CSE 1325: Object-Oriented Programming

Lecture 14 – Chapter 16

Windows and Custom Dialogs
Observer and Factory Patterns

Mr. George F. Rice

george.rice@uta.edu

Based on material by Bjarne Stroustrup

www.stroustrup.com/Programming

ERB 402

Office Hours:

Tuesday Thursday 11 - 12

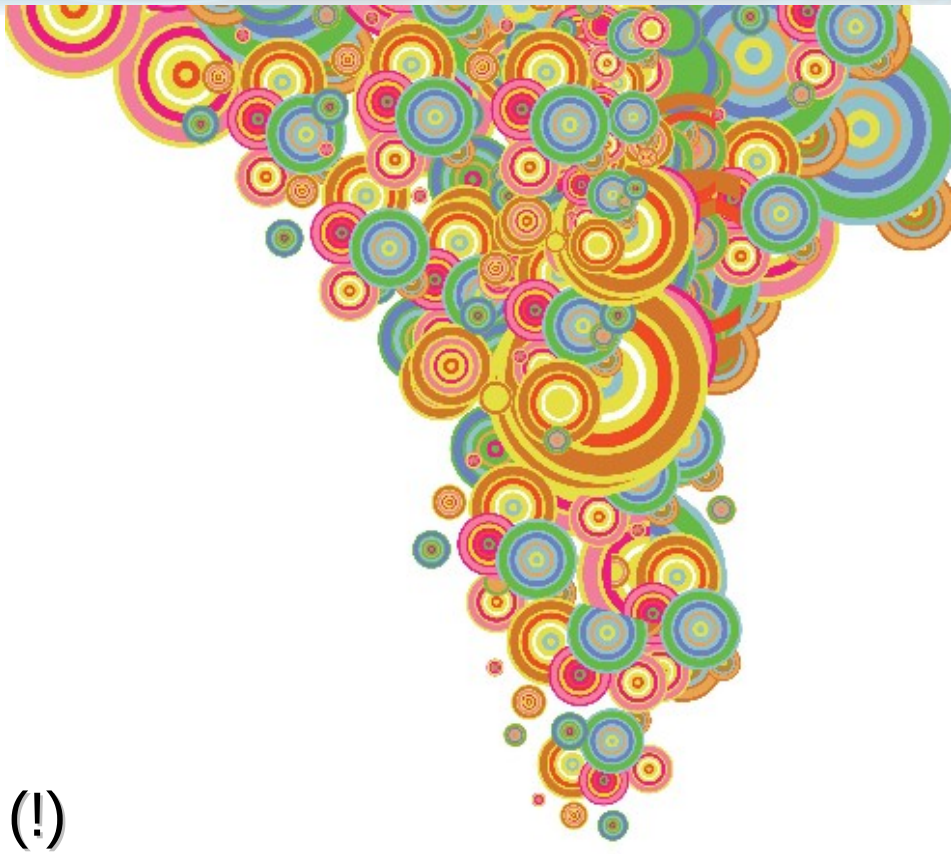
Or by appointment

Quick Review

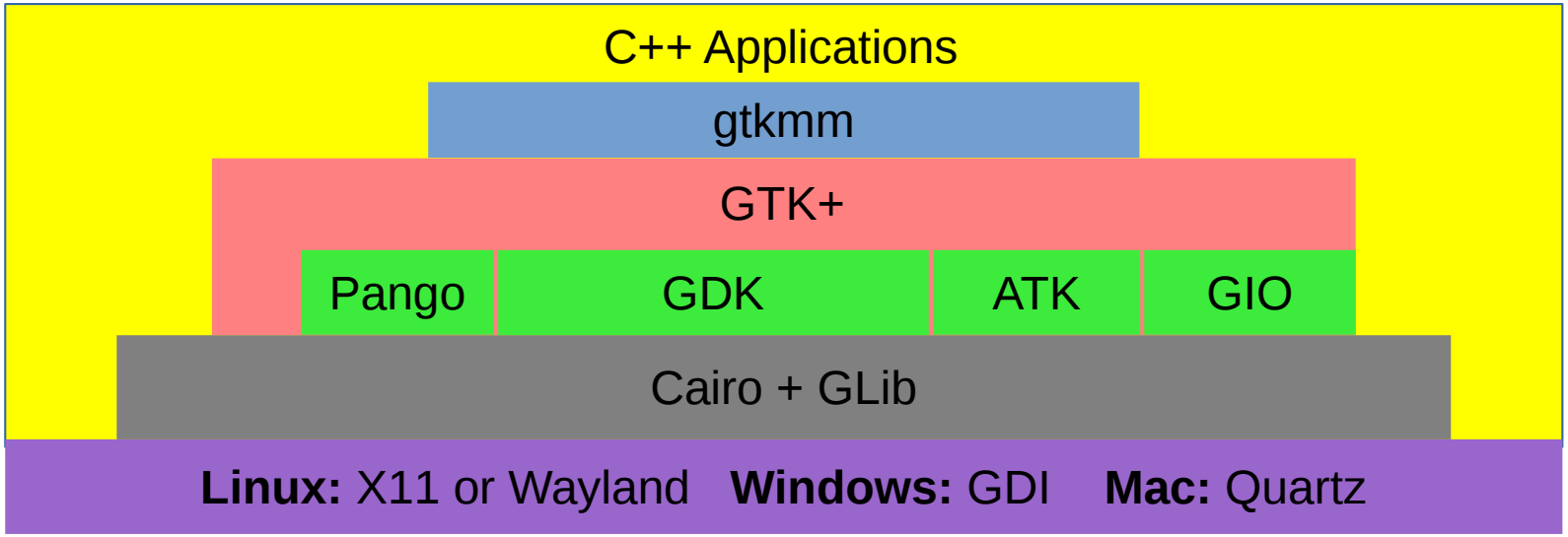
- Each program using gtkmm must include the header file gtkmm.h.
- True or False: `/usr/bin/pkg-config gtkmm-3.0 --cflags –libs` need only be added to the linker line, not the compiler lines, in the Makefile. False. It must also be added to any compiler lines for files including gtkmm.h.
- True or False: The Dialogs class must be instanced before the methods can be called. False. The methods are all static.

Overview: Windows and Dialogs

- gtkmm Architecture
- Dialogs with Callbacks
  - Observer Pattern
  - 4 Dialog Implementations
  - Virtual Destructors
  - Connecting Signals to Methods
  - Gtk's "Application Factory"
  - Factory Pattern
- Full Applications
  - Automatic Memory Management (!)
  - Multiple Widget Layout Management
  - Menu Bars
  - Example Application



gtkmm Architecture



Gtkmm: A thin native C++ wrapper for Gtk+, allowing the use of more natural C++ idioms
GTK+: An object-oriented C library for implementing graphical applications
Pango: A text markup rendering library to convert HTML-like tags into visual attributes
GDK: The GIMP Drawing Toolkit provides the basic graphics primitives for Gtk+
ATK: The Accessibility Toolkit provides accessible technologies like screen readers
GIO: The Gnu Input/Output Library provides a virtualized file system interface
GLib: The Gnu Library provides advanced data structures, Unicode strings, thread support, callback mechanisms, and similar capabilities now more commonly supported by the C++ Standard Template Library (STL) and Boost
Cairo: Vector graphics toolkit that (among other things) actually renders GTK+ widgets
X11, Wayland, GDI, Quartz: Operating System-dependent graphics interfaces

Creating Dialogs with gtkmm

- We can use our own Dialogs convenience class
  - Simple, single-line invocation with few options
- We can use DialogMessage
  - A few lines and a few more options, but limited to text output
- We can use Dialog
  - More lines and many more options
  - Capable of moderately complex dialogs
- We can use Window via the Application factory*
  - Much more complexity but with full flexibility
  - Suitable for the main window and sophisticated dialogs

*A “factory” produces objects via a static method call rather than a constructor. We’ll get there in a few minutes. Hard hats, please!

Custom Actions from a Button

- Our Dialogs class simply returns the button pressed
  - This is called “polling” - our program waits, checks which button was pressed, and then responds – simple, but NOT very scalable
- Modern GUIs do NOT poll – they “observe”
  - Our code tells the button which method(s) to call when it is clicked
  - Similarly, it tells the button which method(s) to call on any other interaction – mouse entry or exit, button pressed or released, etc.
  - Similarly, it tells all of the other widgets in the window which method(s) to call on any interaction with them
  - Similarly, it tells the system which method(s) to call on other events of interest, such as a timer expiration or arriving network packets
  - Then our program hands control to the tool kit, which runs the “main loop” and orchestrates method calls until the program ends

Behavioral

Observer Pattern

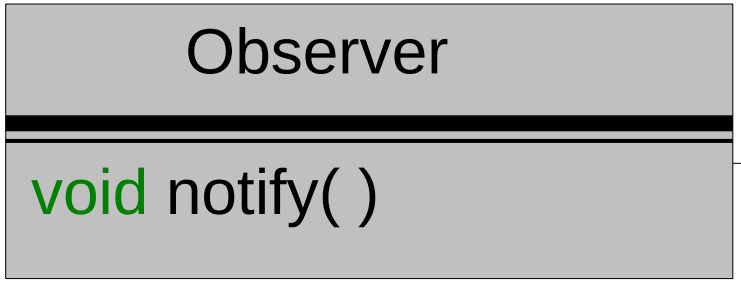
- The **Observer** pattern notifies dependent objects when the observed event occurs
  - This is most useful for implementing distributed event handling systems such as (ahem) gtkmm
  - **Beware** – In managed environments, the Observer's reference to an object, if it is the last reference remaining, will prevent garbage collection and may cause the equivalent of memory leaks

Behavioral

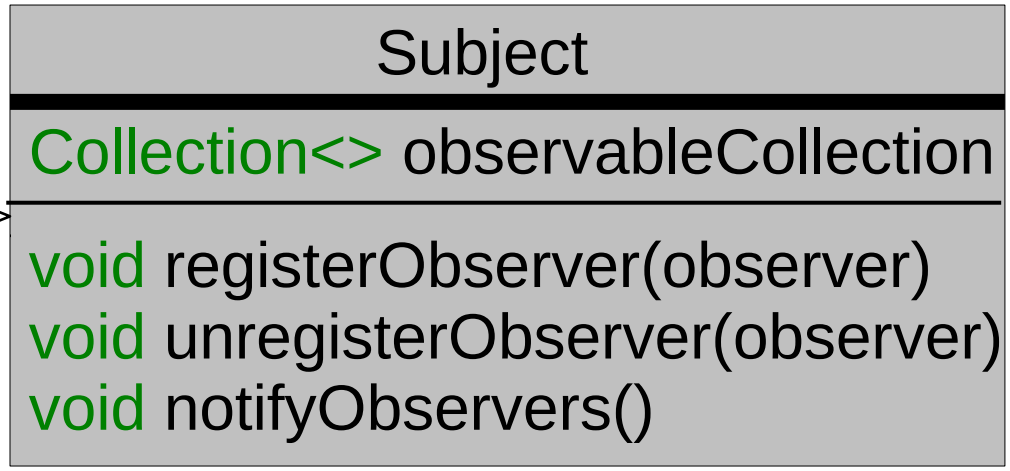
The Observer Pattern

(Slightly Simplified)

The observer, which is notified when Subject changes



Basic notifier



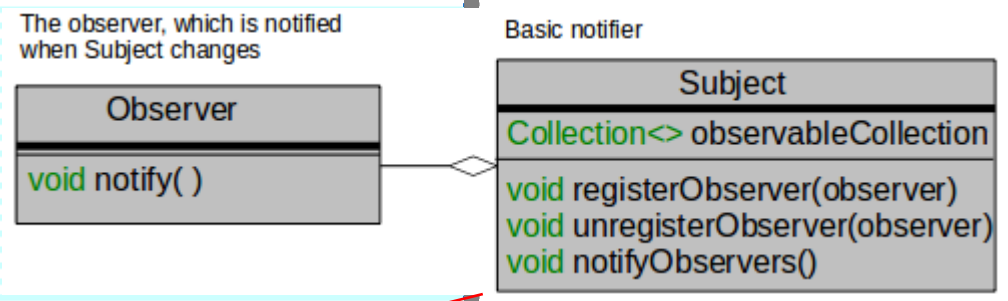
Behavioral

The Observer Pattern

(Slightly Simplified)

observer.cpp

```
#include <vector>
#include <functional>
#include <iostream>
using namespace std;

class Observer_interface {
    public:
        virtual void notify() = 0;
};
class Observer : public Observer_interface {
    public:
        virtual void notify() override {
            cout << "Notified!" << endl;
        }
};
class Subject {
    public:
        void register_observer(Observer_interface& o) {
            observers.push_back(o);
        }
        void notify_observers() {
            for (Observer_interface& o : observers) {
                o.notify();
            }
        }
    private:
        vector<reference_wrapper<Observer_interface>> observers;
};
```



"override" asserts that this method is overriding a method from the base class

"reference_wrapper"
(declared in <functional>)
enables use of a reference as a template type

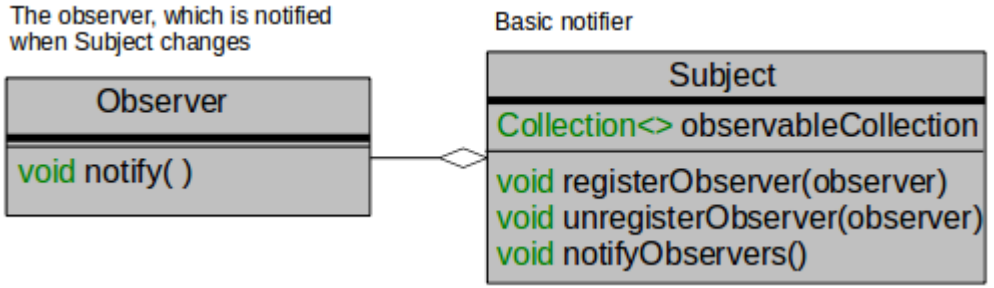
Notifications are often referred to as "Callbacks"

Behavioral

The Observer Pattern

(Slightly Simplified)

Notifications are often referred to as “Callbacks”



observer.cpp

```
int main() {
    // The subject to be observed.
    // Call notify_observers() when a noteworthy event happens.
    Subject subject;

    // The observer watching the subject, waiting for a notification.
    Observer observer;

    // Register the observer with the subject
    subject.register_observer(observer);

    // Now, notify all observers that a noteworty event has happened.
    subject.notify_observers();
}
```

```
ricegf@pluto:~/dev/cpp/201708/11$ make observer
g++ --std=c++11 -o observer observer.cpp
ricegf@pluto:~/dev/cpp/201708/11$ ./observer
Notified!
ricegf@pluto:~/dev/cpp/201708/11$
```

GUIs and the Observer Pattern

- Most GUI toolkits (including gtkmm) use the Observer pattern to handle “behaviors”
- Thus, creating a GUI app involves **6 basic steps**

1) Declare and configure the window
2) Declare the widgets to populate the window
3) Configure each widget and “add” to the window
4) Assign callbacks (register observers) to each widget’s behavior(s) of interest (events) so that they can respond to user interaction
5) Show the window (make it visible to the user)
6) Turn control over the the GUI tool kit’s main loop

Examples

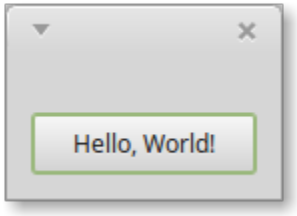
- We'll look at the polling approach (not very scalable), followed by 2 increasingly sophisticated, capable, and scalable observer approaches
  - Our Dialogs Façade
  - gtkmm's Dialog class
  - gtkmm's Window class (Dialog's base class)

Using Our Dialogs Façade
to Create a Single-Button Window

- Very little code, but with very little flexibility
  - Only 3 lines of code required
  - Not much flexibility, e.g., when you click the button, the dialog closes and returns a button number
  - Great for “quick and dirty”

dialog1.cpp

```
#include "dialogs.h"

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

    vector<string> buttons = {"Hello, World!"};
    std::cout << Dialogs::question("", "", buttons) << endl;
}
```



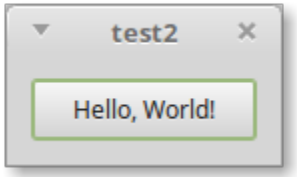
What output does this generate, and when does it generate it?

