Chapter 16
Graphical User Interfaces

Bjarne Stroustrup

www.stroustrup.com/Programming

Overview

- Perspective
  - I/O alternatives
  - GUI
  - Layers of software
- GUI example
- GUI code
  - callbacks

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 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.

Common GUI tasks (cont.)

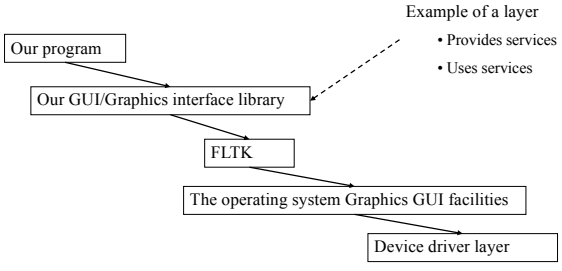
- Display results
  - Shapes
  - Text and numbers
- Make a window "look right"
  - Style and color
    - Note: our windows look different (and appropriate) on different systems
- More advanced
  - Tracking the mouse
  - Dragging and dropping
  - Free-hand drawing

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

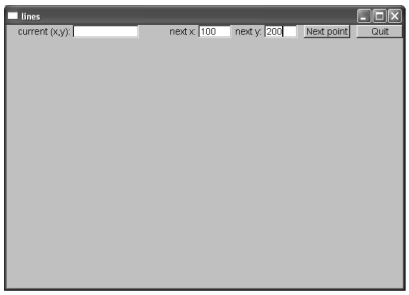
- When we build software, we usually build upon existing code



GUI example

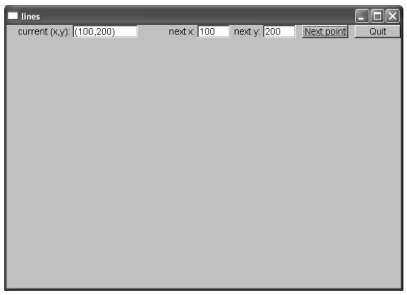
- Window with
  - two **Button**s, two **In_box**es, and an **Out_box**

GUI example



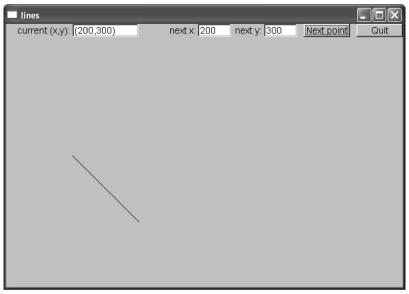
- Enter a point in the **In_box**es

GUI example



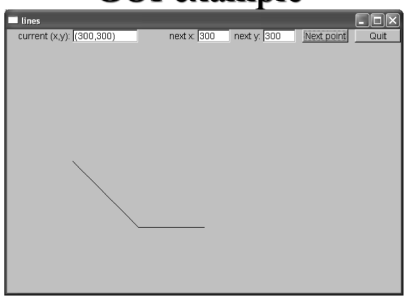
- When you hit **Next point** that point becomes the current (x,y) and is displayed in the **Out_box**

GUI example



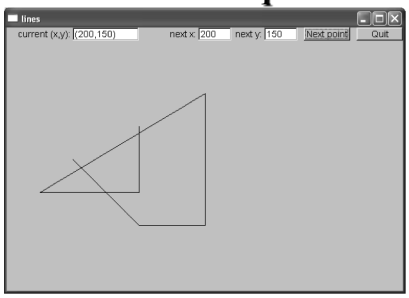
- Add another point and you have a line

GUI example



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

GUI example



- And so on, until you hit **Quit.**

So what? And How?

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

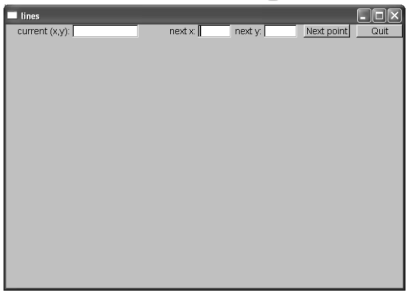
Mapping

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

Define class Lines_window

```
struct Lines_window : Window    // Lines_window inherits from Window
{
    Lines_window(Point xy, int w, int h, const string& title); // declare constructor
    Open_polyline lines;

private:
    Button next_button;         // declare some buttons – type Button
    Button quit_button;
    In_box next_x;              // declare some i/o boxes
    In_box next_y;
    Out_box xy_out;

    void next();                // what to do when next_button is pushed
    void quit();                // what to do when quit_bottom is pushed

    static void cb_next(Address, Address window); // callback for next_button
    static void cb_quit(Address, Address window); // callback for quit_button
};
```

GUI example



- Window with
  - two **Button**s, two **In_box**es, and an **Out_box**

The Lines_window constructor

```
Lines_window::Lines_window(Point xy, int w, int h, const string& title)
    :Window(xy,w,h,title),
        // construct/initialize the parts of the window:
                    // location    size    name    action
    next_button(Point(x_max()-150,0), 70, 20, "Next point", cb_next),
    quit_button(Point(x_max()-70,0), 70, 20, "Quit", cb_quit),      // quit button
    next_x(Point(x_max()-310,0), 50, 20, "next x:"),                // io boxes
    next_y(Point(x_max()-210,0), 50, 20, "next y:"),
    xy_out(Point(100,0), 100, 20, "current (x,y):")
{
    attach(next_button);            // attach the parts to the window
    attach(quit_button);
    attach(next_x);
    attach(next_y);
    attach(xy_out);
    attach(lines);                  // attach the open_polylines to the window
}
```

