CSE 1325: Object-Oriented Programming

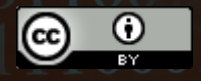
Intro to Graphical User Interfaces and gtkmm

Mr. George F. Rice

george.rice@uta.edu

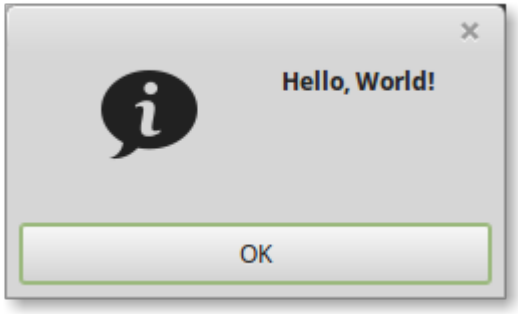
Based on material by Bjarne Stroustrup
www.stroustrup.com/Programming

ERB 402
Office Hours:
Tuesday Thursday 11 - 12
Or by appointment



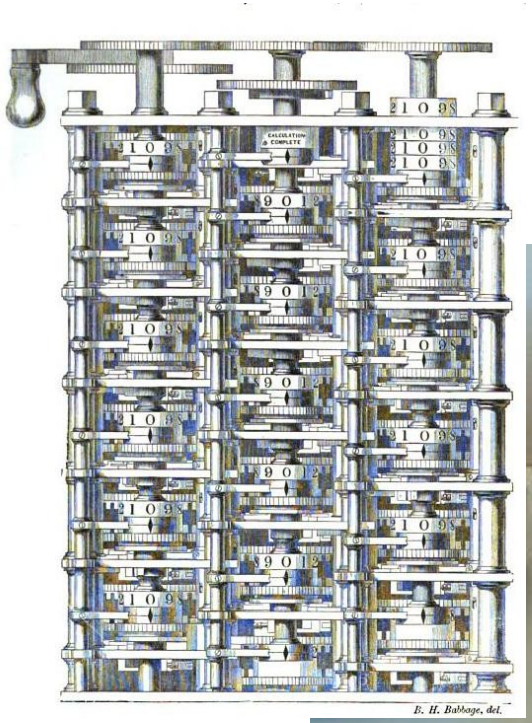
Overview: Intro to GUI and Agile Processes

- Graphical User Interfaces (GUI)
  - History of User Interfaces
  - Principle of Least Astonishment
  - Why GUI?
  - Common Widgets
  - Selecting a GUI Library
- The Façade Pattern



Question

- When was the first user input device deployed for automated computing machines?



Paper Tape

- Early user interfaces were based on paper tape…
  - Paper tape (with 5, 6, 7, or 24 holes per row) date back to **1725** for automating weaving looms
  - First used in 1846 to prepare telegrams for transmission
  - Punch machine separate from reader
  - Excellent confetti generator!

Photo by Bad Germ CC-BY-SA-3.0 Unported

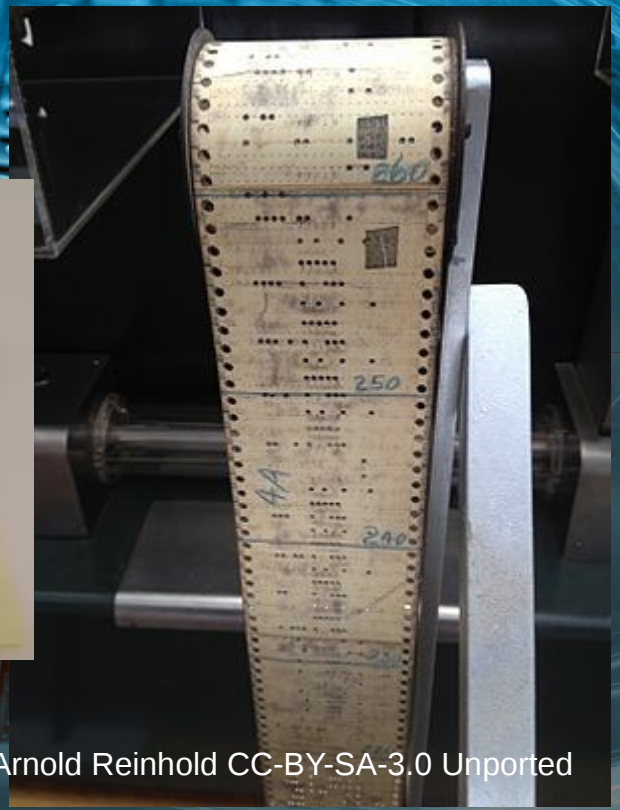
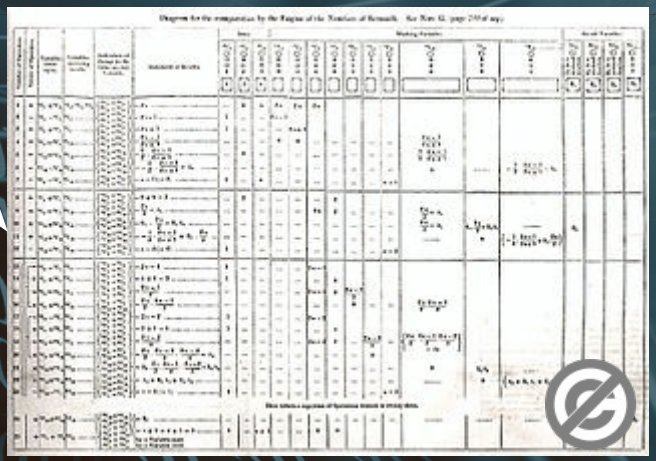


Photo by Arnold Reinhold CC-BY-SA-3.0 Unported

Punch (Hollerith) Cards

- … and punch cards aka Hollerith cards
  - Cards with 80 columns, often driving looms and musical instruments, starting in 1832
  - Charles Babbage proposed use of cards for the Circulating Engine Store in the Analytic Engine
    - Lady Ada Lovelace's algorithm to produce Bernoulli numbers was the first computer program... designed for cards
  - Cards were used to calculate the 1890 census



Fortran code: photo by Arnold Reinhold CC-BY-SA-2.5 Generic

Switches and Tubes

- Digital computers supported physical switches and lights or Nixie tubes on the front panel
  - To boot the computer, just load the program via switches
  - Debug by watching light patterns

PDP-8 photo by Florian Schäffer CC-BY-SA-4.0

Georg-Johann Lay Gnu FDL 1.2+

PDP = Programmable Data Processor

Keyboards and Printers / CRTs

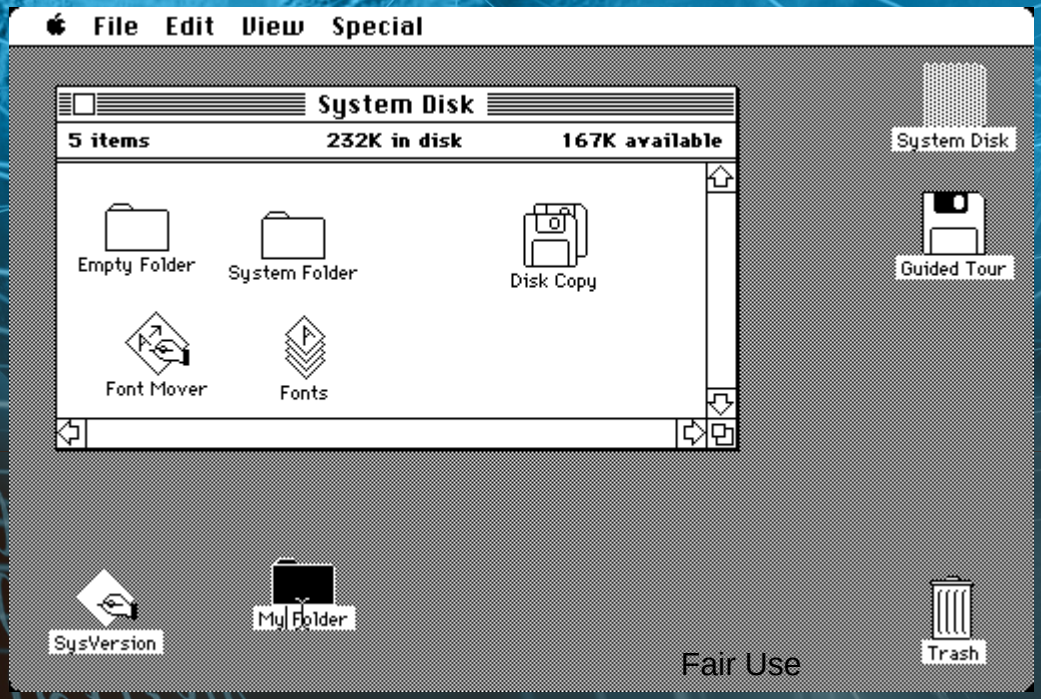
- Keyboards and thermal printers or displays brought the Command Line Interface (CLI)

Dave Fischer CC-BY-SA-3.0

Jason Scott CC-BY-SA-2.0 Generic

Graphical User Interfaces

- In 1984 began the rise of the Graphical User Interface
  (Windows Icon Mouse Pointer, or WIMP)

