# TD/TP 3

## C++ pratique 1
## Les techniques courantes

### Objectifs, méthodes

Enfin du code ! En cours différents « outils » du C++ ont été présentés comme un catalogue. Lors de ce TP vous apprendrez à les utiliser et à les combiner. Nous n'avons pas encore assez de bagage C++ à notre disposition pour attaquer l'implémentation des modèles objets étudiés sur les 2 1ers TD, en particulier le développement de classes sera couvert aux cours 5 et 6. Mais nous pouvons tout de suite être utilisateurs de classes :

- **`std::istream`** avec l'objet « saisies » cin
- **`std::ostream`** avec l'objet « affichage » cout
- **`std::string`** pour les chaines de caractères, avec ses nombreuses méthodes pratiques
- **`std::vector<>`** pour les collections : les « tableaux élastiques »

### Outils, plateformes, format

Dans ce TD/TP nous allons **travailler sur machine.** L'environnement de développement utilisé pour rédiger mes TPs est Code::Blocks 17.12 et son compilateur GCC (TDM-GCC 5.1.0) le tout sur Windows. Il est possible de faire les TPs avec un autre environnement à condition de disposer d'un compilateur C++ récent (C++14), en particulier *si vous développez sur Xcode (macOS) vous n'êtes pas obligés de passer en Windows/VirtualBox etc...* : vous pouvez rester en natif sur macOS ou votre Linux distro préférée. Pour les quelques manipulations commençant par « Sur Code::Blocks allez dans le menu Machin → Truc → Bidule ... » il faudra trouver l'équivalent. **La conséquence de cette portabilité est que nous nous limitons à des programmes en console.** Il y a bien sûr des bibliothèques d'interfaces graphiques portables (cross-platform) en C++ mais leur technicité ne permet pas de les utiliser en même temps que l'apprentissage du C++.

Les TD/TPs sont au format .pdf ce qui garantit que les documents gardent leur mise en page. Malheureusement le copier-coller de code depuis un pdf souffre d'une perte de formatage y compris les sauts de ligne. **Afin de faciliter l'utilisation des codes donnés en exemple dans les énoncés, vous trouverez en correspondance un lien vers un service d'hébergement de code avec le code source bien formaté, par exemple : http://cpp.sh/25p46**
Vous pouvez récupérer le code avec Ctrl-A puis Ctrl-C (tout copier), et Ctrl-V (coller) dans l'éditeur où vous travaillez (par exemple Code::Blocks). Le service permet également de compiler/tester directement de courts extraits de code, mais préférez un IDE complet comme Code::Blocks qui deviendra vite indispensable (projets multi-fichiers...)

- **Dans les exercices vous trouverez un texte d'introduction, en écriture droite.**

- ***La tâche principale à réaliser est décrite en italique souligné.***

- **Suivent souvent des explications sur la façon de réaliser cette tache, en écriture droite : si vous ne savez pas comment réaliser la tâche demandée dans le texte en italique souligné, lisez la suite !**

## 1. Testez votre environnement de développement

*Dans Code::Blocks ou dans votre environnement de développement préféré créez un nouveau projet console **en C++** bien sûr.*

Dans Code::Blocks menu File → New → Project... puis « Console application » puis C++. Sélectionner un répertoire (ne pas mettre tout en vrac sur le bureau !) et préciser un *Project title* (éviter les espaces, les accents, utiliser lettres, chiffres, et underscore_) valider pour le reste : boutons Next puis Finish. Ouvrir main.cpp, voir le code par défaut qui est proposé (vous devriez tout comprendre!) exécuter, ça doit marcher.

Vérifier dans menu Help → About... la dernière version est 17.12 et il est préférable d'avoir cette dernière version. Pour rappel, **sous Windows** le package Code::Blocks à installer se trouve au **4ème lien** de la rubrique Windows : sur http://www.codeblocks.org/downloads/26 c'est le fichier codeblocks-17.12mingw-setup.exe

