CSE 1325: Object-Oriented Programming

Lecture 15 – Chapters 13-15

Designing a Class Library, Lambda, and Adapter Pattern

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

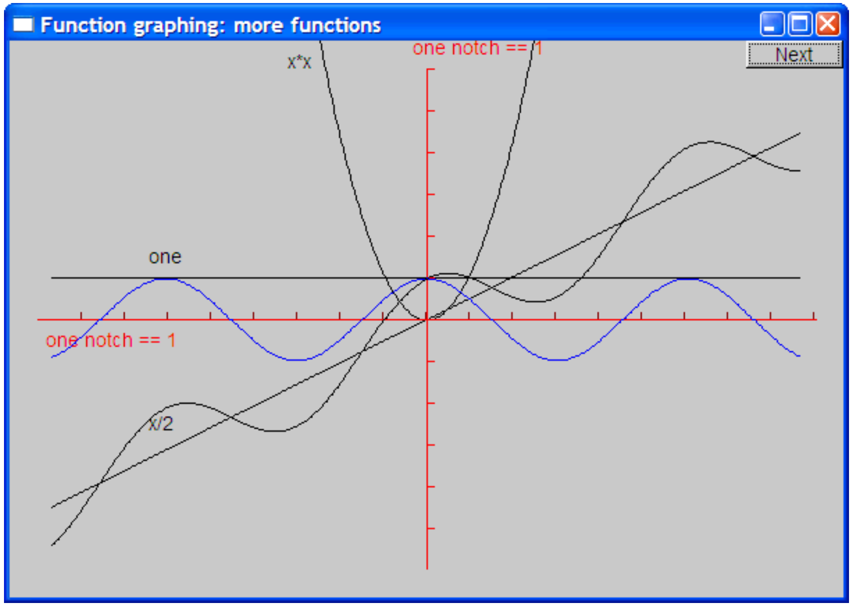
Lecture #11

Quick Review

- The **Observer** pattern notifies dependent objects when the observed event occurs.
- The **Factory** pattern creates new objects without exposing the creation logic to the client.
- The “new” operator allocates memory from the **heap**. To ensure it is returned (and avoid memory leaks), we use the ~ operator to define a **destructor**.
- The **Gtk::Application::create** factory handles most of the routine chores associated with creating a gtkmm GUI application.
- **Gtk::Manage** marks a Gtk widget for automatic deletion when the Gtk container referencing it is deleted.

Overview: Class Libraries

- Class Libraries
- C++ Enablers
  - Friendship
  - Operator Overloading
  - Abstract Classes
  - Typedef
  - Lambda
- Adapter Pattern
- Bug Management
- Generalization and Specialization



Class Libraries

- A **Class Library** is a collection of prewritten classes, often designed as templates, that work together to facilitate writing applications
- Numerous class libraries exist for C++
  - The Standard Library and Standard Template Library (STL
  - Proprietary libraries include Microsoft .NET and Foundation Class Libraries (FCL), Apple iOS Frameworks, and Oracle Class Libraries (OCL)
  - Graphics libraries include gtkmm, Gtk+, Qt, and FLTK
  - General purpose libraries include Boost - http://www.boost.org/doc/libs/

Expertise
/eks-per-tyz/
def. Special skill or knowledge in a particular subject, eg. He has expertise in his field of molecular scien

The best code you'll ever write is code you didn't write, but code you reused, often from a class library

Designing a Library

- Start by addressing the key issues
  - What is the root class?
  - How are real-world concepts specified?
How do they relate?
  - To what precision should they be modeled ?
  - What derived classes should be provided?
  - What utility classes are needed?
- This is an excellent time for prototyping and exploring the use cases!

A Point about Namespaces

```
namespace Graph_lib {   // a graphics interface in Graph_lib
  class Point {      // a Point is simply a pair of ints (the coordinates)
    public:
      Point(int xx, int yy) : x(xx), y(yy) { }
      friend bool operator==(Point a, Point b);
    private:
      int x, y;
    };

// Define equality of points
inline bool operator==(Point a, Point b) { return a.x==b.x && a.y==b.y; }
inline bool operator!=(Point a, Point b) { return !(a==b); }
}
```

Namespaces help to prevent name conflicts in large projects.

Symbols declared in a namespace are allocated to its scope. The same name in a different namespace is a distinct symbol, thus ensuring identically-named symbols do not cause ambiguity.

Multiple namespace blocks with the same name are permitted. All declarations within namespaces with the same name are allocated to the named scope.

Friendship

```
namespace Graph_lib {   // our graphics interface is in Graph_lib
  class Point {      // a Point is simply a pair of ints (the coordinates)
    public:
      Point(int xx, int yy) : x(xx), y(yy) { }
      friend bool operator==(Point a, Point b);
    private:
      int x, y;
    };

// Define equality of points
inline bool operator==(Point a, Point b) { return a.x==b.x && a.y==b.y; }
inline bool operator!=(Point a, Point b) { return !(a==b); }
}
```

In C++, a friend (or friend function) of a class is defined OUTSIDE of class scope, yet has access to all private and protected members of that class.

The friend is declared within class scope to grant friendship, but is **NOT** part of the class! It is usually a function, although it may also be a class. Friendship is one-way unless mutually declared.

Operator Overloading

```
namespace Graph_lib {   // our graphics interface is in Graph_lib
  class Point {      // a Point is simply a pair of ints (the coordinates)
    public:
      Point(int xx, int yy) : x(xx), y(yy) { }
      friend bool operator==(Point a, Point b);
    private:
      int x, y;
    };
```

Here we define “==” and “!=” for Point objects.

```
// Define equality of points
inline bool operator==(Point a, Point b) { return a.x==b.x && a.y==b.y; }
inline bool operator!=(Point a, Point b) { return !(a==b); }
}
```

We define the meaning of Point “equality”, so that we can write:

```
Point p1{5, 3};

Point p2{3, 9};

if (p1 != p2) cout << “Points are not equal!” << endl;
```

This will print “Points are not equal!” when run.

But why isn't operator!= also a friend of Point?

Defining Shapes and Lines

```
// A shape holds lines represented as pairs of points
class Shape {
  public:
    Shape() = 0;                  // Abstract class
    void add(Point p);            // Adds point to the shape
    virtual void draw() = 0;  // May be used polymorphically for derived classes
};

class Line : public Shape {   // a Line is a Shape defined by just two Points
  public:
    Line(Point p1, Point p2) {   // construct a line from p1 to p2
      add(p1);   // add p1 to this shape (add() is provided by Shape)
      add(p2);   // add p2 to this shape
}
```

Expertise
/eks-per-tyz/
def: Special skill or knowledge in a particular subject, eg. He has expertise in his field of molecular

An **Abstract Class** contains one or more pure virtual methods or constructors (in C++, one to which 0 is assigned).

- An abstract class **cannot** be instanced.
- An abstract class **can** be used as a type, whose variables **can** contain objects of derived types.
- An abstract class **can** be used as a base class.

Shape is an abstract class, while Line is a concrete class.
Shape() (the constructor) and draw() are pure virtual.

Defining a Function Plotter

```
typedef double Fct(double);

// Plots the provided function from 0 to count
class Function : public Shape {
  public:
    Function(Fct f, double r1, double r2, Point orig,
        int count = 100, double xscale = 25, double yscale = 25);
};
```

Expertise
/eks-per-tyz/
def. Special skill or knowledge in a particular subject, eg. He has expertise in his field of molecular science

A **typedef** defines an alias for an (often complex) type, simplifying code and increasing readability.

For example, this would define a type X synonymous with an int:

```
typedef int X;
```

Fct above is a function with a parameter of type double and a return type of double (e.g., the sin() function). This allows us to pass the function to Function for plotting. Clever!

Plotting Some Cosines

```
#include<cmath>        // standard mathematical functions

// Graph a cosine
Function s4(cos,-10,11,Point{0,0},400,20,20);

// Graph a sloping cosine
double sloping_cos(double x) { return cos(x)+slope(x); }
Function s5(sloping_cos,-10,11,orig,400,20,20);
```

sloping_cos is only used once. It would be more legible (and supportable) to just define it where it is used.

But how can you define a function in the middle of a different function definition?

Lambda

- A **lambda** is a an anonymous function object. It is usually defined where it is invoked or where it is passed as an argument to a different function.

Expertise
/eks-per-tyz/
def: Special skill or knowledge in a particular subject, eg. He has expertise in his field of molecular scien

  - Lambdas should be short – a few lines of code
  - Lambdas can only be used once – they are anonymous and thus can't be referenced again
- Lambdas originated in functional programming
  - They were added to C++ 11, and enhanced in C++ 14
  - Lambdas are also available in Python 2.5+ and Java 8+

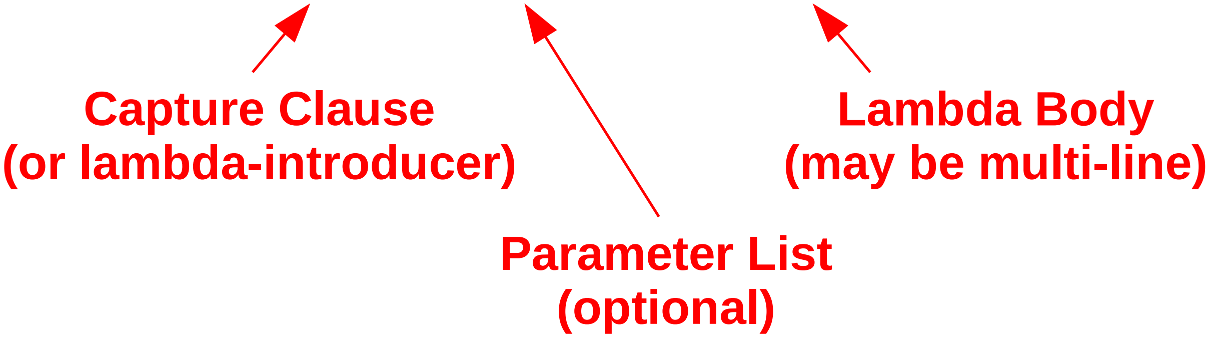
Example C++ Lambda

- So instead of this:

```
double sloping_cos(double x) { return cos(x)+slope(x); }
Function s5(sloping_cos,-10,11,orig,400,20,20);
```

- We have this:

```
Function s5( [ ] (double x) {return cos(x)+slope(x); },-10,11,orig,400,20,20);
```



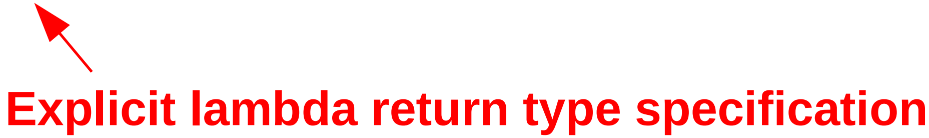
Capture Clause
(or lambda-introducer)

Lambda Body
(may be multi-line)

Parameter List
(optional)

The return type specifier of “auto” is assumed unless explicitly defined, e.g.,

```
[ ] (double x) -> double {return cos(x)+slope(x); }
```



Explicit lambda return type specification

Lambda: Capture Clause
(or lambda-introducer)

- A lambda can access – or *capture* – variables from the surrounding scope
  - If empty, no external variables are captured
  - If prefixed by &, variables are referenced; otherwise, they are copied (by value)
  - A parameter of & or = captures *all* variables in scope as referenced or copied, respectively, unless otherwise specified
  - *this* (if needed) must always be referenced, never copied

