Covers C++11, C++14, and C++17

C++ Templates

The Complete Guide

SECOND EDITION

David VANDEVOORDE
Nicolai M. JOSUTTIS
Douglas GREGOR

C++ Templates

Second Edition

This page intentionally left blank

C++ Templates

The Complete Guide

Second Edition

David Vandevoorde
Nicolai M. Josuttis
Douglas Gregor

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Catalog Number: 2017946531

This book was typeset by Nicolai M. Josuttis using the LaTeX document processing system.

ISBN-13: 978-0-321-71412-1
ISBN-10: 0-321-71412-1

1 17

To Alessandra & Cassandra
—David

To those who care
for people and mankind
—Nico

To Amy, Tessa & Molly
—Doug

This page intentionally left blank

Contents

Preface

The notion of templates in C++ is over 30 years old. C++ templates were already documented in 1990 in "The Annotated C++ Reference Manual" (ARM; see [*EllisStroustrupARM*]), and they had been described before then in more specialized publications. However, well over a decade later, we found a dearth of literature that concentrates on the fundamental concepts and advanced techniques of this fascinating, complex, and powerful C++ feature. With the first edition of this book, we wanted to address this issue and decided to write *the* book about templates (with perhaps a slight lack of humility).

Much has changed in C++ since that first edition was published in late 2002. New iterations of the C++ standard have added new features, and continued innovation in the C++ community has uncovered new template-based programming techniques. The second edition of this book therefore retains the same goals as the first edition, but for "Modern C++."

We approached the task of writing this book with different backgrounds and with different intentions. David (aka "Daveed"), an experienced compiler implementer and active participant of the C++ Standard Committee working groups that evolve the core language, was interested in a precise and detailed description of all the power (and problems) of templates. Nico, an "ordinary" application programmer and member of the C++ Standard Committee Library Working Group, was interested in understanding all the techniques of templates in a way that he could use and benefit from them. Doug, a template library developer turned compiler implementer and language designer, was interested in collecting, categorizing, and evaluating the myriad techniques used to build template libraries. In addition, we all wanted to share this knowledge with you, the reader, and the whole community to help to avoid further misunderstanding, confusion, or apprehension.

As a consequence, you will see both conceptual introductions with day-to-day examples and detailed descriptions of the exact behavior of templates. Starting from the basic principles of templates and working up to the "art of template programming," you will discover (or rediscover) techniques such as static polymorphism, type traits, metaprogramming, and expression templates. You will also gain a deeper understanding of the C++ standard library, in which almost all code involves templates.

We learned a lot and we had much fun while writing this book. We hope you will have the same experience while reading it. Enjoy!

This page intentionally left blank

Acknowledgments for the Second Edition

Writing a book is hard. Maintaining a book is even harder. It took us more than five years—spread over the past decade—to come up with this second edition, and it couldn't have been done without the support and patience of a lot of people.

First, we'd like to thank everyone in the C++ community and on the C++ standardization committee. In addition to all the work to add new language and library features, they spent many, many hours explaining and discussing their work with us, and they did so with patience and enthusiasm.

Part of this community also includes the programmers who gave feedback for errors and possible improvement for the first edition over the past 15 years. There are simply too many to list them all, but you know who you are and we're truly grateful to you for taking the time to write up your thoughts and observations. Please accept our apologies if our answers were sometimes less than prompt.

We'd also like to thank everyone who reviewed drafts of this book and provided us with valuable feedback and clarifications. These reviews brought the book to a significantly higher level of quality, and it again proved that good things need the input of many "wise guys." For this reason, huge thanks to Steve Dewhurst, Howard Hinnant, Mikael Kilpeläinen, Dietmar Kühl, Daniel Krügler, Nevin Lieber, Andreas Neiser, Eric Niebler, Richard Smith, Andrew Sutton, Hubert Tong, and Ville Voutilainen.

Of course, thanks to all the people who supported us from Addison-Wesley/Pearson. These days, you can no longer take professional support for book authors for granted. But they were patient, nagged us when appropriate, and were of great help when knowledge and professionalism were necessary. So, many thanks to Peter Gordon, Kim Boedigheimer, Greg Doench, Julie Nahil, Dana Wilson, and Carol Lallier.

A special thanks goes to the LaTeX community for a great text system and to Frank Mittelbach for solving our LaTeX issues (it was almost always our fault).

David's Acknowledgments for the Second Edition

This second edition was a long time in the waiting, and as we put the finishing touches to it, I am grateful for the people in my life who made it possible despite many obligations vying for attention. First, I'm indebted to my wife (Karina) and daughters (Alessandra and Cassandra), for agreeing to let me take significant time out of the "family schedule" to complete this edition, particularly in the last year of work. My parents have always shown interest in my goals, and whenever I visit them, they do not forget this particular project.