*Le code par défaut proposé par Code::Blocks est le suivant (si vous n'avez pas CodeBlocks, recopiez ce code) vérifiez qu'il compile et s'exécute :*

http://cpp.sh/4wud

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

## 2. Namespace std

*Ça marche. P*ourtant si vous avez suivi le cours, quelque chose doit vous choquer...
En effet on annule l'intérêt d'encapsuler les identifiants de la bibliothèque standard dans un namespace en utilisant un « using namespace ». Imaginons que nous sommes dans une application scientifique de simulation. Un chercheur (pas spécialiste du C++) a *déclaré des variables au début du main*, 3 coefficients a, b et c et leurs futurs logarithmes alog, blog et clog :

```
double a, b, c, alog, blog, clog;
```

Tout se passe bien mais un jour il y a un bug qui nécessite l'intervention d'une collègue plus calée en C++. La collègue voit le using namespace au début mais pas les déclarations qui sont noyées dans le `main{...}` qui est devenu trop grand. Pour faire du debug elle est habituée à afficher des infos avec clog (character log, log = archive des événements) au lieu de cout (character output) parce qu'on peut par exemple facilement rediriger ces messages vers un fichier (ce qui évite d'encombrer la console). Elle *ajoute la ligne suivante vers la fin du main* :

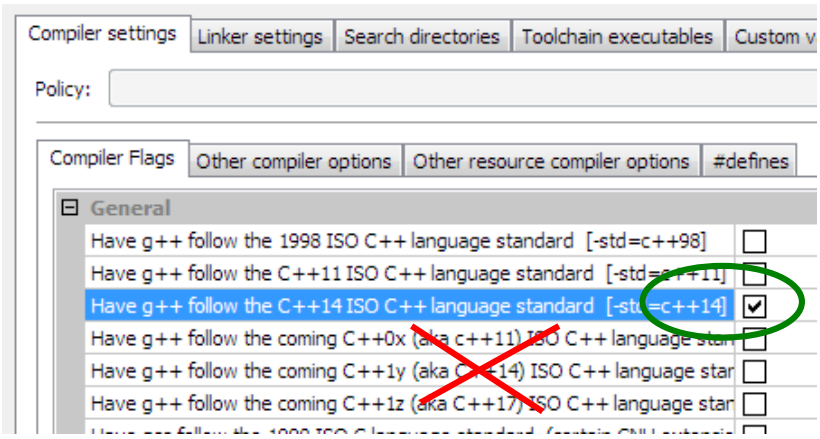
```
clog << "Point de controle 175 Ok" << endl;
```

*Faites les manips en vert. Compiler. Que se passe-t-il ? Le message d'erreur est-il sympathique ? Corriger le problème en supprimant le « using namespace ». Préciser pour chaque utilisation d'un objet de la bibliothèque standard (cout, endl et clog) en préfixant par std:: puis compiler / exécuter*

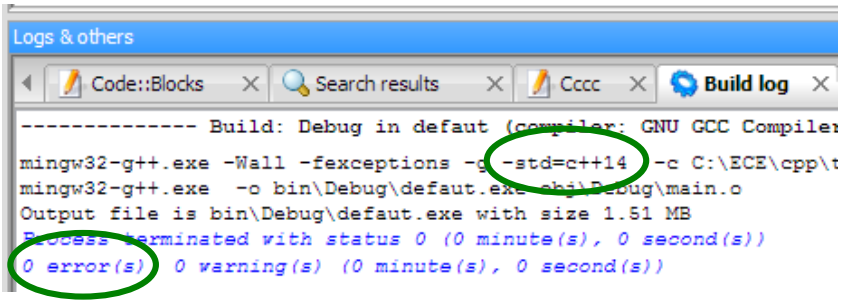
A partir de maintenant à chaque ouverture de projet votre 1er réflexe devrait être de supprimer la ligne using namespace proposée « par commodité » par Code::Blocks. On prend vite l'habitude de préfixer par std:: et c'est la façon normale (sérieuse) de faire du C++.

