Introduction à gtkmm 2.24

Plan de cours

1. Vision d'ensemble
2. Gestion des évènements
3. Présentation des widgets

Ce qui ne sera pas traité *(en particulier)*

- Gtk+ (version C)
- Gtkmm 3.0 ...
- Les détails des librairies sœurs
  - la Glib / Glibmm : Gestion de mémoire, types particuliers, entrées-sorties, ...
  - Cairo, Pango
  - Détails de la libsigc
- Gestion des erreurs et du logging
- Internationalisation
- Les Tree View (pas de MVC en Gtk !)
- Le parsing XML
- *Comment créer une bonne interface ...*

Vision d'ensemble

gtkmm

- Binding C++ de Gtk (Gtkmm = Gtk--)
  - Gtk+ : en C mais basé sur des concepts objet
- Ensemble de librairies pour la programmation d'interfaces graphiques
- Développé et maintenu par *GNOME Foundation*
- Cross-plateform
- Sous licence LGPL 2.1
- http://www.gtkmm.org
  - Téléchargements
  - Documentations
  - Qqs tutoriaux

Gtkmm vs Qt (by Gtkmm developers)

Gtkmm does things in a more C++ way.

- Qt has it's own Qt-specific containers, gtkmm uses std::string, std::list, std::vector, iterators, etc. It therefore duplicates a lot of stuff that is now in the standard library, such as containers and type information.
- Gtkmm was able to use standard C++ to provide signals without changing the C+ + language. gtkmm uses pure C++. Qt requires extensions to C++ that are parsed by the moc pre-processor. Most significantly, they modified the C++ language to provide signals, so that Qt classes can not be used easily with non-Qt classes.
- With gtkmm normal C++ memory management can be used. Qt demands that all widgets are dealt with as pointers, and that deletion of widgets is surrendered to parent widgets.

Arrangement of widgets seems to be simpler in gtkmm. In Qt, Containers and Layouts are separate classes, and child widgets must be added to both.

The **gtkmm API tends to be more explicit**. The behaviour of Qt classes is often dependent upon the implicit effects of confusingly-overridden constructors.

Also, Gtkmm and the other *mm modules allow you to build software which **works more closely with the GNOME desktop**.

Court historique

- 1997 : gtk+ 1.0 (Mattis, Kimball & MacDonald)
  - U. Californie, Berkeley
  - Projet initié par la conception de GIMP
  - GTK = GIMP Tools Kit
- 1998 : gtk+ 1.2 et gtkmm 1.2
- 1998 : gtk+ 2.0 (620,000 lignes) et gtkmm2.0
  - Pango, GdkPixbuf, ATW
  - Nouveaux Widget (Texte et Tree),
- 2005 : Apparition de Cairo 1.0
- 2011 : Gtkmm 3.0

Situation actuelle

- Communauté de développement de Gtk active
- Toujours utilisée par GIMP
- Librairie graphique utilisée pour GNOME, Xcfe
- Encore assez largement utilisée

- Version stable actuelle (3.2.0)
  - Peu de modifications par rapport à la 2.24 (version éprouvée)
  - Version 3.3.2 en développement

Librairies usuelles

- Gtknn ne réimplémente pas les structures usuelles de programmation objet
  - Différence avec Qt
- Utilise la librairie standard C++ (STL)
  - Container (`vector`, `list`, `map`, ...)
  - `Iterator` et `algorithm`
  - **Sauf pour les** String (utiliser les `Glib:ustring` !!)
    - Détail qui a son importance : l'un des rares défaut de STL est de ne pas gérer l'UTF-8 (accents, etc...) !
    - Toutes les librairies graphiques proposent leur version des string !

Librairies associées de Gtkmm

- Pango/Pangomm
  - Gestion de l'affichage du texte
- Gdk
  - Primitives de dessin
- Cairo / Cairomm
  - Librairie de dessin vectoriel

- Facile de s'y retrouver
  - Champs de compétence bien délimités
  - Chaque librairie utilise son propre `namespace`

Organisation des librairies

pkg-config

- `pkg-config` : outil de gestion des dépendances entre les librairies installées sur le système (Linux)
- La description de ces librairies se trouvent dans le dossier :
  `/usr/lib/pkgconfig`
- La commande suivante affiche tous les includes et librairies nécessaires à la compilation d'un fichier utilisation Gtkmm :

```
pkg-config --libs --cflags gtkmm-2.4
```

pkg-config

- Très utile pour faciliter l'écriture des commandes de compilation sans oublier d'includes
- Compilation :

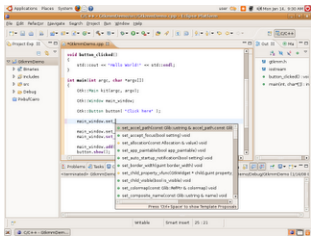
```
gcc -c `pkg-config --flagc gtkmm2.4` -o
  fichier.o fichier.c
```

- Édition de liens :

```
gcc `pkg-config --libs gtkmm2.4` -o main
  fichier.o
```

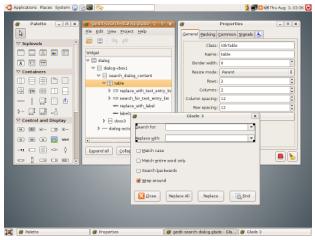
Outils de développement

- Interfaçage avec Eclipse CDT
  - *cf.* tutoriel en ligne pour la création d'un projet C++ sous Eclipse avec gestion des librairies Gtkmm.
    - Pour Helios
    - Ne fonctionne pas bien pour Indigo
  - Facilité de codage de Eclipse
    - Auto-complétion
- Possibilité de passer par CMake
  - Multi-IDE



Outils de développement

- Glade
  - http://glade.gnome.org/
  - Interface graphique pour créer simplement des interfaces GTK+
  - Existe pour Windows
  - Alternative possible : Gazpacho
- GObject Builder
  - préprocesseur qui permet de construire des objets GObject



Documentation

- En ligne :
  - http://www.gtkmm.org/en/documentation.html
    - Documentation générale
    - Exemples
    - Liens vers les autres libriaries (SigC)
  - API : http://developer.gnome.org/gtkmm/
  - Book « **Programming with gtkmm** »
    - http://developer.gnome.org/gtkmm-tutorial/stable/
    - Traduction Française partielle

Autres binds de Gtk+ (v3.0)

- C++ : GtkMM
- C# : Gtk#
- Java : Java-Gnome
- Python : PyGTK
- PHP : PHP-Gtk (partiel, v2.16)
- Ada, OCaml, R, Perl, ...

http://www.gtk.org/language-bindings.php

Portabilité

- Peut être porté sur tout OS pouvant faire fonctionner un serveur X (et disposant d'une version convenable de la XLib)
  - Sous GNU/Linux
  - Sous Windows
    - Avec MinGW
    - Compatibilité partielle avec MSVC (2005 ou +)
  - Sous MacOS X

Licence

- GTK+ est un projet libre : GNU LGPL 2.1 (Library General Public licence)
- Les codes utilisant GTK+ ne doivent pas obligatoirement être licenciés GPL !
  - La LGPL exige uniquement de conserver la trace des personnes ayant contribuer au code
  - Valable aussi bien pour l'utilisation des librairies que du code source
- Licence possible maintenant pour Qt 4.5 !

