Overview/Intro to GUIs

Remember that when we are programming, we are really:

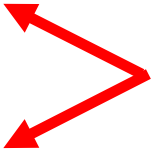
1. Solving a real-world problem (*How can I get songs recommended to me?* or *How can I send disappearing pictures to my friends?)* **or** trying to represent a real-life scenario (*a restaurant with waiters and customers* or *a hotel with staff and guests*)
    a. We are not yet programming at this phase-we are simply identifying what we will be the subject of our programming

2. Using the computer as a tool to help us do this. In order to use the computer as a tool, we need to "speak" to it using one of many programming languages (like C++)
    a. Programming alone is pointless-we need something to program
    b. We are essentially representing our problem or our real world scenario in a programming language to "explain" or "tell" the computer about it

Object-oriented programming:

A style of programming where the world can be broken down into objects. THIS IS NOT A C++ CONCEPT, THIS IS A PROGRAMMING PARADIGM (meaning that different languages can support this type of programming, not just C++). Also note that C++ is not strictly an object-oriented language. If you remember from the beginning of the semester, I was able to write programs in C++ without objects:

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

int wallet_total(std::vector <int> money)
{
    if(money.size()==0)//nothing in wallet yet
    {
      std::cout<<"Nothing in your wallet yet."<<std::endl;
      return 0;
    }

    int total=0;

    for(int i=0;i<money.size();i++)
    {
      total+=money[i];
    }

    return total;
}

bool check_total(int total, int goal)
{
  if(total<=goal)
  {
    std::cout<<"You hit your goal without going over."<<std::endl;
    return true;
  }

  else
  {
    std::cout<<"You went over your goal."<<std::endl;
    return false;
  }
}


int main()
{
  int goal;
  int add;
  std::vector <int> wallet;

  std::cout<<"Enter your goal amount:"<<std::endl;
  std::cin>>goal;

  while(wallet_total(wallet)<goal)
  {
    std::cout<<"Enter amount to add to wallet: "<<std::endl;
    std::cin>>add;
    wallet.push_back(add);
  }

