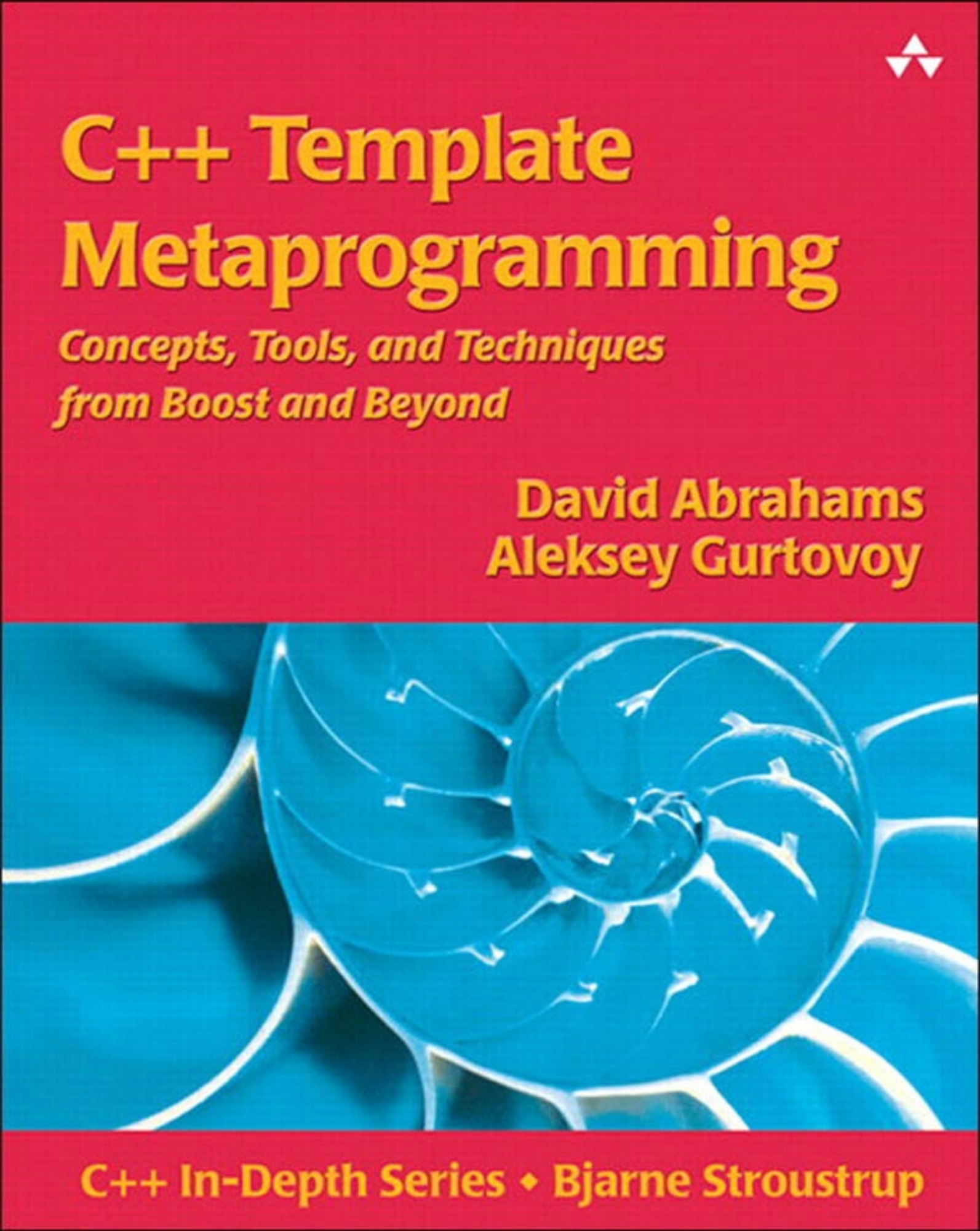
C++ Template Metaprogramming
Concepts, Tools, and Techniques from Boost and Beyond
David Abrahams
Aleksey Gurtovoy
C++ In-Depth Series • Bjarne Stroustrup

