Value-Based Programming with Boost

Jeff Garland & Kevlin Henney

Workshop Goals

1. Discuss the state of the practice
2. Raise awareness of value types in programming
3. Produce a value type library roadmap
   - What libraries and features are needed?
4. Familiarize people with some often overlooked Boost libraries

Session Agenda

1. Intro to Value-Based Programming (Kevlin)
2. Value Type Libraries (Jeff)
3. Workshop - Discussion of Example Value Types
4. Workshop - Discussion / Documentation of Results

Section Goals

- Introduce and clarify what we mean by *values* and *value types*
- Present some common properties of values
- Discuss the design of value types in C++

What Do We Mean by *Value*?

- The term *value* is used in subtly different ways in different contexts
  - E.g. "pass by value" refers to an argument-passing mechanism more accurately termed "pass by copy"
  - E.g. a C++ object that is normally copyable and bound to a scope rather than heap allocated
  - E.g. a kind of stateful object for which identity does not play a dominant role
  - E.g. "the precise number or amount represented by a figure, quantity, etc." (OED)

Complementary Perspectives

- There are three perspectives of interest and value (sic) to us...
  - *The Platonic Perspective* presents an idealized view of what values are in terms of models of the real world
  - *The Object Perspective* is a model-based view of what values are in terms of programming concepts — namely objects — but is not necessarily bound to a single language's view
  - *The C++ Perspective* allows us to realize and use value objects in C++ code

The Platonic Perspective

- Michael Jackson (*Problem Frames*) identifies values as one of six phenomena to model in problem domains
  - Individuals: events, entities and values
  - Relations: states, truths and roles
- Values are considered to be immutable, intangible and immortal individuals
  - The value or content of a value is synonymous with its identity
  - Relations are tuples that may involve values, but some such tuples may also be considered values

The Object Perspective

- From a programming perspective, we can model values as objects
  - Hence value objects and value types
- Value objects are objects with significant state and insignificant identity
  - Note that distinguishing between *values* and *objects* is not normally useful or meaningful
- Historically values have been present in many patterns and practices
  - But value-based programming has not really been acknowledged as a significant topic

The C++ Perspective

- C++ has many features that make value-based programming simple and attractive
  - Operator overloading allows direct expression of many value operations in a convenient form
  - Templates support further generalization of such expressiveness and allows (compile-time) polymorphic use of types with common interface
  - Inlining allows expression of value types without an associated abstraction penalty
  - Value semantics as opposed to reference semantics as the default type model

Core Anatomy of a C++ Value

- C++ value types have the following common characteristics...
  - They are copy constructible and copy assignable
  - They are default constructible
  - The equality (and inequality) comparable
  - They encapsulate their memory management
- In use, C++ value objects are normally...
  - Part of a surround scope (block or object) rather than managed with *new* and *delete*
  - Passed around by copy or by *const* reference
  - Modified through non-*const* references

More Anatomy of a C++ Value

- Value objects are not normally handled via pointer, so operator overloading is practical
  - Of course, which operators are supported is a matter of convention and taste :-)
- Similarly, value types do not normally participate in subclassing hierarchies
  - They may participate in subtyping hierarchies, whether in terms of interface class derivation or concept refinement
- Other conventions guide value type design
  - E.g. moveability, swappability

Generality of Values

- Value types differ in the generality and focus of their domain
  - Some are mathematical in nature, e.g. integers
  - Some are programming focused, e.g. strings
  - Some are real-world focused, e.g. ISBNs
- Value types reflect the constraints of the problem and the solution domains
  - E.g. ISBNs have rules of well formedness that govern legal instances
  - E.g. *int* is a bounded subset of all integers

Systems of Values

- Many value types form systems through their operations and constraints
  - E.g. a point in time is a value, as is the difference between two points in time, but a time point and a time interval are not the same conceptual type
  - E.g. distance divided by time yields speed (and displacement divided by time yields velocity)
- It is possible to express conceptual types as proper types or as degenerate types
  - E.g. an ISBN can be expressed explicitly as a class wrapping a string or directly as a string, in which case the conceptual type is informal

