GTKmm Tutorial

CS247 Spring 2019

Topics

- Resources

- Environment

- Compiling & Linking

- Creating a Window

- Window Layout Managers

- Frames & Panels

- Widgets

- Listeners

- Dialogs

- Menus & Toolbars

- Using glade

- Using valgrind

Resources

- How to use GTKmm:
  https://www.student.cs.uwaterloo.ca/~cs247/current/Assignments/P2/using-gtkmm.pdf

- Some general notes on using GTKmm:
  https://www.student.cs.uwaterloo.ca/~cs247/current/Assignments/P2/gtkmm_Notes.pdf

- GTKmm API documentation: https://developer.gnome.org/gtkmm/stable/pages.html

- GTKmm 3 reference manual: https://developer.gnome.org/gtkmm/stable/

- GTKmm 3 tutorial: https://developer.gnome.org/gtkmm-tutorial/stable/

- GTKmm 3 examples
  - https://www.student.cs.uwaterloo.ca/~cs247/current/Assignments/P2/provided/gtkmm-examples.zip
  - https://www.student.cs.uwaterloo.ca/~cs247/current/Assignments/P2/provided/gtkmm-examples/MVC/MVC.zip

Resources (2)

- GNOME Human Interface guidelines
  - https://developer.gnome.org/hig-book/unstable/windows-alert.html.en

- GTK+/Glade
  - https://developer.gnome.org/glade/stable/index.html.en_GB
  - https://en.wikipedia.org/wiki/Glade_Interface_Designer
  - http://python-gtk-3-tutorial.readthedocs.io/en/latest/builder.html
  - https://git.gnome.org/browse/gtkmm-documentation/plain/examples/book/menus_and_toolbars/toolbar.glade?h=gtkmm-3-22
  - https://developer.gnome.org/gtk3/stable/ch01s04.html

Environment

- Use `linux.student.cs.uwaterloo.ca` since has the GNU C++ compiler that is C++14 compliant (`g++ –std=c++14`) and the GTKmm 3.0 package installed.
  - Enable X-forwarding by using command:
    `ssh –X `*`userid`*`@linux.student.cs.uwaterloo.ca`
  - Can use `dpkg -l "*gtkmm*"` to see version of GTKmm package installed.

- Working from home:
  - Windows: XMing + PuTTY + `ssh -X`
  - Mac: XQuartz + `ssh -X`
  - LINUX: `ssh –X`

- Can install on your home computer, but no guarantee that will be 100%-compatible with version in student environment.
  - See `https://developer.gnome.org/gtkmm-tutorial/stable/chapter-installation.html.en`

- **Make sure you try your project out at least once in MC3022 since your project demos will be held there.**

Compiling & linking

- When compiling and linking, **must** specifiy `pkg-config gtkmm-3.0 --cflags --libs` as the last element of each command.

