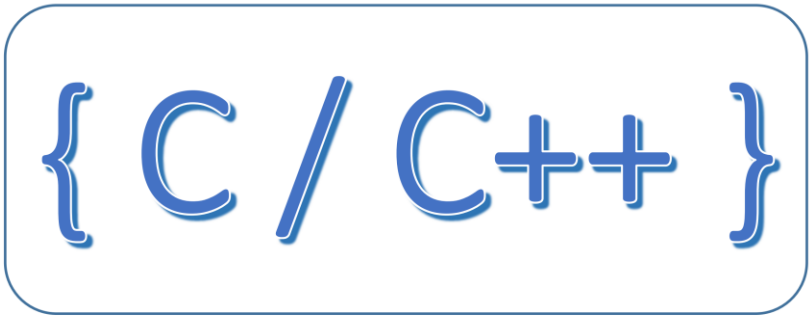
LYCÉE ST-JOSEPH HASPARREN BTS SN-IR	C++ STRUCTURES DE CONTROLE	COURS

Tâches professionnelles : T7.2 Réaliser une maquette, un prototype logiciel/matériel

Compétences du référentiel : C4.4 Développer un module logiciel.

Savoirs - Savoir faires : S4.7 Langage de programmation C++

Table des matières



1 LES INSTRUCTIONS SIMPLES ET CELLES DE CONTROLES

Une instruction C ++ simple, comme les déclarations de variables et expressions vu dans les cours précédents, se terminent toujours par un point-virgule (;), et sont exécutées dans l'ordre dans lequel elles apparaissent dans un programme.

Cependant, les programmes ne sont pas limités à une séquence linéaire d'instructions. Au cours de son processus, un programme peut répéter des segments de code, ou prendre des décisions et bifurquer. A cet effet, C ++ fournit des instructions de contrôle qui servent à préciser ce qui doit être fait par notre programme, quand et dans quelles circonstances.

La structure de contrôle va faire exécuter soit une instruction unique, terminée par un point - virgule (**;**), soit une instruction composée. Une instruction composée est un groupe d'instructions (chacune d'entre elles terminée par son propre point-virgule), mais toutes regroupées dans un bloc, entre accolades **{}** :

```
{ instruction1; instruction2; instruction3; }
```

Le bloc entier est considéré comme une seule instruction (elle-même composée de plusieurs sous-instructions).

2 INSTRUCTION DE SELECTION : IF ET ELSE

Le mot - clé `if` est utilisée pour exécuter une instruction ou un bloc, si, et seulement si, une condition est remplie. Sa syntaxe est:

```
if (condition) instruction
```

Ici, la **condition** est l'expression qui est en cours d' évaluation. Si cette **condition** est vrai, **instruction** est exécutée. Si elle est fausse, **instruction** ne soit pas exécuté (il est simplement ignoré), et le programme continue juste après l'intégralité de la déclaration de sélection.

Par exemple, le fragment de code suivant affiche le message (**x is 100**), seulement si la valeur mémorisée dans la **x** variable **est** en effet **100** :

```
1 if (x == 100)
2   cout << "x is 100";
```

Si **x n'est pas** exactement **100**, cette instruction est ignorée, et rien n'est affiché.

Si vous voulez inclure plus d'une instruction à exécuter lorsque la condition est remplie, ces instructions doivent être enfermées dans des accolades (**{}**), formant un bloc :

```
1 if (x == 100)
2 {
3    cout << "x is ";
4    cout << x;
5 }
```

Comme d'habitude, l'indentation et les sauts de ligne dans le code sont sans effet, de sorte que le code ci-dessus est équivalent à :

```
if (x == 100) { cout << "x is "; cout << x; }
```

L'instruction de sélection avec **`if`** peut également spécifier ce qui se passe lorsque la condition n'est pas remplie, en utilisant le mot- clé **`else`** pour introduire une instruction de remplacement. Sa syntaxe est:

`if (condition) instruction1 else instruction2`

où **`instruction1`** est exécutée si **`condition`** est vraie, et sinon **`instruction2`** est exécutée.

