# *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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

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**

Copyright © 2016 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearsoned.com/permissions/.

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

This is John's dedication to Vittorio for being so great and
writing this book so well.
JL

This is Vittorio dedication to something else.

VR

This is Slava's dedication to something else.

RK

This is Alisdair's dedication to something else.

AM

# Contents

**Foreword** **xv**

**Preface** **xvii**

**Acknowledgements** **xix**

**About the Authors** **xxi**

**Chapter 0 Introduction** **1**
0.1 What Makes This Book Different 1
0.2 Scope for the First Edition 2
0.3 The *EMC++S* White Paper 3
    0.3.1 Facts (Not Opinions) 3
    0.3.2 Elucidation (Not Prescription) 3
    0.3.3 Brevity (Not Verbosity) 3
    0.3.4 Real-World (Not Contrived) Examples 4
    0.3.5 At Scale (Not Overly Simplistic) Programs 4
0.4 What Do We Mean by *Safely*? 4
0.5 A *Safe* Feature 5
0.6 A *Conditionally Safe* Feature 5
0.7 An *Unsafe* Feature 5
0.8 Modern C++ Feature Catalog 6
    0.8.1 Organization 6
0.9 How To Use This Book 7

**Chapter 1 Safe Features** **9**
*C++11*
1.1 Attributes 10
    1.1.1 Description 10
    1.1.2 Use Cases 13
    1.1.3 Potential Pitfalls 17
    1.1.4 Annoyances 18
    1.1.5 See Also 18
    1.1.6 Further Reading 19
1.2 Binary Literals 20

1.2.1 Description 20
1.2.2 Use Cases 21
1.2.3 Potential Pitfalls 23
1.2.4 Annoyances 23
1.2.5 See Also 23
1.2.6 Further Reading 23
1.3 Consecutive Right Angle Brackets 24
1.3.1 Description 24
1.3.2 Use Cases 25
1.3.3 Potential pitfalls 25
1.3.4 Annoyances 26
1.3.5 See Also 26
1.3.6 Further Reading 27
1.4 `decltype` 28
1.4.1 Description 28
1.4.2 Use Cases 29
1.4.3 Potential pitfalls 32
1.4.4 Annoyances 32
1.4.5 See Also 32
1.4.6 Further reading 32
1.5 Deleted Functions 33
1.5.1 Description 33
1.5.2 Use Cases 33
1.5.3 Potential Pitfalls 36
1.5.4 Annoyances 37
1.5.5 See Also 37
1.5.6 Further Reading 37
1.6 `override` 38
1.6.1 Description 38
1.6.2 Use Cases 38
1.6.3 Potential Pitfalls 39
1.6.4 Annoyances 40
1.6.5 See Also 40
1.6.6 Further Reading 40
1.7 Compile-Time Assertions (`static_assert`) 41
1.7.1 Description 41
1.7.2 Use Cases 43
1.7.3 Potential Pitfalls 45
1.7.4 Annoyances 48
1.7.5 See Also 48
1.7.6 Further reading 48
1.8 Trailing Function Return Types 49
1.8.1 Description 49
1.8.2 Use Cases 51
1.8.3 Potential Pitfalls 52

1.8.4 Annoyances 52
1.8.5 See Also 52
1.8.6 Further Reading 52
1.9 Unrestricted Unions 53
1.9.1 Description 53
1.9.2 Use Cases 56
1.9.3 Potential Pitfalls 58
1.9.4 Annoyances 59
1.9.5 See Also 59
1.9.6 Further Reading 59
1.10 The `[[noreturn]]` Attribute 60
1.10.1 Description 60
1.10.2 Use Cases 60
1.10.3 Potential Pitfalls 62
1.10.4 Annoyances 62
1.10.5 See Also 63
1.10.6 Further Reading 63
1.11 Null Pointer Literal (`nullptr`) 64
1.11.1 Description 64
1.11.2 Use Cases 65
1.11.3 Potential Pitfalls 67
1.11.4 Annoyances 67
1.11.5 See Also 67
1.11.6 Further Reading 67
1.12 **`alignas`** 68
1.12.1 Description 68
1.12.2 Use Cases 71
1.12.3 Potential Pitfalls 74
1.12.4 See Also 76
1.12.5 Further Reading 76
1.12.6 Appendix 76
1.13 Delegating Constructors 81
1.13.1 Description 81
1.13.2 Use Cases 82
1.13.3 Potential Pitfalls 85
1.13.4 Annoyances 86
1.13.5 See Also 86
1.13.6 Further Reading 86
1.14 Local Types as Template Arguments 87
1.14.1 Description 87
1.14.2 Use Cases 88
1.14.3 Potential Pitfalls 90
1.14.4 Annoyances 90
1.14.5 See Also 90
1.14.6 Further Reading 91

1.15 `long long` 92
  1.15.1 Description 92
  1.15.2 Use Cases 92
  1.15.3 Potential Pitfalls 94
  1.15.4 See Also 95
  1.15.5 Further Reading 95
  1.15.6 Appendix: Historical Perspective on the Evolution of Use of Fundamental Integral Types 95
1.16 Alias Declarations and Alias Templates 97
  1.16.1 Description 97
  1.16.2 Use Cases 98
  1.16.3 Potential Pitfalls 100
  1.16.4 Annoyances 100
  1.16.5 See Also 100
  1.16.6 Further Reading 101
  1.16.7 Appendix: Brief Review of (C++03) `using` Declarations 101
1.17 Explicit Conversions 104
  1.17.1 Description 104
  1.17.2 Use Cases 106
  1.17.3 Potential Pitfalls 108
  1.17.4 Sometimes a named function is better 109
  1.17.5 Annoyances 109
  1.17.6 See Also 109
  1.17.7 Further Reading 109
1.18 `alignof` 110
  1.18.1 Description 110
  1.18.2 Use Cases 111
  1.18.3 Annoyances 118
  1.18.4 See Also 119
  1.18.5 Further Reading 119
1.19 Inheriting Constructors 120
  1.19.1 Description 120
  1.19.2 Use Cases 124
  1.19.3 Potential Pitfalls 129
  1.19.4 Annoyances 132
  1.19.5 See Also 134
  1.19.6 Further Reading 135
  1.19.7 Appendix: C++17 Improvements Made Retroactive to C++11/14 135
1.20 Unicode String and Character Literals 140
  1.20.1 Description 140
  1.20.2 Use Cases 141
  1.20.3 Potential Pitfalls 141
  1.20.4 Annoyances 142
  1.20.5 See Also 142
  1.20.6 Further Reading 142

1.21 Explicit Enumeration Underlying Type 143
- 1.21.1 Description 143
- 1.21.2 Use Cases 144
- 1.21.3 Potential Pitfalls 145
- 1.21.4 See Also 147
- 1.21.5 Further Reading 147

1.22 `enum class` 148
- 1.22.1 Description 148
- 1.22.2 Use Cases 152
- 1.22.3 Potential Pitfalls 159
- 1.22.4 Annoyances 164
- 1.22.5 See Also 165
- 1.22.6 Further Reading 165

1.23 Opaque Enumeration Declarations 166

*C++14*

1.24 Aggregate Member Initialization Relaxation 167
- 1.24.1 Description 167
- 1.24.2 Use Cases 168
- 1.24.3 Potential Pitfalls 168
- 1.24.4 Annoyances 169
- 1.24.5 See Also 169
- 1.24.6 Further Reading 170

1.25 Digit Separators 171
- 1.25.1 Description 171
- 1.25.2 Use Cases 172
- 1.25.3 Potential Pitfalls 172
- 1.25.4 See Also 172
- 1.25.5 Further Reading 173
- 1.25.6 Appendix: Silent Loss of Precision in Floating-Point Literals 173

1.26 Variable Templates 176
- 1.26.1 Description 176
- 1.26.2 Use Cases 178
- 1.26.3 Potential Pitfalls 180
- 1.26.4 Annoyances 182
- 1.26.5 See Also 183
- 1.26.6 Further Reading 183

1.27 Defaulted Special Member Functions 184
- 1.27.1 Description 184
- 1.27.2 Use Cases 187
- 1.27.3 Potential Pitfalls 193
- 1.27.4 See Also 194
- 1.27.5 Further Reading 194
- 1.27.6 Appendix: Implicit Generation of Special Member Functions 194

1.28 `[[deprecated]]` 196
- 1.28.1 Description 196

- 1.28.2 Use Cases 197
- 1.28.3 Potential Pitfalls 198
- 1.28.4 Annoyances 199
- 1.28.5 See Also 199
- 1.28.6 Further Reading 199

1.29 Relaxed `constexpr` Restrictions 200
- 1.29.1 Description 200
- 1.29.2 Use Cases 201
- 1.29.3 Potential Pitfalls 206
- 1.29.4 Annoyances 206
- 1.29.5 See Also 206
- 1.29.6 Further Reading 207
- 1.29.7 Appendix: Optimized C++11 Example Algorithms 207

1.30 Lambda-Capture Expressions 209
- 1.30.1 Description 209
- 1.30.2 Use Cases 211
- 1.30.3 Potential Pitfalls 214
- 1.30.4 Annoyances 215
- 1.30.5 See Also 216
- 1.30.6 Further Reading 216

1.31 Raw String Literals 217
- 1.31.1 Description 217
- 1.31.2 Use Cases 220
- 1.31.3 Potential Pitfalls 220
- 1.31.4 Annoyances 222
- 1.31.5 See Also 222
- 1.31.6 Further Reading 222

**Chapter 2 Conditionally Safe Features** **223**

2.1 C++11 223
- 2.1 `auto` 224
- 2.2 Braced Initialization 225
- 2.3 Rvalue References 226
- 2.4 Default Member Initializers 227
- 2.5 `constexpr` Variables 228
- 2.6 `constexpr` Functions 229
- 2.7 Variadic Templates 230
- 2.8 Lambdas 231
- 2.9 Forwarding References 232
- 2.10 `noexcept` 233

2.2 C++14 233

2.11 Generic Lambdas 234

**Chapter 3 Unsafe Features** **235**

3.1 C++11 235

3.1 `[[carries_dependency]]` (The `[[carries_dependency]]` Attribute) 236
3.2 C++14 237
3.2 Deduced Return Types (Function Return Type Deduction) 238

**Chapter 4 Parting Thoughts 239**
4.1 Testing Section 239
4.2 Testing Another Section 239

**Todo list 241**

# Foreword

The text of the foreword will go here.

# Preface

The text of the preface will go here.

# Acknowledgements

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

# About the Authors

Author
Photo
here

**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).

Author
Photo
here

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

# Chapter 0

# Introduction

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

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

Richard Feynman famously said: “If it disagrees with experiment, it’s wrong. In that simple statement is the key to science.”[1] There is no better way to experiment with a language feature than letting time do its work. We took that to heart by dedicating *Embracing Modern C++ Safely* to only the features of Modern C++ that have been part of the Standard for at least five years, which grants enough perspective to properly evaluate its practical impact. Thus, we are able to provide you with a thorough and comprehensive treatment based on practical experience and worthy of your limited professional development time. If you’re out there looking for tried and true ways to better use modern C++ features for improving your productivity, we hope this book will be the one you’ll reach for.

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

## 0.1 What Makes This Book Different

The book you’re now reading aims very strongly at being objective, empirical, and practical. We simply present features, their applicability, and their potential pitfalls as reflected by the analysis of millions of human-hours of using C++11 and C++14 in the development of varied large-scale software systems; personal preference matters have been neutralized to our, and our reviewers’, best ability. We wrote down the distilled truth that remains, which should shape your understanding of what modern C++ has to offer to you without being skewed by our subjective opinions or domain-specific inclinations.

---

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

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

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

We are not aware of any similar text amid the rich offering of C++ textbooks; in a very real sense, we wrote it because we needed it.

## 0.2 Scope for the First Edition

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

In this first of an anticipated series of periodically extended volumes, we characterize, dissect, and elucidate most of the modern language features introduced into the C++ Standard starting with C++11. We chose to limit the scope of this first edition to only those features that have been in the language Standard and widely available in practice for at least five years. This limited focus enables us to more fully evaluate the real-world impact of these features and to highlight any caveats that might not have been anticipated prior to standardization and sustained, active, and widespread use in industry.

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

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

C++11. With that horizon in mind, we are actively planning to cover pre-C++11 material in future editions. For the time being, however, we highly recommend *Effective C++* by Scott Meyers[1] as a concise, practical treatment of many important and useful C++98/03 features.

## 0.3 The *EMC++S* White Paper

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

### 0.3.1 Facts (Not Opinions)

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

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

### 0.3.2 Elucidation (Not Prescription)

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

### 0.3.3 Brevity (Not Verbosity)

*Embracing Modern C++ Safely* is neither designed nor intended to be an introduction to modern C++. It is a handy reference for experienced C++ programmers who may have a passing knowledge of the recently added C++ features and a desire to perfect their understanding. Our writing style is intentionally tight, with the goal of providing you with facts, concise objective analysis, and cogent, real-world examples. By doing so we spare you the task of wading through introductory material. If you are entirely unfamiliar with a feature, we suggest you start with a more elementary and language-centric text such as *The C++ Programming Language* by Bjarne Stroustrup.[1]

---

[1] **meyers05**

[1] **stroustrup13**

### 0.3.4 Real-World (Not Contrived) Examples

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

### 0.3.5 At Scale (Not Overly Simplistic) Programs

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

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

## 0.4 What Do We Mean by *Safely*?

The ISO C++ Standards Committee, of which we are members, would be remiss — and downright negligent — if it allowed any feature of the C++ language to be standardized if that feature were not reliably safe when used as intended. Still, we have chosen the word "safely" as the moniker for the signature aspect of our book, by which we indicate a comparatively favorable risk-to-reward ratio for using a given feature in a large-scale development environment. By contextualizing the meaning of the term "safe," we get to apply it to a real-world economy in which everything has a cost in multiple dimensions: risk of misuse, added maintenance burden borne by using a new feature in an older code base, and training needs for developers who might not be familiar with that feature.

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

1. Number and severity of known deficiencies

2. Difficulty in teaching consistent proper use
3. Experience level required for consistent proper use
4. Risks associated with widespread misuse

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

## 0.5 A *Safe* Feature

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

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

## 0.6 A *Conditionally Safe* Feature

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

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

## 0.7 An *Unsafe* Feature

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

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

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

## 0.8 Modern C++ Feature Catalog

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

### 0.8.1 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
- Chapter 2: Conditionally Safe Features
- Chapter 3: Unsafe Features
- Chapter 4: Parting Thoughts

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

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

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

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

## 0.9 How To Use This Book

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

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

2. **Read the chapters in order but slowly over time.** An incremental, priority-driven approach is also possible and recommended, especially if you're feeling less sure-footed. Understanding and applying first the safe features of Chapter 1 gets you the low-hanging fruit. In time, the conditionally safe features of Chapter 2 will allow you to ease into the breadth of useful modern C++ language features, prioritizing those that are least likely to prove problematic.

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

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

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

# Chapter 1

# Safe Features

Intro text should be here.

# 1.1 Attributes

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

## 1.1.1 Description

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

### 1.1.1.1 C++ attribute syntax

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

```
[[attribute_name]]
```

A single annotation can consist of zero or more attributes:

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

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

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

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

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
```

---

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

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

[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.

### 1.1.1.2 C++ attribute placement

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

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

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

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

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

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

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

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

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

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

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

```
[[attr1]];                              // null statement
```

---

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

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

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

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

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

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

### 1.1.1.3 Common compiler-dependent attributes

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

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

**Table 1–1: Some standardized compiler-specific attributes**

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

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

---

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

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

## 1.1.2 Use Cases

### 1.1.2.1 Eliciting useful compiler diagnostics

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

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

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

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

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

For the code above, GCC produces a useful warning:

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

### 1.1.2.2 Hinting at additional optimization opportunities

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

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

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

---

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

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

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

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

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

### 1.1.2.3 Delineating explicit assumptions in code to achieve better optimizations

Although the presence (or absence) of an attribute usually has no effect on the behavior of any well-formed program (besides runtime performance), an attribute sometimes imparts knowledge to the compiler which, if incorrect, could alter the intended behavior of the program (or perhaps mask the defective behavior of an incorrect one). As an example of this more forceful form of attribute, consider the GCC-specific `[[gnu::const]]` attribute (also available on Clang). When applied to a function, this (atypically) powerful (and dangerous, see *Potential Pitfalls: Some attributes, if misused, can affect program correctness* on page 17) attribute instructs the compiler to *assume* that the function is a **pure function** (i.e., that it always returns the same value for any given set of arguments) and has no **side effects** (i.e., the globally reachable state[11] of the program is unaltered by calling this function):

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

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

```
Vector2D vectorLerp(const Vector2D& start, const Vector2D& end, double factor)
{
```

---

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

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

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

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

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

If the implementation of a function tagged with the `[[gnu::pure]]` attribute does not satisfy limitations imposed by the attribute, however, the compiler will not be able to detect this and a runtime defect will be the likely result[12]; see *Potential Pitfalls: Some attributes, if misused, can affect program correctness* on page 17.

### 1.1.2.4 Using attributes to control external static analysis

Since unknown attributes are ignored by the compiler, external static-analysis tools can define their own custom attributes that can be used to embed detailed information to influence or control those tools without affecting program semantics. For example, the Microsoft-specific `[[gsl::suppress(/* rules */)]]` attribute can be used to suppress unwanted warnings from static-analysis tools that verify *Guidelines Support Library*[13] rules. In particular, consider GSL C26481 (Bounds rule #1),[14] which forbids any pointer arithmetic, instead suggesting that users rely on the `gsl::span` type[15]:

```
void hereticalFunction()
{
```

---

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

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

[14] **microsoftC26481**

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

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

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

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

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

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

### 1.1.2.5 Creating new attributes to express semantic properties

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

```
// lib.h

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

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

```
// client.cpp
#include <lib.h>

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

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

### 1.1.2.6 Probing where attributes are permitted in the compiler’s C++ grammar

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

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

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

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

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

## 1.1.3 Potential Pitfalls

### 1.1.3.1 Unrecognized attributes have implementation-defined behavior

Although standard attributes work well and are portable across all platforms, the behavior of compiler-specific and user-specified attributes is entirely implementation defined, with unrecognized attributes typically resulting in compiler warnings. Such warnings can typically be disabled (e.g., on GCC using `-Wno-attributes`), but, if they are, misspellings in even standard attributes will go unreported.[16]

### 1.1.3.2 Some attributes, if misused, can affect program correctness

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

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

```
double myRandom()
{
```

---

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

```
static std::random_device randomDevice;
static std::mt19937 generator(randomDevice());

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

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

### 1.1.3.3 Not every syntactic location is viable for an attribute

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

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

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

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

## 1.1.4 Annoyances

None so far

## 1.1.5 See Also

- “The `[[noreturn]]` Attribute” on page 60 — Safe C++11 standard attribute for functions that never return control flow to the caller
- “`[[carries_dependency]]` (The `[[carries_dependency]]` Attribute)” on page 236 — Unsafe C++11 standard attribute used to communicate release-consume dependency-chain information to the compiler to avoid unnecessary memory-fence instructions
- “`alignas`” on page 68 — Safe C++11 attribute (with a keyword-like syntax) used to widen the alignment of a type or an object

- "`[[deprecated]]`" on page 196 — Safe C++14 standard attribute that discourages the use of an entity via compiler diagnostics

### 1.1.6 Further Reading

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

## 1.2 Binary Literals

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

### 1.2.1 Description

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

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

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

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

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

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

The type of a binary literal[2] is by default a (non-negative) `int` unless that value cannot fit in an `int`. In that case, its type is the first type in the sequence {`unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`}[3] in which it will fit. If neither of those is applicable, then the program is *ill-formed*[4]:

```
// example platform 1:
// (sizeof(int): 4; sizeof(long): 4; sizeof(long long): 8)
auto i32  = 0b0111...[ 24 1-bits]...1111;  // i32 is int.
```

---

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

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

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

[4] Purely for convenience of exposition, we have employed the C++11 `auto` feature to conveniently capture the type implied by the literal itself; for more information, see Section 2.1, “`auto`.”

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

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

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

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

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

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

## 1.2.2 Use Cases

### 1.2.2.1 Bit masking and bitwise operations

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

```
unsigned int lastFourBits(unsigned int value)
{
return value & 0xFu;
}
```

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

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

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

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

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

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

// ...
};
```

#### 1.2.2.2 Replicating constant binary data

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

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

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

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

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

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

// ...
}
}
```

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

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

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

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

## 1.2.3 Potential Pitfalls

None so far

## 1.2.4 Annoyances

None so far

## 1.2.5 See Also

- Section 1.25, "Digit Separators" — Safe C++14 feature that allows a developer to (visually) group together digits in a numerical literal to help readability. Often used in conjunction with binary literals.

## 1.2.6 Further Reading

None so far

## 1.3 Consecutive Right Angle Brackets

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

### 1.3.1 Description

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

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

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

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

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

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

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

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

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

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

## 1.3.2 Use Cases

### 1.3.2.1 Avoiding annoying whitespace when composing template types

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

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

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

## 1.3.3 Potential pitfalls

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

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

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

The easy fix is simply to parenthesize the expression:

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

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

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

Though pathologically rare, the same valid expression can (in theory) have a different interpretation in C++11 than it had when compiled for C++03. Consider the case[1] where the `>>` token is embedded as part of an expression involving templates:

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

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

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

template <typename> struct S
{
```

---

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

```
enum Inner { a = 100, c = 102 };
};

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

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

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

```
// C++03

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

// C++11

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

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

### 1.3.4 Annoyances

None so far

### 1.3.5 See Also

None so far

### 1.3.6 Further Reading

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

## 1.4 decltype

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

### 1.4.1 Description

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

#### 1.4.1.1 Use with (typically named) entities

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

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

#### 1.4.1.2 Use with (unnamed) expressions

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

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

The three integer expressions below illustrate the various value categories:

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

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

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

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

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

## 1.4.2 Use Cases

### 1.4.2.1 Avoiding unnecessary use of explicit typenames

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

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

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

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

### 1.4.2.2 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[1] become infinite.[2]

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)
// ...
```

### 1.4.2.3 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[3] of the function (see Section 1.8, "Trailing Function Return Types") and (2) the auxiliary variable:

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

### 1.4.2.4 Determining the validity of a generic expression

In the context of generic-library development, `decltype` can be used in conjunction with **SFINAE**[4] 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)
```

---

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

[2] 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.

[3] Using `decltype(a + b)` as a return type is significantly different from relying on *automatic return type deduction*. See Section 2.1, "`auto`," for more information.

[4] "Substitution Failure Is Not An Error"

```
{
sortRangeImpl(range, 0);
}
```

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

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

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

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

Note that, by exposing[5] `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.[6]

---

[5] 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.
```

[6] 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

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

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

### 1.4.3 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.

### 1.4.4 Annoyances

None so far

### 1.4.5 See Also

None so far

### 1.4.6 Further reading

None so far

---

commonly known as **expression SFINAE** and is a restricted form of the more general (classical) SFINAE that acts exclusively on expressions visible in a function's signature rather than on (obscure) template-based type computations.

## 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.

### 1.5.1 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.

### 1.5.2 Use Cases

#### 1.5.2.1 Suppressing special member function generation

When instantiating an object of user-defined type, **special member functions** that have not been declared explicitly are often[1] generated automatically by the compiler. For certain kinds of types, the notion of **copy semantics** (including **move semantics**[2]) 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[3]:

---

[1] The generation of individual special member functions can be affected by the existence of other user-defined special member functions or by limitations imposed by the specific types of any data members or base types. For more information, see Section 1.27, "Defaulted Special Member Functions."

[2] The two **special member functions** controlling *move* operations (introduced in C++11) are sometimes implemented as effective optimizations of copy operations and (rarely) with copy operations explicitly deleted; see Section 2.3, "Rvalue References."

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

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

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

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

// ...
};
```

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

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

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

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

// ...
};
```

### 1.5.2.2 Preventing a particular implicit conversion

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

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

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

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

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

Pernicious user errors can now be reported during compilation:

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

### 1.5.2.3 Preventing all implicit conversions

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

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

// ...
};

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

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

---

[4] 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`.

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

Best practices notwithstanding,[5] it can be cost-effective (in the short term) to provide an elided "view" on a concrete class for (trusted) clients. Imagine a class `AudioStream` designed to play sounds and music that — in addition to providing basic "play" and "rewind" operations — sports a large, robust interface:

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

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

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

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

If the need for a `ForwardAudioStream` type persists, we can always consider reimplementing it more carefully later.[6]

## 1.5.3 Potential Pitfalls

None so far

---

[5] 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.

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

### 1.5.4 Annoyances

None so far

### 1.5.5 See Also

- Section 1.27, “Defaulted Special Member Functions” — Companion safe C++11 feature that enables *defaulting* (as opposed to *deleting*) special member functions
- Section 2.3, “Rvalue References” — Conditionally safe C++11 feature that introduces the two *move* variants to *copy* special member functions

### 1.5.6 Further Reading

None so far

## 1.6 `override`

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

### 1.6.1 Description

The **contextual keyword** `override` can be provided at the end of a member-function declaration to ensure that the decorated function is indeed **overriding** a corresponding `virtual` member function in a base class (i.e., not **hiding** it or otherwise inadvertently introducing a distinct function declaration):

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

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

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

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

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

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

### 1.6.2 Use Cases

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

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

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

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

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

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

Without applying the corresponding modification to all classes deriving from `ErrorCategory`, the semantics of the program change due to the derived classes now hiding (instead of overriding) the base class’s `virtual` member function. Both of the errors discussed above would be detected automatically by decorating the `virtual` functions in all derived classes with `override`:

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

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

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

## 1.6.3 Potential Pitfalls

### 1.6.3.1 Lack of consistency across a code base

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

### 1.6.4 Annoyances

None so far

### 1.6.5 See Also

None so far

### 1.6.6 Further Reading

None so far

# 1.7 Compile-Time Assertions (`static_assert`)

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

## 1.7.1 Description

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

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

### 1.7.1.1 Syntax and semantics

We can use *static assertion declarations* to conditionally trigger controlled compilation failures depending on the truthiness of a **constant expression**. Such declarations are introduced by the `static_assert` keyword, followed by a parenthesized list consisting of (1) a constant Boolean expression and (2) a mandatory (see *Annoyances: Mandatory string literal* on page 48) **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
}
```

---

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

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

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

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

### 1.7.1.2 Evaluation of static assertions in templates

The C++ Standard does not explicitly specify at precisely what point (during the compilation process) static assertion declarations are evaluated.[2] 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[3] evaluated immediately — i.e., as soon as it is parsed and irrespective of whether any subsequent template instantiations occur:

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

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

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

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

(However, see *Potential Pitfalls: Static assertions in templates can trigger unintended compilation failures* on page 45.) In the example above, the compiler has no choice but to wait until each time `f3` is instantiated because the truth of the predicate will vary depending on the type provided:

```
void g()
{
f3<double>();               // OK
```

---

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

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

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

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

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

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

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

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

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

Both `f2` and `h2` are ill-formed template functions; the cause of their being ill formed has nothing to do with the template type and hence will always be reported as a compile-time error in practice. Finally, `f3` can be only contextually ill formed whereas `h3` is always necessarily ill formed, and yet neither is reported by typical compilers as such unless and until it has been instantiated. Reliance on a compiler not to notice that a program is ill formed is dubious; see *Potential Pitfalls: Static assertions in templates can trigger unintended compilation failures* on page 45.

## 1.7.2 Use Cases

### 1.7.2.1 Verifying assumptions about the target platform

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

```
#include <ctype>  // CHAR_BIT

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

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

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

### 1.7.2.2 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[4] or, more critically, (2) actively avoiding erroneous runtime behavior.

As an example, consider the `SmallObjectBuffer<N>` class templates, which provide storage for arbitrary objects whose size does not exceed `N`[5]:

```
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.");
```

---

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

[5] 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.

```
new (&d_buffer) T(object);
}
```

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

```
#include <ctype>  // CHAR_BIT

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

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

## 1.7.3 Potential Pitfalls

### 1.7.3.1 Static assertions in templates can trigger unintended compilation failures

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

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

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

template <typename T>
void serialize(char* buffer, const T& object, SerializableTag<false>)  // (2)
{
static_assert(false, "T must be serializable.");  // independent of T
// too obviously ill formed: always a compile-time error
}
```

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

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

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

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

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

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

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

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

### 1.7.3.2 Misuse of static assertions to restrict overload sets

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

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

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

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

To achieve the goal of removing (up front) a specialization from consideration, we will need to employ **SFINAE**. To do that, we must instead find a way to get the failing compile-time expression to be part of the function's **declaration**[6]:

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

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

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

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

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

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

---

[6] **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.

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

This general technique of pairing template specializations is used widely in modern C++ programming. For another, often more convenient way of constraining overloads using **expression SFINAE**, see Section 1.8, “Trailing Function Return Types.”

## 1.7.4 Annoyances

### 1.7.4.1 Mandatory string literal

Many compilation failures caused by static assertions are self-explanatory since the offending line (which necessarily contains the predicate code) is displayed as part of the compiler diagnostic. In those situations, the message required[7] as part of `static_assert`’s grammar is redundant:

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

Developers commonly provide an empty string literal in these cases:

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

## 1.7.5 See Also

- Section 1.8, “Trailing Function Return Types” — Safe C++11 feature that allows fine-grained control over overload resolution by enabling **expression SFINAE** as part of a function’s **declaration**

## 1.7.6 Further reading

None so far

---

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

## 1.8 Trailing Function Return Types

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

### 1.8.1 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[1]:

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

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

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

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

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

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

The same advantage would apply to a nonmember function[2] defined outside of the namespace in which it is declared:

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

---

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

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

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

```
};

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`[3]:

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

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

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

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

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

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

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

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

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

---

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

## 1.8.2 Use Cases

### 1.8.2.1 Function template whose return type depends on a parameter type

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

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

The return type of `linearInterpolation` is the type of expression inside the `decltype` *specifier*, which is identical to the expression returned in the body of the function. Hence, this interface necessarily supports any set of input types for which `a + factor * (b - a)` is valid, including types such as mathematical vectors, matrices, or expression templates. As an added benefit, the presence of the expression in the function's declaration enables **expression SFINAE**, which is typically desirable for generic template functions (see Section 1.4, "`decltype`").

### 1.8.2.2 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.

#### 1.8.2.3 Improving readability of declarations involving function pointers

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

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

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

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

Using this divide-and-conquer approach, writing a **higher-order function** that returns a pointer to a function, member function, or data member as its return type[4] becomes fairly straightforward.

### 1.8.3 Potential Pitfalls

None so far

### 1.8.4 Annoyances

None so far

### 1.8.5 See Also

- Section 1.4, "`decltype`" — Safe C++11 type inference feature that is often used in conjunction with (or in place of) trailing return types
- Section 3.2, "Deduced Return Types (Function Return Type Deduction)" — Unsafe C++14 type inference feature that shares syntactical similarities with trailing return types, leading to potential pitfalls when migrating from C++11 to C++14

### 1.8.6 Further Reading

None so far

---

[4] 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`).

## 1.9 Unrestricted Unions

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

### 1.9.1 Description

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

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

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

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

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

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

As an added safety measure, any non-trivial **special member function** defined — either implicitly or explicitly — for any *member* of a `union` results in the compiler implicitly deleting (see Section 1.5, "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::operator=(const Nt&)
*/
};
```

This same sort of precautionary deletion also occurs for any class containing such a union as a data member (see *Use Cases: Implementing a* ***sum type*** *as a discriminating (or tagged)* `union` on page 56).

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[1]:

```
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;
}
};
```

---

[1] Attempting to restore a `union`'s implicitly deleted special member functions by using the `= default` syntax (see Section 1.27, "Defaulted Special Member Functions") will still result in their being deleted because the compiler cannot know which member of the union is active without a discriminator.

## 1.9.2 Use Cases

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

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

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

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

// ...

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

return ParseResult(oss.str());
}

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

return ParseResult(oss.str());
}

// ...

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

The implementation above relies on `ParseResult` being able to hold a value of type either `int` or `std::string`. By encapsulating a C++ `union` and a Boolean[2] *discriminator* as part of the `ParseResult` **sum type**, we can achieve the desired semantics:

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

bool d_isError;  // discriminator

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

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

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

As discussed in *Description* on page 53, 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[3]:

```
ParseResult::~ParseResult()
{
if(d_isError)
```

---

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

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

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

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

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

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

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

In practice, `ParseResult` would typically be defined as a template and renamed to allow any arbitrary result type `T` to be returned or else implemented in terms of a more general **sum type** abstraction.[4]

## 1.9.3 Potential Pitfalls

### 1.9.3.1 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

---

[4] `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**.

**undefined behavior**. Although forgetting to destroy an object does not necessarily result in **undefined behavior**, failing to do so for any object that manages a resource (such as dynamic memory) will result in a *resource leak* and/or lead to unintended behavior. Note that destroying an object having a trivial destructor is never necessary; there are, however, rare cases where we may choose not to destroy an object having a non-trivial one.[5]

## 1.9.4 Annoyances

None so far

## 1.9.5 See Also

- Section 1.5, “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 subobject of a union or when a class has such a union as a data member.

