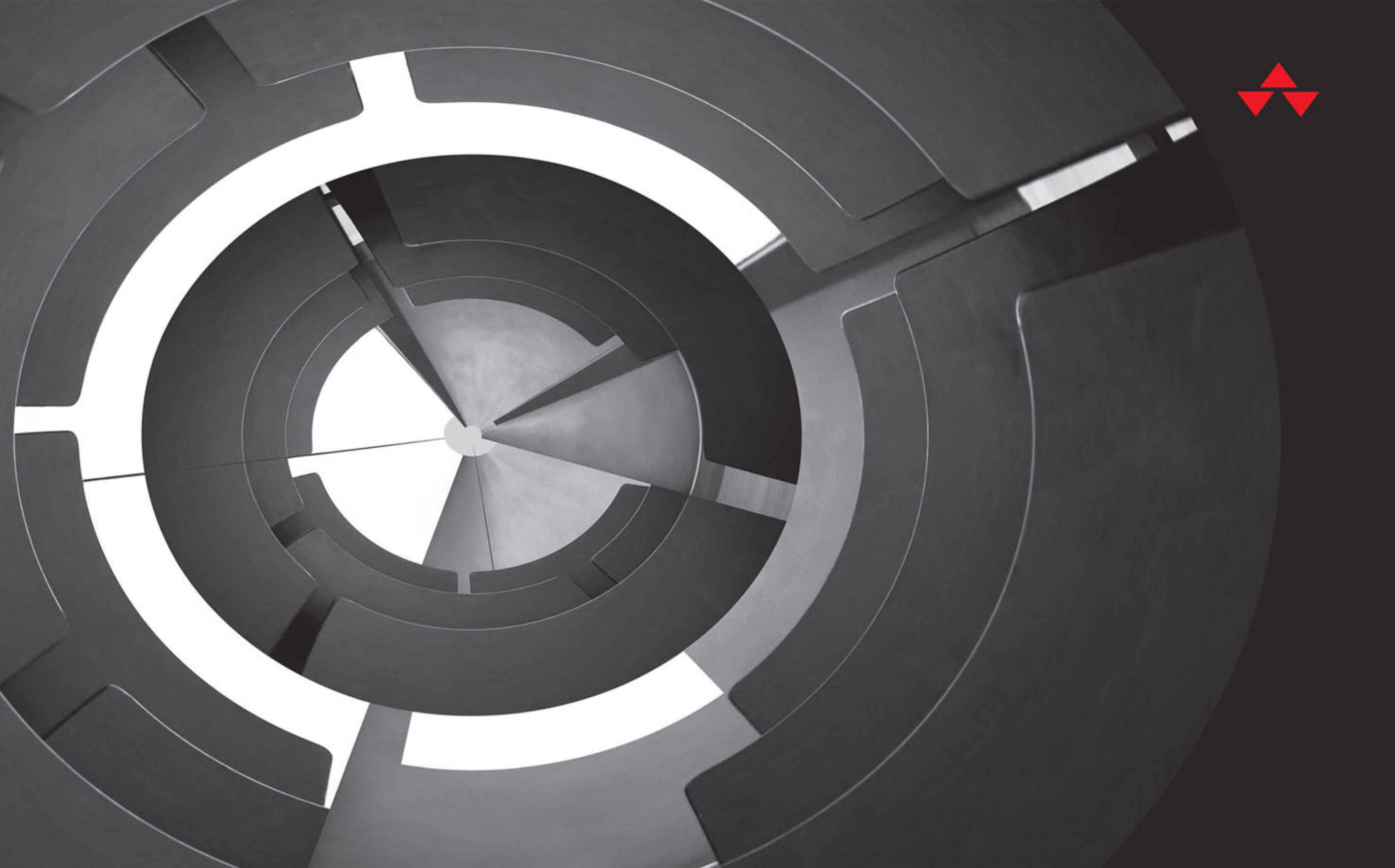

EMBRACING MODERN C++ SAFELY

JOHN LAKOS | VITTORIO ROMEO | ROSTISLAV KHLEBNIKOV | ALISDAIR MEREDITH

Embracing Modern C++ *Safely*

Embracing Modern C++ *Safely*

John Lakos

Vittorio Romeo

Rostislav Khlebnikov

Alisdair Meredith

Addison-Wesley

Boston • Columbus • 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

Cover image: Unconventional/Shutterstock
Pages 109, 130: cocktail glass, Laura Humpfer/OpenMojis

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 Control Number: 2021947542

ISBN-13: 978-0-13-738035-0
ISBN-10: 0-13-738035-6

ScoutAutomatedPrintCode

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

- Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.html.

To my darling wife, Elyse, who I love dearly, always have, and forever will:

"'When I use a word,' Humpty Dumpty said in rather a scornful tone,
'it means just what I choose it to mean—neither more nor less.'"
— Lewis Carroll, *Through the Looking-Glass*

JSL

To my aunts and my dad,
who have always supported me
in every aspect of my life.

VR

To Elena and my parents.

RK

To the late David and Mary Meredith,
loving parents who encouraged me in everything that I did
and would have been so proud to see their son finally in print.

AM

Contents

Contents

Foreword by Shawn Edwards

I have been writing programs in C++ professionally for more than 25 years, even before it was standardized. The C++ language, in its mission to deliver zero overhead and maximum performance, necessarily provides few guardrails; syntax and type safety go only so far. Using C++ features in unsound ways and creating spectacular failures was always easy. But because the language was relatively stable, good developers — over time — learned how to write reliable C++ software.

The first standardized version, C++98, formalized what many already knew about the language. The second version of the Standard, C++03, included some small corrections and enhancements but did not fundamentally alter the way programs were written. What it meant to know how to program in C++, however, changed drastically with the publication of the C++11 Standard. For the first time in many years, the ISO C++ Standards Committee (WG21) added significant new functionality and removed functionality as well. For example, `noexcept` and `std::unique_ptr` were in, and the days of using dynamic exception specifications and `std::auto_ptr` were numbered.

At the same time, the Standards Committee announced its unprecedented commitment to deliver a new version of the C++ Standard every three years! For a large software organization, like Bloomberg, whose software asset lifetimes are measured in decades, relying on a language standard that is updated with such frequency is especially problematic. Bloomberg has been reliably and accurately providing indispensable information to the professional financial community for nearly 40 years, with services that span such diverse needs as financial analytics, trading solutions, and real-time market data.

To support our global business, Bloomberg has developed high-performance software systems that operate at scale and, for more than two decades now, has written them primarily in C++. As you can imagine, incorporating and validating new tool chains that underpin our company's entire code base is no simple task. Each update risks the stability of the very products upon which our customers depend.

Modern C++ has much to offer — both good and bad. Many of its newer features offer the prospect of improving performance, expressiveness, maintainability, and so on. On the other hand, many of these same features come with potential pitfalls, some of which are obvious, and others less so. With each new release of C++, now every three years, the language gets bigger, and the opportunities for misusing a feature, through lack of knowledge and experience, grow ever larger as well.

Foreword by Shawn Edwards

Using new features of an already sophisticated programming language such as C++, with which many developers might not be fully familiar, introduces its own category of risk. Less-seasoned engineers might unwittingly introduce new features into a mature code base where they could add manifestly negative value in that context. As ever, only time and experience can provide proof as to whether and under what conditions using a new C++ language feature would be prudent. We, as senior developers, team leads, and technical managers of a leading financial technology company, bear responsibility for protecting our Software Capital asset from undue risk.

We cannot justify the instability of rewriting all of our software every time a new version of the language appears, nor can we leave it in perpetual stasis and forgo the important benefits modern C++ has to offer. So we move forward but with expertise and caution, adopting features only after we fully understand them. Bloomberg is committed to extracting all of the benefit that it can from modern C++, but as a company, we must do so *safely*.

Bloomberg sponsored this book, *Embracing Modern C++ Safely*, because we felt that, despite all of the books, conferences, blogs, etc., that covered C++11/14 features, we needed to look at each feature from the point of view of how to apply it *safely* as well as effectively in the context of a large, mature corpus of production code. Therefore, this book provides detailed explanations of each C++11/14 language feature, examples of its effective use, and pitfalls to avoid. Moreover, this book could only have been written *now*, after years of gathering real-world experience. What's more, we knew that we had the right people — some of the best engineers and authors in the world — to write it.

As promised, the C++ Standards Committee has been sticking to its schedule, sometimes in the face of major world events, and two additional versions of the Standard, C++17 and C++20, have been published. As the community gains experience using the new features provided in those standards, I expect that future editions of this book will offer similar guidance and critique.

If you've been writing programs in C++ for more than a decade, you've undoubtedly noticed that being an accomplished C++ programmer is a different challenge than it used to be. This book will help you navigate the modern C++ landscape so that you too can feel confident in applying C++11/14 in ways that truly add value without undue risk to your organization's precious Software Capital investment.

— Shawn Edwards
Chief Technology Officer, Bloomberg LP
August 2021

Foreword by Andrei Alexandrescu

Do you like version control systems — Git, Perforce, Mercurial, and such? I love them! I have no idea how any of today's complex software systems could have ever been built without using version control.

One beneficial artifact of version control software is the *diff view*, that quintessential side-by-side view of a change of a large system as a differential from the previous, known version of the system. The diff view is often the best way to review code, to assess complexity of a feature, to find a bug, and, most importantly, to get familiar with a new system. I pore over diff views almost every working day, perusing them for one or more of their advantages. The diff view is proof that we *can* actually have the proverbial nice things.

The novel concept of this book is a diff view between classic C++ — i.e. C++03, the baseline — and modern C++ — i.e., post-2011 C++, with its added features. A diff view of programming language features! Now that's a cool idea with interesting implications.

Embracing Modern C++ Safely addresses a large category of programmers: those who work daily on complex, long-lived C++ systems and who are familiar with C++03 because said systems were written with that technology. Classes. Inheritance. Polymorphism. Templates. The STL. Yep, they know these notions well and work with them every day in complex problem domains. Rehashing those classic features is unnecessary. But some programmers are perhaps less comfortable with the cornucopia of new features standardized every three years, starting with C++11. They have no time to spend on tracking what the C++ Standards Committee is doing. Every hour spent learning new C++ features is an hour not spent on core systems functionality, so that snazzy new feature better be worth it. *Embracing Modern C++ Safely* is cleverly optimized to maximize the ratio of usefulness in production to time spent learning.

Pedagogically, this book achieves an almost impossible challenge: a *partial* diff (to allow this nerd a mathematically motivated metaphor) for each individual new feature added to C++ after 2003. What do I mean by that? When a book teaches language features, cross talk is inevitable: While discussing any one given feature, most other features interfere by necessity. As Scott Meyers once told me, "When you learn a language, all features come at you in parallel." The authors *modularized* the teaching of each new feature, so if you want to read about, say, generic lambdas, you get to read about generic lambdas with minimal interference from any other new language feature. When necessary, the interaction between the feature being discussed and others is narrowly specified, documented, and cross-referenced. The result is a fractally self-consistent book that can be read cover to cover or chunked by themes, interconnected features, or individual topics.

Foreword by Andrei Alexandrescu

Chapters 1, 2, and 3 mimic a sort of reverse *Divine Comedy*, whereby, as you may recall, the poet Dante is led by trusted guides through Hell (*Inferno*), Purgatory (*Purgatorio*), and Heaven (*Paradiso*). The respective chapters help you navigate from *Safe* to *Conditionally Safe* to *Unsafe* features of Modern C++.

Safe features (Chapter 1) will clearly, definitely, pound-the-table improve your code wherever you use them. Acquiring and applying the teachings of Chapter 1 is the fastest way for a team to start leveraging Modern C++ in production. `override`? Enjoy. Digit separators? Have at 'em. Explicit conversion operators? Knock yourself out. Such features are recommended fully and without reservation. Chapter 2 discusses *conditionally safe* features, those that are good for you but come with caveats. Initializer lists? Let's talk. Range `for`? Couple of things to be mindful of. Rvalue references? Long discussion; grab a coffee. And last but not least, *unsafe* features are those that can be challenging and require skill and utmost attention in usage. Their use should be confined as much as possible and wrapped under interfaces. Standard-layout types? Way trickier than it may seem. The `noexcept` specifier? Careful, you're on your own. Inline namespaces? At best, don't. Extensive details, examples, and discussions are available for every single feature added after C++03.

The authors use "unsafe" in a tongue-in-cheek manner here. Nothing taught in this book is unsafe in the traditional computer science sense; instead, think of the casual meaning of "safety" when used, say, in a hardware store. What's the safety of various tools for someone just starting to use them? A screwdriver is safe; a power drill is conditionally safe; and a welding machine is unsafe.

You may be concerned, thinking, "That sounds authoritative. What is the basis of such a ranking?" In fact, *Embracing Modern C++ Safely* is emphatically not authoritative but *objective* and based on the vast community of experience that the authors collected and curated. They intentionally, sometimes painfully, withhold their opinions. The "Use Cases" and "Potential Pitfalls" sections, taken from production code, are empirical evidence as much as instructive examples to learn from.

Only the passage of time can distill the programming community's practical experience with each feature and how well it fared, which is why this book discusses features added up to C++14, even though C++20 is already out. Using features for years can replace passionate debate on language design ideas with cold, hard experience, which guides this book's remarkably clinical approach. In the words of John Lakos, "We explain the degrees of burns you could get if you put your hand on a hot stove, but we won't tell you not to do it." The result is a refreshingly nonideological read, no more partisan than a book on experimental physics. Consistently avoiding injecting one's own ego and opinion in an analysis takes paradoxically a lot of work. *Ars est celare artem*, the Latin proverb goes in typical brief, cryptic, and slightly confusing manner. (Is Latin the APL of natural languages?) That literally translates to "the art is to conceal the art," but the profound meaning is closer to "good art is not emphatically artsy." Good artists don't leave fingerprints all over their work. In a very concrete way, that has been a design goal of *Embracing Modern C++ Safely*, for you won't find in it any opinion, pontification, or even gratuitously flowery language. (Fierce debates occurred about the perfect, most spartan choice of words in one paragraph or another.) This polished clarity will, I'm sure, shine through to any reader.

That Extra Oomph

"The only kind of writing is rewriting," goes the famous quote. That is doubly true for technical books. The strength of a textbook stands in the willingness of its authors to redo their work and in the depth and breadth of its review team feeding the revision process. And rewriting is not easy! Have you ever written some code and then resisted reviews because you fell in love with it? Multiply that by 1024 and you'll know how book authors feel about rewriting passages they've already poured their souls into. You really need to be committed to quality to keep heart during such a trial.

The authors' insistence on quality brings to mind what I like most about this book, which is also the most difficult to explain. I call it *the extra oomph.*

I noticed something about great work — be it in engineering, art, sports, or any other challenging human endeavor. Almost always, great work is the result of talented people making an extra effort that goes beyond what one might consider reasonable. In appreciating such work, we implicitly acknowledge great capability combined with commensurately great effort in realizing it. Good work can be done glibly; great work cannot.

Through an odd turn of events I ended up getting quite involved with this book — first, for one review. And then another, and another, for a total of four thorough passes through the entire book. The quest for perfection is as contagious as the resignation to sloppiness and incomparably more fun. ("Destroy!" John Lakos pithily emailed me along with each new revision. My often caustic reviews motivated him like nothing else.) Other reviewers — C++ Standards Committee denizens, industry C++ experts, C++03 experts with no prior exposure to C++1x, software-architecture experts, multithreading experts, process experts, even LaTeX experts — have done the same, with the net result that each sentence you'll read has been critically considered dozens of times and probably rewritten a few. For my part, I got so enthused with the project and with the authors' uncompromising take on quality, that I ended up writing a full feature for the book. (Any mistake in Section 2.1. "Variadic Templates" is my fault.) This book project has been a lot of work, more than I might have reasonably expected, which is everything I'd hoped for. I thought I've gotten too old to still pull all-nighters; apparently I was wrong.

Having been thusly involved, I can tell: This book does have that extra oomph baked into it. The talk is being walked, there's no fluff, and the code examples are precise and eloquent. I think *Embracing Modern C++ Safely* is Great Work. Aside from learning from this book, I hope you derive from it inspiration to add more oomph into your own work. I know I did.

— Andrei Alexandrescu
May 2021

Acknowledgments

Embracing Modern C++ Safely is the work of the C++ community as a whole, not just the authors. This book comprises knowledge drawn from the depths of language design to the boundaries of sound software development. Those who are expert at one end of that language-design to application-development spectrum might be relatively unfamiliar with the other. Although we, the four authors named on the front of this book, are each professional senior software engineers, our combined knowledge did not initially span everything presented here, and we relied on many of our colleagues — from our fellow developers at Bloomberg to the Core Working Group of the C++ Standards Committee to Bjarne Stroustrup himself — to fill in holes in our understanding and to correct misconceptions we held.

Everyone on Bloomberg's BDE team, founded in December 2001, contributed directly, in one way or another, to the publication of this book: Parsa Amini, Joshua Berne, Harry Bott, Steven Breitstein, Nathan Burgers, Bill Chapman, Attila Feher, Mungo Gill, Rostislav Khlebnikov, Jeffrey Mendelsohn, Alisdair Meredith, Hyman Rosen, and the BDE team's second manager (since April 2019), Mike Verschell.

Nina Ranns, ISO C++ Standards Committee secretary and ISO C++ Foundation director, was our principal researcher and provided a window into the depths of the C++ core language standard. We relied on her to get to *the truth*: With a release coming every three years and defect reports retroactive to previous standards, *the truth* is a contextual, ephemeral, and elusive beast. Nonetheless, Nina provided us with clarity about what was in effect when and thoroughly reviewed each and every core-language-intensive feature in this book; see Section 2.1."**`constexpr`** Functions" on page 257, Section 2.1."Generalized PODs '11" on page 401, Section 2.1."*Rvalue* References" on page 710, and Section 3.1."**`noexcept`** Specifier" on page 1085, as just a few examples.

Joshua Berne, senior software engineer on Bloomberg's BDE team and an active member of the C++ Standards Committee's Core Working Group (CWG) and Contracts Study Group (SG21), served multiple roles: Josh was our bridge between the core language and software development, performing structural rewrites of major features, including the features mentioned and many others. All benchmarking research conducted for this book was designed, performed, and/or reviewed by Josh. He provided the technical expertise needed to make LaTeX function to its fullest capabilities, designing and implementing the glossary, including automating the references back into the individual sections that use the terms. Importantly, Josh was the voice of reason throughout this entire project.

Lori Hughes, our project manager, frontline technical editor, and LaTeX designer and compositor, would probably tell you that herding cats is child's play compared to what she endured during this project. The tenacity, assertiveness, and roll-up-your-sleeves hard work she demonstrated relentlessly is arguably the only reason this book was published in 2021 (if not this decade). In short, Lori's our rock; she is a veteran of **lakos20**, and we look forward to working with her on *all* of our planned future projects — e.g., allocators (**lakos22**), contracts (**lakos23**), Volumes II and III following **lakos20** (**lakos2a** and **lakos2b**), and anticipated future editions of this book incorporating C++17, C++20, etc.

