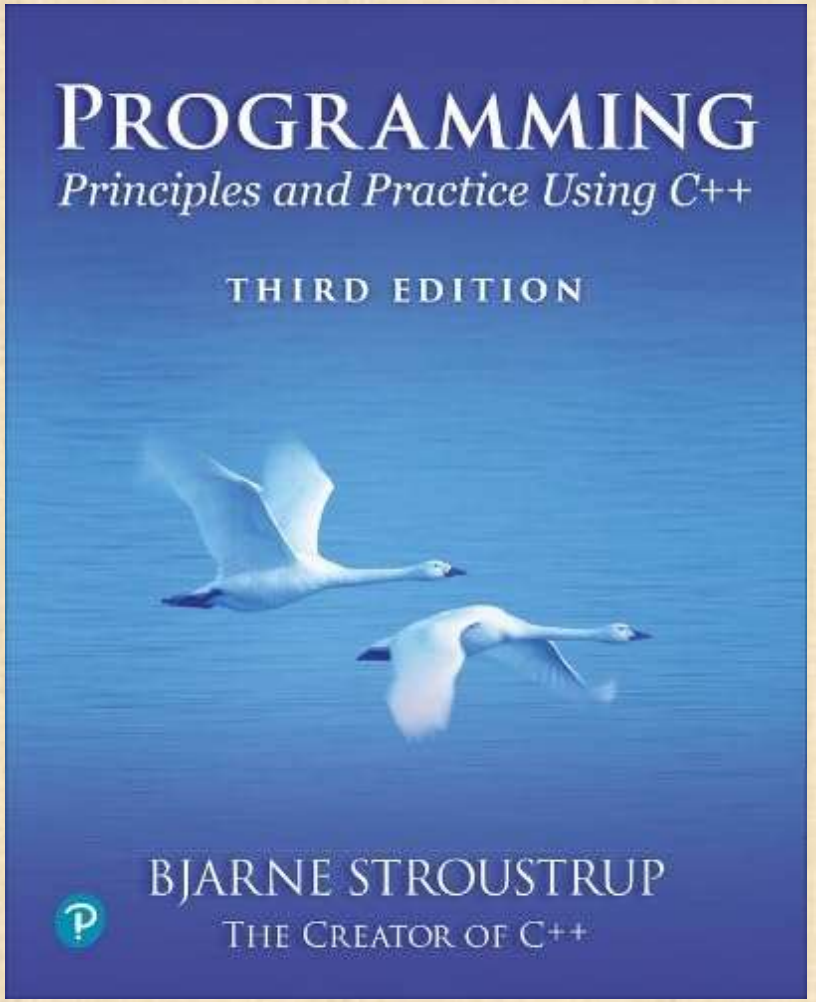
Chapter 14 – Graphical User Interfaces



Computing is not about computers anymore.
It is about living.
– Nicholas Negroponte

Overview

- Perspective
  - I/O alternatives
  - GUI
  - Layers of software
- GUI example
- GUI code
  - Callbacks
- A simple animation

I/O alternatives

- Use console input and output
  - A strong contender for technical/professional work
  - Command line interface
- Menu driven interface
- Graphic User Interface
  - Use a GUI Library
  - To match the “feel” of Windows/Mac/Android/iPhone/… applications
  - When you need drag and drop, WYSIWYG
  - Event driven program design
  - A web browser – this is a GUI library application
    - HTML / a scripting language
    - For remote access (and more)

Common GUI tasks

- Titles / Text
  - Names
  - Prompts
  - User instructions
- Fields / Dialog boxes
  - Input
  - Output
- Buttons
  - Let the user initiate actions
  - Let the user select among a set of alternatives
    - e.g., yes/no, blue/green/red, etc.
- Menus
- Sliders

Common GUI tasks (cont.)

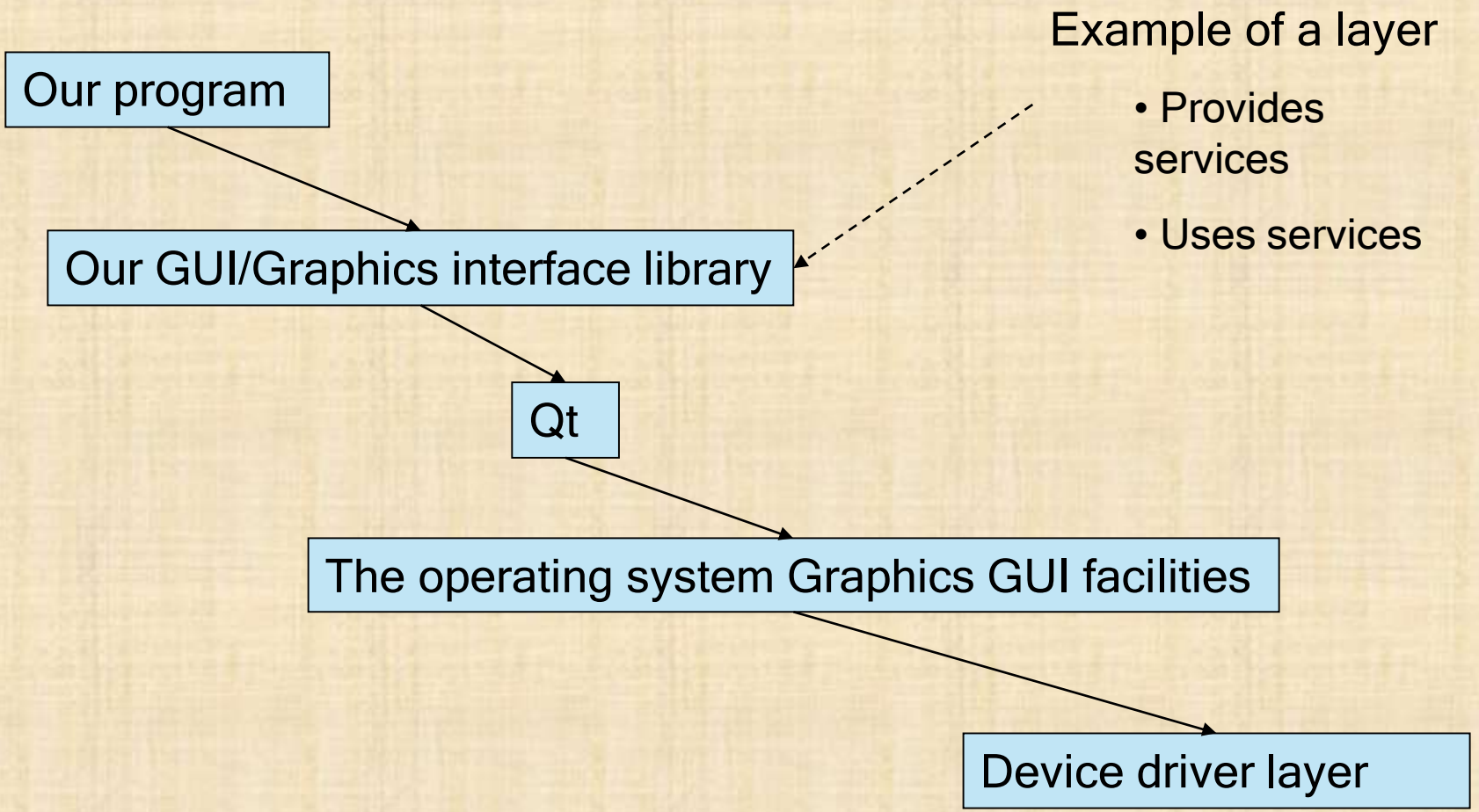
- Display results
  - Shapes
  - Text and numbers
- Make a window “look right”
  - Style and color
    - Note: our windows look different (and appropriate) on different systems
- More advanced
  - Tracking the mouse
  - Dragging and dropping
  - Free-hand drawing
  - Animation

