Type Erasure

Motivation

Ok, but generic Programming solves those!

Wait, there are problems with this approach?

Problems successfully replaced with others

Ah, so its polymorphism!

So, how can we have our cake and eat it too?

Huh?

First contact

You are probably already using it

boost::any

[std|boost]::function

Star Wars

A new Hope ...

Return of the Jedi

... but the empire strikes back

Real Life

boost to the rescue

boost::type_erasure

Having and eating our cake

- Uninstrusive
- Separate compilation
- Run-time selection
- Compile-time concept checks
- Composable
- Value based

Watch "Value Semantics"

*Limitations

- involved concept definition syntax
- complex semantics with multiple parameters
- limitied flexibility in "concept - deduction"
- cannot adapt free functions?

Thank You!

Prezi

Type Erasure

Prezi

Motivation

Ok, but generic Programming solves those!

```
template <class OFFICER>
void boldly_go(OFFICER officer) {
    officer.give_order();
}

picard jean_luc;
boldly_go(jean_luc);
```

Wait, there are problems with this approach?

- It is intrusive
- Applying independent concepts to an object is difficult
- It requires pointer semantics => dynamic memory

Problems succesfully replaced with others

- Static dependency on the full type
- No run-time selection
- No separate compilation

Ah, so its polymorphism!

```
struct captain {
    virtual void give_order();
};
struct kirk : captain {
    void warp4_mr_sulu();
};
struct picard : captain {
    void engage();
};
```

```
void boldly_go(captain* cpt) {
    cpt->give_order();
}

kirk* james = new kirk();
boldly_go(james);
```

So, how can we have our cake and eat it too?

Huh?

"The process of turning a wide variety of types with a common interface into one type with that same interface"

Dave Abrahams and Aleksey Gurtovoy

Type Erasure

Prezi

So, how can we have our cake and eat it too?

Huh?

"The process of turni
common interface int
interface"

Type Erasure

First contact

You are probably already using it

boost::any

[std|boost]::function

Star Wars

A new Hope ...

Prezi

Type Erasure

First contact

You are probably already using it

- boost::any
- [std|boost]::function

Star Wars

A new Hope ...

```
struct young_kirk {
    void misbehave();
    void give_order();
};

young_kirk james;
james.misbehave();
board.emergency(james);
```

```
struct one_who_gives_orders {
    void give_order();
};
using leader = any<one_who_gives_orders>;

void emergency_promote(leader lead) {
    lead.give_order();
}
```

```
emergency_promote(james);
```

```
leader select_someone_suitable();
emergency_promote(select_someone_suitable());
```

```
struct young_spock {
    void be_logical();
    void live_long_and_prosper();
};

leader select_someone_suitable() {
    return young_spock();
}
```

won't compile

Return of the Jedi

boost::type_erasure by Steven Watanabe
Accepted into boost, but not yet released.

Documentation:
http://steven_watanabe.users.sourceforge.net/type_erasure/libs/type_erasure/doc/html/index.html

Code:
http://svn.boost.org/svn/boost/sandbox/type_erasure

boost to the rescue

```
template <class CLASS>
struct one_who_gives_orders {
    static void apply(CLASS& c) { c.give_order(); }
};

using leader = bte::any< mpl::vector<
                                bte::copy_constructible<>,
                                one_who_gives_orders<bte::_self> >>;

void emergency_promote(leader lead) {
    bte::call<one_who_gives_orders<bte::_self>>(lead);   //lead.give_order();
}
```

... but the empire strikes back

```
template<CONCEPT> struct any {


    Unimplementable , without a language feature
};

any<one_who_gives_orders> leader;
```

```
template <class CLASS>
struct one_who_gives_orders {
    static void apply(CLASS& c) { c.give_order(); }
};

namespace boost { namespace type_erasure {
    template<class CLASS, class Base>
        struct concept_interface< one_who_gives_orders<CLASS>, Base, CLASS> : Base {
            void give_order() { call<one_who_gives_orders<CLASS>>(*this); }
        };
} }

using leader = bte::any< mpl::vector<
                                bte::copy_constructible<>,
                                one_who_gives_orders<bte::_self> >>;

void emergency_promote(leader lead) {
    lead.give_order();
}
```

Real Lif

Prezi

Type Erasure

*Limitations

- involved concept definition syntax
- complex semantics with multiple parameters
- limitied flexibility in "concept - deduction"
- cannot adapt free functions?

Real Life

Watch "Value Semantics"

by Sean Parent to see the full power
this technique enables ...

see here ... youtube.com/watch?v=_BpMYeUFXv8

Other contenders ...

