T.P. numéro 2

Types de variable et instructions de controle

1 Introduction

Dans ce TP seront introduites les notions nécessaires à une *première approche de la structuration d'un programme* c'est-à-dire les notions de découpage en blocs, d'instructions de contrôle et de déclaration de variables.

Le terme « *langage structuré* », qui peut paraître nouveau aux gens venant du BASIC ou d'autres langages non structurés, mérite une explication. Les langages non structurés attaquent leurs programmes par le haut et se fraient un chemin, en ligne droite jusqu'au bas du programme, en exécutant une par une chaque instruction. Néanmoins, beaucoup de problèmes sont trop complexes pour être traités par une simple série d'instructions allant de haut en bas et de bas en haut. Le langage C, langage structuré, a été conçu pour pouvoir traiter séparément les tâches d'un programme en les plaçant dans des blocs (délimités par des paires d'accolades `{}`). Le programme possède alors un bloc principal qui dirige le passage de bloc à bloc.

Les variables utilisées par le programme doivent être déclarées avec leur type et nous verrons dans les parties 2 et 3 les *types de données* que possède le langage C.

Normalement, les instructions s'effectuent séquentiellement, c'est à dire l'une après l'autre. Pour accéder à une autre instruction que la suivante, on a trois solutions : le *branchement inconditionnel*, le *branchement conditionnel* et la *boucle*. Ce sont les *instructions de contrôle*, qui seront présentées dans la partie 3.

Une fois ces notions présentées et mises en pratique dans les exercices 1 à 8, nous les appliqueront au *découpage d'un problème (simple) en vue de son codage* dans la partie 4).

2 Types de variables scalaires en C

Nous allons définir les différents types de variables existantes en C. On verra ici uniquement les types scalaires (entiers, ...); les types agrégés (combinaisons de scalaires, tableaux par exemple) seront présentés et manipulés en détail au TP numéro 5.

On appelle variable scalaire une variable ne contenant qu'une seule valeur, sur laquelle on pourra faire un calcul arithmétique. On possède trois types de base (**char**, **int**, **float**) que l'on peut modifier par 3 spécificateurs (**short**, **long**, **unsigned**).

2.1 char : caractère (8 bits)

Une constante caractère est désignée entre apostrophes (simples quotes). 'a' correspond à un octet On peut définir certains caractères spéciaux, par le préfixe \ (antislash) :

- \n nouvelle ligne
- \b backspace
- \r retour chariot (même ligne)
- \' apostrophe
- \" double quote
- \0 nul
- \<nombre> nombre en octal d'un chiffre).
- \x<nombre> nombre en hexadécimal

Les **char** sont considérés dans les calculs comme des **int**. Par défaut en C un **char** est signé donc peut varier de -128 à +127. Pour utiliser les caractères spéciaux du PC (non standard), il vaut mieux utiliser des **unsigned char** ($2^8 = 256$ soit de 0 à 255).

2.2 int : entier

Si l'on désire une taille précise, on utilise **short int** (16 bits) ou **long int** (32 bits). Sans précision, **int** donnera les programmes les plus rapides pour une machine donnée. Par défaut, les **int** sont signés, mais on peut préciser **unsigned int**.

Désormais certains compilateurs considèrent **short** comme 16 bits, **int** comme 32 bits et **long** comme 64 bits.

2.3 float : réel

Un flottant est un nombre stocké en trois parties : un bit de signe **S**, un exposant biaisé **Eb** et une mantisse **M**. La taille de la mantisse définit le nombre de chiffres significatifs, alors que la taille de l'exposant définit le plus grand nombre acceptable par la machine. Les opérations sur les réels sont plus lentes que sur les entiers. Pour une addition par exemple, il faut d'abord décaler la mantisse pour égaliser les exposants puis faire l'addition. Les réels sont toujours signés. On peut par contre utiliser le spécificateur **long** pour des réels avec une précision accrue. On peut également utiliser le nom **double** au lieu de **long float**. Certains compilateurs acceptent même des **long double** (quadruple précision).

En représentation IEEE754, le signe **S** est codé sur 1 bit (0=+ ; 1=-) et exposant **E** et mantisse **M** sont codés sur les 31 ou 63 bits restants pour simple ou double precision.

