
gtkmm

gtkmm

gtkmm is a C++ wrapper for GTK+ (http://www.gtk.org/), a library used to create graphical user interfaces.

It is licensed using the LGPL license, so you can develop open software, free software, or even commercial non-free software using gtkmm without purchasing licenses.

gtkmm was originally named gtk-- because GTK+ already has a + in the name. However, as -- is not easily indexed by search engines, the package generally went by the name gtkmm and that’s what stuck.

gtkmm

gtkmm uses an event-driven programming model.

When the user is doing nothing, gtkmm sits in the main loop and waits for input. If the user performs some action - say, a mouse click - then the main loop “wakes up” and delivers an event to gtkmm.

When widgets (main user interface elements) receive an event, they frequently emit one or more signals.

Signals notify your program that “something interesting happened” by invoking functions you’ve connected to the signal.

Such functions are commonly known as callbacks.

When your callbacks are invoked, you would typically take some action - for example, when an Open button is clicked you might display a file chooser dialog.

After a callback finishes, gtkmm will return to the main loop and await more user input.

gtkmm

You must include the gtkmm header in your program

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

```
student@cse1325:/media/sf_VM/GTKMM$ make
g++ -c -g -std=c++11 WindowDemo.cpp -o WindowDemo.o `/usr/bin/pkg-config
gtkmm-3.0 --cflags --libs`
WindowDemo.cpp: In function ‘int main(int, char**)’:
WindowDemo.cpp:5:2: error: ‘Gtk’ has not been declared
  Gtk::Main kit(argc, argv);
  ^
```

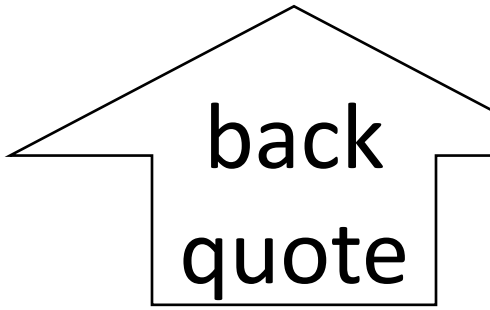
gtkmm

makefile changes

```
SRC1 = gtkmm.cpp
OBJ1 = $(SRC1:.cpp=.o)
EXE = $(SRC1:.cpp=.e)

GTKFLAGS = `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
CFLAGS = -g -std=c++11

all : $(EXE)

$(EXE): $(OBJ1)
	g++ $(CFLAGS) $(OBJ1) -o $(EXE) $(GTKFLAGS)

$(OBJ1) : $(SRC1)
	g++ -c $(CFLAGS) $(SRC1) -o $(OBJ1) $(GTKFLAGS)
```

gtkmm

```
GTKFLAGS = `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
```



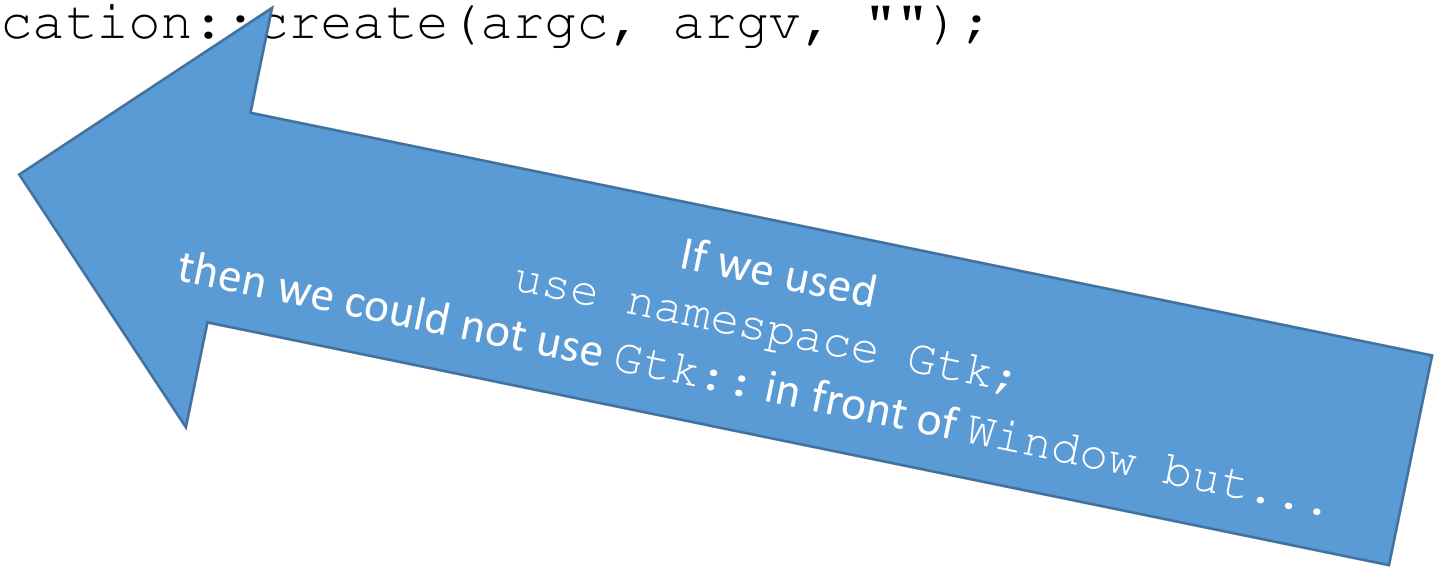
Note that you must surround the `pkg-config` invocation with backquotes.

Backquotes cause the shell to execute the command inside them, and to use the command's output as part of the command line.

gtkmm

Create an empty 200x200 pixel window

```
#include <gtkmm.h>

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

	Gtk::Window MyWindow;

	app->run(MyWindow);

	return 0;
}
```

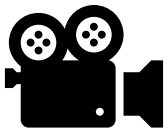


WindowDemo.cpp

gtkmm

```
student@cse1325:/media/sf_VM/GTKMM$ make

g++ -c -g -std=c++11 WindowDemo.cpp -o
WindowDemo.o `/usr/bin/pkg-config gtkmm-3.0 --
cflags --libs`

g++ -g -std=c++11 WindowDemo.o -o WindowDemo.e
`/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
```

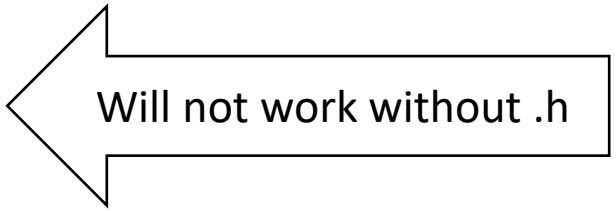

student@cse1325: /media/sf_VM/GTKMM
File Edit Tabs Help
student@cse1325:/media/sf_VM/GTKMM$ ./WindowDemo.e
student@cse1325: /me...
Software Updater
18:26



emo.cpp

gtkmm

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



All gtkmm programs must include certain gtkmm headers; `gtkmm.h` includes the entire gtkmm kit.

This is usually not a good idea, because it includes a megabyte or so of headers, but for simple programs, it suffices. We will use this for now.

You will notice how long the compile takes.

gtkmm

Create an empty 200x200 pixel window

```
#include <gtkmm.h>

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

      Gtk::Window MyWindow;

      app->run(MyWindow);

      return 0;
}
```

WindowDemo.cpp

gtkmm

```
auto app = Gtk::Application::create(argc, argv, "");
```

Initializes gtkmm. Takes in `argc` and `argv`.

This is needed in all gtkmm applications. The constructor for this object initializes gtkmm and checks the arguments passed to your application on the command line.

We can leave the application id empty but, if we do, then some features (like application uniqueness) will be disabled.

```
auto app = Gtk::Application::create(argc, argv, "app.WindowDemo");
```

```
type = class Glib::RefPtr<Gtk::Application> [with T_CppObject = Gtk::Application] {
  private:
    Gtk::Application *pCppObject_;

  public:
    RefPtr(void);
    RefPtr(Gtk::Application *);
    RefPtr(const Glib::RefPtr<Gtk::Application> &);
    RefPtr(Glib::RefPtr<Gtk::Application> &&);
    ~RefPtr();
    void swap(Glib::RefPtr<Gtk::Application> &);
    Glib::RefPtr<Gtk::Application> & operator=(const Glib::RefPtr<Gtk::Application> &);
    Glib::RefPtr<Gtk::Application> & operator=(Glib::RefPtr<Gtk::Application> &&);
    bool operator==(const Glib::RefPtr<Gtk::Application> &) const;
    bool operator!=(const Glib::RefPtr<Gtk::Application> &) const;
    Gtk::Application * operator->(void) const;
    Gtk::Application * get(void) const;
    operator bool(void) const;
    void clear(void);
    void reset(void);
    Gtk::Application * release(void);
    bool operator<(const Glib::RefPtr<Gtk::Application> &) const;
    bool operator<=(const Glib::RefPtr<Gtk::Application> &) const;
    bool operator>(const Glib::RefPtr<Gtk::Application> &) const;
    bool operator>=(const Glib::RefPtr<Gtk::Application> &) const;
}
```

Gtkmm::Application

`GtkApplication` is a class that handles many important aspects of a GTK+ application in a convenient fashion, without enforcing a one-size-fits-all application model.

Currently, `GtkApplication` handles GTK+ initialization, application uniqueness, session management, provides some basic scriptability and desktop shell integration by exporting actions and menus and manages a list of toplevel windows whose life-cycle is automatically tied to the life-cycle of your application.

Gtkmm::Application

`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)

Gtkmm::Application

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

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

      app->run(MyWindow);
```

We simplified

```
Glib::RefPtr<Gtk::Application> app =
Gtk::Application::create(argc, argv, "edu.uta.cse1325.example");
```

by using `auto`

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

```
Glib::RefPtr<Gtk::Application> app =
Gtk::Application::create(argc, argv, "edu.uta.cse1325.example");
```

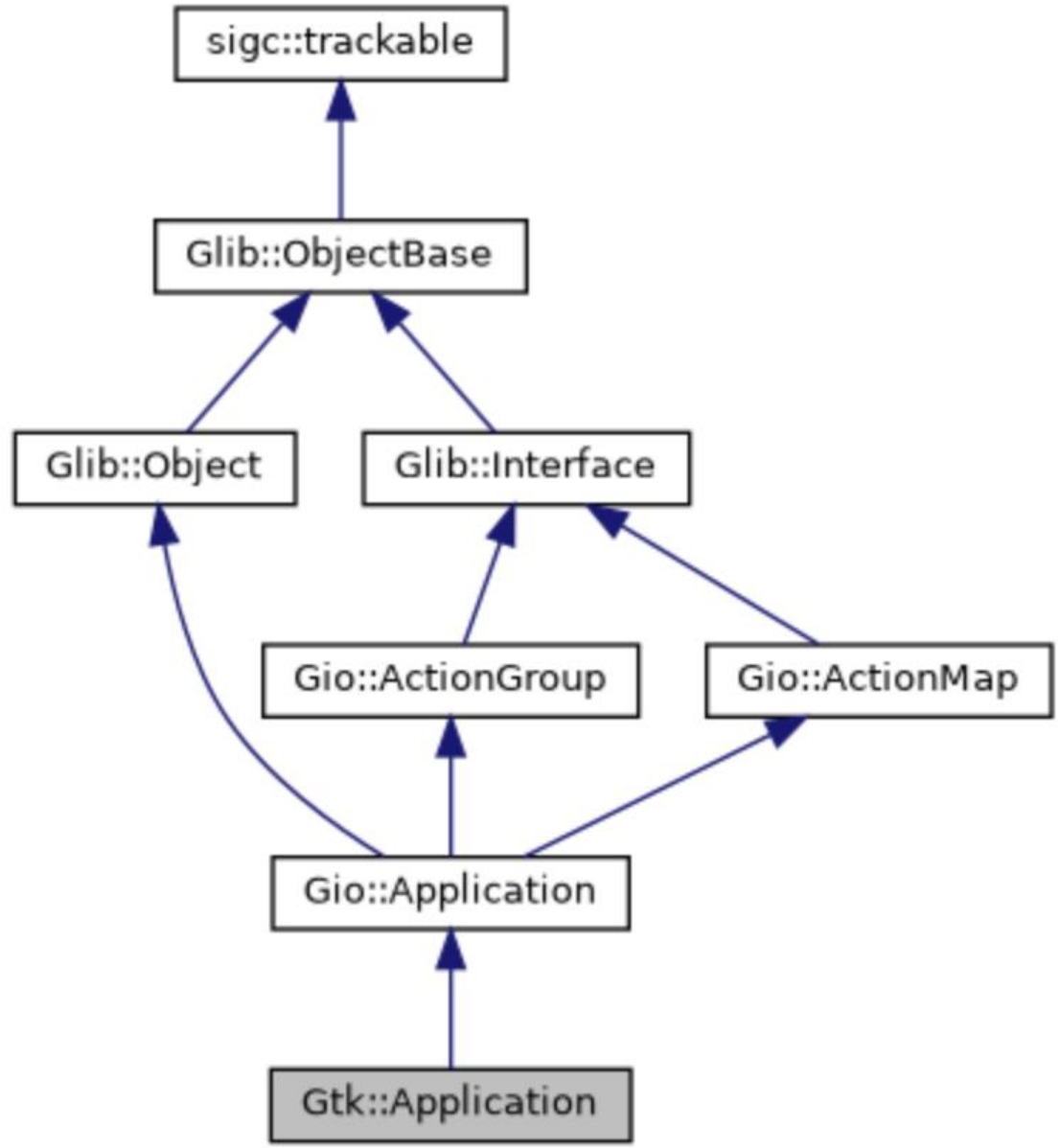
```
auto app = Gtk::Application::create(argc, argv, "edu.uta.cse1325.example");
```

First we instantiate an object stored in a `RefPtr` smartpointer called `app`. This is of type `Gtk::Application`. Every gtkmm program must have one of these.

We pass our command-line arguments to its `create()` method.

It takes the arguments it wants, and leaves you the rest.

Inheritance diagram for Gtk::Application:



gtkmm

Create an empty 200x200 pixel window

```
#include <gtkmm.h>

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

	Gtk::Window MyWindow;

	app->run(MyWindow);

	return 0;
}
```

WindowDemo.cpp

gtkmm

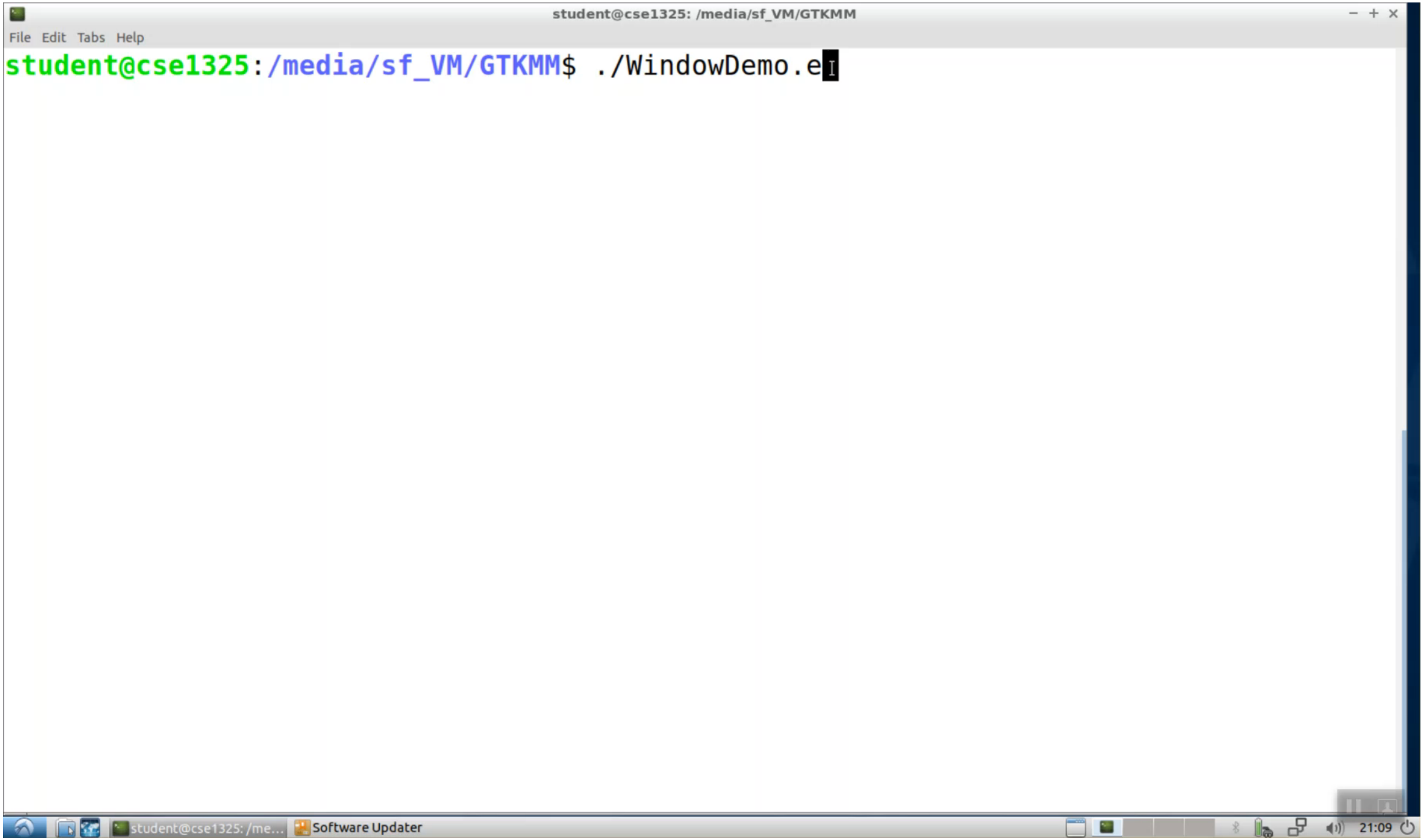
```
Gtk::Window MyWindow;

app->run(MyWindow);
```

This creates an object named `MyWindow` of class `Gtk::Window`.

`app->run` tells `app` to call the inherited member function `run` and passes it the object `MyWindow`.

`WindowDemo.cpp`

```
#include <gtkmm.h>

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

  Gtk::Window MyWindow;