Pablo Halpern, a former member of Bloomberg's BDE team, an active member of the C++ Standards Committee, the creator of the `std::pmr` allocator model, and now a full-time collaborator with BDE working on language-level support for local memory allocators, served as a ghost writer for several features in this book (e.g., see Section 2.1."User-Defined Literals" on page 835) and provided massive restructuring to many others (e.g., see Section 2.1. "Generalized PODs '11" on page 401. Notably, out of all the nonauthors who contributed drafts in final form, only Pablo was able to write in a style approximating the authors' voice. He also performed the research for a paper, commissioned by the authors of this book, demonstrating that move operations, though faster to execute initially, can have negative overall runtime implications due to memory diffusion; see **halpern21c**.

Dr. Andrei Alexandrescu — author of the seminal book *Modern C++ Design* (Addison-Wesley, 2001), coauthor of *C++ Coding Standards* (Addison-Wesley, 2005), and major contributor to the D language — was called upon for multiple assists in this endeavor: (1) as an expert author to provide an approachable guide to using variadic templates for those accustomed to their C++03 counterparts (see Section 2.1."Variadic Templates" on page 873); (2) as a technical reviewer whose primary job was to reduce the tedium of John Lakos' writing style and its numerous parenthetical phrases and footnotes; and (3) as a mascot and champion of our effort to imbue, on C++03 folk, the C++11/14 overlay of features. Andrei also generously agreed to write a foreword to this book, advocating its utility for senior developers familiar with classic C++.

Harold Bott, John's TA in his Advanced C++ course during the 1990s at Columbia University, reconnected with John in 2019. Harry has since been a force in driving this book forward to completion. After a month of research with Nina, John entrusted Harry, a former programmer at Goldman Sachs and Executive Director at JP Morgan, with getting the flagship feature of modern C++ (see Section 2.1."*Rvalue* References" on page 710) ready for review — a daunting task indeed. Once reviews were in and revisions were needed, Harry worked with John, nearly around the clock for almost three straight weeks, to incorporate reviewer feedback and to bring this important feature to the state in which it is presented here.

Mungo Gill is one of the newest full-time contributors on the BDE team and brings with him more than 30 years of professional software experience at such notable organizations as Salomon Brothers, Citigroup, Lehman Brothers, Google, and Citadel Securities. Mungo has reviewed every line of this book and has provided valuable feedback from a senior practitioner's perspective. He also coordinated the process of assembling glossary definitions and gaining consensus among a host of eclectic domain experts.

Clay Wilson, a member of the BDE team since 2003, is another veteran of **lakos20**. Clay has, for the past 18 years, been our "closer" when it comes to reviewing software components.

His attention to detail and accuracy is, in our experience, second to none. Clay has reviewed much of this book, and we look forward to the possibility of working with him on future projects.

Steven Breitstein, a member of BDE since 2004 and an alumnus of **lakos20**, has reviewed every line and code snippet in this book and has made innumerable suggestions for manifestly improving the rendering of the material. He also stepped up and singlehandedly applied all the copy edits to our glossary.

Hyman Rosen retired from Bloomberg's BDE team in April 2021 and was the master of pragmatic real-world use cases for some of the otherwise ostensibly *unsafe* features of modern C++, such as using extended friendship (see Section 3.1."**`friend`** '11" on page 1031) with the **`curiously recurring template pattern` (CRTP)**. You'll find many others scattered throughout this book.

Stephen Dewhurst, an internationally recognized expert in C++ programming and popular repeat C++ author, conference speaker, and professional C++ trainer (including, for more than a decade, at Bloomberg), has reviewed *every* feature in this book and provided copious, practically valuable feedback, including a use case; see *Use Cases — Stateless lambdas* on page 605 within Section 2.1."Lambdas."

Jeffrey Olkin, who joined Bloomberg in 2011, is one of its most senior software architects, was the structural editor of **lakos20**, and has been a welcome advocate of this book from the start, reviewing many features, helping to organize the preliminary material, and providing his insightful and always valuable feedback along the way.

Steve Downey, a senior developer at Bloomberg since 2003, C++ Standards Committee member, and multidomain expert, contributed much of the advanced material found in a somewhat niche, *conditionally safe* feature of C++11; see Section 1.1."Unicode Literals" on page 129. Mike Giroux and Oleg Subbotin fleshed out and provided benchmark material for another *conditionally safe* C++11 feature; see Section 2.1."**`extern template`**" on page 353.

Sean Parent contributed a subsection assessing the strictness of current requirements on moved-from objects for standard containers; see *Annoyances — Standard Library requirements on a moved-from object are overly strict* on page 807 within Section 2.1."*Rvalue* References." Niall Douglas contributed a subsection detailing his experiences at scale with one of the *unsafe* C++ features; see *Appendix — Case study of using **`inline`** namespaces for versioning* on page 1083 within Section 3.1."**`inline namespace`**." Niels Dekker reviewed another *unsafe* C++11 feature (see Section 3.1."**`noexcept`** Specifier" on page 1085) and provided valuable additional information as well as pointers to his own benchmark research. Kevin Klein helped organize and draft the material of yet another *unsafe* C++11 feature; see Section 3.1."**`final`**" on page 1007.

Many senior C++ software engineers, instructors, and professional developers reviewed this work and provided copious feedback: Adil Al-Yasiri, Andrei Alexandrescu, Parsa Amini, Brian Bi, Frank Birbacher, Harry Bott, Steve Breitstein, Tomaz Canabrava, Bill Chapman, Marshall Clow, Stephen Dewhurst, Akshaye Dhawan, Niall Douglas, Steve Downey, Tom Eccles, Attila Feher, Kevin Fleming, J. Daniel Garcia, Mungo Gill, Mike Giroux, Kevin

Klein, Jeff Mendelsohn, Jeffrey Olkin, Nina Ranns, Hyman Rosen, Daniel Ruoso, Ben Saks, Richard Smith, Oleg Subbotin, Julian Templeman, Mike Verschell, Clay Wilson, and JC van Winkel.

In addition to reviewing the features of this book as they were being written, the BDE team fleshed out the first draft of the glossary, after which we relied again on Josh Berne, Brian Bi, Harry Bott, Mungo Gill, Pablo Halpern, and Nina Ranns to refine and consolidate it into a final draft before finally reviewing it ourselves in its totality. Jeff Mendelsohn and Nathan Burgers helped to distill the essence of this book onto its back cover. We want to thank all the Standards Committee members who provided valuable information when researching the details, history, etc., surrounding the various language features presented in this book. In particular, we would like to thank Bjarne Stroustrup for affably answering our pointed questions regarding anything related to C++. Howard Hinnant confirmed, among other things, the details of why and how *xvalues* were originally invented and how they have since morphed (a.k.a. “the delta”) from their original concept to their definition today. Michael Wong, Paul McKenney, and Maged Michael reviewed and signed off on our presentation of the `[[carries_dependency]]` attribute; see Section 3.1.“`carries_dependency`” on page 998. And we cannot thank Richard Smith enough for his thorough review and myriad suggestions on how to correct and improve our treatment of the flagship feature of modern C++ (see Section 2.1.“*Rvalue* References” on page 710). We hope that Richard will review *every* feature in subsequent editions of this book.

The team at Pearson — Greg Doench, our editor and fearless leader; Julie Nahil, our production manager; and Kim Wimpsett, our copy editor — have been very supportive of our efforts to get this book done quickly and accurately, despite its unorthodox workflow. We had originally projected that this book would contain 300–400 pages and would be complete by the end of 2020. That didn’t happen. Somehow, Greg and Julie found a way to accommodate our process and get this book printed in time for the 2021 winter holidays; thank you!

Online compilers, such as Godbolt (Compiler Explorer) and Wandbox, proved invaluable in the development of this work, allowing the team to rapidly evaluate and share code samples tested with various versions of multiple compilers accepting different dialects of the language.

We want to give a shout-out to the folks at Bloomberg involved in making sure that Bloomberg’s intellectual property and customer data were in no way compromised by anything contained herein and that appropriate attributions were made: Tom Arcidiacono, Kevin P. Fleming, and Chaim Haas.

Moreover, we want to recognize and thank our Bloomberg management for providing not just the support but the *imperative* to do this essential work for ourselves and then share it! In 2012, Vladimir Kliatchko, then and still Global Head of Engineering at Bloomberg, directed John Lakos, who collaborated with Rostislav Khlebnikov, to write a paper, **khlebnikov18**, to describe concisely the value proposition of C++11 and how best to exploit it. That short C++11 paper, 11 pages of 11-point type, was indeed well received, widely accepted, and ratified by fully 85 percent of the Standards Committee members who reviewed it. After

that, Andrei Basov, Engineering Manager, Middleware and Core Services; Akshaye Dhawan, Engineering Manager, Training, Documentation, and Work Management; and Adam Wolf, Head of Engineering, Software Infrastructure, encouraged and supported us in pursuing a more all-encompassing, practical-engineering-oriented treatment of modern C++, including C++14, in book form.

Finally, this book would not have been possible without the generous patronage of our Chief Technology Officer, Shawn Edwards. Without his support, and especially his sponsorship, the vast technical resources needed for this book to come to fruition could never have been brought to bear. Shawn, with his illustrious career as a developer, team lead, and technical manager, and now, as a senior executive, has graciously provided a foreword to this book.

About the Authors

John Lakos, author of *Large-Scale C++ Software Design* (Addison-Wesley, 1996) and *Large-Scale C++ Volume I: Process and Architecture* (Addison-Wesley, 2020), serves at Bloomberg in New York City as a senior architect and mentor for C++ software development worldwide. He is also an active voting member of the C++ Standards Committee's Evolution Working Group. From 1997 to 2001, Dr. Lakos directed the design and development of infrastructure libraries for proprietary analytic financial applications at Bear Stearns. From 1983 to 1997, Dr. Lakos was employed at Mentor Graphics, where he developed large frameworks and advanced ICCAD applications for which he holds multiple software patents. His academic credentials include a Ph.D. in Computer Science (1997) and an Sc.D. in Electrical Engineering (1989) from Columbia University. Dr. Lakos received his undergraduate degrees from MIT in Mathematics (1982) and Computer Science (1981).

Vittorio Romeo (B.Sc., Computer Science, 2016) is a senior software engineer at Bloomberg in London, where he builds mission-critical C++ middleware and delivers modern C++ training to hundreds of fellow employees. He began programming at the age of 8 and quickly fell in love with C++. Vittorio has created several open-source C++ libraries and games, has published many video courses and tutorials, and actively participates in the ISO C++ standardization process. He is an active member of the C++ community with an ardent desire to share his knowledge and learn from others: He presented more than 20 times at international C++ conferences (including CppCon, C++Now, ++it, ACCU, C++ On Sea, C++ Russia, and Meeting C++), covering topics from game development to template metaprogramming. Vittorio maintains a website (https://vittorioromeo.info/) with advanced C++ articles and a YouTube channel (https://www.youtube.com/channel/UC1XihgHdkNOQd5IBHnIZWbA) featuring well received modern C++11/14 tutorials. He is active on StackOverflow, taking great care in answering interesting C++ questions (75k+ reputation). When he is not writing code, Vittorio enjoys weightlifting and fitness-related activities as well as computer gaming and sci-fi movies.

Rostislav Khlebnikov is the lead of the BDE Solutions team that works on a variety of BDE libraries, such as the library for HTTP/2 communication, and contributes to other projects, including improving interoperability of BDE libraries with the Standard Library vocabulary types. He is an active member of the C++ Standards Committee and presented at CppCon 2019. Prior to his work at Bloomberg, Dr. Khlebnikov received his undergraduate degrees in Applied Mathematics and Computer Science from St. Petersburg State

About the Authors

Polytechnic University, Russia, and his Ph.D. in Computer Science from Graz University of Technology, Austria. He has worked professionally as a C++ software engineer for over 15 years.

Alisdair Meredith has been a member of the C++ Standards Committee since the inception of C++11 at the Oxford 2003 meeting, focusing on feature integration and actively finding and fixing language inconsistencies. Alisdair was the LWG chair when both C++11 and C++14 were published, for which he credits the hard work of the preceding chair, Howard Hinnant. Alisdair has been a perennial conference speaker for nearly 15 years, elucidating new work from the C++ Standards Committee. Alisdair joined Bloomberg's BDE team in 2009. For a decade prior, Alisdair worked as a professional C++ application programmer in F1 motor racing with the Benetton and Renault teams, winning two world championships! Between the two, Alisdair spent a year or so as a product manager at Borland, marketing their C++ products. Alisdair enjoys traveling, dining, and snorkeling.

Chapter 0

Introduction

Welcome! *Embracing Modern C++ Safely* is a reference book designed for professionals who develop and maintain large-scale, complex C++ software systems and want to leverage modern C++ features.

This book focuses on the productive value of each new language feature, starting with C++11, particularly when the systems and organizations involved are considered at scale. We deliberately left aside ideas and idioms — however clever and intellectually intriguing — that could hurt the bottom line when applied to large-scale systems. Instead, we focus on making wise economic and design decisions, with an understanding of the inevitable trade-offs that arise in any engineering discipline. In doing so, we do our best to steer clear of subjective opinions and recommendations.

Richard Feynman famously said, "If it disagrees with experiment, it's wrong. In that simple statement is the key to science."[1] There is no better way to experiment with a language feature than letting time do its work. We took that to heart and decided to cover only the features of modern C++ that have been part of the Standard for at least five years, which we believe provides enough perspective to properly evaluate the practical impact of new features. Thus, we are able to draw from practical experience to provide a thorough and comprehensive treatment that is worthy of your limited professional development time. If you're looking for ways to improve your productivity by using tried and true modern C++ features, we hope this book will be the one you'll reach for.

What's missing from a book is as important as what's present. *Embracing Modern C++ Safely*, known also as *EMC++S*, is not a tutorial on C++ programming or even on new features of C++. We assume you are an experienced developer, team lead, or manager; that you already have a good command of classic C++98/03; and that you are looking for clear, goal-driven ways to integrate modern C++ features into your toolbox.

What Makes This Book Different

The goal of the book you're now reading is to be objective, empirical, and practical. We simply present features, their applicability, and their potential pitfalls as reflected by the analysis of millions of person-hours of using C++11 and C++14 in the development of

[1] Richard Feynman, lecture at Cornell University, 1964. Video and commentary available at https://fs.blog/2009/12/mental-model-scientific-method.

varied large-scale software systems; personal preference matters have been neutralized to our best ability. We wrote down the distilled truth that remains, which should shape your understanding of what modern C++ has to offer without being skewed by our subjective opinions or domain-specific inclinations.

The final analysis and interpretation of what is appropriate for your context is left to you, the reader. This book is, by design, not a C++ style or coding-standards guide; it does, however, provide valuable input to any development organization seeking to author or enhance one.

Practicality is important to us in a real-world, economic sense. We examine modern C++ features through the lens of a large company developing and using software in a competitive environment. In addition to showing you how to best utilize a given C++ language feature in practice, our analysis takes into account the costs associated with routinely employing that feature in the ecosystem of a software development organization. Most texts omit the costs of using language features. In other words, we weigh the benefits of successfully using a feature against the hidden cost of its widespread ineffective use (or misuse) and/or the costs associated with training and code review required to reasonably ensure that such ill-conceived use does not occur. We are acutely aware that what applies to one person or a small crew of like-minded individuals is quite different from what works with a large, distributed team. The outcome of this analysis is our signature categorization of features based on how safe they are to adopt — namely, *safe*, *conditionally safe*, or *unsafe* features.

We are not aware of any similar text amid the rich offering of C++ textbooks; we wrote this book because we needed it.

Scope for the First Edition

Given the vastness of C++'s already voluminous and rapidly growing standardized libraries, we have chosen to limit this book's scope to just the language features themselves. A companion book, *Embracing Modern C++ Standard Libraries Safely*, is a separate project that we hope to tackle in the future. To be effective, this book, however, must remain focused on what expert C++ developers need to know well to be successful right now.

We chose to limit the scope of this first edition to only those features that have been included in the language standard since C++11 and widely available in practice for at least five years. This limited focus enables us to better evaluate the real-world impact of these features and to highlight any caveats that might not have been anticipated prior to standardization and sustained, active, and widespread use in industry.

We assume you are quite familiar with essentially all of the basic and important special-purpose features of classic C++98/03, so in this book we confine our attention to just the subset of C++ language features introduced in C++11 and C++14. This book is best for you if you need to know how to safely incorporate C++11/14 language features into a predominately C++98/03 codebase, today.

We are actively planning to cover pre-C++11 material in future editions. For the time being, however, we highly recommend *Effective C++* by Scott Meyers[2] as a concise, practical treatment of many important and useful C++98/03 features.

The *EMC++S* Guiding Principles

Throughout the writing of *Embracing Modern C++ Safely*, we have followed a set of guiding principles, which collectively drive the style and content of this book.

Facts, Not Opinions

This book describes only beneficial uses and potential pitfalls of modern C++ features. The content presented is based on objectively verifiable facts, derived either from standards documents or from extensive practical experience; we explicitly avoid subjective opinions on the relative merits of design trade-offs (restraint that is a good exercise in humility). Although such opinions are often valuable, they are inherently biased toward the author's area of expertise.

Note that *safety* — the rating we use to segregate features by chapter — is the one exception to this objectivity guideline. Although the analysis of each feature aims at being entirely objective, each feature's chapter classification — indicating the relative safety of its quotidian use in a large software-development environment — reflects our combined decades of real-world, hands-on experience developing a variety of large-scale C++ software systems.

Elucidation, Not Prescription

We deliberately avoid prescribing any solutions to address specific feature pitfalls. Instead, we merely describe and characterize such concerns in sufficient detail to equip you to devise a solution suitable for your own development environment. In some cases, we might reference techniques or publicly available libraries that others have used to work around such speed bumps, but we do not pass judgment about which workaround should be considered a best practice.

Thorough, Not Superficial

Embracing Modern C++ Safely is neither designed nor intended to be an introduction to modern C++. This book is a handy reference for experienced C++ programmers who have

[2] **meyers92**

familiarity with earlier versions of the language (C++98/03). Our goal is to provide you with facts, detailed objective analysis, and cogent, real-world examples. By doing so, we spare you the task of wading through material that we presume you already know. If you are entirely unfamiliar with the C++ language, we suggest you start with a more elementary and language-centric text such as *The C++ Programming Language* by Bjarne Stroustrup.[3]

Real-World, Not Contrived, Examples

We hope you will find the examples in this book useful in multiple ways. The primary purpose of the examples is to illustrate productive use of each feature as it might occur in practice. We stay away from contrived examples that give equal importance to seldom-used aspects and to the intended, idiomatic uses of the feature. Hence, many of our examples are based on simplified code fragments extracted from real-world codebases. Though we typically change identifier names to be more appropriate to the shortened example (rather than the context and the process that led to the example), we keep the code structure of each example as close as possible to its original, real-world counterpart.

At Scale, Not Overly Simplistic, Programs

As with many aspects of software development, what works for small programs and teams often doesn't scale to larger development efforts. We attempt to simultaneously capture two distinct aspects of size: (1) the sheer product size (e.g., in bytes, source lines, separate units of release) of the programs, systems, and libraries developed and maintained by a software organization; and (2) the size of an organization itself as measured by the number of software developers, quality-assurance engineers, site-reliability engineers, operators, and so on that the organization employs.