Clearly, this is a technical book, and its contents reflect knowledge and experience about a programming topic. However, that is not enough to pull off completing this kind of work. I'm therefore extremely grateful to Nico for having taken upon himself the "management" and "production" aspects of this edition (in addition to all of his technical contributions). If this book is useful to you and you run into Nico some day, be sure to tell him thanks for keeping us all going. I'm also thankful to Doug for having agreed to come on board several years ago and to keep going even as demands on his own schedule made for tough going.

Many programmers in our C++ community have shared nuggets of insight over the years, and I am grateful to all of them. However, I owe special thanks to Richard Smith, who has been efficiently answering my e-mails with arcane technical issues for years now. In the same vein, thanks to my colleagues John Spicer, Mike Miller, and Mike Herrick, for sharing their knowledge and creating an encouraging work environment that allows us to learn ever more.

Nico's Acknowledgments for the Second Edition

First, I want to thank the two hard-core experts, David and Doug, because, as an application programmer and library expert, I asked so many silly questions and learned so much. I now feel like becoming a core expert (only until the next issue, of course). It was fun, guys.

All my other thanks go to Jutta Eckstein. Jutta has the wonderful ability to force and support people in their ideals, ideas, and goals. While most people experience this only occasionally when meeting her or working with her in our IT industry, I have the honor to benefit from her in my day-to-day life. After all these years, I still hope this will last forever.

Doug's Acknowledgments for the Second Edition

My heartfelt thanks go to my wonderful and supportive wife, Amy, and our two little girls, Molly and Tessa. Their love and companionship bring me daily joy and the confidence to tackle the greatest challenges in life and work. I'd also like to thank my parents, who instilled in me a great love of learning and encouraged me throughout these years.

It was a pleasure to work with both David and Nico, who are so different in personality yet complement each other so well. David brings a great clarity to technical writing, honing in on descriptions that are precise and illuminating. Nico, beyond his exceptional organizational skills that kept two coauthors from wandering off into the weeds, brings a unique ability to take apart a complex technical discussion and make it simpler, more accessible, and far, far clearer.

Acknowledgments for the First Edition

This book presents ideas, concepts, solutions, and examples from many sources. We'd like to thank all the people and companies who helped and supported us during the past few years.

First, we'd like to thank all the reviewers and everyone else who gave us their opinion on early manuscripts. These people endow the book with a quality it would never have had without their input. The reviewers for this book were Kyle Blaney, Thomas Gschwind, Dennis Mancl, Patrick Mc Killen, and Jan Christiaan van Winkel. Special thanks to Dietmar Kühl, who meticulously reviewed and edited the whole book. His feedback was an incredible contribution to the quality of this book.

We'd also like to thank all the people and companies who gave us the opportunity to test our examples on different platforms with different compilers. Many thanks to the Edison Design Group for their great compiler and their support. It was a big help during the standardization process and the writing of this book. Many thanks also go to all the developers of the free GNU and egcs compilers (Jason Merrill was especially responsive) and to Microsoft for an evaluation version of Visual C++ (Jonathan Caves, Herb Sutter, and Jason Shirk were our contacts there).

Much of the existing "C++ wisdom" was collectively created by the online C++ community. Most of it comes from the moderated Usenet groups `comp.lang.c++.moderated` and `comp.std.c++`. We are therefore especially indebted to the active moderators of those groups, who keep the discussions useful and constructive. We also much appreciate all those who over the years have taken the time to describe and explain their ideas for us all to share.

The Addison-Wesley team did another great job. We are most indebted to Debbie Lafferty (our editor) for her gentle prodding, good advice, and relentless hard work in support of this book. Thanks also go to Tyrrell Albaugh, Bunny Ames, Melanie Buck, Jacquelyn Doucette, Chanda Leary-Coutu, Catherine Ohala, and Marty Rabinowitz. We're grateful as well to Marina Lang, who first sponsored this book within Addison-Wesley. Susan Winer contributed an early round of editing that helped shape our later work.

Nico's Acknowledgments for the First Edition

My first personal thanks go with a lot of kisses to my family: Ulli, Lucas, Anica, and Frederic supported this book with a lot of patience, consideration, and encouragement.

In addition, I want to thank David. His expertise turned out to be incredible, but his patience was even better (sometimes I ask really silly questions). It is a lot of fun to work with him.

David's Acknowledgments for the First Edition

My wife, Karina, has been instrumental in this book coming to a conclusion, and I am immensely grateful for the role that she plays in my life. Writing "in your spare time" quickly becomes erratic when many other activities vie for your schedule. Karina helped me to manage that schedule, taught me to say "no" in order to make the time needed to make regular progress in the writing process, and above all was amazingly supportive of this project. I thank God every day for her friendship and love.

I'm also tremendously grateful to have been able to work with Nico. Besides his directly visible contributions to the text, his experience and discipline moved us from my pitiful doodling to a well-organized production.

