Boost.Proto v5 Preview

or, "C++11 FTW"

Talk Overview

- Basic Example: Boost.Assign
  - Front Ends, Back Ends
  - Simple Grammars and Transforms
  - User-Defined Expressions
- C++11 and library design
  - Static initialization
  - Library versioning
  - Better template error messages

Example 1

`map_list_of()` from Boost.Assign

map_list_of

```
#include <map>
#include <cassert>
#include <boost/assign/list_of.hpp> // for 'map_list_of()'
using namespace boost::assign; // bring 'map_list_of()' into scope

int main()
{
    std::map<int,int> m = map_list_of(1,2)(2,3)(3,4)(4,5)(5,6);
    assert( m.size() == 5 );
    assert( m[ 1 ] == 2 );
    assert( m[ 5 ] == 6 );
}
```

What is map_list_of?

Proto Front Ends

Plant a seed, grow a tree

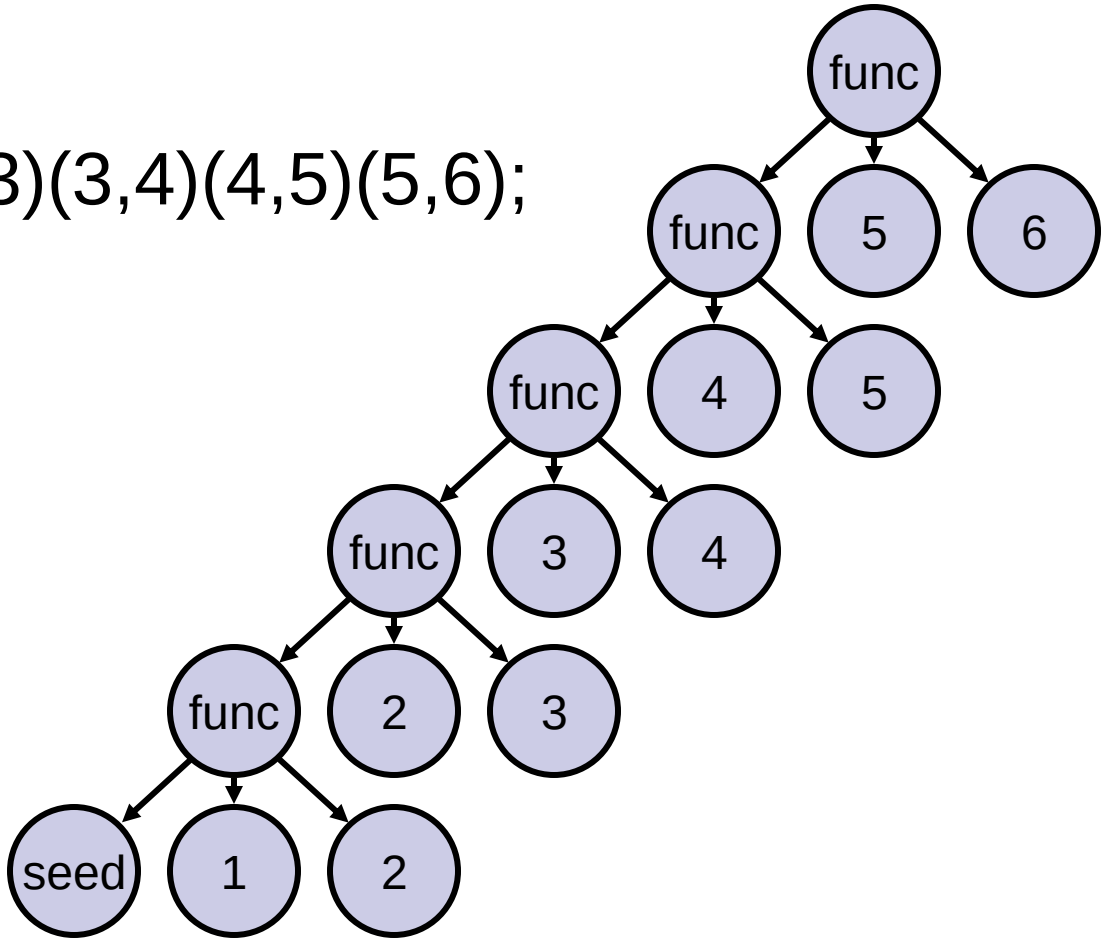
Define a “Seed” Terminal

```
#include <boost/proto/v5/proto.hpp>
namespace proto = boost::proto;

struct map_list_of_ {};
constexpr proto::expr<proto::terminal(map_list_of_)> map_list_of {};

int main()
{
    map_list_of(1,2)(2,3)(3,4)(4,5)(5,6);
}
```

Compiles and runs!
(And does nothing.)

Build a Tree

map_list_of(1,2)(2,3)(3,4)(4,5)(5,6);



Pretty-print trees with display_expr

```
#include <iostream>
#include <boost/proto/v5/proto.hpp>
namespace proto = boost::proto;

struct map_list_of_ {};
constexpr proto::expr<proto::terminal(map_list_of_)> map_list_of {};

int main()
{
    proto::display_expr(
        map_list_of(1,2)(2,3)(3,4)(4,5)(5,6)
    );
}
```

Pretty-print trees with display_expr

```
#include <iostream>
#include <boos
namespace pro

struct map_list_
constexpr proto

int main()
{
    proto::displ
        map_list_(
    );
}
```