Session Agenda

1. Intro to Value-Based Programming (Kevlin)
2. Value Type Libraries (Jeff)
3. Workshop - Discussion of Example Value Types
4. Workshop - Discussion / Documentation of Results

Section Goals

- Survey of value types in Boost (and other) libraries
  - Places to look for inspiration
- Value Type Design Examples
- Current Libraries for Building Value Types

Building Value Types – Preliminary Considerations

- It's difficult do well
  - Domain is often more complex than it appears
  - There's lots of rules to remember for tight class design
  - Hard to keep minimalist interfaces
  - Underlying representation often inherently limits implementation
  - Takes a broad understanding of C++ idioms
  - Often want ‘near-zero’ runtime overhead

Survey of Some Value Types

- C++ Standard
  - std::complex, std::string?
- Boost
  - date-time (a sampling)
    - date, days, weeks, months, years, date_period
    - ptime, hours, minutes, seconds, time_duration
  - rational
  - quaternions, octonion
  - tribool
  - interval (sort of)
  - filesystem::path

What's the Purpose of these Value Types?

- Provide 'domain-specific' capabilities
  - date type: d += weeks(2)
  - Path type: p += p / "file.out";
- Some provide 'different numeric representations'
  - Eg: complex, rational
  - Negative variation from normal numbers
    - Restrict range
    - Layer in typing to restrict calculations
  - Positive variation
    - Allow numeric representations not possible with base types

Taxonomy of Value Types - Primitive

- Basic Numeric Valuetypes
  - Angle (0-360 degrees?)
  - day of month (1-31), day of year (1-366)
- String-Based Value Types
  - 'US States'
  - Zipcodes
- Combo Numeric/Enum Value Types
  - Color (black = #000000, etc)
  - Month (January=1, February=2, etc)

Taxonomy of Value Types - Composites

- Composition of Primitives
  - date
    - Composed of 'year-month-day'
  - LatLong
    - Composed of 'Latitude and Longitude'

Single Numeric Value Types

- Typically a 'unit' of some type
- Usual comparison ops – totally ordered
- May have specialized mathematical rules
  - restricted to other specialized types
  - Throw exception if range exceeded
- Restricted value range common
  - example: day_of_the_week: 0-6

String Based Value Types

- Represented by a fixed set of strings
  - US states - ‘Enumeration’ of 50 string values
- Each value may have multiple strings
  - Long: Arizona
  - Abbreviation: Az
- No ‘math’ interfaces
  - Az + Tx = ?

String/Enum Based Valuetypes

- Typical case is a fixed enumeration of 'values'
  - eg: enum Color { Black, White Red, Green, Blue };
- Ordering is often arbitrary – equality is main test
  - is Red > Green?
- Often multiple string representations for the same value eg: Jan == January
- Strings may need to be localized for i/o
- Sometimes there is numeric association
  - Eg: Black = #000000 in CSS
- Math doesn't necessarily make sense

Composition Numeric Valuetypes

- Usual comparison ops – totally ordered
- Also has specialized math rules
- Each 'composition element' may be single numeric or combo value type
- i/o may have complex ordering rules

Libraries for Valuetype Construction

- Boost Operators
- constrained_value (in date-time)
- MCS Units (recently accepted)
- Smart enum (not reviewed)
- Boost enum (not reviewed)

MCS Units

- New Boost library
  - Helpful in ‘mathematical domains’
- Terminology
  - Fundamental dimension - type of measurement that forms the basis of a unit system (eg: length)
  - Dimension - signature of a collection of fundamental dimensions, each potentially raised to a different rational power (eg: length^2 = area)
  - Units - Units are a specific measure of a dimension (eg: meter)
  - Quantity – Concrete amount of some unit (eg: 3 meters)

MCS Units Example

- 1 circle = 360 degrees = 2 pi radians = 400 gradians
  - Each has it's own uses
  - Numeric type
- Others?
  - ArcSeconds, ArcMinutes
- Taking sine of an angle
  - What units?
  - `Double std::sin(double)`

