

Appendix A An Introduction to Preprocessor Metaprogramming

Copyright: From “C++ Template Metaprogramming,” by David Abrahams and Aleksey Gurtovoy. Copyright (c) 2005 by Pearson Education, Inc. Reprinted with permission.

ISBN: 0321227255

url: <http://www.awprofessional.com/titles/0321227255>

url: <http://www.boost-consulting.com/mplbook>

A.1 Motivation

Even with the full power of template metaprogramming and the [Boost Metaprogramming library](#) at our disposal, some C++ coding jobs still require a great deal of boilerplate code repetition. We saw one example in Chapter 5, when we implemented `tiny_size`:

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

Aside from the repeated pattern in the parameter list of the primary template above, there are three partial specializations below, which also follow a predictable pattern:

```
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> {};
```

In this case there is only a small amount of code with such a “mechanical” flavor, but had we been implementing `large` instead of `tiny`, there might easily have been a great deal more. When the number of instances of a pattern grows beyond two or three, writing them by hand tends to become error-prone. Perhaps more importantly, the code gets hard to read, because the important abstraction in the code is really the pattern, not the individual instances.

A.1.1 Code Generation

Rather than being written out by hand, mechanical-looking code should really be generated mechanically. Having written a program to spit out instances of the code pattern, a library author has two choices: She can either ship pre-generated source code files, or she can ship the generator itself. Either approach has drawbacks. If clients only get the generated source, they are stuck with whatever the library author generated—and experience shows that if they are happy with three instances of a pattern today, someone will need four tomorrow. If clients get the generator program, on the other hand, they also need the resources to execute it (e.g., interpreters), and they must integrate the generator into their build processes...

A.1.2 Enter the Preprocessor

...unless the generator is a preprocessor metaprogram. Though not designed for that purpose, the C and C++ preprocessors can be made to execute sophisticated programs during the preprocessing phase of compilation. Users can control the code generation process with preprocessor `#defines` in code or `-D` options on the compiler’s command line, making build integration trivial. For example, we might parameterize the primary `tiny_size` template above as follows:

```
#include <boost/preprocessor/repetition/enum_params.hpp>

#ifdef TINY_MAX_SIZE
#   define TINY_MAX_SIZE 3 // default maximum size is 3
#endif

template <BOOST_PP_ENUM_PARAMS(TINY_MAX_SIZE, class T)>
struct tiny_size
    : mpl::int_<TINY_MAX_SIZE>
{};
```

To test the metaprogram, run your compiler in its “preprocessing” mode (usually the `-E` option), with the Boost root directory in your `#include` path. For instance:¹

```
g++ -P -E -Ipath/to/boost_1_32_0 -I. test.cpp
```

Given the appropriate metaprograms, users would be able to adjust not only the number of parameters to `tiny_size`, but the maximum size of the entire `tiny` implementation just by `#define`-ing `TINY_MAX_SIZE`.

The Boost Preprocessor library [MK04] plays a role in preprocessor metaprogramming similar to the one played by the MPL in template metaprogramming: It supplies a framework of high-level components (like `BOOST_PP_ENUM_PARAMS`) that make otherwise-painful metaprogramming jobs approachable. In this appendix we won’t attempt to cover nitty-gritty details of how the preprocessor works, nor principles of preprocessor metaprogramming in general, nor even many details of how the Preprocessor *library* works. We *will* show you enough at a high level that you’ll be able to use the library productively and learn the rest on your own.

A.2 Fundamental Abstractions of the Preprocessor

We began our discussion of template metaprogramming in Chapter 2 by describing its metadata (potential template arguments) and metafunctions (class templates). On the basis of those two fundamental abstractions, we built up the entire picture of compile-time computation covered in the rest of this book. In this section we’ll lay a similar foundation for the preprocessor metaprogrammer. Some of what we cover here may be a review for you, but it’s important to identify the basic concepts going into detail.

A.2.1 Preprocessing Tokens

The fundamental unit of data in the preprocessor is the **preprocessing token**. Preprocessing tokens correspond roughly to the tokens you’re used to working with in C++, such as identifiers, operator symbols, and literals. Technically, there are some differences between *preprocessing tokens* and regular *tokens* (see section 2 of the C++ standard for details), but they can be ignored for the purposes of this discussion. In fact, we’ll be using the terms interchangeably here.

A.2.2 Macros

Preprocessor macros come in two flavors. **Object-like macros** can be defined this way:

¹GCC’s `-P` option inhibits the generation of source file and line number markers in preprocessed output.

```
#define identifier replacement-list
```

where the *identifier* names the macro being defined, and *replacement-list* is a sequence of zero or more tokens. Where the *identifier* appears in subsequent program text, it is **expanded** by the preprocessor into its *replacement-list*.

Function-like macros, which act as the “metafunctions of the preprocessing phase,” are defined as follows:

```
#define identifier(a1, a2, ... an) replacement-list
```

where each a_i is an identifier naming a **macro parameter**. When the macro name appears in subsequent program text followed by a suitable argument list, it is expanded into its *replacement-list*, except that each argument is substituted for the corresponding parameter where it appears in the *replacement-list*.²

A.2.3 Macro Arguments

Definition

A **macro argument** is a nonempty sequence of:

- Preprocessing tokens other than commas or parentheses, *and/or*
- Preprocessing tokens surrounded by matched pairs of parentheses.

This definition has consequences for preprocessor metaprogramming that must not be underestimated. Note, first of all, that the following tokens have special status:

```
, ( )
```

As a result, a macro argument can never contain an unmatched parenthesis, or a comma that is not surrounded by matched parentheses. For example, both lines following the definition of FOO below are ill-formed:

```
#define FOO(X) X // Unary identity macro
FOO(,)         // un-parenthesized comma or two empty arguments
FOO()          // unmatched parenthesis or missing argument
```

²We have omitted many details of how macro expansion works. We encourage you to take a few minutes to study section 16.3 of the C++ standard, which describes that process in straightforward terms.