Using gtkmm's Dialog Class
to Create a Single-Button Window

- A bit more code, but more flexibility
  - 8 lines of code required (similar for DialogMessage)
  - Greater flexibility as Dialog offers over a thousand member methods and data elements

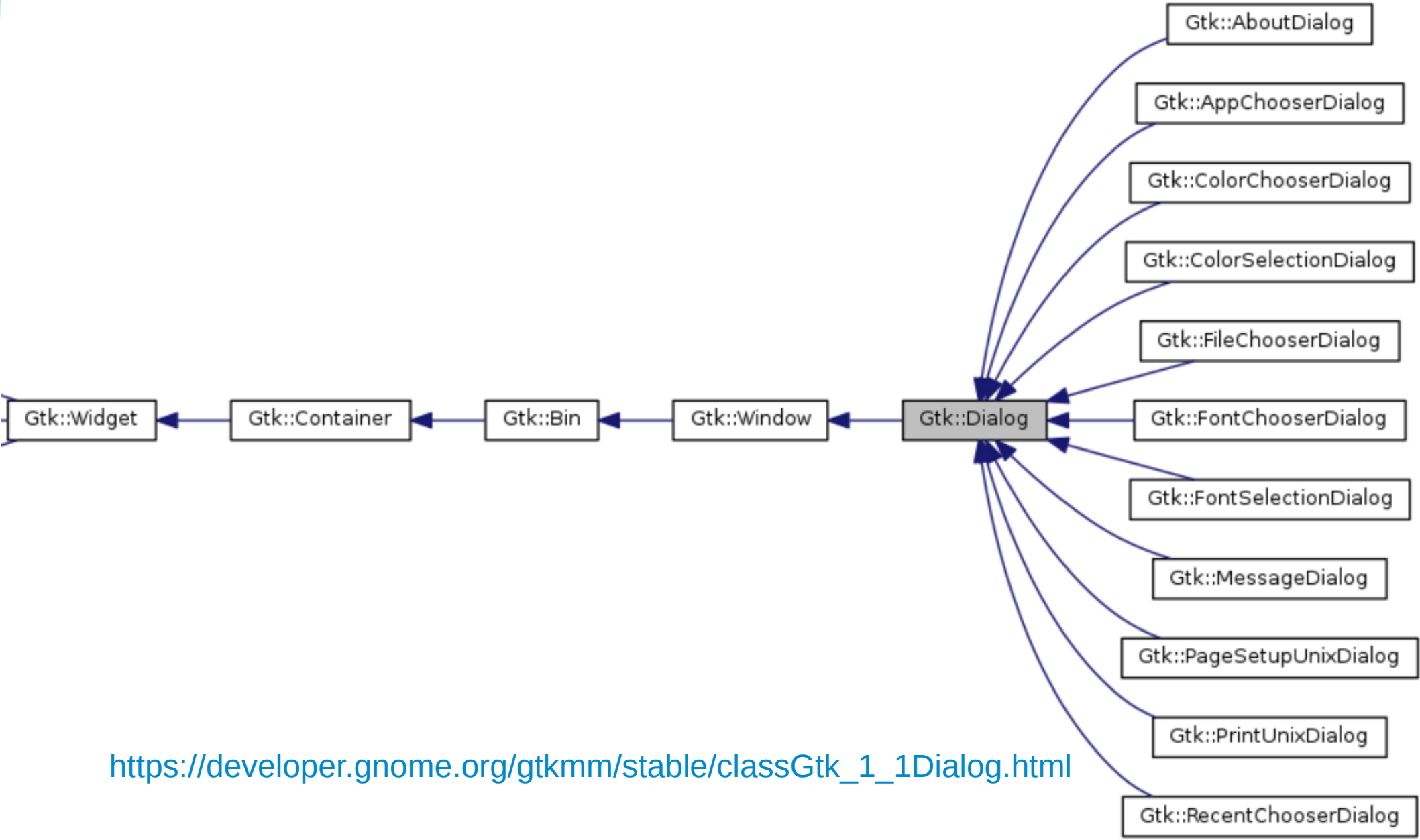
dialog2.cpp

```
#include <gtkmm.h>
#include <iostream>

int main(int argc, char *argv[]) {
    Gtk::Main kit(argc, argv);
    Gtk::Dialog *dialog = new Gtk::Dialog();

    dialog->add_button("Hello, World!", 42);

    std::cout << dialog->run() << std::endl;
    delete dialog;
}
```



What output does this generate, and when does it generate it?

gtkmm Offers Many Predefined Dialogs, Too

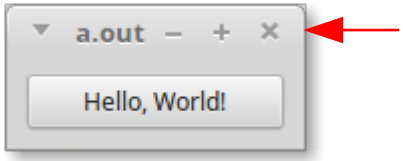


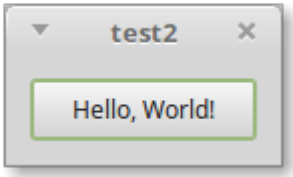
https://developer.gnome.org/gtkmm/stable/classGtk_1_1Dialog.html

Using gtkmm's Window Class
to Create a Single-Button Window

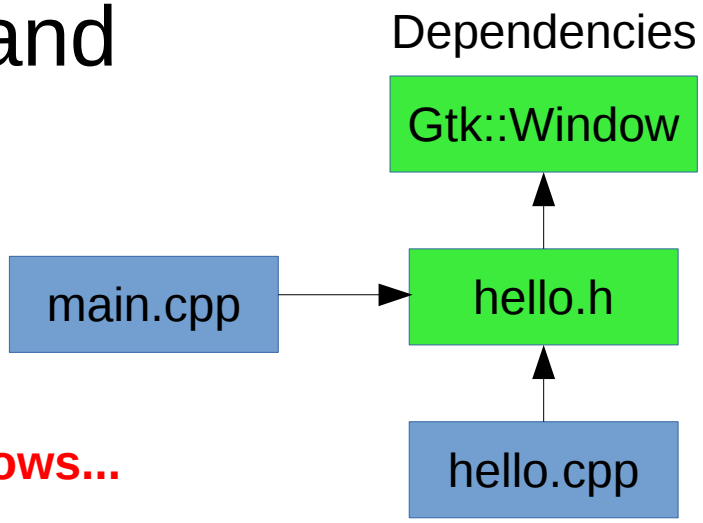
- Significantly more (and more complicated) code, but much more flexibility
  - 28 lines of code required
  - Code does NOT exit on button click, but rather executes a behavior that we specify
- In this case, we derive a "Hello" class from gtkmm's Window class and customize it

Notice the full Window controls on the left, compared the the Dialog on the right





Code follows...

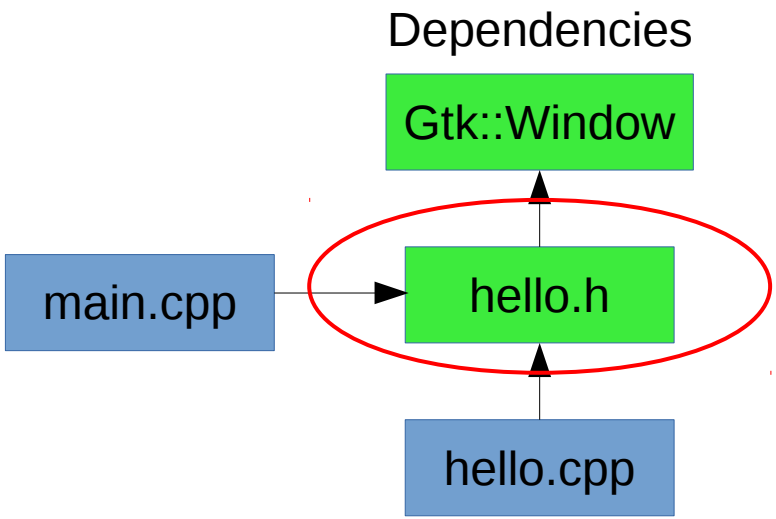


hello.h

- Hello class *inherits* from Gtk::Window. We add:
  - A constructor for customizing the window
  - A virtual destructor
  - A custom behavior when the button is clicked
  - The Gtk::Button itself

hello.h

```
#ifndef _HELLO_H
#define _HELLO_H

#include <gtkmm.h>

class Hello : public Gtk::Window {
  public:
    Hello();
    virtual ~Hello();
  protected:
    void on_button_clicked();
    Gtk::Button button;
};
#endif
```



Virtual Destructor?

- Consider this code:

virtual1.cpp

```
#include <iostream>

class Base {
  public:
    void test() {std::cout << "Base" << std::endl;}
};

class Derived : public Base {
  public:
    void test() {std::cout << "Derived" << std::endl;}
};

int main() {
  Base *b = new Derived{ };
  b->test();
}
```

- Variable b is of type “pointer to Base”, but actually contains a pointer to an instance of Derived
  - What will this program print, “Base” or “Derived”?

Virtual Destructor?

- Running this code:

virtual1.cpp

```
#include <iostream>

class Base {
  public:
    void test() {std::cout << "Base" << std::endl;}
};

class Derived : public Base {
  public:
    void test() {std::cout << "Derived" << std::endl;}
};

int main() {
  Base *b = new Derived{ };
  b->test();
}
```

```
ricegf@pluto:~/dev/cpp/201801/14$ make virtual1
g++ -std=c++14 -o virtual1 virtual1.cpp
ricegf@pluto:~/dev/cpp/201801/14$ ./virtual1
Base
ricegf@pluto:~/dev/cpp/201801/14$
```

- For a normal method, C++ by default follows the type of the *variable as declared*

Virtual Destructor?

- Now consider this code, with “virtual” added:

virtual2.cpp

```
#include <iostream>

class Base {
  public:
    virtual void test() {std::cout << "Base" << std::endl;}
};

class Derived : public Base {
  public:
    void test() {std::cout << "Derived" << std::endl;}
};

int main() {
  Base *b = new Derived{ };
  b->test();
}
```

- Variable b is still of type “pointer to Base”, and still contains a pointer to an instance of Derived
  - What will this program print, “Base” or “Derived?

Virtual Destructor?

- Running this code:

virtual2.cpp

```
#include <iostream>

class Base {
  public:
    virtual void test() {std::cout << "Base" << std::endl;}
};

class Derived : public Base {
  public:
    void test() {std::cout << "Derived" << std::endl;}
};

int main() {
  Base *b = new Derived{ };
  b->test();
}
```

```
ricegf@pluto:~/dev/cpp/201801/14$ make virtual2
g++ -std=c++14 -o virtual2 virtual2.cpp
ricegf@pluto:~/dev/cpp/201801/14$ ./virtual2
Derived
ricegf@pluto:~/dev/cpp/201801/14$
```

- For a *virtual* method, C++ follows the type of the *actual object* (called “polymorphism”*)

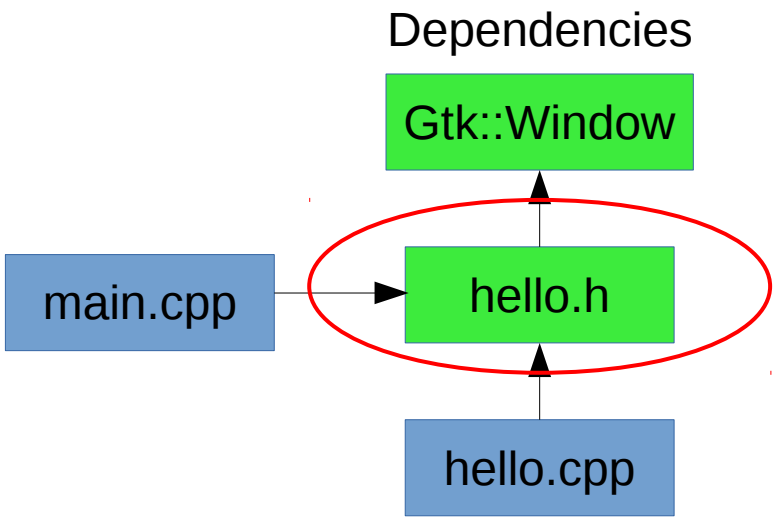
*We’ll come back to polymorphism in much more detail later!

Virtual Destructor

- Similarly, by declaring our destructor as *virtual*, we tell C++ to always use the *derived* destructor rather than the *base* destructor
- tl;dr? You're usually safe to declare an empty virtual destructor when inheriting from Gtk et. al.

hello.h

```
#ifndef _HELLO_H
#define _HELLO_H

#include <gtkmm.h>

class Hello : public Gtk::Window {
  public:
    Hello();
    virtual ~Hello();
  protected:
    void on_button_clicked();
    Gtk::Button button;
};
#endif
```

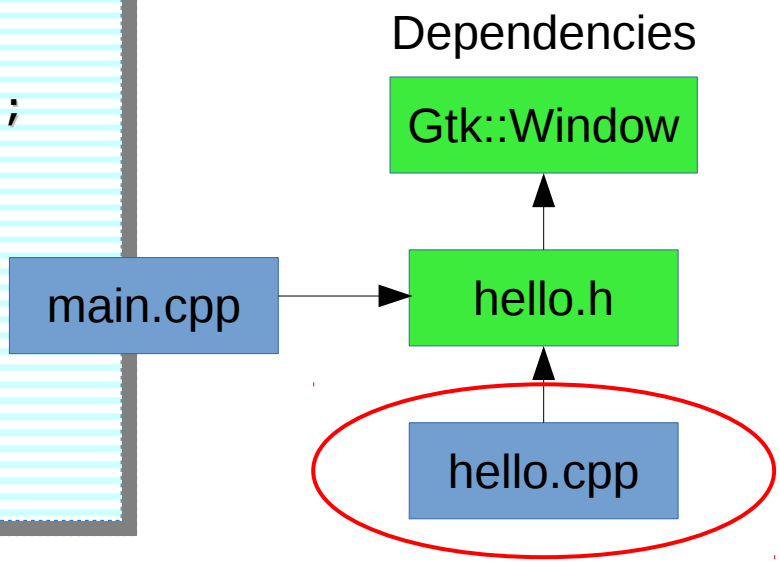


hello.cpp

- Implementations are straightforward
  - Configure window and button in constructor
  - Create an empty virtual destructor (almost always)
  - Define the custom behavior

```
#include "hello.h"
#include <iostream>

Hello::Hello() : button{"Hello, World!"} {
  set_border_width(10);
  button.signal_clicked().connect(
      sigc::mem_fun(*this,&Hello::on_button_clicked));
  add(button);
  button.show();
}

Hello::~Hello() { }

void Hello::on_button_clicked() {
  std::cout << "Hello, World!" << std::endl;
}
```

hello.cpp



button.signal_clicked( ).connect(sigc::mem_fun( )) ??

- Yes, that's a lot of new concepts in one line!
- signal_clicked() is "emitted" when the button is activated (by, you know, *clicking* it)
  - Also available: signal_activate(), _enter(), _leave(), _pressed(), and released()
- connect() accepts a function* that can be called when signal_clicked() is emitted
- sigc::mem_fun() converts an object with one of its methods into a function suitable for connect()

* Technically, a "functor", but the distinction isn't important yet

hello.cpp

- An instance of Hello is thus:
  - A Window (which we inherited from Gtk::Window)
  - ...containing a button (as we instanced and added a Gtk::Button)
  - ...that calls on_button_clicked() when clicked (as we connected)
  - ...that prints “Hello, World!” to STDOUT when called.

