Domain specific language embedding with C++ template metaprogramming

Ábel Sinkovics

Outline

- Introduction to C++ template metaprogramming
- DSL integration
- Real world example: typesafe printf
- Summary

C++ template metaprogramming

- Erwin Unruh, 1994
- Turing-complete

- Concept checking
- Expression templates
- DSL embedding

C++ template metaprogramming

C++ source code

Template metaprogram

C++ template metaprogramming

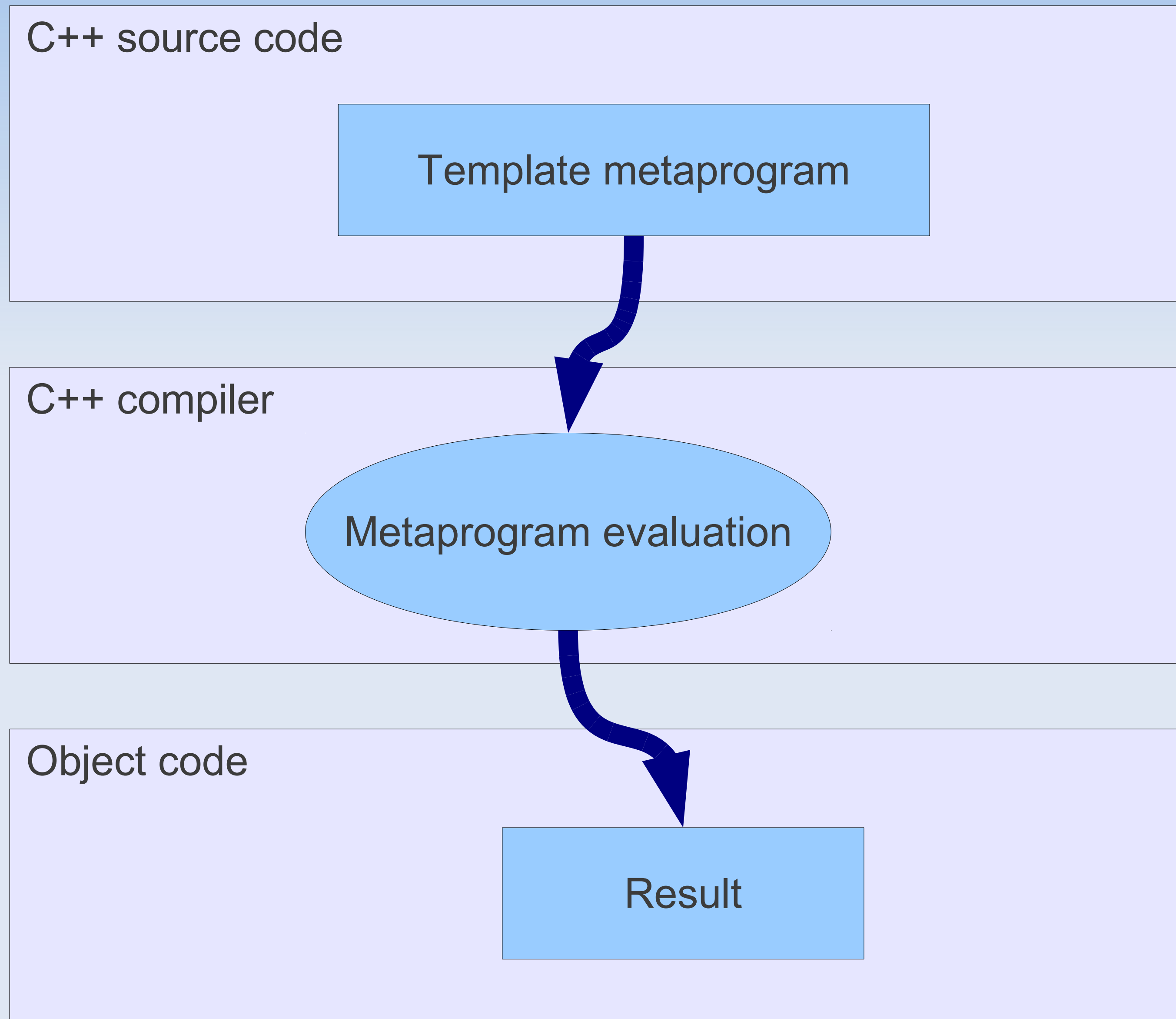
C++ source code

Template metaprogram

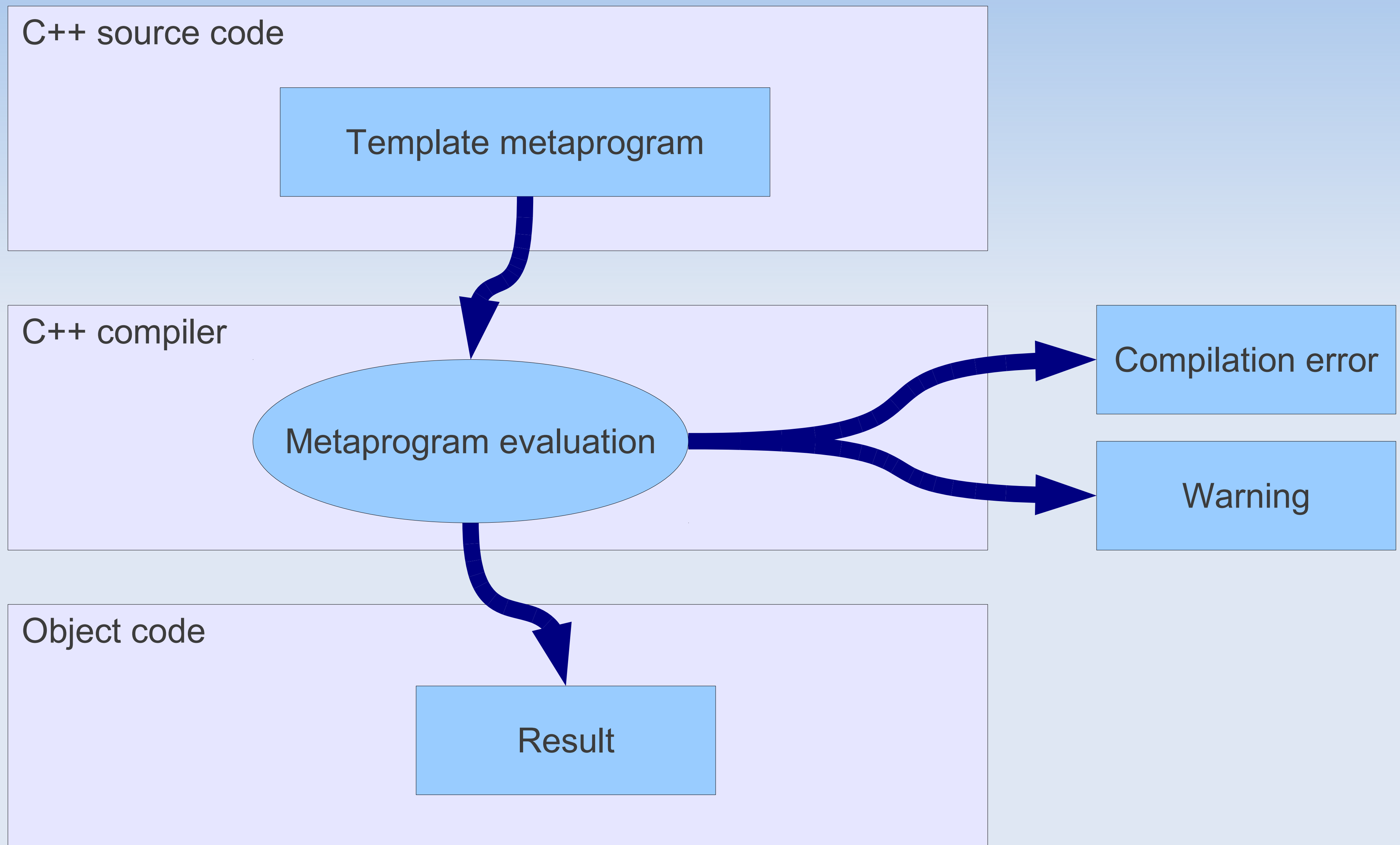
C++ compiler

Metaprogram evaluation

C++ template metaprogramming



C++ template metaprogramming



C++ template metaprogramming

```
template <int n>
struct fact
{
  static const int value =
