d` par `%c`, l'affichage devient :

```
656667686970717273747576777879808182838485868788899 0
```

Dans ma solution, j'ai ajouté un espace après `%d` dans `printf()` pour améliorer la lecture :

```
65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89
90
```

Le programme affiche les codes ASCII des lettres de A à Z.

ex0910

```
#include <stdio.h>

int main()
{
    char alphabet;

    for(alphabet='z'; alphabet>='a'; alphabet=alphabet-1)
    {
        printf("%c", alphabet);
    }
    putchar('\n');
    return(0);
}
```

Remarque

Toutes les retouches à partir du Listing 9.4 se déroulent dans la ligne 7. Le `'A'` devient un `'z'` et le `'Z'` un `'a'`. Le troisième membre de *for* fait une décrémentation (moins un) de la valeur de la variable `alphabet`.

ex0911

```
#include <stdio.h>

int main()
{
    int alpha,code;

    for(alpha='A'; alpha<='G'; alpha=alpha+1)
    {
        for(code=1; code<=7; code=code+1)
        {
            printf("%c%d\t", alpha,code);
        }
        putchar('\n');      /* Saut de ligne final */
    }
    return(0);
}
```

Remarques

La boucle externe (`alpha`) incarne les lignes et la boucle interne (`code`) des colonnes.

Une boucle imbriquée ressemble à une horloge : la boucle externe pour les heures et l'autre pour les minutes. Dans ma solution ex0911, la boucle externe est sur le 'A' puis la boucle interne va de 1 à 7. Ensuite, la boucle externe passe au 'B', etc.

ex0912

```
#include <stdio.h>

int main()
{
    int a,b,c;

    for(a='A'; a<='Z'; a=a+1)
    {
        for(b='A'; b<='Z'; b=b+1)
        {
            for(c='A'; c<='Z'; c=c+1)
            {
                printf("%c%c%c\n", a, b, c);    // L13
            }
        }
    }
    return(0);
}1
```

Remarque

Vous pouviez remplacer en ligne 13 la séquence `\n` par `\t`. Dans ce cas, il faut ajouter un appel `putchar('\n')` en sortie de boucle pour rendre l'affichage propre.

ex0913

```
#include <stdio.h>

int main()
{
    int x;

    x = 0;
    while(x<10)
    {
        puts("Ne vous l'ais-je pas encore dit ?");
        x = x+1;
    }
    return(0);
}
```

Remarques

L'affichage ne vous est pas inconnu puisque nous venons de reformuler avec `while` la solution avec `for` de **ex0901**.

ex0914

```
#include <stdio.h>

int main()
{
    int x;

    x = 13;
    while(x<10)
    {
        puts("Ne vous l'ais-je pas encore dit ?");
        x = x+1;
```

```
    }
    return(0);
}
```

Remarque

La variable étant dès le départ supérieure à la condition `x<10`, la boucle n'est pas même exécutée une fois.

ex0915

```
#include <stdio.h>

int main()
{
    float demipas;                // L05

    demipas = -5.0;               // L07
    while(demipas <= 5.0)
    {
        printf("%f\n", demipas); // L10
        demipas = demipas + 0.5;
    }
    return(0);
}
```

Remarques

Puisque nous voulons progresser d'une demi-unité à la fois, nous devons utiliser le type flottant pour la variable `demipas` en ligne 5.

En ligne 7, nous initialisons `demipas` à la valeur flottante `-5.0`. Nous avons pris soin d'ajouter de quoi prévenir le compilateur qu'il ne s'agit pas d'un entier.

Dans la ligne suivante, nous configurons la boucle `while` : dès que la valeur de `demipas` dépasse `5.0`, on sort (valeur flottante aussi).

En ligne 10,nous affichons la valeur actuelle de `demipas`.

Enfin, en ligne 11, nous incrémentons la variable `demipas` de `0.5`.

ex0916

```
#include <stdio.h>

int main()
{
    int fibo,nacci;

    fibo  = 0;        // L07
    nacci = 1;

    do
    {
        printf("%d ", fibo);
        fibo = fibo+nacci;
        printf("%d ", nacci);
        nacci = nacci+fibo;
    } while( nacci < 300 );

    putchar('\n');
    return(0);
}
```

Remarques

Notez bien l'espace de lisibilité dans les deux appels à `printf()`.