Par exemple :

Affiche **`x is 100`** , si en effet ***x*** est égale à ***100***, sinon il affiche **`x is not 100`**.

```
if (x == 100)
  cout << "x is 100";
else
  cout << "x is not 100";
```

Plusieurs **if + else** peuvent être concaténées pour vérifier une plage de valeurs :

Affiche un message différent suivant que x soit positif, négatif, ou nul, par concaténation de deux structures if-else.

Bien sûr, il aurait également été possible d'exécuter plus d'une instruction par cas en les regroupant en blocs entre accolades `{}`.

```
if (x > 0)
  cout << "x is positive";
else if (x < 0)
  cout << "x is negative";
else
  cout << "x is 0";
```

Testez ces différentes versions de if.

3 INSTRUCTIONS ITERATIVES (BOUCLES)

Les boucles permettent de répéter une instruction un certain nombre de fois, ou tant qu'une condition est remplie. Elles utilisent les mots - clés **`while`**, **`do`**, et **`for`** .

3.1 La boucle while

Le type de boucle le plus simple est la boucle **while**. Sa syntaxe est:

`while (expression) instruction`

La boucle ***`while`*** répète ***`instruction`*** tant que ***`expression`*** est vraie. Dès que ***`expression`*** n'est plus vraie, la boucle se termine, et le programme se poursuit juste après la boucle. Si dès le début ***`expression`*** n'est pas vraie, on ne rentre pas dans la boucle ***`while`*** et ***`instruction`*** n'est pas exécutée, même pas une fois.

Par exemple, ceci est un compte à rebours fait à l'aide d'une boucle while:

```
// custom countdown using while
#include <iostream>
using namespace std;

int main ()
{
  int n = 10;

  while (n>0) {
    cout << n << ", ";
    --n;
  }

  cout << "liftoff!\n";
}
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!
```

La première instruction dans le **`main`** initialise ***n*** à une valeur de **10**. C'est le premier nombre dans le compte à rebours. Ensuite, la boucle while commence : si cette valeur remplit la condition **`n>0`** (ce qui est vrai puisque ***n*** **est égale à 10**), alors le bloc qui suit l'état est exécuté et répété aussi longtemps que la condition (**`n>0`**) reste étant vrai.

L'ensemble du processus du programme précédent peut être décrit selon le script suivant (à partir de `main`):

1. On attribue une valeur (10) à `n`
2. La condition du `while` est vérifiée (`n>0`). À ce stade, il y a deux possibilités :
   - La condition est vraie : l'instruction est exécutée (on va à l'étape 3)
   - La condition est fausse : ignorer et continuer après l'instruction (à l'étape 5)
3. Exécuter le bloc d'instructions: (cela affiche la valeur de `n` et décrémente `n`)
   ```
   cout << n << ", ";
   --n;
   ```
4. Fin du bloc. Retour automatiquement à l'étape 2.
5. Poursuivre le programme juste après le bloc
6. Afficher le `liftoff!` et la fin du programme.

Une chose qu'il faut prendre en considération avec toute boucle est que **la boucle doit se terminer à un moment donné !**
Et donc qu'une des instructions du bloc doit modifier la valeur vérifiée dans la condition, de manière à la forcer à devenir fausse à un moment donné.
Sinon, la boucle continuera en boucle pour toujours ! Dans notre cas, la boucle comprend **`--n`**, qui décrémente la valeur de la variable qui est en cours d'évaluation dans la condition (`n`), ce qui finira par rendre la condition (`n>0`) fausse après un certain nombre d'itérations de la boucle . Pour être plus précis, après 10 itérations, `n` devient égale à 0, ce qui met fin à la boucle while.

3.2 La boucle do-while

La boucle **do-while** est une boucle très similaire, dont la syntaxe est:

`do instructions while (condition);`

La particularité est que la **`condition`** est évaluée après l'exécution de **`instruction`** , et pas avant, garantissant au moins une exécution de **`instruction`** , même si la **`condition`** est jamais remplie.
Notez aussi la présence du point-virgule à la fin !

