# 1 Prise en main

Pour que vous puissiez effectuer vos travaux pratiques, vous devez sauvegarder sur votre compte informatique 4 fichiers de l'*Espace pédagogique*. Pour cela :

- Ouvrez une session sur un poste de travail.
- Créez un répertoire (dossier) que vous appellerez `IN101` dans lequel vous sauvegarderez les fichiers de vos séances de TP.
- Connectez vous à l'*Espace pédagogique*, sélectionnez le cours HLIN101, puis la rubrique `Documents pédagogiques`, puis `TP`, puis `Fichiers pour TP`.
- Recopiez le fichier `definitionsFonctions.cpp` dans votre répertoire local. Pour cela cliquez avec le bouton droit sur `definitionsFonctions.cpp`, sélectionnez « Enregistrer la cible du lien sous ... », sélectionnez le répertoire `IN101` que vous avez créé précédemment et `Enregistrer`.
- De la même façon, recopiez dans votre dossier `IN101` les 3 autres fichiers `listesEtTableaux.cpp`, `programmePrincipal.cpp` et `Makefile`.

Vérifiez que votre répertoire `IN101` contient les 4 fichiers :

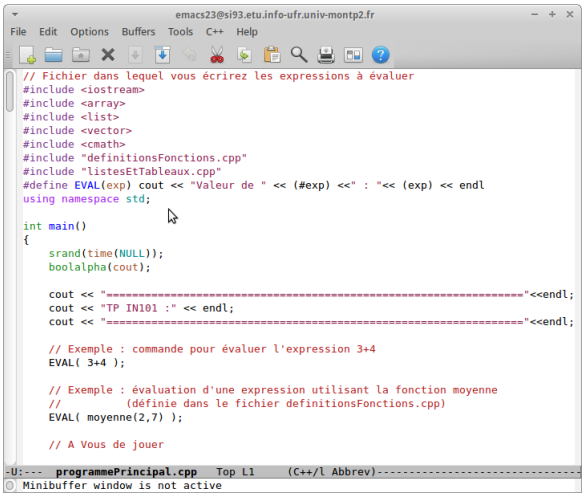
- `definitionsFonctions.cpp` dans lequel vous définirez les fonctions.
- `programmePrincipal.cpp` dans lequel vous saisirez les expressions à évaluer.
- `listesEtTableaux.cpp` qui contient la définition des types liste et tableau. Ce fichier sera nécessaire à l'exécution des fonctions opérant sur les listes. **Il ne faudra pas le modifier.**
- `Makefile` qui contient des directives pour la compilation de vos programmes. **Il ne faudra pas modifier ce fichier.**

Lors des séances de TP vous éditerez, compléterez, corrigerez les fichiers `definitionsFonctions.cpp` et `programmePrincipal.cpp` à l'aide de l'éditeur EMACS. Vous compilerez et exécuterez vos programmes à partir de la fenêtre TERMINAL (ou console). Vous pouvez ouvrir les fenêtres EMACS et TERMINAL à partir du « Menu des applications » puis « Accessoires ».

## 1.1 L'éditeur emacs

Les principales commandes d'EMACS que vous utiliserez sont :

- Dans le menu `File` :
  - « `Open File...` » pour ouvrir un fichier.
  - « `Save` » pour sauvegarder un fichier.
  - « `Split Window` » pour scinder la fenêtre en 2 sous-fenêtres.
  - « `Remove Split` » pour n'afficher qu'une des sous-fenêtres.
- Dans le menu `Buffers`, lorsque plusieurs fichiers sont ouverts (`definitionsFonctions.cpp` ou `programmePrincipal.cpp` dans notre cas) en cliquant sur le nom de l'un d'entre eux, son contenu s'affiche dans la fenêtre courante.