    n * fact<n - 1>::value;
};
```

C++ template metaprogramming

```
template <int n>
struct fact
{
  static const int value =
    n * fact<n - 1>::value;
};

template <>
struct fact<0>
{
  static const int value =
    1;
};
```

C++ template metaprogramming

```
fact<3>::value
```

```
template <int n>
struct fact
{
  static const int value =
    n * fact<n - 1>::value;
};

template <>
struct fact<0>
{
  static const int value =
    1;
};
```

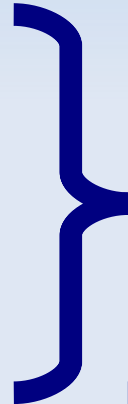
C++ template metaprogramming

fact<3>::value

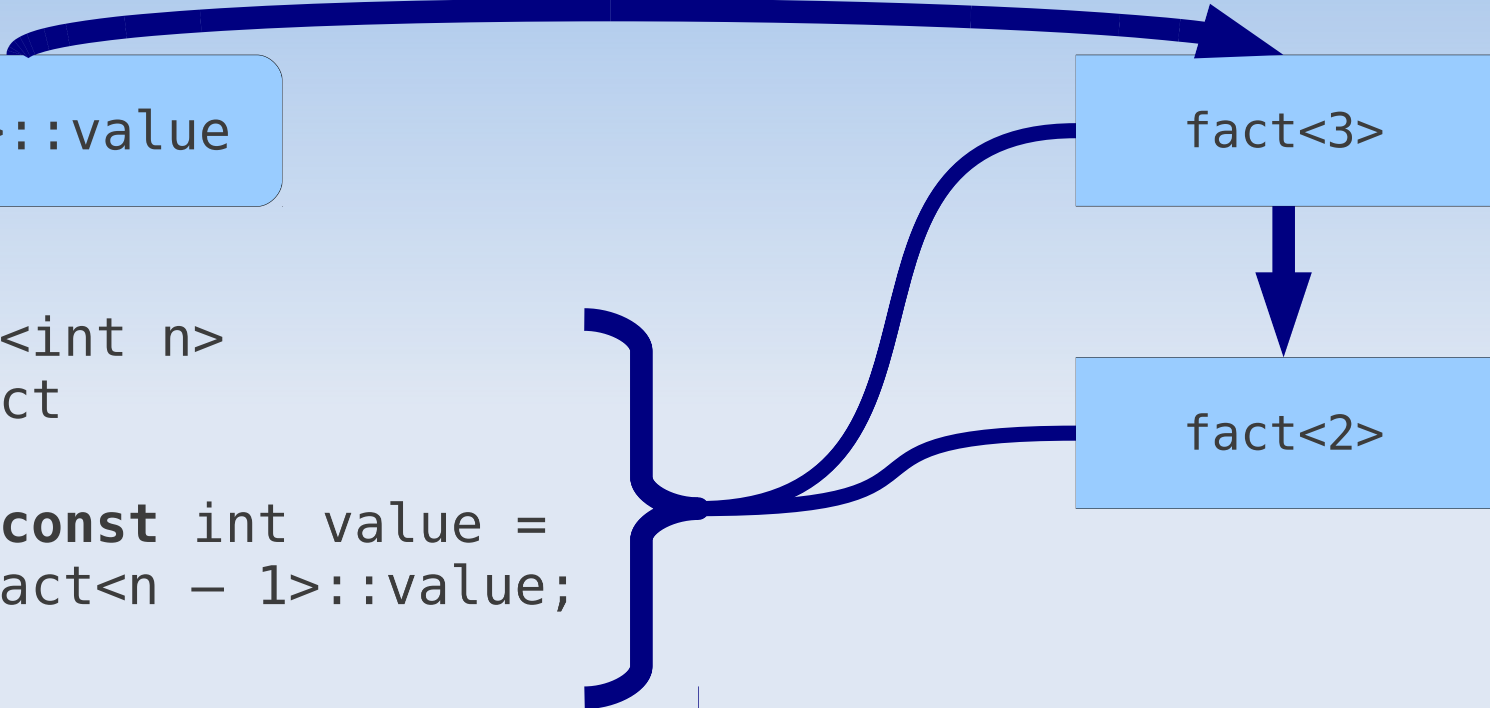
fact<3>

```
template <int n>
struct fact
{
  static const int value =
    n * fact<n – 1>::value;
};

template <>
struct fact<0>
{
  static const int value =
    1;
};
```



