Beyond Objects
Understanding The Software We Write

Sean Parent
Adobe Systems
sparent@adobe.com
http://opensource.adobe.com/
Polymorphism as Implementation

Sean Parent
Adobe Systems
sparent@adobe.com
http://opensource.adobe.com/
Abstract

• Look at existing techniques for handling runtime polymorphism in C++.
  ● Associated complex system of "patterns" that complicate the use of libraries.
• Develop an alternative approach to runtime polymorphism.
  ● Hide polymorphic implementation inside objects that can be copied, assigned, compared for equality, stored in STL containers and used with STL algorithms.
Background

- Improve code through understanding
- Increasing use of generic programming
  - Prefer the term Concept-based programming
- Struggle with object-oriented vs. generic
- Often choice is runtime vs. compile-time
  - An artificial dichotomy
Promise of Concepts

• Algorithms Determine Type Requirements
• Requirements Cluster as Concepts
  • Most Appropriate Algorithm Selected by Concept Match
• Code is Reusable and Efficient
  • Write Algorithms Once
typedef std::pair<int, int> point;

class shape
{
    point center_m;

public:
    shape(const point& center) : center_m(center) { }

    point where() const { return center_m; }
    void move(const point& to) { center_m = to; }

    virtual void draw() const = 0;
};
Combine OO and Generic Code

class circle : public shape
{
    public:
        int radius;

    circle(const point& center, int r) : shape(center), radius(r) {}

    void draw() const {
        std::cout << "circle(point(" << where().first << ", " << where().second << ");" << std::endl;
    }
};
class rectangle : public shape
{
public:
    int width, height;

    rectangle(const point& center, int w, int h)
        : shape(center), width(w), height(h) {}

    void draw() const {
        std::cout << "rectangle(point(" << where().first << ", " <<
                  where().second << ")", " << width << ", " << height
                  << ");" << std::endl;
    }
};
vector<shape> s1;
s1.push_back(circle(point(1, 2), 3));
s1.push_back(circle(point(4, 5), 6));
s1.push_back(rectangle(point(7, 8), 9, 10));

vector<shape> s2(s1);

reverse(s1);

find(s1, circle(point(4, 5), 6))->move(point(10, 20));

for_each(s1, &shape::draw);
for_each(s2, &shape::draw);
First Attempt

```cpp
int main()
{
    using namespace std;

    vector<shape> s1;
    s1.push_back(circle(point(1, 2), 3)); // Error!
        // Cannot allocate an object of abstract type shape

    return 0;
}
```
Classic Problem: Object Slicing
Classic Solution: Indirection

- shape*
- shape*
- shape*

- circle
- rectangle
Second Attempt

```cpp
vector<shape*> s1;
s1.push_back(new circle(point(1, 2), 3));
s1.push_back(new circle(point(4, 5), 6));
s1.push_back(new rectangle(point(7, 8), 9, 10));

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

(*find(s1.begin(), s1.end(), new circle(point(4, 5), 6)))
    ->move(point(10, 20)); // Runtime Error!
```
Identity Is Not Equality

- Comparing Pointer is Checking Identity
- Need to Compare Polymorphic Instances
class object {
    public:
        virtual bool equals(const object&) const = 0;
};
class shape : public object
{ /* ... */};
class circle : public shape {
    /* ... */
    bool equals(const object& x) const {
        if (typeid(x) != typeid(circle)) return false;
        const circle& c(static_cast<const circle&>(x));
        return (c.where() == where()) && (c.radius == radius);
    }
    /* ... */
struct equal_object : std::unary_function<const object*, bool>
{
    const object* object_m;
    equal_object(const object* x) : object_m(x) {} 

    bool operator()(const object* x) const
    { return object_m->equals(*x); }
};
Third Attempt

/* ... */
(*find_if(s1.begin(), s1.end(), equal_object(new circle(point(4, 5), 6))))
    ->move(point(10, 20));

for (vector<shape*>::const_iterator first(s1.begin()), last(s1.end());
    first != last; ++first)
    { (*first)->draw(); }

/* ... draw each shape in s2 */
rectangle(point(7, 8), 9, 10);
circle(point(10, 20), 6);
circle(point(1, 2), 3);
circle(point(1, 2), 3);
circle(point(10, 20), 6);
rectangle(point(7, 8), 9, 10);
Implicit Data Structure
Lesson:

• “A shared pointer is as ‘good’ as a global.”

