Travaux pratiques n° 1 : Premiers programmes

Premières expérimentations avec le C

Une bonne manière d'apprendre le C est de lire et de modifier des programmes existants. Dans cette section, vous trouverez trois petits programmes très simples avec des rappels de cours et des suggestions pour faire vos premières expériences en programmation. Nous vous conseillons de répondre aux questions de cette partie, mais il ne faudra pas nous rendre les listings des programmes correspondants. Les programmes qui nous intéressent sont dans la partie intitulée « Exercices ».

« Hello, world »

Traditionnellement, le premier programme qu'on écrit doit afficher « Hello, world »

```
// tout ce qui suit une double barre de fraction jusqu'à la fin de la
// ligne est un commentaire qui est complètement ignoré par le compilateur.

#include <stdio.h>                  // Standard input/output

int main(void) {                    // La fonction main() ne prend aucun argument (void)
                                    // et renvoie un entier (int)
        printf("Hello, world\n"); // On affiche une chaîne de caractères
        return 0;                   // main() renvoie l'entier 0. Le programme s'arrête.
}
```

1. Lancez un éditeur, et recopiez le programme ci-dessus dans le fichier `hello.c`, puis exécuter la commande

```
gcc -Wall -W hello.c -o hello
```

Si tout se passe bien, le programme `hello` a été créé. Lancez le en tapant

```
./hello
```

Regardons le programme `hello.c` point par point :

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

Cette ligne dit au compilateur de charger l'entête `stdio.h` qui déclare l'existence des fonctions standard comme `printf` que l'on utilise dans le programme. En pratique, tout programme commence par cette ligne. Pour pouvoir utiliser certaines autres fonctions, on doit « inclure » d'autres fichiers. Par exemple, pour utiliser les fonctions mathématiques (`sin`, `log`, ...), on doit rajouter « #**include** <math.h> ». Pour utiliser le générateur de nombres aléatoires `rand`, on a besoin de « #**include** <stdlib.h> ».

2. Essayer d'enlever la ligne avec le #**include** et de recompiler pour voir ce qui se passe.

```
int main(void) {
        ...
}
```

On déclare et définit la fonction `main()`. Le code de cette fonction spéciale (ce qu'il y a entre la paire d'accolades) est la liste d'instructions qui seront exécutées, dans l'ordre, par le programme. La dernière de ces instructions est

```
        return 0;
```

qui signifie que `main()` doit s'arrêter en renvoyant la valeur (entière) 0. Le programme s'arrête et cette valeur de retour est fournie au système d'exploitation. (On peut l'obtenir en tapant « **echo** $? » juste après avoir exécuté le programme.) Traditionnellement, un code de retour valant 0 signifie que tout s'est bien passé.

```
        printf("Hello, world\n");
```

Cette instruction est un appel à la fonction `printf()` qui affiche la *chaîne de caractères* (écrite entre guillemets doubles « " ») à l'écran. Le « `\n` » indique le caractère spécial « retour à la ligne. » La fonction `printf()` sait faire plus que d'afficher des chaînes de caractères ; par exemple

```
        printf("Le résultat de %d + %d est %d\n", 2, 3, 2+3);
```

affiche « `Le résultat de 2 + 3 est 5` ». Les « `%d` » ont été remplacés par les entiers qui sont indiqués après la chaîne.

3. Essayez d'enlever le `\n`, de mettre plusieurs `\n` dans la chaîne de caractères à différents endroits et de mettre plusieurs appels à `printf`. Utilisez le compilateur C pour calculer 985×743.

Boucles et variables

```
#include <stdio.h>

int main(void) {
        int fact=1;     // fact est une variable de type entier initialisée à 1.
        int n;          // n est une variable de type entier de valeur indéterminée.
        n = 1;          // On donne maintenant la valeur 1 à la variable n.
        while (n <= 10) {       // « <= » signifie « est inférieur ou égal à »
                fact = fact * n;        // On peut aussi écrire fact *= n;
                printf("La factorielle de %d est %d\n", n, fact);
                n = n + 1;              // On peut aussi écrire n += 1; ou n++;
        }
        return 0;
}
```

Dans ce programme, on définit deux variables (`fact` et `n`) dans la fonction `main()`. Ces deux variables contiennent des entiers (`int`) dont la valeur initiale est 1. Cette valeur va changer au cours de l'exécution ; par exemple, si `fact` vaut 2 et `n` vaut 3, après l'instruction « fact = fact * n; », `fact` vaut 6. L'appel à `printf()` affiche les valeurs de `n` et `fact` à la place des `%d` dans la chaîne de caractères.