GUI

- From a programming point of view GUI is based on two techniques
  - Object-oriented programming
    - For organizing program parts with common interfaces and common actions
  - Events
    - For connecting an event (like a mouse click) with a program action

Layers of software

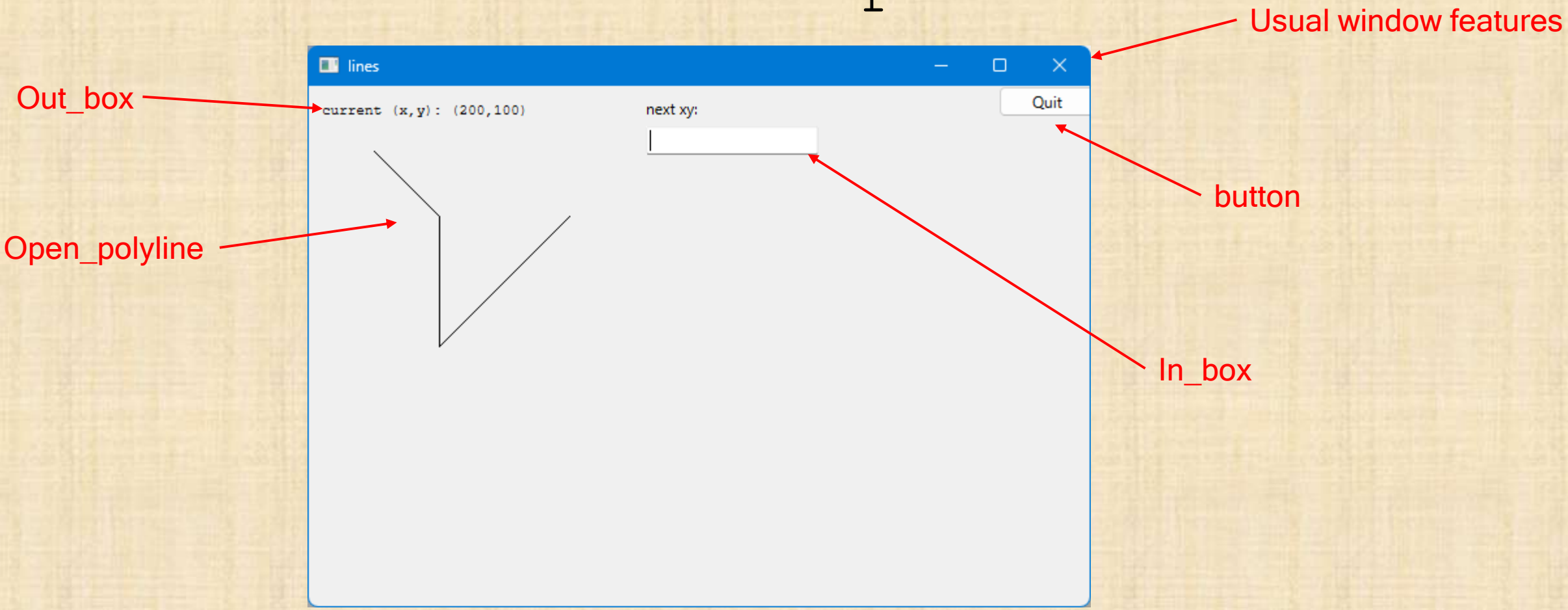
- When we build software, we usually build upon existing code



C++ GUI libraries

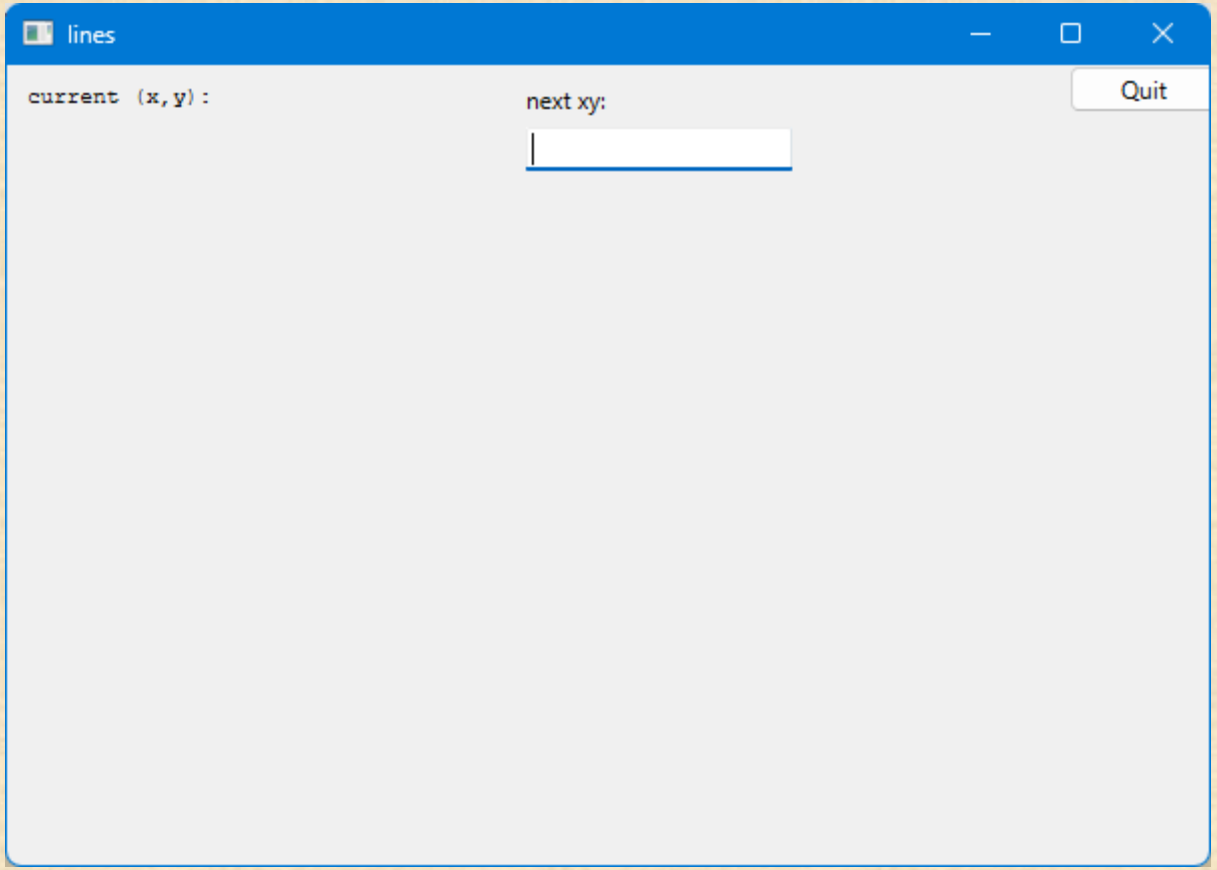
- There are ***many*** C++ GUI and Graphics libraries
  - That's a problem: how to choose?
  - FLTK, Gtkmm, Qt, Ultimate++, VxWorks, …
- Tradeoffs
  - Ease of use
  - Ease of installation
  - Generality
  - Range of platforms: Android, Browsers, iPad, Linux, Mac, Windows, embedded systems, …
  - Footprint
  - Latency
  - …
- We prefer systems that work on many platforms
  - PPP2 used FLTK
  - PPP3 uses Qt

