Héritage

I Introduction

Le but de ce TP est l'application de l'héritage à la bibliothèque SDL2, via les notions suivantes :

- utilisation du système de construction logicielle *Meson*,
- création d'une bibliothèque partagée (DLL sous Windows, dylib sous Mac, so pour le reste),
- binding de langage C++,
- utilisation des séquences, héritage, classe abstraite et pointeurs intelligents du C++11,
- documentation du code avec Doxygen,
- tests unitaires avec *Boost* (module *unit_ test_ framework*),
- analyse statique de code avec *scan-build* fourni avec *clang*,
- couverture de code avec *lcov*,
- utilisation de *valgrind* ou *DrMemory* pour les erreurs de mémoire.

Il est recommandé de compiler avec *g++* et *clang++* car un compilateur peut trouver des avertissements ou des erreurs que l'autre ne détecte pas.

II Utilisation de la *SDL2*

II.1 Conception

La bibliothèque *SDL2* permet de faire des applications graphiques, comme des jeux (en général 2D) en utilisant le langage C. Elle nécessite au moins :

- une initialisation avec `SDL_Init()` au début du programme.
- `SDL_Quit()` à la fin.
- La création d'une fenêtre, avec `SDL_CreateWindow()`.
- Un *renderer* par fenêtre, *i.e.* une manière de faire l'affichage de pixels dans la fenêtre. *SDL2* en propose un avec `SDL_CreateRenderer`, il n'y en a qu'un seul par fenêtre. On doit appeler la fonction `SDL_RenderPresent()` pour appliquer les changements dans la fenêtre.
- une boucle (infinie) utilisant `SDL_PollEvent()` pour récupérer les appels du clavier ou de la souris par exemple.

II.2 Binding C++

Un binding de langage d'une bibliothèque consiste à écrire les appels de cette bibliothèque écrite en un certain langage, dans un autre langage. La bibliothèque *SDL2* étant écrite en C, on va faire un binding en C++.

III Le sytème de construction logicielle *Meson*

Pour plus d'information sur *Meson*, se référer au document envoyépar mail.

III.1 Présentation

Meson est un système de construction logicielle similaire à *CMake*, mais avec une syntaxe plus agréable. Il permet de choisir les compilateurs, vérifier des dépendances, construire des applications et des bibliothèques, faire des tests unitaire et de la couverture de code, etc... Tout ceci est déjà intégré dans *Meson*.

Il utilise exclusivement *ninja*, un outil de construction similaire à *make*, mais bien plus rapide.

Pour effectuer les constructions d'applications ou bibliothèques, il faut écrire un ou plusieurs fichiers *meson.build*.

Un moyen rapide pour commencer à écrire les fichiers *meson.build* est de regarder des projets existants ou des tutoriels, comme celui de *Meson*, ou bien celui-ci.

— Pour détecter SDL2, voir ici.
— Pour les bibliothèques partagées et exécutables, voir ici.
— Pour les tests unitaires voir ici.
— Pour Boost, voir ici.
— Pour *Doxygen* voir ici.
— Pour l'anayse statique de code voir ici.
— Pour la couverture de code, voir ici.

III.2 Invocation

On invoque *Meson* avec un repertoire dans lequel sera compilé le projet. On peut aussi ajouter des options. Pour plus d'information, voir ici. Par exemple :

```
meson setup builddir
ninja -C builddir
```

Si, après le premier appel de *Meson*, un des fichiers *meson.build* est modifié, il est inutile de relancer *meson* : l'appel de *ninja* ci-dessus relancera *meson* avec toutes les options apssées.

IV Tutoriel sur Boost Test

On pourra se baser sur ce tutoriel pour avoir une idée sur l'utilisation des tests unitaires avec Boost. On peut aussi regarder ceux de Boost.

V Travail

V.1 Généralités

Il s'agira d'écrire une bibliothèque dynamique réalisant un binding C++ de la *SDL2*. La structure des répertoires, ainsi que les fichiers devra être la suivante :

```
tp/
  meson.build
  doc/
    meson.build
    Doxyfile
  src/
    lib/
      sdl/  <-- binding C++ (namespace 'sdl')
        meson.build
        sdl_forward.hpp
        sdl_core.cpp/hpp
        sdl_exception.cpp/hpp
        sdl_window.cpp/hpp
    bin/      <-- fichier de test
      meson.build
      main.cpp
    test/     <-- tests unitaires
      meson.build
      suite.cpp
```

- On utilisera le namespace `sdl` pour les fichiers du binding C++.
- Pour chaque méthode, il faudra décider si elle foit être `const` ou non, statique ou non, si des exceptions doivent être envoyées ou non, ainsi que les paramètres et type de retour.
- Il faudra faire en sorte que le fichier *Sdl.h* de la bibliothèque *SDL2* ne soit jamais inclus dans le fichier *main.cpp* (donc il devra être inclus exclusivement dans les fichiers source du binding C++).
- On pensera à documenter avec *Doxygen* toutes les classes et méthodes.