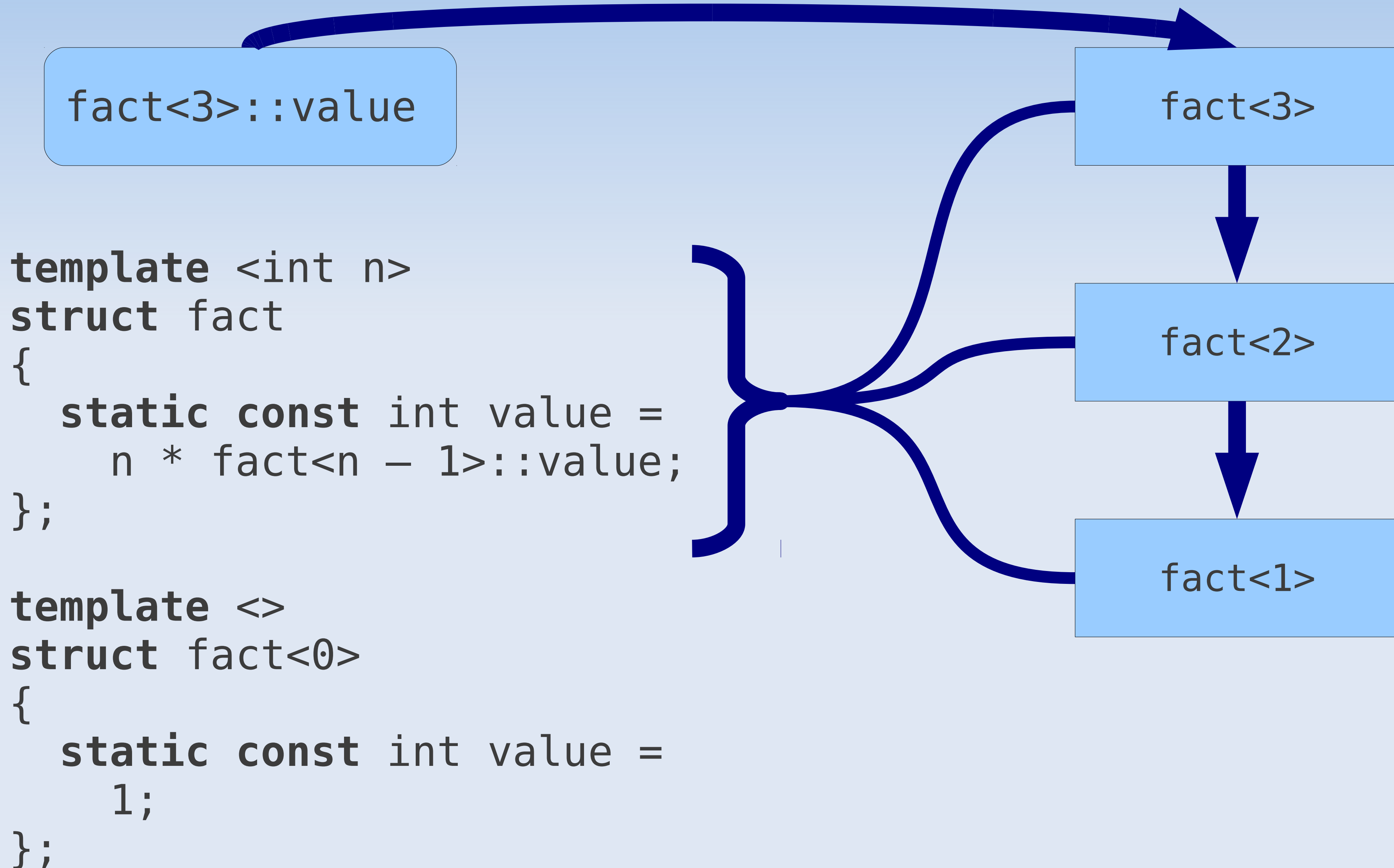
C++ template metaprogramming



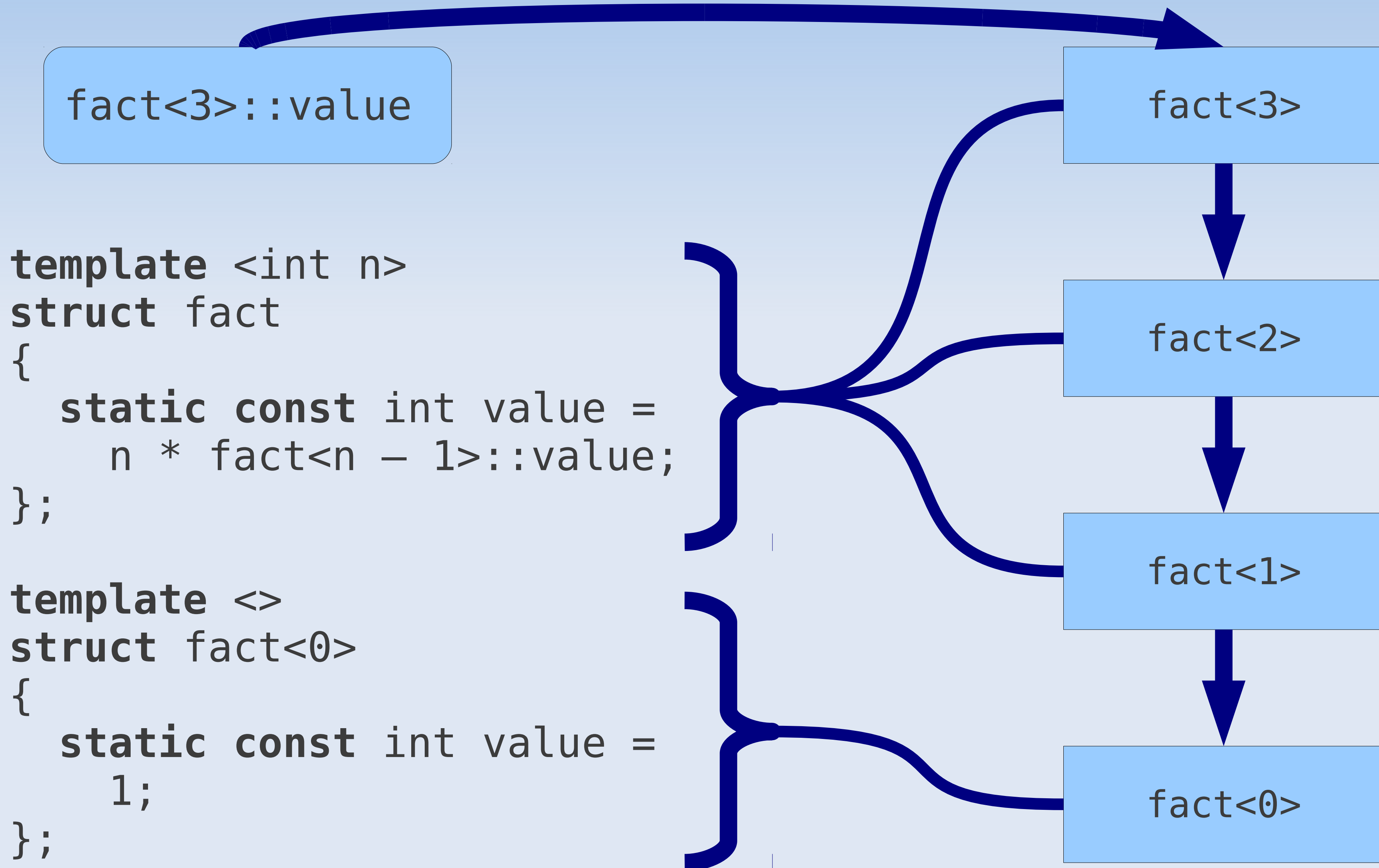

```
template <int n>
struct fact
{
  static const int value =
    n * fact<n – 1>::value;
};

template <>
struct fact<0>
{
  static const int value =
    1;
};
```