  MyWindow.set_default_size(400, 200);
  MyWindow.set_title("This is MY window!!");
  MyWindow.set_position(Gtk::WIN_POS_CENTER);

  app->run(MyWindow);

  return 0;
}
```


student@cse1325: /media/sf_VM/GTKMM
File Edit Tabs Help
student@cse1325:/media/sf_VM/GTKMM$ ./WindowDemo.e
student@cse1325: /me...
Software Updater
21:09

gtkmm
Widgets

gtkmm applications consist of windows containing widgets, such as buttons and text boxes.

In some other systems, widgets are called "controls".

For each widget in your application’s windows, there is a C++ object in your application’s code.

So you just need to call a method of the widget’s class to affect the visible widget.

Packing

gtkmm windows seem "elastic" - they can usually be stretched in many different ways.

This is due to the **widget packing system**.

Many GUI toolkits require you to precisely place widgets in a window using absolute positioning using a visual editor. This leads to several problems…

Packing

The widgets don’t rearrange themselves when the window is resized. Some widgets are hidden when the window is made smaller and lots of useless space appears when the window is made larger.

It’s impossible to predict the amount of space necessary for text after it has been translated to other languages or displayed in a different font.

On Unix, it is also impossible to anticipate the effects of every theme and window manager.

Changing the layout of a window "on the fly", to make some extra widgets appear, for instance, is complex. It requires tedious recalculation of every widget’s position.

Packing

gtkmm uses the **packing** system to solve these problems.

Rather than specifying the position and size of each widget in the window, you can arrange your widgets in rows, columns, and/or tables.

gtkmm can size your window automatically, based on the sizes of the widgets it contains. And the sizes of the widgets are, in turn, determined by the amount of text they contain, or the minimum and maximum sizes that you specify, and/or how you have requested that the available space should be shared between sets of widgets.

You can perfect your layout by specifying padding distance and centering values for each of your widgets.

gtkmm then uses all this information to resize and reposition everything sensibly and smoothly when the user manipulates the window.

Packing

gtkmm arranges widgets hierarchically, using *containers*.

A Container widget contains other widgets.

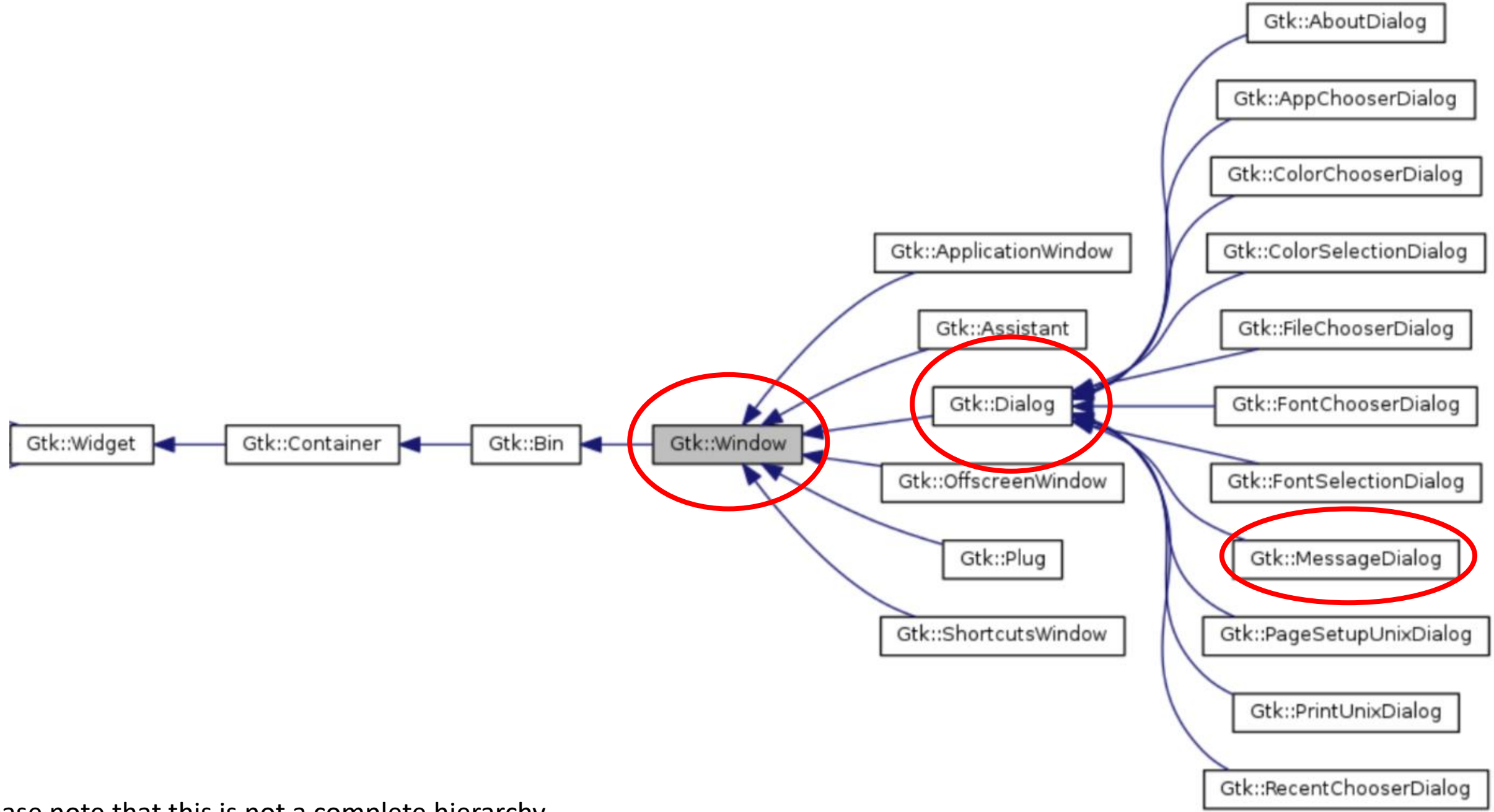
Most gtkmm widgets are containers.

Windows and Buttons are all container widgets.

There are two flavors of containers:

- single-child containers which are all descendants of `Gtk::Bin`
- multiple-child containers which are descendants of `Gtk::Container`

Most widgets in gtkmm are descendants of `Gtk::Bin`, including `Gtk::Window`.



Please note that this is not a complete hierarchy

Packing

`Gtk::Window` can contain at most one widget.

So how can we use a window for anything useful?

By placing a multiple-child container in the window.

The most useful container widgets are `Gtk::Box` and `Gtk::Table`.

`Gtk::Box` can arrange child widgets vertically (GTK_ORIENTATION_VERTICAL)
can arrange child widgets horizontally (GTK_ORIENTATION_HORIZONTAL)

`Gtk::Table` arranges its widgets in a grid.

Dialog

Dialogs are used as secondary windows to provide specific information or to ask questions.

`Gtk::Dialog` windows contain a few pre-packed widgets to ensure consistency and a `run()` method which blocks until the user dismisses the dialog.

There are several derived Dialog classes which you might find useful.

`Gtk::MessageDialog` is used for most simple notifications.

But at other times you might need to derive your own dialog class to provide more complex functionality.

Dialog

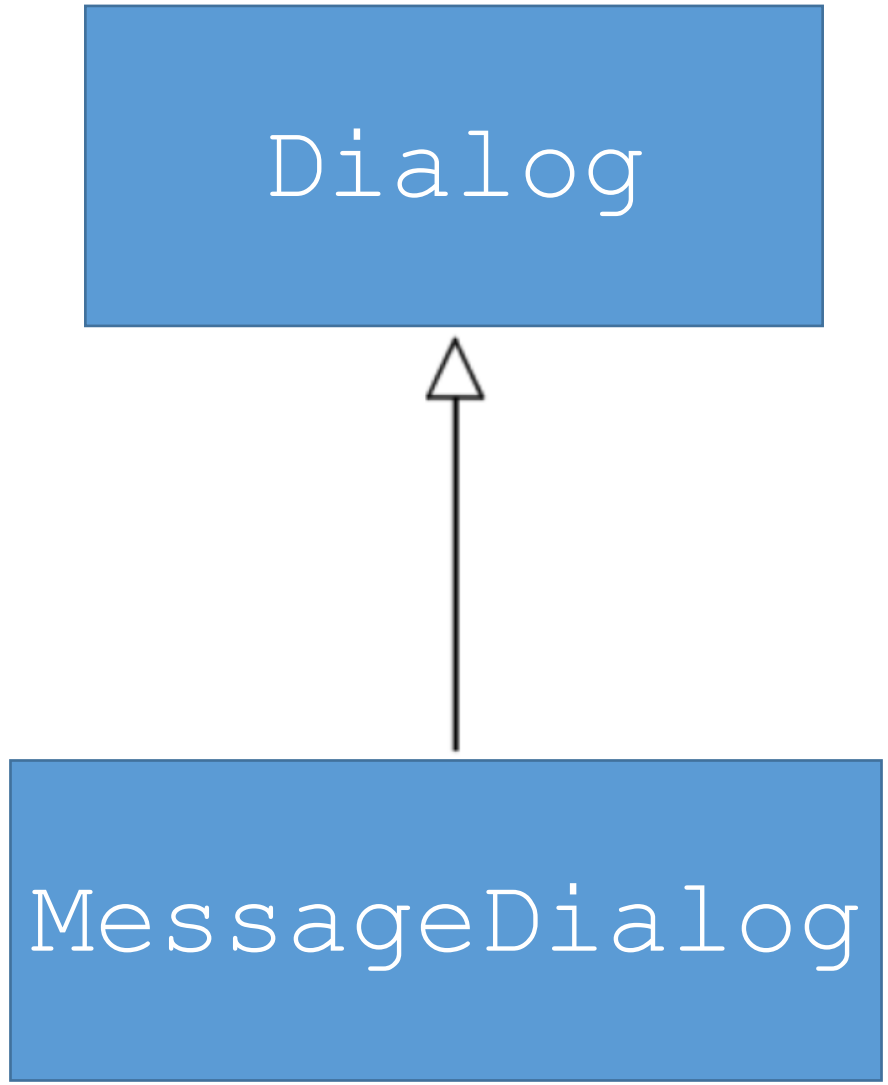
Base class

`Dialog`

Derived class

`MessageDialog`

`Dialog` is derived from `Gtk::Window`



MessageDialog

`MessageDialog` is a convenience class used to create simple, standard message dialogs with a message, an icon and buttons for user response.

You can specify the type of message and the text in the constructor, as well as specifying standard buttons via the `Gtk::ButtonsType` enum.

MessageDialog

```
#include <gtkmm.h>

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

	Gtk::MessageDialog HWdialog{"Hello World"};

	HWdialog.run();
}
```


student@cse1325: /media/sf_VM/GTKMM
File Edit Tabs Help
student@cse1325:/media/sf_VM/GTKMM$ ./HelloWorld.e
student@cse1325: /me...
[Software Updater]
18:43

Gtk Warning

```
Gtk-Message: 19:24:17.043: GtkDialog mapped without a
transient parent. This is discouraged.
```

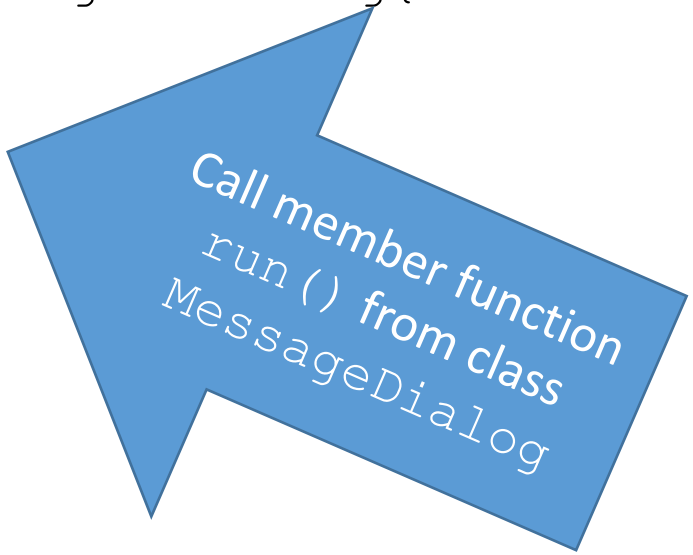
This is telling us that we displayed a dialog that did not have a main window as a root.

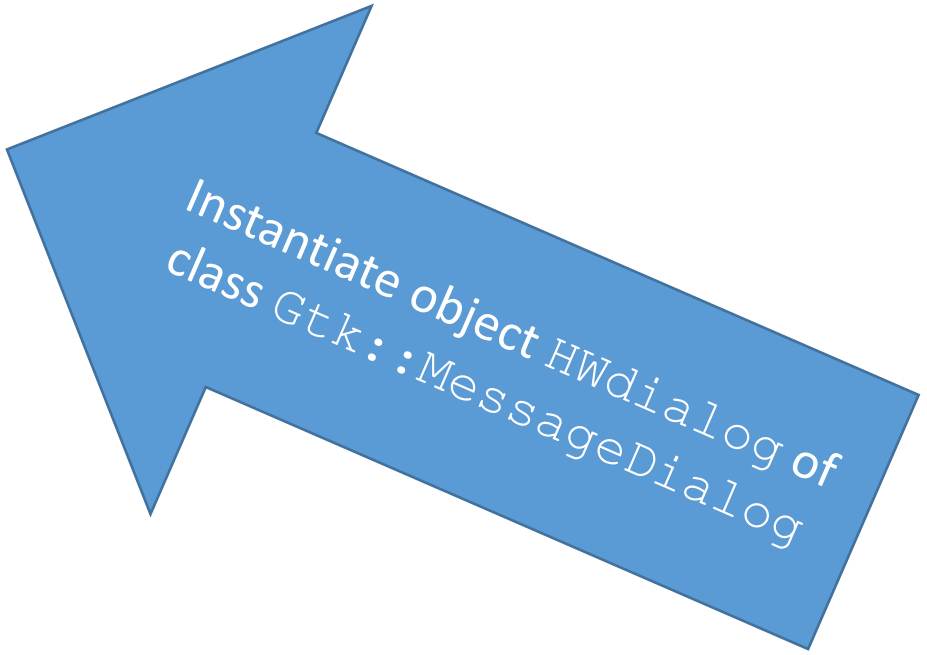
In small examples like this, we don't actually *have* a main window so that warning is inevitable and (happily) innocuous.

But once we have a main window, we will use it to avoid that warning.

MessageDialog

```
#include <gtkmm.h>

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

	Gtk::MessageDialog HWdialog{"Hello World"};

	HWdialog.run();
}
```





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

int main(int argc, char* argv[])
{
	std::ostringstream MyMessage;

	auto app = Gtk::Application::create(argc, argv, "app.hello.world");

	MyMessage << "Hello " << argv[1];

	Gtk::MessageDialog HWdialog{MyMessage.str()};

	HWdialog.run();
}
```


student@cse1325: /media/sf_VM/GTKMM
File Edit Tabs Help
student@cse1325:/media/sf_VM/GTKMM$ ./HelloWorld.e
student@cse1325: /me...
Software Updater
19:34

```
int main(int argc, char* argv[])
{
   std::ostringstream MyMessage;
   int i;

   auto app = Gtk::Application::create(argc, argv, "app.hello.world");

   MyMessage << "Hello ";

   for (i = 1; i < argc; i++)
   {
      MyMessage << argv[i] << " ";

      Gtk::MessageDialog HWdialog{MyMessage.str()};

      HWdialog.run();
   }
}
```

```
./HelloWorld.e How are you?
```


student@cse1325: /media/sf_VM/GTKMM
File Edit Tabs Help
student@cse1325:/media/sf_VM/GTKMM$ ./HelloWorld.e How are you?
student@cse1325: /me...
Software Updater
19:46

```
int main(int argc, char* argv[])
{
   std::ostringstream MyMessage;
   int i;

   auto app = Gtk::Application::create(argc, argv, "app.hello.world");

   MyMessage << "Hello ";

   for (i = 1; i < argc; i++)
   {
      MyMessage << argv[i] << " ";

      Gtk::MessageDialog *HWdialog = new Gtk::MessageDialog{MyMessage.str()};

      HWdialog->run();
   }
}
```

MessageDialog

```
HWdialog.set_secondary_text("this is a test", false);
```

```
set_secondary_text (
const Glib::ustring& text,
bool use_markup=false )
```

secondary text is positioned under the message

In effect, the message becomes the title and the secondary text the message

Set `use_markup` to true if the text should be processed for Pango tags

Pango Markup

Pango works similar to HTML

Accepts attributes such as

font, font_size, font_style, font_weight, etc.
fgcolor and alpha, bgcolor and bgalpha
underline, strikethrough, and their color variants

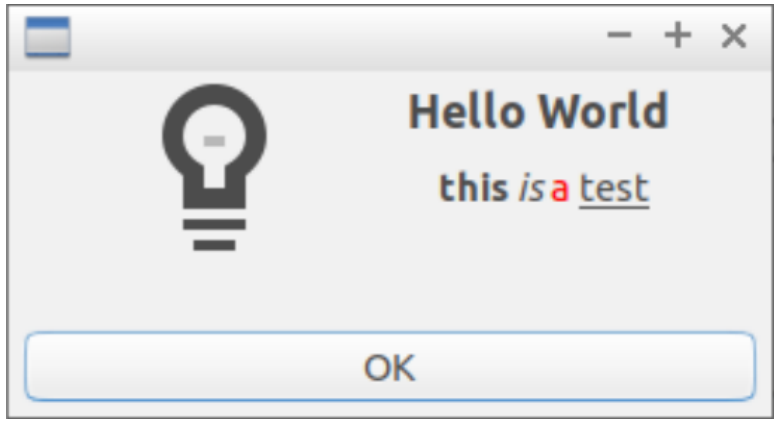
Uses tags

<b>, <big>, <i>, <s>, <sub>, <sup>, <small>, <tt> (monospace), and <u>

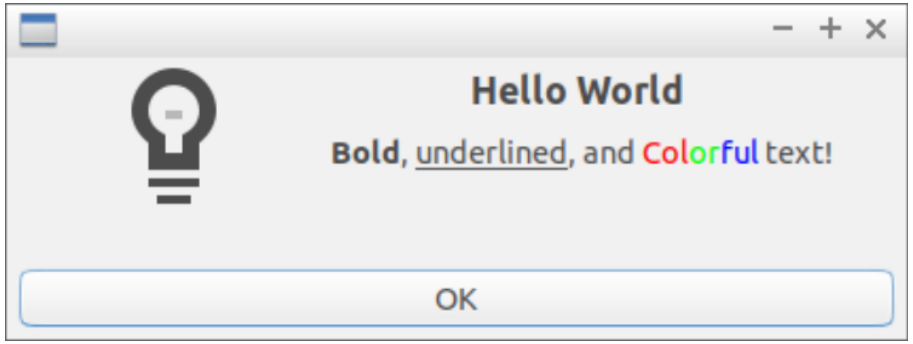
The Pango reference manual is at https://developer.gnome.org/pango/stable/