## 1.9.6 Further Reading

None so far

---

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

# 1.10 The `[[noreturn]]` Attribute

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

## 1.10.1 Description

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

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

The `[[noreturn]]` attribute is not part of a function's type and is also, therefore, not part of the type of a function pointer. Applying `[[noreturn]]` to a function pointer is not an error, though doing so has no actual effect in standard C++; see *Potential Pitfalls: Misuse of* `[[noreturn]]` *on function pointers* on page 62. Use on a pointer might have benefits for external tooling, code expressiveness, and future language evolution:

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

## 1.10.2 Use Cases

### 1.10.2.1 Better compiler diagnostics

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

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

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

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

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

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

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

### 1.10.2.2 Improved runtime performance

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

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

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

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

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

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

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

Notice that even though `[[noreturn]]` appeared only on the first declaration (in the `util.h` header), the `[[noreturn]]` attribute carries over to the redeclaration used in the `throwBadAlloc` function's definition because the header was included in the corresponding `.cpp` file.

## 1.10.3 Potential Pitfalls

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

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

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

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

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

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

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

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

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

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

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

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

## 1.10.4 Annoyances

None so far

### 1.10.5 See Also

- Section 1.1, “Attributes” — To learn more about allowed attribute placement in general

### 1.10.6 Further Reading

None so far

# 1.11 Null Pointer Literal (`nullptr`)

The keyword `nullptr` (unambiguously) denotes the null-pointer-value literal.

## 1.11.1 Description

The `nullptr` keyword is a **prvalue** (pure rvalue) of type `std::nullptr_t` representing the (implementation-defined) bit pattern corresponding to a **null address** on the host platform; `nullptr` and other values of type `std::nullptr_t` (along with the integer literal `0` and the macro `NULL`) can be converted implicitly to any pointer (or pointer-to-member) type:

```
int data;  // non-member data

int *pi0 = &data;    // Initialize with non-null address.
int *pi1 = nullptr;  // Initialize with null address.
int *pi2 = NULL;     //  "          "    "    "
int *pi3 = 0;        //  "          "    "    "

double f(int x);  // non-member function

double (*pf0)(int) = &f;       // Initialize with non-null address.
double (*pf1)(int) = nullptr;  // Initialize with null address.

struct S
{
short d_data;    // member data
float g(int y);  // member function
};

short S::*pmd0 = &S::d_data;  // Initialize with non-null address.
short S::*pmd1 = nullptr;     // Initialize with null address.

float (S::*pmf0)(int) = &S::g;    // Initialize with non-null address.
float (S::*pmf1)(int) = nullptr;  // Initialize with null address.
```

Because `std::nullptr_t` is its own distinct type, overloading on it is possible:

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

void g(void*);           // (1)
void g(int);             // (2)
void g(std::nullptr_t);  // (3)

void f()
{
g("hello");  // OK --- (1) void g(void*)
g(0);        // OK --- (2) void g(int)
g(nullptr);  // OK --- (3) void g(std::nullptr_t)
g(NULL);     // Error: ambiguous --- (1), (2), or (3)
}
```

## 1.11.2 Use Cases

### 1.11.2.1 Improve type safety

In pre-C++11 code bases, use of the `NULL` macro[1] was a common way of indicating (mostly to the human reader) that the literal value it conveys is intended specifically to represent a *null address* rather than the literal `int` value `0`. From a type-safety perspective, its implementation-defined (typically integral[2]) definition, however, makes the use of `NULL` only marginally better suited than a raw literal `0` to represent a null pointer.

As just one specific illustration of the added type safety provided by `nullptr`, imagine that you work for a large software company that has historically required, as one of its coding standards, that values returned via output parameters (as opposed to a `return` statement) are always returned via pointer to a modifiable object.[3] In the illustrative function below, the output parameter's local pointer variable is "zeroed" (shown here in three different ways) to indicate (and ensure) that nothing more is to be written:

```
int illustrativeFunction(int* x)   // pointer to modifiable integer
{
// ...
if (/*...*/)
{
x = 0;       // (1) Set pointer x to null address.
x = NULL;    // (2) Set pointer x to null address.
x = nullptr; // (3) Set pointer x to null address.
}
// ...
return 0;    // success
}
```

Now suppose that the function signature is changed (e.g., due to a change in coding standards in the organization) to accept a reference instead of a pointer:

```
int illustrativeFunction(int& x)  // reference to modifiable integer
{
// ...
if (/*...*/)
{
x = 0;       // (1) always compiles; makes what x refers to 0
x = NULL;    // (2) implementation-defined (might warn)
x = nullptr; // (3) always a compile-time error
```

---

[1] In the C Standard, the macro `NULL` is defined as an **implementation-defined** integral or `void*` constant. Unlike C, C++ forbids conversions from `void*` to arbitrary pointer types and instead, prior to C++11, defined `NULL` as an *"integral constant expression rvalue of integer type that evaluates to zero"*; any integer literal (e.g., `0`, `0L`, `0U`, `0LLU`) satisfies this criterion.

[2] As of C++11, the definition of `NULL` has been expanded to — in theory — permit `nullptr` as a conforming definition; as of this writing, however, no major compiler vendors do so. Both GCC and Clang default to `0L` (`long int`) while MSVC defaults to `0` (`int`). Such definitions are unlikely to change since existing code could cease to compile or (possibly silently) present altered run-time behavior.

[3] Functions that return via argument typically do so to reserve the function's return value to communicate status. See **lakos96**, section 9.1.11, pp. 621–628, specifically the *Guideline* at the bottom of p. 621: "Be consistent about returning values through arguments (e.g., avoid declaring non-`const` reference parameters)."

```
}
// ...
return 0;    // SUCCESS
}
```

As the example above demonstrates, how we represent the notion of a **null address** matters:

1. `0` — Portable across all implementations but minimal type safety.
2. `NULL` — Implemented as a macro; added type safety (if any) is platform specific.
3. `nullptr` — Portable across all implementations and fully type-safe.

Use of `nullptr` instead of `0` or `NULL` to denote a null address maximizes type safety and readability, while avoiding both macros and implementation-defined behavior.

### 1.11.2.2 Disambiguation of `(int) 0` versus `(T*) 0` during overload resolution

The platform-dependent nature of `NULL` presents additional challenges when used to call a function whose overloads differ only in accepting a pointer or an integral type as the same positional argument (which might be the case, e.g., in a poorly designed third-party library):

```
void uglyLibraryFunction(int* p);  // (1)
void uglyLibraryFunction(int  i);  // (2)
```

Calling this function with the literal `0` will always invoke overload (2), but that might not always be what casual clients expect:

```
void f()
{
uglyLibraryFunction(0);         // unambiguously invokes (2)
uglyLibraryFunction((int*) 0);  // unambiguously invokes (1)
uglyLibraryFunction(nullptr);   // unambiguously invokes (1)
uglyLibraryFunction(NULL);      // anything! (platform-defined)
uglyLibraryFunction(0L);        // always ambiguous
uglyLibraryFunction(0U);        // always ambiguous
}
```

`nullptr` is especially useful when such problematic overloads are unavoidable because it obviates explicit casts.[4]

### 1.11.2.3 Overloading for a literal null pointer

Being a distinct type, `std::nullptr_t` can itself participate in an overload set:

```
void f(int* v);          // (1)
void f(std::nullptr_t);  // (2)
```

---

[4] N.B., Explicitly casting `0` to an appropriately typed pointer (other than `void*`) was at one time considered by some to be a best practice, especially in C.

```
void g()
{
int* ptr = nullptr;
f(ptr);      // unambiguously invokes (1)
f(nullptr);  // unambiguously invokes (2)
}
```

Given the relative ease with which a `nullptr` (above) can be converted to a typed pointer having the same null-address value, such overloads are, however, dubious when used to control essential behavior. Nonetheless, one can envision such use to, say, aid in compile-time diagnostics when passing a **null address** would otherwise result in a runtime error[5]:

```
std::size_t strlen(const char* s);
// The behavior is undefined unless s is null-terminated.

std::size_t strlen(std::nullptr_t);
// declared but not defined
```

Another arguably safe use of such an overload for a `nullptr` is to avoid a null-pointer check. However, for cases where the client knows the address is null at compile time, better ways typically exist for avoiding the (often insignificant) overhead to test for a null pointer at runtime.

### 1.11.3 Potential Pitfalls

None so far

### 1.11.4 Annoyances

None so far

### 1.11.5 See Also

None so far

### 1.11.6 Further Reading

None so far

---

[5] see also Section 1.5, “Deleted Functions”

# 1.12 `alignas`

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

## 1.12.1 Description

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

### 1.12.1.1 Restricting the alignment of a particular object

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

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

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

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

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

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

---

[1] The minimum alignment of an entity is the least restrictive memory-address boundary at which the entity can be placed and have the program continue to work properly. This value is platform dependent and often subject to compiler controls but, by default, is often well approximated by **natural alignment**; see *Appendix: Natural Alignment* on page 76.

[2] The notion of the largest supported alignment is characterized by both **maximal alignment** and the maximum **extended alignment**. **Maximal alignment** is defined as that most restrictive alignment that is valid in *all* contexts on the current platform. All fundamental and pointer types necessarily have a minimal alignment that is less than or equal to `alignof(std::max_align_t)` — typically 8 or 16. Any alignment value greater than **maximal alignment** is an **extended alignment** value. Whether any extended alignment is supported (and in which contexts) is implementation defined. On typical platforms, extended alignment will often be as large as $2^{18}$ or $2^{19}$, however implementations may warn when the alignment of a global object exceeds some maximal hardware threshold (such as the size of a physical memory page, e.g., 4096 or 8192). For **automatic variables** (defined on the program stack), making alignment more restrictive than what would naturally be employed is seldom desired because at most one thread is able to access proximately located variables there unless explicitly passed in via address to separate threads; see *Use Cases: Avoiding*

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

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

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

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

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

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

### 1.12.1.2 Restricting the alignment of a user-defined type

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

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

Notice that, for each of `class`, `struct`, and `union` above, the `sizeof` objects of that type increased to match the alignment; in the case of the `enum`, however, the size remains that of the default **underlying type** (e.g., 4 bytes) on the current platform.[3]

---

***false sharing** among distinct objects in a multi-threaded program* on page 73. Note that, in the case of `i` in the first code snippet on page 68, a conforming platform that did not support an extended alignment of 64 would be required to report an error at compile time.

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

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

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

### 1.12.1.3 Restricting the alignment of individual data members

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

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

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

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

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

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

### 1.12.1.4 Matching the alignment of another type

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

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

---

conforming implementations will eventually stop accepting the `alignas` specifier on enumerations in the future.

## 1.12.2 Use Cases

### 1.12.2.1 Creating a sufficiently aligned object buffer

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

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

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

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

void f()
{
// ...

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

// ...

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

// ...
}
```

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

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

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

// ...

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

// ...
}
```

### 1.12.2.2 Ensuring proper alignment for architecture-specific instructions

Architecture-specific instructions or compiler intrinsics might require the data they act on to have a specific alignment. One example of such intrinsics is the *Streaming SIMD Extensions (SSE)*[4] instruction set available on the x86 architecture. SSE instructions operate on groups of four 32-bit single-precision floating-point numbers at a time, which are required to be 16-byte aligned.[5] The `alignas` specifier can be used to create a type satisfying this requirement:

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

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

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

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

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

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

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

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

---

[4] **inteliig**, “Technologies: SSE”

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

```
assert(vResult.d_data[3] == 13.0f);
}
```

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

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

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

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

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

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

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

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

t0.join();
t1.join();
// Wait for both jobs to be completed.
}
```

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

```
// ...

void f()
{
```

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

// ...
```

As an empirical demonstration of the effects of **false sharing**, a benchmark program repeatedly calling `f` completed its execution seven times faster on average when compared to the same program without use of `alignas`.[6]

### 1.12.2.4 Avoiding false sharing within a single thread-aware object

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

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

// ...
};
```

Not aligning `d_head` and `d_tail` (above) to the CPU cache size might result in poor performance of the `ThreadSafeRingBuffer` because CPU cores that need to access only one of the variables will inadvertently load the other one as well, triggering expensive hardware-level coherency mechanisms between the cores' caches. On the other hand, specifying such substantially stricter alignment on consecutive data members necessarily increases the size of the object; see *Potential Pitfalls: Stricter alignment might reduce cache utilization* on page 76.

## 1.12.3 Potential Pitfalls

### 1.12.3.1 Underspecifying alignment is not universally reported

The Standard is clear when it comes to underspecifying alignment[7]:

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

---

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

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

The compiler is required to honor the specified value if it is a **fundamental alignment**,[8] so imagining how this would lead to anything other than an ill-formed program is difficult:

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

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

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

Each of the three errors above are reported by Clang, but GCC doesn’t issue so much as a warning (let alone the required error) — even in the most pedantic warning mode. Thus, one could write a program, involving statements like those above, that happens to work on one platform (e.g., GCC) but fails to compile on another (e.g., Clang).[9]

### 1.12.3.2 Incompatibly specifying alignment is IFNDR

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

```
struct Foo;                  // OK, does not specify an alignment
struct alignas(double) Foo;  // OK, must be equivalent to every definition
struct alignas(8) Foo;       // OK, all definitions must be identical.
struct alignas(8) Foo { };   // OK, equivalent to each decl. specifying alignas
struct Foo;                  // OK, has no effect
struct alignas(8) Foo;       // OK, has no effect; might warn after definition
```

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

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

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

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

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

---

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

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

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

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

### 1.12.3.3 Stricter alignment might reduce cache utilization

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

## 1.12.4 See Also

- Section 1.18, "`alignof`" — Safe C++11 feature that inspects the alignment of a given type
- Section 1.1, "Attributes" — Safe C++11 feature that shows how other attributes (following the conventional attribute notation) are used to annotate source code, improve error diagnostics, and implicitly code generation

## 1.12.5 Further Reading

None so far

## 1.12.6 Appendix

### 1.12.6.1 Natural Alignment

By default, fundamental, pointer, and enumerated types typically reside on an address boundary that divides the size of the object; we refer to such alignment as **natural alignment**[10]:

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

---

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

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

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

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

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

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

struct S4
{
char a[2];  // size 2; alignment 1
};             // size 2; alignment 1
```

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

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

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

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

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

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

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

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

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

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

Modern computers are highly complex systems, and a detailed understanding of their intricacies is unnecessary to achieve most of the performance benefits. Still, certain general themes and rough thresholds aid in understanding how to squeeze just a bit more out of the underlying hardware. In this section, we sketch fundamental concepts that are common to all modern computer hardware; although the precise details will vary, the general ideas remain essentially the same.

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

To exploit the phenomenon of **locality of reference**, computers introduce the notion of a cache that, while much faster than main memory, is also much smaller. Programs that attempt to amplify **locality of reference** will, in turn, often be rewarded with faster run times. The organization of a cache and, in fact, the number of levels of cache (e.g., L1, L2, L3,...) will vary, but the basic design parameters are, again, more or less the same. A given level of cache will have a certain total size in bytes (invariably an integral power of two). The cache will be segmented into what are called **cache lines** whose size (a smaller power of two) divides that of the cache itself. When the CPU accesses main memory, it first looks to see if that memory is in the cache; if it is, the value is returned quickly (known as a **cache hit**). Otherwise, the cache line(s) containing that data is (are) fetched (from the next higher level of cache or from main memory) and placed into the cache (known as a **cache miss**), possibly ejecting other less recently used ones.[11]

Data residing in distinct cache lines is physically independent and can be written concurrently by multiple threads. Logically unrelated data residing in the same cache line, however, is nonetheless physically coupled; two threads that write to such logically unrelated data will find themselves synchronized by the hardware. Such unexpected and typically undesirable sharing of a cache line by unrelated data acted upon by two concurrent threads is known as **false sharing**. One way of avoiding **false sharing** is to align such data on a cache-line boundary, thus rendering accidental collocation of such data on the same cache line impossible. Another (more broad-based) design approach that avoids lowering cache utilization is to ensure that data acted upon by a given thread is kept physically separate — e.g., through the use of local (arena) memory allocators.[12]

Finally, even data that is not currently in cache but resides nearby in MM can benefit from locality. The virtual address space, synonymous with the size of a `void*` (typically 64-bits on modern general-purpose hardware), has historically well exceeded the physical memory available to the CPU. The operating system must therefore maintain a mapping (in main memory) from what is resident in physical memory and what resides in secondary storage (e.g., on disc). In addition, essentially all modern hardware provides a **TLB**[13] that caches the addresses of the most recently accessed physical pages, providing yet another advantage to having the **working set** (i.e., the current set of frequently accessed pages) remain small and densely packed with relevant data.[14] What's more, dense working sets, in addition to facilitating hits for repeat access, increase the likelihood that data that is coresident on a page (or cache line) will be needed soon (i.e., in effect acting as a prefetch).[15] Table 1–1 provides a summary of typical physical parameters found in modern computers today.

---

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

[12] **lakos17**, **lakos19**, **lakos22**

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

[14] Note that memory for handle-body types (e.g., `std::vector` or `std::deque`) and especially node-based containers (e.g., `std::map` and `std::unordered_map`), originally allocated within a single page, can — through deallocation and reallocation (or even move operations) — become scattered across multiple (perhaps many) pages, thus causing what was originally a relatively small **working set** to no longer fit within physical memory. This phenomenon, known as **diffusion** (which is a distinct concept from **fragmentation**),

**Table 1–1: Various sizes and access speeds of typical memory for modern computers**

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

---

is what typically leads to a substantial runtime performance degradation (due to **thrashing**) in large, long-running programs. Such **diffusion** can be mitigated by judicious use of local arena memory allocators (and deliberate avoidance of **move operations** across disparate localities of frequent memory usage).

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

## 1.13 Delegating Constructors

Delegating constructors are constructors of a class that delegate initialization to another constructor of the same class.

### 1.13.1 Description

A **delegating constructor** is a constructor of a **user-defined type** (i.e., `class`, `struct`, or `union`) that invokes another constructor defined for the same **UDT** as part of its initialization of an object of that type. The syntax for invoking another constructor within a type is to specify the name of the type as the only element in the **member initializer list**:

```
struct S0
{
int d_i;

S0(int i) : d_i(i)        { }  // non-delegating constructor
S0()      : S0(0)         { }  // OK, delegates to S0(int)
S0(bool)  : S0(0), d_i(0) { }  // error: delegation must be on its own
};
```

Multiple delegating constructors can be chained together (one calling exactly one other) so long as cycles are avoided (see *Potential Pitfalls: Delegation cycles* on page 85). Once a *target* (i.e., invoked via delegation) constructor returns, the body of the delegator is invoked:

```
#include <iostream>  // std::cout

struct S1
{
S1(int, int)            { std::cout << 'a'; }
S1(int)      : S1(0, 0) { std::cout << 'b'; }
S1()         : S1(0)    { std::cout << 'c'; }
};

void f()
{
S1 s;  // OK, prints "abc" to stdout
}
```

If an exception is thrown while executing a nondelegating constructor, the object being initialized is considered only **partially constructed** (i.e., the object is not yet known to be in a valid state) and hence its destructor will *not* be invoked[1]:

---

[1] The destructor of a **partially constructed** object will not be invoked. However, the destructors of each successfully constructed base and of data members will still be invoked:

```
#include <iostream>

using std::cout;
struct A { A() { cout << "A() "; } ~A() { cout << "~A() "; } };
struct B { B() { cout << "B() "; } ~B() { cout << "~B() "; } };
```

```
#include <iostream>  // std::cout

struct S2
{
S2()  { std::cout << "S2() ";  throw 0; }
~S2() { std::cout << "~S2() ";          }
};

void f() try { S2 s; } catch(int) { }
// prints only "S2() " to stdout (i.e., the destructor of S2 is never
// invoked)
```

However, if an exception is thrown in the body of a delegating constructor, the object being initialized is considered **fully constructed**, as the target constructor must have returned control to the delegator; hence the overall object's destructor *will* be invoked:

```
#include <iostream>  // std::cout

struct S3
{
S3()          { std::cout << "S3() "              }
S3(int) : S3() { std::cout << "S3(int) "; throw 0; }
~S3()         { std::cout << "~S3() "             }
};

void f() try { S3 s(0); } catch(int) { }
// prints "S3() S3(int) ~S3() " to stdout
```

## 1.13.2 Use Cases

### 1.13.2.1 Avoiding code duplication among constructors

Avoiding gratuitous code duplication is considered by many to be a best practice. Having one ordinary member function call another has always been an option, but having one constructor invoke another constructor directly has not. Classic workarounds included repeating the code or else factoring the code into a private member function that would be called from multiple constructors. The drawback with this workaround is that the private

---

```
struct C : B
{
A d;

C()  { cout << "C() "; throw 0; }  // non-delegating constructor that throws
~C() { cout << "~C() ";         }  // destructor that never gets called
};

void f() try { C c; } catch(int) { }
// prints "B() A() C() ~A() ~B()" to stdout
```

Notice that base-class `B` and member `d` of type `a` were fully constructed, and so their respective destructors are called, even though the destructor for class `C` itself is never executed.

method, not being a constructor, would be unable to make use of **member initialization lists** to construct base-class and member objects efficiently. As of C++11, *delegating constructors* can be used to minimize code duplication when some of the same operations are performed across multiple constructors without having to forgo efficient initialization.

As an example, consider an `IPV4Host` class representing a network endpoint that can either be constructed by (1) a 32-bit address and a 16-bit port or (2) an IPV4 string with `XXX.XXX.XXX.XXX:XXXXX` format[2]:

```
#include <cstdint>  // std::uint16_t, std::uint32_t

class IPV4Host
{
// ...

public:
IPV4Host(std::uint32_t address, std::uint16_t port)
{
if (!connect(address, port))  // code repetition: BAD IDEA
{
throw ConnectionException{address, port};
}
}

IPV4Host(const std::string& ip)
{
std::uint32_t address = extractAddress(ip);
std::uint16_t port = extractPort(ip);

if (!connect(address, port))  // code repetition: BAD IDEA
{
throw ConnectionException{address, port};
}
}
};
```

Prior to C++11, working around such code duplication would require the introduction of a separate, subordinate (private) helper function, that would, in turn, be called by each of the constructors:

```
#include <cstdint>  // std::uint16_t, std::uint32_t

class IPV4Host
{
// ...

private:
void validate(std::uint32_t address, std::uint16_t port)  // helper function
```

---

[2] Note that this initial design might itself be suboptimal in that the representation of the IPV4 address and port value might profitably be factored out into a separate **value-semantic** class, say, `IPV4Host`, that itself might be constructed in multiple ways; see *Potential Pitfalls: Suboptimal factoring* on page 85.

```
{
if (!connect(address, port))  // factored implementation of needed logic
{
throw ConnectionException{address, port};
}
}

public:
IPV4Host(std::uint32_t address, std::uint16_t port)
{
validate(address, port);  // Invoke factored private helper function.
}

IPV4Host(const std::string& ip)
{
std::uint32_t address = extractAddress(ip);
std::uint16_t port = extractPort(ip);

validate(address, port);  // Invoke factored private helper function.
}
};
```

Alternatively, the constructor accepting a string can be rewritten to delegate to the one accepting `address` and `port`, avoiding repetition without having to delegate to a private function:

```
#include <cstdint>  // std::uint16_t, std::uint32_t

class IPV4Host
{
// ...

public:
IPV4Host(std::uint32_t address, std::uint16_t port)
{
if(!connect(address, port))
{
throw ConnectionException{address, port};
}
}

IPV4Host(const std::string& ip)
: IPV4Host{extractAddress(ip), extractPort(ip)}
{
}
};
```

Compared to the pre-C++11 workaround of introducing a private `init` function containing the duplicated logic, use of delegating constructors results in less boilerplate and fewer run-time operations, as data members (and base classes) can be initialized directly through the

**member initialization list**, rather than be *assigned-to* in the body of `init` (assuming copy assignment is even supported on that type), but see *Potential Pitfalls: Suboptimal factoring* on page 85.

## 1.13.3 Potential Pitfalls

### 1.13.3.1 Delegation cycles

If a constructor delegates to itself either directly or indirectly, the program is **ill-formed, no diagnostic required**. While some compilers can detect delegation cycles at compile time, they are not required (nor necessarily able) to do so. For example, consider a simple delegation cycle comprising two constructors:

```
struct S  // Object
{
S(int)  : S(true) { }  // delegating constructor
S(bool) : S(0)    { }  // delegating constructor
};
```

Not all popular compilers will warn you that the program above is ill-formed.[3] Therefore the programmer is responsible for ensuring that no delegation cycles are present.

### 1.13.3.2 Suboptimal factoring

The need for delegating constructors might result from initially suboptimal factoring — e.g., in the case where the same **value** is being presented in different forms to a variety of different **mechanisms**. For example, consider the `IPV4Host` class in *Use Cases* (which starts on page 82). While having two constructors to initialize the host might be appropriate, if either (1) the number of ways of expressing the same value increases or (2) the number of consumers of that value increases, we might be well advised to create a separate **value semantic** type, e.g., `IPV4Address`, to represent that value[4]:

```
#include <cstdint>  // std::uint16_t, std::uint32_t

struct IPV4Address
{
std::uint32_t d_address;
std::uint16_t d_port;

IPV4Address(std::uint32_t address, std::uint16_t port)
: d_address{address}, d_port{port}
{
```

---

[3] GCC 10.x does not detect this delegation cycle at compile time and produces a binary that, if run, will necessarily exhibit **undefined behavior**. Clang 10.x, on the other hand, halts compilation with a helpful error message:

```
error: constructor for S creates a delegation cycle
```

[4] The notion that each component in a subsystem ideally performs one focused function well is sometimes referred to as separation of (logical) concerns or fine-grained (physical) factoring; see **lakos20**, sections 0.4, 3.2.7, and 3.5.9, pp. 20–28, 529–530, and 674–676, respectively.

```
}

IPV4Address(const std::string& ip)
: IPV4Address{extractAddress(ip), extractPort(ip)}
{
}
};
```

Note that `IPV4Address` itself makes use of delegating constructors but as a purely private, encapsulated implementation detail. With the introduction of `IPV4Address` into the codebase, `IPV4Host` (and similar components requiring an `IPV4Address` value) can be redefined to have a single constructor (or other previously overloaded member function) accepting an `IPV4Address` object as an argument.

### 1.13.4 Annoyances

None so far

### 1.13.5 See Also

None so far

### 1.13.6 Further Reading

None so far

## 1.14 Local Types as Template Arguments

Local (i.e., function-scope) and unnamed (e.g., *lambda expression*, a.k.a. “closure”) types can, as of C++11, be used (like all other types) as arguments to templates.

### 1.14.1 Description

Historically, types without **linkage** (i.e., local and unnamed types) were forbidden as template arguments due to implementability concerns using the compiler technology available at that time.[1] Modern C++ lifts this restriction, making use of local or unnamed types consistent with nonlocal, named ones, thereby obviating the need to gratuitously name or enlarge the scope of a type.

```
template <typename T>
void f(T) { };            // function template

template <typename T>
class C { };              // class template

struct { } obj;           // object obj of unnamed C++ type

void g()
{
struct S { };         // local type

f(S());               // OK in C++11; was error in C++03
f(obj);               // OK in C++11; was error in C++03

C<S>             cs;  // OK in C++11; was error in C++03
C<decltype(obj)> co;  // OK in C++11; was error in C++03
}
```

Notice that we have used the (C++11) `decltype` *keyword* (see Section 1.4, “`decltype`”) to extract the unnamed type of the object `obj`.

These new relaxed rules for template arguments are essential to the ergonomics of *lambda expressions* (see Section 2.8, “Lambdas”), as such types are both unnamed and local in typical usage:

```
#include <algorithm>  // std::sort

struct Person { std::string d_name; };

void sortByName(std::vector<Person>& people)
{
std::sort(people.begin(), people.end(),
[](const Person& lhs, const Person& rhs)
{
```

---

[1] TODO: Alisdair

```
return lhs.d_name < rhs.d_name;
});
}
```

In the example above, the lambda expression passed to the `std::sort` algorithm is a local unnamed type, and the algorithm itself is a function template.

## 1.14.2 Use Cases

### 1.14.2.1 Encapsulating a type within a function

Limiting the scope and visibility of an **entity** to the body of a function actively prevents its direct use, even when the function body is exposed widely — say, as an `inline` function or function template defined within a header file.

Consider, for instance, an implementation of Dijkstra's algorithm that uses a local type to keep track of metadata for each vertex in the input graph (i.e., the distance of a vertex from the source of the search and whether a vertex is included in the shortest path or not):

```
// dijkstra.h

inline int dijkstra(std::vector<Vertex>* path, const Graph& graph)
{
struct VertexMetadata         // implementation-specific helper class
{
int  d_distanceFromSource;
bool d_inShortestPath;
};

std::vector<VertexMetadata> vertexMetadata(graph.numNodes());
// standard vector of local VertexMetadata objects -- one per vertex

// ... (body of algorithm)
}
```

Defining `VertexMetadata` outside of the body of `dijkstra` — e.g., to comply with C++03 restrictions — would make that implementation-specific helper class directly accessible to anyone including the `dijkstra.h` header file. As Hyrum's law[2] suggests, if the implementation-specific `VertexMetadata` detail is defined outside the function body, it is to be expected that some user somewhere will depend on it in its current form, making it problematic, if not impossible, to change.[3] Conversely, encapsulating the type within the function body avoids unintended use by clients, while improving human cognition by colocating the definition of the type with its sole purpose.[4]

---

[2] "With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody": see **wight**.

[3] The C++20 *modules* facility enables the encapsulation of helper types (such as `metadata` in the `dijkstra.h` example on this page) used in the implementation of other locally defined types or functions, even when the helper types appear at namespace scope within the module.

[4] For a detailed discussion of malleable versus stable software, see **lakos20**, section 0.5, pp. 29-43.

### 1.14.2.2 Instantiating templates with local function objects as type arguments

Suppose that we have a program that makes wide use of an aggregate data type, `City`:

```
struct City
{
int         d_uniqueId;
std::string d_name;
};
```

Consider now the task of writing a function to print unique elements of an `std::vector<City>`, ordered by name:

```
void printUniqueCitiesOrderedByName(const std::vector<City>& cities)
{
struct OrderByName
{
bool operator()(const City& lhs, const City& rhs) const
{
return lhs.d_name < rhs.d_name;
// increasing order (subject to change)
}
};

const std::set<City, OrderByName> tmp(cities.begin(), cities.end());

std::copy(tmp.begin(), tmp.end(),
std::ostream_iterator<City>(std::cout, '\n'));
}
```

Absent any countervailing reasons to make the `OrderByName` function object more generally available, rendering its definition alongside the one place where it is used — i.e., directly within function scope — again enforces and readily communicates its tightly encapsulated (and therefore *malleable*) status.

### 1.14.2.3 Configuring algorithms via lambda expressions

Suppose we are representing a 3D environment using a *scene graph*[5] and managing the graph’s nodes via an `std::vector` of `SceneNode` objects. Our `SceneNode` class supports a variety of `const` member functions used to query its status (e.g., `isDirty` and `isNew`). Our task is to implement a **predicate function**, `mustRecalculateGeometry`, that returns `true` if and only if at least one of the nodes is either “dirty” or “new.”

These days, we might reasonably elect to implement this functionality using the (C++11) standard algorithm `std::any_of`[6]:

```
template <typename InputIterator, typename UnaryPredicate>
bool any_of(InputIterator first, InputIterator last, UnaryPredicate pred);
```

---

[5] A *scene graph* data structure, commonly used in computer games and 3D-modeling software, represents the logical and spatial hierarchy of objects in a scene.

[6] **cppreferencea**

```
// Return true if any of the elements in the range satisfies pred.
```

Prior to C++11, however, use of a function template, such as `any_of`, would have required a separate function or function object (defined *outside* of the scope of the function):

```
namespace {

struct IsNodeDirtyOrNew
{
bool operator()(const SceneNode& node) const
{
return node.isDirty() || node.isNew();
}
};

}  // close unnamed namespace

bool mustRecalculateGeometry(const std::vector<SceneNode>& nodes)
{
return std::any_of(nodes.begin(), nodes.end(), IsNodeDirtyOrNew());
}
```

In C++11, not only can we embed the function object within the scope of the function but, by using a lambda expression, we can remove much of the boilerplate, including the enclosing `struct`:

```
bool mustRecalculateGeometry(const std::vector<SceneNode>& nodes)
{
return std::any_of(nodes.begin(),             // start of range
nodes.end(),               // end of range
[](const SceneNode& node)  // lambda expression
{
return node.isDirty() || node.isNew();
}
);
}
```

By creating a **closure** of unnamed type via a lambda expression, unnecessary boilerplate, excessive scope, and even local symbol visibility are avoided.

## 1.14.3 Potential Pitfalls

None so far

## 1.14.4 Annoyances

None so far

## 1.14.5 See Also

- Section 2.8, “Lambdas” — Conditionally safe C++11 feature providing strong practical motivation for the relaxations discussed here

- Section 1.4, “`decltype`” — Safe C++11 feature that allows developers to query the type of any expression or entity, including objects with unnamed types

### 1.14.6 Further Reading

None so far

## 1.15 long long

`long long` is a **fundamental integral type** guaranteed to have (at least) 64 bits on all platforms.

### 1.15.1 Description

The **integral type** `long long` and its companion type `unsigned long long` are the only two **fundamental integral types** in C++ that are guaranteed to have at least 64 bits on all conforming platforms[1]:

```
#include <climits>  // CHAR_BIT (a.k.a.~8, see below)

long long          a;  // sizeof(a) * CHAR_BIT >= 64
unsigned long long b;  // sizeof(b) * CHAR_BIT >= 64

static_assert(sizeof(a) == sizeof(b), "");
// I.e., a and b necessarily have the same size in every program.
```

On all conforming platforms, `CHAR_BIT` — the number of bits in a byte — is at least 8 and, on virtually all commonly available commercial platforms today, is exactly 8, as is `sizeof(long long)`.

The corresponding integer-literal suffixes indicating type `long long` are `ll` and `LL`; for `unsigned long long`, any of eight alternatives are accepted: `ull`, `ULL`, `uLL`, `Ull`, `llu`, `LLU`, `LLu`, `llU`[2]:

```
auto i = 0LL;  // long long, sizeof(i) * CHAR_BIT >= 64
auto u = 0uLL  // unsigned long long, sizeof(u) * CHAR_BIT >= 64
```

For a historical perspective on how integral types have evolved (and continue to evolve) over time, see *Appendix: Historical Perspective on the Evolution of Use of Fundamental Integral Types* on page 95.

### 1.15.2 Use Cases

#### 1.15.2.1 When your pedestrian four-byte `int` might not cut it

Deciding when an `int` (i.e., exactly 32 bits) is big enough is often a nonissue. For most common things we deal with day to day — miles on our car, years of age, bottles of wine — having more than about a billion of them just isn't worth thinking about, at least not in the interface.[3] Sometimes the size of the virtual address space for the underlying architecture itself dictates how large an integer you will need. For example, specifying the *distance* between two pointers into a contiguous array or the size of the array itself could, on a 64-bit platform, well exceed the size of an `int` or `unsigned int`, respectively. Using either `long long` or `unsigned long long` here would, however, not be indicated as the respective platform-dependent integer types (`typedef`s) `std::ptrdiff_t` and `std::size_t` are provided expressly for such use (and avoid wasting space where it cannot be used by the underlying hardware).

---

[1] `long long` has been available in C since the C99 standard, and many C++ compilers supported it as an extension prior to C++11.

[2] Note that `long long` and `unsigned long long` are also candidates for the type of an integer literal having a large enough value. As an example, the type of the literal `2147483648` (one more than the upper bound of a 32-bit integer) is likely to be `long long` on a 32-bit platform.

[3] For efficient storage in a `class` or `struct`, however, we may well decide to represent such quantities more compactly using a `short` or `char`; see also the aliases found in C++11's `<cstdint>`.

Occasionally, however, the decision of whether to use an `int` is neither platform dependent nor clear cut, in which case using an `int` is almost certainly a bad idea. As part of a financial library, suppose we were asked to provide a function that, given a date, returns the number of shares of some particular stock, identified by its security id (`SecId`) traded on the New York Stock Exchange (NYSE).[4] Since the average daily rate for even the most heavily traded stocks (roughly 70 million) appears to be well under the maximum value a signed `int` supports (more than 2 billion), we might at first think to write the function returning an `int`:

```
int volYMD(SecId equity, int year, int month, int day);  // (1) BAD IDEA
```

One obvious problem with this interface is that the daily fluctuations in turbulent times might exceed the maximum value representable by a 32-bit `int`, which, unless detected internally, would result in **signed integer overflow**, which is both **undefined behavior** and a potential security hole.[5] What's more, the growth rate of some companies, especially technology companies, such as AAPL, GOOG, FB, AMZN, and MSFT, has been at times seemingly exponential. To gain an extra insurance factor of two, we might opt to replace the return type `int` with an `unsigned int`:

```
unsigned volYND(SecId stock, int year, int month, int day);  // (2) BAD IDEA!
```

Use of an `unsigned int`, however, simply delays the inevitable as the number of shares being traded is almost certainly going to grow over time.

Furthermore, the algebra for unsigned quantities is entirely different from what one would normally expect from an `int`. For example, if we were to try to express the day-over-day increase in volume by subtracting two calls to this function and if the number of shares traded were to have decreased, then the `unsigned int` difference would wrap and the result would be a (typically) large unsigned **garbage value**.[6]

If we happen to be on a 64-bit platform, we might choose to return a `long`:

```
long volYMD(SecId stock, int year, int month, int day);  // (3) NOT A GOOD IDEA
```

---

[4] The NYSE consists of 2400 different (Equity) securities. The average daily trading volume (the number of shares traded on a given day) on the NYSE is typically between 2? and 6? billion shares per day, with the maximum volume reaching ??? on ???. (TODO, TBD – will fill this in much later. NOTE: from Harry: "it looks like the numbers you are referencing are for the NYSE Composite – i.e., the volume of all shares traded in companies listed at the NYSE. That number is significantly higher than the number of shares traded on the NYSE exchange. That is because NYSE shares can trade on other exchanges. With that proviso, over the past 5 years, NYSE Composite trading volume has averaged 3.8 billion shares per day. The highest volume day was March 20, 2020, when just over 9 billion shares were traded.")

[5] **Signed integer overflow** is among the most pervasive kinds of defects enabling avenues of deliberate attack from outside sources. For an overview of integer overflow in C++, see **ballman**. For a more focused discussion of secure coding in CPP using CERT standards, see **seacord13**, section x, pp. yy-zz.

[6] Because integer literals are themselves of type `int` and not `unsigned`, comparing an unsigned value with a negative signed one does not typically go well; hence, many compilers will warn when the two types are mixed, which itself is problematic.

The problems using `long` as the return type are that it (1) is not (yet) generally considered a **vocabulary type** (see *Appendix: Historical Perspective on the Evolution of Use of Fundamental Integral Types* on page 95), and (2) would reduce portability (see *Potential Pitfalls: Relying on the relative sizes of* `int`*,* `long`*, and* `long long` on page 94).

Prior to C++11, we might have considered returning a `double`:

```
double volYMD(SecId stock, int year, int month, int day);  // (4) OK
```

At least with `double` we know that we will have (at no additional size) sufficient precision (53 bits) to express integers accurately into the quadrillions, which will certainly cover us for any foreseeable future. The main drawback is that `double` doesn't properly describe the nature of the type that we are returning — i.e., a whole integer number of shares — and so its algebra, although not as dubious as `unsigned int`, isn't ideal either.

With the advent of C++11, we might consider using one of the type aliases in `<cstdint>`:

```
std::int64_t volYMD(SecId stock, int year, int month, int day);  // (4) OK
```

This choice addresses most of the issues discussed above except that, instead of being a specific C++ type, it is a platform-dependent alias that is likely to be a `long` on a 64-bit platform and almost certainly a `long long` on a 32-bit one. Such exact size requirements are often necessary for packing data in structures and arrays but are not as useful when reasoning about them in the interfaces of functions where having a common set of fundamental **vocabulary types** becomes much more important (e.g., for interoperability).

All of this leads us to our final alternative, `long long`:

```
long long volYMD(SecId stock, int year, int month, int day);  // (5) GOOD IDEA
```

In addition to being a signed fundamental integral type of sufficient capacity on all platforms, `long long` is the same C++ type *relative* to other C++ types on all platforms.

## 1.15.3 Potential Pitfalls

### 1.15.3.1 Relying on the relative sizes of `int`, `long`, and `long long`

As discussed at some length in *Appendix: Historical Perspective on the Evolution of Use of Fundamental Integral Types* on page 95, the fundamental integral types have historically been a moving target. On older, 32-bit platforms, a `long` was often 32 bits and, prior to C++11, a (nonstandard) `long long` (or its platform-dependent equivalent) was needed to ensure that 64 bits were available. When the correctness of code depends on either `sizeof(int) < sizeof(long)` or `sizeof(long) < sizeof(long long)`, portability is needlessly restricted. Relying instead on only the guaranteed[7] property that `sizeof(int) < sizeof(long long)` avoids such portability issues since the relative sizes of the (fundamental) `long` and `long long` integral types continue to evolve.

When precise control of size *in the implementation* (as opposed to in the interface) matters, consider using one of the standard signed (`int*n*_t`) or unsigned (`uint*n*_t`) integer aliases (`typedef`s) provided (since C++11) in `<cstdint>` and summarized here in Table 1–1.

---

[7] Due to the unfathomable amount of software that would stop working if an `int` were ever anything but exactly *four* bytes, we — along with the late Richard Stevens of Unix fame (see **stevens93**, section 2.5.1., pp. 31–32, specifically row 6, column 4, Figure 2.2, p. 32) — are prepared to *guarantee* that it will never become as large as a `long long` for any general-purpose computer.

**Table 1–1: Useful `typedef`s found in `<cstdint>` (since C++11)**

| Exact Size | Fastest (signed) integral type having at least N bits | Smallest (signed) integer type having at least N bits |
|---|---|---|
| `int8_t` | `int_fast8_t` | `int_least8_t` |
| `int16_t`[a] | `int_fast16_t` | `int_least16_t` |
| `int32_t` | `int_fast32_t` | `int_least32_t` |
| `int64_t` | `int_fast64_t` | `int_least64_t` |

[a] optional

Note: Also see `intmax_t`, the maximum width integer type, which might be none of the above.

### 1.15.4 See Also

- Section 1.25, "Digit Separators" — Safe C++11 feature that can help with visually separating digits of large `long long` literals
- Section 1.2, "Binary Literals" — Safe C++11 feature that allows programmers to specify binary constants directly in the source code; large binary values might only fit in a `long long` or even `unsigned long long`

### 1.15.5 Further Reading

None so far

### 1.15.6 Appendix: Historical Perspective on the Evolution of Use of Fundamental Integral Types

The designers of C got it right back in 1972 when they created a portable `int` type that could act as a bridge from a single-word (16-bit) integer, `short`, to a double-word (32-bit) integer, `long`. Just by using `int`, one would get the optimal space versus speed trade-off as the 32-bit computer *word* was on its way to becoming the norm.[8]

During the late 1980s and into the 1990s, the word size of the machine and the size of an `int` were synonymous.[9] As cost of *main memory* was decreasing exponentially throughout the final two decades of the 20th century,[10] the need for a much larger *virtual address space* quickly followed. Intel began its work on 64-bit architectures in the early 1990s and realized one a decade later. As we progressed into the 2000s, the common notion of *word size* — i.e., the width (in bits) of typical registers within the CPU itself — began to shift from "the size of an `int`" to "the size of a simple (nonmember) pointer type," e.g., `8 * sizeof(void*)`, on the host platform. By this time, 16-bit `int` types (like 16-bit architectures) were long gone, but `long` was still expected to be 32 bits on a 32-bit platform.[11]

---

[8] The Motorola 68000 series (c. 1979) was a hybrid *CISC* architecture employing a 32-bit instruction set with 32-bit registers and a 32-bit external data bus; internally, however, it used only 16-bit ALUs and a 16-bit data bus.

[9] Some of the earlier mainframe computers, such as IBM 701 (c. 1954), had a word size of 36 characters (1) to allow accurate representation of a signed 10-digit decimal number or (2) to hold up to six 6-bit characters. Smaller computers, such as Digital Equipment Corporation's PDP-1/PDP-9/PDP-15 used 18-bit words (so a double word held 36-bits); memory addressing, however, was limited to just 12–18 bits (i.e., a maximum 4K–256K 18-bit words of *DRAM*). With the standardization of 7-bit ASCII (c. 1967), its adoption throughout the 1970s, and its most recent update (c. 1986), the common typical notion of character size moved from 6 to 7 bytes. Some early conforming implementations (of C) would choose to set `CHAR_BIT` to 9 to allow two characters per half word. (On some early vector-processing computers, `CHAR_BIT` is 32, making every type, including a `char`, at least a 32-bit quantity.) As double-precision floating (and floating-point coprocessors) for type `double` became typical for scientific calculations, machine architectures naturally evolved from 9-, 18-, and 36-bit words to the familiar 8-, 16-, 32-, and now 64-bit addressable

Something new was needed to mean at least 64-bits on all platforms. Enter `long long`. We have now come full circle. On 64-bit platforms, an `int` is still 4 bytes, but a `long` is now — for practical reasons — typically 8 bytes unless requested explicitly[12] to be otherwise. To ensure portability until 32-bit machines go the way of 16-bit ones, we have `long long` to (1) provide a common *vocabulary type*, (2) make our intent clear, and (3) avoid the portability issue for at least the next decade or two; still, see *Potential Pitfalls: Relying on the relative sizes of* `int`*,* `long`*, and* `long long` on page 94 for some alternative ideas.

---

integer words we have today. Apart from embedded systems and *DSPs*, a `char` is now almost universally considered to be exactly 8 bits. Instead of scrupulously and actively using `CHAR_BIT` for the number of bits in a `char`, consider statically asserting it instead:

```
static_assert(CHAR_BIT == 8, "A char is not 8-bits on this CrAzY platform!");
```

[10] Moore's law (c. 1965) — the observation that the number of transistors in densely packed integrated circuits (e.g., DRAM) grows exponentially over time, doubling every 1–2 years or so — held for nearly a half century, until finally saturating in the 2010s.

[11] Sadly, `long` was often used (improperly) to hold an address; hence, the size of `long` is associated with a de facto need (due to immeasurable amounts of legacy code) to remain in lockstep with pointer size.

[12] On 64-bit systems, `sizeof(long)` is typically 8 bytes. Compiling with the `-m32` flag on either GCC or Clang emulates compiling on a 32-bit platform: `sizeof(long)` is likely to be 4, while `sizeof(long long)` remains 8.

## 1.16 Alias Declarations and Alias Templates

Alias declarations and alias templates provide an expanded use of the `using` keyword, offering an alternative syntax (to `typedef`) for creating a **type alias** that can itself be a template.

### 1.16.1 Description

The keyword `using` has historically supported the introduction of an alias for a named entity (e.g., type, function, or data) from some named scope into the current one; see *Appendix: Brief Review of (C++03)* `using` *Declarations* on page 101. As of C++11, we can employ the `using` keyword to achieve everything that could previously be accomplished with a `typedef` declaration but in a syntactic form that many people find more natural and intuitive (but that offers nothing profoundly new):

```
using Type1 = int;     // equivalent to typedef int Type1;
using Type2 = double;  // equivalent to typedef double Type2;
```

In contrast to `typedef`, the name of the synonym created via the `using` syntax always appears on the left side of the `=` token and separate from the type declaration itself — the advantage of which becomes apparent with more involved types, such as *pointer-to-functions*, *pointer-to-member-function*, or *pointer-to-data-member*:

```
struct S { int i; void f(); };  // user-defined type S defined at file scope

using Type3 = void(*)();        // equivalent to typedef void(*Type3)();
using Type4 = void(S::*)();     // equivalent to typedef void(S::*Type4)();
using Type5 = int S::*;         // equivalent to typedef int S::*Type5;
```

Just as with a `typedef`, the name representing the type can be qualified, but the symbol representing the synonym cannot:

```
namespace N { struct S { }; }  // original type S defined with namespace N

using Type6 = N::S;            // equivalent to typedef N::S Type6;
using ::Type7 = int;           // Error: the alias's name must be unqualified.
```

Unlike a `typedef`, however, a type alias employing `using` can itself be a template, known as an *alias template*:

```
template <typename T>
using Type8 = T;  // "identity" alias template

Type8<int>    i;  // equivalent to int i;
Type8<double> d;  // equivalent to double d;
```

Note, however, that neither partial nor explicit specialization of alias templates is supported:

```
template <typename, typename>   // general alias template
using Type9 = char;             // OK
```

```
template <typename T>          // attempted partial specialization of above
using Type9<T, int> = char;    // error: expected = before < token

template <>                    // attempted full specialization of above
using Type10<int, int> = char; // error: expected unqualified-id before using
```

Used in conjunction with existing class templates, alias templates allow programmers to *bind* one or more template parameters to a fixed type, while leaving others open:

```
#include <utility>  // std::pair

template <typename T>
using PairOfCharAnd = std::pair<char, T>;
// alias template that binds char to the first type parameter of std::pair

PairOfCharAnd<int>    pci;  // equivalent to std::pair<char, int> pci;
PairOfCharAnd<double> pcd;  // equivalent to std::pair<char, double> pcd;
```

Finally, note that the equivalent functionality of alias templates can be achieved in C++03, though with additional boilerplate code at both the point of definition and the call site:

```
template <typename T>
struct PairOfCharAnd
// template class holding an alias, Type, to std::pair<char, T>
{
typedef std::pair<char, T> Type;
// type alias binding char to the first type parameter of std::pair
};

PairOfCharAnd<int>::Type    pci;  // equivalent to std::pair<char, int> pci;
PairOfCharAnd<double>::Type pcd;  // equivalent to std::pair<char, double> pcd;
```

## 1.16.2 Use Cases

### 1.16.2.1 Simplifying convoluted `typedef` declarations

Complex `typedef` declarations involving pointers to functions, member functions, or data members require looking in the middle of the declaration to find the alias name. As an example, consider a *callback* type alias intended to be used with asynchronous functions:

```
typedef void(*CompletionCallback)(void* userData);
```

Developers coming from a background other than C or C++03 might find the above declaration hard to parse since the name of the alias (`CompletionCallback`) is embedded in the function pointer type. Replacing `typedef` with `using` results in a simpler, more consistent formulation of the same alias:

```
using CompletionCallback = void(*)(void* userData);
```

The `CompletionCallback` alias declaration (above) reads almost completely left-to-right,[1] and the name of the alias is clearly specified after the `using` keyword.

---

[1] In order to make the `CompletionCallback` alias read left-to-right, a trailing return (see Section 1.8,

### 1.16.2.2 Binding template arguments

An alias template can be used to *bind* one or more template parameters of, say, a commonly used class template, while leaving the other parameters open to variation. Suppose, for example, we have a class, `UserData`, that contains several (e.g., four) distinct instances of `std::map` — each having the same key type, `UserId`, but with different payloads:

```
class UserData  // class having excessive code repetition (BAD IDEA)
{
private:
std::map<UserId, Message>          d_messages;
std::map<UserId, Photos>           d_photos;
std::map<UserId, Article>          d_articles;
std::map<UserId, std::set<UserId>> d_friends;
};
```

The example above, though clear and regular, involves significant repetition, making it more difficult to maintain should we later opt to change data structures. If we were to instead use an **alias template** to bind the `UserId` type to the first type parameter of `std::map`, we could both (1) reduce code repetition and (2) enable the programmer to consistently replace `std::map` to another container (e.g., `std::unordered_map`[2]) by performing the change in only one place:

```
class UserData  // class with well-factored implementation (GOOD IDEA)
{
private:
template <typename V>                  // using a template alias to bind
using Mapping = std::map<UserId, V>;   // UserId as the key type

Mapping<Message>          d_messages;
Mapping<Photos>           d_photos;
Mapping<Article>          d_articles;
Mapping<std::set<UserId>> d_friends;
};
```

### 1.16.2.3 Providing a shorthand notation for type traits

Alias templates can provide a shorthand notation for **type traits**, avoiding **boilerplate code** in the usage site. As an example, consider a simple type trait that adds a pointer to a given type (akin to `std::add_pointer`):

---

“Trailing Function Return Types”) can be used:

```
using CompletionCallback = auto(*)(void* userData) -> void;
```

The alias declaration above can be read as, “`CompletionCallback` is an alias for a pointer to a function taking a `void*` parameter named `userData` and returning `void`.”

[2] An `std::unordered_map` is an STL container type that became available on all conforming platforms along with C++11. The functionality is similar except that, since it is not required to support ordered traversal or (worst case) `O[log(n)]` lookups and `O[n*log(n)]` insertions, `std::unordered_map` can be implemented as a hash table instead of a balanced tree, yielding significantly faster average access times. See **cppreferenceb**.

```
template <typename T>
struct AddPointer
{
typedef T* Type;
};
```

To use the trait above, the `AddPointer` class template must be instantiated and its nested `Type` alias must be accessed by prepending the `typename` keyword:

```
void f()
{
int i;
typename AddPointer<int>::Type p = &i;
}
```

The syntactical overhead of `AddPointer` can be removed by creating an alias template for its nested type alias, such as `AddPointer_t`[3]:

```
template <typename T>
using AddPointer_t = typename AddPointer<T>::Type;
```

Using `AddPointer_t` instead of `AddPointer` results in shorter code devoid of boilerplate:

```
void g()
{
int i;
AddPointer_t<int> p = &i;
}
```

## 1.16.3 Potential Pitfalls

None so far

## 1.16.4 Annoyances

None so far

## 1.16.5 See Also

- Section 1.19, “Inheriting Constructors” — Safe C++11 feature providing another meaning for the `using` keyword to allow base-class constructors to be invoked as part of the derived class
- Section 1.8, “Trailing Function Return Types” — Safe C++11 feature providing an alternative syntax for function declaration, which can help improve readability in type aliases and alias templates involving function types

---

[3] Note that, since C++14, all the standard type traits defined in the `<type_traits>` header provide a corresponding alias template with the goal of reducing boilerplate code. For instance, C++14 introduces the `std::remove_reference_t` alias template for the C++11 `std::remove_reference` type trait:

```
typename std::remove_reference<int&>::type i0 = 5; // OK in both C++11 and C++14
std::remove_reference_t<int&> i1 = 5;              // OK in C++14
```

### 1.16.6 Further Reading

None so far

### 1.16.7 Appendix: Brief Review of (C++03) `using` Declarations

The `using` keyword serves another, similar purpose: It introduces an alias for a (named) entity from a distinct (named) scope into the current scope. The first usage category for employing `using` to create local aliases is with respect to other namespaces:

```
namespace N  // namespace containing various named constructs
{
void f();          // (0) overloaded function f declared at namespace scope
void f(int);       // (1)      "         "       "      "    "       "       "
void f(double);    // (2)      "         "       "      "    "       "       "
void g();          // (3) function g declared at namespace scope
void h();          // (4)     "    h    "       "     "       "
int x;             // (5) integer variable x declared at namespace scope
int y;             // (6)      "       "     y     "     "     "      "
class C;           // (7) class C declared but not defined at namespace scope
};

void h();             // (8) function h declared at file (aka global) scope

void client1()        // client illustrating usage w.r.t. namespaces
{
N::f();           // OK, invokes (0) above
N::f(1);          // OK, invokes (1) above
N::f(2.0);        // OK, invokes (2) above
f(2.0);           // Error: function f is not found.
using N::f();     // Error: using must apply to all overloads at once.
using N::f;       // OK, creates local aliases for all three f overloads
f();              // OK, invokes (0) above
f(1);             // OK, invokes (1) above
f(2.0);           // OK, invokes (2) above

x = 3;            // error: variable x not found
N::x = 3;         // OK, assigns 3 to (5) above

using N::x;       // OK, creates local alias for x
x = 4;            // OK, assigns 4 to (5) above
y = 5;            // error: variable y not found
g();              // error: function g not found
C *p;             // error: Class C not found
N::C *p;          // OK, creates pointer p to incomplete type C (8) above

using namespace N;
// OK, create local aliases for all named entities in namespace N.

y = 6;          // OK, assigns 6 to (6) above
g();           // OK, invokes (3) above
h();           // Error: alias for h is ambiguous; (4) or (8) above.
::h();         // OK, invokes (4) above
N::h();        // OK, invokes (8) above
C *q;          // // OK, creates pointer q to incomplete type C (8) above
}
```

The second usage category for employing `using` to create local aliases is with respect to `public` (or `protected`) members of *privately* (or *protectedly*) inherited base classes into a `public` (or `protected`) region of the derived class[4]:

```
struct B   // base class having various public named entities
{
void fb();         // (10) overloaded member function
void fb(int);      // (11)     "        "       "
void fb(double);   // (12)     "        "       "
void gb();         // (13) member function
static void hb();  // (14) static member function
typedef int Tb;    // (15) type alias for an integer
int xb;            // (16) integer data member
int yb;            // (17) integer data member
};

struct D : private B  // class aliasing private constructs via using
{
using B::fb;      // local aliases for all three overloads of fb
using B::hb;      // local alias for static member function hb
using B::xb;      // local alias for int data member xb
using B::Tb;      // local alias for int type alias

protected:
using B::yb;      // protected local alias for int data member yb
};

void client2()  // client illustrating usage w.r.t. inheritance
{
D d;        // Create an instance of derived type D.
d.fb();     // OK, alias created by using B::fb  invokes (10) above.
d.fb(1);    // OK,   "     "     "     "     "     invokes (11) above.
d.fb(2.0);  // OK,   "     "     "     "     "     invokes (12) above.
d.gb();     // Error: gb is privately inherited without using declaration.
d.hb();     // OK, alias created by using B::hb  invokes (14) above.
D::hb();    // OK,   "      "    "     "     "     invokes (14) above.
D::Tb i;    // OK,   "      "    "     "  B::Tb  aliases (15) above.
D::xb = 1;  // OK,   "      "    "     "  B::xb  assigns (16) above.
D::yb = 1;  // Error, using for yb is protected, not public.
}
```

---

[4] The alternatives, shown here in parentheses, are provided for technical accuracy but are unlikely to be useful in practice.

Finally, for completeness, we note that the using directive for `yb` in the `protected` region of `D` leaves the local alias for `yb` in `D` accessible to classes that are derived from `D`:

```
struct DD : D  // doubly derived class accessing protected alias
{
DD(int v) { yb = v };
// OK, using yb in D exposes protected alias; assigns (17).
};
```

## 1.17 Explicit Conversion Operators

Ensure that a (user-defined) type is convertible to another type only in contexts where the conversion is made obvious in the code.

### 1.17.1 Description

Though sometimes desirable, implicit conversions achieved via (user-defined) *conversion functions* — either (1) **converting constructors** (accepting a single argument) or (2) **conversion operators** — can also be problematic, especially when the conversion involves a commonly used type (e.g., `int` or `double`)[1]:

```
class Point  // implicitly convertible from an int or to a double
{
int d_x, d_y;

public:
Point(int x = 0, int y = 0);  // default, conversion, & value constructor
// ...
operator double() const;  // Return distance from origin as a double.
};
```

As ever, calling a function `g` that takes a `Point` but accidentally passing an `int` can lead to surprises:

```
void g0(Point p);         // arbitrary function taking a Point object by value
void g1(const Point& p);  // arbitrary function taking a Point by const reference

void f1(int i)
{
g0(i);  // oops, called g0 with Point(i, 0) by mistake
g1(i);  // oops, called g1 with Point(i, 0) by mistake
}
```

This problem could have been solved even in C++98 by declaring the constructor to be `explicit`:

```
explicit Point(int x = 0, int y = 0);  // explicit converting constructor
```

If the conversion is desired, it must now be specified explicitly:

```
void f2(int i)
{
g0(i)          // error: could not convert i from int to Point
g1(i);         // error: invalid initialization of reference type
g0(Point(i));  // OK
g1(Point(i));  // OK
}
```

---

[1] Use of a conversion operator to calculate distance from the origin in this unrealistically simple `Point` example is for didactic purposes only. In practice, we would typically use a named function for this purpose; see *Potential Pitfalls: Sometimes a named function is better* on page 109.

The companion problem stemming from an *implicit conversion operator*, albeit less severe, remained:

```
void h(double d);

double f3(const P& p)
{
h(p);      // OK? Or maybe called h with a "hypotenuse" by mistake
return p;  // OK? Or maybe this is a mistake too.
}
```

As of C++11, we can now use the **`explicit` specifier** when declaring **conversion operators** (as well as **converting constructors**), thereby forcing the client to request conversion explicitly — e.g., using **direct initialization** or `static_cast`):

```
struct S0 { explicit operator int(); };

void g()
{
S0 s0;
int i = s0;                      // error (copy initialization)
double d = s0;                   // error (copy initialization)
int j = static_cast<int>(s0);    // OK (static cast)
if (s0) { }                      // error (contextual conversion to bool)
int k(s0);                       // OK (direct initialization)
double e(s0);                    // error (direct initialization)
}
```

In contrast, had the conversion operator above not been declared to be `explicit`, all conversions shown above would compile:

```
struct S1 { /* implicit */ operator int(); };

void f()
{
S1 s1;
int i = s1;                      // OK (copy initialization)
double d = s1;                   // OK (copy initialization)
int j = static_cast<int>(s1);    // OK (static cast)
if (s1) { }                      // OK (contextual conversion to bool)
int k(s1);                       // OK (direct initialization)
double e(s1);                    // OK (direct initialization)
}
```

Additionally, the notion of **contextual convertibility to `bool`**[2] applicable to arguments of logical operations (e.g., `&&`, `||`, and `!`) and conditions of most control-flow constructs

---

[2] Since the early days of C++, a common idiom to test for validity of an object has been to use it in a context where it can (implicitly) convert itself to a type whose value can be interpreted (contextually) as a boolean, with `true` implying validity (and `false` otherwise). Implicit conversion to `bool` (an integral type) was considered too dangerous, so the cumbersome **safe-`bool` idiom** was used instead, converting to a type that — while contextually convertible to `bool` — could not (by design) participate in any other operations. While making the conversion to `bool` (or `const bool`) `explicit` solves the safety issue, the benefit of the

(e.g., `if`, `while`) was extended in C++11 to admit *explicit* (user-defined) `bool` conversion operators (see *Use Cases: Enabling contextual conversions to* `bool` *as a test for validity* on page 106):

```
struct S2 { explicit operator bool(); };

void h()
{
S2 s2;
int i = s2;                   // error (copy initialization)
double d = s2;                // error (copy initialization)
int j = static_cast<int>(s2); // error (static cast)
if (s2) { }                   // OK (contextual conversion to bool)
int k(s2);                    // error (direct initialization)
double fd(s2);                // error (direct initialization)
}
```

Prior to C++11, essentially the same effect as having an *explicit* `operator bool()` member was achieved (albeit far less conveniently) via the **safe-`bool` idiom**.

## 1.17.2 Use Cases

### 1.17.2.1 Enabling contextual conversions to `bool` as a test for validity

Having a conventional test for validity that involves testing whether the object itself evaluates to `true` or `false` is an idiom that goes back to the origins of C++. The `<iostream>` library, for example, uses this idiom to determine if a given stream is valid:

```
// C++03
#include <iostream>  // std::ostream

std::ostream& printTypeValue(std::ostream& stream, double value)
{
if (stream)  // relies on an implicit conversion to bool
{
stream << "double(" << value << ')';
}
else
{
// ... (handle stream failure)
}

return stream;
}
```

Implementing the implicit conversion to `bool` was, however, problematic as the straightforward approach of using a **conversion operator** could easily allow accidental misuse to

---

idiom would be entirely lost if an explicit cast would have to be performed to test for validity. To address this, C++11 extends contextual conversion to `bool` for a given expression `E` to include an application of `static_cast<const volatile bool>` to `E`, thus enabling explicit conversion to `bool` to be used in lieu of the (now deprecated) **safe-`bool` idiom**; see **sharpe13**.

go undetected:

```
class ostream
{
// ...

/* implicit */ operator bool();  // hypothetical (bad) idea
};

int client(std::ostream& out)
{
// ...
return out + 1;  // likely a latent runtime bug: always returns 1 or 2
}
```

The classic workaround, the **safe-`bool` idiom**, was to return some obscure pointer type (e.g., **pointer to member**) that could not possibly be useful in any context other than one in which `false` and a null pointer-to-member value (e.g., `static_cast<(ostream*::operator bool)()>(0)`) are treated equivalently.

When implementing this idiom in a user-defined type ourselves, we need not go to such lengths to avoid inviting unintended use via an *implicit* conversion to `bool`. As discussed in *Description* on page 104, a conversion operator to type `bool` that is declared `explicit` continues to act as if it were *implicit* only in those places where we might want it to do so and nowhere else — i.e., exactly those places that enable **contextual conversion to `bool`**.[3]

As a concrete example, consider a `ConnectionHandle` class that can be in either a *valid* or *invalid* state. For the user's convenience and consistency with other proxy types (e.g., raw pointers) that have a similar *invalid* state, representing the invalid (or null) state via an explicit conversion to `bool` might be desirable:

```
struct ConnectionHandle
{
std::size_t maxThroughput() const;
// Return the maximum throughput (in bytes) of the connection.

explicit operator bool() const;
// Return true if the handle is valid and false otherwise.
};
```

Instances of `ConnectionHandle` will convert to `bool` only where one might reasonably want them to do so, say, as the predicate of an `if` statement:

```
int ping(const ConnectionHandle& handle)
{
if (handle)  // OK (contextual conversion to bool)
{
// ...
```

---

[3] Note that two consecutive `!` operators can be used to synthesize a **contextual conversion to `bool`** — i.e., if `X` is an expression that is explicitly convertible to `bool`, then `(!!(X))` will be `(true)` or `(false)` accordingly.