## 3. Configurer le compilateur pour utiliser C++14 sous Code::Blocks

Nous n'allons pas immédiatement étudier toutes les fonctionnalités les plus récentes du C++ mais nous voulons quand même utiliser une version « à jour » du langage et de ses bibliothèques. La version par défaut remonte au C++98, il est temps de mettre à la retraite cet ex-teenager !
*Sur Code::Blocks aller dans le menu Settings -> Compiler... et cocher l'option -std=c++14*




*Une fois confirmé vérifier que ça recompile bien (menu Build -> Rebuild ou Ctrl + F11 ) et que vous avez l'option voulue dans le « Build log » sous la fenêtre d'édition :*




*Si vous n'êtes pas sur Code::Blocks (xCode ou autre...) vérifiez les options de compilation et trouvez le paramètre équivalent si nécessaire.*

## 4. Nombres parfaits : affichages, vecteurs d'entiers, fonctions

Un collègue du département maths, chercheur en théorie des nombres, nous demande si nous pouvons lui faire un programme qui permet de **« trouver les nombres parfaits inférieurs à 10000 et les afficher sous la forme de la somme de leurs diviseurs stricts »**. Nous venons tout juste d'apprendre les bases du C++ pratique et nous avons hâte d'appliquer les outils vus en cours !

Un nombre parfait est un entier naturel qui est égal à la somme des ses diviseurs stricts (lui même exclu). Par exemple 6 est un nombre parfait, il est divisible par 1 et par 2 et par 3 et on a

```
6 = 1 + 2 + 3
```

Notre collègue souhaite donc une application qui affiche de cette façon tous les nombres parfaits trouvés jusqu'à 10000. Bien sûr on n'ira pas chercher la liste de ces nombres déjà connus pour les afficher de façon fixe avec juste std::cout, on veut écrire l'algorithme de recherche de ces nombres ! Et comme on est là pour apprendre les bases du C++ on va utiliser des **vecteurs d'entiers** pour contenir les listes de diviseurs, et on utilisera ce type pour communiquer avec les sous-programmes.

*Faire un nouveau projet console C++ `nombres_parfaits` et écrire ce programme !*
Consignes et indications page suivante...

Vous utiliserez au moins 3 sous programmes :

- recupDiviseurs qui prend en paramètre entrant un entier et qui retourne un vecteur d'entiers avec tous les diviseurs stricts
- sommeEntiers qui prend en paramètre entrant un vecteur d'entiers et qui retourne la somme
- afficherSommeEntiers qui prend en paramètre entrant un vecteur d'entiers et qui les affiche séparés par des '+'

A l'aide de ces 3 sous-programmes il sera ensuite facile d'écrire l'algorithme principal :

```
Pour chaque entier e de 1 à 10000
        appeler recupDiviseurs et récupérer la liste des diviseurs de cet entier
        appeler sommeEntiers avec cette liste, récupérer la somme
        Si la somme est égal à e alors
                afficher e '='
                appeler afficherSommeEntiers avec la liste des diviseurs
        FinSi
FinPour
```

A moins que ce 1er programme vous semble trivial il est fortement conseillé d'adopter une stratégie de développement incrémentale avec validation successives des sous-programmes par des tests. L'ordre de développement/validation n'est pas nécessairement l'ordre logique d'utilisation des sous-programmes. Je suggère de développer d'abord celui qui semble le plus facile sommeEntiers, puis celui qui « donne de la visibilité » afficherSommeEntiers, et enfin recupDiviseur. Pour un petit programme comme celui-ci il n'est pas nécessaire de faire un projet différent pour chaque étape : dans un 1er temps on peut se contenter de développer les sous-programmes directement au dessus du `main` et le `main` sert de zone de test (code temporaire d'appel) qui sera mis au propre plus tard.

### *4.1 Développer et valider sommeEntiers*

Ce sous-programme doit être tel que le main de test suivant …

http://cpp.sh/7nyf

```cpp
#include <iostream>
#include <vector>

... Ici développez sommeEntiers ...

int main()
{
    std::vector<int> testVec{7, 3, 5};

    std::cout << "7 + 3 + 5 = "
              << sommeEntiers(testVec) << std::endl;

    std::cout << "1 + 2 + 3 + 4 = "
              << sommeEntiers( {1, 2, 3, 4} ) << std::endl;

    return 0;
}
```

doit afficher

```
7 + 3 + 5 = 15
1 + 2 + 3 + 4 = 10
```

### 4.2 Développer et valider afficherSommeEntiers

Sur le même principe, ce sous-programme doit être tel que le main de test suivant …

```
std::vector<int> testVec{7, 3, 5};

afficherSommeEntiers(testVec);
std::cout << std::endl;

afficherSommeEntiers( {1, 2, 3, 4} );
std::cout << std::endl;
```