Note also that the following tokens do *not* have special status; the preprocessor knows nothing about matched pairs of braces, brackets, or angle brackets:

```
{ } [ ] < >
```

As a result, these lines are also ill-formed:

```
FOO(std::pair<int, long>)           // two arguments
FOO({ int x = 1, y = 2; return x+y; }) // two arguments
```

It is possible to pass either string of tokens above as part of a single macro argument, provided it is parenthesized:

```
FOO((std::pair<int,int>))           // one argument
FOO(({ int x = 1, y = 2; return x+y; } )) // one argument
```

However, because of the special status of commas, it is impossible to strip parentheses from a macro argument without knowing the number of comma-separated token sequences it contains.³ If you are writing a macro that needs to be able to accept an argument containing a variable number of commas, your users will either have to parenthesize that argument *and* pass you the number of comma-separated token sequences as an additional argument, or they will have to encode the same information in one of the preprocessor data structures covered later in this appendix.

A.3 Preprocessor Library Structure

Since in-depth coverage of the Boost Preprocessor library is beyond the scope of this book, we'll try to give you the *tools* to gain an in-depth understanding of the library here. To do that, you'll need to use the electronic Preprocessor library documentation, which begins with the `index.html` file in the `libs/preprocessor/` subdirectory of your Boost installation.

On the left of your browser window you'll see an index, and if you follow the "Headers" link, it will reveal the structure of the `boost/preprocessor/` directory. Most of the library's headers are grouped into subdirectories according to related functionality. The top-level directory contains only a few headers that provide general-purpose macros, along with a header for each subdirectory that simply `#includes` all the headers in that subdirectory. For example, `boost/preprocessor/selection.hpp` does nothing more than to `#include` the `min.hpp` and `max.hpp` headers that comprise the contents of `boost/preprocessor/selection/`. The headers whose names *don't* correspond to subdirectories generally declare a macro whose name is the same as the name

³The C99 preprocessor, by virtue of its variadic macros, can do that and more. The C++ standardization committee is likely to adopt C99's preprocessor extensions for the next version of the C++ standard.

of the header, without the extension, and with a `BOOST_PP_` prefix. For example, `boost/preprocessor/selection/max.hpp` declares `BOOST_PP_MAX`.

You'll also notice that often a header will declare an additional macro with a `_D`, `_R`, or `_Z` suffix.⁴ For instance, `boost/preprocessor/selection/max.hpp` also declares `BOOST_PP_MAX_D`. For the purposes of this appendix, you should ignore those macros. Eventually you will want to understand how they can be used to optimize preprocessing speed; consult the Topics section of the library documentation under the subheading “reentrancy” for that information.

A.4 Preprocessor Library Abstractions

In this section we'll discuss the basic abstractions of the Preprocessor library, and give some simple examples of each.

A.4.1 Repetition

The repeated generation of `class T0, class T1... class Tn` that we achieved using `BOOST_PP_ENUM_PARAMS` was a specific case of the general concept of **horizontal repetition**. The library also has a concept of vertical repetition, which we'll get to in a moment. Horizontal repetition macros are all found in the library's `repetition/` subdirectory.

A.4.1.1 Horizontal Repetition

To generate the `tiny_size` specializations using horizontal repetition, we might write the following:

```
#include <boost/preprocessor/repetition.hpp>
#include <boost/preprocessor/arithmetic/sub.hpp>
#include <boost/preprocessor/punctuation/comma_if.hpp>

#define TINY_print(z, n, data) data

#define TINY_size(z, n, unused) \
    template <BOOST_PP_ENUM_PARAMS(n, class T)> \
    struct tiny_size< \
        BOOST_PP_ENUM_PARAMS(n, T) \
        BOOST_PP_COMMA_IF(n) \
```

⁴Macros with `_1ST`, `_2ND`, or `_3RD` suffixes, if they appear, should be ignored for a different reason: They are deprecated and will be removed from the library soon.

```

        BOOST_PP_ENUM(                                \
            BOOST_PP_SUB(TINY_MAX_SIZE,n), TINY_print, none) \
    >                                                \
    : mpl::int_<n> {};
```

```

BOOST_PP_REPEAT(TINY_MAX_SIZE, TINY_size, ~)

#undef TINY_size
#undef TINY_print
```

The code generation process is kicked off by calling `BOOST_PP_REPEAT`, a **higher-order macro** that repeatedly invokes the macro named by its second argument (`TINY_size`). The first argument specifies the number of repeated invocations, and the third one can be any data; it is passed on unchanged to the macro being invoked. In this case, `TINY_size` doesn't use that data, so the choice to pass `~` was arbitrary.⁵

Each time the `TINY_size` macro is invoked by `BOOST_PP_REPEAT`, it generates a different specialization of `tiny_size`. The macro accepts three parameters.

- `z` is related to the `_Z` macro suffix we mentioned earlier. You'll never need to use it except for optimization purposes, and can safely ignore it for now.
- `n` is the repetition index. In repeated invocations of `TINY_size`, `n` will be 0, then 1, then 2, and so on.
- `unused`, in this case, will be `~` on each repetition. In general, the final argument to a macro invoked by `BOOST_PP_REPEAT` is always the same as its invoker's final argument.

Because its *replacement-list* covers several lines, all but the last line of `TINY_size` is continued with a trailing backslash. The first few of those lines just invoke `BOOST_PP_ENUM_PARAMS` (which we already used in the primary template) to generate comma-separated lists, so each invocation of `TINY_size` produces something equivalent to:⁶

```

template <class T0, class T1, ... class Tn-1>
struct tiny_size<
    T0, T1, ... Tn-1
    ...more...
>
: mpl::int_<n> {};
```

⁵`~` is not an *entirely* arbitrary choice. Both `@` and `$` might have been good choices, except that they are technically not part of the basic character set that C++ implementations are required to support. An identifier like ignored might be subject to macro expansion, leading to unexpected results.

`BOOST_PP_COMMA_IF` generates a comma if its numeric argument is not 0. When `n` is 0, the list generated by the preceding line will be empty, and a leading comma directly following the `<` character would be ill-formed.

The next line uses `BOOST_PP_ENUM` to generate `TINY_MAX_SIZE-n` comma-separated copies of `none`. `BOOST_PP_ENUM` is just like `BOOST_PP_REPEAT` except that it generates commas between repetitions, so its second argument (`TINY_print`, here) must have the same signature as `TINY_size`. In this case, `TINY_print` ignores its repetition index `n`, and simply yields its third argument, `none`.