La séquence de Fibonacci est un phénomène mathématique que la nature a adopté. Elle comprend les valeurs 0 et 1 (lignes 7 et 8). La valeur suivante est l'addition des deux précédentes. C'est le travail de la boucle `do-while`.

ex0917

```
#include <stdio.h>

int main()
{
    int x;

    x = 13;
    do
    {
        puts("Ne vous l'ais-je pas encore dit ?");
        x = x+1;
    } while(x<10);
    return(0);
}
```

Remarque

Cet exemple affiche du texte parce que la boucle `do-while` s'exécute au moins une fois avant de tester la condition de sortie qui est pourtant fausse dès le départ.

ex0918

```
#include <stdio.h>

int main()
{
    int x;

    for(x=0; x=10; x=x+1)
    {
        puts("Vous cherchiez quelque chose ?");
    }
    return(0);
}
```

Remarques

Le compilateur vous prévient d'un problème de parenthèses, mais c'est en fait votre confusion entre l'opérateur d' affectation `=` et celui de comparaison `==`. Je ne sais pas en détail pourquoi l'indice fourni concerne une parenthèse.

Par ailleurs, la valeur de `x` va croître, mais ne risque pas de faire exploser la mémoire de la machine ni le programme. A force de `x=x+1`, elle va atteindre la valeur maximale du type `signed int`. Elle va ensuite dérouler les valeurs entières négatives puis revenir à zéro et continuer jusqu'à la fin du monde.

ex0919

```
#include <stdio.h>

int main()
{
    int cmptr;
```

```
    cmptr = 0;
    while(1)
    {
        printf("%d, ", cmptr);
        cmptr = cmptr+1;
        if( cmptr > 50)
            break;
    }
    putchar('\n');
    return(0);
}
```

Remarques

Vous pouvez préférer une boucle `while` avec `while(cmptr<=50)`. Le résultat est le même. Un exemple plus réaliste montrerait que la valeur de `cmptr` n'est plus un compteur, mais une variable modifiée par une fonction appelée.

La condition de sortie de boucle est en général une action qui fait changer la valeur renvoyée par une fonction appelée, ou la simple valeur de la saisie utilisateur.

En pratique ce genre de boucle figée ne sert pas à afficher quoi que ce soit.

ex0920

```
#include <stdio.h>

int main()
{
    int cmptr;

    cmptr = 0;
    for(;;)
    {
        printf("%d, ", cmptr);
        cmptr = cmptr+1;
        if( cmptr > 50)
            break;
    }
    putchar('\n');
    return(0);
}
```

Remarque

La seule retouche concerne la ligne 8.

ex0921

```
#include <stdio.h>

int main()
{
    int x;

    for( x=0; x<10; x=x+1, printf("%d\n", x) )
        ;
    return(0);
}
```

Remarques

Tout les traitements sont réunis dans la seule ligne 7 :

- D'abord, nous forçons à zéro la variable `x`.

- Puis la boucle tourne tant que la valeur de `x` reste inférieure à 10. Tout va bien.

- Enfin, le troisième membre du `for` contient deux instructions exécutées pour chaque tour de boucle. D'abord, nous incrémentons de un la variable compteur par `x=x+1`, puis, après une virgule, nous insérons l'instruction d'affichage complète `printf("%d\n", x)`. Cette approche ultra-compacte est tout à fait correcte.

Pour que tous vos collègues sachent qu'il n'y a pas d'autre instructions dans cette boucle, nous installons un point-virgule isolé sur sa propre ligne, avec la bonne indentation.

Chapitre 10

ex1001

```
#include <stdio.h>

void prompt();      /* Prototype */

int main()
{
    int loop;              // L07
    char input[32];

    loop=0;
    while(loop<5)
    {
        prompt();
        fgets(input, 31, stdin);
        loop = loop+1;
    }                      // L16
    return(0);
}

/* Fonction prompt() */

void prompt()
{
    printf("C:\\DOS> ");           // L24
}
```

Remarques

Vous devez valider cinq fois par Entrée pour sortir de la boucle.