Pango Markup

```
HWDialog.set_secondary_text("<b>this</b> <i>is</i> <span
fgcolor='#ff0000'>a</span> <u>test</u>", true);
```




```
HWdialog.set_secondary_text("<b>Bold</b>, <u>underlined</u>, and <span
fgcolor='#ff0000'>Col</span>""<span fgcolor='#00ff00'>or</span><span
fgcolor='#0000ff'>ful</span>"" text!", true);
```



`MessageDialog` Constructor

```
Gtk::MessageDialog{"Hello World"}
```

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

```
const Glib::ustring& message
```

`ustring` is gtkmm's Unicode version of `std::string` with full conversions to and from `std::string`; therefore, we can just use "`string`")

MessageDialog Constructor

```
MessageDialog{"Hello World"}
```

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

`bool` **`use_markup`**`=false`

`use_markup` determines whether **Pango** markup is interpreted from the message (more on Pango shortly)

MessageDialog

```
MessageDialog{"Hello World"}
```

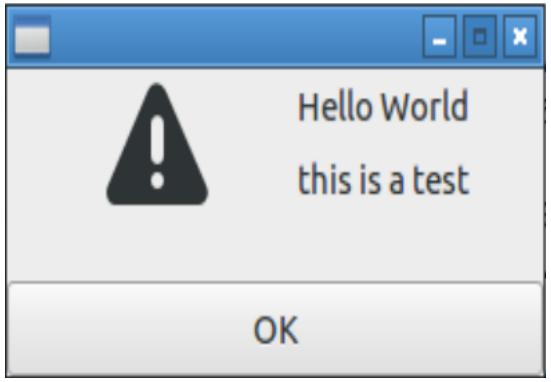
```
MessageDialog (const Glib::ustring& message,
               bool use_markup=false,
               MessageType type=Gtk::MESSAGE_INFO,
               ButtonsType buttons=BUTTONS_OK,
               bool modal=false)
```

```
MessageType type=Gtk::MESSAGE_INFO
```

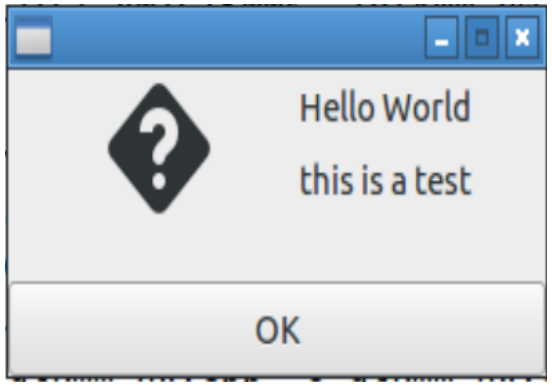
type determines the **icon** to be presented

MESSAGE_INFO, MESSAGE_WARNING, MESSAGE_QUESTION,
MESSAGE_ERROR, MESSAGE_OTHER

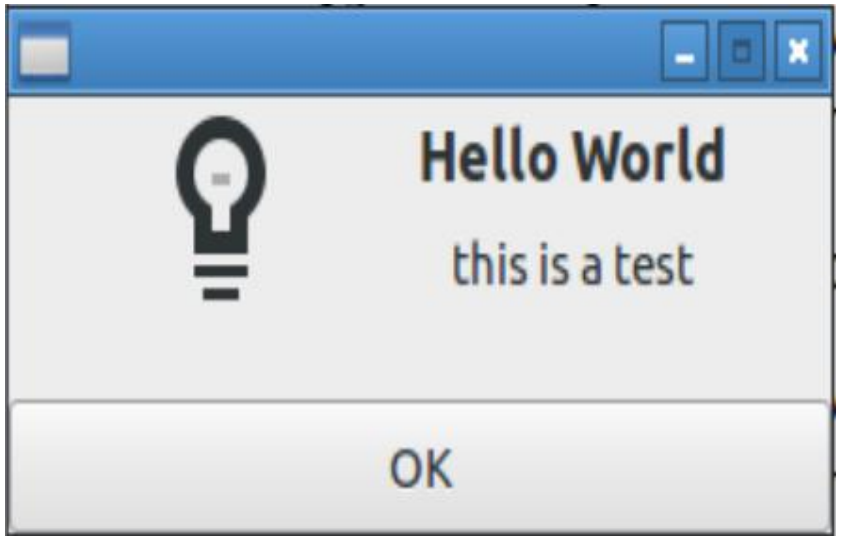
MESSAGE_WARNING



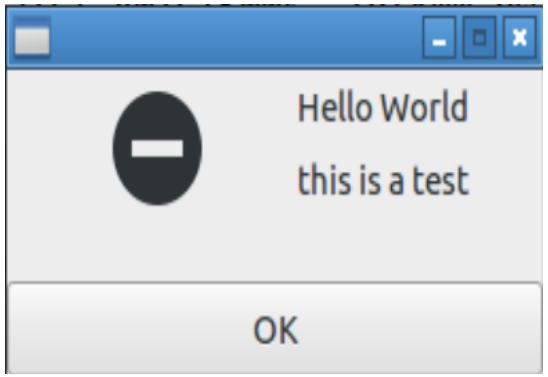
MESSAGE_QUESTION



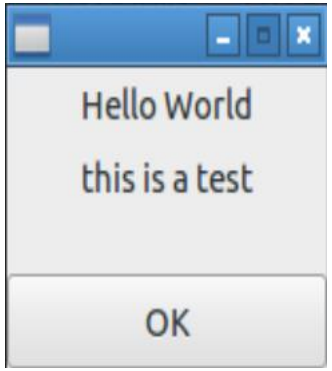
MESSAGE_INFO



MESSAGE_ERROR



MESSAGE_OTHER



MessageDialog

```
MessageDialog{"Hello World"}
```

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

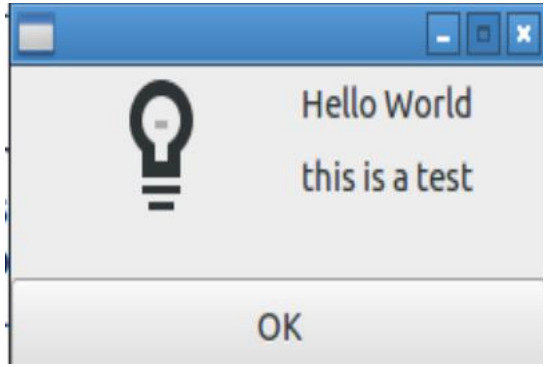
```
ButtonsType buttons=Gtk::BUTTONS_OK
```

buttons determines which buttons to display

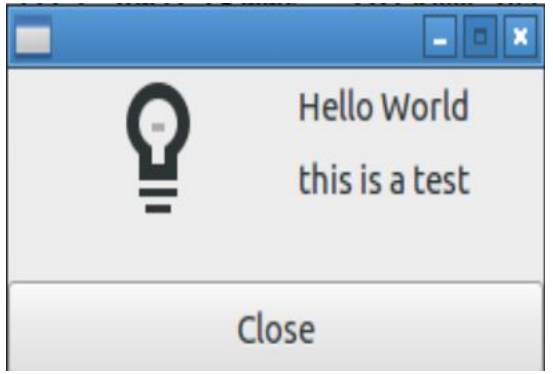
```
BUTTONS_NONE, BUTTONS_OK, BUTTONS_CLOSE,
BUTTONS_CANCEL, BUTTONS_YES_NO, BUTTONS_OK_CANCEL
```

BUTTONS_NONE

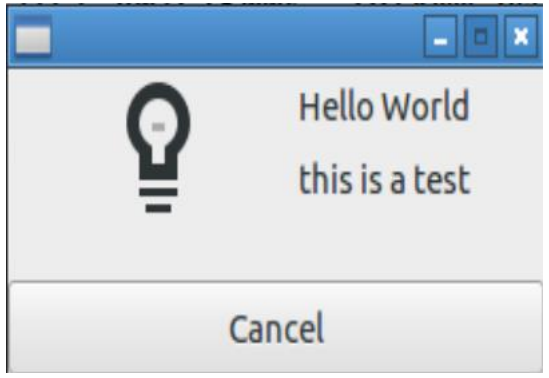
BUTTONS_OK



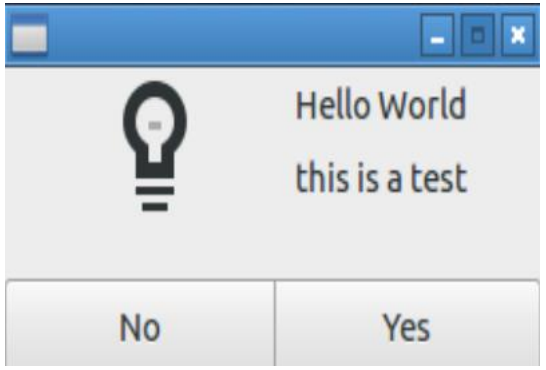
BUTTONS_CLOSE



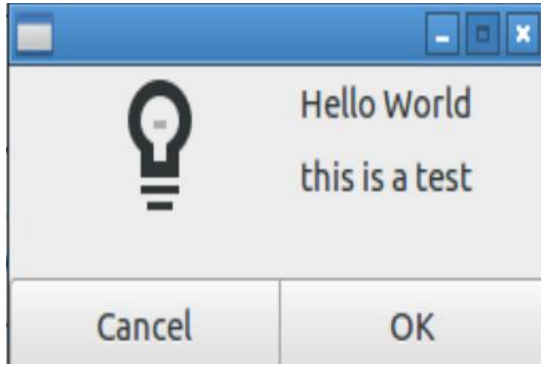
BUTTONS_CANCEL



BUTTONS_YES_NO



BUTTONS_OK_CANCEL



MessageDialog

```
MessageDialog{"Hello World"}
```

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

modal is true if no other dialogs may take focus while this dialog is open, false otherwise

This should almost ALWAYS be false!

MessageDialog

```
MessageDialog{"Hello World"}


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


Gtk::MessageDialog{"Hello World", true,
Gtk::MESSAGE_INFO, Gtk::BUTTONS_OK, false};
```

Dialog

Dialog boxes can be used to create popup windows.

Dialog boxes are a convenient way to prompt the user for a small amount of input

- to display a message
- ask a question
- anything else that does not require extensive effort on the user's part.

Dialog

gtkmm treats a dialog as a window split horizontally.

The top section is a vertical `Gtk::Box` and is where widgets such as a `Gtk::Label` or a `Gtk::Entry` should be packed.

The bottom area is known as the `action_area`. This is generally used for packing buttons into the dialog which may perform functions such as cancel, ok, or apply.

```
#include <gtkmm.h>

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

    Gtk::Dialog MyDialog;

    MyDialog.set_title("Question");
    MyDialog.set_default_size(600,600);

    MyDialog.run();

    MyDialog.close();

    return 0;
}
```


student@cse1325: /media/sf_VM/GTKMM
File Edit Tabs Help
student@cse1325:/media/sf_VM/GTKMM$ ./DialogDemo1.e
student@cse1325: /me...
Software Updater
21:52

Dialog

To add a Button to the bottom of a Dialog, use the `add_button()` method.

When adding buttons using `add_button()`, clicking the button will emit a `signal_response()` with the "response id" you specified.

You are should use the `Gtk::ResponseType` enum for your response id.

`run()` returns that response id as an `int`.

Dialog

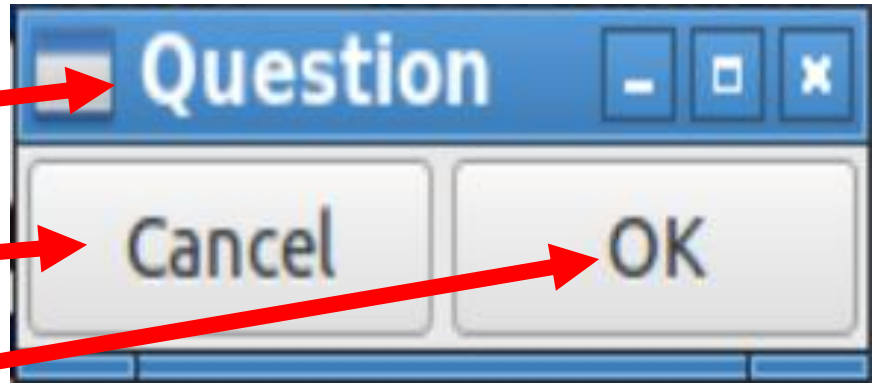
enum Gtk::ResponseType

Predefined values for use as response ids in **Gtk::Dialog::add_button()**.

All predefined values are negative; GTK+ leaves values of 0 or greater for application-defined response ids.

Enumerator	
RESPONSE_NONE	Returned if an action widget has no response id, or if the dialog gets programmatically hidden or destroyed.
RESPONSE_REJECT	Generic response id, not used by GTK+ dialogs.
RESPONSE_ACCEPT	Generic response id, not used by GTK+ dialogs.
RESPONSE_DELETE_EVENT	Returned if the dialog is deleted.
RESPONSE_OK	Returned by OK buttons in GTK+ dialogs.
RESPONSE_CANCEL	Returned by Cancel buttons in GTK+ dialogs.
RESPONSE_CLOSE	Returned by Close buttons in GTK+ dialogs.
RESPONSE_YES	Returned by Yes buttons in GTK+ dialogs.
RESPONSE_NO	Returned by No buttons in GTK+ dialogs.
RESPONSE_APPLY	Returned by Apply buttons in GTK+ dialogs.
RESPONSE_HELP	Returned by Help buttons in GTK+ dialogs.

```
#include <gtkmm.h>

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

    Gtk::Dialog MyDialog;

    MyDialog.set_title("Question");
    MyDialog.set_default_size(225,100);

    MyDialog.add_button("Cancel", Gtk::RESPONSE_CANCEL);
    MyDialog.add_button("OK", Gtk::RESPONSE_OK);
    MyDialog.set_default_response(Gtk::RESPONSE_OK);

    int result = MyDialog.run();

    MyDialog.close();

    return 0;
}
```

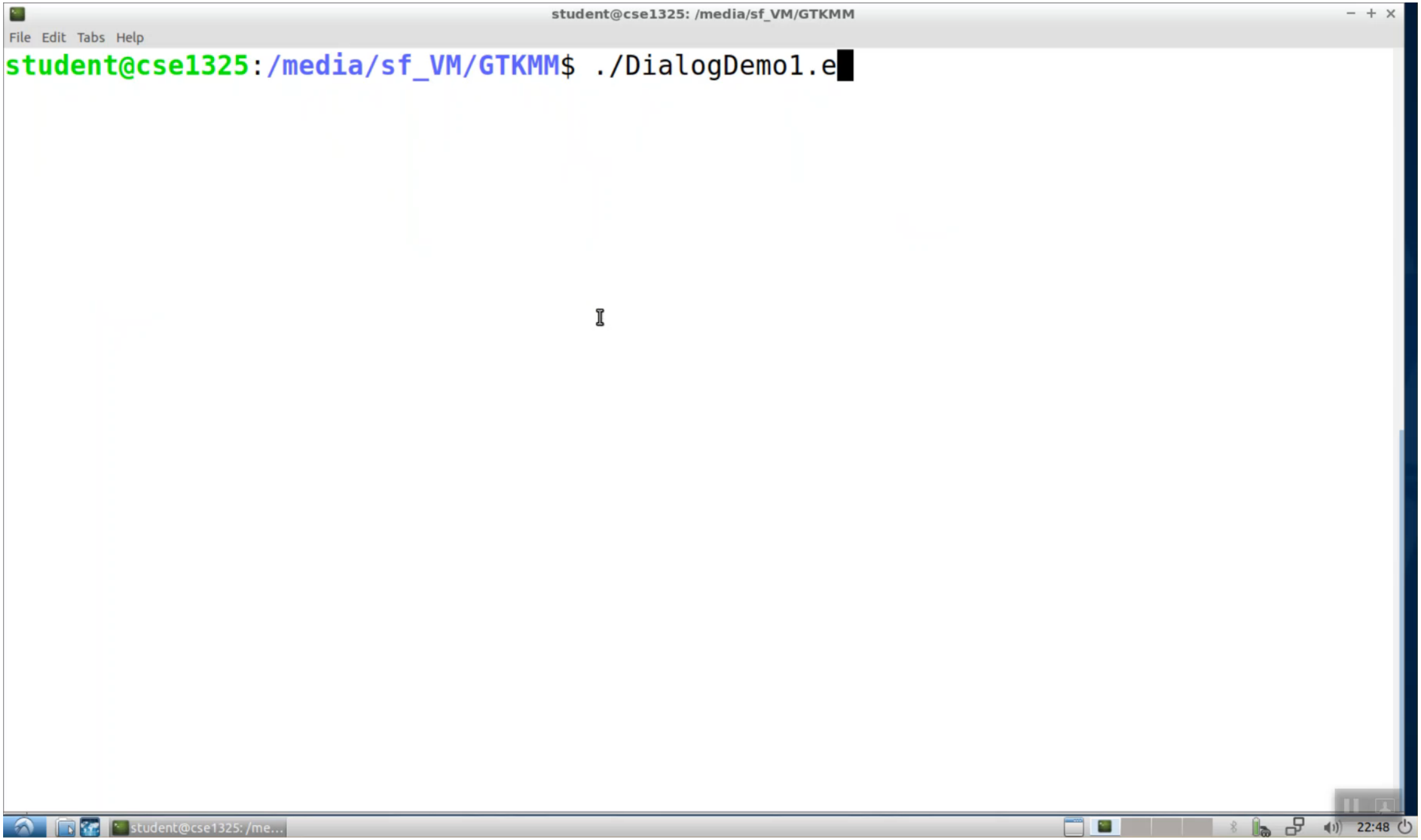


Runs the dialog and captures the return value of the button click.

```
auto app = Gtk::Application::create(argc, argv, "app.Dialog");

Gtk::Dialog MyDialog;

MyDialog.set_title("Question");
MyDialog.set_default_size(225,100);

MyDialog.add_button("Cancel", Gtk::RESPONSE_CANCEL);
MyDialog.add_button("OK", Gtk::RESPONSE_OK);
MyDialog.set_default_response(Gtk::RESPONSE_CANCEL);

int result = MyDialog.run();

MyDialog.close();
```

dialog1Demo.cpp

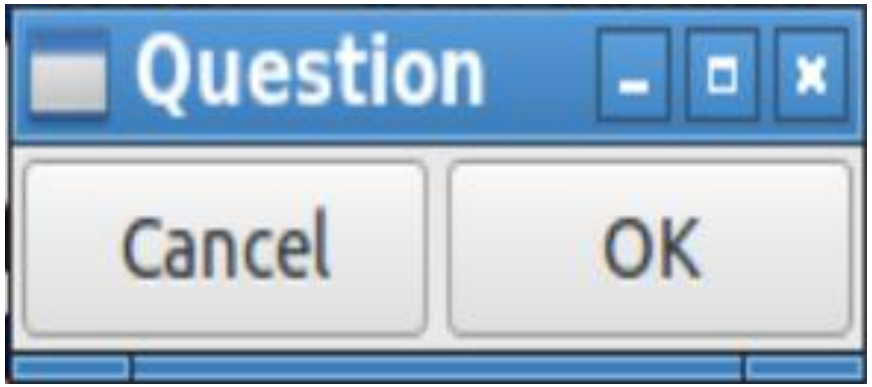
```
if (result == Gtk::RESPONSE_OK)
    std::cout << "User clicked OK or pressed ENTER on keyboard"
              << std::endl;

else if (result == Gtk::RESPONSE_CANCEL)
    std::cout << "User clicked Cancel" << std::endl;

else if (result == Gtk::RESPONSE_DELETE_EVENT)
    std::cout << "User closed dialog box" << std::endl;

else
    std::cout << "User did something - "
              << result << std::endl;
```


student@cse1325: /media/sf_VM/GTKMM
File Edit Tabs Help
student@cse1325:/media/sf_VM/GTKMM$ ./DialogDemo1.e
student@cse1325: /me...
22:48

Dialog

Dialog provides

- a message (title)
- a content area
- a vbox area
- a button (action) area



You can put any number of widgets into Content and Vbox because they are containers.