- In order to not forget, strongly encouraged to use provided sample `Makefile`.

```
CXX = g++ -std=c++14
CXXFLAGS = -Wall -g -MMD
GTKFLAGS = `pkg-config gtkmm-3.0 --cflags --libs`
SOURCES = $(wildcard *.cc)
OBJECTS = ${SOURCES:.cc=.o}
DEPENDS = ${OBJECTS:.o=.d}
EXEC=example

$(EXEC): $(OBJECTS)
        $(CXX) $(CXXFLAGS) $(OBJECTS) -o $(EXEC) $(GTKFLAGS)
%.o: %.cc
        $(CXX) -c -o $@ $< $(CXXFLAGS) $(GTKFLAGS)
-include ${DEPENDS}
.PHONY: clean
clean:
        rm  -f $(OBJECTS) $(DEPENDS) $(EXEC)
```
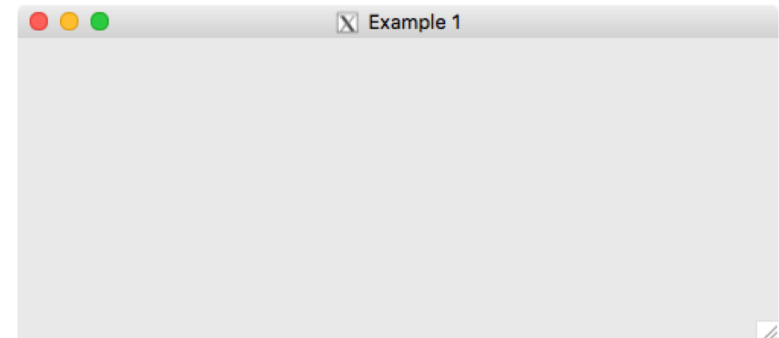
Creating a window

```cpp
// ex1/main.cc
#include <gtkmm/application.h> // Gtk::Application
#include <gtkmm/window.h>      // Gtk::Window
#include <iostream>

int main( int argc, char * argv[] ) {
   auto app = Gtk::Application::create( argc, argv, "GTKmm.Tutorial.Example1" );

   Gtk::Window window;
   window.set_title( "Example 1" );
   window.set_default_size( 500, 200 );
   std::cout << "waiting for window to close" << std::endl;

   return app->run( window );
} // main
```
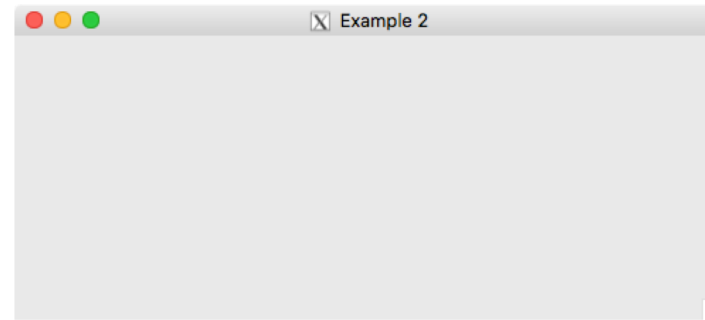
Creating your own window

```
// ex2/window.h
#include <gtkmm/window.h>
class MyWindow : public Gtk::Window {
  public:
    MyWindow();
};
```



```
// ex2/main.cc
#include <gtkmm/application.h>
#include "window.h"
#include <iostream>
int main( int argc, char * argv[] ) {
    auto app = Gtk::Application::create( argc, argv, "GTKmm.Tutorial.Example2" );
    MyWindow window;
    std::cout << "waiting for window to close" << std::endl;
    return app->run( window );
} // main
```

8

Window managers

- Define a layout style for the window.

- Java has a variety of layout managers, GTKmm doesn't. Instead, build by compositing existing "widgets".
  - May find it useful to use Glade to generate the GUI, since GTKmm can import the `.glade` file to define the look of the interface, and then add the code to attach the listeners.

- GTKmm *containers* are defined to contain:
  - single items e.g. `Gtk::Frame`, `Gtk::Window`, `Gtk::Button`
  - multiple items e.g. `Gtk::Paned`, `Gtk::Grid`, `Gtk::Box`, `Gtk::ButtonBox`, `Gtk::Notebook`, `Gtk::Assistant`, `Gtk::TreeView`, `Gtk::HeaderBar`, `Gtk::FlowBox`, etc.
  - `https://developer.gnome.org/gtkmm/stable/classGtk_1_1Container.html`

Panes & frames

```cpp
// ex3/window.h
#include <gtkmm/window.h>            // Gtk::Window
#include <gtkmm/frame.h>             // Gtk::Frame
#include <gtkmm/paned.h>             // Gtk::Paned

class MyWindow : public Gtk::Window {
    Gtk::Paned paned_;
    Gtk::Frame frame1_, frame2_;

  public:
    MyWindow();
    ~MyWindow();
};
```