http://cpp.sh/2vocp

doit afficher

```
7 + 3 + 5
1 + 2 + 3 + 4
```

### 4.3 Développer et valider recupDiviseurs

Sur le même principe, ce sous-programme doit être tel que le main de test suivant …

```
std::vector<int> testVec;

testVec = recupDiviseurs(6);
afficherSommeEntiers( testVec );
std::cout << std::endl;

testVec = recupDiviseurs(60);
afficherSommeEntiers( testVec );
std::cout << std::endl;
```

http://cpp.sh/6g4kn

doit afficher

```
1 + 2 + 3
1 + 2 + 3 + 4 + 5 + 6 + 10 + 12 + 15 + 20 + 30
```

### 4.4 Intégration des sous-programmes validés séparément

Les 3 exercices précédents vous ont donné une (petite) idée d'une méthode de développement qu'on qualifie de « *test driven* » : on écrit les tests **avant** d'écrire le code à développer. Après cette phase de validation de « tests unitaires » il faut maintenant intégrer c'est à dire assembler les sous-programmes et faire des « tests d'intégration ».

Vider le `main` des codes de test et écrire à la place la traduction en C++ de l'algorithme principal indiqué page précédente. Tester. Vous pouvez vérifier que la sortie de votre application est conforme à ce que votre collègue matheux attend : Exemples nombres parfaits sur wikipedia.

### 4.5 Mise au propre, projet multi-fichiers

Cet exercice va se conclure mais on ne peut pas laisser le code dans cet état. Même si ça ne change rien « pour le client » on va pour notre satisfaction personnelle et l'efficacité d'une hypothétique ré-utilisation ultérieure mettre les sous-programmes là où ils doivent être : basculer les 3 sous-programmes dans un nouveau fichier source de projet, par exemple theorie_nombres.cpp, prototyper dans un nouveau fichier d'en-tête theorie_nombres.h, inclure ce dernier dans main.cpp.

### *4.6 Améliorer la lisibilité du code appelant*

Et si on devait montrer le code au collègue matheux qui ne sait pas programmer ? Pour lui qui est habitué à la pureté conceptuelle de l'algèbre, l'algorithme principal n'est pas super-clair, ça sent la graisse d'atelier. Il serait plus clair si il s'écrivait comme ça :

```
Pour chaque entier e de 1 à 10000
        Si estUnEntierParfait( e ) alors
                afficher e '='
                afficherSommeEntiers( recupDiviseurs( e ) )
        FinSi
FinPour
```

Ça nécessite d'*écrire un nouveau sous-programme estUnEntierParfait, qui appellera recupDiviseur et sommeEntiers et retournera un booléen ( type retour* **`bool`** *). Puis ré-écrire le main pour implémenter cet algorithme plus clair.*

On notera que pour les entiers parfaits le travail de déterminer les diviseurs sera fait 2 fois, une 1ère fois par estUnEntierParfait pour savoir si l'entier est parfait et une 2ème fois par le main pour afficher ces diviseurs. En l'occurrence ce n'est pas trop pénalisant vu la faible fréquence des nombres parfait (il y en a peu dans l'intervalle considéré). On peut considérer que cette nouvelle version est un progrès même si elle est un tout petit peu moins performante : le code appelant est nettement amélioré et, à résultat égal, c'est un aspect important. En programmation objet on veut fournir des « composants » faciles à utiliser : on se met autant que possible au service du code appelant même si ça implique de fournir un travail supplémentaire au niveau du code appelé.

## *5. Filtrage divisibles : saisies, affichages, vecteurs d'entiers, fonctions*

Le collègue du département maths, chercheur en théorie des nombres, impressionné par la vitesse de développement dont nous faisons preuve nous demande si nous pouvons lui faire un autre programme qui permet de **« filtrer successivement par divisibilité une liste de nombre entiers, initialement entrés au clavier »**. Le collègue ne nous fourni pas de CDC, mais il a rédigé une **cession typique d'utilisation du logiciel qu'il souhaite, voir page suivante avec en vert les saisies utilisateur**.

Après avoir saisie successivement au clavier des entiers (aucun « blindage » demandé) l'utilisateur termine la saisie de sa liste de nombre en entrant 0 (ou négatif). A partir de là on entre dans la phase de filtrages successifs : l'utilisateur entre un diviseur et seuls les nombre de la liste divisibles par ce diviseur seront conservés à l'étape suivante. Etc... jusqu'à ce que l'utilisateur termine la cession en entrant 0 (ou négatif).