C++ Template Metaprogramming

```
    struct deref< tiny_iterator<Tiny,Pos> >
      : at<Tiny,Pos>
    {
    };

}}
```

The only thing missing now are constant-time specializations of `mpl::advance` and `mpl::distance` metafunctions:

```
namespace boost { namespace mpl {

    // random access iterator requirements
    template <class Tiny, class Pos, class N>
    struct advance<tiny_iterator<Tiny,Pos>,N>
    {
        typedef tiny_iterator<
             Tiny
           , typename mpl::plus<Pos,N>::type
        > type;
     };

     template <class Tiny, class Pos1, class Pos2>
     struct distance<
         tiny_iterator<Tiny,Pos1>
       , tiny_iterator<Tiny,Pos2>
     >
       : mpl::minus<Pos2,Pos1>
     {};

}}
```

Note that we've left the job of checking for usage errors to you in exercise 5-0.

5.11.5 `begin` and `end`

Finally, we're ready to make `tiny` into a real sequence; all that remains is to supply `begin` and `end`. Like `mpl::at`, `mpl::begin` and `mpl::end` use traits to isolate the implementation for a particular family of sequences. Writing our `begin`, then, is straightforward:

```
namespace boost { namespace mpl {

   template <>
   struct begin_impl<tiny_tag>
   {
       template <class Tiny>
       struct apply
       {
            typedef tiny_iterator<Tiny,int_<0> > type;
       };
   };
}}
```

Writing `end` is a little more complicated than writing `begin` was, since we'll need to deduce the sequence length based on the number of `none` elements. One straightforward approach might be:

```
namespace boost { namespace mpl {

   template <>
   struct end_impl<tiny_tag>
   {
       template <class Tiny>
       struct apply
         : eval_if<
               is_same<none,typename Tiny::t0>
             , int_<0>
             , eval_if<
                   is_same<none,typename Tiny::t1>
                 , int_<1>
                 , eval_if<
                       is_same<none,typename Tiny::t2>
                     , int_<2>
                     , int_<3>
                   >
               >
           >
       {};
   };
}}
```

Unfortunately, that code doesn't satisfy the O(1) complexity requirements of `end`: It costs O(N) template instantiations for a sequence of length N, since `eval_if`/`is_same` pairs will be instantiated

until a `none` element is found. To find the size of the sequence in constant time, we need only write a few partial specializations:

```
template <class T0, class T1, class T2>
struct tiny_size
  : mpl::int_<3> {};

template <class T0, class T1>
struct tiny_size<T0,T1,none>
  : mpl::int_<2> {};

template <class T0>
struct tiny_size<T0,none,none>
  : mpl::int_<1> {};

template <>
struct tiny_size<none,none,none>
  : mpl::int_<0> {};

namespace boost { namespace mpl {
   template <>
   struct end_impl<tiny_tag>
   {
       template <class Tiny>
       struct apply
       {
           typedef tiny_iterator<
               Tiny
             , typename tiny_size<
                   typename Tiny::t0
                 , typename Tiny::t1
                 , typename Tiny::t2
               >::type
           >
           type;
       };
   };
}}
```

Here, each successive specialization of `tiny_size` is "more specialized" than the previous one, and only the appropriate version will be instantiated for any given `tiny` sequence. The best-matching `tiny_size` specialization will always correspond directly to the length of the sequence.

If you're a little uncomfortable (or even peeved) at the amount of boilerplate code repetition here, we can't blame you. After all, didn't we promise that metaprogramming would help save us from all that? Well, yes we did. We have two answers for you. First, metaprogramming libraries save their *users* from repeating themselves, but once you start writing new sequences you're now working at the level of a library designer.[8] Your users will thank you for going to the trouble (even if they're just you!). Second, as we hinted earlier, there are other ways to automate code generation. You'll see how even the library designer can be spared the embarrassment of repeating herself in Appendix A.