Adobe::poly
DynamicAny

Star Wars

Having and eating our cake

Best of both worlds*

- Unintrusive
- Separate compilation
- Run-time selection
- Compile-time concept checks
- Composable
- Value based

Hope ...

```
struct one_who_gives_orders {
    void give_order();
};
using leader = any<one_who_gives_orders>;

void emergency_promote(leader lead) {
    lead.give_order();
}
```

```
emergency_promote(james);

leader select_someone_suitable();
emergency_promote(select_someone_suitable());

struct young_spock {
    void be_logical();
    void live_long_and_prosper();
};

leader select_someone_suitable() {
    return young_spock();
}
```

won't compile

boost::type_erasure

building block for any type erased type

Handles

- references
- concept composition
- concept mapping
- (construction)
- (functions with multple arguments)
- (overloading)
- binary compatibility ?

Return of the Jedi

boost::type_erasure by Steven Watanabe
Accepted into boost, but not yet released.

Documentation:
http://steven_watanabe.users.sourceforge.net/type_erasure/libs/type_erasure/doc/html/index.html

Code:
http://svn.boost.org/svn/boost/sandbox/type_erasure

boost to the rescue

```
template <class CLASS>
struct one_who_gives_orders {
    static void apply(CLASS& c) { c.give_order(); }
};

using leader = bte::any< mpl::vector<
                         bte::copy_constructible<>,
                         one_who_gives_orders<bte::_self> >>;

void emergency_promote(leader lead) {
    bte::call(one_who_gives_orders<bte::_self>(), lead);    //lead.give_order();
}
```

```
template <class CLASS>
struct one_who_gives_orders {
    static void apply(CLASS& c) { c.give_order(); }
};
namespace boost { namespace type_erasure {
    template<class CLASS, class Base>
        struct concept_interface< ::one_who_gives_orders<CLASS>, Base, CLASS> : Base {
            void give_order() { call(::one_who_gives_orders<CLASS>(), *this); }
        };
}}

using leader = bte::any< mpl::vector<
                         bte::copy_constructible<>,
                         one_who_gives_orders<bte::_self> >>;

void emergency_promote(leader lead) {
    lead.give_order();
}
```

the empire strikes back

Prezi

Type Erasure

Prezi

Huh?

"The process of turning a wide variety of types with a common interface into one type with that same interface"

Dave Abrahams and Aleksey Gurtovoy

Prezi

Ah, so its polymorphism!

```
struct captain {
    virtual void give_order();
};
struct kirk : captain {
    void warp4_mr_sulu();
};
struct picard : captain {
    void engage();
};
```

```
void boldly_go(captain* cpt) {
    cpt->give_order();
}

kirk* james = new kirk();
boldly_go(james);
```

Type (partially) erased

We do not care how the captain gives his/her orders, we do not even care who he/she is.

Prezi

orphism!

```
void boldly_go(captain* cpt) {
    cpt->give_order();
}
```

Type (partially) erased

We do not care how the captain gives his/her orders, we do not even care who he/she is.

```
kirk* james = new kirk();
```

Prezi

Thank You!

f.fracassi@gmx.net

Prezi

Ah, so its polymorphism!

```
struct captain {
    virtual void give_order();
};
struct kirk : captain {
    void warp4_mr_sulu();
};
struct picard : captain {
    void engage();
};
```

```
void boldly_go(captain* cpt) {
    cpt->give_order();
}

kirk* james = new kirk();
boldly_go(james);
```

Type (partially) erased

We do not care how the captain gives his/her orders, we do not even care who he/she is.

Prezi

Wait, there are problems with this approach?

- It is intrusive
- Applying independent concepts to an object is difficult
- It requires pointer semantics => dynamic memory

Prezi

Ok, but generic Programming solves those!

```
template <class OFFICER>
void boldly_go(OFFICER officer) {
    officer.give_order();
}

picard jean_luc;
boldly_go(jean_luc);
```

Prezi

Problems succesfully replaced with others

- Static dependency on the full type
- No run-time selection
- No separate compilation

Prezi

So, how can we have our cake and eat it too?

Prezi

Type Erasure

Prezi

You are probably already using it

boost::any

```
void black_box(boost::any param) {
    boost::any copy = param;
}

boost::any value = 5;
black_box(value);
```

[std|boost]::function

```
using functions = std::vector<std::function<void (int)>>;
void call_all(functions funs, int value) {
    for(auto& fun : funs) fun(value);
}

void one(int p) { std::cout << "1: " << p << std::endl; }
void two(int p) { std::cout << "2: " << p * 2 << std::endl; }

functions f;
    f.emplace_back(&one);
    f.emplace_back(&two);
call_all(f, 21);
```

Prezi

boost::any

```
void black_box(boost::any param) {
    boost::any copy = param;
}

boost::any value = 5;
black_box(value);
```