Exemple :

Ce nombre réel, dont la représentation binaire IEEE754 float (32 bits) est :

1 10000101 101110110110000000000000

doit être interprété de la manière suivante :

Le bit de signe **S** (à gauche) vaut **1** : c'est un réel négatif

On a **Eb=10000101=133** qui est la valeur de l'exposant biaisé. On a **E=133-127=6** qui est la valeur de l'exposant vrai. Il faudra donc décaler la mantisse de 6 colonnes vers la gauche.

La lecture de la mantisse donne (avec le bit sous-entendu égal à 1 en tête, puisqu'il s'agit d'un nombre normalisé) :

M=1,101010110100000000000000

Le nombre est donc égal à (en base 10) :

$$-1,1010101101 \times 2^6 = -1101010,1101 = -(2 + 8 + 32 + 64 + 0.5 + 0.25 + 0.0625) = -106,8125$$

Exercice 2.1 Tailles et plages de valeurs

Complétez le tableau suivant :

type	taille (en bits)	plage de valeurs
char	8	 à
unsigned char	8	 à
short (shortint)	16	 à
unsigned short	16	 à
long (longint)	32	 à
unsigned long	32	 à
float	32	-3.4e38 à 3.4e38(7 chiffres significatifs)
double (long float)	64	-1.7e308 à 1.7e308 (15 chiffres significatifs)

3 Les instructions de contrôle en C

Normalement, les instructions s'effectuent séquentiellement, c'est à dire l'une après l'autre. Pour accéder à une autre instruction que la suivante, on a trois solutions : le branchement incondi-tionnel, le branchement conditionnel et la boucle.

3.1 Boucles

Une boucle permet de répéter plusieurs fois un bloc d'instructions.

3.1.1 While (tant que)

structure :

```
while (expression) {instructions}
```

Tant que l'expression est vraie ($\neq 0$), on effectue le bloc d'instructions entre accolades. L'expression est au moins évaluée une fois. Tant qu'elle est vraie, on effectue l'instruction, dès qu'elle est fausse, on passe à l'instruction suivante (si elle est fausse dès le début, l'instruction n'est jamais effectuée).

Exemple :

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

```
int main (void)
{
    float nombre, racine=0;
    printf("Entrez un nombre reel entre 0 et 10");
    scanf("%f", \&nombre);
    while (racine*racine<nombre)
    {
        racine+=0.01
    }
    printf("La racine de %f vaut %4.2f à 1%% pres.\n", nombre, racine);
}
```

Exercice 2.2 *Puissances de 2*

Faire un programme qui affiche toutes les puissances de 2, jusqu'à une valeur maximale donnée par l'utilisateur. On calculera la puissance par multiplications successives par 2.

3.1.2 Do While (faire tant que)

Structure :

```
do {instructions} while (expression);
```

Comme **while**, mais le bloc d'instructions est au moins fait une fois, avant la première évaluation de l'expression. Attention au ; final!

Exemple :

```
#include <stdio.h>

int main(void)
{
    int a;
    do
    {
        printf("entrez le nombre 482");
        scanf("%d",&a);
    }
    while (a!=482);
    printf("c'est gentil de m'avoir obei");
}
```

Exercice 2.3 *Devinette*

Ecrivez un programme de jeu demandant de deviner un nombre entre 0 et 9 choisi par l'ordinateur. On ne donnera pas d'indications avant la découverte de la solution, où l'on indiquera le nombre d'essais. La solution sera choisie par l'ordinateur par la fonction rand(). Utilisez le man pour connaître l'utilisation de cette fonction qui rend un entier aléatoire (déclarée dans stdlib.h).

3.1.3 For (pour)

Structure :

```
for (initialisation ;condition;incrementation) {instructions}
```

Cette boucle est surtout utilisée lorsque l'on connaît à l'avance le nombre d'itération à effectuer. L'expression initialisation est effectuée une fois, en premier, puis on teste la condition. On effectue le bloc d'instruction puis l'incrémentation tant que la condition est vraie. Le bloc d'instructions et l'incrémentation peuvent ne jamais être effectués.

Une ou plusieurs des trois expressions peuvent être omises, l'instruction peut être vide. for(;;) est donc une boucle infinie.

