

**TP 9 : compilation séparée, graphiques****Exercice 1** (Preliminaires : compilation séparée).

- (1) Télécharger et extraire l'archive `Semaine9.zip` dans votre répertoire `Info111`, en suivant les instructions usuelles (voir par exemple la fiche de TP 2).
- (2) Consulter le contenu des fichiers suivants : `factorielle.h`, `factorielle.cpp`, `factorielle-exemple.cpp`.
- (3) Pour compiler le programme entier, il faut d'abord compiler chacun des bouts de programme (les fichiers `.cpp`). Pour cela on utilise l'option `-c` :

```
g++ -c factorielle.cpp
g++ -c factorielle-exemple.cpp
```

Ceci nous a créé deux fichiers, `factorielle.o` et `factorielle-exemple.o` qui sont des bouts de programme. Vérifier avec `ls` que ces fichiers ont bien été créés.

Ensuite on les combine ces deux bouts de programme de la façon suivante :

```
g++ factorielle.o factorielle-exemple.o -o factorielle-exemple
```

Vérifier avec `ls` que cette commande crée bien un exécutable `factorielle-exemple`.

- (4) Exécuter le programme `factorielle-exemple`.
- (5) Créer un programme `factorielle-test` en reprenant les mêmes étapes, mais en remplaçant le programme principal `factorielle-exemple.cpp` par `factorielle-test.cpp`, puis testez ce nouveau programme.
- (6) Une autre solution aurait été de remplacer les trois lignes de codes en console précédentes par la seule ligne suivante :

```
g++ factorielle.cpp factorielle-exemple.cpp -o factorielle1
```

Supprimer les fichiers `factorielle.o`, `factorielle-exemple.o` et `factorielle-exemple` de votre dossier avec `rm`. Tester alors l'instruction précédente. Quel(s) fichier(s) ont été créés ? En déduire quelles sont les différences avec la méthode précédente ? En fonction des moments vous pourrez être amené à faire l'une ou l'autre, notamment dans le projet.

**Exercice 2** (Preliminaires : compilation séparée).

- (1) Ouvrir le fichier `fibonacci.cpp` et regarder son contenu. Remarquer que la fonction `main` mélange deux actions de nature très différente : lancer les tests de la fonction `fibonacci`, et utiliser cette fonction pour interagir avec l'utilisateur. Ceci n'est pas très satisfaisant.
- (2) Réorganiser le fichier `fibonacci.cpp` en plusieurs fichiers en suivant le modèle de l'exercice précédent. Il y aura donc quatre fichiers : `fibonacci.h`, `fibonacci.cpp`, `fibonacci-test.cpp` et `fibonacci-exemple.cpp`.  
Faites attention à ne pas dupliquer de code.
- (3) Compiler chacun des deux programmes (test et exemple) grâce à la compilation séparée, les exécuter et vérifier que tout fonctionne correctement.

**Exercice 3** (Premiers graphiques avec MLV).

Attention : dans les salles de TP, la bibliothèque MLV n'est installée que sous Linux. Pour ceux qui souhaitent travailler sur leur machine personnelle, voir les instructions d'installation<sup>1</sup> de la documentation de MLV. À noter que pour utiliser MLV sur CodeBlocks ou tout autre IDE, il faut configurer l'IDE. Plus encore que d'habitude, on fera tout en ligne de commande.

- (1) Ouvrir les fichiers `mlv-exemple.cpp` et `MLV.h`. Consulter le contenu du premier.
- (2) Compiler ce programme depuis le terminal avec :

```
g++ mlv-exemple.cpp -std=c++11 -lMLV -o mlv-exemple
```

puis le lancer avec :

```
./mlv-exemple
```

- (3) Refaire l'exercice 2 du TD en complétant le programme fourni `graphisme-premier-dessin.cpp`. Implantez chacun des items en vérifiant à chaque fois le résultat. N'hésitez pas à changer la valeur de la variable `delai` pour voir le résultat s'afficher plus longtemps.

**Exercice 4** (Souris et clavier).

- (1) Pour vous donner une idée des primitives graphiques de MLV, consulter l'exemple fourni `mlv-exemple3.cpp`.
- (2) Implanter l'exercice 4 du TD.

**Exercice ♣ 5** (Images).

Le but de l'exercice est d'implanter un programme qui lit un fichier au format PPM (*Portable Pix Map*, voir [http://fr.wikipedia.org/wiki/Portable\\_pixmap](http://fr.wikipedia.org/wiki/Portable_pixmap)), et l'affiche à l'écran avec la bibliothèque MLV. L'en-tête de fonction et les programmes à compléter sont donnés dans le fichier `affiche-ppm.cpp`. Les fichiers `pbmlib.ppm` et `blackbuck.ppm` fournis serviront pour les tests. La solution proposée utilise des tableaux 3D, ce qui nécessite une compréhension fine des tableaux.

**Indications :** Avec MLV, on peut créer une nouvelle couleur avec la fonction suivante :

```
/** construit une nouvelle couleur
 * @param red: un entier entre 0 et 255 spécifiant le niveau de rouge
 * @param green: un entier entre 0 et 255 spécifiant le niveau de vert
 * @param blue: un entier entre 0 et 255 spécifiant le niveau de bleu
 * @param alpha: un entier entre 0 et 255 spécifiant la transparence
 */
color_t rgba(UInt8 red, UInt8 green, UInt8 blue, int alpha);
```

Par exemple, `color::rgba(255,0,200,128)` donnera un violet tirant sur le rouge, à moitié transparent.

1. <http://www-igm.univ-mlv.fr/~boussica/mlv/api/French/html/installation.html>

**Exercice ♣ 6** (Jeu du Yams).

Reprendre le jeu du Yams du TP 6 en rajoutant une petite interface graphique.

On affichera les dés (soit avec du texte, soit avec des points) dans la fenêtre. Retourner voir `mlv-exemple3.cpp` pour des fonctions rapides. L'utilisateur pourra cliquer sur les dés qu'il veut combiner pour former une figure. (utiliser par exemple `wait_mouse()` pour cliquer sur les dés, et `wait_keyboard()` pour valider.) Le nombre de points sera ensuite affiché.

À vous de concevoir les fonctions à introduire pour décomposer le problème.