Polymorphic Value Types for C++

Stuart Davis October 2021

Goals for software development

“Write all code as if it were a library you intend to submit for standardization.”
- Sean Parent

- Reliable
- Testable
- Maintainable
- Comprehensible
- Modifiable
- Extensible
- Scalable

Polymorphism

“the provision of a single interface to entities of different types”

- Compile-time **polymorphism**
- Runtime **polymorphism**
- C++ inheritance
- C++ templates
- C++ concepts
- Type erasure

Compile-time Polymorphism

- Fast (execution)
- Rigid (static)
- Bloated
- Opaque

C++ function and operator overloading

```
int64_t add(int64_t x, int64_t y);
double add(double x, double y);
template <typename T> T add(T x, T y);
template <typename X, typename Y, typename Z> Z add(X x, Y y);
```

Runtime Polymorphism

- Slower (execution)
- Flexible (dynamic)
- Compact
- Transparent

C++ function overriding

```
class number
{
    virtual number & add(number const& other) = 0;
};
```

C++ Inheritance

“Choose your parents wisely.” - Bertrand Russell

Pros

- Familiar and predictable
- Helpful compiler error messages
- Allows separation of interface and implementation

Cons

- Reference semantics
- Unreasonable shared / global data
- Allows conflation of interface and implementation

C++ Templates

“When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.” - James Whitcomb Riley

Pros

- Duck typing
- **Value semantics**
- Powerful metaprogramming

Cons

- Can be unintuitive
- Unhelpful compiler error messages
- Encourages conflation of interface and implementation

C++ Concepts

"named boolean predicates on template parameters, evaluated at compile time"

- New feature for C++20
- Friendlier alternative to **SFINAE** (substitution failure is not an error)
- Introduces type-checking to template programming
- Simplifies compiler diagnostics for failed template instantiations
- Allows selection of function template overloads and class template specializations based on type properties
- Helps to distinguish between interface and implementation
- e.g. **`std::copy_constructible`**

Type Erasure

“enables you to use various concrete types through a single generic interface”

Can mean different things:

- `void` pointers
- C++ inheritance
- C++ templates
- `std::any`
- `std::function`
- “Classic” type erasure implementations for C++
- Usually some form of inheritance hidden behind templates...

“Classic” Type Erasure for C++ - Interface

```
struct widget // the interface name
{
    struct base // internal boilerplate
    {
        virtual ~base() {};
        virtual void display() const = 0;
    };

    template <typename T>
    struct derived : base // more internal boilerplate
    {
        T wrapped; // this will be instantiated with an implementation

        void display() const override { wrapped.display(); } // implementation requires
    };

    std::shared_ptr<base> handle; // this will look after memory allocation for any data

    void display() const { handle->display(); } // this defines the external interface
};
```

“Classic” Type Erasure for C++ - Implementation

```
void display_widget(widget w) // example of a free function that takes a widget by value
{
    w.display();
}

struct button // this is a widget implementation and yet it is semi-independent
{
    void display() const
    {
        std::cout << "Don’t push me!" << std::endl;
    }
};

int main() // this is real code that compiles and executes!
{
    widget w1;
    w1.handle = std::make_shared<widget::derived<button>>(); // construct & instantiate T
    display_widget(w1);                                      // prints "Don’t push me!"
    return 0;
}
```

Benefits of “Classic” Type Erasure for C++

- Duck typing checked at compile time (like templates)
- Could easily be combined with C++ concepts
- Wrapper itself can be thought of as a kind of concept definition
- Enforces separation of interface and implementation
- Allows loose coupling between different types
- Allows mixins without resorting to multiple-inheritance
- Lots of options to include different behavior and features in the wrapper
- Wrapper can be designed to exhibit **value semantics**

Drawbacks of “Classic” Type Erasure for C++

- Not a native language feature
- Not part of the standard library
- Lots of boilerplate
- Can be unintuitive, unpredictable and unhelpful (like templates)
- Intention may not be transparent to compilers, debuggers and programmers
- Usually involves heap allocation
- Can end up with multiple layers of nested wrappers (without even realising)
- Performance?

Value Types

“the representation of some entity that can be manipulated by a program”

- Primitives e.g. **`int64_t`**
- **`std::string`**
- **STL** e.g. **`std::vector`** etc.
- Intuitive semantics (construction, copying, assignment, comparison, scope...)
- Easy ownership / lifetime management
- Local reasoning about code
- Lock-free thread-safety (**AKA** copying)
- Can utilise copy-on-write (**COW**)

Reference Types

“a value of reference type is a reference to another value”

- Raw pointers
- C++ references
- `std::shared_ptr` etc.
- Redefined semantics (construction, copying, assignment, comparison...)
- Harder ownership / lifetime management
- Non-local / global reasoning about code
- Thread-safety through locking etc.
- Sometimes... shared state might actually simplify a design! (hmmmm…)

C++ Approaches to Polymorphic Value Types

“Now that you understand the basic procedure that I am circumlocuting…”
- Frank Zappa

- Boost.TypeErasure
- Sean Parent
- Zach Laine
- Sy Brand
- Strange

Boost.TypeErasure

- Probably the best type erasure library for C++
- Reduces the amount of per-type boilerplate
- Replaces it with Boost macros and template-metaprogramming techniques
- Uses explicit v-table construction
- Stores a copy of the v-table with every object
- Produces long compile times and scary compiler error messages
- Stores everything on the heap
- Integrates with other Boost libraries

https://www.boost.org/doc/libs/1_77_0/doc/html/boost_typeerasure.html

Sean Parent

“Inheritance is the base class of evil.”