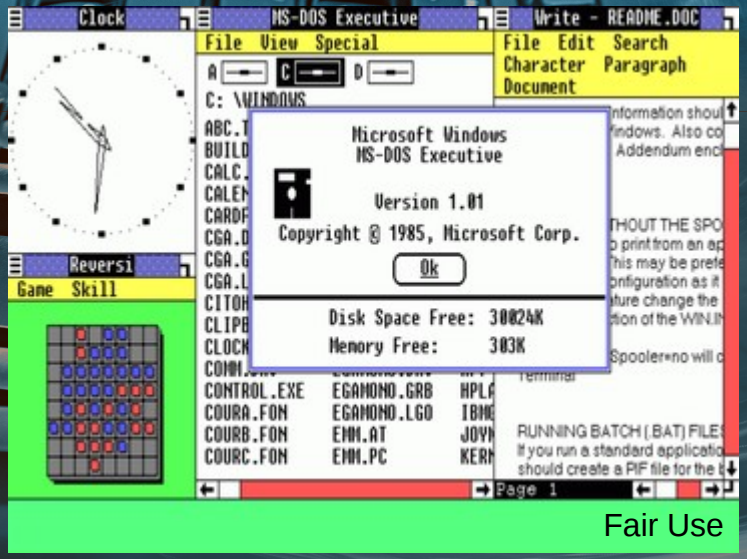


Fair Use

Apple Macintosh OS 1.0 Desktop

Fair Use

Xerox Star Compound Document

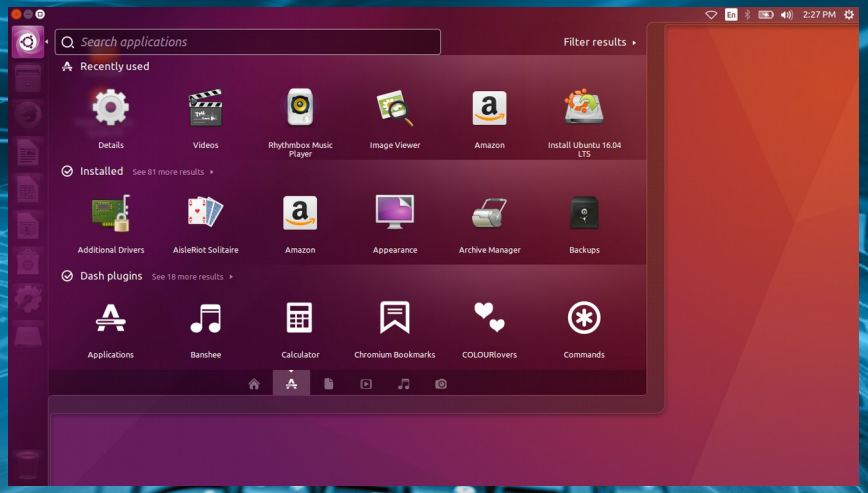


Microsoft Windows 1.01 →

Fair Use

Modern Graphical User Interfaces

- GUIs matured over time into “push button software” with rich mouse interaction



Fair Use

Ubuntu Linux 16.04 Desktop

Fair Use

Apple Macintosh OS X 10.11 Desktop

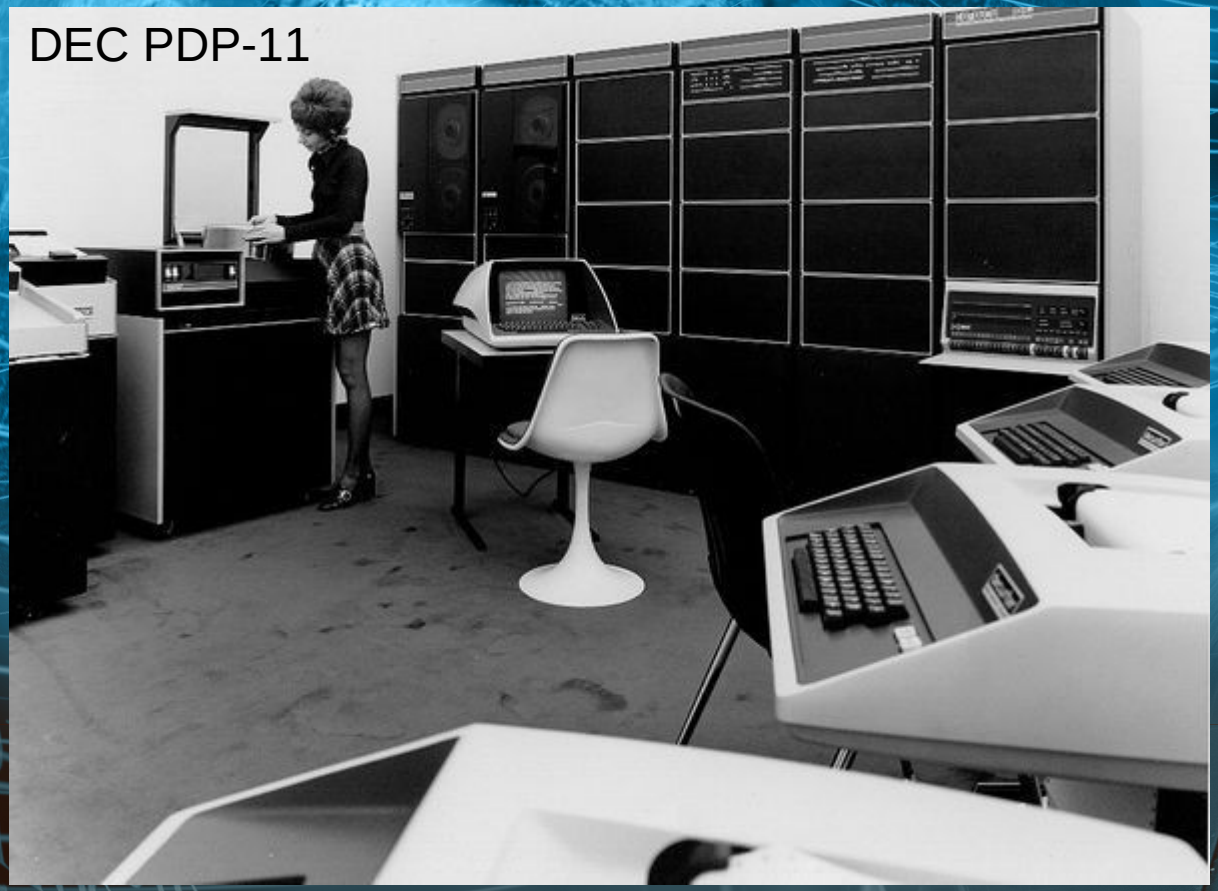
Microsoft Windows 10 →

Fair Use

Input from Persistent Memory

Disk Packs 1965

DEC PDP-11



Fair Use

8” 1971 80K 5¼” 1976 360K

Floppy Disks

3½” 1986 1.44M

Public Domain

https://en.wikipedia.org/wiki/Floppy_disk#/media/File:Floppy_disk_2009_G1.jpg

CF 1994 8M

Flash

Fair Use

Then the reign of the desktop was threatened by newer, more nimble devices...

Voice User Interfaces

- In 1996, Phillips introduced Voice Dial™, an early Voice User Interface (VUI)

- In 2011, Apple announced the Siri general VUI

In 2014, Amazon introduced the Alexa Echo family

Until recently, VUIs have acted as a complement to other user interfaces rather than as a primary interface

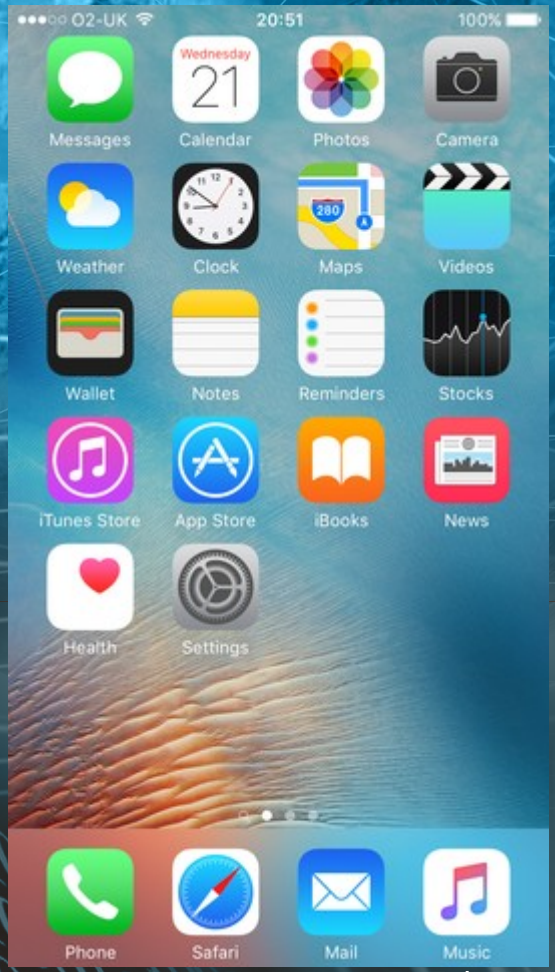
Touch / Gesture User Interfaces

- In 2006, the Touch Interface...
  (also called the Gesture Interface)

