Type Erasure

Nevin ":-)" Liber
nevin@eviloverlord.com

Type Erasure

- Type Erasure Pattern
- boost::any
- boost::function
- expression templates
- adobe::poly

Personal History

- Dave Abrahams
  - boost::function
    - Implemented poor man's boost::function
  - boost::shared_ptr
    - Didn't realize it
- Curiosity
  - Friend asked about boost::any
- BoostCon
  - Sean Parent Keynote
- Classes That Work

Definition

- "Process of turning a wide variety of types with a common interface into one type with that same interface."
  - C++ Template Metaprogramming by Dave Abrahams and Aleksey Gurtovoy

- More than that
  - You can adapt interfaces
  - You can morph inheritance based interfaces into value semantics
    - Copyable, Assignable, etc.
    - Directly stored in STL containers

- Tension in C++ between value semantics and inheritance
  - Interfaces don't mix well
  - Copying vs. cloning
  - Binary operators

```
struct AbstractMatrix {
    virtual ~AbstractMatrix() {}
    virtual bool operator==(AbstractMatrix const&) const = 0;
};
```

    - How do you implement operator== in derived classes?

- Virtual functions
  - Specify interface
  - Implementation detail

- First idea: Non-virtual Interface Idiom
  - Template Method pattern
  - Separate interface from implementation
  - Take virtual functions out of the public interface

Example

```
class MyType {
public:
    virtual void print() const = 0;
    virtual std::string str() const = 0;
};
```

Example

```
class MyType {
public:
    virtual void print() const = 0;
    virtual std::string str() const = 0;
};
```

- Non-virtual Interface Idiom

```
class MyType {
    virtual void printImpl() const = 0;
    virtual std::string strImpl() const = 0;
public:
    void print() const { return printImpl(); }
    std::string str() const { return strImpl(); }
};
```

- Problem:
  - Virtual functions are still part of the interface
  - They just aren't part of the public interface
  - Still have issues with slicing, copying, etc.

- Type Erasure takes this to the next level
  - Take inheritance out of the entire interface: public, protected & private
    - Make it an implementation detail

Example

- Non-virtual Interface Idiom

```
class MyType {
    virtual void printImpl() const = 0;
    virtual std::string strImpl() const = 0;
public:
    void print() const { return printImpl(); }
    std::string str() const { return strImpl(); }
};
```

Example

- Type Erasure

```
class MyType {
    /* ??? */

public:
    void print() const { /* ??? */ }
    std::string str() const { /* ??? */ }
};
```

Type Erasure Pattern

```
class TypeErasure {
    struct Concept {
        virtual ~Concept() {}
        virtual Concept* clone() const = 0;

        // Forwarding functions
        virtual void print() const = 0;
        virtual std::string str() const = 0;
    };

    template<typename T> struct Model : Concept {
        explicit Model(T const& data) : data(data) {}

        virtual Model* clone() const { return new Model(data); }

        // Forwarding functions
        virtual void print() const { return data.print(); }
        virtual std::string str() const { return data.str(); }

        T data;
    };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T>
    explicit TypeErasure(T const& data) : object(new Model<T>(data)) {}
    TypeErasure(TypeErasure const& that) : object(that.object ? that.object->clone() : 0) {}

    friend void swap(TypeErasure& lhs, TypeErasure& rhs)
    { boost::swap(lhs.object, rhs.object); }

    TypeErasure& operator=(TypeErasure rhs) { swap(*this, rhs); return *this; }

    // Forwarding functions
    void print() const { return object->print(); }
    std::string str() const { return object->str(); }
};
```

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T> explicit TypeErasure(T const& data);
    TypeErasure(TypeErasure const& that);

    friend void swap(TypeErasure& lhs, TypeErasure& rhs);
    TypeErasure& operator=(TypeErasure rhs);

    // Forwarding functions
    // ...
};
```

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T> explicit TypeErasure(T const& data);
    TypeErasure(TypeErasure const& that);

    friend void swap(TypeErasure& lhs, TypeErasure& rhs);
    TypeErasure& operator=(TypeErasure rhs);

    // Forwarding functions
    // ...
};
```

- TypeErasure is a concrete non-templated class which holds any type conforming to the interface

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T> explicit TypeErasure(T const& data);
    TypeErasure(TypeErasure const& that);

    friend void swap(TypeErasure& lhs, TypeErasure& rhs);
    TypeErasure& operator=(TypeErasure rhs);

    // Forwarding functions
    // ...
};
```

- Concept is an abstract base class corresponding to the interface being enforced

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T> explicit TypeErasure(T const& data);
    TypeErasure(TypeErasure const& that);

    friend void swap(TypeErasure& lhs, TypeErasure& rhs);
    TypeErasure& operator=(TypeErasure rhs);

    // Forwarding functions
    // ...
};
```

- Model<T> is an aggregate which adapts the interface (if necessary) and holds the actual data

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T> explicit TypeErasure(T const& data);
    TypeErasure(TypeErasure const& that);

    friend void swap(TypeErasure& lhs, TypeErasure& rhs);
    TypeErasure& operator=(TypeErasure rhs);

    // Forwarding functions
    // ...
};
```

- object is a pointer to the instantiated Model<T> holding the data

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T> explicit TypeErasure(T const& data);
    TypeErasure(TypeErasure const& that);

    friend void swap(TypeErasure& lhs, TypeErasure& rhs);
    TypeErasure& operator=(TypeErasure rhs);

    // Forwarding functions
    // ...
};
```

- Constructor which takes any type T whose interface matches
  - Can include fundamental types
  - Enforced by Model<T>

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T> explicit TypeErasure(T const& data);
    TypeErasure(TypeErasure const& that);

    friend void swap(TypeErasure& lhs, TypeErasure& rhs);
    TypeErasure& operator=(TypeErasure rhs);