MCS Units – Angle Example

```
quantity<angle:degree> ang(180.0)*degrees;
auto sin_ang(std::sin(ang));

Todo:
What's happened:
  ang is converted to double for sin
discussion of 'auto'
Discussion of actual type (it's ugly)
```

Limits of MCS Units

- Minimal input-output
  - Basically only ‘output’
- Intended for ‘numeric types’
  - Won’t do anything for ‘string/combined types’
- Unique ‘units type’ require quite a bit of implementation

constrained_value

- Part of date-time library
  - http://www.boost.org/doc/html/constrained_value.html
- Provides policy-based range wrapper for integer types
- Others have extended idea
  - Christopher Diggins in DDJ
  - Discussed in O'Reilly C++ Cookbook
  - http://www.artima.com/weblogs/viewpost.jsp?thread=79470

Configuring constrained_value

template<class value_policies>

class constrained_value

- value_policies:
  - A value_type that holds the value (eg: int)
  - Provides the range limits via the min and max functions
  - an on_error function that determines how errors are handled.
    - typically log, assert, or throw an exception.

constrained_value example

```
#include <iostream>
#include <stdexcept>
#include <boost/lexical_cast.hpp>
#include <boost/date_time/constrained_value.hpp>

class one_to_ten_out_of_range : public std::range_error
{
public:
  one_to_ten_out_of_range(unsigned short value) :
    std::range_error("one_to_ten out of range: "
                     + boost::lexical_cast<std::string>(value))
  {}
};
```

constrained_value continued

```
struct one_to_ten_policies
{
public:
  typedef unsigned short value_type;
  static unsigned short min() { return 1; };
  static unsigned short max() { return 10;};
  static void on_error(unsigned short&,
                       unsigned short value,
                       boost::CV::violation_enum)
  {
    throw one_to_ten_out_of_range(value);
  }
};
```

constrained_value continued

```
typedef boost::CV::constrained_value<one_to_ten_policies>
    one_to_ten;
int main()
{
  one_to_ten v1(1);
  std::cout << v1 << std::endl;
  try {
    one_to_ten v1(11);
  }
  catch(std::exception& e) {
    std::cout << e.what() << std::endl;
  }
  return 0;
}
```

constrained_value output

1

one_to_ten out of range: 11

Boost Operators

- http://www.boost.org/libs/utility/operators.htm
- Simplifies creation of valuetypes by minimizing code required to implement operators
  - Builds on concepts
    - addable, subtractable, dividable, multipliable
    - incrementable, decrementable
    - less_than_comparable, equality_comparable
    - etc.

Issues with Using Operator Lib

- Documentation of value type is harder
  - The class no longer has obvious methods
  - Even though 'addable' documents that += and + work you'll need to write docs for each method
- Object size
  - Can bloat on old compilers
  - not an issue on gcc 4

Boost Enum

- Provides a fancy enum type
  - Relates strings and enum values
  - Or bitfields to strings
  - Creates an index as well
  - Various bit operations
- Problem
  - No docs
  - Development isn't too active…
- **Download: http://tinyurl.com/2hh7po**

Boost Enum: Writing an Enum

```
struct Log
{
    BOOST_ENUM_VALUES(Level, const char*,
      (Abort)("unrecoverable problem")
      (Error)("recoverable problem")
      (Alert)("unexpected behavior")
      (Info) ("expected behavior")
      (Trace)("normal flow of execution")
      (Debug)("detailed object state listings")
    )
};
```

Boost Enum Functions

- Embedded domain enum always contains the enum
- Names as strings – look up by index
- Values as strings – look up by index
- Indexes – look up by string

Boost Enum - Expanded

```
class Level : public boost::detail::enum_base<Level, string>
{
   public:
    enum domain
    {
        Abort, Error, Alert, Info, Trace, Debug,
    };

    BOOST_STATIC_CONSTANT(index_type, size = 6);
```

Boost Enum - Expanded

```
Level() {}
Level(domain index) : boost::detail::enum_base<Level, string>(index) {}

typedef boost::optional<Level> optional;

static optional get_by_name(const char* str)
{
    if(strcmp(str, "Abort") == 0) return optional(Abort);
    if(strcmp(str, "Error") == 0) return optional(Error);
    if(strcmp(str, "Alert") == 0) return optional(Alert);
    if(strcmp(str, "Info") == 0) return optional(Info);
    if(strcmp(str, "Trace") == 0) return optional(Trace);
    if(strcmp(str, "Debug") == 0) return optional(Debug);
    return optional();
}
```