Introduction à Gtkmm

1. Historique et position actuelle de Gtkmm
2. Organisation des librairies de Gtkmm
3. Outils de développement et binds
4. ***Les basiques***
5. Conventions de programmation de Gtkmm

Éléments basiques 1 : les widgets

- Un application Gtkmm est constituée de Fenêtres (windows) contenant des éléments d'interfaces (widgets) tels que :
  - boutons
  - boîtes de textes
  - ...
- Chaque type de widget est représenté par une classe
- Les widgets sont organisés graphiquement
  - en ligne ou en colonne de widgets

Éléments basiques 2 : les signaux

- Gtkmm est basé sur les concepts de signal et de callback (ou handlers)
  - des signaux peuvent être émis par les widgets lors d'évènements comme un clic de souris
  - chaque widget a ses propres signaux qu'il peut émettre (dans la limite des signaux X !)
  - les signaux peuvent être rattrapés pour être traités (*signal handlers*)
- Les signaux de Gtkmm sont basés sur libsigc++

```
m_button1.signal_clicked().connect( sigc::mem_fun(*this,
&HelloWorld::on_button_clicked) );
```

Éléments basiques 2 : les signaux

- Comparaison avec Java/Swing
  - Java/Swing : principe de Listeners

```
m_button1.addActionListerner( new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      system.out.println('coucou');
   }
});
```

  - Gtkmm : un handler est une fonction qui implémente un comportement à réaliser

```
m_button1.signal_clicked().connect( sigc::mem_fun(*this,
&HelloWorld::button_clicked) );

void HelloWorld::button_clicked() {
   cout << 'coucou' << end;
}
```

Conception d'une application Gtkmm

1. Conception de fenêtres et de *widgets* par la combinaison des *widgets* existants
   - Définition de la forme de l'interface
2. Conception des *signal handlers*
   - Définition des comportements potentiels de l'interfaces
3. Connexions des signaux et *signal handlers*
   - Description du comportement de l'interface en fonction des évènements (actions de l'utilisateur)

Éléments basiques 3 : 1er programme

```
#include <gtkmm.h>
int main(int argc, char *argv[])
{
    Gtk::Main kit(argc, argv);
    Gtk::Window window;

    return Gtk::Main::run(window);
}
```

- `Gtk::Main kit(argc, argv)` : Objet `Gtk::Main` indispensable pour créer une application Gtkmm
  - Initialise Gtkmm
  - Lit les arguments destinées aux appli. X (*e.g.* options -display, -geometry)
- `Gtk::Window window` : Créer un objet fenêtre
- `Gtk::Main::run(window)` : Exécute la fenêtre comme fenêtre principale

Première classe d'une « interface »

```
#include <gtkmm.h>
int main(int argc, char *argv[])
{
    Gtk::Main kit(argc, argv);
    MaFenetre window;


    return Gtk::Main::run(window);
}
```

Première classe d'une « interface »

```
maFenetre::maFenetre() : Gtk::Window()
{
    set_title('Ma fenetre');
    Gtk::HBox m_box;
    window.add(m_box);
    Gtk::Label m_label('Cliquer pour quitter');
    Gtk::Button m_button('Quitter');
    m_box.pack_start(m_label);
    m_box.pack_start(m_button);

    m_button.signal_clicked().connect(sigc::ptr_fun(&on_button
    _clicked));
    m_label.show();
    m_button.show();
}

void maFenetre::on_button_clicked()
{
  ...
}
```

Introduction à Gtkmm

1. Historique et position actuelle de Gtkmm
2. Organisation des librairies de Gtkmm
3. Outils de développement et binds
4. Les basiques
5. ***Conventions de programmation de Gtkmm***

Conventions de nommage

- Nom des classes :
  - Pas de préfixe (préfixé par le namespace!)
    - ex: `Button` ou **`Gtk::Button`**
    - *Attention :* `GtkButton` réfère à la *structure* C d'un bouton dans la librairie Gtk+
  - Chaque « mot » commence par une majuscule
    - ex: `Button`, `TextView`, `VBox`, `Viewport`, ...
- Nom des membres
  - Commencent par « `m_` »
  - Mots avec majuscules, séparés par des « _ »
  - ex : `m_Label_Value(''Value :'');`
    - Label contenu dans un widget

Conventions de nommage

- Méthodes
  - Pas de majuscule
  - Mots séparés par des « _ »
- Getter et setter
  - set : Commence par le mot clé « `set` »
    - ex. `void set_label (const Glib::ustring& label)`
  - get : Commence par le mot clé « `get` »
    - ex. `Glib::ustring get_label() const`
- Signaux : identifiés par le mot clé « `signal` »
  - ex: `signal_clicked()`

Conventions de nommage

- Les handlers : commencent par le mot clé « on »
  - ex : `virtual void on_clicked()`
  - Fonctions connectés aux signaux correspondants
  - Fonctions déclarées comme protégées
    - Elles peuvent être surchargées lors de la conception de ses propres widgets qui héritent des widgets par existants
- Propriétés de l'interface (GLib) : commencent par le mot clé « `property` »
  - ex : `Glib::PropertyProxy_ReadOnly<bool> property_focus_on_click() const`
  - Fonctions rarement utiles !
  - Préférer l'usage des `set` et des `get` !

Les includes

- `#include <gtkmm/(widget).h>` par exemples :
  - `#include <gtkmm/button.h>`
  - `#include <gtkmm/window.h>`
  - Sans majuscule !
- Attention aux inclusions provenant des librairies sœurs
  - `#include <gdkmm/cursor.h>`
- `#include <gtkmm.h>`
  - Inclut tous les widgets et les librairies sœeurs
  - Évite de se poser des questions d'inclusion

Plan de cours

1. Vision d'ensemble
2. Gestion des évènements
3. Présentation des widgets

Gestion des évènements

1. La boucle d'évènements
2. Les évènements d'affichage
3. Surcharger les handlers
4. Connexion des signaux
5. Activer les signaux X masqués

La boucle d'évènements

- Gtkmm implémente une boucle de gestion des évènements
  - **Correspond au thread de l'interface**
  - Dans la classe `Gtk::Main`
  - Lancée par `Gtk::Main::run()`

```
int main(int, char **)
{
    Gtk::Main app();
    app.run();
    return 0;
}
```

```
int main(int argc, char *argv[])
{
    Gtk::Main kit(argc, argv);
    Gtk::Window window;

    return Gtk::Main::run(window);
}
```

La boucle d'évènements

- Exemple d'implémentation de la boucle d'évènements

```
while( program == running ) {
  while(not event_table.empty ()) {
    const Event &e = event_table.pop();
    switch ( e.type ()) {
    case Event::BUTTON_CLICKED :
   //Action à réaliser
      ...
    case Event::NEED_REDRAW :
      ...
    default :
  }
}
```

La boucle d'évènements

- Les évènements sont traités au travers des notions de :
  - **signal** : un objet peut « émettre » un évènement décrit par un signal
    - action d'un utilisateur (*e.g.* souris ou clavier)
    - envoi programmé (*e.g.* timer)
  - **callback** (ou **handler**) : un objet peut être déclenché par des évènements, le callback est une fonction indique comment réagir !
- **Connexion** : le programmeur associe un signal à un callback

La boucle d'évènements

- Table de connexions (pour se faire une *vague idée* du fonctionnement)

Objet	Évènement	Action (callback)
Application	Cacher	exit()
Button1	Pressed	fonction1()
Button1	Released	fonction2()
Button2	Pressed	fonction3()
Button2	Pressed	fonction4()
...		