John "Mr. Template" Spicer and Steve "Mr. Overload" Adamczyk are wonderful friends and colleagues, but in my opinion they are (together) also the ultimate authority regarding the core C++ language. They clarified many of the trickier issues described in this book, and should you find an error in the description of a C++ language element, it is almost certainly attributable to my failing to consult with them.

Finally, I want to express my appreciation to those who were supportive of this project without necessarily contributing to it directly (the power of cheer cannot be understated). First, my parents: Their love for me and their encouragement made all the difference. And then there are the numerous friends inquiring: "How is the book going?" They, too, were a source of encouragement: Michael Beckmann, Brett and Julie Beene, Jarran Carr, Simon Chang, Ho and Sarah Cho, Christophe De Dinechin, Ewa Deelman, Neil Eberle, Sassan Hazeghi, Vikram Kumar, Jim and Lindsay Long, R.J. Morgan, Mike Puritano, Ragu Raghavendra, Jim and Phuong Sharp, Gregg Vaughn, and John Wiegley.

About This Book

The first edition of this book was published almost 15 years ago. We had set out to write the definitive guide to C++ templates, with the expectation that it would be useful to practicing C++ programmers. That project was successful: It's been tremendously gratifying to hear from readers who found our material helpful, to see our book time and again being recommended as a work of reference, and to be universally well reviewed.

That first edition has aged well, with most material remaining entirely relevant to the modern C++ programmer, but there is no denying that the evolution of the language—culminating in the "Modern C++" standards, C++11, C++14, and C++17—has raised the need for a revision of the material in the first edition.

So with this second edition, our high-level goal has remained unchanged: to provide the definitive guide to C++ templates, including both a solid reference and an accessible tutorial. This time, however, we work with the "Modern C++" language, which is a significantly bigger beast (still!) than the language available at the time of the prior edition.

We're also acutely aware that C++ programming resources have changed (for the better) since the first edition was published. For example, several books have appeared that develop specific template-based applications in great depth. More important, far more information about C++ templates and template-based techniques is easily available online, as are examples of advanced uses of these techniques. So in this edition, we have decided to emphasize a breadth of techniques that can be used in a variety of applications.

Some of the techniques we presented in the first edition have become obsolete because the C++ language now offers more direct ways of achieving the same effects. Those techniques have been dropped (or relegated to minor notes), and instead you'll find new techniques that show the state-of-the-art uses of the new language.

We've now lived over 20 years with C++ templates, but the C++ programmers' community still regularly finds new fundamental insights into the way they can fit in our software development needs. Our goal with this book is to share that knowledge but also to fully equip the reader to develop new understanding and, perhaps, discover the next major C++ technique.

What You Should Know Before Reading This Book

To get the most from this book, you should already know C++. We describe the details of a particular language feature, not the fundamentals of the language itself. You should be familiar with the concepts of classes and inheritance, and you should be able to write C++ programs using components such as IOstreams and containers from the C++ standard library. You should also be familiar with the basic features of "Modern C++", such as `auto`, `decltype`, move semantics, and lambdas. Nevertheless, we review more subtle issues as the need arises, even when such issues aren't directly related to templates. This ensures that the text is accessible to experts and intermediate programmers alike.

We deal primarily with the C++ language revisions standardized in 2011, 2014, and 2017. However, at the time of this writing, the ink is barely dry on the C++17 revision, and we expect that most of our readers will not be intimately familiar with its details. All revisions had a significant impact on the behavior and usage of templates. We therefore provide short introductions to those new features that have the greatest bearing on our subject matter. However, our goal is neither to introduce the modern C++ standards nor to provide an exhaustive description of the changes from the prior versions of this standard ([*C++98*] and [*C++03*]). Instead, we focus on templates as designed and used in C++, using the modern C++ standards ([*C++11*], [*C++14*], and [*C++17*]) as our basis, and we occasionally call out cases where the modern C++ standards enable or encourage different techniques than the prior standards.

Overall Structure of the Book

Our goal is to provide the information necessary to start using templates and benefit from their power, as well as to provide information that will enable experienced programmers to push the limits of the state-of-the-art. To achieve this, we decided to organize our text in *parts*:

- Part I introduces the basic concepts underlying templates. It is written in a tutorial style.
- Part II presents the language details and is a handy reference to template-related constructs.
- Part III explains fundamental design and coding techniques supported by C++ templates. They range from near-trivial ideas to sophisticated idioms.

Each of these parts consists of several chapters. In addition, we provide a few appendixes that cover material not exclusively related to templates (e.g., an overview of overload resolution in C++). An additional appendix covers *concepts*, which is a fundamental extension to templates that has been included in the draft for a future standard (C++20, presumably).

The chapters of Part I are meant to be read in sequence. For example, Chapter 3 builds on the material covered in Chapter 2. In the other parts, however, the connection between chapters is considerably looser. Cross references will help readers jump through the different topics.