Complete list of talks (including some given at **Bloomberg**):
https://sean-parent.stlab.cc/papers-and-presentations/

“Better Code: Runtime Polymorphism” talk:
https://www.youtube.com/watch?v=QGcVXgEVMJg

- As we increasingly move to heavily threaded systems using futures, reactive programming, and task queues, **value semantics** becomes critical to avoid locking and to reason about code.
- It is my hope that the language (and libraries) will evolve to make creating **polymorphic** types with **value semantics** easier.

Upcoming book: “Better Code: Goals for Software Developers” Paperback – 28 Oct. 2021

Zach Laine

“The error messages will make your eyes bleed.”

Emtypen code generator: https://github.com/tzlaine/type_erasure

- Uses libclang to analyse a simple C++ class interface or ‘archetype’
- Generates C++ code for sophisticated type-erased wrapper classes
- Optional copy-on-write (**COW**)
- Optional small-buffer-optimisation (**SBO**)
- Generated code uses a lot of templates and **SFINAE**
- Supports ‘archetype’ class templates, but not member function templates
- Focused on runtime performance

“Pragmatic Type Erasure” talk: https://www.youtube.com/watch?v=0I0FD3N5cgM

Sy Brand

“Dynamic Polymorphism with Metaclasses and Code Injection” talk:
https://www.youtube.com/watch?v=8c6BAQcYF_E

- Type erasure implemented using manual v-tables and lambdas
- Minimizes boilerplate using static reflection and code injection
- Uses experimental language features that are not part of standard C++ (yet)
- Existing proposals for these features seem unlikely to be accepted soon
- e.g. “Scalable Reflection in C++” - **`reflexpr`**:
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1240r1.pdf
- Recent proposals potentially derailed by the “Circle” **`@meta`** language:
  https://www.circle-lang.org/

Strange

"An 'I' is a strange loop where the brain's symbolic and physical levels feed back into each other and flip causality upside down so that symbols seem to have gained the paradoxical ability to push particles around, rather than the reverse."
- Douglas Hofstadter "I Am A Strange Loop"

- C++17 framework based around regular function pointers
- Much of the boilerplate abstracted out into wrapper class templates
- Supports **constant** and **variable value** types: `con<>` `var<>`
- Also includes **shared** and **weak reference** types: `ptr<>` `adr<>`
- "abstraction" - an interface supporting inheritance and template parameters
- "thing" - an implementation for one or more abstractions

Code (**WIP**): https://github.com/oneish/strange

Strange Abstractions

```
namespace strange
{
//  struct any_o
//  {
//      void (*_free) (con<> const& me);
//      void (*_copy) (con<> const& me, var<> const& copy);
//  };

    struct widget_o : any_o
    {
        void (*display) (con<widget_a> const& me); // only appears once in the interface
    };

    struct widget_a
    {
        using operations = widget_o;
        mutable thing_t* t;             // constness is delegated to the wrappers
        mutable operations const* o;    // v-tables are statically allocated, not copied
    };
}
```

Strange Things

```
namespace strange
{
    struct button_t : thing_t // common base class helps with reference counting etc.
    {
        std::string message_ = "Don't push me!";        // data
        static widget_o const* _operations();           // v-table
        static void display(con<widget_a> const& me);   // extraction - const operation
    };

    widget_o const* button_t::_operations() // v-table
    {
        static widget_o operations = { { _free, _copy }, display };
        return & operations;
    }

    void button_t::display(con<widget_a> const& me) // implementation
    {
        std::cout << static_cast<button_t const* const>(me.t)->message_ << std::endl;
    }
}
```

Strange Values

```
namespace strange
{
    var<widget_a> create_button() // factory function
    {
        return var<widget_a> { new button_t, button_t::_operations() }; // construction
    }
}

using namespace strange;

int main()
{
    con<widget_a> constant = create_button();                 // button_t::create()
    constant.perform(&widget_a::operations::display);   // constant.o->display(constant);
    var<widget_a> variable{ constant };                       // thread-safe copy-on-write
    auto pointer = ptr<widget_a>{ variable };                 // similar to std::shared_ptr
    auto address = pointer.address();                         // similar to std::weak_ptr

    return 0;
}
```

Strange Possibilities

- Implement “things” for collections, iterators and ranges
- Tokenizer, parser and C++ code generator for “strange” language
- Syntax similar to **LISP** with **JSON**
- Step down into native C++
- Step up into macros and metacode
- Reflection
- Serialization
- Code injection
- Compile-time execution via runtime plugins
- Concurrent evaluation of directed acyclic graphs...