Travaux dirigés 3 : expressions booléennes et structures de controle *if else*, *for* et *while*.

1 Révisions rapides

1.1 Affectation

Soit le programme suivant :

```
1  /* Declaration de fonctionnalites supplementaires */
2  #include <stdlib.h> /* EXIT_SUCCESS */
3  #include <stdio.h> /* printf */
4
5  /* Declaration des constantes et types utilisateurs */
6
7  /* Declaration des fonctions utilisateurs */
8
9  /* Fonction principale */
10 int main()
11 {
12     /* Declaration et initialisation des variables */
13     int x;
14
15     x = 3;
16     x = x + 1;
17     printf("x = %d\n", x);
18
19     /* valeur fonction */
20     return EXIT_SUCCESS;
21 }
22
23 /* Definitions des fonctions utilisateurs */
```

1. Que fait ce programme (répondre sans utiliser l'ordinateur) ?

Correction.

- *Il déclare la variable impérative* `x` *(occupe un espace memoire)*
- *Il affecte 3 à la variable* `x`
- *Affecte à x la valeur de l'expression (x + 1) qui vaut 4 : la sous-expression x s'évalue comme la valeur de la variable (3), plus un.*
- *Affiche la valeur de l'expression x qui s'évalue comme la valeur de la variable, cad 4.*

2. Donner la traduction des instructions aux lignes 15 et 16 en langage amil.

Correction.

CP vaut 1
 # Soit la var x correspondant à la case mémoire d'adresse 10 (initialisation).
 # affecte 3 a la var x
1 valeur 3 r0
2 ecriture r0 10
 # x = x + 1
3 lecture 10 r0

4 valeur 1 r1
5 add r1 r0 # r0 vaut l'expression x + 1
6 ecriture r0 10 # affecte la valeur de l'expression à x

1.2 Évaluation d'expressions booléennes

Soit le programme suivant :

```
1  #include <stdlib.h> /* EXIT_SUCCESS */
2  #include <stdio.h> /* printf */
3
4  #define FALSE 0
5  #define TRUE 1
6
7  /* Declaration de fonctions utilisateurs */
8
9  int main()
10 {
11     int beau_temps = TRUE;
12     int pas_de_vent = FALSE;
13
14     printf("%d\n", beau_temps && pas_de_vent);
15     printf("%d\n", beau_temps || pas_de_vent);
16     printf("%d\n", !(beau_temps) || pas_de_vent);
17     printf("%d\n",!(!(beau_temps) || pas_de_vent) == (beau_temps && !(pas_de_vent)));
18
19     return EXIT_SUCCESS;
20 }
21
22 /* Definition de fonctions utilisateurs */
```

Qu'affiche le programme (répondre sans utiliser l'ordinateur) ?

Correction.

```
0
1
0
1 /* toujours vrai : théorème de De Morgan : NON (a OU b) = NON a ET NON b */
```

---

2 Boucles *for* ou *while* ?

Ces boucles ont exactement la même sémantique et on peut facilement réécrire l'une en l'autre. Par convention, on préfère utiliser la boucle *for* lorsque l'on connaît le nombre d'itérations à l'avance ; on utilise *while* dans le cas contraire, lorsque le nombre d'itérations n'est pas connu à l'avance.

Résoudre les problèmes suivants en utilisant soit *for*, soit *while*.

2.1 Élévation à la puissance

Écrire un programme qui demande à l'utilisateur d'entrer deux nombres entiers x et $n \geq 0$ puis calcule x^n et affiche le résultat.

Correction. *Trivial (for).*

2.2 Somme d'une série d'entiers positifs saisis par l'utilisateur

Écrire un programme qui lit des entiers saisis par l'utilisateur et dès que l'utilisateur saisit un zéro (ou un nombre négatif), affiche le total des entiers positifs saisis jusque là.

