Introduction à gtkmm 2.24

Plan de cours

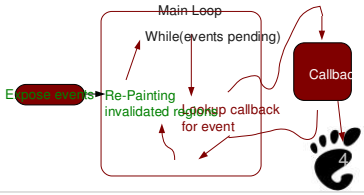
1. Vision d'ensemble
2. Gestion des évènements
3. Présentation des widgets
4. Dessins dans Gtkmm : Gdk + Cairo
5. Drag'n Drop
6. Conception d'interfaces avec Glade
7. Menus et menus contextuels

Dessin et polices

1. Rappels de la gestion des évènements d'affichage
2. Dessin natif dans Gtkmm : Gdk
3. Dessin avec Cairo
4. Gestion des fonts avec Pango

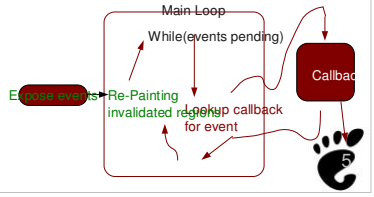
Rappels sur les évènements graphiques

- Les widgets sont redessinés lorsque une région dont ils ont la charge a été invalidée par l'appel d'un `queue_draw()`
  - redimensionnement de fenêtre
  - déplacement de fenêtre
  - superposition d'une autre fenêtre
  - ...



Rappels sur les évènements graphiques

- Que se passe-t-il lorsque le widget doit être redessiné ?
  - La boucle d'évènements appelle la fonction `on_expose_event()`
  - Gtkmm3 : la fonction s'appelle `on_draw()`



Rappels sur les évènements graphiques

- Conclusions
  - Tous les affichages dans des widgets qui sont réalisés en dehors de `on_expose_event` ne seront **pas pérennes**
    - Si un `queue_draw()` est lancé, le dessin réalisé sera perdu
  - Tous les affichages réalisés dans `on_expose_event` sont refaits à chaque fois que la fenêtre est redessinée (suite à l'invalidation de la zone par un `queue_draw()`)
    - Nécessite de créer une nouvelle classe pour surcharger cette fonction
    - Essayer d'être le plus efficace possible dans ces fonctions !

Illustration

- Exemple d'utilisation d'un `on_expose_event`
  - exemples test_cairo1 à test_cairo4

Le dessin dans Gtkmm : Gdk

- Classes usuelles
  - `Gdk::Color / Gdk::Colormap`
  - `Gdk::Point / Gdk::Rectangle`
- Deux classes de gestion des affichages
  - `GC` : Graphics Context : description du 'pinceau' avec lequel dessiner
  - `Drawable` : classe d'un objet qui accepte des primitives de dessin
  - Dans Gtkmm 3, la fonction `on_draw()` s'attend à ce que vous utilisiez `Cairomm` à la place !!
- `Image` et `Pixbuf`

Gdk::Color / Colormap

- Color
  - `Color (const Glib::ustring& value)`
    - Définit une couleur à partir de son nom dans la table des couleurs X
  - Fonctions set
    - `set_red(...), set_green(...), set_rgb(...), set_hsl(...)`
  - Fonctions get correspondantes
  - Gtkmm 3 : remplacement par `Gdk::RGBA`
- Colormap
  - Table de conversion des couleurs permettant d'ajuster les couleurs d'affichage pour les codes rgb.
  - `Gdk::Colormap::get_system()` : récupère le colormap du système
  - Supprimé dans Gtkmm3

Gdk::Point / Gdk::Rectangle

- Point : `x, y`
- Rectangle : `x, y, width, height`
  - `Rectangle& intersect (const Rectangle& src2, bool& rectangles_intersect)`
  - `Rectangle& intersect (const Rectangle& src2)`
  - `Rectangle& join (const Rectangle& src2)`

