Practical C++ Metaprogramming

Book · September 2016

O'REILLY®

Practical C++ Metaprogramming

Modern Techniques for Accelerated Development

Edouard Alligand
& Joel Falcou

O'REILLY®

Additional Resources

4 Easy Ways to Learn More and Stay Current

Programming Newsletter
Get programming related news and content delivered weekly to your inbox.
oreilly.com/programming/newsletter

Free Webcast Series
Learn about popular programming topics from experts live, online.
webcasts.oreilly.com

O'Reilly Radar
Read more insight and analysis about emerging technologies.
radar.oreilly.com

Conferences
Immerse yourself in learning at an upcoming O'Reilly conference.
conferences.oreilly.com

Practical C++ Metaprogramming

Modern Techniques for Accelerated Development

Edouard Alligand and Joel Falcou

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Practical C++ Metaprogramming
by Edouard Alligand and Joel Falcou

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Nan Barber and Brian Foster
Production Editor: Colleen Lobner
Copyeditor: Octal Publishing, Inc.
Proofreader: Rachel Head

Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

September 2016: First Edition

Revision History for the First Edition
2016-09-13: First Release

978-1-491-95504-8

[LSI]

Table of Contents

Preface

Another arcane text about an overly complex language! C++ is already difficult enough to master; why do people feel the need to make it even more difficult?

C++'s power comes at a price, but with the latest revisions of the language, the bar has been drastically lowered. The improvements in C++11 and C++14 have had a positive impact in many areas, from how you write a loop to how you can write templates.

We've had the idea of writing about template metaprogramming for a long time, because we wanted to demonstrate how much easier it has become. We also wanted to prove its usefulness and efficiency. By that we mean that it's not only a valid solution, but sometimes the *best* solution.

Last but not least, even if you don't use metaprogramming every day, understanding its concepts will make you a better programmer: you will learn to look at problems differently and increase your mastery and understanding of the language.

A Journey of a Thousand Miles Begins with a Single Step

Really mastering C++ metaprogramming is difficult and takes a lot of time. You need to understand how compilers work to get around their bugs and limitations. The feedback you can receive when you have an error is more often than not arcane.

That is the bad news.

The good news is that you don't need to master C++ metaprogramming, because you are standing on the shoulders of giants.

In this report, we will progressively expose you to the technique and its practical applications, and give you a list of tools that you can use to get right to it.

Then, depending on your tastes and your aspirations, you can decide how deep down the rabbit hole you want to go.

Understanding Metaprogramming

Metaprogramming is a technique that can greatly increase your productivity when properly used. Improperly used, though it can result in unmaintainable code and greatly increased development time.

Dismissing metaprogramming based on a preconceived notion or dogma is counterproductive. Nevertheless, properly understanding if the technique suits your needs is paramount for fruitful and rewarding use.

An analogy we like to use is that you should see a metaprogram as a robot you program to do a job for you. After you've programmed the robot, it will be happy to do the task for you a thousand times, without error. Additionally, the robot is faster than you and more precise.

If you do something wrong, though, it might not be immediately obvious where the problem is. Is it a problem in how you programmed the robot? Is it a bug in the robot? Or is your program correct but the result unexpected?

That's what makes metaprogramming more difficult: the feedback isn't immediate, and because you added an intermediary you've added more variables to the equation.

That's also why before using this technique you must ensure that you know how to program the robot.

Conventions Used in This Report

The following typographical conventions are used in this report:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

This element signifies a general note.

This element indicates a warning or caution.

Acknowledgments

This report would probably not exist without the work of Aleksey Gurtovoy and David Abrahams, authors of the Boost.MPL library and the reference book *C++ Template Metaprogramming* (Addison-Wesley Professional).

More recently, Eric Niebler and Peter Dimov paved the way to what modern C++ template metaprogramming should look like. They have been greatly influential in our work.

We would also like to thank all of the contributors to the Brigand library and Louis Dionne for his metaprogramming library benchmark.

Finally, we would like to thank Jon Kalb and Michael Caisse for their reviews, as well as our families, friends, and coworkers, who have been incredibly supportive.

CHAPTER 1
Introduction

If you grabbed this report, it means that you have at least *curiosity* about C++ metaprogramming, a topic that often generates outright rejection.

Before we talk about template metaprogramming, let's ask ourselves a question: why do we violently reject some techniques, even before studying them?

There are, of course, many valid reasons to reject something new, because, let's be frank, sometimes concepts are just plain nonsense or totally irrelevant to the task at hand.

However, there is also a lot to be said about managing your own psychology when accepting novelty, and recognizing our own mental barriers is the best way to prevent them from growing.

The purpose of this report is to demonstrate that understanding C++ metaprogramming will make you a better C++ programmer, as well as a better software engineer in general.

A Misunderstood Technique

Like every technique, we can overuse and misunderstand metaprogramming. The most common reproaches are that it makes code more difficult to read and understand and that it has no real benefit.

As you progress along the path of software engineering, the techniques you learn are more and more advanced. You could opt to rely solely on simple techniques and solve complex problems via a com-

position of these techniques, but you will be missing an opportunity to be more concise, more productive, and sometimes more efficient.

Imagine that you are given an array and that you need to fill it with increasing integers. You could write the following function:

```
void f(int * p, size_t l)
{
    for(size_t i = 0; i < l; ++i)
    {
        p[i] = i;
    }
}

// ...

int my_array[5];

f(my_array, 5);
```

Or you could use the Standard Template Library (STL):

```
int my_array[5];

std::iota(my_array, my_array + 5, 0);
```

The generated assembly code may be equivalent, if not identical, yet in the latter case, because you learned about an STL function, you gained both in productivity and in information density. A programmer who doesn't know about the `iota` function just needs to look it up.

What makes a good software engineer is not only the size of his toolbox, but, most importantly, his ability to choose the right tool at the right time.

C++ metaprogramming has, indeed, an *unusual* syntax, although it has significantly improved in the past few years with the release of C++11 and C++14.

The concepts behind C++ metaprogramming, on the other hand, are extremely coherent and logical: it's functional programming! That's why on the surface it might look arcane, but after you are taught the underlying concepts it all makes sense. This is something we will see in more depth in Chapter 3.

In this report, we want to expose you to C++ metaprogramming in a way that is intelligible and practical.

When you are finished, we hope that you that will agree with us that it is both useful and accessible.

What Is Metaprogramming?

By definition, metaprogramming is the design of programs whose input and output are programs themselves. Put another way, it's writing code whose job is to write code itself. It can be seen as the ultimate level of abstraction, as code fragments are actually seen as data and handled as such.

It might sound esoteric, but it's actually a well-known practice. If you've ever written a Bash script generating C files from a boilerplate file, you've done metaprogramming. If you've ever written C macros, you've done metaprogramming. In another sphere, you could debate whether generating Java classes from a UML schema is not actually just another form of metaprogramming.

In some way, you've probably done metaprogramming at various points in your career without even knowing it.

The Early History of Metaprogramming

Throughout the history of computer science, various languages have evolved to support different forms of metaprogramming. One of the most ancient is the LISP family of languages, in which the program itself was considered a piece of data and well-known LISP macros were able to be used to extend the languages from within. Other languages relied on deep reflection capabilities to handle such tasks either during compile time or during runtime.

