# More Information

- http://opensource.adobe.com
- http://stepanovpapers.com
  - Specifically:
    - http://www.stepanovpapers.com/eop/lecture_all.pdf
    - http://www.stepanovpapers.com/notes.pdf
    - http://www.stepanovpapers.com/PAM.pdf





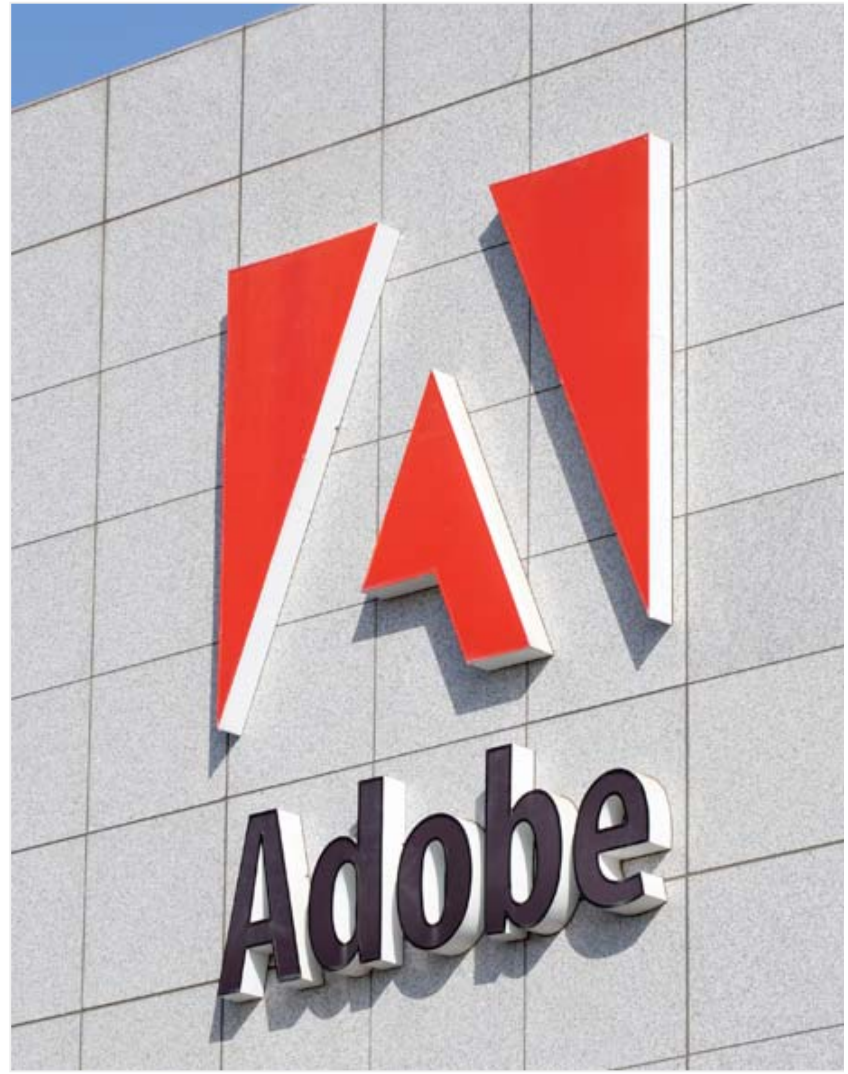
Adobe

# Concept-Based Runtime Polymorphism

Sean Parent

Principal Scientist

May 17, 2007



# Mat's Talk

## Runtime Polymorphic Generic Programming—Mixing Objects and Concepts in ConceptC++

Mat Marcus[1], Jaakko Järvi[2], and Sean Parent[1]

[1] Adobe Systems Inc. {mmarcus|sparent}@adobe.com
[2] Texas A&M University jarvi@cs.tamu.edu

**Abstract.** A long-held goal of software engineering has been the ability to treat software libraries as reusbale components that can be composed with program-specific code to produce applications. The object-oriented programming paradigm offers mechanisms to write libraries that are open for extension, but it tends to impose intrusive interface requirements on the types that will be supplied to the library. The generic programming paradigm has seen much success in C++, partly due to the fact that libraries remain open to extension without imposing the need to intrusively inherit from particular abstract base classes. However, the static polymorphism that is a staple of programming with templates and overloads in C++, limits generic programming's applicability in application domains where more dynamic polymorphism is required. In this paper we present the **poly<>** library, a part of Adobe System's open source library ASL, that combines the object-oriented and generic programming paradigms to provide non-intrusive, transparent, value-based, runtime-polymorphism. Usage, impact on design, and implementation techniques are discussed.