Exemple :

```
#include <stdio.h>

int main(void)
{
    int i, j;
    printf("Quelle table de multiplication afficher (1-10) ?");
    scanf("%d",&i);
    do (j=1;j<=10;j=j+1)
    {
        printf("%d fois %d font %d",j,i,j*i)';
    }
}
```

Exercice 2.4 *Moyenne*

Faire un programme qui calcule la moyenne de N notes. N et les notes seront saisies par scanf. Le calcul de la moyenne s'effectue en initialisant une variable à 0, puis en y ajoutant progressivement les notes saisies puis en divisant par N.

3.2 Branchements conditionnels

On a souvent besoin de n'effectuer certaines instructions que dans certains cas. On dispose pour cela du **if** et du **switch**.

3.2.1 If - Else (Si - Sinon)

Structure :

```
    if (expression) {instructions1}
```

ou

```
    if (expression) {instructions1} else {instructions2}
```

Si l'expression est vraie ($\neq 0$) on effectue le bloc d'instructions1, puis on passe à la suite. Sinon, on effectue le bloc d'instructions2 puis on passe à la suite (dans le cas sans else on passe directement à la suite).

Exercice 2.5 *Devinette (bis)*

Reprendre le jeu de l'exercice 2.3 en précisant au joueur à chaque essai si sa proposition est trop grande ou trop petite.

L'instruction d'un **if** peut être un autre **if** (imbriqué)

Exemple :

```
if (c1)
  {i1}
else if (c2)
  {i2}
  else if (c3)
    {i3}
    else
      {i4}
{i5}
```

Si c1 alors i1 puis i5, sinon mais si c2 alors i2 puis i5, ... Si ni c1 ni c2 ni c3 alors i4 puis i5.

Le **else** étant facultatif, il peut y avoir une ambiguïté s'il y a moins de **else** que de **if**. En fait, un **else** se rapporte toujours au **if** non terminé (c'est à dire à qui on n'a pas encore attribué de **else**) le plus proche. On peut aussi terminer un **if** sans **else** en l'entourant d'accolades.

Exemple :

```
if (c1)
  if (c2)
    {i1}
  else
    {i2}
```

Si c1 et c2 alors i1, si c1 et pas c2 alors i2, si pas c1 alors (quel que soit c2) rien.

```
if (c1)
  {
    if (c2)
      {i1}
  }
else
  {i2}
```

Si c1 et c2 alors i1, si c1 et pas c2 alors rien, si pas c1 alors i2.

3.2.2 Switch - Case (brancher - dans le cas)

Structure :

```
switch (expression_entiere)
{
    case valeur1: {instructions1}
    ...
    case valeurN: {instructionsN}
    default: {instructions_par_defaut}
}
```

L'expression ne peut être qu'entière (**char**, **int**, **long**). L'expression est évaluée, puis on passe directement au « **case** » correspondant à la valeur trouvée. Le cas default est facultatif, mais si il est prévu il doit être le dernier cas.

Exemple : fonction vérifiant si son argument c est une voyelle.

```
int voyelle(char c)
{
    switch (c)
    {
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
        case 'y': return (1);    /* 1 = vrai */
        default: return (0);    /* 0 = faux */
    }
}
```

Exercice 2.6 *Calculatrice*

Faire un programme simulant une calculatrice 4 opérations. On demandera à l'utilisateur de donner deux chiffres et un opérateur.

3.3 Branchements inconditionnels

Quand on arrive sur une telle instruction, on se branche obligatoirement sur une autre partie du programme. Ces instructions sont à éviter si possible, car elles rendent le programme plus complexe à maintenir, le fait d'être dans une ligne de programme ne suffisant plus pour connaître immédiatement quelle instruction on a fait auparavant, et donc ne permet plus d'assurer que ce qui est au dessus est correctement terminé. Il ne faut les utiliser que dans certains cas simples.

3.3.1 Break (interrompre)

Il provoque la sortie immédiate de la boucle ou du **switch** en cours. Il est limité à un seul niveau d'imbrication.

Exemple :

```
do
{
  if (i==0) break;
  instruction;
  ...
}
while (i!=0);
```

Un meilleur codage de ce programme serait

```
while (i!=0)
{
  instruction;
  ...
}
```

3.3.2 Continue (continuer)

Cette instruction provoque le passage à la prochaine itération d'une boucle. Dans le cas d'un **while** ou **do...while**, saut vers l'évaluation du test de sortie de boucle. Dans le cas d'un **for** on passe à l'expression d'incrémentatation puis seulement au test de bouclage. En cas de boucles imbriquées, permet uniquement de continuer la boucle la plus interne.