[&total, factor] – Capture total by reference, factor by value

[&, factor] – Capture factor by value, all others by reference

[&total, =] – Capture total by reference, all others by value

[&, &total] – ERROR, & is the default and cannot be also specified

[factor, factor] – ERROR, cannot capture a variable more than once

Lambda: Return Type and Parameter List

- The parameter list in most respects resembles that for a normal function
  - The parameter list is optional, and may be omitted (including the parentheses)
  - **[ ] {return rand();} // no parameter required**
- The return type is auto by default, and is thus usually deduced from the return statement type
  - **[ ] (int i) {return i*i;} // return type is int**
  - **[ ] {string s; return getline(cin, s); return s;} // return type is string**
  - **If needed, an explicit return type may be specified using “-> type” immediately preceding the body, e.g.,**

    [ ] (int i) -> double {return i*i;} // return type is double

Lambda: Body

- The body of a lambda many contain anything that a normal function body contains
- The body can reference:
  - Captured variables
    - And class data members from *this* (if captured)
  - Parameters
  - Locally-declared variables
  - Static and globally-declared variables

Using Lambdas

- A lambda may be passed as a parameter
  - sort(a_vector.begin(), a_vector.end(),
  -     [ ](const a_class & a, const a_class & b) -> bool { return a.cost < b.cost; }
  -); // sort a vector a_vector containing objects of a_class by their cost fields
- A lambda may be evaluated in place
  - [&, n] (int a) mutable { m = ++n + a; }(4); // m becomes 5, n remains 0
- A lambda may be assigned to a variable and treated as a referenced function (but use a normal function)
  - double x = 3.0, y = 4.0;
    function<double (void)> hypotenuse = [x, y] { return sqrt(x*x + y*y; };
    cout << hypotenuse() << endl; // prints 5.0
    x *= 2.0;
    cout << hypotenuse() << endl; // prints 7.2111

Some more functions (now with lambda!)

```
#include<cmath>        // standard mathematical functions

Function s4(cos,-10,11,orig,400,20,20);
Function s5( [ ] (double x) {return cos(x)+slope(x); },
                 -10, 11, orig, 400, 20, 20);
```

Standard mathematical functions
(<cmath>)

- **double abs(double);** *// absolute value*

- **double ceil(double d);** *// smallest integer* ***>= d***
- **double floor(double d);** *// largest integer* ***<= d***

- **double sqrt(double d);** *//* ***d*** *must be non-negative*

- **double cos(double);**
- **double sin(double);**
- **double tan(double);**
- **double acos(double);** *// result is non-negative; “a” for “arc”*
- **double asin(double);** *// result nearest to 0 returned*
- **double atan(double);**
- **double sinh(double);** *// “h” for “hyperbolic”*
- **double cosh(double);**
- **double tanh(double);**

Standard mathematical functions
(<cmath>)

- **double exp(double);** ***//*** *base e*
- **double log(double d);** ***//*** *natural log (base e)* ***; d*** *must be positive*
- **double log10(double);** ***//*** *base 10 logarithm*

- **double pow(double x, double y);** ***// x*** *to the power of* ***y***
- **double pow(double x, int y);** ***// x*** *to the power of* ***y***
- **double atan2(double y, double x);** ***//*** *atan(y/x)*
- **double fmod(double d, double m);** ***//*** *floating-point remainder*
  // same sign as d%m
- **double ldexp(double d, int i);** ***//*** *d*pow(2,i)*

Question?

- What if the function we want to graph doesn't accept and return a double?

Question?

- What if the function we want to graph doesn't accept and return a double?
  - We could **modify the class library** to include an overloaded Function that accepts our signature
  - We could **create an adapter** to translate between our function and the existing Function
- The latter solution is common enough to exist as a pattern – the **Adapter Pattern**

Structural

Adapter Pattern

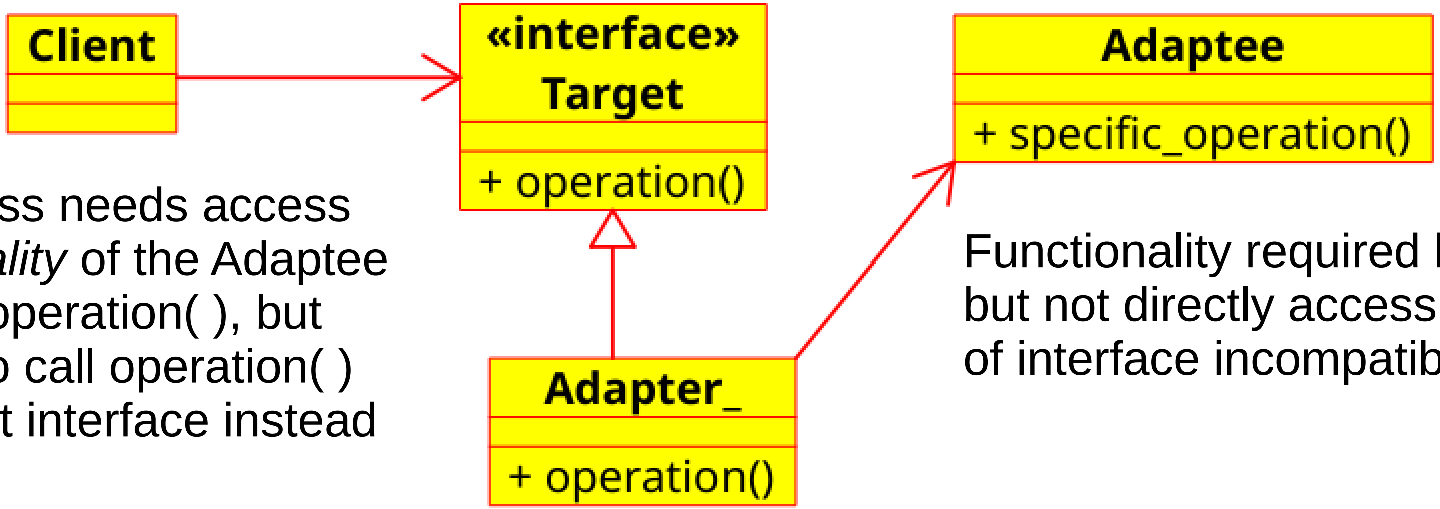
- The **Adapter** pattern implements a bridge between two classes with incompatible interfaces
  - The adapter implements the interface expected by the client class
  - The adapter calls the interface provided by the server (or adaptee) class
  - The adapter translates communication in both directions to meet both interface specifications

Structural

The Adapter Pattern

(Technically, the *Object* Adapter Pattern)

Target is the interface expected by Client



The Client class needs access to the *functionality* of the Adaptee in specific_operation( ), but is designed to call operation( ) from the Target interface instead

Functionality required by the Client, but not directly accessible because of interface incompatibility

The obvious solution would be to “simply” rewrite Client to use the Adaptee (Server) directly.
In the real world, this may be impractical due to cost / schedule constraints, or a need to simultaneously support Target and Adaptee servers.

So the adapter implements the interface expected by the Client, and translates Client calls into Adaptee (Server) calls.
Client, Target, and Adaptee are completely unchanged.

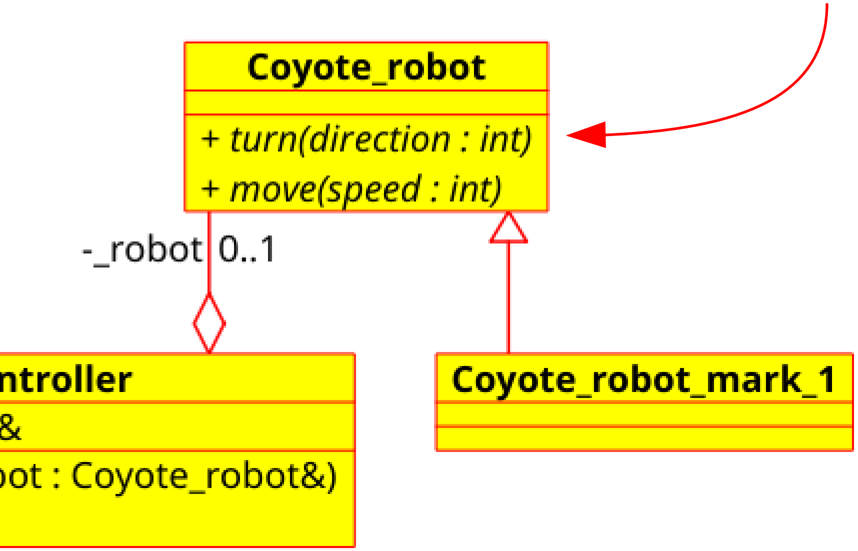
Example: Coyote Seeks Road Runner for Dinner Date

- Coyote has an existing control system that calls two methods, turn(direction) and move(speed)

This ends badly.

(It always ends badly...)

Note the two methods used by controller

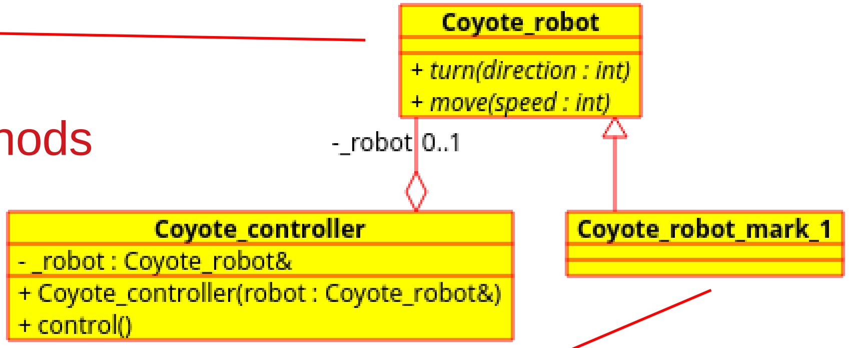


Wile E. Coyote ©1949 by Warner Brothers – Fair use is asserted

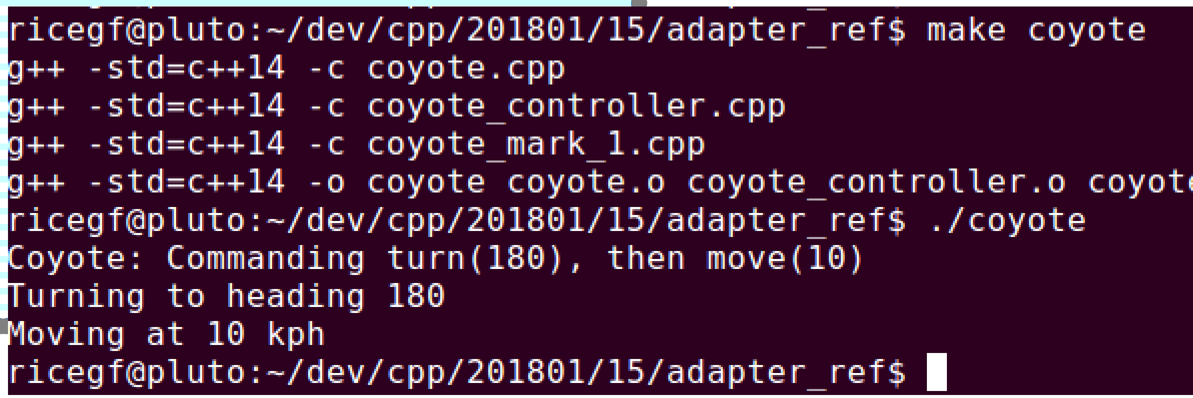
The Coyote System with Desired Server (Adaptee)

```
class Coyote_robot {
  public:
    virtual void turn(int direction) = 0;
    virtual void move(int speed) = 0;
};

class Coyote_controller {
  public:
    Coyote_controller(Coyote_robot& robot);
    void control();
  private:
    Coyote_robot& _robot;
};

class Coyote_robot_mark_1 : public Coyote_robot {
  public:
    virtual void turn(int direction) override;
    virtual void move(int speed) override;
};

int main() {
  Coyote_robot_mark_1 robot{};
  Coyote_controller controller{robot};
  controller.control();
}
```