Outside of the LISP family, C and its preprocessor became a tool of choice to generate code from a boilerplate template. Beyond classical function-like macros, the technique known as *X-macros* was a very interesting one. An X-macro is, in fact, a header file containing a list of similar macro invocations—often called the components—which can be included multiple times. Each inclusion is prefixed by the redefinition of said macro to generate different code fragments for the same list of components. A classic example is structure serialization, wherein the X-macros will enumerate structure members, first during definition and then during serialization:

```
// in components.h
PROCESS(float, x      )
```

```
PROCESS(float, y      )
PROCESS(float, z      )
PROCESS(float, weight )

// in particle.c
typedef struct
{
  #define PROCESS(type, member) type member;
  #include "components.h"
  #undef PROCESS
} particle_t;

void save(particle_t const* p, unsigned char* data)
{
  #define PROCESS(type, member)                  \
  memmove(data, &(p->member), sizeof(p->member)); \
  data += sizeof(p->member);                     \
  /**/

  #include "components.h"

  #undef PROCESS
}
```

X-macros are a well-tested, pure C-style solution. Like a lot of C-based solutions, they work quite well and deliver the performance we expect. We could debate the elegance of this solution, but consider that a very similar yet more automated system is available through the Boost.Preprocessor vertical repetition mechanism, based on self-referencing macros.[1]

Enter C++ Templates

Then came C++ and its generic types and functions implemented through the template mechanism. Templates were originally a very simple feature, allowing code to be parameterized by types and integral constants in such a way that more generic code can emerge from a collection of existing variants of a given piece of code. It was quickly discovered that by supporting partial specialization, compile-time equivalents of recursion or conditional statements were feasible. After a short while, Erwin Unruh came up with a very interesting program[2] that builds the list of every prime number

1 See the documentation for details.

2 The original code is available (in German) at *http://www.erwin-unruh.de/primorig.html*.

between 1 and an arbitrary limit. Quite mundane, isn't it? Except that this enumeration was done through warnings at compile time.

Let's take a moment to ponder the scope of this discovery. It meant that we could turn templates into a very crude and syntactically impractical functional language, which later would actually be proven by Todd Veldhuizen to be Turing-complete. If your computer science courses need to be refreshed, this basically meant that, given the necessary effort, any functions computable by a Turing machine (i.e., a computer) could be turned into a compile-time equivalent by using C++ templates. The era of C++ template metaprogramming was coming.

C++ template metaprogramming is a technique based on the use (and abuse) of C++ template properties to perform arbitrary computations at compile time. Even if templates are Turing-complete, we barely need a fraction of this computational power. A classic roster of applications of C++ template metaprogramming includes the following:

- Complex constant computations
- Programmatic type constructions
- Code fragment generation and replication

Those applications are usually backed up by some libraries, like Boost.MPL or Boost.Fusion, and a set of patterns including tag dispatching, recursive inheritance, and SFINAE.[3] All of those components thrived in the C++03 ecosystem and have been used by a large number of other libraries and applications.

The main goal of those components was to provide compile-time constructs with an STL look and feel. Boost.MPL is designed around compile-time containers, algorithms, and iterators. In the same way, Boost.Fusion provides algorithms to work both at compile time and at runtime on tuple-like structures.

3 *Substitution failure is not an error*.

For some reason, even with those familiar interfaces, metaprogramming tools continued to be used by experts and were often overlooked and considered unnecessarily complex. The compilation time of metaprograms was also often criticized as hindering a normal, runtime-based development process.

Most of the critiques you may have heard about template metaprogramming stem from this limitation—which no longer applies, as we will see in the rest of this report.

How to Get Started with Metaprogramming

When making your first forays into metaprogramming, our advice is to experiment on the side with algorithms and type manipulations, as we show in Chapters 2 and 3, and in actual projects, beginning with the simplest thing: static assertions.

Writing metaprograms to do compile-time checks is the safest and simplest thing you can do when getting started. When you are wrong, you will get a compilation-time error or a check will incorrectly pass, but this will not affect the reliability or correctness of your program in any way.

This will also get your mind ready for the day when concepts land in C++.

Checking Integer Type

Some programs or libraries obfuscate the underlying integer type of a variable. Having a compilation error if your assumption is wrong is a great way to prevent difficult-to-track errors:

```
static_assert(std::is_same<obfuscated_int,
                           std::uint32_t>::value,
              "invalid integer detected!");
```

If your obfuscated integer isn't an unsigned 32-bit integer, your program will not compile and the message "invalid integer detected" will be printed.

You might not care about the precise type of the integer—maybe just the size is important. This check is very easy to write:

```
static_assert(sizeof(obfuscated_int) == 4,
    "invalid integer size detected!");
```

Checking the Memory Model

Is an integer the size of a pointer? Are you compiling on a 32-bit or 64-bit platform? You can have a compile-time check for this:

```
static_assert(sizeof(void *) == 8, "expected 64-bit platform");
```

In this case, the program will not compile if the targeted platform isn't 64-bit. This is a nice way to detect invalid compiler/platform usage.

We can, however, do better than that and build a value based on the platform without using macros. Why not use macros? A metaprogram can be much more advanced than a macro, and the error output is generally more precise (i.e., you will get the line where you have the error, whereas with preprocessor macros this is often not the case).

Let's assume that your program has a read buffer. You might want the value of this read buffer to be different if you are compiling on a 32-bit platform or a 64-bit platform because on 32-bit platforms you have less than 3 GB of user space available.

The following program will define a 100 MB buffer value on 32-bit platforms and 1 GB on 64-bit platforms:

```
static const std::uint64_t default_buffer_size =
  std::conditional<sizeof(void *) == 8,
    std::integral_constant<std::uint64_t, 100 * 1024 * 1024>,
    std::integral_constant<std::uint64_t, 1024 * 1024 * 1024>
>::type::value;
```

Here's what the equivalent in macros would be:

```
#ifdef IS_MY_PLATFORM_64
static const std::uint64_t default_buffer_size
    = 100 * 1024 * 1024;
#else
static const std::uint64_t default_buffer_size
    = 1024 * 1024 * 1024;
#endif
```

The macros will silently set the wrong value if you have a typo in the macro value, if you forget a header, or if an exotic platform on which you compile doesn't have the value properly defined.

Also, it is often very difficult to come up with good macros to detect the correct platform (although Boost.Predef has now greatly reduced the complexity of the task).

Summary

Things changed with the advent of C++11 and later C++14, where new language features like variadic lambdas, `constexpr` functions, and many more made the design of metaprograms easier. This report will go over such changes and show you how you can now build a metaprogramming toolbox, understand it, and use it with far greater efficiency and elegance.

So, let's dive in headfirst—we'll start with a short story that demonstrates what you can do.

CHAPTER 2

C++ Metaprogramming in Practice

Let's imagine that you are responsible for the construction—from the ground up—of a brand new module in a big weather prediction system. Your task is to take care of the distribution of complex computations on a large computing grid, while another team has the responsibility for the actual computation algorithms (in a library created two decades previously).

We will see in this chapter what kinds of problems arise when you try to interface two bricks that were created 20 years apart, examine the typical approaches, and see if the template metaprogramming approach brings any benefit.

A Typical Code Maintenance Assignment

After two years of development, your distributed weather system is at last done! You've been very thorough in applying modern C++ principles all along, and took advantage of pass-by-value everywhere you could. You are happy with the performance, the software is now stable, and you've made the design as sound as possible given the time you had.

But now, you need to interface with "the Thing," aka "The Simulation Library of Awesomeness," or SLA for short.