`BOOST_PP_SUB` implements token subtraction. It's crucial to understand that although the preprocessor *itself* can evaluate ordinary arithmetic expressions:

```
#define X 3
...
#if X - 1 > 0 // OK
    whatever
#endif
```

preprocessor *metaprograms* can only operate on tokens. Normally, when a macro in the Preprocessor library expects a numeric argument, it must be passed as a single token. If we had written `TINY_MAX_SIZE-n` instead of `BOOST_PP_SUB(TINY_MAX_SIZE, n)` above, the first argument to `BOOST_PP_ENUM` would have contained three tokens at each invocation: first `3-0`, then `3-1`, and finally `3-2`. `BOOST_PP_SUB`, though, generates single-token results: first `3`, then `2`, and finally `1`, in successive repetitions.

Naming Conventions

Note that `TINY_print` and `TINY_size` are `#undef'd` immediately after they're used, with no intervening `#includes`. They can therefore be thought of as "local" macro definitions. Because the preprocessor doesn't respect scope boundaries, it's important to choose names carefully to prevent clashes. We recommend `PREFIXED_lower_case` names for local macros and `PREFIXED_UPPER_CASE` names for global ones. The only exceptions are one-letter lowercase names, which are safe to use for local macros: No other header is likely to `#define` a global single-letter lowercase macro—that would be *very bad manners*.

A.4.1.2 Vertical Repetition

If you send the previous example through your preprocessor, you'll see one long line containing something like this:

⁶Note that the line continuation characters *and* the newlines following them are removed by the preprocessor, so the resulting code actually appears on a single line in the preprocessed output.

```
template <> struct tiny_size< none , none , none > : mpl::int_<0>
  {}; template < class T0> struct tiny_size< T0 , none , none > :
  mpl::int_<1> {}; template < class T0 , class T1> struct tiny_size
  < T0 , T1 , none > : mpl::int_<2> {};
```

The distinguishing feature of horizontal repetition is that all instances of the repeated pattern are generated on the same line of preprocessed output. For some jobs, like generating the primary `tiny_size` template, that's perfectly appropriate. In this case, however, there are at least two disadvantages.

1. It's hard to verify that our metaprogram is doing the right thing without reformatting the resulting code by hand.
2. The efficiency of nested horizontal repetitions varies widely across preprocessors. Each specialization generated by means of horizontal repetition contains three other horizontal repetitions: two invocations of `BOOST_PP_ENUM_PARAMS` and one invocation of `BOOST_PP_ENUM`. When `TINY_MAX_SIZE` is 3, you'll probably never care, but on at least one preprocessor still in use today, compilation begins to slow noticeably when `TINY_MAX_SIZE` reaches 8.⁷

The solution to these problems, naturally, is **vertical repetition**, which generates instances of a pattern across multiple lines. The Preprocessor library provides two means of vertical repetition: **local iteration** and **file iteration**.

Local Iteration

The most expedient way to demonstrate local iteration in our example is to replace the invocation of `BOOST_PP_REPEAT` with the following:

```
#include <boost/preprocessor/iteration/local.hpp>

#define BOOST_PP_LOCAL_MACRO(n)    TINY_size(~, n, ~)
#define BOOST_PP_LOCAL_LIMITS    (0, TINY_MAX_SIZE - 1)
#include BOOST_PP_LOCAL_ITERATE()
```

⁷That said, other preprocessors can handle 256 * 256 nested repetitions without any speed problems whatsoever.

Local iteration repeatedly invokes the user-defined macro with the special name `BOOST_PP_LOCAL_MACRO`, whose argument will be an iteration index. Since we already had `TINY_size` lying around, we've just defined `BOOST_PP_LOCAL_MACRO` to invoke it. The range of iteration indices are given by another user-defined macro, `BOOST_PP_LOCAL_LIMITS`, which must expand to a parenthesized pair of integer values representing the *inclusive* range of index values passed to `BOOST_PP_LOCAL_MACRO`. Note that this is one of the rare places where the library expects a numeric argument that can be an expression consisting of multiple tokens.

Finally, the repetition is initiated by `#include-ing` the result of invoking `BOOST_PP_LOCAL_ITERATE`, which will ultimately be a file in the Preprocessor library itself. You may find it surprising that many preprocessors can handle repeated file inclusion more quickly than nested horizontal repetition, but that is in fact the case.

If we throw the new example at our preprocessor, we'll see the following, on three separate lines in the output:

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

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

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

That represents a great improvement in verifiability, but it's still not ideal. As `TINY_MAX_SIZE` grows, it gets harder and harder to see that the pattern is generating what we'd like. If we could get some more line breaks into the output it would retain a more recognizable form.

Both repetition methods we've used so far have another drawback, though it doesn't show up in this example. Consider what would happen if `tiny_size` had a member function that we wanted to debug. If you've ever tried to use a debugger to step through a function generated by a preprocessor macro, you know that it's a frustrating experience at best: The debugger shows you the line from which the macro was ultimately invoked, which usually looks nothing at all like the code that was generated. Worse, as far as the debugger is concerned, *every* statement in that generated function occupies that same line.

File Iteration

Clearly, debuggability depends on preserving the association between generated code and the lines in the source file that describe the code pattern. File iteration generates pattern

instances by repeatedly `#include`-ing the same source file. The effect of file iteration on debuggability is similar to that of templates: Although separate instances appear to occupy the same source lines in the debugger, we do have the experience of stepping through the function's source code.

To apply file iteration in our example, we can replace our earlier local iteration code and the definition of `TINY_size`, with:

```
#include <boost/preprocessor/iteration/iterate.hpp>
#define BOOST_PP_ITERATION_LIMITS (0, TINY_MAX_SIZE - 1)
#define BOOST_PP_FILENAME_1      "tiny_size_spec.hpp"
#include BOOST_PP_ITERATE()
```

`BOOST_PP_ITERATION_LIMITS` follows the same pattern as `BOOST_PP_LOCAL_LIMITS` did, allowing us to specify an inclusive range of iteration indices. `BOOST_PP_FILENAME_1` specifies the name of the file to repeatedly `#include` (we'll show you that file in a moment). The trailing `1` indicates that this is the first nesting level of file iteration—should we need to invoke file iteration again from within `tiny_size_spec.hpp`, we'd need to use `BOOST_PP_FILENAME_2` instead.

