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

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

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

For questions about sales outside the United States, please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

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

Foreword **ix**

Preface **xi**

Acknowledgements **xv**

About the Authors **xvii**

Chapter 0 Introduction **1**
0.1 Scope for the First Edition 1
0.2 What Makes This Book Different 2
0.2.1 The *EMC++S* Manifesto 3
0.2.2 What Do We Mean by *Safely*? 4
0.3 Modern C++ Language Feature Catalog 6
0.3.1 Organization 6
0.4 How To Use This Book 7

Chapter 1 Safe Features **9**
1.1 C++11 9
1.1.1 Attributes 9
1.1.2 Binary Literals 18
1.1.3 Consecutive Right Angle Brackets 22
1.1.4 `decltype` 24
1.1.5 Deleted Functions 29
1.1.6 `override` 33
1.1.7 Compile-Time Assertions (`static_assert`) 36
1.1.8 Trailing Function Return Types 43
1.1.9 Unrestricted Unions 47
1.2 C++14 54
1.2.1 Feature Title 54

Chapter 2 Conditionally Safe Features **57**

Chapter 3 Unsafe Features **59**

Chapter 4 Parting Thoughts **61**

Bibliography **63**

Glossary **65**

INDEX **67**

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.

Acknowledgements

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. You will quickly discover that this book differs from many other C++ books. *Embracing Modern C++ Safely* is a *reference book* for experienced software engineering practitioners working in large organizations and developing complex systems at scale.

- We are practicing software engineering professionals employed by a global software-development enterprise. As senior developers with decades of real-world experience, we are primarily writing for software developers, team leads, and managers working in similar environments.

- We deliberately focus on the productive value that a given language feature affords (or fails to afford), particularly when the systems and the organization that employs the developers writing them are considered at scale. We do not focus on the aesthetics of language features that could hurt the "bottom line" when applied at scale in a large organization.

- Because we focus on only the features of modern C++ that have been part of the Standard and have been actively used for at least five years, we are able to provide you with a thorough and comprehensive treatment based on practical experience and worthy of your limited professional development time.

- By focusing on what is objectively true and relevant to making wise economic and design decisions — with an understanding of the tradeoffs that will arise in real-world practice — we steer clear of subjective opinions and recommendations.

- Finally, this book — a thorough catalog of consequential information written by expert C++ programmers experienced and adept at using certain (early) modern language features in practice — is explicitly and intentionally designed *not* as a tutorial of the latest features for beginners, but as a guide for senior C++ programmers less familiar with these specific features.

0.1 Scope for the First Edition

Given the vastness of C++'s already voluminous and rapidly growing standardized libraries, we have elected 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, exhibit, dissect, and elucidate some, but not all, of the modern language features introduced into the C++ Standard since 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 also decided to confine our attention to just the subset of C++ language features introduced in C++11 and C++14, rather than attempt to cover all the useful features available in C++98/03. Since we can assume with virtual certainty that most of you are quite familiar with essentially all the basic and important special-purpose features of classical C++, we made the tough decision to focus exclusively on what you need to know today: how to safely incorporate C++11/14 language features into a predominately C++98/03 code base.

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[1] as a concise, practical treatment of many important and useful C++98/03 features.

0.2 What Makes This Book Different

Popular books and other material on modern C++ language features typically teach what their authors and/or the proponents of a given feature perceive to be good style and best practices, based largely on their own subjective experiences. These recommendations, in turn, are typically influenced heavily by (and sometimes limited to) the application domain with which the authors are familiar. The factual basis and objective reasoning upon which these recommendations are based can sometimes be tenuous if not entirely unsubstantiated, with the consequences at scale routinely omitted since the authors often have no such experience to share. Such books may be useful for beginners but are far less so for expert developers who are accustomed to — and rewarded for — drawing their own thoughtful conclusions based on objectively verifiable information.

In this atypical book, we instead focus exclusively on elucidating such truths, leaving the final analysis and its application to the reader. The fact-based, data-driven, objective approach — employed throughout this book — ensures that your understanding of what modern C++ has to offer is not skewed by our subjective opinions or domain-specific requirements, thereby facilitating choices more appropriate to your context. Hence, this book is — by design — explicitly *not* a C++ style or coding-standards guide; it would, however, provide valuable input to any development organization seeking to author one.