The SLA was designed in the 1990s by developers who have now gone insane or missing. Every time you install the SLA on a system,

it is no longer possible to run any other kind of software without having a team of senior system administrators perform a week-long ritual to cleanse the machine.

Last but not least, the SLA only believes in one god, and that god is *The Great Opaque Pointer*. All interfaces are made as incoherent as possible to ensure that you join the writers in an unnamable crazy laughter, ready to be one with The Great Opaque Pointer.

If you didn't have several years of experience up your sleeve, you would advocate a complete rewrite of the SLA—but you know enough about software engineering to know that "total rewrite" is another name for "suicide mission."

Are we dramatizing? Yes, we are. But let's have a look at a function of the SLA:

```
// we assume alpha and beta to be parameters to the mathematical
// model underlying the weather simulation algorithms--any
// resemblance to real algorithms is purely coincidental
void adjust_values(double * alpha1,
    double * beta1,
    double * alpha2,
    double * beta2);
```

Now let's have a look at how you designed your application:

```
class reading
{
/* stuff */
public:
   double alpha_value(location l, time t) const;
   double beta_value(location l, time t) const;
/* other stuff */
};
```

Let us not try to determine what those alpha and beta values are, whether the design makes sense, or what exactly `adjust_values` does. What we really want to see is how we adapt two pieces of software that have very different logic.

Creating a Straightforward Interface

Interfacing your software with other software is part of your job. It is easy to mock the lack of logic or cleanliness of a program that has been running and maintained for 25 years, but at the end of the day, it must work; no excuses.

In this case, you might be tempted to take a pragmatic approach and just interface functions as needed, with a wrapper like this:

```
std::tuple<double, double, double, double> get_adjusted_values(
    const reading & r,
    location l, time t1, time t2)
{
  double alpha1 = r.alpha_value(l, t1);
  double beta1 = r.beta_value(l, t1);

  double alpha2 = r.alpha_value(l, t2);
  double beta2 = r.beta_value(l, t2);

  adjust_values(&alpha1, &beta1, &alpha2, &beta2);

  return std::make_tuple(alpha1, beta1, alpha2, beta2);
}
```

NOTE

The std::tuple<> pattern

You can see that we use a tuple to "return a bunch of otherwise unrelated stuff." This is a common pattern in modern C++, and later you will see why using tuples has some advantages when it comes to metaprogramming.

But if we look again at the manual approach, we can see a certain number of issues:

- It's error prone because of the interface of the library we are working with. Tests can catch some but not all of these bugs.
- The code is very repetitive; for a couple of functions, it is doable, but a hundred? Or a thousand?
- How do you maintain the code? If there are changes in any of the functions, maintenance costs will grow exponentially.
- What if the names of the functions change? What if the object changes? What if the methods change?

You could retort, "Fine, let's make it generic," as shown here:

```
template <typename Reading>
std::tuple<double, double, double, double> get_adjusted_values(
    const Reading & r,
    location l, time t1, time t2)
{
  double alpha1 = r.alpha_value(l, t1);
  double beta1 = r.beta_value(l, t1);

  double alpha2 = r.alpha_value(l, t2);
  double beta2 = r.beta_value(l, t2);

  adjust_values(&alpha1, &beta1, &alpha2, &beta2);

  return std::make_tuple(alpha1, beta1, alpha2, beta2);
}
```

Sure, it's an improvement, but not a big improvement. To which you will reply, "Fine, let's make the methods generic!" as in this example:

```
template <typename AlphaValue, typename BetaValue>
std::tuple<double, double, double, double> get_adjusted_values(
    AlphaValue alpha_value, BetaValue beta_value,
    location l, time t1, time t2)
{
  double alpha1 = alpha_value(l, t1);
  double beta1 = beta_value(l, t1);

  double alpha2 = alpha_value(l, t2);
  double beta2 = beta_value(l, t2);

  adjust_values(&alpha1, &beta1, &alpha2, &beta2);

  return std::make_tuple(alpha1, beta1, alpha2, beta2);
}
```

And you would call the function as follows:

```
reading r;

// some code

auto res = get_adjusted_values(
    [&r](double l, double t){ return r.alpha_value(l, t); },
    [&r](double l, double t){ return r.beta_value(l, t); },
    /* values */);
```

What we will see here is how we can push this principle of reusability and genericity much further, thanks to template metaprogramming.

What we want to avoid writing is all the systematic code that takes the results from our C++ methods, puts them in the correct form for the C function, passes them to that C function, and gets the results in a form compatible with our C++ framework.

We can call it the boilerplate.

With template metaprogramming techniques, we will make the compiler work for us and avoid a lot of mistakes and tedious work.

Generating Code Automatically

You may be thinking, "I can write a Python script that will generate the code for me." This is indeed doable, if the wrapping code isn't too complex and you will not require a comprehensive C++ parsing. It will increase the complexity of building and maintaining your application, however, because in addition to requiring a compiler, you will now require a scripting language, probably with a certain set of libraries. This kind of solution is another form of automation.

You might also create an abstraction around the library, or at least a facade. You'll still have one problem left, though: you have to write all of the tedious code.

But… computers are very good at repetitive tasks, so why not program the computer to write the facade for you? Wouldn't that greatly increase your productivity?

Why not give it a try? In other words, let's write a program that will generate the program. Let's metaprogram!

Making Values and Pointers Work Together

If we look at the problem from a higher perspective, we see that we have on one side methods working with values, and on the other side functions working with pointers.

The typical C++ approach for a function that takes one parameter and returns one parameter is straightforward:

```
template <typename ValueFunction, typename PointerFunction>
double magic_wand(ValueFunction vf,
                  PointerFunction pf,
                  double param)
{
    double v = vf(param);
```

```
    pf(&v);
    return v;
}
```

We take a callable, `vf`, that accepts a `double` as a parameter and returns a `double` as a parameter. Because we're using a template, we don't need to be specific about what exactly `vf` is (it can be a function, a functor, or a method bound to an object instance).

The callable `pf` accepts a pointer to a `double` as a parameter and updates the value. We then return the updated value.

We called that function `magic_wand` because it's the magic wand that makes your type problem go away!

But the problem is that we have more than one function and more than one parameter. We therefore need to somehow guess the type of the function, manipulate the type to correctly extract values, pass a pointer to these values to the `PointerFunction`, and return the result.

If you pause to think about it, you'll quickly realize that we need two capabilities:

- Type manipulation
- Being able to work on an arbitrary number of parameters and iterate on them

In other words, we'd like to write C++ that modifies types and not values. Template metaprogramming is the perfect tool for compile-time type manipulations.

Let us take a look at a general case. How could we write a program that takes a `double` and transforms it into a pointer to a `double`?

Type Manipulation 101

Since C++11, the standard library has come with a fair number of functions to manipulate types. For example, if you'd like to transform a `double` into a `double *`, you can do this:

```
#include <type_traits>

// double_ptr_type will be double *
using double_ptr_type = std::add_pointer<double>::type;
```

And vice versa:

```
#include <type_traits>

// double_type will be double
using double_type = std::remove_pointer<double *>::type;

// note that removing a pointer from a nonpointer type is safe
// the type of double_too_type is double
using double_too_type = std::remove_pointer<double>::type;
```

These kinds of type manipulations (adding and removing pointers, references, and constness) are basic building blocks and extremely useful when dealing with type constraints. For example, your template parameter might have to be a `const` reference when you actually need a value. With these tools you can ensure that your type is exactly what you need.