La construction

```
while (n <= 10) {
        ...
}
```

signifie qu'il faut exécuter en boucle ce qu'il y a entre les deux accolades *tant que* `n` est inférieur ou égal à 10. La condition est vérifiée au début de chaque boucle et le corps du `while` n'est exécuté que si cette condition est vraie. Au lieu d'écrire

```
n = 1;
while (n <= 10) {
        ...
        n = n + 1;
}
```

il est *complètement équivalent* d'écrire

```
for (n = 1; n <= 10; n = n + 1) {  // en fait, tout le monde écrirait
        ...                        // ici « n++ » plutôt que « n=n+1 »
}
```

4. Écrivez un programme qui calcule et affiche la somme des n premiers entiers pour n variant de 1 à 30. Le programme doit afficher des lignes qui ressemblent à « La somme des 4 premiers entiers est 10 ». Faites le programme une fois avec une boucle `for` et une fois avec une boucle `while`.

Tests

```
#include <stdio.h>

int main(void) {
        int n = 347;

        printf ("%d", n);
        while (n != 1) {                // « != » signifie « est différent de »
                if (n % 2 == 0) {       // « == » signifie « est égal à »
                        n = n / 2;      // n % 2 est le reste de la division de n par 2
                } else {                // On peut écrire if (...) {...}
                        n = 3 * n + 1;  // sans mettre la partie avec « else »
                }                       // si on n'en a pas besoin.
                printf(" -> %d",n);
        }                       // La conjecture de syracuse dit que quelle que
        printf("\n");           // soit la valeur de départ, la suite de nombres
                                // atteind 1. Si vous arrivez à le démontrer,
        return 0;               // vous devenez célèbre.
}
```

Selon que `n % 2` vaut 0 ou non, c'est le bloc { n = n /2 } ou le bloc { n = 3 * n + 1 } qui est exécuté. L'un ou l'autre, et jamais les deux à la fois. Attention à ne pas confondre « x=3 » (je donne la valeur 3 à la variable `x`) et « x==3 » (je teste si `x` est égal à 3).

Nombres à virgule

On peut aussi manipuler des nombres à virgules :

```
#include <stdio.h>

int main(void) {
        double x = 3.5;
        x = 3.5 / 7.8;
        printf("x vaut %g\n",x);        // affiche x avec 6 chiffres significatifs
        printf("x vaut %.10g\n",x);     // affiche x avec 10 chiffres significatifs
        return 0;
}
```

Attention, la précision d'un nombre à virgule n'est pas infinie ; la machine est obligée de faire des arrondis.

5. Calculez la valeur de $1 + x + \frac{x^2}{2}$ pour x valant (0,1 ; 0,2 ; 0,3 ; 0,4 ; 0,5 ; 0,6 ; 0,7 ; 0,8 ; 0,9).

Exercices

6. Écrivez un programme qui calcule et affiche les sommes des n premiers entiers pour n variant de 1 à 30. Le programme doit afficher des lignes qui ressemblent à « 1 + 2 + 3 + 4 = 10 ». Pour résoudre ce problème, il faudra faire *deux* boucles imbriquées l'une dans l'autre.

7. Écrivez un programme qui affiche tous les diviseurs d'un nombre (par exemple, 55 044).

8. Écrivez un programme qui vérifie si un nombre donné est premier ou non. Combien faut-il faire de tests au maximum ? (Réfléchissez bien...) Vous aurez peut-être besoin de l'instruction `break`, qui permet de sortir immédiatement d'une boucle.

9. Écrivez un programme qui affiche tous les nombres premiers entre 1 et 1 000.

10. Écrivez un programme qui affiche la décomposition en facteurs premiers d'un nombre donné.

11. Calculer des approximations de $\ln(1+x)$ pour x petit en utilisant le développement limité de cette fonction jusqu'à l'ordre x^n, pour n valant (5 ; 10 ; 15 ; 20) et x valant (0,1 ; 0,2 ; 0,3 ; 0,4 ; 0,5 ; 0,6 ; 0,7 ; 0,8 ; 0,9). Faites la même chose pour la fonction e^x.

12. On veut calculer $\sqrt{x}$ pour un nombre x donné. Supposons que r soit une première approximation de $\sqrt{x}$. Les nombres r et x/r encadrent $\sqrt{x}$, c'est-à-dire que si l'un de ces deux nombres est plus petit que $\sqrt{x}$, alors l'autre est plus grand. Le nombre $(r + x/r)/2$, à mi-chemin entre les deux bornes de cet encadrement, est donc une meilleure approximation de $\sqrt{x}$ que ne l'était r. En partant de $r = 1$ et en raffinant plusieurs fois cette approximation de $\sqrt{x}$, calculez avec une bonne précision $\sqrt{2}$ et $\sqrt{55044}$.