What's more, we examine modern C++ features through the lens of a large, for-profit company developing and using software in a real-world economy. In addition to showing

[1] **meyers97**

you how best to 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 commercial software-development organization. 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. The outcome of this analysis is our signature categorization of features in terms of *safety*, namely *safe*, *conditionally safe*, or *unsafe* features. (We'll explain what we mean by *safety* and by each of our categories later in this chapter.)

Finally, the information contained in *EMC++S* is the result of our analysis of millions of human hours of using C++11 and C++14 in the development of large-scale commercial software systems, combined with the wisdom and our expertise as senior software engineers actively working with modern C++ in the industry and participating in its continuous improvement on the Standard C++ ISO committee (Working Group 21, i.e., WG21).

0.2.1 The *EMC++S* Manifesto

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 usages and potential pitfalls of modern C++ features. The content presented is based on objectively verifiable facts; we explicitly avoid subjective opinion such as the relative merits of design tradeoffs. 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: While the analysis of each feature is itself completely objective, its chapter classification — indicting the relative *safety* of its quotidian use in a typical, large, corporate software-development environment — reflects our combined opinions, backed by decades of real-world, hands-on experience developing large-scale C++ software systems in various contexts.

Elucidation (Not Prescription)

We deliberately avoid prescribing any canned solutions to address specific feature pitfalls, and instead merely describe and characterize such concerns in sufficient detail to empower 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.

Brevity (Not Verbosity)

EMC++S is neither designed nor intended to be an introduction to modern C++. It is a handy reference for expert C++ programmers who have at least a passing knowledge of the 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 and sparing 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.[2]

Real-World (Not Contrived) Examples

Our goal is that the examples in this book pull their weight in multiple ways. The primary purpose of examples in our book, unlike many books on C++ programing, is to illustrate productive use of the feature as it might occur *in practice* rather than in a contrived use case that merely exercises seldom used aspects of the feature. Hence, many of our examples are based on simplified code fragments extracted from real-world code bases. While we typically change the identifier names to be more appropriate to the shortened example, rather than the business process that led to the example, we keep the code structure as close as possible to the original real-world counterparts from which we derive our experience.

Large-Scale (Not "Toy") Programs

By scale, we are simultaneously capturing two distinct aspect of size: (1) the 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 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.[3]

What's more, powerful new language features that are handled perfectly well by a few expert programmers working together "in a garage" on a prototype for their new start-up simply 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 (see the next section), we do so with mindfulness that any given feature might be used (or misused) in very large programs and by a very large number of programmers having a wide range of knowledge, skill, and ability.

0.2.2 What Do We Mean by *Safely*?

We, the authors of this book, are each members of the ISO standards committee, and we would be remiss — and downright negligent — were we to allow any feature of the C++ language to be standardized if that feature would be other than reliably safe when used as intended.[4] Still, we have chosen the word *safely* as the moniker for the signature aspect of our book, and by *safely* we indicate a comparatively low risk-to-reward ratio for using a given feature widely in a large-scale development environment. In this way, we have contextualized (hijacked) the meaning of the term *safely* to apply to a real-world economy in which everything has a cost, including the risk and added maintenance burden associated

[2] **stroustrup13**

[3] **lakos96**, **lakos20**

[4] Unfortunately, such features do exist and have since C++ was first standardized in 1998. A specific example is local (per-object) memory allocation. An aggressive effort (by us and others) has been and continues to be underway to ameliorate this specific failing of C++; see **lakos22**.

with widespread use of a new feature in an older code base that is maintained by developers that might not be especially 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:

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

Bottom line: In this book, the degree of *safety* for a given feature is the relative likelihood that widespread use of that feature will have no adverse effect on a large software company's bottom line.

A *Safe* Feature

Some of the new features of modern C++ add considerable value, are easy to use, and are decidedly hard to (unintentionally) misuse; 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 unflappable C++ features as *safe*.

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.

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 defects and deficiencies, requiring explicit training and extra care to circumnavigate their pitfalls.

We consider *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 some 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.

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 would be counterproductive in a typical large-scale software-development enterprise.

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[5]) reuse, which is critically important to the success of a large organization.[6]

0.3 Modern C++ Language 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.

0.3.1 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*	Catalog of
Chapter 2: *Conditionally Safe Features*	Language
Chapter 3: *Unsafe Features*	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 out of scope for this book.