A Generic Function Translator

The generic version of the magic wand can take an arbitrary number of functions, concatenate the results into a structure, pass pointers to these results to our legacy C function that will apply the weather model, and return its output.

In other words, in pseudocode, we want something like this:

```
MagicListOfValues generic_magic_want(OldCFunction old_f,
    ListOfFunctions functions,
    ListOfParameters params)
{

  MagicListOfValues values;

  /* wait, something is wrong, we can't do this
  for(auto f : functions)
  {
      values.push_back(f(params));
  }
  */

  olf_f(get_pointers(values));

  return values;
}
```

The only problem is that we can't do that.

Why? The first problem is that we need a collection of values, but those values might have heterogeneous types. Granted, in our example we return `doubles` and we could use a `vector`.

The other problem is a performance issue—why resize the collection at runtime when you know exactly its size at compile time? And why use the heap when you can use the stack?

That's why we like tuples. Tuples allow for heterogeneous types to be stored, their size is fixed at compile time, and they can avoid a lot of dynamic memory allocation.

That raises some questions, though. How do we build these tuples based on the parameters of our legacy C function? How do we iterate on a tuple? How do we work on the list of functions? How do we pass parameters?

Extracting the C Function's Parameters

The first step of the process is, for a given function `F`, to build a tuple matching the parameters.

We will use the pattern matching algorithms of partial template specialization to do that:

```
template <typename F>
struct make_tuple_of_params;

template <typename Ret, typename... Args>
struct make_tuple_of_params<Ret (Args...)>
{
  using type = std::tuple<Args...>;
};

// convenience function
template <typename F>
using make_tuple_of_params_t =
  typename make_tuple_of_params<F>::type;
```

The Magic ... Operator

In C++11, the semantics of the `...` operator have been changed and greatly extended to enable us to say to the compiler, "I expect a list of types of arbitrary length." It has no relationship anymore with the old C ellipsis operator. This operator is a pillar of modern C++ template metaprogramming.

With our new function, we can therefore do the following:

```
template <typename F>
void magic_wand(F f)
{
   // if F is in the form void(double *, double *)
   // make_tuple_of_params is std::tuple<double *, double *>
   make_tuple_of_params_t<F> params;

   // ...
}
```

We now have a tuple of params we can load with the results of our C++ functions and pass to the C function. The only problem is that the C function is in the form `void(double *, double *, double *, double *)`, and we work on values.

We will therefore modify our `make_tuple_of_params` functor accordingly:

```
template <typename Ret, typename... Args>
struct make_tuple_of_derefed_params<Ret (Args...)>
{
   using type = std::tuple<std::remove_ptr_t<Args>...>;
};
```

Hey! What's Going On with the ... Operator?!

When we wrote `Args...`, we somehow expanded the list of parameters inside our `std::tuple`. That's one of the most straightforward uses of the operator.

The general behavior of the `...` operator is to replicate the code fragment on its left for every type in the parameter pack. In this case, the `remove_ptr_t` will be carried along.

For example, if your arguments are:

```
int i, double d, std::string s
```

expansion with `std::tuple<Args...>` will yield:

```
std::tuple<int, double, std::string>
```

and expansion with `std::( tuple <std::( add_( pointer_t<Args>...>` will yield:

```
std::tuple<int *, double *, std::string *>
```

Now the function works as follows:

```
template <typename F>
void magic_wand(F f)
{
   // if F is in the form void(double *, double *)
   // make_tuple_of_params is std::tuple<double, double>
   make_tuple_of_derefed_params<F> params;

   // ...
}
```

We just need to load up the results!

Getting a List of Functions and Parameters

Now that we can extract the contents of the C function's parameters, we need to assemble them in objects that we can manipulate easily in C++.

Indeed, you might be tempted to write this:

```
template <typename Functions, typename Params>
void magic_wand(/* stuff */, Functions... f, Params... p)
{
 // stuff
}
```

After all, you have a list of functions and a list of parameters, and you want both of them. The only problem is, how can the compiler know when the first list ends and the second list begins?

Again, tuples come to the rescue:

```
template <typename... Functions, typename... Params>
void magic_wand(/* stuff */,
    const std::tuple<Functions...> & f,
    const std::tuple<Params...> & p1,
    const std::tuple<Params...> & p2)
{
 // stuff
}
```

This enables the compiler to know that multiple tuples of arbitrary and unrelated lengths are expected. You could, of course, make a tuple of tuples if you expect more than two sets of parameters, but there's no need to make our example more complex than it needs to be.

Performance Warning

Although compilers are getting very good at removing unnecessary copies, and rvalue references help with moving objects, be mindful of what you put inside your tuples and how many of them you create.

Passing the values, in our example, becomes the following:

```
magic_wand(/* stuff */,
   // our C++ functions
   std::make_tuple(
       [&r](double l, double t){ return r.alpha_value(l, t); },
        [&r](double l, double t){ return r.beta_value(l, t); }),
   // first set of params
   std::make_tuple(l, t1),
   // second set of params
   std::make_tuple(l, t2));
```

Which means that inside the body of the `magic_wand` function, we will have tuples containing the functions we need to call and the parameters we need to pass to them.

Filling the Values for the C Function

We've progressed, but we have not arrived. On one hand we have tuples of values to pass to the C function; on the other hand, we have a tuple of functions and parameters.

We now want to fill the tuple of values with the results, which means calling every function inside the tuple and passing the correct parameters:

```
template <typename LegacyFunction,
          typename... Functions,
          typename... Params>
auto magic_wand(
    LegacyFunction legacy,
    const std::tuple<Functions...> & functions,
    const std::tuple<Params...> & params1,
    const std::tuple<Params...> & params2)
{
    make_tuple_of_derefed_params_t<LegacyFunction> params = {
        /* we would like to do
        for(auto f : functions)
        {
            f(params1);
        }
        for(auto f : functions)
```

```
        {
            f(params2);
        }*/
    };

    // rest of the code
}
```

Returning auto

In C++14 you don't need to be explicit about the return type of a function; the type can be determined at compile time contextually. Using `auto` in this case greatly simplifies the writing of generic functions.

In template metaprogramming, there is no iterative construct. You can't iterate on your list of types by using `for`. You can, however, use recursion to apply a callable on every member of the tuple. This approach has been used since 2003 to great effect, but it has the disadvantage of generating a huge amount of intermediate types and therefore increases compilation time.

Whenever you can, you should use the `...` operator to apply a callable to every member of a list. This is faster, it doesn't generate all the unneeded intermediate types, and the code is often more concise.

How can we use the `...` operator for that? Here, we will create a sequence that matches the size of the tuple in order to apply a functor to each member:

```
template <typename F, typename Params, std::size_t... I>
auto dispatch_params(F f,
                     Params & params,
                     std::index_sequence<I...>)
{
    return f(std::get<I>(params)...);
}
```

What happens here is the following:

```
template <typename F, typename Params, std::size_t... I>
auto dispatch_params(F f,
                     Params & params,
                     std::index_sequence<I...>)
{
    // not real C++ code
    return f(std::get<0>(params),
      std::get<1>(params),
      std::get<2>(params),
```

```
        ...,
        std::get<N>(params)); // where N is the last index
}
```

The advantage is that all of the work is done by the compiler and it's much faster than recursion (or macros).

