Implementing a mini Fusion 0x

Joel de Guzman and Dan Marsden

May 6, 2008

Outline

Outline

Aims

- To provide hands on experience with some C++ 0x features
  - static_assert
  - r-value references
  - Variadic templates
  - decltype

Aims

- To provide hands on experience with some C++ 0x features
  - static_assert
  - r-value references
  - Variadic templates
  - decltype
- To understand some of the internals of a Boost library

Aims

- To provide hands on experience with some C++ 0x features
  - static_assert
  - r-value references
  - Variadic templates
  - decltype
- To understand some of the internals of a Boost library
- An alternative route to learning about Boost.Fusion

Aims

- To provide hands on experience with some C++ 0x features
  - static_assert
  - r-value references
  - Variadic templates
  - decltype
- To understand some of the internals of a Boost library
- An alternative route to learning about Boost.Fusion
- Hopefully to encourage some discussion and experimentation - my "model" answers are probably far from perfect!

Prerequisites

- Experience with C++ 03
- gcc 4.3.0
- A copy of the Boost trunk or 1.35.0
- A copy of the presentation source code

Latest stuff

http://svn.boost.org/trac/boost/wiki/BoostFusion0x

Details

- Source code for the exercises
- Instructions for getting gcc 4.3.0 set up on your machine
- The latest version of this presentation

Outline

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 C++ world

- Conventional C++
  - std :: pair
- C++ 0x / TR1
  - std :: tr1 :: tuple
  - std :: tr1 :: array
- 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, Traversal
- Also suitable for use in application code

Key concepts of Fusion

- Containers
  - fusion :: vector , fusion :: list , fusion :: map
- Algorithms
  - Iteration
  - Querying
  - Transformation
- Iterators
- Views

Duality of runtime and compile time

Duality

- MPL sequences are Fusion sequences
- Fusion sequences are MPL sequences
- Algorithms, intrinsics etc. have compile time and runtime forms

```
fusion::reverse(...)
fusion::result_of::reverse<...>
```

Containers

```
fusion::vector<int, std::string> v(101, "hello");
std::pair<int, char> p(202, 'a');
struct my_struct {
  int a,
  float f
};
```

Intrinsic operations

```
fusion::vector<int, char, std::string> my_seq(
  101, 'a', "hello");

std::cout << fusion::size(my_seq) << '\n'; // 3
std::cout << fusion::at_c<1>(my_seq) << '\n'; // 'a'
std::cout << *fusion::begin(my_seq) << '\n'; // 101
```

Eager algorithms

Iterator based algorithms

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

Eager algorithms

Iterator based algorithms

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

```
fusion::vector<int, char, std::string> my_seq(
  101, 'a', "hello");

fusion::for_each(my_seq, poly_print());
```

Transforming algorithms are lazily evaluated

- Container independent algorithms - fusion::push_back, fusion::pop_back etc. have 1 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 need the whole thing!

Lazy algorithms

View based algorithms

```
fusion::transform(my_seq, f);
```

Lazy algorithms

View based algorithms

```
fusion::transform(my_seq, f);
```

```
fusion::vector<int, char, std::string> my_seq(
  101, 'a', "hello");

// This costs approximately nothing!
fusion::transform(my_seq, make_massive_vector());
```

Fixed at compile time

- The number of types in a container

Fixed at compile time

- The number of types in a container

```
fusion::vector<int, char> my_vec(101, 'a');
```

Fixed at compile time

- The number of types in a container

```
fusion::vector<int, char> my_vec(101, 'a');
```

- The order of types in a container

Fixed at compile time

- The number of types in a container

```
fusion::vector<int, char> my_vec(101, 'a');
```

- The order of types in a container

```
fusion::reverse(my_vec);
```

Fixed at compile time

- The number of types in a container

```
fusion::vector<int, char> my_vec(101, 'a');
```