  check_total(wallet_total(wallet),goal);
}
```



Notice I am just using functions here

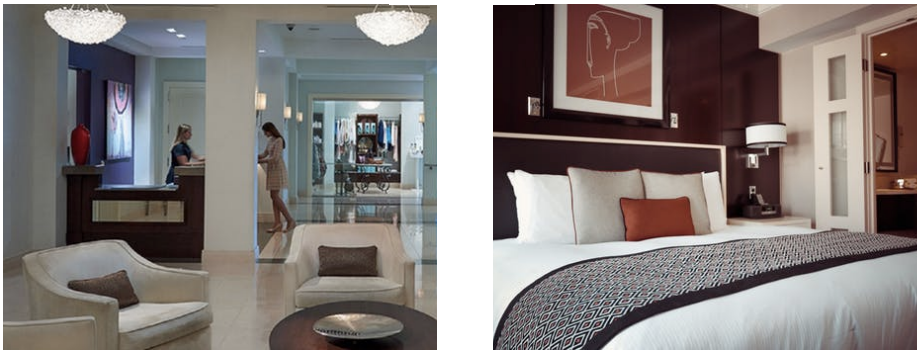
No objects are being created

USING WHAT I mentioned above:

1. Solving a real-world problem **or** trying to represent a real-life scenario
    a. **In object oriented programming, we represent the world around us as objects**
    b. **We are therefore using objects to represent the world around us**
        i. **Objects, just like the real world, have characteristics (like a color or age) and functionality (the ability to walk or talk).**
    c. **We are therefore solving our real-world problem or representing a real-life scenario with objects**

Example: Representing a hotel

Real world:



Objects in the real world:

Possible program using the above objects: Create a program to monitor the overall function of a hotel (like guests checking into the hotel, maintenance etc). Make sure the daily operations stay within budget.

2. Use the computer as a tool to help us do this. In order to use the computer as a tool, we need to "speak" to it using one of many programming languages (like C++)
    a. Languages that support the object-oriented paradigm (like C++ or Java) support the ability to use objects. This means that a language like C++ actually has built in features that allow us to represent the world as objects (discussed below). Other languages, like C, do not naturally have these features built in (meaning we could try to do object-oriented style programming, but it might be unnatural and difficult).

The fundamental concepts that support this object-oriented paradigm are: encapsulation, inheritance and polymorphism

1. Encapsulation:

We can roll up functionality and characteristics into an object. An important aspect of this idea is deciding what functionality and characteristics we want to "share with the world" vs keep only visible to the object itself.
If a human represents encapsulation: we would keep to ourselves our lungs and heart (and the functionality of those organs) and we would make visible to the world our faces and the functionality of walking.

Idea: Encapsulation
Some built-in features in C++ to support this idea (not an exhaustive list): classes with member variables and member functions. We can decide to make these member variables and functions public (available to the world) or private (hidden from the world)

2. Inheritance:

We can include some (or all) of the properties and functionalities of one object in another object.

Idea: Inheritance
Some built in features in C++ to support this idea (not an exhaustive list): the keywords public, private and protected that allow us to get features from a base class

3. Polymorphism:

Different objects can respond differently to the same message, depending on the environment. An object that inherits from another object (in order to include certain properties from that original object) can also define its own unique behavior.

