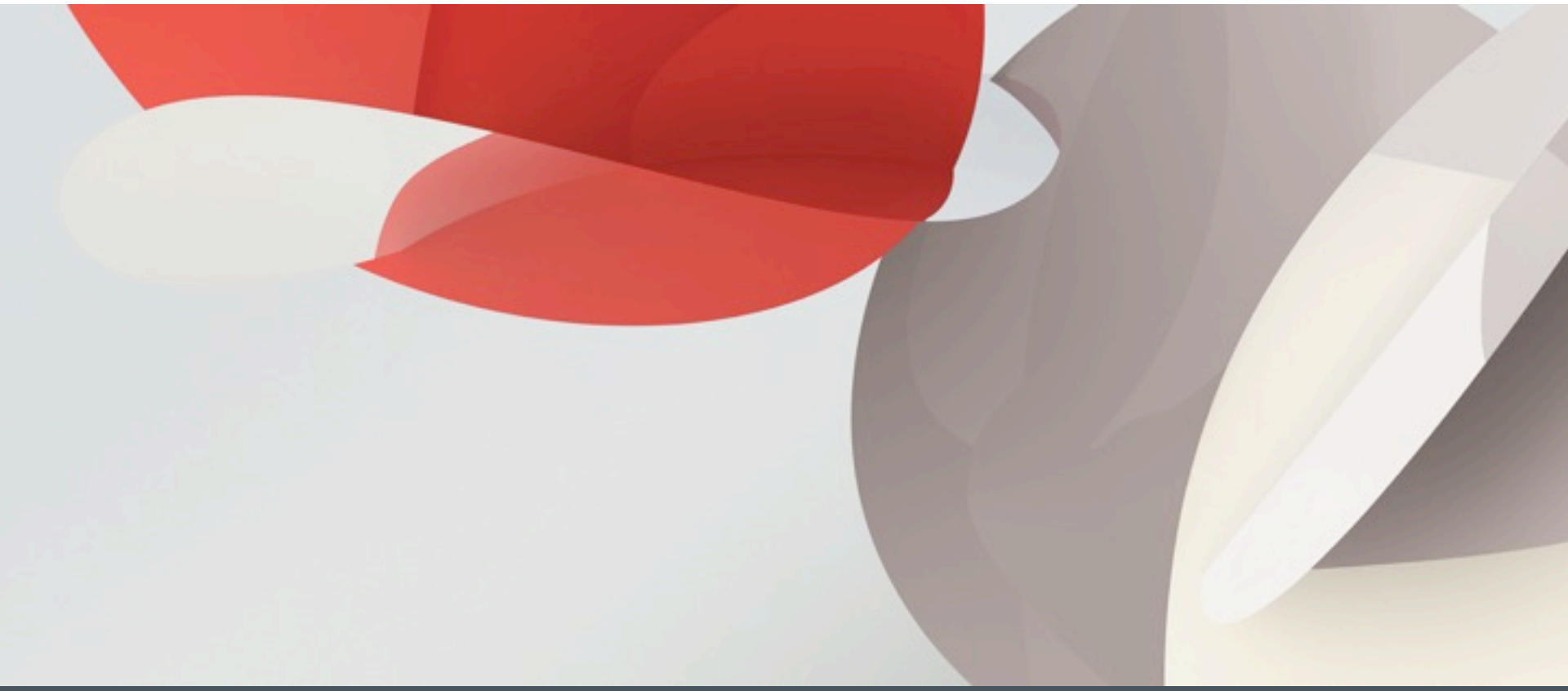
Value Semantics and Concept Based Polymorphism

Sean Parent | Principal Scientist



Outline of Talk

- Defining Value Semantics
- “Polymorphic Types”
- Demo of Photoshop History
- Implementing History

Objects & Types

- A value type is a correspondence between entities with common properties, or species, and a set of values
  - Example species: color, integer, dog
- An object type is a pattern for storing a values of a corresponding value type in memory
  - Since this talk deals with computers and memory, we will simply use type to mean object type
- An object is a representation of a concrete entity as a value in memory

Objects & Types

- The physical nature of the machine imbues a set of properties which are common to all object types - we refer to this collection of properties as the concept of a regular type
  - All types are inherently regular

