Fusion by example

Joel de Guzman and Dan Marsden

Outline

- What is Fusion?
- A quick outline of Fusion
- 4 examples

What is Fusion?

- A library of heterogeneous containers and associated algorithms
- A fusion of compile time and run time programming techniques
- A tuple library

Tuples in the C++ world

- Conventional C++
  - std::pair
- C++ 0x / TR1
  - std::tr1::array
  - std::tr1::tuple
- Fusion
  - Multiple container types
  - Algorithms!

Applications of Fusion

- Originally developed to do some template “heavy lifting” in Boost.Spirit.
- Applicable to a wide range of generic library implementations – Boost.Spirit, Phoenix, Proto.
- Also suitable for use in application code

Key concepts of Fusion

- Containers
  - vector, list, map....
- Algorithms
  - iteration, querying and transformation
- Iterators
- Views

Fixed at compile time

- The number of types in a container
- The order of types in a container
- Iterator positions

Duality of runtime and compile time

- Fusion sequences are MPL sequences
- MPL sequences are Fusion sequences
- Algorithms, intrinsics etc. have both runtime and compile time forms
  - fusion::reverse(...) <=> fusion::result_of::reverse<...>

Algorithms are lazily evaluated

- Container independent algorithms
  - push_back, pop_back etc. have one implementation
- Efficient algorithm chaining
  - filter_if<pred>(transform(push_back(cont, v), f())) does not lead to excessive copying
- Permits infinite length sequences
  - as long as you don't actually need the whole thing...

A serialization example

<int>32</int>

<string>Marsden</string>

<string>Dan</string>

<double>123.45</double>

Our example data type

```
struct employee
{
    int age;
    std::string surname;
    std::string forename;
    double salary;
};
```

The conventional approach

```
std::ostream& format_employee(std::ostream& stream, employee const& e)
{
    return stream << "<int>" << e.age << "</int>"
                  << "<string>" << e.surname << "</string>"
                  << "<string>" << e.forename << "</string>"
                  << "<double>" << e.salary << "</double>";
}
```

Can we eliminate this repetition?

```
std::ostream& format_employee(std::ostream& stream, employee const& e)
{
    return stream << "<int>" << e.age << "</int>"
                  << "<string>" << e.surname << "</string>"
                  << "<string>" << e.forename << "</string>"
                  << "<double>" << e.salary << "</double>";
}
```

What is it we really intend to do?

"for each member write a formatted XML string to the output stream."

- “for each” sounds like we need to apply some sort of algorithm
- How do we produce a “container” of members to operate upon?

Better expressing our intent (1st Attempt)

```
typedef fusion::vector<int, std::string, std::string, double> employee;


std::ostream& format_employee(std::ostream& stream, employee const& e)
    {
        fusion::for_each(e, format_member(stream));
        return stream;
    }
```

The formatting object

```
struct format_member {
    explicit format_member(std::ostream& stream)
      : stream_(stream) {};

    template<typename T>
    void operator()(T const& val) const
    {
        std::string const type = typeid(val).name();
        stream_ << '<' << type << '>'  << val << "</" << type << '>';
    }
    std::ostream& stream_;
};
```

Not for employees only

```
template <typename Sequence>
std::ostream& format_xml(std::ostream& stream, Sequence const& seq)
{
    fusion::for_each(seq, format_member(stream));
    return stream;
}
```

Returning to our original employee

```
std::ostream& format_xml(std::ostream& stream, employee const& e)
{
    return format_xml(
        fusion::vector_tie(e.age, e.surname, e.forename, e.salary));
}
```

The extension alternative

```
BOOST_FUSION_ADAPT_STRUCT(
    employee,
    (int, age)
    (std::string, surname)
    (std::string, forename)
    (double, salary))
```

Summary so far...