C++ template metaprogramming



C++ template metaprogramming



C++ template metafunction

Argument list

Name

Body

C++ template metafunction

```
template <class T>
struct makeConst
{
  typedef const T type;
};
```

Argument list

Name

Body

C++ template metafunction

```
template <class T>
struct makeConst
{
  typedef const T type;
};
```

Argument list

Name

Body

C++ template metafunction

```
template <class T>
struct makeConst
{
  typedef const T type;
};
```

Argument list

Name

Body

```
makeConst<int>::type
```

C++ template metafunction

```
template <class T>
struct makeConst
{
  typedef const T type;
};
```

Argument list

Name

Body

```
makeConst<int>::type
```

Boxing

```
template <int n>
struct int_
{
  static const int value = n;
};
```

Boxing

```
template <int n>
struct int_
{
  static const int value = n;
};
```

```
typedef int_<13> boxed13;
```

Boxing

```
template <int n>
struct int_
{
  static const int value = n;
};
```

```
typedef int_<13> boxed13;
```

```
boxed13::value;
```

Operations on scalars

```
template <class a, class b>
struct times
{

};
```

Operations on scalars

```
template <class a, class b>
struct times
{
                a::value * b::value
};
```

Operations on scalars

```
template <class a, class b>
struct times
{
          int_<a::value * b::value>
};
```

Operations on scalars

```
template <class a, class b>
struct times
{
  typedef int_<a::value * b::value> type;
};
```

Operations on scalars

```
template <class a, class b>
struct times
{
  typedef int_<a::value * b::value> type;
};
```

```
typedef int_<11> boxed11;
typedef int_<13> boxed13;

times<boxed11, boxed13>::type::value;
```

Operations on scalars

```
template <class a, class b>
struct times
{
  typedef int_<a::value * b::value> type;
};
```

```
typedef int_<11> boxed11;
typedef int_<13> boxed13;

times<boxed11, boxed13>::type::value;
```

The real implementation of times is more complicated, it is not covered here.

Writing metafunctions

```
template <class n>
struct doubleNumber
{

};
```

Writing metafunctions

```
template <class n>
struct doubleNumber
{
                   times<int_<2>, n>
};
```

Writing metafunctions

```
template <class n>
struct doubleNumber
{
                   times<int_<2>, n>::type
};
```

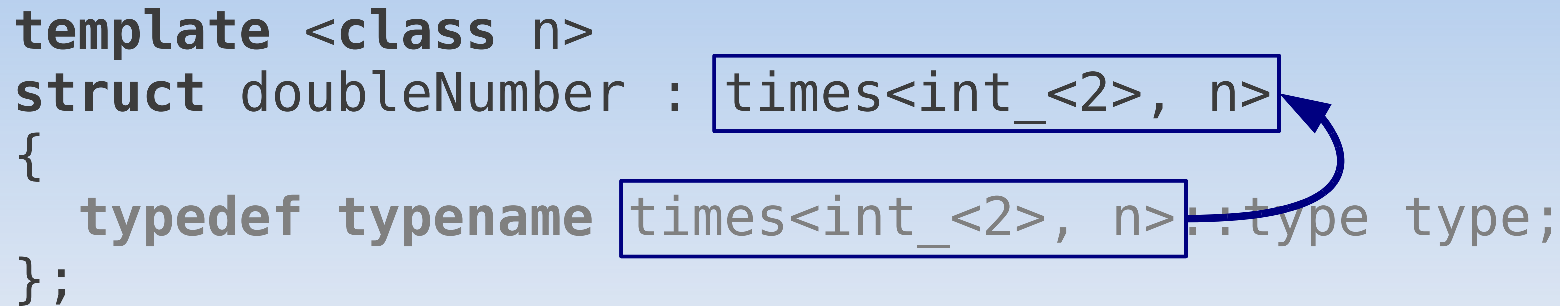
Writing metafunctions

```
template <class n>
struct doubleNumber
{
          typename times<int_<2>, n>::type
};
```

Writing metafunctions

```
template <class n>
struct doubleNumber
{
  typedef typename times<int_<2>, n>::type type;
};
```

Writing metafunctions



```
template <class n>
struct doubleNumber : times<int_<2>, n>
{
  typedef typename times<int_<2>, n>::type type;
};
```

Writing metafunctions

```
template <class n>
struct doubleNumber : times<int_<2>, n> {};
```

Higher order functions

```
template <class T>
struct makeConst
{
  typedef const T type;
};
```

```
makeConst<int>::type
```

Higher order functions

```
struct makeConst
{
  template <class T>
  struct makeConst
  {
    typedef const T type;
  };
};
```

```
makeConst::makeConst<int>::type
```

Higher order functions

```
struct makeConst
{
  template <class T>
  struct apply
  {
    typedef const T type;
  };
};
```

```
makeConst::apply<int>::type
```

Template metafunction class

```
struct makeConst
{
  template <class T>
  struct apply
  {
    typedef const T type;
  };
};
```

Argument list

Name

Body

```
makeConst::apply<int>::type
```

Template metafunction class

```
struct makeConst
{
  template <class T>
  struct apply
  {
    typedef const T type;
  };
};
```

Argument list

Name

Body

```
makeConst::apply<int>::type
```

Higher order functions

```
twice<doubleNumber, int_<13> >::type::value
```

Higher order functions

```
doubleNumber< doubleNumber<int_<13> >        >
```

Higher order functions

```
template <                              >
struct twice

{




};
```

```
twice<doubleNumber, int_<13> >::type::value
```

Higher order functions

```
template <class f, class t>
struct twice

{






};
```

```
twice<doubleNumber, int_<13> >::type::value
```

Higher order functions

```
template <class f, class t>
struct twice

{
  typedef




    type;
};
```

```
twice<doubleNumber, int_<13> >::type::value
```

Higher order functions

```
template <class f, class t>
struct twice

{
  typedef



               f::          apply<t>::type


    type;
};
```

```
twice<doubleNumber, int_<13> >::type::value
```

Higher order functions

```
template <class f, class t>
struct twice

{
  typedef
                f::         apply<

                  f::         apply<t>::type
    >::type
    type;
};
```

```
twice<doubleNumber, int_<13> >::type::value
```

Higher order functions

```
template <class f, class t>
struct twice

{
  typedef
    typename f::template apply<

      typename f::template apply<t>::type
    >::type
    type;
};
```

```
twice<doubleNumber, int_<13> >::type::value
```

Higher order functions

```
template <class f, class t>
struct twice

{
  typedef
    typename f::template apply<

      typename apply<f, t>::type
    >::type
    type;
};
```

```
twice<doubleNumber, int_<13> >::type::value
```

Higher order functions

```
template <class f, class t>
struct twice

{
  typedef
    typename apply<
      f,
      typename apply<f, t>::type
    >::type
    type;
};
```

```
twice<doubleNumber, int_<13> >::type::value
```