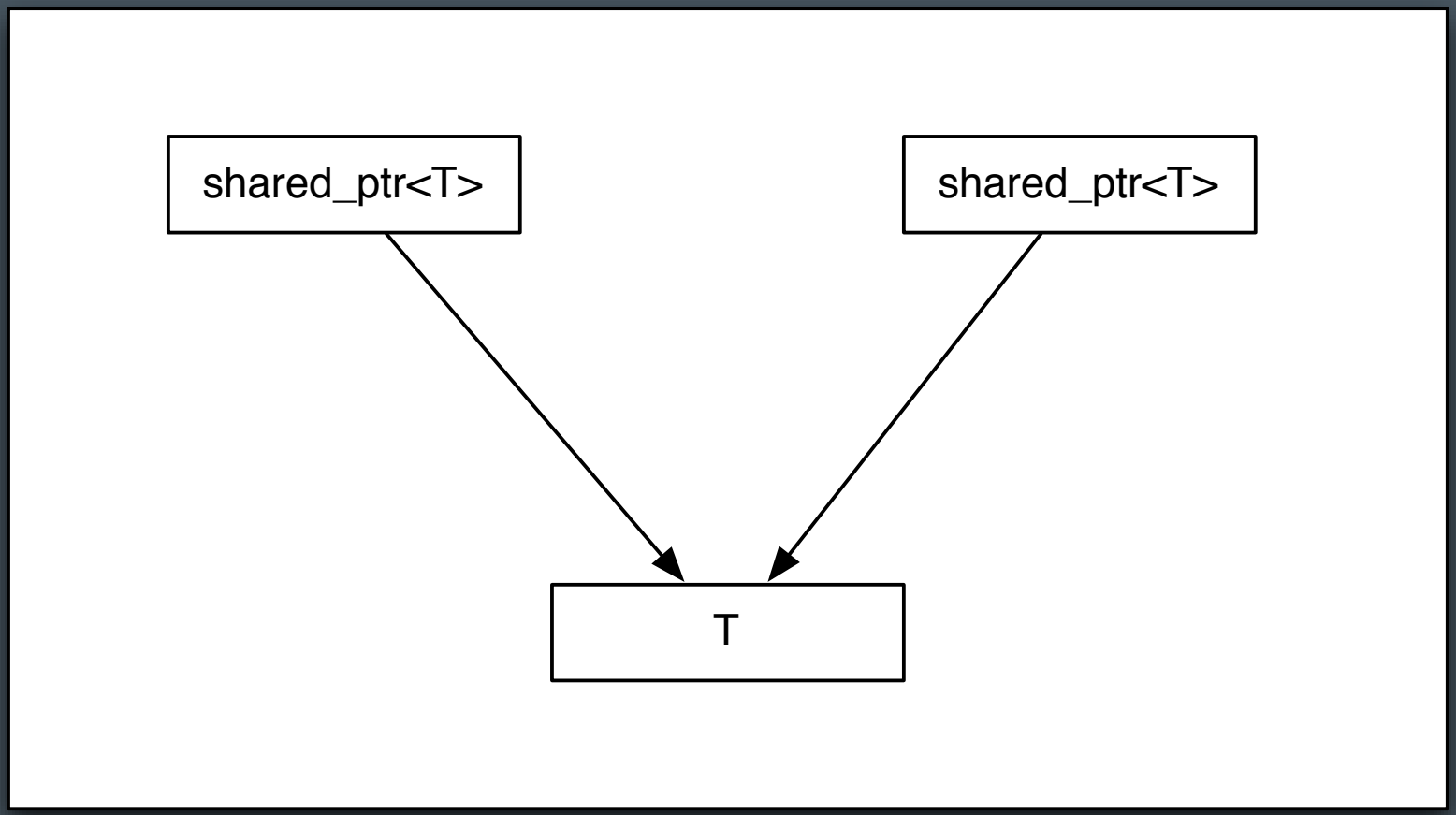
Semantics & Syntax

- We define an operation in terms of the operation's semantics:
  - "Assignment is a procedure taking two objects of the same type that makes the first object equal to the second without modifying the second."
- Choosing the same syntax for the same semantics enables code reuse and avoids combinatoric interfaces
  - If a type has a proper set of basis operations then it can be adapted to any alternative set of basis operations regardless of syntax
- C++ has defined semantics for operations on built-in types, including assignment, copy, equality, address-of
  - By using the same operator names to provide the same semantics on user types enables code reuse

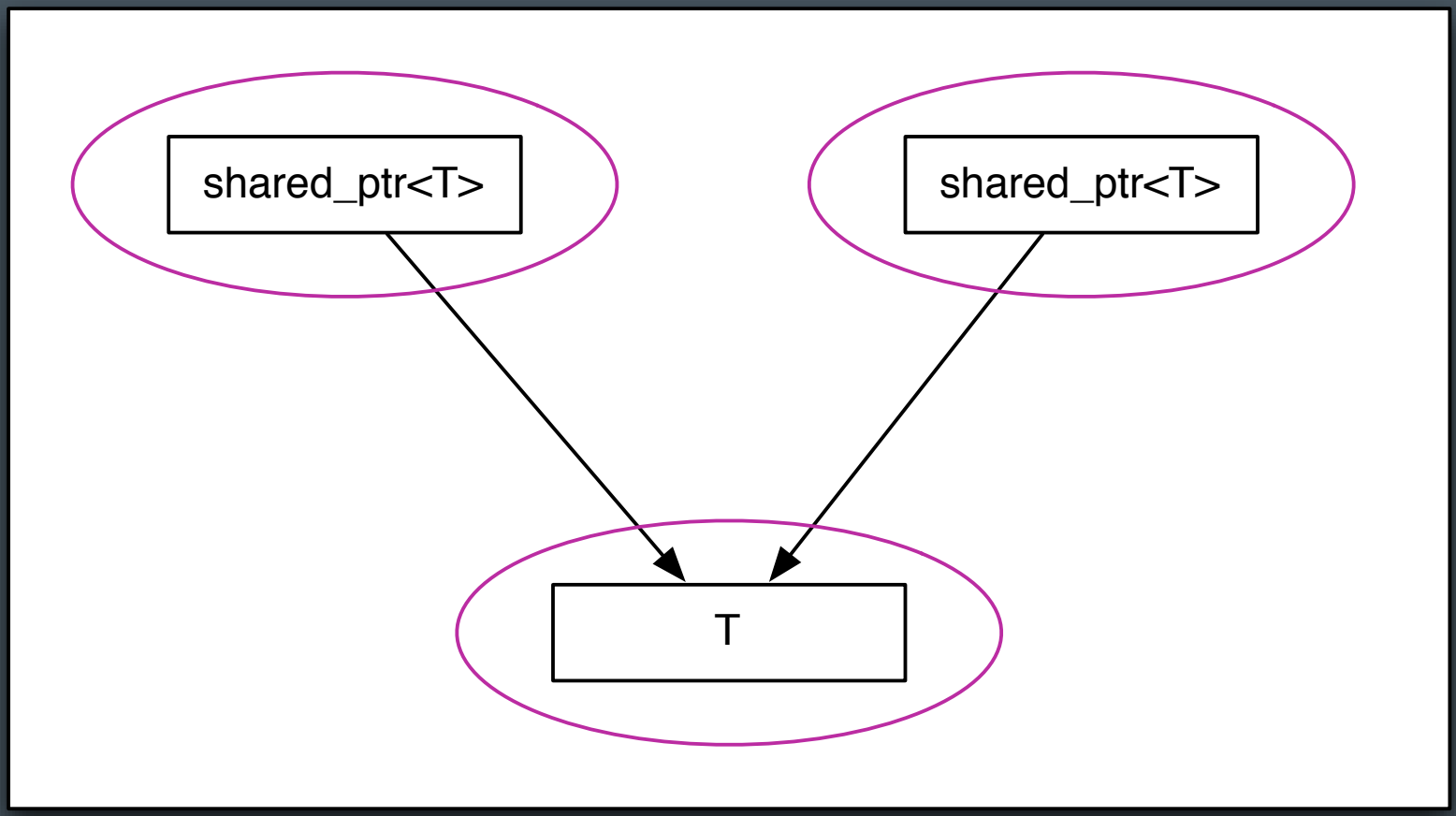
Semantics & Syntax

- Regular types where the regular operations are implemented with the standard names are said to have *value semantics*
- When user object are always referred to indirectly, through a shared reference or pointer, the objects are said to have *reference semantics*
- Value semantics is similar to functional programming, except objects still have addresses and in-situ operations
  - Given a transformation, f, we can define an equivalent action, a, such that a(x) is equivalent to x = f(x)
  - Given an action, a, we can define an equivalent action, f, such that x = f(x) is equivalent to a(x)
  - However, one representation may be more efficient than the other