Each feature resides in its own subsection, rendered in a canonical format:

- Brief description of intended use and purpose
- Real-world motivating example(s)
- Objective analysis
  - Elucidation of less-than-obvious properties
  - Potential risks and undesirable consequences
  - Known feature deficiencies (if any)
  - Workarounds (as needed)

[5] **lakos20**, section 0.4, pp. 20–28
[6] **lakos20**, section 0.9, pp. 86–97

- Cross-references to related features
- External references for 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 features is sought.

0.4 How To Use This Book

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

1. **Read the entire book from front to back.** If you are an expert developer, 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.** If you are a less sure-footed developer, understanding and applying first the *safe* features of Chapter 1, followed in time by the *conditionally safe* features of Chapter 2, will allow you to grow 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.** 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 nonetheless provide timely access to all relevant details of whichever feature is of immediate interest.

The knowledge imbued into this book — irrespective of how it is consumed — can be valuable in many important ways. In addition to helping you become a more knowledgeable and therefore *safer* developer, this book aims to elucidate (to developers and managers alike) which features demand more training, attention to detail, experience, peer review, and so on. 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, after all, we all are). Finally, any C++ software development organization that adopts this book wholesale will be taking the first steps toward levering modern C++ in a way that maximizes reward while minimizing risks, i.e., by embracing modern C++ *safely*.

Chapter 1

Safe Features

Intro text should be here.

1.1 C++11

1.1.1 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 not affect the semantics[1] of a well-written program. Customized annotations targeted at external (e.g., *static-analysis*) tools[2] might benefit 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 token sequence (fails on GCC <= 9.2)
```

[1]By *semantics* here we typically mean any observable behavior apart from runtime performance. Generally, ignoring an attribute is often a valid (and safe) choice for a compiler to make. There are, however, cases where 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*, below).

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

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*, below).

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, compilers do not typically support anything resembling arbitrary placement of attributes (see *Probing where attributes are permitted in the host platform's C++ grammar*, below) outside of a *declaration statement*. In some cases, the entity to which an unrecognized attribute appertains 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 a 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 }; };
num [[attribute_name]] { SUCCESS, FAIL } result;
```

[5]In the future, it is possible that redundancy of standard attributes will not be an error anymore; see **p2156r0** (https://wg21.link/p2156r0).

```
[[a1]][[a1]] int x [[a1]][[a1]] &;    // 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`, that pertains only to functions, would be valid syntactically but not semantically were it placed so as to annotate any other kind 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, there was no standardized syntax to support conveying such externally sourced information and so non-portable 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 [^attributes_prior_to_cpp17) non-fatal diagnostic.

Table 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*, below).

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

The absolute 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, but see *Potential Pitfalls*: *Not every syntactic location is viable for an attribute*, below.

[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 behave differently depending on particular platform.

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 `[[gnu::warn_unused_result]]` GCC-specific attribute[7] can be used to inform the compiler (and developers) that a function's return value should not be ignored[8]:

```
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 non-zero value
        // otherwise.

    void bind(int port);
        // The behavior is undefined unless start has been previously called
        // and returned a non-zero return code.
};
```

Such annotation of the client-facing declaration can prevent defects caused by a client's forgetting to inspect the result of a function:[9]

```
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]
```

Hinting at better optimizations Some annotations can affect compiler optimizations leading to more efficient or smaller binaries. As an 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)
```

[7]For compatibility with `g++`, `clang++` supports `[[gnu::warn_unused_result]]` as well.

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

[9]Because the `[[gnu::warn_unused_result]]` attribute can in no way 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, however, and best practice might argue in favor of consistency regardless.

```
{
    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 runtime, 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 (beside runtime performance), there are cases where an attribute 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 below) 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[10] of the program is unaltered by calling this function):

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

The `vectorLerp` function (below) performs Linear intERPolation (LERP) between two bidimensional vectors. The body of this function comprises two invocations to the `linear-Interpolation` 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));
}
```

[10]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]]` 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 proper use of `[[gnu::cost]]` or `[[gnu::pure]]`.

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 `linearInterpolation` fails to live up to this promise, however, the compiler will not be able to help us and a runtime defect will be the likely result[11]; see *Potential Pitfalls*, below.