Exemple d'exécution :

```
Entier ? 13
Entier ? 2
Entier ? 12
Entier ? 15
Entier ? 0
Total : 42
```

2.3 Test de primalité

Donner **seulement l'algorithme** d'un programme qui demande à l'utilisateur d'entrer un nombre entier positif n et teste si n est premier. Utiliseriez vous un *for* ou un *while* ? Exemples d'exécution :

```
Entrer un entier positif : 23
L'entier 23 est premier

Entrer un entier positif : 25
L'entier 25 n'est pas premier car divisible par 5
```

Correction. _ *L'idée est qu'il faut le faire avec un while plutôt qu'un for. L'algorithme doit s'énoncer à peu près comme ceci :*

- *saisie de n*
- *Pour chaque entier d entre 1 et n exclus :*
  -
  - *Si d divise n, terminer la boucle et afficher n n'est pas premier*
- *Si la boucle précédente s'exécute juste qu'au bout sans avoir rencontré de diviseur de n, afficher que l'entier n est premier.*

La traduction d'une boucle de parcours pouvant avoir une fin prématurée se fera plutôt avec un `while`*. On ne présente pas le* `break` *dans ce cours.*

```
#define TRUE 1
#define FALSE 0
...
int main()
{
  int premier = TRUE;
  int n;
  int d = 2;

  /* saisie */
  printf("n?");
  scanf("%d",&n);

  /* test de primalite */
  while (premier && (d < n))
  {
     if (n % d == 0)
     {
        premier = FALSE;
     }
     d = d + 1;
  }
```

```
  /* affichage */
  if (premier)
  {
    printf("%d est premier\n", n);
  }
  else
  {
    printf("%d n'est pas premier, car divisible par %d\n", n, d - 1);
  }


  return EXIT_SUCCESS;
}
```

---

3 Le nombre secret

Nous voulons programmer le jeu du nombre à découvrir. Le joueur doit deviner un nombre secret choisit par l'ordinateur entre 0 et `NB_MAX` (une constante du programme). S'il propose un nombre trop grand, l'ordinateur lui répond "Plus petit", s'il propose trop petit, l'ordinateur lui répond "Plus grand". Dans ces deux cas, il est invité à proposer un autre nombre. Le jeu s'arrête quand il devine juste. Un exemple d'exécution de ce jeu pourrait être :

```
Votre choix ?
8
Plus petit.
Votre choix ?
4
Plus petit.
Votre choix ?
2
Vous avez trouvé le nombre secret.
```

1. Proposez un algorithme en français pour le jeu.
2. Traduisez-le en langage C et exécutez-le.
3. Pourquoi préférez-vous une boucle `while` ici ?

Pour rendre le jeu intéressant, l'ordinateur doit choisir le nombre secret *au hasard*. La librairie C standard propose des fonctions renvoyant des nombres pseudo-aléatoires [1] déclarées dans `<stdlib.h>`. L'ordinateur va utiliser la fonction : `int rand();` qui renvoie un nombre pseudo-aléatoire entier entre 0 et la constante `RAND_MAX` (égale à 2147483647) inclus. Pour renvoyer un nombre pseudo-aléatoire entre 0 et `NB_MAX`, `NB_MAX` inclus (`NB_MAX` $<<$ `RAND_MAX`), il suffit de calculer le reste de la division entière de `rand()` par `(NB_MAX + 1)`, c'est-à-dire le nombre renvoyé par `rand()` modulo `(NB_MAX + 1)` (opérateur % en C). Le nom `rand` vient de *random* qui veut dire aléatoire en anglais.

Un exemple de programme illustrant l'utilisation de `rand` pour engendrer un nombre pseudo-aléatoire est le suivant :

```
1  #include <stdlib.h> /* EXIT_SUCCESS, rand, srand */
2  #include <stdio.h> /* printf */
3  #include <time.h> /* time */
4
5  #define NB_MAX 15 /* nombre secret entre 0 et NB_MAX inclus */
6
7  int main()
8  {
```