Les deux antibarres `\\` en ligne 24 sont requises pour faire apparaître un signe antibarre dans la sortie, le premier servant à désactiver l'effet normal du suivant. On obtient ainsi l'affichage d'invite que les anciens (déjà !) ont connu au quotidien sous MS-DOS :

```
C:\DOS>
```

Vous pouvez modifier les lignes 3 et 22 pour que le mot clé `void` soit présent entre parenthèses. En ligne 3 :

```
void prompt(void);    /* Prototype */
```

En ligne 22 :

```
void prompt(void)
```

Ce n'est pas obligatoire, mais je préfère ajouter cette mention pour que tout soit clair : ma fonction n'attend aucun argument d'entrée. Avec des parenthèses vides, on se demande par réflexe si on n'a pas oublié quelque choses.

Vous remarquez au passage qu'en revanche, la fonction principale `main()` ne profite pas de cette précaution basée sur `void` dans l'argument. Pourtant, elle peut recevoir des arguments, mais c'est en option. Elle est à géométrie variable. Pas de mention `void` dans les paramètres de *main()* même si vous savez que vous ne fournirez jamais d'arguments au démarrage du programme. L'écriture `main(void)` est interdite par le compilateur.

ex1002

```
#include <stdio.h>

void prompt(void);      /* Prototypes */
void busy(void);

int main()
{
    busy();             // Appel de fonction
    return(0);
}

/* Cinq tours de boucle */

void busy(void)
{
    int loop;
    char input[32];

    loop = 0;
    while(loop<5)
    {
        prompt();
        fgets(input, 31, stdin);
        loop = loop+1;
    }
}

/* Fonction prompt() */

void prompt(void)
{
    printf("C:\\DOS> ");
}
```

Remarques

Vous n'aviez pas oublié de créer le prototype de la nouvelle fonction `busy()` ? Ce n'était pas sympa de ma part de ne pas vous en prévenir dans le livre, mais c'est comme cela que l'on apprend.

L'ordre des prototypes n'a pas d'importance, mais les programmeurs aiment les découvrir à peu près dans l'ordre dans lequel les fonctions sont appelées.

Les variables locales d'une fonction doivent y être déclarées. Le déplacement des lignes 7 à 16 vers la nouvelle fonction `busy()` permet de récupérer les déclarations de `loop` et de `input`. Si nous avions laissé ces déclarations dans la fonction `main()` , la fonction `busy()` n'y aurait pas eu accès. En C, les variables sont par défaut locales à la fonction qui les déclare.

Ce code montre comment appeler une fonction depuis une autre.

ex1003

```
#include <stdio.h>

//void prompt(void);         /* Prototype */

int main()
{
    int loop;
    char input[32];

    loop=0;
    while(loop<5)
    {
        prompt();
        fgets(input, 31, stdin);
        loop = loop+1;
    }
    return(0);
}

/* Fonction prompt() */

void prompt(void)
{
    printf("C:\\DOS> ");
}
```

Remarque

J'ai neutralisé la ligne 3 avec le couple `//`, plus adapté aux lignes isolées. En utilisant `/*` , vous risquez de créer un commentaire imbriqué si vous ajoutez le fermant `*/` en fin de ligne, mais rares sont les éditeurs et compilateurs qui s'en plaignent.

ex1004

```
#include <stdio.h>

/* Fonction prompt() */

void prompt(void)
{
    printf("C:\\DOS> ");
}

int main()
{
    int loop;
    char input[32];

    loop=0;
    while(loop<5)
    {
        prompt();
        fgets(input, 31, stdin);
        loop = loop+1;
    }
    return(0);
}
```

ex1005

```
#include <stdio.h>

void vegas(void);

int main()
{
    int a;

    a = 365;
    printf("Dans la fonction main(), a=%d\n",a);
    vegas();
    printf("De retour dans main(), a=%d\n",a);
    return(0);
}

void vegas(void)
{
    int a;

    a = -10;
    printf("Dans la fonction vegas(), a=%d\n",a);
}
```

Remarques

Les variables locales déclarées dans une fonction ne perturbent pas les homonymes d'une autre fonction, mais elles les masquent. Attention donc lorsque vous créez une homonyme d'une variable de `main()`.