● Exclusions:

  • Explicit Container
    – Shared Pointer as Relationship
  • Pointers to Immutable Values.
References as Values:

• A reference can be used as a value if the value is immutable during the lifetime of the reference.
  • There is no need to copy immutable objects.
  • This is why passing parameters as a const & works.
Borrow a Little More From Java...

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

class circle : public shape {
    /* ... */
    object* clone() const { return new circle(*this); }
};
Fourth Attempt

/* ... */

vector<shape*> s2(s1);

for (vector<shape*>::iterator first(s2.begin()), last(s2.end());
    first != last; ++first)
    { *first = static_cast<shape*>((*first)->clone()); } 

/* ... */
rectangle(point(7, 8), 9, 10);
circle(point(10, 20), 6);
circle(point(1, 2), 3);
circle(point(1, 2), 3);
circle(point(4, 5), 6);
rectangle(point(7, 8), 9, 10);
vector<shape*> s1;
s1.push_back(new circle(point(1, 2), 3));
s1.push_back(new circle(point(4, 5), 6));
s1.push_back(new rectangle(point(7, 8), 9, 10));

vector<shape*> s2(s1);

for (vector<shape*>::iterator first(s2.begin()), last(s2.end()); first != last; ++first)
{ *first = static_cast<shape*>((*first)->clone()); }

reverse(s1.begin(), s1.end());

(*find_if(s1.begin(), s1.end(), equal_object(new circle(point(4, 5), 6))))->move(point(10, 20));

for (vector<shape*>::const_iterator first(s1.begin()), last(s1.end()); first != last; ++first)
{ (*first)->draw(); }

for (vector<shape*>::const_iterator first(s2.begin()), last(s2.end()); first != last; ++first)
{ (*first)->draw(); }
vector<shape> s1;
s1.push_back(circle(point(1, 2), 3));
s1.push_back(circle(point(4, 5), 6));
s1.push_back(rectangle(point(7, 8), 9, 10));

vector<shape> s2(s1);

reverse(s1);

find(s1, circle(point(4, 5), 6))->move(point(10, 20));

for_each(s1, &shape::draw);
for_each(s2, &shape::draw);
Did It Work?

• It shows generic techniques are very flexible.
• How far should we take this?
  ● Parallel standard library that works on classes derived from class object?
    • copy_deep, find_deep, etc.
  ● Add indirect iterators, deep containers, projection functions, function objects for all the operators?
  ● Add a garbage collector and/or reference counted pointers?
  ● Always take great care to avoid implicit structures.
Let’s Try Again...
# A Quick Look At Concepts

<table>
<thead>
<tr>
<th>expression</th>
<th>return type</th>
<th>post-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(t)</td>
<td></td>
<td>t is equal to T(t)</td>
</tr>
<tr>
<td>T(u)</td>
<td></td>
<td>u is equal to T(u)</td>
</tr>
<tr>
<td>t.~T()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;t</td>
<td>T*</td>
<td>denotes address of t</td>
</tr>
<tr>
<td>&amp;u</td>
<td>const T*</td>
<td>denotes address of u</td>
</tr>
</tbody>
</table>

### Table 1 - CopyConstructable

| t = u       | T&          | t is equal to u |

### Table 2 - Assignable

| a==b        | convertible to bool | == is the equality relation |

### Table 3 – EqualityComparable
Value Semantics

For all a, a == a.
If a == b, then b == a.
If a == b, and b == c, then a == c.
T a(b) implies a == b.
T a; a = b ⇔ T a(b).
T a(c); T b(c); a = d; then b == c.
T a(c); T b(c); modify(a) then b == c && a != b.
If a == b then for any regular function f, f(a) == f(b).
!(a == b) ⇔ a != b.
Why Polymorphism?

• Apply an algorithm to heterogeneous objects as if they were homogeneous

• In this example – I need a “Drawable” type that can contain any Drawable object and is also a Regular* type.