V.2 Description des méthodes dans les fichiers d'en-tête

- Dans `sdl_forward.hpp`, les déclarations forward (*i.e.* déclarées avant tout code du binding) des types `position` et `size`, ayant 2 membres entiers, pour gérer la position et la taille d'une feneêtre (ou tout autre objet les demandant). On pourra la compléter avec tout autre type pour faciliter l'écriture du binding et son utilisation.
- Dans `sdl_exception.hpp`, une class nommée `error` gérant les erreurs retournées par la fonction `SDL_GetError()`.
- Dans `sdl_core.hpp`, une classe nommée `core` contenant :
  - La méthode `init()` qui initialise la *SDL2*.
  - La méthode `quit()` qui termine la *SDL2*.
  - La méthode `run()` qui lance la boucle infinie.
  - La méthode `exit_run()` qui permet de sortir de la boucle infinie.
- Dans `sdl_window.hpp`, une class nommée `window` contenant
  - Un constructeur créant une fenêtre ayant un titre, une position et une taille donnée.
  - Deux méthodes `show()` et `hide()` pour respectivement afficher une fenêtre et la cacher.
  - Des accesseurs permettant de récupérer le renderer, la taille et la position, nommés `get_renderer`, `get_size` et `get_position` respectivement.
  - Des méthodes `move()` et `resize()` permettant respectivement de déplacer et redimensionner la fenêtre.
  - La méthode `present()` qui appelle la fonction `SDL_RederPresent()`.

V.3 Tests unitaires

Dans le fichier *test/suite.cpp*, on testera la position, la taille, le titre et le fait que la fenêtre est visible ou non. On effectuera aussi la couverture de code.

V.4 Héritage et classes abstraites, application aux événements

Tout environnement graphique (Windows, Gnome, KDE, *etc...*) est basé sur des événements : quand on presse une touche du clavier, deux événements sont envoyés (quand la touche est pressée, puis quand elle est relachée), ou bien quand on utilise la souris (déplacement, bouton appuyé puis relaché, etc...). La *SDL* permet bien sûr de détecter ces événements : la méthode `core::run()` lance une boucle infinie et on récupère les événements avec la fonction `SDL_PollEvent()`. Une événement est de type `SDL_Event`. Le but sera de lancer nos propres fonctions (en fait des méthodes) quand une touche est pressée. Ce `SDL_Event` contient le *le symbole de touche* (en anglais *keysym*). Ce sera un caractère C. Par exemple quand la touche 'q' est pressée, le keysym sera le caractère C `'q'`.

— On commencera par écrire une classe abstraite nommée `event` dans le namespace `sdl` et dans le fichier *event.hpp*. Elle contiendra une méthode virtuelle pure ayant cette déclaration :

```
virtual void callback(int keysym) const = 0;
```

La méthode `callback()` sera définie dans nos propres événement via une classe fille.

— Comme plusieurs événements (*i.e.* les classes filles ci-dessus) peuvent être définies, il faut pouvoir les stocker. On utilisera une liste doublement chaînée de la STL pour les stocker. Il faudra que le template de cette liste soit `sdl::event`. Mais celui-ci est une classe abstraite et on ne peut pas l'instancier. Il faut pour cela utiliser un pointeur. Pour gérer la destruction automatique de ce pointeur, on va utiliser un pointeur intelligent, comme par exemple un `std::shared_ptr<event>`. La liste doublement chaînée sera mise dans la classe `core`.

— on ajoutera dans la classe `core` la méthode `void add_event()` qui ajoutera un évévement `event` à la liste.

— la méthode `run()` listera tous les éléments de la liste et appellera la méthode `callback()`, et ceci à chaque fois un événement SDL de type `SDL_KEYDOWN` sera envoyé.

— A titre d'exemple, écrire une classe héritant de `sdl::event` et définissant la méthode `callback()` qui arrêtera l'application quand la touche (le keysym) 'q' est pressée. Il faudra aussi ajouter une méthode statique nommée `std::shared_ptr<event> create()` qui renverra le pointeur intelligent. On l'utilisera évidemment avec `void add_event()`.