Fair Use

Modern Touch User Interfaces

- ...which has also evolved



Fair Use

Apple iOS 9 Home Screen

Apache AOSP

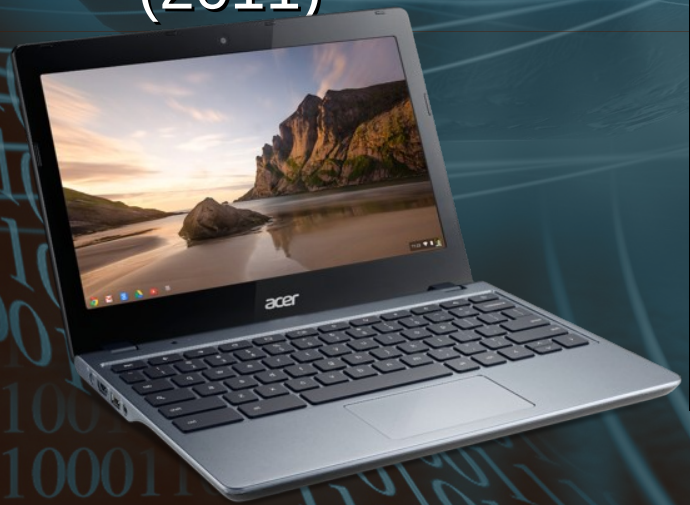
Android 7 Nougat Home Screen

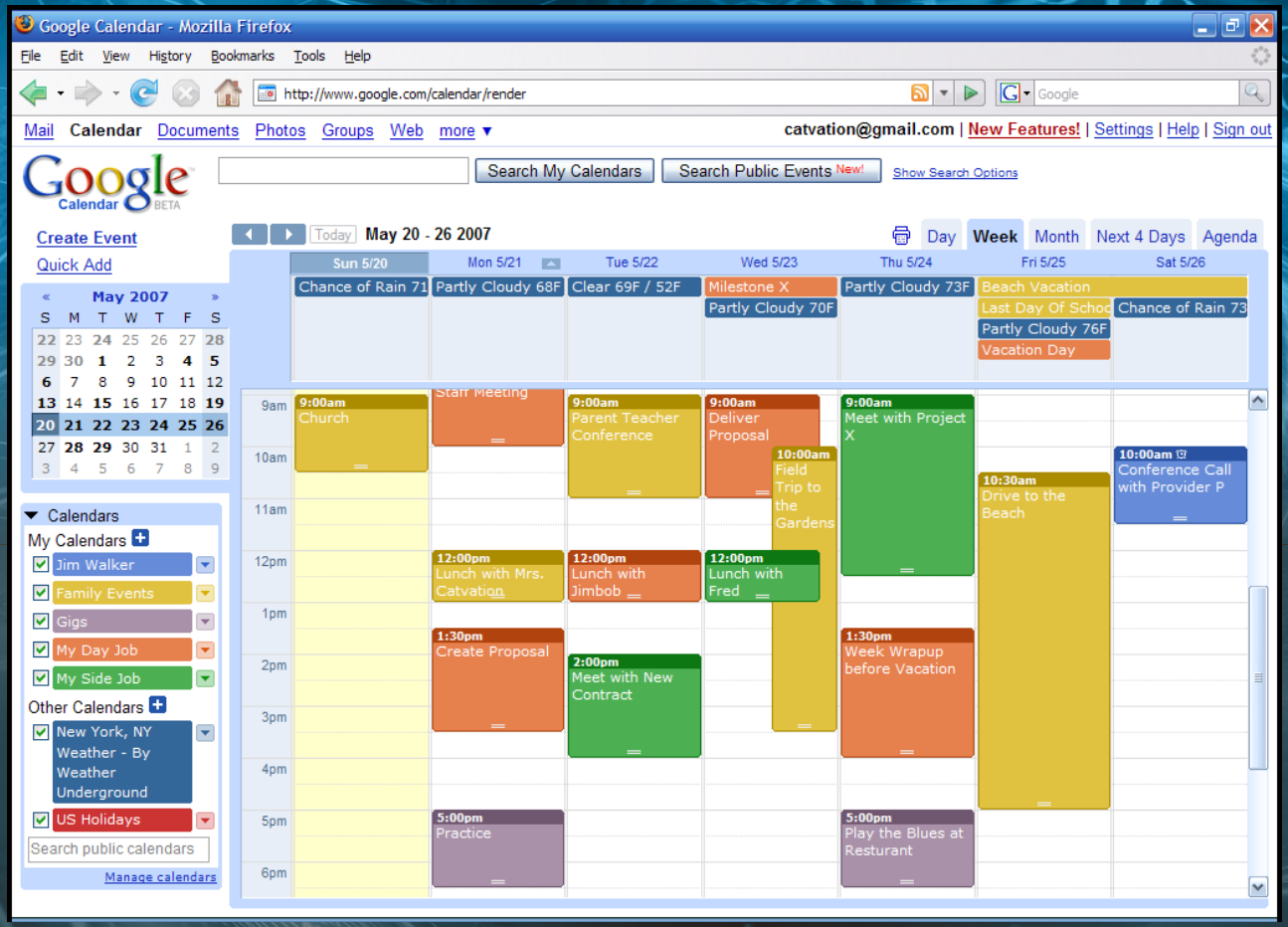
Seeking the Ultimate User Interface

http://xkcd.com/1367/

Modern Web User Interfaces

- In 1995, with the introduction of JavaScript, web applications began to evolve... slowly
- XMLHttpRequest (1999)
- AJAX (2005)
- HTML 5 and Chromebooks (2011)

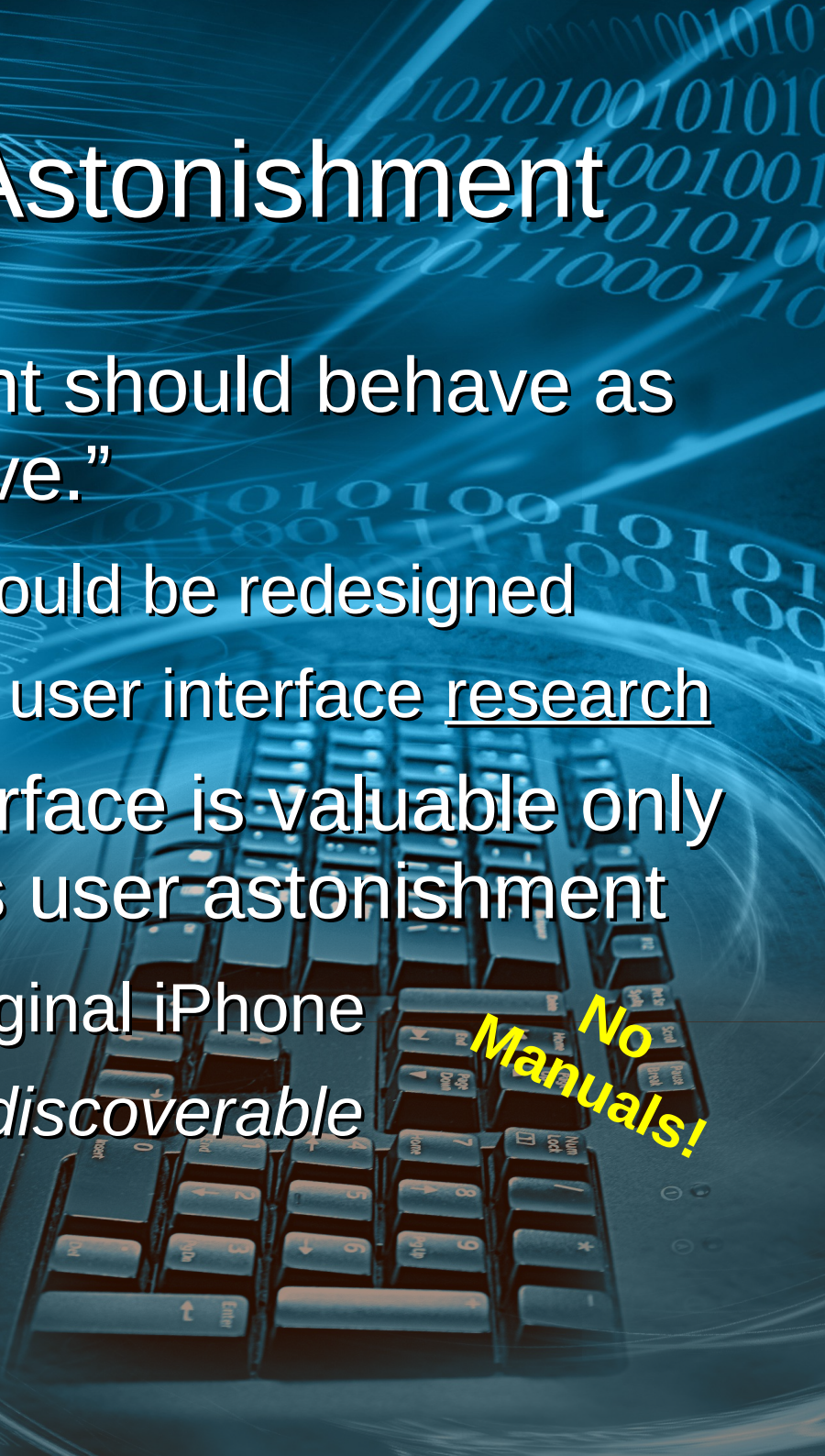




Google Calendar on Firefox Browser

Principle of Least Astonishment

- "A user interface component should behave as the users expect it to behave."
  - If it behaves otherwise, it should be redesigned
  - The natural consequence is user interface research
- Corollary: A novel user interface is valuable only to the extent that it reduces user astonishment
  - A good example was the original iPhone
  - A good interface should be *discoverable*



So... What UI is “Best”?

