Embracing Modern C++ Safely

This is simply a placeholder. Your production team will replace this page with the real series page.

Embracing Modern C++ Safely

John Lakos

Vittorio Romeo

Addison-Wesley

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

Library of Congress Cataloging-in-Publication Data

LIBRARY OF CONGRESS CIP DATA WILL GO HERE; MUST BE ALIGNED AS INDICATED BY LOC

ISBN-13: NUMBER HERE
ISBN-10: NUMBER HERE
Text printed in the United States on recycled paper at PRINTER INFO HERE.
First printing, MONTH YEAR

The lines of text of the dedication will be broken
with the help of the production team
to ensure an attractive
layout.

Author Initials

Should there be a second author, that author's lines of
dedication text will go here and will also be broken
in whatever way is attractive and logical.

Author Initials

Contents

Contents

Foreword

The text of the foreword will go here.

Preface

The text of the preface will go here. All the elements and commands offered within the text will work here as well.

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?

At vero eos et accusamus et iusto odio dignissimos ducimus qui blanditiis praesentium voluptatum deleniti atque corrupti quos dolores et quas molestias excepturi sint occaecati cupiditate non provident, similique sunt in culpa qui officia deserunt mollitia animi, id est laborum et dolorum fuga. Et harum quidem rerum facilis est et expedita distinctio. Nam libero tempore, cum soluta nobis est eligendi optio cumque nihil impedit quo minus id quod maxime placeat facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum hic tenetur a sapiente delectus, ut aut reiciendis voluptatibus maiores alias consequatur aut perferendis doloribus asperiores repellat.

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt.Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?

At vero eos et accusamus et iusto odio dignissimos ducimus qui blanditiis praesentium voluptatum deleniti atque corrupti quos dolores et quas molestias excepturi sint occaecati cupiditate non provident, similique sunt in culpa qui officia deserunt mollitia animi, id est laborum et dolorum fuga. Et harum quidem rerum facilis est et expedita distinctio. Nam

libero tempore, cum soluta nobis est eligendi optio cumque nihil impedit quo minus id quod maxime placeat facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum hic tenetur a sapiente delectus, ut aut reiciendis voluptatibus maiores alias consequatur aut perferendis doloribus asperiores repellat.

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt.Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?

At vero eos et accusamus et iusto odio dignissimos ducimus qui blanditiis praesentium voluptatum deleniti atque corrupti quos dolores et quas molestias excepturi sint occaecati cupiditate non provident, similique sunt in culpa qui officia deserunt mollitia animi, id est laborum et dolorum fuga. Et harum quidem rerum facilis est et expedita distinctio. Nam libero tempore, cum soluta nobis est eligendi optio cumque nihil impedit quo minus id quod maxime placeat facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum hic tenetur a sapiente delectus, ut aut reiciendis voluptatibus maiores alias consequatur aut perferendis doloribus asperiores repellat.

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?

At vero eos et accusamus et iusto odio dignissimos ducimus qui blanditiis praesentium voluptatum deleniti atque corrupti quos dolores et quas molestias excepturi sint occaecati cupiditate non provident, similique sunt in culpa qui officia deserunt mollitia animi, id est laborum et dolorum fuga. Et harum quidem rerum facilis est et expedita distinctio. Nam libero tempore, cum soluta nobis est eligendi optio cumque nihil impedit quo minus id quod maxime placeat facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum hic

tenetur a sapiente delectus, ut aut reiciendis voluptatibus maiores alias consequatur aut perferendis doloribus asperiores repellat.

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?

At vero eos et accusamus et iusto odio dignissimos ducimus qui blanditiis praesentium voluptatum deleniti atque corrupti quos dolores et quas molestias excepturi sint occaecati cupiditate non provident, similique sunt in culpa qui officia deserunt mollitia animi, id est laborum et dolorum fuga. Et harum quidem rerum facilis est et expedita distinctio. Nam libero tempore, cum soluta nobis est eligendi optio cumque nihil impedit quo minus id quod maxime placeat facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum hic tenetur a sapiente delectus, ut aut reiciendis voluptatibus maiores alias consequatur aut perferendis doloribus asperiores repellat.

Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?

At vero eos et accusamus et iusto odio dignissimos ducimus qui blanditiis praesentium voluptatum deleniti atque corrupti quos dolores et quas molestias excepturi sint occaecati cupiditate non provident, similique sunt in culpa qui officia deserunt mollitia animi, id est laborum et dolorum fuga. Et harum quidem rerum facilis est et expedita distinctio. Nam libero tempore, cum soluta nobis est eligendi optio cumque nihil impedit quo minus id quod maxime placeat facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et aut officiis debitis aut rerum necessitatibus saepe eveniet ut et voluptates repudiandae sint et molestiae non recusandae. Itaque earum rerum hic tenetur a sapiente delectus, ut aut reiciendis voluptatibus maiores alias consequatur aut perferendis doloribus asperiores repellat.

Acknowledgments

The text of the author's acknowledgements will go here.

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, 2019), 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, working on mission-critical C++ middleware and delivering 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.

Chapter 0

Introduction

Welcome! *Embracing Modern C++ Safely* is a *reference book* dedicated to professionals who want to leverage modern C++ features in the development and maintenance of large-scale, complex C++ software systems.

This book deliberately concentrates on the productive value afforded by each new language feature added by C++ starting with C++11, particularly when the systems and organizations involved are considered at scale. We left aside ideas and idioms, however clever and intellectually intriguing, that could hurt the bottom line when applied at large. Instead, we focus on what is objectively true and relevant to making wise economic and design decisions, with an understanding of the inevitable tradeoffs 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 by dedicating *Embracing Modern C++ Safely* to only the features of Modern C++ that have been part of the Standard for at least five years, which grants enough perspective to properly evaluate its practical impact. Thus, we are able to provide you with a thorough and comprehensive treatment based on practical experience and worthy of your limited professional development time. If you're out there looking for tried and true ways to better use modern C++ features for improving your productivity, 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* is not a tutorial on programming, on C++, 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 within your and your team's toolbox.

What Makes This Book Different

The book you're now reading aims very strongly at being objective, empirical, and practical. We simply present features, their applicability, and their potential pitfalls as reflected by the analysis of millions of human-hours of using C++11 and C++14 in the development of varied large-scale software systems; personal preference matters have been neutralized to

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

our, and our reviewers', best ability. We wrote down the distilled truth that remains, which should shape your understanding of what modern C++ has to offer to you 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. Hence, this book is, by design, not a C++ style or coding-standards guide; it would, however, provide valuable input to any development organization seeking to author or enhance one.

Practicality is a topic very important to us, too, and in a very 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 having that feature employed routinely in the ecosystem of a software development organization. (We believe that costs of using language features are sadly neglected by most texts.) In other words, we weigh the benefits of successfully using a feature against the risk 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 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 in terms of safety of adoption — namely *safe*, *conditionally safe*, or *unsafe* features.

We are not aware of any similar text amid the rich offering of C++ textbooks; in a very real sense, we wrote it 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. However, to be effective, this book must remain small, concise, and focused on what expert C++ developers need to know well to be successful right now.

In this first of an anticipated series of periodically extended volumes, we characterize, dissect, and elucidate most of the modern language features introduced into the C++ Standard starting with C++11. We chose to limit the scope of this first edition to only those features that have been in the language Standard and widely available in practice for at least five years. This limited focus enables us to more fully 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 confined 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 code base, today.

Over time, we expect, and hope, that practicing senior developers will emerge entirely from the postmodern C++ era. By then, a book that focuses on all of the important features of modern C++ would naturally include many of those that were around before C++11. With that horizon in mind, 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* White Paper

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, either derived from standards documents or from extensive practical experience; we explicitly avoid subjective opinion such as our evaluation of the relative merits of design tradeoffs (restraint that admittedly 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, its chapter classification — indicating the relative safety of its quotidian use in a large software-development environment — reflects our combined accumulated experience totaling decades of real-world, hands-on experience with developing a variety of large-scale C++ software systems.

Elucidation (Not Prescription)

We deliberately avoid prescribing any cut-and-dried 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.

[2] **meyers05**

Brevity (Not Verbosity)

Embracing Modern C++ Safely is neither designed nor intended to be an introduction to modern C++. It is a handy reference for experienced C++ programmers who may have a passing knowledge of the recently added C++ features and a desire to perfect their understanding. Our writing style is intentionally tight, with the goal of providing you with facts, concise objective analysis, and cogent, real-world examples. By doing so we spare you the task of wading through introductory material. If you are entirely unfamiliar with a feature, 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 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 of the feature, as to the intended, idiomatic uses. 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

By scale, 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. As with many aspects of software development, what works for small programs simply doesn't scale to larger development efforts.

What's more, powerful new language features that are handled perfectly well by a few expert programmers working together in the archetypal garage on a prototype for their new start-up don't always fare as well when they are wantonly exercised by numerous members of 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 very large programs and by a very large number of programmers having a wide range of knowledge, skill, and ability.

---

[3] **stroustrup13**

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 "safely" as the moniker for the signature aspect of our book, by which we indicate a comparatively favorable risk-to-reward ratio for using a given feature in a large-scale development environment. By contextualizing the meaning of the term "safe," we get to 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 code base, and training needs for developers who might not be familiar with that feature.

Several aspects conspire to offset 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:

1. Number and severity of known deficiencies
2. Difficulty in teaching consistent proper use
3. Experience level required for consistent proper use
4. Risks associated with widespread misuse

Bottom line: 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 to be 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 easily be misused, and has no serious deficiencies. If someone has heard of this feature and tried to use it and the software compiles, the code base is likely better for it. Using `override` wherever applicable is always a sound engineering decision.

A *Conditionally Safe* Feature

The preponderance of new features available in modern C++ has 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 code base 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 may 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 very important, use cases, yet our experience indicates that routine or widespread use thereof 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

As an essential aspect of its design, this first edition of *Embracing Modern C++ Safely* aims to serve as a comprehensive catalog of C++11 and C++14 language features, presenting vital information for each of them in a clear, concise, consistent, and predictable format to which experienced engineers can readily refer during development or technical discourse.

Organization

This book is divided into five chapters, the middle three of which form the catalog characterizing 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
- Chapter 4: Parting Thoughts

For this first edition, the language-feature chapters (1, 2, and 3) each consist of 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 resides in its own subsection, rendered in a canonical format:

- Description
- Use Cases
- Potential Pitfalls
- Annoyances
- See Also
- Further Reading

By constraining our treatment of each individual feature to this canonized format, we avoid gratuitous variations in rendering, thereby facilitating rapid discovery of whatever particular aspects of a given language feature you are searching for.

How To Use This Book

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

1. **Read the entire book from front to back.** If you are conversant with classic C++, consuming this book in its entirety all at once will provide a complete and nuanced practical understanding of each of the language features introduced by C++11 and C++14.
2. **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.

3. **Read the first 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.

4. **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 subsection in any order will provide timely access to all relevant details of whichever feature is of immediate interest.

We wish you would derive value in several ways from the knowledge 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*. We are very much looking forward to getting feedback and suggestions for future editions of *Embracing Modern C++ Safely* at www.TODOTODOTODO.com. Happy coding!

Chapter 1

Safe Features

Intro text should be here.

Attributes

An *attribute* is an annotation (e.g., of a statement or named **entity**) used to provide supplementary information.

Description

Developers are often aware of information that is not deducible directly from the source code within a given translation unit. Some of this information might be useful to certain compilers, say, to inform diagnostics or optimizations; typical attributes, however, are designed to avoid affecting the semantics[1] of a well-written program. Customized annotations targeted at external (e.g., static-analysis) tools[2] might be beneficial as well.

C++ attribute syntax

C++ supports a standard syntax for attributes, introduced via a matching pair of `[[` and `]]`, the simplest of which is a single attribute represented using a simple identifier, e.g., `attribute_name`:

```
[[attribute_name]]
```

A single annotation can consist of zero or more attributes:

```
[[]]          // permitted in every position where any attribute is allowed
[[foo, bar]]  // equivalent to [[foo]] [[bar]]
```

An attribute may have an (optional) argument list consisting of an arbitrary sequence of tokens:

```
[[attribute_name()]]           // same as attribute_name
[[deprecated("too ugly")]]     // single-argument attribute
[[theoretical(1, "two", 3.0)]] // multiple-argument attributes
[[complicated({1, 2, 3} + 5)]] // arbitrary tokens (fails on GCC <= 9.2)
```

Note that having an incorrect number of arguments or an incompatible argument type is a compile-time error for all standard attributes; the behavior for all other attributes, however, is **implementation-defined** (see *Potential Pitfalls: Unrecognized attributes have implementation-defined behavior* on page 17).

[1] By *semantics*, here we typically mean any observable behavior apart from runtime performance. Generally, ignoring an attribute is a valid (and safe) choice for a compiler to make. Sometimes, however, an attribute will not affect the behavior of a *correct* program but might affect the behavior of a well-formed yet incorrect one (see *Use Cases: Delineating explicit assumptions in code to achieve better optimizations* on page 14).

[2] Such static-analysis tools include Clang sanitizers, Coverity, and other proprietary, open-source, and commercial products.

Any attribute may be namespace qualified[3] (using any arbitrary identifier):

```
[[gnu::const]]  // (GCC-specific) namespace-gnu-qualified const attribute
[[my::own]]     // (user-specified) namespace-my-qualified own attribute
```

C++ attribute placement

Attributes can, in principle, be introduced almost anywhere within the C++ syntax to annotate almost anything, including an entity, statement, code block, and even entire translation unit; however, most contemporary compilers do not support arbitrary placement of attributes (see *Use Cases: Probing where attributes are permitted in the compiler's C++ grammar* on page 17) outside of a declaration statement. Furthermore, in some cases, the entity to which an unrecognized attribute pertains might not be clear from its syntactic placement alone.

In the case of a declaration statement, however, the intended entity is well specified; an attribute placed in front of the statement applies to every entity being declared, whereas an attribute placed immediately after the named entity applies to just that one entity:

```
[[noreturn]] void f(), g();  // Both f() and g() are noreturn.
void u(), v() [[noreturn]];  // Only v() is noreturn.
```

Attributes placed in front of a declaration statement and immediately behind the name[4] of an individual entity in the same statement are additive (for that entity). The behavior of attributes associated with an entity across multiple declaration statements, however, depends on the attributes themselves. As an example, `[[noreturn]]` is required to be present on the *first* declaration of a function. Other attributes might be additive, such as the hypothetical `foo` and `bar` shown here:

```
[[foo]] void f(), g();  // declares both f() and g() to be foo
void f [[bar]](), g();  // Now f() is both foo and bar while
                        //     g() is still just foo.
```

Redundant attributes are not themselves necessarily considered an error; however, most standard attributes do consider redundancy an error[5]:

```
[[attr1]] void f [[attr2]](), f [[attr3]](int);
                                    // f()    is attr1 and attr2.
                                    // f(int) is attr1 and attr3.
```

[3] Attributes having a namespace-qualified name (e.g., `[[gnu::const]]`) were only **conditionally supported** in C++11 and C++14, but historically they were supported by all major compilers, including both Clang and GCC; all C++17-conforming compilers *must* support namespace-qualified names.

[4] There are rare edge cases in which an entity (e.g., an anonymous `union` or `enum`) is declared without a name:

```
struct S { union [[attribute_name]] { int a; float b }; };
enum [[attribute_name]] { SUCCESS, FAIL } result;
```

[5] Redundancy of standard attributes might no longer be an error in future revisions of the C++ Standard; see **iso20a**.

```
[[a1]][[a1]] int [[a1]][[a1]] & x;    // x (the reference itself) is a1.

void g [[noreturn]] [[noreturn]]();   // g() is noreturn.

void h [[noreturn, noreturn]]();      // error: repeated (standard) attribute
```

In most other cases, an attribute will typically apply to the statement (including a block statement) that immediately (apart from other attributes) follows it:

```
[[attr1]];                                // null statement
[[attr2]] return 0;                       // return statement
[[attr3]] for (int i = 0; i < 10; ++i);   // for statement
[[attr4]] [[attr5]] { /* ... */ }         // block statement
```

The valid positions of any particular attribute, however, will be constrained by whatever entities to which it applies. That is, an attribute such as `noreturn`, which pertains only to functions, would be valid syntactically but not semantically were it placed so as to annotate any other kind of entity or syntactic element. Misplacement of standard attributes results in an ill-formed program[6]:

```
void [[noreturn]] g() { throw; }  // error: appertains to type specifier
void i() [[noreturn]] { throw; }  // error: appertains to type specifier
```

Common compiler-dependent attributes

Prior to C++11, no standardized syntax was available to support conveying externally sourced information, and nonportable compiler intrinsics (such as `__attribute__((fallthrough))`, which is GCC-specific syntax) had to be used instead. Given the new standard syntax, vendors are now able to express these extensions in a more (syntactically) consistent manner. If an unknown attribute is encountered during compilation, it is ignored, emitting a (likely[7]) nonfatal diagnostic.

Table 1.0–1 provides a brief survey of popular compiler-specific attributes that have been standardized or have migrated to the standard syntax. (For additional compiler-specific attributes, see *Further Reading* on page 19.)

The requirement (as of C++17) to ignore unknown attributes helps to ensure portability of useful compiler-specific and external-tool annotations without necessarily having to employ conditional compilation so long as that attribute is permitted at that specific syntactic location by all relevant compilers (with some caveats; see *Potential Pitfalls: Not every syntactic location is viable for an attribute* on page 18).

[6] As of this writing, GCC is lax and merely warns when it sees the standard `noreturn` attribute in an unauthorized syntactic position, whereas Clang (correctly) fails to compile. Hence creative use of even a standard attribute might lead to different behavior on different compilers.

[7] Prior to C++17, a conforming implementation was permitted to treat an unknown attribute as ill formed and terminate translation; to the authors' knowledge, however, none of them did.

Table 1.0–1: Some standardized compiler-specific attributes

Compiler	Compiler-Specific	Standard-Conforming
GCC	`__attribute__((pure))`	`[[gnu::pure]]`
Clang	`__attribute__((no_sanitize))`	`[[clang::no_sanitize]]`
MSVC	`declspec(deprecated)`	`[[deprecated]]`

Use Cases

Eliciting useful compiler diagnostics

Decorating entities with certain attributes can give compilers enough additional context to provide more detailed diagnostics. For example, the GCC-specific `[[gnu::warn_unused_result]]` attribute[8] can be used to inform the compiler (and developers) that a function's return value should not be ignored[9]:

```
struct UDPListener
{
    [[gnu::warn_unused_result]] int start();
        // Start the UDP listener's background thread (which can fail for a
        // variety of reasons). Return 0 on success and a nonzero value
        // otherwise.

    void bind(int port);
        // The behavior is undefined unless start was called successfully.
};
```

Such annotation of the client-facing declaration can prevent defects caused by a client's forgetting to inspect the result of a function[10]:

```
void init()
{
    UDPListener listener;
    listener.start();      // Might fail; return value must be checked!
    listener.bind(27015);  // Possible undefined behavior; BAD IDEA!
}
```

For the code above, GCC produces a useful warning:

```
warning: ignoring return value of 'bool HttpClient::start()' declared
         with attribute 'warn_unused_result' [-Wunused-result]
```

[8] For compatibility with GCC, Clang supports `[[gnu::warn_unused_result]]` as well.

[9] The C++17 Standard `[[nodiscard]]` attribute serves the same purpose and is portable.

[10] Because the `[[gnu::warn_unused_result]]` attribute does not affect code generation, it is explicitly *not* ill formed for a client to make use of an unannotated declaration and yet compile its corresponding definition in the context of an annotated one (or vice versa); such is not always the case for other attributes, however, and best practice might argue in favor of consistency regardless.