Ces commandes sont accessibles à partir des menus et à partir du clavier avec les racourcis :

| Commande | Racourci clavier |
|---|---|
| Ouvrir un fichier | <Ctrl-x><Ctrl-f> |
| Sauvegarder le fichier | <Ctrl-x><Ctrl-s> |
| Scinder la fenêtre | <Ctrl-x>2 (et 3) |
| N'afficher qu'une fenêtre | <Ctrl-x>1 |
| Aller à un numéro de ligne | <Alt-g>g |

## 1.2 La fenêtre TERMINAL

Les principales commandes que vous exécuterez dans la fenêtre TERMINAL :

| Commande | Effet |
|---|---|
| **cd IN101** | Se placer dans votre dossier `IN101` |
| **ls** | Lister les fichiers contenus dans le dossier courant |
| **make** | Compiler |
| **./in101** | Éxécuter votre programme `in101` |
| ↑ | Obtenir la dernière commande exécutée |

## 1.3 Évaluer des expressions dans le fichier `programmePrincipal.cpp`

Lisez et exécutez les consignes suivantes :

- Démarrez l'éditeur EMACS.
- Ouvrez une fenêtre TERMINAL et placez-vous dans votre dossier `IN101` (commande `cd IN101`).
- À partir d'EMACS ouvrez le fichier `programmePrincipal.cpp`. Pour cela sélectionnez `Open File` dans le menu `File`. Parcourez l'arborescence de vos fichiers pour vous placer dans votre répertoire `IN101` et sélectionnez `programmePrincipal.cpp`. Sélectionnez la fenêtre courante (menu « File » puis « Remove split » ou avec le clavier <Ctrl-x>1).
- **Edition :** ce fichier aprés quelques déclarations **que vous ne modifierez pas**, contient le programme principal (`main`). Ce programme est réduit à la ligne `EVAL( 3+4 );` qui signifie que le programme calcule et affiche la valeur de l'expression `3+4` et c'est tout ! (les lignes commençant par les caractères // sont des commentaires ; ceux-ci sont ignorés par le compilateur).
- **Compilation :** pour pouvoir exécuter ce programme, il faut au préalable le compiler. Pour cela, placez-vous dans la fenêtre TERMINAL et tapez la commande `make`. Les éventuelles erreurs de compilation sont affichées dans le TERMINAL. S'il ne contient pas d'erreur le programme est traduit en un programme exécutable qui dans notre cas s'appelle `in101`.
- **Exécution :** pour exécuter le programme `in101`, tapez la commande `./in101` dans le TERMINAL. Le résultat de l'exécution est affiché dans le TERMINAL : `Valeur de 3+4 : 7`.
- **A vous :** ajouter au programme principal la ligne `EVAL(true || false);` (avec le point virgule). Sauvegardez le fichier `programmePrincipal.cpp` (avec la souris menu `File`, puis `Save` ou avec le clavier <Ctrl-x><Ctrl-s>).
  Compilez et exécutez le programme modifié (plutôt que de retaper la commande `make` dans le TERMINAL vous pouvez utiliser la flèche ↑, de même pour `./in101`).
  Vous devez obtenir la valeur des expressions `3+4` et `true || false`.
- **Erreur :** lorsqu'une expression est mal formée, mal typée ou non évaluable, le compilateur signale l'erreur. Par exemple ajoutez au programme principal la ligne `EVAL( 3 * );` . Sauvegardez le fichier `programmePrincipal.cpp`, compilez–le. Le compilateur signale qu'il manque une expression avant la parenthèse fermante (`programmePrincipal.cpp:19:3:` signifie que l'erreur se situe à la ligne 19, colonne 3, du fichier `programmePrincipal.cpp`). Corrigez l'erreur en ajoutant un nombre entre * et ) dans `programmePrincipal.cpp`. Sauvegardez et compilez. Il ne doit plus y avoir d'erreur.
  Tapez à présent `EVAL( abs(2,3) );` . Sauvegardez, compilez. Le compilateur signale une erreur car il ne connaît pas de fonction `abs` s'appliquant à 2 entiers. Corrigez l'erreur en supprimant le $2^{ème}$ argument. Sauvegardez, compilez, exécutez. Il ne doit plus y avoir d'erreur.
  Tapez à présent `EVAL(2/(1-1));` . Sauvegardez, compilez. Le compilateur indique que l'expression contient une division par zéro. Transformez `EVAL(2/(1-1));` en commentaire : pour cela insérez // en début de ligne. Les commentaires sont ignorés par le compilateur. Sauvegardez, compilez. Il n'y a plus d'erreur (normalement).

