Cette première présentation du langage de programmation C++ est écrite dans le cadre du cours C++ organisé au club informatique de Loriol-sur-Drôme (http://ciloriol.fr).

Il est écrit et présenté au sein du club par Jean-Claude CATY (jc<année>.caty@laposte.net).

Table des matières

1. Pour commencer, le programme ne fait rien

Voici le squelette du programme « compterCar » qui, à ce stade, ne fait rien !

```
int main () {

    return 0;
    }
```

Ligne « `int` main () { »

- `Int`

  La fonction main (explications sur cette fonction dans le paragraphe ci-dessous), renvoie un entier. `int` est un mot prédéfini du langage. Il s'agit d'un nombre entier signé dont la valeur est comprise entre -32 768 et 32 767. Ce nombre est codé sur 2 octets (16 bits).

- Main ()

  Traduction : « main » = « principal ».
  Le nom « main » de cette fonction sera compris comme le point de départ du programme. Tout programme C++ doit avoir une fonction « main ». Le compilateur cherchera une fonction « main » pour définir le point de départ du programme. Notons que dans une librairie, « main » sera remplacé par « libmain » Les parenthèses laissées vides signifient que la fonction ne prend aucun paramètre. Nous verrons que la fonction « main » peut prendre des paramètres qui correspondront aux paramètres passés sur la ligne de commande qui appellera le programme. Dans ce cas, il faudra modifier cette ligne.

- {

  L'accolade ouvrante délimite le début des instructions de la fonction « main ».

Ligne « `return 0;` »

- Nous avons vu précédemment que la fonction « main » renvoie un entier. Le renvoi est matérialisé par le mot clef « return ». Le nombre entier renvoyé ici est 0. Nous pourrions par exemple décider conventionnellement de renvoyer 0 s'il n'y a pas d'erreur et 1 s'il y a une erreur. Éventuellement, nous pourrions décider de renvoyer différentes valeurs entières dont la signification varierait selon le contexte du programme. Cette valeur de retour peut par exemple être ensuite exploitée dans la ligne de commande.

Ligne « `}` »

- }

  De la même manière que nous avions une accolade ouvrante pour délimiter le début des instructions de la fonction « main ». nous avons maintenant l'accolade fermante pour délimiter la fin de la fonction « main ».

Compilation

Si vous compilez ce programme, il n'y a pas d'erreurs. Contrairement à ce qui a été dit en titre de ce chapitre, il fait quelque chose : il vous renvoie 0 et bouffe un peu d'électricité pour ne presque rien faire. Les fainéants vous diront toujours qu'il faut brasser un peu d'air pour laisser croire que !

En tant que membres du pôle éolien, vous constatez d'ailleurs qu'il faut une page complète pour faire du vent …

2. Introduisons la machine caractère

Notre programme va compter le nombre de caractères sur le ruban d'une machine caractère. Nous allons donc avoir besoin d'une machine caractère et d'un compteur. Ajoutons donc ces deux informations à notre programme.

```
int main () {

      uint16_t lvCompteur;
      CMachineCaractere lvMachine;
      return 0;
      }
```

Ligne « uint16_t lvCompteur; »

- uint16_t

  uint16_t est un type de donnée représentant un nombre entier non signé codé sur 16 bits (2 octets). Le préfixe « u » signifie « unsigned ». Notre compteur n'a pas besoin de compter négativement. Ce sera toujours un nombre débutant à 0. Nous le précisons donc de cette manière. Le suffixe « 16 » précise que notre nombre est codé sur 16 bits. Sa valeur maximale est donc 65 535 (1111 1111 1111 1111 en binaire). Nous estimons donc que notre machine caractère comportera donc moins de 65 536 caractères.

- LvCompteur

  Nous donnons à notre variable un nom explicite nous permettant de mieux comprendre le code source. La variable est préfixée de « lv » (voir conventions personnelles définie en annexe 0).

Ligne « CMachineCaractere lvMachine; »

- CMachineCaractere

  CMachineCaractere est une classe C++ dont nous allons utiliser les méthodes publiques (méthode : nom donné aux fonctions d'une classe). Le nom de la classe est préfixé par le caractère « C » (voir conventions personnelles définie en annexe 0).