Par exemple, le programme suivant fait écho à tout texte saisi par l'utilisateur jusqu'à l'entrée de « Au revoir »:

Testez ce programme

L'utilisation de do-while est recommandée quand le test de fin de boucle ne pourrait pas se faire sans au moins un passage dans la boucle.

```
#include <iostream>
#include <string>

using namespace std;

void main()
{
    string str{ "" };

    do {
        cout << "Saisissez votre nom, ou \"Au Revoir\" pour sortir: ";
        getline(cin, str);
        cout << "Vous avez saisi: " << str << endl;
    } while (str != "Au revoir");
}
```

Dans l'exemple précédent, c'est la saisie par l'utilisateur dans le bloc qui va déterminer si la boucle se termine. Et donc, même si l'utilisateur veut mettre fin à la boucle dès que possible en entrant «Au revoir», le bloc dans la boucle doit être exécuté au moins une fois pour demander la saisie, et la condition de fin ne pourrait pas être testée sans la saisie.

3.3 La boucle for

La boucle **for** est conçue pour itérer dans la boucle un certain nombre de fois. Sa syntaxe est:

`for (initialisation; condition; incrémentation) instruction;`

Comme la boucle **`while`**, cette boucle répète **`instruction`** tant que **`condition`** est vrai. Mais, en outre, la boucle fournit des emplacements spécifiques pour définir une **`initialisation`** et une **`incrémentation`** d'une variable. Cette **`incrémentation`** est exécutée avant que la boucle débute la première fois, et après chaque itération. Par conséquent, il est particulièrement utile d'utiliser des variables de comptage comme **`condition`** .

Cela fonctionne de la manière suivante :

1. l'**`initialisation`** est exécutée. En général, cela déclare une variable compteur, et la fixe à une certaine valeur initiale. Ceci est exécuté une seule fois, au début de la boucle.
2. **`condition`** est vérifiée. Si elle est vraie, la boucle continue ; sinon, la boucle se termine, et **`instruction`** est ignorée, on va directement à l'étape 5.
3. **`instruction`** est exécutée. Comme d'habitude, il peut être soit une seule déclaration ou un bloc entre accolades **`{ }`** .
4. **`incrémentation`** est exécutée, et la boucle revient à l' étape 2.
5. la boucle se termine: l'exécution se poursuit par l'instruction suivante.

Voici l'exemple du compte à rebours en utilisant une boucle for :

```
1  // countdown using a for loop
2  #include <iostream>
3  using namespace std;
4
5  int main ()
6  {
7    for (int n=10; n>0; n--) {
8      cout << n << ", ";
9    }
10   cout << "liftoff!\n";
11 }
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!
```

Les trois champs dans une boucle **`for`** sont facultatifs. Ils peuvent être laissés vides, mais dans tous les cas, les points-virgules entre eux sont nécessaires.

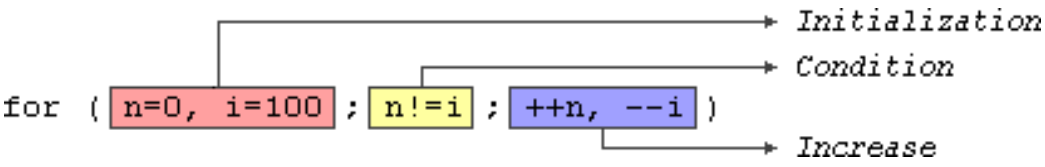
Par exemple, **`for(;n<10;)`** est une boucle sans *initialisation* ni *incrémentation* (ce qui équivaut à une boucle while); et **`for(;n<10;++n)`** est une boucle avec une *incrémentation,* mais aucune *initialisation* (peut-être parce que la variable a déjà été initialisée avant la boucle). Une boucle sans *condition* est équivalente à une boucle avec la condition toujours à `true` (par exemple, une boucle infinie).

Dans le champ initialisation on peut initialiser plusieurs variables, en les séparant par une virgule. De même dans le champ incrémentation on peut agir sur plusieurs variables.