1. http ://fr.wikipedia.org/wiki/Générateur_de_nombres_pseudo-aléatoires

```
 9       int nombre_secret; /* nombre secret à deviner */
10
11       /* initialisation du générateur de nombres pseudo-aléatoires */
12       srand(time(NULL)); /* à ne faire qu'une fois */
13
14       /* tirage du nombre secret */
15       nombre_secret = rand() % (NB_MAX + 1); /* entre 0 et NB_MAX inclus */
16
17       /* exploitation du secret */
18       printf("Tu ne devineras jamais que mon secret est %d\n", nombre_secret);
19
20       return EXIT_SUCCESS;
21   }
```

Correction.

```
#include <stdlib.h> /* EXIT_SUCCESS, rand, srand */
#include <stdio.h> /* printf, scanf */
#include <time.h> /* time */

#define TRUE 1
#define FALSE 0

#define NB_MAX 15 /* nombre secret entre 0 et NB_MAX inclus */

/* declaration de fonctions utilisateurs */

int main()
{
    int nombre_secret; /* nombre secret à deviner */
    int choix; /* choix de l'utilisateur pour le nombre secret */
    int trouve = FALSE; /* TRUE si trouvé nombre secret */

    /* initialisation du générateur de nombres pseudo-aléatoires */
    srand(time(NULL)); /* à ne faire qu'une fois */

    /* tirage du nombre secret */
    nombre_secret = rand() % (NB_MAX + 1); /* entre 0 et NB_MAX inclus */

    /* manche joueur */
    while(!trouve) /* pas trouvé nombre secret */
    {
        /* demande nombre à l'utilisateur */
        printf("Votre choix (nombre entre 0 et %d) ?\n",NB_MAX);
        scanf("%d",&choix);

        if(choix == nombre_secret) /* trouvé */
        {
            trouve = TRUE;
        }
        else /* pas trouvé */
        {
            /* donne indice */
            if(choix > nombre_secret)
            {
                printf("Plus petit.\n");
            }
            else
```

```
            {
                printf("Plus grand.\n");
            }
        }
    }
    /* trouvé nombre secret */

    printf("Vous avez trouvé le nombre secret.\n");

    return EXIT_SUCCESS;
}

/* Definition de fonctions utilisateurs */
```

---

4 Exercices complémentaires

4.1 Faut-il renoncer à rembourser la dette publique ?

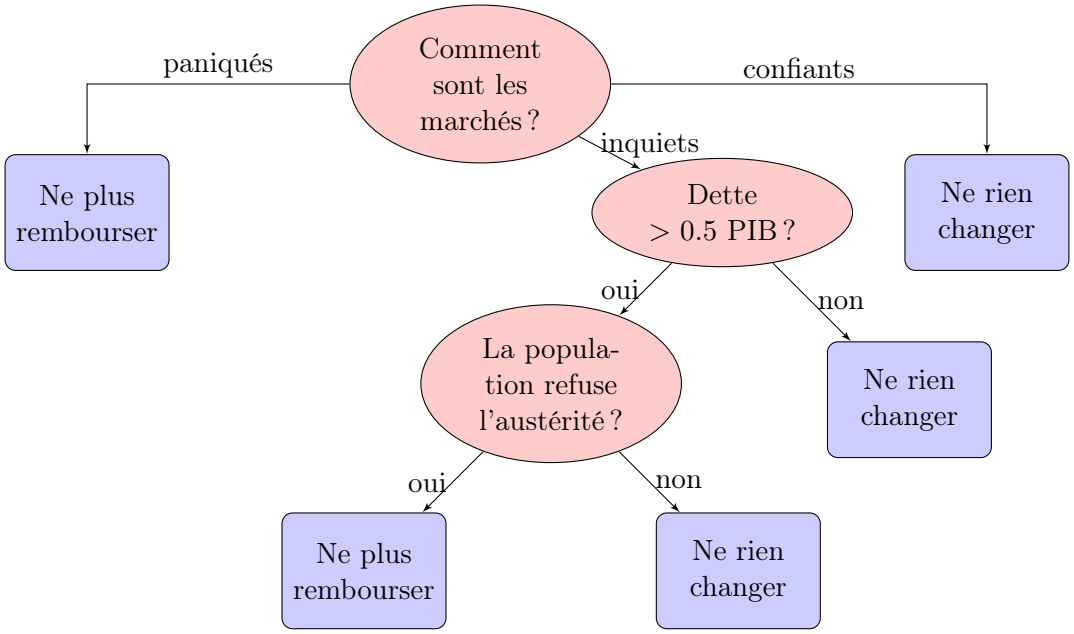

FIGURE 1 – Décider s'il faut continuer de rembourser la dette

Un arbre de décision est un graphe particulier où les nœuds sont des questions et les arêtes sont les réponses à ces questions. Il se lit de haut en bas. On avance dans l'arbre en répondant aux questions. Les nœuds les plus bas jouent le rôle particulier de classes de réponse au problème initial.