- lvMachine

  Nous donnons à l'objet lvMachine un nom explicite. lvMachine est un « représentant de la classe CMachineCaractere.

Compilation

Si vous compilez ce programme, deux erreurs sont signalées par le compilateur.

- *'uint16_t' was not declared in this scope*. (*'uint16_t' n'est pas déclaré dans ce champ de vision*) Le type « uint16_t » est inconnu du compilateur. Il ne sait pas comment l'interpréter. Il faut donc soit définir « uint16_t », soit préciser le fichier dans lequel figure la définition de ce type. C'est la deuxième solution que nous allons choisir. Une rapide recherche sur internet nous conduit sur la page www.cplusplus.com/reference/cstdint/. Là, nous apprenons que le type uint16_t et beaucoup d'autres sont définis dans le fichier **cstdint**. Nous corrigeons donc notre programme en ajoutant la directive **`#include <cstdint>`** sur la première ligne.

- *'CMachineCaractere' was not declared in this scope*. (*'CMachineCaractere' n'est pas déclaré dans ce champ de vision*) Comme précédemment pour le type « uint16_t », « CMachineCaractere » est inconnu du compilateur. Il ne sait pas comment l'interpréter. Nous allons donc, comme précédemment préciser le fichier dans lequel figure la définition de cette classe. La définition figure dans l'entête du fichier « machineCaractere.h » que nous avons intégré au projet. Nous corrigeons donc notre programme en ajoutant la directive **`#include "machineCaractere.h"`** sur la deuxième ligne du programme.

```
#include <cstdint>
#include "machineCaractere.h"

int main () {
```

```
    uint16_t lvCompteur;
    CMachineCaractere lvMachine;
    return 0;
    }
```

Deuxième Compilation

Si vous compilez ce programme, une nouvelle erreur est signalée par le compilateur. Les précédentes ont bien disparues.

- *No matching function for call to 'CMachineCaractere::CMachineCaractere ()'* (pas de fonction *'CMachineCaractere::CMachineCaractere ()' à invoquer*)
  Là, le compilateur nous dit qu'il n'a pas trouvé de fonction « *CMachineCaractere::CMachineCaractere ()* » à invoquer. Que comprendre dans ce message ? Une fonction nommée « CMachineCaractere », préfixée par « CMachineCaractere :: », soit le même nom que la classe dont fait partie notre objet lvMachine. À ce stade, rappelons-nous que le principe d'encapsulation est là pour garantir une cohérence interne à nos objets. En émettant cette erreur, le compilateur nous le rappelle bruyamment. En effet, pour garantir cette cohérence, tout objet d'une classe doit être défini dès sa naissance en appelant un « constructeur » de la classe. Tout constructeur d'une classe que nous appellerons « CClasse » porte le même nom que la classe. Par ailleurs, un constructeur n'est qu'une fonction particulière. À l'intérieur de la classe, il porte le nom de la classe, soit « CClasse () ». À l'extérieur de la classe, il faudra rappeler la classe d'appartenance en préfixant par « CClasse:: ». Un rapide coup d'oeil sur la documentation de la classe nous indique que la prototype d'un constructeur de la classe CMachineCaractere est « CMachineCaractere::CMachineCaractere (const string& kprRuban) »

- Analyse du paramètre « const string& kprRuban »
  Le mot clef const signifie « constant » et donc que le constructeur ne pourra modifier ce paramètre. Le type « string » est une chaine de caractères. Le caractère « & » placé à la suite du type string signifie que le paramètre est une référence à une variable déjà définie par ailleurs (en principe dans la fonction appelante). Il s'agit donc de la même variable à qui nous donnons un autre nom (ici « kprRuban »). En préfixant par « const », le concepteur du constructeur vous précise que cette variable ne sera pas modifiée par ce constructeur. De surcroît, le compilateur y veillera ! Nous ajoutons donc un paramètre string à notre déclaration d'une machine caractère : CMachineCaractere lvMachine ("Elle n'est pas veuve"). Notre programme devient :

```
#include <cstdint>
#include "machineCaractere.h"

int main () {

    uint16_t lvCompteur;
    CMachineCaractere lvMachine ("Elle n'est pas veuve");
    return 0;
    }
```

Pas d'erreur de compilation. Pour commencer à faire autre chose que du vent, il a fallu 2 pages !