Gdk::Drawable

- Fonctions de dessin de lignes (sans surface)
  - `void draw_line (const Glib::RefPtr<const GC>& gc, int x1, int y1, int x2, int y2)`
  - `void draw_lines (const Glib::RefPtr<const GC>& gc, const Glib::ArrayHandle<Point>& points)`
- Fonctions avec surface
  - `void draw_rectangle (const Glib::RefPtr<const GC>& gc, bool filled, int x, int y, int width, int height)`
  - `void draw_polygon (const Glib::RefPtr<const GC>& gc, bool filled, const Glib::ArrayHandle<Point>& points)`
  - `void draw_arc (const Glib::RefPtr<const GC>& gc, bool filled, int x, int y, int width, int height, int angle1, int angle2)`
- Et affichage d'images, ...

Gdk::Drawable

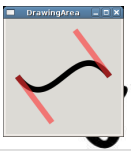
- Toutes les fonctions de dessin prennent en paramètre un contexte graphique (GC)
- Le GC spécifie le « pinceau » avec lequel réaliser une forme
- Il définit en particulier
  - les couleurs de dessins
    - `foreground` : couleur de traits
    - `background` : couleur des fond (filled)
  - l'épaisseur des traits
  - le types de ligne (dashed ou plain)

Gdk::Drawable

- En pratique, il faut créer une instance de GC et la modifier pour ce besoin
  - Mais les constructeurs sont protégés ...
- Comment construire un contexte graphique ?
  - Utilisation la fonction `static create(drawable &)`
  - On peut récupérer le drawable de la fenêtre dans laquelle est un widget par la fonction `get_window()`
    - NB: c'est le drawable qui peut être utilisé pour faire appel aux fonctions de dessin

```
Glib::RefPtr<Gdk::Window> window = get_window();
_gc = Gdk::GC::create(window);
```

Cairo : Généralités



- Librairie de dessin vectoriel
  - Consiste à définir un dessin à partir de primitives simples
  - Les primitives sont définies par des vecteurs
- Plus avancée que Gdk, par exemple
  - Gestion de transformations géométriques
    - Rotation, échelle, translation
    - Par matrice d'homomorphisme (espaces projectifs)
  - Courbes de Bézier (B-spline)
  - Transparence

Le dessin dans Gtkmm : Gdk

```
Glib::RefPtr<Gdk::Window> window = get_window();
gc = Gdk::GC::create(window);

int winx, winy, winw, winh, wind;
window->get_geometry(winx, winy, winw, winh, wind);

window->clear();

// 4 points blancs
gc->set_foreground( Gdk::Color('white') );
window->draw_point(gc, 5, 5);
window->draw_point(gc, 5, winh-5);
window->draw_point(gc, winw-5, winh-5);
window->draw_point(gc, winw-5, 5);

// Une croix épaisse grise
gc->set_foreground( Gdk::Color('grey') );
gc->set_line_attributes(/*line_width*/7,
  /*LineStyle*/Gdk::LINE_SOLID, /*CapStyle*/Gdk::CAP_NOT_LAST,
  /*JoinStyle*/Gdk::JOIN_MITER);

window->draw_line(gc, 10, 10, winw-10, winh-10);
window->draw_line(gc, winw-10, 10, 10, winh-10);
```

Le dessin avec Cairo

- Tout le dessin est construit à partir d'appels de fonctions au `Context` Cairo
  - `Cairo::RefPtr<Cairo::Context> cr`
- Le Context Cairo, fusionne :
  - Le **drawable**, *ie* les primitives de dessin
    - `cr->move_to(x0, y0);`
    - `cr->curve_to(x1, y1, x2, y2, x3, y3);`
    - `cr->stroke();`
    - `cr->fill();`
  - Le **contexte graphique**, *ie* les propriétés de la forme à dessiner
    - `cr->set_line_width(0.05);`
    - `cr->set_source_rgb(1, 0.2, 0.2);`