Last, we provide a rather complete index that encourages additional ways to read this book out of sequence.

How to Read This Book

If you are a C++ programmer who wants to learn or review the concepts of templates, carefully read Part I, The Basics. Even if you're quite familiar with templates already, it may help to skim through this part quickly to familiarize yourself with the style and terminology that we use. This part also covers some of the logistical aspects of organizing your source code when it contains templates.

Depending on your preferred learning method, you may decide to absorb the many details of templates in Part II, or instead you could read about practical coding techniques in Part III (and refer back to Part II for the more subtle language issues). The latter approach is probably particularly useful if you bought this book with concrete day-to-day challenges in mind.

The appendixes contain much useful information that is often referred to in the main text. We have also tried to make them interesting in their own right.

In our experience, the best way to learn something new is to look at examples. Therefore, you'll find a lot of examples throughout the book. Some are just a few lines of code illustrating an abstract concept, whereas others are complete programs that provide a concrete application of the material. The latter kind of examples will be introduced by a C++ comment describing the file containing the program code. You can find these files at the Web site of this book at `http://www.tmplbook.com`.

Some Remarks About Programming Style

C++ programmers use different programming styles, and so do we: The usual questions about where to put whitespace, delimiters (braces, parentheses), and so forth came up. We tried to be consistent in general, although we occasionally make concessions to the topic at hand. For example, in tutorial sections, we may prefer generous use of whitespace and concrete names to help visualize code, whereas in more advanced discussions, a more compact style could be more appropriate.

We do want to draw your attention to one slightly uncommon decision regarding the declaration of types, parameters, and variables. Clearly, several styles are possible:

```
void foo (const int &x);
void foo (const int& x);
void foo (int const &x);
void foo (int const& x);
```

Although it is a bit less common, we decided to use the order `int const` rather than `const int` for "constant integer." We have two reasons for using this order. First, it provides for an easier answer to the question, "*What* is constant?" It's always what is in front of the `const` qualifier. Indeed, although

```
const int N = 100;
```

is equivalent to

```
int const N = 100;
```

there is no equivalent form for

```
int* const bookmark;      // the pointer cannot change, but the value pointed to can
```

that would place the `const` qualifier before the pointer operator `*`. In this example, it is the pointer itself that is constant, not the `int` to which it points.

Our second reason has to do with a syntactical substitution principle that is very common when dealing with templates. Consider the following two type declarations using the `typedef` keyword:[1]

```
typedef char* CHARS;
typedef CHARS const CPTR;     // constant pointer to chars
```

or using the `using` keyword:

```
using CHARS = char*;
using CPTR  = CHARS const;    // constant pointer to chars
```

The meaning of the second declaration is preserved when we textually replace `CHARS` with what it stands for:

```
typedef char* const CPTR;     // constant pointer to chars
```

or:

```
using CPTR = char* const;     // constant pointer to chars
```

However, if we write `const` *before* the type it qualifies, this principle doesn't apply. Consider the alternative to our first two type definitions presented earlier:

```
typedef char* CHARS;
typedef const CHARS CPTR;     // constant pointer to chars
```

Textually replacing `CHARS` results in a type with a different meaning:

```
typedef const char* CPTR;     // pointer to constant chars
```

The same observation applies to the `volatile` specifier, of course.

Regarding whitespaces, we decided to put the space between the ampersand and the parameter name:

```
void foo (int const& x);
```

By doing this, we emphasize the separation between the parameter type and the parameter name. This is admittedly more confusing for declarations such as

```
char* a, b;
```

where, according to the rules inherited from C, `a` is a pointer but `b` is an ordinary `char`. To avoid such confusion, we simply avoid declaring multiple entities in this way.

This is primarily a book about language features. However, many techniques, features, and helper templates now appear in the C++ standard library. To connect these two, we therefore demonstrate

[1] Note that in C++, a type definition defines a "type alias" rather than a new type (see Section 2.8 on page 38). For example:

```
typedef int Length; // define Length as an alias for int
int i = 42;
Length l = 88;
i = l;              // OK
l = i;              // OK
```

template techniques by illustrating how they are used to implement certain library components, *and* we use standard library utilities to build our own more complex examples. Hence, we use not only headers such as `<iostream>` and `<string>` (which contain templates but are not particularly relevant to define other templates) but also `<cstddef>`, `<utilities>`, `<functional>`, and `<type_traits>` (which do provide building blocks for more complex templates).

In addition, we provide a reference, Appendix D, about the major template utilities provided by the C++ standard library, including a detailed description of all the standard type traits. These are commonly used at the core of sophisticated template programming

The C++11, C++14, and C++17 Standards

The original C++ standard was published in 1998 and subsequently amended by a *technical corrigendum* in 2003, which provided minor corrections and clarifications to the original standard. This "old C++ standard" is known as C++98 or C++03.