Panes & frames (2)

```cpp
// ex3/window.cc
#include "window.h"

MyWindow::MyWindow() : paned_{Gtk::Orientation::ORIENTATION_VERTICAL} {
    set_title( "Example 3" );       // Sets the window title.
    set_default_size( 500, 200 );  // Set default size, width and height, in pixels.
    set_border_width( 10 );
    add( paned_ );
    paned_.add1( frame1_ );
    paned_.add2( frame2_ );
    frame1_.set_label( "Gtk::Frame 1 Widget" ); // set the frames label
    frame2_.set_label( "Gtk::Frame 2 Widget" );
    frame1_.set_label_align( 0.0, 0.25 );
    frame2_.set_label_align( 1.0 );
    frame1_.set_shadow_type( Gtk::ShadowType::SHADOW_ETCHED_OUT );
    frame2_.set_shadow_type( Gtk::ShadowType::SHADOW_ETCHED_IN );
    show_all_children();
}

MyWindow::~MyWindow() {}
```
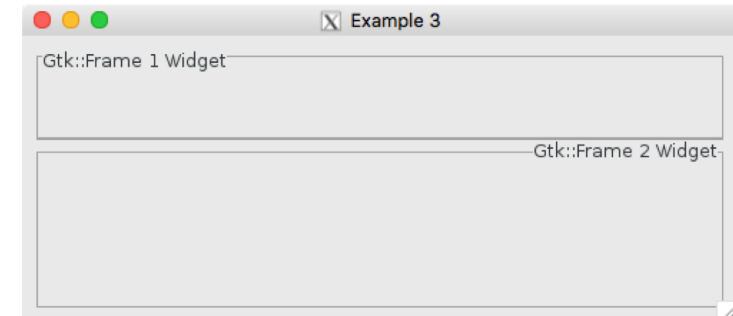
Widgets

★ • Layout: `Separator`, `Paned`, `Box`, `Grid`, `ScrollBar`

• Menus and tool bars: `MenuBar`, `ToolBar`

★ • Text: `Label`, `Entry`, `TextView`

• Information: `ToolTips`, `InfoBar`, `Dialog`
  • `MessageDialog`, `FileChooserDialog`, `ColorChooserDialog`, `FontChooserDialog`, `AboutDialog`

• Numeric values: `SpinButton`, `Range`, `Scale`

★ • Pictures: `Image`, `PixBuf`, `DrawingArea`

★ • Buttons: `Button`, `ToggleButton`, `CheckButton`, `RadioButton`
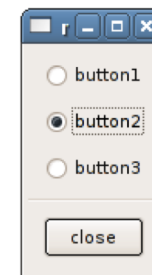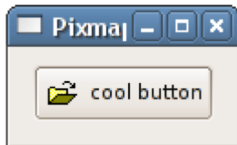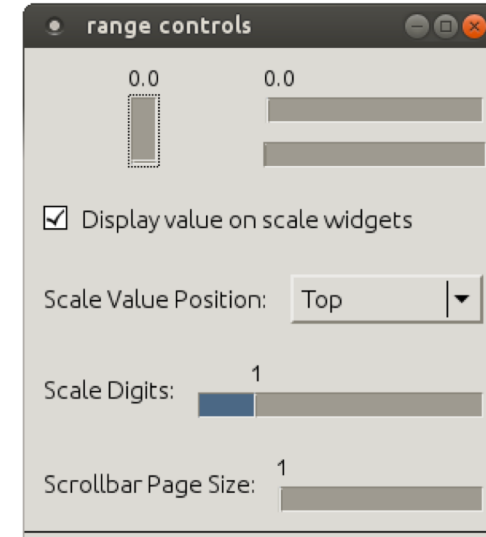
More complex window structure

```cpp
// ex4/window.h
#include <gtkmm.h>
#include <vector>
#include <string>
class MyWindow : public Gtk::Window {
    Gtk::Paned paned_;
    Gtk::Frame frame1_, frame2_;
    Gtk::ButtonBox bbox_;
    Gtk::Box box_;
    Gtk::Label label_;
    Gtk::Button button1_, button2_;
    Gtk::Image image_;
    Gtk::Entry valueField_;
    static std::vector<std::string> imageNames_;
    static int index_;
  public:
    MyWindow();
    ~MyWindow();
};
```