La boucle d'évènements

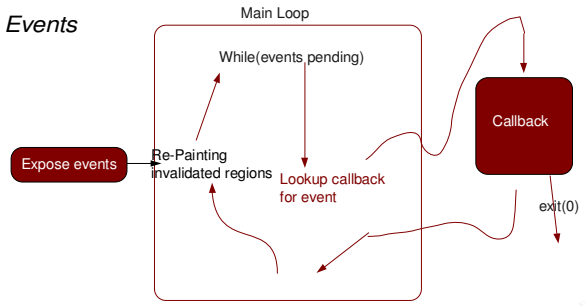


Schéma extrait de Hacking GTK+, -Muthiah A, Octave-Gtk+ team

La boucle d'évènements : 2 piles

- **Une pile d'évènements** (FIFO) : entasse et se dépile au fur et à mesure des demandes de l'utilisateur ou du programme
  - Les signaux créent de nouveaux évènements,
  - Les callbacks les consomment
- **Une *pile* d'évènements d'affichage** (`expose events`)
  - Retient les zones de l'écran à rafraichir,
  - Gestion intelligente : évite des doublons, mutualisation des régions !
- Gtkmm assure la distribution des évènements

Gestion des évènements d'affichage

- En fonction de ce qui se passe dans l'IHM, Gtkmm détermine quels sont les zones à afficher
  - Ajout dans une queue de zones à afficher
  - Chaque zone correspond à un widget
- Lors du traitement d'une zone à afficher Gtkmm fait appel à la fonction `on_expose_event` de la classe `Gtk::Widget`
- En surchargeant la fonction `on_expose_event`, on peut ajouter ou redéfinir l'affichage d'un widget

Gestion des évènements d'affichage

```
bool ZoneDessin::on_expose_event(GdkEventExpose* event)
{
    Glib::RefPtr<Gdk::GC> gc = Gdk::GC::create(get_window());

    Gdk::Color bleu("blue");
    get_default_colormap()->alloc_color( bleu );
    Gdk::Color rouge("red");
    get_default_colormap()->alloc_color( rouge );

    gc->set_foreground(bleu);
    get_window()->set_background(rouge);

    get_window()->clear();

    get_window()->draw_line(gc, 1, 1, 100, 100);
    get_window()->draw_layout(gc, 40, 40, create_pango_layout("Bonjour"));

    return true;
}
```

Gestion des évènements

1. La boucle d'évènements
2. Les évènements d'affichage
3. Connecter des évènements
   1. Surcharger les handlers
   2. Connexion des signaux
4. Activer les signaux X masqués

Connexion d'un évènement

- La connexion des évènements consiste à faire des ajouts dans la « table des connexions »
- Deux méthodes
  - Surcharger les handlers
  - Établir des connexions

Surcharger les handlers

- Les widgets disposent déjà de handlers prédéfinis et connectés par défaut
  - Fonctions identifiées par le préfixe **`on_`**
  - Les fonctions sont **`virtual protected`**
  - Chaque handler est associé a un évènement du widget
  - Exemples `Gtk::Button` :

```
virtual void on_pressed ()
virtual void on_released ()
virtual void on_clicked ()
virtual void on_enter ()
virtual void on_leave ()
virtual void on_activate ()
```

- Il faut surcharger ces fonctions pour implémenter l'action désirée
  - Nécessite de construire une classe spécifique pour chaque bouton qui hérite de `Gtk::Button`

Surcharger les handlers

- `on_clicked()` est appelé suite à un évènement X

```
#include <gtkmm/main.h>
#include <gtkmm/button.h>
#include <gtkmm/window.h>

class myButton : public Gtk::Button
{
  public:
    myButton(){};
    virtual ~myButton(){};

  protected:
    //Surcharge du handler :
    virtual void on_clicked();
};

void myButton::on_clicked()
{
  printf("Clic!\n");
}
```

```
int main(int argc, char *argv[])
{
  Gtk::Main kit(argc, argv);

  Gtk::Window win;
  win.set_title("buttonExample");
  win.set_border_width(10);

  myButton button;
  win.add(button);
  button.show();

  Gtk::Main::run(win);

  return 0;
};
```

Les signaux

```
Glib::SignalProxy0< void > signal_pressed ()
Glib::SignalProxy0< void > signal_released ()
Glib::SignalProxy0< void > signal_clicked ()
Glib::SignalProxy0< void > signal_enter ()
Glib::SignalProxy0< void > signal_leave ()
Glib::SignalProxy0< void > signal_activate ()
```

- Les signaux
  - Un « signal » est une **fonction** :
    - caractérise un type d'évènement qui peut être émis
    - **n'envoie pas l'évènement lorsqu'on l'appelle**
  - Chaque classe de widget dispose de ses propres signaux
  - Identifiés par le préfixe `signal_`
  - On retrouve les mêmes que pour les handlers prédéfinis
  - Exemples ci-dessus : signaux de `Gtk::Button`
- **Il ne faut par faire appel aux signaux,** ils servent à un usage interne pour déclarer des connexions !
- **Les signaux doivent être connectés à des callbacks pour déclencher une action**

Les signaux et les callbacks

- Connexions des signaux (sans paramètres)

  <signal>().connect(sigc::ptr_fun(&handler));

Exemple

```
void on_button_clicked() {
   std::cout << "Hello World" << std::endl;
}
main() {
   Gtk::Button b("Hello World");
   b.signal_clicked().connect( sigc::ptr_fun(&on_button_clicked) );
}
```

Rappel sur les pointeurs de fonctions

- Un pointeur de fonction
  - ou functor
  - ou fonction objet

  est une variable qui représente une fonction
- Permet d'abstraire des fonctions
- Exemple

```
int compare_function(int A, int B) {
   return (A < B);
}
...
/* Déclaration d'une fonction */
void sort_ints(int* items, int nb, int (*cmpfunc)(int, int) );
...
int main() {
    int items[] = {4, 3, 1, 2};
    sort_ints(items, sizeof(items)/sizeof(int), compare_function);
}
```

Pointeur de fonction qui retourne un entier et prend deux paramètres entiers

Rappel sur les pointeurs de fonctions

- Utilisation dans les librairies standards
- En plus, il y a un template !

```
template <class Compare>
  void sort ( Compare comp );
```

http://www.cplusplus.com

```
using namespace std;

// comparison, not case sensitive.
bool compare_nocase (string first, string second)
{
  unsigned int i=0;
  while ( (i<first.length()) && (i<second.length()) )
  {
    if (tolower(first[i])<tolower(second[i]))
      return true;
    else if (tolower(first[i])>tolower(second[i]))
      return false;
    ++i;
  }
  if (first.length()<second.length()) return true;
  else return false;
}

int main ()
{
  list<string> mylist;
  list<string>::iterator it;
  mylist.push_back ("one");
  mylist.push_back ("two");
  mylist.push_back ("Three");

  mylist.sort(compare_nocase);

  cout << "mylist contains:";
  for (it=mylist.begin(); it!=mylist.end(); ++it)
    cout << " " << *it;
  cout << endl;

  return 0;
}
```