Prezi

```
struct any {
  ~any() { delete content; }

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

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

private: // types

  struct placeholder {

    virtual ~placeholder() {}
    virtual placeholder * clone() const = 0;
  };

  template<typename ValueType>
  struct holder : placeholder {

    holder(const ValueType & value) : held(value) {}
    virtual placeholder * clone() const { return new holder(held); }

    private:
      ValueType held;
  };

  placeholder * content;
};
```

Prezi

[std|boost]::function

```
using functions = std::vector<std::function<void (int)>>;
void call_all(functions funs, int value) {
    for(auto& fun : funs) fun(value);
}

void one(int p) { std::cout << "1: " << p << std::endl; }
void two(int p) { std::cout << "2: " << p * 2 << std::endl; }

functions f;
    f.emplace_back(&one);
    f.emplace_back(&two);
call_all(f, 21);
```

Prezi

```
struct any_fun {
  ~any_fun() { delete content; }

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

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

  void operator()(int param) { assert(content); content->call(param); }

private: // types

  struct placeholder {

    virtual ~placeholder() {}
    virtual placeholder * clone() const = 0;

    virtual void call(int param) const = 0;
  };

  template<typename ValueType>
  struct holder : placeholder {

    holder(const ValueType & value) : held(value) {}
    virtual placeholder * clone() const { return new holder(held); }

    virtual void call(int param) const { held(param); }

    private:
      ValueType held;
  };

  placeholder * content;
};
```

Prezi

```
struct any_fun {
  ~any_fun() { delete content; }

  any_fun(const any_fun & other) : content(other.content ? other.conten

  template<typename ValueType>
  any_fun(const ValueType & value) : content(new holder<ValueType>(

  void operator()(int param) { assert(content); content->call(param); }

private: // types

  struct placeholder {

    virtual ~placeholder() {}
    virtual placeholder * clone() const = 0;
```

Prezi

```
    virtual ~placeholder() {}
    virtual placeholder * clone() const = 0;

    virtual void call(int param) const = 0;
};

template<typename ValueType>
struct holder : placeholder {
```

Prezi

```
template<typename ValueType>
struct holder : placeholder {

  holder(const ValueType & value) : held(value) {}
  virtual placeholder * clone() const { return new hol

  virtual void call(int param) const { held(param); }

  private:
    ValueType held;
};

placeholder * content;
```

A new Hope ...

```
struct young_kirk {
    void misbehave();
    void give_order();
};

young_kirk james;
james.misbehave();
board_enterprise(james);
```

```
struct one_who_gives_orders {
    void give_order();
};
using leader = any<one_who_gives_orders>;

void emergency_promote(leader lead) {
    lead.give_order();
}
```

```
emergency_promote(james);
```

```
leader select_someone_suitable();
```

Prezi

A new Hope ...

```
struct young_kirk {
    void misbehave();
    void give_order();
};

young_kirk james;
james.misbehave();
board_enterprise(james);
```

```
struct one_who_gives_orders {
    void give_order();
};
using leader = any<one_who_gives_orders>;

void emergency_promote(leader lead) {
    lead.give_order();
}
```

```
emergency_promote(james);
```

```
leader select_someone_suitable();
emergency_promote(select_someone_suitable());
```

Prezi

A new Hope ...

```
struct young_kirk {
    void misbehave();
    void give_order();
};

young_kirk james;
james.misbehave();
board_enterprise(james);
```

```
struct one_who_gives_orders {
    void give_order();
};
using leader = any<one_who_gives_orders>;

void emergency_promote(leader lead) {
    lead.give_order();
}
```

```
emergency_promote(james);
```

```
leader select_someone_suitable();
emergency_promote(select_someone_suitable());
```

```
struct young_spock {
    void be_logical();
    void live_long_and_prosper();
};

leader select_someone_suitable() {
    return young_spock();
}
```

won't compile

Prezi

… but the empire strikes back

```
template<CONCEPT> struct any {

        Unimplementable , without a language feature

};

any<one_who_gives_orders> leader;
```

Prezi

- A dedicated feature for exactly this use-case
- Static Reflection

Prezi

Return of the Jedi

boost::type_erasure by Steven Watanabe
Accepted into boost, but not yet released.