- We can use fusion::vector as a container for a mixture of types
- We can use fusion::vector_tie to produce vectors of references in an ad-hoc manner
- We can use fusion::for_each to apply an operation to every member of a Fusion sequence

A simple bit of formatting

Surname: Marsden

Forename: Dan

Age: 32

Salary: $ 123.45

The conventional implementation

```
std::ostream& format_employee(std::ostream& s, employee
   const& e)
{
    return s      << "<b>Surname: " << e.surname << "</b>\n"
                  << "Forename: " << e.forename << "\n"
                  << "Age: " << e.years << " years old\n"
                  << "Salary: $" << e.salary << '\n';
}
```

More employees

```
namespace example_lib
{
    struct employee
    {
        std::string last_name;
        std::string first_name;
        double pay;
        long employee_number;
    };
}
```

```
struct employee
{
    int age;
    std::string surname;
    std::string forename;
    double salary;
};
```

Associating data with keys

```
namespace keys
{
    struct surname;
    struct forename;
    struct age;
    struct salary;
    struct employee_number;
}
```

Formatting our original employee

```
std::ostream& format_employee(std::ostream& s,
    example::employee const& e)
{
    return format(s,
      fusion::map_tie<
          keys::surname, keys::forename,
          keys::age, keys::salary>(
              e.surname, e.forename,
              e.age, e.salary));
}
```

Formatting the new guy

```
std::ostream& format_employee(std::ostream& s,
    example_lib::employee const& e)
{
    return format(s,
        fusion::map_tie<
            keys::surname, keys::forename,
            keys::employee_number, keys::salary>(
                e.last_name, e.first_name,
                e.employee_number,e.pay));
}
```

Our flexible formatter version 1.0

```
template<typename Map>
std::ostream& format(std::ostream& s, Map const& m)
{
    format_optional<keys::surname>(s, m, "<b>Surname: ", "</b>\n");
    format_optional<keys::forename>(s, m, "Forename: ", "\n");
    format_optional<keys::age>(s, m, "Age:", " years\n");
    format_optional<keys::salary>(s, m, "Salary: $", "\n");
    format_optional<keys::employee_number>(s, m, "<b>Employee no:
    ", "</b>\n");
    return s;
}
```

If the data is there...

```
template<typename Key, typename Map>

typename boost::enable_if<fusion::result_of::has_key<Map, Key>,
    std::ostream&>::type

format_optional(std::ostream& s, const Map& m, const std::string&
    prefix, const std::string& postfix)

{

    return s << prefix << fusion::at_key<Key>(m) << postfix;

}
```

and if the data is missing...

```
template<typename Key, typename Map>

typename boost::disable_if<fusion::result_of::has_key<Map, Key>,
    std::ostream&>::type

format_optional(std::ostream& s, const Map&, const std::string&, const
    std::string&)

{

    return s;

}
```

Redundancy again

```
template<typename Map>
std::ostream& format(std::ostream& s, Map const& m)
{
    format_optional<keys::surname>(s, m, "<b>Surname: ", "</b>\n");
    format_optional<keys::forename>(s, m, "Forename: ", "\n");
    format_optional<keys::age>(s, m, "Age:", " years\n");
    format_optional<keys::salary>(s, m, "Salary: $", "\n");
    format_optional<keys::employee_number>(s, m, "<b>Employee no:
    ", "</b>\n");
    return s;
}
```

Format specifications

```
template<typename Key>
struct format_spec
{
    format_spec(std::string const& pre, std::string const& post)
        : pre(pre), post(post)
    {}

    std::string pre;
    std::string post;
};
```

Formatting function object

```
template<typename Map>
struct format_member {
    format_member(Map const& lookup, std::ostream& stream)
        : lookup(lookup), stream(stream) {};

    template<typename Key>
    std::ostream& operator()(format_spec<Key> const& spec) const {
        return format_optional<Key>(stream, lookup, spec.pre, spec.post);
    }
    Map const& lookup;
    std::ostream& stream;
};
```