Using attributes to control external static analysis Since unknown attributes do not prevent a well-formed program from compiling, external static-analysis tools can define their own custom attributes that, while having absolutely no effect on program semantics, can nonetheless be used to embed detailed information to influence or control those tools. As an example, consider the `[[gsl::suppress(/* rules */)]]` Microsoft-specific attribute, which can be used to suppress unwanted warnings from static analysis tools that verify *Guidelines Support Library*[12] rules. In particular, consider GSL C26481 (Bounds rule #1; see which forbids any pointer arithmetic, instead suggesting that users rely on the `gsl::span` type[13]:

```
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:

---

[11] 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 pre-conditions (in addition to being runtime checked or ignored) could be *assumed*; this unique use of attribute-like syntax also required that a conforming implementation could not unilaterally ignore these precondition-checking attributes as that would make attempting to test them result in hard (*language*) **undefined behavior**.

[12] *Guidelines Support Library* is an Open-source library, developed by Microsoft, that implements functions and types suggested for use by the "C++ Core Guidelines" (**cppcoreguidelines**); see **gsl**.

[13] `gsl::span` is a lightweight reference type that observes a contiguous sequence (or subsequence) of objects of homogeneous type. Useful 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.

```
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` such that calling the function where `p1` does not refer to an object in the same contiguous block of memory as `p2` is considered a *precondition violation* (as the two addresses are compared internally). Accordingly, we might annotate the function 'f' with our own home-grown 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
#include <lib.h>

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

But, 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, it has enough information to report, at compile time, what might otherwise have resulted in **undefined behavior** at runtime.

Probing where attributes are permitted in the host platforms 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 a 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 statically viable positions for arbitrary attributes on the host platform (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 is allowed (on GCC).
```

Type expressions — e.g., the argument to `sizeof` (above) — are a notable exception; see *Potential Pitfalls*, below.

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 then misspellings in even standard attributes will go unreported[14].

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. There are, however, some that — if misused — can lead to incorrect results and/or **undefined behavior**.

For example, consider the `myRandom` function (below) 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, out of ignorance, decided to use it in production. This use is clearly a misuse because the function doesn't inherently satisfy the property that when it is invoked on the same arguments (in this case none) that it produces the same values. Adding this attribute tells the compile that it need not call this function repeatedly and is free to treat the first value returned as a constant for all time.

As a second example, consider the function `stableSort` that, as part of returning a pure result, needs to allocate and deallocate dynamic memory from the heap:

[14]Ideally there would, on every relevant platform, be a way to silently ignore a specific attribute on a case-by-case bases .

```
std::vector<int> stableSort(std::vector<int> data)
{
    std::stable_sort(data.begin(), data.end());
    return data;
}
```

If `stableSort` were to be decorated with `[[gnu::const]]`, even though it is likely that the program will continue to work as intended, the compiler is (in theory) allowed to presume that no side effects occur and would be within its rights assume that no memory allocation/deallocation is occurring either, perhaps leading to the subtlest of race conditions or other runtime defects.

Not every syntactic location is viable for an attribute There is a fairly limited subset of syntactic location for which most conforming implementation 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 host platform's C++ grammar*, above, 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 [[]] & [[]] ) /**/;
                                                ^~~~
```

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

Annoyances

None so far.

See Also

- `[[noreturn]]` — Safe C++11 standard attribute
- `[[carries_dependency]]` — Unsafe(?) C++11 standard attribute
- `alignas`: Safe C++11 attribute (with a keyword-like syntax)
- `[[deprecated]]` — Safe C++14 standard attribute

Further Reading

For more information on commonly supported function attributes, see

1.1.2 Binary Literals

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 non-empty sequence of binary digits (`0` or `1`): [15]

```
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 == 42, "");  // 1*2^5 + 0*2^4 + 1*2^3 + 1*2^5 + 0*2^4 + 1*2^3
```

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 == 256, "");
```

The type of a binary literal[16] is by default a (non-negative) `int` unless that value cannot fit in an `int`, in which case its type is the first type in the sequence {`unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`}[17] in which it will fit, or else the program is *ill-formed*:[18]

```
// 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.
```

[15] Prior to being introduced in C++14, GCC supported binary literals (with the same syntax as the standard feature) as a non-conforming extension since version 4.3; for more details, see *gnu19*, section xyz, pp. 123–456.

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

[17] 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 below).

[18] 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 `auto`.