What's more, powerful new language features in the hands of a few expert programmers working together on a prototype for their new start-up don't always fare as well when they are wantonly exercised by dozens or hundreds of developers in a large software-development organization. Hence, when we consider the relative safety of a feature, as defined in the next section, we do so with mindfulness that any given feature might be used — and occasionally misused — in large programs and by a large number of programmers having a wide range of knowledge, skill, and ability.

What Do We Mean by *Safely*?

The ISO C++ Standards Committee, of which we are members, would be remiss — and downright negligent — if it allowed any feature of the C++ language to be standardized if that feature were not reliably safe when used as intended. Still, we have chosen the word

[3] **stroustrup13**

"safely" as the moniker for the signature aspect of our book and the method by which we rank the risk-to-reward ratio for using a given feature in a large-scale development environment. By contextualizing the meaning of the term "safe," we apply it to a real-world economy in which everything has a cost in multiple dimensions: risk of misuse, added maintenance burden borne by using a new feature in an older codebase, and training needs for developers who might not be familiar with that feature.

Several factors impact the value added by the adoption and widespread use of any new language feature, thereby reducing its intrinsic safety. By categorizing features in terms of safety, we strive to capture an appropriately weighted combination of the following factors:

- Number and severity of known deficiencies
- Difficulty in teaching consistent proper use
- Experience level required for consistent proper use
- Risks associated with widespread misuse

In this book, the degree of safety of a given feature is the relative likelihood that widespread use of that feature will have positive impact and no adverse effect on a large software company's codebase.

A *Safe* Feature

Some of the new features of modern C++ add considerable value, are easy to use, and are decidedly hard to misuse unintentionally; hence, ubiquitous adoption of such features is productive, relatively unlikely to become a problem in the context of a large-scale development organization, and generally encouraged — even without training. We identify such staunchly helpful, unflappable C++ features as *safe*.

For example, we categorize the **`override`** contextual keyword as a safe feature because it prevents bugs, serves as documentation, cannot be easily misused, and has no serious deficiencies. If someone has heard of this feature and tried to use it and the software compiles, the codebase is likely better for it. Using **`override`** wherever applicable is always a sound engineering decision.

A *Conditionally Safe* Feature

The vast majority of new features available in modern C++ have important, frequently occurring, and valuable uses, yet how these features are used appropriately, let alone optimally, might not be obvious. What's more, some of these features are fraught with inherent dangers and deficiencies, requiring explicit training and extra care to circumnavigate their pitfalls.

For example, we deem default member initializers a *conditionally safe* feature because, although they are easy to use per se, the perhaps less-than-obvious unintended consequences of doing so (e.g., tight compile-time coupling) might be prohibitively costly in certain circumstances (e.g., might prevent relink-only patching in production).

An *Unsafe* Feature

When an expert programmer uses any C++ feature appropriately, the feature typically does no direct harm. Yet other developers — seeing the feature's use in the codebase but failing to appreciate the highly specialized or nuanced reasoning justifying it — might attempt to use it in what they perceive to be a similar way, yet with profoundly less desirable results. Similarly, maintainers might change the use of a fragile feature, altering its semantics in subtle but damaging ways.

Features that are classified as unsafe are those that might have valid — and even important — use cases, yet our experience indicates that routine or widespread use would be counterproductive in a typical, large-scale, software-development enterprise.

For example, we deem the **`final`** contextual keyword an unsafe feature because the situations in which it would be misused overwhelmingly outnumber those vanishingly few isolated cases in which it is appropriate, let alone valuable. Furthermore, its widespread use would inhibit fine-grained (e.g., hierarchical) reuse, which is critically important to the success of a large organization.

Modern C++ Feature Catalog

This first edition of *Embracing Modern C++ Safely* was designed to serve as a comprehensive catalog of C++11 and C++14 language features, presenting vital information for each in a clear, consistent, and predictable format to which experienced engineers can readily refer during development or technical discourse.

Organization

This book is divided into four chapters, the last three of which form the catalog of modern C++ language features grouped by their respective safety classifications:

- Chapter 0: Introduction
- Chapter 1: *Safe* Features

- Chapter 2: *Conditionally Safe* Features
- Chapter 3: *Unsafe* Features

For this first edition, the language-feature chapters (1, 2, and 3) are divided into two sections containing, respectively, C++11 and C++14 features having the safety level (*safe*, *conditionally safe*, or *unsafe*) corresponding to that chapter. Recall, however, that Standard Library features are outside the scope of this book.

Each feature is presented in a separate section, rendered in a canonical format:

- **Description** — A brisk but comprehensive introduction of the feature's syntax and semantics, supplemented with abundant code snippets. We do our best to avoid using other new features concurrently with the one being described, so each feature can be read independently and out of order. This might lead, on occasion, to code that is less fluent than it could otherwise be. Make sure you consult the "See Also" section (described below) to learn about crosstalk between features.
- **Use Cases** — A collection of tried-and-true use cases distilled from libraries and applications.
- **Potential Pitfalls** — Misuses of the feature that might lead to serious bugs and other problems.
- **Annoyances** — Shortcomings of the feature and unpleasant quirks that might make the feature less pleasant to use.
- **See Also** — Cross-references to other related features within this book along with a brief description of the connection.
- **Further Reading** — References to external sources discussing the feature.

Constraining our treatment of each individual feature to this canonized format facilitates rapid discovery of whatever particular aspects of a given language feature you are searching for.

Note that cross-references to subsections within a feature are in italics, and cross-references to other features are in normal text font. We refer to each feature within its relevant chapter and section: For example, Section 1.1."Attribute Syntax" tells you that the"Attributes" feature is located in Chapter 1 (Safe) and within Section 1 (C++11). Terms that are defined within the glossary are set in a different font, with the first use in each feature being set in **bold**.

The commenting style is worth noting because it conveys good information in a terse format. Note that "description" or "details" provides additional descriptive information. Placeholders for irrelevant and/or unspecified code are shown with stylized comments in one of the following ways:

```
/*...*/
// ...
// ...                              (<description>)
```

Code that does not compile will be marked with one of the following two comments:

```
// Error
// Error, <details>
```

Code that does not link will be marked with one of the following two comments:

```
// Link-Time Error
// Link-Time Error, <details>
```

Code that does not behave as expected at run time will be marked with one of the following two comments:

```
// Bug
// Bug, <details>
```

Code that behaves as expected will be marked with one of the following two comments:

```
// OK
// OK, <details>
```

Code that might warn but behaves as expected would be marked “OK, might warn” or similarly. For example, if a feature is deprecated until C++17 and removed in C++20, we might comment it like this:

```
// OK, deprecated[4] (might warn)
```

How to Use This Book

Depending on your needs, *Embracing Modern C++ Safely* can be handy in a variety of ways.

- **Read the entire book from front to back.** If you are conversant with classic C++, consuming this book in its entirety will provide a complete and nuanced practical understanding of each of the language features introduced by C++11 and C++14.
- **Read the chapters in order but slowly over time.** An incremental, priority-driven approach is also possible and recommended, especially if you’re feeling less sure-footed. Understanding and applying first the *safe* features of Chapter 1 gets you the low-hanging fruit. In time, the *conditionally safe* features of Chapter 2 will allow you to ease into the breadth of useful modern C++ language features, prioritizing those that are least likely to prove problematic.

[4]Removed in C++20

- **Read the C++11 sections of each of the three catalog chapters first.** If you are a developer whose organization uses C++11 but not yet C++14, you can focus on learning everything that can be applied now and then circle back and learn the rest later when it becomes relevant to your evolving organization.

- **Use the book as a quick-reference guide if and as needed.** Random access is great, too, especially now that you've made it through Chapter 0. If you prefer not to read the book in its entirety (or simply want to refer to it periodically as a refresher), reading any arbitrary individual feature section in any order will provide timely access to all relevant details of whichever feature is of immediate interest.

We believe that you will derive value in several ways from the knowledge we imbued into *Embracing Modern C++ Safely*, irrespective of how you read it. In addition to helping you become a more knowledgeable and therefore safer developer, this book aims to clarify (whether you are a developer, a lead, or a manager) which features demand more training, attention to detail, experience, peer review, and such. The factual, objective presentation style also makes for excellent input into the preparation of coding standards and style guides that suit the particular needs of a company, project, team, or even just a single discriminating developer (which, of course, we all aim at being). Finally, any C++ software-development organization that adopts this book will be taking the first steps toward leveraging modern C++ in a way that maximizes reward while minimizing risks, i.e., by embracing modern C++ *safely*.

Last but definitely not least, this is *your* book in more than one sense of the word. It has been a collaborative effort with input from many engineers just like you, and it was "designed for maintenance" because we plan future revised editions with new features and improved treatment of the existing ones. Those future editions could greatly benefit from your contributions. Found something broken or missing? A clever use case? A hidden pitfall? An annoyance you can't stand? We'd be happy to add it to the book. Point your browser to http://emcpps.com, and follow the instructions to send us feedback. Your input will be well received. You'll find more information about the book on the website. Thank you, and happy coding!

The override Member-Function Specifier

Decorating a function in a derived class with the contextual keyword **override** ensures that a **virtual** function having a compatible declaration exists in one or more of its base classes.

Description

The **contextual keyword override** can be provided at the end of a member-function declaration to ensure that the decorated function is indeed *overriding* a corresponding **virtual** member function in a base class, as opposed to *hiding* it or otherwise inadvertently introducing a distinct function declaration:

```
struct Base
{
    virtual void f(int);
            void g(int);
    virtual void h(int) const;
    virtual void i(int) = 0;
};

struct DerivedWithoutOverride : Base
{
    void f();             // hides Base::f(int) (likely mistake)
    void f(int);          // OK, implicitly overrides Base::f(int)

    void g();             // hides Base::g(int) (likely mistake)
    void g(int);          // hides Base::g(int) (likely mistake)

    void h(int);          // hides Base::h(int) const (likely mistake)
    void h(int) const;    // OK, implicitly overrides Base::h(int) const

    void i(int);          // OK, implicitly overrides Base::i(int)
};

struct DerivedWithOverride : Base
{
    void f()          override;    // Error, Base::f() not found
    void f(int)       override;    // OK, explicitly overrides Base::f(int)

    void g()          override;    // Error, Base::g() not found
    void g(int)       override;    // Error, Base::g() is not virtual.

    void h(int)       override;    // Error, Base::h(int) not found
    void h(int) const override;    // OK, explicitly overrides Base::h(int)
```

```
    void i(int)       override;    // OK, explicitly overrides Base::i(int)
};
```

Using this feature expresses design intent so that (1) human readers are aware of it and (2) compilers can validate it.

As noted, **override** is a contextual keyword. C++11 introduces keywords that have special meaning only in certain contexts. In this case, **override** is a keyword in the context of a declaration, but not otherwise using it as the identifier for a variable name, for example, is perfectly fine:

```
int override = 1;  // OK
```

Use Cases

Ensuring that a member function of a base class is being overridden

Consider the following polymorphic hierarchy of error-category classes, as we might have defined them using C++03:

```
struct ErrorCategory
{
    virtual bool equivalent(const ErrorCode& code, int condition);
    virtual bool equivalent(int code, const ErrorCondition& condition);
};

struct AutomotiveErrorCategory : ErrorCategory
{
    virtual bool equivalent(const ErrorCode& code, int condition);
    virtual bool equivolent(int code, const ErrorCondition& condition);
};
```

Notice that there is a defect in the last line of the example above: `equivalent` has been misspelled. Moreover, the compiler did not catch that error. Clients calling `equivalent` on `AutomotiveErrorCategory` will incorrectly invoke the base-class function. If the function in the base class happens to be defined, the code might compile and behave unexpectedly at run time. Now, suppose that over time the interface is changed by marking the equivalence-checking function **const** to bring the interface closer to that of `std::error_category`:

```
struct ErrorCategory
{
    virtual bool equivalent(const ErrorCode& code, int condition) const;
    virtual bool equivalent(int code, const ErrorCondition& condition) const;
};
```

Without applying the corresponding modification to all classes deriving from `ErrorCategory`, the semantics of the program change due to the derived classes now hiding the base class's **`virtual`** member function instead of overriding it. Both errors discussed above would be detected automatically if the **`virtual`** functions in all derived classes were decorated with **`override`**:

```
struct AutomotiveErrorCategory : ErrorCategory
{
    bool equivalent(const ErrorCode& code, int condition) override;
        // Error, failed when base class changed

    bool equivolent(int code, const ErrorCondition& code) override;
        // Error, failed when first written
};
```

What's more, **`override`** serves as a clear indication of the derived-class author's intent to customize the behavior of `ErrorCategory`. For any given member function, using **`override`** necessarily renders any use of **`virtual`** for that function syntactically and semantically redundant. The only cosmetic reason for retaining **`virtual`** in the presence of **`override`** would be that **`virtual`** appears to the left of the function declaration, as it always has, instead of all the way to the right, as **`override`** does now.

Potential Pitfalls

Lack of consistency across a codebase

Relying on **`override`** as a means of ensuring that changes to base-class interfaces are propagated across a codebase can prove unreliable if this feature is used inconsistently, i.e., not applied in every circumstance where its use would be appropriate. In particular, altering the signature of a **`virtual`** member function in a base class and then compiling the entire code base will always flag as an error any nonmatching derived-class function where **`override`** was used but might fail even to warn where it is not.

Further Reading

- Various relationships among **`virtual`**, **`override`**, and **`final`** (see Section 3.1."**`final`**" on page 1007) are presented in **boccara20**.
- Scott Meyers advocates the use of the **`override`** specifier in **meyers15b**, "Item 12: Declare overriding functions `override`," pp. 79–85.

The `[[deprecated]]` Attribute

The standard attribute `[[deprecated]]` indicates that the use of the entity to which the attribute pertains is discouraged, typically in the form of a compiler warning.

Description

The standard `[[deprecated]]` attribute is used to portably indicate that a particular **entity** is no longer recommended and to actively discourage its use. Such deprecation typically follows the introduction of alternative constructs that are superior to the original one, providing time for clients to migrate to them *asynchronously* before the deprecated one is removed in some subsequent release.

An asynchronous process for ongoing improvement of legacy codebases, sometimes referred to as **continuous refactoring**, often allows time for clients to migrate — on their own respective schedules and time frames — from existing *deprecated* constructs to newer ones, rather than having every client change in lock step. Allowing clients time to move *asynchronously* to newer alternatives is often the only viable approach unless (1) the codebase is a closed system, (2) all of the relevant code is governed by a single authority, and (3) the change can be made mechanically.

Although not strictly required, the Standard explicitly encourages[1] conforming compilers to produce a diagnostic message in case a program refers to any **entity** to which the `[[deprecated]]` attribute pertains. For instance, most popular compilers emit a warning whenever a `[[deprecated]]` function or object is used:

```
               void f();
[[deprecated]] void g();

               int a;
[[deprecated]] int b;

void h()
{
    f();
    g();  // Warning: g is deprecated.
    a;
    b;    // Warning: b is deprecated.
}
```

[1]The C++ Standard characterizes what constitutes a well-formed program, but compiler vendors require a great deal of leeway to facilitate the needs of their users. In case any feature induces warnings, command-line options are typically available to disable those warnings (`-Wno-deprecated` in GCC), or methods are in place to suppress those warnings locally, e.g., `#pragma GCC diagnostic ignored "-Wdeprecated"`.

The [[deprecated]] attribute can be used portably to decorate other entities: **class**, **struct**, **union**, type alias, variable, data member, function, enumeration, template specialization.[2]

A programmer can supply a **string literal** as an argument to the [[deprecated]] attribute — e.g., [[deprecated("message")]] — to inform human users regarding the reason for the deprecation:

```
[[deprecated("too slow, use algo1 instead")]] void algo0();
                                              void algo1();

void f()
{
    algo0();  // Warning: algo0 is deprecated; too slow, use algo1 instead.
    algo1();
}
```

An entity that is initially *declared* without [[deprecated]] can later be redeclared with the attribute and vice versa:

```
void f();
void g0() { f(); }  // OK, likely no warnings

[[deprecated]] void f();
void g1() { f(); }  // Warning: f is deprecated.

void f();
void g2() { f(); }  // Warning: f is deprecated still.
```

As shown in g2 in the example above, redeclaring an entity that was previously decorated with [[deprecated]] without the attribute leaves the entity still deprecated.

Use Cases

Discouraging use of an obsolete or unsafe entity

Decorating any entity with the [[deprecated]] attribute serves both to indicate a particular feature should not be used in the future and to actively encourage migration of existing uses to a better alternative. Obsolescence, lack of safety, and poor performance are common motivators for deprecation.

As an example of productive deprecation, consider the RandomGenerator class having a static nextRandom member function to generate random numbers:

[2] Applying [[deprecated]] to a specific enumerator or namespace, however, is guaranteed to be supported only since C++17; see **smith15a**.

```
struct RandomGenerator
{
    static int nextRandom();
        // Generate a random value between 0 and 32767 (inclusive).
};
```

Although such a simple random number generator can be useful, it might become unsuitable for heavy use because good pseudorandom number generation requires more state (and the overhead of synchronizing such state for a single **`static`** function can be a significant performance bottleneck), while good random number generation requires potentially high overhead access to external sources of entropy. The `rand` function, inherited from C and available in C++ through the `<cstdlib>` header, has many of the same issues as our `RandomGenerator::nextRandom` function, and similarly developers are guided to use the facilities provided in the `<random>` header since C++11.

One solution is to provide an alternative random number generator that maintains more state, allows users to decide where to store that state (the random number generator objects), and overall offers more flexibility for clients. The downside of such a change is that it comes with a functionally distinct API, requiring that users update their code to move away from the inferior solution:

```
class StatefulRandomGenerator
{
    // ... (internal state of a quality pseudorandom number generator)

public:
    int nextRandom();
       // Generate a quality random value between 0 and 32767, inclusive.
};
```

Any user of the original random number generator can migrate to the new facility with little effort, but that is not a completely trivial operation, and migration will take some time before the original feature is no longer in use. The empathic maintainers of `RandomGenerator` can decide to use the `[[deprecated]]` attribute to discourage continued use of `RandomGenerator::nextRandom()` instead of removing it completely:

```
struct RandomGenerator
{
    [[deprecated("Use StatefulRandomGenerator class instead.")]]
    static int nextRandom();
        // ...
};
```

By using `[[deprecated]]` as shown in the previous example, existing clients of `RandomGenerator` are informed that a superior alternative, `BetterRandomGenerator`, is available, yet they are granted time to migrate their code to the new solution rather than having their code broken by the removal of the old solution. When clients are notified of the deprecation (thanks to a compiler diagnostic), they can schedule time to rewrite their applications to consume the new interface.

Continuous refactoring is an essential responsibility of a development organization, and deciding when to go back and fix what's suboptimal instead of writing new code that will please users and contribute more immediately to the bottom line will forever be a source of tension. Allowing disparate development teams to address such improvements in their own respective time frames, perhaps subject to some reasonable overall deadline date, is a proven real-world practical way of ameliorating this tension.

