WIKIBOOKS

More C++ Idioms/Type Erasure

Contents

Type Erasure

Intent

To provide a type-neutral container that interfaces a variety of concrete types.

Also Known As

"Variant"[*citation needed*] (not to be confused with `std::variant`). This technique is used inside `std::any` and `std::function`.

Motivation

It is often useful to have a variable which can contain more than one type. Type Erasure is a technique to represent a variety of concrete types through a single generic interface.

Implementation and Example

Type Erasure is achieved in C++ by encapsulating a concrete implementation in a generic wrapper and providing virtual accessor methods to the concrete implementation via a generic interface.

The key components in this example interface are var, inner_base and inner classes:

```
struct var{
   struct inner_base{
      using ptr = std::unique_ptr<inner_base>;
   };
   template <typename _Ty> struct inner : inner_base{};
private:
   typename inner_base::ptr _inner;
};
```

The var class holds a pointer to the inner_base class. Concrete implementations on inner (such as inner<int> or inner<std::string>) inherit from inner_base. The var representation will access the concrete implementations through the generic inner_base interface. To hold arbitrary types of data a little more scaffolding is needed:

```
struct var{
   template <typename _Ty> var(_Ty src) : _inner(new inner<_Ty>
(std::forward<_Ty>(src))) {} //construct an internal concrete type accessible
through inner_base
   struct inner_base{
      using ptr = std::unique_ptr<inner_base>;
   };
   template <typename _Ty> struct inner : inner_base{
      inner(_Ty newval) : _value(newval) {}
   private:
      _Ty _value;
   };
private:
   typename inner_base::ptr _inner;
};
```

The utility of an erased type is to assign multiple typed values to it so an assignment operator achieves just that:

```
struct var{
   template <typename _Ty> var& operator = (_Ty src) {
      _inner = std::make_unique<inner<_Ty>>(std::forward<_Ty>(src));
      return *this;
   }
   struct inner_base{
      using ptr = std::unique_ptr<inner_base>;
   };
   template <typename _Ty> struct inner : inner_base{
      inner(_Ty newval) : _value(newval) {}
   private:
      _Ty _value;
   };
private:
```

```
    typename inner_base::ptr _inner;
};
```

Creating an erased type and assigning it various values isn't of much use unless you can interrogate it. One useful method is to query for the underlying type info:

```
struct var{
    const std::type_info& Type() const { return _inner->Type(); }
    struct inner_base{
        using ptr = std::unique_ptr<inner_base>;
        virtual const std::type_info& Type() const = 0;
    };
    template <typename _Ty> struct inner : inner_base{
        virtual const std::type_info& Type() const override { return
typeid(_Ty); }
    };
private:
    typename inner_base::ptr _inner;
};
```

Here the var class forwards calls of Type() to it's inner_base interface which is overridden by the concrete inner<_Ty> subclass which ultimately returns the underlying type. This technique of forwarding accessor methods to a virtual interface which is overridden by concrete implementations is expanded for a fully useful generic type.

Complete Implementation

```
struct var {
    var() : _inner(new inner<int>(0)){} //default construct to an integer

    var(const var& src) : _inner(src._inner->clone()) {} //copy constructor
calls clone method of concrete type

    template <typename _Ty> var(_Ty src) : _inner(new inner<_Ty>
(std::forward<_Ty>(src))) {}

    template <typename _Ty> var& operator = (_Ty src) { //assign to a concrete
type
        _inner = std::make_unique<inner<_Ty>>(std::forward<_Ty>(src));
        return *this;
    }

    var& operator=(const var& src) { //assign to another var type
        var oTmp(src);
        std::swap(oTmp._inner, this->_inner);
        return *this;
    }

    //interrogate the underlying type through the inner_base interface
```

```cpp
    const std::type_info& Type() const { return _inner->Type(); }
    bool IsPOD() const { return _inner->IsPOD(); }
    size_t Size() const { return _inner->Size(); }

    //cast the underlying type at run-time
    template <typename _Ty> _Ty& cast() {
        return *dynamic_cast<inner<_Ty>&>(*_inner);
    }

    template <typename _Ty> const _Ty& cast() const {
        return *dynamic_cast<inner<_Ty>&>(*_inner);
    }

    struct inner_base {
        using Pointer = std::unique_ptr < inner_base > ;
        virtual ~inner_base() {}
        virtual inner_base * clone() const = 0;
        virtual const std::type_info& Type() const = 0;
        virtual bool IsPOD() const = 0;
        virtual size_t Size() const = 0;
    };

    template <typename _Ty> struct inner : inner_base {
        inner(_Ty newval) : _value(std::move(newval)) {}
        virtual inner_base * clone() const override { return new inner(_value);
}
        virtual const std::type_info& Type() const override { return
typeid(_Ty); }
        _Ty & operator * () { return _value; }
        const _Ty & operator * () const { return _value; }
        virtual bool IsPOD() const { return std::is_pod<_Ty>::value; }
        virtual size_t Size() const { return sizeof(_Ty); }
    private:
        _Ty _value;
    };

    inner_base::Pointer _inner;
};

//this is a specialization of an erased std::wstring
template <>
struct var::inner<std::wstring> : var::inner_base{
    inner(std::wstring newval) : _value(std::move(newval)) {}
    virtual inner_base * clone() const override { return new inner(_value); }
    virtual const std::type_info& Type() const override { return
typeid(std::wstring); }
    std::wstring & operator * () { return _value; }
    const std::wstring & operator * () const { return _value; }
    virtual bool IsPOD() const { return false; }
    virtual size_t Size() const { return _value.size(); }
    private:
```

```
    std::wstring _value;
};
```

Example Implementation from Sean Parent talk

```
template<typename T>
void draw(const T& x, std::ostream& out, size_t position) {
    out << std::string(position, ' ') << x << std::endl;
}
class object_t {
public:
    template<typename T>
    object_t(T x) : self_(std::make_shared<model<T>>(std::move(x))) {}
    friend void draw(const object_t& x, std::ostream& out, size_t position) {
        x.self_->draw_(out, position);
    }
private:
    struct concept_t {
        virtual ~concept_t() = default;
        virtual void draw_(std::ostream&, size_t) const = 0;
    };
    template<typename T>
    struct model final : concept_t {
        model(T x) : data_(std::move(x)) {}
        void draw_(std::ostream& out, size_t position) const override {
            draw(data_, out, position);
        }
        T data_;
    };
    std::shared_ptr<const concept_t>self_;
};
```