Les signaux et les callbacks

- Connexions des signaux (sans paramètres)

  <signal>().connect(sigc::ptr_fun(&handler));

Exemple

```
void on_button_clicked() {
   std::cout << "Hello World" << std::endl;
}
main() {
   Gtk::Button b("Hello World");
   b.signal_clicked().connect( sigc::ptr_fun(&on_button_clicked) );
}
```

Il s'agit d'un pointeur sur la fonction on_button_clicked

Les signaux et les callbacks

- La fonction `connect()` prend en paramètre un `slot`
  - `slot` : représentation interne d'un pointeur de fonction
  - Exemple de type : `sigc::slot< bool, double >`
    - Fonction avec un paramètre de type double, et qui retourne un booléen
- La fonction `sigc::ptr_fun` construit un *slot* à partir d'une fonction
  - On donne simplement un pointeur de fonction (*sans parenthèses avec un & devant !*)
  - Exemple :

```
void foo(int)
{
}

sigc::slot<void, int> sl = sigc::ptr_fun(&foo);
```


Les signaux et les callbacks

- Connexions des signaux (sans paramètres)

  <signal>().connect(sigc::ptr_fun(&handler));

Exemple

```
void on_button_clicked() {
   std::cout << "Hello World" << std::endl;
}
main() {
   Gtk::Button b("Hello World");
   b.signal_clicked().connect( sigc::ptr_fun(&on_button_clicked) );
}
```


Connexion à une méthode

- sigc::ptr_fun(&handler) : sert uniquement à connecter une fonction globale
- sigc::mem_fun(objet, &handler) : sert à connecter une méthode d'une classe pour un objet particulier
  - Utilisation identique :

```
void MaClasse::action() {
   std::cout << "Hello World" << std::endl;
}

void main() {
   MaClasse obj, obj2;
   Gtk::Button b("Hello World");
   b.signal_clicked().connect( sigc::mem_fun(obj,&MaClasse::action) );
}
```

Connexion à une méthode

```
void on_button_clicked();

class some_class {
   void on_button_clicked();
};

some_class some_object;

main() {
   Gtk::Button b;
   b.signal_clicked().connect( sigc::ptr_fun(&on_button_clicked) );
   b.signal_clicked().connect( sigc::mem_fun(some_object,
&some_class::on_button_clicked) );
}
```

Connexion des signaux aux callbacks

- Les détails techniques permettent de bien comprendre ce qui se passe mais peuvent être oubliés dans la pratique d'utilisation de Gtkmm
- À retenir, connexions des signaux (sans paramètres)

  <signal>().connect(sigc::ptr_fun(&handler));

  <signal>().connect(sigc::mem_fun(object, &handler));

Exemple d'appel à un signal

```
class myButton : public
Gtk::Button {
  public:
    myButton(){};
    virtual ~myButton(){};

  public:
    void on_button_enter();
  protected:
    virtual void on_clicked();
};

void myButton::on_button_enter(){
  printf("enter!\n");
  signal_clicked();
};

void myButton::on_clicked() {
  printf("Click!\n");
};
```

```
int main(int argc, char *argv[])
{
  Gtk::Main kit(argc, argv);

  Gtk::Window win;
  win.set_title("buttonExample 3");
  win.set_border_width(10);

  myButton button;
  win.add(button);

  button.signal_enter().connect(
    sigc::mem_fun(button,
    &myButton::on_button_enter)
    );
  button.show();

  Gtk::Main::run(win);
  return 0;
};
```

Ne marche pas !

Mettre on_clicked() à la place !!

Signaux/callbacks avec paramètres

- Certains signaux retournent des valeurs ou ont des paramètres
- Les handlers doivent les gérer
- La fonction `connect()` exige la correspondance entre les profils de fonctions (*type safety* à la compilation)

  <signal>().connect(sigc::ptr_fun(&handler));

Profil attendu décrit dans le signal proxy

Profil du handler construit par ptr_fun

Signaux/callbacks avec paramètres

- La fonction `connect()` exige la correspondance entre les profils de fonction

  <signal>().connect(sigc::ptr_fun(&handler));
- Exemple

```
signal_focus().connect( ptr_fun(&ma_fonction) )


Glib::SignalProxy1<bool, Gtk::DirectionType> signal_focus()

bool ma_fonction(Gtk::DirectionType direction);

ptr_fun(&ma_fonction) : slot<bool, Gtk::DirectionType>
```

Signaux/callbacks avec paramètres

- La fonction `connect()` exige la correspondance entre les profils de fonction

  <signal>().connect(sigc::ptr_fun(&handler));
- Exemple

```
signal_focus().connect( ptr_fun(&ma_fonction) )


Glib::SignalProxy1<bool, Gtk::DirectionType> signal_focus()

bool ma_fonction();

ptr_fun(&ma_fonction) : slot<bool>
```

Les profils de fonction ne correspondent pas !

Signaux/callbacks avec paramètres

- De même avec plusieurs paramètres (maximum 7)
- exemple de signal `Gtk::editable`

```
Glib::SignalProxy3<void, const Glib::ustring&, int, int*>
signal_insert_text()
```

```
void on_insert_text(const Glib::ustring& text, int length, int*
position);
```

Signaux/callbacks avec paramètres

- Pour forcer la correspondance entre profils, il faut parfois utiliser des fonctions de *binding*
  - `sigc::bind` : ajouter des paramètres au handler
  - `sigc::hide` : masquer des paramètres du handler
  - `sigc::bind_return` : ajouter un valeur de retour au handler
  - `sigc::hide_return` : masque le type de retour (remplacé par `void`)

Signaux/callbacks avec paramètres

- Exemple d'utilisation de la fonction `sigc::bind()`
  - Fonction : `virtual void mon_handler(Glib::ustring data);`
  - On veut la déclencher par un clic de bouton : `Glib::SignalProxy0<void> signal_clicked()`

```
m_button.signal_clicked().connect(
 sigc::bind<Glib::ustring>( sigc::mem_fun(*this, &mon_handler)
                          , Glib::ustring("button 1") )
);
```

Valeur du paramètre supplémentaire

cf. exemple 4 : `buttonExample_bind`

Signaux/callbacks avec paramètres

- Exemple d'utilisation de la fonction `hide_return()`
  - Il faut que le type de retour de la fonction corresponde également
  - handler : `bool mon_handler();`
  - signal : `Glib::SignalProxy0<void> signal_clicked()`
  - Connexion à réaliser

```
m_button.signal_clicked().connect(
     sigc::hide_return( sigc::mem_fun(*this, &mon_handler) )
);
```

cf. exemple 4 : `buttonExample_bind`

Déconnection d'un signal

- Pour déconnecter un signal, il suffit de faire appel à la fonction `disconnect()` d'une connexion
  - Nécessite d'avoir conserver une trace de la connexion quelque part !
- Les signaux peuvent être connectés et déconnectés **dynamiquement**

```
sigc::connection connexion = button_test.signal_clicked().connect(
    sigc::ptr_fun( &test_connection )
);

(...)

connexion.disconnect();
```

cf. exemple `buttonExample_dynamicdisconnection`

Avantages et inconvénients

- Surcharge des handlers
  - Avantage : facile et rapide
  - Inconvénients :
    - nécessite de surcharger la classe de base pour chaque nouveau comportement
    - solution nécessitant une forte dépendance entre la classe de l'objet appelant et celle de l'objet appelé
- Connexion de signaux
  - Avantages :
    - générique
    - forte séparation appelant/appelé possible
  - Inconvénients :
    - les binds de functors
    - peut nécessiter un peu plus de travail de réflexion préalable !