Hinting at additional optimization opportunities

Some annotations can affect compiler optimizations leading to more efficient or smaller binaries. For example, decorating the function `reportError` (below) with the GCC-specific `[[gnu::cold]]` attribute (also available on Clang) tells the compiler that the developer believes the function is unlikely to be called often:

```
[[gnu::cold]] void reportError(const char* message) { /* ... */ }
```

Not only might the definition of `reportError` itself be optimized differently (e.g., for space over speed), any use of this function will likely be given lower priority during branch prediction:

```
void checkBalance(int balance)
{
    if (balance >= 0)  // likely branch
    {
        // ...
    }
    else  // unlikely branch
    {
        reportError("Negative balance.");
    }
}
```

Because the (annotated) `reportError(const char*)` appears on the else branch of the if statement (above), the compiler knows to expect that `balance` is likely *not* to be negative and therefore optimizes its predictive branching accordingly. Note that even if our hint to the compiler turns out to be misleading at run time, the semantics of every well-formed program remain the same.

Delineating explicit assumptions in code to achieve better optimizations

Although the presence (or absence) of an attribute usually has no effect on the behavior of any well-formed program (besides runtime performance), an attribute sometimes imparts knowledge to the compiler which, if incorrect, could alter the intended behavior of the program (or perhaps mask the defective behavior of an incorrect one). As an example of this more forceful form of attribute, consider the GCC-specific `[[gnu::const]]` attribute (also available on Clang). When applied to a function, this (atypically) powerful (and dangerous, see *Potential Pitfalls: Some attributes, if misused, can affect program correctness* on page 18) attribute instructs the compiler to *assume* that the function is a **pure function** (i.e., that it always returns the same value for any given set of arguments) and has no **side effects** (i.e., the globally reachable state[11] of the program is unaltered by calling this function):

[11] Absolutely no external state changes are allowed in a function decorated with `[[gnu::const]]`, including global state changes or mutation via any of the function's arguments. (The arguments themselves are considered local state and hence can be modified.) The (more lenient) `[[gnu::pure]]` attribute allows changes to the state of the function's arguments but still forbids any global state mutation. For example, any sort of (even temporary) global memory allocation would render a function ineligible for `[[gnu::const]]` or `[[gnu::pure]]`.

```
[[gnu::const]]
double linearInterpolation(double start, double end, double factor)
{
    return (start * (1.0 - factor)) + (end * factor);
}
```

The `vectorLerp` function (below) performs linear interpolation (referred to as LERP) between two bidimensional vectors. The body of this function comprises two invocations to the `linearInterpolation` function (above) — one per vector component:

```
Vector2D vectorLerp(const Vector2D& start, const Vector2D& end, double factor)
{
    return Vector2D(linearInterpolation(start.x, end.x, factor),
                    linearInterpolation(start.y, end.y, factor));
}
```

In the (possibly frequent) case where the values of the two components are the same, the compiler is allowed to invoke `linearInterpolation` only once — even if its body is not visible in `vectorLerp`'s translation unit:

```
// pseudocode (hypothetical compiler transformation)
Vector2D vectorLerp(const Vector2D& start, const Vector2D& end, double factor)
{
    if (start.x == start.y && end.x == end.y)
    {
        const double cache = linearInterpolation(start.x, end.x, factor);
        return Vector2D(cache, cache);
    }

    return Vector2D(linearInterpolation(start.x, end.x, factor),
                    linearInterpolation(start.y, end.y, factor));
}
```

If the implementation of a function tagged with the `[[gnu::pure]]` attribute does not satisfy limitations imposed by the attribute, however, the compiler will not be able to detect this and a runtime defect will be the likely result[12]; see *Potential Pitfalls: Some attributes, if misused, can affect program correctness* on page 18.

Using attributes to control external static analysis

Since unknown attributes are ignored by the compiler, external static-analysis tools can define their own custom attributes that can be used to embed detailed information to influence or control those tools without affecting program semantics. For example, the Microsoft-

[12] The briefly adopted — and then *unadopted* — contract-checking facility proposed for C++20 contemplated incorporating a feature similar in spirit to `[[gnu::const]]` in which preconditions (in addition to being runtime checked or ignored) could be *assumed* to be true by the compiler for the purposes of optimization; this unique use of attribute-like syntax also required that a conforming implementation could not unilaterally ignore these precondition-checking attributes since that would make attempting to test them result in hard (*language*) **undefined behavior**.

specific `[[gsl::suppress(/* rules */)]]` attribute can be used to suppress unwanted warnings from static-analysis tools that verify *Guidelines Support Library*[13] rules. In particular, consider GSL C26481 (Bounds rule #1),[14] which forbids any pointer arithmetic, instead suggesting that users rely on the `gsl::span` type[15]:

```
void hereticalFunction()
{
    int array[] = {0, 1, 2, 3, 4, 5};

    printElements(array, array + 6);  // elicits warning C26481
}
```

Any block of code for which validating rule C26481 is considered undesirable can be decorated with the `[[gsl::suppress(bounds.1)]]` attribute:

```
void hereticalFunction()
{
    int array[] = {0, 1, 2, 3, 4, 5};

    [[gsl::suppress(bounds.1)]]           // Suppress GSL C26481.
    {
        printElements(array, array + 6);  // Silence!
    }
}
```

Creating new attributes to express semantic properties

Other uses of attributes for static analysis include statements of properties that cannot otherwise be deduced within a single translation unit. Consider a function, `f`, that takes two pointers, `p1` and `p2`, and has a **precondition** that both pointers must refer to the same contiguous block of memory (as the two addresses are compared internally). Accordingly, we might annotate the function `f` with our own attribute `home_grown::in_same_block(p1, p2)`:

```
// lib.h

[[home_grown::in_same_block(p1, p2)]]
int f(double* p1, double* p2);
```

Now imagine that some client calls this function from some other translation unit but passes in two unrelated pointers:

```
// client.cpp
```

[13] *Guidelines Support Library* (see **microsoft**) is an open-source library, developed by Microsoft, that implements functions and types suggested for use by the "C++ Core Guidelines" (see **stroustrup20**).

[14] **microsoftC26481**

[15] `gsl::span` is a lightweight reference type that observes a contiguous sequence (or subsequence) of objects of homogeneous type. `gsl::span` can be used in interfaces as an alternative to both pointer/size or iterator-pair arguments and in implementations as an alternative to (raw) pointer arithmetic. Since C++20, the standard `std::span` template can be used instead.

```
#include <lib.h>

void client()
{
    double a[10], b[10];
    f(a, b);  // Oops, this is UB.
}
```

Because our static-analysis tool knows from the `home_grown::in_same_block` attribute that `a` and `b` must point into the same contiguous block, however, it has enough information to report, at compile time, what might otherwise have resulted in **undefined behavior** at run time.

Probing where attributes are permitted in the compiler's C++ grammar

An attribute can generally appear syntactically at the beginning of any statement — e.g., `[[attr]] x = 5;` — or in almost any position relative to a type or expression (e.g., `const int&`) but typically cannot be associated within named objects outside of a declaration statement:

```
[[]] static [[]] int [[]] a [[]], /*[[]]*/ b [[]];  // declaration statement
```

Notice how we have used the empty attribute syntax `[[]]` above to probe for positions allowed for arbitrary attributes by the compiler (in this case, GCC) — the only invalid one being immediately following the comma, shown above as `/*[[]]*/`. Outside of a declaration statement, however, viable attribute locations are typically far more limited:

```
[[]] void [[]] f [[]] ( [[]] int [[]] n [[]] )
[[]] {
    [[]] n /**/ *= /**/ sizeof /**/ ( [[]] const [[]] int [[]] & [[]] ) /**/;
    [[]] for ([[]] int [[]] i [[]] = /**/ 0 /**/ ;
                      /**/ i  /**/ < /**/ n /**/ ;
                /**/ ++ /**/ i /**/ )
    [[]] {
        [[]] ;            // [[]] denotes viable attribute location (on GCC)
    /**/ }
/**/ }                    // /**/ denotes no attribute allowed (on GCC)
```

Type expressions — e.g., the argument to `sizeof` (above) — are a notable exception; see *Potential Pitfalls: Not every syntactic location is viable for an attribute* on page 18.

Potential Pitfalls

Unrecognized attributes have implementation-defined behavior

Although standard attributes work well and are portable across all platforms, the behavior of compiler-specific and user-specified attributes is entirely implementation defined, with unrecognized attributes typically resulting in compiler warnings. Such warnings can typically

be disabled (e.g., on GCC using `-Wno-attributes`), but, if they are, misspellings in even standard attributes will go unreported.[16]

Some attributes, if misused, can affect program correctness

Many attributes are benign in that they might improve diagnostics or performance but cannot themselves cause a program to behave incorrectly. Some, however, if misused, can lead to incorrect results and/or **undefined behavior**.

For example, consider the `myRandom` function that is intended to return a new random number between [0.0 and 0.1] on each successive call:

```
double myRandom()
{
    static std::random_device randomDevice;
    static std::mt19937 generator(randomDevice());

    std::uniform_real_distribution<double> distribution(0, 1);
    return distribution(generator);
}
```

Suppose that we somehow observed that decorating `myRandom` with the `[[gnu::const]]` attribute occasionally improved runtime performance and innocently but naively decided to use it in production. This is clearly a misuse of the `[[gnu::const]]` attribute because the function doesn't inherently satisfy the requirement of producing the same result when invoked with the same arguments (in this case, none). Adding this attribute tells the compiler that it need not call this function repeatedly and is free to treat the first value returned as a constant for all time.

Not every syntactic location is viable for an attribute

For a fairly limited subset of syntactic locations, most conforming implementations are likely to tolerate the double-bracketed attribute-list syntax. The ubiquitously available locations include the beginning of any statement, immediately following a named entity in a declaration statement, and (typically) arbitrary positions relative to a **type expression** but, beyond that, caveat emptor. For example, GCC allowed all of the positions indicated in the example shown in *Use Cases: Probing where attributes are permitted in the compiler's C++ grammar* on pages 17–17, yet Clang had issues with the third line in two places:

```
<source>:3:39: error: expected variable name or 'this' in lambda capture list
    [[]] n /**/ *= /**/ sizeof /**/ ([[]] const [[]] int [[]] & [[]] ) /**/;
                                      ^


<source>:3:48: error: an attribute list cannot appear here
    [[]] n /**/ *= /**/ sizeof /**/ (/**/ const [[]] int [[]] & [[]] ) /**/;
                                               ^~~~
```

[16] Ideally, every relevant platform would offer a way to silently ignore a specific attribute on a case-by-case basis.

Hence, just because an arbitrary syntactic location is valid for an attribute on one compiler doesn't mean that it is necessarily valid on another.a

Annoyances

None so far

See Also

- "The **`[[noreturn]]`** Attribute" on page 64 — Safe C++11 standard attribute for functions that never return control flow to the caller
- "`[[carries_dependency]]` (The `[[carries_dependency]]` Attribute)" on page 104 — Unsafe C++11 standard attribute used to communicate release-consume dependency-chain information to the compiler to avoid unnecessary memory-fence instructions
- "`alignas`" on page 68 — Safe C++11 attribute (with a keyword-like syntax) used to widen the alignment of a type or an object
- "`[[deprecated]]`" on page 81 — Safe C++14 standard attribute that discourages the use of an entity via compiler diagnostics

Further Reading

For more information on commonly supported function attributes, see section 6.33.1, "Common Function Attributes," **freesoftwarefdn20**.

Binary Literals

Binary literals are **integer literals** representing their values in base 2.

Description

A *binary literal* (e.g., `0b1110`) — much like a hexadecimal literal (e.g., `0xE`) or an octal literal (e.g., `016`) — is a kind of *integer literal* (in this case, having the *decimal* value `14`). A binary literal consists of a `0b` (or `0B`) prefix followed by a nonempty sequence of binary digits (`0` or `1`)[17]:

```
int i = 0b11110000;  // equivalent to 240, 0360, or 0xF0
int j = 0B11110000;  // same value as above
```

The first digit after the `0b` prefix is the most significant one:

```
static_assert(0b0     ==  0, "");  // 0*2^0
static_assert(0b1     ==  1, "");  // 1*2^0
static_assert(0b10    ==  2, "");  // 1*2^1 + 0*2^0
static_assert(0b11    ==  3, "");  // 1*2^1 + 1*2^0
static_assert(0b100   ==  4, "");  // 1*2^2 + 0*2^1 + 0*2^0
static_assert(0b101   ==  5, "");  // 1*2^2 + 0*2^1 + 1*2^0
// ...
static_assert(0b11010 == 26, "");  // 1*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0
```

Leading zeros — as with octal and hexadecimal (but not decimal) literals — are ignored but can be added for readability:

```
static_assert(0b00000000 ==   0, "");
static_assert(0b00000001 ==   1, "");
static_assert(0b00000010 ==   2, "");
static_assert(0b00000100 ==   4, "");
static_assert(0b00001000 ==   8, "");
static_assert(0b10000000 == 128, "");
```

The type of a binary literal[18] is by default a (non-negative) `int` unless that value cannot fit in an `int`. In that case, its type is the first type in the sequence {`unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`}[19] in which it will fit. If neither of those is applicable, then the program is *ill-formed*[20]:

[17] Prior to being introduced in C++14, GCC supported binary literals (with the same syntax as the standard feature) as a nonconforming extension since version 4.3 (released between March 2008 and May 2010); for more details, see [CITATION TBD].

[18] Its *value category* is *prvalue* like every other integer literal.

[19] This same type list applies for both `octal` and `hex` literals but not for decimal literals, which, if initially `signed`, skip over any `unsigned` types, and vice versa (see the *Description* section).

[20] Purely for convenience of exposition, we have employed the C++11 `auto` feature to conveniently capture the type implied by the literal itself; for more information, see Section 2, "`auto`."

```
// example platform 1:
// (sizeof(int): 4; sizeof(long): 4; sizeof(long long): 8)
auto i32  = 0b0111...[ 24 1-bits]...1111;  // i32 is int.
auto u32  = 0b1000...[ 24 0-bits]...0000;  // u32 is unsigned int.
auto i64  = 0b0111...[ 56 1-bits]...1111;  // i64 is long long.
auto u64  = 0b1000...[ 56 0-bits]...0000;  // u64 is unsigned long long.
auto i128 = 0b0111...[120 1-bits]...1111;  // error: integer literal too large
auto u128 = 0b1000...[120 0-bits]...0000;  // error: integer literal too large

// example platform 2:
// (sizeof(int): 4; sizeof(long): 8; sizeof(long long): 16)
auto i32  = 0b0111...[ 24 1-bits]...1111;  // i32  is int.
auto u32  = 0b1000...[ 24 0-bits]...0000;  // u32  is unsigned int.
auto i64  = 0b0111...[ 56 1-bits]...1111;  // i64  is long.
auto u64  = 0b1000...[ 56 0-bits]...0000;  // u64  is unsigned long.
auto i128 = 0b0111...[120 1-bits]...1111;  // i128 is long long.
auto u128 = 0b1000...[120 0-bits]...0000;  // u128 is unsigned long long.
```

Separately, the precise initial type of a binary literal, like any other literal, can be controlled explicitly using the common integer-literal suffixes {`u`, `l`, `ul`, `ll`, `ull`} in either lower- or uppercase:

```
auto i   = 0b101;        // type: int;                value: 5
auto u   = 0b1010U;      // type: unsigned int;       value: 10
auto l   = 0b1111L;      // type: long;               value: 15
auto ul  = 0b10100UL;    // type: unsigned long;      value: 20
auto ll  = 0b11000LL;    // type: long long;          value: 24
auto ull = 0b110101ULL;  // type: unsigned long long; value: 53
```

Finally, note that affixing a minus sign (`-`) to a binary literal (e.g., `-b1010`) — just like any other integer literal (e.g., `-10`, `-012`, or `-0xa`) — is parsed as a non-negative value first, after which a unary minus is applied:

```
static_assert(sizeof(int) == 4, "");  // true on virtually all machines today
static_assert(-0b1010 == -10, "");    // as if: 0 - 0b1010 == 0 - 10
static_assert(0x7fffffff != -0x7fffffff, "");  // Both values are signed int.
static_assert(0x80000000 == -0x80000000, "");  // Both values are unsigned int.
```

Use Cases

Bit masking and bitwise operations

Prior to the introduction of binary literals, hexadecimal (and before that octal) literals were commonly used to represent bit masks (or specific bit constants) in source code. As an example, consider a function that returns the least significant four bits of a given `unsigned int` value:

```
unsigned int lastFourBits(unsigned int value)
{
```

```
    return value & 0xFu;
}
```

The correctness of the "bitwise and" operation above might not be immediately obvious to a developer inexperienced with hexadecimal literals. In contrast, use of a binary literal more directly states our intent to mask all but the four least-significant bits of the input:

```
unsigned int lastFourBits(unsigned int value)
{
    return value & 0b1111u;  // The u literal suffix here is entirely optional.
}
```

Similarly, other bitwise operations, such as setting or getting individual bits, might benefit from the use of binary literals. For instance, consider a set of flags used to represent the state of an avatar in a game:

```
struct AvatarStateFlags
{
    enum Enum
    {
        e_ON_GROUND    = 0b0001,
        e_INVULNERABLE = 0b0010,
        e_INVISIBLE    = 0b0100,
        e_SWIMMING     = 0b1000,
    };
};

class Avatar
{
    unsigned char d_state;  // power set of possible state flags

public:
    bool isOnGround() const
    {
        return d_flags & AvatarStateFlags::e_ON_GROUND;
    }

    // ...
};
```

Replicating constant binary data

Especially in the context of *embedded development* or emulation, a programmer will commonly write code that needs to deal with specific "magic" constants (e.g., provided as part of the specification of a CPU or virtual machine) that must be incorporated in the program's source code. Depending on the original format of such constants, a binary representation can be the most convenient or most easily understandable one.

As an example, consider a function decoding instructions of a virtual machine whose opcodes are specified in binary format:

```
#include <cstdint>  // std::uint8_t

void VirtualMachine::decodeInstruction(std::uint8_t instruction)
{
    switch(instruction)
    {
        case 0b00000000u:  // no-op
            break;

        case 0b00000001u:  // add(register0, register1)
            d_register0 += d_register1;
            break;

        case 0b00000010u:  // jmp(register0)
            jumpTo(d_register0);
            break;

        // ...
    }
}
```

Replicating the same binary constant specified as part of the CPU's (or virtual machine's) manual or documentation directly in the source avoids the need to mentally convert such constant data to and from, say, a hexadecimal number.

Binary literals are also suitable for capturing bitmaps. For instance, consider a bitmap representing the uppercase letter "C":

```
const unsigned char letterBitmap_C[] =
{
    0b00011111,
    0b01100000,
    0b10000000,
    0b10000000,
    0b10000000,
    0b01100000,
    0b00011111
};
```

Use of *binary* literals makes the shape of the image that the bitmap represents apparent directly in the source code.

Potential Pitfalls

None so far

Annoyances

None so far

See Also

- Section **??**, "**??**" — Safe C++14 feature that allows a developer to (visually) group together digits in a numerical literal to help readability. Often used in conjunction with binary literals.

Further Reading

None so far

Consecutive Right Angle Brackets

In the context of template argument lists, `>>` is parsed as two (separate) closing angle brackets.