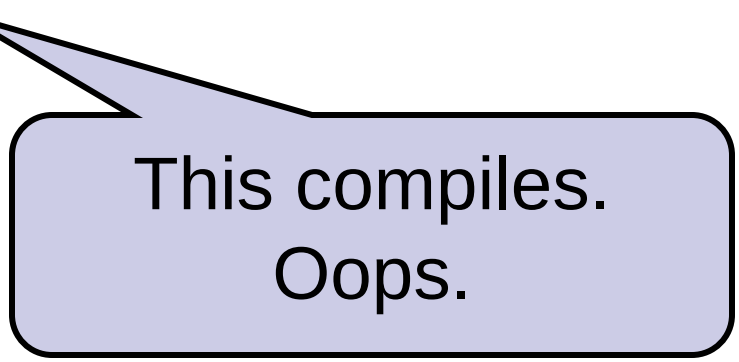
```
C:\proto-0x\libs\proto\v5\scratch\proto.exe
function(
    function(
        function(
            function(
                function(
                    terminal( (map_list_of_) map_list_of_ ) const &
                  , terminal( (int) 1 )
                  , terminal( (int) 2 )
                )
              , terminal( (int) 2 )
              , terminal( (int) 3 )
            )
          , terminal( (int) 3 )
          , terminal( (int) 4 )
        )
      , terminal( (int) 4 )
      , terminal( (int) 5 )
    )
  , terminal( (int) 5 )
  , terminal( (int) 6 )
)
```

Expression Tree Validation

Spotting invalid expressions

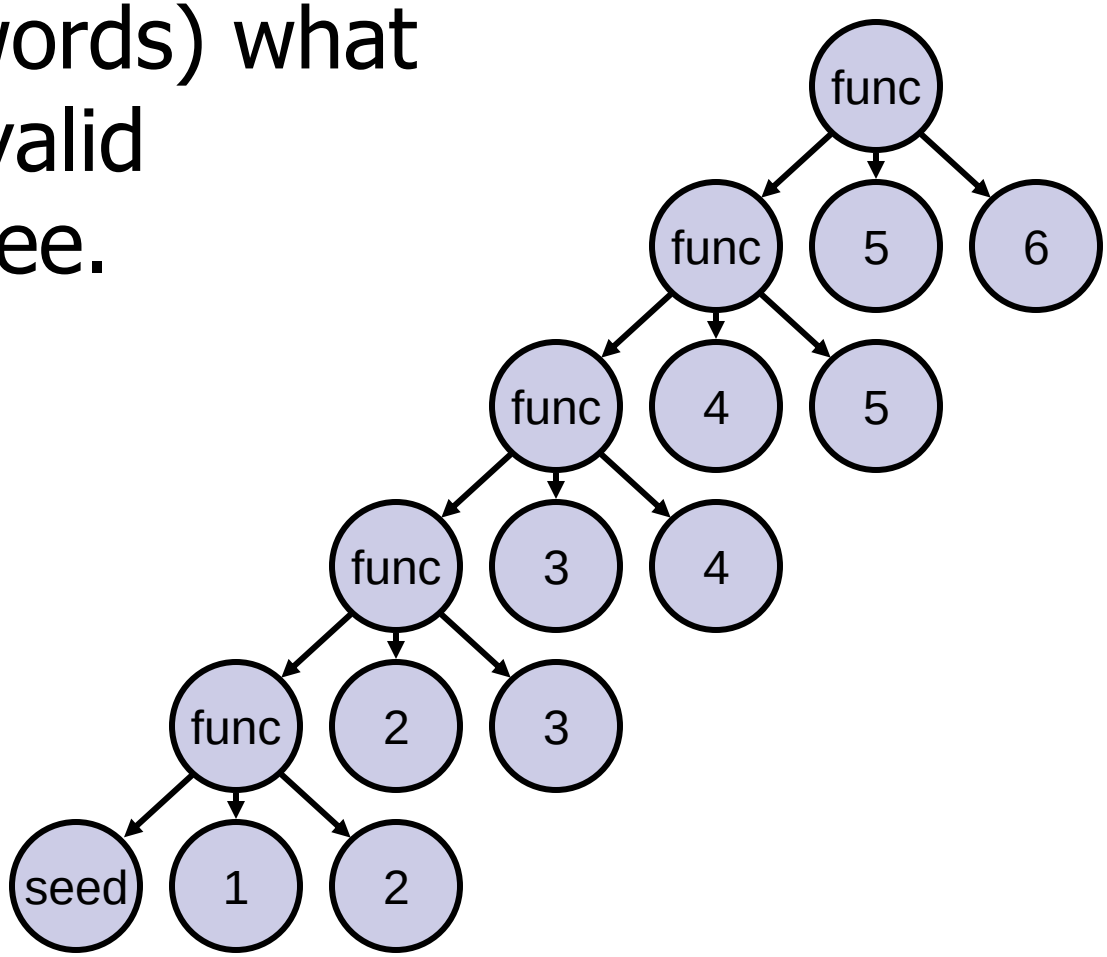
Proto's Promiscuous Operators

```
#include <boost/proto/v5/proto.hpp>
namespace proto = boost::proto;

struct map_list_of_ {};
constexpr proto::expr<proto::terminal(map_list_of_)> map_list_of {};

int main()
{
    map_list_of(1,2) * 32 << map_list_of; // WTF???!!!
}
```



