The Context

- Allows functions to efficiently access data from other stack frames
  - Caller sets up a Context
  - Callee retrieve the Context as needed
- On demand (pull vs. push)
- Data can be polymorphic
- Efficient alternative to passing arguments to functions
- Data can cross multiple stack frames
- Allows multiple contexts to be linked up

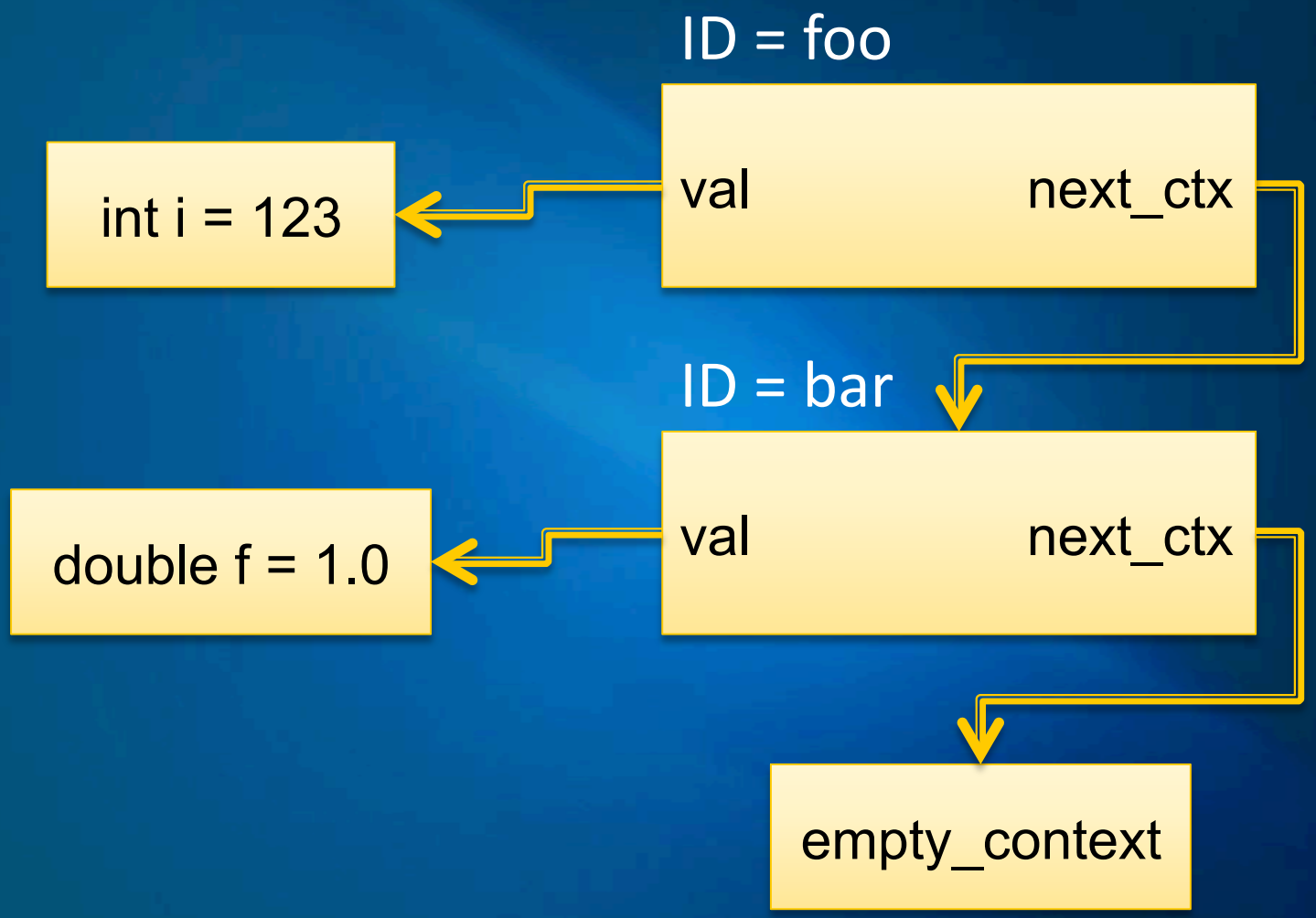
The Context

```
template <typename ID, typename T, typename NextContext>
struct context
{
    context(T const& val, NextContext const& next_ctx)
        : val(val), next_ctx(next_ctx) {}

    T const& get(mpl::identity<ID>) const
    {
        return val;
    }

    template <typename Identity>
    decltype(std::declval<NextContext>().get(Identity()))
    get(Identity id) const
    {
        return next_ctx.get(id);
    }

    T const& val;
    NextContext const& next_ctx;
};
```

The Empty Context

```
struct empty_context
{
   struct undefined {};
   template <typename ID>
   undefined get(ID) const
   {
      return undefined();
   }
};
```