More complex window structure (2)

```cpp
// ex4/window.cc
MyWindow::MyWindow() :
    paned_{Gtk::Orientation::
            ORIENTATION_VERTICAL},
    label_{"Enter text: "},
    image_{imageNames_.at(index_)},
{
    set_title( "Example 4" );
    set_default_size( 500, 300 );
    set_border_width( 10 );
    button1_.set_image( image_ );
    button2_.set_label( "something" );
    bbox_.add( button1_ );
    bbox_.add( button2_ );
    valueField_.set_text( "xxx" );
    box_.add( label_ );
    box_.add( valueField_ );
    frame1_.add( bbox_ );
    frame2_.add( box_ );
    add( paned_ );
    paned_.add1( frame1_ );
    paned_.add2( frame2_ );
    frame1_.set_label("...");
    frame2_.set_label("...");
    frame1_.set_label_align( 0.0, 0.25 );
    frame2_.set_label_align( 1.0 );
    frame1_.set_shadow_type(
        Gtk::ShadowType::SHADOW_ETCHED_OUT );
    frame2_.set_shadow_type( Gtk::
ShadowType::SHADOW_ETCHED_IN );
    show_all_children();
}
```
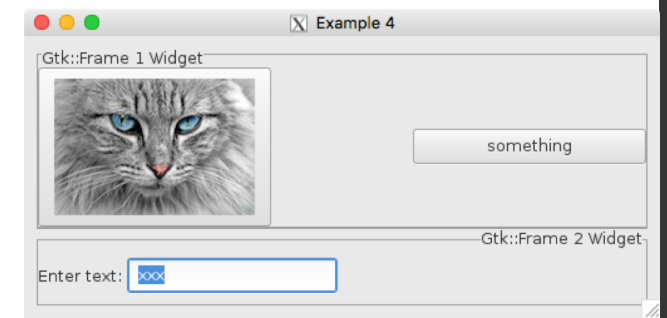
Listeners

- In order to be able to interact with the GUI, we need to associate actions to perform i.e. functions to call to GUI events.
  - Events include: keyboard keys pressed/released, mouse motion, button presses, etc.

- Tie an event received by a widget to a function to call, as in:

  `button.signal_clicked().connect( sigc::mem_fun(*this, &function) );`

- For keyboard events, need to tell window to override `on_key_press_event` and what sort of events it's interested in, combined via bit-wise operators:

  `add_events( Gdk::EventMask );`

  `bool on_key_press_event( GdkEventKey* ) override;`

Adding listeners

```cpp
// ex4/window.h
#include <gtkmm.h>
#include <vector>
#include <string>
class MyWindow : public Gtk::Window {
    ...
    bool on_key_press_event( GdkEventKey* ) override;

  protected:
    // signal handlers
    void b1_clicked();
    void b2_clicked();

  public:
    MyWindow();
    ~MyWindow();
};
```

Adding listeners (2)

```cpp
// ex4/window.cc
MyWindow::MyWindow() ... {
    ...
    // set button listeners
    button1_.signal_clicked().connect(
     sigc::mem_fun(*this,
        &MyWindow::b1_clicked) );
    button2_.signal_clicked().connect(
        sigc::mem_fun(*this,
        &MyWindow::b2_clicked) );
    add_events( Gdk::KEY_PRESS_MASK );
    ...
}

bool MyWindow::on_key_press_event(
GdkEventKey* keyEvent ) {
    static int numTimesReturnPressed = 0;
    if (keyEvent->keyval == GDK_KEY_Return) {
        numTimesReturnPressed++;
        string s = valueField_.get_text();
        cout << "User entered: " << s << endl;
        label_.set_text( s );
        valueField_.set_text( std::to_string(
            numTimesReturnPressed ) );
        return true;
    } // if

    return Gtk::Window::on_key_press_event(
        keyEvent );
}

void MyWindow::b1_clicked() {
    index_++;
    index_ %= imageNames_.size();
    image_.set( imageNames_.at( index_ ) );
}

void MyWindow::b2_clicked() {
    string s = valueField_.get_text();
    cout << "old text = " << s << endl;
    valueField_.set_text( "yyy" );
}
```

17

Dialogs

- Inherit from `Gtk::Dialog`
- ★ `Gtk::MessageDialog`: simple two lines of text plus "OK" button. Can add an image and other buttons that trigger a `Gtk::Dialog::signal_response()` signal plus an id.
  - `Gtk::FileChooserDialog`: for "open" or "save" actions.
  - `Gtk::ColorChooserDialog`: presents a colour palette from which the user can select a colour.
  - `Gtk::FontChooserDialog`: presents a selection of fonts from which the user can select one.
- ★ `Gtk::AboutDialog`: lets the program present some information without freezing the rest of the program (unlike the other dialogs) i.e. *non-modal*. Designed to display program name, credits, version, copyright, comments, authors, artists, website, documenters, translator, and logo. Can add an image and other buttons that trigger a `Gtk::Dialog::signal_response()` signal plus an id.