    // Forwarding functions
    // ...
};
```

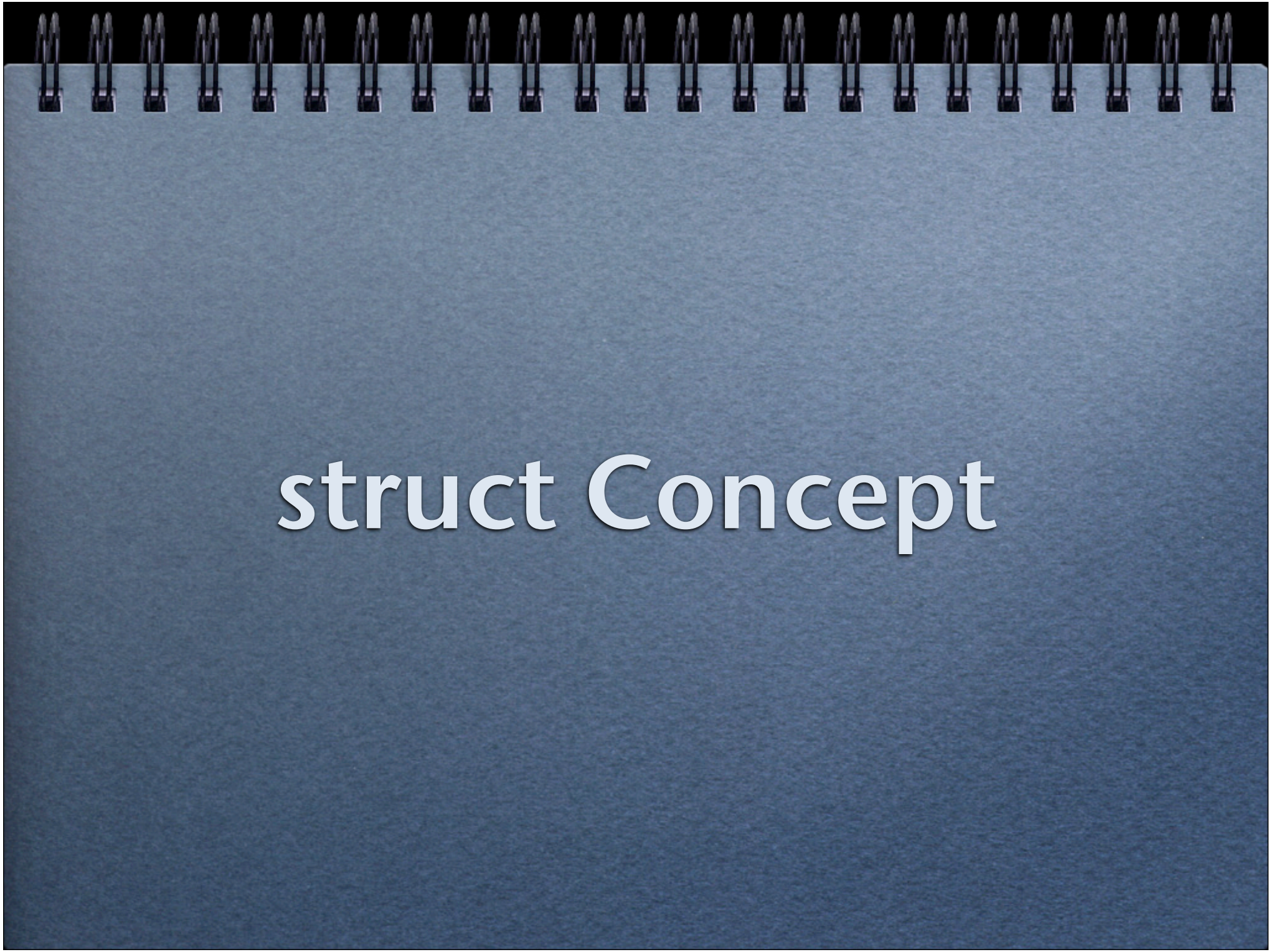
- TypeErasure is Copyable
  - Behind the scenes, clones object

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T> explicit TypeErasure(T const& data);
    TypeErasure(TypeErasure const& that);

    friend void swap(TypeErasure& lhs, TypeErasure& rhs);
    TypeErasure& operator=(TypeErasure rhs);

    // Forwarding functions
    // ...
};
```

- TypeErasure is assignable
  - copy / swap idiom
    - Strong exception safety guarantee

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T> explicit TypeErasure(T const& data);
    TypeErasure(TypeErasure const& that);

    friend void swap(TypeErasure& lhs, TypeErasure& rhs);
    TypeErasure& operator=(TypeErasure rhs);

    // Forwarding functions
    // ...
};
```

- Forwards functions to virtual functions inside object


struct Concept

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T> explicit TypeErasure(T const& data);
    TypeErasure(TypeErasure const& that);

    friend void swap(TypeErasure& lhs, TypeErasure& rhs);
    TypeErasure& operator=(TypeErasure rhs);

    // Forwarding functions
    // ...
};
```

- Concept is an abstract base class corresponding to the interface being enforced

```
class TypeErasure {
    struct Concept {
        virtual ~Concept() {}
        virtual Concept* clone() const = 0;

        // Forwarding functions
        virtual void print() const = 0;
        virtual std::string str() const = 0;
    };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    // ...
};
```

- Concept is an abstract base class corresponding to the interface being enforced

```
class TypeErasure {
    struct Concept {
        virtual ~Concept() {}
        virtual Concept* clone() const = 0;

        // Forwarding functions
        virtual void print() const = 0;
        virtual std::string str() const = 0;
    };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    // ...
};
```

- virtual destructor
  - scoped_ptr will delete through a pointer to Concept (base class)

```
class TypeErasure {
    struct Concept {
        virtual ~Concept() {}
        virtual Concept* clone() const = 0;

        // Forwarding functions
        virtual void print() const = 0;
        virtual std::string str() const = 0;
    };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    // ...
};
```

- clone
  - Copying TypeErasure instances is performed by cloning its object

```
class TypeErasure {
    struct Concept {
        virtual ~Concept() {}
        virtual Concept* clone() const = 0;

        // Forwarding functions
        virtual void print() const = 0;
        virtual std::string str() const = 0;
    };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    // ...
};
```

- Forwarding functions forward the work to the derived instance via virtual functions

struct Model

```
class TypeErasure {
    struct Concept {
        virtual ~Concept() {}
        virtual Concept* clone() const = 0;

        // Forwarding functions
        virtual void print() const = 0;
        virtual std::string str() const = 0;
    };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    // ...
};
```

- Model<T> is an aggregate which adapts the interface (if necessary) and holds the actual data

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept {
        explicit Model(T const& data) : data(data) {}

        virtual Model* clone() const { return new Model(data); }

        // Forwarding functions
        virtual void print() const { return data.print(); }
        virtual std::string str() const { return data.str(); }

        T data;
    };

    boost::scoped_ptr<Concept> object;

public:
    // ...
};
```

- Model<T> is an aggregate which adapts the interface (if necessary) and holds the actual data

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept {
        explicit Model(T const& data) : data(data) {}

        virtual Model* clone() const { return new Model(data); }

        // Forwarding functions
        virtual void print() const { return data.print(); }
        virtual std::string str() const { return data.str(); }

        T data;
    };

    boost::scoped_ptr<Concept> object;

public:
    // ...
};
```

- Model<T> derives from Concept
  - object will hold a pointer to Model<T>

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept {
        explicit Model(T const& data) : data(data) {}

        virtual Model* clone() const { return new Model(data); }

        // Forwarding functions
        virtual void print() const { return data.print(); }
        virtual std::string str() const { return data.str(); }

        T data;
    };

    boost::scoped_ptr<Concept> object;

public:
    // ...
};
```

- Constructor copies the data in

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept {
        explicit Model(T const& data) : data(data) {}

        virtual Model* clone() const { return new Model(data); }

        // Forwarding functions
        virtual void print() const { return data.print(); }
        virtual std::string str() const { return data.str(); }

        T data;
    };

    boost::scoped_ptr<Concept> object;

public:
    // ...
};
```

- clone() makes a copy of itself in the heap
- Constraint on T
  - Copyable

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept {
        explicit Model(T const& data) : data(data) {}

        virtual Model* clone() const { return new Model(data); }

        // Forwarding functions
        virtual void print() const { return data.print(); }
        virtual std::string str() const { return data.str(); }

        T data;
    };

    boost::scoped_ptr<Concept> object;

public:
    // ...
};
```

- No need for `assert(typeid(*this) == typeid(Model));`
- We control the depth of the inheritance
- Know Model<T> is the final class

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept {
        explicit Model(T const& data) : data(data) {}

        virtual Model* clone() const { return new Model(data); }

        // Forwarding functions
        virtual void print() const { return data.print(); }
        virtual std::string str() const { return data.str(); }

        T data;
    };

    boost::scoped_ptr<Concept> object;

public:
    // ...
};
```

- Forwarding functions forward the work to data to be performed

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept {
        explicit Model(T const& data) : data(data) {}

        virtual Model* clone() const { return new Model(data); }

        // Forwarding functions
        virtual void print() const { return data.print(); }
        virtual std::string str() const { return data.str(); }

        T data;
    };

    boost::scoped_ptr<Concept> object;

public:
    // ...
};
```

- The actual data for the instance of type T

class *TypeErasure*

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept {
        explicit Model(T const& data) : data(data) {}

        virtual Model* clone() const { return new Model(data); }

        // Forwarding functions
        virtual void print() const { return data.print(); }
        virtual std::string str() const { return data.str(); }

        T data;
    };

    boost::scoped_ptr<Concept> object;

public:
    // ...
};
```

- TypeErasure is a concrete non-templated class which holds any type conforming to the interface