Par exemple, il serait possible pour une boucle de manipuler deux variables de compteur :

```
for ( n=0, i=100 ; n!=i ; ++n, --i )
{
   // whatever here...
}
```

Cette boucle s'exécutera 50 fois (à condition qu'il n'y ait pas d'autres modifications sur n ou i dans la boucle) :



`n` commence par une valeur de 0, `i` à 100, la condition est `n!=i` (`n` pas égal à `i`). Parce que `n` est incrémenté, et `i` décrémenté à chaque itération, l'état de la condition deviendra false après la 50ème itération, lorsque `n` et `i` sont égaux à 50.

3.4 La boucle for basée sur une gamme (range-base)

La boucle for a une autre syntaxe, qui est exclusivement utilisée avec des gammes, des plages de valeurs.

`for ( déclaration : range ) instructions;`

Ce type de boucle for passe en revue tous les éléments de **`range`** , et la variable déclarée dans la **`déclaration`** prend successivement la valeur de chaque élément dans **`range`** (il faut que cette variable aie un type compatible avec les éléments de **`range)`** .

Une gamme est une séquence d'éléments, comme par exemple une chaine de caractère, un tableau, un conteneur et autres collections d'éléments. La plupart de ces types seront vus plus tard, mais nous connaissons déjà avec au moins un type de gamme : les **string**, qui sont des séquences de caractères.

Un exemple de boucle *range-based* en utilisant des chaînes :

```
// range-based for loop
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string str {"Hello!"};
  for (char c : str)
  {
    std::cout << "[" << c << "]";
  }
  std::cout << '\n';
}
```

```
[H][e][l][l][o][!]
```

La déclaration d'une variable **`c`** de type **`char`** va permettre d'aller lire un par un les éléments du **`string str`** (une chaine est composée de caractères). Nous utilisons ensuite cette variable, dans le bloc d'instruction de la boucle, pour afficher la valeur de chacun des éléments de la gamme.

Cette boucle est automatique, et ne nécessite pas la déclaration explicite d'une variable compteur.

Les boucles à base de **range** font généralement usage de **`auto`** pour la déduction automatique du type de la variable dans **`déclaration`**. La boucle range-based au-dessus peut également être écrite comme ceci :

Ici, le type de **`c`** est automatiquement déduit à partir du type des éléments dans **`str`**.

4 DECLARATIONS DE SAUT

Les déclarations de saut permettent de modifier le flux d'un programme en effectuant des sauts à des endroits précis

4.1 L'instruction break

L'instruction **`break`** permet de quitter une boucle, même si la condition de sa fin n'est pas remplie. Elle peut être utilisée pour mettre fin à une boucle infinie, ou la forcer à se terminer avant sa fin naturelle.

Les instructions qui suivent le **`break`** dans le corps de la boucle, sont **ignorées**. Dès que le **`break`** intervient on quitte **immédiatement** la boucle.

Par exemple, nous allons arrêter le compte à rebours avant sa fin naturelle :

```
1  // break loop example
2  #include <iostream>
3  using namespace std;
4
5  int main ()
6  {
7    for (int n=10; n>0; n--)
8    {
9      cout << n << ", ";
10     if (n==3)
11     {
12       cout << "countdown aborted!";
13       break;
14     }
15   }
16 }
```

```
10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!
```

4.2 Exercice :

Vous devez créer un jeu qui s'appelle le **Nombre mystère**.

On crée un nombre mystère aléatoire entre 0 et 100 (partie de programme déjà écrite).

Le joueur à droit à 10 essais ; à chaque fois on lui dit s'il est trop haut, trop bas ou s'il a gagné (on garde affichés les essais déjà faits).

A la fin on affiche le nombre mystère, et on lui demande s'il veut rejouer.