Description

Prior to C++11, a pair of consecutive right-pointing angle brackets anywhere in the source code was always interpreted as a bitwise right-shift operator, making an intervening space mandatory for them to be treated as separate closing-angle-bracket tokens:

```
// C++03
std::vector<std::vector<int>> v0;   // annoying compile-time error in C++03
std::vector<std::vector<int> > v1;  // OK
```

To facilitate the common use case above, a special rule was added whereby, when parsing a template-argument expression, *non-nested* (i.e., within parentheses) appearances of `>`, `>>`, `>>>`, and so on are to be treated as separate closing angle brackets:

```
// C++11
std::vector<std::vector<int>> v0;               // OK
std::vector<std::vector<std::vector<int>>> v1;  // OK
```

Using the greater-than or right-shift operators within template-argument expressions

For templates that take only type parameters, there's no issue. When the template parameter is a non-type, however, the greater-than or right-shift operators might possibly be useful. In the unlikely event that we need either the greater-than operator (`>`) or the right-shift operator (`>>`) within a (non-type) template-argument expression, we can achieve our goal by nesting that expression within parentheses:

```
const int i = 1, j = 2;  // arbitrary integer values (used below)

template <int I> class C { /*...*/ };
    // class C taking non-type template parameter I of type int

C<i > j>    a1;  // compile-time error (always has been)
C<i >> j>   b1;  // compile-time error in C++11 (OK in C++03)
C<(i > j)>  a2;  // OK
C<(i >> j)> b2;  // OK
```

In the definition of `a1` above, the first `>` is interpreted as a closing angle bracket and the subsequent `j` is (and always has been) a syntax error. In the case of `b1`, the `>>` is, as of C++11, parsed as a pair of separate tokens in this context, so the second `>` is now considered an error. For both `a2` and `b2`, however, the would-be operators appear nested

(within parentheses) and thus are blocked from matching any active open angle bracket to the left of the parenthesized expression.

Use Cases

Avoiding annoying whitespace when composing template types

When using nested templated types (e.g., nested containers) in C++03, having to remember to insert an intervening space between trailing angle brackets added no value. What made it even more galling was that every popular compiler was able to tell you straight-up that you had forgotten to leave the space. With this new feature (rather, this repaired defect), we can now render closing angle brackets — just like parentheses and square brackets — contiguously:

```
// OK in both C++03 and C++11
std::list<std::map<int, std::vector<std::string> > > idToNameMappingList;

// OK in C++11, compile-time error in C++03
std::list<std::map<int, std::vector<std::string>>> idToNameMappingList;
```

Potential pitfalls

Some C++03 programs may stop compiling in C++11

If a right-shift operator is used in a template expression, the newer parsing rules may result in a compile-time error where before there was none:

```
T<1 >> 5>;  // worked in C++03, compile-time error in C++11
```

The easy fix is simply to parenthesize the expression:

```
T<(1 >> 5)>;  // OK
```

This rare syntax error is invariably caught at compile-time, avoiding undetected surprises at runtime.

The meaning of a C++03 program can (in theory) silently change in C++11

Though pathologically rare, the same valid expression can (in theory) have a different interpretation in C++11 than it had when compiled for C++03. Consider the case[21] where the `>>` token is embedded as part of an expression involving templates:

```
S<G< 0 >>::c>::b>::a
//   ^~~~~~
```

In the expression above, `0 >>::c` will be interpreted as a *bitwise right-shift operator* in C++03 but not in C++11. Writing a program that (1) compiles under both C++03 and C++11 and (2) exposes the difference in parsing rules, is possible:

[21] Adaptation of an example from **gustedt13**

```
enum Outer { a = 1, b = 2, c = 3 };

template <typename> struct S
{
    enum Inner { a = 100, c = 102 };
};

template <int> struct G
{
    typedef int b;
};

int main()
{
    std::cout << (S<G< 0 >>::c>::b>::a) << '\n';
}
```

The program above will print `100` when compiled for C++03 and `0` for C++11:

```
// C++03

//     (2) instantiation of G<0>
//    ||~~~~~~~~~~~~
//    || | ||   (4) instantiation of S<int>
//  ~~|| ↓ ||~~~~~~~~~~~~~~↓
    S< G< 0 >>::c > ::b >::a
//    ~~|| ↑ ||~~~~~~~~~↑
//      || | || (3) type alias for int
//      ||~~~~~~~
// (1) bitwise right-shift (0 >> 3)

// C++11

//
//
//  (2) compare (>) Inner::c and Outer::b
//  ↓ ~~~~~~~~~~~~~~~~~~
    S< G< 0 >>::c > ::b >::a
//  ↑ ~~~~~~~~~
//  (1) instantiation of S<G<0>>
//
//
```

Though theoretically possible, programs that are (1) syntactically valid in both C++03 and C++11 and (2) have distinct semantics have not emerged in practice anywhere that we are aware of.

Annoyances

None so far

See Also

None so far

Further Reading

- Daveed Vandevoorde, *Right Angle Brackets,* **vandevoorde05**

decltype

The keyword `decltype` enables the compile-time inspection of the **declared type** of an **entity** *or* the type and **value category** of an expression.

Description

What results from the use of `decltype` depends on the nature of its operand.

Use with (typically named) entities

If an unparenthesized operand is either an **id-expression** that names an entity (or a non-type template parameter) or a **class member access expression** (that identifies a class member), `decltype` yields the *declared type* (the type of the *entity* indicated by the operand):

```
int i;               // decltype(i)   -> int
std::string s;       // decltype(s)   -> std::string
int* p;              // decltype(p)   -> int*
const int& r = *p;   // decltype(r)   -> const int&
struct { char c; } x; // decltype(x.c) -> char
double f();          // decltype(f)   -> double()
double g(int);       // decltype(g)   -> double(int)
```

Use with (unnamed) expressions

When `decltype` is used with any other expression `E` of type `T`, the result incorporates both the expression's type and its **value category**:

Value category of `E`	Result of `decltype(E)`
prvalue	`T`
lvalue	`T&`
xvalue	`T&&`

The three integer expressions below illustrate the various value categories:

```
decltype(0)   // -> int   (*prvalue* category)

int i;
decltype((i)) // -> int&  (*lvalue* category)

int&& g();
decltype(g()) // -> int&& (*xvalue* category)
```

Much like the `sizeof` operator (which too is resolved at compile time), the expression operand to `decltype` is not evaluated:

```
int i = 0;
decltype(i++) j;  // equivalent to int j;
assert(i == 0);   // The function next is never invoked.
```

Use Cases

Avoiding unnecessary use of explicit typenames

Consider two logically equivalent ways of declaring a vector of iterators into a list of `Widgets`:

```
std::list<Widget> widgets;
std::vector<std::list<Widget>::iterator> widgetIterators;
    // (1) The full type of widgets needs to be restated, and iterator
    // needs to be explicitly named.

std::list<Widget> widgets;
std::vector<decltype(widgets.begin())> widgetIterators;
    // (2) Neither std::list nor Widget nor iterator need be named
    // explicitly.
```

Notice that, when using `decltype`, if the C++ type representing the widget changes (e.g., from `Widget` to, say, `ManagedWidget`) or the container used changes (e.g., from `std::list` to `std::vector`), the declaration of `widgetIterators` need not necessarily change.

Expressing type-consistency explicitly

In some situations, repetition of explicit type names might inadvertently result in latent defects caused by mismatched types during maintenance. For example, consider a `Packet` class exposing a `const` member function that returns a `std::uint8_t` representing the length of the packet's checksum:

```
class Packet
{
    // ...
public:
    std::uint8_t checksumLength() const;
};
```

This (tiny) unsigned 8-bit type was selected to minimize bandwidth usage as the checksum length is sent over the network. Next, picture a loop that computes the checksum of a `Packet`, using the same (i.e., `std::uint8_t`) type for its iteration variable (to match the type returned by `Packet::checksumLength`):

```
void f()
{
    Checksum sum;
    Packet data;
```

```
    for (std::uint8_t i = 0; i < data.checksumLength(); ++i)  // brittle
    {
        sum.appendByte(data.nthByte(i));
    }
}
```

Now suppose that, over time, the data transmitted by the `Packet` type grows to the point where the range of a `std::uint8_t` value might not be enough to ensure a sufficiently reliable checksum. If the type returned by `checksumLength()` is changed to, say, `std::uint16_t` without updating the type of the iteration variable `i` in lockstep, the loop might silently[22] become infinite.[23]

Had `decltype(packet.checksumLength())` been used to express the type of `i`, the types would have remained consistent and the ensuing ("truncation") defect would naturally have been avoided:

```
// ...
for (decltype(data.checksumLength()) i = 0; i < data.checksumLength(); ++i)
// ...
```

Creating an auxiliary variable of generic type

Consider the task of implementing a generic `loggedSum` function (template) that returns the sum of two arbitrary objects `a` and `b` after logging both the operands and the result value (e.g., for debugging or monitoring purposes). To avoid computing the (possibly expensive) sum twice, we decide to create an auxiliary function-scope variable, `result`. Since the type of the sum depends on both `a` and `b`, we can use `decltype(a + b)` to infer the type for both (1) the (trailing) return type[24] of the function (see Section 1, "Short Title (Trailing Function Return Types)") and (2) the auxiliary variable:

```
template <typename A, typename B>
auto loggedSum(const A& a, const B& b)
    -> decltype(a + b)                // (1) exploiting trailing return types
{
    decltype(a + b) result = a + b;    // (2) auxiliary generic variable
    LOG_TRACE << a << " + " << b << " = " << result;
    return result;
}
```

[22] As of this writing, neither GCC 9.3 nor Clang 10.0.0 provide a warning (using `-Wall`, `-Wextra`, and `-Wpedantic`) for the comparison between `std::uint8_t` and `std::uint16_t` — even if both (1) the value returned by `checksumLength` does not fit in a 8-bit integer and (2) the body of the function is visible to the compiler. Decorating `checksumLength` with `constexpr` causes `clang++` to issue a warning, but this is clearly not a general solution.

[23] The (tiny) loop variable is promoted to an `unsigned int` for comparison purposes but wraps (to 0) whenever its value prior to being incremented is 255.

[24] Using `decltype(a + b)` as a return type is significantly different from relying on *automatic return type deduction*. See Section 2, "`auto`," for more information.

Determining the validity of a generic expression

In the context of generic-library development, `decltype` can be used in conjunction with **SFINAE**[25] to validate an expression involving a template parameter.

For example, consider the task of writing a generic `sortRange` function template that, given a **range**, invokes either the `sort` member function of the argument (the one specifically optimized for that type) if available, or else falls back to the more general `std::sort`:

```
template <typename Range>
void sortRange(Range& range)
{
    sortRangeImpl(range, 0);
}
```

The client-facing `sortRange` function (above) delegates its behavior to an (overloaded) `sortRangeImpl` function (below), invoking the latter with the `range` and a *disambiguator* of type `int`. The type of this additional parameter (its value is arbitrary) is used to give priority to the `sort` member function (at compile time) by exploiting overload resolution rules in the presence of an implicit (*standard*) conversion (from `int` to `long`):

```
template <typename Range>
void sortRangeImpl(Range& range,
                   long)                  // low priority: standard conversion
{
    // fallback implementation
    std::sort(std::begin(range), std::end(range));
}
```

The fallback overload of `sortRangeImpl` (above) will accept a `long` *disambiguator* (requiring a standard conversion from `int`) and will simply invoke `std::sort`. The more specialized overload of `sortRangeImpl` (below) will accept an `int` *disambiguator* (requiring no conversions) and thus will be a better match, provided a range-specific sort is available:

```
template <typename Range>
void sortRangeImpl(Range& range,
                   int,                           // high priority: exact match
                   decltype(range.sort())* = 0)   // check expression validity
{
    // optimized implementation
    range.sort();
}
```

Note that, by exposing[26] `decltype(range.sort())` as part of `sortRangeImpl`'s declaration,

[25] "Substitution Failure Is Not An Error"

[26] The relative position of `decltype(range.sort())` in the signature of `sortRangeImpl` is not significant, as long as it is visible to the compiler (as part of the function's *logical interface*) during template substitution. This particular example (shown in the main text) makes use of a function parameter that is defaulted to `nullptr`. Alternatives involving a trailing return type or a default template argument are also viable:

```
template <typename Range>
```

the more specialized overload will be discarded during template substitution if `range.sort()` is not a valid expression for the deduced `Range` type.[27]

Putting it all together, we see that exactly two possible outcomes exist for the original client-facing `sortRange` function invoked with a range argument of type `R`:

- If `R` does have a `sort` member function, the more specialized overload (of `sortRangeImpl`) will be viable (as `range.sort()` is a well-formed expression) and preferred because the *disambiguator* `0` (of type `int`) requires no conversion.
- Otherwise, the more specialized overload will be discarded during template substitution (as `range.sort()` is not a well-formed expression) and the only remaining (more general) `sortRangeImpl` overload will be chosen instead.

Potential pitfalls

Perhaps surprisingly, `decltype(x)` and `decltype((x))` will sometimes yield different results for the same expression `x`:

```
int i = 0; // decltype(i) yields int.
           // decltype((i)) yields int&.
```

In the case where the unparenthesized operand is an entity having a declared type `T` and the parenthesized operand is an expression whose value category is represented (by `decltype`) as the same type `T`, the results will coincidentally be the same:

```
int& ref = i;  // decltype(ref) yields int&.
               // decltype((ref)) yields int&.
```

Wrapping its operand with parentheses ensures `decltype` yields the **value category** of a given expression. This technique can be useful in the context of metaprogramming — particularly in the case of **value category** propagation.

Annoyances

None so far

```
auto sortRangeImpl(Range& range, int) -> decltype(range.sort(), void());
    // The comma operator is used to force the return type to void,
    // regardless of the return type of range.sort().

template <typename Range, typename = decltype(std::declval<Range&>().sort()>
auto sortRangeImpl(Range& range, int);
    // std::declval is used to generate a reference to Range that can
    // be used in an unevaluated expression.
```

[27] The technique of exposing a (possibly unused) unevaluated expression (e.g., using `decltype`) in a function's declaration for the purpose of expression-validity detection prior to template instantiation is commonly known as **expression SFINAE** and is a restricted form of the more general (classical) SFINAE that acts exclusively on expressions visible in a function's signature rather than on (obscure) template-based type computations.

See Also

None so far

Further reading

None so far

Deleted Functions

Use of `= delete` in a function's (first) declaration forces a compilation error upon any attempt to use or access it.

Description

Declaring a particular function (or function overload) to result in a fatal diagnostic upon invocation can be useful — e.g., to suppress the generation of a *special function* or to limit the types of arguments a particular function is able to accept. In such cases, `= delete;` can be used in place of the body of any function (on first declaration only) to force a compile-time error if any attempt is made to invoke it or take its address.

```
void g(double) { }
void g(int) = delete;

void f()
{
    g(3.14);  // OK, f(double) is invoked.
    g(0);     // Error: f(int) is deleted.
}
```

Notice that deleted functions participate in *overload resolution* and produce a compile-time error when selected as the best candidate.

Use Cases

Suppressing special member function generation

When instantiating an object of user-defined type, **special member functions** that have not been declared explicitly are often[28] generated automatically by the compiler. For certain kinds of types, the notion of **copy semantics** (including **move semantics**[29]) is not meaningful and hence permitting the generation of copy operations is contraindicated.

Consider a class, `FileHandle`, that uses the **RAII** idiom to safely acquire and release an I/O stream. As *copy semantics* are typically not meaningful for such resources, we will want to suppress generation of both the *copy constructor* and *copy assignment operator*. Prior to C++11, there was no direct way to express suppression of *special functions* in C++. The commonly recommended workaround was to declare the two methods `private` and

[28] The generation of individual special member functions can be affected by the existence of other user-defined special member functions or by limitations imposed by the specific types of any data members or base types. For more information, see Section **??**, "**??**."

[29] The two **special member functions** controlling *move* operations (introduced in C++11) are sometimes implemented as effective optimizations of copy operations and (rarely) with copy operations explicitly deleted; see Section 2, "Rvalue References."

leave them unimplemented, typically resulting in a compile-time (or link-time) error when accessed[30]:

```
class FileHandle
{
private:
    // ...

    FileHandle(const FileHandle&);             // not implemented
    FileHandle& operator=(const FileHandle&);  // not implemented

public:
    explicit FileHandle(FILE* filePtr);
    ~FileHandle();

    // ...
};
```

With the `= delete` syntax, we are able to (1) explicitly express our intention to make these special member functions unavailable, (2) do so directly in the `public` region of the class, and (3) enable more precise compiler diagnostics:

```
class FileHandle
{
private:
    // ...

public:
    explicit FileHandle(FILE* filePtr);
    ~FileHandle();

    FileHandle(const FileHandle&) = delete;             // make unavailable
    FileHandle& operator=(const FileHandle&) = delete;  // make unavailable

    // ...
};
```

Preventing a particular implicit conversion

Certain functions — especially those that take a `char` as an argument — are prone to inadvertent misuse. As a truly classic example, consider the C library function `memset`, which may be used to write the character `*` five times in a row, starting at a specified memory address, `buf`:

```
#include <cstring>
#include <cstdio>
```

[30] Leaving unimplemented a special member function that is declared to be private ensures that there will be at least a link-time error in case that function is inadvertently accessed from within the implementation of the class itself.

```
void f()
{
    char buf[] = "Hello World!";
    memset(buf, 5, '*');  // undefined behavior
    puts(buf);            // expected output: "***** World!"
}
```

Sadly, inadvertently reversing the final two arguments is a commonly recurring error, and the C language provides no help. In C++, we can target such observed misuse using an extra deleted overload:

```
#include <cstring>  // memset()
void* memset(void* str, int ch, size_t n);      // standard library function
void* memset(void* str, int n, char) = delete;  // defensive against misuse
```

Pernicious user errors can now be reported during compilation:

```
// ...
memset(buf, 5, '*');  // Error: memset(void, int, char) is deleted.
// ...
```

Preventing all implicit conversions

The `ByteStream::send` member function below is designed to work with 8-bit unsigned integers only. Providing a deleted overload accepting an `int` forces a caller to ensure that the argument is always of the appropriate type:

```
class ByteStream
{
public:
    void send(unsigned char byte) { /* ... */ }
    void send(int) = delete;

    // ...
};

void f()
{
    ByteStream stream;
    stream.send(0);   // Error: send(int) is deleted.     (1)
    stream.send('a'); // Error: send(int) is deleted.     (2)
    stream.send(0L);  // Error: ambiguous                 (3)
    stream.send(0U);  // Error: ambiguous                 (4)
    stream.send(0.0); // Error: ambiguous                 (5)
    stream.send(
        static_cast<unsigned char>(100));  // OK          (6)
}
```

Invoking `send` with an `int` (noted with (1) in the code above) or any integral type (other than `unsigned char`[31]) that promotes to `int` (2) will map exclusively to the deleted `send(int)` overload; all other integral (3 & 4) and floating-point types (5) are convertible to both (via a **standard conversion**) and hence will be ambiguous. An explicit cast to `unsigned char` (6) can always be pressed into service if needed.

Hiding a structural (nonpolymorphic) base class's member function

Best practices notwithstanding,[32] it can be cost-effective (in the short term) to provide an elided "view" on a concrete class for (trusted) clients. Imagine a class `AudioStream` designed to play sounds and music that — in addition to providing basic "play" and "rewind" operations — sports a large, robust interface:

```
struct AudioStream
{
    void play();
    void rewind();
    // ...
    // ... (large, robust interface)
    // ...
};
```

Now suppose that, on short notice, we need to whip up a very similar class, `ForwardAudioStream`, to use with audio samples that cannot be rewound (e.g., coming directly from a live feed). Realizing that we can readily reuse most of `AudioStream`'s interface, we pragmatically decide to prototype the new class simply by exploiting public **structural inheritance** and then deleting just the lone unwanted `rewind` member function:

```
struct ForwardAudioStream : AudioStream
{
    void rewind() = delete; // make just this one function unavailable
};

void f()
{
    ForwardAudioStream stream = FMRadio::getStream();
    stream.play();   // fine
    stream.rewind(); // Error: rewind() is deleted.
}
```

[31] Note that implicitly converting from `unsigned char` to either a `long` or `unsigned` integer involves a **standard conversion** (not just an **integral promotion**), the same as converting to a `double`.