Semantics & Syntax



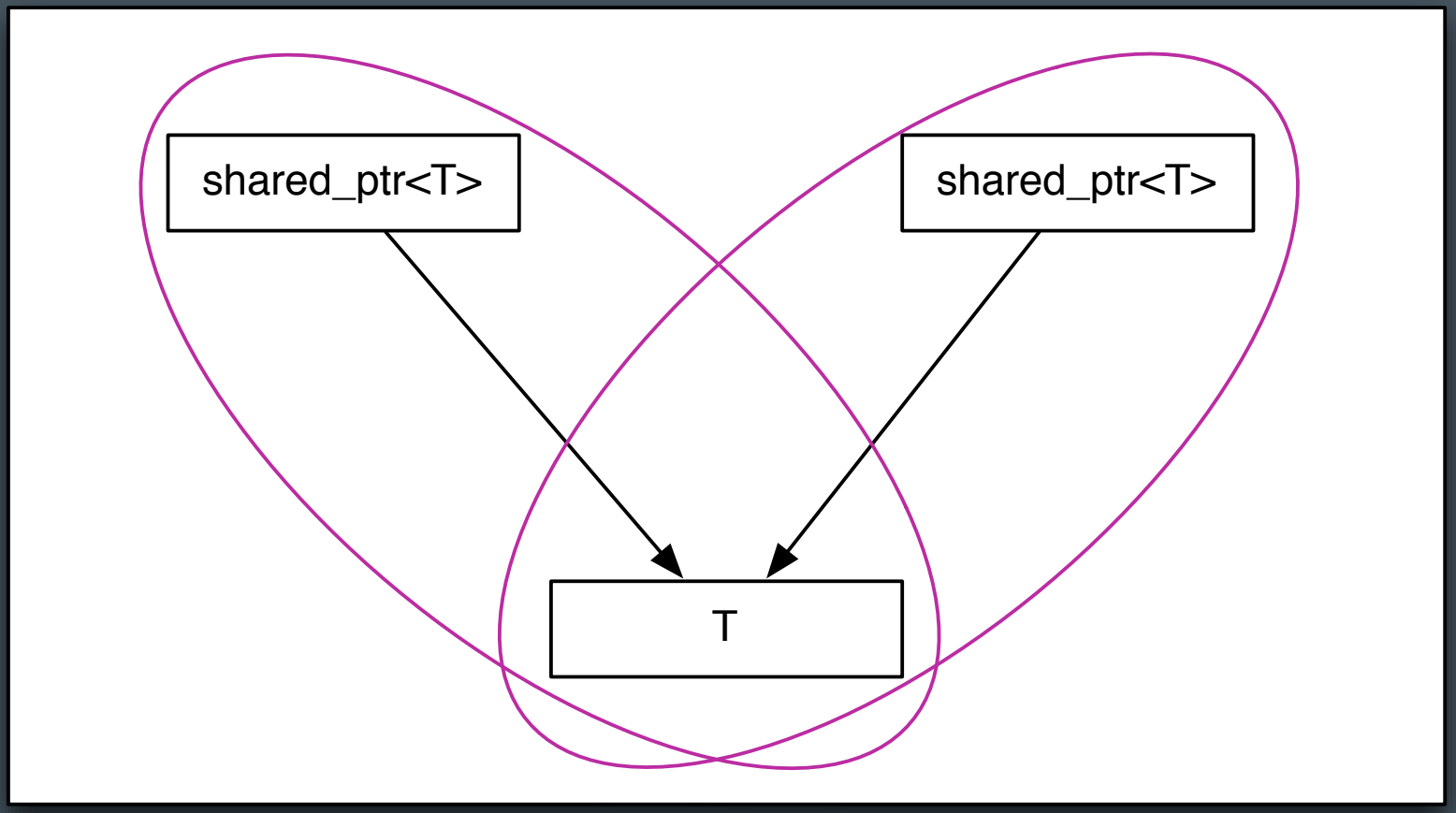
Semantics & Syntax



Semantics & Syntax

- Considered as individual types, assignment and copy hold their regular semantic meanings
  - However, this fails to account for the relationships (the arrows) which form an incidental data-structure. You cannot operate on T through one of the shared pointers without considering the effect on the other shared pointer