The contents of `tiny_size_spec.hpp` should look familiar to you; most of it is the same as `TINY_size`'s *replacement-list*, without the backslashes:

```
#define n BOOST_PP_ITERATION()

template <BOOST_PP_ENUM_PARAMS(n, class T)>
struct tiny_size<
    BOOST_PP_ENUM_PARAMS(n, T)
    BOOST_PP_COMMA_IF(n)
    BOOST_PP_ENUM(BOOST_PP_SUB(TINY_MAX_SIZE, n), TINY_print, none)
>
: mpl::int_<n> {};

#undef n
```

The Library transmits the iteration index to us in the result of `BOOST_PP_ITERATION()`; `n` is nothing more than a convenient local macro used to reduce syntactic noise. Note that we didn't use `#include` guards because we need `tiny_size_spec.hpp` to be processed multiple times.

The preprocessed result should now preserve the line structure of the pattern and be more verifiable for larger values of `TINY_MAX_SIZE`. For instance, when `TINY_MAX_SIZE` is 8, the following excerpt appears in the output of GCC's preprocessing phase:

...

```

template < class T0 , class T1 , class T2 , class T3>
struct tiny_size<
    T0 , T1 , T2 , T3
    ,
    none , none , none , none
>
: mpl::int_<4> {};

template < class T0 , class T1 , class T2 , class T3 , class T4>
struct tiny_size<
    T0 , T1 , T2 , T3 , T4
    ,
    none , none , none
>
: mpl::int_<5> {};
...etc.

```

Self-Iteration

Creating an entirely new file like `tiny_size_spec.hpp` each time we want to express a trivial code pattern for file repetition can be inconvenient. Fortunately, the library provides a macro that allows us to place the pattern right in the file that invokes the iteration. `BOOST_PP_IS_ITERATING` is defined to a nonzero value whenever we're inside an iteration. We can use that value to select between the part of a file that invokes the iteration and the part that provides the repeated pattern. Here's a complete `tiny_size.hpp` file that demonstrates self-iteration. Note in particular the placement and use of the `#include guard` `TINY_SIZE_HPP_INCLUDED`:

```

#ifndef BOOST_PP_IS_ITERATING

#   ifndef TINY_SIZE_HPP_INCLUDED
#       define TINY_SIZE_HPP_INCLUDED

#           include <boost/preprocessor/repetition.hpp>
#           include <boost/preprocessor/arithmetic/sub.hpp>
#           include <boost/preprocessor/punctuation/comma_if.hpp>
#           include <boost/preprocessor/iteration/iterate.hpp>

#           ifndef TINY_MAX_SIZE
#               define TINY_MAX_SIZE 3 // default maximum size is 3
#           endif

```

```

// primary template
template <BOOST_PP_ENUM_PARAMS(TINY_MAX_SIZE, class T)>
struct tiny_size
    : mpl::int_<TINY_MAX_SIZE>
    {};

// generate specializations
#   define BOOST_PP_ITERATION_LIMITS (0, TINY_MAX_SIZE - 1)
#   define BOOST_PP_FILENAME_1      "tiny_size.hpp" // this file
#   include BOOST_PP_ITERATE()

# endif // TINY_SIZE_HPP_INCLUDED

#else // BOOST_PP_IS_ITERATING

#   define n BOOST_PP_ITERATION()

#   define TINY_print(z, n, data) data

// specialization pattern
template <BOOST_PP_ENUM_PARAMS(n, class T)>
struct tiny_size<
    BOOST_PP_ENUM_PARAMS(n,T)
    BOOST_PP_COMMA_IF(n)
    BOOST_PP_ENUM(BOOST_PP_SUB(TINY_MAX_SIZE,n), TINY_print, none)
>
    : mpl::int_<n> {};

#   undef TINY_print
#   undef n

#endif // BOOST_PP_IS_ITERATING

```

More

There's a good deal more to file iteration than what we've been able to show you here. For more details, we encourage you to delve into the library's electronic documentation of `BOOST_PP_ITERATE` and friends. Also, it's important to note that no single technique for repetition is superior to any other: Your choice may depend on convenience, verifiability, debuggability, compilation speed, and your own sense of "logical coherence."

A.4.2 Arithmetic, Logical, and Comparison Operations

As we mentioned earlier, many of the Preprocessor library interfaces require single-token numeric arguments, and when those numbers need to be computed arithmetically, straightforward arithmetic expressions are inappropriate. We used `BOOST_PP_SUB` to subtract two numeric tokens in our `tiny_size` examples. The library contains a suite of operations for non-negative integral token arithmetic in its `arithmetic/` subdirectory, as shown in Table A.1

Table A.1: Preprocessor Library Arithmetic Operations

Expression	Value of Single Token Result
<code>BOOST_PP_ADD(x, y)</code>	$x + y$
<code>BOOST_PP_DEC(x)</code>	$x - 1$
<code>BOOST_PP_DIV(x, y)</code>	x / y
<code>BOOST_PP_INC(x)</code>	$x + 1$
<code>BOOST_PP_MOD(x, y)</code>	$x \% y$
<code>BOOST_PP_MUL(x, y)</code>	$x * y$
<code>BOOST_PP_SUB(x, y)</code>	$x - y$

The `logical/` subdirectory contains the convenient Boolean token operations shown in Table A.2 and the more efficient operations shown in Table A.3, which require that their operands are either 0 or 1 (a single bit).