*En suivant la démarche générale détaillée à l'exo 4, coder ce programme en C++ dans un nouveau projet console* `filtrage_divisibles`. Remarque si vous avez déjà eu le cours 4 : on fera des transmissions du vecteur **par valeur** (en entrée et en retour). On pourrait faire par référence...

Découpage en sous-programmes suggéré :

- Un sous-programme afficherAccueil qui affiche le message d'accueil
- Un sous-programme saisirEntiers qui retourne la liste(vecteur) des nombres saisis
- Un sous-programme afficherEntiers qui affiche la liste(vecteur) des nombres reçus
- Un sous-programme trouverDivisibles qui reçoit un vecteur d'entiers et un entier diviseur et qui retourne un nouveau vecteur « filtré » (uniquement les valeurs de la liste reçue qui sont divisibles par le diviseur).
- L'algo principal directement dans le main ou dans un sous-programme filtragesSuccessifs

```
Bienvenue dans l'application "filtrage divisibles"

Entrer 0 ou negatif pour terminer la saisie
Puis   0 ou negatif pour quitter l'application

Veuillez entrer des entiers
5
21
15
10
30
0

---------------------------------
Voici la liste de vos 5 entiers :

1 -> 5
2 -> 21
3 -> 15
4 -> 10
5 -> 30

Veuillez entrer un diviseur
5

---------------------------------
Voici la liste de vos 4 entiers :

1 -> 5
2 -> 15
3 -> 10
4 -> 30

Veuillez entrer un diviseur
3

---------------------------------
Voici la liste de vos 2 entiers :

1 -> 15
2 -> 30

Veuillez entrer un diviseur
2

---------------------------------
Voici la liste de vos 1 entiers :

1 -> 30

Veuillez entrer un diviseur
0

Process returned 0 (0x0)   execution time : 58.157 s
Press any key to continue.
```

## 6. Générateur de phrases : chaînes, vecteurs de chaînes

Sur le campus c'est maintenant une étudiante en linguistique qui prépare une thèse en « compositionnalité du langage et générativité gestaltiste » qui a besoin de nos talents : elle souhaite générer des séquences de phrases sur des patrons grammaticaux comme « adverbe GN verbe GN » où GN est un groupe nominal, en remplaçant chaque élément de cette structure par un représentant tiré au hasard dans une des listes correspondantes (liste d'adverbes, liste de noms, liste de verbes). Saurons-nous relever le défi ?

Par exemple, sur le patron grammatical « adverbe GN verbe GN »
- on pourrait tirer au hasard dans la liste des adverbes : **loin**
- on pourrait tirer au hasard dans la liste des GN : **le ministre**
- on pourrait tirer au hasard dans la liste des verbes : **intéresser**
- on pourrait tirer au hasard dans la liste des GN : **la passion**

Ce qui après accord au présent donnera la phrase « Loin le ministre intéresse la passion ».

La thésarde Oulipienne souhaite disposer d'échantillons de l'ordre de quelques milliers de phrases types de ce genre pour les soumettre à des ~~cobayes~~ volontaires et enregistrer leurs électroencéphalogrammes devant des phrases grammaticalement correctes mais vides de sens comparativement à des textes de poésie contemporaine et des discours de politique générale. Elle nous fournira des listes de mots (lexiques) par catégories, déjà au format « vecteur de strings et vecteur de vecteur de strings » car en effet elle touche aussi sa bille en langages de programmation.

*Téléchargez ce projet de départ*
https://fercoq.bitbucket.io/cpp/tdtp/tdtp3/phrase_generator_exo.zip ***pour Windows***
https://fercoq.bitbucket.io/cpp/tdtp/tdtp3/phrase_generator_exo_utf8.zip ***autres OS***
( source du lexique : 1500 mots les plus fréquents de la langue française, par catégorie )

*Vérifier que ça compile, essayer de comprendre ce que ça fait, où sont les informations lexicales, dans quel format. Essayer d'afficher d'autres mots précis. Vérifier que les accents passent. ( la console Windows est peu compatible avec la norme Unicode UTF-8 ☠☠☠ )*