GUI example



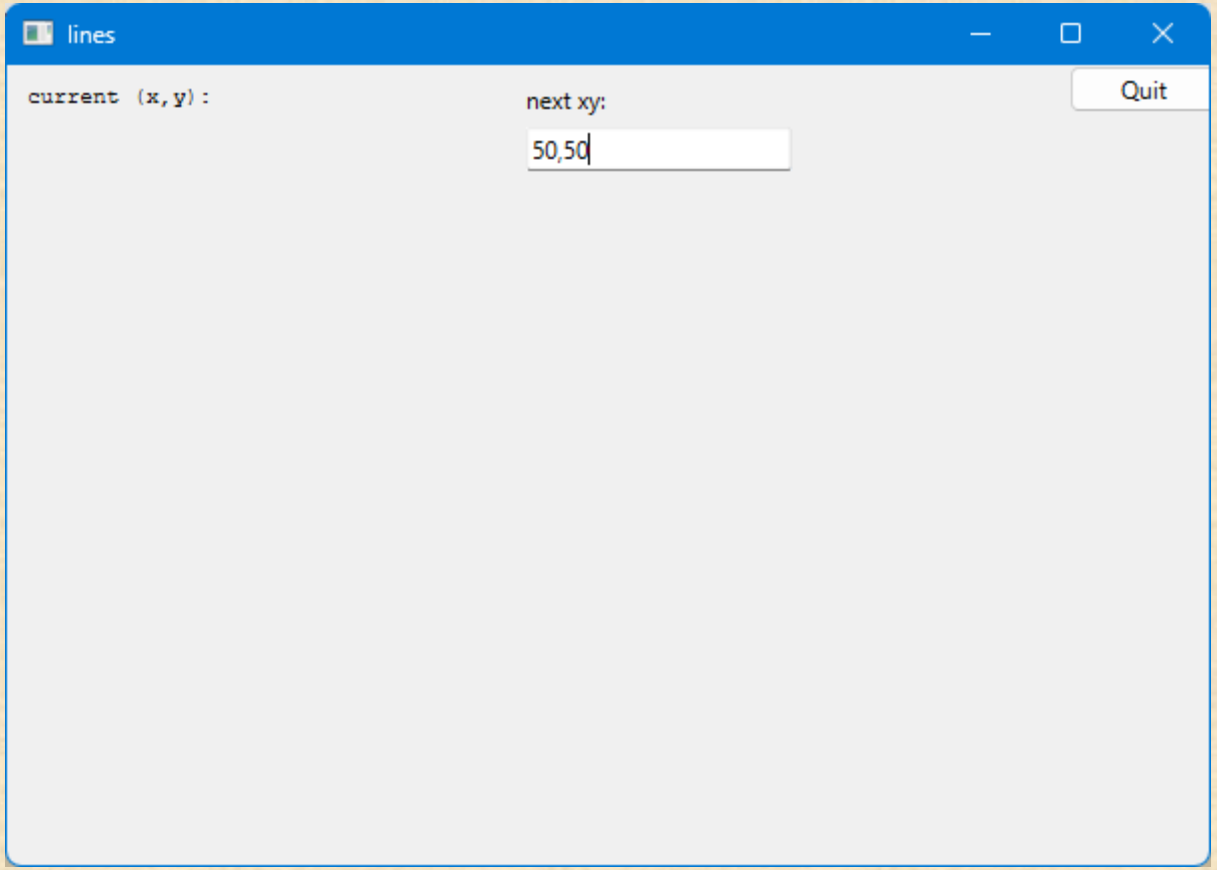
- Window with
  - The usual header: label, minimize, maximize, and close
    - That will look different on a different system
  - A **Button**, an **In_box**, an **Out_box**, and an **Open_polyline**