18
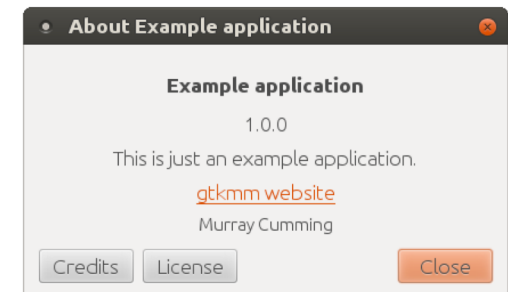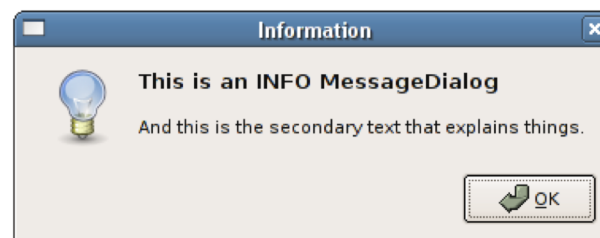
Menus & Tool bars

- GTKmm 3.0 uses an XML string to specify the layout and format of menus and tool bars.
  - Need to add the `Gtk::MenuBar`/`Gtk::Toolbar` to a `Gtk::Builder`, which is then added to the container.

- Create actions, possibly associated with "key accelerators" such as `Ctrl-C` for copy, that are tied to the menu/tool bar selections.
  - Would also be specified in the XML string.

- Can add a menu bar directly to a `Gtk::ApplicationWindow`, which is a subclass of both `Gtk::Window` and `Gio::ActionMap`.
  - Populate the menu bar through the constructor, or by adding `Gtk::MenuItem` objects. Dealing with window closure gets tricky, though.

- Otherwise, the window needs to know about the application, and needs a `Gtk::Builder` to create the menu bar.

- Can create pop-up menus by using `Gtk::Menu::popup`.

Menus & Tool bars (2)

- I would suggest having a "main box" with a vertical orientation, to which you'd add the menu bar (and tool bar if you want one) before adding the other widgets.

- There will likely be an overlap between the actions in the menu bar and the tool bar, so use that to your advantage.

- Can either build the menu incrementally by adding menu items or use XML, which is considerably easier once you understand the format.
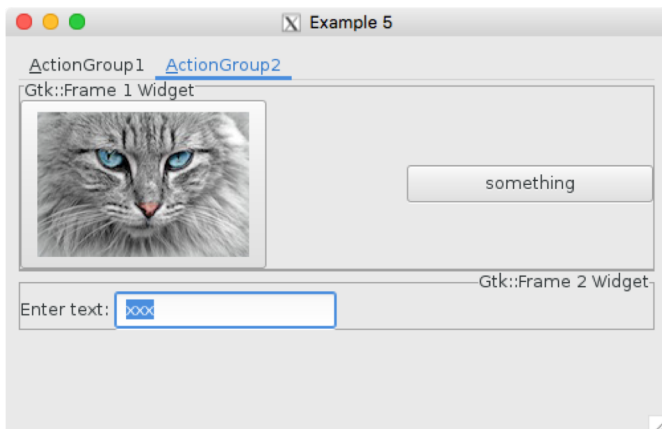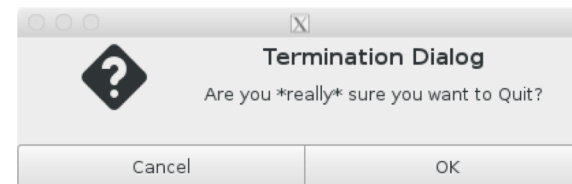
Defining a menu

```
// ex5/window.cc
MyWindow::MyWindow(const Glib::RefPtr<
    Gtk::Application>& app) ... {
    ...
    add( mainBox_ );
    setUpMenu();
    structureGUI();
}
void MyWindow::setUpMenu() {
    actionGroup_=Gio::SimpleActionGroup::
        create();
    builder_ = Gtk::Builder::create();
    insert_action_group("example",
        actionGroup_);
    ...
    actionGroup_->add_action( "quit",
        sigc::mem_fun(*this,
        &MyWindow::sub_action1_quit));
    ...
```

```
const char* ui_info =
    "<interface>"
    "  <menu id='menubar'>"
    "    <submenu>"
    "      <attribute name='label'
translatable='yes'>_ActionGroup1
</attribute>"
    "      <section>"
...
    "        <item>"
    "          <attribute name='label'
translatable='yes'>_Quit</attribute>"
    "          <attribute name='action'>
example.quit</attribute>"
    "          <attribute name='accel'>
&lt;Primary&gt;q</attribute>"
    "        </item>"
...
```