The C++11 standard was the first major revision of C++ driven by the ISO C++ standardization committee, bringing a wealth of new features to the language. A number of these new features interact with templates and are described in this book, including:

- Variadic templates
- Alias templates
- Move semantics, rvalue references, and perfect forwarding
- Standard type traits

C++14 and C++17 followed, both introducing some new language features, although the changes brought about by these standards were not quite as dramatic as those of C++11.[2] New features interacting with templates and described in this book include but are not limited to:

- Variable templates (C++14)
- Generic Lambdas (C++14)
- Class template argument deduction (C++17)
- Compile-time `if` (C++17)
- Fold expressions (C++17)

We even describe *concepts* (template interfaces), which are currently slated for inclusion in the forthcoming C++20 standard.

At the time of this writing, the C++11 and C++14 standards are broadly supported by the major compilers, and C++17 is largely supported also. Still, compilers differ greatly in their support of the different language features. Several will compile most of the code in this book, but a few compilers may not be able to handle some of our examples. However, we expect that this problem will soon be resolved as programmers everywhere demand standard support from their vendors.

[2] The committee now aims at issuing a new standard roughly every 3 years. Clearly, that leaves less time for massive additions, but it brings the changes more quickly to the broader programming community. The development of larger features, then, spans time and might cover multiple standards.

Even so, the C++ programming language is likely to continue to evolve as time passes. The experts of the C++ community (regardless of whether they participate in the C++ Standardization Committee) are discussing various ways to improve the language, and already several candidate improvements affect templates. Chapter 17 presents some trends in this area.

Example Code and Additional Information

You can access all example programs and find more information about this book from its Web site, which has the following URL:

`http://www.tmplbook.com`

Feedback

We welcome your constructive input—both the negative and the positive. We worked very hard to bring you what we hope you'll find to be an excellent book. However, at some point we had to stop writing, reviewing, and tweaking so we could "release the product." You may therefore find errors, inconsistencies, and presentations that could be improved, or topics that are missing altogether. Your feedback gives us a chance to inform all readers through the book's Web site and to improve any subsequent editions.

The best way to reach us is by email. You will find the email address at the Web site of this book:

`http://www.tmplbook.com`

Please, be sure to check the book's Web site for the currently known errata before submitting reports. Many thanks.

Chapter 4

Variadic Templates

Since C++11, templates can have parameters that accept a variable number of template arguments. This feature allows the use of templates in places where you have to pass an arbitrary number of arguments of arbitrary types. A typical application is to pass an arbitrary number of parameters of arbitrary type through a class or framework. Another application is to provide generic code to process any number of parameters of any type.

4.1 Variadic Templates

Template parameters can be defined to accept an unbounded number of template arguments. Templates with this ability are called *variadic templates*.

4.1.1 Variadic Templates by Example

For example, you can use the following code to call `print()` for a variable number of arguments of different types:

basics/varprint1.hpp

```
#include <iostream>

void print ()
{
}

template<typename T, typename... Types>
void print (T firstArg, Types... args)
{
  std::cout << firstArg << '\n';  // print first argument
  print(args...);                 // call print() for remaining arguments
}
```

If one or more arguments are passed, the function template is used, which by specifying the first argument separately allows printing of the first argument before recursively calling `print()` for the remaining arguments. These remaining arguments named `args` are a *function parameter pack*:

```
void print (T firstArg, Types... args)
```

using different "`Types`" specified by a *template parameter pack*:

```
template<typename T, typename... Types>
```

To end the recursion, the nontemplate overload of `print()` is provided, which is issued when the parameter pack is empty.

For example, a call such as

```
std::string s("world");
print (7.5, "hello", s);
```

would output the following:

```
7.5
hello
world
```

The reason is that the call first expands to

```
print<double, char const*, std::string> (7.5, "hello", s);
```

with

- `firstArg` having the value `7.5` so that type `T` is a `double` and
- `args` being a variadic template argument having the values `"hello"` of type `char const*` and `"world"` of type `std::string`.

After printing `7.5` as `firstArg`, it calls `print()` again for the remaining arguments, which then expands to:

```
print<char const*, std::string> ("hello", s);
```

with

- `firstArg` having the value `"hello"` so that type `T` is a `char const*` here and
- `args` being a variadic template argument having the value of type `std::string`.

After printing `"hello"` as `firstArg`, it calls `print()` again for the remaining arguments, which then expands to:

```
print<std::string> (s);
```

with

- `firstArg` having the value `"world"` so that type `T` is a `std::string` now and
- `args` being an empty variadic template argument having no value.

Thus, after printing `"world"` as `firstArg`, we calls `print()` with no arguments, which results in calling the nontemplate overload of `print()` doing nothing.