Documentation:
http://steven_watanabe.users.sourceforge.net/type_erasure/libs/type_erasure/doc/html/index.html

Code:
http://svn.boost.org/svn/boost/sandbox/type_erasure

Prezi

boost to the rescue

```
template <class CLASS>
struct one_who_gives_orders {
    static void apply(CLASS& c) { c.give_order(); }
};


using leader = bte::any< mpl::vector<
                                    bte::copy_constructible<>,
                                    one_who_gives_orders<bte::_self> >>;

void emergency_promote(leader lead) {
    bte::call(one_who_gives_orders<bte::_self>(), lead);    //lead.give_order();
}
```

Prezi

```
template <class CLASS>
struct one_who_gives_orders {
    static void apply(CLASS& c) { c.give_order(); }
};

namespace boost { namespace type_erasure {
    template<class CLASS, class Base>
        struct concept_interface< ::one_who_gives_orders<CLASS>, Base, CLASS> : Base {
            void give_order() { call(::one_who_gives_orders<CLASS>(), *this); }
        };
}}

using leader = bte::any< mpl::vector<
                                    bte::copy_constructible<>,
                                    one_who_gives_orders<bte::_self> >>;

void emergency_promote(leader lead) {
    lead.give_order();
}
```

Prezi

boost::type_erasure

building block for any type erased type

Handles

- references
- concept composition
- concept mapping
- (construction)
- (functions with multple arguments)
- (overloading)
- binary compatibility ?

Prezi

```
// te_stack.cpp
using any_stack_ref = bte::any<stack<bte::_self, int>, bte::_self&>;
using any_stack = bte::any< mpl::vector<
                    bte::copy_constructible<>,
                    stack<bte::_self, int>
                  >, bte::_self>;

//------------------------------------------------------------------------

any_stack create_stack() {
  return std::list<int>();
}

void fill(any_stack_ref stack) {
  stack.push_back(10);
  stack.push_back(6);
  stack.push_back(2);
}

void show(any_stack_ref stack) {
  while (!stack.empty()) {
    std::cout << stack.back();
    stack.pop_back();
    if (!stack.empty()) {std::cout << ", "; }
  }
}

//------------------------------------------------------------------------

int main() {
  any_stack stack = create_stack();

  fill(stack);
  show(stack);
}
```

Prezi

boost::type_erasure

building block for any type erased type

Handles

- references
- concept composition
- concept mapping
- (construction)
- (functions with multple arguments)
- (overloading)
- binary compatibility ?

Prezi

```
// te_stack.hpp

template<class C, class T>
struct push_back {
  static void apply(C& cont, const T& arg) { cont.push_back(arg); }
};

namespace boost { namespace type_erasure {
  template<class C, class T, class Base>
  struct concept_interface< ::push_back<C, T>, Base, C> : Base {
    void push_back(typename rebind_any<Base, const T&>::type arg) {
      call(::push_back<C, T>(), *this, arg);
    }
  };
} }

// ...
// BOOST_TYPE_ERASURE_MEMBER(back ...)
// BOOST_TYPE_ERASURE_MEMBER(empty ...)
// BOOST_TYPE_ERASURE_MEMBER(pop_back ...)

template<class S = bte::_self, class T = int>
struct stack : mpl::vector<
  push_back<S, T>,
  back<S, T>,
  pop_back<S>,
  empty<S>
> {};
```

Prezi

Having and eating our cake

Best of both worlds*

- Unintrusive
- Separate compilation
- Run-time selection
- Compile-time concept checks
- Composable
- Value based

Prezi

```
template <class STACK>
void show_impl(STACK stack) {
  while (!stack.empty()) {
    std::cout << stack.back();
    stack.pop_back();
    if (!stack.empty()) {std::cout << ", "; }
  }
}

void show(const std::vector<int>& stack) { std::cout << "std::vec "; show_impl(stack); }
void show(any_stack_ref stack) { std::cout << "generic "; show_impl(stack); }

std::list<int> list;
  fill(list);
  show(list);

std::vector<int> vec;
  fill(vec);
  show(vec);
```

Prezi

Watch "Value Semantics"

by Sean Parent to see the full power
this technique enables ...

see here ... youtube.com/watch?v=_BpMYeUFXv8

Other contenders ...

Adobe::poly	http://stlab.adobe.com/group__poly__related.html
DynamicAny	http://accu.org/index.php/journals/1502

Prezi

*Limitations

- involved concept definition syntax
- complex semantics with multiple parameters
- limitied flexibility in "concept - deduction"
- cannot adapt free functions?

Prezi

```
struct Base {};
struct Derived : Base {};
using any = bte::any<mpl::vector<bte::copy_constructible<>, bte::typeid_<>>>;
Base* base_ptr = new Derived();
any ap = base_ptr;
Derived* derived_ptr = bte::any_cast<Derived*>(ap);
```

```
any_stack stack = create_stack();
```

Prezi

```
any_stack stack = create_stack();
any_push_back_ref pb = stack;

// feature doesn't exist!
//any_stack_ref sr = dynamic_concept_cast<???>(pb);
```

Prezi

Thank You!

f.fracassi@gmx.net

Prezi