



Type Erasure



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

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

Motivation

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

void engage();

void boldly_go(captain* cpt) (

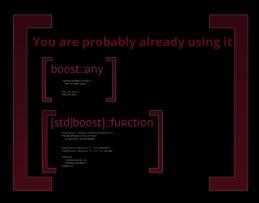
"The process of turning a wide variety of types with a common interface into one type with that same interface"



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



First contact



Type Erasure

Star Wars



First contact



Type Erasure

Real Lif

Star Wars

```
A new Hope ...

even to trans, livid |
| void at stack benefic |
| void ben
```

```
... but the empire strikes back

umplaceCONCEPT- uras my !

Unimplementable, without a language feature
li

mayone, who, gives, orders leader;
```

```
Return of the Jedi

boot:type.crasure by Seven Watanabe
Accepted into boot, but not yet released.

Documentation:
http://seven.watanabe.sacri.courreforce.nst/type.erasure/libs/
type.crasure/doc/htmi/index.html

Code
http://www.bcast.org/en/boot/sondbox/type.crasure
```

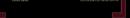
```
boost to the rescue

trapiae class (LASS-
stret ene. who. pire. andro.]

stret ene. who. pire. andro.]

traing leader = locatope replacetime
because in the stretch of the
```





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

Real Life

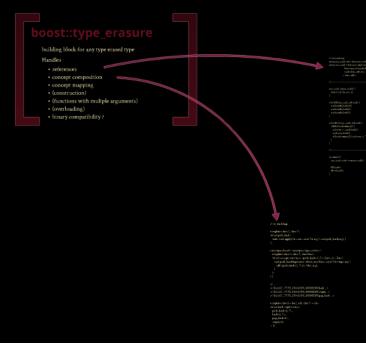
Star Wars

eader select_someone_suitable(); mergency_promote(select_someone_suitable());

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

http://svn.boost.org/svn/boost/sandbox/type.enssure

Run-time selection · Compile-time concept checks Composable Value based



Type Erasure



"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



Ah, so its polymorphism!

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



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();



Thank You!

f.fracassi@gmx.net



Ah, so its polymorphism!

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



Wait, there are problems with this approach?

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



Ok, but generic Programming solves those!

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



Problems succesfully replaced with others

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



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



Type Erasure



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);</pre>
```



DOOST:.a Particular of the second of the sec

```
void black_box(boost::any param) {
    boost::any copy = param;
}
boost::any value = 5;
black_box(value);
```



```
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;
};
```



[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);</pre>
```



```
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); }
   private:
     ValueType held;
  placeholder * content;
```



```
struct any_fun {
 ~any_fun() { delete content; }
 any_fun(const any_fun & other): content(other.content? other.conter
 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;
```

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

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

template<typename ValueType>
struct holder : placeholder {

```
tempiate<typename value i ype>
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;
```

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);



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());
```



A new Hope ...

```
struct young_kirk {
                           struct one_who_gives_orders {
  void misbehave();
                              void give_order();
  void give_order();
                           using leader = any<one_who_gives_orders>;
young_kirk james;
                           void emergency_promote(leader lead) {
james.misbehave();
                              lead.give_order();
board_enterprise(james);
             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() {
won't compile
                   return young_spock();
```



... but the empire strikes back

```
template<CONCEPT> struct any {
     Unimplementable , without a language feature
};
any<one_who_gives_orders> leader;
```



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



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<</pre>
                                       bte::copy_constructible<>,
                                       one_who_gives_orders<br/><br/>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<</pre>
                                      bte::copy_constructible<>,
                                      one_who_gives_orders<br/><br/>bte::_self> >>;
void emergency_promote(leader lead) {
    lead.give_order();
```



boost::type_erasure

building block for any type erased type

Handles

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



```
// 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<br/>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();</pre>
   stack.pop_back();
   if (!stack.empty()) {std::cout << ", "; }</pre>
//-----
int main() {
 any_stack stack = create_stack();
 fill(stack);
 show(stack);
```



boost::type_erasure

building block for any type erased type

Handles

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



```
// 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>
> {};
```



Having and eating our cake

Best of both worlds*

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



```
template <class STACK>
void show_impl(STACK stack) {
 while (!stack.empty()) {
   std::cout << stack.back();</pre>
   stack.pop_back();
   if (!stack.empty()) {std::cout << ", "; }</pre>
void show(const std::vector<int>& stack) { std::cout << "std::vec "; show_impl(stack); }</pre>
void show(any_stack_ref stack) { std::cout << "generic "; show_impl(stack); }</pre>
std::list<int> list;
 fill(list);
 show(list);
std::vector<int> vec;
 fill(vec);
 show(vec);
```



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



*Limitations

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



```
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();
any_push_back_ref pb = stack;

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



Thank You!

f.fracassi@gmx.net