Les signaux et les callbacks

Résumé

- Chaque Widget est associé à des évènements
- un évènement
  - une fonction **`signal_XXX()`** : sert à connecter des actions aux évènements
  - une fonction **`on_XXX( params )`** : peut être surchargée
- Utilisation de la fonction `connect()`

```
signal_XXX().connect( mem_fun( object, &Class::function ) )
```

  - Les fonctions de binding (`sigc::bind()`) servent à modifier les paramètres de la fonction à appeler pour correspondre au profil du signal !
  - La fonction appelée doit récupérer tous les paramètres du signal
- Il est possible de connecter et déconnecter dynamiquement les signaux

Gestion des évènements

1. La boucle d'évènements
2. Les évènements d'affichage
3. Connecter des évènements
   1. Surcharger les handlers
   2. Connexion des signaux
4. Activer les signaux X masqués

Les signaux X usuels

- Dans la classe `Gtk::Widget` : tous les widgets peuvent, en principe, les utiliser
- Les signaux des évènements X (repérés par `_event`)

```
Glib::SignalProxy1<bool, GdkEvent* > signal_event()
Glib::SignalProxy1<bool, GdkEventButton* > signal_button_press_event()
Glib::SignalProxy1<bool, GdkEventButton* > signal_button_release_event()
Glib::SignalProxy1<bool, GdkEventKey* > signal_key_press_event()
...
```


Les signaux X usuels

- Les signaux d'affichage

```
Glib::SignalProxy1< bool, GdkEventExpose* > signal_expose_event()
Glib::SignalProxy1< bool, GdkEventExpose* > signal_configure_event()
Glib::SignalProxy0<void> signal_realize()
Glib::SignalProxy0<void> signal_hide()
Glib::SignalProxy0<void> signal_show()
```

- Les signaux de Drag & Drop

```
Glib::SignalProxy1<void, const Glib::RefPtr<Gdk::DragContext>&> signal_drag_begin()
Glib::SignalProxy1<void, const Glib::RefPtr<Gdk::DragContext >&> signal_drag_end()
Glib::SignalProxy2<void, const Glib::RefPtr<Gdk::DragContext>&,guint >
  signal_drag_leave()
(...)
```


Activation des signaux X usuels

- Utilisation par défaut des signaux X
  - Dans un certain nombre de widget, les signaux X sont déjà utilisés pour les signaux propres à l'objet
  - Pour les autres, ils ne sont pas écoutés par la boucle de Gtk
- Récupérer les signaux X d'un widget
  - Dans le premier cas, on peut utiliser un `EventBox`, et indiquer les signaux à écouter
  - Dans le second, il suffit (mais nécessaire) d'indiquer les signaux à écouter

Activation des signaux X usuels

- Exemple : activation des clics de souris sur un `Gtk:DrawingArea`

```
Gtk::DrawingArea widg;
widg.add_events(Gdk::BUTTON_PRESS_MASK | Gdk::BUTTON_RELEASE_MASK);

widg.signal_button_press_event().connect( ... );
widg.signal_button_release_event().connect( ... );
```

Exemple : rendre un label cliquable

```
bool on_button_clicked(GdkEventButton*)
{
   std::cout << "fonction clicked\n";
};
Gtk::EventBox m_EventBox;
fenetre.add(m_EventBox);
Gtk::Label m_Label("Mon label");
m_EventBox.add(m_Label);
m_Label.set_size_request(110, 20);

//Préparation pour l'action (a faire avant l'affichage pour éviter les
erreurs)
m_EventBox.set_events(Gdk::BUTTON_PRESS_MASK);

(...)

//Connexion : sur l'Eventbox !
m_EventBox.signal_button_press_event().connect( sigc::ptr_fun( &on_but
ton_clicked ) );
```

La boucle d'évènements

- Petit exercice de compréhension
  - On propose d'afficher l'évolution d'un traitement en modifiant le label d'un bouton ainsi
  - Que va-t-il se passer en pratique ?

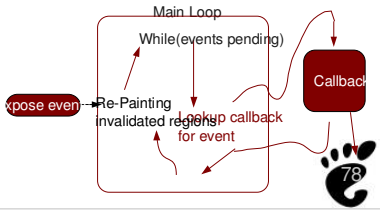
```
void go() {
  long i=LONG_MIN;
  long max= LONG_MAX;
  while( i < max ) {
    double result = cos((double)i*PI/180);
    button->set_label( Glib::ustring::compose("done %1%%",
(double)i/(double)max) );
    i++;
  }
}
```

cf. exemple 5 : `Almostlong.cpp`

La boucle d'évènements

- Forcer le traitement des évènements

```
void go() {
  long i=LONG_MIN/100;
  long max= LONG_MAX/100;
  while( i < max ) {
    double result = cos((double)i*PI/180);
    button->set_label( Glib::ustring::compose("done %1%%",
(double)i/(double)max) );
    while( Gtk::Main::events_pending()) {
      Gtk::Main::iteration( false );
    }
    i++;
  }
}
```



Plan de cours

1. Vision d'ensemble
2. Gestion des évènements
3. Présentation des widgets

Présentation des widgets

- Objectif :
  - Faire une revue rapide, *non-exhaustive* des widgets usuels
  - S'appuyer fortement sur la documentation
    - Pour que je ne recopie pas ce qui y est déjà ! (*principe n°1 de l'informaticien : ne pas refaire ce qui fonctionne bien !*)
    - Pour que vous trouviez facilement les informations dont vous avez besoin
  - Donner quelques exemples pratiques *exemplaires* de l'utilisation de Widgets

Présentation des widgets

1. La classe Widget
2. La hiérarchie des Widgets
3. Présentation de qqs des widgets standard
4. Autres éléments d'une interface

La classe Widget

- Fonction liée à la géométrie du Widget
  - Déterminer la taille d'un widget
    - `void set_size_request (int width=-1, int height=-1)`
    - Les requêtes de taille ne sont pas nécessairement satisfaites !
  - Connaître la taille d'un widget
    - `get_width()` et `get_height()`
    - `Allocation get_allocation () const`
    - `Allocation` : permet de connaître la taille **et la position** du widget (alloué par le windows manager)

La classe Widget

- Modifier le style d'un Widget
  - style = couleurs, polices par défaut, pixmap de fond, ...
  - fonctions `modify_...()`
  - annulé avec la fonction `unset_...()`
- Modifier la couleur de fond d'un Widget
  - `void modify_bg(StateType state, const Gdk::Color& color)`
  - `Gtk::StateType = {STATE_NORMAL, STATE_ACTIVE, STATE_PRELIGHT, STATE_SELECTED, STATE_INSENSITIVE}`
  - `color = Gdk::Color('yellow');` (Voir /usr/share/X11/rgb.txt)
- La modification passe par l'appel d'un évènement X. Le Widget doit pouvoir le recevoir pour que la modification soit effective !