Idea: Polymorphism
Some built in features/abilities in C++ to support this idea (not an exhaustive list): method overloading, operator overloading, method overriding

---

GUIs:

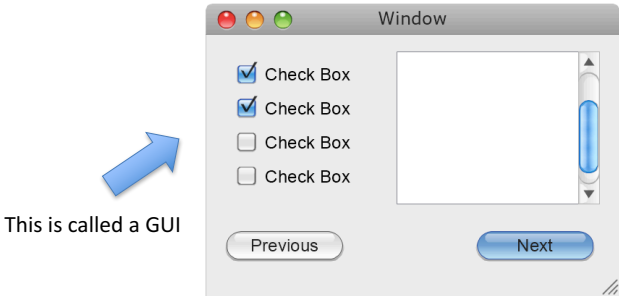
Note-a portion of this is from 1320 so I mention C programs instead of C++-the overall concept about GUIs is the same

GUI

- A GUI is graphical user interface
- Instead of having a program that just looks like text (what we have done so far), our programs will have buttons, be clickable etc.
  - We are more used to our programs looking like this

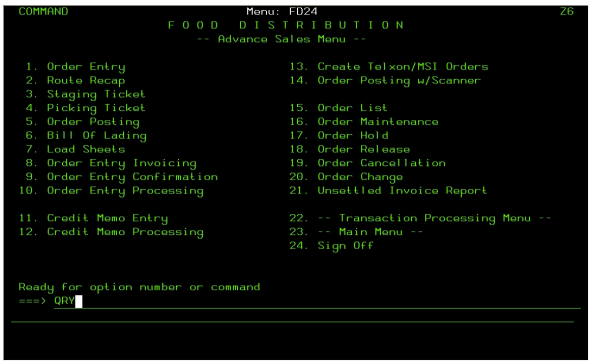
GUI

- In this class, every time we run our program, our output is text
  - We don't create programs that look like this when we run them:



GUI

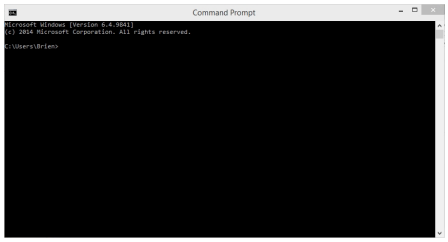
- There was a time GUIs were not used
- Computers looked more like this (similar to the type of programs we have been coding in this class):



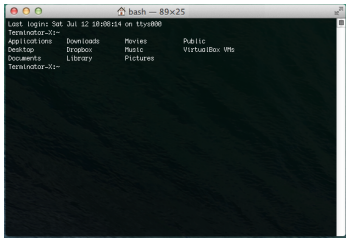
Actual example 1 (your computer Desktop):

Windows Desktop (GUI)

Mac Desktop (GUI)

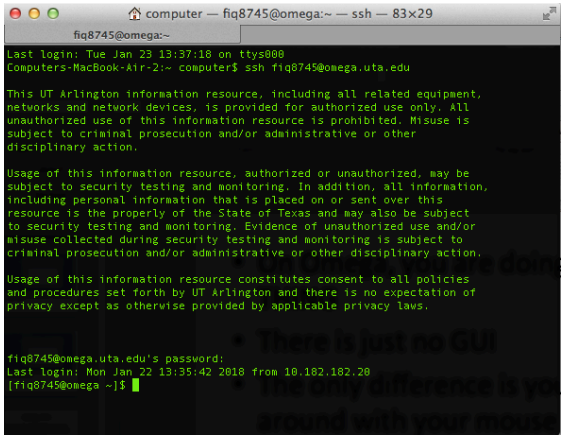


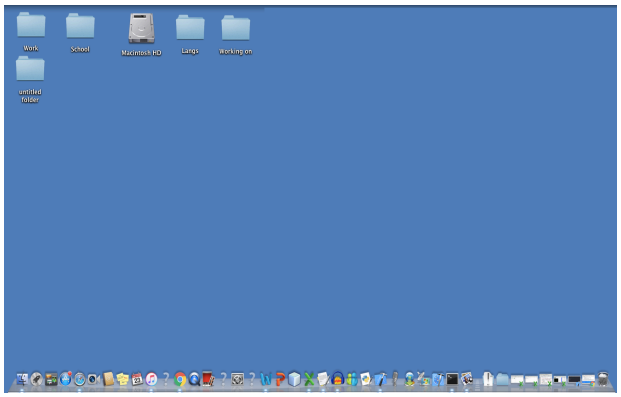
You can use **Command Prompt** on Windows to navigate your computer



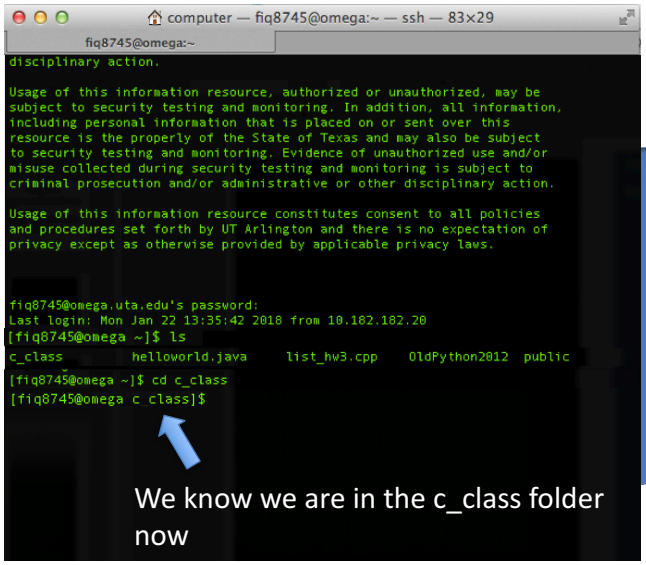
You can use **Terminal** on Mac to navigate your computer

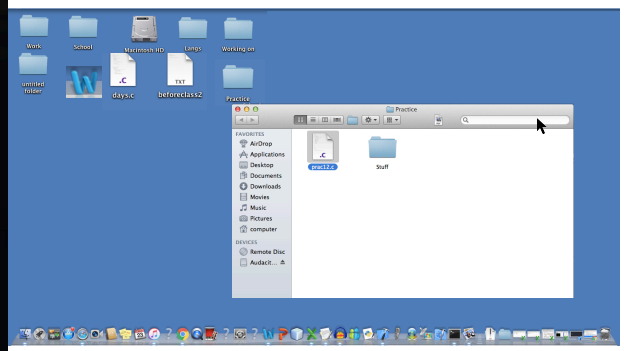
Actual example 2 (Omega desktop vs Computer desktop w/GUI):



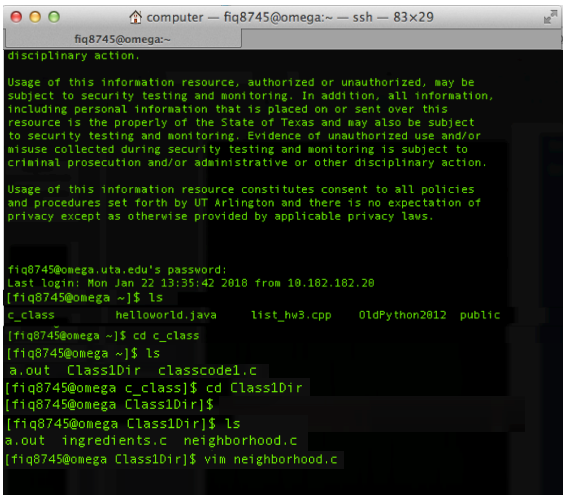


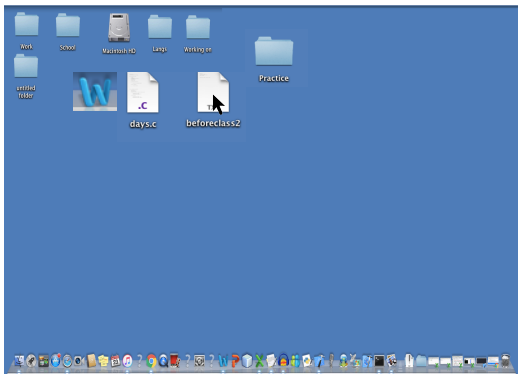
Notice that I can't see everything on my "Omega desktop" like I can on my computer desktop. I would have to use the command ls to see what I have (next slide).





It is the same idea as when I clicked on the folder on my desktop to open a folder. Once again, notice I can't see what is inside-I would need to type ls to see what is inside



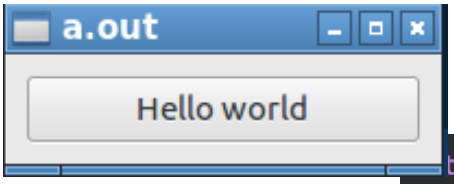


I want to open the file called neighborhood.c. I select the program to open it (the editor VIM). Remember, on my desktop I could just click the file and it would open in the program. Here I have to specify which program first.

IMPORTANT: To open or use a file, you MUST be in the folder with the file (or have the correct path).

When you add a GUI to your program, you are not modifying the actual way the program works conceptually. You are just changing the way that users interact with it.

- For example, instead of just using text, they can now use the mouse
- I always like to say we're putting a GUI *on top of the program* (since the program itself is still going to perform the concept we had originally)
    - Obviously, the way we code it will be different, but the outcome will be the same



We can now put a GUI on it

