Université de Nice-Sophia Antipolis
Polytech ELEC4

Mardi 24 mars 2020
Durée: 2h

Gtkmm - MVC
Travaux Dirigés – Séance n. 12

---

1 Introduction

Les entrées et les sorties des programmes que vous avez écrits jusqu'à maintenant étaient purement textuelles. Aujourd'hui, les programmes communiquent avec leurs utilisateurs au moyen d'interfaces graphiques. Ces interfaces ont complètement transformé le dialogue avec l'utilisateur, grâce à des composants graphiques (*widgets* en anglais), tels que les menus déroulants ou les boutons accessibles avec une souris. L'utilisation des programmes en est bien souvent considérablement simplifiée.

La plupart des langages de programmation possède des bibliothèques de composants graphiques, prêts à l'emploi, pour créer ces interfaces graphiques. Il en existe de nombreuses qui peuvent être utilisées avec C++. Dans ce TD, vous utiliserez *gtkmm* qui est une surcouche, adaptée à C++, de la bibliothèque *GTK+* (conçu pour être interfacée avec le langage C). À l'origine, la bibliothèque graphique *GTK+* (The **G**IMP **T**ool**k**it)) a été conçue pour le logiciel libre de traitement d'images *Gimp*, et utilisée, aujourd'hui, en particulier dans l'environnement de bureau GNOME sous Linux. L'installation de *gtkmm* sous Ubuntu se fait simplement à l'aide de la commande :

```
sudo apt install libgtkmm-3.0-dev
```

Les exercices de ce TD vous permettront d'aborder les bases de *gtkmm*. Pour une utilisation plus approfondie, vous devrez consulter la documentation de cette bibliothèque graphique disponible à l'url `https://www.gtkmm.org/en/documentation.html`.

Dans ce TD, nous verrons les bases de `gtkmm`, et nous mettrons en œuvre, à travers un exemple simple celui d'un convertisseur de températures degrés celsius-farenheit, le modèle *MVC* (Modèle-Vue-Controleur) et le patron de conception *Observateur*.

2 Une simple fenêtre

Le programme suivant crée et fait apparaître une fenêtre graphique.

```
#include <gtkmm.h>
int main(int argc, char *argv[]) {
  auto app =  Gtk::Application::create(argc, argv);
  Gtk::Window window;
  return app->run(window);
}
```

La fonction `main` commence par la création d'objet « application gtkmm » avec la méthode `create`. On lui passe les paramètres programme parmi lesquels elle cherche à reconnaître des options qui lui sont propres, sans traiter les autres. La méthode renvoie un pointeur sur cet objet créé.

La déclaration de la variable `window` crée un objet fenêtre (top-level). Cet objet est ensuite transmis à l'objet application qui assure la gestion de l'ensemble des composants graphiques et des événements de l'application graphique. La méthode `run` renvoie un code de retour. En général, l'appel de cette méthode est la dernière instruction du programme.

Pour compiler et exécuter ce programme, vous devez, bien évidement, avoir installé dans votre environnement *gtkmm*, et plus particulièrement sa version 3.0, *gtkmm3*. Si le texte du programme est placé dans le fichier `simple_fenetre.cpp`, sa compilation se fait par l'instruction :

```
g++ simple_fenetre.cpp $(pkg-config gtkmm-3.0 --cflags --libs)
```

Il est bien évidemment conseillé de fabriquer systématiquement un fichier `Makefile` pour simplifier la compilation de vos programmes.

exercice 1) Testez le programme précédent.

exercice 2) La méthode `set_default_size(`*l*`,` *h*`)` appliquée à une fenêtre permet de spécifier sa taille (en pixels). Modifiez le programme précédent pour créer une fenêtre de largeur 300 pixels, et de hauteur 100 pixels.

3 Composants graphiques

La construction d'une interface graphique se fait par l'agencement de différents composants graphiques (widgets). Les bibliothèques graphiques en proposent de nombreux, prêts à l'emploi, tels que les boutons, les radio-boutons, les menus déroulants, les ascenseurs, les entrées, les canvas, et de très nombreux autres.

Les composants graphiques sont placés dans des conteneurs qui peuvent en définir l'agencement au moyen d'objets spéciaux.

Nous allons commencer par faire un programme qui crée et affiche une fenêtre composée d'une image et d'un bouton, comme le montre la figure suivante :

Cette fenêtre sera représentée par une classe, appelée ici `Vue`, qui contiendra deux wigdets, d'une part une image de type `Gtk::Image` et d'autre part un bouton de type `Gtk::Button`. La constructeur de la fenêtre se fera dans le constructeur de la classe `Vue`.

On passe au constructeur d'images le nom du fichier (une chaîne de caractères) qui contient l'image numérisée au format png, jpeg, *etc.* Pour le bouton, on donne à son constructeur le texte à écrire dessus.

Pour placer, ces wigdets dans une fenêtre, on utilisera un conteneur de type `Gtk::Box` avec un agencement vertical `Gtk::ORIENTATION_VERTICAL`.