3. Boucle de calcul « Tant que »

Pour déterminer le nombre de caractères de la machine caractères, nous allons utiliser une boucle du type « Tant que ». Le schéma algorithmique d'une telle boucle est défini en annexe 1.

Initialiser la boucle

L'initialisation de la boucle comprend deux instructions. L'initialisation de la machine caractère et celle du compteur.

Passons rapidement l'initialisation de la machine caractère. La documentation précise qu'après initialisation, la fenêtre de la machine caractère est positionnée au début du ruban. C'est ce que nous souhaitions !

L'initialisation du compteur est fonction de l'état du système avant d'entrer dans la boucle. A ce stade, nous n'avons pas commencer à compter. Le compteur est donc égal à zéro. L'instruction correspondante est :

```
lvCompteur = 0;
```

Attention : La déclaration de la variable lvCompteur ne fait que réserver un emplacement mémoire. En aucun cas, elle n'initialise la variable à une quelconque valeur. Par contre, nous aurions pu fusionner la déclaration et l'initialisation avec l'instruction suivante :

```
uint16_t lvCompteur = 0;
```

Condition de la boucle

Nous comptons le nombre de caractères du ruban. Aussi devons-nous passer l'intégralité des caractères. C'est à dire continuer tant que la marque n'est pas atteinte. En disant cela, nous venons de dire notre condition : *tant que la marque n'est pas atteinte* ! La machine caractère nous permet de tester si nous avons atteint la marque :

```
bool CMachineCaractere::marque ()
```

Cette méthode nous renvoie une valeur booléenne (bool). Ce type a deux valeurs possibles : vrai (en anglais : true) ou faux (false).

L'instruction C++ correspondante est (nous appliquons la méthode « marque () » à la variable « lvMachine ») :

```
while (!lvMachine.marque ())
```

Le point d'exclamation précédant « lvMachine » représente la négation. Il inverse donc le résultat de la méthode « marque () ».

Traitement de la boucle

Le traitement consiste, dans le contexte, à faire évoluer le comptage de façon à ce que le compteur corresponde au nombre de caractères à gauche de la position de la fenêtre (fenêtre comprise).

L'instruction C++ correspondante consiste soit à ajouter 1 au compteur :

```
lvCompteur = lvCompteur + 1;
```

soit à utiliser l'opérateur d'incrémentation « ++ » :

```
lvCompteur++;
```

Avancement de la boucle

L'avancement, dans le cas du comptage, consiste à passer au caractère suivant qui sera éventuellement la marque lorsque la fenêtre aura parcouru la totalité du ruban. Pour avancer, nous utilisons la méthode « avancer » de la machine caractère :

```
lvMachine.avancer ();
```

La boucle while

En encadrant l'ensemble des instructions « Traitement de la boucle » et « Avancement de la boucle » par des accolades délimitant la portée de l'instruction while, nous obtenons le code source suivant :

```
    uint16_t lvCompteur = 0; // Initialiser
    while (!lvMachine.marque ()) { // Tant que condition
          lvCompteur++; // Traiter
          lvMachine.avancer (); // Avancer
          };
```

Remarquez que l'ensemble des instructions est indenté de façon à mieux visualiser cet ensemble. Cette façon de faire n'a strictement rien d'obligatoire. Vous pourriez aligner ces 5 lignes sur une seule (sans les commentaires). Le compilateur n'y verrait que du feu. Par contre, le jour où vous aurez besoin de corriger, il sera beaucoup plus facile de vous y retrouver avec des indentations.

Notez également la présence des commentaires débutant par « `//` ». Ce type de commentaire se termine à la fin de la ligne.

Respectez systématiquement le schéma algorithmique décrit en annexe 1. Bien souvent, une boucle qui tourne indéfiniment signifie un oubli d'une partie du schéma. Par exemple, oubliez l'instruction « lvMachine.avancer () » et vous n'atteindrez JAMAIS la marque symbolisant la fin du ruban et la condition « !lvMachine.marque () » sera TOUJOURS vraie. Et, tant que c'est vrai, votre programme tournera, tournera, tournera, … et le compteur explosera.

Y a t'il des erreurs de compilation ?