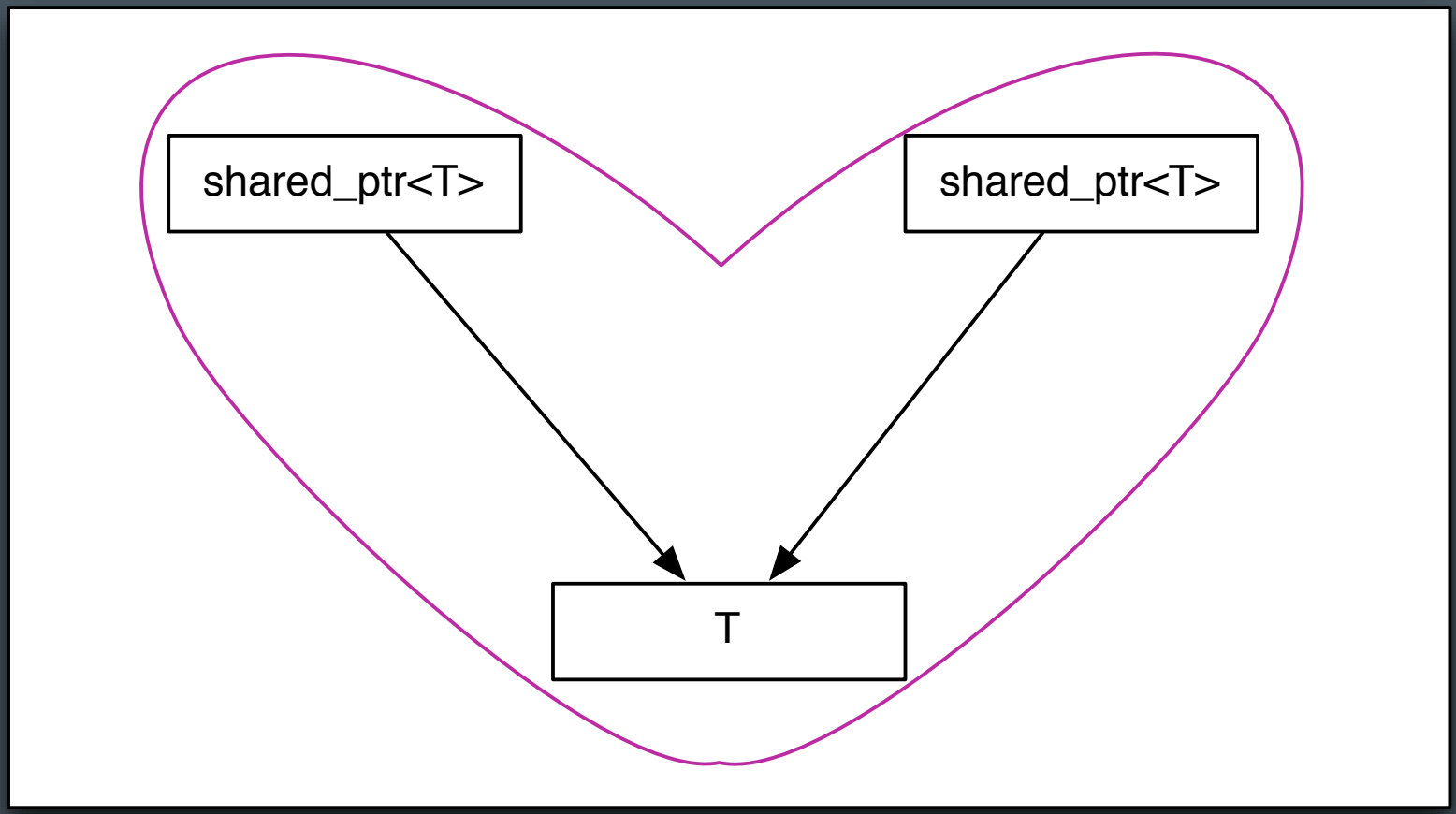
Semantics & Syntax



Semantics & Syntax

- If we extend our notion of object type to include the directly related part then we have intersecting objects which will interfere with each other

Semantics & Syntax



Semantics & Syntax

- When we consider the whole, the standard syntax for copy and assignment no longer have their regular semantics.
  - This structure is still copyable and assignable but these operations must be done through other means

- The shared structure also breaks our ability to reason locally about the code
  - A shared pointer is as good as a global variable

“Polymorphic Types”

- The requirement of a polymorphic type, by definition, comes from it's use -
  - There is no such thing as a polymorphic type, only a polymorphic use of similar types
- By using inheritance to capture polymorphic use, we shift the burden of use to the type implementation, tightly coupling components
- Inheritance implies variable size, which implies heap allocation
- Heap allocation forces a further burden on use to manage the object lifetime
- Object lifetime management leads to garbage collection or reference counting
- This encourages *shared* ownership and the proliferation of *incidental data-structures*

Disclaimer

- In the following code, the proper use of header files, inline functions, and namespaces are ignored for clarity

```
int main()
{
    cout << "Hello World!" << endl;
}
```

client | library

```
int main()
{
    cout << "Hello World!" << endl;
}
```

cout

Hello World!

```
using object_t = int;

void draw(const object_t& x, ostream& out, size_t position)
{ out << string(position, ' ') << x << endl; }

using document_t = vector<object_t>;

void draw(const document_t& x, ostream& out, size_t position)
{
    out << string(position, ' ') << "<document>" << endl;
    for (auto& e : x) draw(e, out, position + 2);
    out << string(position, ' ') << "</document>" << endl;
}
```

```
int main()
{
    document_t document;

    document.emplace_back(0);
    document.emplace_back(1);
    document.emplace_back(2);
    document.emplace_back(3);

    draw(document, cout, 0);
}
```

client | library

```
int main()
{
    document_t document;

    document.emplace_back(0);
    document.emplace_back(1);
    document.emplace_back(2);
    document.emplace_back(3);

    draw(document, cout, 0);
}
```

cout

```
<document>
 0
 1
 2
 3
</document>
```

client | library

```
int main()
{
    document_t document;

    document.emplace_back(0);
    document.emplace_back(1);
    document.emplace_back(2);
    document.emplace_back(3);

    draw(document, cout, 0);
}
```

guidelines

- Write all code as a library.
  - Reuse increases your productivity.
  - Writing unit tests is simplified.

```
using object_t = int;

void draw(const object_t& x, ostream& out, size_t position)
{ out << string(position, ' ') << x << endl; }
```

```
using document_t = vector<object_t>;

void draw(const document_t& x, ostream& out, size_t position)
{
    out << string(position, ' ') << "<document>" << endl;
    for (auto& e : x) draw(e, out, position + 2);
    out << string(position, ' ') << "</document>" << endl;
}
```

```
void draw(const int& x, ostream& out, size_t position)
{ out << string(position, ' ') << x << endl; }

class object_t {
  public:
    object_t(const int& x) : object_(x)
    { }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { draw(x.object_, out, position); }

  private:
    int object_;
};
```

```
using document_t = vector<object_t>;

void draw(const document_t& x, ostream& out, size_t position)
{
    out << string(position, ' ') << "<document>" << endl;
    for (auto& e : x) draw(e, out, position + 2);
    out << string(position, ' ') << "</document>" << endl;
}
```

client | library

```
void draw(const int& x, ostream& out, size_t position)
{ out << string(position, ' ') << x << endl; }

class object_t {
  public:
    object_t(const int& x) : object_(x)
    { }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { draw(x.object_, out, position); }

  private:
    int object_;
};

using document_t = vector<object_t>;

void draw(const d                tream& out, size_t position)
{
    out << string                << "<document>" << endl;
```

guidelines

- The compiler will supply member-wise copy and assignment operators.
  - Let the compiler do the work where appropriate.

```
int main()
{
    document_t document;

    document.emplace_back(0);
    document.emplace_back(1);
    document.emplace_back(2);
    document.emplace_back(3);

    draw(document, cout, 0);
}
```

client | library

```
int main()
{
    document_t document;

    document.emplace_back(0);
    document.emplace_back(1);
    document.emplace_back(2);
    document.emplace_back(3);

    draw(document, cout, 0);
}
```

cout

```
<document>
 0
 1
 2
 3
</document>
```

client | library

```
int main()
{
    document_t document;

    document.emplace_back(0);
    document.emplace_back(1);
    document.emplace_back(2);
    document.emplace_back(3);

    draw(document, cout, 0);
}
```

guidelines

- Write classes that behave like *regular* objects to increase reuse.

```
class object_t {
  public:
    object_t(const int& x) : object_(x)
    { }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { draw(x.object_, out, position); }

  private:
    int object_;
};

using document_t = vector<object_t>;

void draw(const document_t& x, ostream& out, size_t position)
{
    out << string(position, ' ') << "<document>" << endl;
    for (auto& e : x) draw(e, out, position + 2);
    out << string(position, ' ') << "</document>" << endl;
}
```

```
class object_t {
  public:
    object_t(const int& x) : object_(new int_model_t(x))
    { }
    ~object_t() { delete object_; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        int data_;
    };

   int_model_t* object_;
};

using document_t = vector<object_t>;

void draw(const document_t& x, ostream& out, size_t position)
{
```

client | library

```
class object_t {
  public:
    object_t(const int& x) : object_(new int_model_t(x))
    { }
    ~object_t() { delete object_; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        int data_;
    };

    int_model_t* o
};
```