```
auto u64  = 0b1000...[ 56 0-bits]...0000;  // u64  is unsigned long long.
auto i128 = 0b0111...[120 1-bits]...1111;  // Error: integer literal is too large.
auto u128 = 0b1000...[120 0-bits]...0000;  // Error: integer literal is 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: 25.
auto ull = 0b110101ULL;  // Type: unsigned long long; Value: 30.
```

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: assert(0 - 10 == 0 - 0b1010, "");
static_assert(0x7fffffff != -0x7fffffff, "");  // Each literal is an signed int.
static_assert(0x80000000 == -0x80000000, "");  // Each literal is an 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 4 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 who is not experienced 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, it is not uncommon for a programmer to 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 (or virtual machine)'s manual 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

- Digit Separators — Safe C++14 feature which allows to (visually) group together digits in a numerical literal in order to help readability. Often used in conjunction with binary literals.

Further reading

None so far.

1.1.3 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-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 parenthesis) appearances of `>`, `>>`, `>>>` etc. 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, there is at least the possibility that the greater-than or right-shift operators might 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 expressions, we can achieve our goal by nesting that expression within parenthesis:

```
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 and 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 (rather than just pretending you had). With this new feature (or rather a 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 working 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
```

Since this rare syntax error is invariably caught at compile-time, there are no 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) be interpreted differently in C++11 than it was when compiled for C++03. Consider the case[19] 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. It is possible to write a program that (1) compiles under both C++03 and C++11 and (2) exposes the difference in parsing rules:

```
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;
};
```

[19]Adaptation of an example originally written by Jens Gustedt, See **gustedt13**.

```
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 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

- N1757: “Right Angle Brackets” (Revision 2), by Daveed Vandevoorde (2005-01-14)

1.1.4 `decltype`

The keyword `decltype` enables the compile-time inspection of (1) the **declared type** of an **entity** or (2) 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 or a **class member access expression** (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 (which too is evaluated at compile time) `sizeof` operator, 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 `Widget`s:

```
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[20] become infinite[21].

[20] As of this writing, neither `g++ 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.

[21] 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.

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[22] of the function (see trailing 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;
}
```

Determining the validity of a generic expression In the context of generic-library development, `decltype` can be used in conjunction with SFINAE 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.
```

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

```
{
    // 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[23] `decltype(range.sort())` as part of `sortRangeImpl`'s declaration, the more specialized overload will be discarded during template substitution if `range.sort()` is not a valid expression for the deduced `Range` type[24].

Putting it all together, we see that there are exactly two possible outcomes for the original client-facing `sortRange` function invoked with 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`) does not require any 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.

[23] 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>
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.
```

[24] 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 ("Substitution Failure Is Not An Error") that acts exclusively on expressions visible in a function's signature rather than on (obscure) template-based type computations.

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.

See Also

None so far.

Further reading

None so far.

1.1.5 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[25] generated automatically by the compiler. For certain kinds of types, the notion of **copy semantics** (including **move semantics**[26]) 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 leave them unimplemented, typically resulting in a compile-time (or link-time) error when accessed[27]:

```
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 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:
    // ...
```

[25]The generation of individual special member functions can be affected by 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 Default Special Member Function.

[26]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 Move Semantics.

[27]Leaving a special member function that is declared to be private unimplemented 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.

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

    FileHandle(const FileHandle&) = delete;             // Make not available.
    FileHandle& operator=(const FileHandle&) = delete;  // Make not available.

    // ...
};
```

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 is used to write the character `*` five times in a row, starting at a specified memory address, `buf`:

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

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 C provides no language support to 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` (1) or any integral type (other than `unsigned char`[28]) that promotes to `int` (2) will map exclusively to the deleted `sent(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 (non-polymorphic) base classs member function Best practices notwithstanding[29], 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:

[28]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`.

[29]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.

```
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
}
```

If the need for a 'ForwardAudioStream' type persists, we can always consider reimplementing it more carefully later[30].

Potential Pitfalls

None so far.

Annoyances

None so far.