```
t main(int args, char ** argv)
{
  cout<<"Hello world"<<endl;
}
```

This is what we have been doing

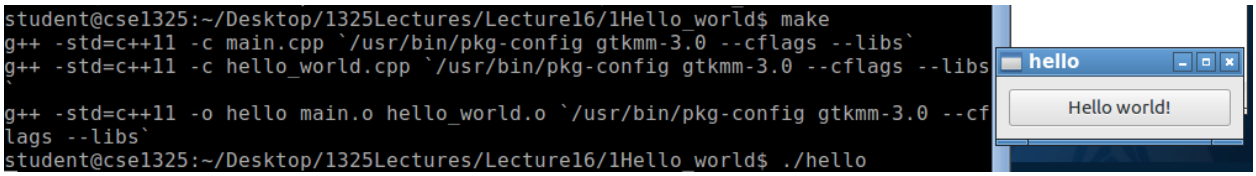
Examples:

Note: I am using the virtual machine for all these examples. Main concepts for today include:

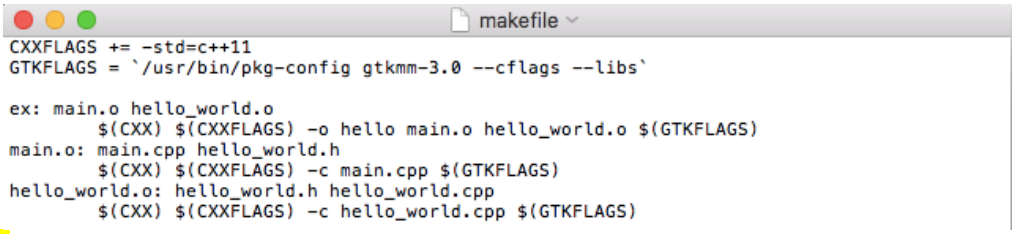
- ***Looking up information***
  - *There is a lot of documentation for gtkmm online (the official C++ interface for the GUI library GTK+)*
    - *https://www.gtkmm.org/en/ (more info about gtkmm)*
    - *For example, you can look at the Window class and all functions available in it when you inherit from it (I just typed in Google: window class gtkmm): https://developer.gnome.org/gtkmm/stable/classGtk_1_1Window.html*

- ***Understanding that we are using the same concepts we previously learned***
  - *Inheritance, creating objects and using these objects to access functions in the class, scope resolution operator for access*

Program 1:



Notice that the title of the window (hello) is the same name we used to run the program



hello_world.h