Potential Pitfalls

Interaction with treating warnings as errors

In some code bases, compiler warnings are promoted to errors using compiler flags, such as `-Werror` for GCC and Clang or `/WX` for MSVC, to ensure that their builds are warning-clean. For such code bases, use of the `[[deprecated]]` attribute by their dependencies as part of the API might introduce unexpected compilation failures.

Having the compilation process completely stopped due to use of a deprecated entity defeats the purpose of the attribute because users of such an entity are given no time to adapt their code to use a newer alternative. On GCC and Clang, users can selectively demote deprecation errors back to warnings by using the `-Wno-error=deprecated-declarations` compiler flag. On MSVC, however, such demotion of warnings is not possible, and the available workarounds, such as entirely disabling the effects of the `/WX` flag or the deprecation diagnostics using the `-wd4996` flag, are often unsuitable.

Furthermore, this interaction between `[[deprecated]]` and treating warnings as errors makes it impossible for owners of a low-level library to deprecate a function when releasing their code requires that they do not break the ability for *any* of their higher-level clients to compile; a single client using the to-be-deprecated function in a code base that treats warnings as errors prevents the release of the code that uses the `[[deprecated]]` attribute. With the frequent advice given in practice to aggressively treat warnings as errors, the use of `[[deprecated]]` might be completely unfeasible.

Explicit-Instantiation Declarations

The **extern template** prefix can be used to suppress *implicit* generation of local object code for the definitions of particular specializations of class, function, or variable templates used within a translation unit, with the expectation that any suppressed object-code-level definitions will be provided elsewhere within the program by template definitions that are instantiated *explicitly*.

Description

Inherent in the current ecosystem for supporting template programming in C++ is the need to generate redundant definitions of fully specified function and variable templates within .o files. For common instantiations of popular templates, such as std::vector, the increased object-file size, a.k.a. **code bloat**, and potentially extended link times might become significant:

```
#include <vector>    // std::vector is a popular template.
std::vector<int> v;  // std::vector<int> is a common instantiation.

#include <string>    // std::basic_string is a popular template.
std::string s;       // std::string, an alias for std::basic_string<char>, is
                     // a common instantiation.
```

The intent of the **extern template** feature is to *suppress* the implicit generation of duplicative object code within every translation unit in which a fully specialized class template, such as std::vector<**int**> in the code snippet above, is used. Instead, **extern template** allows developers to choose a single translation unit in which to explicitly *generate* object code for all the definitions pertaining to that specific template specialization as explained next.

Explicit-instantiation definition

Creating an **explicit-instantiation definition** was possible prior to C++11.[1] The requisite syntax is to place the keyword **template** in front of the name of the fully specialized class template, function template, or, in C++14, variable template (see Section 1.2. "Variable Templates" on page 157):

[1]The C++03 Standard term for the syntax used to create an explicit-instantiation definition, though rarely used, was **explicit-instantiation directive**. The term explicit-instantiation directive was clarified in C++11 and can now also refer to syntax that is used to create a *declaration* — i.e., **explicit-instantiation declaration**.

```
#include <vector>  // std::vector (general template)

template class std::vector<int>;
    // Deposit all definitions for this specialization into the .o for this
    // translation unit.
```

This explicit-instantiation directive compels the compiler to instantiate *all* functions defined by the named `std::vector` class template having the specified **`int`** template argument; any collateral object code resulting from these instantiations will be deposited in the resulting `.o` file for the current translation unit. Importantly, even functions that are never used are still instantiated, so this solution might not be the correct one for many classes; see *Potential Pitfalls — Accidentally making matters worse* on page 373.

Explicit-instantiation declaration

C++11 introduced the explicit-instantiation declaration, a complement to the explicit-instantiation definition. The newly provided syntax allows us to place **`extern template`** in front of the declaration of an explicit specialization of a class template, a function template, or a variable template:

```
#include <vector>  // std::vector (general template)

extern template class std::vector<int>;
    // Suppress depositing of any object code for std::vector<int> into the
    // .o file for this translation unit.
```

Using the modern **`extern template`** syntax above instructs the compiler to *refrain* from depositing any object code for the named specialization in the current translation unit and instead to rely on some other translation unit to provide any missing object-level definitions that might be needed at link time; see *Annoyances — No good place to put definitions for unrelated classes* on page 373.

Note, however, that declaring an explicit instantiation to be an **`extern template`** *in no way* affects the ability of the compiler to instantiate and to inline visible function-definition bodies for that template specialization in the translation unit:

```
// client.cpp:
#include <vector>  // std::vector (general template)

extern template class std::vector<int>;

void client(std::vector<int>& inOut)  // fully specialized instance of a vector
{
    if (inOut.size())          // This invocation of size can inline.
    {
        int value = inOut[0];  // This invocation of operator[] can be inlined.
    }
}
```

In the previous example, the two tiny member functions of `vector`, namely, `size` and **`operator[]`**, will typically be inlined — in precisely the same way they would have been had the **`extern template`** declaration been omitted. The *only* purpose of an **`extern template`** declaration is to suppress object-code generation for this particular template instantiation for the current translation unit.

Finally, note that the use of explicit-instantiation directives has absolutely no effect on the logical meaning of a well-formed program; in particular, when applied to specializations of function templates, they have no effect on overload resolution:

```
template <typename T> bool f(T v) {/*...*/}  // general template definition

extern template bool f(char c);  // specialization of f for char
extern template bool f(int v);   // specialization of f for int

bool bc = f((char)    0);  // exact match: Object code is suppressed locally.
bool bs = f((short)   0);  // not exact match: Object code is generated locally.
bool bi = f((int)     0);  // exact match: Object code is suppressed locally.
bool bu = f((unsigned)0);  // not exact match: Object code is generated locally.
```

As the example above illustrates, overload resolution and template argument deduction occur independently of any explicit-instantiation declarations. Only *after* the template to be instantiated is determined does the **`extern template`** syntax take effect; see also *Potential Pitfalls — Corresponding explicit-instantiation declarations and definitions* on page 371.

A more complete illustrative example

So far, we have seen the use of explicit-instantiation declarations and explicit-instantiation definitions applied to only a standard *class* template, `std::vector`. The same syntax shown in the previous code snippet applies also to full specializations of individual function templates and variable templates.

As a more comprehensive, albeit largely pedagogical, example, consider the overly simplistic `my::Vector` class template along with other related templates defined within a header file, `my_vector.h`:

```
// my_vector.h:
#ifndef INCLUDED_MY_VECTOR  // internal include guard
#define INCLUDED_MY_VECTOR

#include <cstddef>  // std::size_t
#include <utility>  // std::swap

namespace my  // namespace for all entities defined within this component
{

template <typename T>
class Vector
```

```
{
    static std::size_t s_count;     // track number of objects constructed
    T*                 d_data_p;    // pointer to dynamically allocated memory
    std::size_t        d_length;    // current number of elements in the vector
    std::size_t        d_capacity;  // number of elements currently allocated

public:
    // ...

    std::size_t length() const { return d_length; }
        // Return the number of elements.

    // ...
};

// ...          Any partial or full specialization definitions            ...
// ...          of the class template Vector go here.                     ...

template <typename T>
void swap(Vector<T> &lhs, Vector<T> &rhs) { return std::swap(lhs, rhs); }
    // free function that operates on objects of type my::Vector via ADL

// ...            Any [full] specialization definitions                   ...
// ...            of free function swap would go here.                    ...

template <typename T>
const std::size_t vectorSize = sizeof(Vector<T>);  // C++14 variable template
    // This nonmodifiable static variable holds the size of a my::Vector<T>.

// ...           Any [full] specialization definitions                    ...
// ...           of variable vectorSize would go here.                    ...

template <typename T>
std::size_t Vector<T>::s_count = 0;
    // definition of static counter in general template

// ... We might opt to add explicit-instantiation declarations here.
// ...

}  // Close my namespace.

#endif  // Close internal include guard.
```

In the `my_vector` component in the code snippet above, we have defined the following, in the `my` namespace.

1. A **`class`** template, `Vector`, parameterized on element type

2. A free-function template, `swap`, that operates on objects of corresponding specialized `Vector` type

3. A **const** C++14 variable template, `vectorSize`, that represents the number of bytes in the **footprint** of an object of the corresponding specialized `Vector` type

Any use of these templates by a client might and typically will trigger the depositing of equivalent definitions as object code in the client translation unit's resulting `.o` file, irrespective of whether the definition being used winds up getting inlined.

To eliminate object code for specializations of entities in the `my_vector` component, we must first decide where the unique definitions will go; see *Annoyances — No good place to put definitions for unrelated classes* on page 373. In this specific case, we own the component that requires specialization, and the specialization is for a ubiquitous built-in type; hence, the natural place to generate the specialized definitions is in a `.cpp` file corresponding to the component's header:

```
// my_vector.cpp:
#include <my_vector.h>  // We always include the component's own header first.
    // By including this header file, we have introduced the general template
    // definitions for each of the explicit-instantiation declarations below.

namespace my  // namespace for all entities defined within this component
{

template class Vector<int>;
    // Generate object code for all nontemplate member functions and definitions
    // of static data members of template my::Vector having int elements.

template std::size_t Vector<double>::length() const;  // BAD IDEA
    // In addition, we could generate object code for just a particular member
    // function definition of my::Vector (e.g., length) for some other
    // argument type (e.g., double).

template void swap(Vector<int>& lhs, Vector<int>& rhs);
    // Generate object code for the full specialization of the swap free-
    // function template that operates on objects of type my::Vector<int>.

template const std::size_t vectorSize<int>;  // C++14 variable template
    // Generate the object-code-level definition for the specialization of the
    // C++14 variable template instantiated for built-in type int.

template std::size_t Vector<int>::s_count;
    // Generate the object-code-level definition for the specialization of the
    // static member variable of Vector instantiated for built-in type int.

}  // Close my namespace.
```

Each of the constructs introduced by the keyword **template** within the `my` namespace in the previous example represents a separate explicit-instantiation definition. These constructs instruct the compiler to generate object-level definitions for general templates declared in `my_vector.h` specialized on the built-in type **int**. Explicit instantiation of individual member functions, such as `length()` in the example, is, however, only rarely useful; see *Annoyances — All members of an explicitly defined template class must be valid* on page 374.

Having installed the necessary explicit-instantiation definitions in the component's `my_vector.cpp` file, we must now go back to its `my_vector.h` file and, without altering any of the previously existing lines of code, *add* the corresponding explicit-instantiation declarations to suppress redundant local code generation:

```
// my_vector.h:
#ifndef INCLUDED_MY_VECTOR  // internal include guard
#define INCLUDED_MY_VECTOR

namespace my  // namespace for all entities defined within this component
{

// ...
// ...  everything that was in the original my namespace
// ...

                    // ----------------------------------
                    // explicit-instantiation declarations
                    // ----------------------------------

extern template class Vector<int>;
    // Suppress object code for this class template specialized for int.

extern template std::size_t Vector<double>::length() const;  // BAD IDEA
    // Suppress object code for this member, only specialized for double.

extern template void swap(Vector<int>& lhs, Vector<int>& rhs);
    // Suppress object code for this free function specialized for int.

extern template std::size_t vectorSize<int>;  // C++14
    // Suppress object code for this variable template specialized for int.

extern template std::size_t Vector<int>::s_count;
    // Suppress object code for this static member definition w.r.t. int.

}  // Close my namespace.

#endif  // Close internal include guard.
```

Each of the constructs that begins with **extern template** in the example above are explicit-instantiation declarations, which serve only to suppress the generation of any object code emitted to the `.o` file of the current translation unit in which such specializations are used. These added **`extern template`** declarations must appear in `my_header.h` *after* the declaration of the corresponding general template and, importantly, before whatever relevant definitions are ever used.

The effect on various `.o` files

To illustrate the effect of explicit-instantiation declarations and explicit-instantiation definitions on the contents of object and executable files, we'll use a simple `lib_interval` library **component** consisting of a header file, `lib_interval.h`, and an implementation file, `lib_interval.cpp`. The latter, apart from including its corresponding header, is effectively empty:

```
// lib_interval.h:
#ifndef INCLUDED_LIB_INTERVAL  // internal include guard
#define INCLUDED_LIB_INTERVAL

namespace lib  // namespace for all entities defined within this component
{

template <typename T>  // elided definition of a class template
class Interval
{
    T d_low;   // interval's low value
    T d_high;  // interval's high value

public:
    explicit Interval(const T& p) : d_low(p), d_high(p) { }
        // Construct an empty interval.

    Interval(const T& low, const T& high) : d_low(low), d_high(high) { }
        // Construct an interval having the specified boundary values.

    const T& low() const { return d_low; }
        // Return this interval's low value.

    const T& high() const { return d_high; }
        // Return this interval's high value.

    int length() const { return d_high - d_low; }
        // Return this interval's length.

    // ...
};

template <typename T>  // elided definition of a function template
bool intersect(const Interval<T>& i1, const Interval<T>& i2)
    // Determine whether the specified intervals intersect.
```

```
{
    bool result = false;  // nonintersecting until proven otherwise
    // ...
    return result;
}

}  // Close lib namespace.

#endif  // INCLUDED_LIB_INTERVAL
```

```
// lib_interval.cpp:
#include <lib_interval.h>
```

This library component above defines, in the namespace `lib`, an implementation of (1) a class template, `Interval`, and (2) a function template, `intersect`.

Let's also consider a trivial application that uses this library component:

```
// app.cpp:
#include <lib_interval.h>  // Include the library component's header file.

int main(int argv, const char** argc)
{
    lib::Interval<double> a(0, 5);  // instantiate with double type argument
    lib::Interval<double> b(3, 8);  // instantiate with double type argument
    lib::Interval<int>    c(4, 9);  // instantiate with int    type argument

    if (lib::intersect(a, b))  // instantiate deducing double type argument
    {
        return 0;  // Return "success" as (0.0, 5.0) does intersect (3.0, 8.0).
    }

    return 1;  // Return "failure" status as function apparently doesn't work.
}
```

The purpose of this application is merely to exhibit a couple of instantiations of the library *class* template, `lib::Interval`, for type arguments **`int`** and **`double`**, and of the library *function* template, `lib::intersect`, for just **`double`**.

Next, we compile the application and library translation units, `app.cpp` and `lib_interval.cpp`, and inspect the symbols in their respective corresponding object files, `app.o` and `lib_interval.o`:

```
$ gcc -I. -c app.cpp lib_interval.cpp
$ nm -C app.o lib_interval.o

app.o:
0000000000000000 W lib::Interval<double>::Interval(double const&, double const&)
```

```
0000000000000000 W lib::Interval<int>::Interval(int const&, int const&)
0000000000000000 W bool lib::intersect<double>(lib::Interval<double> const&,
                                               lib::Interval<double> const&)
0000000000000000 T main

lib_interval.o:
```

Looking at `app.o` in the previous example, the class and function templates used in the `main` function, which is defined in the `app.cpp` file, were instantiated *implicitly*, and the relevant code was added to the resulting object file, `app.o`, with each instantiated function definition in its own separate **section**. In the `Interval` *class* template, the generated symbols correspond to the two unique instantiations of the constructor, i.e., for **`double`** and **`int`**, respectively. The `intersect` function template, however, was implicitly instantiated for only type **`double`**. Note that all of the implicitly instantiated functions have the `W` symbol type, indicating that they are *weak* symbols, which are permitted to be present in multiple object files. By contrast, this file also defines the *strong* symbol `main`, marked here by a `T`. Linking `app.o` with any other object file containing such a symbol would cause the linker to report a multiply-defined-symbol error. On the other hand, the `lib_interval.o` file corresponds to the `lib_interval` library component, whose `.cpp` file served only to include its own `.h` file, and is again effectively empty.

Let's now link the two object files, `app.o` and `lib_interval.o`, and inspect the symbols in the resulting executable, `app`[2]:

```
$ gcc -o app app.o lib_interval.o
$ nm -C app
000000000040056e W lib::Interval<double>::Interval(double const&, double const&)
00000000004005a2 W lib::Interval<int>::Interval(int const&, int const&)
00000000004005ce W bool lib::intersect<double>(lib::Interval<double> const&,
                                               lib::Interval<double> const&)
00000000004004b7 T main
```

As the textual output above confirms, the final program contains exactly one copy of each weak symbol. In this tiny illustrative example, these weak symbols have been defined in only a single object file, thus not requiring the linker to select one definition out of many.

More generally, if the application comprises multiple object files, each file will potentially contain their own set of weak symbols, often leading to duplicate code sections for implicitly instantiated class, function, and variable templates instantiated on the same type arguments. When the linker combines object files, it will arbitrarily choose at most one of each of these respective and ideally identical weak-symbol sections to include in the final executable.

Imagine now that our program includes a large number of object files, many of which make use of our `lib_interval` component, particularly to operate on **`double`** intervals.

[2]We have stripped out extraneous unrelated information that the `nm` tool produces; note that the `-C` option invokes the symbol demangler, which turns encoded names like `_ZN3lib8IntervalIdEC1ERKdS3_` into something more readable like `lib::Interval<double>::Interval(double const&, double const&)`.

Suppose, for now, we decide we would like to suppress the generation of object code for templates related to just **`double`** type with the intent of later putting them all in one place, i.e., the currently empty `lib_interval.o`. Achieving this objective is precisely what the **`extern template`** syntax is designed to accomplish.