Buttons

Button boxes are a convenient way to quickly arrange a group of buttons.

They come in both horizontal (`Gtk::HButtonBox`) and vertical (`Gtk::VButtonBox`) flavors.

They are exactly alike, except in name and orientation.

`ButtonBoxes` help to make applications appear consistent because they use standard settings such as inter-button spacing and packing.

Buttons are added to a `ButtonBox` with the `add()` method.

Button boxes support several layout styles. The style can be retrieved and changed using `get_layout()` and `set_layout()`.

For Dialog boxes, we are just going to use the `add_button()` method which adds buttons to the bottom of the Dialog box.

```
add_button(const Glib::ustring& button_text, int response_id)
```

`button_text`

text on the button

`response_id`

unique integer returned when a button is clicked

`set_default_response(int response_id)`

sets the default button which is usually activated when the user just presses ENTER

```
MyDialog.add_button("Cancel", Gtk::RESPONSE_CANCEL);
MyDialog.add_button("OK", Gtk::RESPONSE_OK);
MyDialog.set_default_response(Gtk::RESPONSE_CANCEL);
```

```
auto app = Gtk::Application::create(argc, argv, "app.dialog2");

Gtk::Dialog MyDialog;

MyDialog.set_title("Question");

std::vector<std::string> buttons = {"Cancel", "OK", "Maybe", "Maybe Not?",
                                    "Who Cares"};

for (int i = 0; i < buttons.size(); i++)
  MyDialog.add_button(buttons[i], i);

MyDialog.set_default_response(4);

int result = MyDialog.run();

MyDialog.close();

std::cout << "User clicked \"" << buttons[result] << "\"" << std::endl;
```

```
add_button(const Glib::ustring&
button_text, int response_id)
```

dialog2Demo.cpp


student@cse1325: /media/sf_VM/GTKMM
File Edit Tabs Help
student@cse1325:/media/sf_VM/GTKMM$ ./dialog2Demo.e
student@cse1325: /me...
23:17

Packing

Most packing uses boxes which are invisible containers into which we can pack our widgets.

The `pack_start()` and `pack_end()` methods place widgets inside these containers.

When packing widgets into a horizontal box, the objects are inserted horizontally from left to right or right to left depending on whether `pack_start()` or `pack_end()` is used.

In a vertical box, widgets are packed from top to bottom or vice versa.

The `pack_start()` method will start at the top and work its way down in a vertical `Box`, or pack left to right in a horizontal `HBox`.

`pack_end()` will do the opposite - packing from bottom to top in a vertical `Box`, or right to left in a horizontal `Box`.

Using these methods allows us to right justify or left justify our widgets

Packing

Widgets are “pack”ed into a container.

For example, `Gtk::Dialog` can pack widgets into the content area and the vertical container box.

- `dialog->get_content_area()->pack_start(*widget);`

- `dialog->get_vbox()->pack_start(*widget)`

Label Widget

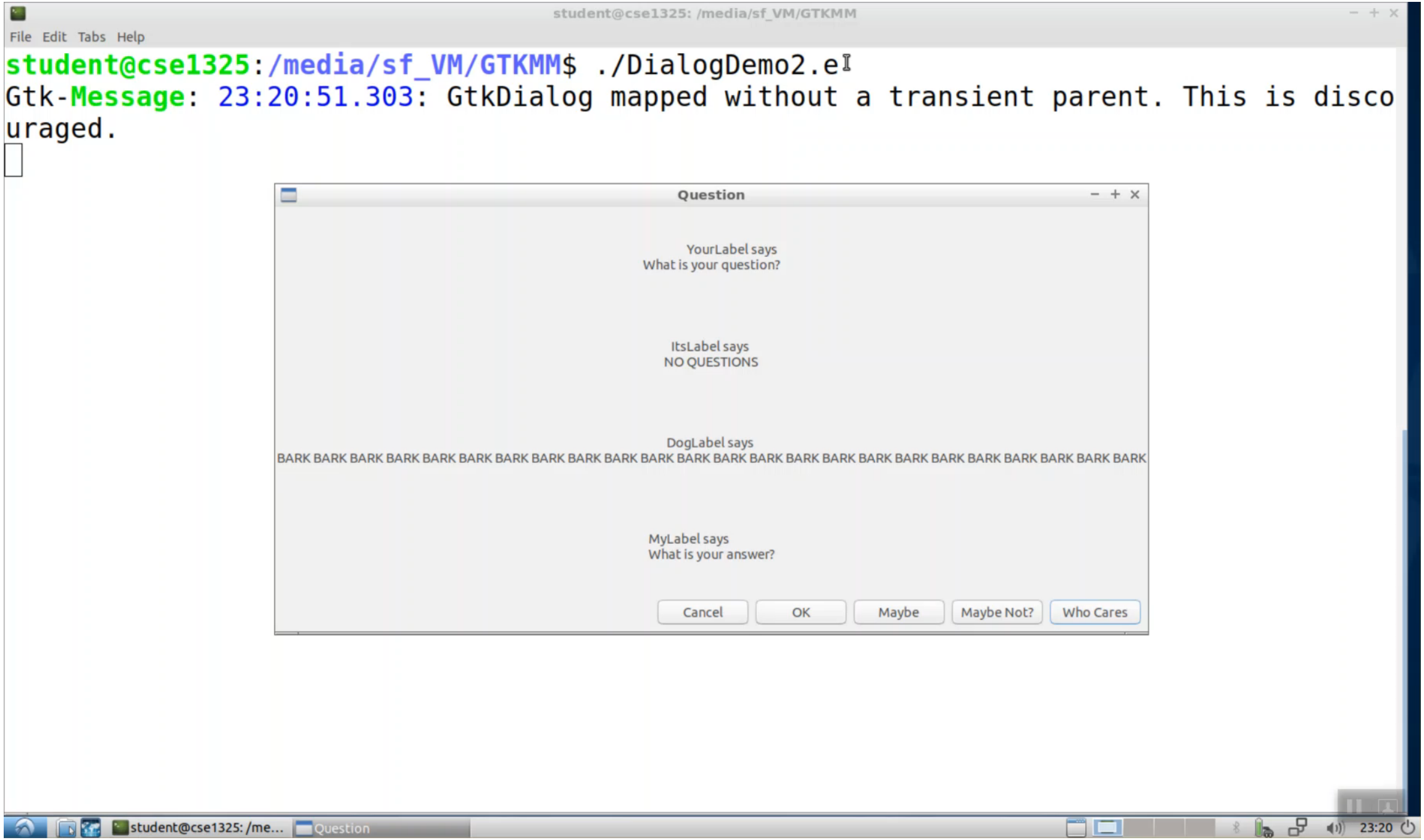
Labels are the main method of placing non-editable text in windows.

- You can specify the text in the constructor or with the `set_text()` method.
- The width of the label will be adjusted automatically. You can produce multi-line labels by putting line breaks ("\n") in the label string.
- The label text can be justified using the `set_justify()` method.
- The widget is also capable of word-wrapping - this can be activated with `set_line_wrap()`.

Label Widget

- Labels can be made selectable with `Gtk.Label.set_selectable`(). Selectable labels allow the user to copy the label contents to the clipboard. Only labels that contain useful-to-copy information — such as error messages — should be made selectable.

- Labels support some simple formatting - making some text bold, colored, or larger. This is done by providing a string to `Gtk.Label.set_markup()` using the Pango Markup syntax.


student@cse1325: /media/sf_VM/GTKMM
File Edit Tabs Help
student@cse1325:/media/sf_VM/GTKMM$ ./DialogDemo2.e
Gtk-Message: 23:20:51.303: GtkDialog mapped without a transient parent. This is disco
uraged.
Question
YourLabel says
What is your question?
ItsLabel says
NO QUESTIONS
DogLabel says
BARK BARK BARK BARK BARK BARK BARK BARK BARK BARK BARK BARK BARK BARK BARK BARK BARK BARK BARK BARK BARK BARK
MyLabel says
What is your answer?
Cancel
OK
Maybe
Maybe Not?
Who Cares
student@cse1325: /me...
Question
23:20

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

int main (int argc, char* argv[])
{
    auto app = Gtk::Application::create(argc, argv, "app.dialog2");

    Gtk::Dialog MyDialog;

    MyDialog.set_title("Question");

    std::vector<std::string> buttons = {"Cancel", "OK", "Maybe", "Maybe Not?", "Who Cares"};

    for (int i = 0; i < buttons.size(); i++)
        MyDialog.add_button(buttons[i], i);
    MyDialog.set_default_response(4);

    Gtk::Label MyLabel("\n\nWhat is your answer?\n\n");
    MyDialog.get_content_area()->pack_start(MyLabel);
    MyLabel.show();

    int result = MyDialog.run();

    MyDialog.close();

    std::cout << "Your answer is \"" << buttons[result] << "\"" << std::endl;

    return 0;
}
```


student@cse1325: /media/sf_VM/GTKMM
File Edit Tabs Help
student@cse1325:/media/sf_VM/GTKMM$ ./DialogDemo2.e
student@cse1325: /me...
22:09

```
Gtk::Label MyLabel("\n\nWhat is your answer?\n\n");
MyDialog.get_content_area()->pack_start(MyLabel);
MyLabel.show();
```

dialog3Demo.cpp

```
Gtk::Label MyLabel("\n\nWhat is your answer?\n\n");
```

Constructs MyLabel with a string.

```
MyDialog.get_content_area()->pack_start(MyLabel);
```

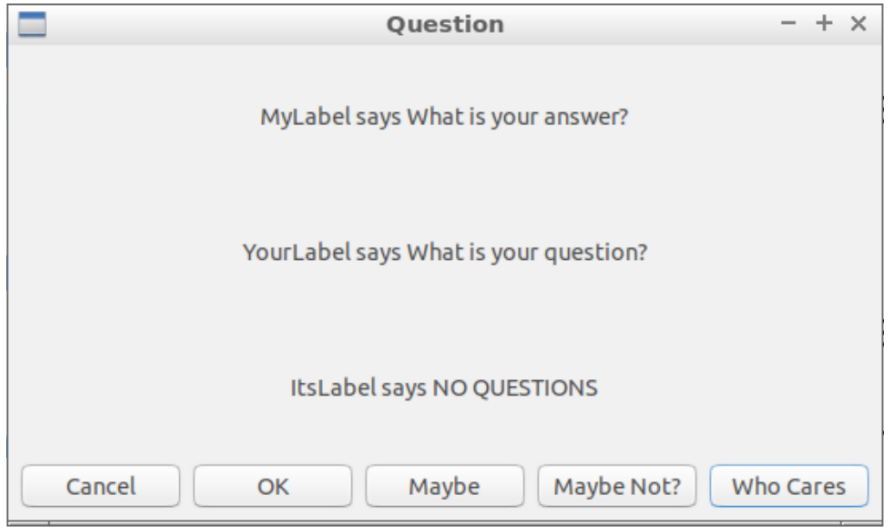
Tells `MyDialog` where to pack `MyLabel`

```
MyLabel.show();
```

`show()` lets *gtkmm* know that we have finished setting the attributes of the widget and that it is ready to be displayed.

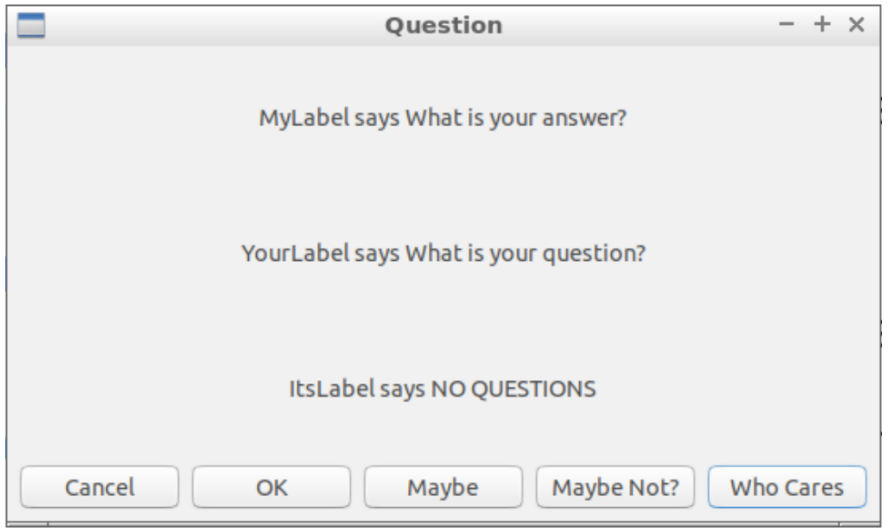
Let's add another Label. We already have..

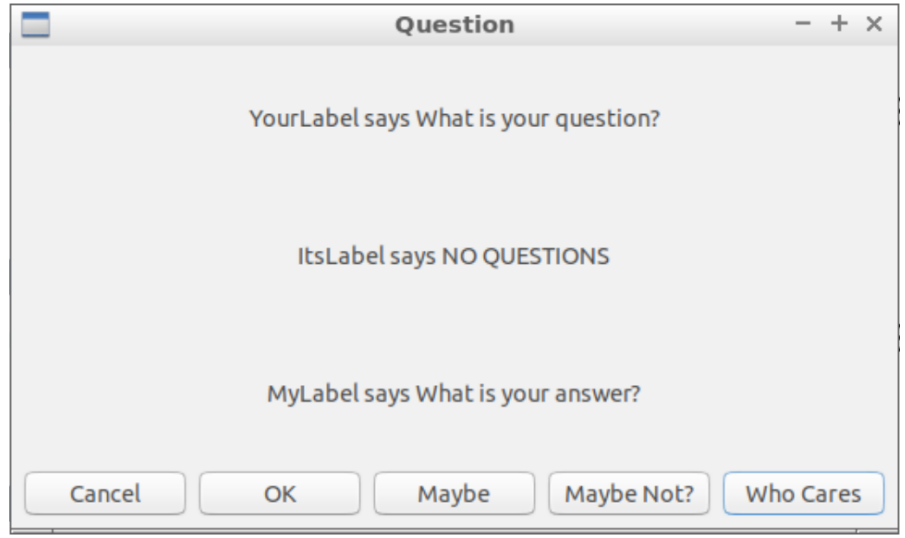
```
Gtk::Label MyLabel("\n\nMyLabel says What is your answer?\n\n");
MyDialog.get_content_area()->pack_start(MyLabel);
MyLabel.show();

Gtk::Label YourLabel("\n\nYourLabel says What is your question?\n\n");
MyDialog.get_content_area()->pack_start(YourLabel);
YourLabel.show();

Gtk::Label ItsLabel("\n\nItsLabel says NO QUESTIONS\n\n");
MyDialog.get_content_area()->pack_start(ItsLabel);
ItsLabel.show();
```



Let's change `pack_start` to `pack_end` for `MyLabel`..