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T>
    explicit TypeErasure(T const& data) : object(new Model<T>(data)) {}
    TypeErasure(TypeErasure const& that) : object(that.object->clone()) {}

    friend void swap(TypeErasure& lhs, TypeErasure& rhs)
    { boost::swap(lhs.object, rhs.object); }

    TypeErasure& operator=(TypeErasure rhs)
    { swap(*this, rhs); return *this; }

    // Forwarding functions
    void print() const { return object->print(); }
    std::string str() const { return object->str(); }
};
```

- TypeErasure is a concrete non-templated class which holds any type conforming to the interface

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T>
    explicit TypeErasure(T const& data) : object(new Model<T>(data)) {}
    TypeErasure(TypeErasure const& that) : object(that.object->clone()) {}

    friend void swap(TypeErasure& lhs, TypeErasure& rhs)
    { boost::swap(lhs.object, rhs.object); }

    TypeErasure& operator=(TypeErasure rhs)
    { swap(*this, rhs); return *this; }

    // Forwarding functions
    void print() const { return object->print(); }
    std::string str() const { return object->str(); }
};
```

- A pointer to Model<T> which holds the actual data

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T>
    explicit TypeErasure(T const& data) : object(new Model<T>(data)) {}
    TypeErasure(TypeErasure const& that) : object(that.object->clone()) {}

    friend void swap(TypeErasure& lhs, TypeErasure& rhs)
    { boost::swap(lhs.object, rhs.object); }

    TypeErasure& operator=(TypeErasure rhs)
    { swap(*this, rhs); return *this; }

    // Forwarding functions
    void print() const { return object->print(); }
    std::string str() const { return object->str(); }
};
```

- Constructor which accepts any type which is a model of the conceptual interface

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T>
    explicit TypeErasure(T const& data) : object(new Model<T>(data)) {}
    TypeErasure(TypeErasure const& that) : object(that.object->clone()) {}

    friend void swap(TypeErasure& lhs, TypeErasure& rhs)
    { boost::swap(lhs.object, rhs.object); }

    TypeErasure& operator=(TypeErasure rhs)
    { swap(*this, rhs); return *this; }

    // Forwarding functions
    void print() const { return object->print(); }
    std::string str() const { return object->str(); }
};
```

- Same expression and heap allocation that Model<T>::clone uses internally

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T>
    explicit TypeErasure(T const& data) : object(new Model<T>(data)) {}
    TypeErasure(TypeErasure const& that) : object(that.object->clone()) {}

    friend void swap(TypeErasure& lhs, TypeErasure& rhs)
    { boost::swap(lhs.object, rhs.object); }

    TypeErasure& operator=(TypeErasure rhs)
    { swap(*this, rhs); return *this; }

    // Forwarding functions
    void print() const { return object->print(); }
    std::string str() const { return object->str(); }
};
```

- If you wish to allow implicit conversions to TypeErasure, remove the explicit keyword

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T>
    explicit TypeErasure(T const& data) : object(new Model<T>(data)) {}
    TypeErasure(TypeErasure const& that) : object(that.object->clone()) {}

    friend void swap(TypeErasure& lhs, TypeErasure& rhs)
    { boost::swap(lhs.object, rhs.object); }

    TypeErasure& operator=(TypeErasure rhs)
    { swap(*this, rhs); return *this; }

    // Forwarding functions
    void print() const { return object->print(); }
    std::string str() const { return object->str(); }
};
```

- Gatekeeper
  - To limit types TypeErasure can hold, just change constructors

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T>
    explicit TypeErasure(T const& data) : object(new Model<T>(data)) {}
    TypeErasure(TypeErasure const& that) : object(that.object->clone()) {}

    friend void swap(TypeErasure& lhs, TypeErasure& rhs)
    { boost::swap(lhs.object, rhs.object); }

    TypeErasure& operator=(TypeErasure rhs)
    { swap(*this, rhs); return *this; }

    // Forwarding functions
    void print() const { return object->print(); }
    std::string str() const { return object->str(); }
};
```

- Copy constructor forwards to clone()
- object can never be NULL

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T>
    explicit TypeErasure(T const& data) : object(new Model<T>(data)) {}
    TypeErasure(TypeErasure const& that) : object(that.object->clone()) {}

    friend void swap(TypeErasure& lhs, TypeErasure& rhs)
    { boost::swap(lhs.object, rhs.object); }

    TypeErasure& operator=(TypeErasure rhs)
    { swap(*this, rhs); return *this; }

    // Forwarding functions
    void print() const { return object->print(); }
    std::string str() const { return object->str(); }
};
```

- swap() just swaps the object pointers
  - Non-throwing

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T>
    explicit TypeErasure(T const& data) : object(new Model<T>(data)) {}
    TypeErasure(TypeErasure const& that) : object(that.object->clone()) {}

    friend void swap(TypeErasure& lhs, TypeErasure& rhs)
    { boost::swap(lhs.object, rhs.object); }

    TypeErasure& operator=(TypeErasure rhs)
    { swap(*this, rhs); return *this; }

    // Forwarding functions
    void print() const { return object->print(); }
    std::string str() const { return object->str(); }
};
```

- Copy / Swap idiom
  - Strong exception safety guarantee

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T>
    explicit TypeErasure(T const& data) : object(new Model<T>(data)) {}
    TypeErasure(TypeErasure const& that) : object(that.object->clone()) {}

    friend void swap(TypeErasure& lhs, TypeErasure& rhs)
    { boost::swap(lhs.object, rhs.object); }

    TypeErasure& operator=(TypeErasure rhs)
    { swap(*this, rhs); return *this; }

    // Forwarding functions
    void print() const { return object->print(); }
    std::string str() const { return object->str(); }
};
```

- To allow implicit assignment, add:

```
    template<typename T>
    TypeErasure& operator=(T const& data)
    { object.reset(new Model<T>(data)); return *this; }
```

```
class TypeErasure {
    struct Concept { /* ... */ };
    template<typename T> struct Model : Concept { /* ... */ };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T>
    explicit TypeErasure(T const& data) : object(new Model<T>(data)) {}
    TypeErasure(TypeErasure const& that) : object(that.object->clone()) {}

    friend void swap(TypeErasure& lhs, TypeErasure& rhs)
    { boost::swap(lhs.object, rhs.object); }

    TypeErasure& operator=(TypeErasure rhs)
    { swap(*this, rhs); return *this; }

    // Forwarding functions
    void print() const { return object->print(); }
    std::string str() const { return object->str(); }
};
```

- Non-virtual forwarding functions have object do the actual work

```
class TypeErasure {
    struct Concept {
        virtual void print() const = 0;
        // ...
    };

    template<typename T> struct Model : Concept {
        virtual void print() const { return data.print(); }
        // ...
        T data;
    };

    boost::scoped_ptr<Concept> object;

public:
    void print() const { return object->print(); }
    // ...
};
```

- Non-virtual forwarding functions have object do the actual work

```
class TypeErasure {
    struct Concept {
        virtual void print() const = 0;
        // ...
    };

    template<typename T> struct Model : Concept {
        virtual void print() const { return data.print(); }
        // ...
        T data;
    };

    boost::scoped_ptr<Concept> object;

public:
    void print() const { return object->print(); }
    // ...
};
```

- Example: what happens when we call TypeErasure::print()?

```
class TypeErasure {
    struct Concept {
        virtual void print() const = 0;
        // ...
    };

    template<typename T> struct Model : Concept {
        virtual void print() const { return data.print(); }
        // ...
        T data;
    };

    boost::scoped_ptr<Concept> object;

public:
    void print() const { return object->print(); }
    // ...
};
```

- Non-virtual TypeErasure::print() calls through object

```
class TypeErasure {
    struct Concept {
        virtual void print() const = 0;
        // ...
    };

    template<typename T> struct Model : Concept {
        virtual void print() const { return data.print(); }
        // ...
        T data;
    };

    boost::scoped_ptr<Concept> object;

public:
    void print() const { return object->print(); }
    // ...
};
```