Le dessin dans Gtkmm : Gdk

- `Gdk::Drawable` : Dessine 'à la volée' sur l'écran
  - Gtkmm3 : supprimé (déplacement dans `Gdk::Window`)
- `Gdk::Pixmap`
  - Un drawable, mais sur lequel on peut dessiner 'off-screen'
  - Le dessin peut être 'recopié' sur un drawable
    - `Gdk::Drawable::draw_drawable()`
  - Gtkmm 3 : remplacement par `Gdk::Pixbuf`

Le dessin avec Cairo

- On créé un contexte Cairo de la même manière qu'un GC

```
Glib::RefPtr<Gdk::Window> window = get_window();
if(window)
{
  Cairo::RefPtr<Cairo::Context> cr = window->create_cairo_context();
  ...
}
```

Le dessin avec Cairo

- Le principe de Cairo : *définir un chemin ou une forme, puis de les faire afficher avec les propriétés choisies*
  - Intermédiaire entre le `Drawable` et le `Pixmap`
- Exemple (test_cairo1, test_cairo2)
  - `Stroke` : dessine les lignes
  - `Fill` : remplit l'intérieur d'un chemin

```
// draw curve
cr->move_to(x0, y0);
cr->line_to(x1, y1);
cr->set_line_width(10);
cr->set_source_rgb(0.8, 0, 0)
cr->stroke();
```

```
// draw filled circle
cr->set_source_rgb(0.3, 0.6, 0.1)
cr->arc(0, 0, m_line_width / 3.0,
   0, 2 * M_PI);
cr->fill();
```


Le dessin avec Cairo

- Les différents types de définition de formes
  - Fonctions standards
    - **Les positions des points sont absolues**
    - **On donne les coordonnées dans le référentiel de contexte graphique (cf. test_cairo1)**
    - `move_to(x,y)` : positionne le début d'un chemin
    - `line_to(x,y)` : relie la position courante au point (x,y) avec une ligne droite
    - `curve_to(x, y, x2, y2, x3, y3)` : relie la position courante au point (x,y) avec une B-Spline (points de contrôle 2 et 3)
  - Fonctions relatives : `rel_`
    - **Les positions des points sont relatives au point courant**
    - `rel_move_to, rel_line_to, rel_curve_to`

Le dessin avec Cairo

- Les différentes fonctions de réalisation
  - `Stroke` et `fill`
    - Réalisent le tracé d'une forme (ligne ou fond)
    - Suppriment la forme du 'buffer' de forme
  - Fonctions `_preserve`
    - `stroke_preserve, fill_preserve`
    - **Préserve la forme pour une nouvelle réalisation**
    - Permet d'utiliser une même forme pour réaliser des tracés avec des propriétés graphiques différentes sans redéfinir celle-ci
    - Exemple : test_cairo3

```
cr->rectangle(x, y, width, height);

cr->set_source_rgb(0.3, 0.6, 0.1);
cr->fill_preserve();
cr->set_source_rgb(0.8, 0, 0);
cr->stroke();
```


Le dessin avec Cairo

- Le `clip()` : définition de la fenêtre à dessiner

```
Cairo::RefPtr<Cairo::Context> cr = window->create_cairo_context();
// clip to the area indicated by the expose event so that we only
// redraw the portion of the window that needs to be redrawn
cr->rectangle(event->area.x, event->area.y,
        event->area.width, event->area.height);
cr->clip();
```


Le dessin avec Cairo

- Affichage d'une image
  - Utilisation de la classe Surface
    - `image_surface_create_from_png('filename');`
  - Fonction `paint()` permet d'afficher une surface inscrite dans un `Context`
- Exemple : test_cairo4

Le dessin avec Cairo

- Les chemins peuvent être complexes
  - Comportent autant de ligne que souhaité
  - Composés de sous-chemins (`sub_path`)
- Possibilité de faire des remplissages (`fill`) complexes
  - `set_fill_rule()`
- Les transformations géométriques des objets
  - `scale`
  - `rotate`