**Évaluez les expressions suivantes, en essayant de deviner les réponses du programme :**

- **Des expressions de type entier** :
  Modifiez le programme pour obtenir les valeurs des 2 expressions `8/2` et `9/2`. Sauvegardez, compilez, exécutez. À quelle opération correspond / ?
  Modifiez le programme pour obtenir les valeurs des 2 expressions `8 % 4` et `9 % 4`. Sauvegardez, compilez, exécutez. À quelle opération correspond `%` ?
- **Des expressions de type réel** :
  Ajoutez au programme `EVAL( 8.5 + 1 );` puis `EVAL( 8.2 / 4.1 );` puis `EVAL( 8.2 / 2 );` Sauvegardez, compilez, exécutez. Vous remarquerez que / désigne ici la division réelle. Il désigne donc à la fois la division réelle et la division entière. Lorsque les 2 opérandes sont de type `int` c'est la division entière (le quotient) qui est appliquée, sinon la division réelle est appliquée.
  Modifiez le programme pour calculer la moyenne des 3 nombres 10.5, 8 et 11.25 .
  `pow` et `sqrt` correspondent aux opérations d'exponentiation et de racine carrée.
  Calculez les valeurs des expressions `pow(3,2)` puis `sqrt(25)` puis `sqrt(pow(12.3,2))`.
- **Des expressions de type booléen** :
  Calculez les valeurs des expressions `true` puis `2 > 3` puis `!(2 > 3)` puis `(2 > 3) && true` puis `(2 > 3) || true`. Les opérateurs « `non` », « `et` », « `ou` » s'écrivent respectivement `!`, `&&`, `||` en C/C++.
  Ajoutez au programme `EVAL( ((2/0)>1) || true);` . La compilation indique une division par 0. Commentez l'expression précédente pour supprimer l'erreur. Évaluez à présent l'expression `true || (2/0)>1`. Sauvegardez, compilez, exécutez. Il n'y a plus d'erreur car, l'évaluation *paresseuse* du « `ou` » n'évalue pas la division par zéro.
- **L'opérateur conditionnel** :
  Évaluez l'expression conditionnelle `2>3? 5 : 7` . Pour cela, écrivez `EVAL(2>3? 5 : 7);` sauvegardez, compilez, exécutez.
  Évaluez l'expression `sqrt(2)>1.4? 5 : 7` .
  Écrivez et évaluez une expression dont la valeur est l'entier 1 dans le cas où la moyenne de 11.4 et 8.5 est strictement supérieure à 10 et l'entier 2 dans le cas contraire.

## 1.4 Définir de nouvelles fonctions dans le fichier `definitionsFonctions.cpp`

Dans la suite des TP vous aurez à définir des nouvelles fonctions C/C++ correspondant à la traduction d'algorithmes. Vous saisirez ces définitions de fonctions dans le fichier `definitionsFonctions.cpp`.

- Ouvrez le fichier `definitionsFonctions.cpp`.
  - Saisissez le texte suivant qui correspond à la définition de la fonction calculant la moyenne de 2 nombres réels :

```
// Fonction moyenne
// Données: a : nombre réel, b : nombre réel
// Résultat: Nombreréel, la moyenne de a et b
float moyenne(float a, float b)
{
  return (a+b)/2
}
```

    EMACS reconnaît la syntaxe C/C++ : pour indenter automatiquement vos définitions tapez sur la touche tabulation à chaque ligne (ou après avoir sélectionné des lignes). Les 3 premières lignes (celles commençant par //) sont un commentaire donnant les spécifications de la fonction. Elles sont ignorées par le compilateur.
  - Sauvegardez votre fichier C/C++ (menu `File` puis `Save` ou bien <`Ctrl-x`><`Ctrl-s`>).

