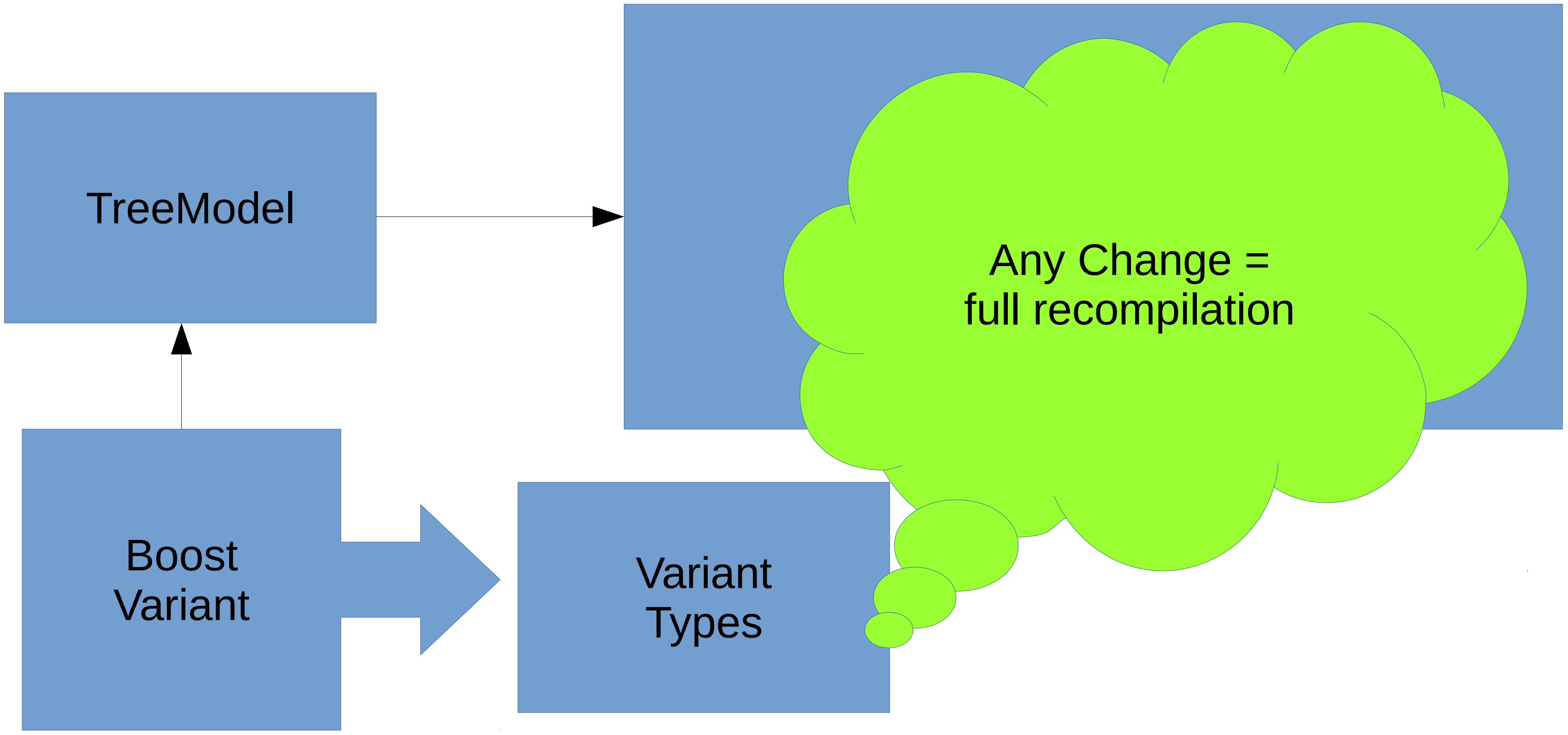
A different way to use boost serialization

Jens Weller
CppCon 2015 lightning talks

About me

- C++ Evangelist
- C++ since '98
- '02-'07 Vodafone
- '07 selfemployed / freelancer in C++
- '12 Meeting C++

Application Overview



Boost Serialization

- My Choice to serialize things
- 2 official ways in the documentation
  - Intrusive
    - Means any change is a recompilation
  - Non intrusive
    - Make your members public!
  - Both are not an option

First try

- Put all members in a tuple
  - tie(member0, … memberN)
  - Expose this through a method
    - tuple<int&> tuple_access(){return tie(m_int);}
  - Don't forget to use references in the tuple...
- Now I only need some way to serialize this tuple
  - Boost.Fusion
  - Fusion.for_each
  - Custom class to serialize each tuple item

Some Helper Macros

```
#define TIE_ELEMENT(TE) TE

#define TIE_MACRO(r, data, i, elem) BOOST_PP_COMMA_IF(i) TIE_ELEMENT(elem)

#define TIE(...) tuple_ns::tie( BOOST_PP_SEQ_FOR_EACH_I(TIE_MACRO, _,\
 BOOST_PP_VARIADIC_TO_SEQ(__VA_ARGS__)) )

#define TUPLE_ACCESS(...) auto tuple_access() -> decltype( TIE(__VA_ARGS__) )\
{ return TIE(__VA_ARGS__);}



TUPLE_ACCESS(member0,member1);
```

Some Helper Macros

```
#define TIE_ELEMENT(TE) TE

#define TIE_MACRO(r, data, i, elem) BOOST_PP_COMMA_IF(i) TIE_ELEMENT(elem)

#define TIE(...) tuple_ns::tie( BOOST_PP_SEQ_FOR_EACH_I(TIE_MACRO, _,\
 BOOST_PP_VARIADIC_TO_SEQ(__VA_ARGS__)) )

#define TUPLE_ACCESS(...) auto tuple_access() -> decltype( TIE(__VA_ARGS__) )\
{ return TIE(__VA_ARGS__);}


TUPLE_ACCESS(member0,member1);
```