### 1 Introduction

Successful development of robust large scale-software applications depends on upon the ability to combine application-specific functionality with independently developed library modules from a variety of sources, with a reasonable amount of application-specific glue code. To support this activity, modules must remain open for extension but closed for modification [18]. Object-oriented programming and generic programming are the two main paradigms available for creating such modules in C++.

In object-oriented programming, libraries typically specify that the types supplied to the library must be derived from a common abstract base class, providing implementations for a collection of pure virtual functions. The library knows only about the abstract base class interface, but can be "extended" to work with new user types derived from the abstract interface. That is, variability is achieved through differing implementations of the virtual functions in the derived classes. This is how object-oriented programming supports modules that are closed for modification, yet remain open for extension. One strength of this paradigm is its support for varying the types supplied to a module at runtime. Composability of modules is limited, however, since independently produced modules generally do not agree on common abstract interfaces from which supplied types must inherit.





Adobe

# Abstract

- Requirement of Polymorphism
  - Compile Time / Runtime Dichotomy
- The Semantics of Inheritance
  - Modeling
  - Refinement
  - Algorithm Refinement
- Problems with Inheritance
  - Intrusive
  - Reference Semantics
    - Object Management
    - Naming Variance
- The Poly Library
  - Goals
  - The Basics
  - Usage in Adobe Source Libraries
  - Future Directions

# Requirement of Polymorphism

- Apply an algorithm to *similar* types

- Apply an algorithm to a heterogeneous collection of *similar* types

# Requirement of Polymorphism

- Apply an algorithm to *similar* types

- Apply an algorithm to a heterogeneous collection of *similar* types

- Similar types are types which satisfy they same semantic requirements

  *Types are similar if they model the same concept*

# Requirement of Polymorphism

- Apply an algorithm to any type which models a given concept
- Apply an algorithm to a collection of types which model the same concept

# Requirement of Polymorphism

- Apply an algorithm to any type which models a given concept

```
swap(x, y); // where x and y are of type T which models Regular
```

- Apply an algorithm to a collection of types which model the same concept

```
vector<any model of Regular> v = { 10, "Hello", true };
find(v.begin(), v.end(), "Hello");
```

# Compile Time / Runtime Dichotomy

- Apply an algorithm to any type which models a given concept
  - Templates work when T is known at compile time - OOP techniques if T is not known
- Apply an algorithm to a collection of types which model the same concept
  - Types cannot be fixed at compile time - OOP techniques required

# Compile Time / Runtime Dichotomy

- Apply an algorithm to any type which models a given concept

```
swap(x, y); // works for object pointers too!
```

- Apply an algorithm to a collection of types which model the same concept

```
vector<object*> v = { new integer(10), new string("Hello"), new boolean(true) };
find_if(v.begin(), v.end(), bind(&object::equals, new string("Hello"), _1));


vector <int> v = { 1, 2, 3 };
find(v.begin(), v.end(), 2);
```

# The Semantics of Inheritance - Concept Definition

- A virtual base class defines a concept:

```
class object {
  public:
        virtual ~object() = 0;
        virtual type_info& get_class() const = 0;
        virtual object* clone() const = 0;
        virtual bool equals(const object*) const = 0;
  };
```

- This base object type corresponds with the Regular concept

# The Semantics of Inheritance - Modeling

- We define a model with inheritance:

```
class boolean : public object {
   public:
        ~object() { };
        type_info get_class() const { return typeid(bool); }
        integer* clone() const { return new boolean(member); }
        bool equals(const object* x) const
        { return x.get_class() == get_class()
             && dynamic_cast<const boolean*>(x)->member == member; }
   private:
        bool member;
   };
```