[32] By publicly deriving from a concrete class, we do not hide the underlying capabilities, which can easily be accessed (perhaps accidentally) via assignment to a pointer or reference to a base class (no casting required). What's more, inadvertently passing such a class to a function taking the base class by value will result in *slicing*, which can be especially problematic when the derived class holds data. Finally, if the derived class purports to maintain *class invariants* that the base class does not preserve, this design technique is beyond dubious; a more robust approach would be to use layering or at least private inheritance. For more on improving compositional designs at scale, see **lakos20**, sections 3.5.10.5 and 3.7.3, pp. 687–703 and 726–727, respectively.

If the need for a `ForwardAudioStream` type persists, we can always consider reimplementing it more carefully later.[33]

Potential Pitfalls

None so far

Annoyances

None so far

See Also

- Section **??**, "**??**" — Companion safe C++11 feature that enables *defaulting* (as opposed to *deleting*) special member functions
- Section 2, "Rvalue References" — Conditionally safe C++11 feature that introduces the two *move* variants to *copy* special member functions

Further Reading

None so far

[33] **lakos20**, sections 3.5.10.5 and 3.7.3, pp. 687–703 and 726–727

override

The override keyword ensures that a member function overrides a corresponding virtual member function in a base class.

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 (i.e., not **hiding** it or otherwise inadvertently introducing a distinct function declaration):

```
struct Base
{
    virtual void f(int);
    void g(int);
};

struct Derived : Base
{
    void f();              // hides Base::f(int) (likely mistake)
    void f() override;     // error: Base::f() not found

    void f(int);           // implicitly overrides Base::f(int)
    void f(int) override;  // explicitly overrides Base::f(int)

    void g();              // hides Base::g(int) (likely mistake)
    void g() override;     // error: Base::g() not found

    void g(int);           // hides Base::g(int) (likely mistake)
    void g(int) override;  // Error: Base::g() is not virtual.
};
```

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

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 runtime. 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 (instead of overriding) the base class's `virtual` member function. Both of the errors discussed above would be detected automatically by decorating the `virtual` functions in all derived classes with `override`:

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

    bool equivolent(int code, const ErrorCondition& code) override;
        // compile-time error when first written
};
```

What's more, `override` serves as a clear indication to the human reader of the derived class's author's intent to customize the behavior of `ErrorCategory`. For any given member function, use of `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 code base

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.,

statically verified 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 world" 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.

Annoyances

None so far

See Also

None so far

Further Reading

None so far

Compile-Time Assertions (`static_assert`)

The `static_assert` keyword allows programmers to intentionally terminate compilation whenever a given compile-time predicate evaluates to `false`.

Description

Assumptions are inherent in every program, whether we explicitly document them or not. A common way of validating certain assumptions at runtime is to use the classic `assert` macro found in `<cassert>`. Such runtime assertions are not always ideal because (1) the program must already be built and running for them to even have a chance of being triggered and (2) executing a **redundant check** at runtime typically[34] results in a slower program. Being able to validate an assertion at compile time avoids several drawbacks:

1. Validation occurs at compile time within a single translation unit, and therefore doesn't need to wait until a complete program is linked and executed.
2. Compile-time assertions can exist in many more places than runtime assertions and are unrelated to program control flow.
3. No runtime code will be generated due to a `static_assert`, so program performance will not be impacted.

Syntax and semantics

We can use *static assertion declarations* to conditionally trigger controlled compilation failures depending on the truthiness of a **constant expression**. Such declarations are introduced by the `static_assert` keyword, followed by a parenthesized list consisting of (1) a constant Boolean expression and (2) a mandatory (see *Annoyances: Mandatory string literal* on page 50) **string literal**, which will be part of the compiler diagnostics if the compiler determines that the assertion fails to hold:

```
static_assert(true, "Never fires.");
static_assert(false, "Always fires.");
```

Static assertions can be placed anywhere in the scope of a namespace, block, or class:

```
static_assert(1 + 1 == 2, "Never fires.");  // (global) namespace scope

template <typename T>
struct S
{
    void f0()
```

[34] It is not unheard of for a program having assertions to run faster with them enabled than disabled — e.g., when asserting that a pointer is not null, thereby enabling the optimizer to elide all code branches that can be reached only if that pointer were null.

```
    {
        static_assert(1 + 1 == 3, "Always fires.");  // block scope
    }

    static_assert(!Predicate<T>::value, "Might fire.");  // class scope
};
```

Providing a non-constant expression to a `static_assert` is itself a compile-time error:

```
extern bool x;
static_assert(x, "Nice try.");  // Error: x is not a compile-time constant.
```

Evaluation of static assertions in templates

The C++ Standard does not explicitly specify at precisely what point (during the compilation process) static assertion declarations are evaluated.[35] In particular, when used within the body of a template, a `static_assert` declaration might not be evaluated until **template instantiation time**. In practice, however, a `static_assert` that does not depend on any template parameters is essentially always[36] evaluated immediately — i.e., as soon as it is parsed and irrespective of whether any subsequent template instantiations occur:

```
void f1()
{
    static_assert(false, "Impossible!");  // always evaluated immediately...
}                                         // even if f1() is never invoked

template <typename T>
void f2()
{
    static_assert(false, "Impossible!");  // always evaluated immediately...
}                                         // even if f2() is never instantiated
```

The evaluation of a static assertion that is (1) located within the body of a class or function template and (2) depends on at least one template parameter is almost always bypassed during its initial parse since the value — true or false — of the assertion will (in general) depend on the nature of the template argument:

```
template <typename T>
void f3()
{
    static_assert(sizeof(T) >= 8, "Size < 8.");  // depends on T
}
```

(However, see *Potential Pitfalls: Static assertions in templates can trigger unintended compilation failures* on page 47.) In the example above, the compiler has no choice but to wait

[35] By "evaluated" here, we mean that the asserted expression is processed and its semantic truth determined.

[36] E.g., GCC 10.1, Clang 10.0, and MSVC 19.24

until each time `f3` is instantiated because the truth of the predicate will vary depending on the type provided:

```
void g()
{
    f3<double>();               // OK
    f3<long double>();          // OK
    f3<std::complex<float>>();  // OK
    f3<char>();                 // Error: static assertion failed: Size < 8.
}
```

The standard does, however, specify that a program containing any template definition for which no valid specialization exists is **ill formed** (no diagnostic required), which was the case for `f2` but not `f3`, above. Contrast each of the `h*n*` definitions (below) with its correspondingly numbered `f*n*` definition (above):

```
void h1()
{
    int a[!sizeof(int) - 1];  // same as int a[-1]; and is ill formed
}

template <typename T>
void h2()
{
    int a[!sizeof(int) - 1];  // always reported as a compile-time error
}

template <typename T>
void h3()
{
    int a[!sizeof(T) - 1];    // typically reported only if instantiated
}
```

Both `f1` and `h1` are ill-formed, non-template functions, and both will always be reported at compile time, albeit typically with decidedly different error messages as demonstrated by GCC 10.x's output:

```
f1: error: static assertion failed: Impossible!
h1: error: size -1 of array a is negative
```

Both `f2` and `h2` are ill-formed template functions; the cause of their being ill formed has nothing to do with the template type and hence will always be reported as a compile-time error in practice. Finally, `f3` can be only contextually ill formed whereas `h3` is always necessarily ill formed, and yet neither is reported by typical compilers as such unless and until it has been instantiated. Reliance on a compiler not to notice that a program is ill formed is dubious; see *Potential Pitfalls: Static assertions in templates can trigger unintended compilation failures* on page 47.

Use Cases

Verifying assumptions about the target platform

Some programs rely on specific properties of the native types provided by their target platform. Static assertions can help ensure portability and prevent such programs from being compiled (into a malfunctioning binary) on, say, an unsupported platform. As an example, consider a program that relies on the size of an `int` to be exactly 32 bits (e.g., due to the use of inline `asm` blocks). Placing a `static_assert` in namespace scope in any of the program's translation units will (1) ensure that the assumption regarding the size of `int` is valid and (2) serve as documentation for readers:

```
#include <ctype>  // CHAR_BIT

static_assert(sizeof(int) * CHAR_BIT == 32,
    "An int must have exactly 32 bits for this program to work correctly.");
```

More typically, statically asserting the *size* of an `int` avoids having to write code to handle an `int` type's having greater or fewer bytes when no such platforms are likely ever to materialize:

```
static_assert(sizeof(int) == 4, "An int must have exactly 4 bytes.");
```

Preventing misuse of class and function templates

Static assertions are often used in practice to constrain class or function templates to prevent their being instantiated with unsupported types by either (1) substantially improving compile-time diagnostics[37] or, more critically, (2) actively avoiding erroneous runtime behavior.

As an example, consider the `SmallObjectBuffer<N>` class templates, which provide storage for arbitrary objects whose size does not exceed `N`[38]:

```
template <std::size_t N>
class SmallObjectBuffer
{
private:
    char d_buffer[N];

public:
    template <typename T>
    void set(const T& object);
```

[37] Syntactically incompatible types often lead to absurdly long and notoriously hard-to-read diagnostic messages, especially when deeply nested template expressions are involved.

[38] A `SmallObjectBuffer` is similar to C++17's `std::any` (**cppref_stdany**) in that it can store any object of any type. Instead of performing dynamic allocation to support arbitrarily sized objects, however, `SmallObjectBuffer` uses an internal fixed-size buffer, which can lead to better performance and cache locality provided (the maximum size of) all of the types involved is known.

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

To prevent buffer overruns, it is important that `set` accepts only those objects that will fit in `d_buffer`. The use of a static assertion in the `set` member function template catches — at compile time — any such misuse:

```
template <std::size_t N>
template <typename T>
void SmallObjectBuffer<N>::set(const T& object)
{
    static_assert(sizeof(T) <= N, "object does not fit in the small buffer.");
    new (&d_buffer) T(object);
}
```

The principle of constraining inputs can be applied to most class and function templates. `static_assert` is particularly useful in conjunction with standard **type traits** provided in `<type_traits>`. In the `rotateLeft` function template (below), we have used two static assertions to ensure that only unsigned integral types will be accepted:

```
#include <ctype>  // CHAR_BIT

template <typename T>
T rotateLeft(T x)
{
    static_assert(std::is_integral<T>::value, "T must be an integral type.");
    static_assert(std::is_unsigned<T>::value, "T must be an unsigned type.");

    return (x << 1) | (x >> (sizeof(T) * CHAR_BIT - 1));
}
```

Potential Pitfalls

Static assertions in templates can trigger unintended compilation failures

As mentioned in the description, any program containing a template for which no valid specialization can be generated is (by definition) **ill formed** (no diagnostic required). Attempting to prevent the use of, say, a particular function template overload by using a static assertion that never holds produces such a program:

```
template <bool>
struct SerializableTag { };

template <typename T>
void serialize(char* buffer, const T& object, SerializableTag<true>);  // (1)

template <typename T>
void serialize(char* buffer, const T& object, SerializableTag<false>)  // (2)
{
```

```
    static_assert(false, "T must be serializable.");  // independent of T
        // too obviously ill formed: always a compile-time error
}
```

In the example above, the second overload (2) of `serialize` is provided with the intent of eliciting a meaningful compile-time error message in the event that an attempt is made to serialize a nonserializable type. The program, however, is technically *ill-formed* and, in this simple case, will likely result in a compilation failure — irrespective of whether either overload of `serialize` is ever instantiated.

A commonly attempted workaround is to make the predicate of the assertion somehow dependent on a template parameter, ostensibly forcing the compiler to withhold evaluation of the `static_assert` unless and until the template is actually instantiated (a.k.a. **instantiation time**):

```
template <typename>  // N.B., we make no use of the (nameless) type parameter:
struct AlwaysFalse   // This class exists only to "outwit" the compiler.
{
    enum { value = false };
};

template <typename T>
void serialize(char* buffer, const T& object, SerializableTag<false>)  // (2)
{
    static_assert(AlwaysFalse<T>::value, "T must be serializable.");  // OK
        // less obviously ill formed: compile-time error when instantiated
}
```

To implement this version of the second overload, we have provided an intermediary class template `AlwaysFalse` that, when instantiated on any type, contains an enumerator named `value`, whose value is `false`. Although this second implementation is more likely to produce the desired result (i.e., a controlled compilation failure only when `serialize` is invoked with unsuitable arguments), sufficiently "smart" compilers looking at just the current translation unit would still be able to know that no valid instantiation of `serialize` exists and would therefore be well within their rights to refuse to compile this still technically *ill-formed* program.

Equivalent workarounds achieving the same result without a helper class are possible.

```
template <typename T>
void serialize(char* buffer, const T& object, SerializableTag<false>)  // (2)
{
    static_assert(0 == sizeof(T), "T must be serializable.");  // OK
         // not too obviously ill formed: compile-time error when instantiated
}
```

Know that use of this sort of obfuscation is not guaranteed to be either portable or future-proof: *caveat emptor*.

Misuse of static assertions to restrict overload sets

Even if we are careful to *fool* the compiler into thinking that a specialization is wrong *only* if instantiated, we still cannot use this approach to remove a candidate from an overload set because translation will terminate if the static assertion is triggered. Consider this (flawed) attempt at writing a `process` function that will behave differently depending on the size of the given argument:

```
template <typename T>
void process(const T& x)  // (1) first definition of process function
{
    static_assert(sizeof(T) <= 32, "Overload for small types");  // BAD IDEA
    // ... (process small types)
}

template <typename T>
void process(const T& x)  // (2) compile-time error: redefinition of function
{
    static_assert(sizeof(T) > 32, "Overload for big types");     // BAD IDEA
    // ... (process big types)
}
```

While the intention of the developer might have been to statically dispatch to one of the two mutually exclusive overloads, the ill-fated implementation above will not compile because the signatures of the two overloads are identical, leading to a redefinition error. The semantics of `static_assert` are not suitable for the purposes of **compile-time dispatch**.

To achieve the goal of removing (up front) a specialization from consideration, we will need to employ **SFINAE**. To do that, we must instead find a way to get the failing compile-time expression to be part of the function's **declaration**[39]:

```
template <bool> struct Check { };
    // helper class template having a (non-type) boolean template parameter
    // representing a compile-time predicate

template <> struct Check<true> { typedef int Ok; };
    // specialization of Check that makes the type Ok manifest *only* if
    // the supplied predicate (boolean template argument) evaluates to true

template <typename T,
          typename Check<(sizeof(T) <= 32)>::Ok = 0>  // SFINAE
void process(const T& x)  // (1)
{
    // ... (process small types)
}

template <typename T,
```

[39] **Concepts** — a language feature introduced in C++20 — provides a far less baroque alternative to SFINAE that allows for overload sets to be governed by the syntactic properties of their (compile-time) template arguments.

```
          typename Check<(sizeof(T) > 32)>::Ok = 0>  // SFINAE
void process(const T& x)  // (2)
{
    // ... (process big types)
}
```

The (empty) `Check` helper class template in conjunction with just one of its two possible specializations (above) conditionally exposes the `Ok` type alias *only* if the provided boolean template parameter evaluates to `true`. (Otherwise, by default, it does not.)

During the substitution phase of template instantiation, exactly one of the two overloads of the `process` function will attempt to access a nonexisting `Ok` type alias via the `Check<false>` instantiation, which again, by default, is nonexistent. Although such an error would typically result in a compilation failure, in the context of template argument substitution it will instead result in only the offending overload's being discarded, giving other (valid) overloads a chance to be selected:

```
void client()
{
    process(SmallType());  // discards (2), selects (1)
    process(BigType());    // discards (1), selects (2)
}
```

This general technique of pairing template specializations is used widely in modern C++ programming. For another, often more convenient way of constraining overloads using **expression SFINAE**, see Section 1, "Short Title (Trailing Function Return Types)."

Annoyances

Mandatory string literal

Many compilation failures caused by static assertions are self-explanatory since the offending line (which necessarily contains the predicate code) is displayed as part of the compiler diagnostic. In those situations, the message required[40] as part of `static_assert`'s grammar is redundant:

```
static_assert(std::is_integral<T>::value, "T must be an integral type.");
```

Developers commonly provide an empty string literal in these cases:

```
static_assert(std::is_integral<T>::value, "");
```

See Also

- Section 1, "Short Title (Trailing Function Return Types)" — Safe C++11 feature that allows fine-grained control over overload resolution by enabling **expression SFINAE** as part of a function's **declaration**

[40] As of C++17, the message argument of a static assertion is optional.

Further reading

None so far

Trailing Function Return Types

Trailing return types provide a new alternate syntax in which the return type of a function is specified at the end of a function declaration (as opposed to at the beginning), thereby allowing it to reference function parameters by name and to reference class or namespace members without explicit qualification.

Description

C++ offers an alternative function-declaration syntax in which the return type of a function is located to the right of its **signature** (name, parameters, and qualifiers), offset by the arrow token (`->`); the function itself is introduced by the keyword `auto`, which acts as a type placeholder[41]:

```
auto f() -> void;  // equivalent to void f();
```

Using a trailing return type allows the parameters of a function to be named as part of the specification of the return type, which can be useful in conjunction with `decltype`:

```
auto g(int x) -> decltype(x);  // equivalent to int g(int x);
```

When using the trailing-return-type syntax in a member function definition outside the class definition, names appearing in the return type, unlike with the classic notation, will be looked up in class scope by default:

```
struct S
{
    typedef int T;
    auto h1() -> T;  // trailing syntax for member function
    T h2();          // classical syntax for member function
};

auto S::h1() -> T { /*...*/ }  // equivalent to S::T S::h1() { /.../ }
T    S::h2()      { /*...*/ }  // Error: T is unknown in this context.
```

The same advantage would apply to a nonmember function[42] defined outside of the namespace in which it is declared:

```
namespace N
{
```

[41] Note that, when using the alternative trailing return syntax for a function (e.g., one returning a `double`), the override keyword would be inserted after call qualifiers and before the arrow:

```
virtual f(int value) const override -> double;
```

[42] A `static` member function of a `struct` can be a viable alternative implementation to a free function declared within a namespace; see **lakos20**, section 1.4, pp. 190–201, especially Figure 1-37c (p. 199), and section 2.4.9, pp. 312–321, especially Figure 2-23 (p. 316).