Contraintes :

- Pas d'utilisation de : using namespace std (dès que ça se complique il y a un risque de confusion entre des noms de variables créées par vous-même, et des noms du namespace std ; autant prendre l'habitude de préciser chaque fois std :: même si c'est lourd).
- Utilisation des instructions de contrôle suivantes (if-else, for, do-while)

```cpp
// génération de nombre aléatoire entre 0 et 100
//
//

#include <cstdlib>              // pour std::rand()
#include <ctime>                // pour std::time()
#include <iostream>             // pour std::cin, std::cout
#include <limits>               // pour cin.ignore
#include <clocale>              // pour gestion des caractères accentués


int main()
{
    setlocale(LC_CTYPE, "fra");     // pour gestion des caractères accentués

    const int maxi(100);            //le max
    const int mini(0);              // le min
    std::srand(std::time(0));       // utilise le temps time courant comme base du générateur aléatoire

    // Générer le nombre mystère et demander de deviner la valeur
    int mystere = std::rand() % maxi + mini;       // génération d'un nombre mystère compris entre mini et maxi

    // dire au joueur s'il est trop haut, trop bas ou s'il a gagné
    // lui dire de retenter (il a droit à 10 essais)
    // ...



    std::cout << "Le nombre mystère était: "        << mystere << '\n';

    // Demander s'il veut rejouer, et si oui recommencer, sinon dire au revoir


    //std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');  // efface ce qui pourrait trainer dans le buffer
    std::cout << "Taper Entrée pour continuer";                          // équivalent à system("pause")
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
}
```

4.3 L'instruction continue

L'instruction **`continue`** fait que le programme ignore les instructions suivantes dans la boucle, pour l'itération courante, comme si la fin du bloc d'instructions avait été atteint, l'amenant ainsi à passer au début de l'itération suivante. Par exemple, nous allons sauter le numéro 5 dans notre compte à rebours :

```
1  // continue loop example
2  #include <iostream>
3  using namespace std;
4
5  int main ()
6  {
7    for (int n=10; n>0; n--) {
8      if (n==5) continue;
9      cout << n << ", ";
10   }
11   cout << "liftoff!\n";
12 }
```

```
10, 9, 8, 7, 6, 4, 3, 2, 1, liftoff!
```

4.4 L'instruction goto

L'instruction ***`goto`*** permet de faire un saut absolu à n'importe quel autre point dans le programme.
Le point de destination est identifié par une ***étiquette,*** qui est ensuite utilisé comme un argument pour l'instruction ***`goto`***. Une ***étiquette*** est faite d'un identificateur valide suivi par deux points (:).

Ce saut inconditionnel ignore les niveaux d'imbrication, et aussi plusieurs autres mécanismes importants.
`goto` est généralement considérée comme une fonction de bas niveau, sans utilité particulière, et **proscrite en application professionnelle**.

Donc son utilisation est formellement interdite dorénavant, vous pouvez l'oublier.

Mais, juste pour l'exemple, voici une version de notre boucle de compte à rebours en utilisant goto :

```
1  // goto loop example
2  #include <iostream>
3  using namespace std;
4
5  int main ()
6  {
7    int n=10;
8  mylabel:
9    cout << n << ", ";
10   n--;
11   if (n>0) goto mylabel;
12   cout << "liftoff!\n";
13 }
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!
```

5 UNE AUTRE INSTRUCTION DE SELECTION : SWITCH

Le but de l'instruction switch est de **tester une valeur parmi un certain nombre d'expressions constantes possibles**. C'est un peu comme si concaténait **plusieurs `if - else`**, mais limité à des expressions constantes.
Sa syntaxe la plus typique est :

```
switch (expression) {
       case constant1: groupe d'instructions 1; break;
       case constant2: groupe d'instructions 2; break;
       . . .
       default:       groupe d'instructions par défaut ;
}
```

Cela fonctionne de la manière suivante :

`switch` évalue **`expression`** :

- Il vérifie si elle est égale à **`constant1`**, il exécute un **`groupe d'instructions 1`** jusqu'à ce qu'il trouve l'instruction **`break`**. Quand il trouve cette instruction **`break`**, le programme saute à la fin de l'ensemble du **`switch`** (l'accolade de fermeture).
- Si non, il vérifie si elle est égale à **`constant2`**. Si oui, il exécute un **`groupe d'instructions 2`** jusqu'à ce qu'il trouve l'instruction **`break`**. Quand il trouve cette instruction **`break`**, le programme saute à la fin de l'ensemble du **`switch`** (l'accolade de fermeture).
- Enfin, si la valeur de **expression** ne correspond à aucune des constantes indiquées précédemment (il peut y en avoir plus que deux), le programme exécute les instructions figurant après l'étiquette **`default:`** si elle existe (car elle est optionnelle).