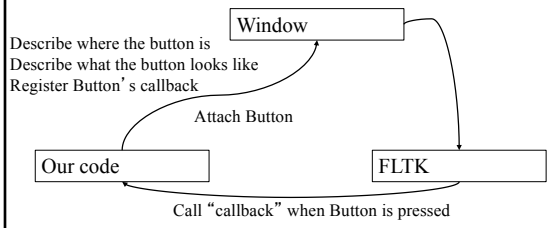

Widgets, Buttons, and Callbacks

- A Widget is something you see in the window which has an action associated with it
- A Button is a Widget that displays as a labeled rectangle on the screen, and when you click on the button, a Callback is triggered
- A Callback connects the button to some function or functions (the action to be performed)

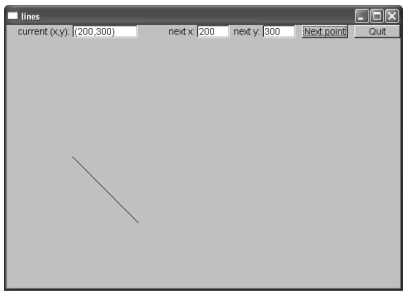
Widgets, Buttons, and Callbacks

```
// A widget is  something you see in the window
// which has an action associated with it

// A Button is a Widget that displays as a labeled rectangle on the screen;
// when you click on the button, a Callback is triggered
// A Callback connects the button to some function

struct Button : Widget {
    Button(Point xy, int w, int h, const string& s, Callback cb)
        :Widget(xy,w,h,s,cb) { }
};
```

How it works



GUI example



- Add another point an you have a line

Widget

- A basic concept in Windows and X windows systems
  - Basically anything you can see on the screen and do something with is a widget (also called a "control" by Microsoft)

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


Button

- A Button is a Widget that
  - displays as a labeled rectangle on the screen;
  - when you click on it, a Callback is triggered

```
struct Button : Widget {
    Button(Point xy, int w, int h, const string& s, Callback cb)
        :Widget(xy,w,h,s,cb) { }
};
```

Callback

- Callbacks are part of our interface to "the system"
  - Connecting functions to widgets is messy in most GUIs
  - It need not be, but
    - "the system" does not "know about" C++
    - the style/mess comes from systems designed in/for C/assembler
    - Major systems always use many languages; this is one example of how to cross a language barrier
  - A callback function maps from system conventions back to C++

```
void Lines_window::cb_quit(Address, Address pw)
    // Call Lines_window::quit() for the window located at address pw
{
    reference_to<Lines_window>(pw).quit();        // now call our function
}
```

Map an address into a reference to the type of object residing at that address – to be explained the following chapters

Our "action" code

```
// The action itself is simple enough to write

void Lines_window::quit()
{
    // here we can do just about anything with the Lines_window
    hide();          // peculiar FLTK idiom for "get rid of this window"
}
```


The next point function

```
// our action for a click ("push") on the next point button

void Lines_window::next()
{
    int x = next_x.get_int();
    int y = next_y.get_int();

    lines.add(Point(x,y));

    // update current position readout:
    stringstream ss;
    ss << '(' << x << ',' << y << ')';
    xy_out.put(ss.str());

    redraw();   // now redraw the screen
}
```


In_box

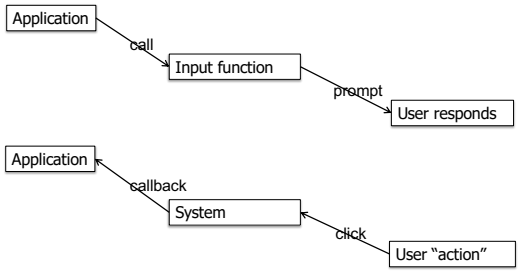
```
// An In_box is a widget into which you can type characters
// Its "action" is to receive characters

struct In_box : Widget {
    In_box(Point xy, int w, int h, const string& s)
        :Widget(xy,w,h,s,0) { }
    int get_int();
    string get_string();
};

int In_box::get_int()
{
    // get a reference to the FLTK FL_Input widget:
    Fl_Input& pi = reference_to<Fl_Input>(pw);
    // use it:
    return atoi(pi.value());          // get the value and convert
                                      // it from characters (alpha) to int
}
```

Control Inversion

- But where is the program?
  - Our code just responds to the user clicking on our widgets
  - No loops? No if-then-else?
- "The program" is simply

```
int main ()
{
    Lines_window win(Point(100,100),600,400,"lines");
    return gui_main();          // an "infinite loop"
}
```


Control Inversion



Summary

- We have seen
  - Action on buttons
  - Interactive I/O
    - Text input
    - Text output
    - Graphical output
- Missing
  - Menu (See Section 16.7)
  - Window and Widget (see Appendix E)
  - 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.