- The order of types in a container

```
fusion::reverse(my_vec);
```

- Iterator positions

Fixed at compile time

- The number of types in a container

```
fusion::vector<int, char> my_vec(101, 'a');
```

- The order of types in a container

```
fusion::reverse(my_vec);
```

- Iterator positions

```
fusion::begin(my_vec);
fusion::next(fusion::begin(my_vec));
```

Fixed at compile time

- The number of types in a container

```
fusion::vector<int, char> my_vec(101, 'a');
```

- The order of types in a container

```
fusion::reverse(my_vec);
```

- Iterator positions

```
fusion::begin(my_vec);
fusion::next(fusion::begin(my_vec));
```

Outline

What to keep in Mini Fusion?

- At least one container
- Iterators
- A small number of intrinsics

What to cut out of Mini Fusion?

- Views
- Algorithms (big decision!)
- Support for reference members and other niceties
- MPL support
- Lots of other bits and pieces

Our starting point

- A cons cell based container
- A limited number of "intrinsic operations"
  at_c, size, begin, end
- An iterator implementation

```
fusion::cons<int, fusion::nil> cell(
   101, fusion::nil());

BOOST_TEST(fusion::size(cell) == 1);
BOOST_TEST(fusion::at_c<0>(cell) == 101);
```

Our target

- Lots of improvements to our cons cell based container

Our target

- Lots of improvements to our cons cell based container
- A list like interface

Our target

- Lots of improvements to our cons cell based container
- A list like interface

```
fusion::list<int, char, std::string> lst(
  101, 'a', "hello");
```

Our target

- Lots of improvements to our cons cell based container
- A list like interface

```
fusion::list<int, char, std::string> lst(
  101, 'a', "hello");
```

- A list builder convenience function

Our target

- Lots of improvements to our cons cell based container
- A list like interface

```
fusion::list<int, char, std::string> lst(
  101, 'a', "hello");
```

- A list builder convenience function

```
fusion::make_list(
  101, 'a', std::string("hello"));
```

Our target

- Lots of improvements to our cons cell based container
- A list like interface

```
fusion::list<int, char, std::string> lst(
  101, 'a', "hello");
```

- A list builder convenience function

```
fusion::make_list(
  101, 'a', std::string("hello"));
```

- Lots of performance and extensibility improvements to the list

Our target

- Lots of improvements to our cons cell based container
- A list like interface

```
fusion::list<int, char, std::string> lst(
  101, 'a', "hello");
```

- A list builder convenience function

```
fusion::make_list(
  101, 'a', std::string("hello"));
```

- Lots of performance and extensibility improvements to the list
- 2 more container types

Outline

C++ 03 warmup exercise

Add a convenience list interface to our cons building blocks. Only C++03 language features should be used for this exercise.

```
typedef fusion::list<int, char> my_list_type;

my_list_type my_list(101, 'a');
```

Exercise - Hints

- We're not going to be able to do arbitrary size lists in C++03 (suggest sticking to length 3 or less)
- Multiple specializations of fusion::list are going to be needed
- Its going to be really repetitive...

Outline

Introducing static_assert

```
static_assert(sizeof(int) == 4,
  "problem!")
```

```
cons.cpp:10: error: static assertion failed: "problem!"
```

Jamfile hints

user-config.jam

```
using gcc : 4.3.0 : g++4.3.0 ;
```

Jamfile hints

user-config.jam

```
using gcc : 4.3.0 : g++4.3.0 ;
```

Jamfile

```
project : requirements
  <cxxflags>--std=c++0x
  <linkflags>--std=c++0x ;
```

Exercise

A C++0x warm up. Add bounds checking to the fusion :: at_c operator, so that clearer error messages are produced

```
fusion::list<int, char> l(int, 'c');
fusion::at_c<3>(l); // Clear error here!
```

