[Math-L312] TP 2 : Structures de contrôle en C

Adrien Semin
adrien.semin@math.u-psud.fr

1 Syntaxe des structures de contrôle

1.1 Opérateurs logiques

Les différentes relations logiques sont : Il est possible de combiner plusieurs relations logiques avec les opérateurs

`a == b`	est vrai si a est égal à b	`a != b`	est vrai si a est différent de b
`a<b`	est vrai si a est strictement inférieur à b	`a <= b`	est vrai si a est inférieur ou égal à b
`a > b`	est vrai si a est strictement supérieur à b	`a >= b`	est vrai si a est supérieur ou égal à b

Table 1 – Tableau des différentes relations logiques

logiques suivants

`(Expression1) && (Expression2)`	est vrai si `Expression1` **et** `Expression2` le sont
`(Expression1) || (Expression2)`	est vrai si `Expression1` **ou** `Expression2` le sont
`!(Expression)`	est vrai si `Expression` est faux

Table 2 – Tableau des différents opérateurs logiques

1.2 Structures conditionnelles

La structure `if` permet d'effectuer une instruction si une condition est réalisée : on parlera de test. Elle peut prendre une clause `else` facultative qui sera exécutée si la condition n'est pas réalisée. La condition doit être une expression de type entier (ou convertible en entier), lorsque sa valeur est nulle la condition sera considérée comme non réalisée et toute valeur non nulle sera considérée comme une condition réalisée.

```
if (Expression) Instruction; /* Instruction sera exécutée si Expression ≠ 0
    */
if (Expression) Instruction1 else Instruction2; /* Instruction1 sera
    exécutée si Expression ≠ 0, sinon on exécute Instruction2 */
```

La structure `switch` permet de réaliser simplement une série de tests imbriqués. Si $Expression = Valeur1$ alors le programme exécutera `Instruction1`, si $Expression = Valeur2$ alors il exécutera `Instruction2`, etc... Si $Expression$ n'est égale à aucune des valeurs de la liste des `case` alors le programme exécutera `InstructionDefault` (si toutefois la clause `default` facultative a été spécifiée, sinon il passe directement à la suite).

```
switch (Expression){
  case Valeur1 : Instruction1;
                 break; /* L'instruction break est obligatoire ici sinon le
                     programme exécutera aussi Instruction2 */
  case Valeur2 : Instruction2;
                 break;
  case Valeur3 : Instruction3;
                 break;
  /* ... */
  default : InstructionDefault;  /* Cette ligne est facultative */
}
```

1.3 Boucles

La boucle `for` est une boucle multi-usages. L'instruction `Initialisation` est d'abord exécutée, puis :

(a) si l'expression *Condition* est nulle alors le programme quitte la boucle, sinon on passe à l'étape (b) ;

(b) l'instruction `Instruction` est exécutée ;

(c) l'instruction `Transition` est exécutée ;

(d) retour à l'étape (a).

```
for (Initialisation;Condition;Transition){
  Instruction;
}
```

Exemple pour afficher les entiers de 0 à 99 (les accolades sont ici inutiles puisqu'il n'y a qu'une seule instruction dans le corps de la boucle) :

```
for (i=0;i<100;i++)
  printf("%i\n",i);
```

La boucle `while` effectue une instruction tant qu'une condition est vérifiée. Elle admet aussi une variante avec `do`, qui elle effectue l'instruction au moins une fois même si la condition est fausse dès le début.

```
while (Condition){
  Instruction; /* Instruction exécutée tant que Condition ≠ 0 */
}

do { /* Variante */
  Instruction; /* Instruction exécutée tant que Condition ≠ 0, mais le test a
       lieu à la fin de la boucle donc Instruction sera toujours exécutée au
       moins une fois */
} while (Condition);
```

Toute boucle peut être interrompue avec l'instruction `break` (auquel cas le programme sort de la boucle et exécute le code qui suit), et on peut forcer le retour au début de la boucle grâce à l'instruction `continue`. Toutefois on préférera, dans la mesure du possible, d'éviter de les utiliser car elles ont tendance à "déstructurer" le programme et à le rendre moins lisible (traduction : un programme utilisant cette instruction sera considéré comme "faux").

2 Exercices

1. `carre.c` ***À Rendre!*** Ecrire un programme qui demande deux entiers m et n strictement positifs, et qui affiche un carré $m \times n$ comme suit (exemple avec $m = 5$ et $n = 8$) :

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

2. `triangle.c` ***À Rendre!*** Ecrire un programme qui demande un entier n strictement positif, et qui affiche un triangle inférieur $n \times (2n)$ comme suit (exemple avec $n = 5$) :

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

3. `trace_m.c` ***À Rendre!*** Ecrire un programme qui demande un entier n stricement positif, et qui affiche un "M" sur n lignes comme suit (exemple avec $n = 5$) :

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

4. `isprime.c` **∗ *À Rendre!* ∗** Ecrire un programme qui demande un nombre entier strictement positif p et qui affiche si p est premier (on vérifiera que p n'est divisible par aucun k, avec k entre 2 et $\sqrt{p}$). On optimisera le programme pour faire le moins de calculs possibles (si on trouve l tel que $k \bmod l = 0$, alors on ne testera pas pour les entiers entre $l+1$ et $\sqrt{p}$).

5. `fibonacci.c` **∗ *À Rendre!* ∗** Ecrire un programme qui demande un entier n positif et qui calcule le $n^{\text{ème}}$ terme de la suite de Fibonacci, défini par

$$u_0 = 1, \quad u_1 = 1, \quad u_{n+2} = u_{n+1} + u_n \tag{1}$$