```
Gtk::Label MyLabel("\n\nMyLabel says What is your answer?\n\n");
MyDialog.get_content_area()->pack_end(MyLabel);
MyLabel.show();

Gtk::Label YourLabel("\n\nYourLabel says What is your question?\n\n");
MyDialog.get_content_area()->pack_start(YourLabel);
YourLabel.show();

Gtk::Label ItsLabel("\n\nItsLabel says NO QUESTIONS\n\n");
MyDialog.get_content_area()->pack_start(ItsLabel);
ItsLabel.show();
```





Label Widget

You can specify the text in the constructor or with the `set_text()` method.

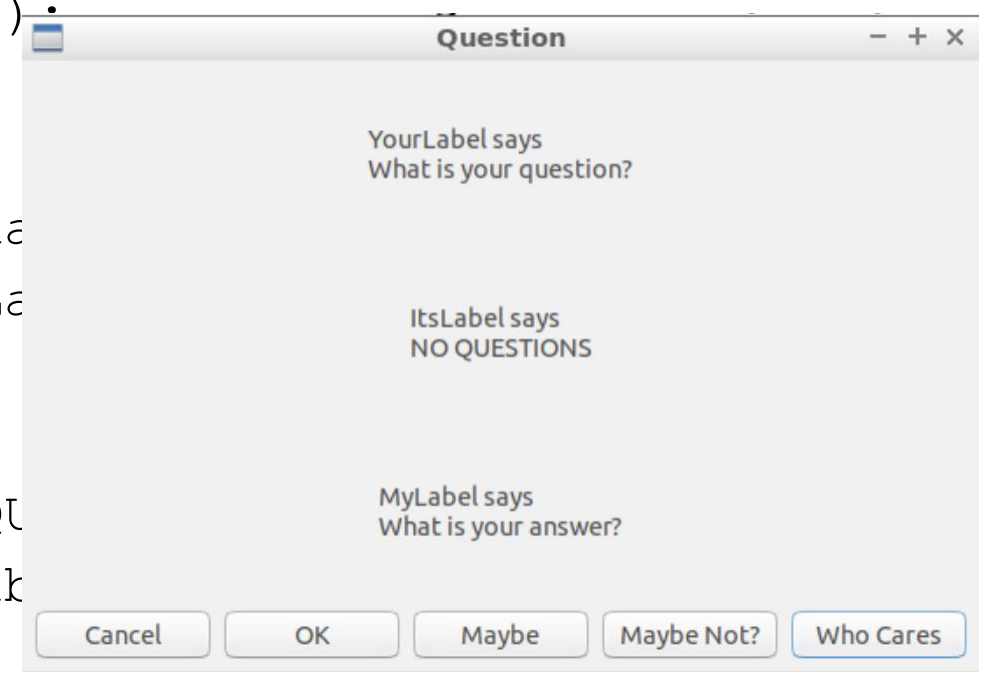
```
Gtk::Label YourLabel("\n\nYourLabel says What is your question?\n\n");
```

```
Gtk::Label MyLabel;
MyLabel.set_text("\n\nMyLabel says What is your answer?\n\n");
```

Label Widget

The width of the label will be adjusted automatically. You can produce multi-line labels by putting line breaks ("\n") in the label string.

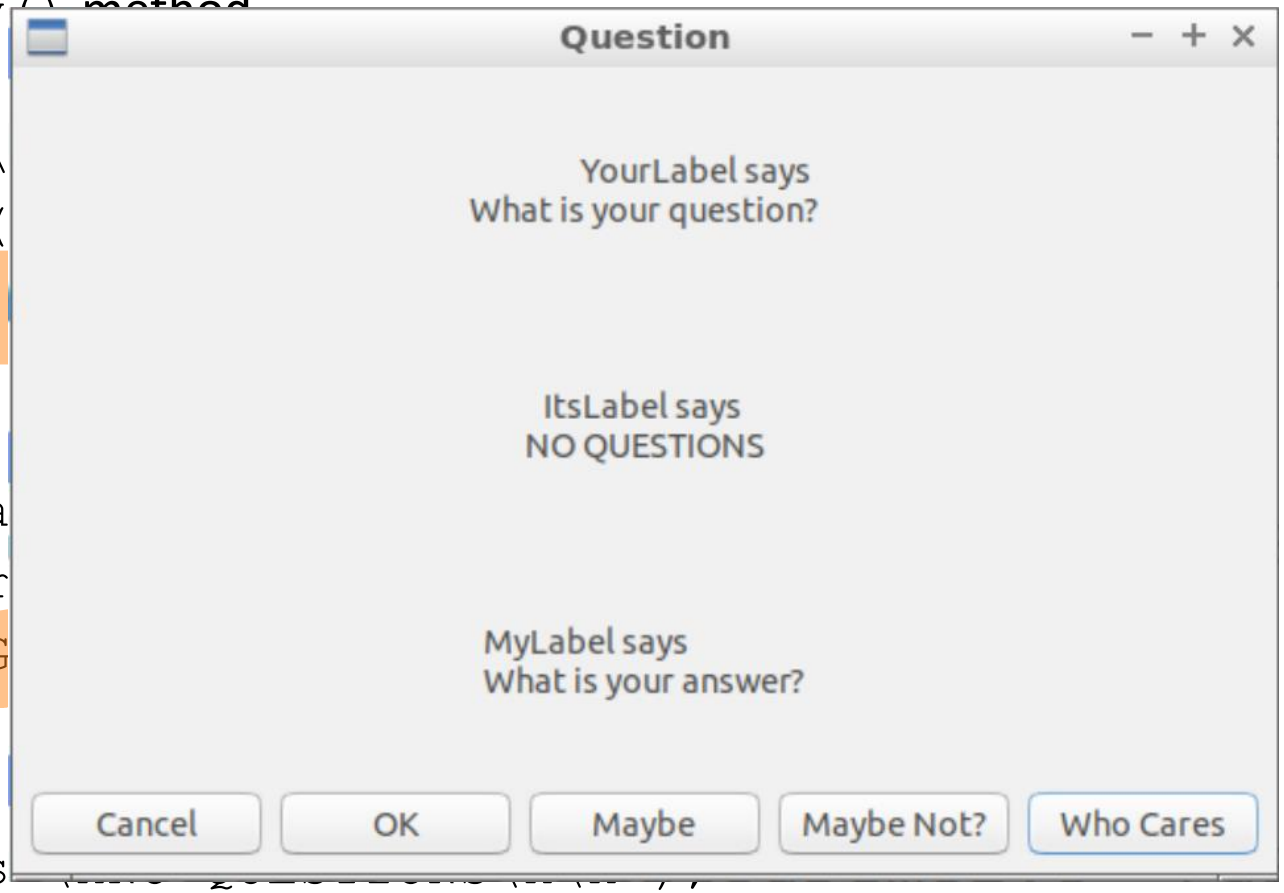
```
Gtk::Label MyLabel("\n\nMyLabel says \nWhat is your answer?\n\n");
MyDialog.get_content_area()->pack_end(MyLabel);
MyLabel.show();

Gtk::Label YourLabel("\n\nYourLabel says \nWha
MyDialog.get_content_area()->pack_start(YourLa
YourLabel.show();

Gtk::Label ItsLabel("\n\nItsLabel says \nNO QU
MyDialog.get_content_area()->pack_start(ItsLab
ItsLabel.show();
```



Label Widget

The label text can be justified using the `set_justify()` method

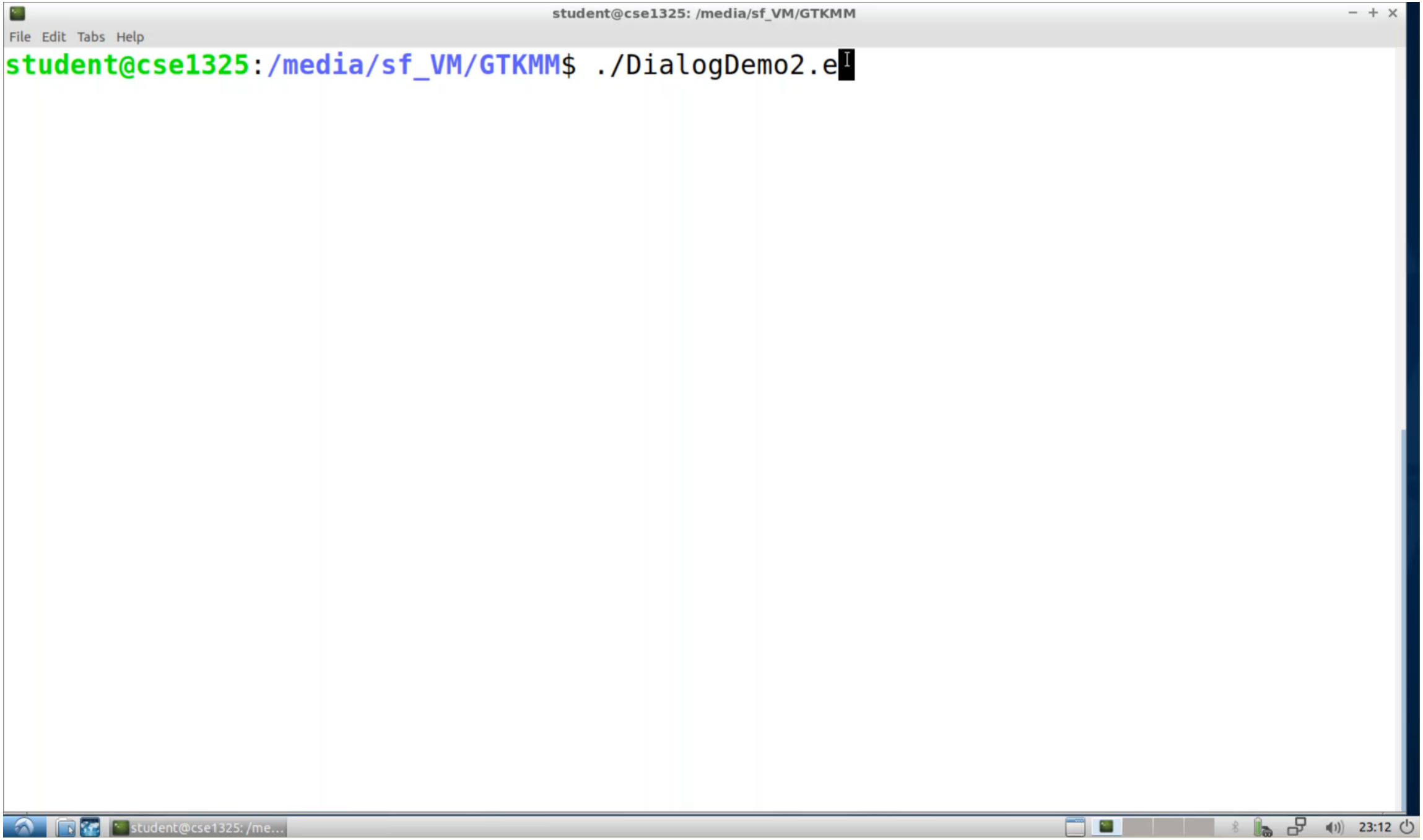
```
Gtk::Label MyLabel("\n\nMyLabel says \
MyDialog.get_content_area()->pack_end(
MyLabel.set_justify(Gtk::JUSTIFY_LEFT)
MyLabel.show();

Gtk::Label YourLabel("\n\nYourLabel sa
MyDialog.get_content_area()->pack_star
YourLabel.set_justify(Gtk::JUSTIFY_RIG
YourLabel.show();

Gtk::Label ItsLabel("\n\nItsLabel says
MyDialog.get_content_area()->pack_start(ItsLabel);
ItsLabel.set_justify(Gtk::JUSTIFY_CENTER);
ItsLabel.show();
```



Label Widget

The widget is also capable of word-wrapping - this can be activated with `set_line_wrap()`.

```
Gtk::Label DogLabel("\n\nDogLabel says \n"
                    "BARK BARK BARK BARK BARK BARK BARK BARK "
                    "BARK BARK BARK BARK BARK BARK BARK BARK "
                    "BARK BARK BARK BARK BARK BARK BARK BARK\n\n");
MyDialog.get_content_area()->pack_start(DogLabel);
DogLabel.set_justify(Gtk::JUSTIFY_CENTER);
DogLabel.show();
```


student@cse1325: /media/sf_VM/GTKMM
File Edit Tabs Help
student@cse1325:/media/sf_VM/GTKMM$ ./DialogDemo2.e
student@cse1325: /me...
23:12

Entry Widget

- Entry widgets allow the user to enter text. You can change the contents with the `set_text()` method, and read the current contents with the `get_text()` method.
- Occasionally you might want to make an Entry widget read-only. This can be done by passing `false` to the `set_editable()` method.
- For the input of passwords, passphrases and other information you don't want echoed on the screen, calling `set_visibility()` with `false` will cause the text to be hidden.
- You can set the max length of the user's entry using the `set_max_length()` method.
- Pango is not supported.

`dialog4Demo.cpp`

Entry Widget

- You might want to be notified whenever the user types in a text entry widget.
- `Gtk::Entry` provides two signals, `activate` and `changed`, for this purpose.
- `activate` is emitted when the user presses the Enter key in a text-entry widget
- `changed` is emitted when the text in the widget changes.
- You can use these, for instance, to validate or filter the text the user types.
- Moving the keyboard focus to another widget may also signal that the user has finished entering text. The `focus_out_event` signal that `Gtk::Entry` inherits from `Gtk::Widget` can notify you when that happens.

```
Gtk::Entry MyEntry;
MyEntry.set_text("Enter your name here");
MyDialog.get_content_area()->pack_start(MyEntry);
MyEntry.set_max_length(50);
MyEntry.show();

int result = MyDialog.run();
std::string TheName = MyEntry.get_text();
std::cout << "The entered name is " << TheName << std::endl;
```

dialog4Demo.cpp

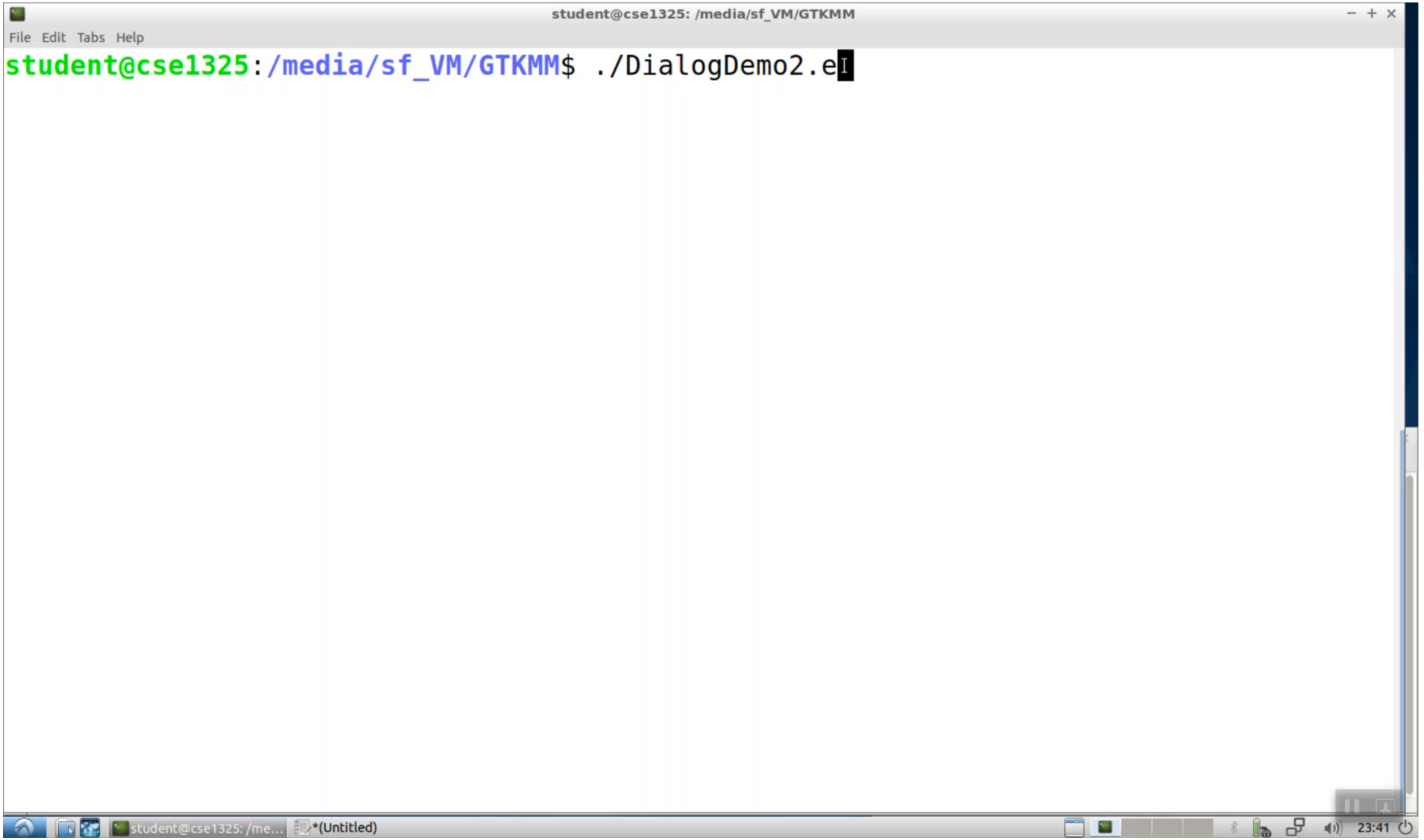
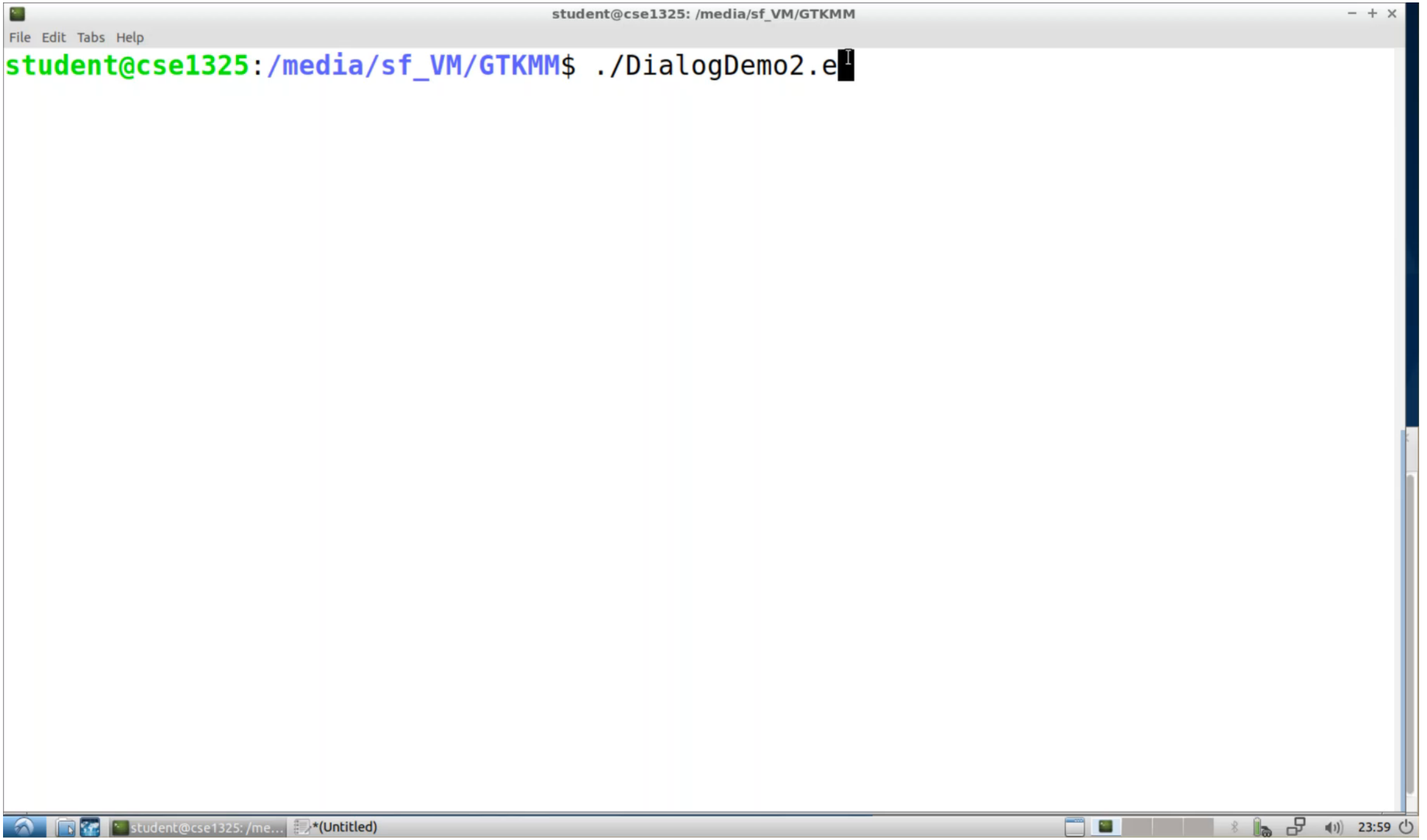

student@cse1325: /media/sf_VM/GTKMM
File Edit Tabs Help
student@cse1325:/media/sf_VM/GTKMM$ ./DialogDemo2.e
student@cse1325: /me...
*(Untitled)
23:41

Image Widget

`Gtk::Image` is a widget that displays an image.