Ne vous refrénez pas sur le nombre de variable locales dans une fonction. S'il en faut douze pour que la fonction atteigne son but, déclarez-en douze !

Personnellement, je choisis des noms de variables courts quand cela m'arrange, comme dans les boucles où je me limite à `x` ou `ch`. Ce sont chez moi des variables locales. En revanche, les variables globales ou à longue portée ont des noms plus explicites comme `statut`, `ligne_courante` ou `totalGen`. Même si ce genre de variable ne sert que dans une fonction, je ne réutilise pas son nom ailleurs, pour plus de lisibilité. Mais tout est affaire de goût.

Bien que je parle de variables locales, ce n'est pas un terme officiel du langage C. Toutes les variables, sauf celles déclarées globalement sont locales à une fonction. Les variables globales seront vues dans le Chapitre 16.

ex1006

```
#include <stdio.h>

void graph(int cmptr);

int main()
{
    int valeur;

    valeur = 2;

    while(valeur<=64)
    {
        graph(valeur);
        printf("La valeur est %d\n", valeur);
        valeur = valeur * 2;                // L15
```

```
    }
    return(0);
}

void graph(int cmptr)
{
    int x;

    for(x=0; x<cmptr; x=x+1)
        putchar('*');
    putchar('\n');
}
```

Remarques

L'équation en ligne 15 double le contenu de `valeur` à chaque tour de boucle. Du fait qu'au départ `valeur` contient 2, nous obtenons un graphe des puissances de deux.

Il est autorisé de déclarer le prototype d'une fonction sans fournir les noms des variables de paramètres, comme ceci :

```
void graph(int);
```

En revanche, dans l'en-tête de fonction (ligne 20), il faut indiquer le ou les paramètres. Ici, nous nous en servons comme variable locale contenant la valeur transmise en entrée.

ex1007

```
#include <stdio.h>

void graph(int cmptr);

int main()
{
    int valeur;

    valeur = 2;

    while(valeur<=64)
    {
        graph(64);
        printf("La valeur est %d\n", valeur);
        valeur = valeur * 2;
    }
    return(0);
}

void graph(int cmptr)
{
    int x;

    for(x=0; x<cmptr; x=x+1)
        putchar('*');
    putchar('\n');
}
```

Remarque

Et un point de bonus Pour les nuls si vous avez créé la constante avec une directive `#define`.

ex1008

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

```
void graph(int cmptr,char ch);

int main()
{
    int valeur;

    valeur = 2;

    while(valeur<=64)
    {
        graph(valeur, '=');
        printf("La valeur est %d\n", valeur);
        valeur = valeur * 2;
    }
    return(0);
}

void graph(int cmptr, char ch)
{
    int x;

    for(x=0; x<cmptr; x=x+1)
        putchar(ch);
    putchar('\n');
}
```

Remarques

L'erreur de retouche la plus probable est l'oubli de modification de l'appel à `graph()` en ligne 13 ; elle attend maintenant deux valeurs.

N'avez-vous pas oublié de retoucher le prototype de `graph()` ?

ex1009

```
#include <stdio.h>

float convertir(float f);

int main()
{
    float temp_f, temp_c;

    printf("Temperature en Fahrenheit : ");
    scanf("%f", &temp_f);
    temp_c = convertir(temp_f);
    printf("%.1fF vaut %.1fC\n", temp_f, temp_c);
    return(0);
}

float convertir(float f)
{
    float t;

    t = (f - 32) / 1.8;
    return(t);
}
```

Remarques

Nous avons choisi `temp_f` et `temp_c` pour éviter que les noms de variables se résument à `f` et `c`. Les noms `fahrenheit` et `celsius` auraient rendu la ligne de `printf()` trop longue pour la version imprimée.