Table A.2: Preprocessor Library Integer Logical Operations

Expression	Value of Single Token Result
<code>BOOST_PP_AND(x, y)</code>	$x \ \&\& \ y$
<code>BOOST_PP_NOR(x, y)</code>	$!(x \ \ y)$
<code>BOOST_PP_OR(x, y)</code>	$x \ \ y$
<code>BOOST_PP_XOR(x, y)</code>	$(\text{bool})x \ != \ (\text{bool})y \ ? \ 1 \ : \ 0$
<code>BOOST_PP_NOT(x)</code>	$x \ ? \ 0 \ : \ 1$
<code>BOOST_PP_BOOL(x)</code>	$x \ ? \ 1 \ : \ 0$

Table A.3: Preprocessor Library Bit Logical Operations

Expression	Value of Single Token Result
<code>BOOST_PP_BITAND(x, y)</code>	$x \ \&\& \ y$

Table A.3: Preprocessor Library Bit Logical Operations

Expression	Value of Single Token Result
<code>BOOST_PP_BITNOR(x, y)</code>	<code>!(x y)</code>
<code>BOOST_PP_BITOR(x, y)</code>	<code>x y</code>
<code>BOOST_PP_BITXOR(x, y)</code>	<code>(bool)x != (bool)y ? 1 : 0</code>
<code>BOOST_PP_COMPL(x)</code>	<code>x ? 0 : 1</code>

Finally, the `comparison/` subdirectory provides the token integral comparison operations shown in Table A.4.

Table A.4: Preprocessor Library Comparison Operations

Expression	Value of Single Token Result
<code>BOOST_PP_EQUAL(x, y)</code>	<code>x == y ? 1 : 0</code>
<code>BOOST_PP_NOT_EQUAL(x, y)</code>	<code>x != y ? 1 : 0</code>
<code>BOOST_PP_LESS(x, y)</code>	<code>x < y ? 1 : 0</code>
<code>BOOST_PP_LESS_EQUAL(x, y)</code>	<code>x <= y ? 1 : 0</code>
<code>BOOST_PP_GREATER(x, y)</code>	<code>x > y ? 1 : 0</code>
<code>BOOST_PP_GREATER_EQUAL(x, y)</code>	<code>x >= y ? 1 : 0</code>

Because it's common to have a choice among several workable comparison operators, it may be useful to know that `BOOST_PP_EQUAL` and `BOOST_PP_NOT_EQUAL` are likely to be $O(1)$ while the other comparison operators are generally slower.

A.4.3 Control Structures

In its `control/` directory, the Preprocessor Library supplies a macro `BOOST_PP_IF(c, t, f)` that fulfills a similar role to the one filled by `mpl::if_`. To explore the “control” group, we'll generate code for a framework of generic function objects: the Boost Function Library.⁸ `boost::function` is partially specialized to match function type arguments of each arity up to the maximum supported by the library:

```
template <class Signature> struct function;    // primary template

template <class R>                                // arity = 0
struct function<R()>
    definition not shown...
```

```

template <class R, class A0>                                // arity = 1
struct function<R(A0)>
    definition not shown...

template <class R, class A0, class A1>                    // arity = 2
struct function<R(A0,A1)>
    definition not shown...

template <class R, class A0, class A1, class A2>          // arity = 3
struct function<R(A0,A1,A2)>
    definition not shown...

etc.

```

We've already covered a few strategies that can be used to generate the pattern above, so we won't belabor that part of the problem; the file iteration approach we used for `tiny_size` would be fine:

```

#ifndef BOOST_PP_IS_ITERATING

#   ifndef BOOST_FUNCTION_HPP_INCLUDED
#       define BOOST_FUNCTION_HPP_INCLUDED

#       include <boost/preprocessor/repetition.hpp>
#       include <boost/preprocessor/iteration/iterate.hpp>

#       ifndef FUNCTION_MAX_ARITY
#           define FUNCTION_MAX_ARITY 15
#       endif

template <class Signature> struct function;    // primary template

// generate specializations
#   define BOOST_PP_ITERATION_LIMITS (0, FUNCTION_MAX_ARITY)
#   define BOOST_PP_FILENAME_1      "boost/function.hpp" // this file
#   include BOOST_PP_ITERATE()

#   endif // BOOST_FUNCTION_HPP_INCLUDED

```

⁸We touched briefly on the design of Boost Function when we discussed type erasure in Chapter 9. See the Function library documentation at boost_1_32_0/libs/function/index.html on the CD that accompanies this book for more information.

```

#else // BOOST_PP_IS_ITERATING

#   define n BOOST_PP_ITERATION()

// specialization pattern
template <class R BOOST_PP_ENUM_TRAILING_PARAMS(n, class A)>
struct function<R ( BOOST_PP_ENUM_PARAMS(n,A) )>
    definition not shown...

#   undef n

#endif // BOOST_PP_IS_ITERATING

```

`BOOST_PP_ENUM_TRAILING_PARAMS`, used above, is just like `BOOST_PP_ENUM_PARAMS` except that when its first argument is not 0, it generates a leading comma.

A.4.3.1 Argument Selection

For the sake of interoperability with C++ standard library algorithms, it might be nice if functions of one or two arguments were derived from appropriate specializations of `std::unary_function` or `std::binary_function`, respectively.⁹ `BOOST_PP_IF` is a great tool for dealing with special cases:

```

#   include <boost/preprocessor/control/if.hpp>
#   include <boost/preprocessor/comparison/equal.hpp>

// specialization pattern
template <class R BOOST_PP_ENUM_TRAILING_PARAMS(n, class A)>
struct function<R ( BOOST_PP_ENUM_PARAMS(n,A) )>
    BOOST_PP_IF (
        BOOST_PP_EQUAL(n,2), : std::binary_function<A0, A1, R>
        , BOOST_PP_IF (
            BOOST_PP_EQUAL(n,1), : std::unary_function<A0, R>
            , ...empty argument...
        )
    )
    { ...class body omitted... };

```

⁹While derivation from `std::unary_function` or `std::binary_function` might be necessary for interoperability with some older library implementations, it may inhibit the Empty Base Optimization (EBO) from taking effect when two such derived classes are part of the same object. For more information, see section 9.4. In general, it's better to expose `first_argument_type`, `second_argument_type`, and `result_type` typedefs directly.

Well, our first attempt has run into several problems. First off, you're not allowed to pass an empty argument to the preprocessor.³ Secondly, because angle brackets don't get special treatment, the commas in the `std::unary_function` and `std::binary_function` specializations above are treated as macro argument separators, and the preprocessor will complain that we've passed the wrong number of arguments to `BOOST_PP_IF` in two places.