La gestion des fonts avec Pango

- Pango permet de gérer l'affichage de différentes polices
- Exemple : test_pango1

```
Cairo::RefPtr<Cairo::Context> cr = window->create_cairo_context();

Glib::RefPtr<Pango::Layout> pangoLayout = Pango::Layout::create(cr);
pangoLayout->set_font_description( Pango::FontDescription("sans bold
20") );
pangoLayout->set_text("text");
pangoLayout->update_from_cairo_context(cr);

cr->move_to(20,10);
pangoLayout->add_to_cairo_context(cr);

cr->set_source_rgb(0.8,0,0);
cr->fill();
```

Drag'n Drop : implémentation

- Définition des zones de Drag et de Drop
  - À la construction d'un objet
    - `drag_source_set` : définit les types de données qui peuvent être prises sur l'objet
    - `drag_dest_set` : définit les types de données qui peuvent être déposées
  - Type d'objet définit par une chaîne de caractères (propre ou standard, `Gtk::Atom`)

```
std::list<Gtk::TargetEntry> listTargets;
listTargets.push_back( Gtk::TargetEntry("MON_TYPE") );
listTargets.push_back( Gtk::TargetEntry("text/plain") );

//Make a DnD drag source:
drag_source_set(listTargets);

//Make a DnD drop destination:
drag_dest_set(listTargets);
```

Drag'n Drop : implémentation

```
void drag_source_set(const ArrayHandle_TargetEntry& targets,
GdkModifierType start_button_mask, GdkDragAction actions);

void drag_dest_set(const ArrayHandle_TargetEntry& targets,
GtkDestDefaults flags, GdkDragAction actions);
```

- On peut affiner la nature du DnD permise sur un objet en définissant
  - `GdkDragAction` : le type d'action possible : `Gdk::ACTION_MOVE, Gdk::ACTION_COPY, ...`
    - Comportements internes pouvant légèrement changer
  - GdkModifierType : comment l'émetteur commence le DnD (souris, keyboard) : `Gdk::MODIFIER_MASK`
  - GtkDestDefaults : comment le récipiendaire répond visuellement à une sollicitation de DnD : `Gtk::DEST_DEFAULT_ALL`

Drag'n Drop : implémentation

- Implémentation de l'action à réaliser au tout début du DnD
  - `void on_drag_begin (const Glib::RefPtr<Gdk::DragContext>& context)`
  - Le contexte contient les informations sur le type de DnD qui pourra être faite (actions et types de données possibles par le Widget source)
- Implémentation des actions pendant le déplacement
  - `void on_drag_leave (const Glib::RefPtr<Gdk::DragContext>& context, guint time)`
  - `bool on_drag_motion (const Glib::RefPtr<Gdk::DragContext>& context, int x, int y, guint time)`
  - **NB : Pas de fonction `on_drag_enter(...)` !!**
  - **NB 2 : Fonction/signal appelée par le widget survolé par la souris (et pas le widget source !!)**

Drag'n Drop : implémentation

- Lorsque l'utilisateur passe au dessus ou lorsqu'il lâche sur un widget cible :
  - Mécanisme ***interne*** de vérification de la faisabilité du DnD basé sur les listes de targets
- Lorsque l'utilisateur lâche le bouton :
  - Si pas ok : c'est la fin du DnD
    - Le **widget source** exécute la fonction `drag_end(...)`
  - Si ok, il faut procéder **manuellement** à l'échange des données,
  - Pas ok = il n'y a pas de point commun entre la liste des types sources et la liste des types destinations

Drag'n Drop : implémentation

- Échange des données : séquence d'appels de fonctions :

1) widget source : `on_drag_data_get`
2) widget cible : `on_drag_data_received`
   1) Doit faire appelle à la fonction `drag_finish` pour indiquer si le DnD c'est bien passé.