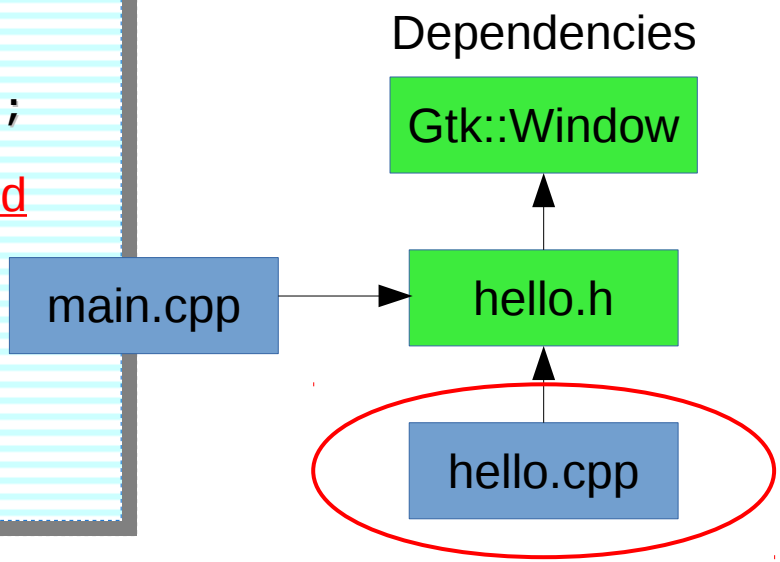
hello.cpp

```
#include "hello.h"
#include <iostream>

Hello::Hello() : button("Hello, World!") {
  set_border_width(10);
  button.signal_clicked().connect(
      sigc::mem_fun(*this,&Hello::on_button_clicked));
  add(button);
  button.show();
}

Hello::~Hello() { }

void Hello::on_button_clicked() {
  std::cout << "Hello, World!" << std::endl;
}
```

Address of the method

Address of the class

In this example, the method is in the same class as the widget

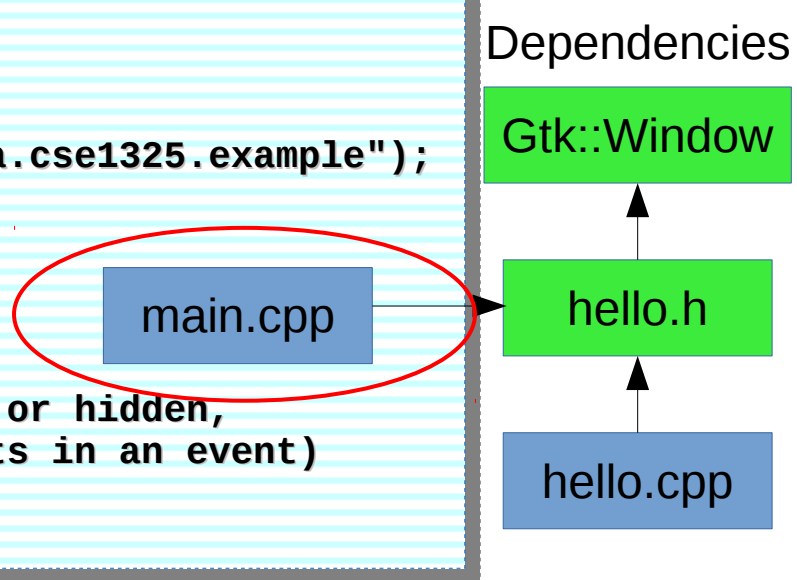


main.cpp

- Main sets up the window and turns over control to the Gtk+ main loop
  - Create an Application using a Gtk+ factory
  - Instance Hello, and pass the Instance to Gtk's run()

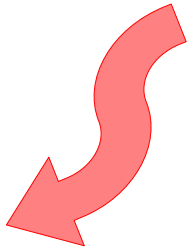
dialog3.cpp

```
#include "hello.h"
#include <gtkmm.h>

int main (int argc, char *argv[])
{
  Glib::RefPtr<Gtk::Application> app =
    Gtk::Application::create(argc, argv, "edu.uta.cse1325.example");
  // NOT Gtk::Main kit(argc, argv);

  // Instance a Window
  Hello hello;

  //Show the window and return when it is closed or hidden,
  //  NOT when a widget is activated (that results in an event)
  return app->run(hello);
}
```



Glib::RefPtr<Gtk::Application> app = Gtk::Application::create( ) ??

- Yes, that's a lot more gibberish. One step at a time.
- Glib::RefPtr is Gtk+'s "reference-counting smart pointer"*
  - We only care because Application::create creates one!**
- The Gtk::Application factory handles a lot of routine aspects of setting up applications for you, saving a lot of rote typing
  - GTK+ and gtkmm initialization (e.g., Gtk::Main kit(argc, argv);)
  - Application uniqueness
  - Session management
  - Basic scriptability and desktop shell integration
  - Manages a list of top-level windows for your application



* Remember unique_ptr and shared_ptr? Yep, they arrived *after* RefPtr. :-(

** Going forward, we'll use "auto app = Gtk::Application::create( )" - much better!
"auto" means "this variables type is whatever type is returned by the expression".

Glib::RefPtr<Gtk::Application> app = Gtk::Application::create( ) ??

- Gtk::Application::create creates a new Application instance
  - The first two parameters are argc and argv, exactly as received via “int main (int argc, char *argv[ ])”
  - The third parameter is the Application ID
  - The last parameter is a set of flags
- All parameters are optional (more or less)
  - See following slides for more info

```
Gtk::Application::create(argc, argv, "edu.uta.cse1325.example");
```

Gtk::Application::create( ) Parameters
argv and argc

- Gtk apps accept standard bash arguments, such as:
  - **--gtk-debug *options*** asks Gtk+ runtime to print certain debug info
    - **Dnd** – Information about drag-and-drop
    - **Nogl** – Turn off OpenGL, as if OpenGL support was not available
    - **Nograbs** – Turn off all pointer and keyboard grabs
    - **Cursor** – Information about cursor objects (only Windows)
    - **Draw** – Information about drawing operations (only Windows)
    - **Eventloop** – Information about event loop operation (mostly Mac)
    - **Xinerama** – Simulate a multi-monitor setup (mostly Linux)
    - **Misc** – Miscellaneous information
  - **--g-fatal-warnings** causes Gtk+ to abort on the first *warning*, useful for catching and analyzing them in a debugger
- Argv/argc may be provide to create( ) OR to run( )

https://developer.gnome.org/gtk3/stable/gtk-running.html

Gtk::Application::create( ) Parameters
Application ID

- This enables Gtk+ to uniquely identify apps as they communicate with each other
- The Application ID is a string following these rules
  - IDs contain 2+ elements separated by periods ('.')
    - Elements consist of letters, numbers, and underscores
    - Elements may not start with a number
    - Elements must contain at least one letter
  - IDs may not begin with a period (‘.’)
- IDs traditionally begin with your reverse domain, followed by the group, application name, and version
  - edu.uta.CSE1325.example_application_id.version1

https://dbus.freedesktop.org/doc/dbus-specification.html#message-protocol-names-interface

Gtk::Application::create( ) Parameters
Flags

- These configure certain aspects of your application's behavior
  - You won't need any of these for homework
  - You won't see any of these on an exam

https://developer.gnome.org/glibmm/unstable/group__giommEnums.html#ga1134403b032ff66482ffece0715c69be

Factory Pattern

Gtk::Application::Create follows the Factory Pattern.
So... What's that?

Creational

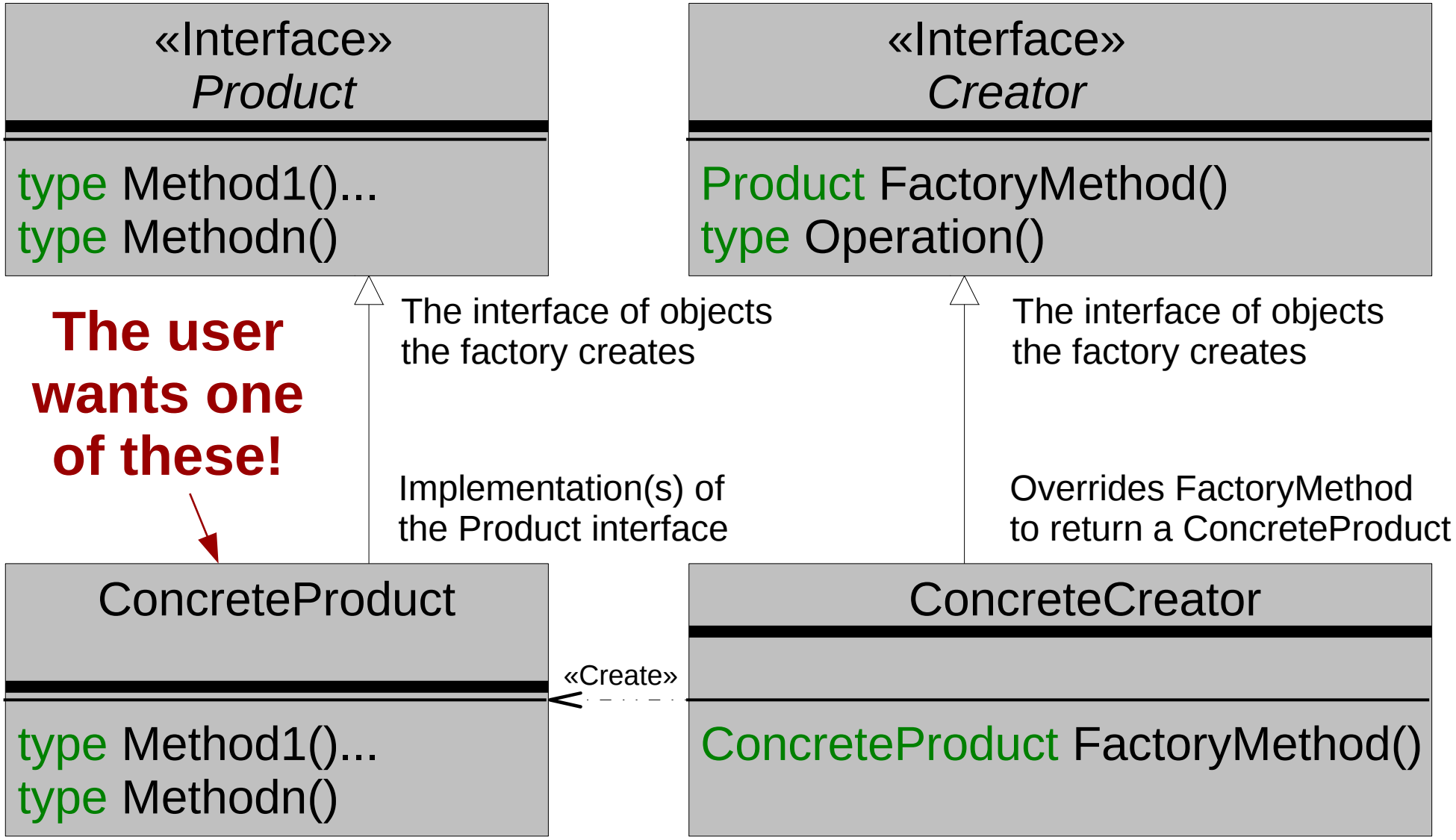
Factory Pattern

- The **Factory** pattern creates new objects without exposing the creation logic to the client
  - Rather than new Object( ), the client calls Factory.getObject(...), which returns the object
  - The client specifies what is needed, and the Factory returns the most appropriate object based on the supplied criteria
  - This may be useful for creating e.g., robot parts!
    - “I need a torso that holds 3 batteries and 1 arm”
    - “I need a locomotor that's fast and low power”

Creational

The Factory Pattern

(Slightly Simplified)

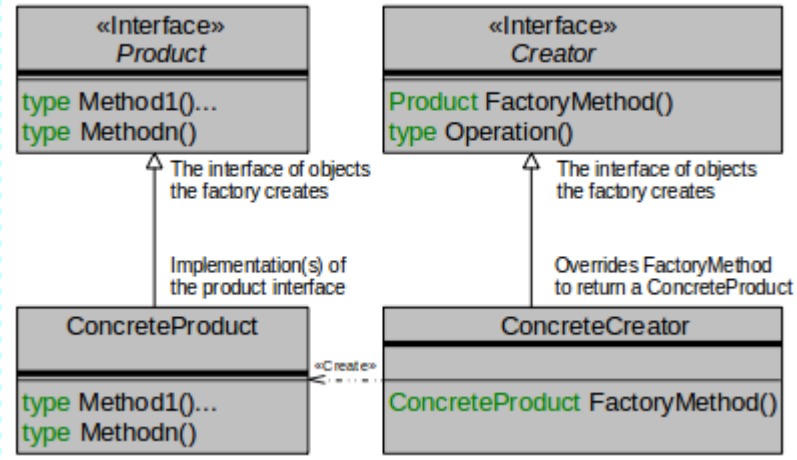


The Factory Pattern
(Slightly Simplified)

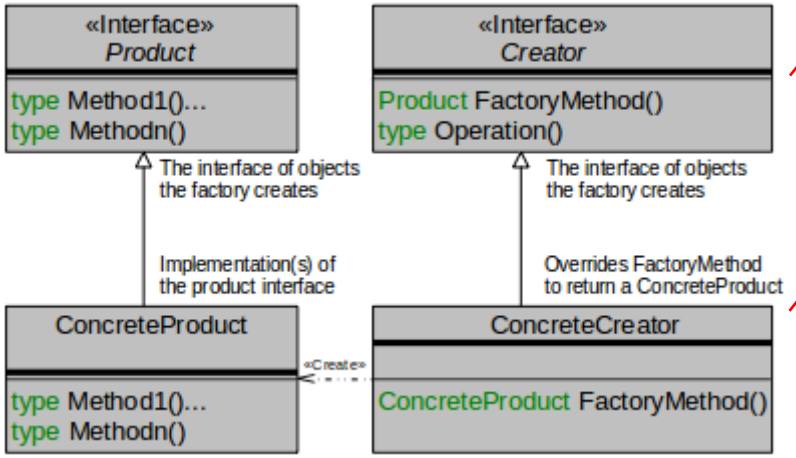
factory.cpp

Our chatty robots return...

```
#include <iostream>
#include <vector>

// The product interface
class Robot {
  public:
    virtual void announce() = 0;
};
// The concrete product classes
class WalkingRobot : public Robot {
  public:
    void announce() override {
      std::cout << "Make way!" << std::endl;
    }
};
class FlyingRobot : public Robot {
  public:
    void announce() override {
      std::cout << "Heads up!" << std::endl;
    }
};
class DivingRobot : public Robot {
  public:
     void announce() override {
       std::cout << "Glub! Glub!" << std::endl;
     }
};
```



The Factory Pattern
(Slightly Simplified)

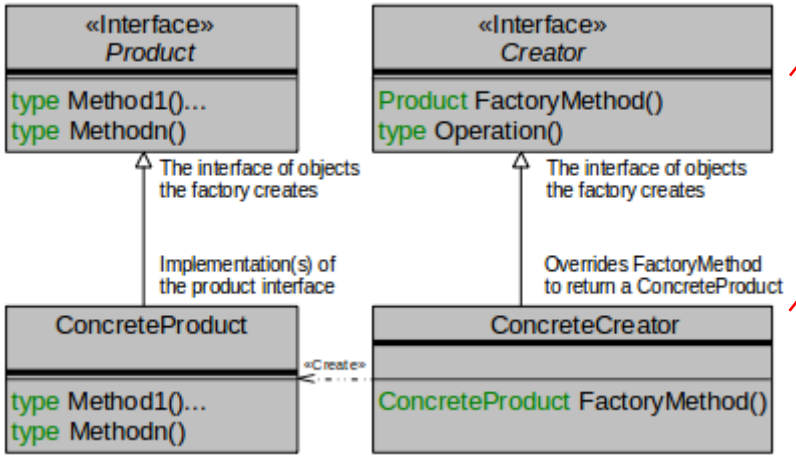


factory.cpp

```
// The Creator interface
class RobotCreator {
  public:
    static Robot* getRobot(std::string criteria);
};

// The concrete creator (factory) class
class RobotFactory : public RobotCreator {
  public:
    static Robot* getRobot(string criteria) {
      if (criteria == "walks")
        return new WalkingRobot();
      else if (criteria == "flies")
        return new FlyingRobot();
      else
        return new DivingRobot();
    }
};
```

```
// Testing the Robot Factory
int main() {
    std::vector<std::string> criteria = {"walks", "flies", "dives"};
    for(std::string s: criteria) {
      Robot* robot = RobotFactory::getRobot(s);
      robot->announce();
    }
};
```

The Factory Pattern
(Slightly Simplified)



factory.cpp

```
// The Creator interface
class RobotCreator {
  public:
    static Robot* getRobot(std::string criteria);
};

// The concrete creator (factory) class
class RobotFactory : public RobotCreator {
  public:
    static Robot* getRobot(string criteria) {
      if (criteria == "walks")
        return new WalkingRobot();
      else if (criteria == "flies")
        return new FlyingRobot();
      else
        return new DivingRobot();
    }
};
```

```
// Testing the Robot Factory
int main() {
    std::vector<std::string> criteria = {"walks", "flies", "dives"};
    for(std::string s: criteria) {
      Robot* robot = RobotFactory::getRobot(s);
      robot->announce();
    }
};
```

```
ricegf@pluto:~/dev/cpp/201801/14$ make factory
g++ -std=c++14 -o factory factory.cpp
ricegf@pluto:~/dev/cpp/201801/14$ ./factory
Make way!
Heads up!
Glub! Glub!
ricegf@pluto:~/dev/cpp/201801/14$
```