- *"is a"* means T is a model of concept C





Adobe

# The Semantics of Inheritance - Refinement

- We use virtual inheritance as refinement:

```
class incrementable : public virtual object {
  public:
        virtual void next() const = 0;
  };


class fast_incrementable : public virtual incrementable {
  public:
        virtual void next(size_t n) const = 0;
  };
```

# The Semantics of Inheritance - Algorithms Refinement

- We can dispatch at runtime based on the concept category:

```
void advance(fast_incrementable* x, size_t count = 1) {
        x->next();
    }

void advance(incrementable* x, size_t count = 1) {
        fast_incrementable* derived = dynamic_cast<fast_incrementable*>x;
        if (derived) advance(derived);
        else while (count != 0) x->next();
    }
```

# Problems with Inheritance - Intrusive

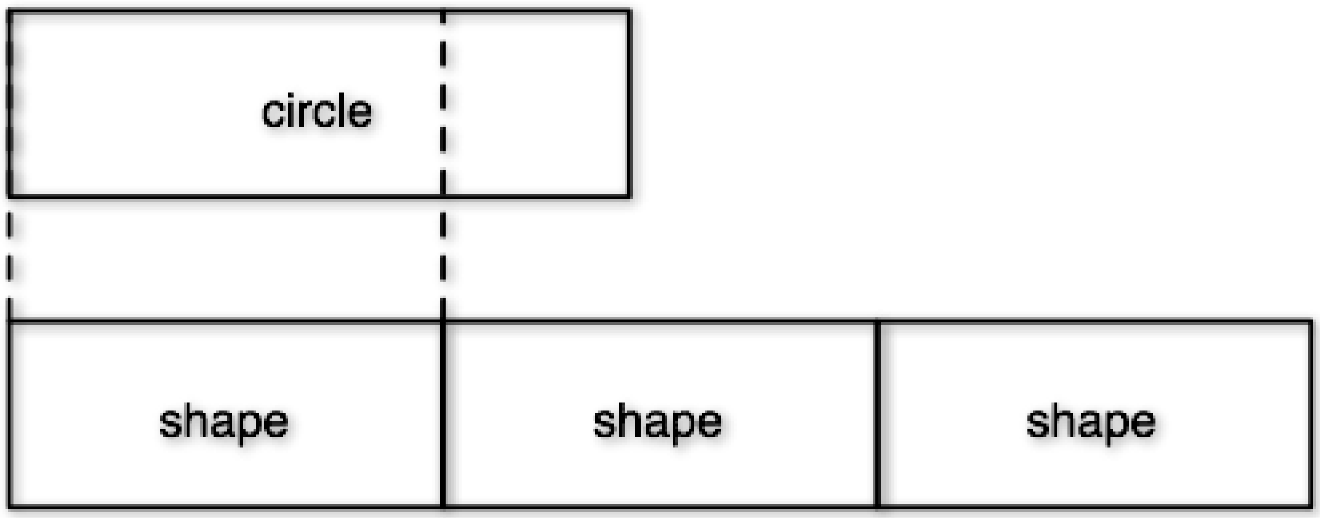
- Inheritance requires modification or wrapping of a class
- Wrapping requires an additional level of indirection through a virtual table
- The requirements of an object come from algorithms
  - imposing requirements of use on the object entangles the object with the application