Higher order functions

```
template <class f, class t>
struct twice

{
  typedef
    typename apply<
      f,
      typename apply<f, t>::type
    >::type
    type;
};
```

```
twice<doubleNumber, int_<13> >::type::value
```

Higher order functions

```
template <class f, class t>
struct twice

{
  typedef
    typename apply<f, apply<f, t> >::type
    type;
};
```

```
twice<doubleNumber, int_<13> >::type::value
```

Higher order functions

```
template <class f, class t>
struct twice :
               apply<f,            apply<f, t> >
{
  typedef
    typename apply<f, apply<f, t> >::type
    type;
};
```

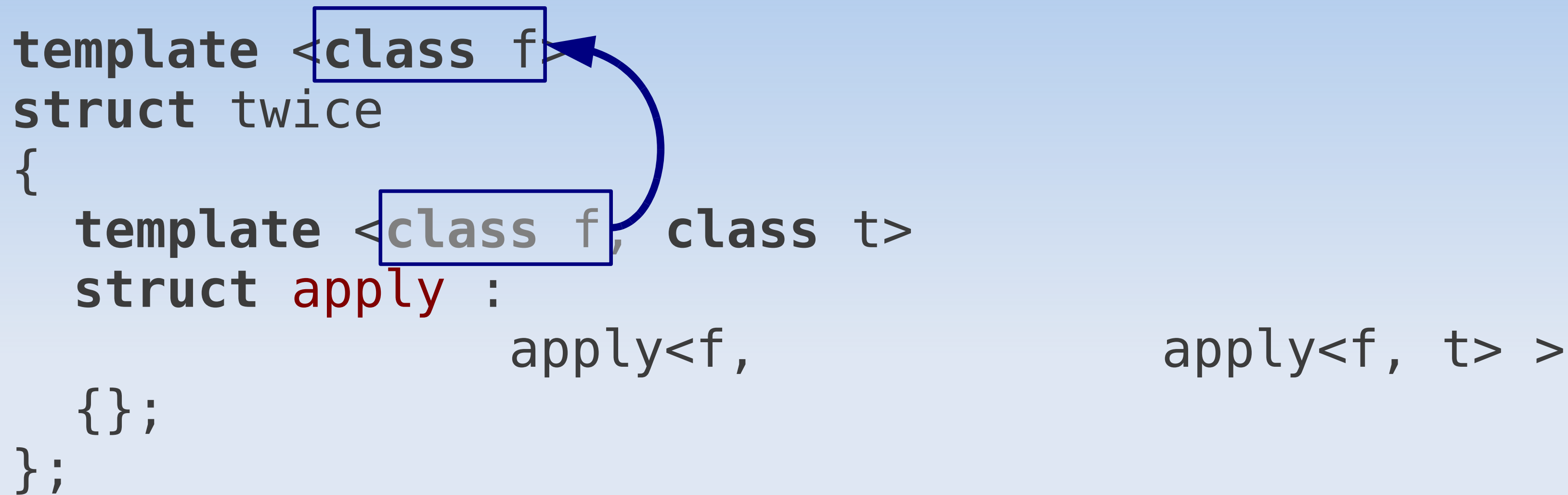
```
twice<doubleNumber, int_<13> >::type::value
```

Higher order functions

```
template <class f, class t>
struct twice :
             apply<f,            apply<f, t> >
{};
```

```
twice<doubleNumber, int_<13> >::type::value
```

Higher order functions




```
twice<doubleNumber>::apply<int_<13> >::type::value
```

Higher order functions

```
template <class f>
struct twice
{
  template <class t>
  struct apply :
    boost::mpl::apply<f, boost::mpl::apply<f, t> >
  {};
};
```

```
twice<doubleNumber>::apply<int_<13> >::type::value
```

Higher order functions

```
template <class f>
struct twice
{
  template <class t>
  struct apply :
    boost::mpl::apply<f, boost::mpl::apply<f, t> >
  {};
};
```

```
apply<twice<doubleNumber>, int_<13> >::type::value
```

boost::mpl

- Design follows STL
- Data structures, iterators, algorithms
- Integral wrappers
- Lambda expressions

DSLs

- Express problems in a particular domain
- Expressive
- Reflecting domain notations
- Embedding into a host language
- C++ examples
  - boost::xpressive
  - boost::proto
  - boost::spirit

Integration techniques

- External frameworks
- Language extensions
- New languages designed for extension
- Generative approach
  - Using the C++ compiler itself
  - No need for external tools
  - DSL interacts with C++ code

Our solution

- Write parsers in C++ template metaprograms
- Parsing happens at compile-time
- The parsed code becomes part of the compiled program
- Imagine spirit ”running” at compile-time