- Compilez le programme (commande `make` dans le TERMINAL). Le compilateur C/C++ signale une erreur dans le fichier `definitionsFonctions.cpp` : il manque un point virgule avant l'accolade fermante. Corrigez l'erreur, sauvegardez, compilez. Il n'y a plus d'erreur.
- Vous pouvez à présent utiliser la fonction `moyenne` dans les expressions.
  Ouvrez le fichier `programmePrincipal.cpp` (menu `Buffers`) et ajoutez-y par exemple la ligne `EVAL(moyenne(2.3 , 8));` . Sauvegardez `programmePrincipal.cpp`, compilez, exécutez. Le programme affiche la moyenne de 2.3 et 8.
  Le compilateur vérifie lors de l'application de la fonction `moyenne` si le nombre et le type des arguments sont corrects. Par exemple testez `EVAL( moyenne(2.3,8,7.7) );` . Le compilateur signale une erreur dans le nombre d'arguments. Supprimez cette erreur en commentant cette ligne (en ajoutant // en début de ligne).
  On peut utiliser la fonction `moyenne` dans une expression conditionnelle. Par exemple, ajoutez au programme principal la ligne `EVAL( moyenne(11.4,8.5)>10 ? 1 : 2 );` . Sauvegardez, compilez, exécutez.

— Une autre fonction à définir.
- Définissez dans `definitionsFonctions.cpp` la fonction `max3` qui étant donnés 3 nombres réels donne comme résultat le plus grand de ces nombres.
- Combien de données admet la fonction `max3` ? Quels sont leurs types ? Quel est le type du résultat ? Les réponses à ces questions déterminent la signature de la fonction `max3` et donc la première ligne de sa définition.
- Pour écrire le corps de la fonction vous pouvez utiliser la fonction prédéfinie `max` qui calcule le maximum de 2 nombres réels.
- Sauvegardez votre fichier, compilez, jusqu'à ce qu'il n'y ait plus d'erreur.
- Testez votre définition en ajoutant à `programmePrincipal.cpp` `EVAL(max3(3,9,2.5));` pour obtenir le maximum parmi 3, 9 et 2.5. Sauvegardez votre fichier, compilez, exécutez.

# 2 Expressions conditionnelles, opérations booléennes

1. Définissez la fonction `multiple` qui étant donné 2 nombres entiers `x` et `y` a pour résultat le booléen `true` si `x` est un multiple de `y` et `false` sinon.

   **Rappel** : un entier `x` est un multiple d'un entier `y` si il existe un entier `k` tel que $x = k \times y$ .

   Vous aurez très certainement besoin de l'opération `%` pour définir `multiple` ainsi que de l'opérateur égalité qui en C/C++ est noté `==` .

   Écrivez la définition de `multiple`, sauvegardez, évaluez jusqu'à ce que sa syntaxe et son type soient corrects. Testez votre définition. Votre fon ction `multiple` doit être définie pour tous les couples d'entiers, en particulier lorsque l'un d'entre eux ou les 2 sont nuls (attention à la division par 0). Par exemple les valeurs respectives de `multiple(2,0)`, `multiple(0,2)` et `multiple(0,0)` doivent être `false`, `true` et `true`. Si les réponses fournies par C/C++ ne correspondent pas à ces valeurs ou si l'évaluation génère une erreur modifiez votre définition. Sauvegardez, évaluez la définition, testez ...

2. Écrivez une fonction C/C++ pour chacune des fonctions suivantes. Vous testerez chacune de vos fonctions avec différents arguments.