GUI example



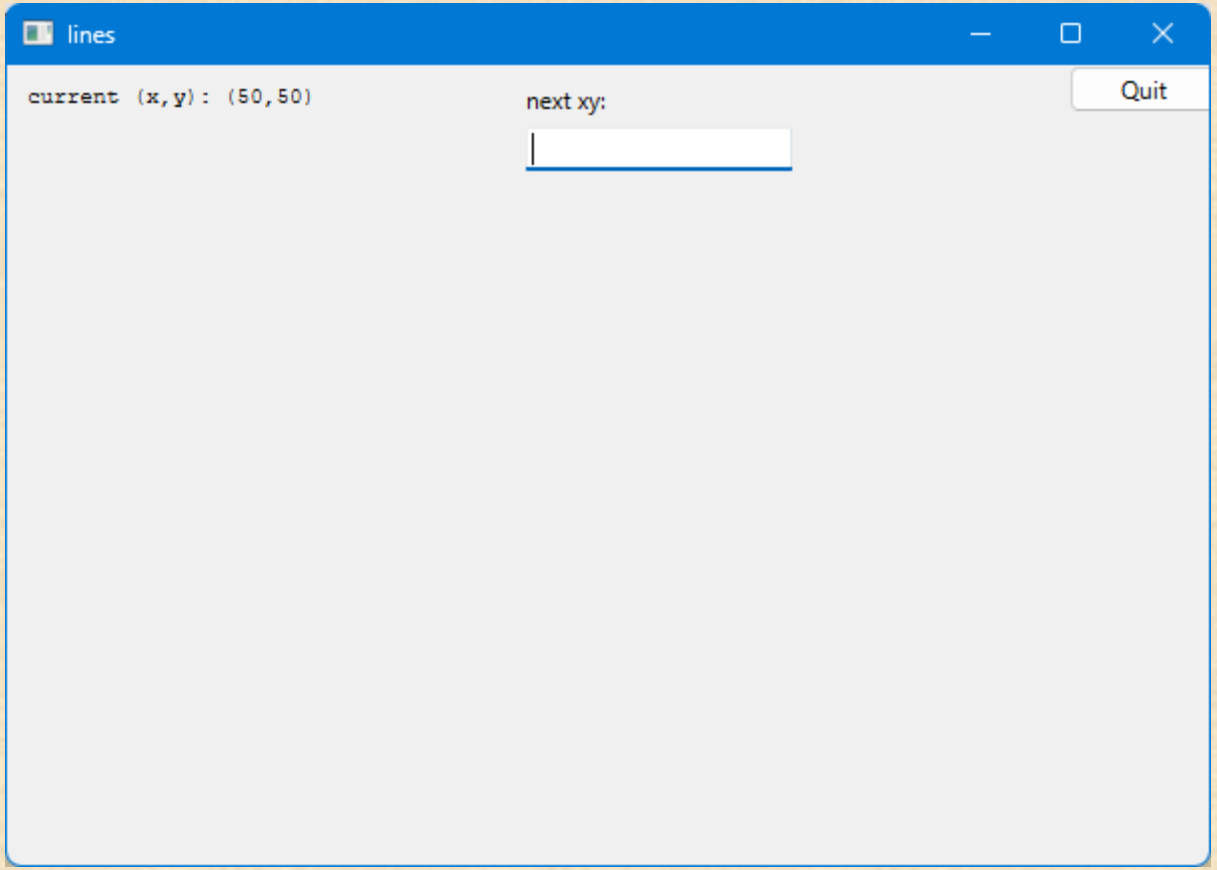
- Initial window

GUI example



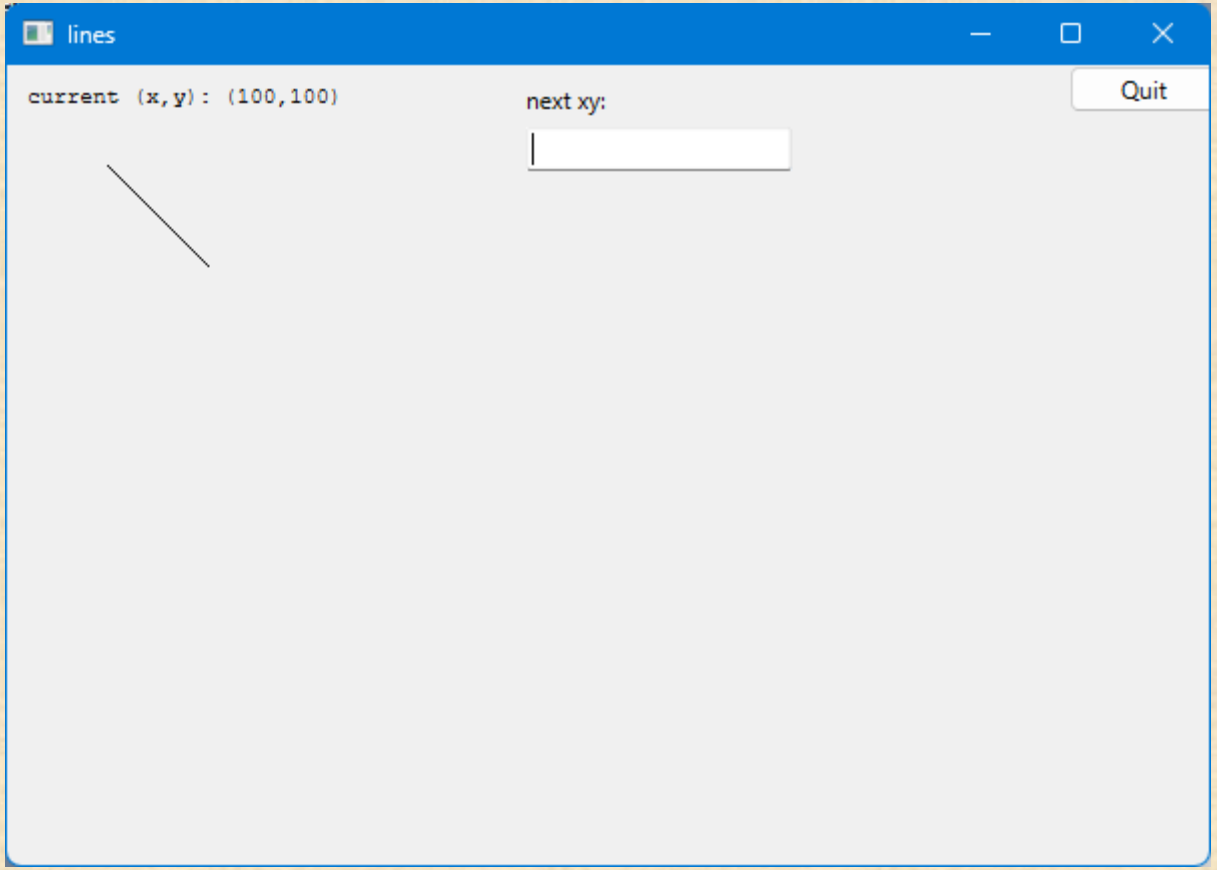
- Enter a point in the **In_box** labelled “next xy:”