· Si ok :
   1) widget source : `on_drag_delete` (Pas fait si l'action est COPY)
   2) widget cible : `on_drag_drop`
   3) widget source : `on_drag_end`

· Sinon :
   1) widget source : `on_drag_end`

Drag'n Drop : implémentation

- `void on_drag_data_get(const Glib::RefPtr<Gdk::DragContext>& context, Gtk::SelectionData& selection_data, guint info, guint time)`
  - Dans cette fonction, on doit remplir le `Gtk::SelectionData` avec la donnée à échanger
    - Précise le type des données qui seront fournies
    - Passe effectivement des données par un emplacement mémoire partagé (inter-applications) :
      - `type` : la chaîne de caractères du type (doit être parmi les types déclaré par `drag_source_set`)
      - `format '=8'` : définit la longueur mémoire des `guint8*` (!)
      - `data` : pointeur sur un emplacement mémoire des données
      - `length` : place en mémoire

```
void Gtk::SelectionData::set( const std::string & type, int format, const guint8 *data, int length)
```

Drag'n Drop : implémentation

- `void on_drag_data_received(const Glib::RefPtr<Gdk::DragContext>& context, int x, int y, Gtk::SelectionData& selection_data, guint info, guint time)`
  - `x, y` : position du pointeur lors du lâché
  - À partir du `Gtk::SelectionData`, récupérer le type de données échangées (fonction `get_target()`) et récupérer effectivement les données à partir du pointeur sur l'espace mémoire (fonction `const guint8 * get_data()`)
  - Finalisé avec un appel à `drag_finish()`
    - `success` : `TRUE` si le DnD a réussi, `FALSE` sinon
    - `del` : Indique si on force un appel à un `drag_delete` ou non

```
void Gdk::DragContext::drag_finish (bool success, bool del, guint32 time)
```

Drag'n Drop : implémentation

- Finalisation,
  - `void on_drag_data_delete (const Glib::RefPtr<Gdk::DragContext>& context)`
    - Fonction qui permet à la source de supprimer sa copie des données transmises
    - pas effectuée si `ACTION_COPY`, mais peut être forcé par `drag_finish`
  - `bool on_drag_drop (const Glib::RefPtr<Gdk::DragContext>& context, int x, int y, guint time)`
    - Finalisation côté destination (succès uniquement)
    - Effectue un `drop_finish();`
  - `void on_drag_end (const Glib::RefPtr<Gdk::DragContext>& context)`
    - Finalisation côté source (avec ou sans succès)

Drag'n Drop : implémentation

- Voir les exemples `DragNDrop` et `DragNDrop2` pour les enchaînements des évènements

Drag'n Drop : remarques ...

- Le Drag'N Drop a été pensé pour les échanges inter-applications
- Attention : dans des cas particuliers, les profils de ces fonctions peuvent se montrer restrictifs
  - lors des mouvements, le widget n'a pas accès aux données sources !
  - pas de fonction `drag_enter(...)`
  - la position du curseur n'est pas directement accessible
- L'affichage du pointeur se gère éventuellement avec la fonction `DragContext::set_icon(...)`

Drag'n Drop d'une couleur

- Exemple applicable au TP
- Comment copier et coller des couleurs entre applications ?
  - qui envoie et qui reçoit ?
  - quel descripteur de target utiliser ?
  - quel est le format des données à transmettre ?

Drag'n Drop de couleurs

- Comment copier et coller des couleurs entre applications ?
  - qui envoie et qui reçoit ?