- Non-virtual TypeErasure::print() calls through object
- object calls through abstract virtual Concept::print()

```
class TypeErasure {
    struct Concept {
        virtual void print() const = 0;
        // ...
    };

    template<typename T> struct Model : Concept {
        virtual void print() const { return data.print(); }
        // ...
        T data;
    };

    boost::scoped_ptr<Concept> object;

public:
    void print() const { return object->print(); }
    // ...
};
```

- object calls through abstract virtual Concept::print()
- concrete call is to Model<T>::print()

```
class TypeErasure {
    struct Concept {
        virtual void print() const = 0;
        // ...
    };

    template<typename T> struct Model : Concept {
        virtual void print() const { return data.print(); }
        // ...
        T data;
    };

    boost::scoped_ptr<Concept> object;

public:
    void print() const { return object->print(); }
    // ...
};
```

- object calls through abstract virtual Concept::print()
- concrete call is to Model<T>::print()
- Model<T>::print() forwards to data

Example

```
struct MyInt
{
    explicit MyInt(int i)
    : i(i) {}

    void print() const
    {
        std::cout << "i=="
                  << i
                  << std::endl;
    }

    std::string str() const
    { return lexical_cast<std::string>(i); }

private:
    int i;
};
```

```
struct MyString
{
    explicit MyString(std::string const& s)
    : s(s) {}

    void print() const
    {
        std::cout << "s==\""
                  << s
                  << '\"' << std::endl;
    }

    std::string str() const
    { return s; }

private:
    std::string s;
};
```

- MyInt and MyString are independent unrelated classes
- Both Copyable, support print() and str()

```
struct MyInt
{
    explicit MyInt(int i)
    : i(i) {}

    void print() const
    {
        std::cout << "i=="
                  << i
                  << std::endl;
    }

    std::string str() const
    { return lexical_cast<std::string>(i); }

private:
    int i;
};
```

```
struct MyString
{
    explicit MyString(std::string const& s)
    : s(s) {}

    void print() const
    {
        std::cout << "s==\""
                  << s
                  << '\"' << std::endl;
    }

    std::string str() const
    { return s; }

private:
    std::string s;
};
```

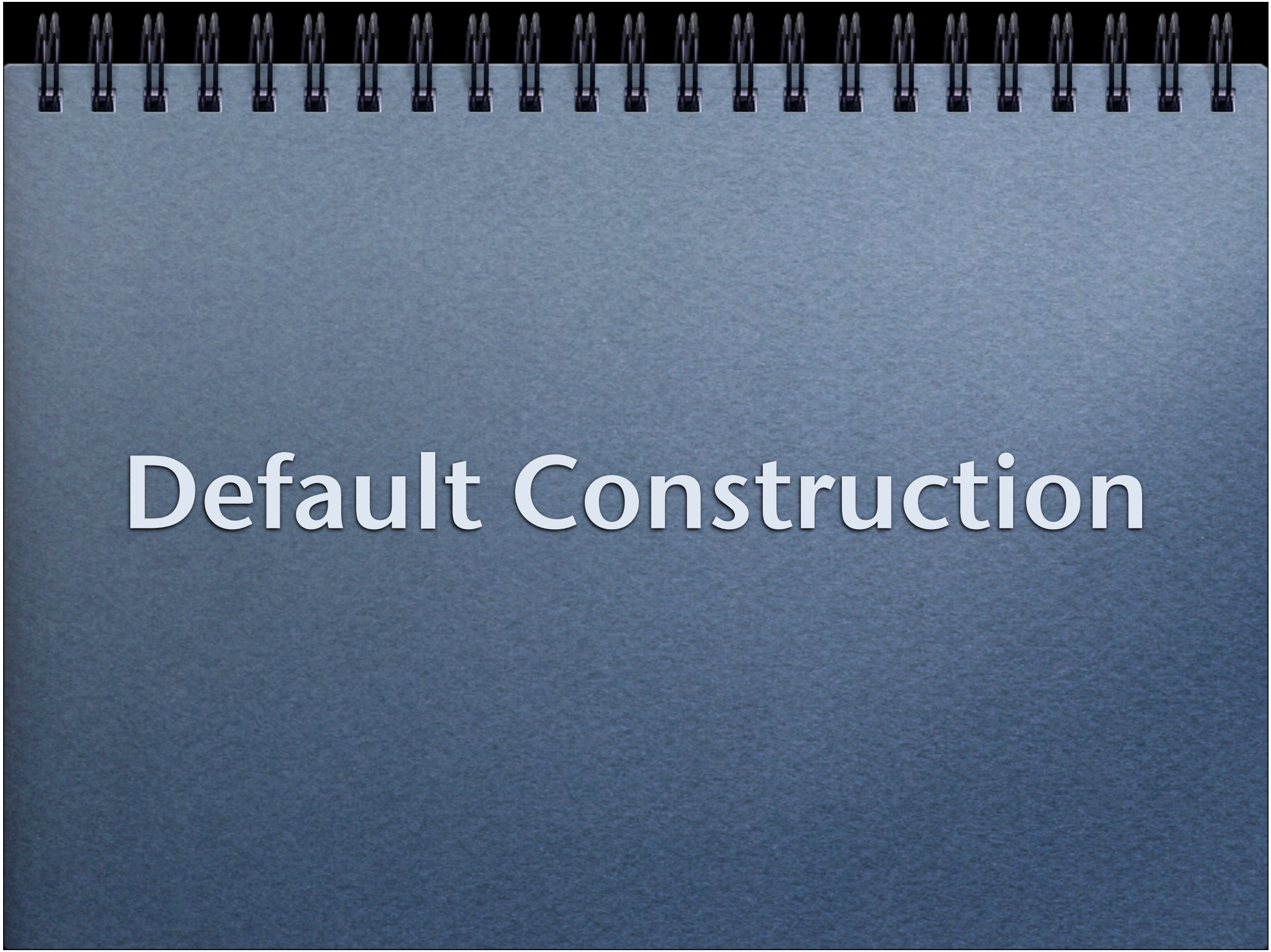
```
int main() {
    std::vector<TypeErasure> vte;
    vte.push_back(TypeErasure(MyInt(2)));
    vte.push_back(TypeErasure(MyString("three")));
    std::for_each(vte.begin(), vte.end(), boost::bind(&TypeErasure::print, _1));
}
```

```
struct MyInt
{
    explicit MyInt(int i)
    : i(i) {}

    void print() const
    {
        std::cout << "i=="
                  << i
                  << std::endl;
    }

    std::string str() const
    { return lexical_cast<std::string>(i); }

private:
    int i;
};
```

```
struct MyString
{
    explicit MyString(std::string const& s)
    : s(s) {}

    void print() const
    {
        std::cout << "s==\""
                  << s
                  << '\"' << std::endl;
    }

    std::string str() const
    { return s; }

private:
    std::string s;
};
```

```
int main() {
    std::vector<TypeErasure> vte;
    vte.push_back(TypeErasure(MyInt(2)));
    vte.push_back(TypeErasure(MyString("three")));
    std::for_each(vte.begin(), vte.end(), boost::bind(&TypeErasure::print, _1));
}
```

```
i==2
s=="three"
```


Default Construction

A: Check Pointers

```
class TypeErasure {
    // ...
    boost::scoped_ptr<Concept> object;

public:
    TypeErasure() {}
    TypeErasure(TypeErasure const& that)
    : object(that.object ? that.object->clone() : 0) {}
    //...
};
```

- When NULL == object, scoped_ptr will assert on dereference
- Easy to add isEmpty() check

B: Add a Null Model

```
class TypeErasure
{
    struct NullModel : Concept {
        virtual NullModel* clone() const { return new NullModel; };
        virtual void print() const { std::cout << "empty"; }
        virtual std::string str() const { return ""; }
    };
public:
    TypeErasure() : object(new NullModel) {}
    // ...
};
```