   **triangleEqui** : $\mathbb{R}^{+*} \times \mathbb{R}^{+*} \times \mathbb{R}^{+*} \longrightarrow$ *Bool*

   $$a, b, c \longmapsto \begin{cases} \texttt{true} \text{ si a, b et c sont les longueurs des côtés} \\ \text{d'un triangle équilatéral,} \\ \texttt{false} \text{ sinon} \end{cases}$$

**triangle** : $\mathbb{R}^{+*} \times \mathbb{R}^{+*} \times \mathbb{R}^{+*} \longrightarrow Bool$

$$a, b, c \longmapsto \begin{cases} \texttt{true} \text{ si a, b et c sont les longueurs des côtés} \\ \text{d'un triangle (chaque longueur est inférieure} \\ \text{à la somme des 2 autres), } \texttt{false} \text{ sinon} \end{cases}$$

**triangleRect** : $\mathbb{R}^{+*} \times \mathbb{R}^{+*} \times \mathbb{R}^{+*} \longrightarrow Bool$

$$a, b, c \longmapsto \begin{cases} \texttt{true} \text{ si } a, b \text{ et } c \text{ sont les longueurs des côtés} \\ \text{d'un triangle rectangle, } \texttt{false} \text{ sinon} \end{cases}$$

**triangleIso** : $\mathbb{R}^{+*} \times \mathbb{R}^{+*} \times \mathbb{R}^{+*} \longrightarrow Bool$

$$a, b, c \longmapsto \begin{cases} \texttt{true} \text{ si } a, b \text{ et } c \text{ sont les longueurs des côtés} \\ \text{d'un triangle isocèle (c'est un triangle et 2 de} \\ \text{ses côtés ont même longueur),} \\ \texttt{false} \text{ sinon} \end{cases}$$

3. En TD vous avez écrit un algorithme pour la fonction booléenne :

   **ouExcl** : $Bool \times Bool \longrightarrow Bool$

   $a, b \longmapsto \{$ `true` si parmi $a$ et $b$, un seul des 2 est `true`, `false` sinon

   Traduisez cet algorithme en fonction C/C++.

4. Utilisez vos fonctions `multiple` et `ouExcl` pour définir en C/C++ la fonction :

   **mul2ou3** : $\mathbb{Z} \longrightarrow Bool$

   $n \longmapsto \{$ `true` si $n$ est multiple de 2 ou de trois mais pas de 6, `false` sinon

5. Écrivez une fonction `memeDizaine` vérifiant si 2 nombres entiers sont dans la même dizaine. C'est le cas par exemple de 38 et 33 ou encore de 922 et 929. Par contre 128 et 325 ne sont pas dans la même dizaine (128 appartient à la $12^{ème}$ dizaine alors que 325 appartient à la $32^{ème}$). Pensez à utiliser la division entière.

6. Écrivez une fonction `memeParite` vérifiant si 2 nombres entiers ont même parité, c'est à dire si ils sont tous les deux pairs ou alors tous les deux impairs.

7. Soit un jeu de roulette simple :

   > On mise une somme sur un numéro. Le gain est calculé en fonction du numéro sorti selon la règle suivante :
   > — Si les numéros sortis et joués sont identiques, on remporte 20 fois la mise.
   > — Si les numéros sortis et joués sont dans la même dizaine, on remporte 5 fois la mise.
   > — Si les numéros sortis et joués sont de même parité, on remporte 2 fois la mise.
   > — Sinon on ne gagne rien ?

   Par exemple si on mise 10 € sur le numéro 12 et que le numéro sorti est 18, on gagne 5 fois la mise, donc 50 €, car les 2 numéros sont dans la même dizaine.

   Complétez la fonction ci-dessous calculant le gain au jeu de roulette :

```
int gainRoulette(int mise, int numJoue, int numSorti)
{
    return ...
```

   Testez votre procédure en évaluant `gainRoulette(100,675,500)`, `gainRoulette(100,675,672)` et `gainRoulette(100,675,675)`.

# 3 Récursivité sur les nombres

1. En TD vous avez écrit un algorithme récursif testant si un entier positif est un entier pair sans utiliser de division. Voici une traduction de cet algorithme en une fonction C/C++ :

```
bool estPair(int n)
{
    return
        n==0 ? true :
        !estPair(n-1);
}
```

Testez cette fonction en évaluant `estPair(9)` puis `estPair(56)`.