```
Gtk::Image MyImage{"Fred.png"};
MyDialog.get_content_area()->pack_start(MyImage);
MyImage.show();
```

dialog5Demo.cpp


student@cse1325: /media/sf_VM/GTKMM
File Edit Tabs Help
student@cse1325:/media/sf_VM/GTKMM$ ./DialogDemo2.e
student@cse1325: /me...
*(Untitled)
23:59

```
int main (int argc, char* argv[])
{
            auto app = Gtk::Application::create(argc, argv, "app.dialog2");

            Gtk::Dialog MyDialog;

            MyDialog.set_title("Question");

            std::vector<std::string> buttons = {"Cancel", "Definitely Correct", "Maybe Correct", "Completely Guessing"};

            for (int i = 0; i < buttons.size(); i++)
                MyDialog.add_button(buttons[i], i);
            MyDialog.set_default_response(4);

            Gtk::Label MyLabel("\nWhat is this cartoon character's name?\n");
            MyDialog.get_content_area()->pack_start(MyLabel);
            MyLabel.set_justify(Gtk::JUSTIFY_LEFT);
            MyLabel.show();

            Gtk::Entry MyEntry;
            MyEntry.set_text("Enter your guess here");
            MyDialog.get_content_area()->pack_start(MyEntry);
            MyEntry.set_max_length(50);
            MyEntry.show();

            Gtk::Image MyImage{"Fred.png"};
            MyDialog.get_content_area()->pack_start(MyImage);
            MyImage.show();

            int result = MyDialog.run();
            std::string TheName = MyEntry.get_text();

            MyDialog.close();

            std::cout << "Your answer is " << TheName
                      << " and you said you are \""
                      << buttons[result] << "\"" << std::endl;

            return 0;
}
```

gtkmm Memory Management

gtkmm allows the programmer to control the lifetime (that is, the construction and destruction) of any widget in the same manner as any other C++ object.

This flexibility allows you to use `new` and `delete` to create and destroy objects dynamically

or

to use regular class members (that are destroyed automatically when the class is destroyed)

or

to use local instances (that are destroyed when the instance goes out of scope).

gtkmm Memory Management

Although, in most cases, the programmer will prefer to allow containers to automatically destroy their children using `Gtk::manage()`.

The programmer is not required to use `Gtk::manage()`. The traditional `new` and `delete` operators may also be used.

Not destroying containers and their children (widgets contained in the container) will lead to memory leaks.

Each widget is created independently on the heap so if you delete a widget's container but not the widget, then you have memory leak.

`Gtk::manage()` is used to avoid memory leaks.

gtkmm Memory Management

You can let a widget's container control when the widget is destroyed.

In most cases, you want a widget to last only as long as the container it is in.

To delegate the management of a widget's lifetime to its container, first create it with `Gtk::manage()` and pack it into its container with `add()`.

Now, the widget will be destroyed whenever its container is destroyed.

gtkmm Memory Management

gtkmm provides the `manage()` function and `add()` methods to create and destroy widgets.

Every widget except a top-level window must be added or packed into a container in order to be displayed.

The `manage()` function marks a packed widget so that when the widget is added to a container, the container becomes responsible for deleting the widget.

```
Gtk::Label *MyLabel = Gtk::manage(new Gtk::Label("What is your decision?"));
MyDialog->get_content_area()->pack_start(*MyLabel);
MyLabel->show();

Gtk::Entry *MyEntry = Gtk::manage(new Gtk::Entry());
MyEntry->set_text("Enter your name here");
MyEntry->set_max_length(50);
MyEntry->show();
MyDialog->get_content_area()->pack_start(*MyEntry);

Gtk::Image *MyImage = Gtk::manage(new Gtk::Image{"Fred.png"});
MyDialog->get_content_area->pack_start(*MyImage);
MyImage->show();
```

You might see
`get_vbox()`
`get_content_area`
is newer and should be used

`MyLabel`, `MyEntry` and `MyImage` were set to managed when they were created and were then packed into `MyDialog`; therefore, when `MyDialog` is deleted, `MyLabel`, `MyEntry` and `MyImage` will be deleted also.

Let's get rid of that error message

```
Gtk-Message: 15:28:36.675: GtkDialog mapped without a transient parent. This
is discouraged.
```

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

   MainWindow MyWindow;

   return app->run(MyWindow);
}

class MainWindow : public Gtk::Window
{
   public:
      MainWindow();
};
```

```
MainWindow::MainWindow()
{
}
```


student@cse1325: /media/sf_VM/GTKMM
File Edit Tabs Help
student@cse1325:/media/sf_VM/GTKMM$ ./Mainwin.e
student@cse1325: /me...
15:37

The `Box` class is an abstract base class defining a widget that encapsulates functionality for a container that organizes a variable number of widgets into a rectangular area.

`Box` currently has two families of derived classes:

- `HBox` and `Vbox`
- `HButtonBox` and `VButtonBox`

The rectangular area of a `Box` is organized into either a single row or a single column of child widgets depending upon whether the box is an `HBox` or a `Vbox`.

`Box` uses packing. Packing refers to adding widgets with reference to a particular position in a `Container`.

For a `Box`, there are two reference positions: the start and the end of the box.

For a `VBox`, the start is defined as the top of the box and the end is defined as the bottom.
For a `HBox`, the start is defined as the left side and the end is defined as the right side.

Repeated calls to `pack_start()` pack widgets into a `Box` from start to end.

The `pack_end()` method adds widgets from end to start. You may intersperse these calls and add widgets from both ends of the same `Box`.

Because `Box` is a `Container`, we can also use `add()` to insert widgets into the box and they will be packed as if with the `pack_start()` method. Use `remove()` to remove widgets from the `Box`.

```
MainWindow::MainWindow()
{
    Gtk::Box *MainVBox =
        Gtk::manage(new Gtk::Box(Gtk::ORIENTATION_VERTICAL, 0));

    add(*MainVBox);
```

Adds our new box to our Window widget

This creates a vertical box container.

It is constructed with “ORIENTATION_VERTICAL” to define it as vertical and “0” for no padding.

This widget is also managed so that it will automatically delete when `window` is deleted.

App2Demo.cpp

`Gtk::Window` can contain at most one widget.

So how can we use a window for anything useful?

By placing a multiple-child container in the window.

The most useful container widgets are `Gtk::Box` and `Gtk::Table`.

`Gtk::Box` can arrange child widgets vertically (GTK_ORIENTATION_VERTICAL)
can arrange child widgets horizontally (GTK_ORIENTATION_HORIZONTAL)

`Gtk::Table` arranges its widgets in a grid.

Now let's create something to pack into our box!!

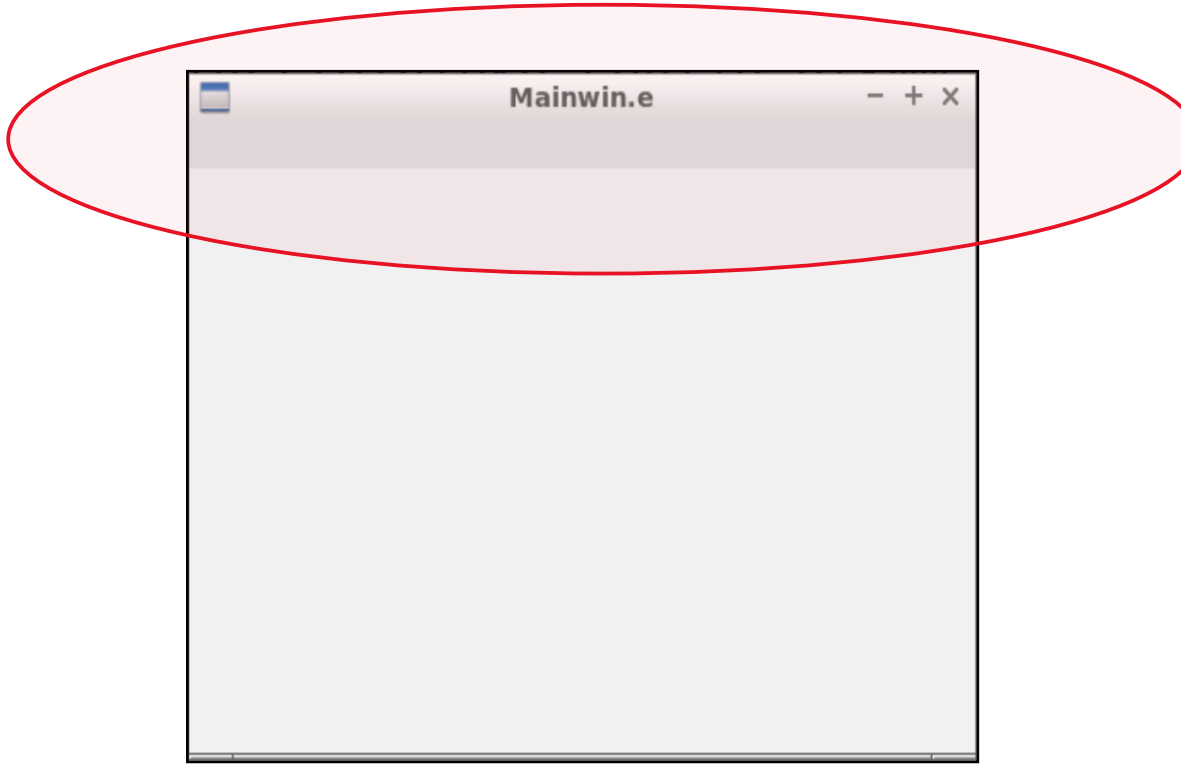
```
Gtk::MenuBar *MyMenuBar = Gtk::manage(new Gtk::MenuBar());

MainVbox->pack_start(*MyMenuBar, Gtk::PACK_SHRINK, 0);
```

This will create a standard menu bar which holds `Gtk::Menu` submenu items.

The `MenuBar` widget looks like

The `pack_start` function is given the pointer to the `MenuBar` so it can pack the `MenuBar` into the top of the vertical box. It also shrinks the cell to the size of the `MenuBar` and does not pad the cell.


Mainwin.e


Mainwin.e

```
Gtk::MenuItem *MyFileMenu =
                Gtk::manage(new Gtk::MenuItem("_File", true));

MyMenuBar->append(*MyFileMenu);
```

`MenuItem` is a child item of menus and it handles highlighting, alignment, events and submenus.

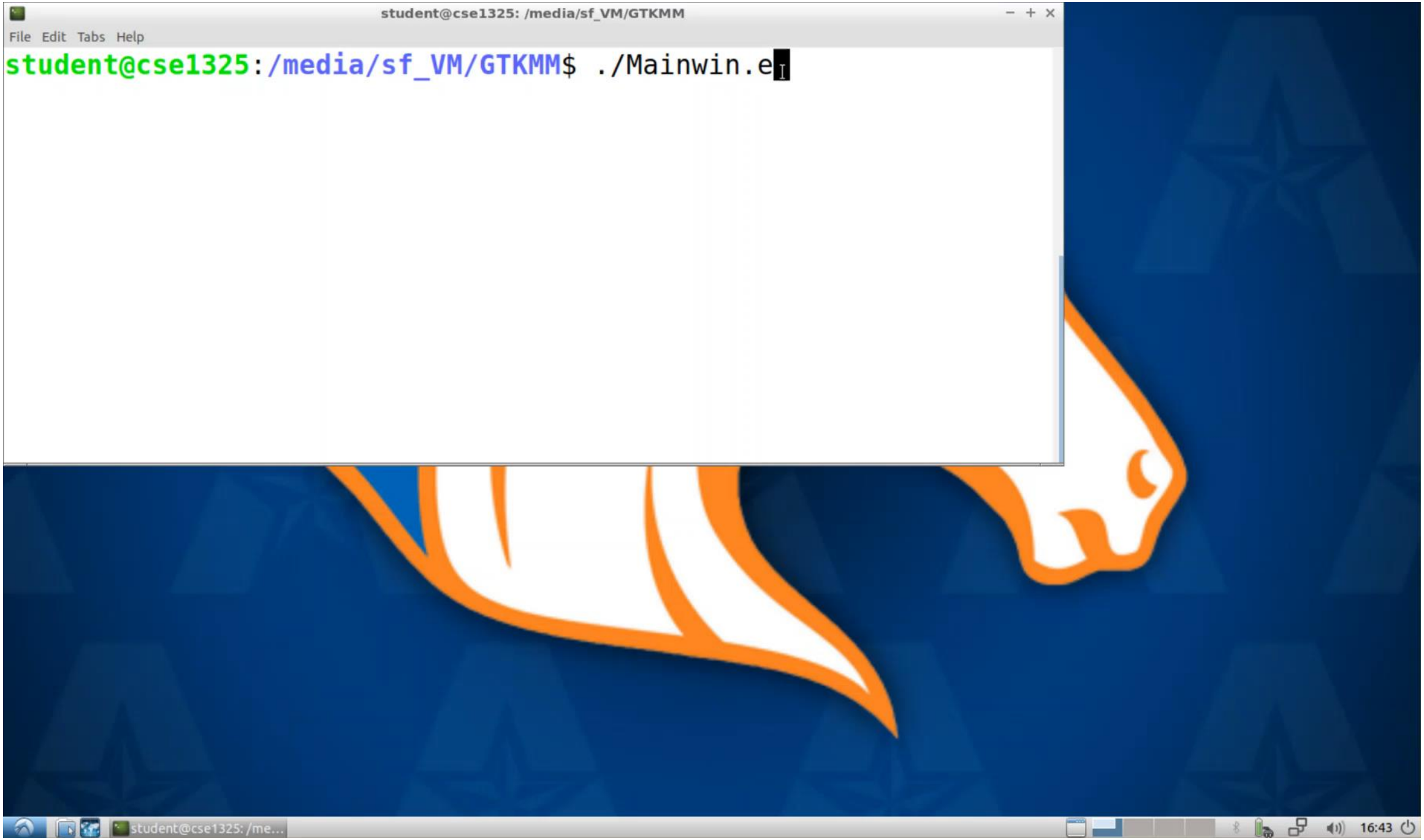
It derives from `Gtk::Bin` so it can hold any valid child widget.

Constructed with the string "`File`" which is what the menu item will display. The second parameter to the constructor turns on mnemonics.

Mnemonics are underlined characters in certain widgets (like menus) used for keyboard navigation.

Mnemonics are created by providing a string with an underscore before the mnemonic character, such as "`_File`".

Inherited member function `append` adds the new `MenuItem` to the end of the `MenuBar`.


student@cse1325: /media/sf_VM/GTKMM
File Edit Tabs Help
student@cse1325:/media/sf_VM/GTKMM$ ./Mainwin.e
student@cse1325: /me...
16:43

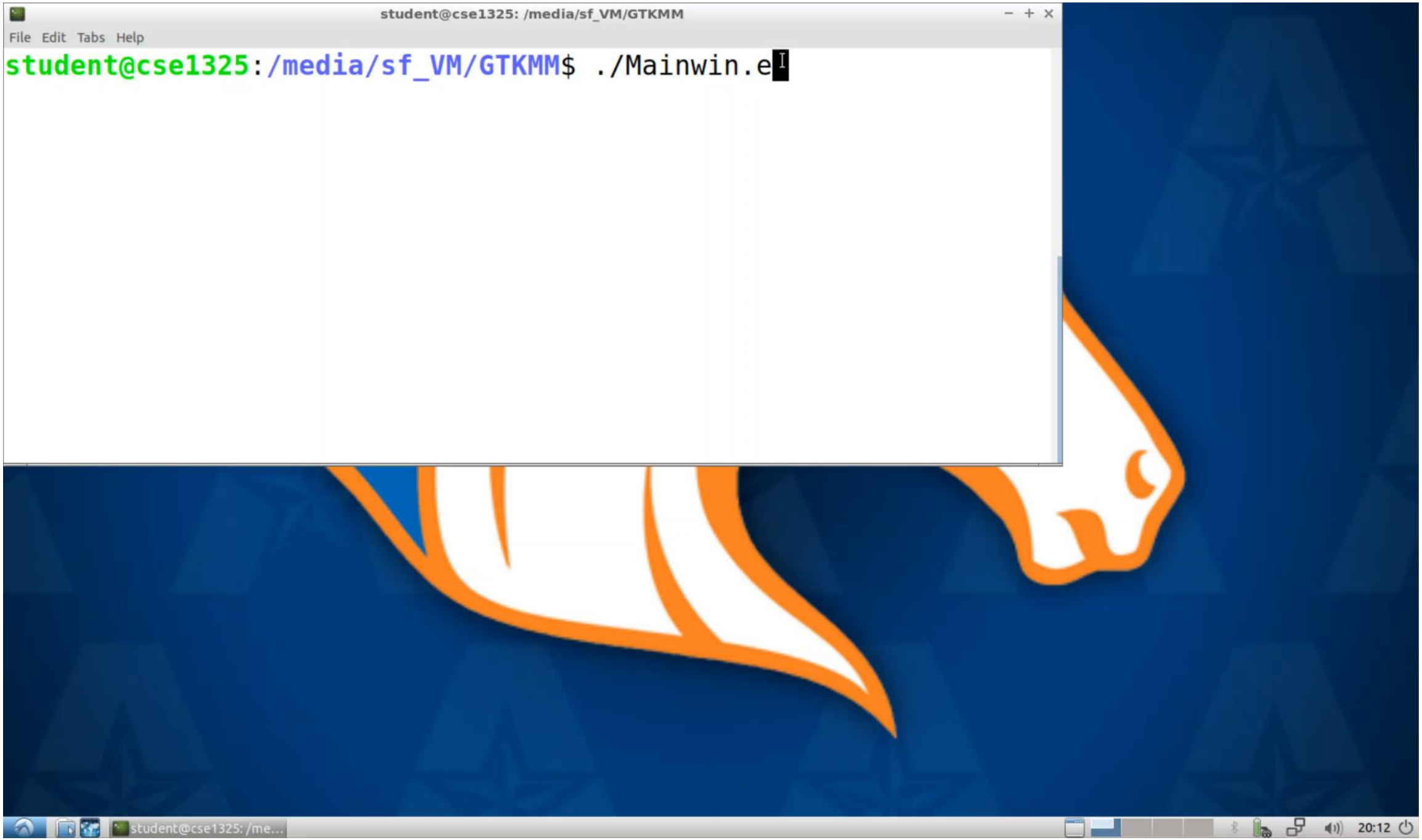
```
Gtk::Menu *MyFileSubMenu = Gtk::manage(new Gtk::Menu());
```

```
MyFileMenu->set_submenu(*MyFileSubMenu);
```