Some Helper Macros

```
#define TIE_ELEMENT(TE) TE

#define TIE_MACRO(r, data, i, elem) BOOST_PP_COMMA_IF(i) TIE_ELEMENT(elem)

#define TIE(...) tuple_ns::tie( BOOST_PP_SEQ_FOR_EACH_I(TIE_MACRO, _,\
 BOOST_PP_VARIADIC_TO_SEQ(__VA_ARGS__)) )

#define TUPLE_ACCESS(...) auto tuple_access() -> decltype( TIE(__VA_ARGS__) )\
{ return TIE(__VA_ARGS__);}


TUPLE_ACCESS(member0,member1);
```

Some Helper Macros

```
#define SERIALIZE_TYPE(Type)\
template<class Archive>
void serialize(Archive& ar, Type &t, const unsigned int )
{
  fusion::for_each(t.tuple_access(),fusion_helper<Archive>(ar));
}
```

```
template<class Archive>
class fusion_helper
{
    Archive& ar;
public:
    explicit fusion_helper(Archive& ar):ar(ar){}
    template<class T>
    void operator()( T&t)const
    {
        ar & t;
    }
};
```

Some Helper Macros

```
#define SERIALIZE_TYPE(Type)\
template<class Archive>
void serialize(Archive& ar, Type &t, const unsigned int )
{
  fusion::for_each(t.tuple_access(),fusion_helper<Archive>(ar));
}
```

```
template<class Archive>
class fusion_helper
{
    Archive& ar;
public:
    explicit fusion_helper(Archive& ar):ar(ar){}
    template<class T>
    void operator()( T&t)const
    {
        ar & t;
    }
};
```

Some Helper Macros

```
#define SERIALIZE_TYPE(Type)\
template<class Archive>
void serialize(Archive& ar, Type &t, const unsigned int )
{
  fusion::for_each(t.tuple_access(),fusion_helper<Archive>(ar));
}
```

```
template<class Archive>
class fusion_helper
{
    Archive& ar;
public:
    explicit fusion_helper(Archive& ar):ar(ar){}
    template<class T>
    void operator()( T&t)const
    {
        ar & t;
    }
};
```

This works o.O

- Advantage
  - Serialization code
    - In one place
    - Classes don't know about serialization

- Disadvantage
  - Feels a little dirty
    - WTF
    - WAT
  - Tuple
    - Suboptimal performance

Lets fix this!

- Feedback from reddit
  - Make the free serialization function a friend!
  - Members can stay private
  - Template method added to each class
    - No extra header
  - Serialization needs not to be a friend
- How to avoid recompilation hell?

Some Helper Macros

```
#define SERIALIZE(Type)  template<class Archive>
friend void serialize(Archive& ar, Type &t, const unsigned int );
#define ELEMENT(TE) TE
#define ELEMENT_MACRO(r, data, i, elem) ar & t. ELEMENT(elem);
#define FRIEND_ELEMENT(...) BOOST_PP_SEQ_FOR_EACH_I(ELEMENT_MACRO, _,\
 BOOST_PP_VARIADIC_TO_SEQ(__VA_ARGS__))
#define SERIALIZE_IMPL(Type,...) \
template<class Archive>
void serialize(Archive& ar, Type &t, const unsigned int )
{ FRIEND_ELEMENT(__VA_ARGS__)}
#define SERIALIZE_DERIVED_IMPL(Type,Base,...) ...
```

Some Helper Macros

```
#define SERIALIZE(Type)  template<class Archive>
friend void serialize(Archive& ar, Type &t, const unsigned int );
#define ELEMENT(TE) TE
#define ELEMENT_MACRO(r, data, i, elem) ar & t. ELEMENT(elem);
#define FRIEND_ELEMENT(...) BOOST_PP_SEQ_FOR_EACH_I(ELEMENT_MACRO, _,\
 BOOST_PP_VARIADIC_TO_SEQ(__VA_ARGS__))
#define SERIALIZE_IMPL(Type,...) \
template<class Archive>
void serialize(Archive& ar, Type &t, const unsigned int )
{ FRIEND_ELEMENT(__VA_ARGS__)}
#define SERIALIZE_DERIVED_IMPL(Type,Base,...) ...
```

Some Helper Macros

```
#define SERIALIZE(Type)  template<class Archive>
friend void serialize(Archive& ar, Type &t, const unsigned int );
#define ELEMENT(TE) TE
#define ELEMENT_MACRO(r, data, i, elem) ar & t. ELEMENT(elem);
#define FRIEND_ELEMENT(...) BOOST_PP_SEQ_FOR_EACH_I(ELEMENT_MACRO, _,\
 BOOST_PP_VARIADIC_TO_SEQ(__VA_ARGS__))
#define SERIALIZE_IMPL(Type,...) \
template<class Archive>
void serialize(Archive& ar, Type &t, const unsigned int )
{ FRIEND_ELEMENT(__VA_ARGS__)}
#define SERIALIZE_DERIVED_IMPL(Type,Base,...) ...
```

Final Solution

- SERIALIZE(TYPE)
  - In every serializable class
- SERIALIZE_IMPL(TYPE,member0,...)
  - In serializer.cpp
- All Serialization code is still in one place
- SERIALIZE_DERIVED_IMPL
  - For derived classes

std::end(slides)

Thanks for listening!