GUI example



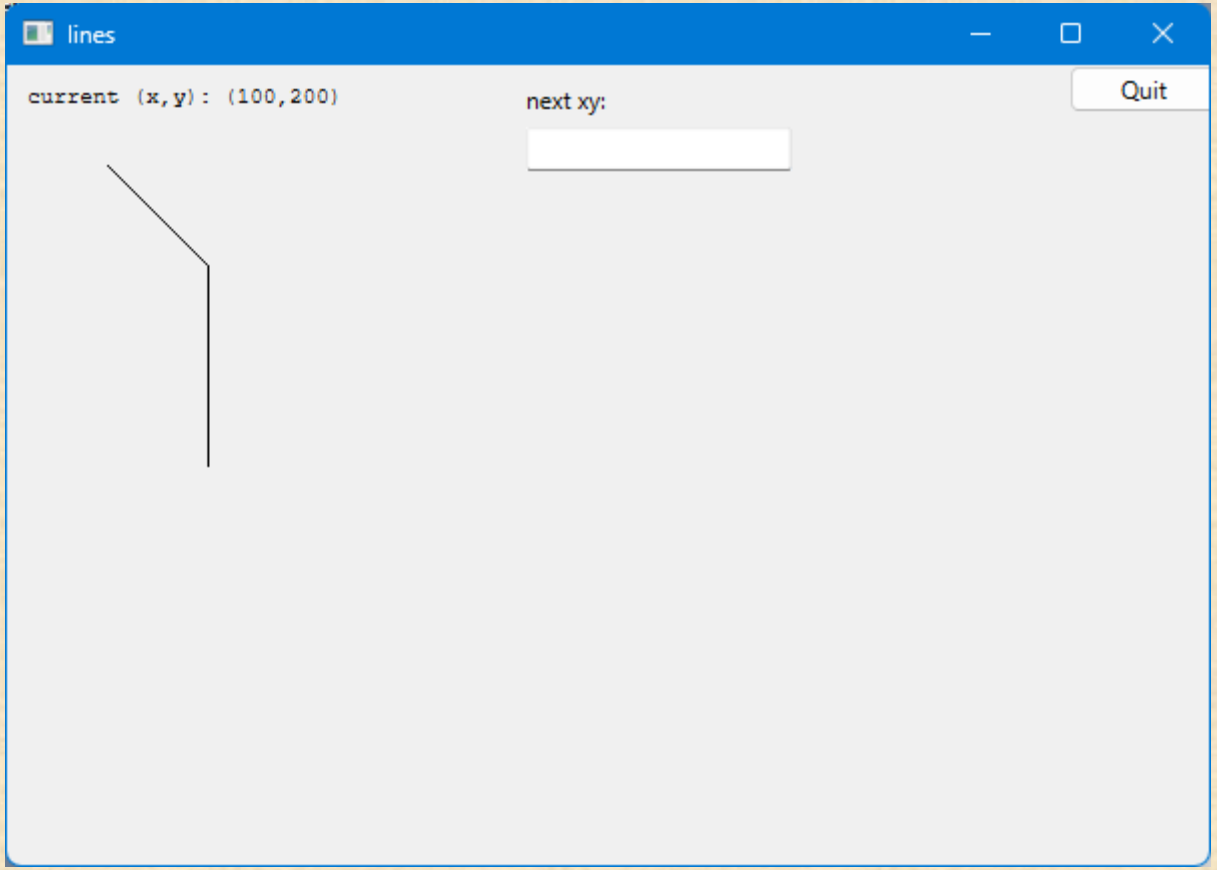
- Hit “enter” and the point entered appear in the **Out_box** labelled “current (x,y):”

GUI example



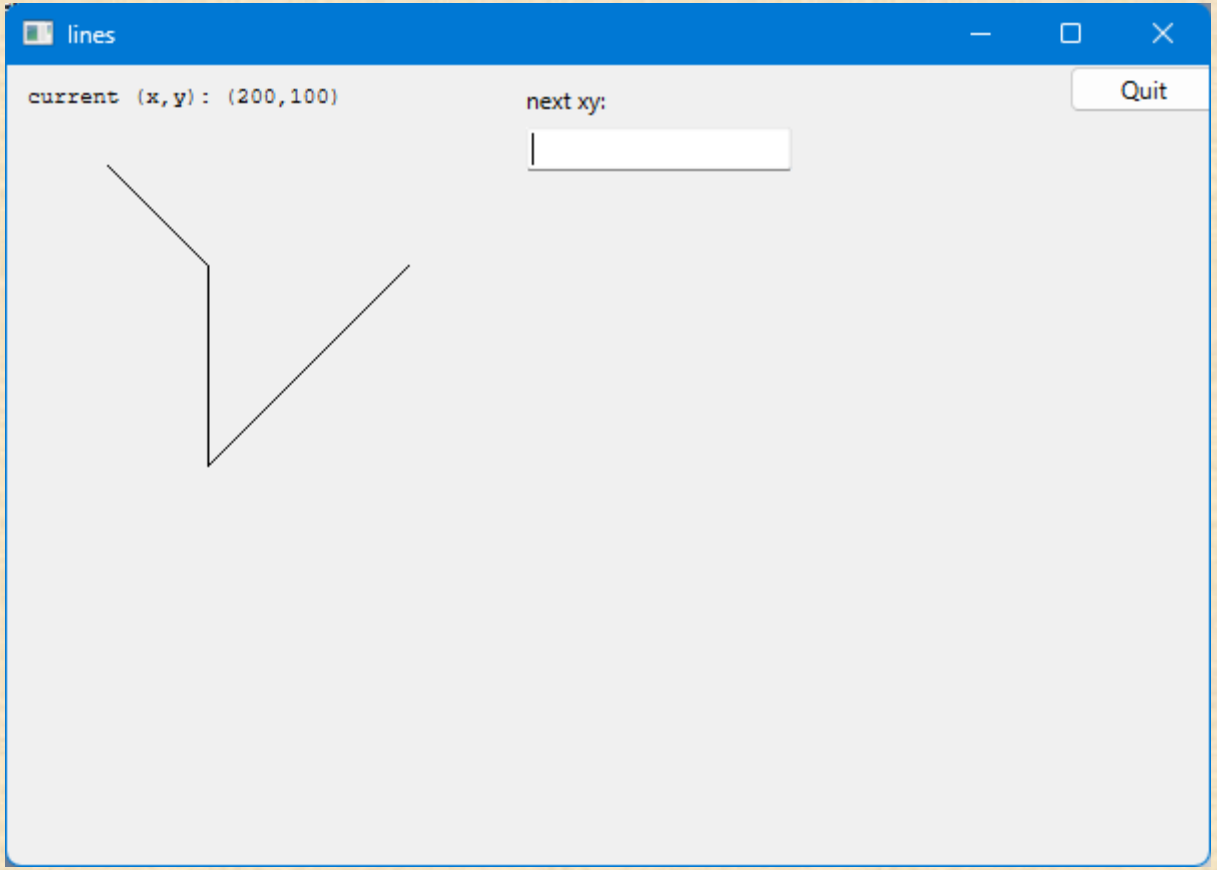
- Add another point and you have a line

GUI example



- Three points give two lines
  - Obviously, we are building a polyline

GUI example



- And so on, until you hit the **Button** labelled “Quit”.

So what? And How?

- We saw buttons, input boxes and an outbox in a window
  - How do we define windows, buttons, input boxes, and output boxes?
- Click on a button and something happens
  - How do we specify that action as code?
  - How do we connect our code to the button?
- You type something into an input box
  - How do we get that value into our code?
  - How do we interpret the string you type as a numbers?
- We saw output in the output box
  - How do we put the values there?
- Lines appeared in our window
  - How do we store the lines?
  - How do we draw them?