```
return 0;  // success
}

std::cerr << "Invalid connection handle.\n";
return -1;  // failure
}
```

Having an `explicit` conversion operator prevents unwanted conversions to `bool` that might otherwise happen inadvertently:

```
bool hasEnoughThroughput(const ClientConnection& connection,
const ResourceHandle&   handle)
{
return connection.throughput() <= handle;  // Compilation error, thankfully
//                                    ^~~~~~
}
```

After the relational operator (`<=`) in the example above, the programmer mistakenly wrote `handle` instead of `handle.maxThroughput()`. Fortunately the conversion operation of `ResourceHandle` was declared to be `explicit` and a compile-time error (thankfully) ensued; if the conversion had been *implicit*, the example code above would have compiled, and, if executed, the very same source for the `hasEnoughThroughput` function would have silently exhibited well-defined but incorrect behavior.

## 1.17.3 Potential Pitfalls

### 1.17.3.1 Sometimes implicit conversion is indicated

Implicit conversions to and from common arithmetic types, especially `int` are generally ill advised, given the likelihood of accidental misuse. However, sometimes *implicit* conversion is exactly what is needed. Such cases occur frequently with wrapper and proxy types that might need to interoperate with a large legacy codebase. Consider, for example, an initial implementation of memory allocators in which each constructor takes, as an optional trailing argument, a pointer to an abstract memory resource that itself provides pure virtual `allocate` and `deallocate` member functions. Later, we decide to move in the direction of the `std::pmr` (C++17) standard and wrap those pointers in classes that support additional operations. Making such constructors on the wrapper explicit would force every client supplying an allocator to a container to rework their code (e.g., by using `static_cast`). An implicit conversion in this case is further justified because the likelihood of accidental spontaneous conversion to an `Allocator` is all but nonexistent.

The same sort of stability argument favors implicit conversion for proxy types intended to be dropped in and used in existing codebases. If, for example, we wanted to provide a proxy for a writeable `std::string` that, say, also logged, we might want an implicit conversion to a `std::string&&` (perhaps using *reference qualifiers*). In such cases, making the conversion explicit would entirely defeat the purpose of the proxy, which is to achieve new functionality with minimal effect on existing client code.

### 1.17.4 Sometimes a named function is better

Other kinds of overuses of even *explicit* conversion operators exist. Like any user defined operator, when the operation being implemented is not somehow either canonical or ubiquitously idiomatic for that operator, expressing that operation by a named (i.e., non-operator) function is often better. Recall from *Description* on page 104 that we used a conversion operator of class `Point` to represent the distance from the origin. This example serves both to illustrate how conversion operators *can* be used and also how they probably should *not* be. Consider that (1) many mathematical operations on a 2-D integral point might return a `double` (e.g., `magnitude`, `angle`) and (2) we might want to represent the same information but in different units (e.g., `angleInDegrees`, `angleInRadians`).[4]

Rather than employing any conversion *operator* (`explicit` or otherwise), consider instead providing a named function, which (1) is automatically `explicit` and (2) affords both flexibility (in writing) and clarity (in reading) for a variety of domain-specific functions — now and in the future — that might well have had overlapping return types:

```
class Point  // only explicitly convertible (and from only an int)
{
int d_x, d_y;

public:
explicit Point(int x = 0, int y = 0);  // explicit converting constructor
// ...
double magnitude() const;  // Return distance from origin as a double.
};
```

Note that defining **nonprimitive functionality**, like `magnitude`, in a separate *utility* at a higher level in the physical hierarchy might be better still.[5]

### 1.17.5 Annoyances

None so far

### 1.17.6 See Also

None so far

### 1.17.7 Further Reading

None so far

---

[4] Another valid design decision is returning an object of type `Angle` that captures the amplitude and provides named accessory to the different units (e.g., `asDegrees`, `asRadians`).

[5] For more on separating out **nonprimitive functionality**, see **lakos20**, sections 3.2.7–3.2.8, pp 529–552.

## 1.18 `alignof`

The keyword `alignof` serves as a compile-time operator used to query the **alignment requirements** of a type on the current platform.

### 1.18.1 Description

The `alignof` operator, when applied to a type, evaluates to an **integral constant expression** that represents the **alignment requirements** of its argument type. Similar to `sizeof`, the (compile-time) value of `alignof` is of type `std::size_t`; unlike `sizeof` (which can accept an arbitrary expressions), `alignof` is defined (in the C++ Standard) on only a type identifier but often works on expressions anyway (see *Annoyances* on page 118). The argument type, `T`, supplied to `alignof` must be either a **complete type**, a **reference type**, or an **array type**. If `T` is a **complete type**, the result is the alignment requirement for `T`. If `T` is a **reference type**, the result is the alignment requirement for the referenced type. If `T` is an **array type**, the result is the alignment requirement for every element in the array[1]:

```
static_assert(alignof(short)    == 2, "");  // complete type   (sizeof is 2)
static_assert(alignof(short&)   == 2, "");  // reference type  (sizeof is 2)
static_assert(alignof(short[5]) == 2, "");  // array type      (sizeof is 2)
static_assert(alignof(short[])  == 2, "");  // array type      (sizeof fails)
```

#### 1.18.1.1 `alignof` Fundamental Types

Like their size, the alignment requirements of a `char`, `signed char`, and `unsigned char` are all guaranteed to be 1 (i.e., 1-byte aligned) on every conforming platform. For any other fundamental or pointer type `FPT`, `alignof(FPT)` (like `sizeof(FPT)`) is platform-dependent but is typically approximated well by the type's **natural alignment** — i.e., `sizeof(FPT) == alignof(FPT)`:

```
static_assert(alignof(char)   == 1, "");  // guaranteed to be 1
static_assert(alignof(short)  == 2, "");  // platform-dependent
static_assert(alignof(int)    == 4, "");  //    "          "
static_assert(alignof(double) == 8, "");  //    "          "
static_assert(alignof(void*)  >= 4, "");  //    "          "
```

#### 1.18.1.2 `alignof` User-Defined Types

When applied to user-defined types, alignment is always at least that of the strictest alignment of any of its arguments' base or member objects. Empty types are defined to have a size (and alignment) of 1 to ensure that every object has a unique address.[2] Compilers will (by default) avoid nonessential padding because any extra padding would be wasteful of (e.g., cache) memory[3]:

---

[1] According to the C++11 Standard, “An object of **array type** contains a contiguously allocated non-empty set of `N` subobjects of type `T`” (**cpp11**, section 8.3.4, “Arrays,” paragraph 1, p. 188). Note that, for every type `T`, `sizeof(T)` is always a multiple of `alignof(T)`; otherwise storing multiple `T` instances in an array would be impossible without padding, and the Standard explicitly prohibits padding between array elements.

```
struct S0 { };                          // sizeof(S0) is  1; alignof(S0) is  1
struct S1 { char c; };                  // sizeof(S1) is  1; alignof(S1) is  1
struct S2 { short s; };                 // sizeof(S2) is  2; alignof(S2) is  2
struct S3 { char c; short s; };         // sizeof(S3) is  4; alignof(S3) is  2
struct S4 { short s1; short s2; };      // sizeof(S4) is  4; alignof(S4) is  2
struct S5 { int i; char c; };           // sizeof(S5) is  8; alignof(S5) is  4
struct S6 { char c1; int i; char c2};   // sizeof(S6) is 12; alignof(S6) is  4
struct S7 { char c; short s; int i; };  // sizeof(S7) is  8; alignof(S7) is  4
struct S8 { double d; };                // sizeof(S8) is  8; alignof(S8) is  8
struct S9 { double d; char c};          // sizeof(S9) is 16; alignof(S9) is  8
struct SA { long double; };             // sizeof(SA) is 16; alignof(SA) is 16
struct SB { long double; char c};       // sizeof(SB) is 32; alignof(SB) is 16
```

## 1.18.2 Use Cases

### 1.18.2.1 Probing the alignment of a type during development

Both `sizeof` and `alignof` are often used informally during development and debugging to confirm the compiler's understanding of those attributes for a given type on the current

---

[2] An exception is made for an object of a type derived from an empty (base) class in that neither the size nor the alignment of the derived object is affected by the derivation:

```
struct S { int i; }      // size = 4; alignment = 4
struct E { };            // size = 1; alignment = 1
struct D : E { int i };  // size = 4; alignment = 4
```

[3] Compilers are permitted to increase alignment (e.g., in the presence of virtual functions) but have certain restrictions on padding. For example, they must ensure that each comprised type is itself sufficiently aligned and that the alignment of the parent type divides its size. This ensures that the fundamental identity for arrays holds for all types, `T`, and positive integers, `N`:

```
T a[N]; static_assert(n == sizeof(a) / sizeof(*a));  // guaranteed
```

The alignment of user-defined types can be made artificially stricter (but not weaker) using the `alignas` (see "`alignas`" on page 68) specifier. Also note that, for **standard-layout types**, the address of the first member object is guaranteed to be the same as that of the parent object:

```
struct S { int i; }
class T { public: S s; }
T t;
static_assert(&t.s == &t,     "");  // guaranteed
static_assert(&t.s == &t.s.i, "");  // guaranteed
```

This property also holds for (e.g., anonymous) unions:

```
struct { union { char c; float f; double d; } } u;
static_assert(&u == &u.c, "");  // guaranteed
static_assert(&u == &u.f, "");  // guaranteed
static_assert(&u == &u.d, "");  // guaranteed
```

platform. For example:

```
#include <iostream>

void f()
{
std::cout << " sizeof(double): " <<  sizeof(double) << '\n';  //  always 8
std::cout << "alignof(double): " << alignof(double) << '\n';  // usually 8
}
```

Printing the size and alignment of a `struct` along with those of each of its individual data members can lead to the discovery of suboptimal ordering of data members (resulting in wasteful extra padding). As an example, consider two `struct`s, `Wasteful` and `Optimal`, having the same three data members but in different order:

```
struct Wasteful
{
char   d_c;  // size =  1;  alignment = 1
double d_d;  // size =  8;  alignment = 8
int    d_i;  // size =  4;  alignment = 4
};               // size = 24;  alignment = 8

struct Optimal
{
double d_d;  // size =  8;  alignment = 8
int    d_i;  // size =  4;  alignment = 4
char   d_c;  // size =  1;  alignment = 1
};               // size = 16;  alignment = 8
```

Both `alignof(Wasteful)` and `alignof(Optimal)` are `8` but `sizeof(Wasteful)` is `24`, whereas `sizeof(Optimal)` is only `16`. Even though these two `struct`s contain the very same data members, the individual alignment requirements of these members forces the compiler to insert more total padding between the data members in `Wasteful` than is necessary in `Optimal`:

```
struct Wasteful
{
char   d_c;           // size =  1;  alignment = 1
char   padding_0[7];  // size =  7
double d_d;           // size =  8;  alignment = 8
int    d_i;           // size =  4;  alignment = 4
char   padding_1[4];  // size =  4
};                        // size = 24;  alignment = 8

struct Optimal
{
double d_d;           // size =  8;  alignment = 8
int    d_i;           // size =  4;  alignment = 4
char   d_c;           // size =  1;  alignment = 1
char   padding_0[3];  // size =  3
};                        // size = 16;  alignment = 8
```

### 1.18.2.2 Determining if a given buffer is sufficiently aligned

The `alignof` operator can be used to determine if a given (e.g., `char`) buffer is suitably aligned for storing an object of arbitrary type. As an example, consider the task of creating a **value-semantic** class, `MyAny`, that represents an object of arbitrary type[4]:

```
void f()
{
MyAny obj = 10;               // can be initialized with values of any type
assert(obj.as<int>() == 10);  // inner data can be retrieved at runtime

obj = std::string{"hello"};   // can be reassigned from a value of any type
assert(obj.as<std::string>() == "hello");
}
```

A straightforward implementation of `MyAny` would be to allocate an appropriately sized block of dynamic memory each time a value of a new type is assigned. Such a naive implementation would force memory allocations even though the vast majority of values assigned in practice are small (e.g., fundamental types), most of which would fit within the space that would otherwise be occupied by just the pointer needed to refer to dynamic memory. As a practical optimization, we might instead consider reserving a small buffer (say, roughly[5] 32 bytes) within the footprint of the `MyAny` object to hold the value provided (1) it will fit and (2) the buffer is sufficiently aligned. The natural implementation of this type — the union of a char array and a `struct` (containing a `char` pointer and a size) — will naturally result in the minimal alignment requirement of the `char*` (i.e., 4 on a 32-bit platform and 8 on a 64-bit one)[6]:

```
class MyAny  // nontemplate class
{
union
{
```

---

[4] The C++17 Standard Library provides the (nontemplate) class `std::any`, which is a type-safe container for single values of *any* **regular type**. The implementation strategies surrounding alignment for `std::any` in both `libstdc++` and `libc++` closely mirror those used to implement the simplified `MyAny` class presented here. Note that `std::any` also records the current `typeid` (on construction or assignment) so that it can implement a `const` template member function, `bool is<T>() const`, to query, at runtime, whether a specified type is currently the active one:

```
void f(const std::any& object)
{
if (object.is<int>()) { /* ... */ }
}
```

[5] We would likely choose a slightly larger value, e.g., 35 or 39, if that space would otherwise be filled with essential padding due to overall alignment requirements.

[6] We could, in addition, use the `alignas` attribute to ensure that the minimal alignment of `d_buffer` was at least 8 (or even 16):

```
// ...
alignas(8) char d_buffer[39];  // small buffer aligned to (at least) 8
// ...
```

```
struct
{
char*       d_buf_p;  // pointer to dynamic memory if needed
std::size_t d_size;   // for d_buf_p; same alignment as (char*)
} d_imp;  // Size/alignment of d_imp is sizeof(d_buf_p) (e.g., 4 or 8).

char d_buffer[39];         // small buffer aligned as a (char*)
};  // Size of union is 39; alignment of union is alignof(char*).

bool d_onHeapFlag;             // boolean (discriminator) for union (above)

public:
template <typename T>
MyAny(const T& x);               // (member template) constructor

template <typename T>
MyAny& operator=(const T& rhs);  // (member template) assignment operator

template <typename T>
const T& as() const;             // (member template) accessor

// ...

};  // Size of MyAny is 40; alignment of MyAny is alignof(char*) (e.g., 8).
```

The (templated) constructor[7] of `MyAny` can then decide (potentially at compile time) whether to store the given object `x` in the internal small buffer storage or on the heap, depending on `x`’s size and alignment:

```
template <typename T>
MyAny::MyAny(const T& x)
{
if (sizeof(x) <= 39 && alignof(T) <= alignof(char*))
{
// Store x in place in the small buffer.
new(d_buffer) T(x);
d_onHeapFlag = false;
}
else
{
// Store x on the heap and a pointer to it in the small buffer.
d_imp.d_buf_p = new T(x);
d_imp.d_size = sizeof(x);
d_onHeapFlag = true;
}
}
```

---

[7] In a real-world implementation, a *forwarding reference* would be used as the parameter type of `MyAny`’s constructor to *perfectly forward* the argument object into the appropriate storage; see “Forwarding References” on page 2.9.

Using the (compile-time) `alignof` operator in the constructor above to check whether the alignment of `T` is compatible with the alignment of the small buffer is necessary to avoid attempting to store overly aligned objects in place — even if they would fit in the 39-byte buffer. As an example, consider `long double`, which on typical platforms has both a size and alignment of `16`. Even though `sizeof(long double)` (16) is not greater than 39, `alignof(long double)` (16) is greater than that of `d_buffer` (8); hence, attempting to store an instance of `long double` in the small buffer, `d_buffer`, might — depending on where the `MyAny` object resides in memory — result in **undefined behavior**. User-defined types that either contain a `long double` or have had their alignments artificially extended beyond 8 bytes are also unsuitable candidates for the internal buffer even if they might otherwise fit:

```
struct Unsuitable1 { long double d_value };
// Size is 16 (<= 39), but alignment is 16 (> 8).

struct alignas(32) Unsuitable2 { };
// Size is  1 (<= 39), but alignment is 32 (> 8).
```

### 1.18.2.3 Monotonic memory allocation

A common pattern in software — e.g., request/response in client/server architectures — is to quickly build up a complex data structure, use it, and then quickly destroy it. A **monotonic allocator** is a special-purpose memory allocator that returns a monotonically increasing sequence of addresses into an arbitrary buffer subject to specific size and alignment requirements.[8] Especially when the memory is allocated by a single thread, there are prodigious[9] performance benefits to having unsynchronized raw memory to be taken directly off the (always hot) program stack. In what follows, we will provide the building blocks of a monotonic memory allocator wherein the `alignof` operator plays an essential role.

As a practically useful example, suppose that we want to create a lightweight `MonotonicBuffer` class template that will allow us to allocate raw memory directly from the footprint of the object. Just by creating an object of an (appropriately sized) instance of this type on the program stack, memory will natural come from the stack. For didactic reasons, we will start with a first pass at this class — ignoring alignment — and then go back and fix it using `alignof` so that it returns properly aligned memory:

```
template <std::size_t N>
struct MonotonicBuffer  // first pass at a monotonic memory buffer
{
char  d_buffer[N];  // fixed-size buffer
char* d_top_p;      //  next available address

MonotonicBuffer() : d_top_p(d_buffer) { }
```

---

[8] C++17 introduces an alternate interface to supply memory allocators via an abstract base class. The C++17 Standard Library provides a complete version of standard containers using this more interoperable design in a sub-namespace, `std::pmr`, where `pmr` stands for **polymorphic memory resource**. Also adopted as part of C++17 are two concrete memory resources, `std::pmr::monotonic_buffer_resource` and `std::pmr::unsynchronized_pool_resource`.

[9] see **lakos16**

```
// Initialize the next available address to be the start of the buffer.

template <typename T>
void* allocate()              // BAD IDEA --- doesn't address alignment
{
void* result = d_top_p;  // Remember the current next-available address.
d_top_p += sizeof(T);    // Reserve just enough space for this type.
return result;           // Return the address of the reserved space.
}
};
```

`MonotonicBuffer` is a class template with one integral template parameter that controls the size of the `d_buffer` member from which it will dispense memory. Note that, while `d_buffer` has an alignment of 1, the `d_top_p` member, used to keep track of the next available address, has an alignment that is typically 4 or 8 (corresponding to 32-bit and 64-bit architectures, respectively). The constructor merely initializes the next-address pointer, `d_top_p` to the start of the local memory pool, `d_buffer[N]`. The interesting part is how the `allocate` function manages to return a sequence of addresses corresponding to objects allocated sequentially from the local pool:

```
MonotonicBuffer<20> mb;  // On a 64-bit platform, the alignment will be 8.
char*   cp = static_cast<char*   >(mb.allocate<char   >());  // &d_buffer[ 0]
double* dp = static_cast<double*>(mb.allocate<double>());  // &d_buffer[ 1]
short*  sp = static_cast<short* >(mb.allocate<short >());  // &d_buffer[ 9]
int*    ip = static_cast<int*   >(mb.allocate<int   >());  // &d_buffer[11]
float*  fp = static_cast<float* >(mb.allocate<float >());  // &d_buffer[15]
```

The predominant problem with this first attempt at an implementation of `allocate` is that the addresses returned do not necessarily satisfy the minimum alignment requirements of the supplied type. A secondary concern is that there is no internal check to see if sufficient room remains. To patch this faulty implementation, we will need a function that, given an initial address and an alignment requirement, returns the amount by which the address must be rounded up (i.e., necessary padding) for an object having that alignment requirement to be properly aligned:

```
std::size_t calculatePadding(const char* address, std::size_t alignment)
// Requires: alignment is a (non-negative, integral) power of 2.
{
// rounding up X to N (where N is a power of 2): (x + N - 1) & ~(N - 1)
const std::size_t maxA = alignof(std::max_align_t);
const std::size_t a = reinterpret_cast<std::size_t>(address) & (maxA - 1);
const std::size_t am1 = alignment - 1;
const std::size_t alignedAddress = (a + am1) & ~am1;  // round up
return alignedAddress - a;                            // return padding
}
```

Armed with the `calculatePadding` helper function (above), we are all set to write the final (correct) version of the `allocate` method of the `MonotonicBuffer` class template:

```
template <typename T>
void* MonotonicBuffer::allocate()
```

```
{
// Calculate just the padding space needed for alignment.
const std::size_t padding = calculatePadding(d_top_p, alignof(T));

// Calculate the total amount of space needed.
const std::size_t delta = padding + sizeof(T);

// Check to make sure the properly aligned object will fit.
if (delta > d_buffer + N - d_top_p)  // if (Needed > Total - Used)
{
return 0;  // not enough properly aligned unused space remaining
}

// Reserve needed space; return the address for a properly aligned object.
void* alignedAddress = d_top_p + padding;  // Align properly for T object.
d_top_p += delta;                          // Reserve memory for T object.
return alignedAddress;                     // Return memory for T object.
}
```

Using this corrected implementation that uses `alignof` to pass the alignment of the supplied type `T` to the `calculatePadding` function, the addresses returned from the benchmark example (above) would be different[10]:

```
MonotonicBuffer<20> mb;  // Assume 64-bit platform (8-byte aligned).
char*   cp = static_cast<char*  >(mb.allocate<char  >());  // &d_buffer[ 0]
double* dp = static_cast<double*>(mb.allocate<double>());  // &d_buffer[ 8]
short*  sp = static_cast<short* >(mb.allocate<short >());  // &d_buffer[16]
int*    ip = static_cast<int*   >(mb.allocate<int   >());  // 0 (out of space)
bool*   bp = static_cast<bool*  >(mb.allocate<bool  >());  // &d_buffer[18]
```

In practice, an object that allocates memory, such as a `vector` or a `list` will be constructed with an object that allocates and deallocates memory that is guaranteed to be either **maximally aligned**, **naturally aligned**, or sufficiently aligned to satisfy an optionally specified alignment requirement.

Finally, instead of returning a null pointer when the buffer was exhausted, we would typically have the concrete allocator fall back to a geometrically growing sequence of dynamically allocated blocks; the `allocate` method would then fail (i.e., a `std::bad_alloc` exception would somehow be thrown) only if all of available memory were exhausted and the **`new` handler** were unable to acquire more memory yet still opted to return control to its caller.

---

[10] Note that on a 32-bit architecture, the `d_top_p` character pointer would be only four-byte aligned, which means that the entire buffer might be only four-byte aligned. In that case, the respective offsets for `cp`, `dp`, `sp`, `ip`, and `bp` in the example for the aligned use case might sometimes instead be 0, 4, 12, 16, and `nullptr`, respectively. If desired, we can use the `alignas` attribute/keyword to artificially constrain the `d_buffer` data member always to reside on a maximally aligned address boundary, thereby improving consistency of behavior, especially on 32-bit platforms.

## 1.18.3 Annoyances

### 1.18.3.1 `alignof` (unlike `sizeof`) is defined only on types

The (compile-time) `sizeof` operator comes in two different forms: one accepting a *type* and the other accepting an *expression*. The C++ Standard currently requires that `alignof` support only the former[11]:

```
static_assert(sizeof(int)  == 4, "");     // OK, int is a type.
static_assert(alignof(int) == 4, "");     // OK, int is a type.
static_assert(sizeof(3 + 2))  == 4, "");  // OK, 3 + 2 is an expression.
static_assert(alignof(3 + 2)) == 4, "");  // Error, 3 + 2 is not a type.
```

This asymmetry can result in a need to leverage `decltype` (see "`decltype`" on page 28) when inspecting an expression instead of a type:

```
int f()
{
enum { e_SUCCESS, e_FAILURES } result;
std::cout << "size: " << sizeof(result) << '\n';
std::cout << "alignment:" << alignof(decltype(result)) << '\n';
}
```

The same sort of issue occurs in conjunction with modern **type inference** features such as `auto` (see "`auto`" on page 224) and generic lambdas (see "Generic Lambdas" on page 234). As a real-world example, consider the generic lambda (C++14) being used to introduce a small *local function* that prints out information regarding the size and alignment of a given `object`, likely for debugging purposes:

```
auto printTypeInformation = [](auto object)
{
std::cout << "     size: " << sizeof(object) << '\n'
<< "alignment: " << alignof(decltype(object)) << '\n';
};
```

Because there is no explicit type available within the body of the `printTypeInformation` lambda,[12] a programmer wishing to remain entirely within the C++ standard[13] is forced to use the `decltype` construct explicitly to first obtain the type of `object` before passing it on to `alignof`.

---

[11] Although the Standard does not require `alignof` to work on arbitrary expressions, `alignof` is a common GNU extension and most compilers support it. Both Clang and GCC will warn only if `-Wpedantic` is set.

[12] In C++20, referring to the type of a generic lambda parameter explicitly is possible (due to the addition to lambdas of some familiar template syntax):

```
auto printTypeInformation = []<typename T>(T object)
{
std::cout << "     size: " << sizeof(T) << '\n'
<< "alignment: " << alignof(T) << '\n';
};
```

[13] Note that `alignof(object)` will work on every major compiler (GCC 10.x, Clang 10.x, and MSVC 19.x) as a nonstandard extension.

## 1.18.4 See Also

- "`alignas`" — Safe C++11 feature that can be used to provide an artificially stricter alignment (e.g., more than **natural alignment**).
- "`decltype`" — Safe C++11 feature that helps work around `alignof`'s limitation of accepting only a type, not an expression (see *Annoyances* on page 118).

## 1.18.5 Further Reading

None so far

# 1.19 Inheriting Constructors

The term *inheriting constructors* refers to the use of a **`using`-declaration** to expose nearly all of the constructors of a base class in the scope of a derived class.

## 1.19.1 Description

In a class definition, a **`using` declaration** naming a base class's constructor results in the derived class "inheriting" all of the nominated base class's constructors, except for *copy* and *move* constructors. Just like other `using` declarations, the nominated base class's constructors will be searched when no matching constructor is found in the derived class. When a base class constructor is selected in this way, that constructor will be used to construct the base class and the remaining bases and data members of the subclass will be initialized as if by the default constructor (e.g., applying default initializers; see "Default Member Initializers" on page 227).

```
struct B0
{
B0() = default;      // public, default constructor
B0(int)         { }  // public, one argument (implicit) value constructor
B0(int, int)    { }  // public, two argument value constructor

private:
B0(const char*) { }  // private, one argument constructor
};

struct D0 : B0
{
using B0::B0;  // using declaration
D0(double d);  // suppress implicit default constructor
};

D0 t(1);     // OK, inherited from B0::B0(int)
D0 u(2, 3);  // OK, inherited from B0::B0(int, int)
D0 v("hi");  // Error: Base constructor is declared private.
```

The only constructors that are explicitly *not* inheritable by the derived class are the (potentially compiler-generated) *copy* and *move* constructors[1]:

```
#include <utility>  // std::move

B0 b1(1);              // OK, base-class object can be created.
B0 b2(2, 3);           // OK, base-class object can be created.
B0 b3(b1);             // OK, base-class object can be copied (from *lvalue*).
B0 b4(std::move(b1));  // OK, base-class object can be moved (from *rvalue).
```

---

[1] Note that we use braced initialization (see "Braced Initialization" on page 225) in `D0 x(B0);` to ensure that a variable `x` of type `D0` is declared. `D0 x(B0());` would instead be interpreted as a declaration of a function `x` returning `D0` and accepting a pointer to a nullary function returning `B0`, which is referred to as the **most vexing parse**.

```
D0 w(b1);    // Error: Base-class copy constructor is not inherited.
D0 v;        // OK, base-class default constructor is inherited.
D0 x(B0{});  // Error: Base-class move constructor is not inherited.

D0 y(B0(4));  // Error: Base-class move constructor is not inherited.
D0 z(t);      // OK, uses compiler-generated D0::D0(const D0&)
D0 j(D0(5));  // OK, uses compiler-generated DO::D0(D&&)
```

The constructors inherited by the derived class have the same effect on whether the compiler implicitly generates special member functions as explicitly implemented ones would. For example, `D0`’s default constructor would be implicitly *deleted* (see “Deleted Functions” on page 33) if `B0` doesn’t have a default constructor. Note that since the copy and move constructors are *not* inherited, their presence in the base class wouldn’t suppress implicit generation of copy and move assignment in the derived class. For instance, `D0`’s implicitly generated assignment operators obliviously **hide** their counterparts in `B0`:

```
void f()
{
B0 b(0), bb(0);  // Create destination and source B0 objects.
D0 d(0), dd(0);  //   "         "       "    "    D0    ".

b = bb;          // OK, assign base from lvalue base.
b = B0(0);       // OK,   "       "   "  rvalue   "

d = bb;          // Error: B0::operator= is hidden by D0::operator=.
d = B0(0);       // Error:       "         "    "     "        "

d.B0::operator=(bb);     // OK, explicit slicing is still possible.
d.B0::operator=(B0(0));  // OK,    "         "      "   "      "

d = dd;          // OK, assign derived from lvalue derived.
d = D0(0);       // OK,   "        "     "  rvalue    "
}
```

Note that, when inheriting constructors, private constructors in the base class are accessed as private constructors of that base class and are subject to the same access controls; see *Annoyances: Access levels of inherited constructors are the same as in base class* on page 133.

Inheriting constructors having the same **signature** from multiple base classes lead to ambiguity errors:

```
struct B1A { B1A(int); };
struct B1B { B1B(int); };

struct D1 : B1A, B1B
{
using B1A::B1A;
using B1B::B1B;
};
```

```
D1 d1(0);  // Error: Call of overloaded D1(int) is ambiguous.
```

Each inherited constructor shares the same characteristics as the corresponding one in the nominated base class’s constructors and then delegates to it. This means the **access specifiers**, the `explicit` specifier, the `constexpr` specifier, the default arguments, and the exception specification are also preserved by constructor inheritance; see “`noexcept`” on page 233 and “`constexpr` Functions” on page 229. For template constructors, the template parameter list and the default template arguments are preserved as well:

```
struct B2
{
template <typename T = int>
explicit B2(T) { }
};

struct D2 : B2 { using B2::B2; };
```

The declaration `using B2::B2` above behaves as if a constructor template that delegates to its nominated base class’s template was provided in `D2`:

```
// pseudocode
struct D2 : B2
{
template <typename T = int>
explicit D2(T i) : B2(i) { }
}
```

When deriving from a base class in which inheriting most (but not all) of its constructors is desirable, suppressing inheritance of one or more of them is possible by providing constructors in the derived class having the same signature as the ones that would be inherited:

```
struct B2
{
B2()          { std::cout << "B2()\n"; }
B2(int)       { std::cout << "B2(int)\n"; }
B2(int, int) { std::cout << "B2(int, int)\n"; }
};

struct D2 : B2
{
using B2::B2;
D2(int) { std::cout << "D2(int)\n"; }
};

D2 d;          // prints "B2()"
D2 e(0);       // Prints "D2(int)" --- The derived constructor is invoked.
D2 f(0, 0);  // prints "B2(int, int)"
```

In other words, we can suppress what would otherwise be an inherited constructor from a nominated base class by simply declaring a replacement with the same signature in the derived class. We can then choose to either implement it ourselves, default it (see “Defaulted Special Member Functions” on page 184), or `delete` it (see “Deleted Functions” on page 33).

If we have chosen to inherit the constructors from multiple base classes, we can disambiguate conflicts by declaring the offending constructor(s) explicitly in the derived class and then delegating to the base classes if and as appropriate:

```
struct B1A { B1A(int); };  // Here we have two bases classes, each of which
struct B1B { B1B(int); };  // provides a conversion constructor from an int.

struct D1 : B1A, B1B
{
using B1A::B1A;  // Inherit the int constructor from base class B1A.
using B1B::B1B;  // Inherit the int constructor from base class B1B.

D1(int i) : B1A(i), B1B(i) { }  // Declare the int conversion constructor
};                                  // explicitly, and then delegate to bases.

D1 d1(0);  // OK, calls D1(int)
```

Lastly, inheriting constructors from a **dependent type** affords a capability over C++03 that is more than just convenience and avoidance of boilerplate code.[2] In all of the example code in *Description* thus far, we know how to spell the base-class constructor; we are simply automating some drudge work. In the case of a *dependent* base class, however, we do *not* know how to spell the constructors, so we *must* rely on **inheriting constructors** if that is the forwarding semantic we seek:

```
template <class T>
struct S : T  // The base type, T, is a *dependent type*.
{
using T::T;  // inheriting constructors (generically) from a dependent type
};

#include <string>
#include <vector>

S<std::string>        ss("hello");     // OK, uses constructor from base
S<std::vector<char>> svc("goodbye");  // Error: no suitable constructor in base
```

In this example, we created a class template, `S`, that derived publicly from its template argument, `T`. Then, when creating an object of type `S` parameterized by `std::string`, we were able to pass it a string literal via the inherited `std::string` constructor overloaded on a `const char*`. Notice, however, that no such constructor is available in `std::vector`; hence, attempting to create the derived class from a literal string results in a compile-time error. See *Use Cases: Incorporating reusable functionality via a mix-in* on page 128.

---

[2] A decidedly more complex alternative affording a different set of tradeoffs would involve variadic template constructors (see “Variadic Templates” on page 230) having forwarding references (see “Forwarding References” on page 232) as parameters. In this alternative approach, all of the constructors from the `public`, `protected`, and `private` regions of the bases class would now appear under the same access specifier — i.e., the one in which the perfectly forwarding constructor is declared. What’s more, this approach would not retain other constructor characteristics, such as `explicit`, `noexcept`, `constexpr`, and so on. The forwarding can, however, be restricted to inheriting just the `public` constructors (without characteristics) by constraining on `std::is_constructible` using **SFINAE**; see *Annoyances: Access levels of inherited constructors are the same as in base class* on page 133.

## 1.19.2 Use Cases

### 1.19.2.1 Abstract use case

Use of this form of `using` declaration to inherit a nominated base class’s constructors — essentially verbatim — suggests that one or more of those constructors is sufficient to initialize the *entire* derived-class object to a valid useful state. Typically, such will pertain only when the derived class adds no member data of its own. While additional derived-class member data could possibly default, this state must be *orthogonal* to any modifiable state initialized in the base class, as such state is subject to independent change via **slicing**, which might in turn invalidate **object invariants**. Derived-class data will need either to default or to have its value set using member initializers (see “Default Member Initializers” on page 227). Hence, most typical use cases will involve wrapping an existing class by deriving from it (either publicly or privately), adding only defaulted data members having orthogonal values, and then adjusting the derived class’s behavior via **overriding** its virtual or **hiding** its and non-virtual member functions.

### 1.19.2.2 Avoiding boilerplate code when employing structural inheritance

A key indication for the use of inheriting constructors is that the derived class addresses only auxiliary or optional, rather than required or necessary, functionality to its self-sufficient base class. As an interesting, albeit mostly pedagogical,[3] example, suppose we want to provide a proxy for a `std::vector` that performs explicit checking of indices supplied to its index operator:

```
#include <vector>
#include <cassert>

template <typename T>
struct CheckedVector : std::vector<T>
{
using std::vector<T>::vector;      // Inherit std::vector's constructors.