La classe Widget

- Les fonctions d'affichage
  - `void show(), void show_all(), void show_now()`
    - `show()` : Flags a widget to be displayed,
      - **Les widgets qui ne sont pas *flaggés* ne seront pas affichés !!**
    - `show_all()` : Recursively shows a widget, and any child widgets (if the widget is a container),
    - `show_now()` : Shows a widget. -> Rentre dans la boucle principale et attend que le widget soit effectivement affiché. Attention à son utilisation (!)
  - `void queue_draw()` et `void queue_draw_area()`
    - Invalide une région de l'interface
    - **Dans la boucle de traitement**, une fois les évènements traités les régions invalidées reçoivent des évènements `expose_event`

Widget standard

Standard Widgets

- Button
- Label
- Entry
- Radio
- Check
- Menubar
- Menu
- Scrollbar
- Progressbar
- Ruler
- Scale
- Dialogs

Layout Widgets

- Window
- Container
- Box
- Table
- Scrolled Window

Drawing Widgets

- Drawing Area
- Window
- Curve

MVC Widgets

- TreeView
- TextView
- ListView

Misc Widgets

- Pane
- Fixed
- Separators
- Tearoff menu items
- Tooltips
- Preview
- Notebook

Hacking GTK+, -Muthiah A, Octave-Gtk+ team

Fenêtre (`Gtk::Window`)

- La classe `Window` est le widget représentant une fenêtre dans l'environnement graphique.
- C'est la seule classe représentant un « vrai » objet X
  - possède les fonctions en relation avec le *window manager*, comme par exemple iconifier la fenêtre, l'enrouler, etc.
- On ne peut lancer qu'un `Gtk::Window` avec `run()`

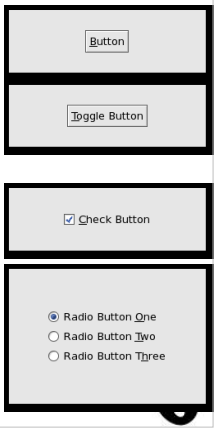
```
int main ( int argc , char ** argv ) {
  Gtk::Main app(argc , argv);
  Gtk::Window w;
  app.run(w);
}
```

Fenêtre (`Gtk::Window`)

- On peut y inclure 1 autre Widget (mais pas 2!)
  - `Gtk::Button button;`
  - `window.add(button);`
- Qqs membres spécifiques
  - `void set_title(const Glib::ustring& title)`
  - `void set_icon (const Glib::RefPtr<Gdk::Pixbuf>& icon)`
  - `void resize(int width, int height)`
    - Pas besoin de passer par la fonction `set_size_request(...)`

Boutons

- Gtk::Button (doc)
  - signal_clicked()
- Gtk::ToggleButton (doc)
  - Bouton qui peut conserver son état
  - toggle() to switch the state
  - void set_active(bool =true)
  - bool get_active()
- Gtk::CheckButton (doc)
- Gtk::RadioButton (doc)



Les mnémonics

- Mnémonic : permet à l'utilisateur d'une interface de déclencher rapidement le bouton par une combinaison de touche
- Exemple: *Alt+B*

- Dans le constructeur :
  - `Button('mon_Button', true)`
- Fonctions `add_label` et `add_mnemonic_label`

ComboBox

- Liste de choix déroulante

- `m_Combo.append_text('...')` : ajoute un item
- `signal_changed()` : signale la modification d'un choix (Sans arguments)
- `Glib::ustring get_active_text()` : récupère le texte de l'item affiché

ComboBox : exemple

```
ExampleWindow::ExampleWindow()
{
  m_Combo.append_text("something");
  m_Combo.append_text("something else");
  m_Combo.append_text("something or other");

  add(m_Combo);

  //Connect signal handler:
  m_Combo.signal_changed().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_combo_changed) );

  show_all_children();
}

void ExampleWindow::on_combo_changed()
{
  Glib::ustring text = m_Combo.get_active_text();
  if(!(text.empty()))
    std::cout << "Combo changed: " << text << std::endl;
}
```

Entry

- Entrée textuelle sur 1 ligne
- Propriétés :
  - `visible` (=`false` : mode password)
  - `editable`
  - `max_length`
- Gestion du contenu textuel :
  - `set_text(Glib::ustring t)`
  - `Glib::ustring get_text()`
- Signaux :
  - `signal_changed()`, `signal_insert_text()`, `signal_delete_text()`
  - `signal_insert_at_cursor()` (retourne la chaîne insérée)
  - `signal_selection_received()`, `signal_selection_ge`

Label

- Le Label est un composant simple qui représente une chaîne de caractères dans une zone graphique.

```
Label (const Glib::ustring& label, AlignmentEnum xalign,
  AlignmentEnum yalign=ALIGN_CENTER, bool mnemonic=false)


Label( 'Bonjour !' );
Label( Glib::ustring('Banzaï') );
```

Label

- Le Label est un composant simple qui représente une chaîne de caractères dans une zone graphique.

```
int main( int argc , char ** argv ) {
  Gtk::Main app(argc , argv);
  Gtk::Window w ;
  Gtk::Label label ;
  label.set_markup("<i>hello</i> <u> world </u> <b>!</b>");
  w.add( label );
  app.run( w );
}
```

Label

- C'est un élément qui ne permet pas de recevoir des évènements X
  - Formellement, il ne s'agit pas d'une fenêtre X
  - exemple (`label_Color.cpp`) :

```
m_label.modify_bg( Gtk::STATE_NORMAL, Gdk::Color('red') )
```

  - Modifie la couleur du fond d'un widget
  - Fait appel à un évènement X pour modifier cette propriété, mais l'évènement n'est pas pris en compte : le label reste gris !!