guidelines

- Do your own memory management - don't create garbage for your client to clean-up.

```
class object_t {
  public:
    object_t(const int& x) : object_(new int_model_t(x))
    { }
    ~object_t() { delete object_; }

    object_t(const object_t& x) : object_(new int_model_t(*x.object_))
    { }
    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        int data_;
    };

   int_model_t* object_;
};

using document_t = vector<object_t>;
```

client | library

```
class object_t {
  public:
    object_t(const int& x) : object_(new int_model_t(x))
    { }
    ~object_t() { delete object_; }

    object_t(const object_t& x) : object_(new int_model_t(*x.object_))
    { }
    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        int data_
    };
```

guidelines

- The semantics of copy are to create a new object which is equal to, and logically disjoint from, the original.
- Copy constructor must copy the object. The compiler is free to elide copies so if the copy constructor does something else the code is incorrect.
- When a type manages *remote parts* it is necessary to supply a copy constructor.
  - If you can, use an existing class (such as a vector) to manage remote parts.

```
class object_t {
  public:
    object_t(const int& x) : object_(new int_model_t(x))
    { }
    ~object_t() { delete object_; }

    object_t(const object_t& x) : object_(new int_model_t(*x.object_))
    { }
    object_t& operator=(const object_t& x)
    { delete object_; object_ = new int_model_t(*x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        int data_;
    };

   int_model_t* object_;
};
```

client | library

```
class object_t {
  public:
    object_t(const int& x) : object_(new int_model_t(x))
    { }
    ~object_t() { delete object_; }

    object_t(const object_t& x) : object_(new int_model_t(*x.object_))
    { }
    object_t& operator=(const object_t& x)
    { delete object_; object_ = new int_model_t(*x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw
        { draw(da
```

guidelines

- Assignment is consistent with copy. Generally:

  `T x; x = y;` is equivalent to `T x = y;`

```
int main()
{
    document_t document;

    document.emplace_back(0);
    document.emplace_back(1);
    document.emplace_back(2);
    document.emplace_back(3);

    draw(document, cout, 0);
}
```

client | library

```
int main()
{
    document_t document;

    document.emplace_back(0);
    document.emplace_back(1);
    document.emplace_back(2);
    document.emplace_back(3);

    draw(document, cout, 0);
}
```

cout

```
<document>
 0
 1
 2
 3
</document>
```

client | library

```
int main()
{
    document_t document;

    document.emplace_back(0);
    document.emplace_back(1);
    document.emplace_back(2);
    document.emplace_back(3);

    draw(document, cout, 0);
}
```

guidelines

- The Private Implementation (Pimpl), or Handle-Body, idom is good for separating the implementation and reducing compile times.

```
class object_t {
  public:
    object_t(const int& x) : object_(new int_model_t(x))
    { }
    ~object_t() { delete object_; }

    object_t(const object_t& x) : object_(new int_model_t(*x.object_))
    { }
    object_t& operator=(const object_t& x)
    { delete object_; object_ = new int_model_t(*x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        int data_;
    };

   int_model_t* object_;
};
```

client | library

```
class object_t {
  public:
    object_t(const int& x) : object_(new int_model_t(x))
    { }
    ~object_t() { delete object_; }

    object_t(const object_t& x) : object_(new int_model_t(*x.object_))
    { }
    object_t& operator=(const object_t& x)
    { delete object_; object_ = new int_model_t(*x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, posit
        int data_;
```

defects

- If new throws an exception, the object will be left in an invalid state.
- If we assign an object to itself this will crash.

```
class object_t {
  public:
    object_t(const int& x) : object_(new int_model_t(x))
    { }

    object_t(const object_t& x) : object_(new int_model_t(*x.object_))
    { }
    object_t& operator=(const object_t& x)
    { object_.reset(new int_model_t(*x.object_)); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        int data_;
    };

    unique_ptr<int_model_t> object_;
};
```

client | library

```
class object_t {
  public:
    object_t(const int& x) : object_(new int_model_t(x))
    { }

    object_t(const object_t& x) : object_(new int_model_t(*x.object_))
    { }
    object_t& operator=(const object_t& x)
    { object_.reset(new int_model_t(*x.object_)); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw
        { draw(da
```

guidelines

- Assignment satisfying the *strong exception guarantee* is a nice property.
  - Either complete successfully or throw an exception, leaving the object unchanged.
- Assignment (like all other operations) must satisfy the basic exception guarantee.
- Don't optimize for rare cases which impact common cases.
  - Don't test for self-assignment to avoid the copy.

```
class object_t {
  public:
    object_t(const int& x) : object_(new int_model_t(x))
    { }

    object_t(const object_t& x) : object_(new int_model_t(*x.object_))
    { }
    object_t& operator=(const object_t& x)
    { object_.reset(new int_model_t(*x.object_)); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        int data_;
    };

   unique_ptr<int_model_t> object_;
};
```

```
class object_t {
  public:
    object_t(const int& x) : object_(new int_model_t(x))
    { cout << "ctor" << endl; }

    object_t(const object_t& x) : object_(new int_model_t(*x.object_))
    { cout << "copy" << endl; }
    object_t& operator=(const object_t& x)
    { object_t tmp(x); object_ = move(tmp.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        int data_;
    };

   unique_ptr<int_model_t> object_;
};
```

```
object_t func() { return 5; }

int main()
{
    /*
        Quiz: What will this print?
    */

    object_t x = func();
}
```

client | library

```
object_t func() { return 5; }

int main()
{
    /*
        Quiz: What will this print?
    */

    object_t x = func();
}
```

cout

ctor

```
object_t func() { return 5; }

int main()
{
    /*
        Quiz: What will this print?
    */

    object_t x = 0;

    x = func();
}
```

client | library

```
object_t func() { return 5; }

int main()
{
    /*
        Quiz: What will this print?
    */

    object_t x = 0;

    x = func();
}
```

cout

```
ctor
ctor
copy
```

```
class object_t {
  public:
    object_t(const int& x) : object_(new int_model_t(x))
    { cout << "ctor" << endl; }

    object_t(const object_t& x) : object_(new int_model_t(*x.object_))
    { cout << "copy" << endl; }
    object_t& operator=(const object_t& x)
    { object_t tmp(x); object_ = move(tmp.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        int data_;
    };

   unique_ptr<int_model_t> object_;
};
```

```
class object_t {
  public:
    object_t(const int& x) : object_(new int_model_t(x))
    { cout << "ctor" << endl; }

    object_t(const object_t& x) : object_(new int_model_t(*x.object_))
    { cout << "copy" << endl; }
    object_t& operator=(object_t x)
    { object_ = move(x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        int data_;
    };

   unique_ptr<int_model_t> object_;
};
```

