Compte-rendu du projet C

« It's time to C Code (2) ! »

Membres du groupe :

- Anto BENEDETTI ;
- Antony DAVID ;
- Anthony JABRE.

Sommaire

Durant ce projet de programmation en langage C, nous avons cherché, pour chaque exercice :

- À avoir le code le plus lisible possible ;
- À ajouter des détails rendant l'expérience utilisateur plus agréable ;
- À sécuriser au maximum le programme, en prenant en compte le maximum de cas différents, lors de la saisie par l'utilisateur.

De plus, nous avons utilisé les types de variables nécessitant le minimum de mémoire pour l'optimisation, hormis certaines où une grande variable était préférable pour avoir une sécurité sur le maximum de cas (ex : `usr_inpt`).

Avant de commencer l'explication de chaque exercice, nous devons décrire trois fonctionnalités communes à tous :

❖ Boucle principale

Quasiment l'ensemble du code est à l'intérieur d'une boucle `do while`, où la condition n'est d'autre qu'une variable `running` valant soit 1, soit 0.

Elle représente donc un booléen, la rendant exploitable par la boucle. Sa valeur est initialisée à 1 au tout début du code, et est mise à 0 uniquement lorsque l'utilisateur choisi d'arrêter le programme, grâce au menu.

❖ Menu

Dans chaque programme, un menu est accessible pour lancer l'exercice, autant de fois que l'on veut, ainsi que pour l'arrêter.

Cette fonctionnalité nécessite, dans un premier temps, à afficher le menu, grâce à un `printf()`, et à récupérer le choix de l'utilisateur, puis dans un second temps, à utiliser un `switch`.

Le premier choix correspond à lancer le programme (*case 1* dans le *switch*), contenant l'algorithme de l'exercice. Le second choix correspond à l'arrêt du programme.

❖ Vérification des saisies

Chaque saisie par l'utilisateur est contrôlée par un algorithme en trois vérifications :

- Type de la variable respectée ;
- Inférieur au minimum ;
- Supérieur au maximum

Lorsqu'une étape est validée, un compteur *check_step* s'incrémente. Cette variable doit être égale à 3 pour que la saisie soit acceptée. Si la vérification échoue, le compteur est réinitialisé et la boucle *while* recommence.

Une variable *type_check* est utilisée pour vérifier si l'utilisateur a bien rentrer une valeur correspondant au type de variable demandé.

Si le programme nécessite une valeur numérique entière (*int* par exemple), mais que l'utilisateur entre une chaîne de caractères, la variable sera égale à 0.

Dans le cas où le type est respecté, la variable sera égale aux nombres de types respectés.

Exemple :

```
type_check = scanf("%d %c", &a, &b);
```

➔ Si la première saisie de l'utilisateur est un nombre entier, et que la deuxième est un caractère, *scanf()* retourne la valeur 2, qui sera affectée à la variable *type_check*.

➔ Si la première saisie est un nombre entier, mais que la deuxième est autre qu'un caractère, *scanf()* retourne la valeur 1, qui sera affectée à la variable *type_check*.

➔ Si aucun des saisies ne correspond au code format, *scanf()* retourne la valeur 0, qui sera affectée à la variable *type_check*.

Après le contrôle réussi, la variable *check_step* est de nouveau réinitialisée, afin que la prochaine vérification ne soit pas ignorée.

Exercices

❖ Exercice 1

Nous avons fixé le nombre maximum de lignes à 100, car nous avons jugé qu'avec cette valeur, l'image était suffisamment grande.

De plus, nous avons ajouté une fonctionnalité de génération de nombre aléatoire, entre 1 et 20 (inclus), rendant le processus légèrement plus automatisé.

```
119     // Cas où l'utilisateur veut générer un nombre aléatoire de lignes
120     if (n == 0)
121     {
122         n = rand() % 20 + 1;
123         printf("Nombre de lignes al%catoirement g%cn%cr%c : %d\n", 130, 130, 130, 130, n);
124     }
```

Ensuite, si le nombre de ligne choisi est égal à 1, il n'y a nullement besoin de réaliser tout un algorithme. Un simple printf suffit, puis nous ignorons le reste du code.

```
126     // Cas particulier où l'utilisateur veut qu'une seule ligne
127     if (n == 1)
128     {
129         printf("* *\n");
130         break; // Permet de passer les étapes ci-dessous
131     }
```

Viens maintenant le cœur de du programme. Pour réaliser le pattern demandé, nous avons diviser le dessin en deux parties si le nombre de lignes est pair, et en trois parties pour le cas impair.

Ceci explique pourquoi nous avons diviser la variable de la boucle, `i`, par deux. Par la suite, nous avons remarqué que chaque moitié est composée de 3 triangles : 2 triangles constitués d'étoiles, et un d'espace entre ces derniers. Nous avons donc utilisé une boucle pour la moitié, et trois autres pour les triangles.

```
133     // Partie haute
134     for (i = 0; i < n / 2; i++) // n / 2 -> nombre de ligne de la partie haute de la figure
135     {
136         for (j = 0; j <= i; j++) // Premier triangle
137             printf("* ");
138 
139         for (j = i + 2; i + j < n; j++) // "pyramide" d'espaces entre les 2 triangles
140             printf("  ");
141 
142         for (k = 0; k <= n - j; k++) // Deuxième triangle
143             printf("* ");
144 
145         printf("\n");
146     }
```

Ensuite, nous arrivons à la ligne centrale si n est impair.

```
148     // Ligne centrale uniquement si n est impair
149     if (n % 2 == 1)
150     {
151         for (i = 0; i < n; i++)
152             printf("* ");
153 
154         printf("\n");
155     }
```

Et pour terminer, nous avons la partie basse, également dotée d'une grande boucle avec une pour chaque triangle.

```
157     // Partie basse
158     for (i = 0; i < n / 2; i++) // n / 2 -> nombre de ligne de la partie basse de la figure
159     {
160         for (j = 0; j + i < n/2; j++) // Premier triangle
161             printf("* ");
162 
163         for (j = n % 2 == 0 ? 1 : 0; j <= 2 * i; j++) // "pyramide" d'espace entre les 2 triangles
164             printf("  ");
165 
166         for (j = i; j < n/2; j++) // Deuxième triangle
167             printf("* ");
168 
169         printf("\n");
170     }
```

❖ Exercice 2

Notre programme peut se diviser en deux parties : une si l'utilisateur rentre un chiffre, et l'autre partie s'il saisit un nombre.

Pour affecter le nombre de barres nécessaires pour chaque chiffre, nous utilisons un *switch*. Les chiffres avec le même nombre de barres ne seront pas espacés par des *break*. Sans ce mot-clé, les cas se suivront jusqu'à en rencontrer un, où une instruction sera utilisable pour tous ces cas. Petit passage pour illustrer :

```
case 0:
case 6:
case 9:
    bars = 6;
    break;
```

Pour le cas du nombre, plus d'instructions sont nécessaires. Tout d'abord, nous devons extraire chaque chiffre.

```
digit1 = n / 10;
digit2 = n % 10;
```

Ensuite, nous avons une boucle permettant de dissocier les traitements du premier chiffre et du second. Cela reprend le *switch* précédent mais avec une addition des barres.

```
case 0:
case 6:
case 9:
    bars = 6;
    bars_sum += bars;
    break;
```

Le code se termine sur l'ensemble des *printf()*, avec tous les détails pour aider l'utilisateur à comprendre tout le processus de calcul (nombre de barres par chiffre, somme des barres et des chiffres et résultat final : magique ou non).

❖ Exercice 3