- Useful when adding behavior to default constructed case
  - Default construction can now throw
  - isEmpty() implemented by forwarding

C: Add a Static Model

```
class TypeErasure {
    static Concept* GetNullModel()
    {
        struct NullModel : Concept {
            virtual NullModel* clone() const
            { return const_cast<NullModel*>(this); }
            virtual void print() const { std::cout << "empty"; }
            virtual std::string str() const { return ""; }
        };

        static NullModel nm;
        return &nm;
    }

    Concept* object;
public:
    TypeErasure() : object(GetNullModel()) {}
    ~TypeErasure() { if (GetNullModel() != object) delete object; }
    // ...
};
```

- Has the advantages of NullModel
- Default construction cannot throw

C: Add a Static Model

```
class TypeErasure {
    static Concept* GetNullModel()
    {
        struct NullModel : Concept {
            virtual NullModel* clone() const
            { return const_cast<NullModel*>(this); }
            virtual void print() const { std::cout << "empty"; }
            virtual std::string str() const { return ""; }
        };

        static NullModel nm;
        return &nm;
    }

    Concept* object;
public:
    TypeErasure() : object(GetNullModel()) {}
    ~TypeErasure() { if (GetNullModel() != object) delete object; }
    // ...
};
```

- NullModel::clone() doesn't
- Threading issues (maybe)

Adapting Interfaces

- What if the interface is different?

```
struct MyDouble
{
    explicit MyDouble(double d)
    : d(d) {}

    friend std::ostream& operator<<(std::ostream& os, MyDouble const& md)
    { return os << "d==" << md.d << std::endl; }

    std::string stringify() const
    { return lexical_cast<std::string>(d); }

private:
    double d;
};
```

- print() is now free function operator<<
- str() is now spelled stringify()

- Solution: create another model

```
class TypeErasure
{
    template<typename T> struct Model2 : Concept {
        explicit Model2(T const& data) : data(data) {}

        virtual Model2* clone() const { return new Model2(data); }

        virtual void print() const { std::cout << data << std::endl; }
        virtual std::string str() const { return data.stringify(); }

        T data;
    };

public:
    explicit TypeErasure(MyDouble const& data)
    : object(new Model2<MyDouble>(data)) {}

    // ...
};
```

- TypeErasure constructors are gatekeepers

- Could also implement via specialization

Reducing copying costs

```
class TypeErasure {
    struct Concept {
        virtual Concept* clone() const = 0;
        // ...
    };
    template<typename T> struct Model : Concept {
        virtual Model* clone() const { return new Model(data); }
        // ...
    };
    boost::scoped_ptr<Concept> object;
public:
    TypeErasure(TypeErasure const& that) : object(that.object->clone()) {}
    TypeErasure& operator=(TypeErasure rhs) { swap(*this, rhs); return *this; }
    // ...
};
```

- Every copy or assignment is a clone()
  - Heap allocation
- How can we reduce this cost?

Reference Counting

```
class TypeErasure {
    struct Concept {
        virtual Concept* clone() const = 0;
        // ...
    };
    template<typename T> struct Model : Concept {
        virtual Model* clone() const { return new Model(data); }
        // ...
    };
    boost::scoped_ptr<Concept> object;
public:
    TypeErasure(TypeErasure const& that) : object(that.object->clone()) {}
    TypeErasure& operator=(TypeErasure rhs) { swap(*this, rhs); return *this; }
    // ...
};
```

Reference Counting

```
class SharedTypeErasure {
    struct Concept {
        virtual Concept* clone() const = 0;
        Concept() : refCount() {}

        friend void intrusive_ptr_add_ref(Concept* content)
        { ++content->refCount; }

        friend void intrusive_ptr_release(Concept* content)
        { if (!--content->refCount) delete content; }

        size_t refCount;
        // ...
    };
    template<typename T> struct Model : Concept {
        virtual Model* clone() const { return new Model(data); }
        // ...
    };
    boost::scoped_ptr<Concept> object;
    boost::intrusive_ptr<Concept> object;
public:
    TypeErasure(TypeErasure const& that) : object(that.object->clone()) {}
    TypeErasure& operator=(TypeErasure rhs) { swap(*this, rhs); return *this; }
    // ...
};
```

Reference Counting

```
class SharedTypeErasure {
    struct Concept {
        virtual Concept* clone() const = 0;
        Concept() : refCount() {}

        friend void intrusive_ptr_add_ref(Concept* content)
        { ++content->refCount; }

        friend void intrusive_ptr_release(Concept* content)
        { if (!--content->refCount) delete object; }

        size_t refCount;
        // ...
    };
    template<typename T> struct Model : Concept {
        virtual Model* clone() const { return new Model(data); }
        // ...
    };
    // ...
};
```

- Replace clone() with refCount

Reference Counting

```
class SharedTypeErasure {
    struct Concept {
        friend void intrusive_ptr_add_ref(Concept* content)
        { ++content->refCount; }

        friend void intrusive_ptr_release(Concept* content)
        { if (!--content->refCount) delete object; }
        // ...
    };
    boost::scoped_ptr<Concept> object;
    boost::intrusive_ptr<Concept> object;
    // ...
};
```

- Replaced scoped_ptr with intrusive_ptr

Reference Counting

```
class SharedTypeErasure {
    struct Concept {
        friend void intrusive_ptr_add_ref(Concept* content)
        { ++content->refCount; }

        friend void intrusive_ptr_release(Concept* content)
        { if (!--content->refCount) delete content; }
        // ...
    };
    boost::scoped_ptr<Concept> object;
    boost::intrusive_ptr<Concept> object;
    // ...
};
```

- Could also have used shared_ptr
  - Thread safe refCount access
  - Extra heap allocation per object

Reference Counting

```
class SharedTypeErasure {
    // ...
    boost::scoped_ptr<Concept> object;
    boost::intrusive_ptr<Concept> object;
public:
    TypeErasure(TypeErasure const& that) : object(that.object->clone()) {}
    TypeErasure& operator=(TypeErasure rhs) { swap(*this, rhs); return *this; }
    // ...
};
```

- Compiler generated copy constructor and assignment operator do the right thing
- Assignment operator non-throwing

```
class SharedTypeErasure {
    // ...
    static Concept* GetNullModel() {
        struct NullModel : Concept {
            virtual NullModel* clone() const
            { return const_cast<NullModel*>(this); }
            NullModel() { intrusive_ptr_add_ref(this); }
            // Forwarding functions
        };

        static NullModel nm;
        return &nm;
    }

    boost::intrusive_ptr<Concept> object;
public:
    SharedTypeErasure() : object(GetNullModel()) {}
    ~SharedTypeErasure() { if (GetNullModel() != object) delete object; }
    //...
};
```

- NullModel changes
  - Increment refCount on construction
  - intrusive_ptr never deletes instance

Reference Counting

```
class SharedTypeErasure {
    struct Concept {
        Concept() : refCount() {}

        friend void intrusive_ptr_add_ref(Concept* content)
        { ++content->refCount; }

        friend void intrusive_ptr_release(Concept* content)
        { if (!--content->refCount) delete content; }

        size_t refCount;
        // ...
    };
    template<typename T> struct Model : Concept {
        // ...
    };
    boost::intrusive_ptr<Concept> object;
public:
    // ...
};
```

- Caveat: shared copy semantics differ
- Might still want public clone()