Ici, il y a deux classes de réponse : « Ne plus rembourser » et « Ne rien changer ». Par exemple, si les marchés sont inquiets, si la dette est strictement supérieure à 0.5 fois le PIB et que la population refuse l'austérité, alors on ne rembourse plus.

Vous utiliserez quatre variables dont vous coderez les valeurs avec des constantes symboliques. Les trois premières représentent les propriétés du jour courant pour prendre la décision, la dernière représente la décision à prendre :

- `marches` est un entier représentant l'état des marchés, elle peut valoir un entier signifiant PANIQUE, ou un entier signifiant INQUIETUDE ou un entier signifiant CONFIANCE.
- `dette` est un nombre représentant la dette en pourcentage du PIB (une valeur de 50 représente 50% du PIB).

– `refus` est un entier représentant le refus de l'austérité qui peut avoir la valeur usuelle TRUE (si la population refuse l'austérité) ou la valeur usuelle FALSE (sinon).
– `rembourser` est un entier représentant la décision à prendre et peut avoir la valeur TRUE (s'il faut continuer de rembourser) ou FALSE (s'il faut arrêter de rembourser). Sa valeur initiale est TRUE.

Votre programme sera en deux parties. Une première partie concernera la prise de décision sans affichage, en fonction des valeurs de `marches`, `dette`, `refus`. Vous donnerez des valeurs de votre choix à ces trois variables mais le programme doit fonctionner pour tous les autres choix possibles. À la fin de cette première partie, `rembourser` contiendra la valeur correcte pour la décision prise (TRUE ou FALSE). La seconde partie exploitera la valeur de `rembourser` pour effectuer l'affichage.

Écrire le programme complet en distinguant bien les deux parties.

Correction. *Correction figure 2.*

4.2 Carré d'étoiles

Écrire un programme qui demande à l'utilisateur d'entrer un nombre entier positif n et affiche un carré creux d'étoiles de côté n. Exemple (l'utilisateur entre 4) :

```
n? 4
****
*  *
*  *
****
```

Correction.

```
 double for et un if comme ça :
if ( (i == 0) || (j == 0) || (i == n - 1) || (j == n - 1) )
{
  printf("*");
}
else
{
  printf(" ");
}
```

```c
/* Declaration de fonctionnalites supplementaires */
#include <stdlib.h> /* pour EXIT_SUCCESS */
#include <stdio.h> /* pour printf() */

/* Declaration des constantes et types utilisateur */
#define TRUE 1
#define FALSE 0
#define PANIQUE 2
#define INQUIETUDE 1
#define CONFIANCE 0

/* Fonction principale */
int main()
{
    /* Declaration et initialisation des variables */
    int marches = PANIQUE;
    int dette = 50;
    int refus = TRUE;
    int rembourser = TRUE;

    if (marches == PANIQUE) /* 1) accroche toi a l'euribor je retire le zinzin */
    {
        rembourser = FALSE;
    }
    if (marches == INQUIETUDE) /* 2) c'est grave docteur ? */
    {
        if (dette > 50)
        {
            if (refus)
            {
                rembourser = FALSE;
            }
        }
    }

    if (rembourser)
    {
        printf("Ne rien changer\n");
    }
    else
    {
        printf("Ne plus rembourser\n");
    }

    /* Valeur fonction */
    return EXIT_SUCCESS;
}

/* Definition des fonctions utilisateur */
```

FIGURE 2 – Faut-il annuler la dette ? – Corrigé.