```
#ifndef HELLOWINDOW_H
#define HELLOWINDOW_H

#include <gtkmm.h>

/*I am inherting from Gtk::Window (you can look this up online like I did in class)*/
class Hellowindow : public Gtk::Window
{

public:
 Hellowindow();
 virtual        ~Hellowindow();

protected:
 /*signal handlers (when the button is clicked, what functionality do I want to occur?)*/
 void           on_button_clicked();
```

```
 /*What will I have in my window?  A button(called a widget)-we create a Button object (once again, you can look up online Gtk::Button)*/
 Gtk::Button          button;
};

#endif
```

hello_world.cpp

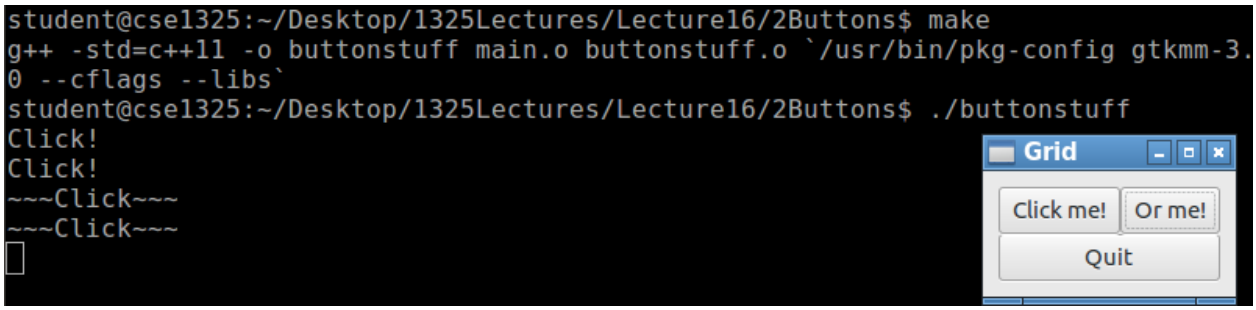
```
#include "hello_world.h"

/*The constructor for Hellowindow.  I am basically saying that whenever I create a Hellowindow object, I am making the little Hello world! button inside.*/
Hellowindow::Hellowindow()
{
 /*The physical characteristics of my window-remember, you can see these functions yourself by looking up the Window class we inherited from*/
 resize(200,50);
 set_border_width(10);

 /*Physical characteristics of my button-what it looks like (with Hello world! written on it)*/
 button.add_label("Hello world!");
/*Functionality of my button-what I want it to do when I click on it*/
 button.signal_clicked().connect(sigc::mem_fun(*this,&Hellowindow::on_button_clicked));

/*We're done making the button, we now go ahead and add it to the window*/
 add(button);
/*We need this extra step of letting our button be visible to the world*/
 button.show();
}


Hellowindow::~Hellowindow()
{}

/*Defining the functionality of the button-we used it when it creating the button, now we're defining it*/
void Hellowindow::on_button_clicked()
{
 std::cout << "You have clicked the Hello World! button." << std::endl;
}
```

main.cpp

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

```
int main(int argc, char* argv[])
{
 /*this starts our Gtk */
   Gtk::Main app(argc, argv);

 /*remember, by creating a window we are creating the button inside (since it was part of the constructor)*/
   Hellowindow w;