The trick is to create an index sequence—whose sole purpose is to give us an index on which to apply the `...` operator—of the right size. This is done as follows:

```
static const std::size_t params_count = sizeof...(Params);
std::make_index_sequence<params_count>();
```

Compile-Time Size of a List

At compile time, when you need to know how many elements you have in your list, you use `sizeof...()`. Note that in this case we stored that into a `static const` variable, but it would actually be better to use a `std::integral_constant`. You will learn more about this in Chapter 3.

We are getting very close to solving our problem; that is, automating the generation of facade code to adapt the simulation library to our distributed system.

But the problem is not fully solved yet because we need to somehow "iterate" on the functions. We will modify our dispatch function so that it accepts the tuple of functions as a parameter and takes an index, as demonstrated here:

```
template <std::size_t FunctionIndex,
          typename FunctionsTuple,
          typename Params,
          std::size_t... I>
auto dispatch_params(FunctionsTuple & functions,
                     Params & params,
                     std::index_sequence<I...>)
{
    return (std::get<FunctionIndex>(functions))
        (std::get<I>(params)...);
}
```

And we will use the same `index_sequence` trick to call `dispatch_params` on every function of the tuple:

```
template <typename FunctionsTuple,
          std::size_t... I,
```

```
          typename Params,
          typename ParamsSeq>
auto dispatch_functions(FunctionsTuple & functions,
                        std::index_sequence<I...>,
                        Params & params,
                        ParamsSeq params_seq)
{
    return std::make_tuple(dispatch_params<I>(functions,
                                              params,
                                              params_seq)...);
}
```

The previous code enables us to aggregate the result of the successive calls to each element of the tuple into a single tuple.

The final code thus becomes:

```
template <typename LegacyFunction,
          typename... Functions,
          typename... Params>
auto magic_wand(
    LegacyFunction legacy,
    const std::tuple<Functions...> & functions,
    const std::tuple<Params...> & params1,
    const std::tuple<Params...> & params2)
{
    static const std::size_t functions_count =
       sizeof...(Functions);
    static const std::size_t params_count = sizeof...(Params);

    make_tuple_of_derefed_params_t<LegacyFunction> params =
        std::tuple_cat(
            dispatch_functions(functions,
                std::make_index_sequence<functions_count>(),
                params1,
                std::make_index_sequence<params_count>()),
            dispatch_functions(functions,
                std::make_index_sequenc<functions_count>(),
                params2,
                std::make_index_sequence<params_count>()));
    /* rest of the code */
}
```

As you can see, the logic of our function makes generalization to an arbitrary list of parameters possible.

Calling the Legacy C Function

We now have loaded in a tuple the results of our C++ method calls. Now we want to pass a pointer to these values to the C function.

With all the concepts we have seen so far, we know how to solve that problem.

We need to determine the size of our results tuple, which we can do by calling the `std::tuple_size` function (which is compile-time) and do exactly what we've done previously to pass all of the parameters:

```
template <typename F, typename Tuple, std::size_t... I>
void dispatch_to_c(F f, Tuple & t, std::index_sequence<I...>)
{
    f(&std::get<I>(t)...);
}
```

The only twist is that we will take the address to the tuple member because the C function requires a pointer to the value to update. It is safe because `std::get<>` returns a reference to the tuple value.

Here is the completed function:

```
template <typename LegacyFunction,
          typename... Functions,
          typename... Params>
auto magic_wand(
    LegacyFunction legacy,
    const std::tuple<Functions...> & functions,
    const std::tuple<Params...> & params1,
    const std::tuple<Params...> & params2)
{
    static const std::size_t functions_count =
        sizeof...(Functions);
    static const std::size_t params_count = sizeof...(Params);

    using tuple_type =
        make_tuple_of_derefed_params_t<LegacyFunction>;

    tuple_type t =
        std::tuple_cat(
            dispatch_functions(functions,
                std::make_index_sequence<functions_count>(),
                params1,
                std::make_index_sequence<params_count>()),
            dispatch_functions(functions,
                std::make_index_sequenc<functions_count>(),
                params2,
                std::make_index_sequence<params_count>()));

    static const std::size_t t_count =
        std::tuple_size<tuple_type>::value;
    dispatch_to_c(legacy,
        params,
```

```
            std::make_index_sequence<t_count>());
    return params;
}
```

Simplifying the Code

Wouldn't it be nice if we didn't need to specify the type of the result of the tuple concatenation? After all, the compiler knows which kind of tuple it's going to be. But in that case, how could we compute the size of the resulting tuple?

We can use the `decltype` directive to access the type of a variable:

```
auto val = /* something */;

decltype(val) // get type of val
```

This simplifies the code and removes the need for the `make_tuples_of_params_t` functor, as shown here:

```
template <typename LegacyFunction,
          typename... Functions,
          typename... Params>
auto magic_wand(LegacyFunction legacy,
                const std::tuple<Functions...> & functions,
                const std::tuple<Params...> & params1,
                const std::tuple<Params...> & params2)
{
    static const std::size_t functions_count =
        sizeof...(Functions);
    static const std::size_t params_count =
        sizeof...(Params);

    auto params = std::tuple_cat(
        dispatch_functions(functions,
            std::make_index_sequence<functions_count>(),
            params1,
            std::make_index_sequence<params_count>()),
        dispatch_functions(functions,
            std::make_index_sequence<functions_count>(),
            params2,
            std::make_index_sequence<params_count>()));

    static constexpr auto t_count =
        std::tuple_size<decltype(params)>::value;

    dispatch_to_c(legacy,
                  params,
                  std::make_index_sequence<t_count>());
```

```
    return params;
}
```

You could also improve the efficiency of the code by using rvalue references and ensuring that you use perfect forwarding semantics.

Putting It All Together

How can we use what we've built to finalize facade generation?

For clarity, we will use an explicit return type, but we could use `auto`. Using an explicit return type has the advantage of generating a compilation error if your type conversions are incorrect.

Another important reason for this decision is that we can consider `get_adjusted_values` as a public API function. Using an `auto` return type makes the function more difficult to use because its return type isn't clear. Your users aren't compilers!

Let's have a look at the code:

```
template <typename Reading>
std::tuple<double, double, double, double>
get_adjusted_values(Reading & r,
                    location l,
                    time t1,
                    time t2)
{
    return magic_wand(adjust_values,
         std::make_tuple(
             [&r](double l, double t)
             {
                 return r.alpha_value(l, t);
             },
             [&r](double l, double t)
             {
                 return r.beta_value(l, t);
             }),
         std::make_tuple(l, t1),
         std::make_tuple(l, t2));
}
```

The power of this new function is that if the legacy C function or the C++ object changes, there will be little to no code rewriting to be done.

Writing the wrappers will also be extremely straightforward, safe, and productive: just call the `magic_wand` function with the required

values. You can make it even more generic by wrapping the parameters in other functors and deducing the right types as needed.

And guess what? It's also possible to write code to generate all the wrappers for you based on the function profiles. We've seen in this chapter all of the building blocks to achieve that.

Summary

Did we accomplish our mission? We'd like to believe that, yes, we did.

With the use of a couple of template metaprogramming tricks, we managed to drastically reduce the amount of code required to get the job done. That's the immediate benefit of automating code generation. Less code means fewer errors, less testing, less maintenance, and potentially better performance.

This is the strength of metaprogramming. You spend more time carefully thinking about a small number of advanced functions, so you don't need to waste your time on many trivial functions.

Now that you have been exposed to template metaprogramming, you probably have many questions. How can I check that my parameters are correct? How can I get meaningful error messages if I do something wrong? How can I store a pure list of types, without values?

More importantly, can these techniques be made reusable?