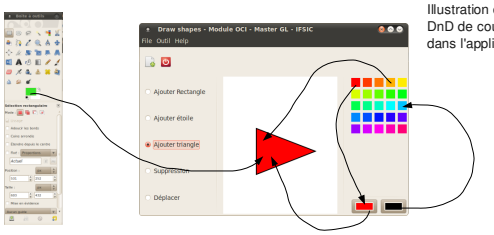


Illustration des DnD de couleurs dans l'application

Drag'n Drop de couleurs

- Comment copier et coller des couleurs entre applications ?
  - quel est le format des données à transmettre ?
    - **Target standard implique l'utilisation de données standards !!!!** (sinon plantage entre applications !!)
    - Format des couleurs définies derrière `application/x-color` ??
      - Tableaux de quatre shorts (entiers sur 16 bits), dans l'ordre RGBA
    - Exemple (très partiel …) de réception d'une couleur X

```
void on_drag_data_received ( ..., const Gtk::SelectionData& selection_data, ...)
{
  Gdk::Color c;
  // application/x-color est en format 16-bit, on fait la conversion :
  const guint16* data = reinterpret_cast<const guint16*>(selection_data.get_data());
  // et on remplit le Gdk::Color
  c.set_red(data[0]);
  c.set_green(data[1]);
  c.set_blue(data[2]);
}
```


Drag'n Drop de couleur

- Comment copier et coller des couleurs entre applications ?
  - quel descripteur de target utiliser ?
    - Il faut utiliser un descripteur compris par les autres applications pour pouvoir échanger avec elles
      - Consultation des standards X (pour Windows ??)
    - Pour l'échange de couleurs, utiliser les targets suivants :

```
Gtk::TargetEntry("application/x-color")
```


Plan de cours

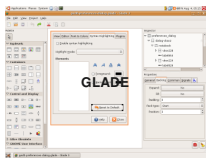
1. Vision d'ensemble
2. Gestion des évènements
3. Présentation des widgets
4. Dessins dans Gtkmm : Gdk + Cairo
5. Drag'n Drop
6. Conception d'interfaces avec Glade
7. Menus et menus contextuels

Drag'n Drop de couleurs

- Comment copier et coller des couleurs entre applications ?
  - quel est le format des données à transmettre ?
    - **Target standard implique l'utilisation de données standards !!!!** (sinon plantage avec un DnD entre applications !!)
    - Format des couleurs définies par `application/x-color` ??
      - Tableaux de quatre shorts (entiers sur 16 bits), dans l'ordre RGBA
    - Exemple d'envoi d'une couleur X

```
void on_drag_data_get (..., Gtk::SelectionData& selection_data, ...)
{
     guint16 data[4];
     data[0]=_color.get_red();
     data[1]=_color.get_green();
     data[2]=_color.get_blue();
     data[3]=65535; //couche alpha
     selection_data.set("application/x-color", 16, (guint8*) data, sizeof(guint16)*4);
}
```


Principe d'une interface avec Glade



`monfichier.glade`

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE glade-interface SYSTEM "glade-2.0.dtd">
<!--Generated with glade3 3.4.5 on Mon Nov  2 19:21:10 2009 -->
<glade-interface>
  <widget class="GtkWindow" id="window1">
    <property name="border_width">6</property>
    <property name="title" translatable="yes">Charter</property>
    <property name="default_width">600</property>
    <property name="default_height">400</property>
    <signal name="destroy" handler="gtk_main_quit()"/>
    <child>
      <widget class="GtkVBox" id="vbox1">
        <property name="visible">True</property>
        <child>
          <widget class="GtkHBox" id="hbox1">
            <property name="visible">True</property>
            <child>
              <widget class="GtkScrolledWindow" id="scrolledwindow1">
                <property name="visible">True</property>
                <property name="can_focus">True</property>
                <property name="hscrollbar_policy">GTK_POLICY_NEVER</property>
                <property name="vscrollbar_policy">GTK_POLICY_AUTOMATIC</property>
                <child>
```

Chargement dynamique des widgets à l'aide de `Gtk::Builder`

Le chargeur d'interface Gtk::Builder

- Création d'un builder et chargement d'un fichier Glade
  - Plusieurs fichiers peuvent être ajoutés
  - Un même fichier glade peut contenir plusieurs fenêtres
  - Il est possible de charger des chaînes de caractères (`add_from_string`)