Évaluez à présent `estPair(-2)`. Un message d'erreur `<<Erreur de segmentation>>` apparaît. Il signifie que l'exécution du programme a été interrompue car l'évaluation de `estPair(-2)` ne termine pas, la suite des valeurs de l'argument (-2, -3, -4, ...) étant infinie. Modifiez la définition de la fonction `estPair` pour que le cas des entiers négatifs soit correctement traité.

2. Écrivez les fonctions suivantes dont les paramètres sont 2 entiers $a$ et $b$ :
   (a) `existeMul11` vérifie si il existe un multiple de 11 dans l'intervalle $[a, b]$. Testez votre fonction : les valeurs de `existeMul11(2,10)`, `existeMul11(22,22)` et `existeMul11(22,2)` doivent être respectivement `false, true, false`.
   (b) `maxMul11` qui calcule le plus grand entier multiple de 11 de l'intervalle $[a, b]$. Dans le cas où l'intervalle $[a, b]$ ne contient pas de multiple de 11, le résultat de `maxMul11` sera -1. Par exemple les valeurs respectives de `maxMul11(3,43)` et `maxMul11(3,4)` sont 33 et -1.
   (c) `nbMul11` calcule le nombre d'entiers multiples de 11 dans l'intervalle $[a, b]$. Par exemple les valeurs respectives de `nbMul11(3,43)` et `nbMul11(3,4)` sont 3 et 0.
   (d) `somMul11` calcule la somme des entiers multiples de 11 appartenant à l'intervalle $[a, b]$. Par exemple les valeurs respectives de `somMul11(3,54)` et `somMul11(3,4)` sont 110 et 0.
3. Soit $n$ un entier positif. Dans l'écriture décimale de $n$, on appelle :
   — *chiffre de rang 1*, le chiffre des unités ;
   — *chiffre de rang 2*, le chiffre des dizaines ;
   — *chiffre de rang k*, le $k^{ème}$ chiffre en lisant l'écriture décimale de $n$ de la droite vers la gauche. Si ce chiffre n'existe pas, le *chiffre de rang k* de $n$ est 0.

   Par exemple pour l'entier 3249 : le chiffre de rang 1 est 9, le chiffre de rang 2 est 4, le chiffre de rang 4 est 3 et le chiffre de rang 12 est 0.

   Quelle expression permet d'obtenir le chiffre des unités de 3249 ? Comment obtenir le chiffre de rang 2 de 3249 ? (pensez à composer les opérations `%` et /).
   (a) Écrivez une fonction `nbChifDec` calculant le nombre de chiffres de l'écriture décimale d'un entier $n$ (`nbChifDec(3279)` vaut 4).
   (b) Écrivez la fonction `chifRang` qui étant donné un entier positif $n$ et un entier strictement positif $k$ calcule le chiffre de rang $k$ de l'écriture décimale de $n$.
   (c) (***) Écrivez une fonction `somChif` qui étant donné un entier positif $n$ calcule la somme de ses chiffres. Par exemple `somChif(3249)` vaut 18.
   (d) (***) La *racine numérique* d'un nombre est un nombre à 1 chiffre. Il s'obtient en calculant la somme des chiffres du nombre jusqu'à ce que le nombre obtenu n'ait qu'un chiffre. Par exemple la racine numérique de 3249 est 9. Elle s'obtient en calculant 3+2+4+9=18 puis 1+8=9. Écrivez une fonction calculant la racine numérique d'un entier naturel.
   (e) (****) Écrivez une fonction `invChif` qui étant donné un entier positif $n$ calcule le nombre dont l'écriture décimale est celle de $n$ lue en sens inverse. Par exemple `invChif(3249)` vaut 9423. Vous aurez besoin de la fonction puissance `pow`.
4. (***) Écrivez la fonction `estCarre`, qui vérifie si un entier est un carré (d'entier). Par exemple 4, 9, 16, 25, 36 sont des carrés.