Imperfect forwarding in C++ 03

```
template<typename Func, typename T0>
void forward1(Func f, T& t0);

template<typename Func, typename T0>
void forward1(Func f, T const& t0);

template<typename Func, typename T0, typename T1>
void forward2(Func f, T0& t0, T1& t1);

template<typename Func, typename T0, typename T1>
void forward2(Func f, T0 const& t0, T1& t1);

// And 2 more!
```

r-value references 101

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

template<typename T>
void lvalue(T&) {}

template<typename T>
void crvalue(T const&&) {}

template<typename T>
void rvalue(T&&) {}

clvalue(A()); // Good
lvalue(A());  // Error
crvalue(A()); // Good
rvalue(A());  // Still good!
```

R-value references summary

Details

- r-values will not bind to a non-const reference
- r-values will bind to an r-value reference
- Allows us to identify temporaries
- Also permits us to implement "perfect forwarding"

Perfect forwarding in C++ 0x

```
template<typename Func, typename T0, typename T1>
void forward2(Func f, T0&& t0, T1&& t1)
{
  f(std::forward<T0>(t0),
    std::forward<T1>(t1));
}
```

Details

- r-value references combined with std::forward allow us to recover all the necessary type information
- We can do even better with variadic templates later

Exercise

Exercise

- Add perfect forwarding support to the cons constructor

```
fusion::list<std::string, my_type> l(
    make_string(), make_mine());
```

Moving

```
// Pass results along a chain of calls
std::string my_string(
  f1(f2(f3(f4(101, more_stuff)))));
...
// Do lots of exciting stuff with my_string
...
std::string my_other_string(std::move(my_string));
```

Details

- r-values allow efficient chaining of function return values
- std::move allow us to confer temporary status on l-values

Move

```
template <class T>
typename remove_reference<T>::type&&
move(T&& a) {
  return a;
}
```

Details

- Converts both l-value and r-values to r-values
- remove_reference strips off any l-value references

Exercise

Add constructors and assignment operators with move support to the cons implementation.

```
// No unnecessary copying
fusion::cons<my_type, fusion::nil> c(make_my_cons());

// Still no unnecessary copying
c = make_my_cons();
```

Variadic class templates

```
template<typename... Args>
struct my_template;
```

Template parameter packs

- The ellipsis indicates Args is a template parameter pack
- Parameters packs for types, non type parameters and template template parameters

Packing and unpacking

```
template<typename... Args>
struct my_template
  : my_base<Args...>
{};
```

Details

- An ellipsis to the left packs parameters
- An ellipsis to the right unpacks parameters again

Variadic function templates

```
template<typename... Args>
void f(Args... args);
{
  g(args...);
}
```

Details

- Args... args captures a function parameter pack
- An ellipsis to the left packs together arguments
- An ellipsis to the right unpacks arguments again

Disassembling variadic templates

```
template<typename A0, typename... Args>
void g(A0 a0, Args... args)
{
  something_interesting(a0);
  g(args...);
}

void g(){}
```

Details

- Disassembly similar to pattern matching in functional languages
- Can pattern match to break up type and value parameter packs

Exercise

Use variadic templates to make the list interface extensible to an arbitrary number of values

```
fusion::list<int,char,int,char> lst(
    1,'a',2,'b');
```

Exercise

Use variadic templates to provide a converting constructor to the list type.

```
fusion::list<int, float> lst(101, 0.123);
fusion::list<long, double> lst2(lst1);
```

Variadic templates again

```
template<typename... Args>
void func(Args&... args)
{
  gunc(&args...);
}
```

Unpacking patterns

- Expanded for each parameter in a parameter pack
- Can be used to influence template argument deduction

Exercise

Add move construction and move assignment to the list interface

```
// No unnecessary copying
fusion::list<std::string, my_type> lst(
  make_mine());

// Still no unnecessary copying
lst = make_mine();
```

Exercise - Hints

Hints

- We need to identify temporary lists during copying and assignment
- We probably want to enforce consistent list sizes using `static_assert`
- We need to pass on the individual members of the list as temporaries
- `std::move` sounds good for this!

Exercise

Add forwarding to the list constructor

```
fusion::list<std::string, my_type> lst(
    make_string(), make_mine());
```

Decay type trait

```
template<typename U>
struct decay<U>;
```

Summary

- Decays array types to the associated point type
- Decays function reference types to the associated function pointer
- Strips references

Exercise

Add a `fusion::make_list` convenience function for building lists. Ensure it uses perfect forwarding to pass the arguments on to the underlying list.