  Les deux bouts de code suivants ont le même comportement, ce qui démontre l'équivalence avec une instruction if-else :

switch example	if-else equivalent

`switch (x) {`
`  case 1:`
`  cout << "x is 1";`
`  break;`
`  case 2:`
`  cout << "x is 2";`
`  break;`
`  default:`
`  cout << "value of x unknown";`
`  }`
	`if (x == 1) {`
	`  cout << "x is 1";`
	`}`
	`else if (x == 2) {`
	`  cout << "x is 2";`
	`}`
	`else {`
	`  cout << "value of x unknown";`
	`}`

L'instruction **`switch`** a une syntaxe assez particulière héritée des premiers temps du langage **C**, car il utilise des étiquettes (**label**) au lieu de blocs. Dans l'utilisation la plus classique (ci-dessus), cela signifie que l'instruction **`break`** est nécessaire après chaque groupe d'instructions pour une étiquette **`case`** donnée. Si le **`break`** n'est pas inclus, toutes les instructions des **`case`** suivants sont également exécutées, jusqu'à la fin du bloc de **`switch`**.

Dans l'exemple ci-dessus s'il manquait l'instruction break après le premier groupe dans le **case 1:** , le programme ne sauterait pas automatiquement à la fin du bloc de **switch** après l'affichage de **`x is 1`**, et continuerait à exécuter les instructions dans les deux autres blocs **case** (donc également l'affichage de **`x is 2`** et **`x is 2`**).

L'utilisation des accolades **`{}`** pour encadrer les instructions d'un **case** est inutile.

On peut aussi faire exécuter le même groupe d'instructions pour différentes valeurs possibles.

Par exemple :

```
switch (x) {
  case 1:
  case 2:
  case 3:
    cout << "x is 1, 2 or 3";
    break;
  default:
    cout << "x is not 1, 2 nor 3";
  }
```

Notez que le `switch` est limité à comparer l'expression évaluée (ici x) avec des labels qui sont des expressions constantes. Il est impossible d'utiliser des variables comme labels elles ne sont pas des expressions constantes valides de C. Dans ce cas il est préférable d'utiliser des concaténations de `if` et d' `else`.

5.1 Exercice :

Détection de touche clavier

Demander à l'utilisateur de taper sur une des touches de la rangée supérieure du clavier (de A à Y) en majuscules ou minuscules.

Le programme affiche : **Vous avez tapez sur la 4ème touche : « R » !**

Ou **Vous avez tapez sur la 4ème touche : « r » !**

Si appui sur ***** sortie du programme, sinon recommencer.

Contraintes :

- Message d'erreur si autre touche appuyée
- Utilisation de switch, do-while

6 EXERCICES

6.1 Exercice Animation tournante

Vous devez écrire le programme qui produit l'animation tournante que vous présente le professeur.

Pour que l'animation soit visible vous devrez utiliser une temporisation entre les différents affichages. Pour cela il faudra utiliser l'instruction Sleep() en n'oubliant pas d'inclure la bibliothèque (library) qui va avec :