Two methods



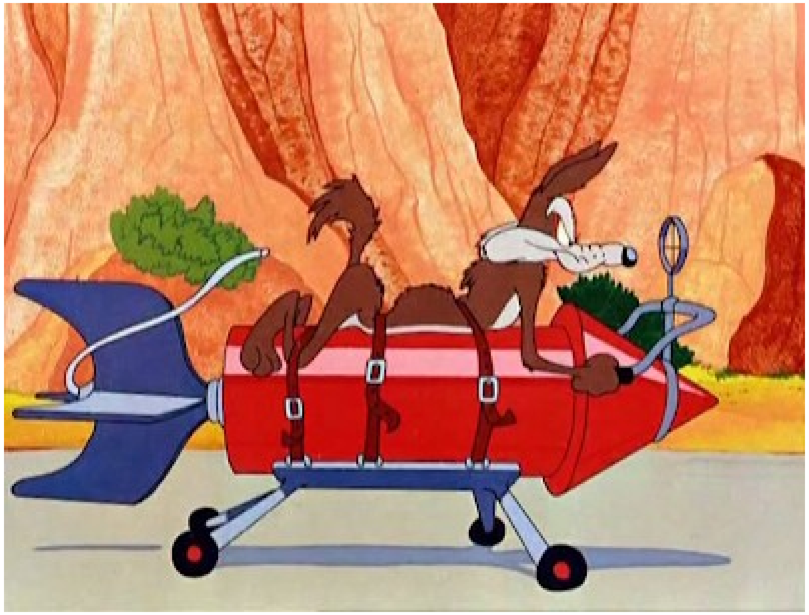
Implementations are attached to the slides on Blackboard.



```
ricegf@pluto:~/dev/cpp/201801/15/adapter_ref$ make coyote
g++ -std=c++14 -c coyote.cpp
g++ -std=c++14 -c coyote_controller.cpp
g++ -std=c++14 -c coyote_mark_1.cpp
g++ -std=c++14 -o coyote coyote.o coyote_controller.o coyote
ricegf@pluto:~/dev/cpp/201801/15/adapter_ref$ ./coyote
Coyote: Commanding turn(180), then move(10)
Turning to heading 180
Moving at 10 kph
ricegf@pluto:~/dev/cpp/201801/15/adapter_ref$
```

Acme Offers a Superior Controller

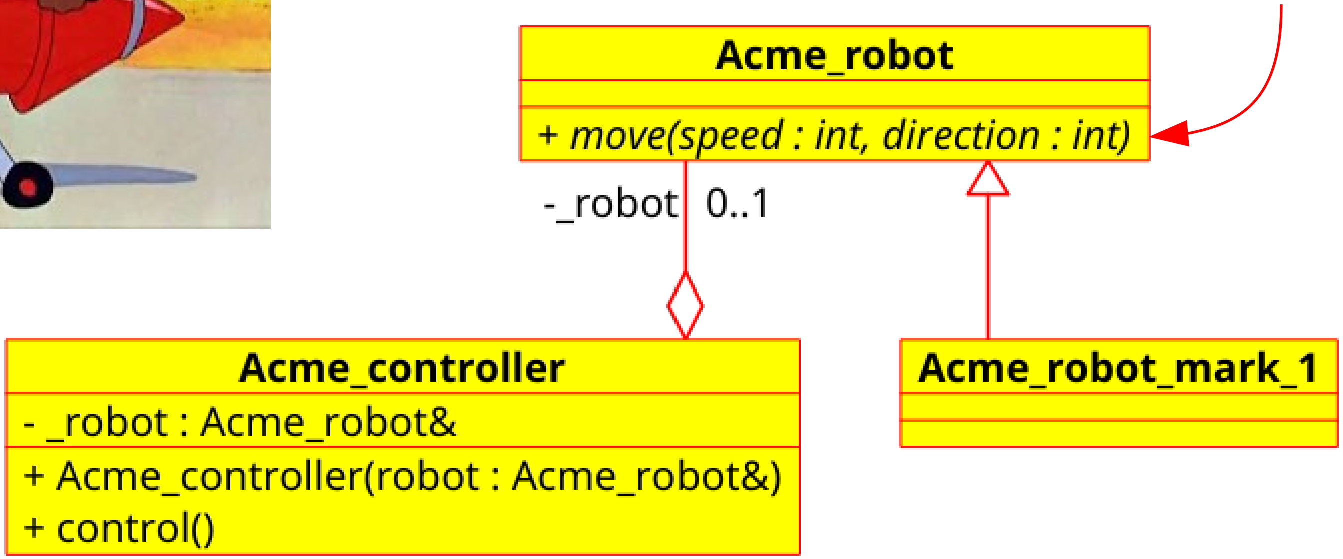
- The Acme controller is better, but the Acme Robot Mark 1 is... iffy



This also ends badly.

(It *always* ends badly...)

New, improved, highly integrated technology!

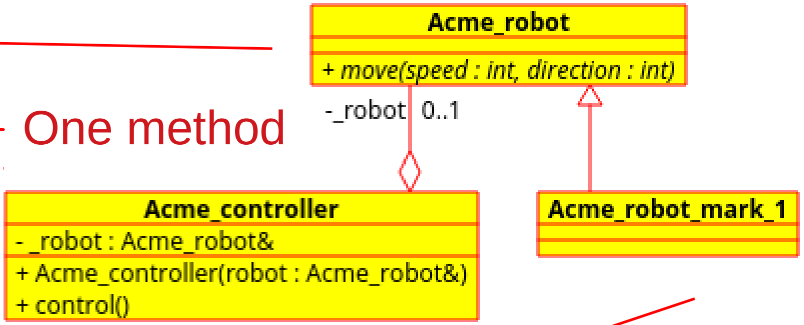


Wile E. Coyote ©1949 by Warner Brothers – Fair use is asserted

The Acme System
with Improved Controller (Client)

```
class Acme_robot {
  public:
    virtual void move(int speed, int direction) = 0;
};

class Acme_controller {
  public:
    Acme_controller(Acme_robot& robot);
    void control();
  private:
    Acme_robot& _robot;
};

class Acme_robot_mark_1 : public Acme_robot {
  public:
    virtual void move(int speed, int direction) override;
};

int main() {
  Acme_robot_mark_1 robot{};
  Acme_controller controller{robot};
  controller.control();
}
```

One method



Acme_robot

+ *move(speed : int, direction : int)*

-_robot 0..1

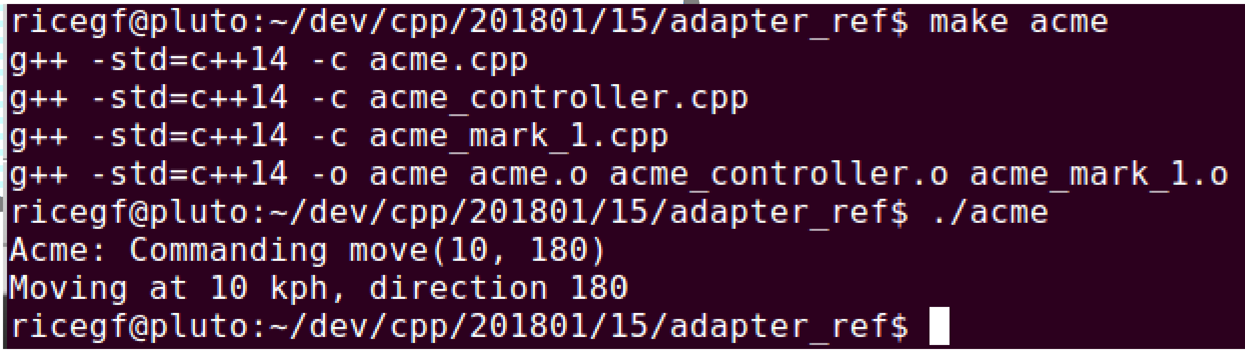
Acme_controller

- _robot : Acme_robot&

+ Acme_controller(robot : Acme_robot&)

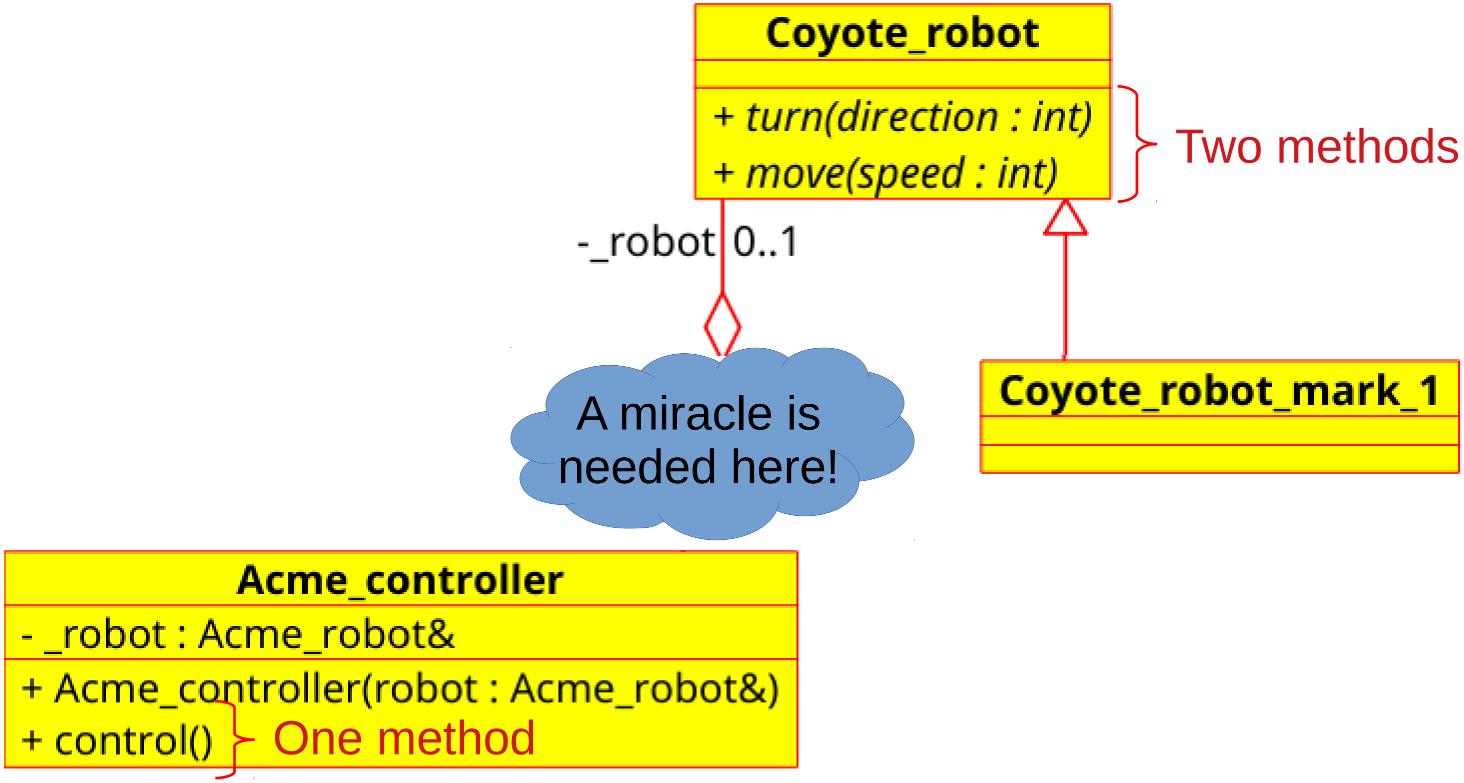
+ control()