Returning to our `lib_interval.h` file, we need not change one line of code; we need only to *add* two explicit-instantiation declarations — one for the *class* template, `Interval<double>`, and one for the *function* template, `intersect<double>(const double&, const double&)` — to the header file anywhere *after* their respective corresponding general template declaration and definition:

```
// lib_interval.h:  // No change to existing code.
#ifndef INCLUDED_LIB_INTERVAL  // internal include guard
#define INCLUDED_LIB_INTERVAL

namespace lib  // namespace for all entities defined within this component
{

template <typename T>
class Interval
{
    // ...  (same as before)
};

template <typename T>
bool intersect(const Interval<T>& i1, const Interval<T>& i2)
{
    // ...  (same as before)
}

extern template class Interval<double>;  // explicit-instantiation declaration

extern template                          // explicit-instantiation declaration
bool intersect(const Interval<double>&, const Interval<double>&);

}  // close lib namespace

#endif  // INCLUDED_LIB_INTERVAL
```

Let's again compile the two `.cpp` files and inspect the corresponding `.o` files:

```
$ gcc -I. -c app.cpp lib_interval.cpp
$ nm -C app.o lib_interval.o

app.o:
                 U lib::Interval<double>::Interval(double const&, double const&)
```

```
0000000000000000 W lib::Interval<int>::Interval(int const&, int const&)
                 U bool lib::intersect<double>(lib::Interval<double> const&,
                                               lib::Interval<double> const&)
0000000000000000 T main

lib_interval.o:
```

Notice that this time some of the symbols, specifically those relating to the class and function templates instantiated for type **double**, have changed from `W`, indicating a *weak* symbol, to `U`, indicating an *undefined* one. This symbol type change means that instead of generating a weak symbol for the explicit specializations for **double**, the compiler left those symbols undefined, as if only the *declarations* of the member and free-function templates had been available when compiling `app.cpp`, yet inlining of the instantiated definitions is in no way affected. **Undefined symbols** are expected to be made available to the linker from other object files. Attempting to link this application expectedly fails because no object files being linked contain the needed definitions for those instantiations:

```
$ gcc -o app app.o lib_interval.o

app.o: In function 'main':
app.cpp:(.text+0x38): undefined reference to
  `lib::Interval<double>::Interval(double const&, double const&)'
app.cpp:(.text+0x69): undefined reference to
  `lib::Interval<double>::Interval(double const&, double const&)'
app.cpp:(.text+0xa1): undefined reference to
  `bool lib::intersect<double>(lib::Interval<double> const&,
                               lib::Interval<double> const&)'

collect2: error: ld returned 1 exit status
```

To provide the missing definitions, we will need to instantiate them explicitly. Since the type for which the class and function are being specialized is the ubiquitous built-in type, **double**, the ideal place to sequester those definitions would be within the object file of the `lib_interval` library component itself, but see *Annoyances — No good place to put definitions for unrelated classes* on page 373. To force the needed template definitions into the `lib_interval.o` file, we will need to use explicit-instantiation definition syntax, i.e., the **template** prefix:

```
// lib_interval.cpp:
#include <lib_interval.h>

template class lib::Interval<double>;
    // example of an explicit-instantiation definition for a class

template bool lib::intersect(const Interval<double>&, const Interval<double>&);
    // example of an explicit-instantiation definition for a function
```

We recompile once again and inspect our newly generated object files:

```
$ gcc -I. -c app.cpp lib_interval.cpp
$ nm -C app.o lib_interval.o

app.o:
                 U lib::Interval<double>::Interval(double const&, double const&)
0000000000000000 W lib::Interval<int>::Interval(int const&, int const&)
                 U bool lib::intersect<double>(lib::Interval<double> const&,
                                               lib::Interval<double> const&)
0000000000000000 T main

lib_interval.o:
0000000000000000 W lib::Interval<double>::Interval(double const&)
0000000000000000 W lib::Interval<double>::Interval(double const&, double const&)
0000000000000000 W lib::Interval<double>::low() const
0000000000000000 W lib::Interval<double>::high() const
0000000000000000 W lib::Interval<double>::length() const
0000000000000000 W bool lib::intersect<double>(lib::Interval<double> const&,
                                               lib::Interval<double> const&)
```

The application object file, `app.o`, naturally remained unchanged. What's new here is that the functions that were missing from the `app.o` file are now available in the `lib_interval.o` file, again as *weak* (`W`), as opposed to strong (`T`), symbols. Notice, however, that explicit instantiation forces the compiler to generate code for all of the member functions of the class template for a given specialization. These symbols might all be linked into the resulting executable unless we take explicit precautions to exclude those that aren't needed[3]:

```
$ gcc -o app app.o lib_interval.o -Wl,--gc-sections
$ nm -C app
00000000004005ca W lib::Interval<double>::Interval(double const&, double const&)
000000000040056e W lib::Interval<int>::Interval(int const&, int const&)
000000000040063d W bool lib::intersect<double>(lib::Interval<double> const&,
                                               lib::Interval<double> const&)
00000000004004b7 T main
```

The **`extern template`** feature is provided to enable software architects to reduce code bloat in individual object files for common instantiations of class, function, and, as of C++14, variable templates in large-scale C++ software systems. The practical benefit is in reducing the physical size of libraries, which *might* lead to improved link times. Explicit-instantiation declarations do *not* (1) affect the meaning of a program, (2) suppress inline template implicit instantiation, (3) impede the compiler's ability to **`inline`**, or (4) meaningfully improve

[3]To avoid including the explicitly generated definitions that are being used to resolve undefined symbols, we have instructed the linker to remove all unused code sections from the executable. The `-Wl` option passes comma-separated options to the linker. The `--gc-sections` option instructs the compiler to compile and assemble and instructs the linker to omit individual unused sections, where each section contains, for example, its own instantiation of a function template.

compile time. To be clear, the *only* purpose of the **extern template** syntax is to suppress object-code generation for the current translation unit, which is then selectively overridden in the translation unit(s) of choice.

Use Cases

Reducing template code bloat in object files

The motivation for the **extern template** syntax is as a purely compile-time, not runtime, optimization, i.e., to reduce the amount of redundant code within individual object files resulting from common template instantiations in client code. As an example, consider a fixed-size-array class template, `FixedArray`, that is used widely, i.e., by many clients from separate translation units, in a large-scale `game` project for both integral and floating-point calculations, primarily with type arguments **int** and **double** and array sizes of either `2` or `3`:

```
// game_fixedarray.h:
#ifndef INCLUDED_GAME_FIXEDARRAY  // internal include guard
#define INCLUDED_GAME_FIXEDARRAY

#include <cstddef>  // std::size_t

namespace game  // namespace for all entities defined within this component
{

template <typename T, std::size_t N>  // widely used class template
class FixedArray
{
    // ... (elided private implementation details)
public:
    FixedArray()                                { /*...*/ }
    FixedArray(const FixedArray<T, N>& other)   { /*...*/ }
    T& operator[](std::size_t index)            { /*...*/ }
    const T& operator[](std::size_t index) const { /*...*/ }
};

template <typename T, std::size_t N>
T dot(const FixedArray<T, N>& a, const FixedArray<T, N>& b) { /*...*/ }
    // Return the scalar ("dot") product of the specified 'a' and 'b'.

// Explicit-instantiation declarations for full template specializations
// commonly used by the game project are provided below.

extern template class FixedArray<int, 2>;              // class template
extern template int dot(const FixedArray<int, 2>& a,   // function template
                        const FixedArray<int, 2>& b);  // for int and 2
```

```
extern template class FixedArray<int, 3>;              // class template
extern template int dot(const FixedArray<int, 3>& a,   // function template
                        const FixedArray<int, 3>& b);  // for int and 3

extern template class FixedArray<double, 2>;           // for double and 2
extern template double dot(const FixedArray<double, 2>& a,
                           const FixedArray<double, 2>& b);

extern template class FixedArray<double, 3>;           // for double and 3
extern template double dot(const FixedArray<double, 3>& a,
                           const FixedArray<double, 3>& b);

}  // Close game namespace.

#endif  // INCLUDED_GAME_FIXEDARRAY
```

Specializations commonly used by the `game` project are provided by the `game` library. In the component header in the example above, we have used the **`extern template`** syntax to suppress object-code generation for instantiations of both the class template `FixedArray` and the function template `dot` for element types **`int`** and **`double`**, each for array sizes `2` and `3`. To ensure that these specialized definitions are available in every program that might need them, we use the **`template`** syntax counterpart to *force* object-code generation within just the one `.o` corresponding to the `game_fixedarray` library component[4]:

```
// game_fixedarray.cpp:
#include <game_fixedarray.h>  // included as first substantive line of code

// Explicit-instantiation definitions for full template specializations
// commonly used by the game project are provided below.

template class game::FixedArray<int, 2>;              // class template
template int game::dot(const FixedArray<int, 2>& a,   // function template
                       const FixedArray<int, 2>& b);  // for int and 2

template class game::FixedArray<int, 3>;              // class template
template int game::dot(const FixedArray<int, 3>& a,   // function template
                       const FixedArray<int, 3>& b);  // for int and 3

template class game::FixedArray<double, 2>;           // for double and 2
template double game::dot(const FixedArray<double, 2>& a,
                          const FixedArray<double, 2>& b);
```

[4]Notice that we have chosen *not* to nest the explicit specializations — or any other definitions — of entities already declared directly within the `game` namespace, preferring instead to qualify each entity explicitly to be consistent with how we render free-function definitions to avoid self-declaration; see **lakos20**, section 2.5, "Component Source-Code Organization," pp. 333–342, specifically Figure 2-36b, p. 340. See also *Potential Pitfalls — Corresponding explicit-instantiation declarations and definitions* on page 371.

```
template class game::FixedArray<double, 3>;          // for double and 3
template double game::dot(const FixedArray<double, 3>& a,
                          const FixedArray<double, 3>& b);
```

Compiling `game_fixedarray.cpp` and examining the resulting object file shows that the code for all explicitly instantiated classes and free functions was generated and placed into the object file, `game_fixedarray.o`, of which we show a subset of the relevant symbols:

```
$ gcc -I. -c game_fixedarray.cpp
$ nm -C game_fixedarray.o
0000000000000000 W game::FixedArray<double, 2ul>::FixedArray(
  game::FixedArray<double, 2ul> const&)
0000000000000000 W game::FixedArray<double, 2ul>::FixedArray()
0000000000000000 W game::FixedArray<double, 2ul>::operator[](unsigned long)
0000000000000000 W game::FixedArray<double, 3ul>::FixedArray(
  game::FixedArray<double, 3ul> const&)
0000000000000000 W game::FixedArray<int, 3ul>::FixedArray()
                                    :
0000000000000000 W double game::dot<double, 2ul>(
  game::FixedArray<double, 2ul> const&, game::FixedArray<double, 2ul> const&)
0000000000000000 W double game::dot<double, 3ul>(
  game::FixedArray<double, 3ul> const&, game::FixedArray<double, 3ul> const&)
0000000000000000 W int game::dot<int, 2ul>(
  game::FixedArray<int, 2ul> const&, game::FixedArray<int, 2ul> const&)
                                    :
0000000000000000 W game::FixedArray<int, 2ul>::operator[](unsigned long) const
0000000000000000 W game::FixedArray<int, 3ul>::operator[](unsigned long) const
```

This `FixedArray` class template is used in multiple translation units within the `game` project. The first one contains a set of geometry utilities:

```
// app_geometryutil.cpp:

#include <game_fixedarray.h>  // game::FixedArray
#include <game_unit.h>        // game::Unit

using namespace game;

void translate(Unit* object, const FixedArray<double, 2>& dst)
    // Perform precise movement of the object on 2D plane.
{
    FixedArray<double, 2> objectProjection;
    // ...
}

void translate(Unit* object, const FixedArray<double, 3>& dst)
    // Perform precise movement of the object in 3D space.
```

```
{
    FixedArray<double, 3> delta;
    // ...
}

bool isOrthogonal(const FixedArray<int, 2>& a1, const FixedArray<int, 2>& a2)
    // Return true if 2d arrays are orthogonal.
{
    return dot(a1, a2) == 0;
}

bool isOrthogonal(const FixedArray<int, 3>& a1, const FixedArray<int, 3>& a2)
    // Return true if 3d arrays are orthogonal.
{
    return dot(a1, a2) == 0;
}
```

The second one deals with physics calculations:

```
// app_physics.cpp:

#include <game_fixedarray.h>  // game::FixedArray
#include <game_unit.h>        // game::Unit

using namespace game;

void collide(Unit* objectA, Unit* objectB)
    // Calculate the result of object collision in 3D space.
{
    FixedArray<double, 3> centerOfMassA = objectA->centerOfMass();
    FixedArray<double, 3> centerOfMassB = objectB->centerOfMass();
    // ..
}

void accelerate(Unit* object, const FixedArray<double, 3>& force)
    // Calculate the position after applying a specified force for the
    // duration of a game tick.
{
    // ...
}
```

Note that the object files for the application components throughout the `game` project do not contain any of the implicitly instantiated definitions that we had chosen to uniquely sequester externally, i.e., within the `game_fixedarray.o` file:

```
$ nm -C app_geometryutil.o
000000000000003e T isOrthogonal(game::FixedArray<int, 2ul> const&,
  game::FixedArray<int, 2ul> const&)
```

```
0000000000000068 T isOrthogonal(game::FixedArray<int, 3ul> const&,
  game::FixedArray<int, 3ul> const&)
0000000000000000 T translate(game::Unit*, game::FixedArray<double, 2ul> const&)
000000000000001f T translate(game::Unit*, game::FixedArray<double, 3ul> const&)
                 U game::FixedArray<double, 2ul>::FixedArray()
                 U game::FixedArray<double, 3ul>::FixedArray()
                 U int game::dot<int, 2ul>(game::FixedArray<int, 2ul> const&,
  game::FixedArray<int, 2ul> const&)
                 U int game::dot<int, 3ul>(game::FixedArray<int, 3ul> const&,
  game::FixedArray<int, 3ul> const&)

$ nm -C app_physics.o
0000000000000039 T accelerate(game::Unit*,
  game::FixedArray<double, 3ul> const&)
0000000000000000 T collide(game::Unit*, game::Unit*)
                 U game::FixedArray<double, 3ul>::FixedArray()
0000000000000000 W game::Unit::centerOfMass()
```

Whether optimization involving explicit-instantiation directives reduces library sizes on disk has no noticeable effect or actually makes matters worse will depend on the particulars of the system at hand. Having this optimization applied to frequently used templates across a large organization has been known to decrease object file sizes, storage needs, link times, and overall build times, but see *Potential Pitfalls — Accidentally making matters worse* on page 373.

Insulating template definitions from clients

Even before the introduction of explicit-instantiation declarations, strategic use of explicit-instantiation definitions made it possible to **insulate** the *definition* of a template from client code, presenting instead just a limited set of instantiations against which clients may link. Such insulation enables the definition of the template to change without forcing clients to recompile. What's more, new explicit instantiations can be added without affecting existing clients.

As an example, suppose we have a single free-function template, `transform`, that operates on only floating-point values:

```
// transform.h:
#ifndef INCLUDED_TRANSFORM
#define INCLUDED_TRANSFORM

template <typename T>  // declaration only of free-function template
T transform(const T& value);
    // Return the transform of the specified floating-point value.

#endif
```

Initially, this function template will support just two built-in types, **float** and **double**, but it is anticipated to eventually support the additional built-in type **long double** and perhaps even supplementary user-defined types (e.g., `Float128`) to be made available via separate headers (e.g., `float128.h`). By placing only the declaration of the `transform` function template in its component's header, clients will be able to link against only two supported explicit specializations provided in the `transform.cpp` file:

```
// transform.cpp:
#include <transform.h>  // Ensure consistency with client-facing declaration.

template <typename T>   // redeclaration/definition of free-function template
T transform(const T& value)
{
    // insulated implementation of transform function template
}

// explicit-instantiation definitions
template float transform(const float&);    // Instantiate for type float.
template double transform(const double&);  // Instantiate for type double.
```

Without the two explicit-instantiation declarations in the `transform.cpp` file above, its corresponding object file, `transform.o`, would be empty.

Note that, as of C++11, we *could* place the corresponding explicit-instantiation declarations in the header file for, say, documentation purposes:

```
// transform.h:
#ifndef INCLUDED_TRANSFORM
#define INCLUDED_TRANSFORM

template <typename T>  // declaration only of free-function template
T transform(const T& value);
    // Return the transform of the specified floating-point value.

// explicit-instantiation declarations, available as of C++11
extern template float transform(const float&);    // user documentation only;
extern template double transform(const double&);  // has no effect whatsoever

#endif
```

Because no definition of the `transform` free-function template is visible in the header, no *implicit* instantiation can result from client use; hence, the two explicit-instantiation declarations above for **float** and **double**, respectively, do nothing.

Potential Pitfalls

Corresponding explicit-instantiation declarations and definitions

To realize a reduction in object-code size for individual translation units and yet still be able to link all valid programs successfully into a well-formed program, four moving parts have to be brought together correctly.

1. Each general template, `C<T>`, whose object code bloat is to be optimized must be declared within some designated component's header file, `c.h`.
2. The specific definition of each `C<T>` relevant to an explicit specialization being optimized — including general, partial-specialization, and full-specialization definitions — must appear in the header file prior to its corresponding explicit-instantiation declaration.
3. Each explicit-instantiation declaration for each specialization of each separate top-level — i.e., class, function, or variable — template must appear in the component's `.h` file *after* the corresponding general template declaration and the relevant general, partial-specialization, or full-specialization definition, but, in practice, always after *all* such definitions, not just the relevant one.
4. Each template specialization having an explicit-instantiation declaration in the header file must have a corresponding explicit-instantiation definition in the component's implementation file, `c.cpp`.

Absent items (1) and (2), clients would have no way to safely separate out the usability and inlineability of the template definitions yet consolidate the otherwise redundantly generated object-level definitions within just a single translation unit. Moreover, failing to provide the relevant definition would mean that any clients using one of these specializations would either fail to compile or, arguably worse, pick up the general definitions when a more specialized definition was intended, likely resulting in an ill-formed program.

Failing item (3), the object code for that particular specialization of that template will be generated locally in the client's translation unit as usual, negating any benefits with respect to local object-code size, irrespective of what is specified in the `c.cpp` file.

Finally, unless we provide a matching explicit-instantiation definition in the `c.cpp` file for each and every corresponding explicit-instantiation declaration in the `c.h` file as in item (4), our optimization attempts might well result in a library component that compiles, links, and even passes some unit tests but, when released to our clients, fails to link. Additionally, any explicit-instantiation definition in the `c.cpp` file that is not accompanied by a corresponding explicit-instantiation declaration in the `c.h` file will inflate the size of the `c.o` file with no possibility of reducing code bloat in client code:

```
// c.h:
#ifndef INCLUDED_C                            // internal include guard
#define INCLUDED_C

template <typename T> void f(T v) {/*...*/}  // general template definition

extern template void f<int>(int v);           // OK, matched in c.cpp
extern template void f<char>(char c);         // Error, unmatched in .cpp file

#endif


// c.cpp:
#include <c.h>                                 // incorporate own header first

template void f<int>(int v);                   // OK, matched in c.h
template void f<double>(double v);             // Bug, unmatched in c.h file


// client.cpp:
#include <c.h>

void client()
{
    int    i = 1;
    char   c = 'a';
    double d = 2.0;

    f(i);  // OK, matching explicit-instantiation directives
    f(c);  // Link-Time Error, no matching explicit-instantiation definition
    f(d);  // Bug, size increased due to no matching explicit-instantiation
           // declaration.
}
```

In the example above, `f(i)` works as expected, with the linker finding the definition of `f<int>` in `c.o`; `f(c)` fails to link because no definition of `f<char>` is guaranteed to be found anywhere; and `f(d)` accidentally works by silently generating a *redundant* local copy of `f<double>` in `client.o`, while another, identical definition is generated explicitly in `c.o`. These extra instantiations do not result in multiply-defined symbols because they still reside in their own sections and are marked as *weak* symbols. Importantly, note that **`extern template`** has *absolutely no effect* on overload resolution because the call to `f(c)` did *not* resolve to `f<int>`.

Accidentally making matters worse

When making the decision to explicitly instantiate common specializations of popular templates within some designated object file, it is important to consider that not all programs necessarily need every (or even any) such instantiation. Classes that have many member functions but typically use only a few require special attention.

For such classes, it might be beneficial to explicitly instantiate individual member functions instead of the entire class template. However, selecting *which* member functions to explicitly instantiate and with *which* template arguments they should be instantiated without carefully measuring the effect on the overall object size might result in not only overall pessimization, but also to an unnecessary maintenance burden. Finally, remember that one might need to explicitly tell the linker to strip unused sections resulting, for example, from forced instantiation of common template specializations, to avoid inadvertently bloating executables, which could adversely affect load times.