Boost Enum - Expanded

```
private:
  friend class boost::detail::enum_base<Level, string>;

 static const char* names(domain index)
  {
        switch(index)
        {
        case Abort: return "Abort";
        case Error: return "Error";
        case Alert: return "Alert";
        case Info: return "Info";
        case Trace: return "Trace";
        case Debug: return "Debug";
        default: return NULL;
        }
  }
```

Boost Enum - Expanded

```
typedef boost::optional<value_type> optional_value;
static optional_value values(domain index)
{
    switch(index)
    {
    case Abort: return optional_value("unrecoverable problem");
    case Error: return optional_value("recoverable problem");
    case Alert: return optional_value("unexpected behavior");
    case Info: return optional_value("expected behavior");
    case Trace: return optional_value("normal flow of execution");
    case Debug: return optional_value("detailed object state listings");
    default: return optional_value();
    }
}
```

Boost Smart Enum

- Another enum library
  - http://cryp.to/smart-enum/libs/smart_enum/doc/smart_enum.htm
- Provides strong type safety
- Increment/Decrement

```
//this won't work today
enum Junk { j1, j2, j3 };

Junk j(j1);
j++; //error!
```

Summary of Current Libraries

- It's a mess…
- Mish-mash of libraries
  - Half not accepted into Boost (or buried)
  - Don't work together
- Lots of holes
  - Not much support for writing i/o code

Session Agenda

1. Intro to Value-Based Programming (Kevlin)
2. Value Type Libraries (Jeff)
3. Workshop - Discussion of Example Value Types
4. Workshop - Discussion / Documentation of Results

Section Goals

- Workshop Problem Value Types
  - SafeInt
  - Money

Safe-Int Requirements

- Core concept: Safe-Int will signal programmer when math operations fail due to overflow
- Considerations
  - Family of Types?
    - different sizes?
    - signed and unsigned?
    - user definable range?
  - input – output
  - immutable?
  - conversions to/from built-in types?

Money Example

```
#include "boost/operators.hpp"

template<typename ValType>
class money :
  boost::equality_comparable<money<ValType>,
  boost::less_than_comparable<money<ValType>
  >  >
{
```

Money (cont)

```
{
public:
  //...
  bool operator<(const money& rhs) const
  {
    return val < rhs.val;
  }
  bool operator==(const money& rhs) const
  {
    return val == rhs.val;
  }
private:
  ValType val;
};
```

Money Type

- Core Concept: Type that holds an amount of money including information about currency
  - One type or one per currency?
  - How to represent currency?
- No round-off errors on calculations?
- Input/Output including currency units?
  - strings like 'USD'
  - symbols like '$'
  - facets and manipulators?
  - strings for units and fractional units (eg: 'dollars' and 'cents')?

Money Type (continued)

- References
  - Rogue Wave's solution:
    - http://www.roguewave.com/support/docs/hppdocs//mnyug/2-9.html
  - ISO 4217 – currency abbreviations
    - http://www.jhall.demon.co.uk/currency/
  - Java Currency
    - http://java.sun.com/j2se/1.4.2/docs/api/java/util/Currency.html
    - http://jscience.org/api/org/jscience/economics/money/package-summary.html
    - www.objectivelogic.com/Articles/Java%20and%20Monetary%20Data/Java%20and%20Monetary%20Data.pdf

Cooperating types?

```
money m (100, USD);
m += money(100, USD); //$200
m += dollars(100); //$300
m += cents(25);
percent tax_rate(5.25);
m = m*tax_rate;
```

Problems

```
money us(100, USD);
money eu(100, EUR);
if (us > eu) { //what now?
```

Session Agenda

1. Intro to Value-Based Programming (Kevlin)
2. Value Type Libraries (Jeff)
3. Workshop - Discussion of Example Value Types
4. Workshop - Discussion / Documentation of Results