Defining a menu (2)

```
app_->set_accel_for_action(
    "example.quit", "<Primary>q");
...

try {
    builder_->add_from_string( ui_info );
} catch( const Glib::Error& ex) {
    std::cerr << "Building menu failed: "
        <<  ex.what();
}

auto object = builder_->get_object(
    "menubar" );
auto gmenu = Glib::RefPtr<Gio::Menu>
    ::cast_dynamic( object );
```

```
// did the conversion fail?
if ( !gmenu )
    g_warning( "GMenu not found" );
else {
    auto menuBar = Gtk::manage( new
        Gtk::MenuBar( gmenu ) );

    mainBox_.pack_start( *menuBar,
        Gtk::PACK_SHRINK );
} // if
} // MyWindow::setUpMenu
```

Defining a menu (3)

```cpp
void MyWindow::sub_action1_quit() {
    Gtk::MessageDialog dialog( *this, "Termination Dialog", false,
        Gtk::MESSAGE_QUESTION, Gtk::BUTTONS_OK_CANCEL);
    dialog.set_secondary_text( "Are you *really* sure you want to Quit?" );
    int result = dialog.run();
    switch(result) {
        case( Gtk::RESPONSE_OK ):
            hide();
            break;
        case( Gtk::RESPONSE_CANCEL ):
            std::cout << "Cancel clicked." << std::endl;
            break;
        default:
            std::cout << "Unexpected button clicked." << std::endl;
            break;
    } // switch
}
```

Tool bars

- `Gtk::Toolbar` contains `Gtk::ToolItem`, which has subclasses `Gtk::SeparatorToolItem`, `Gtk::ToolButton`, `Gtk::MenuToolButton`, `Gtk::ToggleToolButton`, and `Gtk::RadioToolButton`.

- Can configure if a tool item is visible (or not) when the toolbar is laid out horizontally or vertically.

- `Gtk::ToolItem` can be decorated with an image and/or a text label.

- Simplest way to define the tool bar layout is to use XML in a `.glade` file, which can be used in combination with previously defined actions to specify what to do when the tool bar button is pressed.
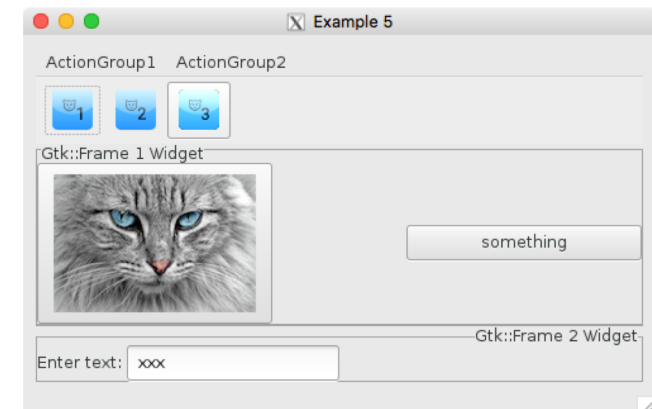
toolbar.glade

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- Generated with glade 3.18.3 -->
<interface>
  <requires lib="gtk+" version="3.8"/>
  <object class="GtkImage" id="image_1">
    <property name="visible">True</property>
    <property name="can_focus">False</property>
    <property name="pixbuf">img/icon1.png</property>
  </object>
...
<object class="GtkToolbar" id="toolbar">
    <property name="visible">True</property>
    <property name="can_focus">False</property>
```

toolbar.glade

```
...
    <child>
      <object class="GtkToolButton" id="show1">
        <property name="visible">True</property>
        <property name="can_focus">False</property>
        <property name="tooltip_text" translatable="yes">Show image 1</property>
        <property name="action_name">example.show1</property>
        <property name="label_widget">image_1</property>
      </object>
      <packing>
        <property name="expand">False</property>
        <property name="homogeneous">True</property>
      </packing>
    </child>
...
</interface>
```

window.cc