Because it captures all of the issues, let's focus on the inner `BOOST_PP_IF` invocation for a moment. The strategy that `mpl::eval_if` uses, of selecting a nullary function to invoke, could work nicely here. The preprocessor doesn't have a direct analogue for `mpl::eval_if`, but it doesn't really need one: We can get the right effect by adding a second set of parentheses to `BOOST_PP_IF`.

```
#define BOOST_FUNCTION_unary()      : std::unary_function<A0,R>
#define BOOST_FUNCTION_empty()     // nothing

...

    , BOOST_PP_IF (
        BOOST_PP_EQUAL(n,1), BOOST_FUNCTION_unary
    , BOOST_FUNCTION_empty
    ) ( )

#undef BOOST_FUNCTION_empty
#undef BOOST_FUNCTION_unary
```

A nullary macro that generates nothing is so commonly needed that the library's "facilities" group provides one: `BOOST_PP_EMPTY`. To complete the example we'll need to delay evaluation all the way to the outer `BOOST_PP_IF` invocation, because `std::binary_function<A0,A1,R>` also has a "comma problem":

```
# include <boost/preprocessor/facilities/empty.hpp>

# define BOOST_FUNCTION_binary() : std::binary_function<A0,A1,R>
# define BOOST_FUNCTION_unary() : std::unary_function<A0,R>

// specialization pattern
template <class R BOOST_PP_ENUM_TRAILING_PARAMS(n, class A)>
struct function<R ( BOOST_PP_ENUM_PARAMS(n,A) )>
    BOOST_PP_IF (
        BOOST_PP_EQUAL(n,2), BOOST_FUNCTION_binary
    , BOOST_PP_IF (
        BOOST_PP_EQUAL(n,1), BOOST_FUNCTION_unary
```

```

        , BOOST_PP_EMPTY
    )
) ()
{
    ...class body omitted...
};

# undef BOOST_FUNCTION_unary
# undef BOOST_FUNCTION_binary
# undef n

```

Note that because we happened to be using file iteration, we could have also used `#if` on `n`'s value directly:

```

template <class R BOOST_PP_ENUM_TRAILING_PARAMS(n, class A)>
    struct function<R ( BOOST_PP_ENUM_PARAMS(n,A) )>
    #if n == 2
        : std::binary_function<A0, A1, R>
    #elif n == 1
        : std::unary_function<A0, R>
    #endif

```

`BOOST_PP_IF` has the advantage of enabling us to encapsulate the logic in a reusable macro, parameterized on `n`, that is compatible with all repetition constructs:

```

#define BOOST_FUNCTION_BASE(n) \
    BOOST_PP_IF(BOOST_PP_EQUAL(n,2), BOOST_FUNCTION_binary \
        , BOOST_PP_IF(BOOST_PP_EQUAL(n,1), BOOST_FUNCTION_unary \
            , BOOST_PP_EMPTY \
        ) \
    ) \
) ()

```

A.4.3.2 Other Selection Constructs

`BOOST_PP_IDENTITY`, also in the “facilities” group, is an interesting cousin of `BOOST_PP_EMPTY`:

```

#define BOOST_PP_IDENTITY(tokens) tokens BOOST_PP_EMPTY

```

You can think of it as creating a nullary macro that returns tokens: When empty parentheses are appended, the trailing `BOOST_PP_EMPTY` is expanded leaving just tokens behind. If we had wanted inheritance from `mpl::empty_base` when function's arity is not one or two, we could have used `BOOST_PP_IDENTITY`:

```
// specialization pattern
template <class R BOOST_PP_ENUM_TRAILING_PARAMS(n, class A)>
struct function<R ( BOOST_PP_ENUM_PARAMS(n,A) )>
    BOOST_PP_IF(
        BOOST_PP_EQUAL(n,2), BOOST_FUNCTION_binary
    , BOOST_PP_IF(
        BOOST_PP_EQUAL(n,1), BOOST_FUNCTION_unary
    , BOOST_PP_IDENTITY(: mpl::empty_base)
    )
    ) ( )
{
    ...class body omitted...
};
```

It's also worth knowing about `BOOST_PP_EXPR_IF`, which generates its second argument or nothing, depending on the Boolean value of its first:

```
#define BOOST_PP_EXPR_IF(c,tokens) \
    BOOST_PP_IF(c,BOOST_PP_IDENTITY(tokens),BOOST_PP_EMPTY) ()
```

So `BOOST_PP_EXPR_IF(1,foo)` expands to `foo`, while `BOOST_PP_EXPR_IF(0,foo)` expands to nothing.

A.4.4 Token Pasting

It would be nice if there were a generic way to access the return and parameter types of *all* function objects, rather than just the unary and binary ones. A metafunction returning the signature as an MPL sequence would do the trick. We could just specialize `signature` for each function arity:

```
template <class F> struct signature; // primary template

// partial specializations for boost::function
template <class R>
struct signature<function<R()> >
    : mpl::vector1<R> {};
```

```

template <class R, class A0>
struct signature<function<R(A0)>> >
    : mpl::vector2<R,A0> {};

template <class R, class A0, class A1>
struct signature<function<R(A0,A1)>> >
    : mpl::vector3<R,A0,A1> {};

...

```

To generate these specializations, we might add the following to our pattern:

```

template <class R BOOST_PP_ENUM_TRAILING_PARAMS(n, class A)>
struct signature<function<R( BOOST_PP_ENUM_PARAMS(n,A) )>> >
    : mpl::BOOST_PP_CAT(vector,n)<
        R BOOST_PP_ENUM_TRAILING_PARAMS(n,A)
    > {};

```

BOOST_PP_CAT implements **token pasting**; its two arguments are “glued” together into a single token. Since this is a general-purpose macro, it sits in `cat.hpp` at the top level of the library’s directory tree.

Although the preprocessor has a built-in token-pasting operator, `##`, it only works within a macro definition. If we’d used it here, it wouldn’t have taken effect at all:

```

template <class R>
struct signature<function<R()>> >
    : mpl::vector##1<R> {};

template <class R, class A0>
struct signature<function<R(A0)>> >
    : mpl::vector##2<R,A0> {};

template <class R, class A0, class A1>
struct signature<function<R(A0,A1)>> >
    : mpl::vector##3<R,A0,A1> {};

...