Bottom Line

- For connect and Application, just follow these templates for the 2 “magic” lines below for now

Registering an Observer

```
Hello::Hello() : button("Hello, World!") {
  set_border_width(10);
  button.signal_clicked().connect(
      sigc::mem_fun(*this,&Hello::on_button_clicked));
  add(button);
  button.show();
}
```

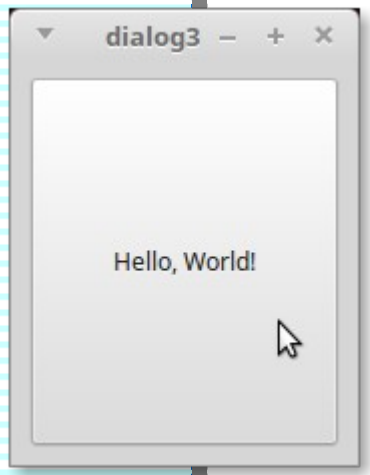
Creating an Application

```
#include "hello.h"
#include <gtkmm.h>

int main (int argc, char *argv[])
{
  auto app = Gtk::Application::create(argc, argv, "edu.uta.cse1325.example");

  Hello hello;

  return app->run(hello);
}
```

Running Hello

Makefile

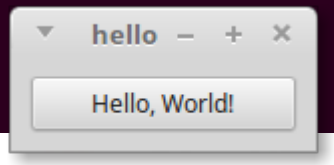
```
CXXFLAGS += -std=c++14
GTKFLAGS = `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`

all: dialog1 dialog2 dialog3 virtual1 virtual2 factory observer

debug: CXXFLAGS += -g
debug: rebuild

dialog3: dialog3.cpp hello.o *.h
	$(CXX) $(CXXFLAGS) -o dialog3 dialog3.cpp hello.o  $(GTKFLAGS)

hello.o: hello.cpp *.h
	$(CXX) $(CXXFLAGS) -c hello.cpp $(GTKFLAGS)
```



Resizing, too!

```
ricegf@pluto:~/dev/cpp/201801/14$ make dialog3
g++ -std=c++14 -c hello.cpp `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
g++ -std=c++14 -o dialog3 dialog3.cpp hello.o  `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
ricegf@pluto:~/dev/cpp/201801/14$ ./dialog3
Hello, World!
Hello, World!
ricegf@pluto:~/dev/cpp/201801/14$
```

I clicked the “Hello, World!” button twice, then ‘x’!



Managing Widget Memory

- Consider our constructor:

```
Hello::Hello() : button{"Hello, World!"} {
  set_border_width(10);
  button.signal_clicked().connect(
      sigc::mem_fun(*this,&Hello::on_button_clicked));
  add(button);
  button.show();
}
```

- ‘`button{"Hello, World!"}`’ instances a Button, passing “Hello, World!” to the constructor
  - The button object is stored as part of the Window object, and is thus deleted with the Window object

(Mis)Managing Widget Memory

- It's common to NOT store the widget as part of the Window object, when we don't need access later – we "create, connect, add, and forget"

```
Hello::Hello() {
  set_border_width(10);
  Gtk::Button *button = new Gtk::Button("Hello, World!"));
  button->signal_clicked().connect(
      sigc::mem_fun(*this,&Hello::on_button_clicked));
  add(*button);
  button->show();
}
```

Create
Connect
Add
Forget

- This works just as well, and keeps the interface (in hello.h) cleaner
- Question: When is the button above deleted?

(Mis)Managing Widget Memory

- Answer: It isn't!
  - The object is independently created on the heap
  - When the Window object is deleted, no references remain to the button – **memory leak!**
- Gtk offers a solution – the "managed" widget
  - Gtk will mark a Gtk widget as "managed"
  - When a widget containing one or more "managed" widgets (e.g., our Window) is deleted, it will also delete all of the "managed" widgets added to it, recursively

Managed Widgets

- We mark an instance of a Gtk widget as “managed” by passing a pointer to the widget to the Gtk::manage function

```
Hello::Hello() {
  set_border_width(10);
  Gtk::Button *button = Gtk::manage(new Gtk::Button("Hello, World!"));
  button->signal_clicked().connect(
      sigc::mem_fun(*this,&Hello::on_button_clicked));
  add(*button);
  button->show();
}
```

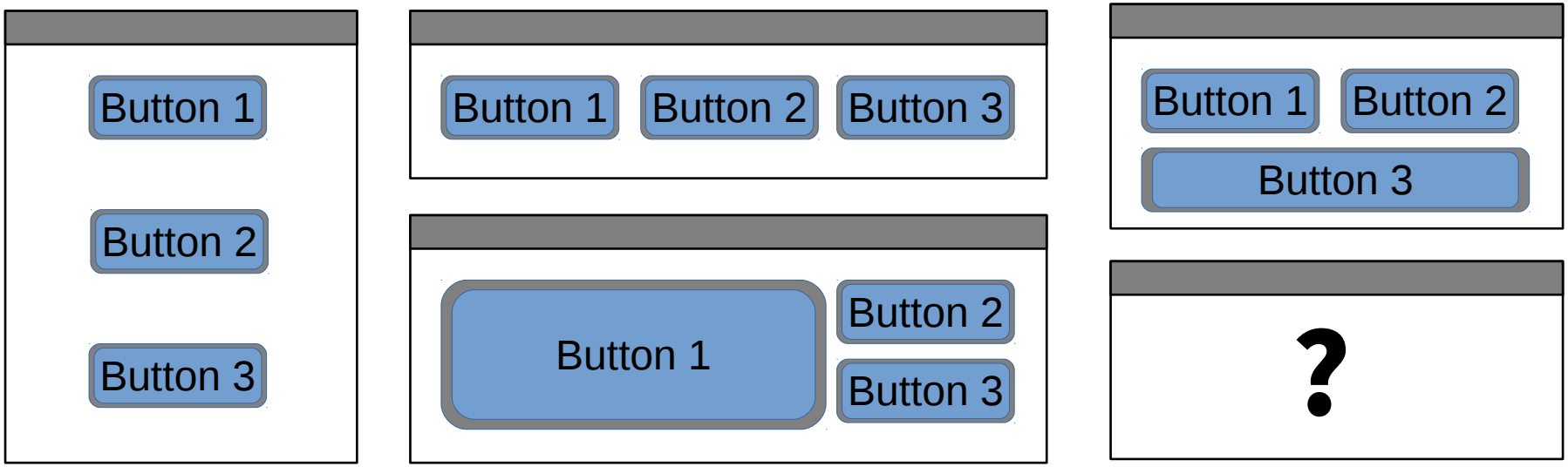
Create
Connect
Add
Forget

Now, the Window’s destructor will “know” to delete the managed button we added!

https://developer.gnome.org/gtkmm-tutorial/stable/sec-memory-widgets.html.en#memory-managed-widgets

What If We Need 3 Buttons Instead of than 1 Button?

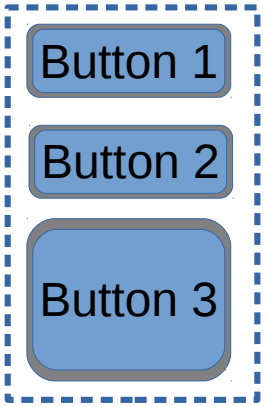
- First problem: A Window object can contain exactly 1 widget. (Seriously?)
  - We need a way to add as many widgets as we like
- Second problem: How will gtkmm arrange the buttons in our window?



Multi-Widget Containers

- Some widgets are (invisible) “containers”
  - Unlike Window, they can contain multiple widgets
  - They include the ability to specify layout

“VBox”



Gtk::ORIENTATION_VERTICAL

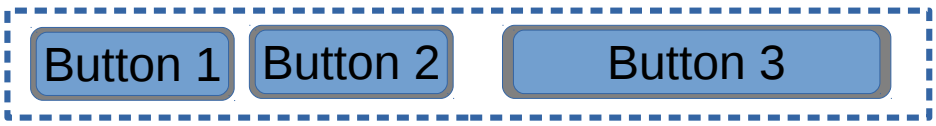
Gtk::Box

A single row or column of widgets, each of which may be *individually* configured.

Gtk::Toolbar

An array of buttons et. al.

“HBox”

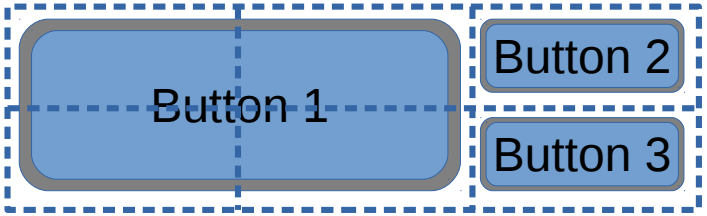


Gtk::ORIENTATION_HORIZONTAL

Gtk::ButtonBox

A single row or column of widgets that are configured *as a group*.

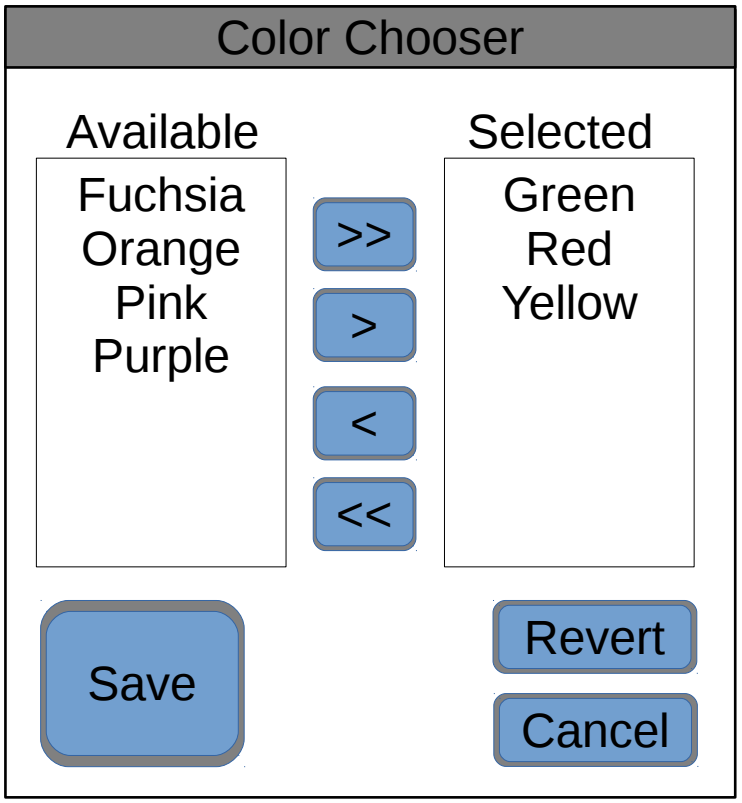
Gtk::Grid



Grid is a 2D array of widgets, each *individually* configured.