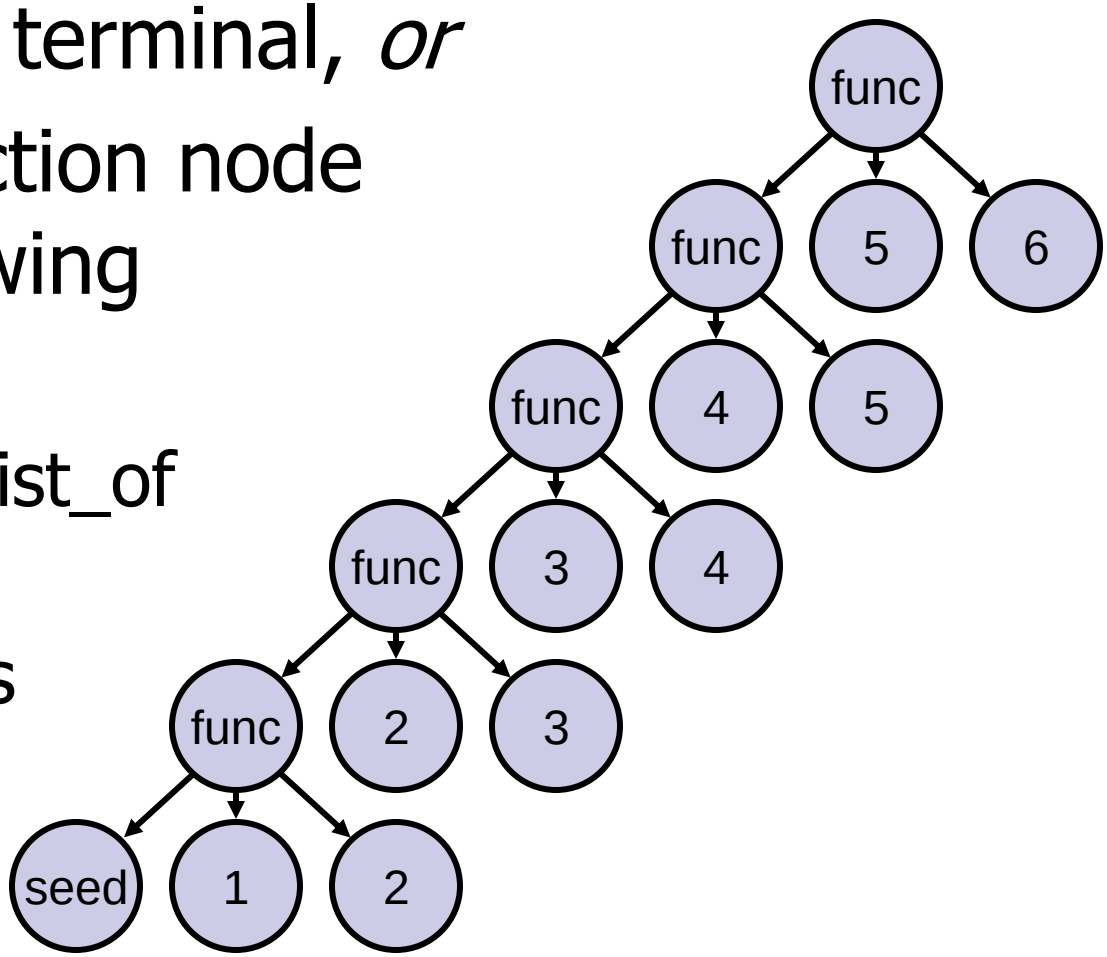
A valid map_list_of tree is …

- Describe (in words) what makes this a valid map_list_of tree.



A valid map_list_of tree is …

- A map_list_of terminal, *or*
- A ternary function node with the following children:
  1. A valid map_list_of tree
  2. Two terminals



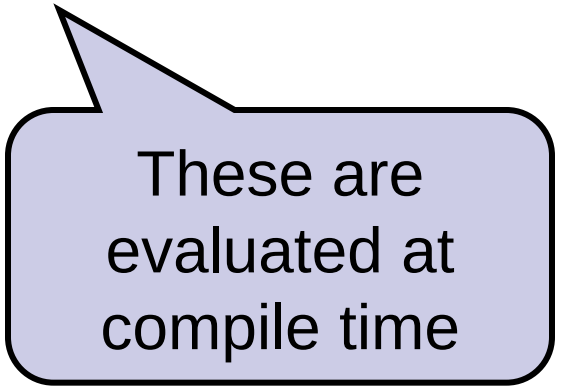
A valid map_list_of tree is …

- A map_list_of terminal, *or*
- A ternary function node with the following children:
  1. A valid map_list_of tree
  2. Two terminals

```
using proto::_;

struct MapListOf : proto::def<
  proto::match(
        proto::terminal(map_list_of_),
        proto::function(
            MapListOf,
            proto::terminal(_),
            proto::terminal(_)
        )
    )
> {};
```

Detecting Wild Expressions

```
struct MapListOf : /* as before */ {};

int main()
{
  BOOST_PROTO_ASSERT_MATCHES(
      map_list_of(1,2)(2,3)(3,4)(4,5)(5,6), MapListOf );

  BOOST_PROTO_ASSERT_MATCHES_NOT(
      map_list_of(1,2) * 32 << map_list_of, MapListOf );
}
```