L'ajout des widgets dans le conteneur `Gtk::Box` se fait à l'aide d'une méthode `pack_start`, et celui du conteneur dans la fenêtre courante se fait à l'aide d'une méthode `add`.

Enfin, pour visualiser tous les widgets qui composent la fenêtre, il faut exécuter la méthode `show_all_children`.

On peut donner maintenant la déclaration de la classe `Vue` :

```
#pragma once

#include <gtkmm/button.h>
#include <gtkmm/window.h>
#include <gtkmm/image.h>
#include <gtkmm/box.h>

class Vue : public Gtk::Window {
private:
  Gtk::Box box;
  Gtk::Button bClose;
  Gtk::Image img;
public:
  Vue() : box(Gtk::ORIENTATION_VERTICAL), bClose("Close"), img("GTK.png")
  {
    box.pack_start(img);
    box.pack_start(bClose);
    add(box);
    show_all_children();
  }
  // le destructeur
  virtual ~Vue() {}
};
```

La fonction `main` du programme principal consiste simplement à passer à la méthode `run` de l'application une instance de `Vue`.

```
#include <gtkmm/application.h>
#include "Vue.hpp"
int main(int argc, char *argv[]) {
  auto app = Gtk::Application::create(argc, argv);
  return app->run(* new Vue());
}
```

---

exercice 3) Récupérez une image de votre choix et testez le programme précédent.

Événements – Signaux Jusqu'à présent, l'exécution de vos précédents programmes suivait l'ordre séquentiel et immuable défini par son algorithme. Au contraire, celle d'un programme contrôlé par une interface graphique est dirigée par l'utilisateur, et plus précisément par les *événements* auxquels réagissent les composants graphiques. Un événement est par exemple le déplacement de la souris, la sélection d'un menu, l'appui sur un bouton, *etc.*

Tous les composants graphiques peuvent *émettre* des événements, appelés *signaux* dans *gtkmm*. On pourra associer aux widgets des gestionnaires de signaux (*signal handlers*) qui pourront déclencher des actions à l'arrivée d'un signal.

Par exemple, pour déclencher la méthode `action` appartenant à la classe `C` à appliquer sur l'objet `obj`, lorsqu'on clique sur le bouton `b`, on écrira :

```
b.signal_clicked().connect(sigc::mem_fun(obj, &C::action));
```

Pour que le bouton « Close » de notre application ferme la fenêtre (et donc termine l'exécution de l'application), on va ajouter au bouton `bClose`, le gestionnaire suivant :

```
bClose.signal_clicked().connect(sigc::mem_fun(*this, &Vue::close));
```

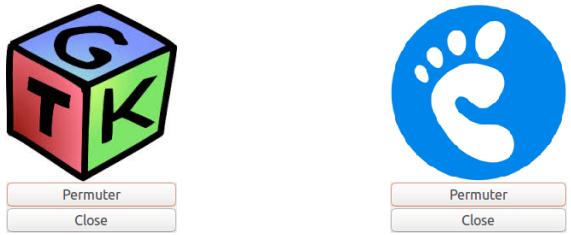
où la fonction `close`, déclarée locale à la classe `Vue`, est définie de la façon suivante :

```
void close() { hide(); }
```


---

exercice 4) Dans la votre classe `Vue`, ajoutez le gestionnaire de signal précédent. Testez votre programme.

On veut maintenant, visualiser deux images à tour de rôle en les faisant permuter à l'aide d'un bouton. La figure ci-dessous montre la fenêtre avec la première image (à gauche), et la seconde image (à droite), le bouton « Permuter » faisant passer de l'une à l'autre.



exercice 5) Récupérez une seconde image. Ajoutez le bouton « Permuter » à votre classe `Vue`, et associez-lui le gestionnaire de signal qui permettra la permutation des deux images.

4 Convertisseur de degrés

Nous allons maintenant écrire une petite application qui convertit des degrés celsius en degrés farenheit, et réciproquement. Nous allons voir comment structurer cette application graphique à l'aide du modèle *MVC* et du patron de conception *Observateur*.

4.1 Modèle MVC

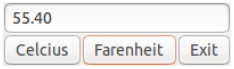
Le modèle MVC (Modèle-Vue-Contrôleur, *Model-View-Controller*) structure l'application en 3 parties :

1. le *Modèle*, qui représente le noyau de l'application. Il contient les données de l'application, leur traitement et fournit des résultats sans forme de présentation. Son interface permet l'accès et la mise à jour des données.
2. la *Vue*, qui visualise les données et les résultats calculés par le modèle. Il traite les événements transmis par le contrôleur. La séparation de la *Vue* du *Modèle* permet, en particulier, d'avoir plusieurs vues différentes d'un même modèle.
3. le *Contrôleur* qui assure la connexion entre les 2 parties précédentes. En particulier, il enregistre les gestionnaires d'événements qui traitent les événements émis par l'utilisateur.