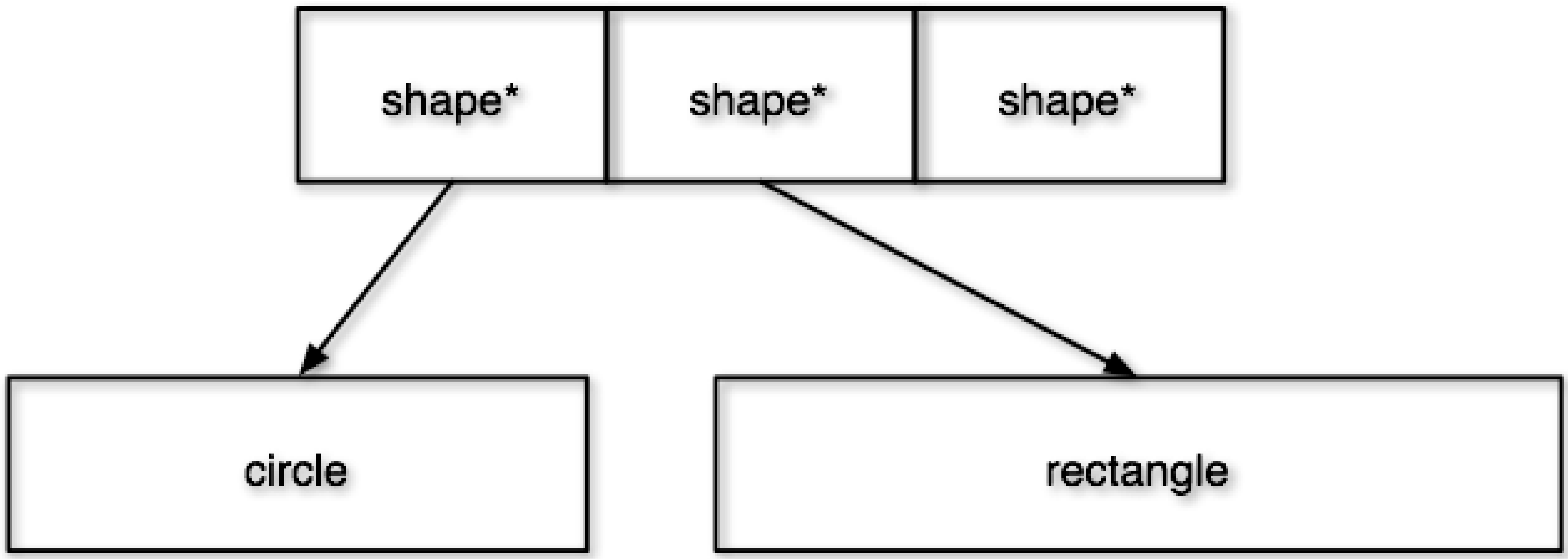


Adobe

# Problems with Inheritance - Reference Semantics



# Problems with Inheritance - Reference Semantics








Adobe

# Problems with Inheritance - Reference Semantics

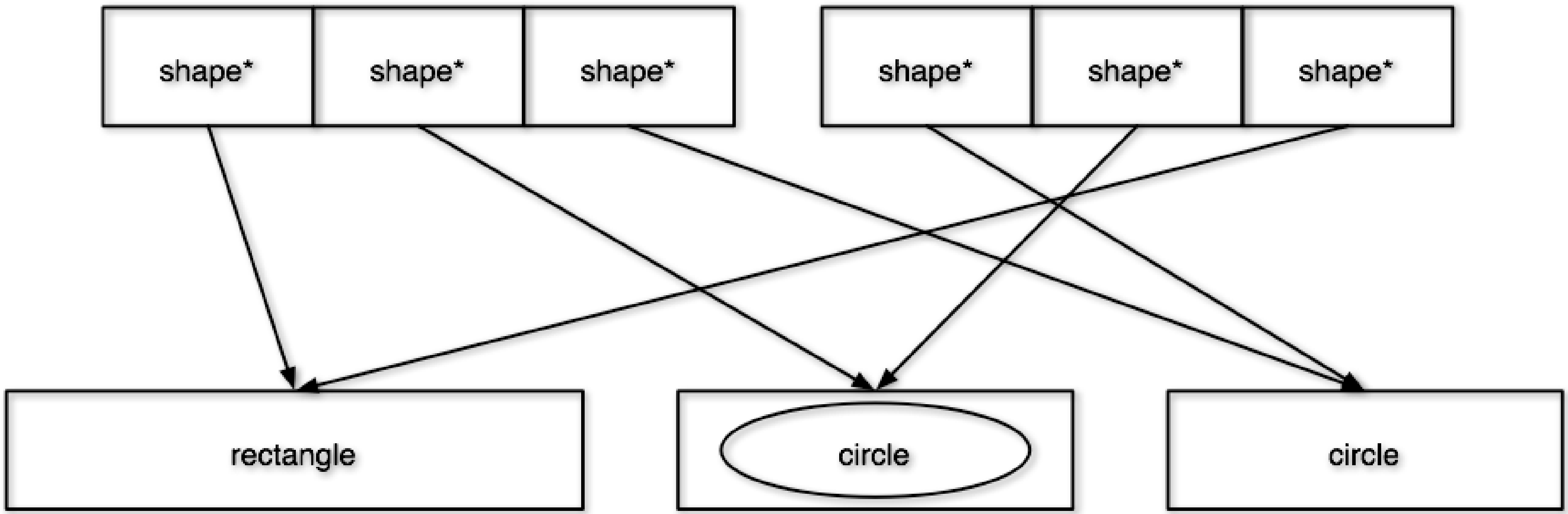
- A polymorphic use of an object imposes the burden of reference semantics on all users of the class
  - Memory management
    - reference counted pointers
    - garbage collection
- Memory management only manages the destruction of the shared object
  - All mutable operations on the object must be managed
  - Threading further complicates the management issue
- Shared writable references make reasoning about code difficult