 /*we're now running it*/
   Gtk::Main::run(w);
   return 0;
}
```

Program 2:



Click! Is printed to screen when I click the button *Click me!*
~~~Click~~~ is printed to screen when I click the button *Or me!*

buttonstuff.h

```
#ifndef BUTTONS_H
#define BUTTONS_H

#include <gtkmm.h>

//see previous program for comments-same thing going on here
class Button_window : public Gtk::Window
{

public:
```
~~~

```
 Button_window();
 virtual         ~Button_window();

protected:

 void            on_button_1();
 void            on_button_2();

 Gtk::Button         button1;
 Gtk::Button         button2;
 Gtk::Button         quit;
 Gtk::Grid           grid; /*we now have something called a grid-since we have multiple buttons, we can
organize them in something called a grid*/
};

#endif
```

buttonstuff.cpp

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


/*see previous program for comments-same thing*/
Button_window::Button_window()
{

 set_title("Grid"); /*Here, I am setting the title of the window*/
 this->set_border_width(10); /*just showing you that you can use this-> also */


 button1.add_label("Click me!");
 button1.signal_clicked().connect(sigc::mem_fun(*this,&Button_window::on_button_1));
 grid.attach(button1,0,0,1,1); /*you can look up what the numbers mean here (look at the attach
function in the Grid class).  it talks about the location of the button in the grid- first two numbers are
the column and row, last two numbers are how many columns and rows it will span):
https://developer.gnome.org/gtkmm/stable/classGtk_1_1Grid.html#a9c425e95660daff60a77fc0cafc18
115 NOTE THIS WEBSITE IS  A GREAT REFERENCE FOR LOOKING UP THE CLASSES AVAILABLE TO
YOU!!!!*/

 button2.add_label("Or me!");
 button2.signal_clicked().connect(sigc::mem_fun(*this,&Button_window::on_button_2));
 grid.attach(button2,1,0,1,1);

 quit.add_label("Quit");
 quit.signal_clicked().connect(sigc::mem_fun(*this,&Button_window::close));
 grid.attach(quit,0,1,2,1);

 grid.show_all();
```

```
 add(grid);
}


Button_window::~Button_window()
{}

void Button_window::on_button_1()
{
 std::cout << "Click!" << std::endl;
}


void Button_window::on_button_2()
{
 std::cout << "~~~Click~~~" << std::endl;
}
```

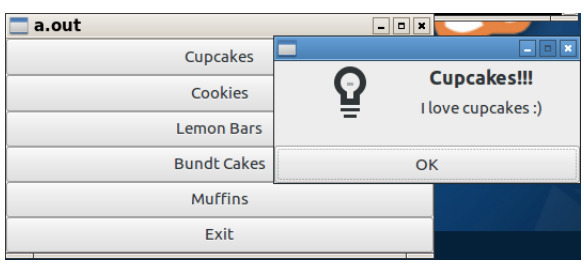
main.cpp

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


int main(int argc, char* argv[])
{
   /*using a smart pointer (different one from class, but same idea) to start gtkmm:
https://developer.gnome.org/gtkmm-tutorial/stable/chapter-refptr.html.en*/
   Glib::RefPtr<Gtk::Application> app = Gtk::Application::create(argc, argv, "www.uta.edu");

   Button_window w;

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

Program 3:

```
student@cse1325:~/Desktop/1325Lectures/Lecture16/3Desserts$ g++ -std=c++11  main.cpp
esserts.cpp `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
student@cse1325:~/Desktop/1325Lectures/Lecture16/3Desserts$ ./a.out
```

(I used the command line here, no makefile)



desserts.h

```
#ifndef DESSERTS_H
#define DESSERTS_H

#include <gtkmm.h>

/*see first program for comments*/
class Dessert_window : public Gtk::Window
{

public:
 Dessert_window();
 virtual      ~Dessert_window();

protected:
 void              cupcake_info();
 void              cookie_info();
 void              lemon_info();
 void              bundt_info();
 void              muffin_info();


 Gtk::Button          cupcake;
 Gtk::Button          cookie;
 Gtk::Button          lemon;
 Gtk::Button          bundt;
 Gtk::Button          muffin;
 Gtk::Button          exit;
 Gtk::VBox             layout; /*another way to hold our buttons-we used a Grid last time, now I'm going
to use something called VBox (once again, feel free to look this up online)*/

};


#endif
```

desserts.cpp

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


/*see first program for comments*/
Dessert_window::Dessert_window()
{
 resize(400,200);

 cupcake.add_label("Cupcakes");
 cupcake.signal_pressed().connect(sigc::mem_fun(*this,&Dessert_window::cupcake_info));
```