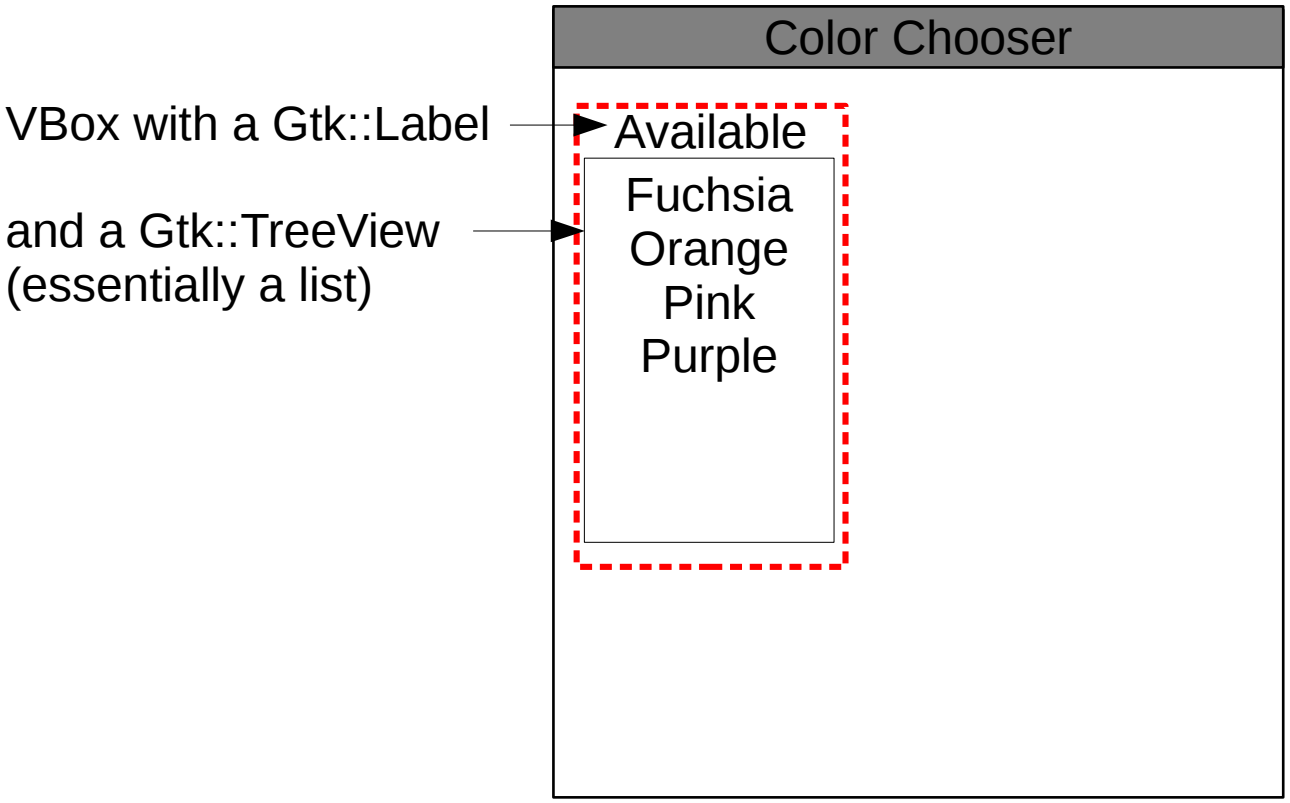
Nested Containers

- Container widgets may be nested to create complex dialogs and windows



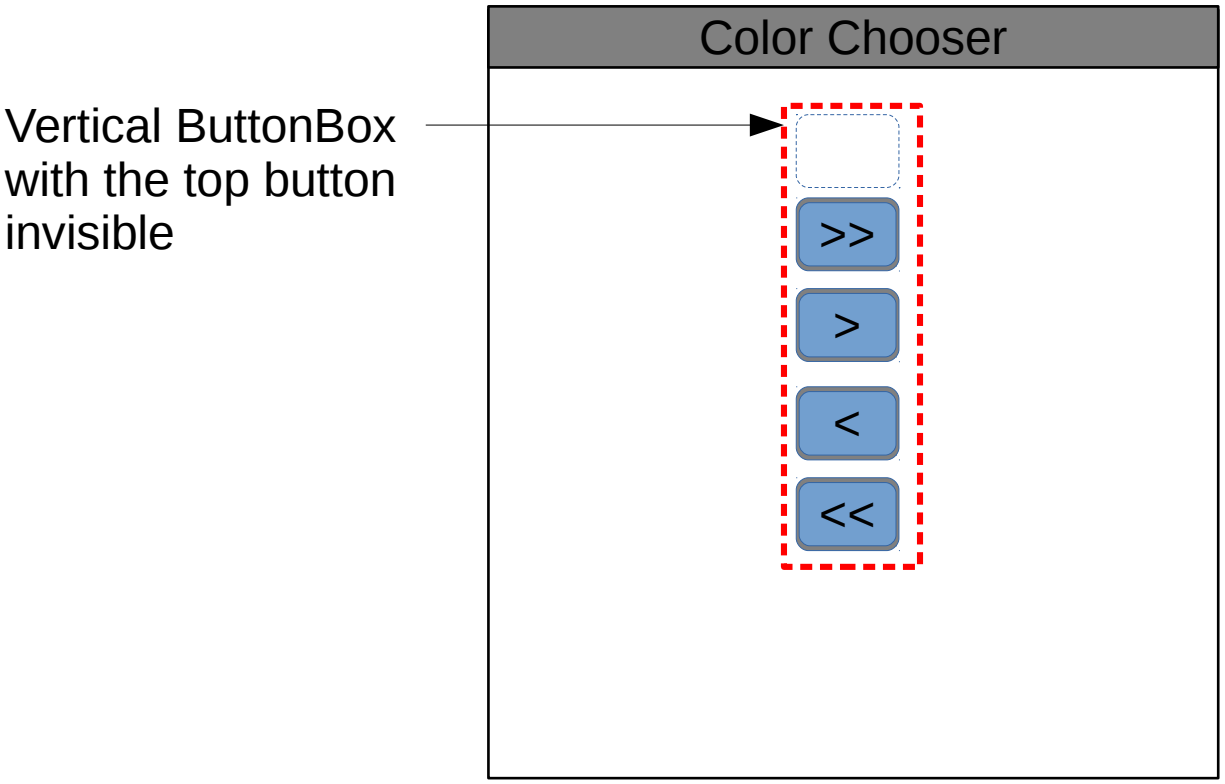
Nested Containers

- Container widgets may be nested to create complex dialogs and windows



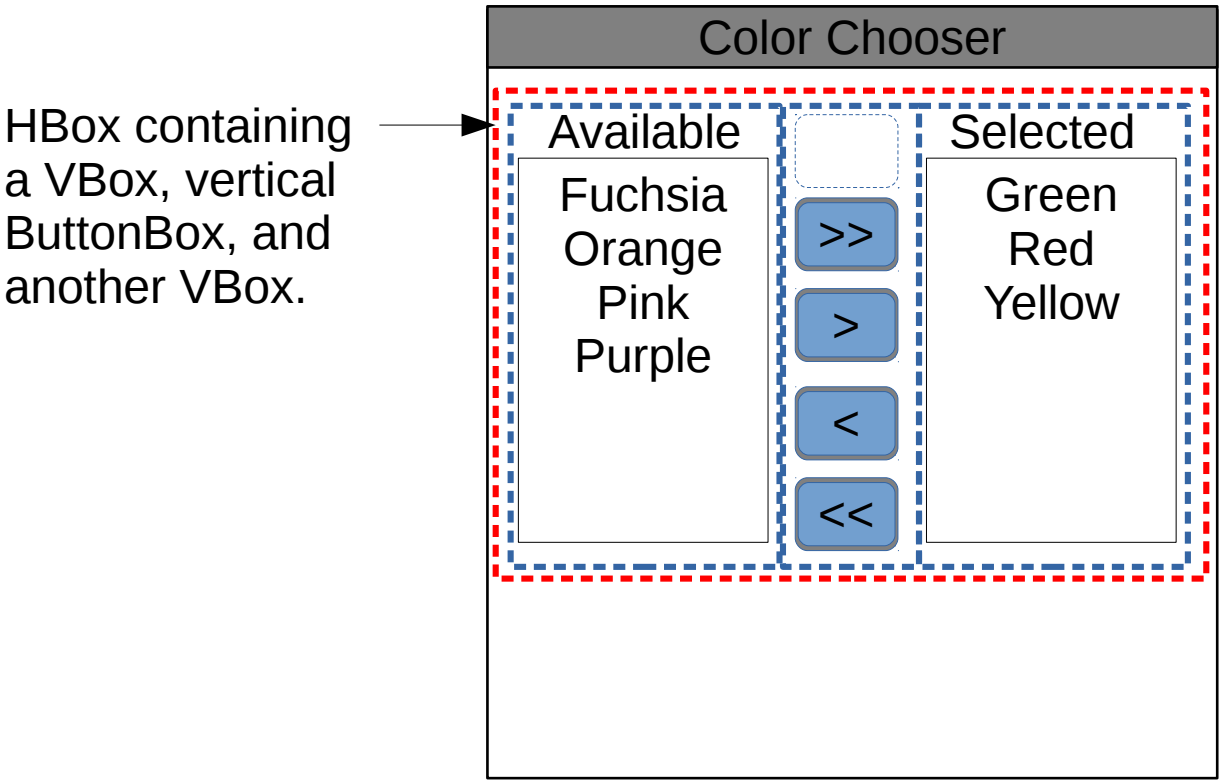
Nested Containers

- Container widgets may be nested to create complex dialogs and windows



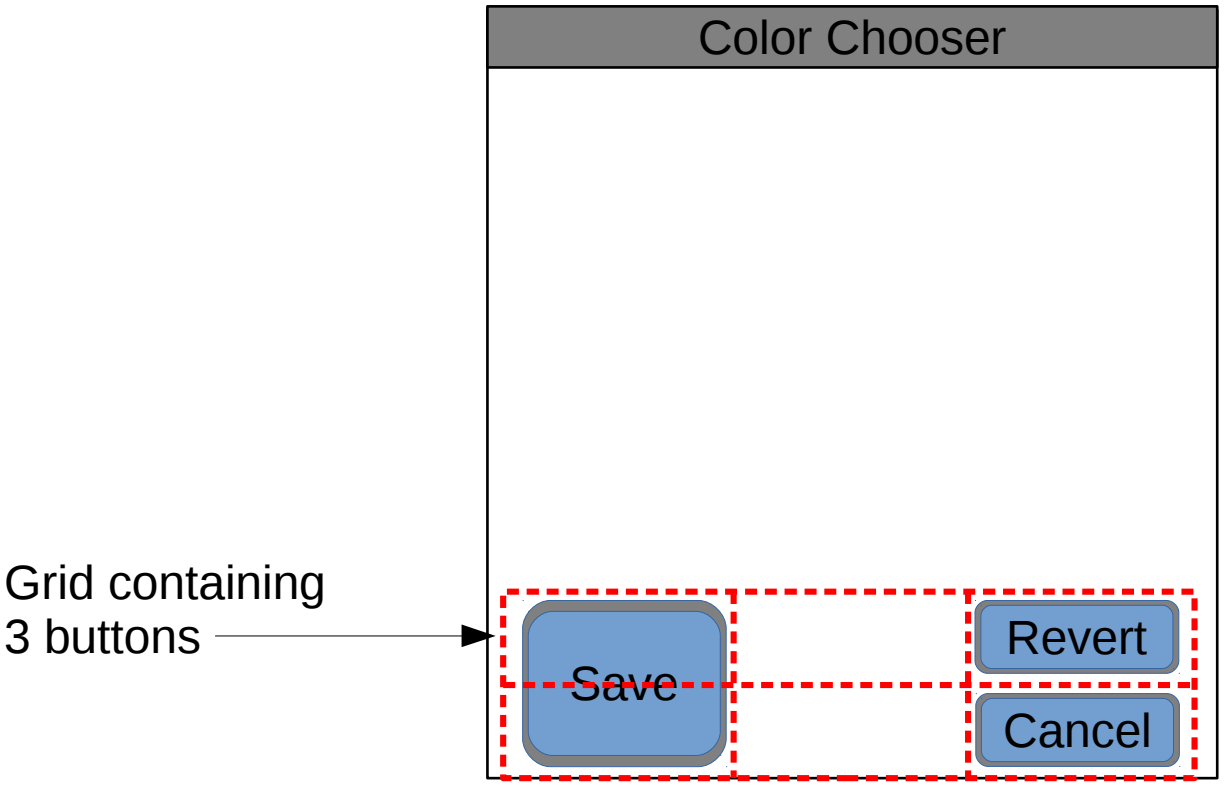
Nested Containers

- Container widgets may be nested to create complex dialogs and windows



Nested Containers

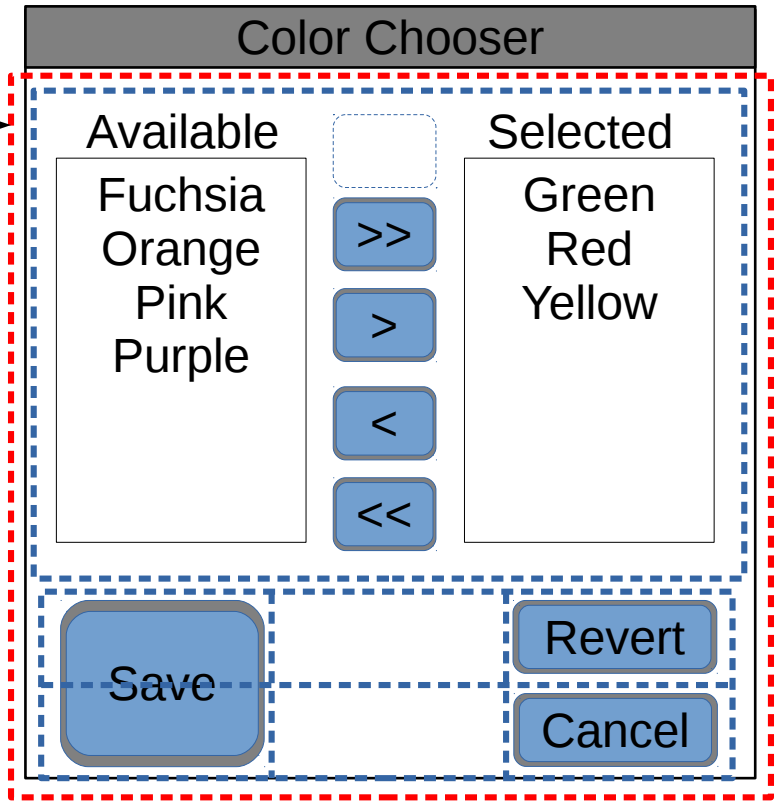
- Container widgets may be nested to create complex dialogs and windows



Nested Containers

- Container widgets may be nested to create complex dialogs and windows

Finally, a VBox containing the HBox and Grid

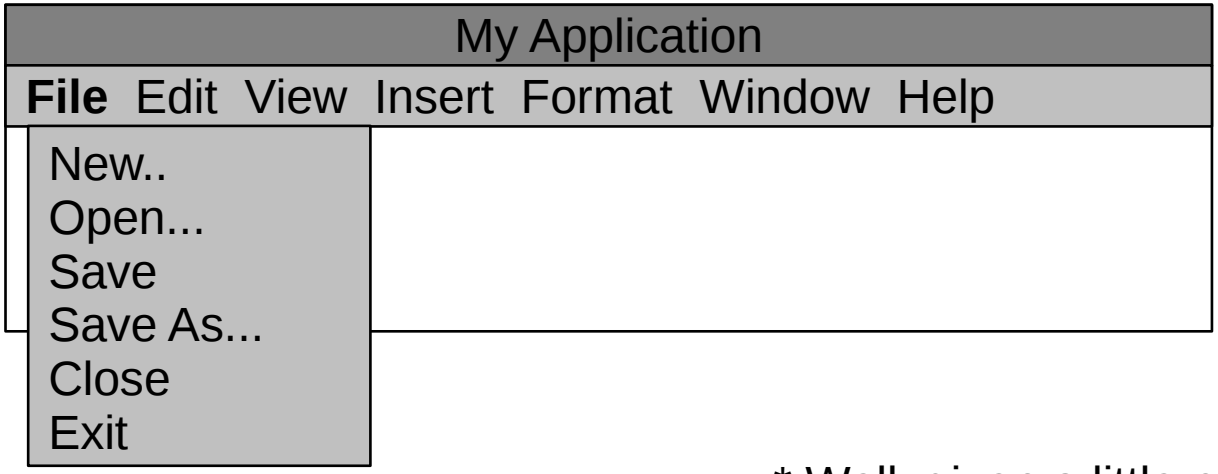


And the VBox is the ONLY widget in the Window!

The reason Window can contain only 1 widget is because we can't know in advance how to lay out multiple widgets in Window. YOU decide – by selecting the container widget to add to the Window!

Creating a Menu Bar

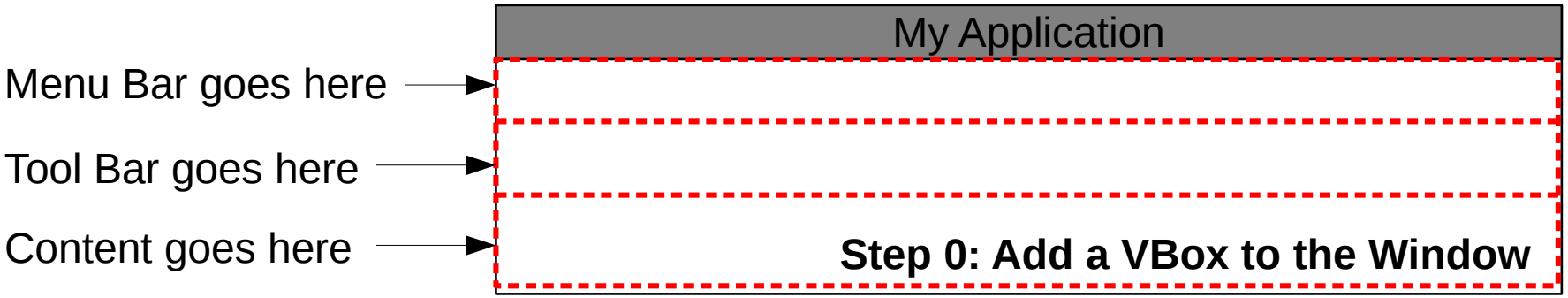
- We* can now create arbitrarily complex widget arrangements
  - Boxes and Grids of widgets
  - Tool Bars via Toolbar
- All we need now is a Menu Bar
  - Step by step procedure follows



* Well, given a little practice, you can! ☺

Creating a Menu Bar

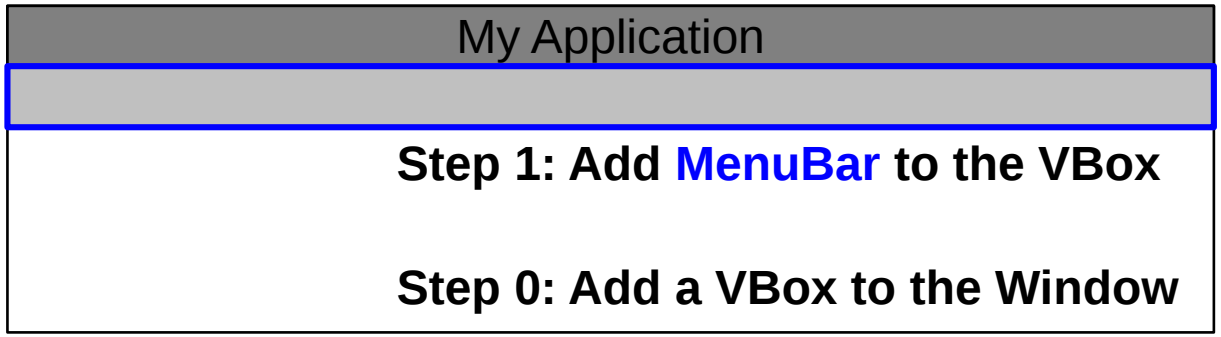
- First, add a vertical Box to the Window
  - This will allow us to add (top to bottom) a menu bar, a tool bar, and a container for our content



https://developer.gnome.org/gtkmm/stable/classGtk_1_1Box.html

Creating a Menu Bar

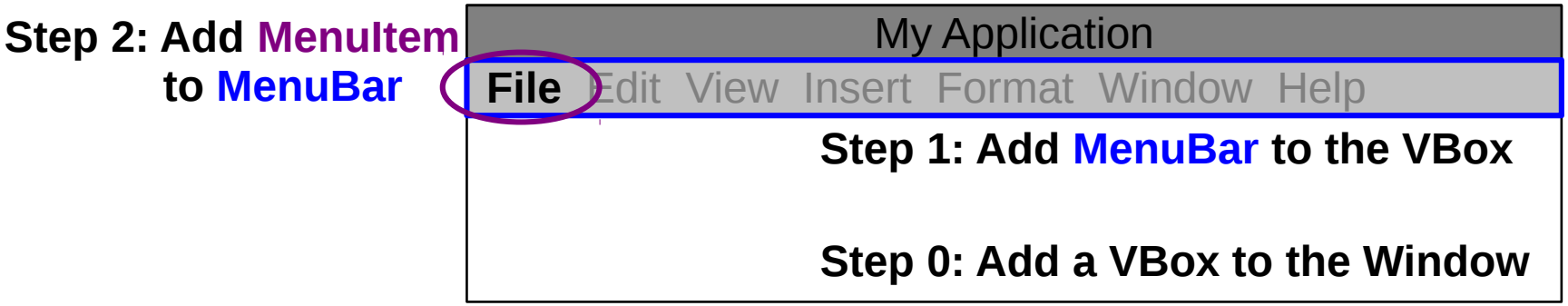
- Second, instance a Gtk::MenuBar and add it to the vertical Gtk::Box instance in the Window
  - Its append method will enable you to add drop-down menu items such as File, Edit, View, ...