4.1.2 Overloading Variadic and Nonvariadic Templates

Note that you can also implement the example above as follows:

basics/varprint2.hpp

```
#include <iostream>

template<typename T>
void print (T arg)
{
  std::cout << arg << '\n';   // print passed argument
}

template<typename T, typename... Types>
void print (T firstArg, Types... args)
{
  print(firstArg);            // call print() for the first argument
  print(args...);             // call print() for remaining arguments
}
```

That is, if two function templates only differ by a trailing parameter pack, the function template without the trailing parameter pack is preferred.[1] Section C.3.1 on page 688 explains the more general overload resolution rule that applies here.

4.1.3 Operator `sizeof...`

C++11 also introduced a new form of the `sizeof` operator for variadic templates: `sizeof...`. It expands to the number of elements a parameter pack contains. Thus,

```
template<typename T, typename... Types>
void print (T firstArg, Types... args)
{
  std::cout << sizeof...(Types) << '\n';   // print number of remaining types
  std::cout << sizeof...(args) << '\n';    // print number of remaining args
  …
}
```

twice prints the number of remaining arguments after the first argument passed to `print()`. As you can see, you can call `sizeof...` for both template parameter packs and function parameter packs.

This might lead us to think we can skip the function for the end of the recursion by not calling it in case there are no more arguments:

[1] Initially, in C++11 and C++14 this was an ambiguity, which was fixed later (see [*CoreIssue1395*]), but all compilers handle it this way in all versions.

```
template<typename T, typename... Types>
void print (T firstArg, Types... args)
{
  std::cout << firstArg << '\n';
  if (sizeof...(args) > 0) {       // error if sizeof...(args)==0
    print(args...);                // and no print() for no arguments declared
  }
}
```

However, this approach doesn't work because in general both branches of all *if* statements in function templates are instantiated. Whether the instantiated code is useful is a *run-time* decision, while the instantiation of the call is a *compile-time* decision. For this reason, if you call the `print()` function template for one (last) argument, the statement with the call of `print(args...)` still is instantiated for no argument, and if there is no function `print()` for no arguments provided, this is an error.

However, note that since C++17, a compile-time `if` is available, which achieves what was expected here with a slightly different syntax. This will be discussed in Section 8.5 on page 134.

4.2 Fold Expressions

Since C++17, there is a feature to compute the result of using a binary operator over *all* the arguments of a parameter pack (with an optional initial value).

For example, the following function returns the sum of all passed arguments:

```
template<typename... T>
auto foldSum (T... s) {
  return (... + s);   // ((s1 + s2) + s3) ...
}
```

If the parameter pack is empty, the expression is usually ill-formed (with the exception that for operator `&&` the value is `true`, for operator `||` the value is `false`, and for the comma operator the value for an empty parameter pack is `void()`).

Table 4.1 lists the possible fold expressions.

Fold Expression	**Evaluation**
(... *op pack*)	(((*pack1 op pack2*) *op pack3*) ... *op packN*)
(*pack op* ...)	(*pack1 op* (... (*packN-1 op packN*)))
(*init op* ... *op pack*)	(((*init op pack1*) *op pack2*) ... *op packN*)
(*pack op* ... *op init*)	(*pack1 op* (... (*packN op init*)))

Table 4.1. Fold Expressions (since C++17)

You can use almost all binary operators for fold expressions (see Section 12.4.6 on page 208 for details). For example, you can use a fold expression to traverse a path in a binary tree using operator `->*`:

basics/foldtraverse.cpp

```
// define binary tree structure and traverse helpers:
struct Node {
  int value;
  Node* left;
  Node* right;
  Node(int i=0) : value(i), left(nullptr), right(nullptr) {
  }
  ...
};
auto left = &Node::left;
auto right = &Node::right;

// traverse tree, using fold expression:
template<typename T, typename... TP>
Node* traverse (T np, TP... paths) {
  return (np ->* ... ->* paths);      // np ->* paths1 ->* paths2 ...
}

int main()
{
  // init binary tree structure:
  Node* root = new Node{0};
  root->left = new Node{1};
  root->left->right = new Node{2};
  ...
  // traverse binary tree:
  Node* node = traverse(root, left, right);
  ...
}
```

Here,

```
(np ->* ... ->* paths)
```

uses a fold expression to traverse the variadic elements of `paths` from `np`.

With such a fold expression using an initializer, we might think about simplifying the variadic template to print all arguments, introduced above:

```
template<typename... Types>
void print (Types const&... args)
{
  (std::cout << ... << args) << '\n';
}
```

However, note that in this case no whitespace separates all the elements from the parameter pack. To do that, you need an additional class template, which ensures that any output of any argument is extended by a space:

basics/addspace.hpp

```
template<typename T>
class AddSpace
{
  private:
    T const& ref;                  // refer to argument passed in constructor
  public:
    AddSpace(T const& r): ref(r) {
    }
    friend std::ostream& operator<< (std::ostream& os, AddSpace<T> s) {
      return os << s.ref << ' ';   // output passed argument and a space
    }
};

template<typename... Args>
void print (Args... args) {
  ( std::cout << ... << AddSpace(args) ) << '\n';
}
```

Note that the expression `AddSpace(args)` uses class template argument deduction (see Section 2.9 on page 40) to have the effect of `AddSpace<Args>(args)`, which for each argument creates an `AddSpace` object that refers to the passed argument and adds a space when it is used in output expressions.