Acme_robot_mark_1

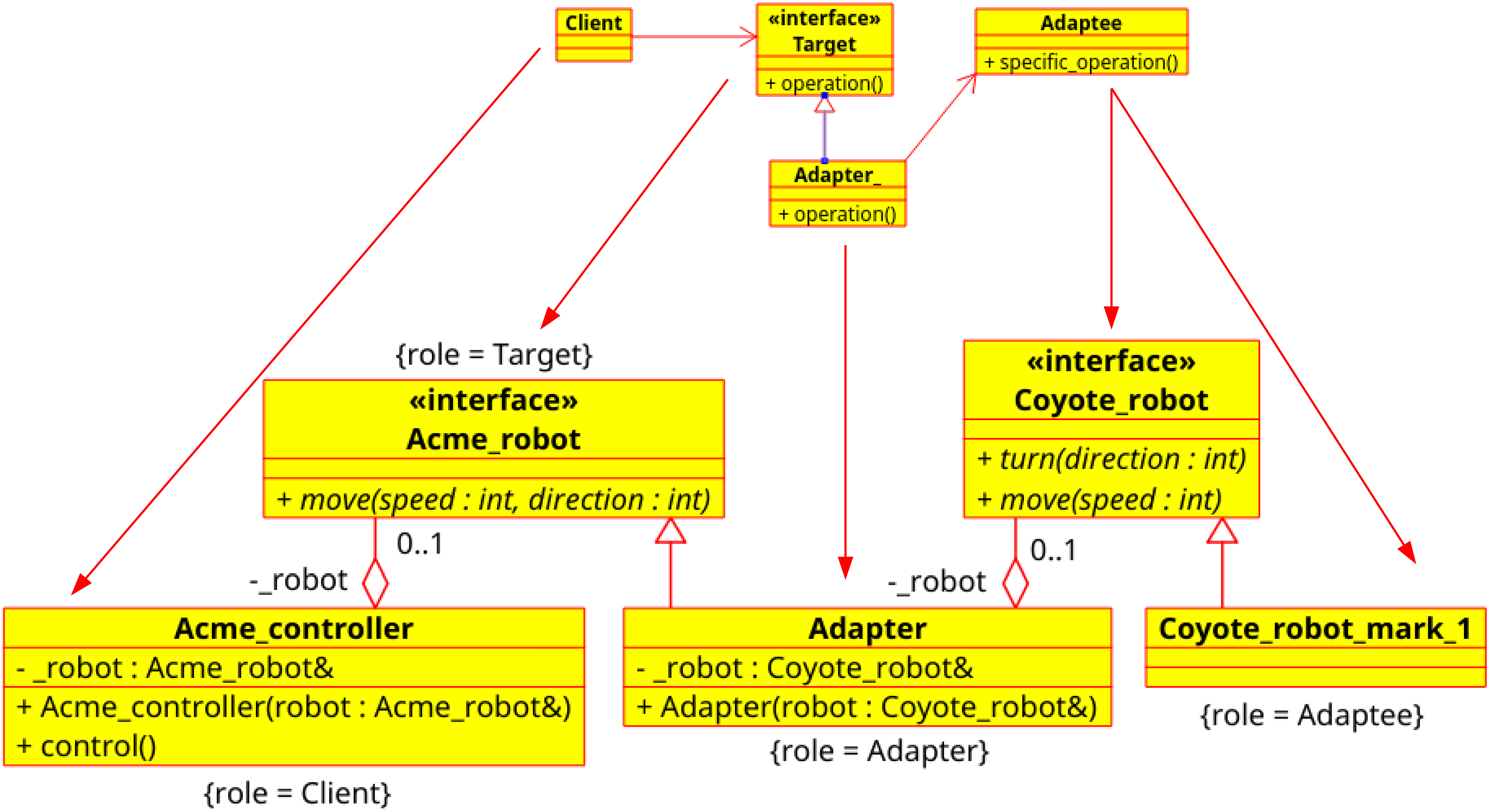


```
ricegf@pluto:~/dev/cpp/201801/15/adapter_ref$ make acme
g++ -std=c++14 -c acme.cpp
g++ -std=c++14 -c acme_controller.cpp
g++ -std=c++14 -c acme_mark_1.cpp
g++ -std=c++14 -o acme acme.o acme_controller.o acme_mark_1.o
ricegf@pluto:~/dev/cpp/201801/15/adapter_ref$ ./acme
Acme: Commanding move(10, 180)
Moving at 10 kph, direction 180
ricegf@pluto:~/dev/cpp/201801/15/adapter_ref$ 
```

The Challenge: Control the Coyote Mark 1 with the Acme Controller

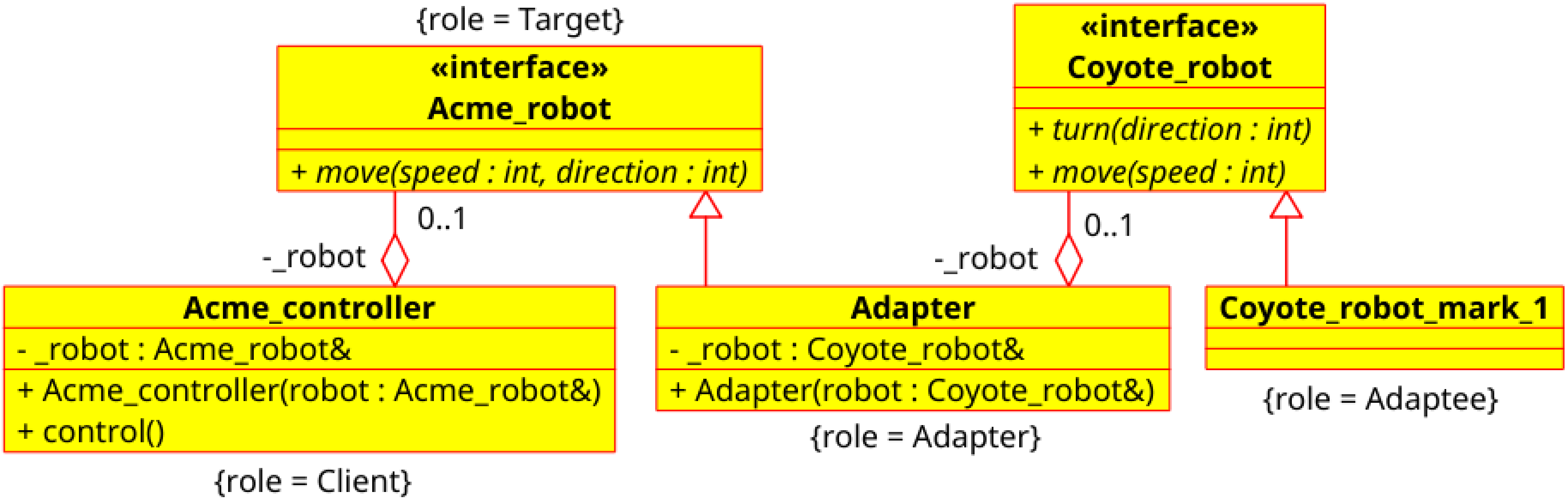


Cue the Adapter!



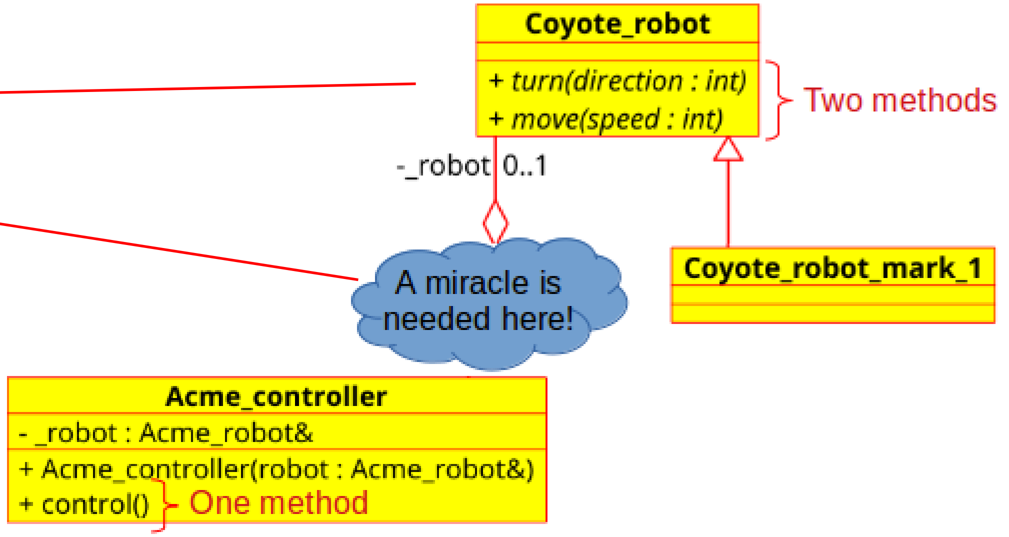
Some Observations

- No existing class changes – not one line!
  - Only the Adapter class is added, and main updated to instance an Adapter
- Adapter “is a” Acme_robot
  - So Acme_controller can “control” it
- Adapter can reference and control a Coyote_robot
  - So it can control the Coyote_robot_mark_1
- Adapter connects **any** Acme-compatible controller to **any** Coyote-compatible robot



The Acme System with Improved Controller (Client)

```
#include "acme_robot.h"
#include "coyote_robot.h"

class Adapter : public Acme_robot {
  public:
    Adapter(Coyote_robot& robot);
    virtual void move(int speed,
                      int direction) override;
  private:
    Coyote_robot& _robot;
};

Adapter::Adapter(Coyote_robot& robot) : _robot{robot} { }

void Adapter::move(int speed, int direction) {
  _robot.turn(direction);
  _robot.move(speed);  // move S at 10 kph
}

int main() {
  Coyote_robot_mark_1 robot{};
  Adapter adapter{robot};
  Acme_controller controller{adapter};
  controller.control();
}
```



Coyote_robot

+ *turn(direction : int)*
+ *move(speed : int)*

Two methods

-_robot 0..1

A miracle is needed here!