```
void MyWindow::setUpMenu() {
    ...
    builder_->add_from_file( "toolbar.glade" );
    Gtk::Toolbar * toolbar = nullptr;
    builder_->get_widget( "toolbar", toolbar );
    if ( !toolbar )
        g_warning( "toolbar not found" );
    else
        mainBox_.pack_start( *toolbar, Gtk::PACK_SHRINK );
    ...
}

void MyWindow::show_image_1() { image_.set( imageNames_.at( 0 ) ); }
```

Glade

- It's pretty painful to create your own custom widgets that Glade will understand and let you use.
  - Need to define custom Glade libraries in XML, and add to the Glade search path.
  - https://developer.gnome.org/gladeui/unstable/catalogintro.html

- Instead, it's probably easier to use Glade to create the GUI as if the default window is your custom window, and in your program, use the `Gtk::Builder` to find the window, and create it as your derived class window.

- You can then create your action group and connect the necessary signal bindings to your window's private/protected methods.

- See `ex6/gui.glade`.

window.h

```
// ex6/window.h
class MyWindow : public Gtk::Window {
    Glib::RefPtr<Gtk::Application> & app_;
    Glib::RefPtr<Gtk::Builder> & builder_;
    Glib::RefPtr<Gio::SimpleActionGroup> actionGroup_;
    Gtk::Box     * box_;
    Gtk::Image   * image_;
    Gtk::Button  * button1_, * button2_;
    Gtk::Label   * label_;
    Gtk::Entry   * valueField_;

    ...

  public:
    MyWindow( BaseObjectType *, Glib::RefPtr<Gtk::Builder> &,
        Glib::RefPtr<Gtk::Application> & );
    ~MyWindow();
};
```

window.cc

```
MyWindow::MyWindow( BaseObjectType * cobject,
  Glib::RefPtr<Gtk::Builder> & builder,
  Glib::RefPtr<Gtk::Application> & app ) :
  Gtk::Window{cobject}, app_{app}, builder_{builder},
  ...
{
  set_title( "Example 6" );
  set_default_size( 300, 400 );
  set_border_width( 10 );
  buildUI();
  buildMenu();
  add_events( Gdk::KEY_PRESS_MASK );
  show_all_children();
}
```

window.cc (2)

```cpp
void MyWindow::buildUI() {
    builder_->get_widget( "button1", button1_ );
    ...
    if ( button1_ == nullptr || button2_ == nullptr || ... label_ == nullptr ) {
      g_warning("unable to extract window sub-components"); return;
    } // if
    image_->set( imageNames_.at( 0 ) );
    button1_->signal_clicked().connect( sigc::mem_fun(*this, &MyWindow::b1_clicked) );
    button2_->signal_clicked().connect( sigc::mem_fun(*this, &MyWindow::b2_clicked) );
} // MyWindow::buildUI
```

window.cc (3)

```
void MyWindow::buildMenu() {
    actionGroup_ = Gio::SimpleActionGroup::create();
    Gtk::Window::insert_action_group( "example", actionGroup_ );
    actionGroup_->add_action( "sub_act_1.1", sigc::mem_fun( *this,
        &MyWindow::sub_action1_1 ) );
     ...

    app_->set_accel_for_action("example.sub_act_1.1", "<Primary>1");
    ...
    actionGroup_->add_action( "show1", sigc::mem_fun( *this,
        &MyWindow::show_image_1 ) );
    ...
} // MyWindow::buildMenu
```

main.cc

```cpp
int main( int argc, char * argv[] ) {
    try {
        auto app = Gtk::Application::create( argc, argv, "gtkmm.tutorial.example6" );
        auto builder = Gtk::Builder::create_from_file( "gui.glade" );

        MyWindow *window_ = nullptr;
        builder->get_widget_derived( "window", window_, app );
        if ( window_ == nullptr ) {
            g_warning("unable to extract window");
            return -1;
        } // if

        app->run( *window_ );
        delete window_;

} // main
```

Valgrind

- GTKmm doesn't play well with `valgrind`.

- Libraries don't free memory allocated once, apparently relying upon OS to clean it up.

- Developers claim appropriate suppression files and exporting variables such as `G_SLICE` and `G_DEBUG` will make it workable.

- Resources:
  - `https://wiki.gnome.org/Valgrind`
  - `https://stackoverflow.com/questions/16659781/memory-leaks-in-gtk-hello-world-program`
  - `https://github.com/dtrebbien/GNOME.supp`