Our refactored version

```
template<typename Map>
    std::ostream& format(std::ostream& s, Map const& m) {
        fusion::for_each(
          fusion::make_vector(
            format_spec<keys::surname>("<b>Surname: ", "</b>\n"),
            format_spec<keys::forename>("Forename: ", "\n"),
            format_spec<keys::age>("<b>Age: ", "\n"),
            format_spec<keys::salary>("<b>Salary: $", "\n"),
            format_spec<keys::employee_number>("<b>Employee no: ", "</b>\n")),
          format_member<Map>(m, s));
        return s;
    }
```

The final version

```
template<typename Map, typename FwdSeq>
std::ostream& format(std::ostream& s, Map const& m, FwdSeq const& formatting)
{
    fusion::for_each(
      formatting,
      format_member<Map>(m, s));
    return s;
}
```

Going the extension route again

```
BOOST_FUSION_ADAPT_ASSOC_STRUCT(employee,
    (int, age, keys::age)
    (std::string, surname, keys::surname)
    (std::string, forename, keys::forename)
    (double, salary, keys::salary))
```

Another summary

- A fusion::map allows us to associate values with key types
- We can use fusion::map_tie to form maps in an ad-hoc manner
- Fusion maps provide an alternative to C++ structs to which many types can be adapted

A change to our data format

<surname>Marsden</surname>

<forename>Dan</forename>

<age>32</age>

<salary>123.45</salary>

The conventional approach

```
std::ostream& format_employee(std::ostream& stream, employee const& e)
{
    return stream << "<age>" << e.age << "</age>"
                  << "<surname>" << e.surname << "</surname>"
                  << "<forename>" << e.forename << "</forename>"
                  << "<salary>" << e.salary << "</salary>";
}
```

Augmenting the original data

```
std::ostream& format_xml(std::ostream& stream, employee const& e)
{
    fusion::for_each(
        fusion::zip(
            fusion::vector_tie(e.age, e.surname, e.forename, e.salary),
            fusion::make_vector("age", "surname", "forename", "salary")),
        format_member(stream));
    return stream;
}
```

Extracting the extra data

```
template<typename T>
   void operator()(T const& t) const
   {
      std::string fieldname = fusion::at_c<1>(t);
      stream_ << '<' << fieldname << '>'
              << fusion::at_c<0>(t)
              << "</" << fieldname << '>';
   }
```

Yet another summary

- fusion::zip can be used to form a single sequence of tuples from a collection of sequences
- Fusion's transformation algorithms return lazily evaluated views which permits efficient algorithm chaining

Can we improve our syntax

- Formatting arbitrary collections of values is clumsy:

```
format_xml(std::cout, fusion::vector_tie(a, b, c));

format_xml(std::cout, fusion::vector_tie(d, e));
```

- Can we make this work like a conventional function?

```
format_xml(std::cout, a, b, c);

format_xml(std:cout, d, e);
```

Formatting function object pt.1

```
struct format_xml_impl
{
    template<typename Seq>
    struct result
    {
        typedef std::ostream& type;
    };
...
};
```

Formatting function object pt.2

```
struct format_xml_impl  {
   ...
      template<typename Seq>
      std::ostream& operator()(Seq const& seq) const
      {
         std::ostream& stream = fusion::front(seq);
         boost::fusion::for_each(fusion::pop_front(seq),
   formatter(stream));
         return stream;
      }
   };
```

Variadic formatting

```
fusion::unfused_generic<format_xml_impl> const
format_xml((format_xml_impl()));

...

format_xml(std::cout, 32, 123.45, "marsden", "dan");

format_xml(std::cout, 'a', std::string("cat"), 32000);
```

Summary time again

- Fusion sequences can be parameter lists
- Parameter lists can be Fusion sequences
- Fusion permits implementation of variadic functions until C++0x gives us the real thing
  - Users prefer to work with argument lists
  - Implementation of variadics is easier to achieve with Fusion sequences and algorithms

We've only scratched the surface

- Built in support for standard types
  - boost::array, std::pair, boost::variant, boost::tuple
- Loads more containers, views and algorithms
- Extension support for easy adaptation of your own data types
- Lots lots more....

Thanks!