Annoyances

No good place to put definitions for unrelated classes

When we consider the implications of physical dependency,[5,6] determining in which component to deposit the specialized definitions can be problematic. For example, consider a codebase implementing a core library that provides both a nontemplated `String` class and a `Vector` container class template. These fundamentally unrelated entities would ideally live in separate physical components (i.e., `.h`/`.cpp` pairs), neither of which depends physically on the other. That is, an application using just one of these components could be compiled, linked, tested, and deployed entirely independently of the other. Now, consider a large codebase that makes heavy use of `Vector<String>`: In what component should the object-code-level definitions for the `Vector<String>` specialization reside?[7] There are two obvious alternatives.

1. `vector` — In this case, `vector.h` would hold `extern template class Vector<String>;` — the explicit-instantiation declaration. `vector.cpp` would hold `template class Vector<String>;` — the explicit-instantiation definition. With this approach, we would create a physical dependency of the `vector` component on `string`. Any client program wanting to use a `Vector` would also depend on `string` regardless of whether it was needed.

[5] See **lakos96**.
[6] See **lakos20**.
[7] Note that the problem of determining in which component to instantiate the object-level implementation of a template for a user-defined type is similar to that of specializing an arbitrary user-defined trait for a user-defined type.

2. `string` — In this case, `string.h` and `string.cpp` would instead be modified so as to depend on `vector`. Clients wanting to use a `string` would also be forced to depend physically on `vector` *at compile time*.

Another possibility might be to create a third component, called `stringvector`, that itself depends on both `vector` and `string`. By **escalating**[8] the mutual dependency to a higher level in the physical hierarchy, we avoid forcing any client to depend on more than what is actually needed. The practical drawback to this approach is that only those clients that proactively include the composite `stringvector.h` header would realize any benefit; fortunately, in this case, there is no **one-definition rule (ODR)** violation if they don't.

Finally, complex machinery could be added to both `string.h` and `vector.h` to conditionally include `stringvector.h` whenever both of the other headers are included; such heroic efforts would, nonetheless, involve a **cyclic physical dependency** among all three of these components. Circular intercomponent collaborations are best avoided.[9]

All members of an explicitly defined template class must be valid

In general, when using a class template, only those members that are actually used get implicitly instantiated. This hallmark allows class templates to provide functionality for parameter types having certain capabilities, e.g., default constructible, while also providing partial support for types lacking those same capabilities. When providing an **explicit-instantiation definition**, however, *all* members of a class template are instantiated.

Consider a simple class template having a data member that can be either default-initialized via the template's default constructor or initialized with an instance of the member's type supplied at construction:

```
template <typename T>
class W
{
    T d_t;  // a data member of type T

public:
    W() : d_t() {}
        // Create an instance of W with a default-constructed T member.

    W(const T& t) : d_t(t) {}
        // Create an instance of W with a copy of the specified t.

    void doStuff() { /* do stuff */ }
};
```

This class template can be used successfully with a type, such as `U` in the following code snippet, that is not default constructible:

[8] **lakos20**, section 3.5.2, "Escalation," pp. 604–614

[9] **lakos20**, section 3.4, "Avoiding Cyclic Link-Time Dependencies," pp. 592–601

```
struct U
{
    U(int i) { /* construct from i */ }
    // ...
};

void useWU()
{
    W<U> wu1(U(17));  // OK, using copy constructor for U
    wu1.doStuff();
}
```

As it stands, the code above is well formed even though `W<U>::W()` would fail to compile if instantiated. Consequently, although providing an explicit-instantiation declaration for `W<U>` is valid, a corresponding explicit-instantiation definition for `W<U>` fails to compile, as would an implicit instantiation of `W<U>::W()`:

```
extern template class W<U>;  // Valid: Suppress implicit instantiation of W<U>.

template class W<U>;         // Error, U::U() not available for W<U>::W()

void useWU0()
{
    W<U> wu0;       // Error, U::U() not available for W<U>::W()
}
```

Unfortunately, the only workaround to achieve a comparable reduction in code bloat is to provide explicit-instantiation directives for each valid member function of `W<U>`, an approach that would likely carry a significantly greater maintenance burden:

```
extern template W<U>::W(const U& u);    // suppress individual member
extern template void W<U>::doStuff();  //     "          "        "
// ... Repeat for all other functions in W except W<U>::W().

template W<U>::W(const U& u);           // instantiate individual member
template void W<U>::doStuff();          //      "            "        "
// ... Repeat for all other functions in W except W<U>::W().
```

The power and flexibility to make it all work — albeit annoyingly — are there nonetheless.

See Also

- “Variable Templates” (§1.2, p. 157) covers an extension of the template syntax for defining a family of like-named variables or static data members that can be instantiated explicitly.

Further Reading

- For a different perspective on this feature, see **lakos20**, section 1.3.16, "`extern` Templates," pp. 183–185.
- For a more complete discussion of how compilers and linkers work with respect to C++, see **lakos20**, Chapter 1, "Compilers, Linkers, and Components," pp. 123–268.

Transparently Nested Namespaces

An **inline** namespace is a nested namespace whose member entities closely behave as if they were declared directly within the enclosing namespace.

Description

To a first approximation, an **inline namespace** (e.g., v2 in the code snippet below) acts a lot like a conventional nested namespace (e.g., v1) followed by a **using** directive for that namespace in its enclosing namespace[1]:

```
// example.cpp:
namespace n
{
    namespace v1  // conventional nested namespace followed by using directive
    {
        struct T { };     // nested type declaration (identified as ::n::v1::T)
        int d;            // ::n::v1::d at, e.g., 0x01a64e90
    }

    using namespace v1;   // Import names T and d into namespace n.
}

namespace n
{
    inline namespace v2   // similar to being followed by using namespace v2
    {
        struct T { };     // nested type declaration (identified as ::n::v2::T)
        int d;            // ::n::v2::d at, e.g., 0x01a64e94
    }

    // using namespace v2;  // redundant when used with an inline namespace
}
```

[1] C++17 allows developers to concisely declare nested namespaces with shorthand notation:

```
namespace a::b { /*...*/ }
// is the same as
namespace a { namespace b { /*...*/ } }
```

C++20 expands on the above syntax by allowing the insertion of the **inline** keyword in front of any of the namespaces, except the first one:

```
namespace a::inline b::inline c { /*...*/ }
// is the same as
namespace a { inline namespace b { inline namespace c { /*...*/ } } }

inline namespace a::b { }  // Error, cannot start with inline for compound namespace names
namespace inline a::b { }  // Error, inline at front of sequence explicitly disallowed
```

Four subtle details distinguish these approaches.

1. Name collisions with existing names behave differently due to differing name-lookup rules.
2. **Argument-dependent lookup (ADL)** gives special treatment to **inline** namespaces.
3. Template specializations can refer to the primary template in an **inline** namespace even if written in the enclosing namespace.
4. Reopening namespaces might reopen an **inline** namespace.

One important aspect that all forms of namespaces share, however, is that (1) nested symbolic names (e.g., `n::v1::T`) at the **API** level, (2) **mangled names** (e.g., `_ZN1n2v11dE`, `_ZN1n2v21dE`), and (3) assigned relocatable addresses (e.g., `0x01a64e90`, `0x01a64e94`) at the **ABI** level remain unaffected by the use of either **inline** or **using** or both. To be precise, source files containing, alternately, `namespace n { inline namespace v { int d; } }` and `namespace n { namespace v { int d; } using namespace v; }`, will produce identical assembly.[2] Note that a **using** directive immediately following an **inline** namespace is superfluous; name lookup will always consider names in **inline** namespaces before those imported by a **using** directive. Such a directive can, however, be used to import the contents of an **inline** namespace to some other namespace, albeit only in the conventional, **using directive** sense; see *Annoyances — Only one namespace can contain any given* ***inline*** *namespace* on page 1082.

More generally, each namespace has what is called its ***inline*** *namespace set*, which is the transitive closure of all **inline** namespaces within the namespace. All names in the **inline** namespace set are roughly intended to behave as if they are defined in the enclosing namespace. Conversely, each **inline** namespace has an *enclosing namespace set* that comprises all enclosing namespaces up to and including the first non**inline** namespace.

Loss of access to duplicate names in enclosing namespace

When both a type and a variable are declared with the same name in the same scope, the variable name hides the type name — such behavior can be demonstrated by using the form of **sizeof** that accepts a nonparenthesized *expression* (recall that the form of **sizeof** that accepts a *type* as its argument requires parentheses):

```
struct A { double d; };  static_assert(sizeof(  A) == 8, "");  // type
                      // static_assert(sizeof   A  == 8, "");  // Error

int A;                   static_assert(sizeof(  A) == 4, "");  // data
                         static_assert(sizeof   A  == 4, "");  // OK
```

[2]These mangled names can be seen with GCC by running `g++ -S <file>.cpp` and viewing the contents of the generated `<file>.s`. Note that Compiler Explorer is another valuable tool for learning about what comes out the other end of a C++ compiler: see https://godbolt.org/.

Unless both type and variable entities are declared within the same scope, no preference is given to variable names; the name of an entity in an inner scope hides a like-named entity in an enclosing scope:

```
void f()
{
    double B;               static_assert(sizeof(B) == 8, "");  // variable
    {                       static_assert(sizeof(B) == 8, "");  // variable
        struct B { int d; }; static_assert(sizeof(B) == 4, ""); // type
    }                       static_assert(sizeof(B) == 8, "");  // variable
}
```

When an entity is declared in an enclosing **namespace** and another entity having the same name hides it in a *lexically* nested scope, then (apart from **inline** namespaces) access to a hidden element can generally be recovered by using scope resolution:

```
struct C { double d; };  static_assert(sizeof(  C) == 8, "");

void g()
{                        static_assert(sizeof(  C) == 8, "");  // type
    int C;               static_assert(sizeof(  C) == 4, "");  // variable
                         static_assert(sizeof(::C) == 8, "");  // type
}                        static_assert(sizeof(  C) == 8, "");  // type
```

A conventional nested namespace behaves as one might expect:

```
namespace outer
{
    struct D { double d; }; static_assert(sizeof(       D) == 8, ""); // type

    namespace inner
    {                       static_assert(sizeof(       D) == 8, ""); // type
        int D;              static_assert(sizeof(       D) == 4, ""); // var
    }                       static_assert(sizeof(       D) == 8, ""); // type
                            static_assert(sizeof(inner::D) == 4, ""); // var
                            static_assert(sizeof(outer::D) == 8, ""); // type
    using namespace inner;//static_assert(sizeof(       D) == 0, ""); // Error
                            static_assert(sizeof(inner::D) == 4, ""); // var
                            static_assert(sizeof(outer::D) == 8, ""); // type
}                           static_assert(sizeof(outer::D) == 8, ""); // type
```

In the example above, the inner variable name, `D`, hides the outer type with the same name, starting from the point of `D`'s declaration in `inner` until `inner` is closed, after which the unqualified name `D` reverts to the type in the `outer` namespace. Then, right after the subsequent **`using namespace`** `inner;` directive, the meaning of the unqualified name `D` in `outer` becomes ambiguous, shown here with a **`static_assert`** that is commented out; any attempt to refer to an unqualified `D` from here to the end of the scope of `outer` will fail to compile. The type entity declared as `D` in the `outer` namespace can, however, still be accessed — from inside or outside of the `outer` namespace, as shown in the example — via its qualified name, `outer::D`.

If an **`inline`** namespace were used instead of a nested namespace followed by a **`using`** directive, however, the ability to recover by name the hidden entity in the enclosing namespace is lost. Unqualified name lookup considers the inline namespace set and the used namespace set simultaneously. Qualified name lookup first considers the **`inline`** namespace set and *then* goes on to look into used namespaces. These lookup rules mean we can still refer to `outer::D` in the example above, but doing so would still be ambiguous if `inner` were an inline namespace. This subtle difference in behavior is a byproduct of the highly specific use case that motivated this feature and for which it was explicitly designed; see *Use Cases — Link-safe ABI versioning* on page 1067.

Argument-dependent–lookup interoperability across `inline namespace` boundaries

Another important aspect of **`inline`** namespaces is that they allow ADL to work seamlessly across **`inline`** namespace boundaries. Whenever unqualified function names are being resolved, a list of *associated namespaces* is built for each argument of the function. This list of associated namespaces comprises the namespace of the argument, its enclosing namespace set, plus the **`inline`** namespace set.

Consider the case of a type, `U`, defined in an `outer` namespace, and a function, `f(U)`, declared in an `inner` namespace nested within `outer`. A second type, `V`, is defined in the `inner` namespace, and a function, `g`, is declared, after the close of `inner`, in the `outer` namespace:

```
namespace outer
{
    struct U { };

    // inline                // Uncommenting this line fixes the problem.
    namespace inner
    {
        void f(U) { }
        struct V { };
    }

    using namespace inner;  // If we inline inner, we don't need this line.

    void g(V) { }
}

void client()
{
    f(outer::U());          // Error, f is not declared in this scope.
    g(outer::inner::V());   // Error, g is not declared in this scope.
}
```

In the example above, a `client` invoking `f` with an object of type `outer::U` fails to compile because `f(outer::U)` is declared in the nested `inner` namespace, which is not the same as declaring it in `outer`. Because ADL does not look into namespaces added with the **`using`** directive, ADL does not find the needed `outer::inner::f` function. Similarly, the type `V`, defined in namespace `outer::inner`, is not declared in the same namespace as the function `g` that operates on it. Hence, when `g` is invoked from within `client` on an object of type `outer::inner::V`, ADL again does not find the needed function `outer::g(outer::V)`.

Simply making the `inner` namespace **`inline`** solves both of these ADL-related problems. All transitively nested **`inline`** namespaces — up to and including the most proximate non-**`inline`** enclosing namespace — are treated as one with respect to ADL.

The ability to specialize templates declared in a nested `inline` namespace

The third property that distinguishes **`inline`** namespaces from conventional ones, even when followed by a **`using`** directive, is the ability to specialize a class template defined within an **`inline`** namespace from within an enclosing one; this ability holds transitively up to and including the most proximate non**`inline`** namespace:

```
namespace out                   // proximate noninline outer namespace
{
    inline namespace in1        // first-level nested inline namespace
    {
        inline namespace in2    // second-level nested inline namespace
        {
            template <typename T> // primary class template general definition
            struct S { };

            template <>           // class template full specialization
            struct S<char> { };
        }

        template <>               // class template full specialization
        struct S<short> { };
    }

    template <>                   // class template full specialization
    struct S<int> { };
}

using namespace out;              // conventional using directive

template <>
struct S<int> { };                // Error, cannot specialize from this scope
```

Note that the conventional nested namespace `out` followed by a **`using`** directive in the enclosing namespace does not admit specialization from that outermost namespace, whereas all of the **inline** namespaces do. Function templates behave similarly except that — unlike class templates, whose definitions must reside entirely within the namespace in which they are declared — a function template can be *declared* within a nested namespace and then be *defined* from anywhere via a **qualified name**:

```
namespace out                       // proximate noninline outer namespace
{
    inline namespace in1            // first-level nested inline namespace
    {
        template <typename T>       // function template declaration
        void f();

        template <>                 // function template (full) specialization
        void f<short>() { }
    }

    template <>                     // function template (full) specialization
    void f<int>() { }
}

template <typename T>               // function template general definition
void out::in1::f() { }
```

An important takeaway from the examples above is that every template entity — be it class or function — *must* be declared in *exactly* one place within the collection of namespaces that comprise the **inline** namespace set. In particular, declaring a class template in a nested **inline** namespace and then subsequently defining it in a containing namespace is not possible because, unlike a function definition, a type definition cannot be placed into a namespace via name qualification alone:

```
namespace outer
{
    inline namespace inner
    {
        template <typename T>       // class template declaration
        struct Z;                   // (if defined, must be within same namespace)

        template <>                 // class template full specialization
        struct Z<float> { };
    }

    template <typename T>           // inconsistent declaration (and definition)
    struct Z { };                   // Z is now ambiguous in namespace outer.

    const int i = sizeof(Z<int>);  // Error, reference to Z is ambiguous.

    template <>                     // attempted class template full specialization
```

```
    struct Z<double> { };          // Error, outer::Z or outer::inner::Z?
}
```

Reopening namespaces can reopen nested inline ones

Another subtlety specific to **inline** namespaces is related to reopening namespaces. Consider a namespace `outer` that declares a nested namespace `outer::m` and an **inline** namespace `inner` that, in turn, declares a nested namespace `outer:inner::m`. In this case, subsequent attempts to reopen namespace `m` cause an ambiguity error:

```
namespace outer
{
    namespace m { }      // opens and closes ::outer::m

    inline namespace inner
    {
        namespace n { }  // opens and closes ::outer::inner::n
        namespace m { }  // opens and closes ::outer::inner::m
    }

    namespace n          // OK, reopens ::outer::inner::n
    {
        struct S { };    // defines ::outer::inner::n::S
    }

    namespace m          // Error, namespace m is ambiguous.
    {
        struct T { };    // with clang defines ::outer::m::T
    }
}

static_assert(std::is_same<outer::n::S, outer::inner::n::S>::value, "");
```

In the code snippet above, no issue occurs with reopening `outer::inner::n` and no issue would have occurred with reopening `outer::m` but for the `inner` namespaces having been declared **inline**. When a new namespace declaration is encountered, a lookup determines if a matching namespace having that name appears anywhere in the ***inline** namespace set* of the current namespace. If the namespace is ambiguous, as is the case with `m` in the example above, one can get the surprising error shown.[3] If a matching namespace is found

[3]Note that reopening already declared namespaces, such as `m` and `n` in the `inner` and `outer` example, is handled incorrectly on several popular platforms. Clang, for example, performs a name lookup when encountering a new namespace declaration and give preference to the outermost namespace found, causing the last declaration of `m` to reopen `::outer::m` instead of being ambiguous. GCC, prior to 8.1 (c. 2018), does not perform name lookup and will place *any* nested namespace declarations directly within their enclosing namespace. This defect causes the last declaration of `m` to reopen `::outer::m` instead of `::outer::inner::m` and the last declaration of `n` to open a new namespace, `::outer::n`, instead of reopening `::outer::inner::n`.

unambiguously inside an **inline** namespace, n in this case, then it is that nested namespace that is reopened — here, ::outer::inner::n. The inner namespace is reopened even though the last declaration of n is not lexically scoped within inner. Notice that the definition of S is perhaps surprisingly defining ::outer::inner::n::S, not ::outer::n::S. For more on what is *not* supported by this feature, see *Annoyances — Inability to redeclare across namespaces impedes code factoring* on page 1079.

Use Cases

Facilitating API migration

Getting a large codebase to *promptly* upgrade to a new version of a library in any sort of timely fashion can be challenging. As a simplistic illustration, imagine that we have just developed a new library, parselib, comprising a class template, Parser, and a function template, analyze, that takes a Parser object as its only argument:

```
namespace parselib
{
    template <typename T>
    class Parser
    {
        // ...

    public:
        Parser();
        int parse(T* result, const char* input);
            // Load result from null-terminated input; return 0 (on
            // success) or nonzero (with no effect on result).
    };

    template <typename T>
    double analyze(const Parser<T>& parser);
}
```

To use our library, clients will need to specialize our Parser class directly within the parselib namespace:

```
struct MyClass { /*...*/ };  // end-user-defined type

namespace parselib  // necessary to specialize Parser
{
    template <>             // Create full specialization of class
    class Parser<MyClass>  // Parser for user-type MyClass.
    {
        // ...
```

```
    public:
        Parser();
        int parse(MyClass* result, const char* input);
            // The contract for a specialization typically remains the same.
    };

    double analyze(const Parser<MyClass>& parser);
}
```

Typical `client` code will also look for the `Parser` class directly within the `parselib` namespace:

```
void client()
{
    MyClass result;
    parselib::Parser<MyClass> parser;

    int status = parser.parse(&result, "...( MyClass value )...");
    if (status != 0)
    {
        return;
    }

    double value = analyze(parser);
    // ...
}
```

Note that invoking `analyze` on objects of some instantiated type of the `Parser` class template will rely on ADL to find the corresponding overload.