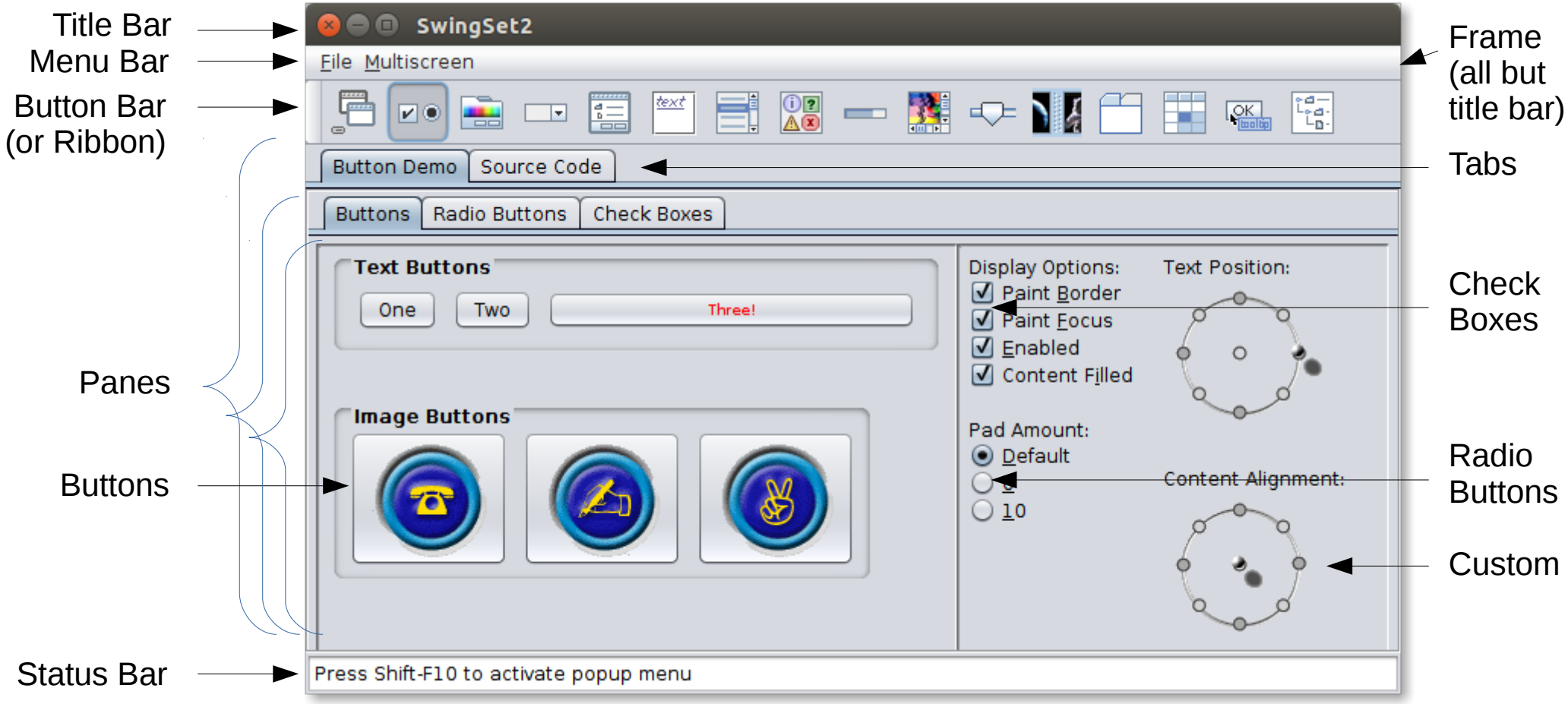
- Define “best”!
- For “home projects”, write to what you use
  - Usually between GUI, iOS, Android, or web app
  - MVC keeps your “business logic” separate for porting
- For commercial apps, (at least) 3 strategies to start
  - 1 - iOS users will pay for apps – so start there
  - 2 - Use a portable desktop framework (FLTK) and seek sales
  - 3 - Write web apps first (smartphone and tableau layouts) and seek profit from advertisements or subscriptions
- This is a business decision – Your Business May Vary!

Why bother with GUI and graphics?

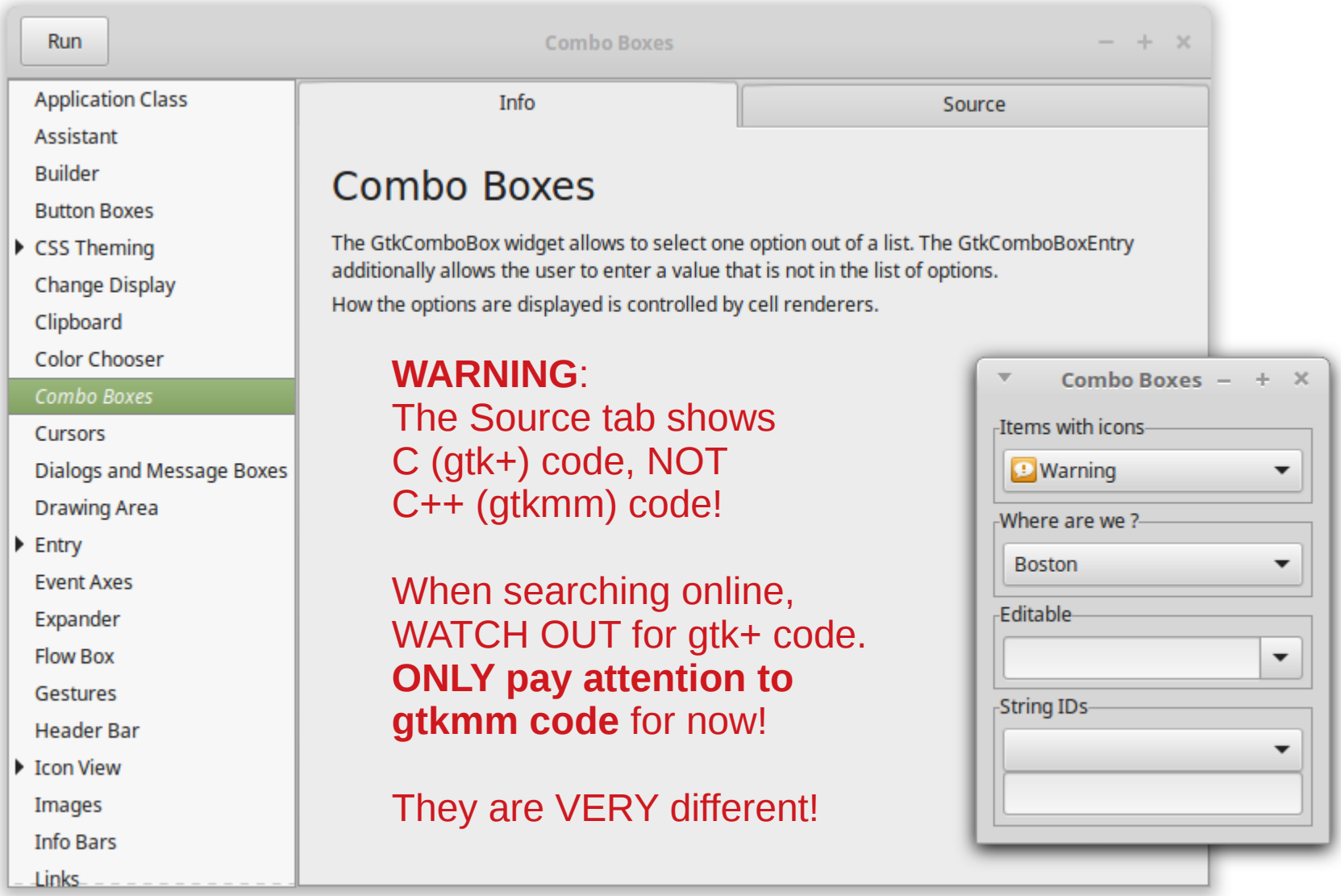
- It's very common
  - If you write conventional PC applications, you'll have to do it
- It's useful
  - What You See Is What You Get ("whizzy-wig")
  - Instant feedback
  - Graphing functions and visual metaphors
  - Displaying results
- It illustrates useful concepts and techniques
  - Deep, rich class hierarchies
  - Callbacks, singletons, and factories
  - Portable, non-sequential user interfaces

Common GUI “Widgets”

- This also demos a conventional desktop layout
  (although one created in Java with Swing)



Using gtk3-demo



Philosophy: CLI vs GUI

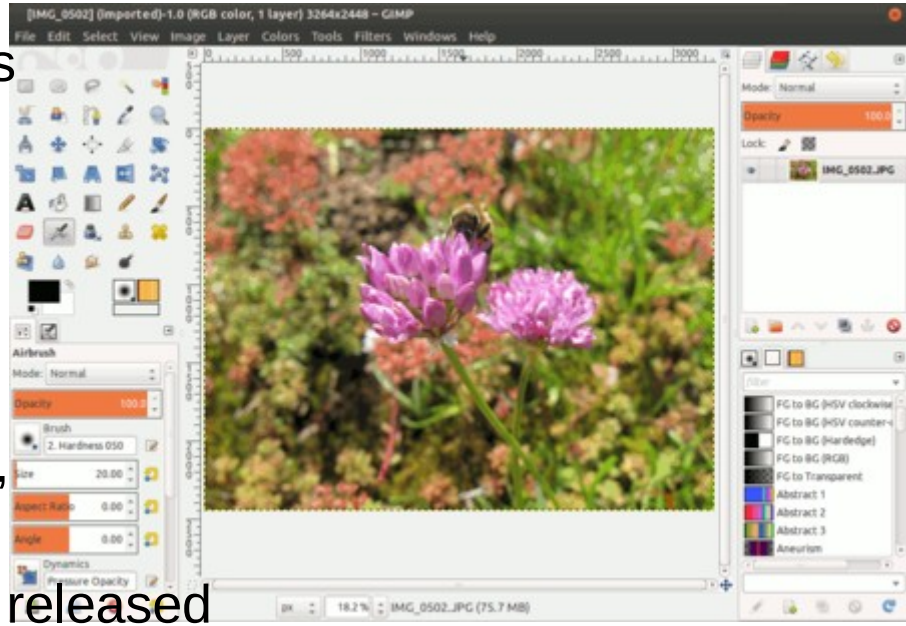
- CLI: The **program** is in control
  - Offers menus and accepts selections
  - Asks questions and accepts answers
  - Limited, keyboard-centric choices
- GUI: The **user** is in control
  - Menus, buttons, and components all available
  - Extensive, mouse or touch-centric choices
  - Click, drag, multi-touch or video gesture, shortcut keystroke, and voice control simultaneously available