Son écriture nécessite l'utilisation d'une autre fonction qu'il vous faudra définir et écrire. Par exemple une fonction `existeRac` à 2 paramètres $n$ et $k$ testant si l'entier $n$ admet une racine entière supérieure ou égale à l'entier $k$. Définissez alors la fonction `estCarre`. La définition de `existeRac` devra précéder celle de `estCarre` dans la fenêtre éditeur.

# 4 Listes

## 4.1 Fonctions et opérateurs de base sur les listes

Ajoutez au fichier `programmePrincipal.cpp` la ligne

```
const list<int> exli={1,33,67,12,1,22};
```

qui a pour effet de définir la constante `exli` dont la valeur est la liste d'entiers `(1 33 67 12 1 22)`. Cette définition simplifiera l'écriture des expressions que vous aurez à évaluer.

1. Après la définition de `exli` ajoutez la ligne `EVAL(exli);` pour obtenir la valeur de `exli`.
   Modifiez le programme principal pour obtenir les valeurs des expressions suivantes. Lorsque l'évaluation d'une expression provoque une erreur, vous commenterez la ligne correspondante pour pouvoir évaluer les autres expressions.

   | | | | |
   |---|---|---|---|
   | tete(exli) | queue(exli) | tete(queue(exli)) | tete(tete(exli)) |
   | queue(queue(exli)) | cons(11,liVide<int>()) | cons(11, exli) | cons(1.1, exli) |

2. Trouvez et évaluez une expression dont la valeur est :
   — La somme des 2 premiers éléments de `exli`.
   — La liste `exli` dans laquelle les 2 premiers éléments ont été remplacés par leur somme.
   — La liste `exli` sans son deuxième élément.
3. L'expression qui permet d'obtenir la liste `(1 2 3)` est `cons(1,cons(2,cons(3,liVide<int>())))`. Pour alléger cette écriture vous pourrez utiliser la fonction `liste` qui étant donné une séquence non vide d'entiers séparés par des points virgules encadrée par des accolades, a pour résultat la liste d'entiers correspondante. Ainsi pour obtenir la liste d'entiers `( 1 2 3 )` on pourra écrire `liste({1, 2, 3})` . Attention cette fonction n'est définie que pour des listes d'entiers.

## 4.2 Définir de nouvelles fonctions sur les listes d'entiers

Ajoutez au fichier `definitionsFonctions.cpp` les définitions des fonctions suivantes :

1. Écrivez la fonction `longLi` qui calcule la longueur d'une liste (son nombre d'éléments) d'entiers. Calculez la longueur de `exli`.
2. Écrivez la fonction `minLi` qui calcule le plus petit élément d'une liste non vide d'entiers. Testez votre fonction en évaluant `minLi(exli)`.
3. Écrivez la fonction `oterLi` qui a pour paramètres un entier $n$ et une liste d'entiers $li$ et dont le résultat est la liste $li$ sans le premier élément de valeur $n$. Dans le cas où $li$ ne contient pas d'élément de valeur $n$, le résultat est la liste $li$. Par exemple les résultats de `oterLi(1,exli)`, `oterLi(67,exli)` et `oterLi(9,exli)` sont les listes `(33 67 12 1 22)`, `(1 33 12 1 22)` et `(1 33 67 12 1 22)`.
4. Écrivez la fonction `estTriee` qui vérifie si une liste d'entiers est triée par ordre croissant, c'est à dire si tout élément de la liste est inférieur ou égal à tous les éléments situés après lui dans la liste. La liste vide, les listes composées d'un seul élément et `(3 3 12 13 16)` sont des exemples de listes triées. `exli` n'est pas une liste triée. Testez votre fonction en évaluant les expressions : `estTriee(exli)`, `estTriee(liste({2}))`, `estTriee(liste({5,1}))` et `estTriee(liste({3, 3, 12, 13, 16}))`.