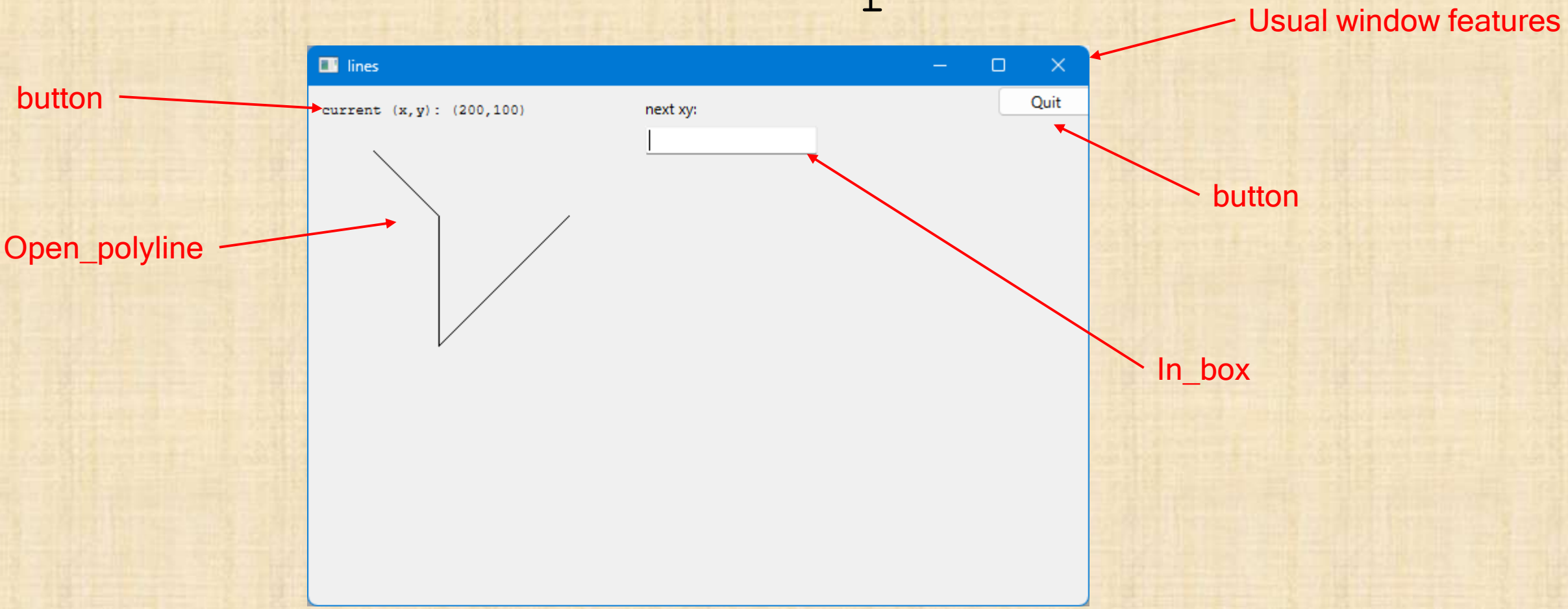
Mapping

- We map our ideas onto the Qt version of the conventional Graphics/GUI ideas

Define class Lines_window

```
struct Lines_window : Window {
        Lines_window(Application* application, Point xy, int w, int h, const string& title);
        Open_polyline lines;
        void wait_for_button();
private:
    Application* app = nullptr;
    Button quit_button;
    In_box next_xy;
    Out_box xy_out;
    void next();            // action for next_xy
    void quit();            // action for quit
};
```

GUI example



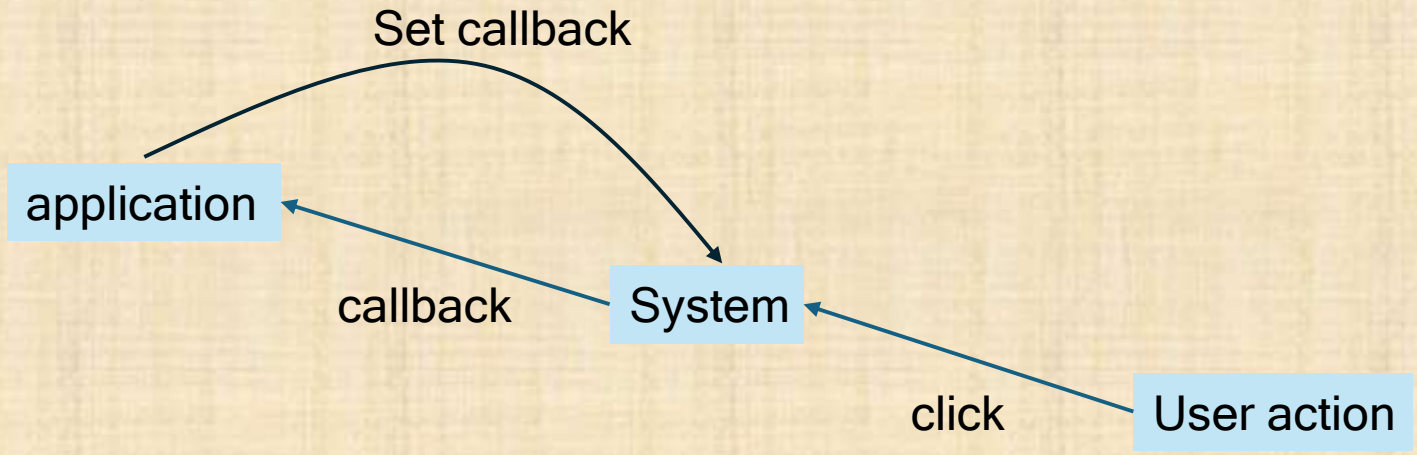
- Window with
  - A **Button**, an **In_box**, and an **Out_box**

The Lines_window constructor

```
Lines_window::Lines_window(Application* application, Point xy, int w, int h, const string& title)
    :Window{ xy,w,h,title },
    app(application),
    quit_button{ Point{x_max() - 70,0}, 70, 20, "Quit", [this]() { quit(); } },
    next_xy{ Point{250,0}, 50, 20, "next xy:", [this]() { next(); } },
    xy_out{ Point{10,10}, "current (x,y): " }
{
    xy_out.label.set_font_size(8);          // we control the appearance of the label
    xy_out.data.set_font_size(8);
    attach(quit_button);
    next_xy.hide_buttons();                 // our In_box shouldn't offer buttons (just use "enter")
    attach(next_xy);
    attach(xy_out);
    next_xy.show();                         // make sure our In_box can be seen
    attach(lines);
}
```

Widgets, Buttons, and Callbacks

- How it works
  - The application gives a “callback” function to the system to call for an event
  - When the event happens (e.g., a user clicks on a button), the system calls the callback