*"A shared pointer is as good as a global variable."*

# Problems with Inheritance - Reference Semantics

```
/*...*/
vector<shape*> s2(s1);
reverse(s1.begin(), s1.end());

(*find_if(s1.begin(), s1.end(), bind(&object::equals, new circle(10), _1)))
                 ->move(point(10, 20));
```






Adobe

# Problems with Inheritance - Naming Variance

- Compare two non-polymorphic value

```
a == b
```

- Compare two polymorphic values

```
a->equals(b)
```

- The difference in naming requires separate libraries (or constant adaptation) to deal with the two cases.
- If a and b are polymorphic then the same name has different semantics

```
a == b // is a the same instance as b (&a == &b)
```

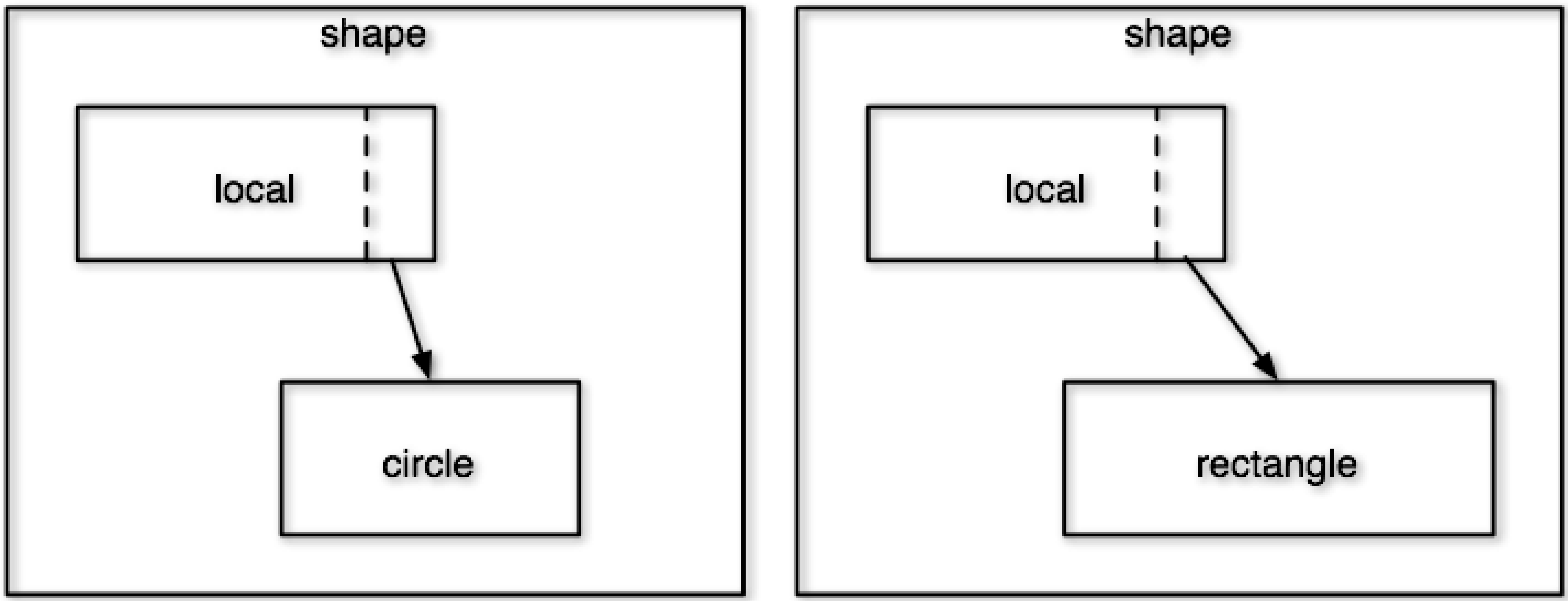
- Using the same name with different semantics (likely in the same context) causes confusion