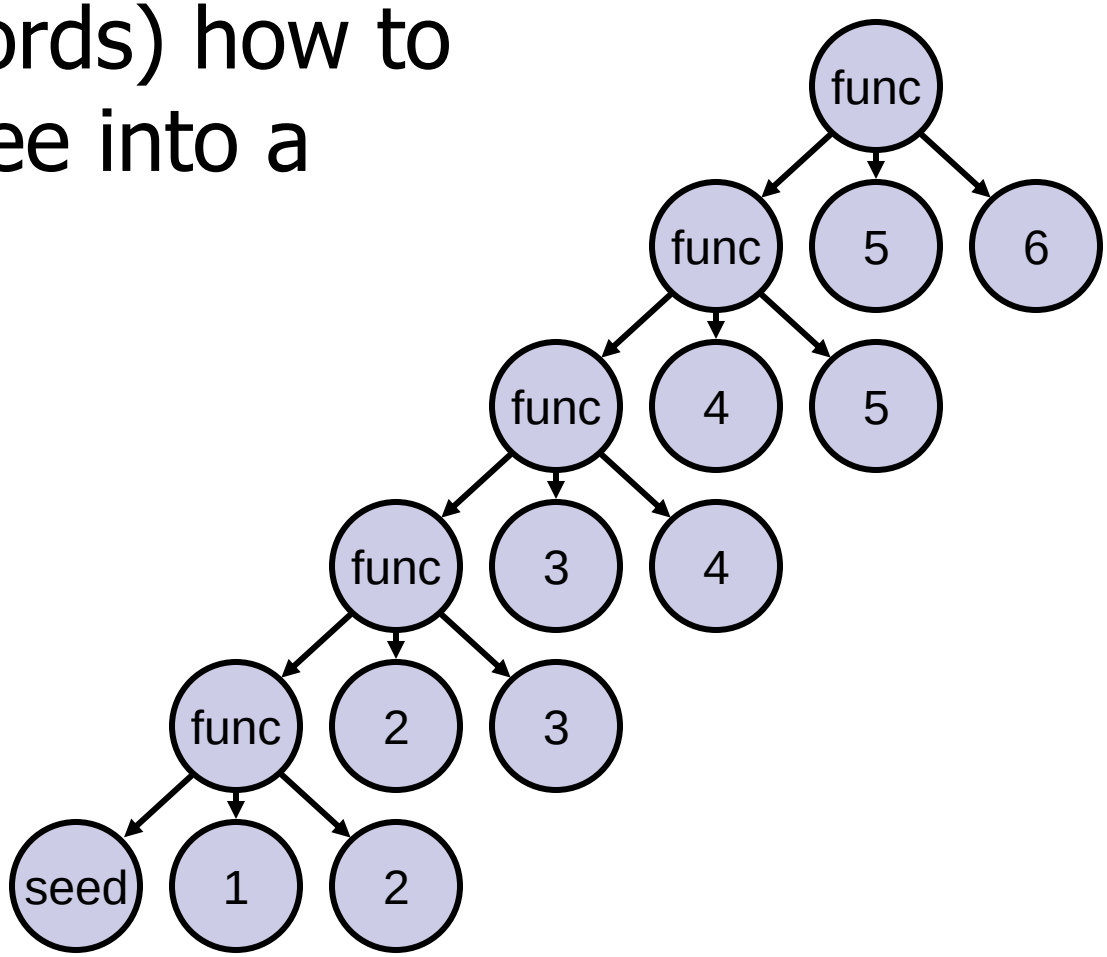


Back Ends

Doing *stuff* with expressions

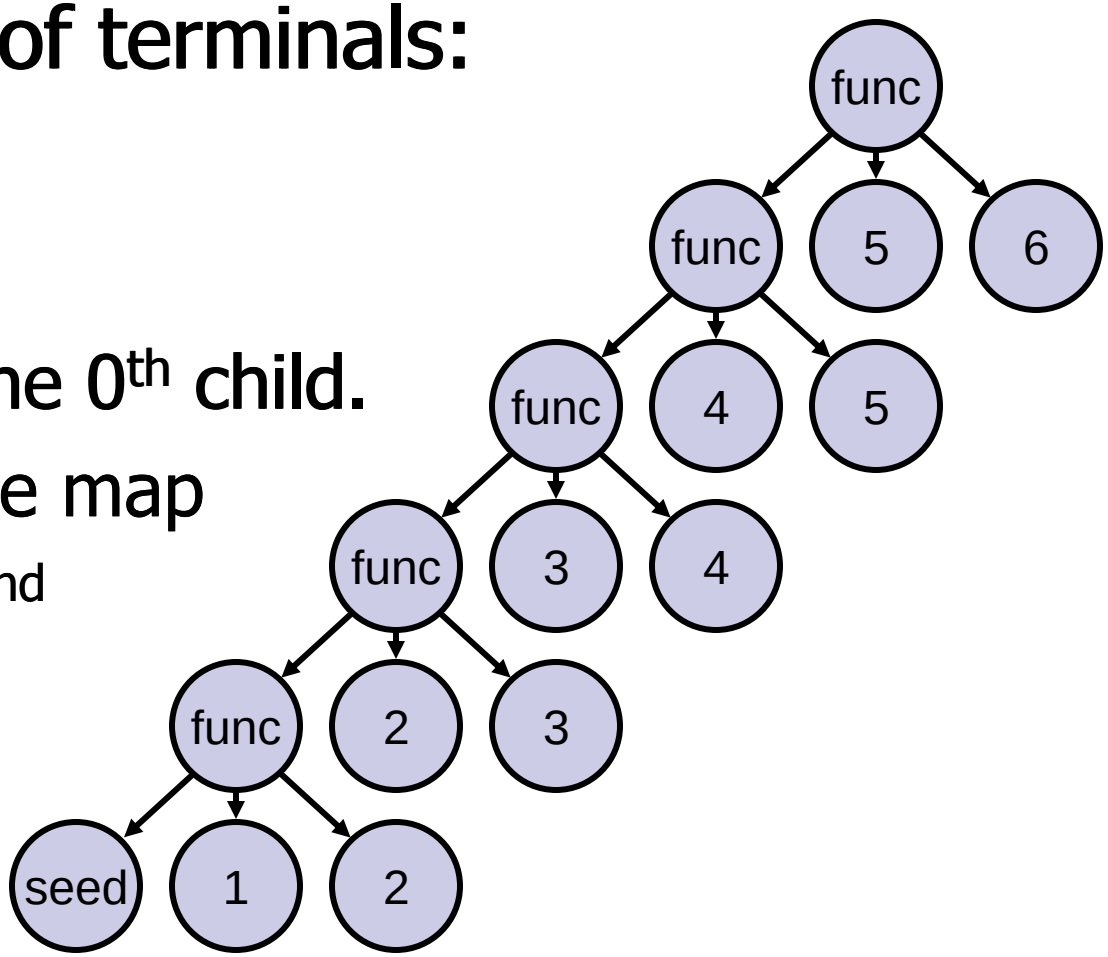
Populate a map from a tree …

- Describe (in words) how to turn the this tree into a std::map.



Populate a map from a tree …

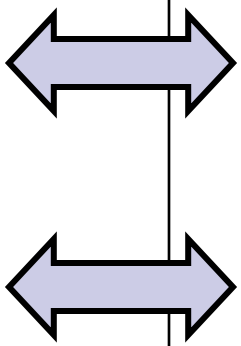
- For map_list_of terminals:
  1. Do nothing.
- Otherwise:
  1. Recurse on the 0th child.
  2. Insert into the map the 1st and 2nd children.