Exemple :

```
for (i=0; i<10; i=i+1)
{
  if (i==j) continue;
  instruction;
  ...
}
```

Un meilleur codage serait

```
for (i=0; i<10; i=i+1)
  if (i!=j)
  {
    instruction;
    ...
  }
```

3.3.3 Return (retourner)

structure :

```
return;
```

ou

```
return(valeur);
```

Permet de sortir de la fonction en cours (y compris **main**), en se branchant à sa dernière accolade fermante. **return** permet également (et surtout) de rendre la valeur résultat de la fonction.

Exemple :

```
int max(int a, int b)
{
    if (a>b) return a;
    return b;
}
```

3.3.4 Exit (sortir)

structure :

exit;

ou

exit(valeur);

Ceci n'est pas un mot clef du C mais une fonction disponible dans la plupart des compilateurs (définie par ANSI, dans STDLIB.H). Elle permet de quitter directement le programme (même depuis une fonction). On peut lui donner comme argument le code de sortie (celui que l'on aurait donné à **return** dans **main**). Cette fonction libère la mémoire utilisée par le programme (variables + allocations) et ferme (sur beaucoup de compilateurs) les fichiers ouverts.

4 Application : Suite de Fibonacci, calcul du nombre d'or

.

Pourquoi les boutons d'or ont-ils 5 pétales ?

Pourquoi les ananas ont-ils 8 diagonales dans un sens et 13 dans l'autre ?

Pourquoi les marguerites ont-elles généralement 34, 55 ou 89 pétales ?

.

Tous ces nombres font partie de la suite de Fibonacci (1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,...), suite de nombres qui conduit au nombre d'or, et où chacun est la somme des deux précédents. Ce n'est pas le fruit du hasard mais le résultat d'un processus physique de croissance des plantes et des fruits. Les nombres de pétales de beaucoup de fleurs sont en moyenne 2 nombres consécutifs de cette suite (ou leur double), le nombre des spirales de graines des fruits (pomme de pin, ananas ...) sont exactement 2 de ces nombres.

Le long de la tige de certaines plantes, les feuilles poussent en décrivant une courbe qui tourne en hélice du bas vers le haut. Si, à partir de n'importe quelle feuille, on compte le nombre n de feuilles entre deux feuilles disposées l'une sous l'autre et le nombre p de tours compris entre ces deux feuilles, on obtient deux nombres de Fibonacci.

Autre propriété étrange de cette suite : le quotient de deux nombres consécutifs est une valeur approchée du nombre d'or $(1 + \sqrt{5})/2$ que l'on retrouve un peu partout dans la nature.

La spirale de la coquille du nautilus est une construction géométrique basée sur ce nombre.

Même Stradivarius utilisa ce nombre pour construire ses fameux violons !

Dans un rectangle d'or, le rapport longueur/largeur est égal au nombre d'or.

Le nombre d'or remonte très loin dans l'Histoire. On le trouve déjà dans les faces triangulaires de la pyramide de Khéops, 2800 ans avant J.-C.. On le trouve aussi dans beaucoup de monuments comme la cathédrale de Chartres.

La suite de Fibonacci :

Si l'on note (u_n) la suite de Fibonacci, elle est définie par : $u_1 = 1$; $u_2 = 1$ et $u_n = u_{n-1} + u_{n-2}$ pour tout $n > 2$.

La suite a entre autre les deux propriétés suivantes :

1) u_n et u_{n+1} sont premiers entre eux

2) $\lim (u_n/u_{n+1}) = (\sqrt{5} - 1) / 2$ (nombre d'or)

Exercice 2.7 *Suite de Fibonacci*

Ecrire un programme calculant les 20 premiers termes de la suite de Fibonacci $\{u_n\}$ et les rapports u_n/u_{n-1} . Verifier que ce rapport tend vers le nombre d'or. Le programme sera *structuré* en trois blocs : un bloc « Fibonacci » calculant les termes de la suite, un bloc « Remplir Tableau » servant à mettre les termes de la suite dans un tableau et un bloc « Ecrire Tableau » servant à afficher les résultats demandés.