```
 layout.pack_start(cupcake); /*think that we are packing in our buttons each time we make them on
top each other in our Vbox (that's why they are all stacked one on the other)*/


 cookie.add_label("Cookies");
 cookie.signal_pressed().connect(sigc::mem_fun(*this,&Dessert_window::cookie_info));
 layout.pack_start(cookie);

 lemon.add_label("Lemon Bars");
 lemon.signal_pressed().connect(sigc::mem_fun(*this,&Dessert_window::lemon_info));
 layout.pack_start(lemon);


 bundt.add_label("Bundt Cakes");
 bundt.signal_pressed().connect(sigc::mem_fun(*this,&Dessert_window::bundt_info));
 layout.pack_start(bundt);

 muffin.add_label("Muffins");
 muffin.signal_pressed().connect(sigc::mem_fun(*this,&Dessert_window::muffin_info));
 layout.pack_start(muffin);


 exit.add_label("Exit");
 exit.signal_pressed().connect(sigc::mem_fun(*this,&Dessert_window::close));
 layout.pack_start(exit);

 layout.show_all(); /*when done packing buttons, we show them*/
 add(layout); /*then add the whole thing to our window*/

}


Dessert_window::~Dessert_window()
{}

/*defining the functionality when we click our buttons*/
void Dessert_window::cupcake_info()
{
 Gtk::MessageDialog dialog(*this, "Cupcakes!!!",false,Gtk::MESSAGE_INFO); /*we can create a message
dialog box from the MessageDialog class.  We're basically saying that when we click the cupcake button,
a message dialog box will be created (unlike before, where we just printed to screen). */
 dialog.set_secondary_text("I love cupcakes :)"); /*check online for the functions available to you in
your message dialog box*/
 dialog.run();
}

void Dessert_window::cookie_info()
{
 Gtk::MessageDialog dialog(*this, "Warning-you pressed cookie! (this is a warning message
dialog)",false,Gtk::MESSAGE_WARNING); /*there are different types of message dialog boxes-the
fourth parameter in the constructor is letting you know this is a warning message dialog box.  check
```

```
online for what MESSAGE_WARNING (and MESSAGE_INFO from the first one) mean-you can look up
the MessageDialog class online)*/
 dialog.run();
}

void Dessert_window::lemon_info()
{
 Gtk::MessageDialog dialog(*this, "Do you like lemon
bars?",false,Gtk::MESSAGE_QUESTION,Gtk::BUTTONS_YES_NO); /*remember we looked up in class the
BUTTONS_YES_NO-it was an enum*/
 int Answer=dialog.run(); /*so based on whatever this function returned…*/

 switch(Answer) /*…we can do something to respond*/
 {
 case(Gtk::RESPONSE_YES):
 std::cout << "Yes!" << std::endl;
 break;
 case(Gtk::RESPONSE_NO):
 std::cout << "No." << std::endl;
 break;
 default:
 std::cout << "Corner button clicked." << std::endl;
 break;
 }
}

void Dessert_window::bundt_info()
{
 Gtk::MessageDialog dialog(*this, "Bundt error (this is a warning message dialog
box)",false,Gtk::MESSAGE_ERROR);
 dialog.run();
}

void Dessert_window::muffin_info()
{
 Gtk::MessageDialog dialog(*this, "<u>You chose the</u> <b>muffin button!</b>
<i>Muffin!!!</i>",true,Gtk::MESSAGE_OTHER);
 dialog.set_secondary_text("Notice you can use <b>HTML</b>!!!.",true);
 dialog.run();
}
```

main.cpp

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

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

 Dessert_window w;
```

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