client | library

```
class object_t {
  public:
    object_t(const int& x) : object_(new int_model_t(x))
    { cout << "ctor" << endl; }

    object_t(const object_t& x) : object_(new int_model_t(*x.object_))
    { cout << "copy" << endl; }
    object_t& operator=(object_t x)
    { object_ = move(x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(da                    on); }

        int data
```

guidelines

- Pass *sink* arguments by value and swap or *move* into place.
- A sink argument is any argument consumed or returned by the function.
  - The argument to assignment is a sink argument.

```
object_t func() { return 5; }

int main()
{
    /*
        Quiz: What will this print?
    */

    object_t x = 0;

    x = func();
}
```

client | library

```
object_t func() { return 5; }

int main()
{
    /*
        Quiz: What will this print?
    */

    object_t x = 0;

    x = func();
}
```

cout

```
ctor
ctor
```

client | library

```
int main()
{
    document_t document;
    document.reserve(5);

    document.emplace_back(0);
    document.emplace_back(1);
    document.emplace_back(2);
    document.emplace_back(3);

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

    draw(document, cout, 0);
}
```

cout | guidelines | defects

client | library

```
int main()
{
    document_t document;
    document.reserve(5);

    document.emplace_back(0);
    document.emplace_back(1);
    document.emplace_back(2);
    document.emplace_back(3);

    std::copy(document.begin(), document.end());
```

cout

```
ctor
ctor
ctor
ctor
copy
copy
copy
copy
<document>
 3
 2
 1
 0
</document>
```

```
class object_t {
  public:
    object_t(const int& x) : object_(new int_model_t(x))
    { cout << "ctor" << endl; }

    object_t(const object_t& x) : object_(new int_model_t(*x.object_))
    { cout << "copy" << endl; }
    object_t(object_t&& x) : object_(move(x.object_)) { }
    object_t& operator=(object_t x)
    { object_ = move(x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        int data_;
    };

   unique_ptr<int_model_t> object_;
};
```

```
class object_t {
  public:
    object_t(const int& x) : object_(new int_model_t(x))
    { cout << "ctor" << endl; }

    object_t(const object_t& x) : object_(new int_model_t(*x.object_))
    { cout << "copy" << endl; }
    object_t(object_t&& x) = default;
    object_t& operator=(object_t x)
    { object_ = move(x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        int data_;
    };

   unique_ptr<int_model_t> object_;
};
```

```
int main()
{
    document_t document;
    document.reserve(5);

    document.emplace_back(0);
    document.emplace_back(1);
    document.emplace_back(2);
    document.emplace_back(3);

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

    draw(document, cout, 0);
}
```

client | library

```
int main()
{
    document_t document;
    document.reserve(5);

    document.emplace_back(0);
    document.emplace_back(1);
    document.emplace_back(2);
    document.emplace_back(3);

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

    draw(document, cout, 0);
}
```

cout

```
ctor
ctor
ctor
ctor
<document>
 3
 2
 1
 0
</document>
```

client | library

```
int main()
{
    document_t document;
    document.reserve(5);

    document.emplace_back(0);
    document.emplace_back(1);
    document.emplace_back(2);
    document.emplace_back(3);

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

    draw(document, cout, 0);
}
```

guidelines

- Providing a move constructor allows copies to be elided where the compiler couldn't otherwise ovoid them.

client | library

```
class object_t {
  public:
    object_t(const int& x) : object_(new int_model_t(x))
    { cout << "ctor" << endl; }

    object_t(const object_t& x) : object_(new int_model_t(*x.object_))
    { cout << "copy" << endl; }
    object_t(object_t&& x) = default;
    object_t& operator=(object_t x)
    { object_ = move(x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        int data_;
    };

   unique_ptr<int_model_t> object_;
};
```

cout | guidelines | defects

client | library

```
class object_t {
  public:
    object_t(const int& x) : object_(new int_model_t(x))
    { }

    object_t(const object_t& x) : object_(new int_model_t(*x.object_))
    { }
    object_t(object_t&& x) = default;
    object_t& operator=(object_t x)
    { object_ = move(x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        int data_;
    };

   unique_ptr<int_model_t> object_;
};
```

cout | guidelines | defects

```
class object_t {
  public:
    object_t(const int& x) : object_(new int_model_t(x))
    { }

    object_t(const object_t& x) : object_(new int_model_t(*x.object_))
    { }
    object_t(object_t&& x) = default;
    object_t& operator=(object_t x)
    { object_ = move(x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        int data_;
    };

   unique_ptr<int_model_t> object_;
};
```

```
class object_t {
  public:
    object_t(const string& x) : object_(new string_model_t(x))
    { }
    object_t(const int& x) : object_(new int_model_t(x))
    { }

    object_t(const object_t& x) : object_(new int_model_t(*x.object_))
    { }
    object_t(object_t&& x) = default;
    object_t& operator=(object_t x)
    { object_ = move(x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct string_model_t {
        string_model_t(const string& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        string data_;
    };
```

```
    object_t& operator=(object_t x)
    { object_ = move(x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct string_model_t {
        string_model_t(const string& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        string data_;
    };

    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        int data_;
    };

    unique_ptr<int_model_t> object_;
};
```

```
    object_t& operator=(object_t x)
    { object_ = move(x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct string_model_t {
        string_model_t(const string& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        string data_;
    };

    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        int data_;
    };

    unique_ptr<concept_t> object_;
};
```

```
    object_t& operator=(object_t x)
    { object_ = move(x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct string_model_t {
        string_model_t(const string& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        string data_;
    };

    struct int_model_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        int data_;
    };

   unique_ptr<concept_t> object_;
};
```

```
    object_t& operator=(object_t x)
    { object_ = move(x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct concept_t {
        virtual ~concept_t() = default;
    };

    struct string_model_t : concept_t {
        string_model_t(const string& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        string data_;
    };

    struct int_model_t : concept_t {
        int_model_t(const int& x) : data_(x) { }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        int data_;
```

```
  object_t& operator=(object_t x)
  { object_ = move(x.object_); return *this; }

  friend void draw(const object_t& x, ostream& out, size_t position)
  { x.object_->draw_(out, position); }

private:
  struct concept_t {
      virtual ~concept_t() = default;
      virtual void draw_(ostream&, size_t) const = 0;
  };

  struct string_model_t : concept_t {
      string_model_t(const string& x) : data_(x) { }
      void draw_(ostream& out, size_t position) const
      { draw(data_, out, position); }

      string data_;
  };

  struct int_model_t : concept_t {
      int_model_t(const int& x) : data_(x) { }
      void draw_(ostream& out, size_t position) const
      { draw(data_, out, position); }
```

```
void draw(const string& x, ostream& out, size_t position)
{ out << string(position, ' ') << x << endl; }

void draw(const int& x, ostream& out, size_t position)
{ out << string(position, ' ') << x << endl; }

class object_t {
  public:
    object_t(const string& x) : object_(new string_model_t(x))
    { }
    object_t(const int& x) : object_(new int_model_t(x))
    { }

    object_t(const object_t& x) : object_(new int_model_t(*x.object_))
    { }
    object_t(object_t&& x) = default;
    object_t& operator=(object_t x)
    { object_ = move(x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct concept_t {
        virtual ~concept_t() = default;
```

```
void draw(const string& x, ostream& out, size_t position)
{ out << string(position, ' ') << x << endl; }

void draw(const int& x, ostream& out, size_t position)
{ out << string(position, ' ') << x << endl; }

class object_t {
  public:
    object_t(const string& x) : object_(new string_model_t(x))
    { }
    object_t(const int& x) : object_(new int_model_t(x))
    { }

    object_t(const object_t& x) : object_(new int_model_t(*x.object_))
    { }
    object_t(object_t&& x) = default;
    object_t& operator=(object_t x)
    { object_ = move(x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct concept_t {
        virtual ~concept_t() = default;
```

```
void draw(const string& x, ostream& out, size_t position)
{ out << string(position, ' ') << x << endl; }

void draw(const int& x, ostream& out, size_t position)
{ out << string(position, ' ') << x << endl; }

class object_t {
  public:
    object_t(const string& x) : object_(new string_model_t(x))
    { }
    object_t(const int& x) : object_(new int_model_t(x))
    { }

    object_t(const object_t& x) : object_(x.object_->copy_())
    { }
    object_t(object_t&& x) = default;
    object_t& operator=(object_t x)
    { object_ = move(x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct concept_t {
        virtual ~concept_t() = default;
```

```
private:
  struct concept_t {
      virtual ~concept_t() = default;
      virtual concept_t* copy_() = 0;
      virtual void draw_(ostream&, size_t) const = 0;
  };

  struct string_model_t : concept_t {
      string_model_t(const string& x) : data_(x) { }
      concept_t* copy_() { return new string_model_t(*this); }
      void draw_(ostream& out, size_t position) const
      { draw(data_, out, position); }

      string data_;
  };

  struct int_model_t : concept_t {
      int_model_t(const int& x) : data_(x) { }
      concept_t* copy_() { return new int_model_t(*this); }
      void draw_(ostream& out, size_t position) const
      { draw(data_, out, position); }

      int data_;
  };
```