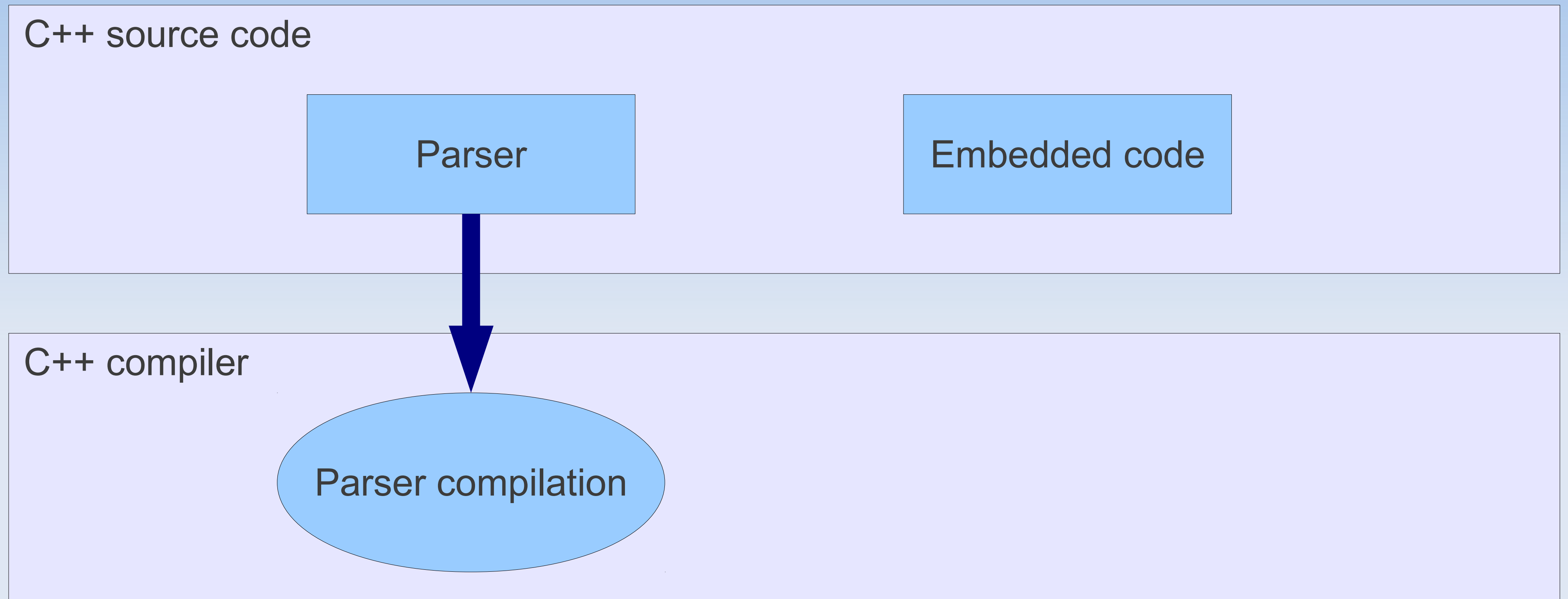
Our solution

C++ source code

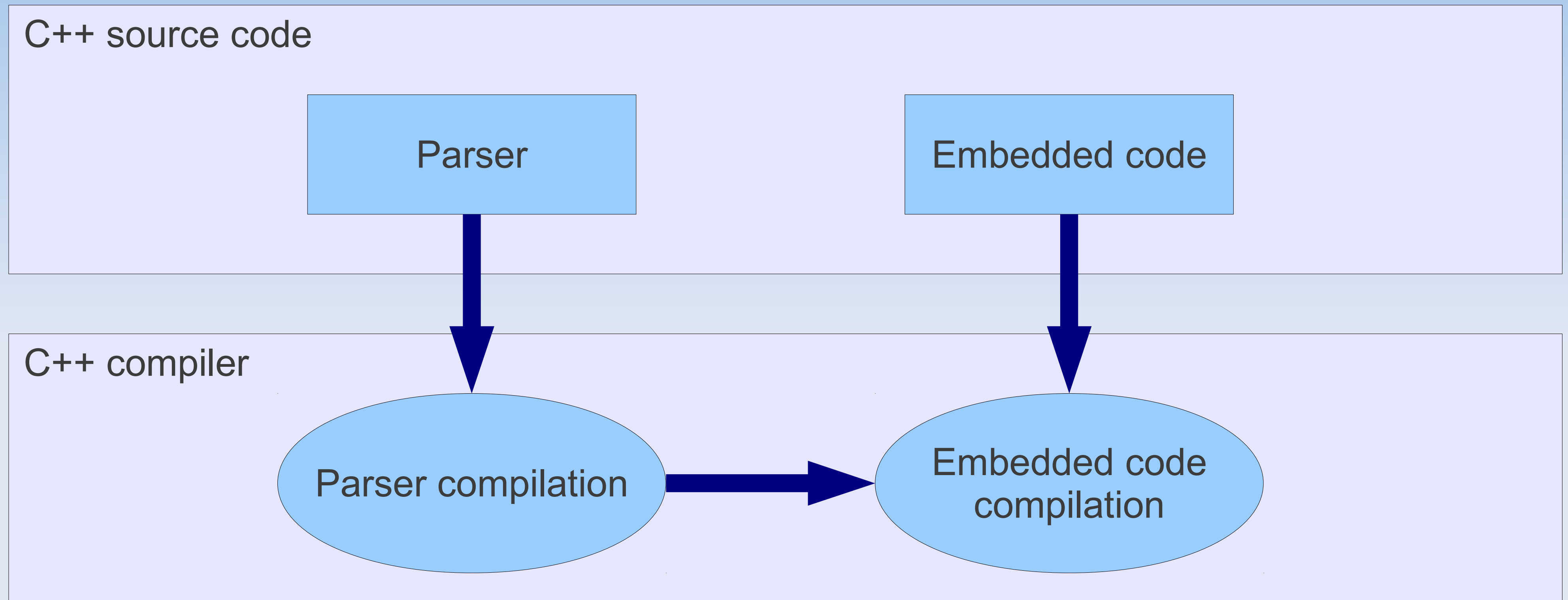
Parser

Embedded code

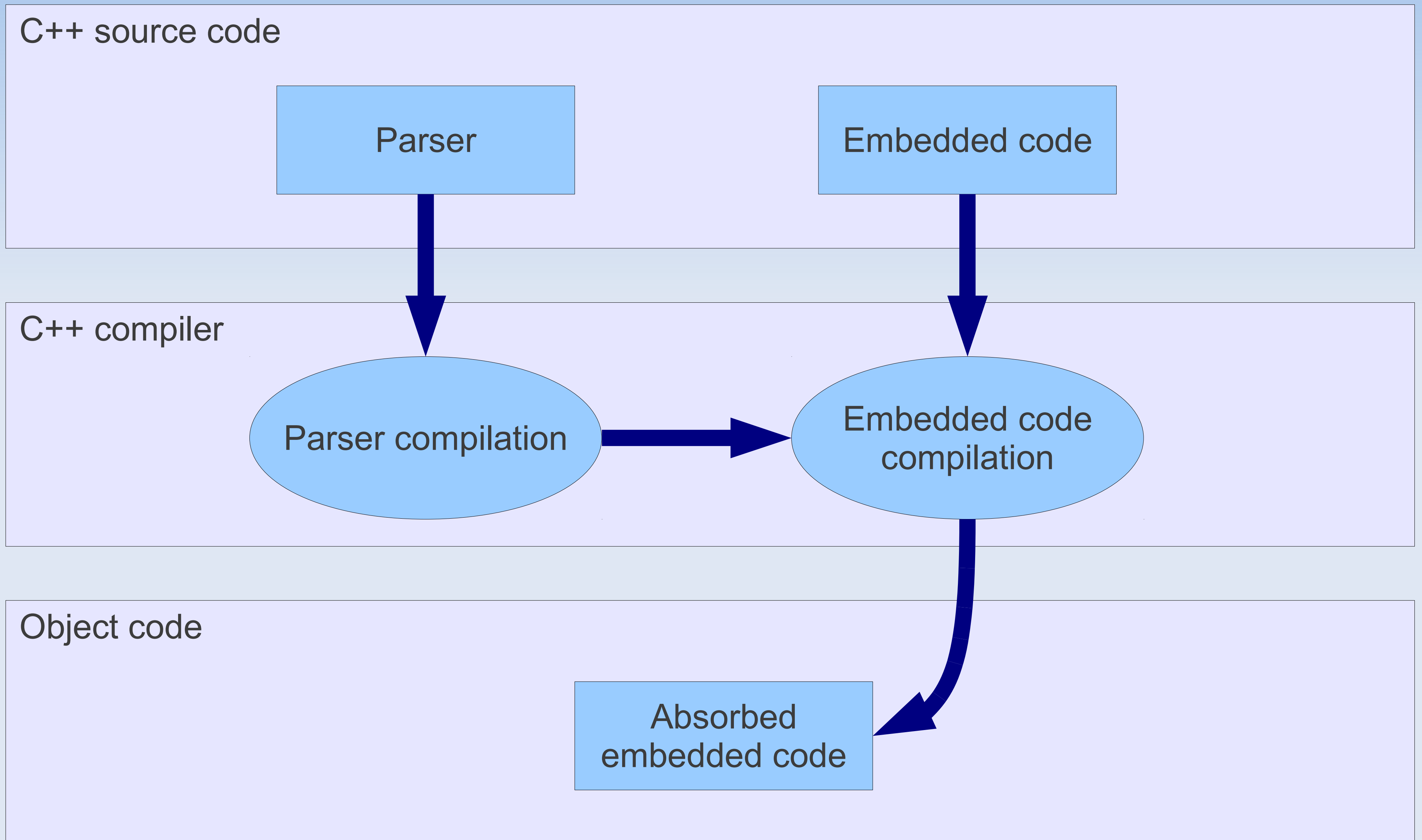
Our solution



Our solution



Our solution



The input of the parsers

- Input of the parser is the embedded source code

```
boost::mpl::string<'Hell','o Wo','rld'>
```

The input of the parsers

- Input of the parser is the embedded source code

```
boost::mpl::string<'Hell','o Wo','rld'>
```

```
_S("Hello World")
```

Building the parser

- Parser generator library
- Strong connection between C++ template metaprogramming and the functional paradigm
- Port a parser generator library written in Haskell
- Based on parser combinators

The translation process

```
type Parser a = String -> Maybe (a, String)
```

The translation process

```
type Parser a = String -> Maybe (a, String)
```

- Template metafunction class
- Takes input string as argument
- Returns a pair of parsed object and remaining string or a special class, nothing

Simple parsers