We anticipate that our library's API will evolve over time, so we want to enhance the design of `parselib` accordingly. One of our goals is to somehow encourage clients to move essentially all at once, yet also to accommodate both the early adopters and the inevitable stragglers that make up a typical adoption curve. Our approach will be to create, within our outer `parselib` namespace, a nested **`inline`** namespace, `v1`, which will hold the current implementation of our library software:

```
namespace parselib
{
    inline namespace v1             // Note our use of inline namespace here.
    {
        template <typename T>
        class Parser
        {
            // ...
```

```
        public:
            Parser();
            int parse(T* result, const char* input);
                // Load result from null-terminated input; return 0 (on
                // success) or nonzero (with no effect on result).
        };

        template <typename T>
        double analyze(const Parser<T>& parser);
    }
}
```

As suggested by the name `v1`, this namespace serves primarily as a mechanism to support library evolution through API and ABI versioning (see *Link-safe ABI versioning* on page 1067 and *Build modes and ABI link safety* on page 1071). The need to specialize **`class`** `Parser` and, independently, the reliance on ADL to find the free function template `analyze` require the use of **`inline`** namespaces, as opposed to a conventional namespace followed by a **`using`** directive.

Note that, whenever a subsystem starts out directly in a first-level namespace and is subsequently moved to a second-level nested namespace for the purpose of versioning, declaring the inner namespace **`inline`** is the most reliable way to avoid inadvertently destabilizing existing clients; see also *Enabling selective **`using`** directives for short-named entities* on page 1074.

Now suppose we decide to enhance `parselib` in a non–backwards-compatible manner, such that the signature of `parse` takes a second argument `size` of type `std::size_t` to allow parsing of non–null-terminated strings and to reduce the risk of buffer overruns. Instead of unilaterally removing all support for the previous version in the new release, we can create a second namespace, `v2`, containing the new implementation and then, at some point, make `v2` the **`inline`** namespace instead of `v1`:

```
#include <cstddef>  // std::size_t

namespace parselib
{
    namespace v1  // Notice that v1 is now just a nested namespace.
    {
        template <typename T>
        class Parser
        {
            // ...

        public:
            Parser();
            int parse(T* result, const char* input);
```

```
                // Load result from null-terminated input; return 0 (on
                // success) or nonzero (with no effect on result).
        };

        template <typename T>
        double analyze(const Parser<T>& parser);
    }

    inline namespace v2    // Notice that use of inline keyword has moved here.
    {
        template <typename T>
        class Parser
        {
            // ...

        public:  // Note incompatible change to Parser's essential API.
            Parser();
            int parse(T* result, const char* input, std::size_t size);
                // Load result from input of specified size; return 0
                // on success) or nonzero (with no effect on result).
        };

        template <typename T>
        double analyze(const Parser<T>& parser);
    }
}
```

When we release this new version with `v2` made **`inline`**, all existing clients that rely on the version supported directly in `parselib` will, by design, break when they recompile. At that point, each client will have two options. The first one is to upgrade the code immediately by passing in the size of the input string (e.g., `23`) along with the address of its first character:

```
void client()
{
    // ...
    int status = parser.parse(&result, "...( MyClass value )...", 23);
    // ...                                                       ^^^^ Look here!
}
```

The second option is to change all references to `parselib` to refer to the original version in `v1` explicitly:

```
namespace parselib
{
    namespace v1  // specializations moved to nested namespace
    {
```

```
        template <>
        class Parser<MyClass>
        {
            // ...

        public:
            Parser();
            int parse(MyClass* result, const char* input);
        };

        double analyze(const Parser<MyClass>& parser);
    }
}

void client1()
{
    MyClass result;
    parselib::v1::Parser<MyClass> parser;  // reference nested namespace v1

    int status = parser.parse(&result, "...( MyClass value )...");
    if (status != 0)
    {
        return;
    }

    double value = analyze(parser);
    // ...
}
```

Providing the updated version in a new **`inline`** namespace `v2` provides a more flexible migration path — especially for a large population of independent client programs — compared to manual targeted changes in client code.

Although new users would pick up the latest version automatically either way, existing users of `parselib` will have the option of converting immediately by making a few small syntactic changes or opting to remain with the original version for a while longer by making all references to the library namespace refer explicitly to the desired version. If the library is released before the **`inline`** keyword is moved, early adopters will have the option of opting in by referring to `v2` explicitly until it becomes the default. Those who have no need for enhancements can achieve stability by referring to a particular version in perpetuity or until it is physically removed from the library source.

Although this same functionality can sometimes be realized without using **`inline`** namespaces (i.e., by adding a **`using namespace`** directive at the end of the `parselib` namespace), any benefit of ADL and the ability to specialize templates from within the enclosing `parselib` namespace itself would be lost. Note that, because specialization doesn't kick in until overload resolution is completed, specializing overloaded functions is dubious at best; see *Potential Pitfalls — Relying on **inline** namespaces to solve library evolution* on page 1077.

Providing separate namespaces for each successive version has an additional advantage in an entirely separate dimension: avoiding inadvertent, difficult-to-diagnose, latent linkage defects. Though not demonstrated by this specific example, cases do arise where simply changing which of the version namespaces is declared **inline** might lead to an **ill formed, no-diagnostic required (IFNDR)** program. This issue might ensue when one or more of its translation units that use the library are not recompiled before the program is relinked to the new static or dynamic library containing the updated version of the library software; see *Link-safe ABI versioning* below.

For distinct nested namespaces to guard effectively against accidental link-time errors, the symbols involved have to (1) reside in object code (e.g., a **header-only library** would fail this requirement) and (2) have the same **name mangling** (i.e., linker symbol) in both versions. In this particular instance, however, the signature of the `parse` member function of `parser` did change, and its mangled name will consequently change as well; hence the same `undefined symbol` link error would result either way.

Link-safe ABI versioning

inline namespaces are not intended as a mechanism for source-code versioning; instead, they prevent programs from being **ill formed** due to linking some version of a library with client code compiled using some other, typically older version of the same library. Below, we present two examples: a simple pedagogical example to illustrate the principle followed by a more real-world example. Suppose we have a library component `my_thing` that implements an example type, `Thing`, which wraps an **int** and initializes it with some value in its default constructor defined out-of-line in the `cpp` file:

```
struct Thing  // version 1 of class Thing
{
    int i;    // integer data member (size is 4)
    Thing();  // original noninline constructor (defined in .cpp file)
};
```

Compiling a source file with this version of the header included might produce an object file that can be incompatible yet linkable with an object file resulting from compiling a different source file with a different version of this header included:

```
struct Thing   // version 2 of class Thing
{
    double d;  // double-precision floating-point data member (size is 8)
    Thing();   // updated noninline constructor (defined in .cpp file)
};
```

To make the problem that we are illustrating concrete, let's represent the client as a `main` program that does nothing but create a `Thing` and print the value of its only data member, `i`.

```
// main.cpp:
#include <my_thing.h>  // my::Thing (version 1)
#include <iostream>    // std::cout

int main()
{
     my::Thing t;
     std::cout << t.i << '\n';
}
```

If we compile this program, a reference to a locally undefined linker symbol, such as `_ZN2my7impl_v15ThingC1Ev`,[4] which represents the `my::Thing::Thing` constructor, will be generated in the `main.o` file:

```
$ g++ -c main.cpp
```

Without explicit intervention, the spelling of this linker symbol would be unaffected by any subsequent changes made to the implementation of `my::Thing`, such as its data members or implementation of its default constructor, even after recompiling. The same, of course, applies to its definition in a separate translation unit.

We now turn to the translation unit implementing type `my::Thing`. The `my_thing` **component** consists of a `.h`/`.cpp` pair: `my_thing.h` and `my_thing.cpp`. The header file `my_thing.h` provides the physical interface, such as the definition of the principal type, `Thing`, its member and associated free function declarations, plus definitions for inline functions and function templates, if any:

```
// my_thing.h:
#ifndef INCLUDED_MY_THING
#define INCLUDED_MY_THING

namespace my                  // outer namespace (used directly by clients)
{
    inline namespace impl_v1  // inner namespace (for implementer use only)
    {
        struct Thing
        {
            int i;     // original data member, size = 4
            Thing();  // default constructor (defined in my_thing.cpp)
        };
    }
```

[4] On a Unix machine, typing `nm main.o` reveals the symbols used in the specified object file. A symbol prefaced with a capital `U` represents an undefined symbol that must be resolved by the linker. Note that the linker symbol shown here incorporates an intervening **inline** namespace, `impl_v1`, as will be explained shortly.

```
}

#endif
```

The implementation file `my_thing.cpp` contains all of the non**inline** function bodies that will be translated separately into the `my_thing.o` file:

```
// my_thing.cpp:
#include <my_thing.h>

namespace my                      // outer namespace (used directly by clients)
{
    inline namespace impl_v1      // inner namespace (for implementer use only)
    {
        Thing::Thing() : i(0)     // Load a 4-byte value into Thing's data member.
        {
        }
    }
}
```

Observing common good practice, we include the header file of the component as the first substantive line of code to ensure that — irrespective of anything else — the header always compiles in isolation, thereby avoiding insidious include-order dependencies.[5] When we compile the source file `my_thing.cpp`, we produce an object file `my_thing.o` containing the definition of the same linker symbol, such as `_ZN2my7impl_v15ThingC1Ev`, for the default constructor of `my::Thing` needed by the client:

```
$ g++ -c my_thing.cpp
```

We can then link `main.o` and `my_thing.o` into an executable and run it:

```
$ g++ -o prog main.o my_thing.o
$ ./prog

0
```

Now, suppose we were to change the definition of `my::Thing` to hold a **double** instead of an **int**, recompile `my_thing.cpp`, and then relink with the original `main.o` without recompiling `main.cpp` first. None of the relevant linker symbols would change, and the code would recompile and link just fine, but the resulting binary `prog` would be IFNDR: the client would be trying to print a 4-byte, **int** data member, `i`, in `main.o` that was loaded by the library component as an 8-byte, **double** into `d` in `my_thing.o`. We can resolve this problem by changing — or, if we didn't think of it in advance, by adding — a new **inline** namespace and making that change there:

[5] See **lakos20**, section 1.6.1, "Component Property 1," pp. 210–212.

```
// my_thing.cpp:
#include <my_thing.h>

namespace my                    // outer namespace (used directly by clients)
{
    inline namespace impl_v2    // inner namespace (for implementer use only)
    {
        Thing::Thing() : d(0.0)  // Load 8-byte value into Thing's data member.
        {
        }
    }
}
```

Now clients that attempt to link against the new library will not find the linker symbol, such as `_Z...impl_v1...v`, and the link stage will fail. Once clients recompile, however, the undefined linker symbol will match the one available in the new `my_thing.o`, such as `_Z...impl_v2...v`, the link stage will succeed, and the program will again work as expected. What's more, we have the option of keeping the original implementation. In that case, existing clients that have not as yet recompiled will continue to link against the old version until it is eventually removed after some suitable deprecation period.

As a more realistic second example of using **`inline`** namespaces to guard against linking incompatible versions, suppose we have two versions of a `Key` class in a security library in the enclosing namespace, `auth` — the original version in a regular nested namespace `v1`, and the new current version in an **`inline`** nested namespace `v2`:

```
#include <cstdint>  // std::uint32_t, std::unit64_t

namespace auth      // outer namespace (used directly by clients)
{
    namespace v1    // inner namespace (optionally used by clients)
    {
        class Key
        {
        private:
            std::uint32_t d_key;
                // sizeof(Key) is 4 bytes.

        public:
            std::uint32_t key() const;  // stable interface function

            // ...
        };
    }

    inline namespace v2    // inner namespace (default current version)
    {
        class Key
```

```
        {
        private:
            std::uint64_t d_securityHash;
            std::uint32_t d_key;
                // sizeof(Key) is 16 bytes.

        public:
            std::uint32_t key() const;  // stable interface function

            // ...
        };
    }
}
```

Attempting to link together older binary artifacts built against version 1 with binary artifacts built against version 2 will result in a link-time error rather than allowing an ill formed program to be created. Note, however, that this approach works only if functionality essential to typical use is defined out of line in a `.cpp` file. For example, it would add absolutely no value for libraries that are shipped entirely as header files, since the versioning offered here occurs strictly at the binary level (i.e., between object files) during the link stage.

Build modes and ABI link safety

In certain scenarios, a class might have two different memory layouts depending on compilation flags. For instance, consider a low-level `ManualBuffer` class template in which an additional data member is added for debugging purposes:

```
template <typename T>
struct ManualBuffer
{
private:
    alignas(T) char d_data[sizeof(T)];  // aligned and big enough to hold a T

#ifndef NDEBUG
    bool d_engaged;  // tracks whether buffer is full (debug builds only)
#endif

public:
    void construct(const T& obj);
        // Emplace obj. (Engage the buffer.) The behavior is undefined unless
        // the buffer was not previously engaged.

    void destroy();
        // Destroy the current obj. (Disengage the buffer.) The behavior is
        // undefined unless the buffer was previously engaged.

    // ...
};
```

Note that we have employed the C++11 **alignas** attribute (see Section 2.1.“**alignas**” on page 168) here because it is exactly what’s needed for this usage example.

The `d_engaged` flag in the example above serves as a way to detect misuse of the `ManualBuffer` class but only in debug builds. The extra space and run time required to maintain this Boolean flag is undesirable in a release build because `ManualBuffer` is intended to be an efficient, lightweight abstraction over the direct use of **placement new** and explicit destruction.

The linker symbol names generated for the methods of `ManualBuffer` are the same irrespective of the chosen build mode. If the same program links together two object files where `ManualBuffer` is used — one built in debug mode and one built in release mode — the **one-definition rule (ODR)** will be violated, and the program will again be IFNDR.

Prior to **inline** namespaces, it was possible to control the ABI-level name of linked symbols by creating separate template instantiations on a per-build-mode basis:

```
#ifndef NDEBUG
enum { is_debug_build = 1 };
#else
enum { is_debug_build = 0 };
#endif

template <typename T, bool Debug = is_debug_build>
struct ManualBuffer { /*...*/ };
```

While the code above changes the interface of `ManualBuffer` to accept an additional template parameter, it also allows debug and release versions of the same class to coexist in the same program, which might prove useful, e.g., for testing.

Another way of avoiding incompatibilities at link time is to introduce two **inline** namespaces, the entire purpose of which is to change the ABI-level names of the linker symbols associated with `ManualBuffer` depending on the build mode:

```
#ifndef NDEBUG            // perhaps a BAD IDEA
inline namespace release
#else
inline namespace debug
#endif
{
    template <typename T>
    struct ManualBuffer
    {
        // ... (same as above)
    };
}
```

The approach demonstrated in this example tries to ensure that a linker error will occur if any attempt is made to link objects built with a build mode different from that of `manualbuffer.o`. Tying it to the `NDEBUG` flag, however, might have unintended consequences; we might introduce unwanted restrictions in what we call **mixed-mode builds**. Most modern platforms support the notion of linking a collection of object files irrespective of their optimization levels. The same is certainly true for whether or not C-style `assert` is enabled. In other words, we might want to have a mixed-mode build where we link object files that differ in their optimization and assertion options, as long as they are binary compatible — i.e., in this case, they all must be uniform with respect to the implementation of `ManualBuffer`. Hence, a more general, albeit more complicated and manual, approach would be to tie the noninteroperable behavior associated with this "safe" or "defensive" build mode to a different switch entirely. Another consideration would be to avoid ever inlining a namespace into the global namespace since no method is available to recover a symbol when there is a collision:

```
namespace buflib  // GOOD IDEA: enclosing namespace for nested inline namespace
{
#ifdef SAFE_MODE  // GOOD IDEA: separate control of non-interoperable versions
    inline namespace safe_build_mode
#else
    inline namespace normal_build_mode
#endif
    {
        template <typename T>
        struct ManualBuffer
        {
        private:
            alignas(T) char d_data[sizeof(T)];  // aligned/sized to hold a T

#ifdef SAFE_MODE
            bool d_engaged;  // tracks whether buffer is full (safe mode only)
#endif

        public:
            void construct(const T& obj);  // sets d_engaged (safe mode only)
            void destroy();                // sets d_engaged (safe mode only)
            // ...
        };
    }
}
```

And, of course, the appropriate conditional compilation within the function bodies would need to be in the corresponding `.cpp` file.

Finally, if we have two implementations of a particular entity that are sufficiently distinct, we might choose to represent them in their entirety, controlled by their own bespoke conditional-compilation switches, as illustrated here using the `my::VersionedThing` type (see *Link-safe ABI versioning* on page 1067):

```
// my_versionedthing.h:
#ifndef INCLUDED_MY_VERSIONEDTHING
#define INCLUDED_MY_VERSIONEDTHING

namespace my
{
#ifdef MY_THING_VERSION_1  // bespoke switch for this component version
    inline
#endif
    namespace v1
    {
        struct VersionedThing
        {
            int d_i;
            VersionedThing();
        };
    }

#ifdef MY_THING_VERSION_2  // bespoke switch for this component version
    inline
#endif
    namespace v2
    {
        struct VersionedThing
        {
            double d_i;
            VersionedThing();
        };
    }
}
#endif
```

However, see *Potential Pitfalls—**`inline`**-namespace–based versioning doesn't scale* on page 1076.

Enabling selective `using` directives for short-named entities

Introducing a large number of small names into client code that doesn't follow rigorous nomenclature can be problematic. Hoisting these names into one or more nested namespaces so that they are easier to identify as a unit and can be used more selectively by clients, such as through explicit qualification or using directives, can sometimes be an effective way of organizing shared codebases. For example, `std::literals` and its nested namespaces, such as `chrono_literals`, were introduced as **`inline`** namespaces in C++14. As it turns out, clients of these nested namespaces have no need to specialize any templates defined in these namespaces nor do they define types that must be found through ADL, but one can at least imagine special circumstances in which such tiny-named entities are either templates that require specialization or operator-like functions, such as `swap`, defined for local types within those nested namespaces. In those cases, **inline** namespaces would be required to preserve the desired "as if" properties.