See Also

- Default Special Member Function — Companion safe C++11 feature which enables *defaulting* (as opposed to *deleting*) special member functions.
- Move Special Member Functions — Conditionally safe C++11 feature that introduces the two *move* variants to *copy* special member functions. <!– TODO: revisit title for this one, we might want to link to rvalue references instead –>

Further Reading

None so far.

1.1.6 `override`

Ensure 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):

[30] 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.

```
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 intent of the author of the derived class 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 codebase Relying on override as a means of ensuring that changes to base-class interfaces are propagated across a codebase can prove unreliable if this feature is not used consistently — 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 non-matching 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.

1.1.7 Compile-Time Assertions (`static_assert`)

Terminate compilation whenever a given compile-time *predicate* evaluates to `false`.

Description

Assumptions — whether we explicitly document them or not — are inherent in every program. 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[31] 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; and
2. Compile-time assertions can exist in many more places than runtime assertions, and are unrelated to program control flow; and
3. No runtime code will be generated due to a `static_assert`, hence 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* below) **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()
    {
        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:

[31]It is not unheard of for a program having assertions to run faster with them enabled than disable — 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.

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

Evaluation of static assertions in templates The standard does not explicitly specify at precisely what point (during the compilation process) static assertion declarations are evaluated[32]. 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 [33] 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 assertions 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 as the truthiness of the assertion will (in general) depend on the nature of the template argument, but see *Potential Pitfalls*, below:

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

In the example above, the compiler has no choice but to wait until each time `f3` is instantiated, as 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):

[32] By “evaluated” here we mean that the asserted expression is processed and its semantic truth determined.

[33] E.g., `g++ 10.1`, `clang++ 10.0`, and `msvc v19.24`.

```
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 in which 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*, below.

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`'s being exactly 32 bits (e.g., due to the use 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[34] or more critically (2) actively avoiding erroneous runtime behavior.

As an example, consider the `SmallObjectBuffer<N>` class templates, which provides storage for arbitrary objects whose size does not exceed `N`[35]:

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

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

    // ...
};
```

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)
{
```

[34] Syntactically incompatible types often lead to absurdly long and notoriously hard-to-read diagnostic messages — especially when deeply nested template expressions are involved; see **TODO**.

[35] 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.

```
    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 non-serializable 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 has been 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 a 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 a 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, as 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 as 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**. [36]:

[36]**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.

```
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 allows 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,
          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 non-existing `Ok` type alias via the `Check<false>` instantiation, which again, by default, is non-existent. 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 (1), selects (2)
    process(BigType());    // Discards (2), selects (1)
}
```

This general technique of paring template specializations is used widely in modern C++ programming. For another, often more convenient way of constraining overloads using **expression SFINAE**, see trailing return types .

Annoyances

Mandatory string literal Many compilation failures caused by static assertions are self-explanatory, as the offending line (which necessarily contains the predicate code) is displayed as part of the compiler diagnostic. In those situations, the message required[37] as part of `static_assert`'s grammar is redundant:

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

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

It is common to see developers provide an empty string literal in these cases:

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

See Also

- Trailing Return Types — Safe C++11 feature which allows fine-grained control over overload resolution by enabling **expression SFINAE** as part of a function's **declaration**.

Further reading

None so far.

1.1.8 Trailing Function Return Types

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 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:

```
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 non-member function[38] defined outside of the namespace in which it is are declared:

```
namespace N
{
    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`:[39]

```
// [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 that of the return type, call it `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 representations in the classic one:

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

[38] 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 and section 2.4.9, pp. 312–321, especially Figure 2-23, p. 316.

[39] The verbose declaration notation shown was first used by John Lakos while teaching his course, *Advanced Design and Programming using C++* at Columbia University during seven consecutive spring semesters (1991-1997).

```
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.

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 `decltype`).

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 functions, member functions, or data members 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 an `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, it becomes fairly straightforward to write a **higher-order function** that returns a pointer to a function, member function, or member as its return type[40].

Potential Pitfalls

None so far.

Annoyances

None so far.

See Also

- `decltype` — Safe C++11 type inference feature which is often used in conjunction with (or in place of) trailing return types.
- Automatic 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.

[40]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`).

1.1.9 Unrestricted Unions

Any non-reference 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's implicitly deleting (see Deleted Functions) 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& Nt::Nt(const Nt&).
    */
};
```

This same sort of precautionary deletion also occur for any class containing such a union as a data member (see *Use Cases*, below).