Coyote_robot_mark_1

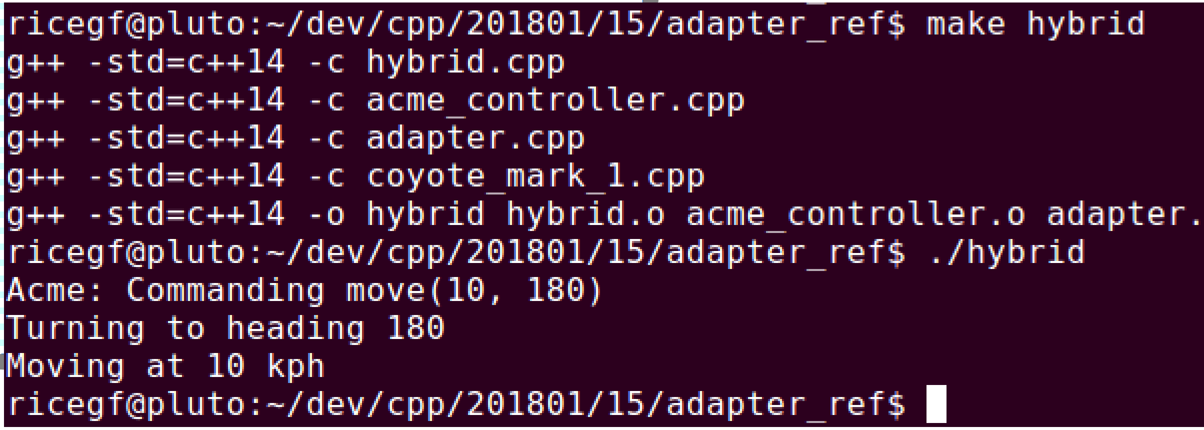
Acme_controller

- _robot : Acme_robot&

+ Acme_controller(robot : Acme_robot&)
+ control()

One method

You need to UNDERSTAND this page!



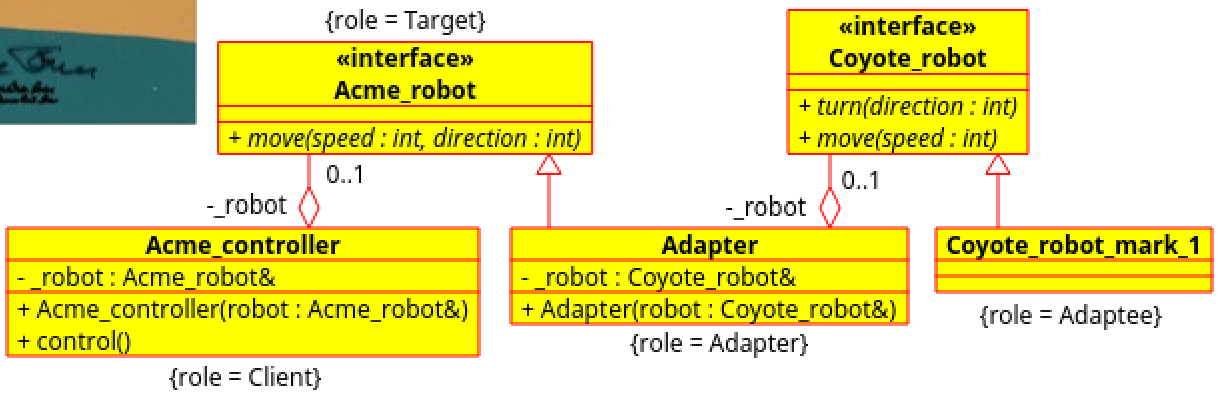
```
ricegf@pluto:~/dev/cpp/201801/15/adapter_ref$ make hybrid
g++ -std=c++14 -c hybrid.cpp
g++ -std=c++14 -c acme_controller.cpp
g++ -std=c++14 -c adapter.cpp
g++ -std=c++14 -c coyote_mark_1.cpp
g++ -std=c++14 -o hybrid hybrid.o acme_controller.o adapter.
ricegf@pluto:~/dev/cpp/201801/15/adapter_ref$ ./hybrid
Acme: Commanding move(10, 180)
Turning to heading 180
Moving at 10 kph
ricegf@pluto:~/dev/cpp/201801/15/adapter_ref$ 
```

Result: Coyote Uses Acme Controller with Coyote Robot Mark 1

- Adapter satisfies the Acme_robot interface while translating to the Coyote_robot interface

This *also* ends badly.

(It ***always, always*** ends badly...)



{role = Target}

«interface»
Acme_robot

+ *move(speed : int, direction : int)*

«interface»
Coyote_robot

+ *turn(direction : int)*
+ *move(speed : int)*

0..1
-_robot

0..1
-_robot

Acme_controller
- _robot : Acme_robot&
+ Acme_controller(robot : Acme_robot&)
+ control()

{role = Client}

Adapter
- _robot : Coyote_robot&
+ Adapter(robot : Coyote_robot&)

{role = Adapter}

Coyote_robot_mark_1

{role = Adaptee}

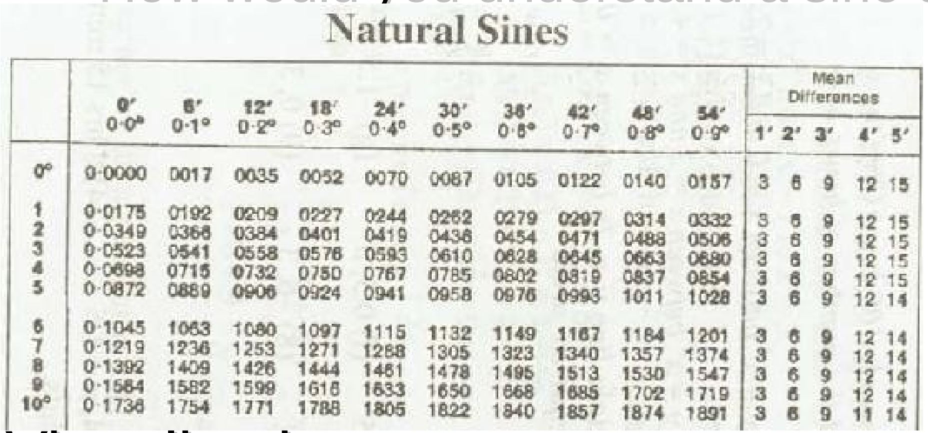
Wile E. Coyote and Road Runner ©1949 by Warner Brothers –
Fair use is asserted

Simple Adapters with Lambdas

```
#include<cmath>        // standard mathematical functions

Function s4( [ ] (double x) {return fmod(x, 16.0); },
             -10,11,orig,400,20,20);
Function s5( [ ] (double x) {return rand(); },
             -10, 11, orig, 400, 20, 20);
```

Why Plot?

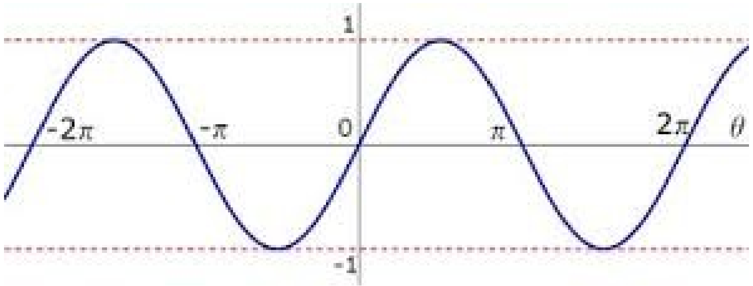
- Because you can see things in a graph that are not obvious from a set of numbers
  - How would you understand a sine curve if you had never seen one?



Natural Sines

	0′ 0·0°	6′ 0·1°	12′ 0·2°	18′ 0·3°	24′ 0·4°	30′ 0·5°	36′ 0·6°	42′ 0·7°	48′ 0·8°	54′ 0·9°	Mean Differences				
											1′	2′	3′	4′	5′
0°	0·0000	0017	0035	0052	0070	0087	0105	0122	0140	0157	3	6	9	12	15
1	0·0175	0192	0209	0227	0244	0262	0279	0297	0314	0332	3	6	9	12	15
2	0·0349	0366	0384	0401	0419	0436	0454	0471	0488	0506	3	6	9	12	15
3	0·0523	0541	0558	0576	0593	0610	0628	0645	0663	0680	3	6	9	12	15
4	0·0698	0715	0732	0750	0767	0785	0802	0819	0837	0854	3	6	9	12	15
5	0·0872	0889	0906	0924	0941	0958	0976	0993	1011	1028	3	6	9	12	14
6	0·1045	1063	1080	1097	1115	1132	1149	1167	1184	1201	3	6	9	12	14
7	0·1219	1236	1253	1271	1288	1305	1323	1340	1357	1374	3	6	9	12	14
8	0·1392	1409	1426	1444	1461	1478	1495	1513	1530	1547	3	6	9	12	14
9	0·1564	1582	1599	1616	1633	1650	1668	1685	1702	1719	3	6	9	12	14
10°	0·1736	1754	1771	1788	1805	1822	1840	1857	1874	1891	3	6	9	11	14

VS



- Visualization
  - Is key to understanding in many fields
  - Is used in most research and business areas
    - Science, medicine, business, telecommunications, control of large systems
  - Can communicate large amounts of data simply

Images are in the public domain

Back to Our Library
Defining Polygons

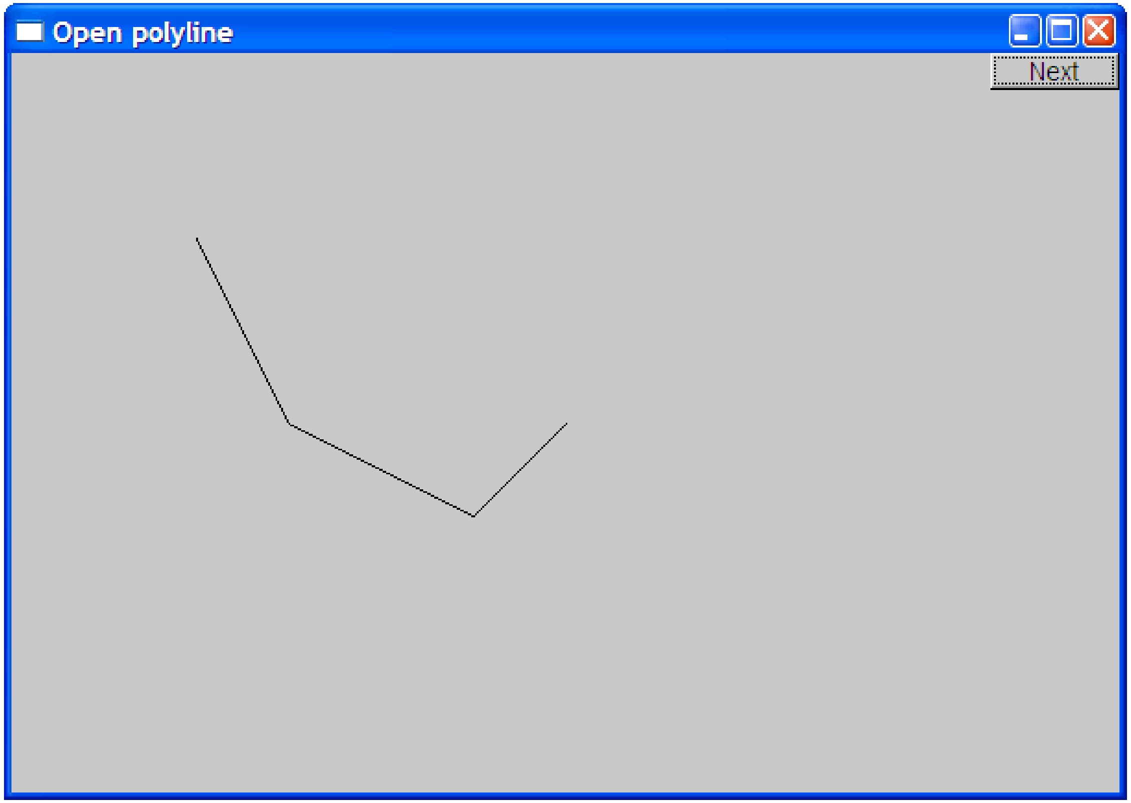
```
class Open_polyline : Shape {         // open sequence of lines
    void add(Point p) { Shape::add(p); }
};

struct Closed_polyline : Open_polyline { // closed sequence of lines
};

struct Polygon : Closed_polyline {    // closed seq of non-intersecting lines
    void add(Point p);
};
```

Note that a Closed_poly**line** is not necessarily a Poly**gon**.
A Polygon is a Closed_polyline where no lines cross.