```
Glib::RefPtr<Gtk::Builder> refBuilder = Gtk::Builder::create();
try {
      refBuilder->add_from_file("ProjetOCI.glade");
} catch(const Glib::FileError& ex) {
      std::cerr << "File Error: " << ex.what() << std::endl;
      return EXIT_FAILURE;
} catch(const Gtk::BuilderError& ex) {
      std::cerr << "GtkBuilder Error: " << ex.what() << std::endl;
      return EXIT_FAILURE;
}
```


Le chargeur d'interface Gtk::Builder

- Création d'un builder et chargement d'un fichier Glade
- Le chargement des fichiers Glade est dynamique
  - Pas de compilation nécessaire
  - Il est possible de modifier le fichier Glade pour modifier l'apparence générale de l'application et de relancer directement celle-ci (sans compilation)
- Deux types de fichiers Glade (version 2 et 3) non compatibles
  - utiliser `gtk_builder-convert` au besoin !

Le chargeur d'interface Gtk::Builder

- Fonction `get_widget`
  - Chargement d'un widget « pur »
  - Il est nécessaire de connaître le nom du widget qu'on souhaite charger et son type
  - Le widget désigné par un pointeur est construit et initialisé par le Builder

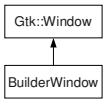
```
Gtk::Button *_button;
refBuilder->get_widget("button1", _button);
if(!_button) {
    std::cerr << "MainDrawingWindow: Button not found" << std::endl;
    exit(1);
}
_button->signal_clicked()
.connect( sigc::mem_fun(this,&BuilderWindow::on_button_clicked) );
```


Le chargeur d'interface Gtk::Builder

- Fonction `get_widget`
  - Chargement d'un widget « pur »
  - Il est nécessaire de connaître le nom du widget qu'on souhaite charger et son type
  - Le widget désigné par un pointeur est construit et initialisé par le Builder
  - Tous les widgets dont vous avez besoin dans le code doivent être chargés
    - Pour faire des connexions (*cf.* exemple précédent),
    - Pour modifier leur apparences depuis le code (par exemple, chargement des valeurs d'un ComboBox).

Le chargeur d'interface Gtk::Builder

- Fonction `get_widget_derived`
  - Chargement de widget pour initialiser un objet d'une classe héritée d'une classe de widget
  - Exemple : chargement d'une fenêtre partiellement définie par Glade (à finaliser avec le code)
  - La classe héritée doit implémenter un constructeur ayant un profil similaire à l'exemple ci-dessous
    - Ce constructeur est appelé par `get_widget_derived`
    - Il n'est pas possible d'y ajouter des paramètres !!




```
BuilderWindow(BaseObjectType* , const Glib::RefPtr<Gtk::Builder>&);
```


Plan de cours

1. Vision d'ensemble
2. Gestion des évènements
3. Présentation des widgets
4. Dessins dans Gtkmm : Gdk + Cairo
5. Drag'n Drop
6. Conception d'interfaces avec Glade
7. Menus et menus contextuels

Les menus et barre d'outils

- Les menus et les icônes d'une barre d'outils sont associés à des `Gtk::Action`
- Pour créer un menu, il faut créer un groupe d'actions `Gtk::ActionGroup`
- Le groupe d'actions est *géré* par le UIManager
  - Disposition dans le menu
  - Affichage dans la barre d'outils

Les menus et barre d'outils

- Création d'un groupe d'actions
  - Utilisation de la fonction `create()`

```
Glib::RefPtr<Gtk::ActionGroup> m_refActionGroup =
Gtk::ActionGroup::create();


m_refActionGroup->add( Gtk::Action::create("MenuFile", "_File") );
```

  - Ajoute une action identifiée par « MenuFile » et qui affiche le texte « File »
  - C'est un menu racine : contient des sous-menus et n'est pas associé à une action
  - Mnémonic : « _ » pour les touches d'accès rapide

Les menus et barre d'outils

- Création d'un groupe d'actions
  - Utilisation de la fonction `create()`