See Section 12.4.6 on page 207 for details about fold expressions.

4.3 Application of Variadic Templates

Variadic templates play an important role when implementing generic libraries, such as the C++ standard library.

One typical application is the forwarding of a variadic number of arguments of arbitrary type. For example, we use this feature when:

- Passing arguments to the constructor of a new heap object owned by a shared pointer:

```
// create shared pointer to complex<float> initialized by 4.2 and 7.7:
auto sp = std::make_shared<std::complex<float>>(4.2, 7.7);
```

- Passing arguments to a thread, which is started by the library:

```
std::thread t (foo, 42, "hello");   // call foo(42,"hello") in a separate thread
```

- Passing arguments to the constructor of a new element pushed into a vector:

```
std::vector<Customer> v;
...
v.emplace("Tim", "Jovi", 1962);   // insert a Customer initialized by three arguments
```

Usually, the arguments are "*perfectly forwarded*" with move semantics (see Section 6.1 on page 91), so that the corresponding declarations are, for example:

```
namespace std {
  template<typename T, typename... Args> shared_ptr<T>
  make_shared(Args&&... args);

  class thread {
   public:
    template<typename F, typename... Args>
    explicit thread(F&& f, Args&&... args);
    ...
  };

  template<typename T, typename Allocator = allocator<T>>
  class vector {
   public:
    template<typename... Args> reference emplace_back(Args&&... args);
    ...
  };
}
```

Note also that the same rules apply to variadic function template parameters as for ordinary parameters. For example, if passed by value, arguments are copied and decay (e.g., arrays become pointers), while if passed by reference, parameters refer to the original parameter and don't decay:

```
// args are copies with decayed types:
template<typename... Args> void foo (Args... args);
// args are nondecayed references to passed objects:
template<typename... Args> void bar (Args const&... args);
```

4.4 Variadic Class Templates and Variadic Expressions

Besides the examples above, parameter packs can appear in additional places, including, for example, expressions, class templates, using declarations, and even deduction guides. Section 12.4.2 on page 202 has a complete list.

4.4.1 Variadic Expressions

You can do more than just forward all the parameters. You can compute with them, which means to compute with all the parameters in a parameter pack.

For example, the following function doubles each parameter of the parameter pack `args` and passes each doubled argument to `print()`:

```
template<typename... T>
void printDoubled (T const&... args)
{
  print (args + args...);
}
```

If, for example, you call

```
printDoubled(7.5, std::string("hello"), std::complex<float>(4,2));
```

the function has the following effect (except for any constructor side effects):

```
print(7.5 + 7.5,
      std::string("hello") + std::string("hello"),
      std::complex<float>(4,2) + std::complex<float>(4,2);
```

If you just want to add 1 to each argument, note that the dots from the ellipsis may not directly follow a numeric literal:

```
template<typename... T>
void addOne (T const&... args)
{
  print (args + 1...);    // ERROR: 1... is a literal with too many decimal points
  print (args + 1 ...);   // OK
  print ((args + 1)...);  // OK
}
```

Compile-time expressions can include template parameter packs in the same way. For example, the following function template returns whether the types of all the arguments are the same:

```
template<typename T1, typename... TN>
constexpr bool isHomogeneous (T1, TN...)
{
  return (std::is_same<T1,TN>::value && ...);  // since C++17
}
```

This is an application of fold expressions (see Section 4.2 on page 58): For

```
isHomogeneous(43, -1, "hello")
```

the expression for the return value expands to

```
std::is_same<int,int>::value && std::is_same<int,char const*>::value
```

and yields `false`, while

```
isHomogeneous("hello", " ", "world", "!")
```

yields `true` because all passed arguments are deduced to be `char const*` (note that the argument types decay because the call arguments are passed by value).

4.4.2 Variadic Indices

As another example, the following function uses a variadic list of indices to access the corresponding element of the passed first argument:

```
template<typename C, typename... Idx>
void printElems (C const& coll, Idx... idx)
{
  print (coll[idx]...);
}
```

That is, when calling

```
std::vector<std::string> coll = {"good", "times", "say", "bye"};
printElems(coll,2,0,3);
```

the effect is to call

```
print (coll[2], coll[0], coll[3]);
```

You can also declare nontype template parameters to be parameter packs. For example:

```
template<std::size_t... Idx, typename C>
void printIdx (C const& coll)
{
  print(coll[Idx]...);
}
```

allows you to call

```
std::vector<std::string> coll = {"good", "times", "say", "bye"};
printIdx<2,0,3>(coll);
```

which has the same effect as the previous example.