Thus, a Polygon has stronger invariant rules than a Closed_polyline.

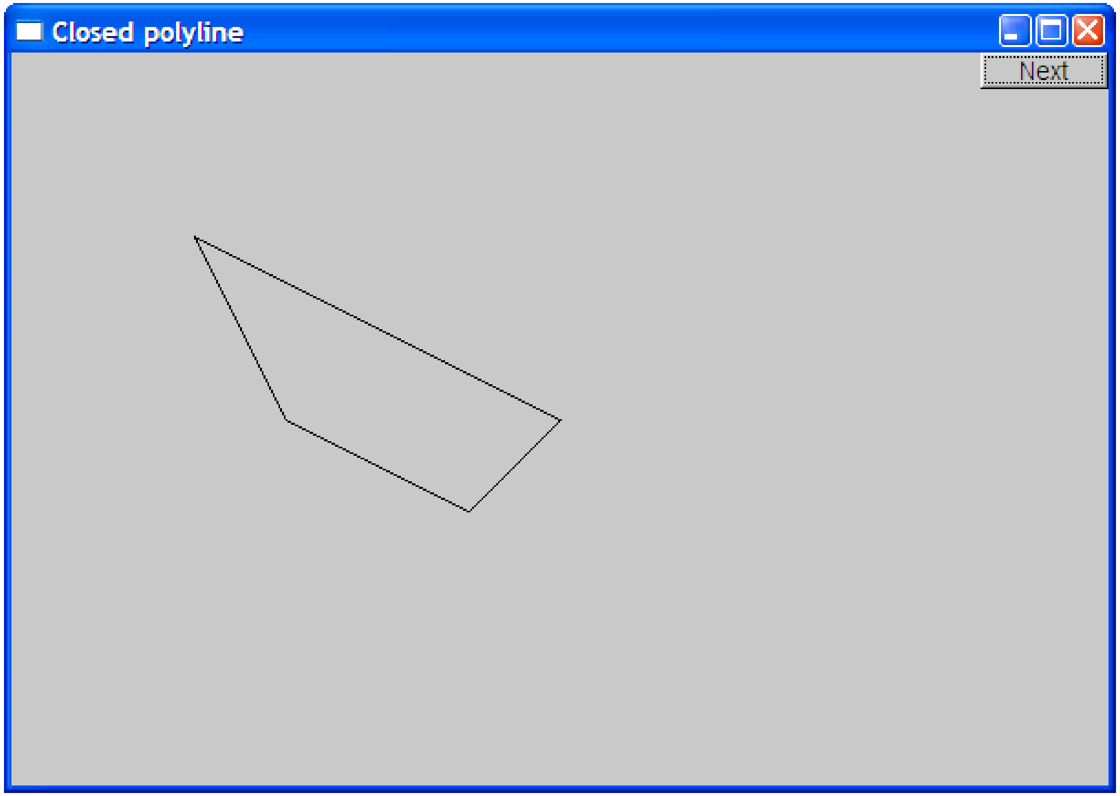
Open_polyline

```
Open_polyline opl;
opl.add(Point(100,100));
opl.add(Point(150,200));
opl.add(Point(250,250));
opl.add(Point(300,200));
```



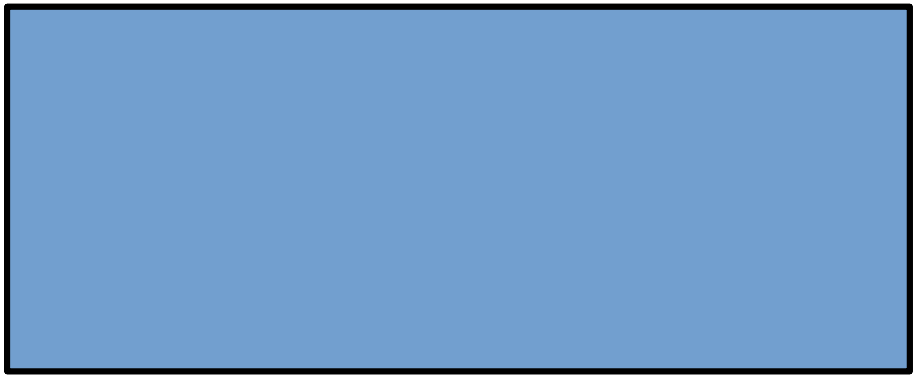
Closed_polyline

```
Closed_polyline cpl;
cpl.add(Point(100,100));
cpl.add(Point(150,200));
cpl.add(Point(250,250));
cpl.add(Point(300,200));
```



Question: What’s a Rectangle?

- Is a Rectangle a Polygon with 4 points?
- Or a Closed_polyline with 4 points?
- Or a Shape with 2 points and parallel opposite sides?
- Or something else?

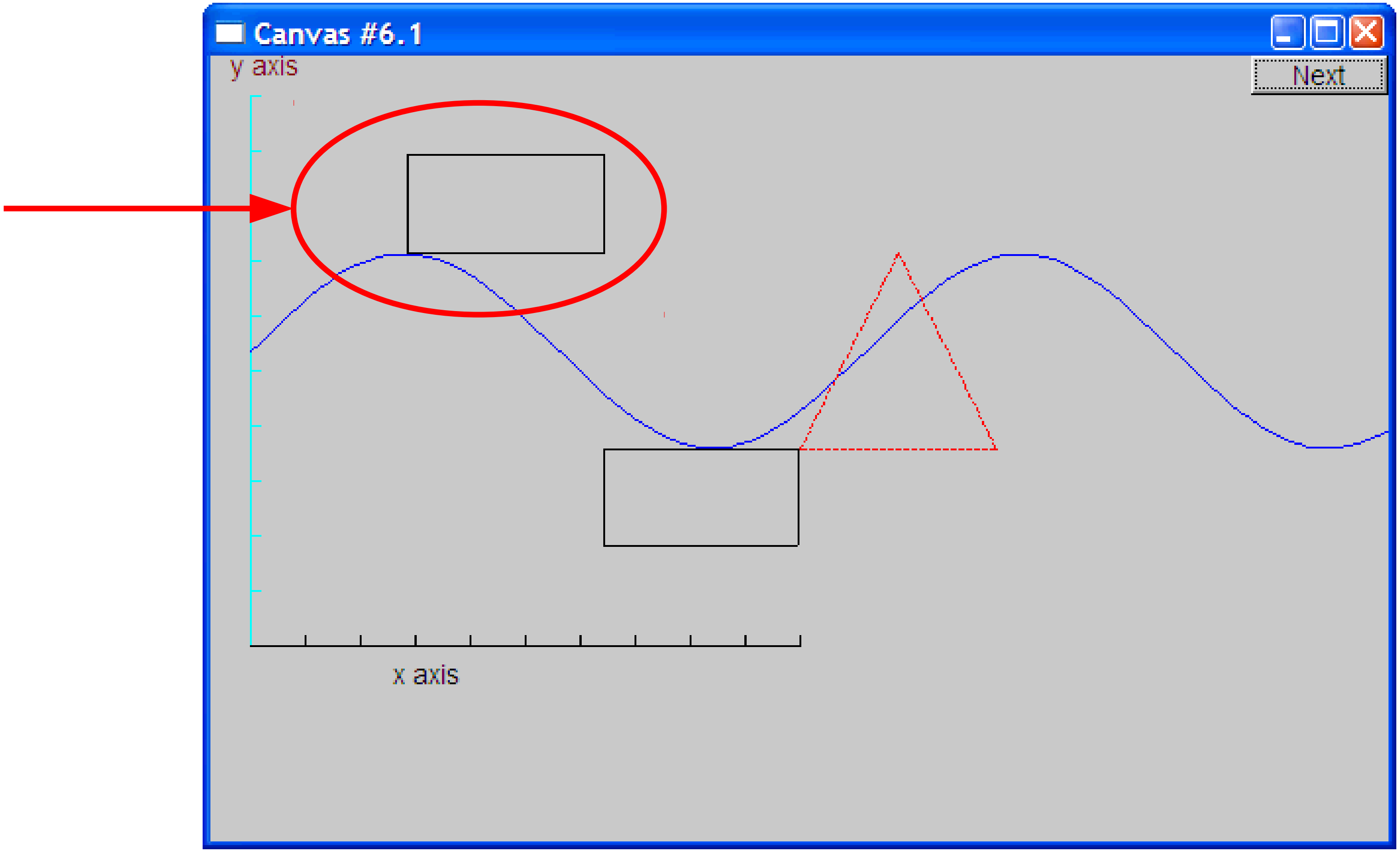


Prototype Implementation for Rectangle : Closed_polyline

- Add a shape that looks like a rectangle

```
Closed_polyline poly_rect;
poly_rect.add(Point(100,50));
poly_rect.add(Point(200,50));
poly_rect.add(Point(200,100));
poly_rect.add(Point(100,100));

win.set_label("Canvas #6.1");
```

Add a Rectangle-Looking Shape



But is it a rectangle?

Transform Rectangle-ish Shape to a Non-Rectangle

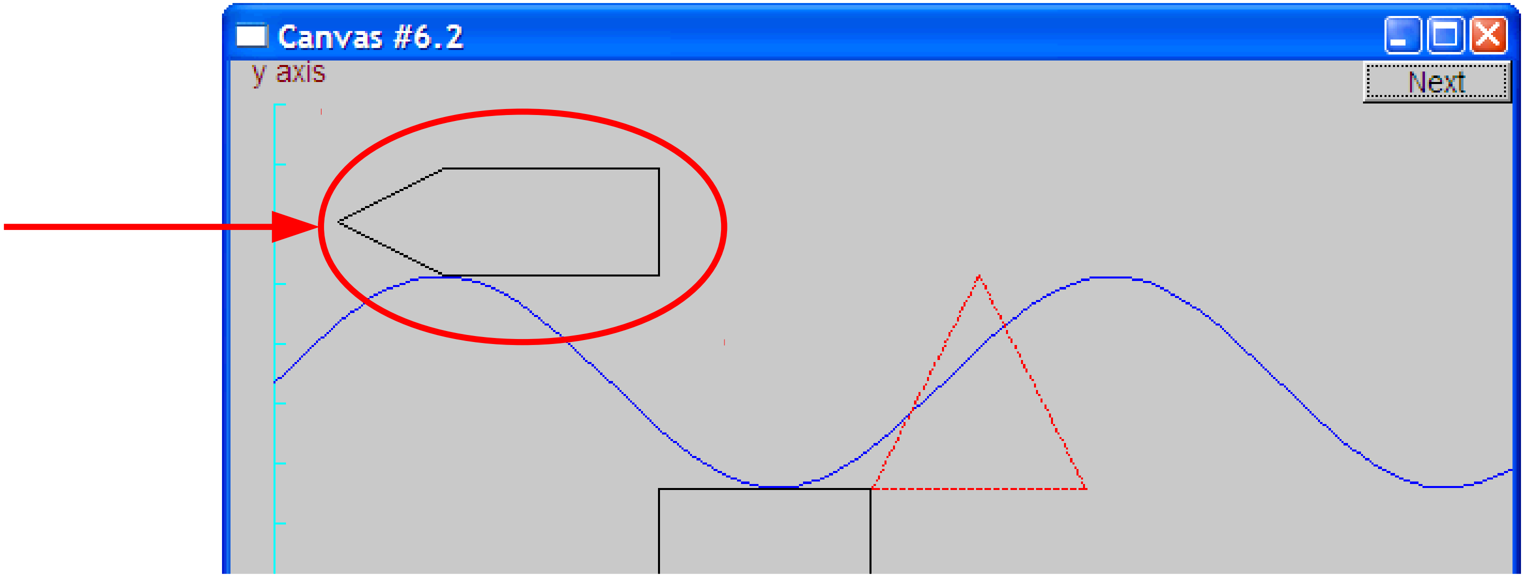
- We can add a point

```
poly_rect.add(Point(50,75));      // now poly_rect has 5 points
```

  - But we could override and null add()

```
void add(Point p) override { }; // rm add from Rectangle : Polygon
```

Obviously a Polygon
Obviously NOT a Rectangle



- “looking like” is not the same as “is”
  - Enforced data constraints are trustworthy
  - “Please use it this way” isn't worth the electrons wasted on the request

Bugs are Inevitable. How do you **manage** them?