Some Popular C++ UI Libraries

- For CLI
  - Built-in: iostream
  - 3rd Party: Gnu readline, CLI toolkit, CLAP / TCLAP
- For GUI
  - Built-in: None
  - 3rd Party native: Windows Foundation Classes (WFC), Windows Template Library (WTL)
  - 3rd Party cross-platform: Qt, GTK+ / **gtkmm**, wxWidgets, FLTK (in textbook)
  - 3rd Party web: CppCMS, Wt

GIMP Tool Kit + (Gtk+)

- Gtk+ is a popular cross-platform GUI toolkit written in C
  - 1995: The GIMP, a free high-end graphics program, was released
  - 1997: Its key graphics primitives were factored out, named the GNU Tool Kit (Gtk), and used to implement GIMP 0.60
  - 1998: The Gtk library was rebuilt using an object-oriented paradigm (in native C), renamed Gtk+, and used for GIMP 0.99
  - 2000: A C++ wrapper named gtkmm was released



- Gtk+ is widely used for desktops (GNOME, Unity, Cinnamon, MATE, and Xfce) as well as all GNOME applications
  - gtkmm is used for applications such as Inkscape and VMware Workstation
- Gtk+ is licensed under the GNU Library General Public License

https://gtkmm.org/en/

Why gtkmm?

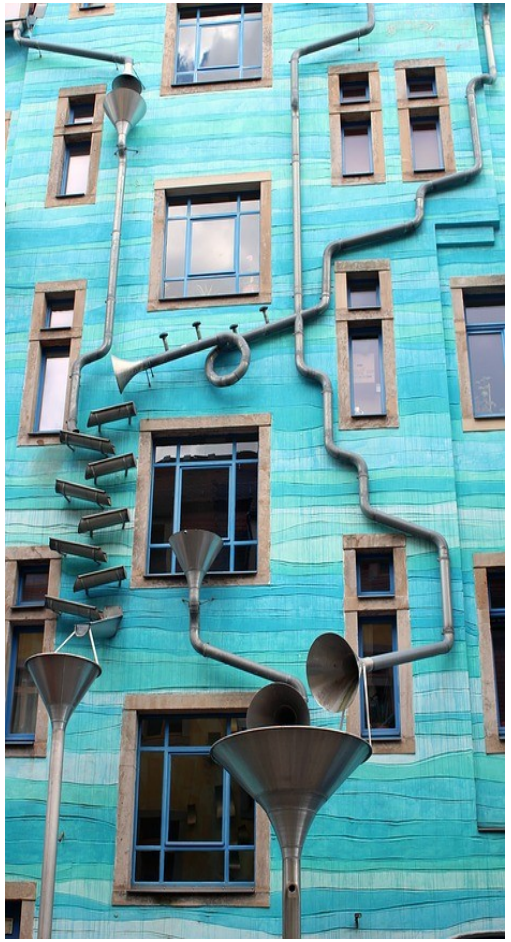
- Must use native C++ object-oriented capabilities
  - “Real” classes with “real” inheritance among widgets
  - “Real” constructors
- Must be widely used in industry for “real” applications
  - Wide use implies (but doesn’t prove) fewer bugs
  - Looks much better on a resume! :-)
- Must have excellent on-line resources
  - Searchable documentation
  - Answer base in Stack Overflow et. al.
- Must be portable to major OS platforms – Linux, Windows, Mac
- Must be simple enough to learn to use in practical applications in only 3½ weeks!

Why Not Something Else?

- Fast Light Tool Kit (FLTK)
  - Used in the textbook, though with a rather extensive facade
  - Rarely used in the “real world”, thus few on-line resources
  - Most common student feedback: “Don’t use FLTK again!”
- Qt
  - Widely used and native C++
  - Implemented early, and thus custom implementation of some features, thus redundant with a lot of subsequently standardized C++
  - Relatively complex even for simple programs
- Anything Else (that’s reasonably popular)
  - wxWidgets is popular, native C++, and a reasonable choice – but rather complex for simple programs
  - Tk is native C and quite dated at this point
  - <Your Favorite Goes Here>

The Ins and (Mostly) Outs of the Façades

- I've mentioned that Dr. Stroustrop's textbook uses a façade named std_lib_facilities.h
  - This attempts to “simplify” learning C++
  - We've carefully avoided using this in class
  - You're experienced enough for “raw C++”
- The textbook also uses a façade for the GUI
  - We will also carefully avoid using this in class
  - You're also experienced enough for “raw FLTK”, although FLTK is significantly tougher to chew
  - Due to popular demand, however, we won’t cover raw FLTK *in class,* but rather, raw gtkmm



But what is a façade, *exactly*?

Structural

Façade Pattern

- The **Façade** pattern implements a simplified interface to a complex class or package
  - The Façade class may simplify the interface by making good default assumptions or eliminating rarely used options, reduce dependencies on external libraries, or reorganize the interface for clarity or familiarity

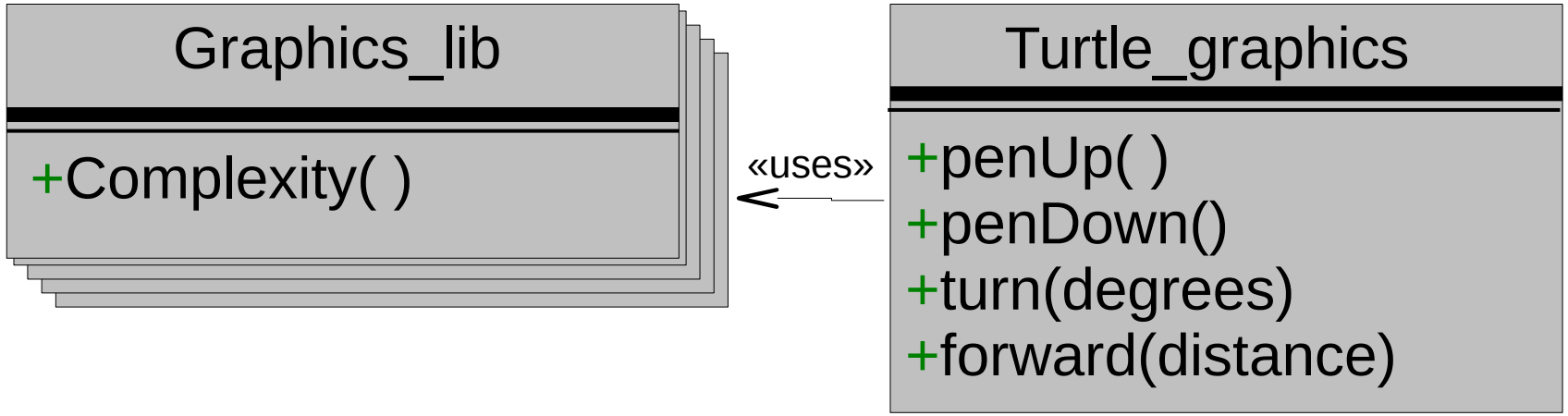
Structural

The Façade Pattern

(Slightly Simplified)

Complex class library

Very simple Façade



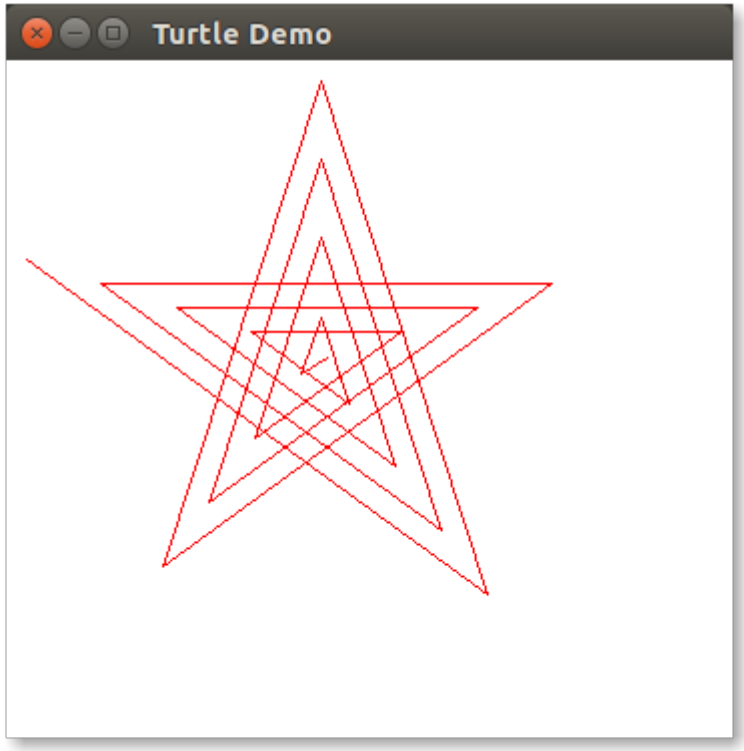
A Turtle Graphics Façade

```
class Line {    // Define a line from (x1, y1) to (x2, y2)
     double _x1, _y1, _x2, _y2;
   public:
     Line(double p_x1, double p_y1,double p_x2, double p_y2)
       : _x1{p_x1}, _y1{p_y1}, _x2{p_x2}, _y2{p_y2} { }
     double x1() {return _x1;}
     double y1() {return _y1;}
     double x2() {return _x2;}
     double y2() {return _y2;}
};
class Turtle_graphics {
       const double d2r = M_PI/180.0; // Degrees to radians
       double x, y, angle; // Current turtle position
       bool pen_is_down;
       vector<Line> lines;
    public:
        void penUp() {pen_is_down = false;}
        void penDown() {pen_is_down = true;}
        void turn(double degrees) {angle += degrees * d2r;}
        void forward(double distance) {
          double x2 = x + distance*cos(angle);
          double y2 = y + distance*sin(angle);
          if (pen_is_down) lines.push_back(Line(x, y, x2, y2));
          x = x2;
          y = y2;
        }
        vector<Line> get_lines() {return lines;}
};
```

gtkmm's line drawing capability
is needed for an actual implementation
(in just a few minutes...)