Ce modèle permet de structurer clairement une Interface Homme/Machine (IHM), et en réduisant, autant que faire se peut, les dépendances entre les parties *Vue* et *Modèle* (découplage).

Dans notre programme de conversion de degrés :

1. le *Modèle* assurera la conversion selon les relations $F = 1.8 \times C + 32$ et $C = (F - 32)/1.8$.
2. la *Vue* sera graphique, programmée avec gtkmm. Elle possédera une entrée pour la saisie des degrés et l'affichage de la valeur convertie, plus deux boutons pour convertir en degrés celsius ou farenheit. La figure ci-dessous donne une *Vue* possible du *Modèle*.



3. le *Contrôleur* enregistrera les gestionnaires de signaux.

4.2 Patron de conception « Observateur »

Le patron de conception *Observateur* permet de synchroniser l'état d'un composant, appelé l'*observé*, avec d'autres composants, appelés les *observateurs*. Quand l'état de l'*observé* change, celui-ci en informe les *observateurs* qui se mettent à jour. L'ensemble des *observateurs* s'enregistrent au préalable auprès de l'*observé*.

Ce patron est bien souvent utilisé avec le modèle MVC. Le modèle est le composant *observé* et la/les vue(s) est/sont le(s) composant(s) *observateur(s)*. Chaque modification des données sera notifiée aux vues qui pourront alors la visualiser.

Pour mettre en œuvre ce patron, on peut définir deux classes génériques `Observateur` et `Observable`.

La première classe est abstraite (donnée ci-dessous), et doit être mise en œuvre par héritage par chaque observateur. Ce dernier implémente la méthode `update` dont le paramètre fournit l'information qu'il doit mettre à jour.

```
#pragma once

template<typename T>
class Observateur {
public:
  virtual void update(T info) =0;
};
```

L'*observé* hérite de la classe `Observable`. Cette classe propose deux méthodes. Chaque observateur s'enregistre auprès de l'observé à l'aide de `ajouterObservateur`. L'observé conserve l'ensemble des observateurs dans une liste. La seconde méthode, `notifierObservateurs`, sera exécutée par l'observé pour informer chaque observateur de la liste qu'un changement a eu lieu. Elle transmet l'information modifiée que les observateurs mettront à jour. Si l'observateur est une vue, l'information modifiée sera mise à jour, par exemple, par l'interface graphique.

```
#pragma once

#include <list>
#include "Observateur.hpp"

template<typename T>
class Observable {
private:
  std::list<Observateur<T> *> list_observateurs;

public:
  void notifierObservateurs(T info) {
    for (auto obs : this->list_observateurs) obs->update(info);
  }
  void ajouterObservateur(Observateur<T> * observateur) {
    this->list_observateurs.push_back(observateur);
  }
};
```


4.3 Le modèle

exercice 6) Écrivez la classe `Modele` qui hérite de la classe `Observable`. Elle conserve le nombre de degrés courant et propose les deux méthodes suivantes :

```
// Rôle : convertit c degrés Celsius en Fahrenheit
void convertirEnFahrenheit(double c);
// Rôle : convertit f degrés Fahrenheiten Celsius
void convertirEnCelsius(double f);
```

4.4 La vue

exercice 7) Écrivez la classe `VueGraphique` qui hérite de la classe `Observateur`. Vous organiserez la fenêtre graphique comme le montre la figure donnée plus haut.

La classe `Gtk::Entry` définit une zone de saisie de texte. Dans cette classe, la méthode `get_text()` renvoie le texte contenu dans la zone de saisie. La méthode `set_text(t)` insère le texte `t` dans la zone de saisie. Attention, la valeur n'est pas de type `std::string` mais d'un type spécifique de la bibliothèque, `Glib::ustring`. Cette dernière classe fournit la méthode `c_str()` qui renvoie sa valeur convertie en `std::string`.

exercice 8) Redéfinissez la méthode `update` issue de la classe abstraite `Observateur`

4.5 Le contrôleur

exercice 9) Écrivez la classe `Controleur` qui enregistrera les gestionnaires de signaux pour les deux boutons de conversion de la vue graphique. Puisque le contrôleur établit la connexion entre le modèle et la vue, il doit avoir accès aux instances de ces deux classes.

La méthode `main` du programme principal pourra avoir la forme suivante :

```
#include <gtkmm/application.h>
#include "VueGraphique.hpp"
#include "Modele.hpp"
#include "Controleur.hpp"

int main(int argc, char *argv[])
{
  auto app = Gtk::Application::create(argc, argv);
  Modele *m= new Modele();
  VueGraphique *vg = new VueGraphique();  // la vue graphique
  Controleur *c = new Controleur(m, vg);
  m->ajouterObservateur(vg); // la vue graphique observe le modèle

  return app->run(*vg);
}
```

exercice 10) Ajoutez une vue textuelle de façon à afficher les conversions, *simultanénment*, sur l'interface graphique et sur la sortie standard.