The Context


ID = foo
val
next_ctx
int i = 123
ID = bar
val
next_ctx
double f = 1.0
empty_context

Example Context Usage

```
struct foo_id;

template <typename Context>
void bar(Context const& ctx)
{
      std::cout << ctx.get(mpl::identity<foo_id>()) << std::endl;
}

void foo()
{
   int i = 123;
   empty_context empty_ctx;
   context<foo_id , int, empty_context> ctx(i, empty_ctx);
   bar(ctx);
}
```

Example Context Usage

```
struct foo_id;

template <typename Context>
void bar(Context const& ctx)
{
      std::cout << ctx.get(mpl::identity<foo_id>()) << std::endl;
}

void foo()
{
   int i = 123;
   empty_context empty_ctx;
   context<foo_id , int, empty_context> ctx(i, empty_ctx);
   bar(ctx);
}
```

The Rule Definition

```
template <typename ID, typename RHS>
struct rule_definition : parser<rule_definition<ID, RHS>>
{
    rule_definition(RHS rhs)
        : rhs(rhs) {}

    template <typename Iterator, typename Context>
    bool parse(Iterator& first, Iterator last, Context const& ctx) const
    {
        context<ID, RHS, Context> this_ctx(rhs,  ctx);
        return rhs.parse(first, last, this_ctx);
    }

    RHS rhs;
};
```

The Rule

```
template <typename ID>
struct rule : parser<rule<ID>>
{
    template <typename Derived>
    rule_definition<ID, Derived>
    operator=(parser<Derived> const& definition) const
    {
        return rule_definition<ID, Derived>(definition.derived());
    }

    template <typename Iterator, typename Context>
    bool parse(Iterator& first, Iterator last, Context const& ctx) const
    {
        return ctx.get(mpl::identity<ID>()).parse(first, last, ctx);
    }
};
```

The main parse function

```
template <typename Iterator, typename Derived>
inline bool parse(parser<Derived> const& p, Iterator& first, Iterator last)
{
    empty_context ctx;
    return p.derived().parse(first, last, ctx);
}
```

Our Recursive Rule X3 style

```
rule<class x> const x;
auto const ax = char_('a') >> x;
auto const start =
    x = char_('x') | ax;
```

Encapsulating a Grammar

```
namespace parser
{
  namespace g_definition
  {
    rule<class x> const x;
    auto const ax = char_('a') >> x;

    auto const g =
      x = char_('x') | ax;
  }
  using g_definition::g;
}
```

Walk-through Spirit X3

- Basic Parsers
  - Eps Parser
  - Int Parser
- Composite Parsers
  - Kleene Parser
  - Sequence Parser
  - Alternative Parser
- Nonterminals
  - Rule
  - Grammar
- Semantic Actions

Eps Parser

```
struct eps_parser : parser<eps_parser>
{
    typedef unused_type attribute_type;
    static bool const has_attribute = false;

    template <typename Iterator, typename Context, typename Attribute>
    bool parse(Iterator& first, Iterator const& last
      , Context const& context, Attribute& /*attr*/) const
    {
        x3::skip_over(first, last, context);
        return true;
    }
};
```

Attributes

- Parsers expose an attribute specific to their type
  - int_ → int
  - char_ → char
  - *int_ → std::vector<int>
  - int_ >> char_ → fusion::deque<int, char>
- Some parsers may have *unused* "don't care" attributes
  - literals: e.g. 'z', "hello"
  - eps, eoi, predicates: e.g. !p, &p

Attribute Categories

unused_attribute	unused
plain_attribute	int, char, double
container_attribute	std::vector<int>
tuple_attribute	fusion::list<int, char>
variant_attribute	variant<int, X>
optional_attribute	optional<int>

Attribute Propagation

a >> b

- Attribute Synthesis
  - a -> T, b -> U → (a >> b) -> tuple<T, U>
- Attribute Collapsing
  - a -> T, b ->unused → T
  - a -> unused, b ->U → U
  - a -> unused, b ->unused → unused
- Attribute Compatibility
  - (a >> b) := vector<T> → a := T, b := T
    → a := vector<T>, b := T
    → a := T, b := vector<T>
    → a := vector<T>, b := vector<T>

unused_type

```
struct unused_type
{
    unused_type() {}

    template <typename T>
    unused_type(T const&) {}

    template <typename T>
    unused_type const& operator=(T const&) const { return *this; }

    template <typename T>
    unused_type& operator=(T const&) { return *this; }

    unused_type const& operator=(unused_type const&) const { return *this; }
    unused_type& operator=(unused_type const&) { return *this; }
};
```