Let's take it from the beginning…

CHAPTER 3

C++ Metaprogramming and Application Design

Chapter 2 gave you a taste of how powerful type manipulation can be and how it can make code more compact, generic, and elegant, and less error prone.

As in all automation systems, metaprogramming requires some tools to avoid rewriting the same code over and over. This chapter will go over the basic components of such a toolbox. We also will try to extract *axioms* (small, self-contained elements of knowledge) to apply in our everyday jobs as metaprogram developers.

Compile-Time Versus Runtime Paradigms

C++ runtime code is based on the fact that users can define functions that will operate on values represented by some data type (either native or user defined) to produce new values or side effects. A C++ program is then the orchestration of said function calls to advance the application's goal at runtime.

If those notions of functions, values, and types defined by runtime behavior are pretty straightforward, things become blurry when speaking about their *compile-time* equivalents.

Values at Compile Time

Compile-time computations need to operate on values defined as valid at compile time. Here's what this notion of compile-time values covers:

- Types, which are entities defined at compile time
- Integral constants, defined by using the `constexpr` keyword or passed to the template class as template parameters.

Unfortunately, types and integral constants are two completely different beasts in C++. Moreover, there is currently no way in C++11/14/17 to specify a template parameter to be either a type or an integral constant.[1] To be able to work with both kinds of values (types and integral constants), we need a way to make both kinds of values homogeneous.

The standard way to do so, which stems from Boost.MPL, is to wrap the value in a type. We can do this by using `std::integral_constant`, which we can implement roughly as follows:

```
template<typename T, T Value>
struct integral_constant
{
  using type = T;
  static constexpr T value = Value;
};
```

This structure is a simple box containing a value. This box's type is dependent on both the value and the value type, making it unambiguous.[2] We can later retrieve either the value type through the `::type` internal type definition or the numerical value through the `::value` constant.

Because integral constants can be turned easily into types, we can consider that the only required flavor of compile-time values is type. This is such a strong proposition that we will define this as one of our basic axioms in metaprogramming.

1 If you think it could be useful, feel free to write such a proposal.

2 If you're an astute reader with a more theoretical background, you might have recognized a crude implementation of Church numerals.

NOTE

Meta-Axiom #1

Types are first-class values inside compile-time programs.

Functions at Compile Time

We can view runtime functions as mappings between values of some types and results of a given type (which might be `void`). In a perfect world, such functions would behave the same way as their mathematical, pure equivalents, but in some cases we might need to trigger side effects (like I/O or memory access).

Functions at compile time live in a world where no side effects can occur. They are, by their very nature, pure functions living in a small functional language embedded in C++. As per our first axiom, the only interesting values in metaprogramming are types. Thus, compile-time functions are components that map types to other types.

This statement looks familiar. It is basically the same definition as that for runtime functions, with "values" replaced by "types." The question now is, how can we specify such a component when it's clear C++ syntax won't let us write `return float` somewhere?

Again, we take advantage of the pioneering work of Boost.MPL by reusing its notion of the *metafunction*. Quoting from the documentation, a metafunction is "*a class or a class template that represents a function invocable at compile-time*." Such classes or class templates follow a simple protocol. The inputs of the metafunctions are passed as template parameters and the returned type is provided as an internal type definition.

A simple metafunction can be written as follows:

```
template<class T>
struct as_constref
{
  using type = T const&;
};
```

As its name implies, this metafunction turns a type into a reference to a constant value of said type.[3] Invoking a metafunction is just a matter of accessing its internal `::type`, as demonstrated here:

```
using cref = as_constref<float>::type;
```

This principle has already leaked from MPL into the standard. The `type_traits` header provides a large number of such metafunctions, supporting for analyzing, creating, or modifying types based on their properties.

Pro Tip

Most of the basic needs for type manipulation are provided by `type_traits`. We strongly advise any metaprogrammer-in-training to become highly familiar with this standard component.

Type Containers

C++ runtime development relies on the notion of containers to express complex data manipulations. Such containers can be defined as data structures holding a variable number of values and following a given schema of storage (contiguous cells, linked cells, and so on). We can then apply operations and algorithms to containers to modify, query, remove, or insert values. The STL provides pre-made containers, like `list`, `set`, and `vector`.

How can we end up with a similar concept at compile time? Obviously, we cannot request memory to be allocated to store our values. Moreover, our "values" actually being types, such storage makes little sense. The logical leap we need to make is to understand that containers are also values, which happen to contain zero or more other values; if we apply our systematic "values are types" motto, this means that compile-time containers must be *types* that contain zero or more other types. But how can a type contain another type? There are multiple solutions to this issue.

3 Let's ignore the potential issue of adding a reference to a reference type.

The first idea could be to have a compile-time container be a type with a variable number of internal `using` statements, as in the following example:

```
struct list_of_ints
{
  static constexpr std::size_t size = 4;
  using element0 = char;
  using element1 = short;
  using element2 = int;
  using element3 = long;
};
```

There are a few issues with this solution, though. First, there is no way to add or remove types without having to construct a new type. Then, accessing a given type is complex because it requires us to be able to map an integral constant to a type name.

Another idea is to use *variadic templates* to store types as the parameter pack of a variadic type. Our `list_of_ints` then becomes the following:

```
template<typename... Values> struct meta_list {};
using list_of_ints = meta_list<chr,short,int,long>;
```

This solution has neither of the aforementioned drawbacks. Operations on this `meta_list` can be carried out by using the intrinsic properties of the parameter pack, because no name mapping is required. Insertion and removal of elements is intuitive; we just need to play with the contents of the parameter pack.

Those properties of variadic templates define our second axiom of metaprogramming: the fact that any variadic template structure is de facto a compile-time container.

Meta-Axiom #2

Any template class accepting a variable number of type parameters can be considered a type container.

Compile-Time Operations

We now have defined type containers as arbitrary template classes with at least a template parameter pack parameter. Operations on such containers are defined by using the intrinsic C++ support for template parameter packs.

We can do all of the following:

- Retrieve information about the pack.
- Expand or shrink the pack's contents.
- Rewrap the parameter pack.
- Apply operations on the parameter pack's contents.

Using Pack-Intrinsic Information

Let's try to make a simple metafunction that operates on a type container by writing a way to access a container's size:

```
template<class List> struct size;

template<template<class...> class List, class... Elements>
struct size<List<Elements...>>
     : std::integral_constant<std::size_t, sizeof...(Elements)>
{};
```

Let's go over this snippet. First, we declare a `size` structure that takes exactly one template parameter. At this point, the nature of this parameter is unknown; thus, we can't give `size` a proper definition. Then, we partially specialize `size` for all of the types of the form `List<Elements...>`. The syntax is a bit daunting, so let's decompose it. The template parameters of this specialization comprise the following:

`List`
: A template template parameter awaiting a template parameter pack as an argument

`Elements`
: A template parameter pack

From those two parameters, we specialize `size<Elements...>`. We can write this as `Elements...`, which will trigger an expansion of every type in the pack, which is exactly what `List` requires in its own parameters. This technique of describing the variadic structure of a type container so that an algorithm can be specified will be our main tool from now on.

Take a look at how we can use this compile-time algorithm and how the compiler interprets this call. Consider the following as we try to evaluate the size of `std::tuple<int,float,void>`:

```
constexpr auto s = size<std::tuple<int,float,void>>::value;
```

By the definition of `std::tuple`, this call will match the `size<List<Elements...>>` specialization. Rather trivially, `List` will be substituted by `std::tuple` and `Elements` will be substituted by the parameter pack `{int, float, void}`. When it is there, the `sizeof...` operator will be called and will return `3`. `size` will then inherit publicly from `std::integral_constant<std::size_t,3>` and forward its internal value constant. We could have used any kind of variadic structure instead of a tuple, and the process would have been similar.