Small Object Optimization

- For "small" derived objects, embed them as data in the TypeErasure class
  - Can change exception safety guarantees
  - More on this in upcoming examples

```
class TypeErasure {
    struct Concept {
        virtual ~Concept() {}
        virtual Concept* clone() const = 0;

        // Forwarding functions
        virtual void print() const = 0;
        virtual std::string str() const = 0;
    };

    template<typename T> struct Model : Concept {
        explicit Model(T const& data) : data(data) {}

        virtual Model* clone() const { return new Model(data); }

        // Forwarding functions
        virtual void print() const { return data.print(); }
        virtual std::string str() const { return data.str(); }

        T data;
    };

    boost::scoped_ptr<Concept> object;

public:
    template<typename T>
    explicit TypeErasure(T const& data) : object(new Model<T>(data)) {}
    TypeErasure(TypeErasure const& that) : object(that.object ? that.object->clone() : 0) {}

    friend void swap(TypeErasure& lhs, TypeErasure& rhs)
    { boost::swap(lhs.object, rhs.object); }

    TypeErasure& operator=(TypeErasure rhs) { swap(*this, rhs); return *this; }

    // Forwarding functions
    void print() const { return object->print(); }
    std::string str() const { return object->str(); }
};
```

boost::any

boost::any

- Safe, generic container for single values of different value types
- Concrete type
- Can hold any type that is
  - Copy Constructible

- boost::any operations
  - Store a value
    - construction/assignment/swap
  - Retrieve a value
    - any_cast
  - Query
    - typeid
    - is_empty

boost::any interface

```
class any {
public:
  // construct/copy/destruct
  any();
  any(const any &);
  template<typename ValueType> any(const ValueType &);
  any & operator=(const any &);
  template<typename ValueType> any & operator=(const ValueType &);
  ~any();

  // modifiers
  any & swap(any &);

  // queries
  bool empty() const;
  const std::type_info & type() const;
};

// extract
template<typename T> T any_cast(any & operand);
template<typename T> T any_cast(const any & operand);
template<typename ValueType> const ValueType * any_cast(const any * operand);
template<typename ValueType> ValueType * any_cast(any * operand);
```

```
class any
{
public: // structors

    any()
      : content(0)
    {
    }

    template<typename ValueType>
    any(const ValueType & value)
      : content(new holder<ValueType>(value))
    {
    }

    any(const any & other)
      : content(other.content ? other.content->clone() : 0)
    {
    }

    ~any()
    {
        delete content;
    }

public: // modifiers

    any & swap(any & rhs)
    {
        std::swap(content, rhs.content);
        return *this;
    }

    template<typename ValueType>
    any & operator=(const ValueType & rhs)
    {
        any(rhs).swap(*this);
        return *this;
    }

    any & operator=(any rhs)
    {
        rhs.swap(*this);
        return *this;
    }

public: // queries

    bool empty() const
    {
        return !content;
    }

    const std::type_info & type() const
    {
        return content ? content->type() : typeid(void);
    }

#ifndef BOOST_NO_MEMBER_TEMPLATE_FRIENDS
private: // types
#else
public: // types (public so any_cast can be non-friend)
#endif
```

```
    class placeholder
    {
    public: // structors

        virtual ~placeholder()
        {
        }

    public: // queries

        virtual const std::type_info & type() const = 0;

        virtual placeholder * clone() const = 0;

    };

    template<typename ValueType>
    class holder : public placeholder
    {
    public: // structors

        holder(const ValueType & value)
          : held(value)
        {
        }

    public: // queries

        virtual const std::type_info & type() const
        {
            return typeid(ValueType);
        }

        virtual placeholder * clone() const
        {
            return new holder(held);
        }

    public: // representation

        ValueType held;

    private: // intentionally left unimplemented
        holder & operator=(const holder &);
    };

#ifndef BOOST_NO_MEMBER_TEMPLATE_FRIENDS

    private: // representation

        template<typename ValueType>
        friend ValueType * any_cast(any *);

        template<typename ValueType>
        friend ValueType * unsafe_any_cast(any *);

#else

    public: // representation (public so any_cast can be non-friend)

#endif

        placeholder * content;
};
```

```
    class placeholder
    {
    public: // structors

        virtual ~placeholder()
        {
        }

    public: // queries

        virtual const std::type_info & type() const = 0;

        virtual placeholder * clone() const = 0;

    };

    template<typename ValueType>
    class holder : public placeholder
    {
    public: // structors

        holder(const ValueType & value)
          : held(value)
        {
        }

    public: // queries

        virtual const std::type_info & type() const
        {
            return typeid(ValueType);
        }

        virtual placeholder * clone() const
        {
            return new holder(held);
        }

    public: // representation

        ValueType held;

    private: // intentionally left unimplemented
        holder & operator=(const holder &);
    };

#ifndef BOOST_NO_MEMBER_TEMPLATE_FRIENDS

    private: // representation

        template<typename ValueType>
        friend ValueType * any_cast(any *);

        template<typename ValueType>
        friend ValueType * unsafe_any_cast(any *);

#else

    public: // representation (public so any_cast can be non-friend)

#endif

        placeholder * content;
};
```

```
class any
{
    // ...
    // types
    class placeholder
    {
    public: // structors

        virtual ~placeholder()
        {
        }

    public: // queries

        virtual const std::type_info & type() const = 0;

        virtual placeholder * clone() const = 0;

    };
    // ...
};
```

- placeholder is just another way to spell Concept
  - The forwarding function is type()

```
class any
{
    // ...
    template<typename ValueType>
    class holder : public placeholder
    {
    public: // structors
        holder(const ValueType & value)
          : held(value)
        {
        }
    public: // queries
        virtual const std::type_info & type() const
        {
            return typeid(ValueType);
        }

        virtual placeholder * clone() const
        {
            return new holder(held);
        }
    public: // representation
        ValueType held;

    private: // intentionally left unimplemented
        holder & operator=(const holder &);
    };
};
```

- holder<ValueType> is just another way to spell Model<T>
  - held is just another way to spell data
- The "work" to return type is just using typeid

```
class any
{
    // ...
    ~any()
    {
        delete content;
    }

    // ...

    placeholder * content;
};
```

- This has the same functionality as boost::scoped_ptr
  - content would be object in our nomenclature

```
class any
{
public: // structors

    any()
      : content(0)
    {
    }

    template<typename ValueType>
    any(const ValueType & value)
      : content(new holder<ValueType>(value))
    {
    }

    any(const any & other)
      : content(other.content ? other.content->clone() : 0)
    {
    }

    ~any()
    {
        delete content;
    }

public: // modifiers

    any & swap(any & rhs)
    {
        std::swap(content, rhs.content);
        return *this;
    }

    template<typename ValueType>
    any & operator=(const ValueType & rhs)
    {
        any(rhs).swap(*this);
        return *this;
    }

    any & operator=(any rhs)
    {
        rhs.swap(*this);
        return *this;
    }

public: // queries

    bool empty() const
    {
        return !content;
    }

    const std::type_info & type() const
    {
        return content ? content->type() : typeid(void);
    }

#ifndef BOOST_NO_MEMBER_TEMPLATE_FRIENDS
private: // types
#else
public: // types (public so any_cast can be non-friend)
#endif
```

```
    class placeholder
    {
    public: // structors

        virtual ~placeholder()
        {
        }

    public: // queries

        virtual const std::type_info & type() const = 0;

        virtual placeholder * clone() const = 0;

    };

    template<typename ValueType>
    class holder : public placeholder
    {
    public: // structors

        holder(const ValueType & value)
          : held(value)
        {
        }

    public: // queries

        virtual const std::type_info & type() const
        {
            return typeid(ValueType);
        }

        virtual placeholder * clone() const
        {
            return new holder(held);
        }

    public: // representation

        ValueType held;

    private: // intentionally left unimplemented
        holder & operator=(const holder &);
    };

#ifndef BOOST_NO_MEMBER_TEMPLATE_FRIENDS

    private: // representation

        template<typename ValueType>
        friend ValueType * any_cast(any *);

        template<typename ValueType>
        friend ValueType * unsafe_any_cast(any *);

#else

    public: // representation (public so any_cast can be non-friend)

#endif

        placeholder * content;
};
```

```
class any
{
public: // structors

    any()
      : content(0)
    {
    }

    template<typename ValueType>
    any(const ValueType & value)
      : content(new holder<ValueType>(value))
    {
    }

    any(const any & other)
      : content(other.content ? other.content->clone() : 0)
    {
    }

    ~any()
    {
        delete content;
    }

public: // modifiers

    any & swap(any & rhs)
    {
        std::swap(content, rhs.content);
        return *this;
    }

    template<typename ValueType>
    any & operator=(const ValueType & rhs)
    {
        any(rhs).swap(*this);
        return *this;
    }

    any & operator=(any rhs)
    {
        rhs.swap(*this);
        return *this;
    }

public: // queries

    bool empty() const
    {
        return !content;
    }

    const std::type_info & type() const
    {
        return content ? content->type() : typeid(void);
    }

#ifndef BOOST_NO_MEMBER_TEMPLATE_FRIENDS
private: // types
#else
public: // types (public so any_cast can be non-friend)
#endif
```

```
class any
{
public: // structors
    any()
      : content(0)
    {
    }

    template<typename ValueType>
    any(const ValueType & value)
      : content(new holder<ValueType>(value))
    {
    }

    any(const any & other)
      : content(other.content ? other.content->clone() : 0)
    {
    }
```