The Context Refined

```
template <typename ID, typename T,
    typename Next = unused_type>
struct context
{
    context(T& val, Next const& next)
        : val(val), next(next) {}

    template <typename ID_,
        typename Unused = void>
    struct get_result
    {
        typedef typename Next::template
         get_result<ID_>::type type;
    };

    template <typename Unused>
    struct get_result<mpl::identity<ID>, Unused>
    {
        typedef T& type;
    };

    T& get(mpl::identity<ID>) const
    {
        return val;
    }

    template <typename ID_>
    typename Next::template get_result<ID_>::type
    get(ID_ id) const
    {
        return next.get(id);
    }

    T& val;
    Next const& next;
};
```

The Context Refined

```
// unused_type can also masquerade as an empty context (see context.hpp)

template <typename ID>
struct get_result : mpl::identity<unused_type> {};

template <typename ID>
unused_type get(ID) const
{
    return unused_type();
}
```

skip_over

```
template <typename Iterator, typename Context>
inline void skip_over(
    Iterator& first, Iterator const& last, Context const& context)
{
    detail::skip_over(first, last, spirit::get<skipper_tag>(context));
}


template <typename Iterator, typename Skipper>
inline void skip_over(
    Iterator& first, Iterator const& last, Skipper const& skipper)
{
    while (first != last && skipper.parse(first, last, unused, unused))
        /***/;
}
```

Eps Parser

```
struct eps_parser : parser<eps_parser>
{
    typedef unused_type attribute_type;
    static bool const has_attribute = false;

    template <typename Iterator, typename Context, typename Attribute>
    bool parse(Iterator& first, Iterator const& last
      , Context const& context, Attribute& /*attr*/) const
    {
        x3::skip_over(first, last, context);
        return true;
    }
};

eps_parser const eps = eps_parser();
```

Int Parser

```
template <typename T, unsigned Radix = 10, unsigned MinDigits = 1 , int MaxDigits = -1>
struct int_parser : parser<int_parser<T, Radix, MinDigits, MaxDigits>>
{
   typedef T attribute_type;
   static bool const has_attribute = true;

   template <typename Iterator, typename Context, typename Attribute>
   bool parse(Iterator& first, Iterator const& last
     , Context const& context, Attribute& attr) const
   {
      typedef extract_int<T, Radix, MinDigits, MaxDigits> extract;
      x3::skip_over(first, last, context);
      return extract::call(first, last, attr);
   }
};

int_parser<int> const int_= int_parser<int>();
```

Kleene Parser

```
template <typename Subject>
struct kleene : unary_parser<Subject, kleene<Subject>>
{
   typedef unary_parser<Subject, kleene<Subject>> base_type;
   typedef typename traits::attribute_of<Subject>::type subject_attribute;
   static bool const handles_container = true;

   typedef typename
      traits::build_container<subject_attribute>::type
   attribute_type;

   kleene(Subject const& subject)
     : base_type(subject) {}

   template <typename Iterator, typename Context, typename Attribute>
   bool parse(Iterator& first, Iterator const& last
     , Context const& context, Attribute& attr) const;
};
```

unary_parser

```
template <typename Subject, typename Derived>
struct unary_parser : parser<Derived>
{
    typedef unary_category category;
    typedef Subject subject_type;
    static bool const has_attribute = Subject::has_attribute;
    static bool const has_action = Subject::has_action;

    unary_parser(Subject subject)
      : subject(subject) {}

    unary_parser const& get_unary() const { return *this; }

    Subject subject;
};
```

Kleene ET

```
template <typename Subject>
inline kleene<typename extension::as_parser<Subject>::value_type>
operator*(Subject const& subject)
{
    typedef
        kleene<typename extension::as_parser<Subject>::value_type>
    result_type;

    return result_type(as_parser(subject));
}
```

as_parser

```
namespace extension
{
    template <typename T, typename Enable = void>
    struct as_parser {};
}

template <typename T>
inline typename extension::as_parser<T>::type
as_parser(T const& x)
{
    return extension::as_parser<T>::call(x);
}
```

as_parser

```
template <>
struct as_parser<unused_type>
{
    typedef unused_type type;
    typedef unused_type value_type;
    static type call(unused_type)
    {
        return unused;
    }
};
```

as_parser

```
template <typename Derived>
struct as_parser<Derived
    , typename enable_if<is_base_of<parser_base, Derived>>::type>
{
    typedef Derived const& type;
    typedef Derived value_type;
    static type call(Derived const& p)
    {
        return p;
    }
};
```

as_parser

```
template <>
struct as_parser<char>
{
    typedef literal_char<
        char_encoding::standard, unused_type>
    type;

    typedef type value_type;

    static type call(char ch)
    {
        return type(ch);
    }
};
```

Kleene Parser Implementation

```
template <typename Iterator, typename Context, typename Attribute>
bool parse(Iterator& first, Iterator const& last
  , Context const& context, Attribute& attr) const
{
    while (detail::parse_into_container(
        this->subject, first, last, context, attr))
        ;
    return true;
}
```