4.4.3 Variadic Class Templates

Variadic templates can also be class templates. An important example is a class where an arbitrary number of template parameters specify the types of corresponding members:

```
template<typename... Elements>
class Tuple;

Tuple<int, std::string, char> t;      // t can hold integer, string, and character
```

This will be discussed in Chapter 25.

Another example is to be able to specify the possible types objects can have:

```
template<typename... Types>
class Variant;

Variant<int, std::string, char> v;   // v can hold integer, string, or character
```

This will be discussed in Chapter 26.

You can also define a class that ***as a type*** represents a list of indices:

```
// type for arbitrary number of indices:
template<std::size_t...>
struct Indices {
};
```

This can be used to define a function that calls `print()` for the elements of a `std::array` or `std::tuple` using the compile-time access with `get<>()` for the given indices:

```
template<typename T, std::size_t... Idx>
void printByIdx(T t, Indices<Idx...>)
{
  print(std::get<Idx>(t)...);
}
```

This template can be used as follows:

```
std::array<std::string, 5> arr = {"Hello", "my", "new", "!", "World"};
printByIdx(arr, Indices<0, 4, 3>());
```

or as follows:

```
auto t = std::make_tuple(12, "monkeys", 2.0);
printByIdx(t, Indices<0, 1, 2>());
```

This is a first step towards meta-programming, which will be discussed in Section 8.1 on page 123 and Chapter 23.

4.4.4 Variadic Deduction Guides

Even deduction guides (see Section 2.9 on page 42) can be variadic. For example, the C++ standard library defines the following deduction guide for `std::arrays`:

```
namespace std {
  template<typename T, typename... U> array(T, U...)
    -> array<enable_if_t<(is_same_v<T, U> && ...), T>,
             (1 + sizeof...(U))>;
}
```

An initialization such as

```
std::array a{42,45,77};
```

deduces `T` in the guide to the type of the element, and the various `U...` types to the types of the subsequent elements. The total number of elements is therefore `1 + sizeof...(U)`:

```
std::array<int, 3> a{42,45,77};
```

The `std::enable_if<>` expression for the first `array` parameter is a fold expression that (as introduced as `isHomogeneous()` in Section 4.4.1 on page 62) expands to:

```
is_same_v<T, U1> && is_same_v<T, U2> && is_same_v<T, U3> ...
```

If the result is not `true` (i.e., not all the element types are the same), the deduction guide is discarded and the overall deduction fails. This way, the standard library ensures that all elements must have the same type for the deduction guide to succeed.

4.4.5 Variadic Base Classes and `using`

Finally, consider the following example:

basics/varusing.cpp

```
#include <string>
#include <unordered_set>

class Customer
{
  private:
    std::string name;
  public:
    Customer(std::string const& n) : name(n) { }
    std::string getName() const { return name; }
};

struct CustomerEq {
    bool operator() (Customer const& c1, Customer const& c2) const {
      return c1.getName() == c2.getName();
    }
};

struct CustomerHash {
    std::size_t operator() (Customer const& c) const {
      return std::hash<std::string>()(c.getName());
    }
};

// define class that combines operator() for variadic base classes:
template<typename... Bases>
struct Overloader : Bases...
{
    using Bases::operator()...;  // OK since C++17
};

int main()
{
  // combine hasher and equality for customers in one type:
  using CustomerOP = Overloader<CustomerHash,CustomerEq>;

  std::unordered_set<Customer,CustomerHash,CustomerEq> coll1;
  std::unordered_set<Customer,CustomerOP,CustomerOP> coll2;
  ...
}
```

Here, we first define a class `Customer` and independent function objects to hash and compare `Customer` objects. With

```
template<typename... Bases>
struct Overloader : Bases...
{
    using Bases::operator()...;  // OK since C++17
};
```

we can define a class derived from a variadic number of base classes that brings in the `operator()` declarations from each of those base classes. With

```
using CustomerOP = Overloader<CustomerHash,CustomerEq>;
```

we use this feature to derive `CustomerOP` from `CustomerHash` and `CustomerEq` and enable both implementations of `operator()` in the derived class.

See Section 26.4 on page 611 for another application of this technique.

4.5 Summary

- By using parameter packs, templates can be defined for an arbitrary number of template parameters of arbitrary type.
- To process the parameters, you need recursion and/or a matching nonvariadic function.
- Operator `sizeof...` yields the number of arguments provided for a parameter pack.
- A typical application of variadic templates is forwarding an arbitrary number of arguments of arbitrary type.
- By using fold expressions, you can apply operators to all arguments of a parameter pack.

Index

C

E

G

H

I

M

P

Q

R

S

T

W

X

Z