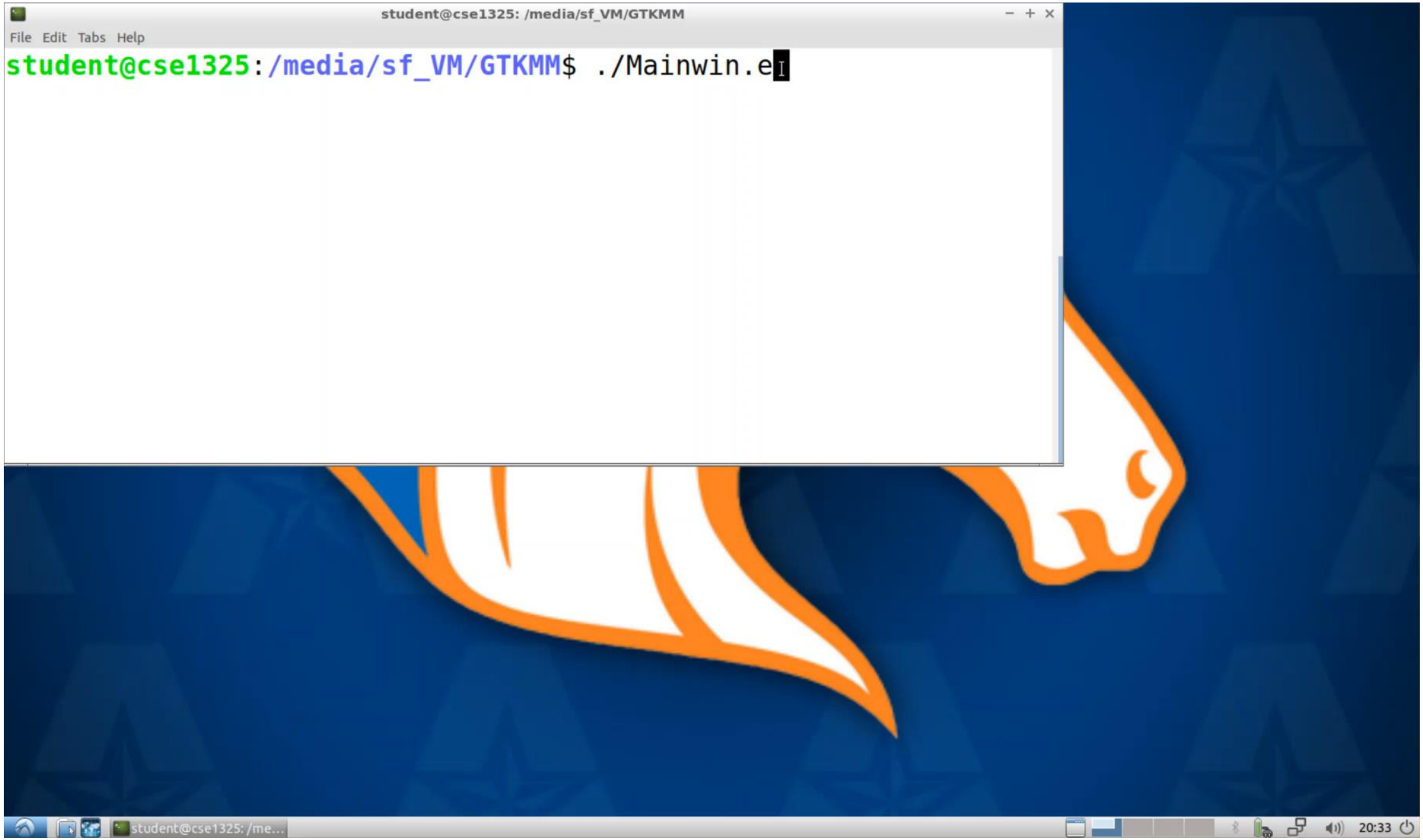
Instantiate a new `Menu` object called `MyFileSubMenu`. We will make this a submenu under `File`.

Member function
`set_submenu()`
sets object `MyFileSubMenu` as a submenu of `MyFileMenu`.

```
Gtk::Menu *MyFileSubMenu = Gtk::manage(new Gtk::Menu());
MyFileMenu->set_submenu(*MyFileSubMenu);

Gtk::MenuItem *SubMenuItem_New =
                Gtk::manage(new Gtk::MenuItem("_New", true));
MyFileSubMenu->append(*SubMenuItem_New);

Gtk::MenuItem *SubMenuItem_Open =
                Gtk::manage(new Gtk::MenuItem("_Open", true));
MyFileSubMenu->append(*SubMenuItem_Open);

Gtk::MenuItem *SubMenuItem_Save =
                Gtk::manage(new Gtk::MenuItem("_Save", true));
MyFileSubMenu->append(*SubMenuItem_Save);

Gtk::MenuItem *SubMenuItem_Close =
                Gtk::manage(new Gtk::MenuItem("_Close", true));
MyFileSubMenu->append(*SubMenuItem_Close);
```


student@cse1325: /media/sf_VM/GTKMM
File Edit Tabs Help
student@cse1325:/media/sf_VM/GTKMM$ ./Mainwin.e
student@cse1325: /me...
20:12

```
Gtk::MenuItem *MyEditMenu =
          Gtk::manage(new Gtk::MenuItem("_Edit", true));
MyMenuBar->append(*MyEditMenu);

Gtk::Menu *MyEditSubMenu = Gtk::manage(new Gtk::Menu());
MyEditMenu->set_submenu(*MyEditSubMenu);

Gtk::MenuItem *SubMenuItem_Cut =
          Gtk::manage(new Gtk::MenuItem("_Cut", true));
MyEditSubMenu->append(*SubMenuItem_Cut);

Gtk::MenuItem *SubMenuItem_Copy =
          Gtk::manage(new Gtk::MenuItem("_Copy", true));
MyEditSubMenu->append(*SubMenuItem_Copy);

Gtk::MenuItem *SubMenuItem_Paste =
          Gtk::manage(new Gtk::MenuItem("_Paste", true));
MyEditSubMenu->append(*SubMenuItem_Paste);
```


student@cse1325: /media/sf_VM/GTKMM
File Edit Tabs Help
student@cse1325:/media/sf_VM/GTKMM$ ./Mainwin.e
student@cse1325: /me...
20:33

We made a vertical `Box` object at the start of the function and called `add()` to add it to our `Window` object.

```
Gtk::Box *MainVBox =
     Gtk::manage(new Gtk::Box(Gtk::ORIENTATION_VERTICAL, 0));

add(*MainVBox);
```

Now we want to make a horizontal `Box` object and add it our `Window` object.

```
Gtk::Box *hBox =
     Gtk::manage(new Gtk::Box(Gtk::ORIENTATION_HORIZONTAL, 0));

add(*hBox);
```

```
student@cse1325:/media/sf_VM/GTKMM$ make
g++ -c -g -std=c++11  Mainwin.cpp -o Mainwin.o `/usr/bin/pkg-config
gtkmm-3.0 --cflags --libs`
g++ -g -std=c++11  Mainwin.o -o Mainwin.e `/usr/bin/pkg-config gtkmm-3.0
--cflags --libs`

student@cse1325:/media/sf_VM/GTKMM$ ./Mainwin.e

(Mainwin.e:2454): Gtk-WARNING **: 20:41:54.072: Attempting to add a
widget with type gtkmm__GtkBox to a gtkmm__GtkWindow, but as a GtkBin
subclass a gtkmm__GtkWindow can only contain one widget at a time; it
already contains a widget of type gtkmm__GtkBox
```

`Gtk::Window` can contain at most one widget.

```
Gtk::Box *hBox =
      Gtk::manage(new Gtk::Box(Gtk::ORIENTATION_HORIZONTAL, 0));

add(*hBox);
```

Caused WARNING during runtime.

Since we cannot add `hBox` to our main `Window`, how do we add it?

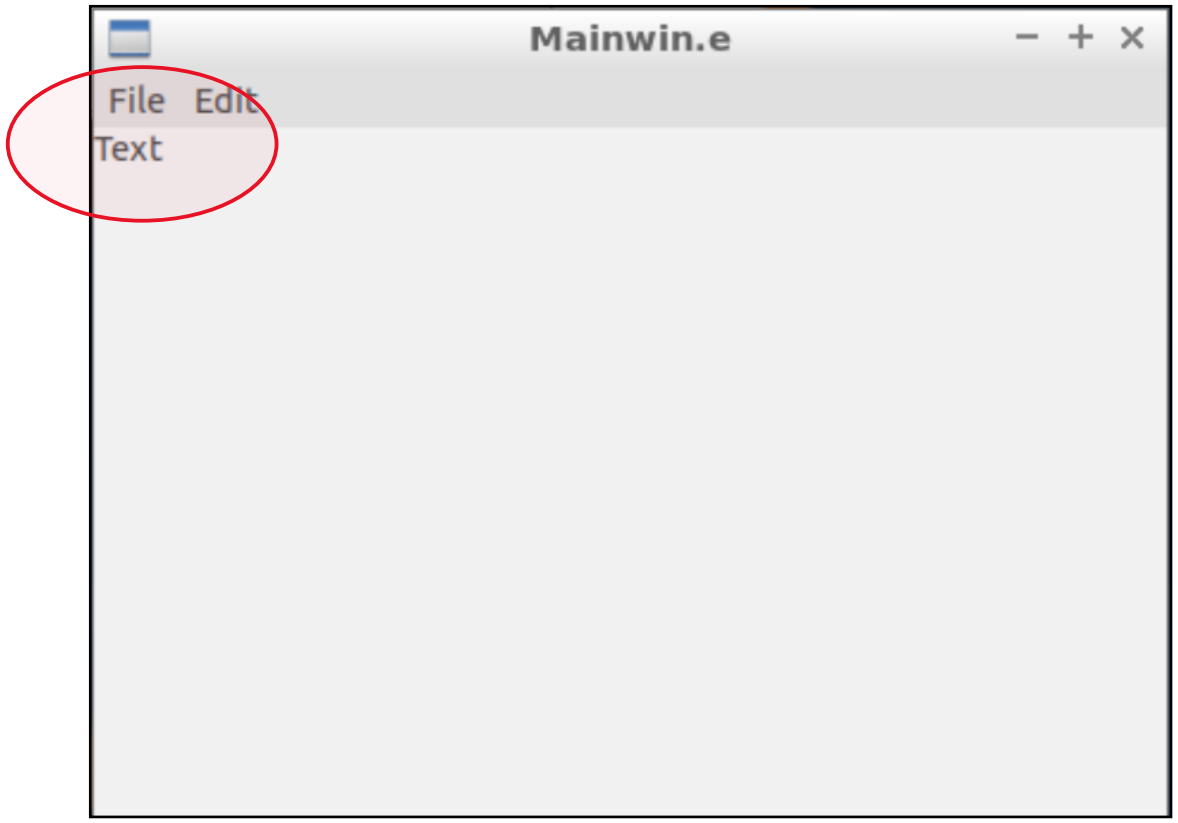
By adding it to the multiple-child container that we already added to our `Window`.

```
MainVBox->add(*hBox);
```

```
Gtk::Label *TextMsg = Gtk::manage(new Gtk::Label{"Text"});
hBox->add(*TextMsg);
```

Why did `Text` show up on the left side of our window?

For a `HBox`, the start is defined as the left side and the end is defined as the right side.



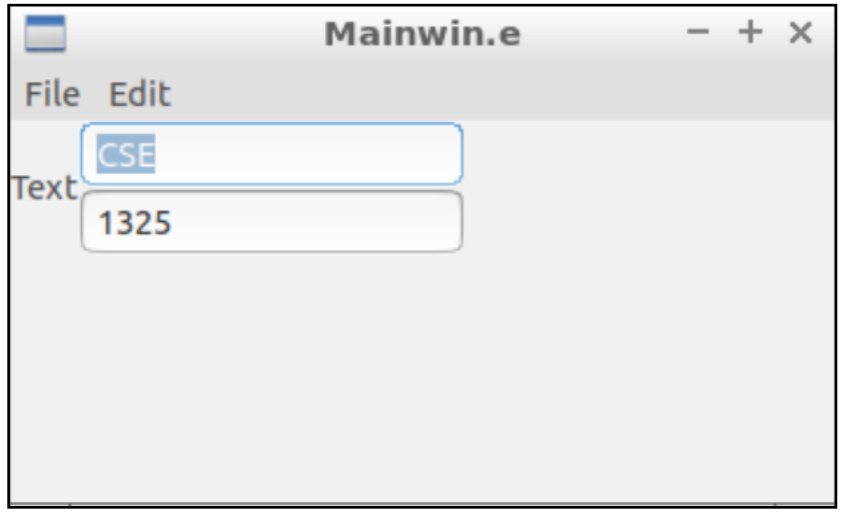
```
Gtk::Label *TextMsg2 = Gtk::manage(new Gtk::Label{"Text2"});
hBox->add(*TextMsg2);
```

Where would `TextMsg2 appear?`

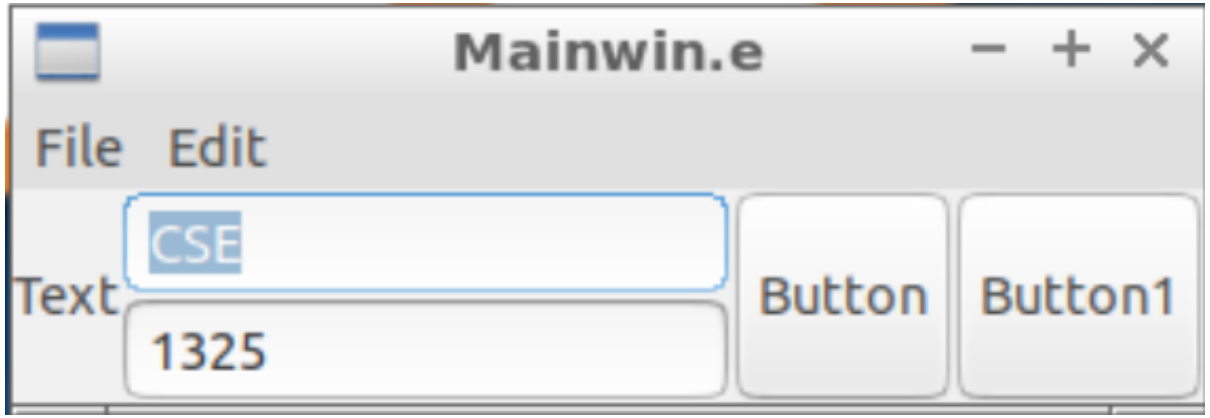
What if we want to add some Entry widgets that stack vertically rather than arranged horizontally?

```
Gtk::Box *vBox1 = Gtk::manage(new Gtk::Box(Gtk::ORIENTATION_VERTICAL, 0));
hBox->add(*vBox1);

Gtk::Entry *entry1 = Gtk::manage(new Gtk::Entry());
entry1->set_text("CSE");
vBox1->add(*entry1);

Gtk::Entry *entry2 = Gtk::manage(new Gtk::Entry());
entry2->set_text("1325");
vBox1->add(*entry2);
```

When `add()` was used to pack `vBox1` into `hBox`, `pack_start()` was used as the default and it was packed from left to right so right next to our `Text` label .



```
Gtk::Button* button = Gtk::manage(new Gtk::Button("Button"));
hBox->add(*button);
```



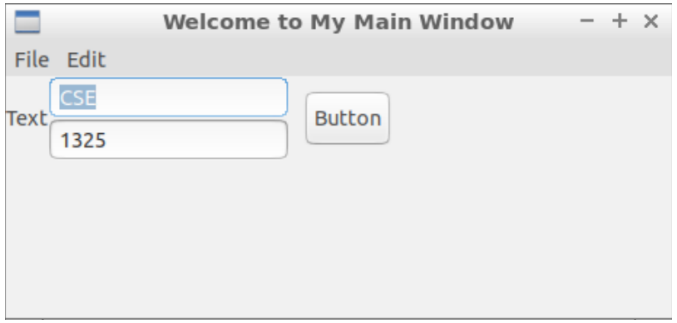
How would this code change the window?

```
Gtk::Button* button1 = Gtk::manage(new Gtk::Button("Button1"));
hBox->add(*button1);
```

```
Gtk::Button* button = Gtk::manage(new Gtk::Button("Button"));
hBox->add(*button);
button->set_border_width(10);
```

The button is given a border width to put some space between the button and the other widgets.

set_border_width(10)



set_border_width(3)

App2Demo.cpp

At the end of the constructor function, we need this line

```
MainVBox->show_all();
```

This function will show the `MainVBox` and all of its children.

If the `show_all()` function was not called, an empty window would be displayed.