Vous disposez dans la bibliothèque utilitaire util.h et util.cpp d'une fonction de tirage aléatoire uniforme telle qu'on aimerait l'avoir quand on débute (pas de srand bizarroïde, pas de max-min+1) juste un intervalle fermé **alea(min, max)**, on obtient un entier « aléatoire » entre min et max, bornes incluses. Le hasard en programmation est un sujet délicat parce que le modèle de conception des machines programmables est intrinsèquement déterministe. Des progrès ont été faits dans le domaine de l'exploitation de l'entropie numérique du système (device *dev/random* sur Linux... ). J'espérais vous conduire à explorer la doc pour mettre en œuvre les innovations récentes de la bibliothèque C++ en matière de hasard (rand/srand sont obsolètes) mais malheureusement la nouvelle norme est *outrageusement paramétrable et compliquée*.

Donc pour faire simple on a emballé le problème dans une fonction (*wrapper*) dans util.cpp, elle même emballée dans un namespace util:: pour éviter toute confusion avec std:: Vous écrirez util::alea pour l'utiliser, par exemple util::alea(1, 6) simule un lancé de dé. *Tester en affichant une 20$^{aine}$ de tirages successifs avec une boucle.* Attention pour tirer aléatoirement un indice dans un vecteur il faudra prévoir qu'un vecteur de 10 cases ( vecteur.size( ) retourne 10 ) a des cases numérotées de 0 à 9, donc la borne max sera vecteur.size( )-1

*Tester en affichant une 10$^{aine}$ de conjonctions au hasard avec une boucle*
*(les exemples donnés sont aléatoires donc vous n'aurez pas exactement le même résultat!)*

```
et mais car comment ou or or que ou quand
```

*Encapsuler le code de tirage aléatoire de conjonction en faisant un sous-programme faireConjonction de telle sorte que le résultat précédent puisse être obtenu avec le code de test suivant. Au lieu d'afficher elle même, la fonction faireConjonction* ***retourne*** *à l'appelant une des chaine du vecteur conjonctions, choisie au hasard. C'est l'appelant qui affiche :*

```
for (size_t i=0; i<10; ++i)
    std::cout << faireConjonction() << " ";
```

*Sur le même principe développer un sous-programme faireAdverbe qui retourne une chaine adverbe au hasard.* C'est un peu plus difficile, regardez bien dans adverbes.cpp, on a un vecteur de vecteur de string. Vous voudrez peut-être utiliser un alias de type VecVecStr par exemple (mais ce n'est pas du tout obligatoire). En tout cas il faudra 2 tirage aléatoire, un 1er tirage pour déterminer la catégorie d'adverbe (de manière, de quantité, de temps...) puis un 2ème tirage pour en sélectionner un seul dans la catégorie. *Tester, valider.*

*Un peu plus difficile encore, développer un sous-programme faireGN (groupe nominal) qui retourne une chaîne en juxtaposant un déterminant et un substantif* ***qui s'accordent en genre****.* On laissera de côté pour l'instant les problèmes de liaisons (le ami → l'ami, ta habitude → ton habitude) Gérer informatiquement une langue comme le français est notoirement difficile, on s'autorise quelques accidents ! On restera sur du singulier...

```
son monde
ta habitude
un instinct
ton éclat
votre fonction
aucun haut
le ami
chaque patron
leur choix
notre image
```

Nous voulons maintenant générer des verbes. Là on va choisir la facilité : on ne va garder que les verbes en « er » ( manger, bouger, tourner …) qui sont les verbes transitifs (sujet verbe COD !) du 1er groupe faciles à conjuguer. On se limite pour l'instant à la 3ème personne du singulier au présent : il suffit donc d'enlever le r (il mange, elle bouge, ça tourne). Ça ne marche pas à tous les coups, par exemple avec aller : « il alle » au lieu de « il va ». Bon, on s'en contentera. Le problème c'est que la liste des verbes (voir fichier source lexique_verbes.cpp) ne fait pas le tri entre différents genre de verbe.