- return_, fail
- one_char

Simple parsers

- return_, fail
- one_char

```
struct one_char
{
  template <class s>
  struct apply
  {
    typedef
      mpl::pair<
        typename mpl::front<s>::type,
        typename mpl::pop_front<s>::type
      >
      type;
  };
};
```

Parser combinators

- transform
- accept_when
- any, any1
- sequence

Parser combinators

- transform
- accept_when
- any, any1
- sequence

```
typedef accept_when<one_char, is_digit> accept_digit;

typedef any1<digit> accept_digit_sequence;
```

Typesafe printf

```
printf("%d + %d = %d", 7, 6, 13);
```

Typesafe printf

```
printf("%d + %d = %d", 7, 6, 13);
```

```
safe::printf<_S("%d + %d = %d")>(7, 6, 13);
```

Typesafe printf

```
printf("%d + %d = %d", 7, 6, 13);
```

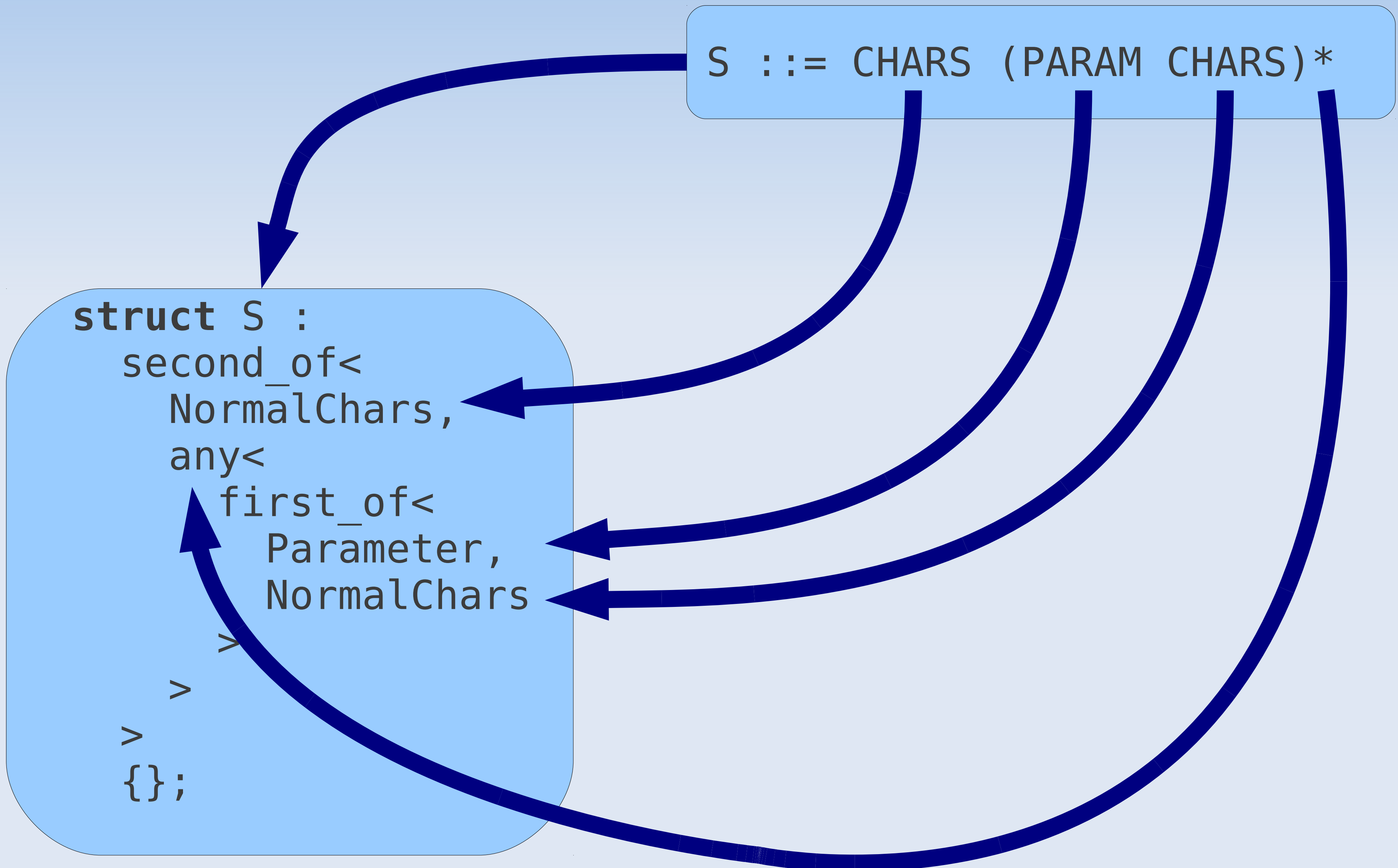
```
safe::printf<_S("%d + %d = %d")>(7, 6, 13);
```

Grammar

```
S ::= CHARS (PARAM CHARS)*
```

```
struct S :
  second_of<
    NormalChars,
    any<
      first_of<
        Parameter,
        NormalChars
      >
    >
  >
  {};
```