Widgets, Buttons, and Callbacks

- A Widget is a basic concept in window-based systems
  - Basically, anything you can see on the screen and can do something is a widget
    - called a “control” by Microsoft

```
using Callback = std::function<void()>;        // can be used to pass a function to the operating system

struct Widget {                 // a connection to an operating system
    Widget(Point xy, int w, int h, const string& s, Callback cb)
      :loc(xy), width(w), height(h), label(s), do_it(cb)
    { }
};

struct Button : Widget {                // Displays as a rectangle and has an action associated with it
    Button(Point xy, int w, int h, const string& label, Callback cb);
};
```

Widgets, Buttons, and Callbacks

- The constructor for a Widget (here a Button) must do a lot of system-specific setup
  - Fortunately, we don't often have to see that
    - just like we rarely have to see how the operating system does it sophisticated "magic"
    - Just like we rarely have to see how our computer hardware really works

```
Button::Button(Point xy, int w, int h, const string& label, Callback cb)
        :Widget(xy,w,h,label,cb)
{
        WidgetPrivate& w_impl = get_impl();          // Qt specific implementation code
        QPushButton* button = new QPushButton();
        w_impl.widget.reset(button);
        button->setText(QString::fromStdString(label));
        QObject::connect(button, &QPushButton::clicked, [this]{ do_it(); });
}
```

Our "action" code: quit()

```
// The action itself is simple enough to write if (and only if) you know the underlying system

void Lines_window::quit()
{
        end_button_wait();
        next_xy.dismiss();
        app->quit();
}
```

In_box

```
struct In_box : Widget {      // An In_box is a widget into which you can type characters
        In_box(Point xy, int w, int h, const string& s, Callback cb = {});
        // … for details see the code …
        void attach(Window& win) override;
        void hide_buttons();          // A Qt input window can be used with keyboard and/or buttons
        void show_buttons();
        enum State {idle, accepted, rejected};
        State last_result();
        void clear_last_result();
        string last_string_value();
        struct ResultData { /* … */ };
private:
        ResultData result;
        bool waiting = false;
};
```

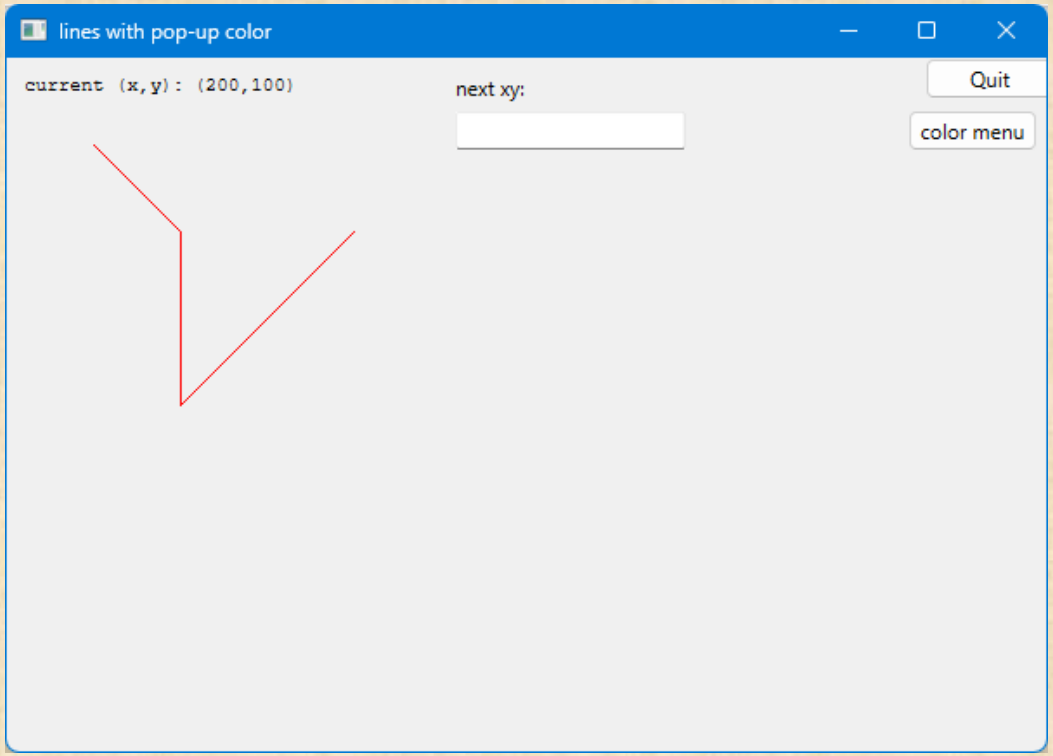
Our "action" code: next()

```
void Lines_window::next()          // read "int,int", e.g. "200,300" from out In_box
{
        if (next_xy.last_result() == In_box::accepted) {
                string s = next_xy.last_string_value();          // get the string typed in (if any)
                istringstream iss { s };
                int x;
                char ch;
                 int y;
                iss >> x >> ch >> y;          // read coordinates from the In_box's string
                lines.add(Point{ x,y });          // draw the new line
                ostringstream oss;
                oss << '(' << x << ',' << y << ')';
                xy_out.put(oss.str());          // update current position readout
          }
          next_xy.clear_last_result();          // remove the string from the In_box
}
```

We can build Windows out of Windows

```
struct Color_window : Lines_window {
    Color_window(Application* app, Point xy, int w, int h, const string& title);
private:
    void change(Color c) { lines.set_color(c); }
    void hide_menu() { color_menu.hide(); menu_button.show(); }

    Button menu_button;
    Menu color_menu;        // See §14.5.2
};
```



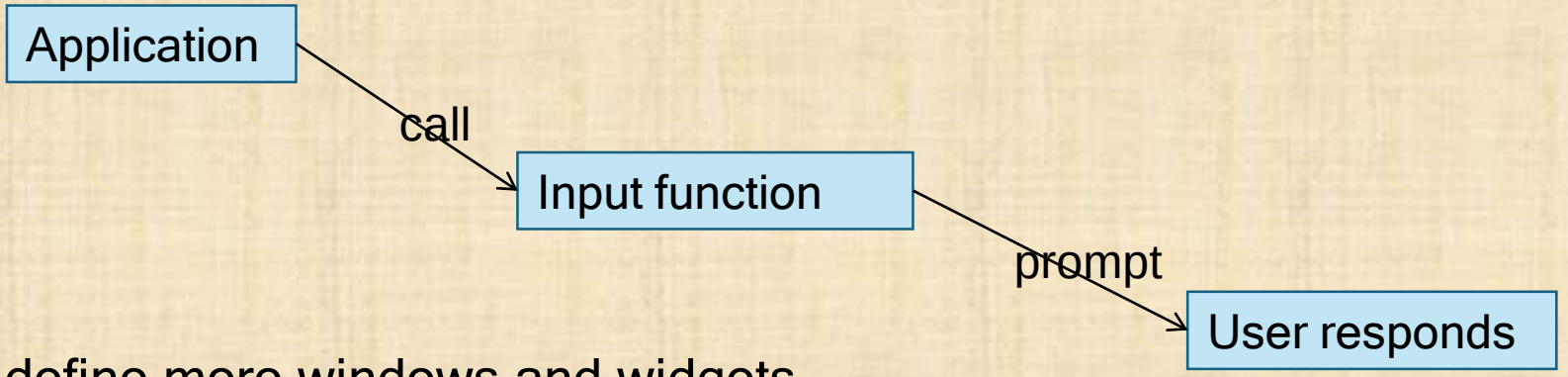
Control Inversion

- But where is the program?
  - Our code just responds to the user using our widgets
  - No loops?
  - No if-then-else?