Adding and Removing Pack Elements

The next natural step is to try to modify the elements inside a type container. We can do this by using the structural description of a parameter pack. As an example, let's try to write `push_back<List,Element>`, which inserts a new element at the end of a given type container.

The implementation starts in a now-familiar way:

```
template<class List, class New> struct push_back;
```

As for `size`, we declare a `push_back` structure with the desired type interface but no definition. The next step is to specialize this type so that it can match type containers and proceed:

```
template<template<class...> class List,
class... Elements, class New>

struct push_back<List<Elements...>, New>
{
  using type = List<Elements...,New>;
};
```

As compile-time metaprogramming has no concept of values, our only way to add an element to an existing type container is to rebuild a new one. The algorithm is pretty simple: expand the existing parameter pack inside the container and add one more element at the end. By the definitions of `List` and `...`, this builds a new valid type where the `New` element has been inserted at the end.

Exercise

Can you infer the implementation for `push_front`?

Removal of existing elements in a type container follows a similar reasoning but relies on the recursive structure of the parameter pack. Fear not! As we said earlier, recursion in template metaprogramming is usually ill advised, but here we will only exploit the structure of the parameter pack and we won't do any loops. Let's begin with the bare-bones code for a hypothetical `remove_front` algorithm:

```
template<class List> struct remove_front;

template<template<class...> class List, class... Elements>
struct remove_front<List<Elements...>>
{
  using type = List</* what goes here??? */>;
};
```

As you can see, we haven't diverged much from what we've seen so far. Now, let's think about how we can remove the first type of an arbitrary parameter pack so that we can complete our implementation. Let's enumerate the cases:

`List<Elements...>`
: Contains at least one element (the head) and a potentially empty pack of other types (the tail). In this case, we can write it as `List<Head, Tail...>`

`List<Elements...>`
: This is empty. In this case, it can be written as `List<>`.

If we know that a head type exists, we can remove it. If the list is empty, the job is already done. The code then reflects this process:

```
template<class List> struct remove_front;

template<template<class...> class List
        , class Head, class... Elements>
struct remove_front<List<Head,Elements...>>
{
  using type = List<Elements...>;
};

template<template<class...> class List>
struct remove_front<List<>>
{
  using type = List<>;
};
```

This introspection of the recursive nature of the parameter pack is another tool in our belt. It has some limitations, given that decomposing a pack into a list of heads and a single tail type is more complex, but it helps us build basic blocks that we can reuse in more complex contexts.

Pack Rewrapping

So far, we've dealt mostly with accessing and mutating the parameter pack. Other algorithms might need to work with the enclosing type container.

As an example, let's write a metafunction that turns an arbitrary type container into a `std::tuple`. How can we do that? Because the difference between `std::tuple<T...>` and `List<T...>` is the enclosing template type, we can just change it, as shown here:

```
template<class List> struct as_tuple;

template<template<class..> class List, class... Elements>
struct as_tuple<List<Elements...>>
{
  using type = std::tuple<Elements...>;
};
```

But wait: there's more! Changing the type container to `tuple` or `variant` or anything else can actually be generalized by passing the new container type as a parameter. Let's generalize `as_tuple` into `rename`:

```
struct rename;

template<template<class..> class Container
        , template<class..> class List
        , class... Elements
        >
struct rename<Container, List<Elements...>>
{
  using type = Container<Elements...>;
};
```

The code is rather similar. We use the fact that a template template parameter can be passed naturally to provide `rename` with its actual target. A sample call can then be as follows:

```
using my_variant = rename<boost::variant
                          , std::tuple<int,short>
                          >;
```

NOTE

This technique was explained by Peter Dimov in his blog in 2015 and instigated a lot of discussion around similar techniques.

Container Transformations

These tools—rewrapping, iteration, and type introspection for type containers—lead us to the final and most interesting metaprograms: container transformations. Such transformations, directly inspired by the STL algorithms, will help introduce the concept of structured metaprogramming.

Concatenating containers

A first example of transformation is the concatenation of two existing type containers. Considering any two lists `L1<T1...>` and `L2<T2...>`, we wish to obtain a new list equivalent to `L1<T1...,T2...>`.

The first intuition we might have coming from our runtime experience is to find a way to "loop" over types as we repeatedly call `push_back`. Even if it's a correct implementation, we need to fight this compulsion of thinking with loops. Loops over types will require a linear number of intermediate types to be computed, leading to unsustainable compilation times. The correct way of handling this use case is to find a natural way to exploit the variadic nature of our containers.

In fact, we can look at `append` as a kind of rewrapping in which we push into a given variadic structure more types than it contained before. A sample implementation can then be as follows:

```
template<typename L1, typename L2> struct append;

tempate< template<class...> class L1, typename... T1
       , template<class...> class L2, typename... T2
       >
struct append< L1<T1...>, L2<T2...> >
{
  using type = L1<T1...,T2...>;
};
```

After the usual declaration, we define `append` as awaiting two different variadic structures filled with two distinct parameter packs. Note that, as with regular specialization on nonvariadic templates, we can use multiple parameter packs as long as they are wrapped properly in the specialization. We now have access to all of the elements required. The result is computed as the first variadic type instantiated with both parameter packs expanded.

Pro Tip

Dealing with compile-time containers requires no loops. Try to express your algorithm as much as possible as a direct manipulation of parameter packs.

Toward a compile-time transform

The `append` algorithm was rather straightforward. Let's now hop to a more complex example: a compile-time equivalent to `std::transform`. Let's first state what the interface of such a metaprogram could be. In the runtime world, `std::transform` calls a callable object over each and every value of the target container and fills another container with the results. Again, this must be transposed to a metafunction that will iterate over types inside a parameter pack, apply an arbitrary metafunction, and generate a new parameter pack to be returned.

Even if "iterating over the contents of a parameter pack using `...`" is a well-known exercise, we need to find a way to pass an arbitrary metafunction to our compile-time `transform` variant. A runtime callable object is an object providing an overload for the so-called function call operator—usually denoted `operator()`. Usually those objects are regular functions, but they can also be anonymous functions (aka lambda functions) or full-fledged user-defined classes providing such an interface.

Generalizing metafunctions

In the compile-time world, we can pass metafunctions directly by having our `transform` metaprogram await a template template parameter. This is a valid solution, but as for runtime functions, we might want to bind arbitrary parameters of existing metafunctions to maximize code reuse.

Let's introduce the Boost.MPL notion of the *metafunction class*. A metafunction class is a structure, which might or might not be a template, that contains an internal template structure named `apply`. This internal metafunction will deal with actually computing our new type. In a way, this `apply` is the equivalent of the generalized `operator()` of callable objects. As an example, let's turn `std::remove_ptr` into a metafunction class:

```
struct remove_ptr
{
  template<typename T> struct apply
  {
    using type = typename std::remove_ptr<T>::type;
  };
};
```

How can we use this so-called metafunction class? It's a bit different than with metafunctions:

```
using no_ptr = remove_ptr::apply<int*>::type;
```

Note the requirement of accessing the internal `apply` template structure. Wrapping this so that the end user is shielded from complexity is tricky.