Solution : les EventBox

- Exemple : modification de la couleur de fond d'un label

```
Gtk::EventBox m_eventbox;
m_eventbox.add(m_label);
m_eventbox.modify_bg(Gtk::STATE_NORMAL, Gdk::Color('red') );
m_eventbox.show();
m_label.show();
```

- Uniquement pour les « non-bin »
- Généralement utilisé sur les widget `Misc`
  - `Label`, `Image` et `Arrow`

Solution : les EventBox

- Exemple: récupérer les *évènements X* d'un Label
  - `set_events` : Définit les types d'évènement X qui doivent être considérés
  - `signal_button_press_event()` : correspond à un signal X de pression de bouton
    - Nom de fonction préfixée par '`_event`'

```
m_label("Click here");
add(m_eventbox);
m_eventbox.add(m_Label);
m_label.set_size_request(110, 20);
```

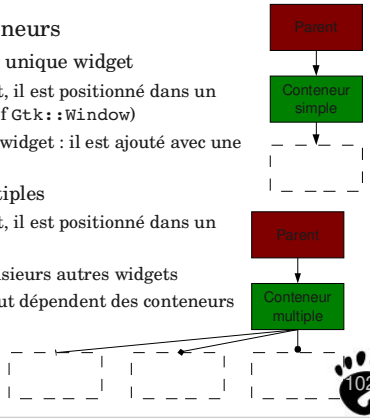
```
m_eventbox.set_events(Gdk::BUTTON_PRESS_MASK);
m_eventbox.signal_button_press_event().connect( sigc::mem_fu
n(*this, &ExampleWindow::on_eventbox_button_press) );
```

Les containers / conteneurs

- Deux types de conteneurs
  - Les conteneurs à un unique widget
    - `Frame`
    - `Dialog`
    - `Alignement`
    - `Window`
    - `EventBox`
  - Les conteneurs multiples, dont
    - `HBox` : une liste dynamique d'espaces horizontaux
    - `VBox` : une liste dynamique d'espaces verticaux
    - `Table` : Grille de Widgets
    - `NoteBook` : un widget à onglets
    - `Menus/Toolbar`
    - `Fixed` : un conteneur à placement libre

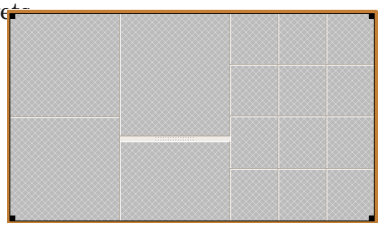
Les containers / conteneurs

- Deux types de conteneurs
  - Les conteneurs à un unique widget
    - En tant que widget, il est positionné dans un widget parent (sauf `Gtk::Window`)
    - Contient un autre widget : il est ajouté avec une fonction **`add`**
  - Les conteneurs multiples
    - En tant que widget, il est positionné dans un widget parent
    - Contient un ou plusieurs autres widgets
    - Les fonctions d'ajout dépendent des conteneurs



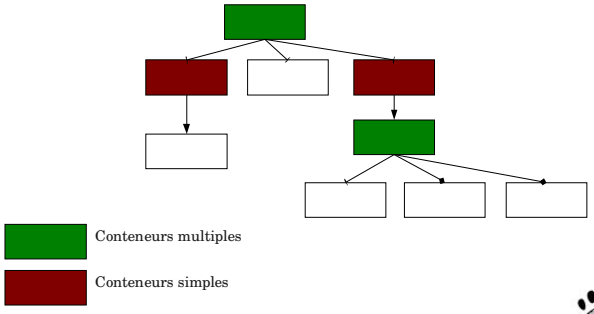
Les containers / conteneurs

- Les containers permettent de structurer une interface graphique en « zones »
  - Les zones sont disjointes
  - Les séparations sont uniquement verticales et horizontales
- Exception `Gtk::Fixed` : container de placement libre des widgets



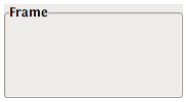
Les containers / conteneurs

- Représentation des zones sous la formes d'arbres



Frame

- Élément décoratif



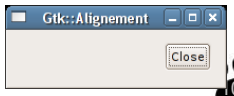
- Propriétés
  - le label

```
Gtk::Window m_window;
Gtk::Frame m_Frame;
Gtk::DrawingArea m_DrawingArea;

m_Frame.set_label('Frame');

// Ajout d'un widget fils
m_Frame.add(m_DrawingArea);

// Ajout du cadre à la fenêtre
m_window.add(m_Frame);
```


Alignement

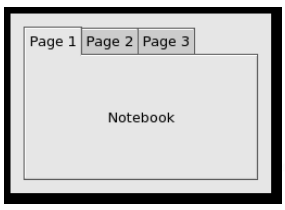
- Alignement d'un widget dans un conteneur
- Contient le widget à aligner
- Constructeur :
  - `Alignment (AlignmentEnum xalign, AlignmentEnum yalign=Gtk::ALIGN_CENTER, float xscale=1.0, float yscale=1.0)`

```
Gtk::Window m_window;
Gtk::Alignment m_Alignment(Gtk::ALIGN_RIGHT, Gtk::ALIGN_CENTER,
0.0, 0.0);
Gtk::Button m_Button(''Close'');

m_window.add(m_Alignment);
m_Alignment.add(m_Button);
```



Containers multiples

- Les conteneurs multiples, dont
  - `HBox` : une liste dynamique d'espaces horizontaux
  - `VBox` : une liste dynamique d'espaces verticaux
  - `Table` : Grille de Widgets
  - `NoteBook` : un widget à onglets
  - `Menus/Toolbar`
  - `Fixed`



Les Boxes (`HBox` ou `VBox`)

- `HBox` – `VBox` : conteneurs multiples pour organiser des widgets verticalement ou horizontalement dans le plan
- Deux propriétés
  - **Homogénéité**
    - « homogeneous property of box, controlling whether or not all children of box are given equal space in the box »
    - **Propriété définie à la création du conteneur**
  - **Espacement**
    - « spacing property of box, which is the number of pixels to place between children of box »

Les Boxes (HBox ou VBox)

- Fonctions d'addition de widget
  - `void pack_start(Gtk::Widget& w);`

ou `void add(Gtk::Widget& w);`

  - `void pack_end(Gtk::Widget& w);`
- Fonction de suppression de widget
  - `void remove(Gtk::Widget& w);`
- Fonction de réordonnancement :
  - `void reorder_child(Gtk::Widget& child, int pos)`

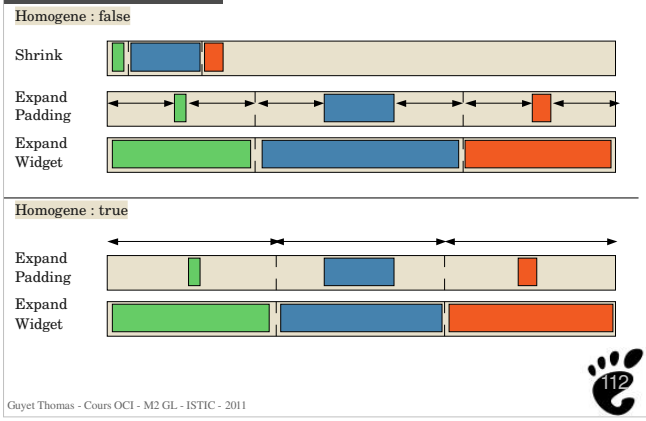
Les Boxes (HBox ou VBox)

```
void pack_start (Widget& child, PackOptions options=PACK_EXPAND_WIDGET,
guint padding=0)
void pack_start (Widget& child, bool expand, bool fill, guint padding=0)
```