*Ecrire une procédure void initialiserVerbesT1( ) qui remplit le vecteur global verbesT1 (déjà déclaré dans lexique_verbes.cpp, initialement vide) avec les verbes du vecteur verbes qui se terminent en «er», en supprimant le r final (pour obtenir la conjugaison attendue).* Après avoir appelé ce sous-programme dans le main, on pourra vérifier que le vecteur verbesT1 contient bien :

```
alle
trouve
donne
parle
passe
regarde
aime
```

**Etc...**

*Enfin développer un sous-programme faireVerbeT1 qui retourne une chaine verbeT1 au hasard.Tester, valider.*

*Ecrire un sous-programme fairePhrase1 qui génère une phrase « adverbe GN verbe GN » en appelant successivement les sous-programmes correspondant et en concaténant les résultats dans une chaîne (sans oublier les espaces pour séparer). Appeler ce sous programme 10 fois et afficher les résultats. On doit obtenir des choses dans ce style :*

```
quelquefois notre secret doute le fusil
là cette suite espére aucun titre
avant toute seconde installe tout instrument
assurément ton demain engage telle histoire
plutôt notre ville marque tout exemple
bien chaque mur étouffe notre jardin
soit quelque rayon trouve mon cheval
assurément ma dame rêve votre membre
loin leur commencement dure nul éclat
au-devant tout horizon sépare votre réponse
```

*Améliorer en ajoutant un point à la fin des phrases et une majuscule au début. Attention les adverbes commencent parfois par un « à » qui n'est pas dans l'intervalle ASCII habituel. Pour simplifier ne convertir que les minuscules non accentuées (entre 'a' et 'z')*

*Sur le même principe écrire un sous-programme fairePhrase2 qui génère une phrase de la forme « Adverbe GN verbe GN, conjonction GN verbe GN ! ». Appeler ce sous programme 10 fois et afficher les résultats. On doit obtenir des choses dans ce style :*

```
Derechef toute folie salue un désespoir, donc notre chien étudie tel officier !
En vérité nulle ligne dresse notre tache, quand ma folie inspire son livre !
Gratis ce empire étale chaque trésor, ni ce cours observe le jour !
Mieux toute valeur prouve tel début, si une salle aborde une vague !
Proche quelque fer efface nul mort, si leur ami éloigne sa voiture !
Premièrement sa mine veille votre bureau, or tel blanc affirme tout tour !
Non ce vol importe toute chute, et votre nouveau présente la chemise !
Impromptu nul commencement mérite nul oeil, or aucun hôtel joue son genou !
Ci notre mari évite nulle mode, parce que cette guerre change notre argent !
Précisément le honneur noye ma rose, si notre fille remplace telle douleur !
```

Finalement on s'approche d'une grammaire générative en utilisant un système de balises et en décrivant les formats de phrases de façon plus simple qu'en les codant : un sous-programme général fairePhrasePatron fabrique une phrase aléatoire en suivant le patron qu'on lui donne en paramètre, les patrons peuvent être regroupés en vecteurs de chaînes. *Cette suite est facultative.*

```
std::vector<std::string> patrons    {
    "<ADV> <GN> <VERBET1>e <GN>.",
    "<ADV> <GN> <VERBET1>e <GN>, <CONJ> <GN> <VERBET1>e <GN> !",
    "<GN> <VERBET1>ait <GN> ...",
    "<GN> <VERBET1>era <GN> ?"
};

for (size_t i=0; i<20; ++i)
{
    std::cout << fairePhrasePatron( patrons[util::alea(0, patrons.size()-1)] )
              << std::endl;
}
```

```
Ma poussière respirera mon patron ?
Tellement toute révolution discute nulle poitrine.
Au-delà telle morte dresse chaque chemin, ou ton bruit trompe son voile !
Aucun paysage penchera quelque sommeil ?
Davantage un vers juge quelque toit, comment chaque désert profite notre bras !
Joliment son genre salue sa fatigue.
Pas ta terreur enleve ton nombre.
Tel aspect menera aucun combat ?
Tout objet inquiétait cette famille ...
Parfois ma cuisine étudie ma montagne, puisque chaque année trace tel étranger !
Nul cas retirait la demande ...
Guère telle cour réveille notre dame, ni notre titre prie quelque loi !
Toute mine versera leur chose ?
Tout intérêt tirait sa faveur ...
Mon soin élevera tout vide ?
Nul mariage rassurait nul regard …
Votre peine demeurera telle paysanne ?
à demi tout hasard habite nul désir, comment telle habitude compose leur louve !
La vue rencontrera leur esprit ?
Céans chaque chair assiste quelque instant, ou ce signe forme une chasse !
```