4. Et si nous affichions le résultat

Pour afficher le résultat, nous allons utiliser les bibliothèques de flux d'entrées-sorties. Ces flux sont composés de multiples classes C++ dont nous n'allons utiliser ici qu'une partie infime. Un flux permet soit d'envoyer, soit de lire des informations dans des fichiers, des flux en mémoire, sur la console, …

En premier lieu, nous prendrons le flux prédéfini « cout » permettant d'afficher des informations sur la console. Il est défini dans le fichier d'en-tête « iostream » (cf http://www.cplusplus.com/reference/iostream/cout/). Voici un exemple de ligne à insérer dans notre programme :

```
cout << "La machine caractère contient " << lvCompteur << " caractères." << endl;
```

Cette ligne mérite de nombreuses explications ! Avant de poursuivre, préparez-vous un café, un thé à la menthe ou toute autre boisson qui vous donnera le tonus nécessaire pour la suite. Installez-vous confortablement – mais pas trop – pour éviter de plonger dans les bras de morphée. Prenez une première gorgée et en avant !

- **`cout << "La machine caractère contient "`**

  Nous utilisons ici un opérateur matérialisé par un chevron « << »dirigé vers « **`cout`** ». En C++, de nombreux opérateurs peuvent être redéfinis par une classe. Le lien ci-dessus nous donne la définition de cout : « *extern ostream cout;* ». Donc, cout est un objet de type ostream défini ailleurs (« extern »). La définition de l'opérateur « << »utilisant abondamment la généricité, simplifions la définition que nous pourrions trouver dans les fichiers de la bibliothèque. Sans généricité, la définition de cet opérateur serait de la forme « ostream& ostream::operator<< (const string& kprTexte) » à interpréter de la même manière que la fonction « main » vu au premier chapitre :

  - ostream::operator<<

    Il s'agit d'une méthode (fonction) de la classe ostream dont le nom est « operator<< »

  - ostream&

    Cette fonction renvoie une référence (symbolisé par « **&** ») à un objet de type « ostream ». Nous allons bientôt voir qu'il est important que cet opérateur renvoie une référence. Systématiquement, l'opérateur renverra une référence vers l'objet lui-même. Dans notre cas, ce sera une référence vers « cout ».

  - const string& kprTexte

    Dans la parenthèse, nous trouvons un paramètre nommé kprTexte et de type « const string& » : référence à une chaîne de caractère constante. Le fait d'avoir une référence signifie que l'objet de type « string » existe déjà et que nous ne faisons que lui donner un nouveau nom. De plus, afin d'interdire la modification de cet objet, cette référence est préfixée par le mot clef « const ». Si le paramètre n'était pas une référence, une copie de l'objet serait créée. Éviter la copie permet d'économiser le temps de copie et donc d'améliorer la rapidité du programme.

  Une autre manière d'écrire cette partie est « *cout.operator<< ("La machine caractère contient ")* ». Le résultat est donc une référence vers cout (la console) dans laquelle le programme a écrit « "La machine caractère contient " ». Puisque le résultat est une référence vers « cout », nous pouvons continuer à invoquer d'autres méthodes, en particulier celle qui envoie le résultat de notre compteur via un opérateur similaire à ceci près qu'il va prendre un autre type de paramètre : le type constant de lvCompteur soit « const uint16_t& » plutôt que le type chaîne constante « const string& ». La partie

  ```
  cout << "La machine caractère contient " << lvCompteur
  ```

  peut donc s'écrire

  ```
  (cout.operator<< ("La machine caractère contient ")).operator<< (lvCompteur) ;
  ```

  À chacun de voir quelle écriture il préfère ! Mais attention, si d'autres personnes doivent modifier le programme dans le cadre d'une équipe de développement, mieux vaut choisir la syntaxe compréhensible par le plus grand nombre et le plus souvent utilisée.

- **`cout << endl`**

  endl est un « manipulateur ». Il permet d'ajouter une fin de ligne (de passer à la ligne) dans un flux textuel.

Nous verrons, au fil de cet atelier de programmation, d'autres manipulateurs.

Première Compilation

Deux erreurs sont signalées par le compilateur :

- *'cout' was not declared in this scope*.

  Nous retrouvons un type d'erreur que nous avons déjà rencontré. Pour corriger cette erreur, il faut inclure le fichier contenant la définition de la variable « cout ». Ce fichier de définition est « iostream ». La ligne suivante doit donc être ajoutée en tête du programme :

  ```
  #include <iostream>
  ```

- *'endl' was not declared in this scope*.