https://developer.gnome.org/gtkmm/stable/classGtk_1_1MenuBar.html

Creating a Menu Bar

- Instance a Gtk::MenuItem, e.g., for File, and append it to the Gtk::MenuBar instance
  - Do this for each additional top-level menu, e.g., Edit, View, …, Help

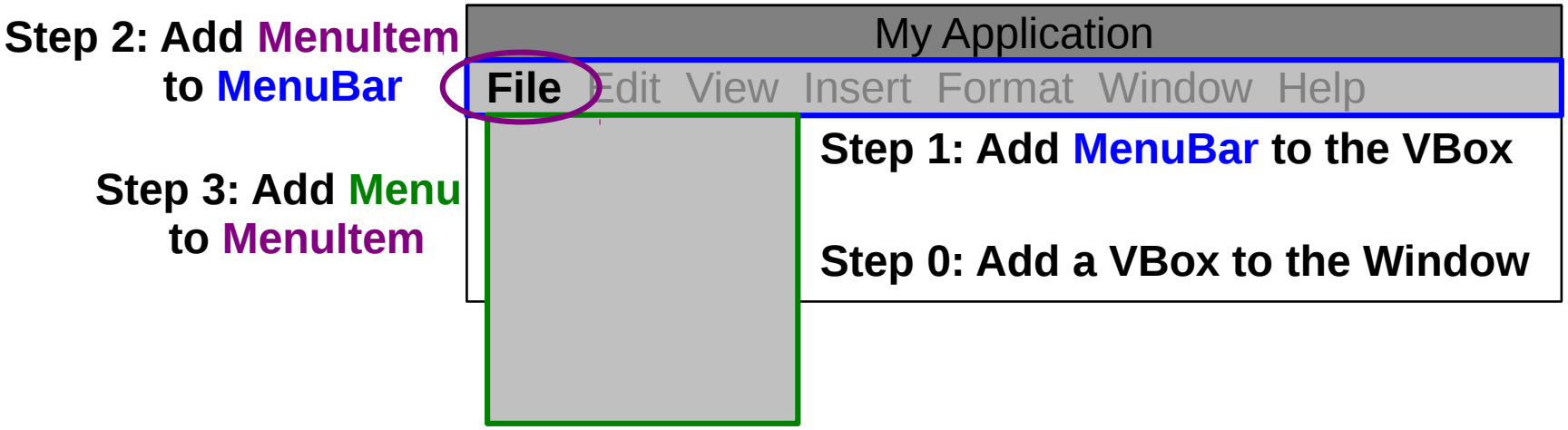


https://developer.gnome.org/gtkmm/stable/classGtk_1_1MenuItem.html

Creating a Menu Bar

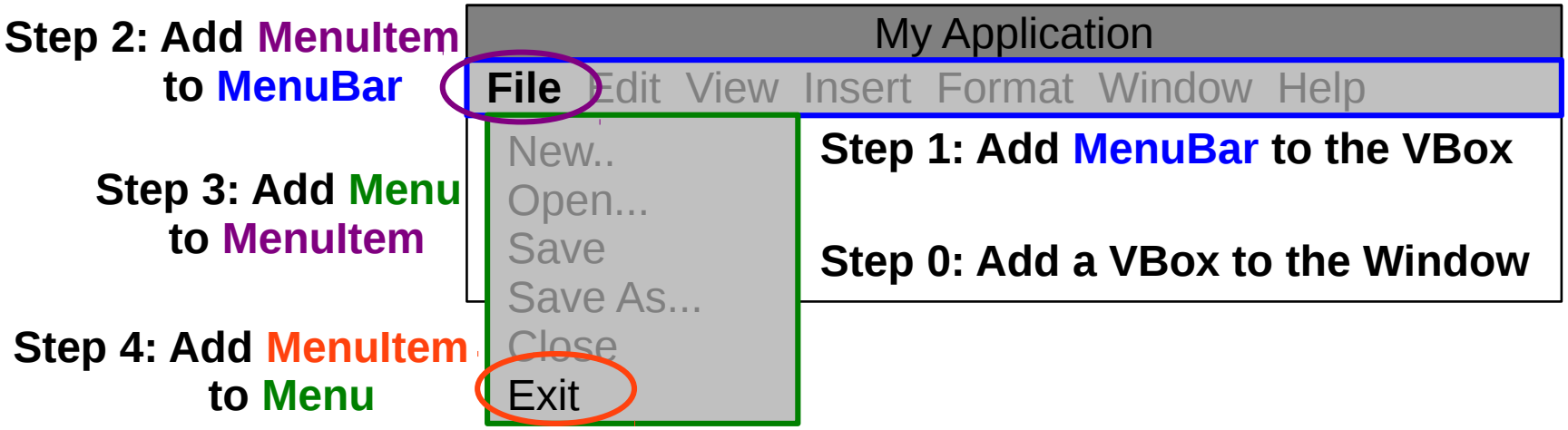
- For each top-level menu item, instance Gtk::Menu, which is the “drop down” functionality
  - Add using Gtk::MenuItem’s set_submenu method

https://developer.gnome.org/gtkmm/stable/classGtk_1_1Menu.html



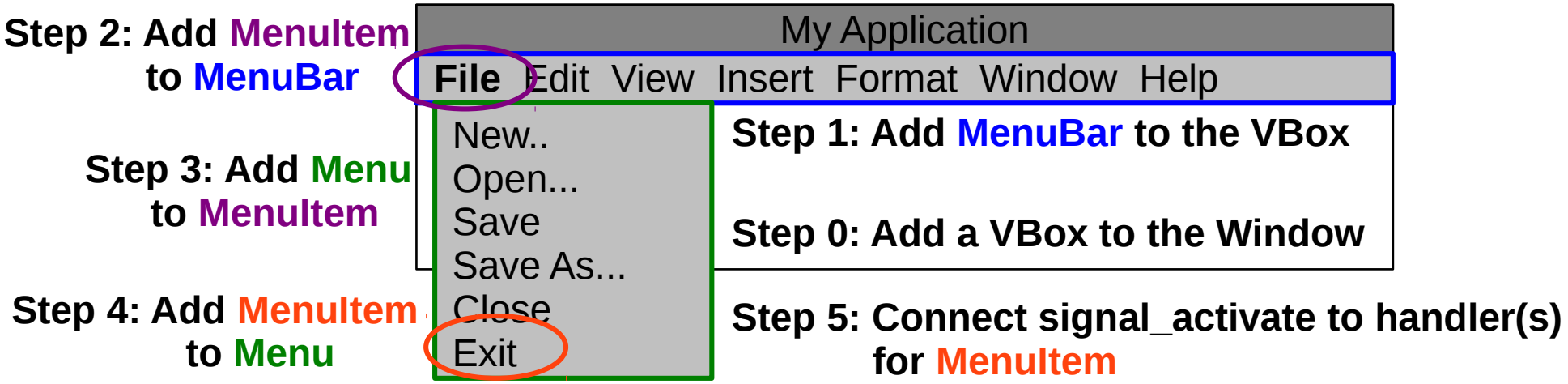
Creating a Menu Bar

- To populate the drop-down menu, instance more Gtk::MenuItem objects
  - Add to the Gtk::Menu instance using its append method



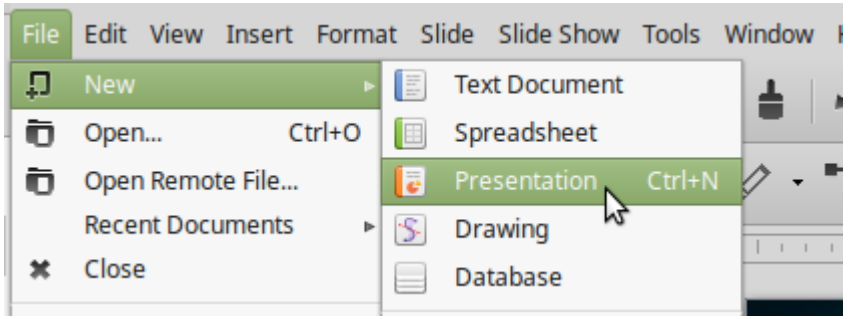
Creating a Menu Bar

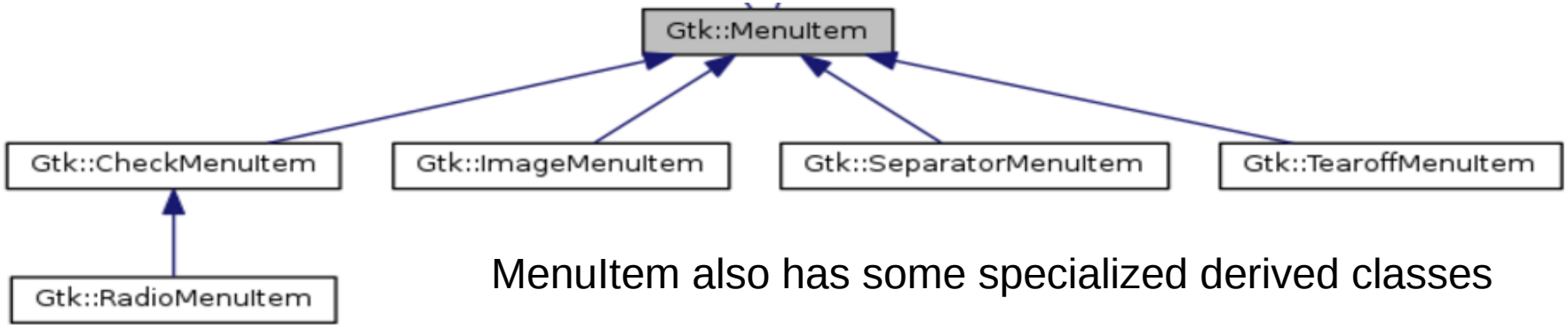
- Finally, connect your handler methods for each menu item to the associated Gtk::MenuItem instance
  - The same handler can service both menu items and tool bar buttons



Fancier Gtk::MenuItem Methods

- Menu items can change in response to events
  - Additional levels can be added via set_submenu
    - Same as above
  - Attributes can change
    - set_label to change text
    - set_visible to show or hide
    - set_sensitive to enable or disable (grey) response to click
    - set_tooltip_text sets tool tip displayed on mouse hover

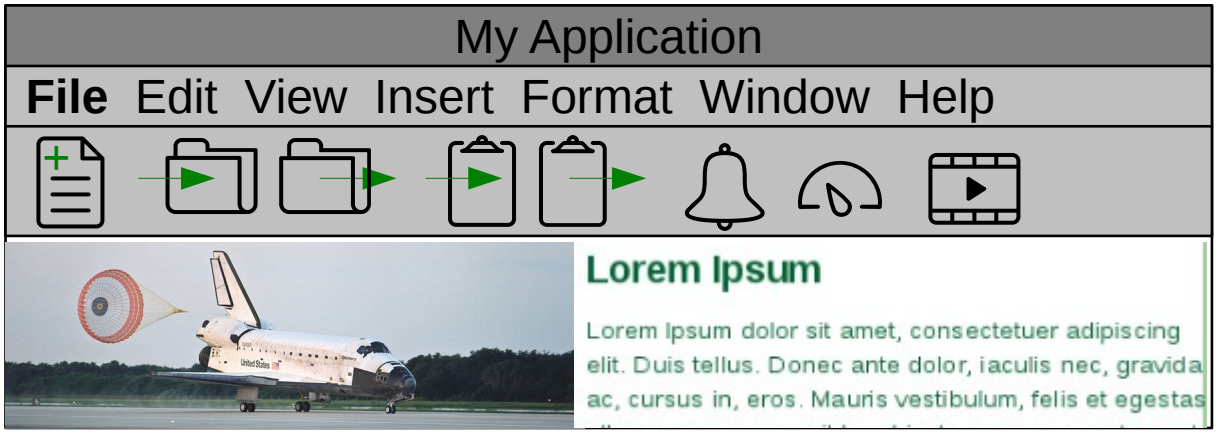




MenuItem also has some specialized derived classes

Putting It All Together

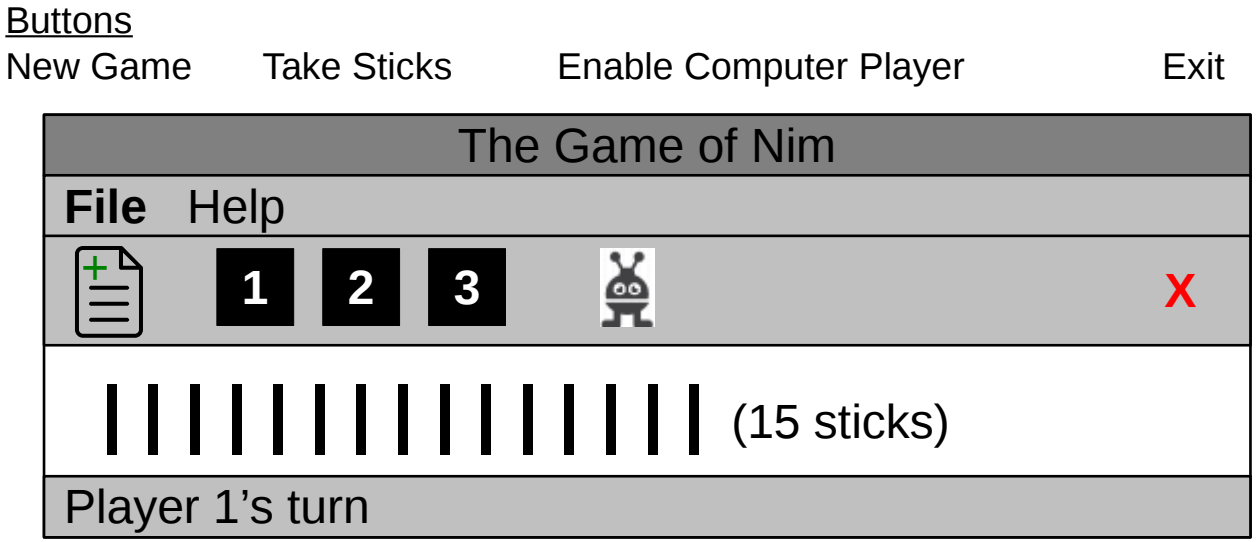
- We can now create a “main window”
  - Menu bar and tool bar
  - Boxes and Grids of widgets
  - Content
- Let’s review code for simple C++ GUI application

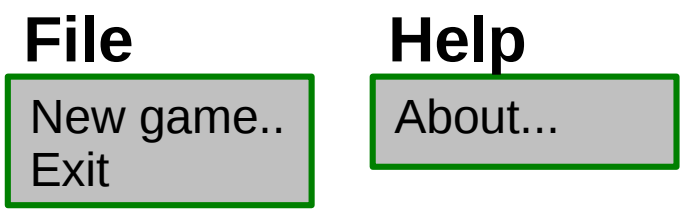


Space Shuttle Discovery image is public domain per https://www.nasa.gov/multimedia/guidelines/index.html
Icons adapted from Umar Irshad’s “Bubbles” icon set, https://iconfinder.com, “Free for Commercial Use”

The Game of Nim

- A set of sticks is piled between the 2 players
  - Each player in turn takes 1, 2, or 3 sticks
  - The player who draws the last stick loses
- We'll write this game for 1 or 2 human players





Moves that are unavailable (e.g., fewer than 3 sticks remain) will be insensitive with buttons "greyed out".