Grammars and Algorithms

A valid map_list_of tree is:

- A map_list_of terminal, or
- A ternary function node with the following children:
  1. A valid map_list_of tree
  2. Two terminals



Populate a map from a tree:

- For map_list_of terminals:
  1. Do nothing.
- Otherwise:
  1. Recurse on the 0th child.
  2. Insert into the map the 1st and 2nd children.

Populate a map from a tree…

- For map_list_of terminals:
  1. Do nothing.
- Otherwise:
  1. Recurse on the 0^{th} child.
  2. Insert into the map the 1^{st} and 2^{nd} children.

```
struct MapListOf : def<
  proto::match(
        proto::terminal(map_list_of_),
        proto::function(
            MapListOf,
            proto::terminal(_),
            proto::terminal(_)
        )
    )
> {};
```

Populate a map from a tree…

- For map_list_of terminals:
  1. Do nothing.

```
proto::case_(
    proto::terminal(map_list_of_),
    void()
)
```

Use proto::case_ to associate an action with a grammar rule.

Populate a map from a tree…

- For map_list_of terminals:
  1. Do nothing.
- Otherwise:
  1. Recurse on the 0^{th} child.
  2. Insert into the map the 1^{st} and 2^{nd} children.

```
struct MapListOf : proto::def<
  proto::match(
        proto::terminal(map_list_of_),
        proto::function(
            MapListOf,
            proto::terminal(_),
            proto::terminal(_)
        )
    )
> {};
```

Populate a map from a tree…

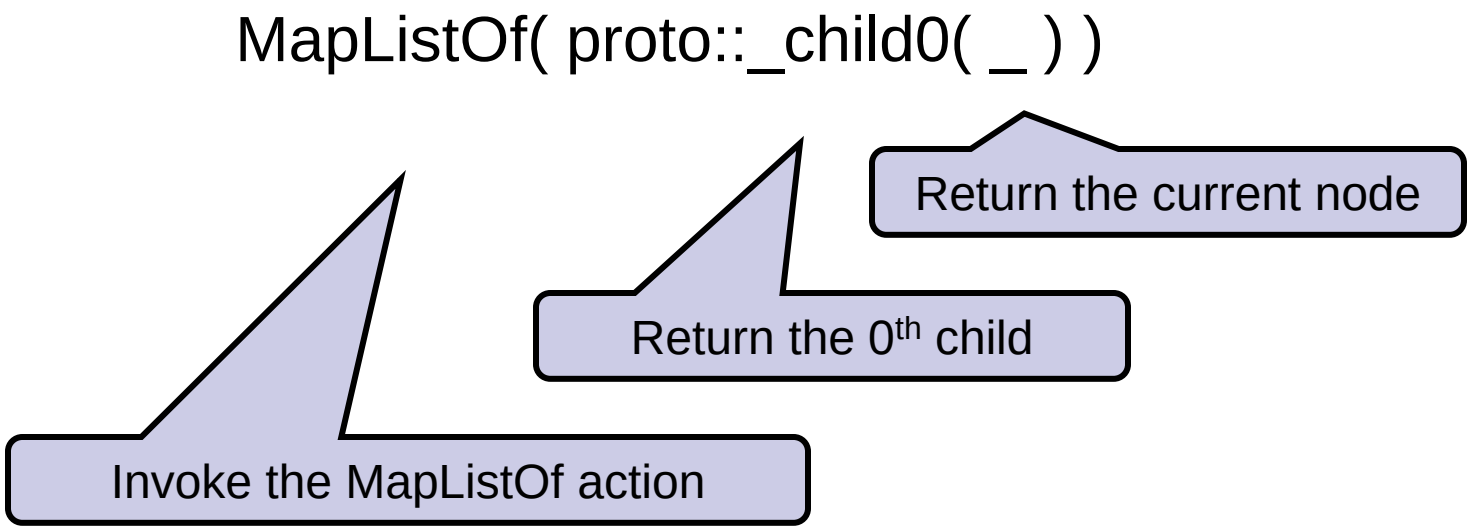
Use function types to compose actions.

- Otherwise:
  1. Recurse on the 0th child.
  2. Insert into the map the 1st and 2nd children.

```
proto::case_(
    proto::function(
        MapListOf,
        proto::terminal(_),
        proto::terminal(_)
    ),
    MapListOf(proto::_child0(_)),
    /* … */
)
```

Composite Actions

- Use function *types* to represent function *invocations*.



Composite Actions

MapListOf(proto::_child0)

All actions operate on the current node by default.

Populate a map from a tree…

Define your own actions.

- Otherwise:
  1. Recurse on the 0^{th} child.
  2. Insert into the map the 1^{st} and 2^{nd} children.

```
proto::case_(
  proto::function(
    MapListOf,
    proto::terminal(_),
    proto::terminal(_)
  ),
  MapListOf(proto::_child0),
  map_insert(
    proto::_data,
    proto::_value(proto::_child1),
    proto::_value(proto::_child2)
  )
)
```

Populate a map from a tree…

Define your own actions.

```
// A simple function object that
// inserts a (key, value) pair into a map.
struct map_insert
{
    template<class M, class K, class V>
    void operator()(M & m, K k, V v) const
    {
        m[ k ] = v;
    }
};
```

```
proto::case_(
    proto::function(
        MapListOf,
        proto::terminal(_),
        proto::terminal(_)
    ),
    MapListOf(proto::_child0),
    map_insert(
        proto::_data,
        proto::_value(proto::_child1),
        proto::_value(proto::_child2)
    )
)
```

Populate a map from a tree…

Pass extra “data” to your actions, like, say, a std::map.