client | library

```
int main()
{
    document_t document;

    document.emplace_back(0);
    document.emplace_back(1);
    document.emplace_back(2);
    document.emplace_back(3);

    draw(document, cout, 0);
}
```

cout | guidelines | defects

client | library

```
int main()
{
    document_t document;

    document.emplace_back(0);
    document.emplace_back(string("Hello!"));
    document.emplace_back(2);
    document.emplace_back(3);

    draw(document, cout, 0);
}
```

cout | guidelines | defects

client | library

```
int main()
{
    document_t document;

    document.emplace_back(0);
    document.emplace_back(string("Hello!"));
    document.emplace_back(2);
    document.emplace_back(3);

    draw(document, cout, 0);
}
```

cout

```
<document>
 0
 Hello!
 2
 3
</document>
```

client | library

```
int main()
{
    document_t document;

    document.emplace_back(0);
    document.emplace_back(string("Hello!"));
    document.emplace_back(2);
    document.emplace_back(3);

    draw(document, cout, 0);
}
```

guidelines

- Don't allow polymorphism to complicate the client code
  - Polymorphism is an implementation detail

```
void draw(const string& x, ostream& out, size_t position)
{ out << string(position, ' ') << x << endl; }

void draw(const int& x, ostream& out, size_t position)
{ out << string(position, ' ') << x << endl; }

class object_t {
  public:
    object_t(const string& x) : object_(new string_model_t(x))
    { }
    object_t(const int& x) : object_(new int_model_t(x))
    { }

    object_t(const object_t& x) : object_(x.object_->copy_())
    { }
    object_t(object_t&& x) = default;
    object_t& operator=(object_t x)
    { object_ = move(x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct concept_t {
        virtual ~concept_t() = default;
```

```
template <typename T>
void draw(const T& x, ostream& out, size_t position)
{ out << string(position, ' ') << x << endl; }

class object_t {
  public:
    template <typename T>
    object_t(const T& x) : object_(new model<T>(x))
    { }

    object_t(const object_t& x) : object_(x.object_->copy_())
    { }
    object_t(object_t&& x) = default;
    object_t& operator=(object_t x)
    { object_ = move(x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct concept_t {
        virtual ~concept_t() = default;
        virtual concept_t* copy_() = 0;
        virtual void draw_(ostream&, size_t) const = 0;
    };
```

client | library

```
        virtual ~concept_t() = default;
        virtual concept_t* copy_() = 0;
        virtual void draw_(ostream&, size_t) const = 0;
    };

    struct string_model_t : concept_t {
        string_model_t(const string& x) : data_(x) { }
        concept_t* copy_() { return new string_model_t(*this); }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        string data_;
    };

    struct int_model_t : concept_t {
        int_model_t(const int& x) : data_(x) { }
        concept_t* copy_() { return new int_model_t(*this); }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        int data_;
    };

    unique_ptr<concept_t> object_;
};
```

cout | guidelines | defects

```
        virtual ~concept_t() = default;
        virtual concept_t* copy_() = 0;
        virtual void draw_(ostream&, size_t) const = 0;
    };

    template <typename T>
    struct model : concept_t {
        model(const T& x) : data_(x) { }
        concept_t* copy_() { return new model(*this); }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        T data_;
    };

   unique_ptr<concept_t> object_;
};

using document_t = vector<object_t>;

void draw(const document_t& x, ostream& out, size_t position)
{
    out << string(position, ' ') << "<document>" << endl;
    for (auto& e : x) draw(e, out, position + 2);
    out << string(position, ' ') << "</document>" << endl;
```

```
class my_class_t {
    /* ... */
};

void draw(const my_class_t&, ostream& out, size_t position)
{ out << string(position, ' ') << "my_class_t" << endl; }

int main()
{
    document_t document;

    document.emplace_back(0);
    document.emplace_back(string("Hello!"));
    document.emplace_back(2);
    document.emplace_back(my_class_t());

    draw(document, cout, 0);
}
```

client | library

```
class my_class_t {
    /* ... */
};

void draw(const my_class_t&, ostream& out, size_t position)
{ out << string(position, ' ') << "my_class_t" << endl; }

int main()
{
    document_t document;

    document.emplace_back(0);
    document.emplace_back(string("Hello!"));
    document.emplace_back(2);
    document.emplace_back(my_class_t());

    draw(document, cout, 0);
}
```