5. (***) Il s'agit d'écrire une fonction `triListe` qui étant donné une liste d'entiers *li* calcule la liste constituée des éléments de *li* rangés dans l'ordre croissant. Par exemple le résultat de `triListe(exli)` est la liste `(1 1 12 22 33 67)`.

   Il existe plusieurs algorithmes de tri. On peut définir l'un d'entre eux de façon récursive :
   — **Cas de base** : quand *li* est la liste vide, que vaut `triListe(li)` ?
   — **Équation de récurrence** : quand *li* n'est pas vide, pour trouver la valeur de `triListe(li)`, vous chercherez à définir sa tête de liste et sa queue de liste (utilisez les fonctions précédemment définies).

   Écrivez la définition de `triListe` et testez-la.

6. Écrivez la fonction `lgPrefEg` qui étant donné une liste non vide d'entier *li*, calcule la longueur du plus long préfixe de *li* constitué d'éléments tous égaux.

   *Exemples* : les longueurs des plus longs préfixes des listes `(2 7 7 2)` et `(2 2 2 7 2)` sont respectivement 1 et 3. Vérifiez si vous trouvez ces valeurs lorsque vous évaluez les expressions `lgPrefEg(liste({2, 7, 7, 2})` et `lgPrefEg(liste({2, 2, 2, 7, 2}))`.

7. Écrivez la fonction `supPrefEg` qui étant donné *li*, une liste d'entiers non vide, supprime le 1er élément de *li*, ainsi que le $2^{ème}$ s'il est égal au 1er, ... jusqu'au premier élément différent du 1er de *li*. Autrement dit le résultat est le plus long suffixe de *li* dont le 1er élément, s'il existe, est différent de la tête de *li*.

   *Exemples* :

| Expression | Valeur |
|---|---|
| `supPrefEg(liste({2, 7, 7, 2}))` | la liste `(7 7 2)` |
| `supPrefEg(liste({2, 2, 7, 2}))` | la liste `(7 2)` |
| `supPrefEg(liste({2, 2, 2}))` | la liste vide |
| `supPrefEg(liste({2, 2, 2, 7, 3, 7, 7}))` | la liste `(7 3 7 7)` |

8. Écrivez la fonction `supRepet` qui supprime les répétitions d'éléments consécutifs égaux d'une liste d'entiers.

   *Exemples* :

| Expression | Valeur |
|---|---|
| `supRepet(liste({2,7,7,7,2})` | la liste `(2 7 2)` |
| `supRepet({2,2,7,7,2})` | la liste `(2 7 2)` |
| `supRepet({2,7,7,7,2})` | `(2 7 2)` |
| `supRepet({2,2})` | `(2)` |
| `supRepet({2,7,3,7})` | `(2 7 3 7)` |
| `supRepet(liVide<int>())` | `()` |

9. (***) On peut coder une liste en remplaçant chaque séquence maximale d'éléments consécutifs égaux par 2 éléments : la longueur de la séquence et la valeur de l'élément répété.

   Par exemple la liste `(7 7 7 7 8 8 7 7 7)` est constituée de quatre 7, suivis de deux 8, suivis de trois 7. Le codage de cette liste est la liste `(4 7 2 8 3 7)`. Le codage de la liste `(5 6 6 6 6 5)` est la liste `(1 5 4 6 1 5)`.

   Écrivez une fonction `codeLi` calculant selon ce principe le codage d'une liste d'entiers. Vous aurez besoin des fonctions `lgPrefEg` et `supPrefEg`.

10. (*****) Écrivez la fonction `decodeLi`, la fonction réciproque de la fonction `codeLi` : étant donné une liste d'entiers de longueur paire *li*, `decodeLi` calcule la liste *lidc* telle que `codeLi(lidc)` vaut *li*.

    *Exemples* :

| Expression | Valeur |
|---|---|
| `decodeLi(liste({3,7,2,1,1,2}))` | `(7 7 7 1 1 2)` |
| `decodeLi(liste({1,1,2,2,3,3}))` | `(1 2 2 3 3 3)` |