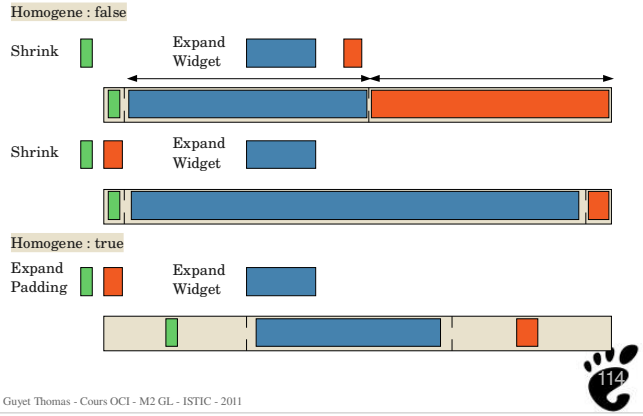
- Détails d'une fonction pack
  - Padding : espace fixé entre les widgets
  - PackOptions
    - `PACK_SHRINK` : Space is contracted to the child widget size.
    - `PACK_EXPAND_PADDING` : Space is expanded, with extra space filled with padding.
    - `PACK_EXPAND_WIDGET` : Space is expanded, with extra space filled by increasing the child widget size.
  - Correspondance entre fonctions

Option	Expand	fill
SHRINK	false	//
EXPAND_PADDING	true	false
EXPAND_WIDGET	true	true

Les Boxes (HBox ou VBox)

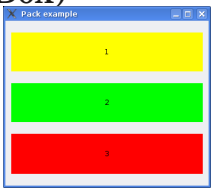


Les Boxes (HBox ou VBox)



Les Boxes (HBox ou VBox)

- Exemple : pack_example.cpp
  - Tester les différentes solutions

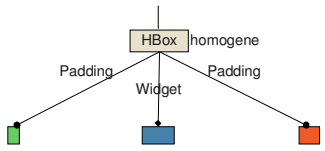


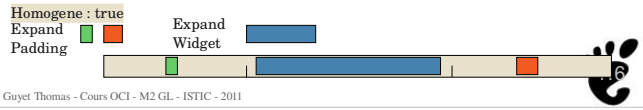

```
Gtk::VBox m_box;
Gtk::Label m_label1("1"), m_label2("2"), m_label3("3");

bool expand=true; //Indique si le box doit s'étirer avec le
parent
bool fill=true; //Indique si les widgets du box doivent
s'étirer avec le box ! (uniquement si expanding = true)
int padding=10;  //minimum d'espace entre les widgets du box

m_box.pack_start(m_label1, expand, fill, padding);
m_box.pack_start(m_label2, true, fill, padding);
m_box.pack_start(m_label3, expand, fill, padding);
```

Représentation dans les arbres

- SHRINK
- EXPAND_PADDING
- EXPAND_WIDGET

- Exemple





Fixed : un container particulier !

- Permet de positionner les widget où l'on veut, avec les tailles que l'on veut
  - Sort des contraintes `VBox` ou `Hbox`,
- Fonctions
  - `void move(Widget& w, int x, int y)`
  - `void put(Widget& w, int x, int y)`
  - `void remove(Widget &w)`
- L'utilisation de ce container est peu recommandée dans les interfaces classiques
  - problèmes d'overlapping

Widget standard

Hacking GTK+, -Muthiah A, Octave-Gtk+ team

Présentation des widgets

1. La classe Widget
2. La hiérarchie des Widgets
3. Présentation de qqs des widgets standard
4. Autres éléments d'une interface

Autres éléments d'une interface

- Autres éléments permettant de construire une interface
  - Les boîtes de dialogue
  - Les menus
  - La barre d'outils
- ou encore (*cf.* documentation)
  - *La barre d'état*
  - *Les documents récents*
  - *Le clipboard*
  - *Les impressions*

Les boîtes de dialogue

- Boîte de dialogue : affiche des messages courts ou récupère des informations courtes auprès de l'utilisateur
- Classe `Gtk::Dialog`
  - Fenêtre composée verticalement avec des `Gtk::Label` et des `Gtk::Entry`
    - Fonctions de pack sur le `VBox` récupéré par : `Vbox* get_vbox()`
  - Associée à une action de retour
    - Des boutons (`Button* add_button (const Glib::ustring& button_text, int response_id)`)
    - ou d'autres actions (*cf.* `add_action_widget(...)`)
  - Est exécutée avec la fonction `int run()`
    - retourne uniquement un entier correspondant à l'action de l'utilisateur

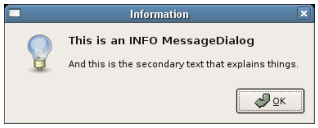
Les boîtes de dialogue

- `MessageDialog`
  - Permet d'afficher des messages à l'utilisateur

```
MessageDialog (Gtk::Window& parent, const Glib::ustring & message, bool
use_markup = false, MessageType type = MESSAGE_INFO, ButtonsType buttons
= BUTTONS_OK, bool modal = false )
```

`MESSAGE_INFO`	`BUTTONS_NONE`
`MESSAGE_WARNING`	`BUTTONS_OK`
`MESSAGE_QUESTION`	`BUTTONS_CLOSE`
`MESSAGE_ERROR`	`BUTTONS_CANCEL`
`MESSAGE_OTHER`	`BUTTONS_YES_NO`
	`BUTTONS_OK_CANCEL`

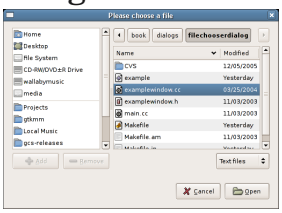
Les boîtes de dialogue

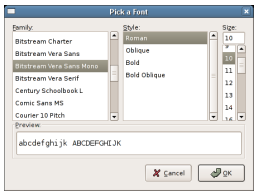


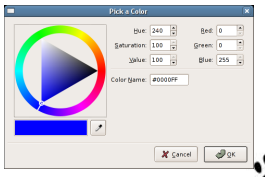

```
Gtk::MessageDialog dialog(*this, "This is a QUESTION MessageDialog", false /* use_mark
*/, Gtk::MESSAGE_QUESTION, Gtk::BUTTONS_OK_CANCEL);
dialog.set_secondary_text( "And this is the secondary text that explains things.");
int result = dialog.run(); //Handle the response:
switch(result) {
    case(Gtk::RESPONSE_OK): {
    std::cout << "OK clicked." << std::endl;
    break;
    }
    case(Gtk::RESPONSE_CANCEL): {
    std::cout << "Cancel clicked." << std::endl;
    break;
    }
    default: {
    std::cout << "Unexpected button clicked." << std::endl;
    break;
    }
}
```


Autres boîtes de dialogue

- FileChooserDialog
- ColorSelectionDialog
- FontSelectionDialog
- AboutDialog







Bibliographie

- **Cours de C++ et Gtk--**, Quesnel Gauthier, 2006
  - http://www-lil.univ-littoral.fr/~quesnel/cplusplus.php
- **Programming-with-gtkmm**, *Murray Cumming, Bernhard Rieder, Jonathon Jongsma, Jason M'Sadoques, Ole Laursen, Gene Ruebsamen, Cedric Gustin, Marko Anastasov, and Alan Ott*, 379 pages, 2006
  - http://www.gtkmm.org/docs/gtkmm-2.4/docs/tutorial/pdf/programming-with-gtkmm.pdf
- **Fondations of Gtk+ development**, Andrew Krause, Apress, 629 pages, 2007.
  - www.gtkbook.com/
- Sites internet
  - www.gtk.org
  - www.gtkmm.org
- Pointeur sur les ressources de ce cours
  - http://www.irisa.fr/dream/Pages_Pros/Thomas.Guyet/enseignements/OCI/