T& operator[](std::size_t index)   // Hide std::vector's index operator.
{
assert(index < std::vector<T>::size());
return std::vector<T>::operator[](index);
}

const T& operator[](std::size_t index) const  // Hide const index operator.
{
assert(index < std::vector<T>::size());
return std::vector<T>::operator[](index);
}
};
```

---

[3] Although this example might be compelling, it suffers from inherent deficiencies making it insufficient for general use in practice: Passing the derived class to a function — whether by value or reference – will strip it of its auxiliary functionality. The best-known solution — a C++2x language-based contract-checking facility — is exactly what’s needed ubiquitously. We plan to cover this topic in **lakos23**.

In the example above, inheriting constructors allowed us to use public (structural) inheritance to readily create a distinct new type having all of the functionality of its base type except for a couple of functions where we chose to augment the original behavior.

### 1.19.2.3 Avoiding boilerplate code when employing implementation inheritance

Sometimes it can be cost effective to adapt **concrete class** having virtual functions[4] to a specialized purpose using inheritance.[5] As an example, consider a **concrete** base class, `NetworkDataStream`, that allows overriding its virtual functions for processing a stream of data from an expanding variety of arbitrary sources over the network:

```
class NetworkDataStream
{
private:
// ...                  (member data)

public:
explicit NetworkDataStream(TCPConnection* tcpConnection);
explicit NetworkDataStream(UDPConnection* udpConnection);
explicit NetworkDataStream(RawDataStreamHandle* rawDataStreamHandle);

virtual ~NetworkDataStream();

virtual void onPacketReceived(DataPacket& dataPacket) = 0;
// Derived classes must override this method.
```

The concrete class above now provides three constructors (with more under development) that can be used assuming no per-packet processing is required. Now, imagine the need for logging information about received packets (e.g., for auditing purposes). Inheriting constructors make deriving from `NetworkDataStream` and overriding (see “`override`” on page 38) `onPacketReceived(DataPacket&)` more convenient — i.e., without having to reimplement each of the constructors, which are anticipated to increase in number over time:

```
class LoggedNetworkDataStream : public NetworkDataStream
{
public:
using NetworkDataStream::NetworkDataStream;

void onPacketReceived(DataPacket& dataPacket) override
{
LOG_TRACE << "Received packet " << dataPacket;    // local log facility
NetworkDataStream::onPacketReceived(dataPacket);  // Delegate to base.
}
};
```

---

[4] Useful design patterns exist where a **partial implementation** class, derived from a pure abstract interface (a.k.a. a **protocol**), contains data, constructors, and pure virtual functions; see **lakos2a**, section 4.7.

[5] Such inheritance, known as **implementation inheritance** is decidedly distinct from pure **interface inheritance**, which is often the preferred design pattern in practice; see **lakos2b**, section 4.6.

### 1.19.2.4 Implementing a strong `typedef`

Classic `typedef` declarations — just like C++11 `using` declarations (see “Alias Declarations and Alias Templates” on page 97) — are just synonyms; they offer absolutely no type safety. A commonly desired capability is to provide an alias to an existing type `T` that is uniquely interoperable with itself, explicitly convertible from `T`, but not implicitly convertible from `T`. This somewhat *more* “type-safe” form of alias is sometimes referred to as a **strong `typedef`**.[6]

As a practical example, suppose we are exposing, to a fairly wide and varied audience, a class, `PatientInfo`, that associates two `Date` objects to a given hospital patient:

```
class Date
{
// ...

public:
Date(int year, int month, int day);

// ...
};

class PatientInfo
{
private:
Date d_birthday;
Date d_appointment;

public:
PatientInfo(Date birthday, Date appointment);
// Please pass the birthday as the first date and the appointment as
// the second one!
};
```

For the sake of argument, imagine that our users are not as assiduous as they should be in reading documentation to know which constructor argument is which:

```
PatientInfo client1(Date birthday, Date appointment)
{
return PatientInfo(birthday, appointment);  // OK
}

int client2(PatientInfo* result, Date birthday, Date appointment)
{
*result = PatientInfo(appointment, birthday);  // Oops! wrong order
```

---

[6] A so-called **strong `typedef`** is similar to a classic, C-style enumeration in that it is (1) its own type and (2) implicitly convertible to its base type (which for enumerators corresponds to its **underlying type**; see “Explicit Enumeration Underlying Type” on page 143). Unlike a classic `enum`, however, a typical implementation of a **strong `typedef`** allows only for explicit conversion from its base type. An analogy to the more strongly typed `enum class` (see “`enum class`” on page 148) would suppress conversion in either direction, e.g., via private inheritance and then `explicit` conversion constructors and `explicit` conversion operators (see “Explicit Conversions” on page 104).

```
return 0;
}
```

Now suppose that we continue to get complaints, from folks like `client2` in the example above, that our code doesn't work. What can we do?[7]

One way is to force clients to make a conscious and explicit decision in their own source code as to which `Date` is the birthday and which is the appointment. Employing a **strong `typedef`** can help us to achieve this goal. Inheriting constructors provide a concise way to define a **strong `typedef`**; for the example above, they can be used to define two new types to represent, uniquely, a birthday and an appointment date:

```
struct Birthday : Date  // somewhat type-safe alias for a Date
{
using Date::Date;  // inherit Date's three integer ctor
explicit Birthday(Date d) : Date(d) { }  // explicit conversion from Date
};

struct Appointment : Date  // somewhat type-safe alias for a Date
{
using Date::Date;  // inherit Date's three integer ctor
explicit Appointment(Date d) : Date(d) { }  // explicit conv. from Date
};
```

The `Birthday` and `Appointment` types expose the same interface of `Date`, yet, given our inheritance-based design, `Date` is not implicitly convertible to either. Most importantly, however, these two new types are not implicitly convertible to each other:

```
Birthday b0(1994, 10, 4);  // OK, thanks to inheriting constructors
Date d0 = b0;              // OK, thanks to public inheritance
Birthday b1 = d0;          // error: no implicit conversion from Date
Appointment a0;            // Error: Appointment has no default ctor.
Appointment a1 = b0;       // error: no implicit conversion from Birthday
Birthday n2(d0);           // OK, thanks to an explicit constructor in Birthday
Birthday b3(a0);           // OK, an Appointment (unfortunately) is a Date.
```

We can now reimagine a `PatientInfo` class that exploits this newfound (albeit artificially manufactured[8]) type-safety:

```
class PatientInfo
{
private:
Birthday d_birthday;
Appointment d_appointment;
```

---

[7] Although this example is presented lightheartedly, misuse by clients is a perennial problem in large-scale software organizations. Choosing the same type for both arguments might well be the right choice in some environments but not in others. We are not advocating use of this technique; we are merely acknowledging that it exists.

[8] Replicating types that have identical behavior in the name of type safety can run afoul of interoperability. Distinct types that are otherwise physically similar are often most appropriate when their respective behaviors are inherently distinct and unlikely to interact in practice (e.g., a `CartesianPoint` and a `RationalNumber`, each implemented as having two integral data members); see **lakos2a**, section 4.4.

```
public:
PatientInfo(Birthday birthday, Appointment appointment);
// Why should I bother to write documentation you won't read anyway!?
};
```

Now our clients have no choice but to make their intentions clear at the call site:

```
PatientInfo client0(Date birthday, Date appointment)
{
return PatientInfo(birthday, appointment);  // Sorry, doesn't compile.
}

int client1(PatientInfo* result, Date birthday, Date appointment)
{
*result = PatientInfo(appointment, birthday);  // Nope! Doesn't compile.
return 0;
}

PatientInfo client3(Date birthday, Date appointment)
{
return PatientInfo(Birthday(birthday), Appointment(appointment));  // OK
}

int client4(PatientInfo* result, Date birthday, Date appointment)
{
Birthday b(birthday);
Appointment a(appointment)
*result = PatientInfo(b, a);  // OK
}
```

This example works because the **value constructor** takes three arguments and cannot be invoked as part of an implicit conversion sequence; see *Potential Pitfalls: Beware of inheriting implicit constructors* on page 130. Note that, in an ideal world where thorough unit testing is ubiquitous, such machinations would most likely be supererogatory.

### 1.19.2.5 Incorporating reusable functionality via a mix-in

Some classes are designed to generically enhance the behavior of a class just by inheriting from it; such classes are sometimes referred to as *mix-ins*. If we wish to adapt a class to support the additional behavior of the mix-in, with no other change to its behavior, we can use simple **structural inheritance** (e.g., to preserve reference compatibility through function calls). To preserve the public interface, however, we will need it to inherit the constructors as well.

Consider, for example, a simple class to track the total number of objects created:

```
template <typename T>
struct CounterImpl  // mix-in used to augment implementation of arbitrary type
{
static int s_constructed;  // count of the number of T objects constructed
CounterImpl()                   { ++s_constructed; }
CounterImpl(const CounterImpl&) { ++s_constructed; }
};

template <typename T>
CounterImpl<T>::s_constructed;  // required member definition
```

The class template `CounterImpl`, in the example above, counts the number of times an object of type `T` was constructed during a run of the program. We can then write a generic adapter, `Counted`, to facilitate use of `CounterImpl` as a *mix-in*:

```
template <class T>
struct Counted : T, CounterImpl<T>
{
using T::T;
};
```

Note that the `Counted` adaptor class inherits all of the constructors of the *dependent* class, `T`, that it wraps, without its having to know what those constructors are:

```
#include <string>   // std::string
#include <vector>   // std::vector
#include "myfoo.h"  // MyFoo

Counted<std::string>        cs;  // Construct a counted string.
Counted<std::vector<char>> cvc;  // Construct a counted vector of char.
Counted<MyFoo>             cmf;  // Construct a counted MyFoo object.
```

While inheriting constructors are a convenience in nongeneric programming, they can be an essential tool for generic idioms.

## 1.19.3 Potential Pitfalls

### 1.19.3.1 Newly introduced constructors in the base class can silently alter program behavior

The introduction of a new constructor in a base class might silently change a program's run-time behavior if that constructor happens to be a better match during overload resolution of an existing instantiation of a derived class. Consider a `Session` class that initially provides only two constructors:

```
struct Session
{
Session();
explicit Session(RawSessionHandle* rawSessionHandle);
};
```

Now, imagine that a class, `AuthenticatedSession`, derived from `Session`, inherits the two constructors of its base class and provides its own constructor that accepts an integral authentication token:

```
struct AuthenticatedSession : Session
{
using Session::Session;
explicit AuthenticatedSession(long long authToken);
};
```

Finally, consider an instantiation of `AuthenticatedSession` in user-facing code:

```
AuthenticatedSession authSession(45100);
```

In the line above and the example above that, `authSession` will be initialized by invoking the constructor accepting a `long long` (see "`long long`" on page 92) authentication token. If, however, a new constructor, having the signature `Session(int)` is added to the base class, it will be invoked instead because it is a better match to the literal `45100` (of type `int`) than the constructor taking a `long long` supplied explicitly in the derived class; hence, adding a constructor to a base class might lead to a potential latent (runtime) defect that would go unreported at compile time.

Note that this problem with shifting implicit conversions is not unique to inheriting constructors; any form of `using` declaration or invocation of an overloaded function carries a similar risk. Imposing stronger typing — e.g., by using **strong `typedefs`** (see *Use Cases: Implementing a strong* `typedef` on page 126) — might sometimes, however, help to prevent such unfortunate missteps.

### 1.19.3.2 Beware of inheriting implicit constructors

Inheriting from a class that has implicit constructors can cause surprises. Consider again the use of inheriting constructors to implement a **strong `typedef`** from *Use Cases: Implementing a strong* `typedef` on page 126. This time, however, let's suppose we are exposing, to a fairly wide and varied audience, a class `PointOfInterest`, that associates the name and address of a given popular tourist attraction:

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

class PointOfInterest
{
private:
std::string d_name;
std::string d_address;

public:
PointOfInterest(const std::string& name, const std::string& address);
// Please pass the name as the *first* and the address *second*!
};
```

Again imagine that our users are not always careful about inspecting the function prototype:

```
PointOfInterested client1(const char* name, const char* address)
{
return PointOfInterest(name, address);  // OK
}
```

```
int client2(PointOfInterest* result, const char* name, const char* address)
{
*result = PointOfInterest(address, name);  // Oops! wrong order
return 0;
}
```

We might think to again use **strong `typedef`s** here as we did for `PatientAppointment` in *Use Cases: Implementing a strong* `typedef` on page 126:

```
struct Name : std::string  // somewhat type-safe alias for a std::string
{
using std::string::string;  // Inherit, as is, all of std::string's ctors.
explicit Name(const std::string& s) : std::string(s) { }  // conversion
};

struct Address : std::string  // somewhat type-safe alias for a std::string
{
using std::string::string;  // Inherit, as is, all of std::string's ctors.
explicit Address(const std::string& s) : std::string(s) { }  // conversion
};
```

The `Name` and `Address` types are not interconvertible; they expose the same interfaces as `std::string` but are not implicitly convertible from it:

```
Name n0 = "Big Tower";  // OK, thanks to inheriting constructors
std::string s0 = n0;    // OK, thanks to public inheritance
Name n1 = s0;           // error: no implicit conversion from std::string
Address a0;             // OK, unfortunately a std::string has a default ctor.
Address a1 = n0;        // error: no implicit conversion from Name
Name n2(s0);            // OK, thanks to an explicit constructor in Name
Name b3(a0);            // OK, an Address (unfortunately) is a std::string.
```

We can rework the `PointOfInterest` class to use the the **strong `typedef`** idiom:

```
class PointOfInterest
{
private:
Name    d_name;
Address d_address;

public:
PointOfInterest(const Name& name, const Address& address);
};
```

Now if our clients use the base class itself as a parameter, they will again need to make their intentions known:

```
PointOfInterested client1(const std::string& name, const std::string address)
{
return PointOfInterest(address, name);  // sorry, doesn't compile
}

PointOfInterested client2(const char* name, const char* address)
```

```
{
return PointOfInterest(Name(name), Address(address));  // OK
}
```

But suppose that some clients instead pass the arguments by `const char*` instead of `const std::string&`:

```
PointOfInterested client3(const char* name, const char* address)
{
return PointOfInterest(address, name);  // Oops! compiles but runtime error
}
```

In the case of `client3` in the code snippet above, passing the arguments through *does* compile because the `const char*` constructors are inherited; hence, there is no attempt to convert to a `std::string` before matching the *implicit* conversion constructor. Had the `std::string` conversion constructor been declared to be `explicit`, the code would not have compiled. In short, inheriting constructors from types that perform implicit conversions seriously undermine the effectiveness of the **strong `typedef`** idiom.

## 1.19.4 Annoyances

### 1.19.4.1 Inherited constructors cannot be selected individually

The inheriting-constructors feature does not allow the programmer to select a subset of constructors to inherit; all of the base class’s eligible constructors are always inherited unless a constructor with the same signature is provided in the derived class. If the programmer desires to inherit all constructors of a base class except for perhaps one or two, the straightforward workaround would be to declare the undesired constructors in the derived class and then use deleted functions (see “Deleted Functions” on page 33) to explicitly exclude them.

For example, suppose we have a general class, `Datum`, that can be constructed from a variety of types:

```
struct Datum
{
Datum(bool);
Datum(char);
Datum(short);
Datum(int);
Datum(long);
Datum(long long);
};
```

If we wanted to create a version of `Datum`, call it `NumericalDatum`, that inherits all but the one constructor taking a `bool`, our derived class would (1) inherit publicly, (2) declare the unwanted constructor, and then (3) mark it with `= delete`:

```
struct NumericalDatum : Datum
{
using Datum::Datum;               // Inherit all the constructors...
NumericalDatum(bool) = delete;  // ...except the one taking a bool.
};
```

Note that the subsequent addition of any non-numerical constructor to `Datum` (e.g., a constructor taking `std::string`) would defeat the purpose of `NumericalDatum`.

### 1.19.4.2 Access levels of inherited constructors are the same as in base class

Unlike base-class member functions that can be introduced with a `using` directive with an arbitrary access level into the derived class (as long as they are accessible by the derived class), the access level of the `using` declaration for inherited constructors is ignored.[9] The inherited constructor overload is instead accessible *if* the corresponding base-class constructor would be accessible:

```
struct Base
{
private:
Base(int) { }  // This constructor is declared private in the base class.
void pvt0() { }
void pvt1() { }

public:
Base() { }     // This constructor is declared public in the base class.
void pub0() { }
void pub1() { }
};
```

Note that, when employing `using` to (1) inherit constructors or (2) elevate base-class definitions in the presence of private inheritance, public clients of the class might find it necessary to look at what are ostensibly private implementation details of the derived class to make proper use of that type through its public interface:

```
struct Derived : private Base
{
using Base::Base;  // OK, inherited Base() as public constructor
// and Base(int) as private constructor

private:
using Base::pub0;  // OK, pub0 is declared private in derived class.
using Base::pvt0;  // Error: pvt0 was declared private in base class.

public:
using Base::pub1;  // OK, pub1 is declared public in derived class.
using Base::pvt1;  // Error: pvt1 was declared private in base class.
};

void client()
{
```

---

[9] Alisdair Meredith, one of the authors of the Standards paper that proposed this feature (**meredith08**) suggests that placing the `using` declaration for **inheriting constructors** as the very first member declaration and preceding any **access specifiers** might be the least confusing location. Programmers might still be confused by the disparate default access levels of `class` versus `struct`.

```
Derived x(0);  // Error: Constructor was declared private in base class.
Derived d;     // OK, constructor was declared public in base class.
d.pub0();      // Error: pub0 was declared private in derived class.
d.pub1();      // OK, pub1 was declared public in derived class.
d.pvt0();      // Error: pvt0 was declared private in base class.
d.pvt1();      // Error: pvt1 was declared private in base class.
}
```

This C++11 feature was itself created because the previously proposed solution — which also involved a couple of features new in C++11, namely forwarding the arguments to base-class constructors with forwarding references (see "Forwarding References" on page 232) and variadic templates (see "Variadic Templates" on page 230) — made somewhat different tradeoffs and was considered too onerous and fragile to be practically useful:

```
#include <utility>  // std::forward

struct Base
{
Base(int) { }
};

struct Derived : private Base
{
protected:
template <typename... Args>
Derived(Args&&... args) : Base(std::forward<Args>(args)...)
{
}
};
```

In the example above, we have used forwarding references (see "Forwarding References" on page 232) to properly delegate the implementation of a constructor that is declared `protected` in the derived class to a `public` constructor of a privately inherited base class. Although this approach fails to preserve many of the characteristics of the inheriting constructors (e.g., `explicit`, `constexpr`, `noexcept`, and so on), the functionality described in the code snippet above is simply not possible using the C++11 inheriting-constructors feature.

## 1.19.5 See Also

- "`override`" on page 38 — Used to ensure that a member function intended to override a virtual function actual does
- "Deleted Functions" on page 33 — Can be used to exclude inherited constructors that are unwanted entirely
- "Defaulted Special Member Functions" on page 184 — Used to implement functions that might otherwise have been suppressed by inherited constructors
- "Delegating Constructors" on page 81 — Related feature used to call one constructor from another from within the same user-defined type

- "Default Member Initializers" on page 227 — Useful in conjunction with this feature when a derived class adds member data
- "Forwarding References" on page 232 — Used in alternative (workaround) when access levels differ from those for base-class constructors
- "Variadic Templates" on page 230 — Used in alternative (workaround) when access levels differ from those for base-class constructors
- "Default Member Initializers" on page 227 — Can be used to provide nondefault values for data members in derived classes that make use of inheriting constructors

### 1.19.6 Further Reading

None so far

### 1.19.7 Appendix: C++17 Improvements Made Retroactive to C++11/14

The original specification of inheriting constructors in C++11 had a significant number of problems with general use. As originally specified, inherited constructors were treated as if they were redeclared in the derived class. For C++17, a significant rewording of this feature[10] happened to instead find the base class constructors and then define how they are used to construct an instance of the derived class, as we have presented here. With a final fix in C++20 with the resolution of CWG issue #2356,[11] a complete working feature was specified. All of these fixes for C++17 were accepted as defect reports and thus apply retroactively to C++11 and C++14. For the major compilers, this was either standardizing already existing practice or quickly adopting the changes.[12]

The subsections that follow describe the subtle bugs that came with the previous specification, both for completeness and to give a better understanding of what to expect on very old compilers, though none fully implemented the original specification as written.

#### 1.19.7.1 Inheriting constructors declared with a C-style ellipsis

Forwarding arguments from a constructor declared using a C-style ellipsis cannot forward correctly. Arguments passed through the ellipsis are not available as named arguments but must instead be accessed through the `va_arg` family of macros. Without named arguments, no easily supported way is available to call the base-class constructor with the additional arguments:

```
struct Base
{
Base(int x, ...) { }  // constructor taking C-style variadic args
};

struct Derived : Base
```

---

[10] **smith15b**

[11] **smith18**

[12] For example, GCC versions above 7.0 and Clang versions above 4.0 all have the modern behavior fully implemented regardless of which standard version is chosen when compiling.

```
{
using Base::Base;  // Error: Prior to C++17 fixes, standard wording
// does not allow forwarding C-style variadic args.
};
```

This problem is sidestepped in C++17 because the base-class constructor becomes available just like any other base-class function made available through a `using` declaration in the derived class.

### 1.19.7.2 Inheriting constructors that rely on friendship to declare function parameters

When a constructor depends on access to a `private` member of a class (e.g., a `typedef`), an inheriting constructor does not implicitly grant friendship that the base class might have that makes the constructor valid. For example, consider the following class template, which grants friendship to class `B`:

```
template <typename T>
struct S
{
private:
typedef int X;
friend struct B;
};
```

Then, we can create a class with a constructor that relies on that friendship. In this case, we consider a constructor template using the dependent member `X`, assuming that, in the normal case, `X` would be publicly accessible:

```
struct B
{
template <typename T>
B(T, typename T::X);
};
```

Now consider class `D` derived from `B` and inheriting its constructors:

```
struct D : B
{
using B::B;
};
```

Without friendship, we cannot construct a `D` from an `S`, but we can construct a `B` from an `S`, suggesting something is wrong with the inheritance. Note that the SFINAE rules for templates mean that the inheriting constructor is a problem only if we try to construct an `S` with the problem type and does not cause a hard error without that use case. The following example illustrates the problematic usage:

```
S<int> s;    // full specialization of S for type int
B b(s, 2);   // OK, thanks to friendship
D d(s, 2);   // Error: Prior to C++17 fixes, friendship is not inherited.
```

As C++17 redefines the semantics of the inheriting constructor as if the base class’s constructors were merely exposed in the derived one, friendship is evaluated within the scope of the base class.

#### 1.19.7.3 Inheriting constructor templates would be ill formed for a local class

A class declared within a function is a **local class**. Local classes have many restrictions, one of which is that they cannot declare member templates. If we inherit constructors from a base class with constructor templates, even `private` ones, the implicit declaration of a constructor template to forward arguments to the base-class constructor would be **ill formed**:

```
struct Base
{
template <typename T>
Base(T);
};

void f()
{
class Local : Base
{
using Base::Base;  // Error: Prior to C++17 fixes, we cannot redeclare
// the constructor template in local class.
};
}
```

C++17 resolves this by directly exposing the base class constructors, rather than defining new constructors to forward arguments.

#### 1.19.7.4 SFINAE evaluation context with default function arguments

Constructors that employ **SFINAE** tricks in default function arguments perform **SFINAE** checks in the wrong context and therefore inherit ill-formed constructors. No such issues occur when these **SFINAE** tricks are performed on default template arguments instead. As an example, consider a class template `Wrap` that has a template constructor with a **SFINAE** constraint:

```
struct S { };

template <typename T>
struct Wrap
{
template <typename U>
Wrap(U, typename std::enable_if<
std::is_constructible<T, U>::value>::type* = nullptr)
// This constructor is enabled only if T is constructible from U.
{
std::cout << "SFINAE ctor\n";
}

Wrap(S)
{
std::cout << "S ctor\n";
}
};
```

If we derive from `Wrap` and inherit its constructors, we would expect the **SFINAE** constraint to behave exactly as in the base class, i.e., the template constructor overload would be silently discarded if `std::is_constructible<T, U>::value` evaluates to `false`:

```
template <typename T>
struct Derived : Wrap<T>
{
using Wrap<T>::Wrap;
};
```

However, prior to C++17's retroactive fixes, **SFINAE** was triggered only for `Wrap`, not for `Derived`:

```
void f()
{
S s;
Wrap<int> w(s);     // prints "S ctor"
Derived<int> d(s);  // error prior to fixes; prints "S ctor" afterward
}
```

### 1.19.7.5 Suppression of constructors in the presence of default arguments

A constructor having one or more default arguments in the derived class does not suppress any corresponding constructors matching only the nondefaulted arguments in the base class, leading to ambiguities:

```
struct B          // base class
{
B(int, int);  // value constructor with two (required) int parameters
};

struct D : B
{
using B::B;
D(int, int, int = 0);  // doesn't suppress D(int, int) from B(int, int)
};
```

In the code example above, the original defective behavior was that there would be two overloaded constructors in `D`; attempting to construct a `D` from two integers became ambiguous. In the corrected behavior, the inheriting `D(int, int)` from the base-class constructor `B(int, int)`, whose domain is fully subsumed by the derived class's explicitly specified constructor `D(int, int, int = 0)`, is suppressed.

### 1.19.7.6 Suprising behavior with unary constructor templates

Because inherited constructors are redeclarations within the derived class and expect to forward properly to the corresponding base-class constructors, constructor templates may do very surprising things. In particular, a gregarious, templated constructor can appear to cause inheritance of a base-class copy constructor. Consider the following class with a constructor template:

```
struct A
{
A() = default;
A(const A&) { std::cout << "copy\n"; }

template <typename T>
A(T) { std::cout << "convert\n"; }
};
```

This simple class can convert from any type and prints those of its constructors that were called. Now consider we want to make a **strong `typedef`** for `A`:

```
struct B : A
{
using A::A;  // inherited base class A's constructors
};
```

The problem is that because `A` can convert from anything, when `B` inherits `A`'s constructor template, `B` can then use the inherited constructor to construct an instance of `B` from `A`. Perhaps more surprising, because the definition of the inherited constructor in `B` is to initialize the `A` subobject with its parameters, the nontemplate inherited constructor will be chosen as the best match, not the templated, converting constructor![13]

---

[13] Note that if the template constructor for `A` were a *copy* or *move* constructor for `A`, then it would be excluded from being an inherited constructor and this odd behavior would be avoided. The by-value parameter of this constructor is also why `"copy"` is output twice in this example.

```
A x;
B y = x;  // Surprise! This compiles, and it prints "copy" twice!
```

## 1.20 Unicode String and Character Literals

C++11 introduces a portable mechanism for ensuring that a literal is encoded as UTF-8, UTF-16, or UTF-32.

### 1.20.1 Description

According to the C++ Standard, the character encoding of string literals is unspecified and can vary with the target platform or the configuration of the compiler. In essence, the C++ Standard does not guarantee that the string literal `"Hello"` will be encoded as the ASCII[1] sequence `0x48, 0x65, 0x6C, 0x6C, 0x6F` or that the the character literal `'X'` has the value `0x58`.

Table 1–1 illustrates three new kinds of *unicode*-compliant *string literals*, each delineating the precise encoding of each character.

**Table 1–1: Three new Unicode-compliant literal strings**

| Encoding | Syntax | Underlying Type |
|---|---|---|
| UTF-8 | `u8"Hello"` | `char` (`char8_t` in C++20) |
| UTF-16 | `u"Hello"` | `char16_t` |
| UTF-32 | `U"Hello"` | `char32_t` |

A unicode literal value is guaranteed to be encoded in UTF-8, UTF-16, or UTF-32, for `u8`, `u`, and `U` literals, respectively:

```
char s0[] = "Hello";
// unspecified encoding (albeit very likely ASCII)

char s1[] = u8"Hello";
// guaranteed to be encoded as {0x48, 0x65, 0x6C, 0x6C, 0x6F, 0x0}
```

C++11 also introduces *universal character names* that provide a reliably portable way of embedding Unicode code points in a C++ program. They can be introduced by the `\u` character sequence followed by four hexadecimal digits or by the `\U` character sequence followed by eight hexadecimal digits:

```
std::puts(u8"\U0001F378");  // Unicode code point in a UTF-8 encoded literal
```

This output statement is guaranteed to emit the cocktail emoji (🍸) to `stdout`, assuming that the receiving end is configured to interpret output bytes as UTF-8.

---

[1] In fact, C++ still fully supports platforms using EBCDIC, a rarely used alternative encoding to ASCII, as their primary text encoding.

## 1.20.2 Use Cases

### 1.20.2.1 Guaranteed-portable encodings of literals

The encoding guarantees provided by the Unicode literals can be useful, such as in communication with other programs or network/IPC protocols that expect character strings having a particular encoding.

As an example, consider an instant-messaging program, in which both the client and the server expect messages to be encoded in UTF-8. As part of broadcasting a message to all clients, the server code uses UTF-8 Unicode literals to guarantee that every client will receive a sequence of bytes they are able to interpret and display as human-readable text:

```
void Server::broadcastServerMessage(const std::string& message)
{
Packet data;
data << u8"Message from the server: '" << message << u8"'\n";

broadcastPacket(data);
}
```

Not using `u8` literals in the code snippet above could potentially result in nonportable behavior and might require compiler-specific flags to ensure that the source is UTF-8 encoded.

## 1.20.3 Potential Pitfalls

### 1.20.3.1 Embedding Unicode graphemes

The addition of Unicode string literals to the language did not bring along an extension of the basic source character set: Even in C++11, the default **basic source character set** is a subset of ASCII.[2]

Developers might be fooled into thinking that `u8"🍸"` is a portable way of embedding a string literal representing the cocktail emoji in a C++ program, but they would be mistaken. The representation of the string literal depends on what encoding the compiler assumes for the source file, which can generally be controlled through compiler flags. The only portable way of embedding the cocktail emoji is to use its corresponding Unicode code point escape sequence (`u8"\U0001F378"`), as demonstrated in *Description* on page 140.

### 1.20.3.2 Lack of library support for Unicode

Essential **vocabulary types**, such as `std::string`, are completely unaware of encoding. They treat any stored string as a sequence of bytes. Even when correctly using Unicode string literals, programmers unfamiliar with Unicode might be surprised by seemingly innocent operations, such as asking for the size of a string representing the cocktail emoji:

```
void f()
{
std::string cocktail(u8"\U0001F378");  // big character (!)
assert(cocktail.size() == 1);          // assertion failure (!)
}
```

---

[2] Implementations are free to map characters outside the basic source character set to sequences of its members, resulting in the possibility of embedding other characters, such as emojis, in a C++ source file.

Even though the cocktail emoji is a *single* code point, `std::string::size` returns the number of code units required to encode it. The lack of Unicode-aware vocabulary types and utilities in the Standard Library can be a source of defects and misunderstandings, especially in the context of international program localization.

#### 1.20.3.3 UTF-8 quirks

UTF-8 string literals use `char` as their **underlying type**. Such a choice is inconsistent with UTF-16 and UTF-32 literals, which provide their own unique character types (`char16_t` and `char32_t`), and prevents any overloading or template specialization on UTF-8 strings because it would be indistinguishable from default, narrow literal encoding. Furthermore, whether the underlying type of `char` is a `signed` or `unsigned` type is itself implementation defined.[3]

C++20 fundamentally changes how UTF-8 string literals work, by introducing a new nonaliasing `char8_t` character type whose representation is guaranteed to match `unsigned char`. The new character type provides several benefits:

- Ensures an `unsigned` and distinct type for UTF-8 character data
- Enables overloading for regular string literals versus UTF-8 string literals
- Potentially achieves better performance due to the lack of special aliasing rules

Unfortunately, the changes brought by C++20 are not backward-compatible and might cause code targeting previous versions of the language using `u8` literals either to fail to compile or to silently change its behavior when targeting C++20:

```
template <typename T> void print(const T*);  // (0)
void print(const char*);                     // (1)

void f()
{
print(u8"text");  // invokes (1) prior to C++20, (0) afterwards
}
```

### 1.20.4 Annoyances

None so far

### 1.20.5 See Also

None so far

### 1.20.6 Further Reading

None so far

---

[3] Note that `char` is distinct from both `signed char` and `unsigned char`, but its behavior is guaranteed to be the same as one of those.

## 1.21 Explicit Enumeration Underlying Type

The underlying type of an enumeration is the fundamental **integral type** used to represent its enumerated values, which can be specified explicitly in C++11.

### 1.21.1 Description

Every enumeration employs an integral type, known as its **underlying type**, to represent its compile-time-enumerated values.[1] By default, the **underlying type** of an `enum`[2] is chosen by the implementation to be large enough to represent all of the values in an enumeration and is allowed to exceed the size of an `int` *only* if there are enumerators having values that cannot be represented as an `int` or `unsigned int`:

```
enum RGB { e_RED, e_GREEN, e_BLUE };                      // fits any char

enum Port { e_LEFT = -81, e_RIGHT = -82 };                // fits signed char

enum Mask { e_LOW = 32767, e_HIGH = 65535 };              // fits unsigned short

enum Big { e_32 = 1<<32 };                                // too big for int

enum Err { K = 1024, M = K*K, G = M*K, T = G*K };         // G*K overflows int...

enum OK { K = 1<<10, M = 1<<20, G = 1<<30, T = 1<<40 }; // ...but 1<<40 is OK
```

The default underlying type chosen for an `enum` is always sufficiently large to represent *all* enumerator values defined for that `enum`. If the value doesn't fit in an `int`, it will be selected deterministically as the first type able to represent all values from the sequence: `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`.[3]

#### 1.21.1.1 Specifying underlying type explicitly

As of C++11, we have the ability to specify the **integral type** that is used to represent an `enum`. This is achieved by providing the type explicitly in the `enum`'s declaration following the enumeration's (optional) name and preceded by a colon:

```
enum Port : unsigned char
{
// Each enumerator of Port is represented as an unsigned char type.
```

---

[1] Note that `char` and `wchar_t`, like enumerations, are their own distinct types (as opposed to `typedef`-like aliases such as `std::uint8_t`) and have their own implemenation-defined underlying integral types. With `char`, for example, the **underlying type** will always be either `signed char` or `unsigned char` (both of which are also distinct C++ types). The same is true in C++11 for `char16_t` and `char32_t` and in C++20 for `char8_t`.