```
my_type mine;
fusion::make_list(101, std::string, mine);
```

Hints

- Our output `list` should not contain references
- We should cope with arrays as argument types

Exercise

Write a metafunction to calculate the length of an `fusion::list`

```
fusion::list_size<
  fusion::list<int, char>>::value == 2;
fusion::list_size<
  fusion::list<int,int,int,int>>::value == 4;
```

Parameter pack convenience

Details

- Checking the number of parameters in a parameter pack is a common operation
- **sizeof** works for parameter packs

```
template<typename... Args>
struct count_args
{
  static const int value = sizeof...(Args);
};
```

decltype 101

```
const int&& foo();
int i;
struct A { double x; }
const A* a = new A();
decltype(foo()); // type is const int&&
decltype(i); // type is int
decltype(a->x); // type is double
decltype((a->x)); // type is const double&
```

Details

- decltype gives the declared type of an expression
- The current gcc implementation lacks some related features

The missing friends of decltype

```
template<typename Lhs, typename Rhs>
auto f(Lhs const& lhs, Rhs const& rhs)
  -> decltype(lhs + rhs);

for( auto it = v.begin(); it != v.end(); ++it ) {
  // use *it ...
}
```

Details

- The new function syntax permits access to function parameter names in a subsequent decltype expression
- **auto** permits inference of the type of variables from their initializers

Exercise

```
// Order N templates instantiated
fusion::at_c<10>(my_list);
fusion::at_c<101>(my_list);
```

Details

- fusion :: at_c is currently O(N) at compile time on fusion :: list
- Use decltype to reduce this to O(1)

Exercise - Hints

```
// Pseudo code
typedef decltype(expr(sequence, index)) member_type;
```

Hints

- If we're going to use decltype, we're going to need a family of expressions that give us the data types

Exercise - Hints

```
// Pseudo code
typedef decltype(expr(sequence, index)) member_type;
```

Hints

- If we're going to use decltype, we're going to need a family of expressions that give us the data types
- A family of overloaded (member) functions might be suitable

Exercise - Hints

```
// Pseudo code
typedef decltype(expr(sequence, index)) member_type;
```

Hints

- If we're going to use decltype, we're going to need a family of expressions that give us the data types
- A family of overloaded (member) functions might be suitable
- How would we build up such a family?

Outline

Extended Exercise

```
fusion::array<std::string, 10, 10> arr;
fusion::at_c<2,3>(arr) = "hello";
```

Details

- Design a multidimensional array like interface
- Provide a Mini Fusion like interface to the array (including iterators)
- Extend `fusion::at_c` for multidimensional indexing
- Plenty of bounds checking
- Be creative!

Extended Exercise - Hints

Details

- Non type variadic template parameters will help with our array dimensions
- `fusion::at_c` can be extended in a similar way
- `static_assert` will be needed in many places for bounds checking
- Iteration should (probably) flatten the multiple dimensions back to one linear set

Extended Exercise

```
fusion::map<
  fusion::pair<int, std::string>,
  fusion::pair<char, int> > my_map(
    "hello", 101);

assert(fusion::at_key<int>(my_map) == "hello");
assert(fusion::at_key<char>(my_map) == 101);
```

Details

- Design a hetrogenous map like container
- Aim for 0(1) lookups at both compile time and runtime

Extended Exercise - Hints

Details

- We need a type to value pair to attach labels to data
- We only want pairs as type arguments - can we enforce this?
- We can use the decltype trick seen previously to implement fast lookups
- The map can still be considered as a sequence for iteration / random access