Drawing is Simple with Turtle Graphics

```
#include “Turtle_graphics.h”
int main() {
    Turtle_graphics g;
    for (double d=0; d<20; ++d) {
        g.forward(d * 10);
        g.right(144);
    }
}
```



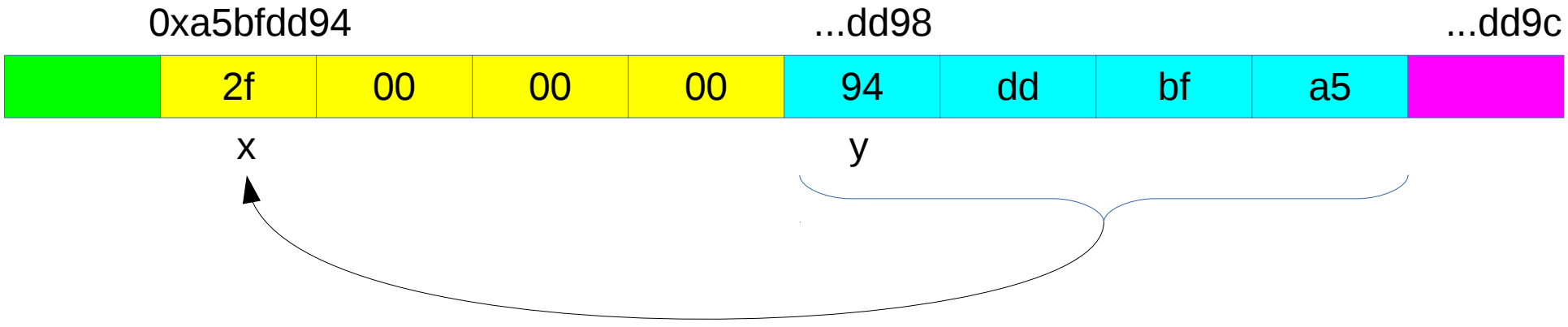
A Brief Digression

GUIs Often Rely on Pointers
So Let’s Point!

Pointers – (Likely) A Review

- A pointer is an “unsafe reference”
  - It references a memory location containing a type

```
int x = 47; // Assume the linker assigns x to address a5bfdd94
int *y;     // y contains the address of an integer – but initially, garbage
y = &x;     // Now y contains the address of x
    cout << hex << "y is at " << &y << " and points to " << y << endl;
    cout << y << " contains *y = " << *y << ", which is also x= " << x << endl;
}
```




```
ricegf@pluto:~/dev/cpp/13$ g++ pointer.cpp
ricegf@pluto:~/dev/cpp/13$ ./a.out
y is at 0x7ffca5bfdd98 and points to 0x7ffca5bfdd94
0x7ffca5bfdd94 contains *y = 2f, which is also x= 2f
```

Unlike References, Pointers can Point *Anywhere*

- References must be initialized on definition
- Pointers need not be initialized...

```
int *y;
cout << hex << "y by default points to " << y << endl;
cout << y << " contains " << *y << endl;
```

```
ricegf@pluto:~/dev/cpp/13$ g++ pointer2.cpp
ricegf@pluto:~/dev/cpp/13$ ./a.out
y by default points to 0x400860
0x400860 contains 8949ed31
```

- ...or they can be initialized to any address

```
int *y;
y = 0;
cout << hex << "y now points to " << y << endl;
cout << y << " contains " << *y << endl;
```

```
ricegf@pluto:~/dev/cpp/13$ g++ pointer3.cpp
ricegf@pluto:~/dev/cpp/13$ ./a.out
y now points to 0
Segmentation fault (core dumped)
```

Accessing the value pointed to by a pointer is called *dereferencing*

Get used to this...

Creating “New” Objects

- Pointers are often used to reference *unmanaged* variables
- “double x[3];” creates a *managed* variable
  - The memory is released when x goes out of scope
- “double *x = new double[3];” creates an *unmanaged* variable

Malloc, free, et. al. are C functions –
never use them in an OOP C++ program

  - The memory is released when you explicitly delete it
  - Delete it using “delete[ ] x;”
- Advantage: You can create an unmanaged variable (an object) inside of a constructor or method, and return without its memory being released when the scope exits

Accessing Object Members via Pointers

- You reference an object member with .
- You reference a pointer's object member with ->

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

class Foo {
  public:
    int bar = 42;
};

int main()
{
    Foo foo1;
    cout << "Direct access  -  foo1.bar:   " << foo1.bar << endl;
    Foo *foo2;
    foo2 = &foo1;
    cout << "Pointer access - (*foo2).bar: " << (*foo2).bar << endl;
    cout << "Arrow access   -  foo2->bar:   " << foo2->bar << endl;
    Foo *foo3 = new Foo{ };
    cout << "Pointer access - (*foo3).bar: " << (*foo3).bar << endl;
    cout << "Arrow access   -  foo3->bar:   " << foo3->bar << endl;
    delete[ ] foo3;
}
```

```
ricegf@pluto:~/dev/cpp/201801/12$ make pointers
g++ -std=c++14 -o pointers pointers.cpp
ricegf@pluto:~/dev/cpp/201801/12$ ./pointers
Direct access  -  foo1.bar:   42
Pointer access - (*foo2).bar: 42
Arrow access   -  foo2->bar:   42
Pointer access - (*foo3).bar: 42
Arrow access   -  foo3->bar:   42
ricegf@pluto:~/dev/cpp/201801/12$
```

Stack vs Heap

- **Stack** is “scratch memory” for a thread
  - LIFO (Last-In, First-Out) allocation and deallocation

    - Allocation when a scope is entered
    - Automatically deallocated when that scope exits
  - Simple to track and so rarely leaks memory
- **Heap** is memory shared by all threads for dynamic allocation
  - Allocation and deallocation can happen at any time – but only when *specifically* invoked!
    - Allocate using “new” to a pointer
    - Deallocate using “delete” (“delete[]” for vectors and arrays)

“delete” vs “delete[ ]”

```
void calc() {
    int result;
    int *i = new int{};           // allocate one integer from the heap
    double *d = new double[10];   // allocate 10 doubles in an array from the heap

    // … use i and d to calculate result

    delete[] d;    // mark the heap array that d points to as free
    delete i;      // mark the integer i on the heap as free
    return result; // result automatically deallocates when it goes out of scope!
}
```

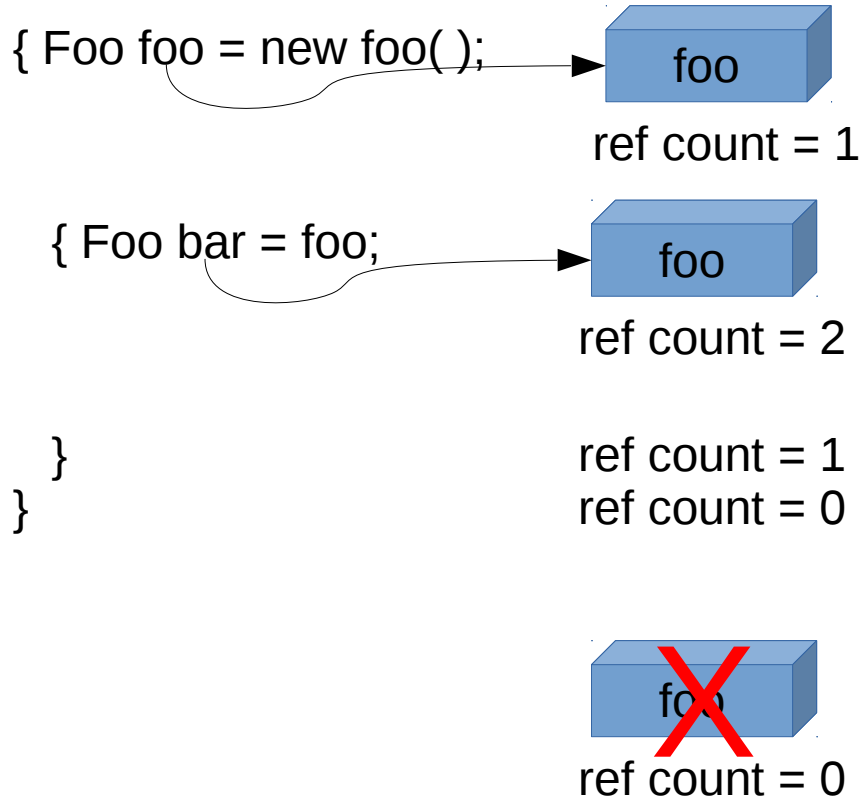
- Manually delete heap variables when you no longer need them
  - “delete” de-allocates a simple variable (like i)
  - “delete[ ]” de-allocates a vector or array (like d)
- The heap is useful for allocating memory that survives the “return” statement

Deleting Unmanaged Variables

- Problem: When do you call delete x?
  - And how can you ensure you never forget?
- Solution: In C++, it's... complicated
  - We can use “reference counted” variables via “smart pointers” (unique_ptr and shared_ptr)
    - Managed languages such as Java, Python, and C# do this (almost) always
  - We can used gtkmm’s Gtk::Manage for gtkmm objects
  - We can (and otherwise will) ignore the problem for now, as memory is freed by Ubuntu when your program exits

Reference Counting

- Java, Python, and (mostly) C# implement managed memory via reference counters
  - When an object is instanced, memory is allocated and a **“reference counter”** is set to 1
  - When a reference to that object is created, the counter is incremented
  - When a reference to that object is deleted (e.g., goes out of scope), the counter is decremented
  - When more memory is needed, the **“garbage collector”** finds all reference counters == 0, frees the associated memory, and moves all in-use objects to free up a contiguous block of memory for future use



C++ Offers Managed Memory for Pointers via Library Classes

- std::unique_ptr<T> ensures the (single referenced) memory allocated from heap is released when the managing object goes out of scope