[2] Note that the default underlying type of an `enum class` is ubiquitously `int`, and it is not implementation defined; see “`enum class`” on page 148.

[3] While specifying an enumeration's underlying type was impossible before C++11, the compiler could be forced to choose at least a 32-bit or 64-bit signed integral type by adding an enumerator having a sufficiently large negative value — e.g., `-1 << 31` for a 32-bit and `-1 << 63` for a 64-bit *signed* integer (assuming such is available on the target platform).

```
e_INPUT        =  37,  // OK, would have fit in a signed char too
e_OUTPUT       = 142,  // OK, would not have fit in a signed char
e_CONTROL      = 255,  // OK, barely fits in an 8-bit unsigned integer
e_BACK_CHANNEL = 256,  // error, doesn't fit in an 8-bit unsigned integer
};
```

If any of the values specified in the definition of the `enum` is outside the boundaries of what the provided **underlying type** is able to represent, the compiler will emit an error, but see *Potential Pitfalls: Subtleties of integral promotion* on page 146.

## 1.21.2 Use Cases

### 1.21.2.1 Ensuring a compact representation where enumerator values are salient

When the enumeration needs to have an efficient representation, e.g., when it is used as a data member of a widely replicated type, restricting the width of the underlying type to something smaller than would occur by default on the target platform might be justified.

As a concrete example, suppose that we want to enumerate the months of the year, for example, in anticipation of placing that enumeration inside a date class having an internal representation that maintains the year as a two-byte signed integer, the month as an enumeration, and the day as an 8-bit signed integer:

```
#include <cstdint>  // std::int8_t, std::int16_t

class Date
{
std::int16_t d_year;
Month d_month;
std::int8_t day;

public:
Date(int year, Month month, int day);

// ...

int year() const    { return d_year; }
Month month() const { return d_month; }
int day() const     { return d_day; }
}
```

Within the software, the `Date` is typically constructed using the values obtained through the GUI, where the month is always selected from a drop-down menu. Management has requested that the month be supplied to the constructor as an `enum` to avoid recurring defects where the individual fields of the date are supplied in month/day/year format. New functionality will be written to expect the month to be enumerated. Still, the date class will be used in contexts where the numerical value of the month is significant, such as in calls to legacy functions that accept the month as an integer. Moreover, iterating over a range of months is common and requires that the enumerators convert automatically to their (integral) **underlying type**, thus contraindicating use of the more strongly typed `enum class`:

```
enum Month  // defaulted underlying type (BAD IDEA)
{
e_JAN = 1, e_FEB, e_MAR,    // winter
e_APR    , e_MAY, e_JUN,    // spring
e_JUL    , e_AUG, e_SEP,    // summer
e_OCT    , e_NOV, e_DEC     // autumn
};

static_assert(sizeof(Month) == 4 && alignof(Month) == 4, "");
```

As it turns out, date values are used widely throughout this code base, and the proposed `Date` type is expected to be used in large aggregates. The underlying type of the `enum` in the code snippet above is implementation-defined and could be as small as a `char` or as large as an `int` despite all the values fitting in a `char`. Hence, if this enumeration were used as a data member in the `Date` class, `sizeof(Date)` would likely balloon to 12 bytes on some relevant platforms due to **natural alignment**! (See "`alignas`" on page 68.)

While reordering the data members of `Date` such that `d_year` and `d_day` were adjacent would ensure that `sizeof(Date)` would not exceed 8 bytes, a better approach is to explicitly specify the enumeration's underlying type to ensure `sizeof(Date)` is exactly the 4 bytes needed to accurately represent the value of the `Date` object. Given that the values in this enumeration fit in an 8-bit signed integer, we can specify its **underlying type** to be, e.g., `std::int8_t` or `signed char`, on every platform:

```
enum Month : std::int8_t  // user-provided underlying type (GOOD IDEA)
{
e_JAN = 1, e_FEB, e_MAR,    // winter
e_APR    , e_MAY, e_JUN,    // spring
e_JUL    , e_AUG, e_SEP,    // summer
e_OCT    , e_NOV, e_DEC     // autumn
};

static_assert(sizeof(Month) == 1 && alignof(Month) == 1, "");
```

With this revised definition of `Month`, the size of a `Date` class is 4 bytes, which is especially valuable for large aggregates:

```
Date timeSeries[1000 * 1000];  // sizeof(timeSeries) is now 4Mb (not 12Mb)
```

## 1.21.3 Potential Pitfalls

### 1.21.3.1 External use of opaque enumerators

Providing an explicit underlying type to an `enum` enables clients to declare or redeclare it as a complete type with or without its enumerators. Unless the opaque form of its definition is exported in a header file separate from its full definition, external clients wishing to exploit the opaque version will be forced to locally declare it with its **underlying type** but without its enumerator list. If the underlying type of the full definition were to change, any program incorporating *its own* original and now inconsistent elided definition and the *new* full one would become silently ill formed, no diagnostic required (**IFNDR**). (See "Opaque Enumeration Declarations" on page 166.)

#### 1.21.3.2 Subtleties of integral promotion

When used in an arithmetic context, one might naturally assume that the type of a classic `enum` will first convert to its **underlying type**, which is not always the case. When used in a context that does not explicitly operate on the `enum` type itself, such as a parameter to a function that takes that enum type, **integral promotion** comes into play. For unscoped enumerations without an explicitly specified underlying type and for character types such as `wchar_t`, `char16_t`, and `char32_t`, integral promotion will directly convert the value to the first type in the list `int`, `unsigned int`, `long`, `unsigned long`, `long long`, and `unsigned long long` that is sufficiently large to represent all of the values of the underlying type. Enumerations having a fixed underlying type will, as a first step, behave as if they had decayed to their underlying type.

In most arithmetic expressions, this difference is irrelevant. Subtleties arise, however, when one relies on overload resolution for identifying the underlying type:

```
void f(signed char x);
void f(short x);
void f(int x);
void f(long x);
void f(long long x);

enum E1         { a, b, c, d, e };
enum E2 : short { f, g, h, i ,j };

void g()
{
f(E1::a);  // always calls f(int) on all platforms
f(E2::f);  // always calls f(short) on all platforms
}
```

The overload resolution for `f` considers the type to which each *individual* enumerator can be directly integrally promoted. This conversion for `E1` can be only to `int`. For `E2`, the conversion will consider `int` *and* `short`, and `short`, being an exact match, will be selected. Note that even though both enumerations are small enough to fit into a `signed char`, that overload of `f` will never be selected.

One might want to get to the implementation-defined underlying type though, and the standard does provide a trait to do that: `std::underlying_type` in C++11 and the corresponding `std::underlying_type_t` alias in C++14. This trait can safely be used in a cast without risking loss of value[4]:

```
template <typename E>
std::underlying_type<E>::type toUnderlying(E value)
{
```

---

[4] See "**??**" on page **??**.

```
return static_cast<std::underlying_type<E>::type>(value);
}

void h()
{
auto e1 = toUnderlying(E1::a); // might be anywhere from signed char to int
auto e2 = toUnderlying(E2::f); // always deduced as short
}
```

As of C++20, however, the use of a classic enumerator in a context in which it is compared to or otherwise used in a binary operation with either an enumerator of another type or a nonintegral type (i.e., a floating-point type, such as `float`, `double`, or `long double`) is deprecated, with the the possibility of being removed in C++23. Platforms might decide to warn against such uses retroactively:

```
enum { k_GRAMS_PER_OZ = 28 };  // not the best idea

double gramsFromOunces(double ounces)
{
return ounces * k_GRAMS_PER_OZ;  // deprecated in C++20; might warn
}
```

Casting to the **underlying type** is *not* necessarily the same as direct integral promotion. In this context, we might want to change our `enum` to a `constexpr int`[5] in the long term:

```
constexpr int k_GRAMS_PER_OZ = 28;  // future proof
```

### 1.21.4 See Also

- "`enum class`" on page 148 — a scoped, more strongly typed enumeration
- "Opaque Enumeration Declarations" on page 166 — a means of **insulating** individual enumerators from clients
- "`constexpr` Variables" on page 228 — an alternative way of declaring compile-time constants

### 1.21.5 Further Reading

TODO

---

[5] See "`constexpr` Variables" on page 228.

# 1.22 Strongly Typed Enumerations (`enum class`)

`enum class` is an alternative to the classic `enum` construct that simultaneously provides both stronger typing and an enclosing scope for its enumerated values.

## 1.22.1 Description

Classic, C-style enumerations are useful and continue to fulfill important engineering needs:

```
enum EnumName { e_Enumerator0 /*= value0 */, e_EnumeratorN /* = valueN */ };
// classic, C-style enum: enumerators are neither type-safe nor scoped
```

For more examples where the classic `enum` shines, see *Potential Pitfalls: Strong typing of an* `enum class` *can be counterproductive* on page 159 and *Annoyances: Scoped enumerations do not necessarily add value* on page 164. Still, innumerable practical situations occur in which enumerators that are both scoped and more type-safe would be preferred; see *Introducing the C++11* `enum class` on page 150.

### 1.22.1.1 Drawbacks and workarounds relating to unscoped C++03 enumerations

Since the enumerators of a classic `enum` leak out into the enclosing scope, if two unrelated enumerations that happen to use the same enumerator name appear in the same scope, an ambiguity could ensue:

```
enum Color { e_RED, e_ORANGE, e_YELLOW };    // OK
enum Fruit { e_APPLE, e_ORANGE, e_BANANA };  // Error: e_ORANGE is redefined.
```

The problems associated with the use of unscoped enumerations is exacerbated when those enumerations are placed in their own respective header files in the global or some other large namespace scope, such as `std`, for general reuse. In such cases, latent defects will typically not manifest unless and until the two enumerations are included in the same translation unit.[1]

If the only issue were the leakage of the enumerators into the enclosing scope, then the long-established workaround of enclosing the enumeration within a `struct` would suffice:

```
struct Color { enum Enum { e_RED, e_ORANGE, e_YELLOW };  };   // OK
struct Fruit { enum Enum { e_APPLE, e_ORANGE, e_BANANA }; };  // OK (scoped)
```

Employing the C++03 workaround in the above code snippet implies that, when passing such an explicitly scoped, classical `enum` into a function, the distinguishing name of the `enum` is subsumed by its enclosing `struct` and the `enum` name itself, such as `Enum`, becomes **boilerplate code**:

```
int enumeratorValue1 = Color::e_ORANGE;  // OK
int enumeratorValue2 = Fruit::e_ORANGE;  // OK

void colorFunc(Color::Enum color);  // enumerated (scoped) Color parameter
void fruitFunc(Fruit::Enum fruit);  // enumerated (scoped) Fruit parameter
```

---

[1] Note that we use a lowercase, single-letter prefix, such as `e_`, to ensure that the uppercase enumerator name is less likely to collide with a legacy macro, which is especially useful in header files.

Hence, adding *just* scope to a classic, C++03 `enum` is easily doable and might be exactly what is indicated; see *Potential Pitfalls: Strong typing of an* `enum class` *can be counterproductive* on page 159.

### 1.22.1.2 Drawbacks relating to weakly typed, C++03 enumerators

Historically, C++03 enumerations have been employed to represent at least two distinct concepts:

1. A collection of related, but not necessarily unique, named integral values
2. A pure, perhaps ordered, set of named entities in which cardinal value has no relevance

It will turn out that the modern `enum class` feature, which we will discuss in *Description: Introducing the C++11* `enum class`, is more closely aligned with this second concept.

A classic enumeration, by default, has an implementation-defined **underlying type** (see “Explicit Enumeration Underlying Type” on page 143), which it uses to represent variables of that enumerated type as well as the values of its enumerators. While implicit conversion *to* an enumerated type is never permitted, when implicitly converting *from* a classical `enum` type to some arithmetic type, the `enum` promotes to integral types in a way similar to how its underlying type would promote using the rules of **integral promotion** and **standard conversion**:

```
void f()
{
enum A { e_A0, e_A1, e_A2 };  // classic, C-style C++03 enum
enum B { e_B0, e_B1, e_B2 };  //    "       "       "     "

A a;  // Declare object a to be of type A.
B b;  //   "       "    b  " "  "       B.

a = e_B2;  // error: cannot convert e_B2 to enum type A
b = e_B2;  // OK, assign the value e_B2 (numerically 2) to b.
a = b;     // error: cannot convert enumerator b to enum type A
b = b;     // OK, self-assignment
a = 1;     // error: invalid conversion from int 1 to enum type A
a = 0;     // error: invalid conversion from int 0 to enum type A

bool     v = a;     // OK
char     w = e_A0;  // OK
unsigned y = e_B1;  // OK
float    x = b;     // OK
double   z = e_A2;  // OK
char*    p = e_B0;  // error: unable to convert e_B0 to char*
char*    q = +e_B0; // error: invalid conversion of int to char*
}
```

Notice that, in this example, the final two diagnostics for the attempted initializations of `p` and `q`, respectively, differ slightly. In the first, we are trying to initialize a pointer, `p`, with an enumerated type, `B`. In the second, we have creatively used the built-in unary-plus operator to explicitly promote the enumerator to an integral type before attempting to assign it to a pointer, `q`. Even though the numerical value of the enumerator is `0` and such is known at at compile time, implicit conversion to a pointer type from anything but the literal integer constant `0`[2] is not permitted.

C++ fully supports comparing values of *classic* `enum` types with values of arbitrary **arithmetic type** as well as those of the same enumerated type; the operands of a comparator will be promoted to a sufficiently large integer type and the comparison will be done with those values. Comparing values having distinct enumerated types, however, is deprecated and will typically elicit a warning.[3]

### 1.22.1.3 Introducing the C++11 `enum class`

With the advent of modern C++, we now have a new, alternative enumeration construct, `enum class`, that simultaneously addresses strong type safety and lexical scoping, two distinct and often desirable properties:

```
enum class Name { e_Enumerator0 /* = value0 */, e_EnumeratorN /* = valueN */ };
// enum class enumerators are both type-safe and scoped
```

Another major distinction is that the default **underlying type** for a C-style `enum` is **implementation defined**, whereas, for an `enum class`, it is always an `int`. See *Description:* `enum class` *and underlying type* on page 152 and *Potential Pitfalls: External use of opaque enumerators* on page 164.

The enumerators within an `enum class` are all scoped by its name, while classic enumerations leak the enumerators into the enclosing scope:

```
enum Vehicle  { e_CAR,   e_TRAIN, e_PLANE };
enum Geometry { e_POINT, e_LINE,  e_PLANE };  // Error: e_PLANE is redefined.
```

Unlike unscoped enumerations, `enum class` does not leak its enumerators into the enclosing scope and can therefore help avoid collisions with other enumerations having like-named enumerators defined in the same scope:

```
enum       VehicleUnscoped  { e_CAR, e_TRAIN, e_PLANE };
struct     VehicleScopedExplicitly { enum { e_CAR, e_TRAIN, e_PLANE }; };
enum class VehicleScopedImplicitly { e_CAR, e_BOAT,  e_PLANE };
```

---

[2] Excluding esoteric user-defined types, only a literal `0` or, as of C++11, a value of type `std::nullptr_t` is implicitly convertible to an arbitrary pointer type; see "Null Pointer Literal (`nullptr`)" on page 64.

[3] As of C++20, attempting to compare two values of distinct classically enumerated types is a compile-time error. Note that explicitly converting at least one of them to an integral type — for example, using built-in unary plus — both makes our intentions clear and avoids warnings.

```
if (e_A0 < 0)       { /* ... */ }  // OK, comparison with integral type
if (1.0 != e_B1)    { /* ... */ }  // OK, comparison with arithmetic type
if (A() <= e_A2)    { /* ... */ }  // OK, comparison with same enumerated type
if (e_A0 == e_B0)   { /* ... */ }  // warning, deprecated (error as of C++20)
if ( e_A0 == +e_B0) { /* ... */ }  // OK, unary + converts to integral type
if (+e_A0 ==  e_B0) { /* ... */ }  // OK,   "        "     "     "      "
if (+e_A0 == +e_B0) { /* ... */ }  // OK,   "        "     "     "      "
```

Just like an unscoped `enum` type, an object of type `enum class` is passed as a parameter to a function using the enumerator name itself[4]:

```
void f1(VehicleUnscoped value);          // classic enumeration passed by value
void f2(VehicleScopedImplicitly value);  // modern enumeration passed by value
```

Qualifying the enumerators of a scoped enumeration is the same, irrespective of whether the scoping is explicit or implicit:

```
void g()
{
f1(VehicleUnscoped::e_PLANE);
// call f1 with an explicitly scoped enumerator

f2(VehicleScopedImplicitly::e_PLANE);
// call f2 with an implicitly scoped enumerator
}
```

Apart from implicit scoping, the modern, C++11 `enum class` deliberately does *not* support implicit conversion, in any context, to its **underlying type**:

```
int i1 = VehicleScopedExplicitly::e_PLANE;
// OK, scoped C++03 enum (implicit conversion)

int i2 = VehicleScopedImplicitly::e_PLANE;
// error: no implicit conversion to underlying type

if (VehicleScopedExplicitly::e_PLANE > 3) { /* OK */ }
if (VehicleScopedImplicitly::e_PLANE > 3) { /* error: implicit conversion */ }
```

Enumerators of an `enum class` do, however, admit equality and ordinal comparisons within their own type:

```
enum class E { e_A, e_B, e_C };  // By default, enumerators increase from 0.

static_assert(E::e_A < E::e_C, "");  // OK, comparison between same-type values
static_assert(0 == E::e_A, "");      // error: no implicit conversion from E
static_assert(0 == static_cast<int>(E::e_A), "");  // OK, explicit conversion

void f(E v)
{
if (v > E::e_A) { /* ... */ }  // OK, comparing values of same type, E
}
```

---

[4] If we use the approach for adding scope to enumerators that is described in *Description: Drawbacks relating to weakly typed, C++03 enumerators* on page 149, the name of the enclosing `struct` together with a consistent tag, such as `Enum`, has to be used to indicate an enumerated type:

```
void f1(VehicleScopedExplicitly::Enum value);
// classically scoped enum passed by value
```

Note that incrementing an enumerator variable from one strongly typed enumerator's value to the next requires an explicit cast; see *Potential Pitfalls: Strong typing of an* `enum class` *can be counterproductive* on page 159.

### 1.22.1.4 `enum class` and underlying type

Since C++11, both scoped and unscoped enumerations permit explicit specification of their integral **underlying type**:

```
enum Ec : char { e_X, e_Y, e_Z };
// underlying type is char

static_assert(1 == sizeof(Ec),     "");
static_assert(1 == sizeof Ec::E_X, "");

enum class Es : short { e_X, e_Y, e_Z };
// underlying type is short int

static_assert(sizeof(short) == sizeof(Es),     "");
static_assert(sizeof(short) == sizeof Es::E_X, "");
```

Unlike a classic `enum`, which has an **implementation-defined** default **underlying type**, the default **underlying type** for an `enum class` is always `int`:

```
enum class Ei { e_X, e_Y, e_Z };
// When not specified the underlying type of an enum class is int.

static_assert(sizeof(int) == sizeof(Ei),     "");
static_assert(sizeof(int) == sizeof Ei::E_X, "");
```

Note that, because the default **underlying type** of an `enum class` is specified by the Standard, eliding the enumerators[5] of an `enum class` in a local redeclaration is *always* possible; see *Potential Pitfalls: External use of opaque enumerators* on page 164.

## 1.22.2 Use Cases

### 1.22.2.1 Avoiding unintended implicit conversions to arithmetic types

Suppose that we want to represent the result of selecting one of a fixed number of alternatives from a drop-down menu as a simple unordered set of uniquely valued named integers. For example, this might be the case when configuring a product, such as a vehicle, for purchase:

```
struct Trans
{
enum Enum { e_MANUAL, e_AUTOMATIC };  // classic, C++03 scoped enum
};
```

Although automatic promotion of a classic enumerator to `int` works well when typical use of the enumerator involves knowing its cardinal value, such promotions are less than ideal when cardinal values have no role in intended usage:

---

[5] See "Opaque Enumeration Declarations" on page 166.

```
class Car { /* ... */ };

struct Trans
{                                          // explicitly scoped
enum Enum { e_MANUAL, e_AUTOMATIC };  // classic enum
};                                         // (BAD IDEA)

int buildCar(Car* result, int numDoors, Trans::Enum trans)
{
int status = Trans::e_MANUAL;    // Oops, accidental misuse!

for (int i = 0; i < trans; ++i)  // Oops, accidental misuse!
{
attachDoor(i);
}

return status;
}
```

As shown in the example above, it is never correct for a value of type `Trans::Enum` to be assigned to, compared with, or otherwise modified like an integer; hence, *any* such use would necessarily be considered a mistake and, ideally, flagged by the compiler as an error. The stronger typing provided by `enum class` achieves this goal:

```
class Car { /* ... */ };

enum class Trans { e_MANUAL, e_AUTOMATIC };  // modern enum class (GOOD IDEA)

int buildCar(Car* result, int numDoors, Trans trans)
{
int status = Trans::e_MANUAL;    // error: incompatible types

for (int i = 0; i < trans; ++i)  // error: incompatible types
{
attachDoor(i);
}

return status;
}
```

By deliberately choosing the `enum class` over the *classic* `enum` above, we automate the detection of many common kinds of accidental misuse. Secondarily, we slightly simplify the interface of the function signature by removing the extra `::Enum` boilerplate qualifications required of an explicitly scoped, less-type-safe, classic `enum`, but see *Potential Pitfalls: Strong typing of an* `enum class` *can be counterproductive* on page 159.

In an unlikely event that the numeric value of a strongly typed enumerator is needed (e.g., for serialization), it can be extracted explicitly via a `static_cast`:

```
const int manualIntegralValue    = static_cast<int>(Trans::e_MANUAL);
const int automaticIntegralValue = static_cast<int>(Trans::e_AUTOMATIC);
```

```
static_assert(0 == manualIntegralValue,    "");
static_assert(1 == automaticIntegralValue, "");
```

### 1.22.2.2 Avoiding namespace pollution

Classic, C-style enumerations do not provide scope for their enumerators, leading to unintended latent name collisions:

```
// vehicle.h
// ...
enum Vehicle  { e_CAR, e_TRAIN, e_PLANE };  // classic, C-style enum
// ...

// geometry.h
// ...
enum Geometry { e_POINT, e_LINE,  e_PLANE };  // classic, C-style enum
// ...

// client
#include <vehicle.h>  // OK
#include <geometry.h> // error: e_PLANE redefined
// ...
```

The common workaround is to wrap the `enum` in a `struct` or `namespace`:

```
// vehicle.h
// ...
struct Vehicle {                            // explicitly scoped
enum Enum { e_CAR, e_TRAIN, e_PLANE };  // classic, C-style enum
};
// ...

// geometry.h
// ...
struct Geometry {                            // explicitly scoped
enum Enum { e_POINT, e_LINE, e_PLANE };  // classic, C-style enum
};
// ...

// client
#include <vehicle.h>    // OK
#include <geometry.h>   // OK, enumerators are scoped explicitly.
// ...
```

If implicit conversions of enumerators to integral types are not required, we can achieve the same scoping effect with much more type safety and slightly less boilerplate — i.e., without the `::Enum` when declaring a variable — by employing `enum class` instead:

```
// vehicle.h
// ...
enum class Vehicle { e_CAR, e_TRAIN, e_PLANE };
```

```
// ...

// geometry.h
// ...
enum class Geometry { e_POINT, e_LINE, e_PLANE };
// ...

// client
#include <vehicle.h>  // OK
#include <geometry.h>   // OK, enumerators are scoped implicitly.
// ...
```

### 1.22.2.3 Improving overloading disambiguation

Overloaded functions are notorious for providing opportunities for misuse. Maintenance difficulties are exacerbated when arguments for these overloads are convertible to more than a single parameter in the function. As an illustration of the compounding of such maintenance difficulties, suppose that we have a widely used, named type, `Color`, and the numeric values of its enumerators are small, unique, and irrelevant. Imagine we have chosen to represent `Color` as a *classic* `enum`:

```
struct Color
{                                            // explicitly scoped
enum Enum { e_RED, e_BLUE /*, ...*/ };  // classic, C-style enum
};                                           // (BAD IDEA)
```

Suppose further that we have provided two overloaded functions, each having two parameters, with one signature's parameters including the enumeration `Color`:

```
void clearScreen(int pattern, int orientation);          // (0)
void clearScreen(Color::Enum background, double alpha);  // (1)
```

Depending on the types of the arguments supplied, one or the other functions will be selected or else the call will be ambiguous and the program will fail to compile[6]:

```
void f0()
{
clearScreen(1           , 1           );  // calls (0) above
```

---

[6] GCC version 7.4.0 incorrectly diagnoses both ambiguity errors as warnings, although it states in the warning that it is an error:

```
warning: ISO C++ says that these are ambiguous, even though the worst conversion for the
first is better than the worst conversion for the second:

note: candidate 1: void clearScreen(int, int)
void clearScreen(int pattern, int orientation);
^~~~~~~~~~~
note: candidate 2: void clearScreen(Color::Enum, double)
void clearScreen(Color::Enum background, double alpha;
^~~~~~~~~~~
```

```
clearScreen(1           , 1.0        );  // calls (0) above
clearScreen(1           , Color::e_RED);  // calls (0) above

clearScreen(1.0         , 1          );  // calls (0) above
clearScreen(1.0         , 1.0        );  // calls (0) above
clearScreen(1.0         , Color::e_RED);  // calls (0) above

clearScreen(Color::e_RED, 1          );  // error: ambiguous call
clearScreen(Color::e_RED, 1.0        );  // calls (1) above
clearScreen(Color::e_RED, Color::e_RED);  // error: ambiguous call
}
```

Now suppose that we had instead defined our `Color` enumeration as a modern `enum class`:

```
enum class Color { e_RED, e_BLUE /*, ...*/ };

void clearScreen(int pattern, int orientation);    // (2)
void clearScreen(Color background, double alpha);  // (3)
```

The function that will be called from a given set of arguments becomes clear:

```
void f1()
{
clearScreen(1           , 1          );  // calls (2) above
clearScreen(1           , 1.0        );  // calls (2) above
clearScreen(1           , Color::e_RED);  // error: no matching function

clearScreen(1.0         , 1          );  // calls (2) above
clearScreen(1.0         , 1.0        );  // calls (2) above
clearScreen(1.0         , Color::e_RED);  // error: no matching function

clearScreen(Color::e_RED, 1          );  // calls (3) above
clearScreen(Color::e_RED, 1.0        );  // calls (3) above
clearScreen(Color::e_RED, Color::e_RED);  // error: no matching function
}
```

Returning to our original, classic-`enum` design, suppose that we find we need to add a third parameter, `bool z`, to the second overload:

```
void clearScreen(int pattern, int orientation);                  // (0)
void clearScreen(Color::Enum background, double alpha, bool z);  // (4) classic
```

If our plan is that any existing client calls involving `Color::Enum` will now be flagged as errors, we are going to be very disappointed:

```
void f2()
{
clearScreen(Color::e_RED, 1.0);  // calls (0) above
}
```

In fact, every combination of arguments above — all nine of them — will call function (0) above with no warnings at all:

```
void f3()
{
clearScreen(1          , 1          );  // calls (0) above
clearScreen(1          , 1.0        );  // calls (0) above
clearScreen(1          , Color::e_RED);  // calls (0) above

clearScreen(1.0        , 1          );  // calls (0) above
clearScreen(1.0        , 1.0        );  // calls (0) above
clearScreen(1.0        , Color::e_RED);  // calls (0) above

clearScreen(Color::e_RED, 1          );  // calls (0) above
clearScreen(Color::e_RED, 1.0        );  // calls (0) above
clearScreen(Color::e_RED, Color::e_RED);  // calls (0) above
}
```

Finally, let's suppose again that we have used `enum class` to implement our `Color` enumeration:

```
void clearScreen(int pattern, int orientation);           // (2)
void clearScreen(Color background, double alpha, bool z);  // (5) modern

void f4()
{
clearScreen(Color::e_RED, 1.0);  // error: no matching function
}
```

And in fact, the *only* calls that succeed unmodified are precisely those that do not involve the enumeration `Color`, as desired:

```
void f5()
{
clearScreen(1          , 1          );  // calls (2) above
clearScreen(1          , 1.0        );  // calls (2) above
clearScreen(1          , Color::e_RED);  // error: no matching function

clearScreen(1.0        , 1          );  // calls (2) above
clearScreen(1.0        , 1.0        );  // calls (2) above
clearScreen(1.0        , Color::e_RED);  // error: no matching function

clearScreen(Color::e_RED, 1          );  // error: no matching function
clearScreen(Color::e_RED, 1.0        );  // error: no matching function
clearScreen(Color::e_RED, Color::e_RED);  // error: no matching function
}
```

Bottom line: Having a *pure* enumeration — such as `Color`, used widely in function signatures — be strongly typed can only help to expose accidental misuse but, again, see *Potential Pitfalls: Strong typing of an* `enum class` *can be counterproductive* on page 159.

Note that strongly typed enumerations help to avoid accidental misuse by requiring an explicit *cast* should conversion to an arithmetic type be desired:

```
void f6()
```

```
{
clearScreen(Color::e_RED, 1.0);                      // error: no match
clearScreen(static_cast<int>(Color::e_RED), 1.0); // OK, calls (2) above
clearScreen(Color::e_RED, 1.0, false);               // OK, calls (5) above
}
```

### 1.22.2.4 Encapsulating implementation details within the enumerators themselves

In rare cases, providing a pure, ordered enumeration having unique, but not necessarily contiguous, numerical values that exploit lower-order bits[7] to categorize (and make readily available) important individual properties might offer an advantage, such as in performance.

For example, suppose that we have a `MonthOfYear` enumeration that encodes the months that have 31 days in their least-significant bit and an accompanying `inline` function to quickly determine whether a given enumerator represents such a month[8]:

```
enum class MonthOfYear : unsigned char  // optimized to flag long months
{
e_JAN = ( 1 << 4) + 0x1,
e_FEB = ( 2 << 4) + 0x0,
e_MAR = ( 3 << 4) + 0x1,
e_APR = ( 4 << 4) + 0x0,
e_MAY = ( 5 << 4) + 0x1,
e_JUN = ( 6 << 4) + 0x0,
e_JUL = ( 7 << 4) + 0x1,
e_AUG = ( 8 << 4) + 0x1,
e_SEP = ( 9 << 4) + 0x0,
e_OCT = (10 << 4) + 0x1,
e_NOV = (11 << 4) + 0x0,
e_DEC = (12 << 4) + 0x1
};

bool hasThirtyOneDays(MonthOfYear month)
{
return static_cast<std::underlying_type<MonthOfYear>::type>(month) & 0x1;
}
```

In such cases, the public clients are not intended to make use of the cardinal values; hence clients are well advised to treat them as implementation details, potentially subject to change without notice. Representing this enumeration using the modern `enum class`, instead of an explicitly scoped classic `enum`, deters clients from making any use (apart from same-type comparisons) of the cardinal values assigned to the enumerators. Notice that implementors of the `hasThirtyOneDays` function will require a verbose but runtime efficient `static_cast` to resolve the cardinal value of the enumerator and thus make the requested determination as efficiently as possible.

---

[7] To preserve the ordinality of the enumerators overall, the higher-level bits must encode their relative order. The lower-level bits are then available for arbitrary use in the implementation.

[8] In this example, we are using a new cross-cutting feature of all enumerated types that allows the client defining the type to specify its underlying type precisely. In this case, we have chosen an `unsigned char` to maximize the number of flag bits while keeping the overall size to a single byte. Three bits remain available. Had we needed more flag bits, we could have just as easily used a larger underlying type, such as `unsigned short`; see “Explicit Enumeration Underlying Type” on page 143.

## 1.22.3 Potential Pitfalls

### 1.22.3.1 Strong typing of an `enum class` can be counterproductive

The additive value in using a modern `enum class` is governed *solely* by whether its stronger typing, *not* its implicit scoping, of its enumerators would be beneficial in its anticipated typical usage. If the expectation is that the client will never to need to know the specific values of the enumerators, then use of the modern `enum class` is often just what's needed. But if the cardinal values themselves are ever needed during typical use, extracting them will require the client to perform an explicit cast. Beyond mere inconvenience, encouraging clients to use casts invites defects.