Bug Management

- Bug trackers offer end-to-end bug management
  - Create bug reports (even by end users!)
  - Triage bug reports: **Verified** (needs to be fixed), **Could Not Duplicate** (developers can't verify that bug), **Duplicate** (already reported, close with link to original), **Won't Fix** (it's a feature, not a bug)
  - Prioritize bug reports by Severity: **Showstopper**, **Critical**, **Severe**, **Important**, **Informational**
    - Showstopper and (usually) Critical bugs must be fixed prior to release
  - Report progress on and close out bug reports with notification
- A <u>verified</u> bug becomes a task on your Sprint Backlog
  - It's priority is determined in part by its severity
  - Fixing a bug means not implementing a new feature
- Innumerable bug trackers exist
  - Atlassian JIRA, Bugzilla, IBM ClearQuest, Excel (ahem), etc.

Defining Rectangles : Shape

```
class Rectangle : public Shape {
  public:
    Rectangle(Point xy, int ww, int hh) : w(ww), h(hh) {
        add(xy);
        if (h<=0 || w<=0)
          throw runtime_error("Bad rectangle: non-positive side");
    }
    Rectangle(Point x, Point y) : w(y.x-x.x), h(y.y-x.y) {
        add(x);
        if (h<=0 || w<=0)
          throw runtime_error("Bad rectangle: non-positive width or height");
    }

    int height() const { return h; }
    int width() const { return w; }
  private:
    int h;    // height
    int w;    // width
};
```

Constructors may be overloaded.
Here we define a rectangle using two points (opposite corners), or using one point with a width and height.

Defining Triangles?

Why a Rectangle class but not a Triangle class?

Selecting the “right” classes and “optimum” system design is the role of an Architect.

Expertise
/eks-per-tyz/
def: Special skill or knowledge in a particular subject, eg. He has expertise in his field of molecular scien

Architecture – The set of implementation elements and associated integration mechanisms necessary to meet the system requirements.

Architect – The engineering role that specifies a system's architecture.

Defining Ellipses

```
class Ellipse : public Shape {
    Ellipse(Point p, int w, int h) // center, min, and max distance from center
        : w(w), h(h) {
        add(Point(p.x-w,p.y-h));
    }

    void draw_lines() const;

    Point center() const { return Point(point(0).x+w,point(0).y+h); }
    Point focus1() const {
      return Point(center().x+int(sqrt(double(w*w-h*h))),center().y);
    }
    Point focus2() const {
      return Point(center().x-int(sqrt(double(w*w-h*h))),center().y);
    }

    void set_major(int ww) { w=ww; }
    int major() const { return w; }
    void set_minor(int hh) { h=hh; }
    int minor() const { return h; }
private:
    int w;
    int h;
};
```

Defining Circles

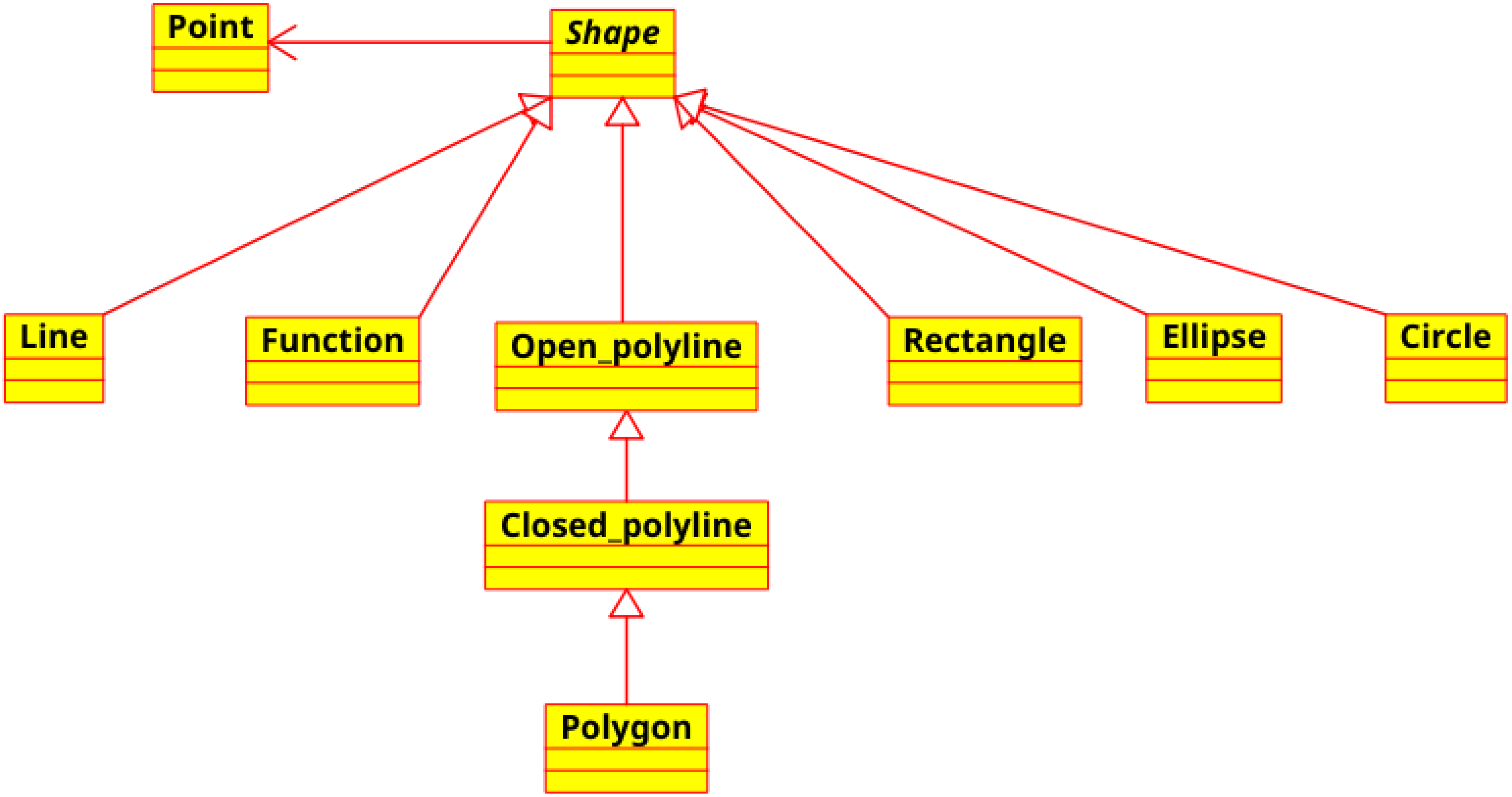
```
class Circle : Shape {
  public:
    Circle(Point p, int rr);    // center and radius
    void draw_lines() const;
    Point center() const ;
    int radius() const { return r; }
    void set_radius(int rr) { r=rr; }
private:
    int r;
};
```

How else could Circle be defined?

- What would be the advantages?
- What would be the disadvantages?

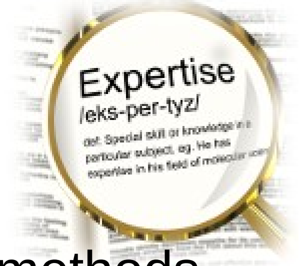
Baseline Class Hierarchy

- We chose to use a simple (and reasonably shallow) class hierarchy for our shapes
  - Based on the root abstract class Shape



Ideals

- Our ideal of program design is to represent the concepts of the application domain directly in code.
  - If you understand the application domain, you understand the code, and *vice versa*. For example:
    - **Point** – a coordinate point
    - **Shape** – what's common for all shapes in our Graph/GUI view of the world
    - **Line** – a line as you see it on the screen
    - **Rectangle** – A true rectangel
    - **Color –** as you see it on the screen – but color is a real rat's nest!
- **Shape** is different in that it is a *generalization*.
  - You can't make an object that's "just a Shape"



Generalization is the process of extracting shared characteristics from two or more classes, and combining them into a generalized base class. Shared characteristics can be attributes, associations, or methods.

Generalization vs Specialization

- Generalization creates a base class from shared characteristics of two or more classes, making the latter derived classes
  - Sometimes called “bottom-up design”
- Specialization creates derived classes from a base class by repeatedly adding the unique characteristics of each
  - Sometimes called “top-down design”

Either is “correct”

Use whichever gets you the best design in the least amount of time

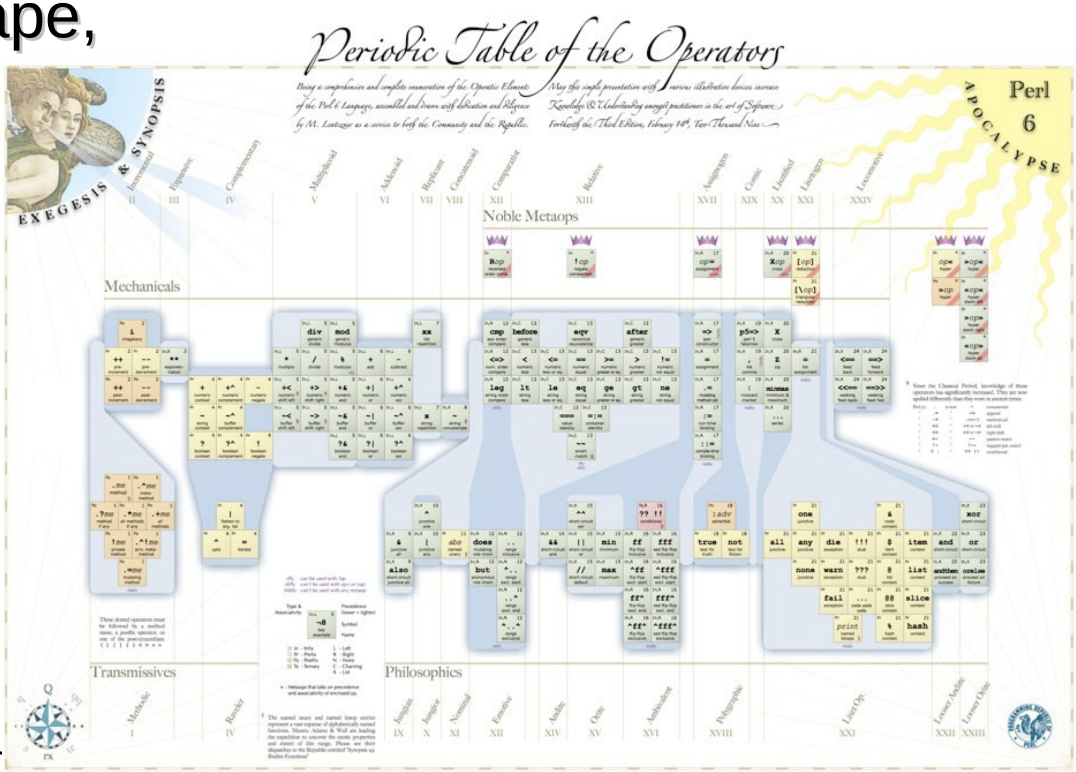
Logically identical operations should have the same name