```
    typedef int T;
    auto h3() -> T;  // trailing syntax for free function
    T h4();          // classical syntax for free function
};

auto N::h3() -> T { /*...*/ }  // equivalent to N::T N::h3() { /.../ }
T    N::h4()      { /*...*/ }  // Error: T is unknown in this context.
```

Finally, since the syntactic element to be provided after the arrow token is a separate type unto itself, return types involving pointers to functions are (somewhat) simplified. Suppose, for example, we want to describe a **higher-order function**, `f`, that takes as its argument a `long long` and returns a pointer to a function that takes an `int` and returns a `double`[43]:

```
// [function(long long) returning]
//     [pointer to] [function(int x) returning] double   f;
//     [pointer to] [function(int x) returning] double   f(long long);
//                  [function(int x) returning] double  *f(long long);
//                                              double (*f(long long))(int x);
```

Using the alternate trailing syntax, we can conveniently break the declaration of `f` into two parts: (1) the declaration of the function's signature, `auto f(long long)`, and (2) that of the return type, say, `R` for now:

```
// [pointer to] [function (int) returning] double   R;
//              [function (int) returning] double  *R;
//                                         double (*R)(int);
```

The two equivalent forms of the same declaration are shown below:

```
double (*f(long long))(int x);         // classic return-type syntax
auto f(long long) -> double (*)(int);  // trailing return-type syntax
```

Note that both syntactic forms of the same declaration may appear together within the same scope. Note also that not all functions that can be represented in terms of the trailing syntax have a convenient equivalent representation in the classic one:

```
template <typename A, typename B>
auto foo(A a, B b) -> decltype(a.foo(b));
    // trailing return-type syntax

template <typename A, typename B>
decltype(std::declval<A&>().foo(std::declval<B&>())) foo(A a, B b);
    // classic return-type syntax (using C++11's std::declval)
```

In the example above, we were essentially forced to use the (C++11) standard library template `std::declval` (**cppref_declval**) to express our intent with the classic return-type syntax.

[43] Co-author John Lakos first used the shown verbose declaration notation while teaching Advanced Design and Programming using C++ at Columbia University (1991-1997).

Use Cases

Function template whose return type depends on a parameter type

Declaring a function template whose return type depends on the types of one or more of its parameters is not uncommon in generic programming. For example, consider a mathematical function that linearly interpolates between two values of (possibly) different type:

```
template <typename A, typename B, typename F>
auto linearInterpolation(const A& a, const B& b, const F& factor)
    -> decltype(a + factor * (b - a))
{
    return a + factor * (b - a);
}
```

The return type of `linearInterpolation` is the type of expression inside the *`decltype` specifier*, which is identical to the expression returned in the body of the function. Hence, this interface necessarily supports any set of input types for which `a + factor * (b - a)` is valid, including types such as mathematical vectors, matrices, or expression templates. As an added benefit, the presence of the expression in the function's declaration enables **expression SFINAE**, which is typically desirable for generic template functions (see Section 1, "`decltype` (Long Title)").

Avoiding having to qualify names redundantly in return types

When defining a function outside the `class`, `struct`, or `namespace` in which it is first declared, any unqualified names present in the return type might be looked up differently depending on the particular choice of function-declaration syntax used. When the return type precedes the qualified name of the function definition (as is the case with classic syntax), all references to types declared in the same scope where the function itself is declared must also be (redundantly) qualified. By contrast, when the return type follows the qualified name of the function (as is the case when using the trailing-return-type syntax), the return type (just like any parameter types) is — by default — looked up in the same scope in which the function was first declared. Avoiding such redundancy can be beneficial, especially when the (redundant) qualifying name is not short.

As an illustration, consider a class (representing an abstract syntax tree node) that exposes a type alias:

```
struct NumericalASTNode
{
    using ElementType = double;
    auto getElement() -> ElementType;
};
```

Defining the `getElement` member function using traditional function-declaration syntax would require repetition of the `NumericalASTNode` name:

```
NumericalASTNode::ElementType NumericalASTNode::getElement() { /*...*/ }
```

Using the trailing-return-type syntax handily avoids the repetition:

```
auto NumericalASTNode::getElement() -> ElementType { /*...*/ }
```

By ensuring that name lookup within the return type is the same as for the parameter types, we avoid needlessly having to qualify names that should be found correctly by default.

Improving readability of declarations involving function pointers

Declarations of functions returning a pointer to either (1) a function, (2) a member function, or (3) a data member are notoriously hard to parse — even for seasoned programmers. As an example, consider a function called `getOperation` that takes, as its argument, a `kind` of (enumerated) `Operation` and returns a pointer to a member function of `Calculator` that takes a `double` and returns a `double`:

```
double (Calculator::*getOperation(Operation kind))(double);
```

As we saw in the description, such declarations can be constructed systematically but do not exactly roll off the fingers. On the other hand, by partitioning the problem into (1) the declaration of the function itself and (2) the type it returns, each individual problem becomes far simpler than the original:

```
auto getOperation(Operation kind)  // (1) function taking a kind of Operation
    -> double (Calculator::*)(double);
        // (2) returning a pointer to a Calculator member function taking a
        //     double and returning a double
```

Using this divide-and-conquer approach, writing a **higher-order function** that returns a pointer to a function, member function, or data member as its return type[44] becomes fairly straightforward.

Potential Pitfalls

None so far

Annoyances

None so far

See Also

- Section 1, "`decltype` (Long Title)" — Safe C++11 type inference feature that is often used in conjunction with (or in place of) trailing return types

[44] Declaring a **higher-order function** that takes a function pointer as an argument might be even easier to read if a type alias is used (e.g., via `typedef` or, as of C++11, `using`).

- Section 3, "Deduced Return Types (Function Return Type Deduction)" — Unsafe C++14 type inference feature that shares syntactical similarities with trailing return types, leading to potential pitfalls when migrating from C++11 to C++14

Further Reading

None so far

Unrestricted Unions

Any nonreference type is permitted to be a member of a `union`.

Description

Prior to C++11, only **trivial types** — e.g., **fundamental types**, such as `int` and `double`, enumerated or pointer types, or a C-style array or `struct` (a.k.a. a **POD**) — were allowed to be members of a `union`. This limitation prevented any (user-defined) type having a **non-trivial special member function** from being a member of a `union`:

```
union U0
{
    int         d_i;  // OK
    std::string d_s;  // compile-time error in C++03 (OK as of C++11)
};
```

C++11 relaxes such restrictions on `union` members, such as `d_s` above, allowing any type other than a **reference type** to be a member of a `union`.

A `union` type is permitted to have user-defined special member functions but — by design — does not initialize any of its members automatically. Any member of a `union` having a **non-trivial constructor**, such as `struct Nt` (below), must be constructed manually (e.g., via **placement `new`** implemented within the body of a constructor of the union itself) before it can be used:

```
struct Nt  // used as part of a union (below)
{
    Nt();   // non-trivial default constructor
    ~Nt();  // non-trivial destructor

    // Copy construction and assignment are implicitly defaulted.
    // Move construction and assignment are implicitly deleted.
};
```

As an added safety measure, any non-trivial **special member function** defined — either implicitly or explicitly — for any *member* of a `union` results in the compiler implicitly deleting (see Section 1, "Deleted Functions (Long Title)") the corresponding **special member function** of the `union` itself:

```
union U1
{
    int d_i;   // fundamental type having all trivial special member functions
    Nt  d_nt;  // user-defined type having non-trivial special member functions

    // Implicitly deleted special member functions of U1:
    /*
        U1()                    = delete; // due to explicit Nt::Nt()
```

```
        U1(const U1&)            = delete; // due to implicit Nt::Nt(const Nt&)
        ~U1()                    = delete; // due to explicit Nt::~Nt()
        U1& operator=(const U1&) = delete; // due to implicit
                                           // Nt::operator=(const Nt&)
    */
};
```

This same sort of precautionary deletion also occurs for any class containing such a union as a data member (see *Use Cases: Implementing a **sum type** as a discriminating (or tagged)* `union` on page 60).

A special member function of a `union` that is implicitly deleted can be restored via explicit declaration, thereby forcing a programmer to think about how non-trivial members should be managed. For example, we can start providing a *value constructor* and corresponding *destructor*:

```
struct U2
{
    union
    {
        int  d_i;   // fundamental type (trivial)
        Nt   d_nt;  // non-trivial user-defined type
    };

    bool d_useInt;  // discriminator

    U2(bool useInt) : d_useInt(useInt)       // value constructor
    {
        if (d_useInt) { new (&d_i) int(); }  // value initialized (to 0)
        else          { new (&d_nt) Nt(); }  // default constructed in place
    }

    ~U2()  // destructor
    {
        if (!d_useInt) { d_nt.~Nt(); }
    }
};
```

Notice that we have employed **placement `new`** syntax to control the lifetime of both member objects. Although assignment would be permitted for the (trivial) `int` type, it would be **undefined behavior** for the (non-trivial) `Nt` type:

```
U2(bool useInt) : d_useInt(useInt)  // value constructor
{
    if (d_useInt) { d_i = int(); }  // value initialized (to 0)
    else          { d_nt = Nt(); }  // undefined behavior
}
```

Now if we were to try to copy-construct or assign an object of type `U2` to another, the operation would fail because we have not (yet) specifically addressed those **special member**

functions:

```
void f()
{
    U2 a(false), b(true);  // OK (construct both instances of U2)
    U2 c(a);               // compile-time error: no U2(const U2&)
    a = b;                 // compile-time error: no U2& operator=(const U2&)
}
```

We can restore these implicitly deleted special member functions too, simply by adding appropriate copy-constructor and assignment-operator definitions for `U2` explicitly[45]:

```
union U2
{
    // ... (everything in U2 above)

    U2(const U2& original) : d_useInt(original.d_useInt)
    {
        if (d_useInt) { new (&d_i) int(original.d_i);  }
        else          { new (&d_nt) Nt(original.d_nt); }
    }

    U2& operator=(const U2& rhs)
    {
        if (this == &rhs) // Prevent self-assignment.
        {
            return *this;
        }

        // Resolve all possible combinations of active types between the
        // left-hand side and right-hand side of the assignment:

        if (d_useInt)
        {
            if (rhs.d_useInt) { d_i = rhs.d_i; }
            else              { new (&d_nt) Nt(rhs.d_nt); }
        }
        else
        {
            if (rhs.d_useInt) { d_nt.~Nt(); new (&d_i) int(rhs.d_i); }
            else              { d_nt = rhs.d_nt; }
        }

        return *this;
    }
};
```

[45] Attempting to restore a `union`'s implicitly deleted special member functions by using the `= default` syntax (see Section **??**, "**??**") will still result in their being deleted because the compiler cannot know which member of the union is active without a discriminator.

Use Cases

Implementing a sum type as a discriminating (or tagged) `union`

A **sum type** is an abstract data type that provides a choice among a fixed set of specific types. Although other implementations are possible, using the storage of a single object to accommodate one out of a set of types along with a (typically integral) discriminator enables programmers to implement a **sum type** (a.k.a. *discriminating* or *tagged* union) efficiently (e.g., without necessarily involving memory allocation or virtual dispatch) and nonintrusively (i.e., the individual types comprised need not be related in any way). A C++ `union` can serve as a convenient and efficient way to define storage for a **sum type** as alignment and size calculations are performed (by the compiler) automatically.

As an example, consider writing a parsing function `parseInteger` that, given a `std::string input`, will return, as a **sum type** `ParseResult` (see below), either an `int` result (on success) or an informative error message (on failure):

```
ParseResult parseInteger(const std::string& input)  // Return a sum type.
{
    int result;     // accumulate result as we go
    std::size_t i;  // current character index

    // ...

    if (/* Failure case (1). */)
    {
        std::ostringstream oss;
        oss << "Found non-numerical character '" << input[i]
            << "' at index '" << i << "'.";

        return ParseResult(oss.str());
    }

    if (/* Failure case (2). */)
    {
        std::ostringstream oss;
        oss << "Accumulating '" << input[i]
            << "' at index '" << i
            << "' into the current running total '" << result
            << "' would result in integer overflow.";

        return ParseResult(oss.str());
    }

    // ...

    return ParseResult(result);  // Success!
}
```

The implementation above relies on `ParseResult` being able to hold a value of type either

`int` or `std::string`. By encapsulating a C++ `union` and a Boolean[46] *discriminator* as part of the `ParseResult` **sum type**, we can achieve the desired semantics:

```
class ParseResult
{
    union  // storage for either the result or the error
    {
        int         d_value;  // trivial result type
        std::string d_error;  // non-trivial error type
    };

    bool d_isError;  // discriminator

public:
    explicit ParseResult(int value);                // value constructor (1)
    explicit ParseResult(const std::string& error);  // value constructor (2)

    ParseResult(const ParseResult& rhs);            // copy constructor
    ParseResult& operator=(const ParseResult& rhs);  // copy assignment

    ~ParseResult();                                 // destructor
};
```

As discussed in *Description* on page 57, having a non-trivial type within a `union` forces the programmer to provide each desired special member function and define it manually; note, although, that the use of placement `new` is not required for either of the two *value constructors* (above) because the initializer syntax (below) is sufficient to begin the lifetime of even a non-trivial object:

```
ParseResult::ParseResult(double value) : d_value(value), d_isError(false)
{
}

ParseResult::ParseResult(const std::string& error)
    : d_error(error), d_isError(true)
    // Note that placement new was not necessary here because a new
    // std::string object will be created as part of the initialization of
    // d_error.
{
}
```

Placement `new` and explicit destructor calls are, however, required for destruction and both copy operations[47]:

```
ParseResult::~ParseResult()
{
```

[46] For **sum types** comprising more than two types, a larger integral or enumerated type may be used instead.

[47] For more information on initiating the lifetime of an object, see **iso14**, section 3.8, "Object Lifetime," pp. 66–69.

```
    if(d_isError)
    {
        d_error.std::string::~string();
            // An explicit destructor call is required for d_error because its
            // destructor is non-trivial.
    }
}

ParseResult::ParseResult(const ParseResult& rhs) : d_isError(rhs.d_isError)
{
    if (d_isError)
    {
        new (&d_error) std::string(rhs.d_error);
            // Placement new is necessary here to begin the lifetime of a
            // std::string object at the address of d_error.
    }
    else
    {
        d_value = rhs.d_value;
            // Placement new is not necessary here as int is a trivial type.
    }
}

ParseResult& ParseResult::operator=(const ParseResult& rhs)
{
    // Destroy lhs's error string if existent:
    if (d_isError) { d_error.std::string::~string(); }

    // Copy rhs's object:
    if (rhs.d_isError) { new (&d_error) std::string(rhs.d_error); }
    else               { d_value = rhs.d_value; }

    d_isError = rhs.d_isError;
    return *this;
}
```

In practice, `ParseResult` would typically be defined as a template and renamed to allow any arbitrary result type `T` to be returned or else implemented in terms of a more general **sum type** abstraction.[48]

[48] `std::variant`, introduced in C++17, is the standard construct used to represent a **sum type** as a *discriminating union*. Prior to C++17, `boost::variant` was the most widely used *tagged* union implementation of a **sum type**.

Potential Pitfalls

Inadvertent misuse can lead to latent undefined behavior at runtime

When implementing a type that makes use of an unrestricted union, forgetting to initialize a non-trivial object (using either a *member initialization list* or **placement `new`**) or accessing a different object than the one that was actually initialized can result in tacit **undefined behavior**. Although forgetting to destroy an object does not necessarily result in **undefined behavior**, failing to do so for any object that manages a resource (such as dynamic memory) will result in a *resource leak* and/or lead to unintended behavior. Note that destroying an object having a trivial destructor is never necessary; there are, however, rare cases where we may choose not to destroy an object having a non-trivial one.[49]

Annoyances

None so far

See Also

- Section 1, "Deleted Functions (Long Title)" — Safe C++11 feature that forbids the invocation of a particular function. Similar effects to deleting a function happen when we specify a special function within a subobject of a union or when a class has such a union as a data member.

Further Reading

None so far

[49] A specific example of where one might deliberately choose *not* to destroy an object occurs when a collection of related objects are allocated from the same local memory resource and then deallocated unilaterally by releasing the memory back to the resource. No issue arises if the only resource that is "leaked" by not invoking each individual destructor is the memory allocated from that memory resource, and that memory can be reused without resulting in **undefined behavior** if it is not subsequently referenced in the context of the deallocated objects.

The [[noreturn]] Attribute

The [[noreturn]] attribute promises that the function to which it pertains never returns.

Description

The presence of the standard [[noreturn]] attribute as part of a function declaration informs both the compiler and human readers that such a function never returns control flow to the caller:

```
[[noreturn]] void f()
{
    throw;
}
```

The [[noreturn]] attribute is not part of a function's type and is also, therefore, not part of the type of a function pointer. Applying [[noreturn]] to a function pointer is not an error, though doing so has no actual effect in standard C++; see *Potential Pitfalls: Misuse of* [[noreturn]] *on function pointers* on page 66. Use on a pointer might have benefits for external tooling, code expressiveness, and future language evolution:

```
void (*fp [[noreturn]])() = f;
```

Use Cases

Better compiler diagnostics

Consider the task of creating an assertion handler that, when invoked, always aborts execution of the program after printing some useful information about the source of the assertion. Since this specific handler will never return, it is a viable candidate for [[noreturn]]:

```
[[noreturn]] void abortingAssertionHandler(const char* filename, int line)
{
    LOG_ERROR << "Assertion fired at " << filename << ':' << line;
    std::abort();
}
```

The additional information provided by the attribute will allow a compiler to warn if it determines that a code path in the function would allow it to return:

```
[[noreturn]] void abortingAssertionHandler(const char* filename, int line)
{
    if (filename)  // just being safe, but see "Further Reading," below
    {
        LOG_ERROR << "Assertion fired at " << filename << ':' << line;
        std::abort();
    }
}  // compile-time warning made possible
```

This information can also be used to warn in case unreachable code is present after `abortingAssertionHandler` is invoked:

```
int main()
{
    // ...
    abortingAssertionHandler("main.cpp", __LINE__);
    std::cout << "We got here.\n";  // compile-time warning made possible
    // ...
}
```

Note that this warning is made possible by decorating just the declaration of the handler function — i.e., even if the definition of the function is not visible in the current translation unit.

Improved runtime performance

If the compiler knows that it is going to invoke a function that is guaranteed not to return, the compiler is within its rights to optimize that function by removing what it can now determine to be dead code. As an example, consider a utility component, `util`, that defines a function, `throwBadAlloc`, that is used to **insulate** the throwing of an `std::bad_alloc` exception in what would otherwise be template code fully exposed to clients:

```
// util.h:
[[noreturn]] void throwBadAlloc();

// util.cpp:
#include <util.h>  // [[noreturn]] void throwBadAlloc()

void throwBadAlloc()  // This redeclaration is also [[noreturn]].
{
    throw std::bad_alloc();
}
```

Irrespective of whether the client compiler warns, the compiler is within its rights to elide code that is rendered unreachable by the call to the `throwBadAlloc` function due to the function being decorated with the `[[noreturn]]` attribute on its declaration:

```
#include <util.h>  // [[noreturn]] void throwBadAlloc()

void client()
{
    // ...
    throwBadAlloc();
    // ... (Everything below here can be optimized away.)
}
```

Notice that even though `[[noreturn]]` appeared only on the first declaration (in the `util.h` header), the `[[noreturn]]` attribute carries over to the redeclaration used in the

`throwBadAlloc` function's definition because the header was included in the corresponding `.cpp` file.

Potential Pitfalls

`[[noreturn]]` can inadvertently break an otherwise working program

Unlike many attributes, use of `[[noreturn]]` *can* alter the semantics of a well-formed program, potentially introducing a runtime defect and/or making the program ill-formed. If a function that can potentially return is decorated with `[[noreturn]]` and then, in the course of executing a program, it ever does return, that behavior is **undefined**.

Consider a `printAndExit` function whose role is to print a fatal error message before aborting the program:

```
[[noreturn]] void printAndExit()
{
    std::cout << "Fatal error. Exiting the program.\n";
    assert(false);
}
```

The programmer chose to (sloppily) implement termination by using an assertion, which would not be incorporated into a program compiled with the preprocessor definition `NDEBUG` active, and thus `printAndExit` would (in that build mode) return normally. If the compiler of the client is informed that function will not return, the compiler is free to optimize accordingly. If the function then does return, any number of hard-to-diagnose defects (e.g., due to incorrectly elided code) might materialize as a consequence of the ensuing **undefined behavior**. Furthermore, within a program, if a function is declared `[[noreturn]]` in some translation units but not in others, that program is (inherently) **ill-formed, no diagnostic required**.

Misuse of `[[noreturn]]` on function pointers

Although the `[[noreturn]]` attribute is permitted to appertain to a function pointer (syntactically) for the benefit of external tools, it has no effect in standard C++; fortunately, most compilers will warn:

```
void (*fp [[noreturn]])();  // not supported by standard C++ (will likely warn)
```

What's more, assigning (the address of) a function that is not decorated with `[[noreturn]]` to an otherwise suitable function pointer that is so decorated is perfectly fine:

```
void f() { return; };  // function that always returns

void g()
{
    fp = f;  // "OK" -- that fp is [[noreturn]] is silently ignored
}
```

Any reliance on `[[noreturn]]` to have any effect in standard C++ when applied to other than a function's declaration is misguided.

Annoyances

None so far

See Also

- Section 1, "Attributes (Long Title)" — To learn more about allowed attribute placement in general

Further Reading

None so far

alignas Long Title

alignas, a keyword that acts like an attribute, is used to widen (make more strict) the **alignment** of a **variable**, **user-defined type**, or **data member**.

Description

The alignas specifier provides a means of further restricting the granularity at which (1) a particular object of arbitrary type, (2) a user-defined type (class, struct, union, or enum), or (3) an individual data member is permitted to reside within the virtual-memory-address space.

Restricting the alignment of a particular object

In its most basic form, alignas acts like an attribute that accepts (as an argument) an **integral constant expression** representing an explicitly supplied minimum alignment value:

```
alignas(64) int i;    // OK, i is aligned on a 64-byte address boundary.
int j alignas(8), k;  // OK, j is 8-byte aligned; k remains naturally aligned.
```

If more than one alignment pertains to a given object, the most restrictive alignment value is applied:

```
alignas(4) alignas(8) alignas(2) char m;  // OK, m is 8-byte aligned.
alignas(8) int n alignas(16);             // OK, n is 16-byte aligned.
```

For a program to be **well formed**, a specified alignment value must satisfy several requirements:

1. Be either zero or a non-negative integral power of two of type std::size_t (0, 1, 2, 4, 8, 16…).
2. Be at least the minimum alignment[50] required by the decorated entity.
3. Be no more than the largest alignment[51] supported on the platform in the context in which the entity appears.

[50]The minimum alignment of an entity is the least restrictive memory-address boundary at which the entity can be placed and have the program continue to work properly. This value is platform dependent and often subject to compiler controls but, by default, is often well approximated by **natural alignment**; see *Appendix: Natural Alignment* on page 77.

[51]The notion of the largest supported alignment is characterized by both **maximal alignment** and the maximum **extended alignment**. **Maximal alignment** is defined as that most restrictive alignment that is valid in *all* contexts on the current platform. All fundamental and pointer types necessarily have a minimal alignment that is less than or equal to alignof(std::max_align_t) — typically 8 or 16. Any alignment value greater than **maximal alignment** is an **extended alignment** value. Whether any extended alignment is supported (and in which contexts) is implementation defined. On typical platforms, extended alignment will often be as large as 2^{18} or 2^{19}, however implementations may warn when the alignment of a global object

Additionally, if the specified alignment value is zero, the `alignas` specifier is ignored:

```
// Static variables declared at namespace scope
alignas(32) int i0;  // OK, aligned on a 32-byte boundary (extended alignment)
alignas(16) int i1;  // OK, aligned on a 16-byte boundary (maximum alignment)
alignas(8)  int i2;  // OK, aligned on an 8-byte boundary
alignas(7)  int i3;  // error: not a power of two
alignas(4)  int i4;  // OK, no change to alignment boundary
alignas(2)  int i5;  // error: less than minimum alignment on this platform
alignas(0)  int i6;  // OK, alignas specifier ignored

alignas(1024 * 16) int i7;
    // OK, might warn: e.g., exceeds (physical) page size on current platform

alignas(1024 * 1024 * 512) int i8;
    // (likely) compile-time error: e.g., exceeds maximum size of object file

alignas(8) char buf[128];  // create 8-byte-aligned, 128-byte character buffer

void f()
{
    // automatic variables declared at function scope
    alignas(4)  double e0;  // error: less than minimum alignment on this platform
    alignas(8)  double e1;  // OK, no-change to (8-byte) alignment boundary
    alignas(16) double e2;  // OK, aligned to maximum (fundamental) alignment value
    alignas(32) double e3;  // OK, maximum alignment value exceeded; might warn
}
```

Restricting the alignment of a user-defined type

The `alignas` specifier can also be used to specify alignment for user-defined types (UDTs), such as a `class`, `struct`, `union`, or `enum`. When specifying the alignment of a UDT, the `alignas` keyword is placed *after* the type specifier (e.g., `class`) and just before the name of the type (e.g., `C`):

```
class alignas(2)  C { };  // OK, aligned on a 2-byte boundary; size = 2
struct alignas(4) S { };  // OK, aligned on a 4-byte boundary; size = 4
union alignas(8)  U { };  // OK, aligned on an 8-byte boundary; size = 8
enum alignas(16)  E { };  // OK, aligned on a 16-byte boundary; size = 4
```

Notice that, for each of `class`, `struct`, and `union` above, the `sizeof` objects of that type increased to match the alignment; in the case of the `enum`, however, the size remains that of

exceeds some maximal hardware threshold (such as the size of a physical memory page, e.g., 4096 or 8192). For **automatic variables** (defined on the program stack), making alignment more restrictive than what would naturally be employed is seldom desired because at most one thread is able to access proximately located variables there unless explicitly passed in via address to separate threads; see *Use Cases: Avoiding **false sharing** among distinct objects in a multi-threaded program* on page 73. Note that, in the case of `i` in the first code snippet on page 68, a conforming platform that did not support an extended alignment of 64 would be required to report an error at compile time.

the default **underlying type** (e.g., 4 bytes) on the current platform.[52]

Again, specifying an alignment that is less than what would occur naturally or else is restricted explicitly is ill formed:

```
struct alignas(2) T0 { int i; };
    // Error: Alignment of T0 (2) is less than that of int (4).
struct alignas(1) T1 { C c; };
    // Error: Alignment of T1 (1) is less than that of C (2).
```

Restricting the alignment of individual data members

Within a user-defined type (`class`, `struct`, or `union`), using the attribute-like syntax of the `alignas` keyword to specify the alignments of individual data members is possible:

```
struct T2
{
    alignas(8)  char   x;  // size   1; alignment 8
    alignas(16) int    y;  // size   4; alignment 16
    alignas(64) double y;  // size   8; alignment 64
};  // size 128; alignment 64
```

The effect here is the same as if we had added the padding explicitly and then set the alignment of the structure overall:

```
struct alignas(64) T3
{
    char   x;      // size   1; alignment 8
    char   a[15];  // padding
    int    y;      // size   4; alignment 16
    char   b[44];  // padding
    double z;      // size   8; alignment 64
    char   c[56];  // padding (optional)
};  // size 128; alignment 64
```

Again, if more than one attribute pertains to a given data member, the maximum applicable alignment value is applied:

```
struct T4
{
    alignas(2) char
        c1 alignas(1),  // size 1; alignment 2
        c2 alignas(2),  // size 1; alignment 2
        c4 alignas(4);  // size 1; alignment 4
};                      // size 8; alignment 4
```

[52] When `alignas` is applied to an enumeration `E`, the Standard does not indicate whether padding bits are added to `E`'s object representation or not, affecting the result of `sizeof(E)`. The implementation variance resulting from this lack of clarity in the Standard was captured in **miller17**. The outcome of the core issue was to completely remove permission for `alignas` to be applied to enumerations (see **iso18a**). Therefore, conforming implementations will eventually stop accepting the `alignas` specifier on enumerations in the future.

Matching the alignment of another type

The `alignas` specifier also accepts (as an argument) a type identifier. In its alternate form, `alignas(T)` is strictly equivalent to `alignas(alignof(T))`:

```
alignas(int) char c;  // equivalent to alignas(alignof(int)) char c;
```

Use Cases

Creating a sufficiently aligned object buffer

When writing low-level, system-infrastructure code, constructing an object within a raw buffer is sometimes useful. As a minimal example, consider a function that uses a local character buffer to create an object of type `std::complex<long double>` on the program stack using placement `new`:

```
void f()
{
    // ...
    char objectBuffer[sizeof(std::complex<long double>)];  // BAD IDEA
    // ...
    new(objectBuffer) std::complex<long double>(1.0, 0.0);  // Might dump core!
    // ...
}
```

The essential problem with the code above is that `objectBuffer`, being an array of characters (each having an alignment of 1), is itself byte aligned. The compiler is therefore free to place it on any address boundary. On the other hand, `std::complex<long double>` is an aggregate consisting of two `long double` objects and therefore necessarily requires (at least) the same strict alignment (typically 16) as the two `long double` objects it comprises. Previous solutions to this problem involved creating a `union` of the object buffer and some maximally aligned type (e.g., `std::max_align_t`):

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

void f()
{
    // ...

    union {                                              // awkward workaround
        std::max_align_t dummy;  // typedef to maximally aligned type
        char objectBuffer[sizeof(std::complex<long double>)];
    } objectBuffer;

    // ...

    new(&objectBuffer) std::complex<long double>(1.0, 0.0);  // OK

    // ...
```

```
}
```

Using the alternate syntax for `alignas`, we can avoid gratuitous complexity and just state our intentions explicitly:

```
void f()
{
    // ...

    alignas(std::complex<long double>) char objectBuffer[
                              sizeof(std::complex<long double>)];  // GOOD IDEA

    // ...

    new(objectBuffer) std::complex<long double>(1.0, 0.0);  // OK

    // ...
}
```

Ensuring proper alignment for architecture-specific instructions

Architecture-specific instructions or compiler intrinsics might require the data they act on to have a specific alignment. One example of such intrinsics is the *Streaming SIMD Extensions (SSE)*[53] instruction set available on the x86 architecture. SSE instructions operate on groups of four 32-bit single-precision floating-point numbers at a time, which are required to be 16-byte aligned.[54] The `alignas` specifier can be used to create a type satisfying this requirement:

```
struct SSEVector
{
    alignas(16) float d_data[4];
};
```

Each object of the `SSEVector` type above is guaranteed always to be aligned to a 16-byte boundary and can therefore be safely (and conveniently) used with SSE intrinsics:

```
#include <xmmintrin.h>  // __m128 and _mm_XXX functions

void f()
{
    const SSEVector v0 = {0.0f, 1.0f, 2.0f, 3.0f};
    const SSEVector v1 = {10.0f, 10.0f, 10.0f, 10.0f};

    __m128 sseV0 = _mm_load_ps(v0.d_data);
    __m128 sseV1 = _mm_load_ps(v1.d_data);
        // _mm_load_ps requires the given float array to be 16-byte aligned.
        // The data is loaded into a dedicated 128-bit CPU register.
```

[53] **inteliig**, "Technologies: SSE"

[54] "Data must be 16-byte aligned when loading to and storing from the 128-bit XMM registers used by SSE/SSE2/SSE3/SSSE3": see **intel16**, section 4.4.4, "Data Alignment for 128-Bit Data," pp. 4-19–4-20.

```
    __m128 sseResult = _mm_add_ps(sseV0, sseV1);
        // sum two 128-bit registers; typically generates an addps instruction

    SSEVector vResult;
    _mm_store_ps(vResult.d_data, sseResult);
        // Store the result of the sum back into a float array.

    assert(vResult.d_data[0] == 10.0f);
    assert(vResult.d_data[1] == 11.0f);
    assert(vResult.d_data[2] == 12.0f);
    assert(vResult.d_data[3] == 13.0f);
}
```

Avoiding false sharing among distinct objects in a multi-threaded program

In the context of an application where multithreading has been employed to improve performance, seeing a previously single-threaded workflow become even less performant after a parallelization attempt can be surprising (and disheartening). One possible insidious cause of such disappointing results comes from **false sharing** — a situation in which multiple threads unwittingly harm each other's performance while writing to logically independent variables that happen to reside on the same **cache line**; see *Appendix: Cache lines; L1, L2, and L3 cache; pages; and virtual memory* on page 78.

As a simple (purely pedagogical) illustration of the potential performance degradation resulting from **false sharing**, consider a function that spawns separate threads to repeatedly increment (concurrently) logically distinct variables that happen to reside in close proximity on the program stack:

```
#include <thread>  // std::thread

volatile int target = 0;  // updated asynchronously from multiple threads

void incrementJob(int* p);
    // Repeatedly increment *p a large, fixed number of times;
    // periodically write its current value to target.

void f()
{
    int i0 = 0;  // Here, i0 and i1 likely share the same cache line,
    int i1 = 0;  // i.e., byte-aligned memory block on the program stack.

    std::thread t0(&incrementJob, &i0);
    std::thread t1(&incrementJob, &i1);
        // Spawn two parallel jobs incrementing the respective variables.

    t0.join();
    t1.join();
```

```
        // Wait for both jobs to be completed.
}
```

In the simplistic example above, the proximity in memory between `i0` and `i1` can result in their belonging to the same **cache line**, thus leading to **false sharing**. By prepending `alignas(64)` to the declaration of both integers, we ensure that the two variables reside on distinct cache lines:

```
// ...

void f()
{
    alignas(64) int i0 = 0;   // Assuming a cache line on this platform is 64
    alignas(64) int i1 = 0;   // bytes, i0 and i1 will be on separate ones.

    // ...
```

As an empirical demonstration of the effects of **false sharing**, a benchmark program repeatedly calling `f` completed its execution seven times faster on average when compared to the same program without use of `alignas`.[55]

Avoiding false sharing within a single thread-aware object

A real-world scenario where the need for preventing **false sharing** is fundamental occurs in the implementation of high-performance concurrent data structures. As an example, a thread-safe ring buffer might make use of `alignas` to ensure that the indices of the head and tail of the buffer are aligned at the start of a cache line (typically 64, 128, or 256 bytes), thereby preventing them from occupying the same one.

```
class ThreadSafeRingBuffer
{
    alignas(cpuCacheSize) std::atomic<std::size_t> d_head;
    alignas(cpuCacheSize) std::atomic<std::size_t> d_tail;

    // ...
};
```

Not aligning `d_head` and `d_tail` (above) to the CPU cache size might result in poor performance of the `ThreadSafeRingBuffer` because CPU cores that need to access only one of the variables will inadvertently load the other one as well, triggering expensive hardware-level coherency mechanisms between the cores' caches. On the other hand, specifying such substantially stricter alignment on consecutive data members necessarily increases the size

[55] The benchmark program was compiled using Clang 11.0.0 using `-Ofast`, `-march=native`, and `-std=c++11`. The program was then executed on a machine running Windows 10 x64, equipped with an Intel Core i7-9700k CPU (8 cores, 64-byte cache line size). Over the course of multiple runs, the version of the benchmark without `alignas` took 18.5967ms to complete (on average), while the version with `alignas` took 2.45333ms to complete (on average). See [**PRODUCTION: CODE PROVIDED WITH BOOK**] **alignasbenchmark** for the source code of the program.

of the object; see *Potential Pitfalls: Stricter alignment might reduce cache utilization* on page 76.

Potential Pitfalls

Underspecifying alignment is not universally reported

The Standard is clear when it comes to underspecifying alignment[56]:

> The combined effect of all *alignment-specifiers* in a declaration shall not specify an alignment that is less strict than the alignment that would be required for the entity being declared if all *alignment-specifiers* were omitted (including those in other declarations).

The compiler is required to honor the specified value if it is a **fundamental alignment**,[57] so imagining how this would lead to anything other than an ill-formed program is difficult:

```
alignas(4) void* p;             // (1) Error: alignas(4) is below minimum, 8.

struct alignas(2) S { int x; };  // (2) Error: alignas(2) is below minimum, 4.

struct alignas(2) T { };
struct alignas(1) U { T e; };    // (3) Error: alignas(1) is below minimum, 2.
```

Each of the three errors above are reported by Clang, but GCC doesn't issue so much as a warning (let alone the required error) — even in the most pedantic warning mode. Thus, one could write a program, involving statements like those above, that happens to work on one platform (e.g., GCC) but fails to compile on another (e.g., Clang).[58]

Incompatibly specifying alignment is IFNDR

It is permissible to forward declare a user-defined type (UDT) without an `alignas` specifier so long as all defining declarations of the type have either no `alignas` specifier or have the same one. Similarly, if any forward declaration of a user-defined type has an `alignas` specifier, then all defining declarations of the type must have the same specifier and that specifier must be *equivalent to* (not necessarily *the same as*) that in the forward declaration:

```
struct Foo;                 // OK, does not specify an alignment
struct alignas(double) Foo; // OK, must be equivalent to every definition
struct alignas(8) Foo;      // OK, all definitions must be identical.
struct alignas(8) Foo { };  // OK, equivalent to each decl. specifying alignas
```

[56] **cpp11**, section 7.6.2, "Alignment Specifier," paragraph 5, pp. 179

[57] "If the constant expression evaluates to a fundamental alignment, the alignment requirement of the declared entity shall be the specified fundamental alignment": **cpp11**, section 7.6.2, "Alignment Specifier," paragraph 2, item 2, p. 178.

[58] Underspecifying alignment is not reported at all by GCC 10.1, using the `-std=c++11 -Wall -Wextra -Wpedantic` flags. With the same set of options, Clang 10.0 produces a compilation failure. MSVC v19.24 will produce a warning and ignore any alignment less than the minimum one.

```
struct Foo;                 // OK, has no effect
struct alignas(8) Foo;      // OK, has no effect; might warn after definition
```

Specifying an alignment in a forward declaration without specifying an equivalent one in the defining declaration is **ill formed; no diagnostic is required (IFNDR)** if the two declarations appear in distinct translation units:

```
struct alignas(4) Bar;      // OK, forward declaration
struct Bar { };             // error: missing alignas specifier

struct alignas(4) Baz;      // OK, forward declaration
struct alignas(8) Baz { };  // error: non-equivalent alignas specifier
```

Both of the errors above are flagged by Clang, but neither of them is reported by GCC. Note that when the inconsistency occurs across translation units, no mainstream compiler is likely to diagnose the problem:

```
// file1.cpp
struct Bam { char ch; } bam, *p = &bam;

// file2.cpp
struct alignas(int) Bam;  // Error: definition of Bam lacks alignment specifier.
extern Bam* p;            //        (no diagnostic required)
```

Any program incorporating both translation units above is **ill formed, no diagnostic required**.

Stricter alignment might reduce cache utilization

User-defined types having artificially stricter alignments than would naturally occur on the host platform means that fewer of them can fit within any given level of physical cache within the hardware. Types having data members whose alignment is artificially widened tend to be larger and thus suffer the same lost cache utilization. As an alternative to enforcing stricter alignment to avoid **false sharing**, consider organizing a multithreaded program such that tight clusters of repeatedly accessed objects are always acted upon by only a single thread at a time, e.g., using local (arena) memory allocators; see *Appendix: Cache lines; L1, L2, and L3 cache; pages; and virtual memory* on page 78.

See Also

- Section **??**, "**??**" — Safe C++11 feature that inspects the alignment of a given type
- Section 1, "Attributes (Long Title)" — Safe C++11 feature that shows how other attributes (following the conventional attribute notation) are used to annotate source code, improve error diagnostics, and implicitly code generation

Further Reading

None so far

Appendix

Natural Alignment

By default, fundamental, pointer, and enumerated types typically reside on an address boundary that divides the size of the object; we refer to such alignment as **natural alignment**[59]:

```
char   c;  // size 1; alignment 1; boundaries: 0x00, 0x01, 0x02, 0x03, ...
short  s;  // size 2; alignment 2: boundaries: 0x00, 0x02, 0x04, 0x06, ...
int    i;  // size 4; alignment 4; boundaries: 0x00, 0x04, 0x08, 0x0c, ...
float  f;  // size 4; alignment 4; boundaries: 0x00, 0x04, 0x08, 0x0c, ...
double d;  // size 8; alignment 8; boundaries: 0x00, 0x08, 0x10, 0x18, ...
```

For aggregates (including arrays) or user-defined types, the alignment is typically that of the most strictly aligned subelement:

```
struct S0
{
    char a;  // size 1; alignment 1
    char b;  // size 1; alignment 1
    int  c;  // size 4; alignment 4
};           // size 8; alignment 4

struct S1
{
    char a;  // size  1; alignment 1
    int  b;  // size  4; alignment 4
    char c;  // size  1; alignment 1
};           // size 12; alignment 4

struct S2
{
    int  a;  // size 4; alignment 4
    char b;  // size 1; alignment 1
    char c;  // size 1; alignment 1
};           // size 8; alignment 4

struct S3
{
    char a;  // size 1; alignment 1
    char b;  // size 1; alignment 1
};           // size 2; alignment 1

struct S4
{
```

[59]Sizes and alignment shown here are typical but not specifically required by the standard. On some platforms, one can request that all types be **byte aligned**. While such a representation is more compact, entities that span memory boundaries can require multiple fetch operations leading to run times that are typically significantly (sometimes as much as an order of magnitude) slower when run in this "packed" mode.

```
    char a[2];  // size 2; alignment 1
};              // size 2; alignment 1
```

Size and alignment behave similarly with respect to **structural inheritance**:

```
struct D0 : S0
{
    double d;  // size  8; alignment 8
};             // size 16; alignment 8

struct D1 : S1
{
    double d;  // size  8; alignment 8
};             // size 24; alignment 8

struct D2 : S2
{
    int d;  // size  4; alignment 4
};          // size 12; alignment 4

struct D3 : S3
{
    int d;  // size 4; alignment 4
};          // size 8; alignment 4

struct D4 : S4
{
    char d;  // size 1; alignment 1
};           // size 3; alignment 1
```

Finally, virtual functions invariably introduce an implicit virtual-table-pointer member having a size and alignment corresponding to that of a memory address (e.g., 4 or 8) on the host platform:

```
struct S5
{
    virtual ~S5();
};                  // size 8; alignment 8

struct D5 : S5
{
    char d;  // size  1; alignment 1
};           // size 16; alignment 8
```

Cache lines; L1, L2, and L3 cache; pages; and virtual memory

Modern computers are highly complex systems, and a detailed understanding of their intricacies is unnecessary to achieve most of the performance benefits. Still, certain general themes and rough thresholds aid in understanding how to squeeze just a bit more out of

the underlying hardware. In this section, we sketch fundamental concepts that are common to all modern computer hardware; although the precise details will vary, the general ideas remain essentially the same.