```

Also, `##` often yields surprising results by taking effect before its arguments have been expanded:

```

#define N          10
#define VEC(i)    vector##i

VEC(N)           // vectorN

```

By contrast, `BOOST_PP_CAT` delays concatenation until after its arguments have been fully evaluated:

```

#define N          10
#define VEC(i)    BOOST_PP_CAT(vector,i)

VEC(N)           // vector10

```

A.4.5 Data Types

The Preprocessor library also provides **data types**, which you can think of as being analogous to the MPL's type sequences. Preprocessor data types store *macro arguments* instead of C++ types.

A.4.5.1 Sequences

A **sequence** (or **seq** for short) is any string of nonempty parenthesized *macro arguments*. For instance, here's a three-element sequence:

```

#define MY_SEQ    (f(12))(a + 1)(foo)

```

Here's how we might use a sequence to generate specializations of the `is_integral` template from the Boost Type Traits library (see Chapter 2):

```

#include <boost/preprocessor/seq.hpp>

template <class T>
struct is_integral : mpl::false_ {};

// a seq of integral types with unsigned counterparts
#define BOOST_TT_basic_ints      (char)(short)(int)(long)

// generate a seq containing "signed t" and "unsigned t"
#define BOOST_TT_int_pair(r,data,t)  (signed t)(unsigned t)

// a seq of all the integral types

```

```

#define BOOST_PP_SEQ_FOR_EACH_BOOST_PP_SEQ_FOR_EACH(BOOST_PP_SEQ_FOR_EACH, ~, BOOST_PP_SEQ_FOR_EACH)
    (bool) (char)
    BOOST_PP_SEQ_FOR_EACH_BOOST_PP_SEQ_FOR_EACH(BOOST_PP_SEQ_FOR_EACH, ~, BOOST_PP_SEQ_FOR_EACH)

// generate an is_integral specialization for type t
#define BOOST_PP_SEQ_FOR_EACH_BOOST_PP_SEQ_FOR_EACH(r,data,t) \
    template <> \
    struct is_integral<t> : mpl::true_ {};

BOOST_PP_SEQ_FOR_EACH_BOOST_PP_SEQ_FOR_EACH(BOOST_PP_SEQ_FOR_EACH, ~, BOOST_PP_SEQ_FOR_EACH)

#undef BOOST_PP_SEQ_FOR_EACH_BOOST_PP_SEQ_FOR_EACH
#undef BOOST_PP_SEQ_FOR_EACH
#undef BOOST_PP_SEQ_FOR_EACH_BOOST_PP_SEQ_FOR_EACH
#undef BOOST_PP_SEQ_FOR_EACH

```

`BOOST_PP_SEQ_FOR_EACH` is a higher-order macro, similar to `BOOST_PP_REPEAT`, that invokes its first argument on each element of its third argument.

Sequences are the most efficient, most flexible, and easiest-to-use of the library's data structures, provided that you never need to make an empty one: An empty sequence would contain no tokens, and so couldn't be passed as a macro argument. The other data structures covered here all have an empty representation.

The facilities for manipulating sequences are all in the library's `seq/` subdirectory. They are summarized in Table A.5, where t is the sequence $(t_0) (t_1) \dots (t_k)$. Where s , r , and d appear, they have a similar purpose to the z parameters we discussed earlier (and suggested you ignore for now).

Table A.5: Preprocessor Sequence Operations

Expression	Result
<code>BOOST_PP_SEQ_CAT(t)</code>	$t_0 t_1 \dots t_k$
<code>BOOST_PP_SEQ_ELEM(n,t)</code>	t_n
<code>BOOST_PP_SEQ_ENUM(t)</code>	t_0, t_1, \dots, t_k
<code>BOOST_PP_SEQ_FILTER(pred, data, t)</code>	t without the elements that don't satisfy <code>pred</code>
<code>BOOST_PP_SEQ_FIRST_N(n,t)</code>	$(t_0) (t_1) \dots (t_{n-1})$
<code>BOOST_PP_SEQ_FOLD_LEFT(op, x, t)</code>	$\dots \text{op}(s, \text{op}(s, \text{op}(s, x, t_0), t_1), t_2) \dots$
<code>BOOST_PP_SEQ_FOLD_RIGHT(op, x, t)</code>	$\dots \text{op}(s, \text{op}(s, \text{op}(s, x, t_k), t_{k-1}), t_{k-2}) \dots$
<code>BOOST_PP_SEQ_FOR_EACH(f, x, t)</code>	$f(r, x, t_0) f(r, x, t_1) \dots f(r, x, t_k)$

Table A.5: Preprocessor Sequence Operations

Expression	Result
BOOST_PP_SEQ_FOR_EACH_I(g, x, t)	$g(r, x, 0, t_0) g(r, x, 1, t_1) \dots g(r, x, k, t_k)$
BOOST_PP_SEQ_FOR_EACH_PRODUCT(h, x, t)	Cartesian product — see online docs
BOOST_PP_SEQ_INSERT(t, i, tokens)	$(t_0) (t_1) \dots (t_{i-1}) (\text{tokens}) (t_i) (t_{i+1}) \dots (t_k)$
BOOST_PP_SEQ_POP_BACK(t)	$(t_0) (t_1) \dots (t_{k-1})$
BOOST_PP_SEQ_POP_FRONT(t)	$(t_1) (t_2) \dots (t_k)$
BOOST_PP_SEQ_PUSH_BACK(t, tokens)	$(t_0) (t_1) \dots (t_k) (\text{tokens})$
BOOST_PP_SEQ_PUSH_FRONT(t, tokens)	$(\text{tokens}) (t_0) (t_1) \dots (t_k)$
BOOST_PP_SEQ_REMOVE(t, i)	$(t_0) (t_1) \dots (t_{i-1}) (t_{i+1}) \dots (t_k)$
BOOST_PP_SEQ_REPLACE(t, i, tokens)	$(t_0) (t_1) \dots (t_{i-1}) (\text{tokens}) (t_{i+1}) \dots (t_k)$
BOOST_PP_SEQ_REST_N(n, t)	$(t_n) (t_{n+1}) \dots (t_k)$
BOOST_PP_SEQ_REVERSE(t)	$(t_k) (t_{k-1}) \dots (t_0)$
BOOST_PP_SEQ_HEAD(t)	t_0
BOOST_PP_SEQ_TAIL(t)	$(t_1) (t_2) \dots (t_k)$
BOOST_PP_SEQ_SIZE(t)	$k+1$
BOOST_PP_SEQ_SUBSEQ(t, i, m)	$(t_i) (t_{i+1}) \dots (t_{i+m-1})$
BOOST_PP_SEQ_TO_ARRAY(t)	$(k+1, (t_0, t_1, \dots, t_k))$
BOOST_PP_SEQ_TO_TUPLE(t)	(t_0, t_1, \dots, t_k)
BOOST_PP_SEQ_TRANSFORM(f, x, t)	$(f(r, x, t_0))$ $(f(r, x, t_1)) \dots (f(r, x, t_k))$