- For every class,
  - **draw_lines()** does the drawing
  - **move(dx,dy)** does the moving
  - **add(p)** adds some **p** (*e.g.*, a point)
- For every attribute **x** of a Shape,
  - **x()** gives its current value and
  - **set_x()** gives it a new value
  - e.g.,

```
Color c = s.color();
s.set_color(Color::blue);
```

Consistency fosters easier learning
and less buggy implementations

The Perl 6
“Periodic Table of the Operators”
emphasizes the consistency
of the operator set
defined for the new version of Perl

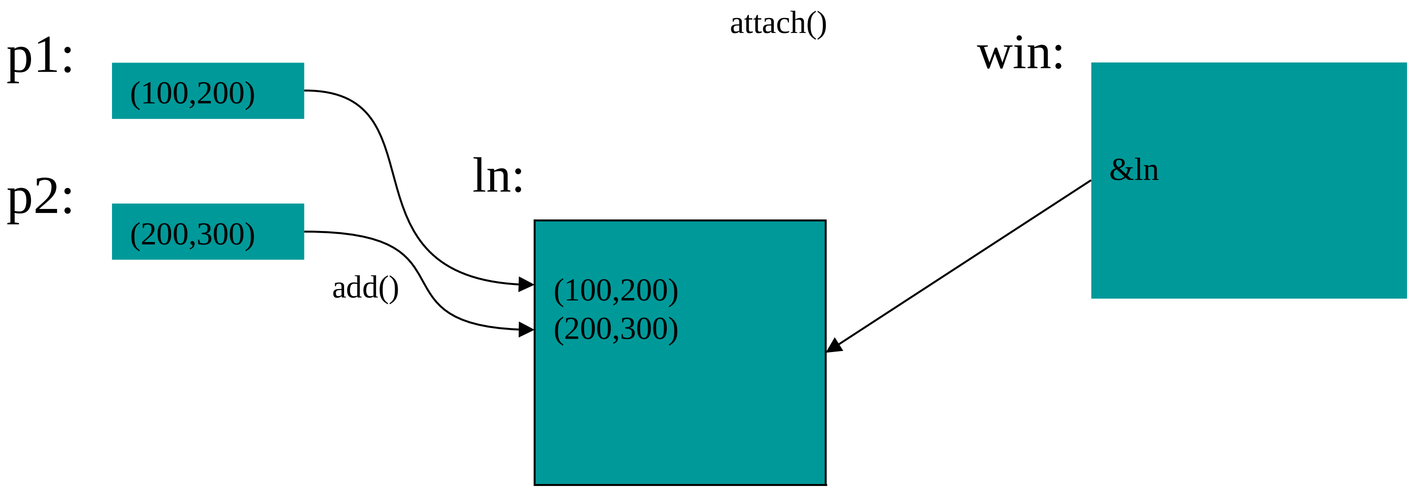


Periodic Table of the Perl 6 Operators by Mark Lentczner is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License.
http://ozonehouse.com/mark/periodic/

Logically different operations have different names

```
Lines ln;
Point p1(100,200);
Point p2(200,300);
ln.add(p1,p2);                // add points to ln (make copies)
win.attach(ln);               // attach ln to window
```

- Why not **win.add(ln)**?
  - **add()** copies information; **attach()** just creates a reference
  - we can change a displayed object after attaching it, but not after adding it



Expose uniformly

- Data should virtually ALWAYS be private (const is the exception)
  - Data hiding – so it will not be changed inadvertently
  - Use **private** data, and pairs of public access functions to get and set the data

```
c.set_radius(12);                     // set radius to 12
c.set_radius(c.radius()*2);           // double the radius (fine)
c.set_radius(-9);                     // set_radius() could check for negative,
double r = c.radius();        // returns value of radius
c.radius = -9;                // error: radius is a method (good!)
c.r = -9;                     // error: r is private (good!)
```

- Our methods can be **private**, **protected**, or **public**
  - Public for interface (note that friend functions are effectively public)
  - Protected for methods used in the class and its derived classes
  - Private for methods used only internally to a class

What does “private” buy us?

- We can change our implementation after release
- We don’t expose gtkmm types used in the library to our users
  - We could replace gtkmm with another library without affecting user code
- We could provide checking in access functions
- Functional interfaces can be nicer to read and use
  - E.g., **s.add(x)** rather than **s.points.push_back(x)**
- We enforce immutability of shape
  - Only color and style change; not the relative position of points
  - **const** member functions
- The value of this “encapsulation” varies with application domains
  - Is often most valuable
  - Is the ideal
    - i.e., hide representation unless you have a good reason not to

“Regular” interfaces

```
Line ln{Point{100,200},Point{300,400}};
Mark m{Point{100,200}, 'x'};     // display a single point as an 'x'
Circle c{Point{200,200},250};

// Alternative (not supported):
Line ln2{x1, y1, x2, y2};                          // from (x1,y1) to (x2,y2)

// How about? (not supported):
Rectangle s1{Point{100,200},200,300       // width==200 height==300
Rectangle s2{Point{100,200},Point{200,300}};   // width==100 height==100

Rectangle s3{100,200,200,300}; // is 200,300 a point or a width plus a height?
```

A library

- A collection of classes and functions designed to be used together effectively is a Class Library (still!)
  - Building blocks for applications
  - Build more such “building blocks” - evolving and / or forking

- A good library models some aspect of a domain
  - It doesn’t try to do everything
  - Our library aims at simplicity and small size for graphing data and for very simple GUI

- We can’t define each library class and function in isolation
  - A good library exhibits a uniform style (“regularity”)

A second distinct development path for a product is called a **Branch** if followed by the **same** company or project, and a **Fork** if followed by a **different** company or project.

Expertise
/eks-per-tyz/
def. Special skill or knowledge in a particular subject, eg. He has expertise in his field of molecular

Basic Branching in git

- You create a local branch using ‘-b’

```
git checkout -b bug_013  # Fix bug #13
```

- While in (branch) bug_013, commits are made *to that branch*, not (branch) master
  - You can switch back to “master” using

    ```
    git checkout master
    ```

  - Then back to bug_013 using

    ```
    git checkout bug_013
    ```

- You can merge your bug fix back into master

```
git checkout master
git merge bug_013
```

Learn (a Lot!) More

https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging

Examples of Product Forks

- Netscape → Firefox
- Debian → Ubuntu → Mint
- Unix System V → BSD Unix → Mac OS X
- MySQL → MariaDB
- OpenOffice → LibreOffice

Quick Review

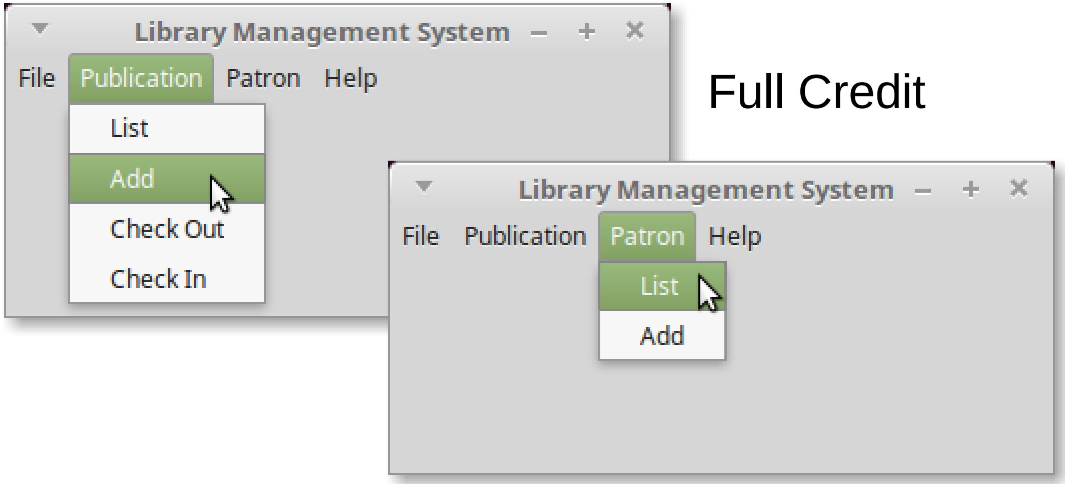
- A ____ ____is a collection of prewritten classes, often designed as templates, that work together to facilitate writing applications.
  - Name some popular ones for C++.
- True or False: Multiple namespace blocks with the same name are permitted.
- A ________ defines an alias for an (often complex) type, simplifying code and increasing readability.
- The ______ ______implements a bridge between two classes with incompatible interfaces.
- The set of implementation elements and associated integration mechanisms necessary to meet the system requirements is called the system's ___________. The engineering role primarily responsible for it is the ______.
- ______________ is the process of extracting shared characteristics from two or more classes, and combining them into a generalized base class.

Quick Review

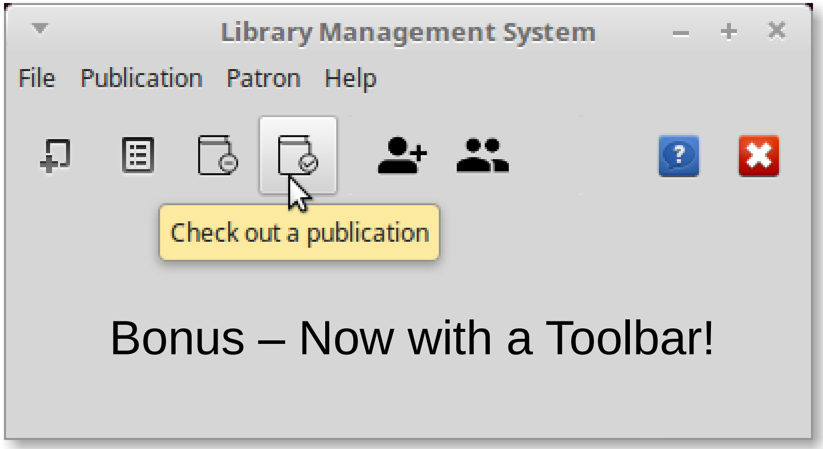
- An anonymous function object is called a ________. They have a _______ _______ to determine which outer scope variables are visible, an optional parameter list, and a body.
- True or False: The parameter list of a lambda function, including the parentheses, are optional.
- True or False: By default, a lambda function can access all variables from the enclosing scope.
- True or False: By default, a captured variable is copied. Prepend an & to capture the variable by reference.
- Describe how can all variables in the surrounding scope can be captured, first by reference, and then by access (copied).
- While the return type of a lambda function is usually inferred by the compiler, describe how to explicitly define the return type

Sprint #3 / Homework #7

- **Homework #6 is due Thursday, March 22 at 8 am (this is a rare 2-week sprint)**
  - **Replace the menu dialog with a main window**
  - **You have much leeway in your GUI design – simply meet the requirements**
  - **Doing the bonus is *strongly* recommended**



Full Credit



Bonus – Now with a Toolbar!