cout

```
<document>
 0
 Hello!
 2
 my_class_t
</document>
```

client | library

```
class my_class_t {
    /* ... */
};

void draw(const my_class_t&, ostream& out, size_t position)
{ out << string(position, ' ') << "my_class_t" << endl; }

int main()
{
    document_t document;

    document.emplace_back(0);
    document.emplace_back(string("Hello!"));
    document.emplace_back(2);
    document.emplace_back(my_class_t());

    draw(document, cout, 0);
}
```

guidelines

- The runtime-concept idiom allows polymorphism when needed without inheritance.
  - Client isn't burdened with inheritance, factories, class registration, and memory management.
  - Penalty of runtime polymorphism is only payed when needed.
  - Polymorphic types are used like any other types, including built-in types.

```
class my_class_t {
    /* ... */
};

void draw(const my_class_t&, ostream& out, size_t position)
{ out << string(position, ' ') << "my_class_t" << endl; }

int main()
{
    document_t document;

    document.emplace_back(0);
    document.emplace_back(string("Hello!"));
    document.emplace_back(document);
    document.emplace_back(my_class_t());

    draw(document, cout, 0);
}
```

client | library

```
class my_class_t {
    /* ... */
};

void draw(const my_class_t&, ostream& out, size_t position)
{ out << string(position, ' ') << "my_class_t" << endl; }

int main()
{
    document_t document;

    document.emplace_back(0);
    document.emplace_back(string("Hello!"));
    document.emplace_back(document);
    document.emplace_back(my_class_t());
```

cout

```
<document>
 0
 Hello!
 <document>
  0
  Hello!
 </document>
 my_class_t
</document>
```

Polymorphic Use

- Shifting polymorphism from type to use allows for greater reuse and fewer dependencies
- Using regular semantics for the common basis operations, copy, assignment, and move helps to reduce shared objects
- Regular types promote interoperability of software components, increases productivity as well as quality, security, and performance
- There is no necessary performance penalty to using regular semantics, and often times there are performance benefits from a decreased use of the heap

Demo

Photoshop History

```
        model(const T& x) : data_(x) { }
        concept_t* copy_() { return new model(*this); }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        T data_;
    };

   unique_ptr<concept_t> object_;
};

using document_t = vector<object_t>;

void draw(const document_t& x, ostream& out, size_t position)
{
    out << string(position, ' ') << "<document>" << endl;
    for (auto& e : x) draw(e, out, position + 2);
    out << string(position, ' ') << "</document>" << endl;
}
```

```
        model(const T& x) : data_(x) { }
        concept_t* copy_() { return new model(*this); }
        void draw_(ostream& out, size_t position) const
        { draw(data_, out, position); }

        T data_;
    };

   unique_ptr<concept_t> object_;
};

using document_t = vector<copy_on_write<object_t>>;

void draw(const document_t& x, ostream& out, size_t position)
{
    out << string(position, ' ') << "<document>" << endl;
    for (auto& e : x) draw(e.read(), out, position + 2);
    out << string(position, ' ') << "</document>" << endl;
}

using history_t = vector<document_t>;

void commit(history_t& x) { assert(x.size()); x.push_back(x.back()); }
void undo(history_t& x) { assert(x.size()); x.pop_back(); }
document_t& current(history_t& x) { assert(x.size()); return x.back(); }
```

```
class object_t {
  public:
    template <typename T>
    object_t(const T& x) : object_(new model<T>(x))
    { }

    object_t(const object_t& x) : object_(x.object_->copy_())
    { }
    object_t(object_t&& x) = default;
    object_t& operator=(object_t x)
    { object_ = move(x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct concept_t {
        virtual ~concept_t() = default;
        virtual concept_t* copy_() = 0;
        virtual void draw_(ostream&, size_t) const = 0;
    };

    template <typename T>
    struct model : concept_t {
        model(const T& x) : data_(x) { }
```

```
class object_t {
  public:
    template <typename T>
    object_t(const T& x) : object_(new model<T>(x))
    { }

    object_t(const object_t& x) : object_(x.object_->copy_())
    { cout << "copy" << endl; }
    object_t(object_t&& x) = default;
    object_t& operator=(object_t x)
    { object_ = move(x.object_); return *this; }

    friend void draw(const object_t& x, ostream& out, size_t position)
    { x.object_->draw_(out, position); }

  private:
    struct concept_t {
        virtual ~concept_t() = default;
        virtual concept_t* copy_() = 0;
        virtual void draw_(ostream&, size_t) const = 0;
    };

    template <typename T>
    struct model : concept_t {
        model(const T& x) : data_(x) { }
```

client | library

```
class my_class_t {
    /* ... */
};

void draw(const my_class_t&, ostream& out, size_t position)
{ out << string(position, ' ') << "my_class_t" << endl; }

int main()
{
    document_t document;

    document.emplace_back(0);
    document.emplace_back(string("Hello!"));
    document.emplace_back(document);
    document.emplace_back(my_class_t());

    draw(document, cout, 0);
}
```

cout | guidelines | defects

client | library

```
{ out << string(position, ' ') << "my_class_t" << endl; }

int main()
{
    document_t document;

    document.emplace_back(0);
    document.emplace_back(string("Hello!"));
    document.emplace_back(document);
    document.emplace_back(my_class_t());

    draw(document, cout, 0);
}
```

cout | guidelines | defects

client | library

```
{ out << string(position, ' ') << "my_class_t" << endl; }

int main()
{
    history_t h(1);

    current(h).emplace_back(0);
    current(h).emplace_back(string("Hello!"));

    draw(current(h), cout, 0);
    cout << "------------------------" << endl;

    commit(h);

    current(h).emplace_back(current(h));
    current(h).emplace_back(my_class_t());
    current(h)[1] = string("World");

    draw(current(h), cout, 0);
    cout << "------------------------" << endl;

    undo(h);

    draw(current(h), cout, 0);
}
```

cout | guidelines | defects

client | library

```
{ out << string(position, ' ') << "my_class_t" << endl; }

int main()
{
```

cout

```
;
```

```
<document>
 0
 Hello!
</document>
------------------------
<document>
 0
 World
 <document>
  0
  Hello!
 </document>
 my_class_t
</document>
------------------------
<document>
 0
 Hello!
</document>
```

Concluding Remarks

- Generalize the language so we can write a library:

```
struct drawable {
    void draw(ostream& out, size_t position);
};

using object_t = poly<drawable>;
```

- Such a generalization would also allow the Objective-C runtime to be packaged as a usable C++ library


Adobe