# The Poly Library - Goals

- Shift the burden of polymorphism to the point of use (non-intrusive)
- Encapsulate the object management (no GC required, thread safe)
- Normalize naming (polymorphic objects work correctly with STL)
- Equal or better efficiency than than traditional inheritance
- Equal or better expressiveness than traditional inheritance

*Can we build complete applications were everything exists in a container?*

# The Poly Library - The Basics



# The Poly Library - Basics

- There exists a transition point from having complete type information to having limited type information
  - We refer to this as the virtualization boundary

- We can leverage type erasure to capture type properties carry then across the boundary

# The Poly Library - Basics

```
class poly_copyable {
    struct concept {
            virtual ~concept() { }
            virtual concept* clone() const = 0;
    };

    template <typename T>
    struct model : concept {
            model(const T x) : instance(x) { };
            concept* clone() const { return new model(instance); }
            T instance;
    };

    concept* object;
 public:
    template<typename T>
    poly_copyable(const T& x) : object(new model<T>(x)) { }
    poly_copyable(const poly_copyable& x) : object(x.object->clone()) { }
    ~poly_copyable() { delete object; }
};
```





Adobe

# The Poly Library - Basics

```
int main()
{
   poly_copyable x(10);   // Capture copy-ctor here
   poly_copyable y = x;   // Use copy-ctor here
}
```

- The overhead is *exactly* that of traditional inheritance
- Overhead is only paid for why polymorphism is required

# The Poly Library - Usage in Adobe Source Libraries

- ASL provides a few special purpose poly types:
  - any_regular_t
    - All operations on the Concept Regular including O(1), non-throwing swap()
    - Small object optimization (small objects with non-throwing default ctor stored locally)
    - Leverages type promotion as well as virtualization
      - Most numeric types promote to double
      - char* promotes to std::string
  - GIL makes use of an any_image<> type which can be parametersed with a set of specific types for which optimal algorithms can be instantiated
  - There is an any_iterator library which experiments with concept refinement and polymorphism
- The poly library incorporates many of the above ideas into a single library

# The Poly Library - Usage in Adobe Source Libraries

- The poly library allows client specified concept descriptions
  - Concept descriptions can inherit from each other to allow refinement
- `poly<Placable>`, `poly<View>`, `poly<Controller>` are used to connect widgets to the property model and layout libraries - each of these are refinements of `poly<Regular>` which will soon replace `any_regular_t`.
- The `any_regular_t` is used as the "dynamic type" for the property model library
  - Allowing the client to create property models with any regular type, including using the type in the property model language

# The Poly Library - Usage in Adobe Source Libraries

```
struct checkbox_t
{
  typedef any_regular_t                                   model_type;
  typedef boost::function<void (const model_type&)>       setter_type;

  checkbox_t( const std::string&  name,
        const any_regular_t&      true_value,
        const any_regular_t&      false_value,
        theme_t                   theme,
        const std::string&        alt_text);

  void measure(extents_t& result);
  void place(const place_data_t& place_data);
  void display(const any_regular_t& value);
  void enable(bool make_enabled);
  void monitor(setter_type proc);
};

bool operator==(const checkbox_t&, const checkbox_t&);
```

# The Poly Library - Future Directions

- Learning and exploring how to assemble systems with value semantics
  - We do have pointers under the hood
  - References between objects are managed with in a container that holds the objects
    - All data structures are explicit
- We are collaborating with Texas A&M and others to explore new techniques and understand the theoretical limitations
  - Techniques such as runtime compilation (compile when the types are known) is an interesting future direction
- You can find more information on our website http://opensource.adobe.com. Keep on eye on the Papers and Presentations section of our wiki for current and upcoming papers.

# Better by Adobe.™