```
{   T *p = new T{42, "meaning"};  // T is a class type that we defined earlier
    my_function(p);
    delete[] p;
}
```

my_function may throw an exception, skipping the delete[] and "leaking" memory

```
using namespace std;
{
    unique_ptr<T> p{new T(42, "meaning")};
    my_function(p);
} // p's destructor is guaranteed to run "here", calling delete
```

Even if my_function throws an exception, memory is freed when local scope exits

- std::shared_ptr<T> implements a reference counter and releases the memory when the counter reaches 0

```
{
    std::shared_ptr<T> p(new T(3.14, "pi"));
    my_object.may_add(p);
} // p's destructor will only delete the T if number_storage didn't copy
```

may_add may keep a reference to p. If so, memory is freed only when both references are deleted.

Know that these exist, but we will NOT code with them in class or on exams.

Gtk::Manage

- Gtkmm will manage nested widgets for you, deleting them when their container is deleted
  - This greatly simplifies memory management
  - Works with all gtkmm widgets
  - Just create the new widget as a parameter to the static method Gtk::manage, e.g.,
    - Gtk::Box *box = **Gtk::manage**(new Gtk::Box{ });

Your new best friend!

Once added to a container, the (gtkmm) object's destructor will be called as part of the container's destructor code.

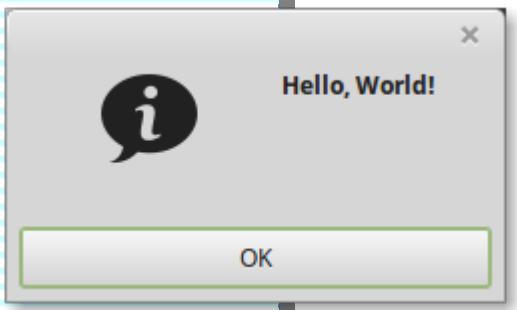
Create a new "box" object on the heap, and return its address (i.e., pointer)

Enough Pointers
Let's Write a GUI!

Writing “Hello, World” in gtkmm

Without pointers

```
#include <gtkmm.h>

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

    // Create a simple dialog containing "Hello, World!"
    Gtk::MessageDialog dialog("Hello, World!");

    // Turn control over to gtkmm until the user clicks OK
    dialog.run();
}
```



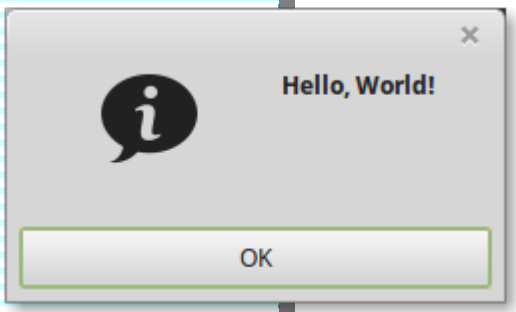
With pointers

```
#include <gtkmm.h>

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

    // Create a simple dialog containing "Hello, World!"
    Gtk::MessageDialog *dialog = new Gtk::MessageDialog{"Hello, World!"};

    // Turn control over to gtkmm until the user clicks OK
    dialog->run();
}
```

Writing “Hello, World” in gtkmm

Without pointers

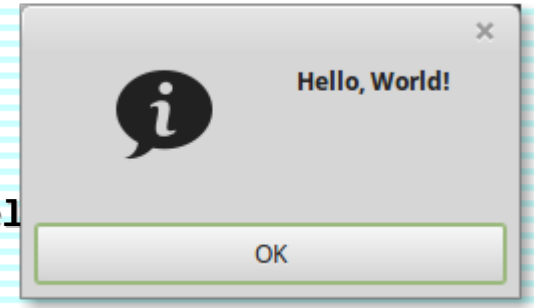
```
#include <gtkmm.h>

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

    // Create a simple dialog containing "Hello, World!"
    Gtk::MessageDialog dialog("Hello, World!");
```




```
ricegf@pluto:~/dev/cpp/201801/12$ make hello
g++ -std=c++14 -o hello hello.cpp `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
g++ -std=c++14 -o hellop hellop.cpp `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
ricegf@pluto:~/dev/cpp/201801/12$ ./hello
Gtk-Message: GtkDialog mapped without a transient parent. This is discouraged.
ricegf@pluto:~/dev/cpp/201801/12$ ./hellop
Gtk-Message: GtkDialog mapped without a transient parent. This is discouraged.
ricegf@pluto:~/dev/cpp/201801/12$
```

With pointers

```
{
    // Initialize GTK
    Gtk::Main kit(argc, argv);

    // Create a simple dialog containing "Hello, World!"
    Gtk::MessageDialog *dialog = new Gtk::MessageDialog{"Hel

    // Turn control over to gtkmm until the user clicks OK
    dialog->run();
}
```



Thoughts on “Hello, World”

- The Gtk::Main instance is traditionally called kit
  - Because it’s a tool *kit*, I suppose
  - Gtk::Main is a Singleton class – every time you “instance” it, you actually get a reference to the same object
- We will use pointers going forward
  - It’s common to need to create long-lived windows and widgets in a narrow scope (e.g., a constructor)
  - Usually created on the heap (i.e., “new” with pointers), that are not automatically deleted by C++ itself
    - Gtk::Manage takes care of gtkmm widgets (so use it!)
    - You must handle non-gtkmm-derived classes yourself

Compiling and Running gtkmm Programs

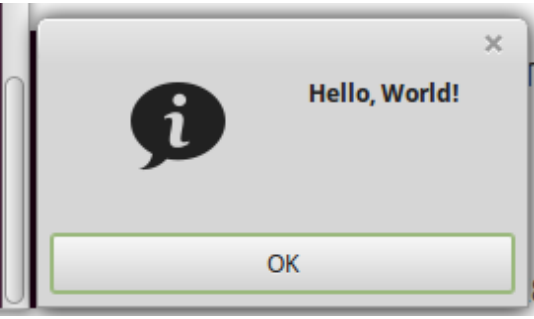
Makefile

```
# Remember to use the leading TABS, not spaces, for commands!
main: main.o
	g++ -o hello main.o `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
	./hello

main.o: main.cc
	g++ -c main.cc `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`

clean:
	-rm -f *.o *~ hello
```

*Back*tick!

The bash backtick operator simply replaces the enclosed command with its output.

```
$ make clean
rm -f *.o *~ hello
$ make
g++ -c main.cc `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
g++ -o hello main.o `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
./hello
Gtk-Message: GtkDialog mapped without a transient parent. This is discouraged.
```



Note: The “Gtk-Message” is complaining that we’re displaying a *dialog* without first displaying a *main window* (where the menus and tool bars would be). We’ll ignore this warning until we get to main windows a couple of lectures hence.

Review

A Turtle Graphics Façade

```
class Line {    // Define a line from (x1, y1) to (x2, y2)
     double _x1, _y1, _x2, _y2;
   public:
     Line(double p_x1, double p_y1,double p_x2, double p_y2)
       : _x1{p_x1}, _y1{p_y1}, _x2{p_x2}, _y2{p_y2} { }
     double x1() {return _x1;}
     double y1() {return _y1;}
     double x2() {return _x2;}
     double y2() {return _y2;}
};
class Turtle_graphics {
       const double d2r = M_PI/180.0; // Degrees to radians
       double x, y, angle; // Current turtle position
       bool pen_is_down;
       vector<Line> lines;
    public:
        void penUp() {pen_is_down = false;}
        void penDown() {pen_is_down = true;}
        void turn(double degrees) {angle += degrees * d2r;}
        void forward(double distance) {
          double x2 = x + distance*cos(angle);
          double y2 = y + distance*sin(angle);
          if (pen_is_down) lines.push_back(Line(x, y, x2, y2));
          x = x2;
          y = y2;
        }
        vector<Line> get_lines() {return lines;}
};
```

line.h
line.cpp

gtkmm's line drawing capability
is needed for an actual implementation
(now...)

Let's take a quick overview
of how to use our façade
with the gtkmm library!

Much more gtkmm to follow

turtle.h
turtle.cpp

Create a gtkmm Drawing Area...

view.h

```
#ifndef _VIEW_H
#define _VIEW_H

#include "line.h"
#include <vector>
#include <gtkmm/drawingarea.h>

using namespace std;

class View : public Gtk::DrawingArea {
     vector<Line> lines;
  public:
     View(vector<Line> L);
     bool on_draw(const Cairo::RefPtr<Cairo::Context>& cr) override;
};
#endif
```

Class View "is a" DrawingArea – it inherits members from DrawingArea

It stores the vector of lines generated by our Turtle Graphics class

It *overrides* (replaces) DrawingArea's drawing method with our method

When Ubuntu needs for a DrawingArea to redraw itself, such as when we resize the window, or minimize and then restore the window, or put another window over it and then bring it back to the front, it calls the DrawingArea's on_draw method.

IMPORTANT: We don't **call** on_draw – we just **write** on_draw.
It's automatically called as needed.

...and Draw On It

view.cpp