Even without either of these two needs, another property of an **inline** namespace differentiates it from a non**inline** one followed by a **using** directive. Recall from *Description — Loss of access to duplicate names in enclosing namespace* on page 1056 that a name in an outer namespace will hide a duplicate name imported via a **using** directive, whereas any access to that duplicate name within the enclosing namespace would be ambiguous when that symbol is installed by way of an **inline** namespace. To see why this more forceful clobbering behavior might be preferred over hiding, suppose we have a communal namespace `abc` that is shared across multiple disparate headers. The first header, `abc_header1.h`, represents a collection of logically related small functions declared directly in `abc`:

```
// abc_header1.h:
namespace abc
{
    int i();
    int am();
    int smart();
}
```

A second header, `abc_header2.h`, creates a suite of many functions having tiny function names. In a perhaps misguided effort to avoid clobbering other symbols within the `abc` namespace having the same name, all of these tiny functions are sequestered within a `nested` namespace:

```
// abc_header2.h:
namespace abc
{
    namespace nested  // Should this namespace have been inline instead?
    {
        int a();  // lots of functions with tiny names
        int b();
        int c();
        // ...
        int h();
        int i();  // might collide with another name declared in abc
        // ...
        int z();
    }

    using namespace nested;  // becomes superfluous if nested is made inline
}
```

Now suppose that a client application includes both of these headers to accomplish some task:

```
// client.cpp:
#include <abc_header1.h>
#include <abc_header2.h>

int function()
{
    if (abc::smart() < 0) { return -1; }  // uses smart() from abc_header1.h
    return abc::z() + abc::i() + abc::a() + abc::h() + abc::c();  // Oops!
        // Bug, silently uses the abc::i() defined in abc_header1.h
}
```

In trying to cede control to the client as to whether the declared or imported `abc::i()` function is to be used, we have, in effect, invited the defect illustrated in the above example whereby the client was expecting the `abc::i()` from `abc_header2.h` and yet picked up the one from `abc_header1.h` by default. Had the `nested` namespace in `abc_header2.h` been declared **`inline`**, the qualified name `abc::i()` would have automatically been rendered *ambiguous* in namespace `abc`, the translation would have failed *safely*, and the defect would have been exposed at compile time. The downside, however, is that no method would be available to recover nominal access to the `abc::i()` defined in `abc_header1.h` once `abc_header2.h` is included, even though the two functions (e.g., including their mangled names at the ABI level) remain distinct.

Potential Pitfalls

inline-namespace–based versioning doesn't scale

The problem with using **`inline`** namespaces for ABI link safety is that the protection they offer is only partial; in a few major places, critical problems can linger until run time instead of being caught at compile time.

Controlling which namespace is **`inline`** using macros, such as was done in the `my::VersionedThing` example in *Use Cases — Link-safe ABI versioning* on page 1067, will result in code that directly uses the unversioned name, `my::VersionedThing` being bound directly to the versioned name `my::v1::VersionedThing` or `my::v2::VersionedThing`, along with the class layout of that particular entity. Sometimes details of using the **`inline`** namespace member are not resolved by the linker, such as the object layout when we use types from that namespace as member variables in other objects:

```
// my_thingaggregate.h:

// ...
#include <my_versionedthing.h>
// ...

namespace my
{
```

```
    struct ThingAggregate
    {
        // ...
        VersionedThing d_thing;
        // ...
    };
}
```

This new `ThingAggregate` type does not have the versioned **`inline`** namespace as part of its mangled name; it does, however, have a completely different layout if built with `MY_THING_VERSION_1` defined versus `MY_THING_VERSION_2` defined. Linking a program with mixed versions of these flags will result in runtime failures that are decidedly difficult to diagnose.

This same sort of problem will arise for functions taking arguments of such types; calling a function from code that is wrong about the layout of a particular type will result in stack corruption and other undefined and unpredictable behavior. This macro-induced problem will also arise in cases where an old object file is linked against new code that changes which namespace is **`inline`**d but still provides the definitions for the old version namespace. The old object file for the client can still link, but new object files using the headers for the old objects might attempt to manipulate those objects using the new namespace.

The only viable workaround for this approach is to propagate the **`inline`** namespace hierarchy through the entire software stack. Every object or function that uses `my::VersionedThing` needs to also be in a namespace that differs based on the same control macro. In the case of `ThingAggregate`, one could just use the same `my::v1` and `my::v2` namespaces, but higher-level libraries would need their own `my`-specific nested namespaces. Even worse, for higher-level libraries, every lower-level library having a versioning scheme of this nature would need to be considered, resulting in having to provide the full cross-product of nested namespaces to get link-time protection against mixed-mode builds.

This need for layers above a library to be aware of and to integrate into their own structure the same namespaces the library has removes all or most of the benefits of using **`inline`** namespaces for versioning. For an authentic real-world case study of heroic industrial use — and eventual disuse — of **`inline`**-namespaces for versioning, see *Appendix — Case study of using **`inline`** namespaces for versioning* on page 1083.

Relying on `inline` namespaces to solve library evolution

Inline namespaces might be misperceived as a complete solution for the owner of a library to evolve its API. As an especially relevant example, consider the C++ Standard Library, which itself does not use inline namespaces for versioning. Instead, to allow for its anticipated essential evolution, the Standard Library imposes certain special restrictions on what is permitted to occur within its own `std` namespace by dint of deeming certain problematic uses as either ill formed or otherwise engendering **undefined behavior**.

Since C++11, several restrictions related to the Standard Library were put in place.

- Users may not add any new declarations within namespace `std`, meaning that users cannot add new *functions*, *overloads*, *types*, or *templates* to `std`. This restriction gives the Standard Library freedom to add new *names* in future versions of the Standard.
- Users may not specialize member functions, member function templates, or member class templates. Specializing any of those entities might significantly inhibit a Standard Library vendor's ability to maintain its otherwise encapsulated implementation details.
- Users may add specializations of top-level Standard Library templates only if the declaration depends on the name of a nonstandard user-defined type and only if that user-defined type meets all requirements of the original template. Specialization of function templates is allowed but generally discouraged because this practice doesn't scale since function templates cannot be partially specialized. Specializing of standard class templates when the specialization names a nonstandard user-defined type, such as `std::vector<MyType*>`, is allowed but also problematic when not explicitly supported. While certain specific types, such as `std::hash`, are designed for user specialization, steering clear of the practice for any other type helps to avoid surprises.

Several other good practices facilitate smooth evolution for the Standard Library.[6]

- Avoid specializing variable templates, even if dependent on user-defined types, except for those variable templates where specialization is explicitly allowed.[7]
- Other than a few specific exceptions, avoiding the forming of pointers to Standard Library functions — either explicitly or implicitly — allows the library to add overloads, either as part of the Standard or as an implementation detail for a particular Standard Library, without breaking user code.[8]
- Overloads of Standard Library functions that depend on user-defined types are permitted, but, as with specializing Standard Library templates, users must still meet the requirements of the Standard Library function. Some functions, such as `std::swap`, are designed to be customization points via overloading, but leaving functions not specifically designed for this purpose to vendor implementations only helps to avoid surprises.

Finally, upon reading about this **`inline`** namespace feature, one might think that all names in namespace `std` could be made available at a global scope simply by inserting an

[6]These restrictions are normative in C++20, having finally formalized what were long identified as best practices. Though these restrictions might not be codified in the Standard for pre-C++20 software, they have been recognized best practices for as long as the Standard Library has existed and adherence to them will materially improve the ability of software to migrate to future language standards irrespective of what version of the language standard is being targeted.

[7]C++20 limits the specialization of variable templates to only those instances where specialization is explicitly allowed and does so only for the mathematical constants in `<numbers>`.

[8]C++20 identifies these functions as `addressable` and gives that property to only `iostream` manipulators since those are the only functions in the Standard Library for which taking their address is part of normal usage.

inline namespace std {} before including any standard headers. This practice is, however, explicitly called out as ill-formed within the C++11 Standard. Although not uniformly diagnosed as an error by all compilers, attempting this forbidden practice is apt to lead to surprising problems even if not diagnosed as an error immediately.

Inconsistent use of inline keyword is ill formed, no diagnostic required

It is an ODR violation, IFNDR, for a nested namespace to be **inline** in one translation unit and non**inline** in another. And yet, the motivating use case of this feature relies on the linker to actively complain whenever different, incompatible versions — nested within different, possibly **inline**-inconsistent, namespaces of an ABI — are used within a single executable. Because declaring a nested namespace **inline** does not, by design, affect linker-level symbols, developers must take appropriate care, such as effective use of header files, to defend against such preventable inconsistencies.

Annoyances

Inability to redeclare across namespaces impedes code factoring

An essential feature of an **inline** namespace is the ability to declare a template within a nested **inline** namespace and then specialize it within its enclosing namespace. For example, we can declare

- a type template, `S0`
- a couple of function templates, `f0` and `g0`
- and a member function template `h0`, which is similar to `f0`

in an **inline** namespace, `inner`, and specialize each of them, such as for **int**, in the enclosing namespace, `outer`:

```
namespace outer                                         // enclosing namespace
{
    inline namespace inner                              // nested namespace
    {
        template<typename T> struct S0;                 // declarations of
        template<typename T> void f0();                 // various class
        template<typename T> void g0(T v);              // and function
        struct A0 { template <typename T> void h0(); }; // templates
    }

    template<> struct S0<int> { };                      // specializations
    template<> void f0<int>() { }                       // of the various
    void g0(int) { }  /* overload not specialization */ // class and function
    template<> void A0::h0<int>() { }                   // declarations above
}                                                       // in outer namespace
```

Note that, in the case of `g0` in this example, the "specialization" **`void`** `g0(`**`int`**`)` is a non-template *overload* of the function template `g0` rather than a specialization of it. We *cannot*, however, portably[9] declare these templates within the `outer` namespace and then specialize them within the `inner` one, even though the `inner` namespace is **`inline`**:

```
namespace outer                                  // enclosing namespace
{
    template<typename T> struct S1;              // class template
    template<typename T> void f1();              // function template
    template<typename T> void g1(T v);           // function template

    struct A1 { template <typename T> void h1(); }; // member function template

    inline namespace inner                       // nested namespace
    {                                            // BAD IDEA
        template<> struct S1<int> { };           // Error, S1 not a template
        template<> void f1<int>() { }            // Error, f1 not a template
        void g1(int) { }                         // OK, overloaded function
        template<> void A1::h1<int>() { }        // Error, h1 not a template
    }
}
```

Attempting to declare a template in the `outer` namespace and then define, effectively redeclaring, it in an **`inline`** inner one causes the name to be inaccessible within the `outer` namespace:

```
namespace outer                                       // enclosing namespace
{                                                     // BAD IDEA
    template<typename T> struct S2;                   // declarations of
    template<typename T> void f2();                   // various class and
    template<typename T> void g2(T v);                // function templates

    inline namespace inner                            // nested namespace
    {
        template<typename T> struct S2 { };           // definitions of
        template<typename T> void f2() { }            // unrelated class and
        template<typename T> void g2(T v) { }         // function templates
    }

    template<> struct S2<int> { };     // Error, S2 is ambiguous in outer.
    template<> void f2<int>() { }      // Error, f2 is ambiguous in outer.
    void g2(int) { }                   // OK, g2 is an overload definition.
}
```

[9]GCC provides the `-fpermissive` flag, which allows the example containing specializations within the inner namespace to compile with warnings. Note again that `g1(`**`int`**`)`, being an *overload* and not a *specialization*, wasn't an error and, therefore, isn't a warning either.

Finally, declaring a template in the nested **inline** namespace `inner` in the example above and then subsequently defining it in the enclosing `outer` namespace has the same effect of making declared symbols ambiguous in the `outer` namespace:

```
namespace outer                                             // enclosing namespace
{                                                           // BAD IDEA
    inline namespace inner                                  // nested namespace
    {
        template<typename T> struct S3;                     // declarations of
        template<typename T> void f3();                     // various class
        template<typename T> void g3(T v);                  // and function
        struct A3 { template <typename T> void h3(); };     // templates
    }

    template<typename T> struct S3 { };                     // definitions of
    template<typename T> void f3() { }                      // unrelated class
    template<typename T> void g3(T v) { }                   // and function
    template<typename T> void A3::h3() { }                  // templates

    template<> struct S3<int> { };     // Error, S3 is ambiguous in outer.
    template<> void f3<int>() { }      // Error, f3 is ambiguous in outer.
    void g3(int) { }                   // OK, g3 is an overload definition.
    template<> void A3::h3<int>() { }  // Error, h2 is ambiguous in outer.
}
```

Note that, although the definition for a member function template must be located directly within the namespace in which it is declared, a class or function template, once declared, may instead be defined in a different scope by using an appropriate name qualification:

```
template <typename T> struct outer::S3 { };         // OK, enclosing namespace
template <typename T> void outer::inner::f3() { }   // OK, nested namespace
template <typename T> void outer::g3(T v) { }       // OK, enclosing namespace
template <typename T> void outer::A3::h3<T>() { }   // Error, ill-formed

namespace outer
{
    inline namespace inner
    {
        template <typename T> void A3::h3() { }     // OK, within same namespace
    }
}
```

Also note that, as ever, the corresponding definition of the declared template must have been seen before it can be used in a context requiring a complete type. The importance of ensuring that all specializations of a template have been seen before it is used substantively (i.e., **ODR-used**) cannot be overstated, giving rise to the only limerick, which is actually part of the normative text, in the C++ Language Standard[10]:

[10]See **iso11a**, section 14.7.3, “Explicit specialization,” paragraph 7, pp. 375–376, specifically p. 376.

When writing a specialization,
be careful about its location;
or to make it compile
will be such a trial
as to kindle its self-immolation.

Only one namespace can contain any given inline namespace

Unlike conventional **using** directives, which can be used to generate arbitrary many-to-many relationships between different namespaces, **inline** namespaces can be used only to contribute names to the sequence of enclosing namespaces up to the first non**inline** one. In cases in which the names from a namespace are desired in multiple other namespaces, the classical **using** directive must be used, with the subtle differences between the two modes properly addressed.

As an example, the C++14 Standard Library provides a hierarchy of nested **inline** namespaces for literals of different sorts within namespace `std`.

- `std::literals::complex_literals`
- `std::literals::chrono_literals`
- `std::literals::string_literals`
- `std::literals::string_view_literals`

These namespaces can be imported to a local scope in one shot via a **using** `std::literals` or instead, more selectively, by **using** the nested namespaces directly. This separation of the types used with user-defined literals, which are all in namespace `std`, from the user-defined literals that can be used to create those types led to some frustration; those who had a **using namespace** `std;` could reasonably have expected to get the user-defined literals associated with their `std` types. However, the types in the nested namespace `std::chrono` did *not* meet this expectation.[11]

Eventually *both* solutions for incorporating literal namespaces, **inline** from `std::literals` and non**inline** from `std::chrono`, were pressed into service when, in C++17, a **using namespace** `literals::chrono_literals;` was added to the `std::chrono` namespace. The Standard does not, however, benefit in any objective way from any of these namespaces being **inline** since the artifacts in the `literals` namespace neither depend on ADL nor are templates in need of user-defined specializations; hence, having all non**inline** namespaces with appropriate **using** declarations would have been functionally indistinguishable from the bifurcated approach taken.

[11] CWG issue 2278; **hinnant17**

See Also

- "**alignas**" (§2.1, p. 168) provides properly aligned storage for an object of arbitrary type `T` in the example in *Use Cases — Build modes and ABI link safety* on page 1071.

Further Reading

- **sutter14a** uses inline namespaces as part of a proposal for a portable ABI across compilers.

- **lopez-gomez20** uses inline namespaces as part of a solution to avoid ODR violation in an interpreter.

Appendix

Case study of using `inline` namespaces for versioning

By Niall Douglas

Let me tell you what I (don't) use them for. It is not a conventional opinion.

At a previous well-regarded company, they were shipping no less than forty-three copies of Boost in their application. Boost was not on the approved libraries list, but the great thing about header-only libraries is that they don't obviously appear in final binaries, unless you look for them. So each individual team was including bits of Boost quietly and without telling their legal department. Why? Because it saved time. (This was C++98, and `boost::shared_ptr` and `boost::function` are both extremely attractive facilities.)

Here's the really interesting part: Most of these copies of Boost were not the same version. They were varying over a five-year release period. And, unfortunately, Boost makes no API or ABI guarantees. So, theoretically, you could get two different incompatible versions of Boost appearing in the same program binary, and BOOM! there goes memory corruption.

I advocated to Boost that a simple solution would be for Boost to wrap up their implementation into an internal inline namespace. That inline namespace ought to mean something.

- `lib::v1` is the *stable*, version-1 ABI, which is guaranteed to be compatible with all past and future `lib::v1` ABIs, forever, as determined by the ABI-compliance-check tool that runs on CI. The same goes for `v2`, `v3`, and so on.

- `lib::v2_a7fe42d` is the *unstable*, version-2 ABI, which may be incompatible with any other `lib::* ABI`; hence, the seven hex chars after the underscore are the git short SHA, permuted by every commit to the git repository but, in practice, per CMake configure, because nobody wants to rebuild everything per commit. This ensures that no symbols from any revision of `lib` will *ever* silently collide or otherwise interfere with any other revision of `lib`, when combined into a single binary by a dumb linker.

I have been steadily making progress on getting Boost to avoid putting anything in the global namespace, so a straightforward find-and-replace can let you "fix" on a particular version of Boost.

That's all the same as the pitch for **`inline`** namespaces. You'll see the same technique used in `libstdc++` and many other major modern C++ codebases.

But I'll tell you now, I don't use **`inline`** namespaces anymore. Now what I do is use a macro defined to a uniquely named namespace. My build system uses the git SHA to synthesize namespace macros for my namespace name, beginning the namespace and ending the namespace. Finally, in the documentation, I teach people to always use a namespace alias to a macro to denote the namespace:

```
namespace output = OUTCOME_V2_NAMESPACE;
```

That macro expands to something like `::outcome_v2_ee9abc2`; that is, I don't use **`inline`** namespaces anymore.

Why?

Well, for *existing* libraries that don't want to break backward source compatibility, I think **`inline`** namespaces serve a need. For *new* libraries, I think a macro-defined namespace is clearer.

- It causes users to publicly commit to "I know what you're doing here, what it means, and what its consequences are."
- It declares to *other* users that something unusual (i.e., go read the documentation) is happening here, instead of silent magic behind the scenes.
- It prevents accidents that interfere with ADL and other customization points, which induce surprise, such as accidentally injecting a customization point into `lib`, not into `lib::v2`.
- Using macros to denote namespace lets us reuse the preprocessor machinery to generate C++ modules using the exact same codebase; C++ modules are used if the compiler supports them, else we fall back to inclusion.

Finally, and here's the real rub, because we now have namespace aliases, if I were tempted to use an **`inline`** namespace, nowadays I probably would instead use a uniquely named namespace instead, and, in the `include` file, I'd alias a user-friendly name to that uniquely named namespace. I think that approach is less likely to induce surprise in the typical developer's likely use cases than **`inline`** namespaces, such as injecting customization points into the wrong namespace.

So now I hope you've got a good handle on **`inline`** namespaces: I was once keen on them, but after some years of experience, I've gone off them in favor of better-in-my-opinion alternatives. Unfortunately, if your type `x::S` has members of type `a::T` and macros decide if that is `a::v1::T` or `a::v2::T`, then no linker protects the higher-level types from ODR bugs, unless you also version `x`.

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

O

Index

P

Index

Index

Index

Index

Index

Index

Index

Index

Index