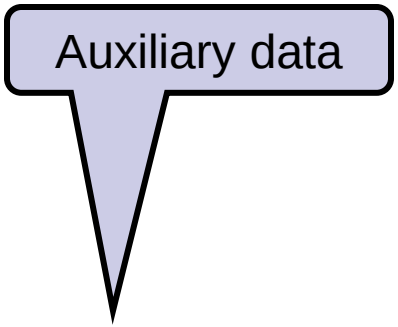
```
proto::case_(
    proto::function(
        MapListOf,
        proto::terminal(_),
        proto::terminal(_)
    ),
    MapListOf(proto::_child0),
    map_insert(
        proto::_data,
        proto::_value(proto::_child1),
        proto::_value(proto::_child2)
    )
)
```

Putting the Pieces Together

```
// Match valid map_list_of expressions and populate a map
struct MapListOf : def<
  match(
    case_(  terminal(map_list_of_),
        void()
    ),
    case_(  function( MapListOf, terminal(_), terminal(_) ),
        MapListOf(_child0),
        map_insert(_data, _value(_child1), _value(_child2))
    )
  )
> {};
```

Using Grammars and Algorithms

```
// Match valid map_list_of expressions and populate a map
struct MapListOf  : /* as before */ {};

int main()
{
    // Use MapListOf as a grammar:
    BOOST_PROTO_ASSERT_MATCHES(
        map_list_of(1,2)(2,3)(3,4)(4,5)(5,6), MapListOf );

    // Use MapListOf as an algorithm:
    std::map< int, int > m;
    MapListOf()( map_list_of(1,2)(2,3)(3,4)(4,5)(5,6), proto::data = m );
    assert( m.size() == 5 );
    assert( m[ 1 ] == 2 );
    assert( m[ 5 ] == 6 );
}
```



Expression Tree Extensibility

Adding members to trees

How to eliminate MapListOf?

```
#include <map>
#include <cassert>
#include <boost/assign/list_of.hpp>
using namespace boost::assign;

int main()
{
    std::map<int,int> m = map_list_of(1,2)(2,3)(3,4)(4,5)(5,6);
    assert( m.size() == 5 );
    assert( m[ 1 ] == 2 );
    assert( m[ 5 ] == 6 );
}
```

This tree must be convertible to a std::map.

User-defined Expressions

```
// Define an expression wrapper that provides a conversion to a map
template<typename ExprDesc>
struct map_list_of_expr  : proto::expr< map_list_of_expr<ExprDesc> >
{
  using proto::expr< map_list_of_expr >::expr;

  template<class K, class V, class C, class A>
  operator std::map<K,V,C,A>() const
  {
    /* … */
  }
};
```

map_list_of_expr< T > is just like expr< T >, except:

- it has a conversion to std::map.
- operations on it produce other map_list_of_expr trees

A Working Expression Extension

```
template<typename ExprDesc>
struct map_list_of_expr  : proto::expr< map_list_of_expr<ExprDesc> >
{
  using proto::expr< map_list_of_expr >::expr;

  template<class K, class V, class C, class A>
  operator std::map<K,V,C,A>() const
  {
    BOOST_PROTO_ASSERT_MATCHES(*this, MapListOf);
    std::map<K,V,C,A> m;
    MapListOf()( *this, m );
    return m;
  }
};
```

```
constexpr map_list_of_expr<proto::terminal(map_list_of_)> map_list_of {};

int main()
{
  std::map<int,int> m0 = map_list_of;                          // OK!
  std::map<int,int> m1 = map_list_of(1,2)(2,3)(3,4)(4,5)(5,6);// OK!
}
```

Proto's Promiscuous Operators

```
template<typename ExprDesc>
struct map_list_of_expr
  : proto::expr< map_list_of_expr<ExprDesc> >
{
    /*… as before… */
};

int main()
{
    map_list_of(1,2) * 32 << map_list_of; // WTF???!!!
}
```

Yeah, but this *still* compiles.

Proto's Promiscuous Operators

```
template<typename ExprDesc>
struct map_list_of_expr
  : proto::expr< map_list_of_expr<ExprDesc>, domain<_,MapListOf > >
{
   /*… as before… */
};

int m
{
   m
}
```

```
main.cpp:75:22: error: invalid operands to binary expression ('declty
      pe(v5::make_expr<v5::function,map_list_of_domain>(static_cast<c
      onst map_list_of_expr<boost::proto::v5::tags::terminal (map_lis
      t_of_)> &>(*this), static_cast<int &&>(u), static_cast<int &&>(
      u)))' (aka 'map_list_of_expr<utility::uncvref<function> (const
      map_list_of_expr<boost::proto::v5::tags::terminal (map_list_of_
      )> &, map_list_of_expr<boost::proto::v5::tags::terminal (int)>,
      map_list_of_expr<boost::proto::v5::tags::terminal (int)>)>') an
      d 'int')
    map_list_of(1,2) * 32 << map_list_of;
    ~~~~~~~~~~~~~~~~ ^  ~~
```

The Complete Solution

```
#include <map>
#include <boost/proto/v5/proto.hpp>
using namespace boost::proto;

struct map_list_of_ {};

struct MapListOf : def<
  match(
    case_(  terminal(map_list_of_),
      void()
    ),
    case_(  function(MapListOf, terminal(_), terminal(_)),
      MapListOf(_child0),
      assign(subscript(_data, _value(_child1)), _value(_child2))
    )
  )
> {};
```

~26 Lines of code

```
template<typename ExprDesc>
struct map_list_of_expr
  : expr<map_list_of_expr<ExprDesc>, domain<_,MapListOf>>
{
    using expr<map_list_of_expr, domain<_,MapListOf>>::expr;

    template< class K, class V, class C, class A>
    operator std::map<K, V, C, A> () const
    {
        std::map<K, V, C, A> map;
        MapListOf()(*this, data = map);
        return map;
    }
};

constexpr map_list_of_expr<terminal(map_list_of_)> map_list_of {};

int main()
{
    std::map<int, int> next = map_list_of(1,2)(2,3)(3,4)(4,5)(5,6);
}
```

map_list_of: Take-Away

- Proto is useful even for small problems
- It makes your code:
  - short
  - declarative
  - efficient