Icons adapted from Umar Irshad's "Bubbles" icon set, https://iconfinder.com, "Free for Commercial Use"

Let's Review One Possible gtkmm Implementation of Nim

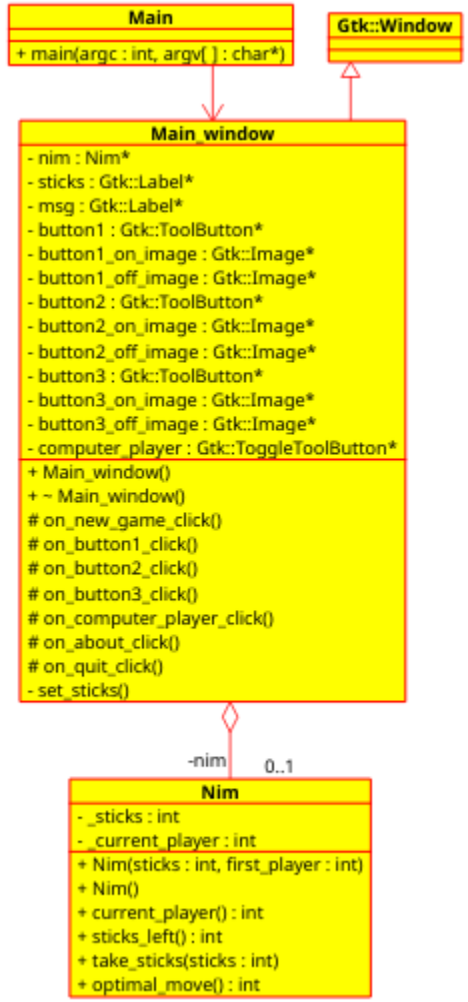
- IMPORTANT: This app was developed in "baby steps", using git, ddd, and test code, as usual
  - Here we're reviewing *code only*, as one way that gtkmm can be used to implement a classic GUI interface
  - We'll review the *process* in lectures 16 & 17
- We'll look at a "code-centric" GUI implementation, though others exist, e.g.,
  - Use GLADE to drag-and-drop the interface design, save as an XML file, and use Gtk::Builder to load and show
  - Use helper libraries that pre-select sane defaults to reduce the amount of code we have to write

Nim Interface

- An instance of this class manages one game
  - To create a new game, instance Nim again

nim.h

```
#ifndef _NIM_H_
#define _NIM_H_

class Nim {
  public:
    Nim(int sticks, int first_player = 1);
    Nim();                          // All options randomized
    int current_player();           // Player to move next
    int sticks_left();              // Number of sticks in pile
    void take_sticks(int sticks); // Remove sticks from pile
    int optimal_move();             // Best move to win
  private:
    int _sticks;                    // Number of sticks in pile
    int _current_player;            // Player to move next
};

#endif
```



Nim Implementation

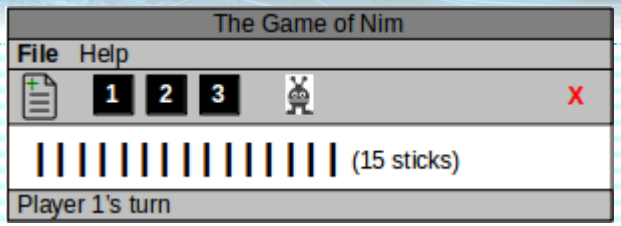
(The “Model” - Irrelevant to the GUI, of course)

nim.cpp

```
#include "nim.h"
#include <stdexcept>
#include <cstdlib>

Nim::Nim(int sticks, int first_player)
       : _sticks{sticks}, _current_player{first_player} {
}

Nim::Nim() {
  srand (time(NULL));
  _sticks = std::rand() % 9 + 7;
  _current_player = std::rand() % 2 + 1;
}

int Nim::current_player() {
  return _current_player;
}

int Nim::sticks_left() {
  return _sticks;
}

void Nim::take_sticks(int sticks) {
  if (_sticks >= sticks) _sticks -= sticks;
  else throw std::runtime_error("Out of sticks!");
  if (_sticks > 0) _current_player = 3 - _current_player;
}
```

```
int Nim::optimal_move() {
  if (_sticks <= 1) return 1;
  switch(_sticks%4) {
    case 0: return 3;
    case 1: return std::rand()%3+1;
    case 2: return 1;
    case 3: return 2;
    default: throw std::runtime_error
       ("Internal calculation error!");
  }
}
```

Main Window Interface
(Page 1 of 2)



main_window.h

```
#ifndef MAIN_WINDOW_H
#define MAIN_WINDOW_H

#include <gtkmm.h>
#include "nim.h"

class Main_window : public Gtk::Window {
    public:
        Main_window();
        virtual ~Main_window();
    protected:
        void on_new_game_click();          // Create a new game
        void on_button1_click();           // Select one stick from pile
        void on_button2_click();           // Select two sticks from pile
        void on_button3_click();           // Select three sticks from pile
        void on_computer_player_click(); // Enable / disable computer player
        void on_about_click();             // Display About dialog
        void on_quit_click();              // Exit the game

    private:
        void set_sticks();                 // Update display, robot move
        Nim *nim;
```

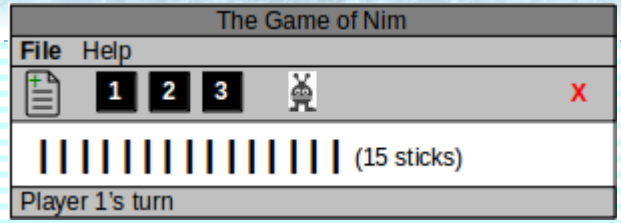
Constructor and Destructor

Callbacks (from gtkmm widgets)

Utility and pointer to game instance

(Continued on next slide)

Main Window Interface
(Page 2 of 2)



main_window.h

GUI widgets

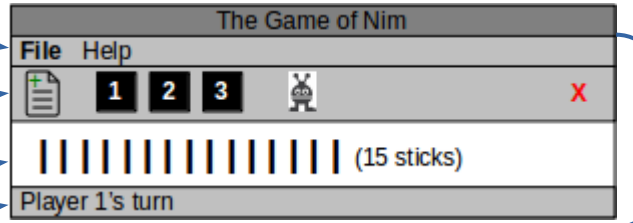
```
        Gtk::Label *sticks;                    // Display of sticks on game board
        Gtk::Label *msg;                       // Status message display
        Gtk::ToolButton *button1;              // Button to select 1 stick
        Gtk::Image *button1_on_image;          //   Image when active
        Gtk::Image *button1_off_image;         //   Image when inactive
        Gtk::ToolButton *button2;              // Button to select 2 sticks
        Gtk::Image *button2_on_image;
        Gtk::Image *button2_off_image;
        Gtk::ToolButton *button3;              // Button to select 3 sticks
        Gtk::Image *button3_on_image;
        Gtk::Image *button3_off_image;
        Gtk::ToggleToolButton *computer_player;  // Button to enable robot
};
#endif
```

Notes:

1) You only need to retain pointers to widgets that you plan to modify in callbacks. gtkmm retains references to all widgets that you add to containers.
2) If the window closes, all Gtk-managed widgets will be deleted automatically. If you try to dereference them later via these pointers, you'll likely cause a segfault!

Main_window Constructor Implementation Outline

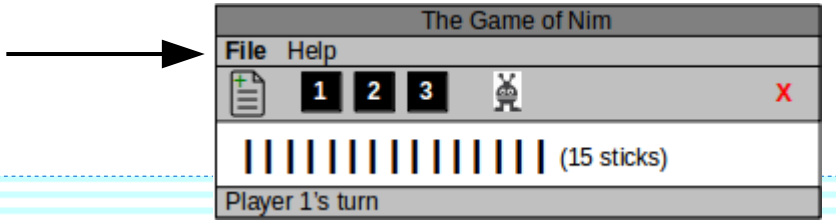
- Add a VBox to the Window (from which Main_window is derived)
- Add the menu at the top of the vbox
  - Add a File drop-down menu
    - Add New Game to the File menu
    - Add Quit to the File menu
  - Add a Help drop-down menu
    - Add About to the Help menu
- Add the toolbar just below the menu
  - Add a New Game button
  - Add 1 stick, 2 stick, and 3 stick buttons
  - Add a 2-state Computer Player button – when down, computer is Player 2
  - Add a Quit button aligned to the right of the toolbar
- Add the Sticks display below the toolbar, which resizes with the window
- Add the Status Bar display at the bottom



Main_window Constructor
(Page 1 of 7)

Here's the code (in 7 slides)
to implement that outline

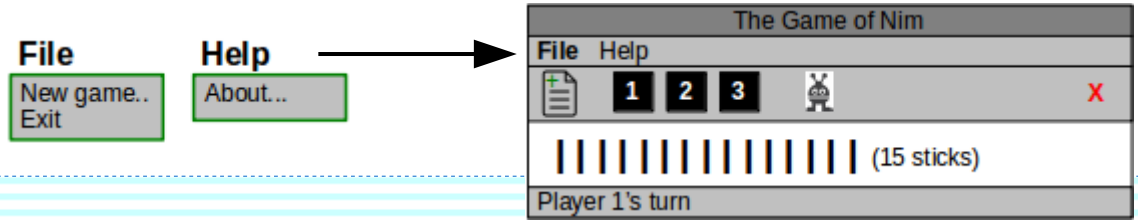
main_window.cpp




```
Main_window::Main_window() {

    // ////////////////
    // G U I   S E T U P
    // ////////////////

    set_default_size(400, 200);

    // Put a vertical box container as the Window contents
    Gtk::Box *vbox = Gtk::manage(new Gtk::Box(Gtk::ORIENTATION_VERTICAL, 0));
    add(*vbox);

    // ///////
    // M E N U
    // Add a menu bar as the top item in the vertical box
    Gtk::MenuBar *menubar = Gtk::manage(new Gtk::MenuBar());
    vbox->pack_start(*menubar, Gtk::PACK_SHRINK, 0);
```

Main_window Constructor
(Page 2 of 7)

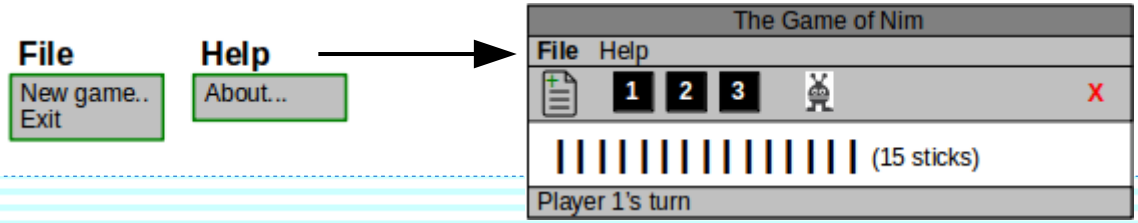
main_window.cpp




```
//     F I L E
// Create a File menu and add to the menu bar
Gtk::MenuItem *menuitem_file = Gtk::manage(new Gtk::MenuItem("_File", true));
menubar->append(*menuitem_file);
Gtk::Menu *filemenu = Gtk::manage(new Gtk::Menu());
menuitem_file->set_submenu(*filemenu);

//         N E W   G A M E
// Append New to the File menu
Gtk::MenuItem *menuitem_new = Gtk::manage(new Gtk::MenuItem("_New Game", true));
menuitem_new->signal_activate().connect(sigc::mem_fun(*this,
    &Main_window::on_new_game_click));
filemenu->append(*menuitem_new);

//         Q U I T
// Append Quit to the File menu
Gtk::MenuItem *menuitem_quit = Gtk::manage(new Gtk::MenuItem("_Quit", true));
menuitem_quit->signal_activate().connect(sigc::mem_fun(*this,
    &Main_window::on_quit_click));
filemenu->append(*menuitem_quit);
```

Main_window Constructor
(Page 3 of 7)

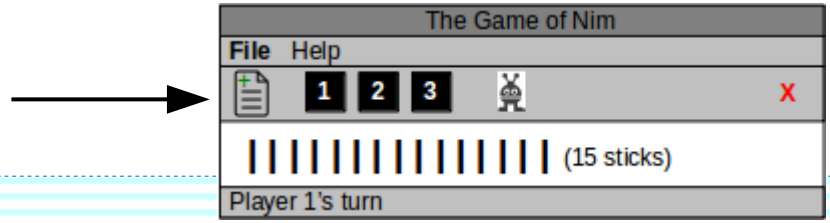
main_window.cpp




```
//     H E L P
// Create a Help menu and add to the menu bar
Gtk::MenuItem *menuitem_help = Gtk::manage(new Gtk::MenuItem("_Help", true));
menubar->append(*menuitem_help);
Gtk::Menu *helpmenu = Gtk::manage(new Gtk::Menu());
menuitem_help->set_submenu(*helpmenu);

//          A B O U T
// Append About to the Help menu
Gtk::MenuItem *menuitem_about = Gtk::manage(new Gtk::MenuItem("About", true));
menuitem_about->signal_activate().connect(sigc::mem_fun(*this,
    &Main_window::on_about_click));
helpmenu->append(*menuitem_about);
```

Main_window Constructor
(Page 4 of 7)

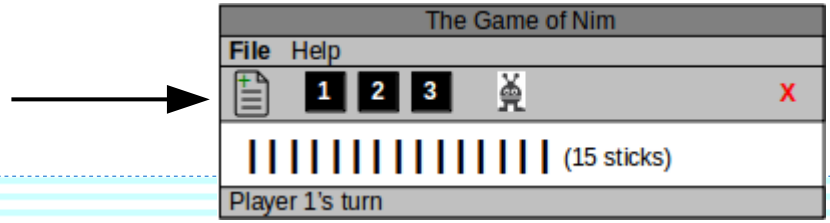
main_window.cpp




```
// ////////////
// T O O L B A R
// Add a toolbar to the vertical box, right below the menu
Gtk::Toolbar *toolbar = Gtk::manage(new Gtk::Toolbar);
vbox->add(*toolbar);

//     N E W   G A M E
// Add a "new game" button
Gtk::ToolButton *new_game_button = Gtk::manage
    (new Gtk::ToolButton(Gtk::Stock::NEW));    // Use the default NEW icon
new_game_button->set_tooltip_markup
    ("Create a new name, discarding any in progress");  // Add a tooltip
new_game_button->signal_clicked().connect(sigc::mem_fun
    (*this, &Main_window::on_new_game_click));   // Call this when clicked
toolbar->append(*new_game_button);
```

Main_window Constructor

main_window.cpp

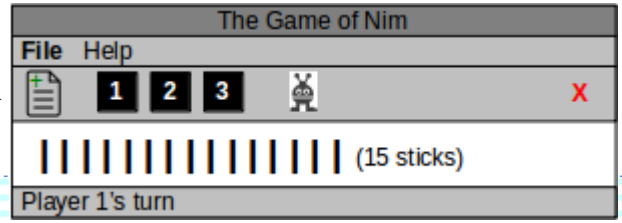



```
//     O N E   S T I C K
// Add a icon for taking one stick
button1_on_image = Gtk::manage(new Gtk::Image{"button1_on.png"});   // Load icon
button1_off_image = Gtk::manage(new Gtk::Image{"button1_off.png"}); //   images
button1 = Gtk::manage(new Gtk::ToolButton{*button1_on_image});  // Create button
button1->set_tooltip_markup("Select one stick");                //   with image
button1->signal_clicked().connect(sigc::mem_fun(*this,
    &Main_window::on_button1_click));
toolbar->append(*button1);

//     T W O   S T I C K S
// Add a icon for taking two sticks
button2_on_image = Gtk::manage(new Gtk::Image{"button2_on.png"});
button2_off_image = Gtk::manage(new Gtk::Image{"button2_off.png"});
button2 = Gtk::manage(new Gtk::ToolButton{*button2_on_image});
button2->set_tooltip_markup("Select two sticks");
button2->signal_clicked().connect(sigc::mem_fun(*this,
    &Main_window::on_button2_click));
toolbar->append(*button2);

//     T H R E E   S T I C K S  (Analogous to the above code)
```

Main_window Constructor
(Page 6 of 7)

Note: A ToggleButton retains state, alternating between “up” and “down” on each click. It’s current state is read using its get_active method.