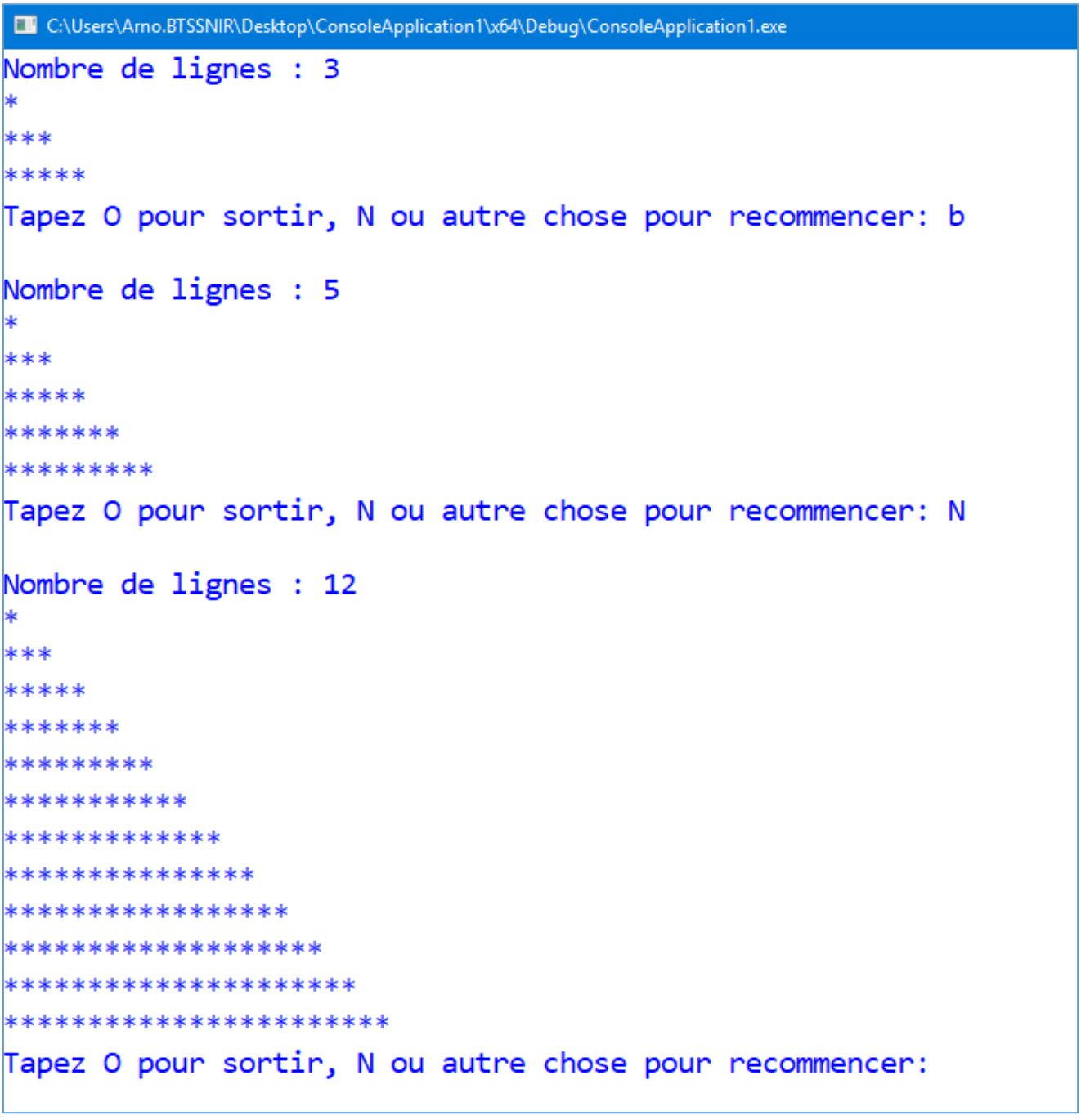
```
#include <windows.h>

Sleep(number of milliseconds);
```

6.2 Exercice Pyramide d'étoiles version 1

Vous devez écrire le programme qui produit le résultat suivant :

On saisit le nombre de lignes et le programme dessine la pyramide correspondante.

Le programme doit fonctionner quel que soit le nombre de lignes.



6.3 Exercice Pyramide d'étoiles version 2

Vous devez écrire le programme qui produit le résultat suivant :

On saisit le nombre de lignes et le programme dessine la pyramide correspondante.

Le programme doit fonctionner quel que soit le nombre de lignes.