Proto and C++11

C++11 and advanced library design

C++11 Features Used by Proto

Rvalue references

Rvalue references for *this

Variadic templates

Initializer lists

Static assertions

auto-typed variables

Decltype

Default template arguments for function templates

SFINAE for expressions

Uniform initialization syntax

Template aliases

Generalized constant expressions

Delegating constructors

Inheriting constructors

User-defined literals

Literal class types

Defaulted and deleted functions

Allowing move constructors to throw [noexcept]

Inline namespaces

Alternative function syntax

Better errors

Static initialization

Library versioning

Library Versioning with Inline Namespaces

Goal: Use Proto.v4 and Proto.v5 side-by-side

Proto v4

```
#ifdef PROTO_HAS_INLINE_NS
# if PROTO_VER == 4
#  define PROTO_VER_NS_BEG inline namespace v4 {
# else
#  define PROTO_VER_NS_BEG namespace v4 {
# endif
# define PROTO_VER_NS_END }
#else
# define PROTO_VER_NS_BEG
# define PROTO_VER_NS_END
#endif

namespace boost { namespace proto {
PROTO_VER_NS_BEG
   /* … */
PROTO_VER_NS_END
} }
```

Proto v5

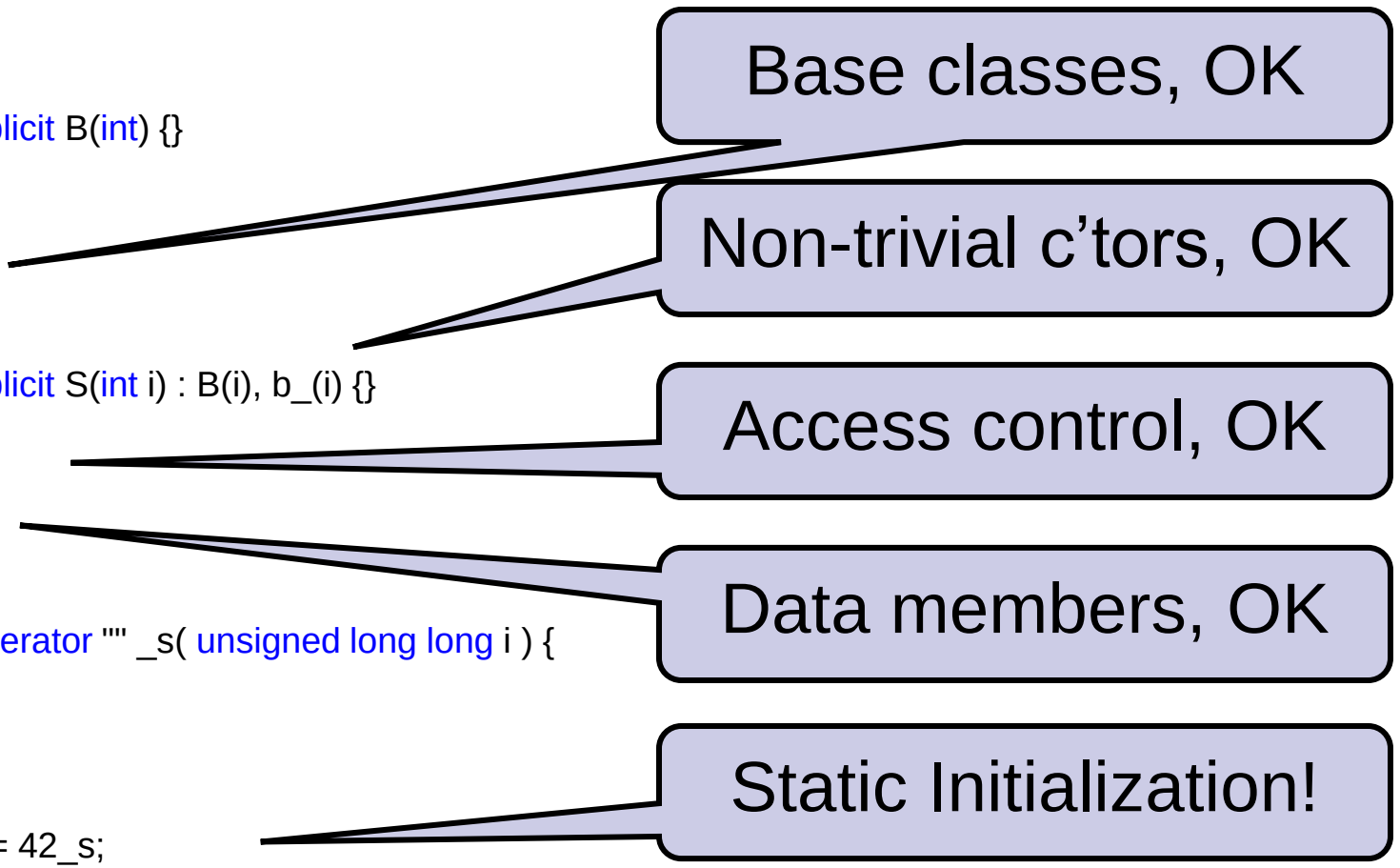
```
#if PROTO_VER == 5
# define PROTO_INLINE_NS inline
#else
# define PROTO_INLINE_NS
#endif

namespace boost { namespace proto {
  PROTO_INLINE_NS namespace v5
  {
    /* … */
  }
}}
```

Static Initialization

- Goal: No initialization order problems with global constants.
- Definition: *Literal Class Types*
  - Trivial copy/move c'tors and d'tor
  - At least one constexpr c'tor (not copy/move)
  - All bases and members are literal