At the end of `main()`, we have

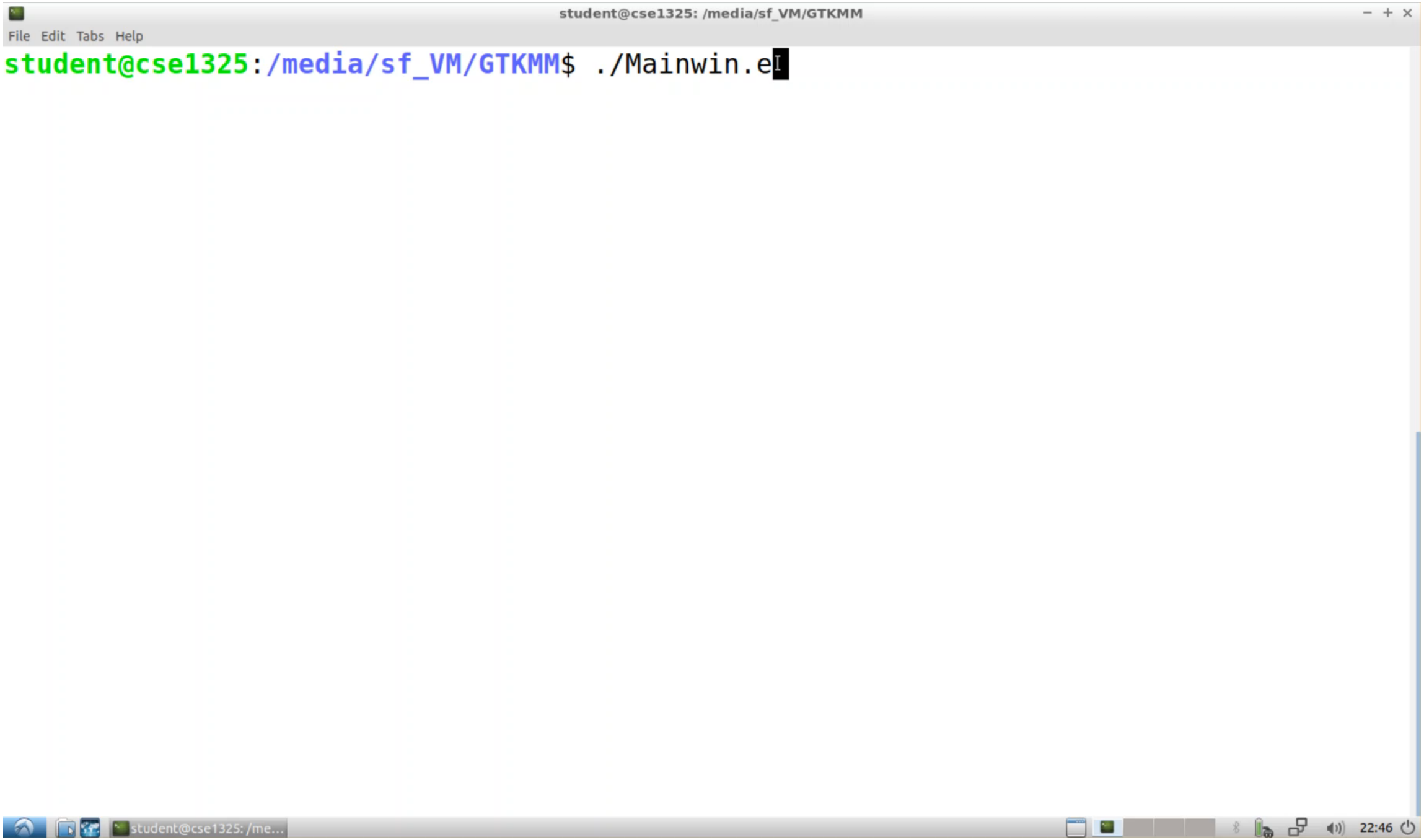
```
app->run(MyWindow);
```

This is the main gtkmm loop. The function is given the window so it can be created and handle all of the control buttons such as minimize, maximize and close. The `run` function will also return a result which is then returned by the main function.

```
MainWindow::MainWindow()
{
    set_title("Welcome to My Main Window");
    set_default_size(800,400);
```

Because the `MainWindow` class is a `Gtk::Window`, the functions from `Gtk::Window` can be used directly without the need for a . or -> operator.

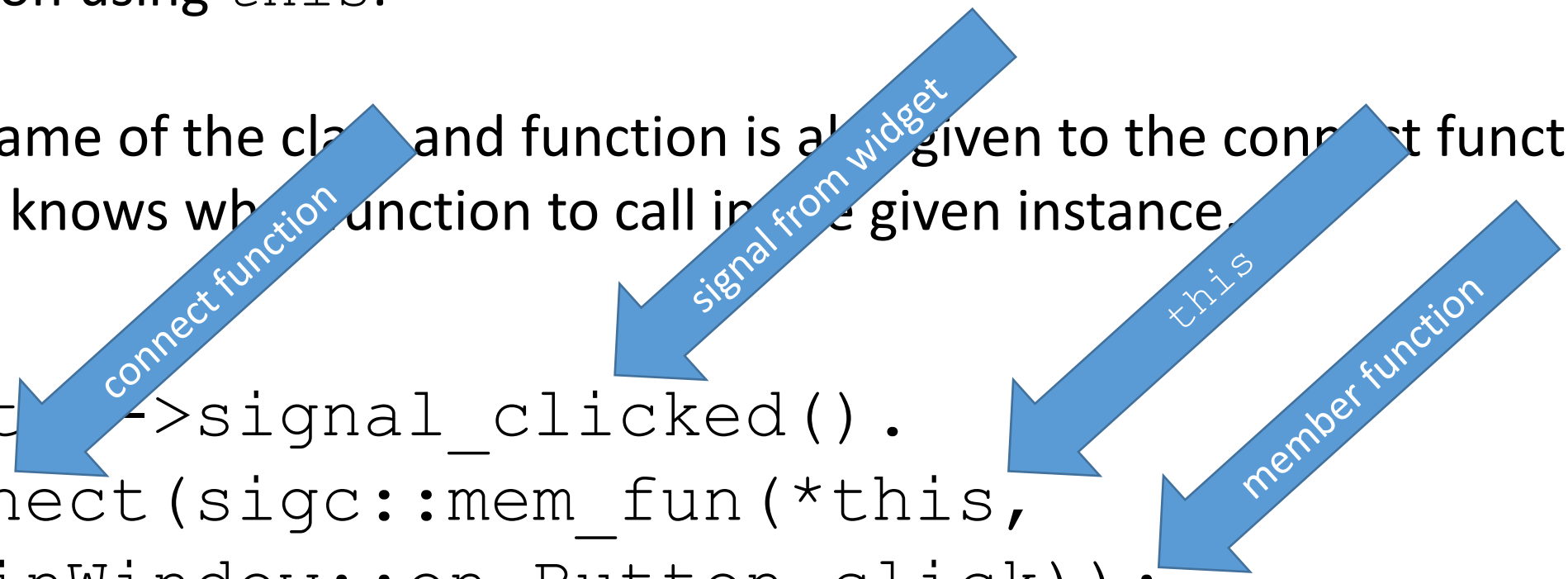
App3Demo.cpp


student@cse1325: /media/sf_VM/GTKMM
File Edit Tabs Help
student@cse1325:/media/sf_VM/GTKMM$ ./Mainwin.e
student@cse1325: /me...
22:46

gtkmm makes it very easy to connect widget signals to functions.

The instance of the class that will handle the event is handed to the connect function using `this`.

The name of the cla and function is al given to the conn t function so the event knows wh unction to call in e given instance.




```
butt ->signal_clicked().
connect(sigc::mem_fun(*this,
&MainWindow::on_Button_click));
```

App3Demo.cpp

```
Gtk::Button* button = Gtk::manage(new Gtk::Button("Button"));
hBox->add(*button);
button->set_border_width(3);
button->signal_clicked().connect(sigc::mem_fun(*this,
                                 &MainWindow::on_Button_click));
```

When `button` is clicked, member function `on_Button_click()` will called using the current instance object.

```
void MainWindow::on_Button_click()
{
    hide();
}
```

Reverses the effects of `show()`, causing the widget to be hidden (invisible to the user).

App3Demo.cpp


student@cse1325: /media/sf_VM/GTKMM
File Edit Tabs Help
student@cse1325:/media/sf_VM/GTKMM$ ./Mainwin.e
student@cse1325: /me...
11:43

```
button->signal_clicked().connect(sigc::mem_fun(*this,
                                  &MainWindow::on_Button_click));
```

`on_Button_click()` is a member function of `MainWindow`; therefore, needs to be added to the class

```
class MainWindow : public Gtk::Window
{
     public:
          MainWindow();
          void on_Button_click();
};

void MainWindow::on_Button_click()
{
     hide();
}
```

```
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);
Gtk::MenuItem *menuitem_quit = Gtk::manage(new Gtk::MenuItem("_Quit", true));
menuitem_quit->signal_activate().
                connect(sigc::mem_fun(*this, &MyWindow::on_quit_click));
filemenu->append(*menuitem_quit);
```

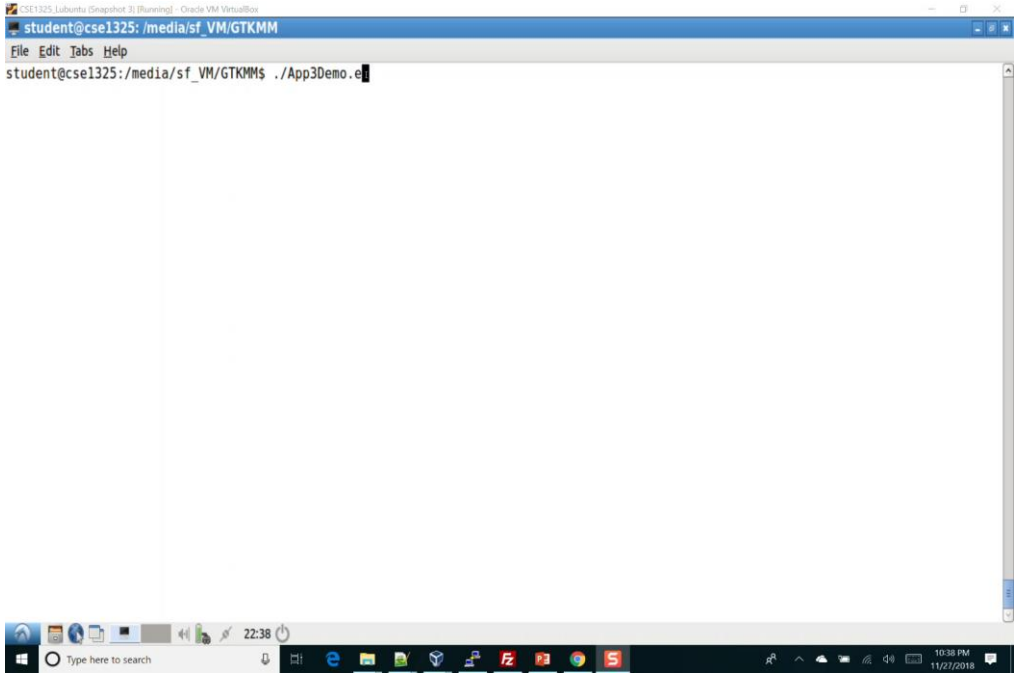
gtkmm makes it very easy to connect signals to functions.

The instance of the class that will handle the event is handed to the connect function which will be this instance.

The name of the class and function is also given to the connect function so the event knows what function to call in the given instance.

App3Demo.cpp

```
void MyWindow::on_big_button1_click()
{
    dialog("Big Button 1 Pressed!");
}

void MyWindow::on_button2_click()
{
    dialog("Button 2 Pressed!");
}

void MyWindow::on_button3_click()
{
    dialog("Button 3 Pressed!");
}

void MyWindow::dialog(Glib::ustring msg)
{
    Gtk::MessageDialog dlg(msg, false, Gtk::MESSAGE_INFO,
                           Gtk::BUTTONS_OK, true);
    dlg.set_title("Message Dialog - Part 3");
    dlg.run();
}
```

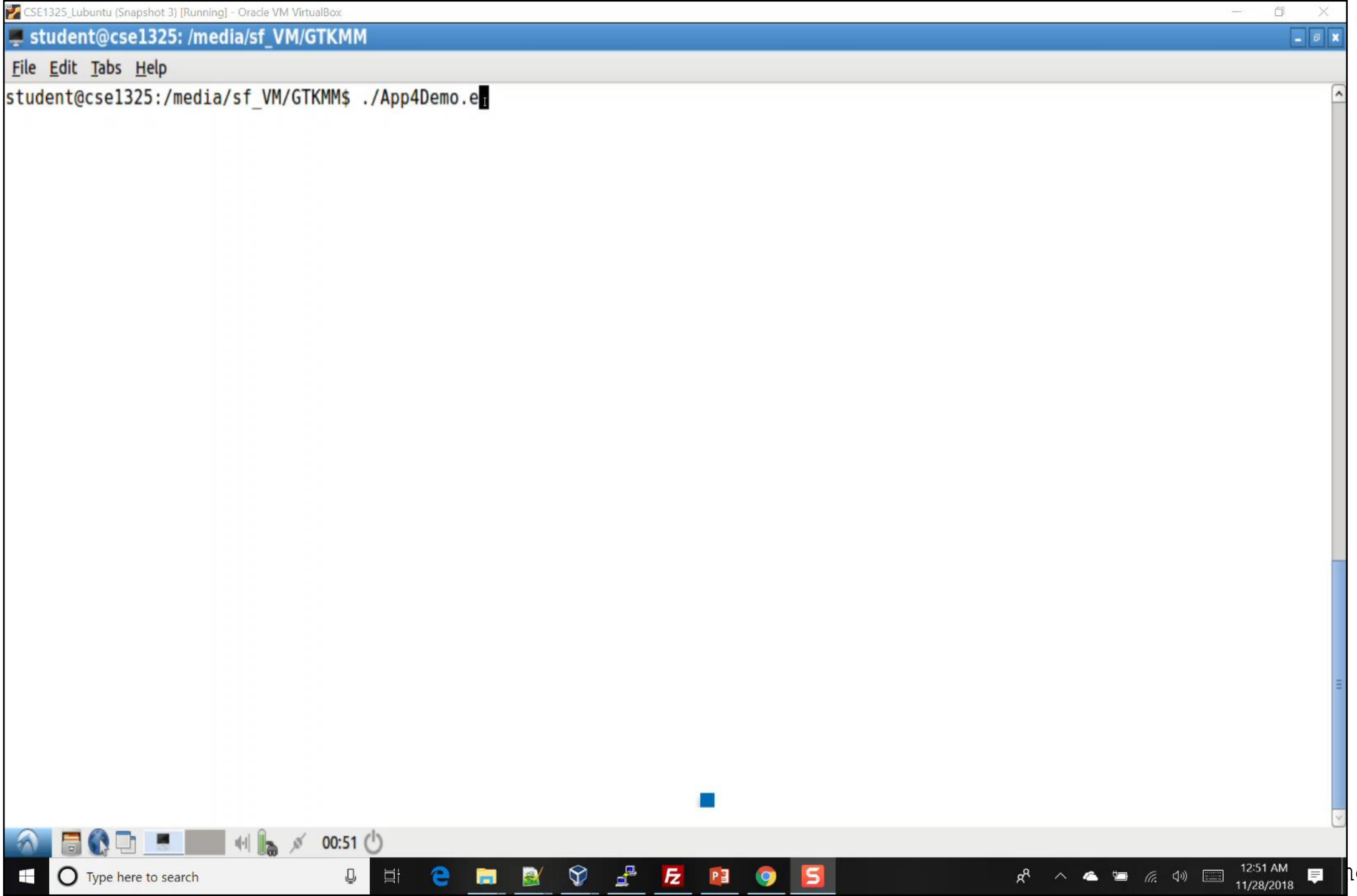


App3Demo.cpp

```
menuitem_quit->signal_activate().
                connect(sigc::mem_fun(*this, &MyWindow::on_quit_click));


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

This is the function connected to the drop down menu quit event. Hiding the main window in a Gtkmm application will end it’s execution. hide is a function in `Gtk::Window` which `MyWindow` has inherited so it can be called directly.

App3Demo.cpp


CSE1325_Lubuntu (Snapshot 3) [Running] - Oracle VM VirtualBox
student@cse1325: /media/sf_VM/GTKMM
File Edit Tabs Help
student@cse1325:/media/sf_VM/GTKMM$ ./App4Demo.e
00:51
Type here to search
12:51 AM
11/28/2018

no.cpp

```
#ifndef MYWINDOW_H
#define MYWINDOW_H

#include <gtkmm.h>

class MyWindow : public Gtk::Window
{
  public:
     MyWindow();
     virtual ~MyWindow();
     void on_login_click(Gtk::Entry *, Gtk::Entry *);
     void on_quit_click();
};
#endif // MYWINDOW_H
```

This is a new function to handle the login button click. The function will require the entry widgets which will be the username and password.

App4Demo.cpp

```
Gtk::Label *lusername = Gtk::manage(new Gtk::Label("Username: "));
grid->attach(*lusername, 0, 0, 1, 1);

Gtk::Label *lpassword = Gtk::manage(new Gtk::Label("Password: "));
grid->attach(*lpassword, 0, 1, 1, 1);
```

This will create 2 labels with the given text and place them in the far left cells with the password label being below the username (0,0 and 0,1).

Both label widgets will only span 1 cell wide and 1 cell in height.

App4Demo.cpp

```
Gtk::Entry *eusername = Gtk::manage(new Gtk::Entry());
eusername->set_hexpand(true);
grid->attach(*eusername, 1, 0, 2, 1);

Gtk::Entry *epassword = Gtk::manage(new Gtk::Entry());
epassword->set_hexpand(true);
epassword->set_visibility(false);
grid->attach(*epassword, 1, 1, 2, 1);
```

This will create 2 entry widgets for accepting text.

The username and password entry boxes will be placed on the cell right of the labels (1,0 and 1,1).

Both entry boxes will span 2 cells wide so the login button will only be half the size of the entry boxes.

Both entry boxes will be told to expand horizontally and take up all available space (`set_hexpand(true)`).

The password entry box is set to not show the text as it is a password box (`set_visibility(false)`).

```
Gtk::Button *blogin = Gtk::manage(new Gtk::Button("Login"));

blogin->signal_clicked().connect(sigc::bind<Gtk::Entry*, Gtk::Entry*>
                         (sigc::mem_fun(*this, &MyWindow::on_login_click),
                          eusername, epassword));
grid->attach(*blogin, 2, 2, 1, 1);
```

This will create a button with the label “Login” and place it on the third row both horizontally and vertically (2,2).

The button will only span 1 row horizontally and vertically.

The connect function is given a cast of `<Gtk::Entry*, Gtk::Entry*>` and the 2 entry boxes are given at the end.

This will automatically send the pointers of the 2 entry boxes when the event is triggered.

App4Demo.cpp

```
void MyWindow::on_login_click(Gtk::Entry *euname, Gtk::Entry *epword)
{
    if(euname->get_text().compare("admin") == 0 &&
       epword->get_text().compare("password") == 0)
    {
        Gtk::MessageDialog dlg("You are now logged in.", false,
                               Gtk::MESSAGE_INFO, Gtk::BUTTONS_OK, true);
        dlg.set_title("My App - Part 4");
        dlg.run();
    }
    else
    {
        Gtk::MessageDialog dlg("Unknown username or password.", false,
                               Gtk::MESSAGE_ERROR, Gtk::BUTTONS_OK, true);
        dlg.set_title("My App - Part 4");
        dlg.run();
    }
}
```

This will compare the text of both the entry boxes with the text “admin” and “password”.

If the compare is true it will equal 0.
If both compares equal 0 the info dialog to show “You are now logged in.” will be shown.
If both compares do not equal 0 the error dialog to show “Unknown username or password.” will be shown.
To change the username and password, edit this line.

App4Demo.cpp