Ne soyez pas choqué par le formateur `%.1f` qui fait bien le travail de `%f`, mais avec une indication de précision avant le `f`. Le F majuscule qui le suit est affiché tel quel pour indiquer l'unité.

ex1010

```
#include <stdio.h>

float convertir(float f);

int main()
{
    float temp_f;

    printf("Temperature en Fahrenheit : ");
    scanf("%f", &temp_f);
    printf("%.1fF vaut %.1fC\n", temp_f, convertir(temp_f));
    return(0);
}

float convertir(float f)
{
    float t;

    t = (f - 32) / 1.8;
    return(t);
}
```

Remarques

L'autre retouche demandée consiste à se passer de la variable `temp_c` devenue inutile. Il faut supprimer sa déclaration en ligne 7 et ses utilisations en lignes 11 et 12.

Le compilateur ne vous prévient pas obligatoirement qu'il a détecté une variable déclarée mais jamais utilisée.

ex1011

```
#include <stdio.h>

float convertir(float f);

int main()
{
    float temp_f;

    printf(" Temperature en Fahrenheit : ");
    scanf("%f", &temp_f);
    printf("%.1fF vaut %.1fC\n", temp_f, convertir(temp_f));
    return(0);
}

float convertir(float f)
{
    return(f - 32) / 1.8;
}
```

Remarque

Pour être encore plus clair, vous pouvez délimiter toute l'équation dans la fonction entre parenthèses :

`return((f - 32) / 1.8);`

Rappelons que les parenthèses sont facultatives dans `return`.

ex1012

```
#include <stdio.h>

void limiter(int stop);

int main()
{
    int s;

    printf("Indiquez une valeur pour stopper (0-100): ");
    scanf("%d",&s);
    limiter(s);
    return(0);
}

void limiter(int stop)
{
    int x;

    for(x=0; x<=100; x=x+1)
    {
        printf("%d ", x);
        if(x == stop)
        {
            puts("Vous gagnez !");
            return;
        }
    }
    puts("Je gagne !");
}
```

Remarque

L'instruction `return` en ligne 25 a le même effet qu'un `break` dans une boucle, sauf que le `break` laisse l'exécution se poursuivre après la boucle alors que le `return` fait ressortir du niveau de sous-programme et revient donc au niveau supérieur (donc au système si on est dans `main()`).

ex1013

```
#include <stdio.h>

#define TRUE 1
#define FALSE 0

void limiter(int stop);
int verifier(int valTest);

int main()
{
    int x;

    printf("Indiquez une valeur pour stopper (0-100): ");
    scanf("%d", &x);            // L14
```

```
    if(verifier(x))
    {
        limiter(x);
    }
    else
    {
        printf("%d est en dehors de la plage !\n", x);
    }
    return(0);
}

int verifier(int valTest)
{
    if(valTest < 0 || valTest > 100)
        return FALSE;
    return TRUE;
}

void limiter(int stop)
{
    int x;

    for(x=0; x<=100; x=x+1)
    {
        printf("%d ", x);
        if(x == stop)
        {
            puts("Vous gagnez !");
            return;
        }
    }
    puts("Je gagne !");
}
```

Remarques

Les constantes `TRUE` et `FALSE` sont définies en début de code, mais nous aurions pu directement indiquer la valeur 1 pour TRUE et 0 pour FALSE, non ? C'est plus lisible ainsi.

Les prototypes de fonctions viennent ensuite.

En ligne 14, une structure conditionnelle vérifie la validité de la saisie puis nous appelons la fonction `verifier()` en lui donnant la valeur lue par `scanf()`. La fonction (lignes 26 à 31) renvoie `TRUE` ou `FALSE` selon que la valeur est ou pas dans la plage. Si elle l'est, nous appelons la fonction `limiter()` et sinon, nous affichons un message d'erreur et quittons sur le champ.

Le seul moyen pour faire gagner l'ordinateur consisterait à faire générer une valeur aléatoire et demander au joueur de deviner une valeur inférieure. Amusez-vous à ajouter une fonction à cet effet.

Suite en Partie 3, Chapitre 11