  Nous avons ici un cas identique au précédent. De plus, le fichier à inclure est situé dans le même fichier de définition que la variable « cout », soit « iostream ».

Deuxième Compilation

Là, surprise ! Malgré l'ajout de la directive d'inclusion du fichier définissant « cout » et « endl », la même erreur persiste. Cependant, les deux lignes suivantes nous proposent une alternative :

> note : suggested alternative :
> note : 'std::cout'

Le compilateur nous suggère de préfixer le mot « cout » par « std :: ». Quelle signification accorder à cette proposition ?

Les variables « cout » et « endl » sont encapsulées dans un « espace de nommage ». Cette possibilité qu'offre le langage C++ consiste à inclure certaines définitions dans un espace nommée. Pour accéder aux informations incluses dans cet espace, nous avons deux solutions :

- Solution 1 : préfixer le nom de la variable par le nom de l'espace de nommage suivi de « :: ». Dans notre cas, chaque référence à « cout » devrait alors être remplacée par « std::cout »
- Solution 2 : Dire expressément que nous travaillons dans un ou plusieurs espaces à l'aide de la directive « using namespace » placée après les directives « includes ». Dans notre cas, ce serait « using namespace std ». Inutile dans ce cas d'ajouter le préfixe « std :: ». Attention cependant au fait que différents espaces de noms peuvent contenir des identifiants de même noms. Si chacun de ces espaces est accessible via une directive « using namespace », le compilateur, ne sachant quel identifiant choisir, pourra vous renvoyer une erreur. Dans un tel cas, il faudra supprimer les directives « using namespace » et préfixer par le nom de l'espace lorsque ce sera néccessaire.

Appliquons cette deuxième solution. Notre programme devient :

```
#include <iostream>
#include <cstdint>
#include "machineCaractere.h"

using namespace std;

int main () {

     uint16_t lvCompteur = 0;
     CMachineCaractere lvMachine ("Elle n'est pas veuve");

     while (!lvMachine.marque ()) {
          lvCompteur++;
          lvMachine.avancer ();
          };
     cout << "La machine caractère contient " << lvCompteur << " caractères" << endl;
     return 0;
     }
```

5. Initialisons le ruban à partir de la ligne de commande

Nous avons évoqué, au chapitre 1, la possibilité de passer des paramètres via la ligne de commande. Nous allons donc initialiser la machine caractères à partir d'informations passées sur la ligne de commande à l'appel du programme.

Pour commencer cette phase, nous commençons par modifier la fonction « main » pour y intégrer les informations passées sur la ligne de commande. Le prototype de la fonction main modifiée est **obligatoirement** le suivant :

```
int main (int pvArgc, char* paArgv[])
```

Vous constatez l'apparition de deux paramètres alors que nous n'en avions aucun jusqu'à présent. Les noms donnés (ici **pvArgc** et **paArgv**[]) aux paramètres importe peu pourvu que leurs types soient respectivement **int** et **char***

- `int pvArgc`

  Le premier paramètre, ici nommé pvArgc est un nombre entier (int). Sa valeur correspond au nombre d'arguments + 1 passés sur la ligne de commande. Le premier argument sera le chemin du programme. Nous pourrons vérifier cette assertion en affichant cet élément dans notre programme.

- `char* paArgv[]`

  Le second paramètre est un tableau nommé « paArgv ». C'est un tableau car le nom du tableau est suivi de crochets « [] ». Il n'y a pas de nombre entre les crochets. A priori, nous ne connaissons donc pas la taille du tableau. Cependant, cette information nous est donné par le premier paramètre « pvArgc. Chacun des éléments de ce tableau est du type « char* ». Le suffixe « * » de ce type signifie qu'il s'agit d'une adresse. Cette adresse désigne un élément de type « char » (caractère). En fait, en C, et le C++ a hérité de ce principe, les chaînes de caractères sont définies par une adresse (un pointeur) désignant une liste de caractères terminée par un caractère de code 0. Chacun des éléments du tableau « `paArgv [ ]` » désigne donc une telle chaîne de caractères, chaque chaîne représentant un argument de la ligne de commande.