```
Glib::RefPtr<Gtk::ActionGroup> m_refActionGroup =
Gtk::ActionGroup::create();


m_refActionGroup->add( Gtk::Action::create("MenuFile", "_File") );
m_refActionGroup->add( Gtk::Action::create("ExportData", "Export Data"),
sigc::mem_fun(*this, &ExampleWindow::on_action_file_open) );
```

  - Création d'une « vraie » action
    - Associe l'action à un handler : action à réaliser lors du clic !

Les menus et barre d'outils

- Création d'un groupe d'actions
  - Utilisation de la fonction `create()`

```
Glib::RefPtr<Gtk::ActionGroup> m_refActionGroup =
Gtk::ActionGroup::create();


m_refActionGroup->add( Gtk::Action::create("MenuFile", "_File") );
m_refActionGroup->add( Gtk::Action::create("ExportData", "Export Data"),
sigc::mem_fun(*this, &ExampleWindow::on_action_file_open) );
m_refActionGroup->add( Gtk::Action::create("New", Gtk::Stock::NEW),
sigc::mem_fun(*this, &ExampleWindow::on_action_file_new) );
m_refActionGroup->add( Gtk::Action::create("Quit", Gtk::Stock::QUIT),
sigc::mem_fun(*this, &ExampleWindow::on_action_file_quit) );
```

  - `Gtk::Stock` : standardisation et internationalisation dans Gtkmm

Les menus et barre d'outils

- UIManager
  - `insert_action_group` : associe les actions à gérer
  - `add_accel_group` : ajout de la gestion, par la fenêtre, des touches de raccourcis du menu

```
Glib::RefPtr<Gtk::UIManager> m_refUIManager = Gtk::UIManager::create();
m_refUIManager->insert_action_group(m_refActionGroup);
add_accel_group(m_refUIManager->get_accel_group());
```

Les menus et barre d'outils

- UIManager

```
Glib::ustring ui_info =
    "<ui>"
    " <menubar name='MenuBar'>"
       " <menu action='MenuFile'>"
          " <menuitem action='New'/>"
          " <menuitem action='Open'/>"
          " <menuitem action='ExportData'/>"
          " <separator/>"
          " <menuitem action='Quit'/>"
       " </menu>"
       " <menu action='MenuEdit'>"
          " <menuitem action='Cut'/>"
          " <menuitem action='Copy'/>"
          " <menuitem action='Paste'/>"
       " </menu>"
    " </menubar>"
    " <toolbar name='ToolBar'>"
       " <toolitem action='Open'/>"
       " <toolitem action='Quit'/>"
    " </toolbar>"
    "</ui>";
m_refUIManager->add_ui_from_string(ui_info);
```

Les menus et barre d'outils

- Réalisation du menu et de la barre d'outil
- Les menus et les barres d'outils s'insèrent comme les autres widgets
  - La fenêtre doit pouvoir contenir plusieurs widgets
    - utilisation usuelle d'un VBox
    - possibilité de mettre votre barre de menu n'importe où ...

```
Gtk::Widget* pMenubar = m_refUIManager->get_widget("/MenuBar");
m_Box.pack_start(*pMenuBar, Gtk::PACK_SHRINK);
Gtk::Widget* pToolbar = m_refUIManager->get_widget("/ToolBar") ;
m_Box.pack_start(*pToolbar, Gtk::PACK_SHRINK);
```

Plan de cours … la suite

1. Vision d'ensemble
2. Gestion des évènements
3. Présentation des widgets
4. Dessins dans Gtkmm : Gdk + Cairo
5. Drag'n Drop
6. Conception d'interfaces avec Glade
7. Menus et menus contextuels
8. Création de signaux personnalisés
9. Module dynamique : mise en place d'un système de plugins