It's worth noting that while there is no upper limit on the length of a sequence, operations such as BOOST_PP_SEQ_ELEM that take numeric arguments will only work with values up to 256.

A.4.5.2 Tuples

A **tuple** is a very simple data structure for which the library provides random access and a few other basic operations. A tuple takes the form of a parenthesized, comma-separated list of *macro arguments*. For example, this is a three-element tuple:

```
#define TUPLE3      (f(12), a + 1, foo)
```

The operations in the library’s `tuple/` subdirectory can handle tuples of up to 25 elements. For example, a tuple’s *N*th element can be accessed via `BOOST_PP_TUPLE_ELEM`, as follows:

```

// length index tuple
BOOST_PP_TUPLE_ELEM( 3 , 1 , TUPLE3) // a + 1

```

Notice that we had to pass the tuple’s length as the second argument to `BOOST_PP_TUPLE_ELEM`; in fact, *all* tuple operations require explicit specification of the tuple’s length. We’re not going to summarize the other four operations in the “tuple” group here—you can consult the Preprocessor library’s electronic documentation for more details. We note, however, that sequences can be transformed into tuples with `BOOST_PP_SEQ_TO_TUPLE`, and nonempty tuples can be transformed back into sequences with `BOOST_PP_TUPLE_TO_SEQ`.

The greatest strength of tuples is that they conveniently take the same representation as a macro argument list:

```

#define FIRST_OF_THREE(a1,a2,a3)    a1
#define SECOND_OF_THREE(a1,a2,a3)   a2
#define THIRD_OF_THREE(a1,a2,a3)    a3

// uses tuple as an argument list
# define SELECT(selector, tuple)    selector tuple

SELECT(THIRD_OF_THREE, TUPLE3) // foo

```

A.4.5.3 Arrays

An **array** is just a tuple containing a non-negative integer and a tuple of that length:

```

#define ARRAY3      ( 3, TUPLE3 )

```

Because an array carries its length around with it, the library’s interface for operating on arrays is much more convenient than the one used for tuples:

```

BOOST_PP_ARRAY_ELEM(1, ARRAY3) // a + 1

```

The facilities for manipulating arrays of up to 25 elements are all in the library’s `array/` subdirectory. They are summarized in Table A.6, where *a* is the array $(k, (a_0, a_1, \dots, a_{k-1}))$.

Table A.6: Preprocessor Array Operations

Expression	Result
BOOST_PP_ARRAY_DATA (a)	$(a_0, a_1, \dots a_{k-1})$
BOOST_PP_ARRAY_ELEM (i, a)	a_i
BOOST_PP_ARRAY_INSERT (a, i, tokens)	$(k+1, (a_0, a_1, \dots a_{i-1}, \text{tokens}, a_i, a_{i+1}, \dots a_{k-1}))$
BOOST_PP_ARRAY_POP_BACK (a)	$(k-1, (a_0, a_1, \dots a_{k-2}))$
BOOST_PP_ARRAY_POP_FRONT (a)	$(k-1, (a_1, a_2, \dots a_{k-1}))$
BOOST_PP_ARRAY_PUSH_BACK (a, tokens)	$(k+1, (a_0, a_1, \dots a_{k-1}, \text{tokens}))$
BOOST_PP_ARRAY_PUSH_FRONT (a, tokens)	$(k+1, (\text{tokens}, a_1, a_2, \dots a_{k-1}))$
BOOST_PP_ARRAY_REMOVE (a, i)	$(k-1, (a_0, a_1, \dots a_{i-1}, a_{i+1}, \dots a_{k-1}))$
BOOST_PP_ARRAY_REPLACE (a, i, tokens)	$(k, (a_0, a_1, \dots a_{i-1}, \text{tokens}, a_{i+1}, \dots a_{k-1}))$
BOOST_PP_ARRAY_REVERSE (a)	$(k, (a_{k-1}, a_{k-2}, \dots a_1, a_0))$
BOOST_PP_ARRAY_SIZE (a)	k

A.4.5.4 Lists

A **list** is a two-element tuple whose first element is the first element of the list, and whose second element is a list of the remaining elements, or BOOST_PP_NIL if there are no remaining elements. Lists have access characteristics similar to those of a runtime linked list. Here is a three-element list:

```
#define LIST3      (f(12), (a + 1, (foo, BOOST_PP_NIL)))
```

The facilities for manipulating lists are all in the library's `list/` subdirectory. Because the operations are a subset of those provided for sequences, we're not going to summarize them here—it should be easy to understand the list operations by reading the documentation on the basis of our coverage of sequences.

Like sequences, lists have no fixed upper length bound. Unlike sequences, lists can also be empty. It's rare to need more than 25 elements in a preprocessor data structure, and lists tend to be slower to manipulate and harder to read than any of the other structures, so they should normally be used only as a last resort.

A.5 Exercise

A-0 Fully preprocessor-ize the `tiny` type sequence implemented in Chapter 5 so that all boilerplate code is eliminated and the maximum size of a `tiny` sequence can be adjusted by changing `TINY_MAX_SIZE`.

References

[MK04] Paul Mensonides and Vesa Karvonen. “The Boost Preprocessor Library.”
<http://www.boost.org/libs/preprocessor>.