In its most basic form, a computer consists of central processing unit (CPU) having internal registers that access main memory (MM). Registers in the CPU (on the order of hundreds of bytes) are among the fastest forms of memory, while main memory (typically many gigabytes) is orders of magnitude slower. An almost universally observed phenomenon is that of **locality of reference**, which suggests that data that resides in close proximity (in the virtual address space) is more likely to be accessed together in rapid succession than more distant data.

To exploit the phenomenon of **locality of reference**, computers introduce the notion of a cache that, while much faster than main memory, is also much smaller. Programs that attempt to amplify **locality of reference** will, in turn, often be rewarded with faster run times. The organization of a cache and, in fact, the number of levels of cache (e.g., L1, L2, L3,...) will vary, but the basic design parameters are, again, more or less the same. A given level of cache will have a certain total size in bytes (invariably an integral power of two). The cache will be segmented into what are called **cache lines** whose size (a smaller power of two) divides that of the cache itself. When the CPU accesses main memory, it first looks to see if that memory is in the cache; if it is, the value is returned quickly (known as a **cache hit**). Otherwise, the cache line(s) containing that data is (are) fetched (from the next higher level of cache or from main memory) and placed into the cache (known as a **cache miss**), possibly ejecting other less recently used ones.[60]

Data residing in distinct cache lines is physically independent and can be written concurrently by multiple threads. Logically unrelated data residing in the same cache line, however, is nonetheless physically coupled; two threads that write to such logically unrelated data will find themselves synchronized by the hardware. Such unexpected and typically undesirable sharing of a cache line by unrelated data acted upon by two concurrent threads is known as **false sharing**. One way of avoiding **false sharing** is to align such data on a cache-line boundary, thus rendering accidental collocation of such data on the same cache line impossible. Another (more broad-based) design approach that avoids lowering cache utilization is to ensure that data acted upon by a given thread is kept physically separate — e.g., through the use of local (arena) memory allocators.[61]

[60] Conceptually, the cache is often thought of as being able to hold any arbitrary subset of the most recently accessed cache lines. This kind of cache is known as **fully associative**. Although it provides the best hit rate, a **fully associative** cache requires the most power along with significant additional chip area to perform the fully parallel lookup. **Direct-mapped** cache associativity is at the other extreme. In direct mapped, each memory location has exactly one location available to it in the cache. If another memory location mapping to that location is needed, the current cache line must be flushed from the cache. Although this approach has the lowest hit rate, lookup times, chip area, and power consumption are all minimized (optimally). Between these two extremes is a continuum that is referred to as **set associative**. A **set associate** cache has more than one (typically 2, 4, or 8; see **solihin15**, section 5.2.1, "Placement Policy," pp. 136–141, and **hruska20**) location in which each memory location in main memory can reside. Note that, even with a relatively small N, as N increases, an N-way **set associative** cache quickly approaches the hit rate of a fully associative cache at greatly reduced collateral cost; for most software-design purposes, any loss in hit rate due to set associativity of a cache can be safely ignored.

[61] **lakos17**, **lakos19**, **lakos22**

Finally, even data that is not currently in cache but resides nearby in MM can benefit from locality. The virtual address space, synonymous with the size of a `void*` (typically 64-bits on modern general-purpose hardware), has historically well exceeded the physical memory available to the CPU. The operating system must therefore maintain a mapping (in main memory) from what is resident in physical memory and what resides in secondary storage (e.g., on disc). In addition, essentially all modern hardware provides a **TLB**[62] that caches the addresses of the most recently accessed physical pages, providing yet another advantage to having the **working set** (i.e., the current set of frequently accessed pages) remain small and densely packed with relevant data.[63] What's more, dense working sets, in addition to facilitating hits for repeat access, increase the likelihood that data that is coresident on a page (or cache line) will be needed soon (i.e., in effect acting as a prefetch).[64] Table 1.0–2 provides a summary of typical physical parameters found in modern computers today.

[62]A translation-lookaside buffer (TLB) is a kind of address-translation cache that is typically part of a chip's memory management unit (MMU). A TLB holds a recently accessed subset of the complete mapping (itself maintained in MM) from virtual memory address to physical ones. A TLB is used to reduce access time when the requisite pages are already resident in memory; its size (e.g., 4K) is capped at the number of bytes of physical memory (e.g., 32Gb) divided by the number of bytes in each physical page (e.g., 8Kb), but could be smaller. Because it resides on chip, is typically an order of magnitude faster (SRAM versus DRAM), and requires only a single lookup (as opposed to two or more when going out to MM), there is an enormous premium on minimizing TLB misses.

[63]Note that memory for handle-body types (e.g., `std::vector` or `std::deque`) and especially node-based containers (e.g., `std::map` and `std::unordered_map`), originally allocated within a single page, can — through deallocation and reallocation (or even move operations) — become scattered across multiple (perhaps many) pages, thus causing what was originally a relatively small **working set** to no longer fit within physical memory. This phenomenon, known as **diffusion** (which is a distinct concept from **fragmentation**), is what typically leads to a substantial runtime performance degradation (due to **thrashing**) in large, long-running programs. Such **diffusion** can be mitigated by judicious use of local arena memory allocators (and deliberate avoidance of **move operations** across disparate localities of frequent memory usage).

[64]We sometimes lightheartedly refer to the beneficial prefetch of unrelated data that is accidentally needed subsequently (e.g., within a single thread) due to high locality within a cache line (or a physical page) as **true sharing**.

Table 1.0–2: Various sizes and access speeds of typical memory for modern computers

Memory Type	Typical Memory Size (Bytes)	Typical Access Times
CPU Registers	512 ... 2048	∼250ps
Cache Line	64 ... 256	NA
L1 Cache	16Kb ... 64Kb	∼1ns
L2 Cache	1Mb ... 2Mb	∼10ns
L3 Cache	8Mb ... 32Mb	∼80ns–120ns
L4 Cache	32Mb ... 128Mb	∼100ns–200ns
Set Associativity	2 ... 64	NA
TL	4 words ... 65536	10ns ... 50ns
Physical Memory Page	512 ... 8192	100ns ... 500ns
Virtual Memory	2^{32} bytes ... 2^{64} bytes	∼10μs–50μs
Solid-State Disc (SSD)	256Gb ... 16Tb	∼25μs–100μs
Mechanical Disc	Huge	∼5ms–10ms
Clock Speed	NA	∼4GHz

The Standard `[[deprecated]]` Attribute

The `[[deprecated]]` attribute discourages the use of a decorated **entity**, typically via the emission 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, in (ideally) all ways, are superior to the original one, providing time for clients to migrate to them (*asynchronously*[65]) before the deprecated one is (in some subsequent release) removed. Although not strictly required, the Standard explicitly encourages[66] 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

[65] A process for ongoing improvement of legacy code bases, sometimes known 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 code base is a closed system, (2) all of the relevant code governed by a single authority, and (3) there is some sort of mechanical way to make the change.

[66] 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"`).

object[67] 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.
}
```

A programmer can (optionally) 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 seen in `g2` (above), redeclaring an **entity** that was previously decorated with `[[deprecated]]` without the attribute does not un-deprecate the entity.

[67] The `[[deprecated]]` attribute can be used portably to decorate other entities: `class`, `struct`, `union`, type alias, variable, data member, function, enumeration, template specialization. Applying `[[deprecated]]` to a specific enumerator or namespace, however, is guaranteed to be supported only since C++17; see **smith14** for more information.

Use Cases

Discouraging use of an obsolete or unsafe entity

Decorating any **entity** with `[[deprecated]]` 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:

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

Although such a simple random number generator can be very 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 very high overhead access to external sources of entropy.[68] 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 BetterRandomGenerator
{
    // ... (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, instead of removing it completely, to use the `[[deprecated]]` attribute to (gently) discourage continued use of `RandomGenerator::nextRandom()`:

```
struct RandomGenerator
{
    [[deprecated("Use BetterRandomGenerator::nextRandom() instead.")]]
    static int nextRandom();
```

[68] The C Standard Library provides `rand`, available in C++ through the `<cstdlib>` header. It has similar issues to our `RandomGenerator::nextRandom` function, and similarly developers are guided to use the facilities provided in the `<random>` header since C++11.

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

By using `[[deprecated]]` as shown above, 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 their code being 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 (eventually) rewrite their applications to consume the new interface.[69]

Potential Pitfalls

Interaction with `-Werror` (e.g., GCC, Clang) or `/WX` (MSVC)

To prevent warnings from being overlooked, the `-Werror` flag (`/WX` on MSVC) is sometimes used, which promotes warnings to errors. Consider the case where a project has been successfully using `-Werror` for years, only to one day face an unexpected compilation failure due to one of the project's dependencies using `[[deprecated]]` as part of their API.

Having the compilation process completely stopped due to use of a deprecated **entity** defeats the purpose of the attribute because users of such **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: The (unsatisfactory) workarounds are to disable (entirely) either `/WX` or deprecation diagnostics (using the `-wd4996` flag).

Furthermore, this interaction between `[[deprecated]]` and `-Werror` 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 along with `-Werror` prevents the release of the code with the `[[deprecated]]` attribute on it. With the default behaviors of compilers and the frequent advice given in practice to use `-Werror` aggressively, this can make any use of `[[deprecated]]` completely unfeasible.

Annoyances

None so far

See Also

None so far

[69] All joking aside, **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.

Further Reading

None so far

Longer Wordier Feature Title

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse non nisi ex. Ut non sollicitudin est. Donec laoreet, risus pretium egestas condimentum, ex eros tempor diam, quis congue nisi metus in tellus. Proin tempor ac lectus nec elementum. Maecenas augue turpis, pellentesque sed eros sit amet, tincidunt pretium nulla. Nunc lacus ligula, ullamcorper a porta eget, fringilla sed orci. Maecenas eget ultricies risus. Donec varius vehicula diam.

Description

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse non nisi ex. Ut non sollicitudin est. Donec laoreet, risus pretium egestas condimentum, ex eros tempor diam, quis congue nisi metus in tellus. Proin tempor ac lectus nec elementum. Maecenas augue turpis, pellentesque sed eros sit amet, tincidunt pretium nulla. Nunc lacus ligula, ullamcorper a porta eget, fringilla sed orci. Maecenas eget ultricies risus. Donec varius vehicula diam.

C++ attribute syntax

Lorem ipsum dolor sit amet, consectetur adipiscing elit.[70] Suspendisse non nisi ex. Ut non sollicitudin est. Donec laoreet, risus pretium egestas condimentum, ex eros tempor diam, quis congue nisi metus in tellus. Proin tempor ac lectus nec elementum. Maecenas augue turpis, pellentesque sed eros sit amet, tincidunt pretium nulla. Nunc lacus ligula, ullamcorper a porta eget, fringilla sed orci. Maecenas eget ultricies risus. Donec varius vehicula diam.

Nested unordered list:

- One
  - Sub
  - Sub
- Two
- Three

Donec laoreet, risus pretium egestas condimentum, ex eros tempor diam, quis congue nisi metus in tellus. Proin tempor ac lectus nec elementum. Maecenas augue turpis, pellentesque sed eros sit amet, tincidunt pretium nulla:

Nested ordered list:

1. One
   (a) Sub

[70] Authors' Note: We will have some footnotes that are authors' notes.

   (b) Sub

2. Two

3. Three

This feature, when used in conjunction with *explicit instantiation definitions*, can significantly improve compilation times for a set of translation units that often instantiate common templates:

Listing 1.0–1: code 1

```
void code()
{

}
```

Listing 1.0–2: code 2

```
void code()
{

}
```

Nullam nibh tortor, finibus ut lectus eu, convallis vehicula libero. Cras maximus ligula nisl, a eleifend mauris venenatis nec. Suspendisse potenti. Maecenas viverra laoreet mauris ut laoreet. Donec non felis risus. Ut faucibus, justo id luctus lobortis, nisi lacus pharetra est, id pretium arcu ligula ac magna. Morbi a nisl sit amet lacus vehicula tempor nec sit amet nibh.

Listing 1.0–3: code 1 with a long and wrapping title

```
void code()
{

}
```

Listing 1.0–4: code 2 with a long and wrapping title

```
void code()
{

}
```

Pellentesque id lorem ac sem ullamcorper semper. Donec imperdiet sapien in nisi faucibus, et tempus lacus hendrerit. Nulla laoreet risus eu tortor ultrices, sed bibendum neque sodales. Ut faucibus ipsum id convallis convallis. Vivamus vehicula rutrum metus in semper. Morbi fringilla ex vel vulputate vehicula. Maecenas ut porttitor massa.

Appendix to the Feature

Suspendisse lorem libero, egestas non semper eleifend, finibus ut purus. Quisque leo nulla, lacinia vel dui et, vestibulum facilisis erat. Vestibulum scelerisque auctor diam, ut fringilla diam elementum non. Vivamus sed mauris lobortis, blandit urna quis, iaculis neque. Phasellus sit amet venenatis sapien, vel vestibulum augue. In pellentesque sit amet enim a rutrum. Fusce nec nulla et nisl faucibus efficitur vel non diam. Nulla lobortis feugiat augue, a lobortis nunc imperdiet eget. Sed in neque ultricies, efficitur mi eu, sollicitudin massa. Vivamus sed dui convallis, mattis sem a, scelerisque augue. Sed non ultricies neque. Praesent tincidunt feugiat lorem, a sagittis dui eleifend ac. Nullam molestie nulla quis risus molestie dignissim. Aenean egestas enim ut tellus vulputate finibus. Integer iaculis sodales gravida.

Table 1.0–3: Testing Table Numbering

Compiler	Compiler-Specific	Standard-Conforming
GCC	`__attribute__((pure))`	`[[gnu::pure]]`
Clang	`__attribute__((no_sanitize))`	`[[clang::no_sanitize]]`
MSVC	`declspec(deprecated)`	`[[deprecated]]`

Donec mattis, ex a ornare auctor, tortor felis pulvinar odio, eu mollis tortor orci id ligula. Vestibulum dignissim magna vitae lectus cursus, ac sollicitudin mi pulvinar. Praesent vulputate lorem lectus, eu pulvinar mi finibus eu. Donec sit amet massa massa. Sed eget molestie quam. Praesent iaculis diam ut nunc condimentum auctor. Morbi porttitor varius ante vel venenatis. Integer ut porttitor elit, id lacinia turpis. Integer id iaculis ante, placerat ornare tortor. Sed et consectetur est, sit amet rhoncus turpis.

Table 1.0–4: Testing Table Numbering

Compiler	Compiler-Specific	Standard-Conforming
GCC	`__attribute__((pure))`	`[[gnu::pure]]`
Clang	`__attribute__((no_sanitize))`	`[[clang::no_sanitize]]`
MSVC	`declspec(deprecated)`	`[[deprecated]]`

Phasellus ac tellus ipsum. Donec ornare id urna a blandit. Pellentesque finibus nulla augue, vitae pretium justo imperdiet ac. Aenean condimentum nibh eget turpis viverra gravida. Nullam diam lectus, egestas id ex quis, blandit facilisis libero. Proin laoreet ante dictum tristique finibus. Vestibulum pretium varius ipsum, sed blandit ex venenatis a. Nam pharetra pretium accumsan. Integer accumsan purus elementum tortor aliquam, commodo semper leo commodo. Nunc finibus varius erat, non hendrerit leo dapibus at. Donec mattis porta ex, eu ornare nibh condimentum id. Sed sit amet erat sit amet urna volutpat ullamcorper. Aenean feugiat eget orci ac feugiat. Nulla facilisi. Cras augue lacus, placerat sit amet vestibulum at, mollis sit amet nisl.

Nam id sem sagittis, placerat justo eget, eleifend justo. Sed ultrices rhoncus quam ut auctor. Nam pellentesque risus orci, a rhoncus arcu lobortis in. Vestibulum tristique nisi sed sollicitudin aliquam. Praesent ut odio sapien. Ut pretium sollicitudin nisi sit amet blandit. Cras ut eleifend elit. Aliquam vel tincidunt lacus, non sodales justo. Sed euismod sapien non diam euismod, sit amet vulputate nisi rhoncus. Vestibulum leo diam, dapibus suscipit blandit molestie, suscipit eget ante. Donec tristique rhoncus purus, nec pellentesque lorem volutpat non. Quisque purus dui, egestas ut ultricies sed, pharetra sit amet turpis. Praesent ut ligula porttitor, convallis velit sit amet, rhoncus urna.

Chapter 2

Conditionally Safe Features

auto

placeholder text.....

Braced Initialization

placeholder text.....

Rvalue References

placeholder text.....

Default Member Initializers

placeholder text.....

constexpr Variables

placeholder text.....

`constexpr` Functions

placeholder text.....

Variadic Templates

placeholder text.....

Lambda Expressions

placeholder text.....

Forwarding References

placeholder text.....

Generic Lambdas

placeholder text.....

Longer Wordier Title for Test

Section text; just to test where/how the C++ change (from C++11 to C++14) in the RH must be placed in relation to sectioning commands such that the correct one displays

Table 2.0–1: Testing Table Numbering

Compiler	Compiler-Specific	Standard-Conforming
GCC	`__attribute__((pure))`	`[[gnu::pure]]`
Clang	`__attribute__((no_sanitize))`	`[[clang::no_sanitize]]`
MSVC	`declspec(deprecated)`	`[[deprecated]]`

Donec mattis, ex a ornare auctor, tortor felis pulvinar odio, eu mollis tortor orci id ligula. Vestibulum dignissim magna vitae lectus cursus, ac sollicitudin mi pulvinar. Praesent vulputate lorem lectus, eu pulvinar mi finibus eu. Donec sit amet massa massa. Sed eget molestie quam. Praesent iaculis diam ut nunc condimentum auctor. Morbi porttitor varius ante vel venenatis. Integer ut porttitor elit, id lacinia turpis. Integer id iaculis ante, placerat ornare tortor. Sed et consectetur est, sit amet rhoncus turpis.

Table 2.0–2: Testing Table Numbering

Compiler	Compiler-Specific	Standard-Conforming
GCC	`__attribute__((pure))`	`[[gnu::pure]]`
Clang	`__attribute__((no_sanitize))`	`[[clang::no_sanitize]]`
MSVC	`declspec(deprecated)`	`[[deprecated]]`

This feature, when used in conjunction with *explicit instantiation definitions*, can significantly improve compilation times for a set of translation units that often instantiate common templates:

Listing 2.0–1: code 1