Suppose, for example, we have a function, `setPort`, from an external library that takes an integer port number:

```
int setPort(int portNumber);
// Set the current port; return 0 on success and a nonzero value otherwise.
```

Suppose further that we have used the modern `enum class` feature to implement an enumeration, `SysPort`, that identifies well-known ports on our system:

```
enum class SysPort { e_INPUT = 27, e_OUTPUT = 29, e_ERROR = 32, e_CTRL = 6 };
// enumerated port values used to configure our systems
```

Now suppose we want to call the function `f` using one of these enumerated values:

```
void setCurrentPortToCtrl()
{
setPort(SysPort::e_CTRL);  // error: cannot convert SetPort to int
}
```

Unlike the situation for a *classic* `enum`, no implicit conversion occurs from an `enum class` to its underlying integral type, so anyone using this enumeration will be forced to somehow explicitly **cast** the enumerator to some arithmetic type. There are, however, multiple choices for performing this cast:

```
#include <type_traits>  // std::underlying_type

setPort(int(SysPort::e_CTRL));                                        // (1)
setPort((int)SysPort::e_CTRL);                                        // (2)
setPort(static_cast<int>(SysPort::e_CTRL));                           // (3)
setPort(static_cast<std::underlying_type<SysPort>::type>(             // (4)
SysPort::e_CTRL));
setPort(static_cast<int>(                                             // (5)
static_cast<std::underlying_type<SysPort>::type>(SysPort::e_CTRL)));
```

Any of the above casts would work in this case, but consider a future where a platform changed `setPort` to take a `long` and the control port was changed to a value that cannot be represented as an `int`:

```
int setPort(long portNumber);
enum class SysPort : unsigned { e_INPUT = 27, e_OUTPUT = 29, e_ERROR = 32,
e_CTRL = 0x80000000 };
// enumerated port values used to configure our systems
```

Only casting method (4) above will pass the correct value for `e_CTRL` to this new `setPort` implementation. The other variations will all pass a negative number for the port, which would certainly not be the intention of the user writing this code. A classic C-style `enum` would have avoided any manually written cast entirely and the proper value would propagate into `setPort` even as the range of values used for ports changes:

```
struct SysPort  // explicit scoping for a classic, C-style enum
{
enum Enum { e_INPUT = 27, e_OUTPUT = 29, e_ERROR = 32,
e_CTRL = 0x80000000 };

// Note that the underlying type of Enum is implicit and will be
// large enough to represent all of these values.
static_assert(
std::is_same<std::underlying_type<Enum>::type,unsigned>::value, "");
};

void setCurrentPortToCtrl()
{
setPort(SysPort::e_CTRL);  // OK, SysPort::Enum promotes to long.
}
```

When the intended client will depend on the cardinal values of the enumerators during routine use, we can avoid tedious, error-prone, and repetitive casting by instead employing a classic, C-style `enum`, possibly nested within a `struct` to achieve explicit scoping of its enumerators. The subsections that follow highlight specific cases in which classic, C-style, C++03 `enum`s are appropriate.

### 1.22.3.2 Misuse of `enum class` for collections of named constants

When constants are truly independent, we are often encouraged to avoid enumerations altogether, preferring instead individual constants; see “Default Member Initializers” on page 227. On the other hand, when the constants all participate within a coherent theme, the expressiveness achieved using a *classic* `enum` to aggregate those values is compelling.[9]

For example, suppose we want to collect the coefficients for various numerical suffixes representing *thousands*, *millions*, and *billions* using an enumeration:

```
enum class S0 { e_K = 1000, e_M = e_K * e_K, e_G = e_M * e_K };  // (BAD IDEA)
```

A client trying to access one of these enumerated values would need to cast it explicitly:

```
void client0()
{
```

---

[9] Another advantage of an enumerator over an individual constant is that the enumerator is guaranteed to be a **compile-time constant** (see “`constexpr` Variables” on page 228) and a **prvalue** (see “Rvalue References” on page 226), which never needs static storage and cannot have its address taken.

```
int distance = 5 * static_cast<int>(S0::e_K);  // casting is error-prone
// ...
}
```

By instead making the enumeration an explicitly scoped, *classic* `enum` nested within a `struct`, no casting is needed during typical use:

```
struct S1  // scoped
{
enum Enum { e_K = 1000, e_M = e_K * e_K, e_G = e_M * e_K };
// *classic* enum (GOOD IDEA)
};

void client1()
{
int distance = 5 * S1::e_K;  // no casting required during typical use
// ...
}
```

If the intent is that these constants will be specified and used in a purely local context, we might choose to drop the enclosing scope, along with the name of the enumeration itself[10][11]:

```
void client2()
{
enum { e_K = 1000, e_M = e_K * e_K, e_G = e_M * e_K };  // function scoped

double salary = 95 * e_K;
double netWorth = 0.62 * e_M;
double companyRevenue = 47.2 * e_G;
// ...
}
```

### 1.22.3.3 Misuse of `enum class` in association with bit flags

Using `enum class` to implement enumerators that are intended to interact closely with arithmetic types will typically require the definition of arithmetic and bitwise operator overloads between values of the same enumeration and between the enumeration and arithmetic types, leading to yet more code that needs to be written, tested, and maintained. This is often the case for bit flags. Consider, for example, an enumeration used to control a file system:

```
enum class Ctrl { e_READ = 0x1, e_WRITE = 0x2, e_EXEC = 0x4 };  // (BAD IDEA)
// low-level bit flags used to control file system
```

---

[10] See "Local Types as Template Arguments" on page 87.

[11] We sometimes use the lowercase prefix `k_` instead of `e_` to indicate salient **compile-time constants** that are not considered part of an enumerated set, irrespective of whether they are implemented as enumerators:

```
enum { k_NUM_PORTS = 500, k_PAGE_SIZE = 512 };      // compile-time constants
static const double k_PRICING_THRESHOLD = 0.03125; // compile-time constant
```

```
void chmodFile(int fd, int access);
// low-level function used to change privileges on a file
```

We could conceivably write a series of functions to combine the individual flags in a type-safe manner:

```
#include <type_traits>  // std::underlying_type

int flags() { return 0; }
int flags(Ctrl a) { return static_cast<std::underlying_type<a>::type>(a); }
int flags(Ctrl a, Ctrl b) { return flags(a) | flags(b); }
int flags(Ctrl a, Ctrl b, Ctrl c) { return flags(a, b) | flags(c); }

void setRW(int fd)
{
chmodFile(fd, flags(Ctrl::e_READ, Ctrl::e_WRITE));  // (BAD IDEA)
}
```

Alternatively, a *classic*, C-style `enum` nested within a `struct` achieves what's needed:

```
struct Ctrl // scoped
{
enum Enum { e_READ = 0x1, e_WRITE = 0x2, e_EXEC = 0x4 };  // classic enum
// low-level bit flags used to control file system (GOOD IDEA)
};

void chmodFile(int fd, int access);
// low-level function used to change privileges on a file

void setRW(int fd)
{
chmodFile(fd, Ctrl::e_READ | Ctrl::e_WRITE);  // (GOOD IDEA)
}
```

### 1.22.3.4 Misuse of `enum class` in association with iteration

Sometimes the relative values of enumerators are considered important as well. For example, let's again consider enumerating the months of the year:

```
enum class MonthOfYear  // modern, strongly typed enumeration
{
e_JAN, e_FEB, e_MAR,  // winter
e_APR, e_MAY, e_JUN,  // spring
e_JUL, e_AUG, e_SEP,  // summer
e_OCT, e_NOV, e_DEC,  // autumn
};
```

If all we need to do is compare the ordinal values of the enumerators, there's no problem:

```
bool isSummer(MonthOfYear month)
{
return MonthOfYear::e_JUL <= month && month <= MonthOfYear::e_AUG;
}
```

Although the `enum class` features allow for relational and equality operations between like-typed enumerators, no arithmetic operations are supported directly, which becomes problematic when we need to iterate over the enumerated values:

```
void doSomethingWithEachMonth()
{
for (MonthOfYear i =  MonthOfYear::e_JAN;
i <= MonthOfYear::e_DEC;
++i)  // error: no match for ++
{
// ...
}
}
```

To make this code compile, an explicit cast from and to the enumerated type will be required:

```
void doSomethingWithEachMonth()
{
for (MonthOfYear i =  MonthOfYear::e_JAN;
i <= MonthOfYear::e_DEC;
i = static_cast<MonthOfYear>(static_cast<int>(i) + 1)
{
// ...
}
}
```

Alternatively, an auxiliary, helper function could be supplied to allow clients to bump the enumerator:

```
MonthOfYear nextMonth(MonthOfYear value)
{
return static_cast<MonthOfYear>(static_cast<int>(value) + 1);
}

void doSomethingWithEachMonth()
{
for (MonthOfYear i =  MonthOfYear::e_JAN;
i <= MonthOfYear::e_DEC;
i = nextMonth(i)
{
// ...
}
}
```

If, however, the cardinal value of the `MonthOfYear` enumerators is likely to be relevant to clients, an explicitly scoped *classic* `enum` should be considered as a viable alternative:

```
struct MonthOfYear  // explicit scoping for enum
{
enum Enum
{
e_JAN, e_FEB, e_MAR,  // winter
e_APR, e_MAY, e_JUN,  // spring
e_JUL, e_AUG, e_SEP,  // summer
e_OCT, e_NOV, e_DEC,  // autumn
};
};

bool isSummer(MonthOfYear::Enum month)  // must now pass nested Enum type
{
return MonthOfYear::e_JUL <= month && month <= MonthOfYear::e_AUG;
}

void doSomethingWithEachMonth()
{
for (int i =  MonthOfYear::e_JAN;  // iteration variable is now an int
i <= MonthOfYear::e_DEC;
++i)  // OK, convert to underlying type
{
// ... (might require cast back to enumerated type)
}
}
```

Note that such code presumes that the enumerated values will (1) remain in the same order and (2) have contiguous numerical values irrespective of the implementation choice.

#### 1.22.3.5 External use of opaque enumerators

Since `enum class` types have an **underlying type** of `int` by default, clients are always able to (re)declare it, as a **complete type**, without its enumerators. Unless the opaque form of an `enum class`'s definition is exported in a header file separate from the one implementing the publicly accessible full definition, external clients wishing to exploit the opaque version will experience an *attractive nuisance* in that they can provide it locally, along with its **underlying type**, if any.

If the underlying type of the full definition were to subsequently change, any program incorporating the original elided definition locally and also the new, full one from the header would become silently **ill formed, no diagnostic required (IFNDR)**; see "Opaque Enumeration Declarations" on page 166.

### 1.22.4 Annoyances

#### 1.22.4.1 Scoped enumerations do not necessarily add value

When the enumeration is local, say, within the scope of a given function, forcing an additional scope on the enumerators is superfluous. For example, consider a function that returns an integer status `0` on success and a nonzero value otherwise:

```
int f()
{
enum { e_ERROR = -1, e_OK = 0 } result = e_OK;
// ...
if (/* error 1 */) { result = e_ERROR; }
// ...
if (/* error 2 */) { result = e_ERROR; }
// ...
return result;
}
```

Use of `enum class` in this context would require potentially needless qualification — and perhaps even casting — where it might not be warranted:

```
int f()
{
enum class RC { e_ERROR = -1, e_OK = 0 } result = RC::e_OK;
// ...
if (/* error 1 */) { result = RC::e_ERROR; } // undesirable qualification
// ...
if (/* error 2 */) { result = RC::e_ERROR; } // undesirable qualification
// ...
return static_cast<int>(result);  // undesirable explicit cast
}
```

### 1.22.5 See Also

- "Explicit Enumeration Underlying Type" on page 143 — The underlying integral representation enumerator variables and values
- "Opaque Enumeration Declarations" on page 166 — A means of **insulating** individual enumerators from clients

### 1.22.6 Further Reading

TODO

## 1.23 Opaque Enumeration Declarations

placeholder text.........

## 1.24 Aggregate Member Initialization Relaxation

C++14 enables the use of **aggregate initialization** with classes employing Default Member Initializers (see Section 2.4, “Default Member Initializers”).

### 1.24.1 Description

Prior to C++14, classes that made use of Default Member Initializers — i.e., initializers that appear directly within the scope of the class — were not considered **aggregate** types:

```
struct S                // aggregate type in C++14 but not C++11
{
int i;
bool b = false;     // uses default member initializer
};

struct A                // aggregate type in C++11 and C++14
{
int  i;
bool b;             // does not use default member initializer
};
```

Because `A` (but not `S`) is considered an **aggregate** in C++11, instances of `A` can be created via **aggregate initialization** (whereas instances of `S` cannot):

```
A a{100, true};  // OK in both C++11 and C++14
S s{100, true};  // error in C++11; OK in C++14
```

As of C++14, the requirements for a type to be categorized as an **aggregate** are relaxed, allowing classes employing default member initializers to be considered as such; hence both `A` and `S` are considered **aggregates** in C++14 and eligible for **aggregate initialization**:

```
void f()
{
S s0{100, true};        // OK in C++14 but not in C++11
assert(s0.i == 100);    // set via explicit aggregate initialization (above)
assert(s0.b == true);   // set via explicit aggregate initialization (above)

S s1{456};              // OK in C++14 but not in C++11
assert(s1.i == 456);    // set via explicit aggregate initialization (above)
assert(s1.b == false);  // set via default member initializer
}
```

In the code snippet above, the C++14 aggregate `S` is initialized in two ways: `s0` is created using aggregate initialization for both data members; `s` is created using aggregate initialization for only the first data member (and the second is set via its default member initializer).

## 1.24.2 Use Cases

### 1.24.2.1 Configuration `struct`s

**Aggregates** in conjunction with Default Member Initializers can be used to provide concise customizable configuration `struct`s, packaged with typical default values. As an example, consider a configuration `struct` for an HTTP request handler:

```
struct HTTPRequestHandlerConfig
{
int maxQueuedRequests = 1024;
int timeout           = 60;
int minThreads        = 4;
int maxThreads        = 8;
};
```

**Aggregate initialization** can be used when creating objects of type `HTTPRequestHandlerConfig` (above) to override one or more of the defaults in definition order[1]:

```
HTTPRequestHandlerConfig getRequestHandlerConfig(bool inLowMemoryEnvironment)
{
if (inLowMemoryEnvironment)
{
return HTTPRequestHandlerConfig{128};
// timeout, minThreads, and maxThreads have their default value.
}
else
{
return HTTPRequestHandlerConfig{2048, 120};
// minThreads, and maxThreads have their default value.
}
}

// ...
```

## 1.24.3 Potential Pitfalls

None so far

---

[1] In C++20, the Designated Initializers feature adds flexibility (e.g., for configuration `struct`s, such as `HTTPRequestHandlerConfig`) by enabling explicit specification of the names of the data members:

```
HTTPRequestHandlerConfig lowTimeout{.timeout = 15};
// maxQueuedRequests, minThreads, and maxThreads have their default value.

HTTPRequestHandlerConfig highPerformance{.timeout = 120, .maxThreads = 16};
// maxQueuedRequests and minThreads have their default value.
```

### 1.24.4 Annoyances

#### 1.24.4.1 Syntactical ambiguity in the presence of brace elision

During the initialization of multilevel **aggregates**, braces around the initialization of a nested aggregate can be omitted (**brace elision**):

```
struct S
{
int arr[3];
};

S s0{{0, 1, 2}};  // OK, nested arr initialized explicitly
S s1{0, 1, 2};    // OK, brace elision for nested arr
```

The possibility of **brace elision** creates an interesting syntactical ambiguity when used alongside **aggregates** with Default Member Initializers. Consider a `struct X` containing three data members, one of which has a default value:

```
struct X
{
int a;
int b;
int c = 0;
};
```

Now, consider various ways in which an array of elements of type `X` can be initialized:

```
X xs0[] = {{0, 1}, {2, 3}, {4, 5}};
// OK, clearly 3 elements having the respective values:
// {0, 1, 0}, {2, 3, 0}, {4, 5, 0}

X xs1[] = {{0, 1, 2}, {3, 4, 5}};
// OK, clearly 2 elements with values:
// {0, 1, 2}, {3, 4, 5}

X xs2[] = {0, 1, 2, 3, 4, 5};
// ...?
```

Upon seeing the definition of `xs2`, a programmer not versed in the details of the C++ Language Standard might be unsure as to whether the initializer of `xs2` is three elements (like `xs0`) or two elements (like `xs1`). The Standard is, however, clear that the compiler will interpret `xs2` the same as `xs1`, and, thus, the default values of `X::c` for the two array elements will be replaced with `2` and `5`, respectively.

### 1.24.5 See Also

- Section 2.4, “Default Member Initializers” — Conditionally safe C++11 feature that allows developers to provide a default initializer for a data member directly in the definition of a class

### 1.24.6 Further Reading

None so far

## 1.25 Digit Separators

A digit separator is a single-character token (`'`) that can appear as part of a numeric literal without altering its value.

### 1.25.1 Description

A digit separator — i.e., an instance of the single-quote character (`'`) — may be placed anywhere within a numeric literal to visually separate its digits without affecting its value:

```
int          i = -12'345;                    // same as -12345
unsigned int u = 1'000'000u;                 // same as 1000000u
long         j = 5'0'0'0'0'0L;               // same as 500000L
long long    k = 9'223'372'036'854'775'807;  // same as 9223372036854775807
float        f = 10'00.42'45f;               // same as 1000.4245f
double       d = 3.1415926'53589793;         // same as 3.141592653589793
long double  e = 3.1415926'53589793'23846;   // same as 3.14159265358979323846
int        hex = 0x8C25'00F9;                // same as 0x8C2500F9
int        oct = 044'73'26;                  // same as 0447326
int        bin = 0b1001'0110'1010'0111;      // same as 0b1001011000110001
```

Multiple digit separators within a single literal are allowed, but they cannot be contiguous, nor can they appear either before or after the *numeric* part (i.e., digit sequence) of the literal[1]:

```
int e0 = 10''00;  // error: consecutive digit separators
int e1 = -'1000;  // error: before numeric part
int e2 = 1000'u;  // error: after numeric part
int e3 = 0x'abc;  // error: before numeric part
int e4 = 0'xdef;  // error: way before numeric part
int e5 = 0'89;    // error: non-octal digits
int e6 = 0'67;    // OK, valid octal literal
```

As a side note, remember that on some platforms an integer literal that is too large to fit in a `long long int` but that does fit in an `unsigned long long int` might generate a warning[2]:

```
unsigned long long big1 = 9'223'372'036'854'775'808;  // 2^63
// warning: integer constant is so large that it is an
// unsigned long long big1 = 9'223'372'036'854'775'808;
//                           ^~~~~~~~~~~~~~~~~~~~~~~~~
```

Such warnings can typically be suppressed by adding a `ull` suffix to the literal:

```
unsigned long long big2 = 9'223'372'036'854'775'808ull;  // OK
```

Warnings like the one above, however, are not typical when the implied precision of a floating-point literal exceeds what can be represented:

---

[1] Although the leading `0x` and `0b` prefixes for hexadecimal and binary literals, respectively, are not considered part of the *numeric* part of the lateral, a leading `0` in an octal literal is.

[2] Tested on GCC 7.4.0.

```
float reallyPrecise = 3.141'592'653'589'793'238'462'643'383'279'502'884;  // OK
// Everything after 3.141'592'6 is typically ignored silently.
```

For more information, see *Appendix: Silent Loss of Precision in Floating-Point Literals* on page 173.

## 1.25.2 Use Cases

### 1.25.2.1 Grouping digits together in large constants

When embedding large constants in source code, consistently placing digit separators (e.g., every thousand) might improve readability, as illustrated in Table 1–1.

**Table 1–1: Use of digit separators to improve readability**

| Without Digit Separator | With Digit Separators |
|---|---|
| `10000` | `10'000` |
| `100000` | `100'000` |
| `1000000` | `1'000'000` |
| `1000000000` | `1'000'000'000` |
| `18446744073709551615ull` | `18'446'744'073'709'551'615ull` |
| `1000000.123456` | `1'000'000.123'456` |
| `3.141592653589793238462l` | `3.141'592'653'589'793'238'462l` |

Use of digit separators is especially useful with binary literals, as shown in Table 1–2.

**Table 1–2: Use of digit separators in binary data**

| Without Digit Separator | With Digit Separators |
|---|---|
| `0b1100110011001100` | `0b1100'1100'1100'1100` |
| `0b0110011101011011` | `0b0110'0111'0101'1011` |
| `0b1100110010101010` | `0b11001100'10101010` |

## 1.25.3 Potential Pitfalls

None so far

## 1.25.4 See Also

- Section 1.2, “Binary Literals” — Safe C++14 feature representing a binary constant for which digit separators are commonly used to group bits in octets (**bytes**) or quartets (**nibbles**)

### 1.25.5 Further Reading

- William Kahan. "Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic," **kahan97**
- *IEEE Standard for Floating-PointArithmetic*, **ieee19**

### 1.25.6 Appendix: Silent Loss of Precision in Floating-Point Literals

Just because we can keep track of precision in floating-point literals doesn't mean that the compiler can. As an aside, it is worth pointing out that the binary representation of floating-point types is not mandated by the Standard, nor are the precise minimums on the ranges and precisions they must support. Although the C++ Standard says little that is normative, the macros in `<cfloat>` are defined by reference to the C Standard.[3]

There are, however, normal and customary minimums that one can typically rely upon in practice. On conforming compilers that employ the IEEE 754 floating-point standard representation[4] (as most do), a `float` can typically represent up to 7 significant decimal digits accurately, while a `double` typically nearly 15 decimal digits of precision. For any given program, `long double` is required to hold whatever a `double` can hold, but is typically larger (e.g., 10, 12, or 16 bytes) and typically adds at least 5 decimal digits of precision (i.e., supports a total of at last 20 decimal digits). A table summarizing typical precisions for various IEEE-conforming floating-point types is presented for convenient reference in Table 1–3. The actual bounds on a given platform can be found using the standard `std::numeric_limits` class template found in `<limits>`.

---

[3] **PRODUCTION: WAITING FOR THESE REFERENCES TO DOUBLECHECK; CONSIDER THESE UNCONFIRMED. iso20**, sections [basic.fundamental] Fundamental types (6.8.1p12); [numeric.limits.members] `numeric_limits` members 17.3.5.1; [cfloat.syn] Header `<cfloat>` synopsis (17.3.7p1); **iso18b**, section 5.2.4.2.2 Characteristics of floating types `<float.h>`

[4] **ieee19**

**Table 1–3: Available precisions for various IEEE-754 floating-point types**

| Name | Common Name | Significant Bits[a] | Decimal Bits | Exponent Bits | Dynamic Range |
|---|---|---|---|---|---|
| binary16 | Half precision | 11 | 3.31 | 5 | $\sim$ 6.50e5 |
| binary32 | Single precision | 24 | 7.22 | 8 | $\sim$ 3.4e38 |
| binary64 | Double precision | 53 | 15.95 | 11 | $\sim$ 1.e308 |
| binary80 | Extended precision | 69 | 20.77 | 11 | $\sim 10^{308}$ |
| binary128 | Quadruple precision | 113 | 34.02 | 15 | $\sim 10^{4932}$ |

[a] Note that the most significant bit of the **mantissa** is always a `1` and, hence, is not stored explicitly, leaving 1 additional bit to represent the sign of the overall floating-point value (the sign of the exponent is encoded using **excess-n** notation).

Determining the minimum number of decimal digits needed to accurately approximate a transcendental value, such as $\pi$, for a given type on a given platform can be tricky (requiring some binary-search-like detective work), which is likely why overshooting the precision without warning is the default on most platforms. One way to establish that *all* of the decimal digits in a given floating-point literal are relevant for a given floating-point type is to compare that literal and a similar one with its least significant decimal digit removed[5]:

```
static_assert(3.1415926535f != 3.141592653f, "too precise for float");
// This assert will fire on a typical platform.

static_assert(3.141592653f != 3.14159265f, "too precise for float");
// This assert too will fire on a typical platform.

static_assert(3.14159265f != 3.1415926f, "too precise for float");
// This assert will NOT fire on a typical platform.

static_assert(3.1415926f != 3.141592f, "too precise for float");
// This assert too will NOT fire on a typical platform.
```

If the values are *not* the same, then that floating-point type *can* make use of the precision suggested by the original literal; if they *are* the same, however, then it is likely that the available precision has been exceeded. Iterative use of this technique by developers can help them to empirically narrow down the maximal number of decimal digits a particular platform will support for a particular floating-point type and value.

One final useful tidbit pertains to the safe (lossless) conversion between binary and decimal floating-point representations; note that “Single” (below) corresponds to a single-precision IEEE-754-conforming (32-bit) `float`[6]:

---

[5] Note that affixing the `f` (*literal suffix*) to a floating-point literal is equivalent to applying a `static_cast<float>` to the (unsuffixed) literal:

```
static_assert(3.14'159'265'358f == static_cast<float>(3.14'159'265'358));
```

[6] **kahan97**, section “Representable Numbers,” p. 4

> If a decimal string with at most 6 sig. dec. is converted to Single and then converted back to the same number of sig. dec., then the final string should match the original. Also, ...
>
> If a Single Precision floating-point number is converted to a decimal string with at least 9 sig. dec. and then converted back to Single, then the final number must match the original.

The ranges corresponding to 6–9 for a single-precision (32-bit) `float` (described above), when applied to a double-precision (64-bit) `double` and a quad-precision (128-bit) `long long`, are 15–17 and 33–36, respectively.

## 1.26 Variable Templates

Variable templates extend traditional template syntax to define, in namespace or class (but not function) scope, a family of like-named variables that can subsequently be instantiated explicitly.

### 1.26.1 Description

By beginning a variable declaration with the familiar **template-head** syntax — e.g., `template <typename T>` — we can create a *variable template*, which defines a family of variables having the same name (e.g., `typeid`):

```
template <typename> int typeId;  // template variable defined at file scope
```

Like any other kind of template, a variable template can be instantiated (explicitly) by providing an appropriate number (one or more) of type or non-type arguments:

```
void f1()
{
typeId<bool> = -1;    // typeId<bool> is an int
typeId<char> = 1000;  // typeId<char> is an int
typeId<void> = -666;  // typeId<void> is an int

assert(typeId<bool> ==   -1);
assert(typeId<char> == 1000);
assert(typeId<void> == -666);
}
```

In the example above, the type of each instantiated variable — i.e., `typeId<bool>` and `typeId<char>` — is `int`. Such need not be the case[1]:

```
template <typename T> const T pi(3.1415926535897932385);  // distinct types
```

In the example above, the type of the instantiated non-`const` variable is that of its (type) argument, and its (mutable) value is initialized to the best approximation of $\pi$ offered by that type:

```
void f2()
{
bool        pi_as_bool        = 1;                      // ( 1 bit)
int         pi_as_int         = 3;                      // (32 bits)
float       pi_as_float       = 3.1415927;              // (32 bits)
double      pi_as_double      = 3.141592653589793;      // (64 bits)
long double pi_as_long_double = 3.1415926535897932385;  // (80 bits)

assert(pi<bool>        == pi_as_bool);
assert(pi<int>         == pi_as_int);
assert(pi<float>       == pi_as_float);
```

---

[1] Use of `constexpr` variables would allow the instantiated variables to be usable as a constant in a compile-time context (see *Use Cases: Parameterized constants* on page 178).

```
assert(pi<double>      == pi_as_double);
assert(pi<long double> == pi_as_long_double);
}
```

For examples involving immutable variable templates, see *Use Cases: Parameterized constants* on page 178.

Variable templates, like **C-style functions**, may be declared at namespace-scope or as `static` members of a `class`, `struct`, or `union` but are not permitted as non`static` members nor at all in function scope:

```
template <typename T> T vt1;             // OK (external linkage)
template <typename T> static T vt2;      // OK (internal linkage)

namespace N
{
template <typename T> T vt3;           // OK (external linkage)
template <typename T> T vt4;           // OK (internal linkage)
}

struct S
{
template <typename T> T vt5;         // error: not static
template <typename T> static T vt6;  // OK (external linkage)
};

void f3()  // Variable templates cannot be defined in functions.
{
template <typename T> T vt7;         // compile-time error
template <typename T> static T vt8;  // compile-time error

vt1<bool> = true;                    // OK (to use them)
}
```

Like other templates, variable templates may be defined with multiple parameters consisting of arbitrary combinations of type and non-type parameters (including a **parameter pack**):

```
namespace N
{
template <typename V, int I, int J> V factor;  // namespace scope
}
```

Variable templates can even be defined recursively (but see *Potential Pitfalls: Recursive variable template initializations require* `const` *or* `constexpr` on page 180):

```
template <int N>
const int sum = N + sum<N - 1>;    // recursive general template

template <> const int sum<0> = 0;  // base-case specialization

void f()
{
```

```
std::cout << sum<4> << '\n';  // prints 10
std::cout << sum<5> << '\n';  // prints 15
std::cout << sum<6> << '\n';  // prints 21
}
```

Note that variable templates do not enable any novel patterns; anything that can be achieved using them could also have been achieved in C++11 along with some additional boilerplate. The initial `typeId` example could have instead been implemented using a `struct`:

```
template <typename> struct TypeId { static int value; };
```

And used with just a bit more syntax:

```
void f1b()
{
TypeId<bool>::value = -1;    // TypeId<bool>::value is an int.
TypeId<char>::value = 1000;  // TypeId<char>::value is an int.
TypeId<void>::value = -666;  // TypeId<void>::value is an int.

assert(TypeId<bool>.value ==   -1);
assert(TypeId<char>.value == 1000);
assert(TypeId<void>.value == -666);
}
```

## 1.26.2 Use Cases

### 1.26.2.1 Parameterized constants

A common effective use of variable templates is in the definition of type-parameterized constants. As discussed in *Description* on page 176, the mathematical constant $\pi$ serves as our example. Here we want to initialize the constant as part of the variable template (the literal chosen is the shortest decimal string to do so accurately for an 80-bit `long double`)[2]:

```
template <typename T>
constexpr T pi(3.14159265358979323851);
// smallest digit sequence to accurately represent pi as a long double
```

Notice that we have elected to use `constexpr` variables (from C++11) in place of a classic `const` as a stronger guarantee that the provided initializer is a compile-time constant and that `pi` itself will be usable as part of a constant expression.

With the definition above, we can provide a `toRadians` function template that performs at maximum runtime efficiency by avoiding needless type conversions during the computation:

```
template <typename T>
constexpr T toRadians(T degrees)
{
return degrees * (pi<T> / T(180));
}
```

---

[2] For portability, a floating-point literal value of $\pi$ that provides sufficient precision for the longest `long double` on any relevant platform (e.g., 128 bits or 34 decimal digits: `3.141'592'653'589'793'238'462'643'383'279'503`) should be used; see Section 1.25, “Digit Separators.”

### 1.26.2.2 Reducing verbosity of type traits

A **type trait** is an empty type carrying compile-time information about one or more aspects of another type. The way in which type traits have been specified historically has been to define a class template having the trait name and a public `static` (or `enum`) data member, that is conventionally called `value`, which is initialized in the primary template to `false`. Then, for each type that wants to advertise that it has this trait, the header defining the trait is included and the trait is specialized for that type, initializing `value` to `true`. We can achieve precisely this same usage pattern replacing a trait `struct` with a variable template whose name represents the type trait and whose type of variable itself is always `bool`. Preferring variable templates in this use case decreases the amount of **boilerplate code** — both at the point of definition and at the call site.[3]

Consider, for example, a boolean trait designating whether a particular type `T` can be serialized to JSON:

```
// isSerializableToJson.h

template <typename T>
constexpr bool isSerializableToJson = false;
```

The header above contains the general variable template trait that, by default, concludes that a given type is not serializable to JSON. Next we consider the streaming utility itself:

```
// serializeToJson.h
#include <isSerializableToJson.h>  // general trait variable template

template <typename T>
JsonObject serializeToJson(const T& object)  // serialization function template
{
static_assert(isSerializableToJson<T>,
"T must support serialization to JSON.");

// ...
}
```

Notice that we have used the C++11 `static_assert` feature to ensure that any type used to instantiate this function will have specialized (see the next code snippet) the general variable template associated with the specific type to be `true`.

Now imagine that we have a type, `CompanyData`, that we would like to advertise at compile time as being serializable to JSON. Like other templates, variable templates can be specialized explicitly:

---

[3] As of C++17, the Standard Library provides a more convenient way of inspecting the result of a type trait, by introducing variable templates named the same way as the corresponding traits but with an additional _v suffix:

```
// C++11/14
std::is_default_constructible<T>::value

// C++17
std::is_default_constructible_v<T>
```

```
// companyData.h
#include <isSerializableToJson.h>  // general trait variable template

struct CompanyData { /* ... */ };  // type to be JSON serialized

template <>
constexpr bool isSerializableToJson<CompanyData> = true;
// Let anyone who needs to know that this type is JSON serializable.
```

Finally, our `client` function incorporates all of the above and attempts to serialize both a `CompanyData` object and an `std::map<int, char>>`:

```
// client.h
#include <isSerializableToJson.h>  // general trait template
#include <companyData.h>           // JSON serializable type
#include <serializeToJson.h>       // serialization function
#include <map>                     // std::map (not JSON serializable)

void client()
{
auto jsonObj0 = serializeToJson<CompanyData>();         // OK
auto jsonObj1 = serializeToJson<std::map<int, char>>(); // compile-time error
}
```

In the `client()` function above, `CompanyData` works fine, but, because the variable template `isSerializableToJson` was never specialized to be `true` for type `std::map<int, char>>`, the client header will — as desired — fail to compile.