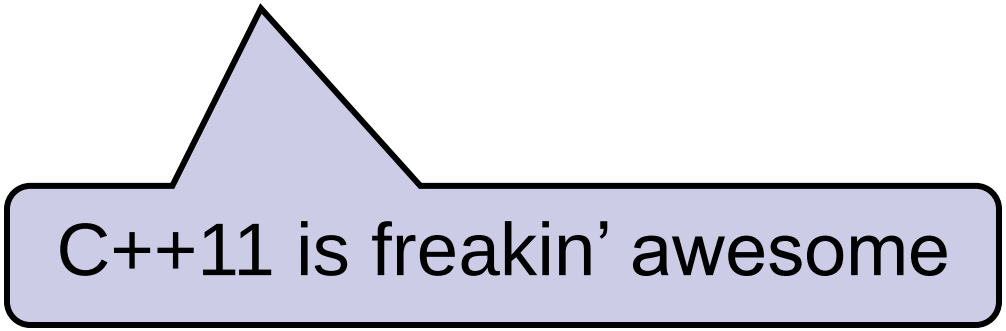
Literal Class Types: Example

```
struct B {
  B() {}
  constexpr explicit B(int) {}
};

struct S : B {
  S() {}
  constexpr explicit S(int i) : B(i), b_(i) {}
private:
  B b_;
};

constexpr S operator "" _s( unsigned long long i ) {
  return S(i);
}

constexpr S s = 42_s;
```



Static Initialization in Proto

```
// Proto's built-in expression type is statically-initialize-able
constexpr expr<terminal(map_list_of_)> map_list_of { };

// User-defined expression type get the same treatment
constexpr map_list_of_expr<terminal(map_list_of_)> map_list_of { };

// All of Proto's operator overloads are also constexpr
constexpr auto omg_wtf_srsly = map_list_of(1,2)(2,3)(3,4)(4,5)(5,6);
```



Stupid constexpr Tricks

```
constexpr auto omg = map_list_of(1,2)(2,3)(3,4)(4,5)(5,6);

// square all int terminals, just 'cause
struct Square : def<
  everywhere(
    case_( terminal(int),
      terminal(multiplies(_value, _value))
    )
  )
> {};

constexpr auto sq = Square()(omg);
static_assert(value(child<1>(child<0>(sq))) == 4*4, "whoa");
```

Run arbitrary Proto algorithms *at compile time*

Better Template Error Msgs

- Goals:
  - Shorter, more readable template error messages.
  - Keep leakage of Proto's implementation details to a minimum.

The Enemy: SFINAE for exprs

```
#define RETURN(…) -> \
  decltype(__VA_ARGS__) { return __VA_ARGS__; }

struct S0 {
   template<typename T>
   auto operator()(T t) const RETURN( t + 1 )
};

struct S1 {
   template<typename T>
   auto operator()(T t) const RETURN( S0()(t) )
};

struct S2 {
   template<typename T>
   auto operator()(T t) const RETURN( S1()(t) )
};

int main() {
   auto x = S2()( std::string("huh?") );
}
```

```
main.cpp:41:14: error: no matching function for call to object of type 'S2'
    auto x = S2()( std::string("foo") );
             ^~~~
main.cpp:32:10: note: candidate template ignored: substitution failure
      [with T = foo]: no matching function for call to object of type 'S1'
    auto operator()(T t) const RETURN( S1()(t) )
         ^                             ~~
1 error generated.
```

The Solution: SFINAE for exprs

```
template<typename Sig>
struct SUBSTITUTION_FAILURE;

template<typename Fun, typename...Args>
struct SUBSTITUTION_FAILURE<Fun(Args...)> {
  virtual void what(Args&&…args) {
    Fun()( std::forward<Args>(args)... );
  }
};

template<typename Fun>
struct try_call {
  template<typename...Args>
  auto operator()(Args&&...args) const
    RETURN( Fun()( std::forward<Args>(args)... ) )

  template<typename...Args>
  SUBSTITUTION_FAILURE<Fun(Args...)>
  operator()(Args&&...) const volatile;
};
```

A function obj wrapper. It either calls the function and returns the result, or returns "SUBSTITUTION_FAILURE"

The Solution: SFINAE for exprs

```
struct S0 {
    template<typename T>
    auto operator()(T t) const RETURN( t + 1 ) // line 38
};

struct S1 {
    template<typename T>
    auto operator()(T t) const RETURN( try_call<S0>()(t) )
};
```

```
struct S2 {
    template<typename T>
    auto operator()(T t) const RETURN( try_call<S1>()(t) )
};

int main() {
    auto x = S2()( std::string("huh?") );
}
```

The Solution: SFINAE for exprs

```
struct S0 {
   template<typename T>
   auto operator()(T t) const RETURN( t + 1 ) // line 38
};

struct S1 {
   template<typename T>
   auto operator()(T t) const RETURN( try_call<S0>()(t) )
};
```

```
struct S2 {
   template<typename T>
   auto operator()(T t) const RETURN( try_call<S1>()(t) )
};

int main() {
   auto x = S2()( std::string("huh?") );
}
```

```
main.cpp:20:40: error: no matching function for call to object of type 'S0'
        Fun()(std::forward<Args>(args)...);
        ^~~~~
main.cpp:32:13: note: in instantiation of member function 'SUBSTITUTION_FAILURE<
S0(std::basic_string<char> &)>::what' requested here
    auto x = S2()(std::string("huh?"));
             ^
main.cpp:38:8: note: candidate template ignored: substitution failure [with T =
      std::basic_string<char>]: invalid operands to binary expression
('std::basic_string<char>' and 'int')
  auto operator()(T t) const RETURN( t + 1 )
       ^                             ~
1 error generated.
```

C++11 and Proto Take-Away

- Proto v5 uses C++11 to work better and report errors better
- C++11 solves many common problems in library design

WE WANT YOU

- … to try Proto v5!
- https://github.com/ericniebler/proto-0x

Questions?