Note how the metafunction class is no longer a template but relies on its internal `apply` to do its bidding. If you're an astute reader, you will see that we can generalize this to convert any metafunction into a metafunction class. Let's introduce the `lambda` metafunction:

```
template<template<class...> class MetaFunction>
struct lambda
{
  struct type
  {
    template<typename Args...> struct apply
    {
      using type = typename MetaFunction<Args...>::type;
    };
  };
};
```

This `lambda` structure is indeed a metafunction because it contains an internal `type` to be retrieved. This type structure is using pack expansion to adapt the template template parameter of `lambda` so that its usage is correct. Notice also that, like runtime lambda functions, this internal type is actually anonymous.

Implementing transform

We now have a proper protocol to pass metafunctions to our compile-time `transform`. Let's write a unary `transform` that works on type containers:

```
template<typename List, typename F> struct transform;

template<template<class...> class List,
struct transform<List<Elems...>,F>
{
  using call = typename F::template apply<T>::type;
  using type = List< call<Elems>... >;
};
```

This code is both similar to what we wrote earlier and a bit more complex. It begins as usual by declaring and defining `transform` as acting on container types using a parameter pack. The actual code performs iterations over elements of the container using the classic `...` approach. The addition we need to make is to call the metafunction class `F` over each type. We do this by taking advantage of the fact that `...` will unpack and apply the code fragment on its left for every type in the pack. For clarity, we use an intermediate template using a statement to hold the actual metafunction class application to a single type.

Now, as an example, let's call `std::remove_ptr` on a list of types:

```
using no_pointers = transform< meta_list<int*,float**, double>
                             , lambda<std::remove_ptr>
                             >::type;
```

Note the abstraction power of algorithms being transposed to the compile-time world. Here we used a high-level metafunction to apply a well-known pattern of computation on a container of types. Observe also how the `lambda` construct can help us make the use and reuse of existing metafunctions easier.

Pro Tip

Metafunctions follow similar rules to those for functions: they can be composed, bound, or turned into various similar yet different interfaces. The transition between metafunctions and metafunction classes is only the tip of the iceberg.

Advanced Uses of Metaprogramming

With a bit of imagination and knowledge, you can do things much more advanced than performing compile-time checks with template metaprogramming. The purpose of this section is just to give you an idea of what is possible.

A Revisited Command Pattern

The command pattern is a behavioral design pattern in which you encapsulate all the information required to execute a command into an object or structure. It's a great pattern, which in C++ is often written with runtime polymorphism.

Putting aside the tendency of runtime polymorphism to induce the "factory of factories" antipattern, there can be a nonnegligible performance cost induced by vtables because they prevent the compiler from aggressively optimizing and inlining code.

The "Factory of Factories" Antipattern

This antipattern can happen in object-oriented programming when you spend more time writing code to manage abstractions than you do writing code to solve problems.

From a strictly software design point of view, it also forces you to relate objects together just because they will go through the same function at some point in time.

If generic programming has taught us anything, it's that you don't need to create a relation between objects just to make them use a function.

All you need to do is make the objects share common properties:

```
struct first_command
{
    std::string operator()(int) { /* something */ }
};

struct second_command
{
    std::string operator()(int) { /* something */ }
};
```

And have a function that accepts a command:

```
template <typename Command>
void execute_command(const Command & c, int param)
{
    c(param);
}
```

To which you will retort, "How do I transmit those commands through a structure, given that I know only at *runtime* which command to run?"

There are two ways to do it: manually, by using an unrestricted union, or by using a variant such as Boost.Variant. Template metaprogramming comes to the rescue because you can safely list the types of the commands in a type list and build the variant (or the union) from that list.

Not only will the code be more concise and more efficient, but it will also be less error prone: at compile time you will get an error if you forgot to implement a function, and the "pure virtual function call" is therefore impossible.[4]

Compile-Time Serialization

What do we mean by compile-time serialization? When you want to serialize an object, there are a lot of things you already know at compile time—and remember, everything you do at compile time doesn't need to be done any more at runtime.

That means much faster serialization and more efficient memory usage.

For example, when you want to serialize a `std::uint64_t`, you know exactly how much memory you need, whereas when you serialize a `std::vector<std::uint64_t>`, you must read the size of the `vector` at runtime to know how much memory you need to allocate.

Recursively, it means that if you serialize a structure that is made up strictly of integers, you are able, at *compile time*, to know exactly how much memory you need, which means you can allocate the required intermediate buffers at compile time.

4 To be fair, C++11 introduced a series of enhancements that allow the programmer to ensure at compile time that she hasn't forgotten to implement a virtual function. That doesn't eliminate the risk of invalid dynamic casts, though.

With template metaprogramming, you can branch, at compile time, the right code for serialization. This means that for every structure for which you are able to exactly compute the memory requirements, you will avoid dynamic memory allocation altogether, yielding great performance improvements and reduced memory usage.

Helper Functions and Libraries

Must you reinvent the wheel and write all your own basic functions, like we've seen in this chapter? Fortunately, no. Since C++11, a great number of helper functions have been included in the standard, and we strongly encourage you to use them whenever possible.

The standard isn't yet fully featured when it comes to metaprogramming; for example, it lacks an official "list of types" type, algorithms, and more advanced metafunctions.

Fortunately, there are libraries that prevent you from needing to reinvent the wheel and that will work with all major compilers. This will save you the sweat of working around compilers' idiosyncrasies and enable you to focus on writing metaprograms.

Boost comes with two libraries to help you with template metaprogramming:

MPL, written by Aleksey Gurtovoy and David Abrahams
: A complete C++03 template metaprogramming toolbox that comes with containers, algorithms, and iterators. Unless you are stuck with a C++03 compiler, we would recommend against using this library.

Hana, written by Louis Dionne
: A new metaprogramming paradigm, which makes heavy use of lambdas. Hana is notoriously demanding of the compiler.

The authors of this report are also the authors of Brigand, a C++11/14 metaprogramming library that nicely fills the gap between Boost.MPL and Boost.Hana.

We strongly encourage you to use existing libraries because they will help you structure your code and give you ideas of what you can do with metaprogramming.

Summary

In this chapter, we took a journey into the land of types within C++ and we saw that they can be manipulated like runtime values.

We defined the notion of type values and saw how such a notion can lead to the definition of type containers; that is, types containing other types. We saw how the expressiveness of parameter packs can lead to a no-recursion way of designing metaprograms. The small yet functional subset of classic container operators we defined showed the variety of techniques usable to design such metaprograms in a systemic way.

We hope we reached our goal of giving you a taste for metaprogramming and proving that it isn't just some arcane technique that should not be used outside of research institutes. Whether you want to check out the libraries discussed herein, write your own first metaprogram, or revisit some code you wrote recently, we have only one hope: that by reading this report you learned something that will make you a better programmer.

About the Authors

Edouard Alligand is the founder and CEO of Quasardb, an advanced, distributed hyper scalable database. He has more than 15 years of professional experience in software engineering. Edouard combines an excellent knowledge of low-level programming with a love for template metaprogramming, and likes to come up with uncompromising solutions to seemingly impossible problems. He lives in Paris, France.

Joel Falcou is CTO of NumScale, an Associate Professor at the University of Paris-Sud, and a researcher at the Laboratoire de Recherche d'Informatique in Orsay, France. He is a member of the C++ Standards Committee and the author of Boost.SIMD and NT2. Joel's research focuses on studying generative programming idioms and techniques to design tools for parallel software development.