  *CopyConstructible, Assignable, and EqualityComparable.
Why Polymorphism?

• Apply an algorithm to heterogeneous typed objects as if they were homogeneous

• Part of a continuum between “static and dynamic”
  ● We use variables for heterogeneous values.
  ● We use Concepts for heterogeneous types.
Root Cause

circle

shape  shape  shape
Another Solution – Remote Parts

shape

local

circle

shape

local

rectangle
struct circle {
    int radius;

    circle(int r) : radius(r) { }

    void draw(const point&) const {
        std::cout << "shape(point(" << center_m.first << ", ", "
          << center_m.second << "), circle(" 
          << radius << ") ); " << std::endl;
    }
};

inline bool operator == (const circle& x, const circle& y) {
    return x.radius == y.radius;
}
Same for rectangle

```cpp
struct rectangle {
  int width, height;

  rectangle(int w, int h) : width(w), height(h) {}

  void draw(const point&) const {
    std::cout << "shape(point(" << center_m.first << ", \\
                     " << center_m.second << "), rectangle(" << width << height << ") ); " << std::endl;
  }
};

inline bool operator == (const rectangle& x, const rectangle& y) {
  return x.width == y.width && x.height == y.height; }
```
Shape (Pseudo Code)

class shape {
    point center_m;
    Drawable object_m;

    public:
    shape(const point& center, const Drawable& s) :
            center_m(center), object_m(s) { }

    void draw() const { object_m.draw(center_m); }

    point where() const { return center_m; }
    void move(const point& to) { center_m = to; }
};

inline bool operator==(const shape& x, const shape& y)  
{ return (x.center_m == y.center_m) && (x.object_m == y.object_m); }
class shape {
    /* …MAGIC STUFF HERE… */
    point center_m;
    regular_object<drawable_interface, drawable_instance> object_m;
public:
    template <typename T> // T models Drawable
    shape(const point& center, const T& s) : center_m(center), object_m(s) {} 
    void draw() const { object_m->draw(center_m); } 
    point where() const { return center_m; } 
    void move(const point& to) { center_m = to; } 
};
inline bool operator==(const shape& x, const shape& y) 
{ return (x.center_m == y.center_m) && (x.object_m == y.object_m); }
vector<shape> s1;
s1.push_back(shape(point(1, 2), circle(3)));
s1.push_back(shape(point(4, 5), circle(6)));
s1.push_back(shape(point(7, 8), rectangle(9, 10)));

vector<shape> s2(s1);

reverse(s1.begin(), s1.end());

find(s1.begin(), s1.end(), shape(point(4, 5), circle(6)))->move(point(10, 20));

for_each(s1.begin(), s1.end(), mem_fun_ref(&shape::draw));
for_each(s2.begin(), s2.end(), mem_fun_ref(&shape::draw));
shape(point(7, 8), rectangle(9, 10));
shape(point(10, 20), circle(6));
shape(point(1, 2), circle(3));
shape(point(1, 2), circle(3));
shape(point(4, 5), circle(6));
shape(point(7, 8), rectangle(9, 10));
Inheritance

drawable_interface
v draw() = 0

regular_interface<>
v equals() = 0; v clone() = 0; v type() = 0
v ~regular_interface() {}

drawable_instance<>
value_type value
draw() {}

regular_instance<>
equals() {} clone() {} type() {}
class shape {
    struct drawable_interface {
        virtual void draw(const point&) const = 0;
    };

template <typename T> // T models Drawable
struct drawable_instance : regular_interface<drawable_interface> {
    typedef T value_type;
    value_type value; // ← circle or rectangle goes here
    drawable_instance(const value_type& x) : value(x) {
    }
    void draw(const point& where) const { value.draw(where); }
};

point center_m;
regular_object<drawable_interface, drawable_instance> object_m;
/* ... */
regular_interface<>

template <typename I> // I is a pure virtual interface
struct regular_interface : I
{
    virtual bool equals(const regular_interface&) const = 0;
    virtual regular_interface* clone() const = 0;
    virtual const std::type_info& type() const = 0;
    virtual ~regular_interface();
};

template <typename I> regular_interface<I>::~regular_interface() {}
regular_instance<>

