TP 10 : compilation séparée, graphiques

Comme d'habitude, utilisez la commande `info-111` pour télécharger les fichiers du TP.

Exercice 1 (Préliminaires : compilation séparée)**.**

(1) Consulter le contenu des fichiers suivants : `factorielle.h`, `factorielle.cpp`, `factorielle-exemple.cpp`.

(2) 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`.

(3) Exécuter le programme `factorielle-exemple`.

(4) Consulter le fichier `factorielle-test.cpp`. Créer un exécutable `factorielle-test` en adaptant les étapes ci-dessus, puis testez ce nouveau programme.

(5) Une autre méthode pour construire `factorielle-exemple` est de remplacer les trois commandes de la question (2) par la seule commande :

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

Supprimer les fichiers `factorielle.o`, `factorielle-exemple.o` et `factorielle-exemple` de votre dossier avec `rm`. Tester alors la commande 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 (Préliminaires : 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) Quels exécutables allez-vous construire ? Pour chaque exécutable, quels fichiers allez-vous combiner pour l'obtenir ? Vérifier pour chaque exécutable que chaque fonction utilisée (dont la fonction `main`) est définie une et une seule fois dans l'ensemble de fichiers correspondant.

(4) Compiler chacun des deux programmes (test et exemple) grâce à la compilation séparée, les exécuter et vérifier que tout fonctionne correctement. En cas d'erreur, lire le message d'erreur puis comparer attentivement vos fichiers et commandes avec ceux donnés pour la fonction factorielle (si vous avez votre feuille de TD sous la main, consulter l'exercice 1 vous permet de visualiser facilement tous les fichiers factorielle).

Exercice 3 (Premiers graphiques avec SFML)**.**
Attention : vous ne pourrez pas utiliser le serveur JupyterHub pour cet exercice. Si vous souhaitez utiliser votre machine personnelle, voir les instructions sur le site du cours pour installer SFML. À noter que, pour utiliser la SFML sur CodeBlocks ou tout autre IDE, il faut configurer l'IDE. Plus encore que d'habitude, nous recommandons de compiler en ligne de commande dans le terminal. Pour l'instant, en salle de TP, il est nécessaire d'utiliser `info-111 g++` pour compiler un programme utilisant SFML.

(1) Ouvrir les fichiers `exemple-graphisme1.cpp` et `primitives.h` et consulter le premier.

(2) Compiler ce programme depuis le terminal avec (en une seule ligne !) :

```
info-111 g++ exemple-graphisme1.cpp primitives.cpp -o exemple-graphisme1
             -lsfml-system -lsfml-window -lsfml-graphics
```

puis le lancer avec :

```
./exemple-graphisme1
```

Explication : le code binaire de SFML est réparti dans trois bibliothèques `sfml-system`, `sfml-window` et `sfml-graphics`. Les arguments `-lsfml-system`, ... indiquent à g++ de les lier au programme.

(3) Refaire l'exercice 2 du TD en complétant le programme fourni `premier-dessin.cpp`. Pour compiler ce dernier, reprendre en l'adaptant la commande utilisée ci-dessus pour compiler `exemple-graphisme1.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 de l'utilisation de la SFML et de notre bibliothèque de primitives, lire attentivement `primitives.h` et sa documentation.

(2) Consulter ensuite `exemple-graphisme2.cpp` et `exemple-graphisme3.cpp` pour en voir des exemples d'utilisation.

(3) Implanter l'exercice 4 du TD.

Exercice ♣ 5 (Couche d'abstraction)**.**
Pour vous approprier la couche d'abstraction, consultez-en l'implémentation dans `primitives.cpp`. En vous inspirant de ce qui est déjà fait, complétez l'implémentation de la fonction `draw_filled_rectangle` (sa documentation est dans `primitives.h`).
Vous pouvez vous aider de la documentation en ligne de la SFML accessible ici : `https://www.sfml-dev.org/learn-fr.php`.

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 `exemple-graphisme3.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.