  Dans ce tableau, à l'indice 0 (`paArgv [0]`), le premier paramètre est le chemin complet du nom du programme. Les éléments suivants, donc de l'indice 1 (`paArgv [1]`) à l'indice `pvArgc` (`paArgV [pvArgc-1]`), nous trouverons les paramètres de la ligne de commande.

Afficher le nom du programme

Nous commençons par exploiter le premier paramètre et affichons le nom du programme. Cette ligne du code source devient une habitude :

```
cout << "Lancement du programme " << paArgv[0] << endl;
```

Ensuite, pour chaque argument de la ligne de commande, nous allons créer et initialiser une machine caractères avec cet argument. La boucle « `while` »précédemment écrite va donc être elle-même incluse dans une autre boucle parcourant les arguments. Nous retrouvons les principes énoncés dans l'annexe 1 : initialiser, traiter, avancer. Cependant, nous allons les utiliser dans un autre type de boucle : la boucle « for ». Celle-ci se présente sous la forme suivante :

```
for (initialiser ; condition ; avancer) {
     traiter
     };
```

Prenons un à un l'initialisation, la condition et l'avancement :

- Initialiser

  Nous allons parcourir le tableau des arguments de l'indice 1 à l'indice pvArgc-1. En effet, le compteur d'arguments comprend l'élément « Nom du programme » à l'indice 0. Si nous passons deux chaînes de caractères sur la ligne de commande, le programme y ajoutera celui à l'indice 0 et nous aurons « pvArgc = 3 ». Nos deux arguments chaînes seront respectivement aux indices 1 et 2.
  Notre initialisation consistera donc à initialiser une variable (appelons là lvNoArg) à 1. Nous n'oublions pas de déclarer son type (**int**) :

  ```
  int lvNoArg = 1
  ```

- Condition

La condition d'arrêt de la boucle vient d'être énoncée : le traitement sera réalisé tant que nous n'aurons pas dépassé les limites du tableau des arguments. C'est-à-dire tant que la variable « lvNoArg » sera inférieure à pvArgc :

```
lvNoArg < pvArgc
```

- Avancer

  Avancer consistera à passer à l'argument suivant. Il suffira donc d'incrémenter la variable lvNoArg :

```
lvNoArg++
```

- Traiter

  Le traitement sera celui déjà écrit consistant à initialiser la machine caractère avec l'argument courant et compter le nombre de caractères.

Dans ce contexte, notre boucle « **for** » devient :

```
for (int lvNoArg = 1 ; lvNoArg < pvArgc ; lvNoArg++) {
     traiter
     };
```

Et le programme :

```
#include <iostream>
#include <cstdint>
#include "machineCaractere.h"

using namespace std;

int main (int pvArgc, char* paArgv[]) {
     uint16_t lvCompteur;

     cout << "Lancement du programme " << paArgv[0] << endl;

     for (int lvNoArg = 1; lvNoArg < pvArgc; lvNoArg++) {

          string lvChaine (paArgv[lvNoArg]);
          CMachineCaractere lvMachine (lvChaine);
          cout << "La machine caractères est initialisée avec \"" << lvChaine << "\"" << endl;
          lvCompteur = 0;

          while (!lvMachine.marque ()) {

               lvCompteur++;
               lvMachine.avancer ();
               };

          cout << "La machine caractère contient " << lvCompteur << " caractères" << endl;
          };
     return 0;
     }
```

Et le résultat est :

```
~/cvs/compterCar/bin/Debug$ ./compterCar "Quelle belle épée" "Elle n'est pas veuve"
Lancement du programme ./compterCar
La machine caractères est initialisée avec "Quelle belle épée"
La machine caractère contient 18 caractères
La machine caractères est initialisée avec "Elle n'est pas veuve"
La machine caractère contient 20 caractères
```

6. Un minimum de documentation

Nous allons maintenant insérer des directives que le logiciel « doxygen » exploitera pour générer la documentation du programme. Nous avons cependant un petit problème ! Comment « mélanger » des instructions pour que le programme soit compilé et des instructions pour générer la documentation ?

La seule façon de faire est passer ces directives de documentation dans les commentaires du programme afin de ne pas perturber le compilateur. Ces directives seront superbement ignorées par le compilateur.