It's so easy to do at this point, that we may as well implement a specialized `mpl::size`. It's entirely optional; MPL's default implementation of `size` just measures the distance between our `begin` and `end` iterators, but since we are going for efficiency, we can save a few more template instantiations by writing our own:

```
namespace boost { namespace mpl {
   template <>
   struct size_impl<tiny_tag>
   {
       template <class Tiny>
       struct apply
         : tiny_size<
              typename Tiny::t0
            , typename Tiny::t1
            , typename Tiny::t2
           >
       {};
   };
}}
```

You've probably caught on by now that the same tag-dispatching technique keeps cropping up over and over. In fact, it's used for all of the MPL's intrinsic sequence operations, so you can always take advantage of it to customize any of them for your own sequence types.

5.11.6 Adding Extensibility

In this section we'll write some of the operations required for `tiny` to fulfill the *Extensible Sequence* requirements. We won't show you all of them because they are so similar in spirit. Besides, we need to leave something for the exercises at the end of the chapter!

8. This need for repetition, at least at the metaprogramming library level, seems to be a peculiarity of C++. Most other languages that support metaprogramming don't suffer from the same limitation, probably because their metaprogramming capabilities are more than just a lucky accident.

First let's tackle `clear` and `push_front`. It's illegal to call `push_front` on a full `tiny`, because our `tiny` sequence has a fixed capacity. Therefore, any valid `tiny<T0,T1,T2>` passed as a first argument to `push_front` must always have length <= 2 and `T2 = none`, and it's okay to just drop `T2` off the end of the sequence:[9]

```
namespace boost { namespace mpl {
   template <>
   struct clear_impl<tiny_tag>
   {
       template <class Tiny>
       struct apply : tiny<>
       {};
   };
   template <>
   struct push_front_impl<tiny_tag>
   {
       template <class Tiny, class T>
       struct apply
         : tiny<T, typename Tiny::t0, typename Tiny::t1>
       {};
   };
}}
```

That was easy! Note that because every `tiny` sequence is a metafunction returning itself, we were able to take advantage of metafunction forwarding in the body of `apply`.

> **Recommendation**
>
> For maximum MPL interoperability, when writing a class template that isn't already a metafunction, consider making it one by adding a nested `::type` that refers to the class itself. When writing a metafunction that will always return a class type, consider deriving it from that class and having the metafunction return itself.

Writing `push_back` isn't going to be such a cakewalk: The transformation we apply depends on the length of the input sequence. Not to worry; we've already written one operation whose implementation depended on the length of the input sequence: `end`. Since we have the length computation conveniently at hand, all we need is a `tiny_push_back` template, specialized for each sequence length:

9. Actually *enforcing* our assumption that the sequence is not full when `push_front` is invoked is left for you as an exercise.

```
template <class Tiny, class T, int N>
struct tiny_push_back;

template <class Tiny, class T>
struct tiny_push_back<Tiny,T,0>
  : tiny<T,none,none>
{};

template <class Tiny, class T>
struct tiny_push_back<Tiny,T,1>
  : tiny<typename Tiny::t0,T,none>
{};

template <class Tiny, class T>
struct tiny_push_back<Tiny,T,2>
  : tiny<typename Tiny::t0,typename Tiny::t1,T>
{};

namespace boost { namespace mpl {
   template <>
   struct push_back_impl<tiny_tag>
   {
       template <class Tiny, class T>
       struct apply
         : tiny_push_back<
             Tiny, T, size<Tiny>::value
           >
       {};
   };
}}
```

Note that what is missing here is just as important as what is present. By not defining a `tiny_push_back` specialization for sequences of length 3, we made it a compile-time error to `push_back` into a full sequence.

5.12 Details

By now you should have a fairly clear understanding of what goes into an MPL sequence—and what comes out of it! In upcoming chapters you can expect to get more exposure to type sequences and their practical applications, but for now we'll just review a few of this chapter's core concepts.