```
void code()
{

}
```

Listing 2.0–2: code 2

```
void code()
{

}
```

Nullam nibh tortor, finibus ut lectus eu, convallis vehicula libero. Cras maximus ligula nisl, a eleifend mauris venenatis nec. Suspendisse potenti. Maecenas viverra laoreet mauris ut laoreet. Donec non felis risus. Ut faucibus, justo id luctus lobortis, nisi lacus pharetra est, id pretium arcu ligula ac magna. Morbi a nisl sit amet lacus vehicula tempor nec sit amet nibh.

Listing 2.0–3: code 1 with a long and wrapping title

```
void code()
{

}
```

Listing 2.0–4: code 2 with a long and wrapping title

```
void code()
{

}
```

Pellentesque id lorem ac sem ullamcorper semper. Donec imperdiet sapien in nisi faucibus, et tempus lacus hendrerit. Nulla laoreet risus eu tortor ultrices, sed bibendum neque sodales. Ut faucibus ipsum id convallis convallis. Vivamus vehicula rutrum metus in semper. Morbi fringilla ex vel vulputate vehicula. Maecenas ut porttitor massa.

Chapter 3

Unsafe Features

The [[carries_dependency]] Attribute

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin in consectetur ante. Proin est risus, iaculis vitae nisl finibus, gravida scelerisque quam. Nullam urna mauris, eleifend sed arcu vitae, laoreet gravida elit. Maecenas varius dolor id lectus elementum, sed posuere nisi malesuada. Etiam ornare egestas commodo. Pellentesque sed tortor in sapien pellentesque luctus. Ut tempor nisl quis ex convallis scelerisque eu a lacus. Mauris quis feugiat urna. Cras id ante sed metus gravida laoreet quis in nisi. Sed suscipit ac massa sit amet laoreet. Maecenas sapien urna, tincidunt ac lectus et, ultricies aliquam lectus. Nam vel dictum ligula, et condimentum risus. Sed convallis ullamcorper massa sed rutrum. Donec dignissim facilisis vulputate. Duis sit amet facilisis odio.

Sed volutpat magna turpis, in mollis purus scelerisque nec. Proin rhoncus magna quis porttitor convallis. Aliquam finibus sit amet eros in suscipit. Integer tristique faucibus placerat. Etiam finibus commodo tortor in dictum. Morbi vel eros enim. Duis pellentesque varius sapien, eget porttitor augue posuere nec. Nulla quis cursus quam, eu molestie turpis. Aenean non condimentum augue. Sed ac ornare ligula.

Ut non tempus tellus. Aenean sit amet purus eu sapien accumsan viverra a nec tellus. Suspendisse tincidunt eleifend fringilla. Nulla dapibus molestie nisi, id placerat eros. Pellentesque ultrices sapien risus, vulputate dignissim purus tempor a. Phasellus blandit laoreet orci, at accumsan ligula. Morbi fringilla auctor suscipit. Aenean et velit a ante lobortis mollis eu id mauris. Sed interdum dapibus lectus et scelerisque. Duis non lacus justo. Suspendisse dui diam, efficitur at orci non, commodo viverra felis. Sed at tempor tellus. Morbi viverra arcu neque, nec viverra lorem consequat ac. Vestibulum at blandit elit.

Praesent dapibus libero ullamcorper, consectetur mi vel, hendrerit quam. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Mauris cursus lacinia orci sit amet rutrum. Aenean non pharetra urna. Nunc pulvinar nunc eget finibus porta. Sed dignissim nunc arcu, non porttitor enim euismod ac. Morbi placerat risus in feugiat bibendum. Curabitur in feugiat augue. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia curae; Pellentesque congue justo sed ante congue egestas. Nullam dolor turpis, vehicula id elit sed, iaculis pellentesque dui. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Morbi sed est eros.

Cras vel cursus ante, quis feugiat tortor. Quisque pulvinar vel justo et fringilla. Etiam euismod vel dolor nec consequat. Quisque hendrerit elit tortor, et maximus tellus vulputate at. Pellentesque rhoncus tempor laoreet. Mauris eget tristique nisl. Aliquam erat nibh, tincidunt sit amet arcu in, iaculis efficitur felis. Nulla eget orci eget nibh vulputate sagittis a sit amet ex. Duis quis dictum libero, sit amet facilisis magna. Duis sed sollicitudin quam, eget placerat nibh. Duis a vehicula massa, vel ultricies leo.

Phasellus et bibendum lacus. Pellentesque elementum lectus sed elit auctor, eget condimentum nisi accumsan. Integer tellus elit, pellentesque eget tincidunt vitae, varius et dolor. Curabitur vel lorem mi. Maecenas sed mauris ultricies, ornare purus non, mattis felis. Quisque porta sapien id nibh semper finibus. Fusce pharetra nunc a cursus egestas. Aliquam viverra metus libero. Quisque ultrices metus non neque fringilla, sit amet ultricies sem varius. Ae-

nean sagittis nisi non metus blandit, quis laoreet dolor vestibulum. Proin ultrices nulla quis vehicula porta. Maecenas ut magna vitae lacus suscipit imperdiet sed quis nulla.

Fusce tincidunt vel purus non ultrices. Donec in vulputate lorem. Vestibulum sed mauris ac tellus rutrum tempor. Mauris enim massa, tincidunt vel ullamcorper ac, vulputate a nunc. Duis accumsan magna in lorem euismod, at volutpat lorem hendrerit. Curabitur in mi sit amet purus viverra volutpat id eu ligula. Aliquam nec sodales nibh. Maecenas eu porta tellus, nec tincidunt quam. Sed fringilla nunc orci, at auctor mi feugiat sed. Integer rhoncus viverra maximus. Suspendisse iaculis tincidunt nibh vitae vestibulum. Mauris ut ex fringilla, fermentum diam vel, dapibus elit.

Suspendisse justo enim, rhoncus sit amet lectus a, efficitur egestas turpis. Donec pulvinar ipsum lorem. Donec aliquam sem nec enim facilisis mollis. Donec vel scelerisque nisl, eu euismod est. Maecenas ultrices, leo at ultricies lacinia, diam felis accumsan leo, a lobortis quam arcu vel lorem. Proin ut purus vitae nulla tincidunt iaculis. Pellentesque vitae nunc mattis, consectetur ligula vitae, pharetra nulla. Ut viverra tortor aliquet ligula accumsan aliquet. Duis lacus odio, euismod porttitor egestas quis, pulvinar non dui. Aliquam eget risus tempor, pellentesque enim vel, sodales tellus.

Morbi ut justo metus. Vivamus fringilla nisl nec cursus fermentum. Nunc massa neque, aliquam ac ultrices et, pharetra nec libero. Suspendisse at elementum ligula. Nullam vehicula urna nec sapien vestibulum dapibus. Nunc lorem sapien, mollis nec velit ullamcorper, suscipit accumsan orci. Pellentesque molestie mauris ut elit fringilla ullamcorper. Sed in ligula sit amet neque consectetur blandit in nec leo. Aliquam dolor nulla, semper quis porta in, tincidunt id massa. Nulla molestie turpis dui, non condimentum dolor pretium tristique. Integer suscipit gravida urna, a varius nulla.

Donec dapibus nulla at euismod aliquam. Suspendisse ultricies, dolor id elementum lobortis, lectus diam tristique neque, ut euismod nulla felis vitae massa. Pellentesque consequat nisi ut augue rutrum gravida. Aliquam in congue neque. Nulla tincidunt, quam et convallis varius, enim tellus iaculis nisl, et lobortis sem leo vitae eros. Quisque ac imperdiet leo, et vulputate nulla. Nulla vitae urna eget erat efficitur porttitor vel sit amet dui. Proin pharetra metus ac ornare dignissim. Praesent enim orci, iaculis id lectus vel, consequat fermentum arcu. Duis quam metus, tristique ac ante eu, porttitor lobortis mi.

Function Return Type Deduction

placeholder text.....

Chapter 4

Parting Thoughts

Testing Section

Testing Another Section

Bibliography

ballman
Aaron Ballman. "Rule 03. Integers (INT)," *SEI CERT C++ Coding Standard* (Pittsburgh, PA: Carnegie Mellon University Software Engineering Institute)
https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046333

calabrese20
Matt Calabrese and Ryan McDougall. "`any_invocable`." Technical Report P0288R6. (Geneva, Switzerland: International Standards Organization, 2020)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0288r6.html

cpp11
Information Technology — Programming Languages — C++ (Geneva, Switzerland: International Standards Organization, 2011)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf

cppreferencea
"std::all_of, std::any_of, std::none_of," *C++ Algorithm Library*, cppreference.com
https://en.cppreference.com/w/cpp/algorithm/all_any_none_of

cppreferenceb
"std::unordered_map," *C++ Containers Library*, cppreference.com
https://en.cppreference.com/w/cpp/container/unordered_map

freesoftwarefdn20
Using the GNU Compiler Collection (GCC) (Boston, MA: Free Software Foundation, Inc., 2020)
https://gcc.gnu.org/onlinedocs/gcc/
RETAIN IN CASE WE NEED THIS; USED IN ATTRIBUTES FEATURE section 6.33.1, "Common Function Attributes,"
https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#Common-Function-Attribut

hinnant02
Howard Hinnant. *A Proposal to Add Move Semantics Support to the C++ Language* (Geneva, Switzerland: International Standards Organization, 2002)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1377.htm

hinnant14

Bibliography

Howard Hinnant. "Everything You Ever Wanted To Know About Move Semantics (and Then Some)" *Conference of the ACCU*, Bristol, England, April 8–12, 2014
https://accu.org/content/conf2014/Howard_Hinnant_Accu_2014.pdf

hinnant16
Howard Hinnant. "Everything You Ever Wanted to Know About Move Semantics" *Bloomberg Engineering Distinguished Speaker Series*, New York, NY, 2016
https://www.youtube.com/watch?v=vLinb2fgkHk&t=28s

hruska20
Joel Hruska. "How L1 and L2 CPU Caches Work, and Why They're an Essential Part of Modern Chips," *Extreme Tech*. April 14, 2020.
https://www.extremetech.com/extreme/188776-how-l1-and-l2-cpu-caches-work-and-why-theyre-an-essential-part-of-modern-chips

ieee19
IEEE Standard for Floating-Point Arithmetic (New York, NY: Institute of Electrical and Electronics Engineers, Inc., 2019)

intel16
Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Publication Number 248966-033. (Santa Clara, CA: Intel Corporation, 2016)
https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf

inteliig
Intel. *The Intel Intrinsics Guide* (Santa Clara, CA: Intel Corporation)
https://software.intel.com/sites/landingpage/IntrinsicsGuide/#

iso14
ISO/IEC 14882:2014 Programming Language C++ (Geneva, Switzerland: International Standards Organization, 2014)
http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2013/n3797.pdf

iso18a
C++ Standard Core Language Active Issues, Revision 100 (Geneva, Switzerland: International Standards Organization, 2018)
http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html

iso18b
ISO/IEC 9899:2018 XXXXXXXXXXXXXXXXX (Geneva, Switzerland: International Standards Organization, 2018)
URL

iso20a
ISO/IEC P2156R1:2020 Allow Duplicate Attributes (Geneva, Switzerland: International Standards Organization, 2020
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2156r1.pdf

iso20b
ISO/IEC 14882:2020 XXXXXXXXXXXXXXXXX (Geneva, Switzerland: International Standards Organization, 2020)
URL

kahan97
W. Kahan, *Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic* (Berkeley, CA: Electrical Engineering and Computer Science Department, University of California, 1997)
https://people.eecs.berkeley.edu/ wkahan/ieee754status/IEEE754.PDF

lakos16
John Lakos, Jeffrey Mendelsohn, Alisdair Meredith, and Nathan Myers, "On Quantifying Memory-Allocation Strategies (Revision 2)," P0089R1, February 12, 2016
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0089r1.pdf

lakos17
John Lakos, "Local ('Arena') Memory Allocators - Part II," *Meeting C++*, 2017
https://www.youtube.com/watch?v=fN7nVzbRiEk

lakos19
John Lakos, "Value Proposition: Allocator-Aware (AA) Software," *C++Now*, 2019
https://www.youtube.com/watch?v=dDR93TfacHc

lakos20
John Lakos, *Large-Scale C++ — Volume I: Process and Architecture* (Reading, MA: Addison-Wesley, 2019)

lakos22
John Lakos and Joshua Berne, *C++ Allocators for the Working Programmer* (Boston, MA: Addison-Wesley, forthcoming)

lakos96
John Lakos, *Large-Scale C++ Software Design* (Reading, MA: Addison-Wesley, 1996)

meyers97
Scott Meyers. *Effective C++*, second ed. (Boston, MA: Addison Wesley, 1997)

microsoft
Microsoft. *Guidelines Support Library.*
https://github.com/Microsoft/GSL

microsoftC26481
Microsoft. "C26481 NO_POINTER_ARITHMETIC," *Guidelines Support Library*, 2020.
https://docs.microsoft.com/en-us/cpp/code-quality/c26481?view=vs-2019

miller13

Bibliography

Mike Miller. "Issue 1655: Line Endings in Raw String Literals," 26 April 2013, included within *C++ Standard Core Language Active Issues, Revision 100* (Geneva, Switzerland: International Standards Organization, 2018)
http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html#1655

miller17
Mike Miller. "Issue 2354: Extended Alignment and Object Representation," 05 September 2017, included within *Core Language Working Group Tentatively Ready Issues for the February, 2019 (Kona) Meeting* (Geneva, Switzerland: International Standards Organization, 2019)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1359r0.html#2354

meyers05
Scott Meyers. *Effective C++* (Boston, MA: Addison-Wesley, 2005)

xxxxxxxxxx
p2156r0
https://wg21.link/p2156r0

pusz20
Mateusz Pusz.*Enable variable template template Parameters*, Technical Report P2008R0 (Geneva, Switzerland: C++ Standards Committee Working Group, 2005)

solihin15
Yan Solihin. *Fundamentals of Parallel Multi-Core Architecture* (Boca Raton, FL: CRC Press, 2015)

seacord13
Robert C. Seacord. *Secure Coding in C and C++*, 2nd ed. (Boston, MA: Addison Wesley, 2013)

sharpe13
Chris Sharpe. "Contextually converted to bool," *Chris's C++ Thoughts* (blog). (Blogspot, July 28, 2013)
http://chris-sharpe.blogspot.com/2013/07/contextually-converted-to-bool.html#:~:text=Certain%20language%20constructs%20require%20that,temporary%20variable%20t%20(8.5)

smith14
Richard Smith. *Attributes for namespaces and enumerators*, Technical Report N4196 (Geneva, Switzerland: C++ Standards Committee Working Group, 2015)

stevens93
W. Richard Stevens. *Advanced Programming in the UNIX Environment* (Boston, MA: Addison-Wesley, 1993)

stroustrup13
Bjarne Stroustrup. *The C++ Programming Language*, 4th ed. (Boston, MA: Addison-Wesley, 2013)

stroustrup20
Bjarne Stroustrup and Herb Sutter, Eds. *C++ Core Guidelines* (**CITY**, WA: Standard C++ Foundation), 2020
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

vandevoorde05
Daveed Vandevoorde. *Right Angle Brackets*, Technical Report N1757, Revision 2, (Geneva, Switzerland: C++ Standards Committee Working Group, 2005)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1757.html

wight
Hyrum Wight. *Hyrum's Law.*
https://www.hyrumslaw.com/

Glossary

access modifier

aggregate

aggregate initialization

algebra

alias template

alignment

alignment requirements

array type

automatic variables

basic source character set

benchmark test

Glossary

boilerplate code

brace elision

bytes

C-style functions

cache hit

cache line

cache miss

callable object

Class member access expression
TODO (include any expression that is used to refer to a class member, such as `object.member`, `object->member`, `object.*member`.)

closures

Compile-time dispatch
TODO

complete type

component

Concepts

conditionally supported

Constant expression
An expression that can be evaluated at compile time. Mention `constexpr` and state that `const` variables that are initialized from a compile-time constants are themselves required to be compile-time constants. New info to me in June 2020, worthwhile to have.

contextual convertibility to `bool`

Contextual keyword
A *contextual keyword* is a special identifier that acts like a *keyword* when used in particular contexts. `override` is an example as it can be used as a regular identifier outside of member-function declarators.

continuous refactoring

contract

conventional string literals

converting constructors

conversion operators

copy initialization

Copy semantics
TODO

Glossary

data member

Declaration
TODO

declared interface

Declared type
TODO The type of the *entity* named by the given expression.

delegating constructor

deleted function

diffusion

direct initialization

direct mapped

Entity
TODO

excess-*n*

Expression SFINAE
TODO

explicit

`explicit` **specifier**

extended alignment

false sharing

fragmentation

full specialization

fully associative

fully constructed
fully constructed

function object

fundamental alignment

fundamental integral types

Fundamental type
TODO

garbage value

golden output

Hiding
Function-name **hiding** occurs when a member function in a derived class has the same

name as one in the base class, but it is not overriding it due to a difference in the function signature or because the member function in the base class is not `virtual`. The hidden member function will **not** participate in dynamic dispatch; the member function of the base class will be invoked instead when invoked via a pointer or reference to the base class . The same code would have invoked the derived class's implementation had the member function of the base class had been **overridden** rather than **hidden**.

higher-order function

Id-expression
TODO are most commonly **Identifiers**; other forms include overloaded operator names (in function notation), names of user-defined-conversion or literal operators, and destructor names, and ??template names followed by their argument lists??.

ill formed
TODO ([`temp.res`]p8)

ill formed, no diagnostic required (IFNDR)

imperative

implementation-defined

incomplete type

instantiation time

insulate

insulating

Insulation

TODO

Integer literal
TODO

integral constant expression

integral promotion

integral type

invocable

lambda expression

lambda-introducer (adj)

linkage

list initialization

literal type

locality of reference

lvalue

lvalue reference

Glossary

mantissa

maximal alignment

maximally aligned

member initialization lists

member initializer list

metafunction

monotonic allocator

move operations

Move semantics
TODO

natural alignment

naturally aligned

new **handler**

nibbles

nonprimitive functionality

non-trivial constructor

non-trivial special member function
TODO

null address

ODR-used

Overriding
TODO

parameter pack

partial class template specialization

partially constructed

perfectly forwarded

Placement
TODO

placement new
TODO

POD type
TODO

pointer to member

Glossary

polymorphic memory resource

precondition

predicate function

prvalue

RAII
"Resource Acquisition is Initialization"

Range
TODO

raw string literals

Redundant check
TODO

Reference type
TODO

regular type

rvalue

safe-bool-idiom

set associative

SFINAE
TODO

shadowed

side effects

Signature
TODO

signed integer overflow

Special member function
TODO

standard conversion

standard-layout types

static data space

`std::unique_ptr`

String literal
TODO

Structural inheritance
TODO

Sum type
Abstract data type allowing the representation of one of multiple possible alternative types. Each alternative has its own type (and state), and only one alternative can be "active" at

any given point in time. Sum types automatically keep track of which choice is "active," and properly implement value-sematic special member functions (even for non-trivial types). They can be implemented efficiently as a C++ `class` using a C++ `union` and a separate (integral) discriminator. This sort of implementation is commonly referred to as a discriminating (or "tagged") union.

synthetization

template-head

Template instantiation time
TODO

Template instantiation
TODO

template template parameter

test driver

thrashing

thread pool

TLB

trivial copy constructor

trivial operation

trivially copyable

Trivial type
TODO

type alias

typedef

type expression

type inference

type list

Type trait
TODO

UDT

Undefined behavior
TODO

underlying type

user-defined type

user-provided special member function

value

Glossary

value category

value constructor

value-semantic

variable

variadic pack

vocabulary type

well formed

working set

Index

Index