- Default constructor uses a NULL pointer
- Templated constructor takes any CopyConstructible value
- Copy constructor clones

```
public: // modifiers

    any & swap(any & rhs)
    {
        std::swap(content, rhs.content);
        return *this;
    }

    template<typename ValueType>
    any & operator=(const ValueType & rhs)
    {
        any(rhs).swap(*this);
        return *this;
    }

    any & operator=(any rhs)
    {
        rhs.swap(*this);
        return *this;
    }
```

- Copy/Swap idiom
  - swap is a member function here
- No surprises here...

```
public: // queries

    bool empty() const
    {
        return !content;
    }

    const std::type_info & type() const
    {
        return content ? content->type() : typeid(void);
    }
```

- empty() just checks for NULL pointer
- type() checks for NULL pointer
  - If NULL, return typeid(void)
  - If not NULL, forward the request to any::holder<valueType>::type()

- How does any_cast work?
  - Look at any_cast<ValueType>(any*)
    - minus compiler workarounds
  - Other versions of any_cast are similar

```
template<typename ValueType>
ValueType * any_cast(any * operand)
{
    return operand &&
#ifdef BOOST_AUX_ANY_TYPE_ID_NAME
        std::strcmp(operand->type().name(), typeid(ValueType).name()) == 0
#else
        operand->type() == typeid(ValueType)
#endif
        ? &static_cast<any::holder<ValueType> *>(operand->content)->held
        : 0;
}

template<typename ValueType>
inline const ValueType * any_cast(const any * operand)
{
    return any_cast<ValueType>(const_cast<any *>(operand));
}

template<typename ValueType>
ValueType any_cast(any & operand)
{
    typedef BOOST_DEDUCED_TYPENAME remove_reference<ValueType>::type nonref;

#ifdef BOOST_NO_TEMPLATE_PARTIAL_SPECIALIZATION
    // If 'nonref' is still reference type, it means the user has not
    // specialized 'remove_reference'.

    // Please use BOOST_BROKEN_COMPILER_TYPE_TRAITS_SPECIALIZATION macro
    // to generate specialization of remove_reference for your class
    // See type traits library documentation for details
    BOOST_STATIC_ASSERT(!is_reference<nonref>::value);
#endif

    nonref * result = any_cast<nonref>(&operand);
    if(!result)
        boost::throw_exception(bad_any_cast());
    return *result;
}

template<typename ValueType>
inline ValueType any_cast(const any & operand)
{
    typedef BOOST_DEDUCED_TYPENAME remove_reference<ValueType>::type nonref;

#ifdef BOOST_NO_TEMPLATE_PARTIAL_SPECIALIZATION
    // The comment in the above version of 'any_cast' explains when this
    // assert is fired and what to do.
    BOOST_STATIC_ASSERT(!is_reference<nonref>::value);
#endif

    return any_cast<const nonref &>(const_cast<any &>(operand));
}
```

```
template<typename ValueType>
ValueType * any_cast(any * operand)
{
    return operand &&
        operand->type() == typeid(ValueType)
        ? &static_cast<any::holder<ValueType> *>(operand->content)->held
        : 0;
}
```

- If operand is not NULL and the type of the data held (determined by using the forwarding function type) is correct, return a pointer to the held data; otherwise, return NULL

- Useful if you need to return the underlying object in your own TypeErasure class

Extending boost::any

- Suppose we wanted to be able to stream out boost::any?
  - If the underlying type has a stream insertion operator, use it
  - If not, use the (mangled static) name
- How would we add the functionality?
  - (No, I'm not proposing this...)

- We'd need a type trait to determine if the stream insert operator exists

- We'd need a type trait to determine if the stream insert operator exists

```
/*****************************************************************************/
/*! \class tsoob::is_ostreamable<T, charT = char, traits = std::char_traits<charT> >
*
*  \brief   metafunction type trait testing if a type T is output streamable
*
*  Requires: Given s of type std::basic_ostream<charT, traits>&
*  and x of type T const&,
*  if the expression s << x is well-formed
*  it must have complete type; otherwise, it must neither be ambiguous
*  nor violate access.
*
*  Inherits: If T is a (possibly cv-qualified) type which is output streamable,
*  then tsoob::is_streamable<T, charT, traits> inherits from boost::true_type;
*  otherwise inherits from boost::false_type.
*
*****************************************************************************/
```

```
#include <iosfwd>
#include <boost/type_traits/integral_constant.hpp>
#include <boost/type_traits/remove_cv.hpp>

namespace tsoob
{
    namespace detail
    {
        // This namespace ensures that ADL doesn't mess things up.
        namespace is_ostreamable_
        {
            // a type returned from operator<< when no operator<< is found in
            // the type's own namespace
            struct tag {};

            // any soaks up implicit conversions and makes the following
            // operator<< less-preferred than any other such operator that
            // might be found via ADL.
            struct any { template <typename T> any(T const&); };

            template<typename charT, typename traits>
            tag operator<<(::std::basic_ostream<charT, traits>&, any const&);

            // In case an operator<< is found that returns void,
            // we'll use s << x, 0
            tag operator,(tag, int);

            // two check overloads help us identify which operator<< was picked
            char (& check(tag) )[2];

            template<typename T>
            char check(T const&);

            template<typename T, typename charT, typename traits>
            struct impl
            {
                static ::std::basic_ostream<charT, traits>&     s;
                static typename ::boost::remove_cv<T>::type&    x;
                static const bool value =
                    1 == sizeof(::tsoob::detail::is_ostreamable_::check((s << x, 0)));
                typedef ::boost::integral_constant<bool, value> type;
            };
        }
    }

    // tsoob::is_ostreamable<T, charT = char, traits = std::char_traits<charT> >
    // metafunction
    template<typename T,
             typename charT = char,
             typename traits = ::std::char_traits<charT> >
    struct is_ostreamable
    : ::tsoob::detail::is_ostreamable_::impl<T, charT, traits>::type
    {};

}
```

