Fusion Authors Corner

Dan Marsden

May 1, 2009

Outline

1. Introduction

2. Structure of Boost.Fusion

3. Code time

Outline

1 Introduction

2 Structure of Boost.Fusion

3 Code time

Introduction

What is Boost.Fusion?

- A Fusion of compile time and runtime programming

Introduction

What is Boost.Fusion?

- A Fusion of compile time and runtime programming
- A library of heterogenous containers and algorithms

Introduction

What is Boost.Fusion?

- A Fusion of compile time and runtime programming
- A library of heterogenous containers and algorithms
- A tuple library

Some history

Chronology

- Standard library with std::pair
- Boost.Tuple accepted into Boost 1.24
- Fusion 1.x emerges under Boost.Spirit
- Fusion 2.x accepted into Boost 1.35

Credits

Joel de Guzman

- Primary author
- Structure, ideas and techniques
- Motivated by Spirit II needs

Credits

Joel de Guzman

- Primary author
- Structure, ideas and techniques
- Motivated by Spirit II needs

Tobias Schwinger

- Functional
- Infinite sequences

Built using Fusion

Fusion as infrastructure

- Spirit 2
- Boost.Proto
- Boost.Phoenix
- Boost.TR1
- Traversal library
- Dataflow library

Outline

1 Introduction

2 Structure of Boost.Fusion

3 Code time

Standing on the shoulders of giants

Borrowed from the STL

- Containers
- Algorithms
- Iterators

Standing on the shoulders of giants

Borrowed from the STL

- Containers
- Algorithms
- Iterators

Borrowed from the MPL

- Algorithm taxonomy
- Views
- Many naming conventions

Motivations for using Tuples

Reasons

Motivations for using Tuples

Reasons

- You cannot be bothered implementing a specific type

Motivations for using Tuples

Reasons

- You cannot be bothered implementing a specific type
- Naming is irrelevant or unhelpful

Motivations for using Tuples

Reasons

- You cannot be bothered implementing a specific type
- Naming is irrelevant or unhelpful
- You want to abstract away from names to just structure

Size and types are fixed at compile time

```
std::pair<int, float>
fusion::vector<int, char, std::string>
boost::array<int, 10>
```

Size and types are fixed at compile time

```
std::pair<int, float>
fusion::vector<int, char, std::string>
boost::array<int, 10>

std::vector<int>
boost::variant<int, std::string>
boost::optional<double>
```

Intrinsic operations

```
fusion::front(seq);
fusion::begin(seq);
fusion::empty(seq);
fusion::size(seq);
fusion::at<mpl::int_<0> >(seq);
fusion::at_key<my_type>(assoc_seq);
```

Polymorphic function object

```
struct print_strings
{
  typedef void result_type;

  void operator()(std::string const& s)
  {
    std::cout << s << '\n';
  }

  template<typename T>
  void operator()(T const&)
  {}
};
```

Algorithms

Iteration

```
fusion::for_each(seq, f);
```

Algorithms

Iteration

```
fusion::for_each(seq, f);
```

Querying

```
fusion::find<std::string>(seq);
fusion::any(seq, pred);
```

Algorithms

Iteration

```
fusion::for_each(seq, f);
```

Querying

```
fusion::find<std::string>(seq);
fusion::any(seq, pred);
```

Transformation

```
fusion::transform(xs, f);
fusion::remove_if<is_pointer<mpl::_> >(seq);
```

Outline

1 Introduction

2 Structure of Boost.Fusion

3 Code time

Tiers

```
std::string useful;
int handy;
double valuable;

// Break up results from a function
fusion::vector_tie(
  useful, handy, valuable) = f();

// Skip some stuff
fusion::vector_tie(
  fusion::ignore, handy, fusion::ignore) = f();
```

A simple example

```
bool operator<(T const& lhs, T const& rhs)
{
  if(lhs.a < rhs.a)
    return true;
  else if(rhs.a < lhs.a)
    return false;
  else if(lhs.b < rhs.b)
    return true;
  else if(rhs.b < lhs.b)
    return false;
  else
    return lhs.c < rhs.c;
}
```

A cheap excuse to use Tuples

```
bool operator<(T const& lhs, T const& rhs)
{
  return fusion::vector_tie(lhs.a, lhs,b, lhs.c)
    < fusion::vector_tie(rhs.a, rhs.b, rhs.c);
}
```

Lots of things are Fusion sequences

```
std::pair<std::string, int> p("dan", 303);
boost::tuple<int, char, double> t(101, 'a', 1.23);
boost::array<int, 3> a = {1,2,3};
```

A quick sequence of your own

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

And now stuff is convenient

```
employee a = ...;
employee b = ...;

bool const compared = a == b;

std::cout << a << std::endl;

int i;
fusion::vector_tie(
  i, fusion::ignore, fusion::ignore) = f();
```

Integration with other libraries

```
template <typename Iterator>
struct employee_parser
  : grammar<Iterator, employee(), space_type>
{
  ... // Amazing Spirit II stuff...
};
```

Details

- Employees are now first class Spirit II citizens
- Similar possibilities abound

Tuples of arguments

```
std::plus<int> add;
fusion::invoke(
  add,
  fusion::make_vector(1,1));

fusion::fused<std::plus<long> > f;
f(fusion::make_vector(1,2l));
```

More tuples of arguments

```
fusion::vector<int, float> a(2,2.0f);
fusion::vector<int, float> b(1,1.5f);

fusion::transform(
  fusion::zip(a,b),
  fusion::make_fused(std::minus<float>()));
```

Where do you get them all from?

```
fusion::unfused_generic<Func> g;

g(1,2,3);
g(1,2,3,4);
g(1,2,3,4,5);
...
```

Details

- Different variations depending on l-value / r-value needs
- A concretely typed version unfused_typed

Associative containers

```
format_address(
 fusion::map_tie<surname_t, address_t, zip_t>(
  lib_type.surname,
  lib_type.user_address,
  lib_type.zip_code));

format_address(
 fusion::map_tie<surname_t, address_t>(
  my_type.last_name,
  my_type.location));
```

Associative containers

- A “more flexible struct”
- Can use as a markup mechanism for data types

Cheap associative containers

```
BOOST_FUSION_ADAPT_ASSOC_STRUCT(
    demo::employee
    (std::string, last, surname_t)
    (std::string, address, address_t))

demo::employee const e = ...;
format_address(my_employee);
```