```
bool View::on_draw(const Cairo::RefPtr<Cairo::Context>& cr) {
    // If nothing to draw, we're done – EDGE CASE!
    if (lines.size() == 0) return true;

    // Create a Cairomm context to manage our drawing
    Gtk::Allocation allocation = get_allocation();

    // Center the drawing in the window
    const double width = allocation.get_width();
    const double height = allocation.get_height();
    cr->translate(width / 2, height / 2);

    // Use a 3 pixel wide red line
    cr->set_line_width(3.0);
    cr->set_source_rgb(0.8, 0.0, 0.0); // Set lines to red

    // Draw the lines
    for(Line l : lines) {
        cr->move_to(l.x1(), l.y1());
        cr->line_to(l.x2(), l.y2());
    }

    // Apply the colors to the window
    cr->stroke();

    // Drawing was successful
    return true;
}
```

A “context” tracks our “crayon” - color, width, dash pattern, etc.

These are straight lines, but we can also draw (and fill) curves and complex shapes

Nothing appears until we “stroke” (fill in) the lines.

Finally and inevitably, main()

main.cpp

```
#include "turtle.h"
#include "view.h"
#include <gtkmm.h>

int main(int argc, char** argv)
{
   // Create the application
   auto app = Gtk::Application::create(argc, argv, "edu.uta.cse1325.turtle1");

   // Instance a window
   Gtk::Window win;
   win.set_title("Turtle Graphics");

   // Create the vector of lines using Turtle Graphics
   Turtle_graphics g;
   g.penDown();
   for (double d=0; d<20; ++d) {
     g.forward(d*15.0);
     g.turn(144);
   }

   // Instance a drawing surface containing the lines and add to the window
   View view{g.get_lines()};
   win.add(view);
   view.show();

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

Gtk::Application::create is a static method that handles routine gtkmm application initialization (including Gtk::Main kit).

Type “auto” just means “use whatever type the assignment is assigning”. Convenient!

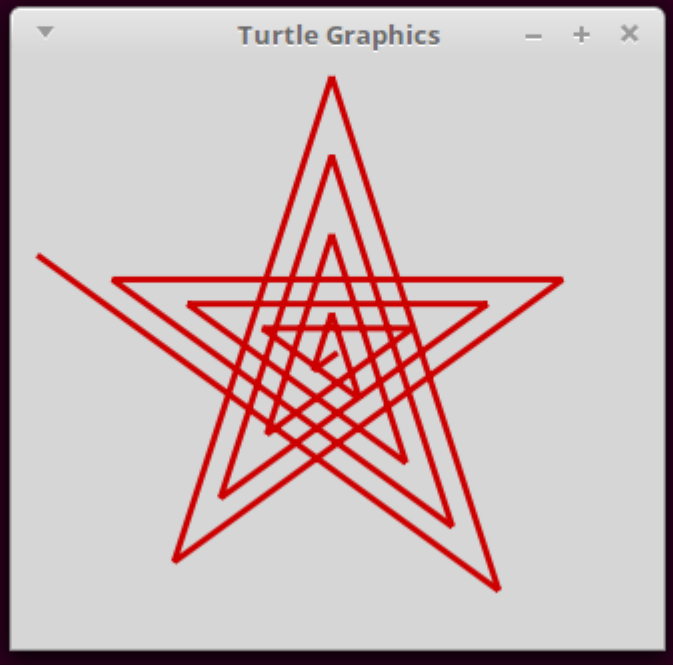
We draw (define line objects for) the same “star” shape as when we introduced the Turtle Graphics façade earlier.

Then we use those line objects to instance our turtle’s drawing area, add it to the window, and make it visible.

Finally, we tell gtkmm to go run our app!

Turtles in Action!

```
ricegf@pluto:~/dev/cpp/201801/12$ make clean
rm -f *.o star pointers hello hellop
ricegf@pluto:~/dev/cpp/201801/12$ make star
g++ -std=c++14 -c main.cpp `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
g++ -std=c++14 -c line.cpp
g++ -std=c++14 -c turtle.cpp
g++ -std=c++14 -c view.cpp `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
g++ -std=c++14 -o star main.o line.o turtle.o view.o `/usr/bin/pkg-config gtkmm-3.0 --cflags --libs`
ricegf@pluto:~/dev/cpp/201801/12$ ./star
```



Again, this is just an introductory overview.
We'll cover this a few more times. Don't Panic!

Quick Review

- Put the following user interface technologies in chronological order of introduction: Voice, GUI, Touch / Gesture, CLI, Punch Card, Paper Tape
  - How do web apps fit in?
- What is the Principle of Least Astonishment?
- A pointer variable contains the _____ of the value of interest. Accessing the value of interest via the pointer value is called _____.
- Memory for the value of interest for pointer variables is usually allocated from the ____ using the ____ keyword, and freed using the ___ keyword.
- Access a member of an object via a pointer uses the ___ operator.
- The _____ pattern implements a simplified interface to a complex class or package.

Quick Review

- What is the primary philosophical difference between CLI and GUI applications?
- Why is the main program loop part of gtkmm rather than written by you?
- To ensure your gtkmm program is compiled and linked using the right libraries, use the _____ tool with a _____.
- How is memory allocated from the stack? How (and when) is it subsequently deallocated?
- How is memory allocated from the heap? How (and when) is it subsequently deallocated?
- When are the two variants of “delete” used?

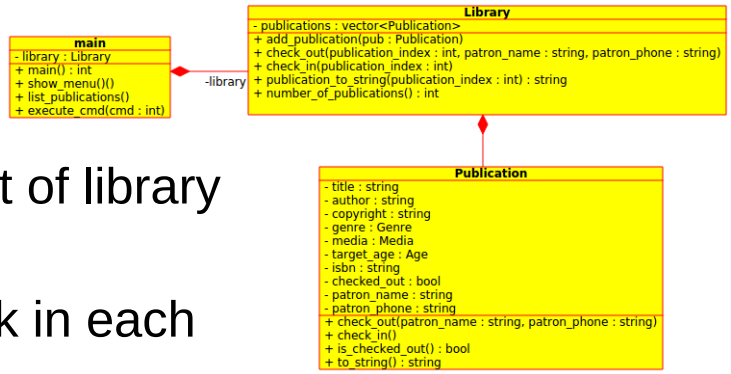
Next Class

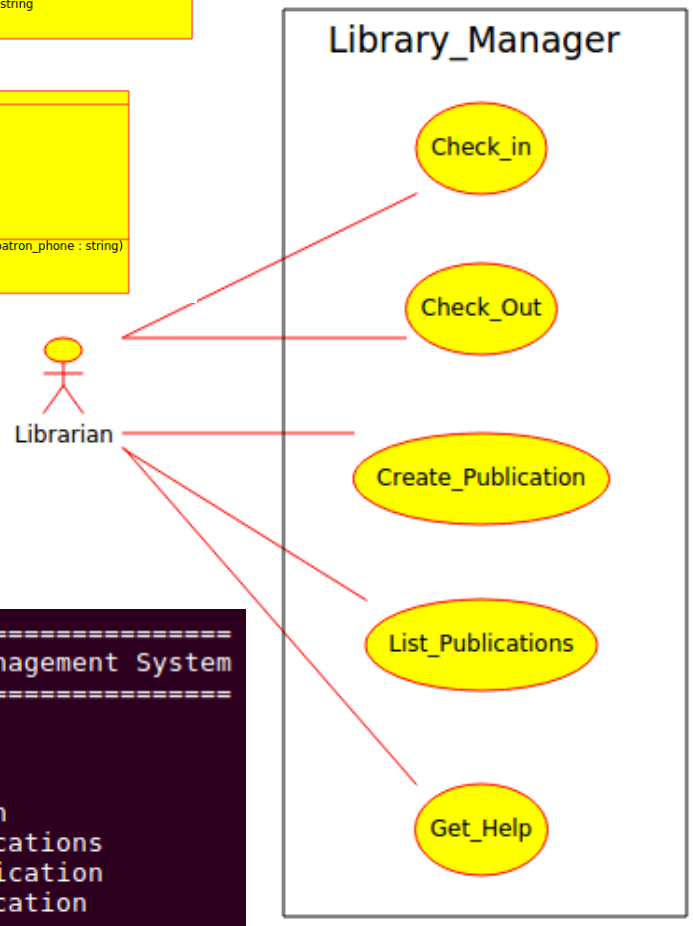
- Homework #5 (the LMS “model” with CLI) is due!
- (Optional) Review chapter 12 in Stroustrop
  - Remember: He is using FLTK and a façade!
  - Do the drills
- (Optional) Read chapter 16
  - We’ll create a Dialogs class that can display text messages and images, and get input from the user via text inputs and buttons
  - We’ll use these Dialogs to “GUI-ize” the Library Management System in Sprint #2

Homework #5

Sprint 1 of 3

- Build a simple Library Management System
  - Add publications to the list of library assets
  - Check out and check back in each publication
  - Provide basic help
- As always, use git for version management
- Manage the sprint with the Scrum spreadsheet
- Details are on Blackboard
- **Due March 1 at 8 am**






```
==================================
CSE1325 Library Management System
==================================

Publications
------------
(1) Add publication
(2) List all publications
(3) Check out publication
(4) Check in publication

Utility
-------
(9) Help
(0) Exit

Command?
```

Sprints

Feature ID	Priority	Planned	Status	As a...	I want to...
AP	1	1		Librarian	Add a publication
LP	2	1		Librarian	List all publications
CO	3	1		Librarian	Check out a publication
CI	4	1		Librarian	Check in a publication
HE	5	1		Librarian	Get help