- We'd need a type trait to determine if the stream insert operator exists
- What additions would we make to boost::any?

```
/*****************************************************************************/
/*! \class tsoob::is_ostreamable<T, charT = char, traits = std::char_traits<charT> >
*
*  \brief   metafunction type trait testing if a type T is output streamable
*
*  Requires: Given s of type std::basic_ostream<charT, traits>&
*  and x of type T const&,
*  if the expression s << x is well-formed
*  it must have complete type; otherwise, it must neither be ambiguous
*  nor violate access.
*
*  Inherits: If T is a (possibly cv-qualified) type which is output streamable,
*  then tsoob::is_streamable<T, charT, traits> inherits from boost::true_type;
*  otherwise inherits from boost::false_type.
*
*****************************************************************************/
```

```
#include <iosfwd>
#include <boost/type_traits/integral_constant.hpp>
#include <boost/type_traits/remove_cv.hpp>

namespace tsoob
{
    namespace detail
    {
        // This namespace ensures that ADL doesn't mess things up.
        namespace is_ostreamable_
        {
            // a type returned from operator<< when no operator<< is found in
            // the type's own namespace
            struct tag {};

            // any soaks up implicit conversions and makes the following
            // operator<< less-preferred than any other such operator that
            // might be found via ADL.
            struct any { template <typename T> any(T const&); };

            template<typename charT, typename traits>
            tag operator<<(::std::basic_ostream<charT, traits>&, any const&);

            // In case an operator<< is found that returns void,
            // we'll use s << x, 0
            tag operator,(tag, int);

            // two check overloads help us identify which operator<< was picked
            char (& check(tag) )[2];

            template<typename T>
            char check(T const&);

            template<typename T, typename charT, typename traits>
            struct impl
            {
                static ::std::basic_ostream<charT, traits>&     s;
                static typename ::boost::remove_cv<T>::type&    x;
                static const bool value =
                    1 == sizeof(::tsoob::detail::is_ostreamable_::check((s << x, 0)));
                typedef ::boost::integral_constant<bool, value> type;
            };
        }
    }

    // tsoob::is_ostreamable<T, charT = char, traits = std::char_traits<charT> >
    // metafunction
    template<typename T,
             typename charT = char,
             typename traits = ::std::char_traits<charT> >
    struct is_ostreamable
    : ::tsoob::detail::is_ostreamable_::impl<T, charT, traits>::type
    {};

}
```

```
class placeholder
{
public: // structors

    virtual ~placeholder()
    {
    }

public: // queries

    virtual const std::type_info & type() const = 0;

    virtual placeholder * clone() const = 0;

    virtual void inserter(std::ostream&) const = 0;

};
```

- virtual inserter forwarding function
  - Because inserter is virtual, parameter cannot be a templated basic_ostream
    - Could use type erasure to allow different types

```
template< typename ValueType,
          typename IsOstreamable = typename tsoob::is_ostreamable<ValueType>::type >
class holder : public placeholder
{
    virtual void inserter(std::ostream& os) const
    {
        os << typeid(ValueType).name();
    }
    //...
};
template<typename ValueType>
class holder<ValueType, boost::true_type> : public placeholder
{
    virtual void inserter(std::ostream& os) const
    {
        os << held;
    }
    // ...
};
```

- Pick the holder implementation based upon whether or not the held type is ostreamable
- Other functions in holder are identical

```
class any
{
    // ...
    friend std::ostream& operator<<(std::ostream& os, any const& a)
    {
        if (a.content)
            a.content->inserter(os);
        else
            os << typeid(void).name();
        return os;
    }
};
```

- Add a stream insertion operator to any which forwards its work to inserter()
  - Pesky NULL pointer check, though

Test Cases

```
any a;
std::cout << a << std::endl;     // v

a = 5;
std::cout << a << std::endl;     // 5

a = MyString("seven");
std::cout << a << std::endl;     //8MyString
```

- That's it!

boost::function

boost::function

- Function object wrappers for deferred calls or callbacks
- boost::function<void()> can hold:
  - Concrete (non-templated) type

boost::function<void()>

- Can hold and call anything with that calling signature
  - Function
    - `void Hello();`
  - Function object
    - `struct Goodbye { void operator()() const; /* ... */ };`
  - boost::bind
  - boost::lambda
  - lambda

- Won't go through actual implementation
  - Lots of macros
  - Lots of stuff not pertaining to type erasure
    - Function pointer or function object
    - Number of parameters
    - Compiler workarounds

- One interesting note from a type erasure point of view...
- Small object optimization
  - "small" function objects are stored in the instance directly, not on the heap
  - Changes exception safety guarantees
    - Basic for assignment
    - swap can throw

Expression Templates

Expression Templates

- Capture parse tree of a C++ expression
- Typically used for delayed evaluation
  - boost::lambda
  - boost::spirit
  - boost::proto
- How can we store an expression template?
  - Type erasure

adobe::poly

adobe::poly

- Type erasure on steroids
- Take Concept and Model out of the class and pass them in as template parameters
  - Externally "Model" inherits from "Concept"

adobe::poly

- Throw in small object optimization
  - Keep same exception safety guarantees
- Move semantics for C++03
- Concept checking
- It is a lot of machinery…

```
struct poly_copyable_interface;
struct Concept : poly_copyable_interface { /* ... */ }

template <typename T>
struct Model : optimized_storage_type<T, Concept>::type { /* ... */ };

struct MyType : poly_base<Concept, Model> { /* ... */ };

typedef poly<MyType> MyRegularType;
```

- Lots of magic...

Small Object Optimization

Picking the storage type

```
typedef double storage_t[2];

template<typename T, int N=sizeof(storage_t)>
struct is_small
{
    enum { value = sizeof(T) <= N &&
          (boost::has_nothrow_constructor<typename T::value_type>::value ||
           boost::is_same<std::string, typename T::value_type>::value) };
};



template <typename ConcreteType, typename Interface>
struct optimized_storage_type :
    boost::mpl::if_<implementation::is_small<
                    implementation::poly_state_local<ConcreteType, Interface> >,
                    implementation::poly_state_local<ConcreteType, Interface>,
                    implementation::poly_state_remote<ConcreteType, Interface> > {
};
```

```
template <typename I, template <typename> class Instance>
struct poly_base {
    // ...
    friend inline void swap(poly_base& x, poly_base& y)
    {
        interface_type& a(x.interface_ref());
        interface_type& b(y.interface_ref());

        if (a.type_info() == b.type_info()) { a.exchange(b); return; }

        // x->tmp
        poly_base tmp(move(x));
        a.~interface_type();

        // y->x
        b.move_clone(x.storage());
        b.~interface_type();

        // tmp->y
        tmp.interface_ref().move_clone(y.storage());
    }
```

- Complicated because storage for a and b may be different (heap vs. small object)

- adobe::any_iterator implemented this way
  - Performance just wasn't there
  - What is the iterator category?

References

Classes That Work, Eric Berdahl, Sean Parent, *http://stlab.adobe.com/wiki/index.php/Image:2008_06_26_classes_that_work.pdf*

Valued Conversions, Kevlin Henney, *http://www.two-sdg.demon.co.uk/curbralan/papers/ValuedConversions.pdf*

http://www.boost.org/

References *(background stuff)*

C++ Template Metaprogramming, David Abrahams & Aleksey Gurtovoy, *http://www.boostpro.com/mplbook/*

Virtuality, Herb Sutter, *http://www.gotw.ca/publications/mill18.htm*

References *(Adobe stuff)*

Runtime Polymorphic Generic Programming—Mixing Objects and Concepts in ConceptC++, Mat Marcus, Jaakko Järvi & Sean Parent, *http://www.emarcus.org/papers/MPOOL2007-marcus.pdf*

Library Composition and Adaptation using C++ Concepts, Jaakko Järvi, Mat Marcus and Jacob N. Smith, *http://portal.acm.org/citation.cfm?doid=1289971.1289984*

http://stlab.adobe.com/

References (alternate any_iter)

On the Tension Between Object-Oriented and Generic Programming in C++, Thomas Becker, *http://www.artima.com/cppsource/type_erasure.html*

Type Erasure in C++: The Glue Between Object-Oriented and Generic Programming, Thomas Becker, *http://homepages.fh-regensburg.de/~mpool/mpool07/proceedings/4.pdf*

any_iterator: Type Erasure for C++ Iterators, Thomas Becker, *http://thbecker.net/free_software_utilities/type_erasure_for_cpp_iterators/any_iterator.html*

Special Thanks

Jon Kalb
Marshall Clow
Mat Marcus

(thank them / blame me)


Q&A