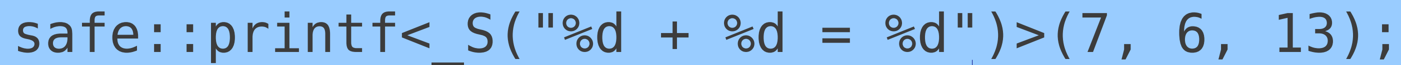
Grammar



Example usage

```
safe::printf<_S("%d + %d = %d")>(7, 6, 13);
```

Example usage



```
mpl::list<int, int, int>
```

Example usage

```
safe::printf<_S("%d + %d = %d")>(7, 6, 13);
```

Compile-time parser

```
mpl::list<int, int, int>
```

=

```
mpl::list<int, int, int>
```

Implementation

```
template <                                   >
int safe_printf(                   )
{




}
```

Implementation

```
template <class S                              >
int safe_printf(                  )
{








}
```

Implementation

```
template <class S                              >
int safe_printf(   a1,    a2,    a3)
{






}
```

Implementation

```
template <class S, class T1, class T2, class T3>
int safe_printf(T1 a1, T2 a2, T3 a3)
{







}
```

Implementation

```
template <class S, class T1, class T2, class T3>
int safe_printf(T1 a1, T2 a2, T3 a3)
{


      typename mpl::apply<printf_grammar, S>::type





}
```

Implementation

```
template <class S, class T1, class T2, class T3>
int safe_printf(T1 a1, T2 a2, T3 a3)
{


      typename mpl::apply<printf_grammar, S>::type
      mpl::list<T1, T2, T3>



}
```

Implementation

```
template <class S, class T1, class T2, class T3>
int safe_printf(T1 a1, T2 a2, T3 a3)
{

    mpllibs::test::equal_sequence<
      typename mpl::apply<printf_grammar, S>::type,
      mpl::list<T1, T2, T3>
    >


}
```

Implementation

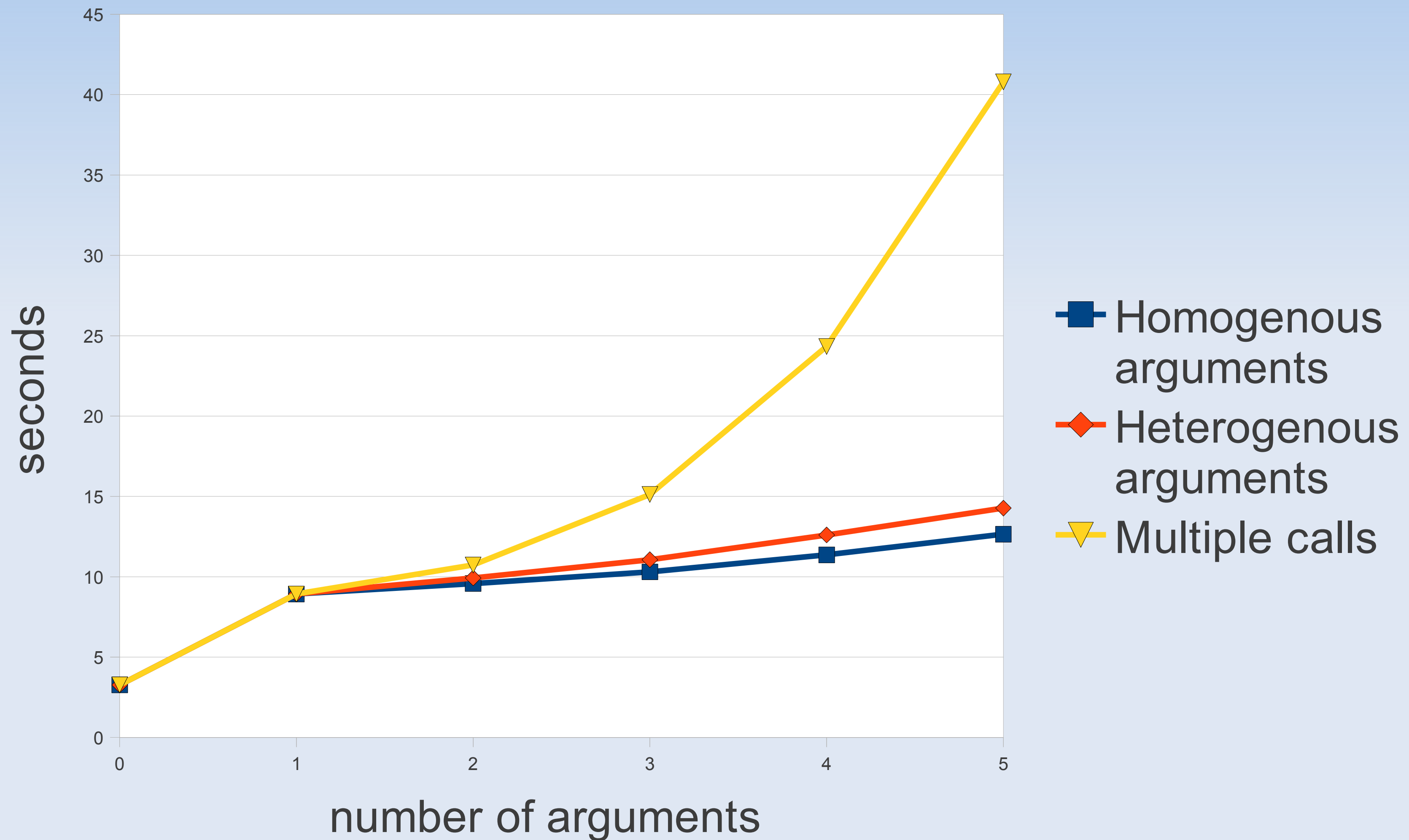
```
template <class S, class T1, class T2, class T3>
int safe_printf(T1 a1, T2 a2, T3 a3)
{
  BOOST_STATIC_ASSERT(
    mpllibs::test::equal_sequence<
      typename mpl::apply<printf_grammar, S>::type,
      mpl::list<T1, T2, T3>
    >
  );

}
```

Implementation

```
template <class S, class T1, class T2, class T3>
int safe_printf(T1 a1, T2 a2, T3 a3)
{
  BOOST_STATIC_ASSERT(
    mpllibs::test::equal_sequence<
      typename mpl::apply<printf_grammar, S>::type,
      mpl::list<T1, T2, T3>
    >
  );
  return printf(mpl::c_str<S>::type::value, a1, a2, a3);
}
```

Performance



Conclusion

- Embedded languages are often used
- Integration of an embedded language is difficult
- The C++ compiler can be utilised to parse the embedded language
- This has a heavy cost at compile time
- It has no cost at runtime

Long term plans

- C++ template metaprograms are not maintenable
- They follow the functional paradigm
- They should be written in a high-level functional language
- C++ source code should remain assembly of metaprograms
- We are planning to compile Haskell code to template metaprograms using this tool

Q & A

http://abel.web.elte.hu/mpllibs/