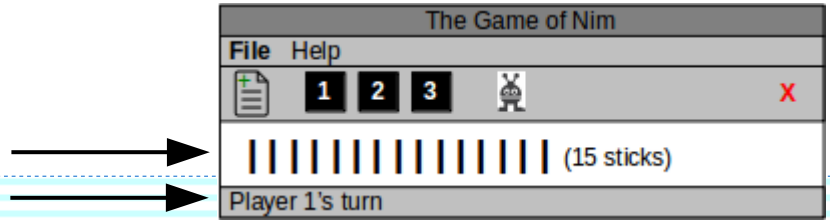
main_window.cpp




```
//     C O M P U T E R   P L A Y E R
// Add a toggle button to enable computer to play as Player 2
Gtk::Image *robot_image = Gtk::manage(new Gtk::Image{"freepik_robot.png"});
computer_player = Gtk::manage(new Gtk::ToggleToolButton{*robot_image});
computer_player->set_tooltip_markup("Enable for computer to be Player 2");
computer_player->signal_clicked().connect(sigc::mem_fun(*this,
    &Main_window::on_computer_player_click));
Gtk::SeparatorToolItem *sep1 = Gtk::manage(new Gtk::SeparatorToolItem{});
toolbar->append(*sep1);    // Add a little space between this and the stick buttons
toolbar->append(*computer_player);

//     Q U I T
// Add a icon for quitting
Gtk::ToolButton *quit_button = Gtk::manage(new Gtk::ToolButton{Gtk::Stock::QUIT});
quit_button->set_tooltip_markup("Exit game");
quit_button->signal_clicked().connect(sigc::mem_fun(*this,
    &Main_window::on_quit_click));
Gtk::SeparatorToolItem *sep = Gtk::manage(new Gtk::SeparatorToolItem{});
sep->set_expand(true);  // The expanding sep forces the Quit button to the right
toolbar->append(*sep);
toolbar->append(*quit_button);
```

Main_window Constructor
(Page 7 of 7)

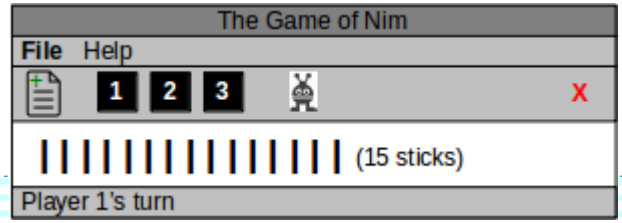
main_window.cpp




```
    // S T I C K S   D I S P L A Y
    // Provide a text entry box to show the remaining sticks
    sticks = Gtk::manage(new Gtk::Label{});
    sticks->set_hexpand(true);  // Fill to the right of the window
    sticks->set_vexpand(true);  // Fill any extra vertical space (as the window resizes)
    vbox->add(*sticks);

    // S T A T U S   B A R   D I S P L A Y
    // Provide a status bar for game messages
    msg = Gtk::manage(new Gtk::Label{});
    msg->set_hexpand(true);
    vbox->add(*msg);

    // Make the box and everything in it visible
    vbox->show_all();

    // Start a new game
    on_new_game_click();
}
```

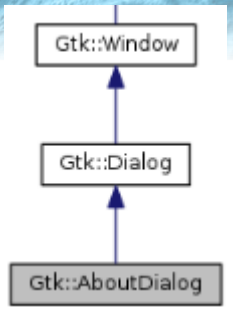
Main_window Callbacks
(Page 1 of 2)

main_window.cpp



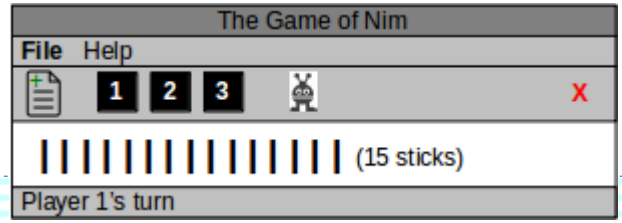

```
// ////////////////
// C A L L B A C K S
// ////////////////

void Main_window::on_button1_click() {
    nim->take_sticks(1);
    set_sticks();
}

void Main_window::on_button2_click() {
    nim->take_sticks(2);
    set_sticks();
}

void Main_window::on_button3_click() {
    nim->take_sticks(3);
    set_sticks();
}
```

Main_window Callbacks
(Page 2 of 2)



Hint: Use Gtk::AboutDialog instead of MessageDialog in real applications!

main_window.cpp



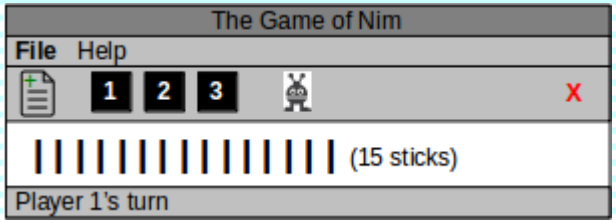

```
void Main_window::on_computer_player_click() {
    set_sticks();
}

void Main_window::on_new_game_click() {
    nim = new Nim();
    set_sticks();
}

void Main_window::on_quit_click() {
    hide();
}
```

The Pango formatting below produces this (after a little trial and error!)

```
void Main_window::on_about_click() {
    Glib::ustring s = "<span size='24000' weight='bold'>Nim</span>\n<span
size='large'>Copyright 2017 by George F. Rice</span>\n<span size='small'>Licensed under
Creative Commons Attribution 4.0 International\nRobot icon created by Freepik, used under
free attribution license</span>";
    Gtk::MessageDialog dlg(*this, s, true, Gtk::MESSAGE_INFO, Gtk::BUTTONS_OK, true);
    dlg.run();
}
```

Main_window Utilities
(Page 1 of 2)

main_window.cpp

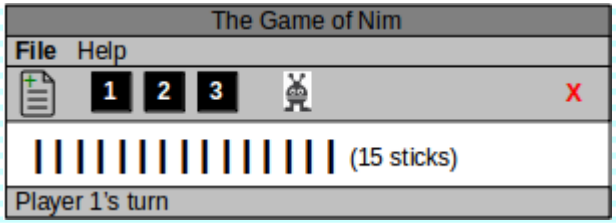



```
// //////////////////
// U T I L I T I E S
// //////////////////

void Main_window::set_sticks() {
    // s collects the status message
    Glib::ustring s = "";

    // If the robot is enabled and it's their turn, move the robot
    if (nim->sticks_left() > 0) {
        if (computer_player->get_active() && nim->current_player() == 2) {
           s += "Robot plays " + std::to_string(nim->optimal_move()) + ", ";
           nim->take_sticks(nim->optimal_move());
        }
    }

    // Report who's turn it is, or (if all of the sticks are gone) who won
    if (nim->sticks_left() > 0) {
        s += "Player " + std::to_string(nim->current_player()) + "'s turn";
    } else {
        s += "<span size='16000' weight='bold'>Player "
             +  std::to_string(3-nim->current_player())
             +  " wins!</span>";
    }
```

(Continued on next slide)

Main_window Utilities
(Page 2 of 2)

main_window.cpp




```
    // Display the collected status on the status bar
    msg->set_markup(s);

    // Update the visual display of sticks
    s = "<span size='24000' weight='bold'>";
    for(int i=0; i<nim->sticks_left(); ++i) s.append("| ");
    s.append("</span>  (" + std::to_string(nim->sticks_left()) + " sticks)");
    sticks->set_markup(s);

    // Set sensitivity of the human stick selectors so user can't make an illegal move
    button1->set_sensitive(nim->sticks_left() > 0);
    button2->set_sensitive(nim->sticks_left() > 1);
    button3->set_sensitive(nim->sticks_left() > 2);
}
```

Main

main.cpp

```
#include "main_window.h"
#include <gtkmm.h>

int main (int argc, char *argv[])
{
  auto app = Gtk::Application::create(argc, argv, "edu.uta.cse1325.nim");

  // Instance a Window
  Main_window win;

  // Set the window title
  win.set_title("Nim");

  //Show the window and returns when it is closed or hidden
  return app->run(win);
}
```

Nim Makefile

Makefile

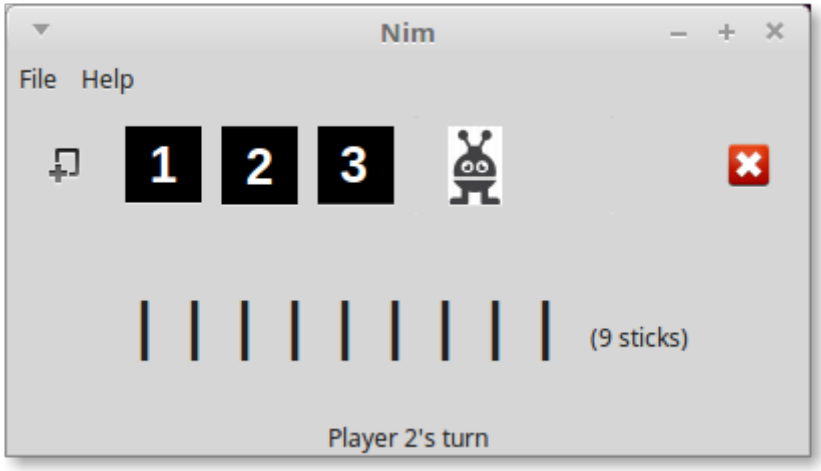
```
CXXFLAGS = --std=c++14
GTKFLAGS = `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`

all: main_window

debug: CXXFLAGS += -g
debug: main_window

main_window: main.o main_window.o nim.o
	$(CXX) $(CXXFLAGS) -o nim main.o main_window.o nim.o $(GTKFLAGS)

main.o: main.cpp main_window.h
	$(CXX) $(CXXFLAGS) -c main.cpp $(GTKFLAGS)

main_window.o: main_window.cpp main_window.h
	$(CXX) $(CXXFLAGS) -c main_window.cpp $(GTKFLAGS)

nim.o: nim.cpp nim.h
	$(CXX) $(CXXFLAGS) -c nim.cpp

clean:
	-rm -f *.o *.gch *~ nim
```

git Log and Make

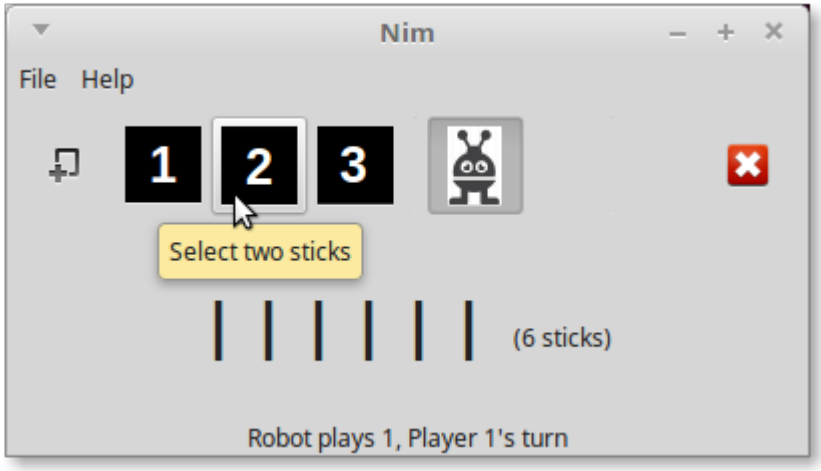
```
ricegf@pluto:~/dev/cpp/201708/gtkmm3/19_nim$ git log --oneline
bb59c0e Fix a few bugs
3ab31a3 Improve code for classroom use
ce73ec6 Refactor
18afe8a Robot player works
c16093e Runs smoothly for 2 players
3a84b82 Runs but buggy
a4df26c Add status line & large sticks
fbdde35 Force Quit button to the right
5f743b9 Add toolbar
2802dc8 First draft of adapted UI
2908203 First draft of Nim class
```

```
ricegf@pluto:~/dev/cpp/201801/14/nim$ make clean
rm -f *.o *.gch *~ nim
ricegf@pluto:~/dev/cpp/201801/14/nim$ make
g++ --std=c++14 -c main.cpp `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
g++ --std=c++14 -c main_window.cpp `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
g++ --std=c++14 -c nim.cpp
g++ --std=c++14 -o nim main.o main_window.o nim.o `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
ricegf@pluto:~/dev/cpp/201801/14/nim$
```

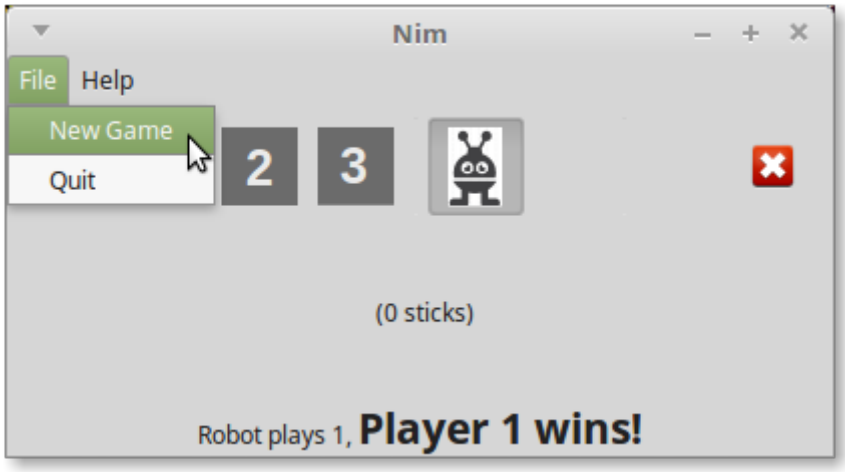
Interactively Testing Nim



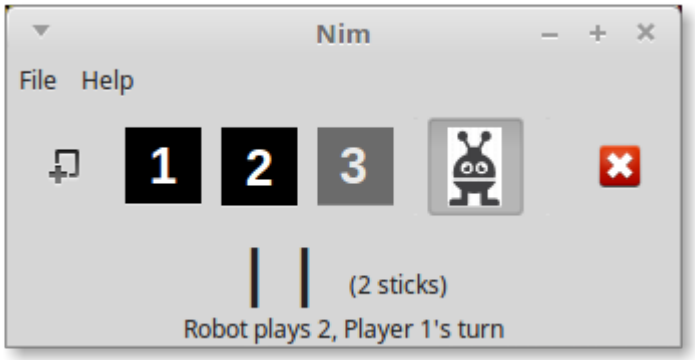
Two-player mode



One-player mode



Detecting a win



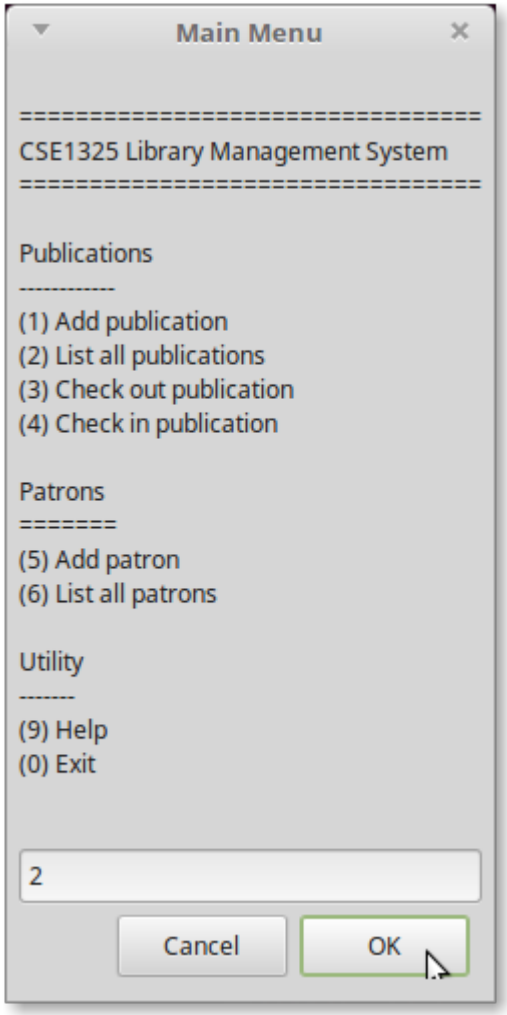
Resizing the window

Quick Review

- The _____ pattern notifies dependent objects when the observed event occurs.
- The _____ pattern creates new objects without exposing the creation logic to the client.
- The “new” operator allocates memory from the _____. To ensure it is returned (and avoid memory leaks), we use the ~ operator to define a _____.
- The _____ factory handles most of the routine chores associated with creating a gtkmm GUI application.
- _____ marks a Gtk widget for automatic deletion when the Gtk container referencing it is deleted.

Next Class

- Review chapter 13 - 15 in Stroustrop
  - Notice he is using a façade!
  - Do the drills
- We will discuss class library design
  - Plotting
  - Friendship
  - Lambdas
  - The Adapter Pattern
  - How to Manage Bugs
  - Generalization and Specialization



Homework #6 Questions?

Sprint 2 of 3

- Rework your LMS user interface with **dialogs**
  - NO console I/O at all – dialogs ONLY!
  - Rich text at the bonus level
  - A true custom dialog at the extreme bonus level
- As always, use git for version management
- Manage all 3 sprints via the Scrum spreadsheet
- Details are on Blackboard
- **Due March 8 at 8 am**