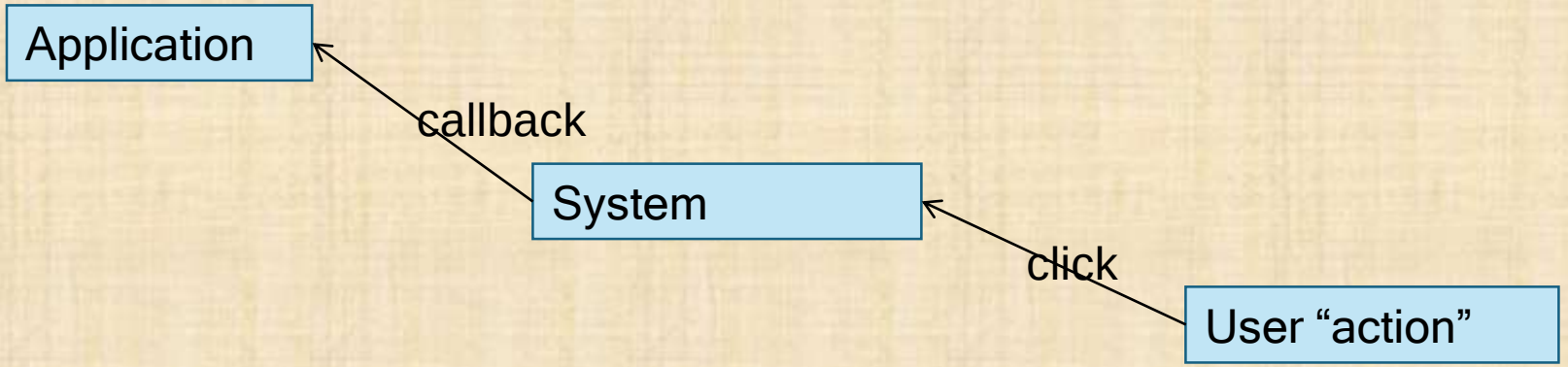
```
int main ()
{
    Application app;                    // the connection to the underlying system
    Lines_window win{ &app, Point{100,100},600,400,"lines" };    // our window with Widgets
    app.gui_main();                     // an "infinite loop"
}
```

Control Inversion

- Conventional control flow
  - To many things at once, see concurrency



- Event-driven control flow:
  - To do many things at once, define more windows and widgets



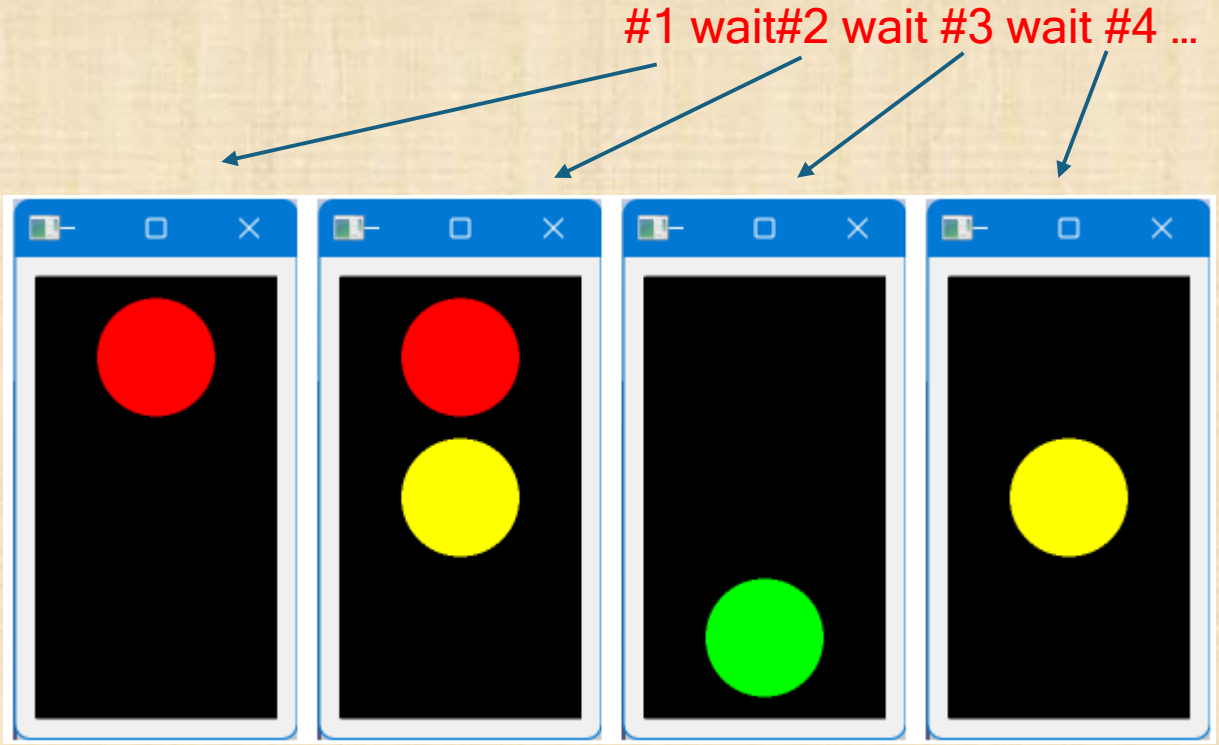
Simple animation

- Often, we want a Widget to change without user intervention
  - Examples: a clock face, a traffic light, a bouncing ball, a growing graph, …
- A key function (see §14.6):

```
void Window::timer_wait(int milliseconds)
{
        impl->timer_wait(milliseconds);
}

do_something()
win.timer_wait (1’000);
do_something_else();

// usually in a loop
```



Summary

- We have seen
  - Action on buttons
  - Interactive I/O
    - Text input
    - Text output
    - Graphical output
  - Simple animation
- Missing
  - Menu (See §14.5)
  - Window and Widget (see the code)
  - Anything to do with tracking the mouse
    - Dragging
    - Hovering
    - Free-hand drawing
  - What we haven't shown, you can pick up if you need it

Next lecture

- The next three lectures will show how the standard vector is implemented using basic low-level language facilities.
- This is where we really get down to the hardware and work our way back up to a more comfortable and productive level of programming.