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[41]:

```
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;
    }
};
```

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 non-intrusively (i.e. the

[41]Attempting to restore an `union`'s implicitly deleted special member functions by using the `= default` syntax (see Defaulted Functions) will still result in their being deleted, as the compiler cannot know which member of the union is active without a discriminator.

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 else 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`'s being able to hold a value of type either `int` or `std::string`. By encapsulating a C++ `union` and a boolean[42] *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
```

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

```
        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* (above), 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[43]:

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

[43] For more information on initiating the lifetime of an object, see cpp14, basic.life.

```
            // 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[44].

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 it is never necessary to destroy an object having a trivial destructor; there are, however, rare cases where we may choose not to destroy an object having a non-trivial one[45].

[44] `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**.

[45] 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. As long as the only resource that is "leaked" by not invoking each individual destructor is the memory allocated from that memory resource, there is no issue and that memory can be reused without resulting in **undefined behavior** so long as it is not subsequently referenced in the context of the deallocated objects.

Annoyances

None so far.

See Also

- Deleted Functions — 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 sub-object of a union or when a class has such a union as a data member.

Further Reading

None so far.

TEST-ONLY MATERIAL STARTS HERE

1.2 C++14

Small amount of intro text here. 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.

1.2.1 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.[46] 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

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

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
   (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 1.1: code 1

```
void code()
{

}
```

Listing 1.2: 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. 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.

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.

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

Chapter 3

Unsafe Features

Chapter 4

Parting Thoughts

Bibliography

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)

stroustrup13

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

Glossary

word to be defined
Definition text follows. 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.

word to be defined
Definition text follows. Itaque earum rerum hic tenetur a sapiente delectus, ut aut reiciendis voluptatibus maiores alias consequatur aut perferendis doloribus asperiores repellat.

word to be defined
Definition text follows.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.

word to be defined
Definition text follows. 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.

Index

Symbols
$, 31
%, 12
&, 31

A
a very long entry to test the column width, 31
anise, 31
apple
  braebrun, 31
  cameo, 31
  fuji, 31
  gala, 31
  granny smith, 31
  red delicious, 31
apricots, 31
avocado, 31

B
banana, 31
basil, 31
bibendum, 12
  dapibus, 12
blackberry, 31

C
cabbage, 31
celery, 31
chervil, 31
chives, 31
cilantro, 31
`codeword`, 12
corn, 31
cucumber, 31

D
dates, 31
dill, 31

E
eggplant, 31
endive, 31

F
fennel, 31
fig, 31
`function`, 31

G
garlic, 31
grapes, 31

H
horseradish, 31
how about another long entry, 31
huckleberry, 31

J
jicama, 31

K
kale, 31
kiwi, 31

L
leeks, 31
lemon, 31
lettuce
  boston bibb, 31
  iceberg, 31
  mesclun, 31
  red leaf, 31
lime, 31
lorem, *See* lobortis

M
majoram, 31
mango, 31
maybe another long entry for this test, 31
melon
  canary, 31
  cantaloupe, 31
  honeydew, 31
  watermelon, 31
mushrooms
  button, 31
  porcini, 31
  portabella, 31
  shitake, 31

N
nectarine, 31
nutmeg, 31

O
okra, 31

onion
red, 31
vidalia, 31
yellow, 31
orange, 31

P
papaya, 31
parsley, 31
peaches, 31
peppers
ancho, 31
bell, 31
habeñeros, 31
jalapeños, 31
pablaños, 31
perhaps yet another long entry for this test, 31
plantains, 31
plums, 31
potatoes
red-skinned, 31
russet, 31
yukon gold, 31
pumpkin, 31

Q
quince, 31

R
radicchio, 31
radish, 31
raspberry, 31
rosemary, 31
rutabaga, 31

S
shallots, 31
spinach, 31
squash, 31
still another long entry for column width testing, 31

T
thyme, 31
tomatillo, 31
tomatoes
cherry, 31
grape, 31
heirloom, 31
hybrid, 31
roma, 31
`typeof`, 31

U
ugly fruit, 31

V
verbena, 31
viverra, *See also* neque

X
xacuti masala, 31

Y
yams, 31
yet another very long entry for column width test, 31

Z
zucchini, 31