template <typename F> // F is an instance of regular_interface
struct regular_instance : F {
    typedef typename F::value_type value_type;

    regular_instance(const value_type& x): F(x) {}  

    bool equals(const interface_type& x) const {
        return (x.type() == typeid(value_type))
            && (static_cast<const regular_instance&>(x).value == this->value);
    }

    interface_type* clone() const { return new regular_instance(this->value); }  

    const std::type_info& type() const { return typeid(value_type); }  
};
template <typename I, // I is a pure virtual interface class
template<class> class D > // D is instance template
class regular_object {
  typedef regular_interface<I> interface_type;
  interface_type* interface_m;
public:
  template<typename T> explicit regular_object(const T& x) :
    interface_m(new regular_instance<D<T> >(x)) { }

  regular_object(const object& x) :
    interface_m(x.interface_m->clone()) { }

/* ... */
regular_object<> (Part 2)

    /* ... */

    regular_object& operator=(const regular_object& x) {
        interface_type* tmp = x.interface_m->clone();
        std::swap(tmp, interface_m);
        delete tmp;
        return *this;
    }

    ~regular_object() { delete interface_m; }

    const interface_type* operator->() const { return interface_m; }
    interface_type* operator->() { return interface_m; }

    friend inline
    bool operator==(const regular_object& x, const regular_object& y)
    { return x.interface_m->equals(*y.interface_m); }
Tradeoffs

• Pros (of value semantics):
  ● Simpler Client Interface
    • Writing a new Drawable class is trivial
    • Using Shapes is simple
  ● Cleanly Extensible
    • Not Intrusive - works for Integer
  ● Types Model RegularType
    • (usable with STL/Boost/ASL…)
  ● No External Dependencies
    • Easier to Reuse
Tradeoffs

• Cons (of value semantics):
  • Lost Fast Move
    • reverse() is slower
      – Fixed this by specializing std::swap()
  • Heavy Meta-Machinery
    • The Magic Stuff
  • No Language Concept Support
    • Fail to satisfy Drawable requirements and stare in awe at the error message!
  • No Large Scale Examples
    • But I’m working on it!
What’s Next?

• **Language Concept Support**
  • How do we define and enforce the semantics of Concepts.

• **Move Support**
  • More importantly, we need to develop the underlying axioms and incorporate into our regular Concept.

• **Semantic Spaces**
  • How do I state “when I say swap, I mean a swap for this type with the same semantics as std::swap”.
One More Bit...
vector<shape> s1;
s1.push_back(shape(point(1, 2), circle(3)));
s1.push_back(shape(point(4, 5), circle(6)));
s1.push_back(shape(point(7, 8), rectangle(9, 10)));

vector<shape> s2(s1);

reverse(s1.begin(), s1.end());

find(s1.begin(), s1.end(), shape(point(4, 5), circle(6)))
    ->move(point(10, 20));

for_each(s1.begin(), s1.end(), mem_fun_ref(&shape::draw));
for_each(s2.begin(), s2.end(), mem_fun_ref(&shape::draw));
vector<shape> s1;
s1.push_back(circle(point(1, 2), 3));
s1.push_back(circle(point(4, 5), 6));
s1.push_back(rectangle(point(7, 8), 9, 10));

vector<shape> s2(s1);

reverse(s1);

find(s1, circle(point(4, 5), 6))->move(point(10, 20));

for_each(s1, &shape::draw);
for_each(s2, &shape::draw);
vector<shape> s1;
s1.push_back(shape(point(1, 2), circle(3)));
s1.push_back(shape(point(4, 5), circle(6)));
s1.push_back(shape(point(7, 8), rectangle(9, 10)));

vector<shape> s2(s1);

reverse(s1);

find(s1, shape(point(4, 5), circle(6)))->move(point(10, 20));

for_each(s1, &shape::draw);
for_each(s2, &shape::draw);
Links

- Alex Stepanov’s Collected Works

- Fundamentals of Generic Programming
  - [http://www.stepanovpapers.com/DeSt98.pdf](http://www.stepanovpapers.com/DeSt98.pdf)
Polymorphism as Implementation

Sean Parent
Adobe Systems
sparent@adobe.com
http://opensource.adobe.com/