Ce premier problème résolu, nous en avons maintenant un deuxième qui pointe le bout de son nez. Nous ne souhaitons pas que tous les commentaires du programme soient à l'usage de doxygen. Nous devons donc différencier les parties documentant le programme des parties exploitées par doxygen.

Voyons tout d'abord le format des commentaires dans le code C++ puis les commentaires à l'usage de doxygen.

Commentaires de fin de ligne

Nous disposons d'un premier format de commentaire situé en fin de ligne. Ce type de commentaires débute par deux caractères « / » et se termine par la fin de ligne. Exemple :

```
uint16_t lvCompteur = 0; // Initialiser
```

Commentaires entre balises « /* » et « */ »

Le deuxième format de commentaire débute par « /* » et se termine par « */ ». Les sauts de ligne situés entre le début et la fin n'interrompent pas le commentaire qui peut donc se poursuivre sur plusieurs lignes.

```
uint16_t lvCompteur = 0; /* Nous sommes sur la première ligne du commentaire,
Celui- ci se poursuit sur cette deuxième ligne.
Puis sur cette troisième.
Mais le scribe de ce cours espère fortement ne pas devoir prolonger ce commentaire sur mille
lignes !
Voilà, terminons donc : */ int main () {
```

Directives de documentations

Comme pour les commentaires, nous disposons des deux mêmes types de directives de documentation. Celles sur une ligne, celles sur plusieurs lignes.

Directives de fin de ligne

Nous disposons d'un premier format situé en fin de ligne. Ce type de commentaires débute par trois caractères « / » et se termine par la fin de ligne. Exemple :

```
uint16_t lvCompteur = 0; /// Ceci est un commentaire à destination de doxygen.
```

Commentaires entre balises « /** » et « */ »

Un exemple valant mieux qu'un long discours, insérez les directives suivantes avant la fonction « main », lancer doxygen et voyez le résultat.

```
/** @brief int main (int pvArgc, char* paArgv[]) \n
 * Ce que fait la fonction main
 *
 * @param pvArgc : Une explication du paramètre pvArgc
 * @param paArgv : Une explication du paramètre paArgv
 * @return Une explication du résultat de la fonction main.
 */
int main (int pvArgc, char* paArgv[]) { ... }
```

7. Licence

Annexe 0 : Préfixes (conventions)

Classes

Les noms des classes sont préfixées par la lettre majuscule « C ». Exemple : class **C**MachineCaractere

Constantes

Les noms des constantes sont préfixées par la lettre minuscule « k ». Exemple const string& **k**prRuban. Suivent ensuite les même préfixes que ceux des variables.

Variables

Les noms des variables sont préfixées de 2 lettres minuscules :

- La première lettre est fonction de la portée de la variable.
  - « g » : variable globale (portée dans tout le programme). Exemple : uint16_t **g**vCompteur
  - « l » : variable locale (portée dans la paire d'accolades {} encadrante). Exemple : uint16_t **l**vCompteur
  - « p » : paramètre (portée dans la fonction courante). Exemple : lvCompteur = **p**vCompteur
  - « m » : donnée membre d'une classe. Ex : **m**vCompteur = kpvCompteur
- La deuxième lettre est fonction du type de la variable (adresse, référence, ...)
  - « v » : Le nom de la variable désigne directement une information. Exemple : uint16_t l**v**Compteur
  - « a » : Le nom de la variable désigne une adresse. Exemple : uint16_t* l**a**Compteur
  - « r » : Le nom de la variable désigne une référence. Exemple : uint16_t& l**r**Compteur
  - « i » : Le nom de la variable désigne un itérateur. Exemple : vector<int>::iterator l**i**Element

Annexe 1 : Schéma algorithmique de la boucle « Tant que »

Schéma

Le schéma algorithmique général d'une boucle « Tant que » est le suivant :

Initialiser
Tant que Condition
    Traiter
    Avancer

Exemple

Pour compter le nombre de lettres dans une phrase, nous transposons le schéma précédent de la manière suivante :

Initialiser le compteur à 0 `// Initialiser 1/2`
Pointer le début de la phrase `// Initialiser 2/2`
Tant que ***la fin de la phrase n'est pas atteinte*** `// Condition`
    Incrémenter le compteur `// Traiter`
    Passer au caractère suivant de la phrase `// Avancer`