## 1.26.3 Potential Pitfalls

### 1.26.3.1 Recursive variable template initializations require `const` or `constexpr`

When discussing the intricacies of the C++ language with your peers, consider quizzing them on why the example below, having no undefined behavior, might produce different results with popular compilers[4]:

```
#include <iostream>

template <int N>
int fib = fib<N - 1> + fib<N - 2>;

template <> int fib<2> = 1;
template <> int fib<1> = 1;

int main()
{
std::cout << fib<4> << '\n';  // 3 expected
std::cout << fib<5> << '\n';  // 5 expected
```

---

[4] For example, GCC version 4.7.0 (2017) produces the expected results whereas Clang version 10.x (2020) produces 1, 3, and 4, respectively.

```
std::cout << fib<6> << '\n';  // 8 expected

return 0;
}
```

The didactic value in answering this question dwarfs any potential practical value that recursive template variable instantiation can offer. First, consider that this same issue could, in theory, have occurred in C++03 using nested `static` members of a `struct`:

```
#include <iostream>

template <int N> struct Fib
{
static int value;                             // BAD IDEA: not const
};

template <> struct Fib<2> { static int value; };  // BAD IDEA: not const
template <> struct Fib<1> { static int value; };  // BAD IDEA: not const

template <int N> int Fib<N>::value = Fib<N - 1>::value + Fib<N - 2>::value;
int Fib<2>::value = 1;
int Fib<1>::value = 1;

int main()
{
std::cout << Fib<4>::value << '\n';  // 3 expected
std::cout << Fib<5>::value << '\n';  // 5 expected
std::cout << Fib<6>::value << '\n';  // 8 expected

return 0;
};
```

The problem did not manifest, however, because the simpler solution of using `enum`s (below) obviated separate initialization of the local `static` and didn't admit the possibility of failing to make the initializer a compile-time constant:

```
#include <iostream>

template <int N> struct Fib
{
enum { value = Fib<N - 1>::value + Fib<N - 2>::value };  // OK - const
};

template <> struct Fib<2> { enum { value = 1 }; };           // OK - const
template <> struct Fib<1> { enum { value = 1 }; };           // OK - const

int main()
{
std::cout << Fib<4>::value << '\n';  // 3 guaranteed
std::cout << Fib<5>::value << '\n';  // 5 guaranteed
std::cout << Fib<6>::value << '\n';  // 8 guaranteed
return 0;
};
```

It was not until C++14 that the variable templates feature readily exposed this latent pitfall involving recursive initialization of non-`const` variables. The root cause of the instability is that the relative order of the initialization of the (recursively generated) variable instantiations is not guaranteed because they are not defined explicitly *within the same translation unit*. The magic sauce that makes everything work is the C++ language requirement that any variable that is declared `const` and initialized with a compile-time constant is itself to be treated as a compile-time constant within the translation unit. This compile-time-constant propagation requirement imposes the needed ordering to ensure that the expected results are portable to all conforming compilers:

```
#include <iostream>

template <int N>
const int fib = fib<N - 1> + fib<N - 2>;  // OK - compile-time const.

template <> const int fib<2> = 1;         // OK - compile-time const.
template <> const int fib<1> = 1;         // OK - compile-time const.

int main()
{
std::cout << fib<4> << '\n';  // guaranteed to print out 3
std::cout << fib<5> << '\n';  // guaranteed to print out 5
std::cout << fib<6> << '\n';  // guaranteed to print out 8

return 0;
}
```

Note that replacing each of the three `const` keywords with `constexpr` in the example above also achieves the desired goal and does not consume memory in the **static data space**.

## 1.26.4 Annoyances

### 1.26.4.1 Variable templates do not support template template parameters

While a class or function template can accept a **template template class parameter**, no equivalent construct is available for variable templates[5]:

```
template <typename T> T vt(5);

template <template <typename> class>
struct S { };

S<vt> s1;  // compile-time error
```

---

[5] Pusz has proposed for C++23 a way to increase consistency between variable templates and class templates when used as template template parameters; see **pusz20**.

Providing a wrapper `struct` around a variable template might therefore be necessary in case the variable template needs to be passed to an interface accepting a **template template parameter**:

```
template <typename T>
struct Vt { static constexpr T value = vt<T>; }

S<Vt> s2;  // OK
```

### 1.26.5 See Also

- Section 2.5, "`constexpr` Variables" — Conditionally safe C++11 feature providing an alternative to `const` template variables that can reduce unnecessary consumption of the **static data space**

### 1.26.6 Further Reading

None so far

## 1.27 Defaulted Special Member Functions

Use of `= default` in a **special member function**'s declaration instructs the compiler to attempt generating the function automatically.

### 1.27.1 Description

Intrinsic to the design of C++ classes is the understanding that the compiler will attempt to generate certain member functions pertaining to *creating*, *copying*, *destroying*, and now *moving* (see Section 2.3, "Rvalue References") an object unless developers intercede by implementing some or all of these functions themselves. Determining which of the **special member functions** will continue to be generated and which will be suppressed in the presence of other **user-provided special member functions** requires remembering the same rules the compiler uses.

#### 1.27.1.1 Declaring a special member function explicitly

The rules specifying what happens in the presence of one or more user-provided special member functions are inherently complex and not necessarily intuitive; in fact, some are already deprecated.[1] Here, we will briefly illustrate a few common cases and then refer you to Howard Hinnant's now famous table (see page 195 of *Appendix: Implicit Generation of Special Member Functions*) to demystify what's going on under the hood.

**Example 1: Providing just the default constructor** Consider a `struct` with a user-provided default constructor:

```
struct S1
{
S1();  // user-provided default constructor
};
```

A user-provided default constructor has no effect on other special member functions. Providing any other constructor, however, will suppress automatic declaration[2] of the default constructor. We can, however, use `= default` to restore the constructor as a **trivial operation**; see *Use Cases: Restoring the generation of a special member function suppressed by another* on page 187.

**Example 2: Providing just a copy constructor** Now, consider a `struct` with a user-provided copy constructor:

```
struct S2
```

---

[1] Even in the presence of a user-provided destructor, both the copy constructor and copy-assignment operator have historically been generated implicitly. Relying on such generated behavior is not recommended because it is unlikely that a class requiring a user-provided destructor will function correctly without corresponding user-provided copy operations. As of C++11, reliance on such dubious implicitly generated behavior is deprecated.

[2] A nondeclared function is nonexistent, which means that it will *not* participate in overload resolution at all. In contrast, a **deleted function** participates in overload resolution and, if selected, results in a compilation failure; see Section 1.5, "Deleted Functions."

```
{
S2(const S2&);  // user-provided copy constructor
};
```

A user-provided copy constructor (1) suppresses the declaration of the default constructor and both move operations and (2) allows implicit generation of both the copy-assignment operator and the destructor. Similarly, providing just the copy-assignment operator would allow the compiler to implicitly generate both the copy constructor and the destructor, but, in this case, it would also generate the default constructor (see `S6` in the `example.h` code snippet on page 186). Note that — in either of these cases — relying on the compiler's implicitly generated copy operation is deprecated.

**Example 3: Providing just the destructor** Finally, consider a `struct` with a user-provided destructor:

```
struct S3
{
~S3();  // user-provided destructor
};
```

A user-provided destructor suppresses the declaration of move operations but still allows copy operations to be generated. Again, relying on either of these (implicitly) compiler-generated copy operations is deprecated.

**Example 4: Providing more than one special member function** When more than one special member function is declared explicitly, the *union* of their respective declaration suppressions and the *intersection* of their respective implicit generations pertain — e.g., if just the default constructor and destructor are provided (`S1` + `S3` in Examples 1 and 3), then the declarations of both move operations are suppressed and both copy operations are generated implicitly.

### 1.27.1.2 Defaulting the first declaration of a special member function explicitly

Using the `= default` syntax with the first declaration of a special member function instructs the compiler to synthesize such a function automatically (if possible) without treating it as being user provided.[3]

For example, consider struct `S4` (in the code snippet below) in which we have chosen to make explicit that the copy operations are to be autogenerated by the compiler; note, in particular, that implicit declaration and generation of each of the other special member functions is left unaffected.

```
struct S4
{
S4(const S4&) = default;              // copy constructor
S4& operator=(const S4&) = default;  // copy-assignment operator
```

---

[3] The compiler-generated version for a special member function is required to call the corresponding special member functions on (1) every base class in base-class-declaration order and then (2) every data member of the encapsulating type in declaration order (regardless of any access specifiers). Note that the destructor calls will be in exactly the opposite order of the other special-member-function calls.

```
// has no effect on other other four special member functions, i.e.,
// implicitly generates the default constructor, the destructor,
// the move constructor, and the move-assignment operator
};
```

A defaulted declaration may appear with any **access specifier** — i.e., `private`, `protected`, or `public` — and, hence, access to that generated function will be regulated accordingly:

```
struct S5
{
private:
S5(const S5&) = default;             // private copy constructor
S5& operator=(const S1&) = default;  // private copy-assignment operator

protected:
~S5() = default;                     // protected destructor

public:
S5() = default;                     // public default constructor
};
```

In the example above, copy operations exist for use by *member* and *friend* functions only. Declaring the destructor `protected` or `private` limits which functions can create automatic variables of the specified type to those functions with the appropriately privileged access to the class. Declaring the default constructor `public` is necessary to avoid its declaration's being suppressed by another constructor (e.g., the private copy constructor in the code snippet above) or *any* move operation.

In short, use of `= default` on first declaration denotes that a special member function is intended to be generated by the compiler — irrespective of any user-provided declarations; in conjunction with `= delete` (see Section 1.5, “Deleted Functions”), use of `= default` affords the fine-grained control over which special member functions are to be generated and/or made publicly available.

### 1.27.1.3 Defaulting the implementation of a user-provided special member function

The `= default` syntax can also be used after the first declaration, but with a distinctly different meaning: The compiler will treat the first declaration as a **user-provided special member function** and thus will suppress the generation of other **special member functions** accordingly.

```
// example.h

struct S6
{
S6& operator=(const S6&);  // user-provided copy-assignment operator

// suppresses the declaration of both move operations
// implicitly generates the default and copy constructors, and destructor
};

inline S6& S6::operator=(const S6&) = default;
// Explicitly request the compiler to generate the default implementation
// for this copy-assignment operator. This request might fail (e.g., if S6
// were to contain a non-copyable-assignable data member).
```

Alternatively, an explicitly defaulted non-inline implementation of this copy-assignment operator may appear in a separate (`.cpp`) file; see *Use Cases: Physically decoupling the interface from the implementation* on page 192.

## 1.27.2 Use Cases

### 1.27.2.1 Restoring the generation of a special member function suppressed by another

Incorporating `= default` in the declaration of a special member function instructs the compiler to generate its definition regardless of any other user-provided special member functions. As an example, consider a **value-semantic** `SecureToken` class that wraps a standard string (`std::string`) and an arbitrary-precision-integer (`BigInt`) token code that satisfy certain invariants:

```
class SecureToken
{
std::string d_value;  // The default-constructed value is the empty string.
BigInt d_code;        // The default-constructed value is the integer zero.

public:
// All six special member functions are (implicitly) defaulted.

void setValue(const char* value);
const char* value() const;
BigInt code() const;
};
```

By default, a secure token’s `value` will be the empty-string value and the token’s `code` will be the numerical value of zero (because those are, respectively, the **default initialized** values of the two data members, `d_value` and `d_tokenCode`):

```
void f()
{
SecureToken token;                        // default constructed        (1)
assert(token.value() == std::string());  // default value: empty string (2)
assert(token.code() == BigInt());        // default value: zero         (3)
}
```

Now suppose that we get a request to add a **value constructor** that creates and initializes a `SecureToken` from a specified token string:

```
class SecureToken
{
```

```
std::string d_value;  // The default-constructed value is the empty string.
BigInt d_tokenCode;   // The default-constructed value is the integer zero.

public:
SecureToken(const char* value);  // newly added value constructor

// suppresses the declaration of just the default constructor --- i.e.,
// implicitly generates all of the other five special member functions

void setValue(const char* value);
const char* value() const;
const BigInt& code() const;
};
```

Attempting to compile function `f` (from page 187) would now fail on the first line, where it attempts to default-construct the token. Using the `= default` feature, however, we can reinstate the default constructor to work trivially, just as it did before:

```
class SecureToken
{
std::string d_value;  // The default-constructed value is the empty string.
BigInt d_code;        // The default-constructed value is the integer zero.

public:
SecureToken() = default;         // newly defaulted default constructor
SecureToken(const char *value);  // newly added value constructor

// implicitly generates all of the other five special member functions

void setValue(const char *value);
const char *value() const;
const BigInt& code() const;
};
```

### 1.27.2.2 Making class APIs explicit at no runtime cost

In the early days of C++, coding standards sometimes required that each special member function be declared explicitly so that it could be documented or even just to know that it hadn't been forgotten:

```
class C1
{
// ...

public:
C1();
// Create an empty object.

C1(const C1& rhs);
// Create an object having the same value as the specified rhs object.
~C1();
// Destroy this object.

C1& operator=(const C1& rhs);
// Assign to this object the value of the specified rhs object.
};
```

Over time, explicitly writing out what the compiler could do more reliably itself became more clearly an inefficient use of developer time. What's more, even if the function definition was empty, implementing it explicitly often had performance implications over allowing implementations to provide a **trivial** default. Hence, such standards tended to evolve toward conventionally commenting out (e.g., using `//!`) the declarations of a function having an empty body rather than providing it explicitly:

```
class C2
a{
// ...

public:
//! C2();
// Create an empty object.

//! C2(const C2& rhs);
// Create an object having the same value as the specified rhs object.

//! ~C2();
// Destroy this object.

//! C2& operator=(const C2& rhs);
// Assign to this object the value of the specified rhs object.
};
```

Note, however, that the compiler does not check the commented code, which is easily susceptible to copy-paste and other errors. By uncommenting the code and defaulting it explicitly in class scope, we regain the compiler's syntactic checking of the function signatures without incurring the cost of turning what would have been **trivial** (i.e., compiler-generated) functions into equivalent non-**trivial** ones:

```
class C3
{
// ...

public:
C3() = default;
// Create an empty object.

C3(const C3& rhs) = default;
// Create an object having the same value as the specified rhs object.
~C3() = default;
// Destroy this object.

C3& operator=(const C3& rhs) = default;
// Assign to this object the value of the specified rhs object.
};
```

### 1.27.2.3 Preserving trivial copyability

In some situations, a particular type *must* be usable with `std::memcpy` (e.g., runtime performance, serialization to binary, or interoperability with C code). Only **trivially copyable** types are safe to use with `std::memcpy`; use with any other types results in **undefined behavior**. A type `T` is **trivially copyable** if it exposes a **trivial copy constructor**:

1. the copy constructor for `T` is not user provided
2. the type `T` itself has no `virtual` member functions or `virtual` base classes
3. any member or base class of `T` is itself **trivially copyable** (recursively).

As an example, the `EntityHandle` class (in the code snippet below) represents an integer handle (to an entity of opaque type) that must be usable with `std::memcpy` for the purpose of efficient serialization (the capacity of the encapsulated fundamental integral type is subject to change)[4]:

```
class EntityHandle
{
short int d_id;  // Note: Implementation size may increase over time.

public:
EntityHandle(int id);  // value constructor

// suppresses the declaration of just the default constructor --- i.e.,
// implicitly generates all of the other five special member functions

// ...
}
```

The presence of any other constructor, except a *move constructor*, never affects implicit generation of a copy constructor, and `short int` (like all *enumerated*, *pointer*, and other *fundamental* types) is a **trivial type**, thus establishing the *triviality* of copying an `EntityHandle`. Now imagine that, to monitor the places in the codebase where *temporary* entity handles are exchanged (with the goal of ultimately optimizing those), a user-provided *move constructor* is added[5]:

---

[4] Objects of this type are sometimes said to hold "dumb data"; see **lakos20**, section 3.5.5, pp. 629–633.

[5] Note that a move constructor will be preferred over a copy constructor when the type category of the argument is an *xvalue* (i.e., expiring value) or *prvalue* (i.e., pure rvalue), which are the value categories to which a temporary can pertain. See Section 2.3, "Rvalue References," for more information.

```
class EntityHandle
{
short int d_id;  // Note: Implementation size may increase over time.

public:
EntityHandle(int id);              // value constructor
EntityHandle(EntityHandle&& rhs);  // user-provided move constructor

// implicitly generates only the destructor
// suppresses synthesis of all four other special member functions

// ...
};
```

As illustrated by Table 1–1 on page 195, the presence of a user-provided *move constructor* suppressed the automatic generation of a copy constructor along with the destructor and both the copy- and move-assignment operators, thereby rendering the `EntityHandle` unusable. Replacing these four previously generated functions with seemingly equivalent user-provided ones might appear to work as intended:

```
class EntityHandle
{
short int d_id;  // Note: Implementation size may increase over time.

public:
EntityHandle(int id);  // value constructor

EntityHandle(const EntityHandle& rhs);  // user-provided copy constructor
EntityHandle(EntityHandle&& rhs);       // user-provided move constructor

EntityHandle& operator=(const EntityHandle& rhs);
// user-provided copy-assignment operator

EntityHandle& operator=(EntityHandle&& rhs);
// user-provided move-assignment operator

// implicitly generates only the destructor
// suppresses synthesis of the default constructor

// ...
};
```

The user-provided nature of the copy constructor, however, renders the `EntityHandle` class ineligible for copy triviality — even if the definitions are identical! Hence, any direct use of `std::memcpy` with an `EntityHandle` object will result in **undefined behavior**. We could have instead explicitly requested that these four special member functions be generated using `= default`:

```
class EntityHandle
{
    short int d_id;  // Note: Implementation size may increase over time.

  public:
    EntityHandle(int id);  // value constructor

    EntityHandle(const EntityHandle& rhs) = default;
        // defaulted (trivial) copy constructor

    EntityHandle(EntityHandle&& rhs);
        // user-provided move constructor

    EntityHandle& operator=(const EntityHandle& rhs) = default;
        // default (trivial) copy-assignment operator

    EntityHandle& operator=(EntityHandle&& rhs);
        // user-provided move-assignment operator

    // Implicitly generates only the destructor.
    // suppresses synthesis of the default constructor

    // ...
};
```

By explicitly defaulting these three special member functions in class scope, we (1) re-enable their generation and (2) preserve the *copy triviality* of the class.

#### 1.27.2.4 Physically decoupling the interface from the implementation

Sometimes, especially during large-scale development, avoiding compile-time coupling clients to the implementations of individual methods offers distinct maintenance advantages. Specifying that a special member function is defaulted on its first declaration (i.e., in class scope) implies that making any change to this implementation will force all clients to recompile[6]:

```
// smallscale.h

struct SmallScale
{
    SmallScale() = default;  // explicitly defaulted default constructor
};
```

Alternatively, we can choose to declare the function but deliberately *not* default it in class scope (or anywhere in the `.h` file):

```
// largescale.h

struct LargeScale
{
    LargeScale();  // user-provided default constructor
};
```

---

[6] The issue here is not just compile time, per se, but compile-time *coupling*; see **lakos20**, section 3.10.5, pp. 783–789.

We can then default just the (non-inline) implementation in a corresponding[7] `.cpp` file:

```
// largescale.cpp
#include <largescale.h>

LargeScale::LargeScale() = default;
// Generate the default implementation for this default destructor.
```

Using this *insulation* technique, we are free to change our minds and implement the default constructor ourselves in any way we see fit without necessarily forcing our clients to recompile.

## 1.27.3 Potential Pitfalls

### 1.27.3.1 Generation of defaulted functions is not guaranteed

Use of `= default` does not guarantee that the special member function of a type, `T`, will be generated. For example, a noncopyable member variable (or base class) of `T` will inhibit generation of `T`'s copy constructor even when `= default` is used. Such behavior can be observed in the presence of a `std::unique_ptr`[8] data member:

```
class Connection
{
private:
std::unique_ptr<Impl> d_impl;  // noncopyable data member

public:
Connection() = default;
Connection(const Connection&) = default;
};
```

Despite the defaulted copy constructor, `Connection` will not be copy-constructible as `std::unique_ptr` is a noncopyable type. Some compilers *may* produce a warning on the declaration of `Connection(const Connection&)`, but they are not required to do so since the example code above is well formed and would produce a compilation failure only if an attempt were made to default-construct or copy `Connection`.[9]

If desired, a possible way to ensure that a defaulted special member function has indeed been generated is to use `static_assert` (see Section 1.7, “Compile-Time Asser-

---

[7] In practice, every `.cpp` file (other than the one containing `main`) typically has a unique associated header (`.h`) file and often vice versa (a.k.a., a **component**); see **lakos20**, sections 1.6 and 1.11, pages 209–216 and 256–259, respectively.

[8] `std::unique_ptr<T>` is a move-only (movable but noncopyable) class template introduced in C++11. It models unique ownership over a dynamically allocated `T` instance, leveraging rvalue references (see Section 2.3, “Rvalue References”) to represent ownership transfer between instances:

```
int* p = new int(42);
std::unique_ptr<int> up(p);                   // OK, take ownership of p.
std::unique_ptr<int> upCopy = up;             // compile-time error
std::unique_ptr<int> upMove = std::move(up);  // OK, transfer ownership.
```

[9] Clang 8.x produces a diagnostic with no warning flags specified. GCC 8.x produces no warning, even with both `-Wall` and `-Wextra` enabled.

tions (`static_assert`)") in conjunction with an appropriate trait from the `<type_traits>` header:

```
class IdCollection
{
std::vector<int> d_ids;

public:
IdCollection() = default;
IdCollection(const IdCollection&) = default;
// ...
};

static_assert(std::is_default_constructible<IdCollection>::value,
"IdCollection must be default constructible.");

static_assert(std::is_copy_constructible<IdCollection>::value,
"IdCollection must be copy constructible.");

// ...
```

Routine use of such compile-time testing techniques can help to ensure that a type will continue to behave as expected (at no additional runtime cost) even when member (and base) types evolve as a result of ongoing software maintenance.

### 1.27.4 See Also

- Section 2.3, "Rvalue References" — Conditionally Safe C++11 feature that is the foundation of **move semantics** — the move-constructor and move-assignment special member functions can be defaulted
- Section 1.5, "Deleted Functions" — Safe C++11 feature that, among other use cases, allows the prevention of generation of special member functions, providing fine-grained control over the interface of a class if used in conjunction with `= default`
- Section 1.7, "Compile-Time Assertions (`static_assert`)" — Safe C++11 feature that checks a predicate at compile time; useful to verify that a class's special copy and move operations are available as expected

### 1.27.5 Further Reading

- Howard Hinnant, "Everything You Ever Wanted to Know About Move Semantics (and Then Some)," **hinnant14**
- Howard Hinnant, "Everything You Ever Wanted to Know About Move Semantics," **hinnant16**

### 1.27.6 Appendix: Implicit Generation of Special Member Functions

The rules a compiler uses to decide if a special member function should be generated implicitly are not entirely intuitive. Howard Hinnant, lead designer and author of the C++11 proposal for move semantics[10] (among other proposals), produced a tabular representation of such rules in the situation where the user provides a single special member function and leaves the rest to the compiler. To understand Table 1–1, after picking a special member function in the first column, the corresponding row will show what is implicitly generated by the compiler. (When selecting multiple rows, the intersection of the defaulted functions results.)

**Table 1–1: Implicit generation of special member functions. NEEDS A CREDIT LINE TO HINNANT.**

| | **Default Ctor** | **Destructor** | **Copy Ctor** | **Copy Assignment** | **Move Ctor** | **Move Assignment** |
|---|---|---|---|---|---|---|
| **Nothing** | Defaulted | Defaulted | Defaulted | Defaulted | Defaulted | Defaulted |
| **Any Ctor** | Not Declared | Defaulted | Defaulted | Defaulted | Defaulted | Defaulted |
| **Default Ctor** | User Declared | Defaulted | Defaulted | Defaulted | Defaulted | Defaulted |
| **Destructor** | Defaulted | User Declared | Defaulted[a] | Defaulted[a] | Not Declared | Not Declared |
| **Copy Ctor** | Not Declared | Defaulted | User Declared | Defaulted[a] | Not Declared | Not Declared |
| **Copy Assignment** | Defaulted | Defaulted | Defaulted[a] | User Declared | Not Declared | Not Declared |
| **Move Ctor** | Not Declared | Defaulted | Deleted | Deleted | User Declared | Not Declared |
| **Move Assignment** | Defaulted | Defaulted | Deleted | Deleted | Not Declared | User Declared |

[a] Deprecated behavior: compilers might warn upon reliance of this implicitly generated member function.

As an example, explicitly declaring a *copy-assignment* operator would result in the *default constructor*, *destructor*, and *copy constructor* being defaulted and in the *move* operations not being declared. If more than one *special member function* is user declared (regardless of whether or how it is implemented), the remaining generated member functions are those in the intersection of the corresponding rows. For example, explicitly declaring both the destructor and the default constructor would still result in the copy constructor and the copy-assignment operator being defaulted and both move operations not being declared. Relying on the compiler-generated copy operations when the destructor is anything but defaulted is dubious; if correct, defaulting them explicitly makes both their existence and intended definition clear.

---

[10] **hinnant02**

## 1.28 The Standard `[[deprecated]]` Attribute

The `[[deprecated]]` attribute discourages the use of a decorated **entity**, typically via the emission of a compiler warning.

### 1.28.1 Description

The standard `[[deprecated]]` attribute is used to portably indicate that a particular **entity** is no longer recommended and to actively discourage its use. Such deprecation typically follows the introduction of alternative constructs that, in (ideally) all ways, are superior to the original one, providing time for clients to migrate to them (*asynchronously*[1]) before the deprecated one is (in some subsequent release) removed. Although not strictly required, the Standard explicitly encourages[2] conforming compilers to produce a diagnostic message in case a program refers to any **entity** to which the `[[deprecated]]` attribute pertains. For instance, most popular compilers emit a warning whenever a `[[deprecated]]` function or object[3] is used:

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

int a;
[[deprecated]] int b;

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

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

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

---

[1] A process for ongoing improvement of legacy code bases, sometimes known as **continuous refactoring**, often allows time for clients to migrate — on their own respective schedules and time frames — from existing *deprecated* constructs to newer ones, rather than having every client change in lock step. Allowing clients time to move *asynchronously* to newer alternatives is often the only viable approach unless (1) the code base is a closed system, (2) all of the relevant code governed by a single authority, and (3) there is some sort of mechanical way to make the change.

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

[3] The `[[deprecated]]` attribute can be used portably to decorate other entities: `class`, `struct`, `union`, type alias, variable, data member, function, enumeration, template specialization. Applying `[[deprecated]]` to a specific enumerator or namespace, however, is guaranteed to be supported only since C++17; see **smith15a** for more information.

```
void algo1();

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

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

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

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

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

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

## 1.28.2 Use Cases

### 1.28.2.1 Discouraging use of an obsolete or unsafe entity

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

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

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

Although such a simple random number generator can be very useful, it might become unsuitable for heavy use because good pseudorandom number generation requires more state (and the overhead of synchronizing such state for a single `static` function can be a significant performance bottleneck) while good random number generation requires potentially very high overhead access to external sources of entropy.[4] One solution is to provide an alternative random number generator that maintains more state, allows users to decide where to store that state (the random number generator objects), and overall offers more flexibility for clients. The downside of such a change is that it comes with a functionally distinct API, requiring that users update their code to move away from the inferior solution:

---

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

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

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

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

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

By using `[[deprecated]]` as shown above, existing clients of `RandomGenerator` are informed that a superior alternative, `BetterRandomGenerator`, is available, yet they are granted time to migrate their code to the new solution (rather than their code being broken by the removal of the old solution). When clients are notified of the deprecation (thanks to a compiler diagnostic), they can schedule time to (eventually) rewrite their applications to consume the new interface.[5]

## 1.28.3 Potential Pitfalls

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

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

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

---

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

Furthermore, this interaction between `[[deprecated]]` and `-Werror` makes it impossible for owners of a low-level library to deprecate a function when releasing their code requires that they do not break the ability for *any* of their higher-level clients to compile; a single client using the to-be-deprecated function along with `-Werror` prevents the release of the code with the `[[deprecated]]` attribute on it. With the default behaviors of compilers and the frequent advice given in practice to use `-Werror` aggressively, this can make any use of `[[deprecated]]` completely unfeasible.

### 1.28.4 Annoyances

None so far

### 1.28.5 See Also

None so far

### 1.28.6 Further Reading

None so far

## 1.29 Relaxed `constexpr` Restrictions

C++14 lifts restrictions regarding use of many language features in the body of a `constexpr` function (see “`constexpr` Functions” on page 229).

### 1.29.1 Description

The cautious introduction (in C++11) of `constexpr` functions — i.e., functions eligible for compile-time evaluation — was accompanied by a set of strict rules that, despite making life easier for compiler implementers, severely narrowed the breadth of valid use cases for the feature. In C++11, `constexpr` function bodies were restricted to essentially a single `return` statement and were not permitted to have any modifiable local state (variables) or **imperative** language constructs (e.g., assignment), thereby greatly reducing their usefulness:

```
constexpr int fact11(int x)
{
static_assert(x >= 0, "");
// Error in C++11/14: x is not a constant expression.

static_assert(sizeof(x) >= 4, "");  // OK in C++11/14

return x < 2 ? 1 : x * fact11(x - 1);  // OK in C++11/14
}
```

Notice that recursive calls were supported, often leading to convoluted implementations of algorithms (compared to an **imperative** counterpart); see *Use Cases: Nonrecursive* `constexpr` *algorithms* on page 201.

The C++11 `static_assert` feature (see “Compile-Time Assertions (`static_assert`)” on page 41) was always permitted in a C++11 `constexpr` function body. However, because the input variable `x` in `fact11` (in the code snippet above) is inherently not a compile-time constant expression, it can never appear as part of a `static_assert` predicate. Note that a `constexpr` function returning `void` was also permitted:

```
constexpr void no_op() { };  // OK in C++11/14
```

Experience gained from the release and subsequent real-world use of C++11 emboldened the standard committee to lift most of these (now seemingly arbitrary) restrictions for C++14, allowing use of (nearly) *all* language constructs in the body of a `constexpr` function. In C++14, familiar non-expression-based control-flow constructs, such as `if` statements and `while` loops, are also available, as are modifiable local variables and assignment operations:

```
constexpr int fact14(int x)
{
if (x <= 2)        // error in C++11; OK in C++14
{
return 1;
}
int temp = x - 1;  // error in C++11; OK in C++14
return x * fact14(temp);
}
```

Some useful features remain disallowed in C++14; most notably, any form of dynamic allocation is not permitted, thereby preventing the use of common standard container types, such as `std::string` and `std::vector`[1]:

1. `asm` declarations
2. `goto` statements
3. Statements with labels other than `case` and `default`
4. `try` blocks
5. Definitions of variables
    (a) of other than a **literal type** (i.e., fully processable at compile time)
    (b) decorated with either `static` or `thread_local`
    (c) left uninitialized

The restrictions on what can appear in the body of a `constexpr` that remain in C++14 are reiterated here in codified form[2]:

```
template <typename T>
constexpr void f()
try {                      // Error: try outside body isn't allowed (until C++20).
std::ifstream is;  // Error: objects of *non-literal* types aren't allowed.
int x;             // error: uninitialized vars. disallowed (until C++20)
static int y = 0;  // Error: static variables are disallowed.
thread_local T t;  // Error: thread_local variables are disallowed.
try{}catch(...){}  // error: try/catch disallowed (until C++20)
if (x) goto here;  // Error: goto statements are disallowed.
[]{};              // Error: lambda expressions are disallowed (until C++17).
here: ;                // Error: labels (except case/default) aren't allowed.
asm("mov %r0");    // Error: asm directives are disallowed.
} catch(...) { }       // error: try outside body disallowed (until C++20)
```

## 1.29.2 Use Cases

### 1.29.2.1 Nonrecursive `constexpr` algorithms

The C++11 restrictions on the use of `constexpr` functions often forced programmers to implement algorithms (that would otherwise be implemented iteratively) in a recursive manner.

---

[1] In C++20, even more restrictions were lifted, allowing, for example, some limited forms of dynamic allocation, `try` blocks, and uninitialized variables.

[2] Note that the degree to which these remaining forbidden features are reported varies substantially from one popular compiler to the next.

Consider, as a familiar example, a naive[3] C++11-compliant `constexpr` implementation of a function, `fib11`, returning the *n*-th Fibonacci number[4]:

```
constexpr long long fib11(long long x)
{
return
x == 0 ? 0
: (x == 1 || x == 2) ? 1
: fib11(x - 1) + fib11(x - 2);
}
```

The implementation of the `fib11` function (above) has various undesirable properties.

1. *Reading difficulty* — Because it must be implemented using a single `return` statement, branching requires a chain of *ternary operators*, leading to a single long expression that might impede human comprehension.

2. *Inefficiency and lack of scaling* — The explosion of recursive calls is taxing on compilers: (1) the time to compile is markedly slower for the *recursive* (C++11) algorithm than it would be for its *iterative* (C++14) counterpart, even for modest inputs,[5] and (2) the compiler might simply refuse to complete the compile-time calculation if it exceeds some internal (platform-dependent) *threshold* number of operations.[6]

3. *Redundancy* — Even if the recursive implementation were suitable for small input values during compile-time evaluation, it would be unlikely to be suitable for any runtime evaluation, thereby requiring programmers to provide and maintain *two* separate

---

[3] For a more efficient (yet less intuitive) C++11 algorithm, see *Appendix: Optimized C++11 Example Algorithms, Recursive Fibonacci* on page 207.

[4] We used `long long` (instead of `long`) here to ensure a unique C++ type having at least 8 bytes on all conforming platforms for simplicity of exposition (avoiding an internal copy). We deliberately chose *not* to make the value returned `unsigned` because the extra bit does not justify changing the **algebra** (from `signed` to `unsigned`). For more discussion on these specific topics, see "`long long`" on page 92.

[5] As an example, Clang 10.0.0, running on an x86-64 machine, required more than 80 times longer to evaluate `fib(27)` implemented using the *recursive* (C++11) algorithm than to evaluate the same functionality implemented using the *iterative* (C++14) algorithm.

[6] The same Clang 10.0.0 compiler discussed in the previous footnote failed to compile `fib11(28)`:

```
error: static_assert expression is not an integral constant expression
static_assert(fib11(28) == 317811, "");
^~~~~~~~~~~~~~~~~~

note: constexpr evaluation hit maximum step limit; possible infinite loop?
```

GCC 10.x fails at `fib(36)`, with a similar diagnostic:

```
error: 'constexpr' evaluation operation count exceeds limit of 33554432
(use '-fconstexpr-ops-limit=' to increase the limit)
```

Clang 10.x fails to compile any attempt at constant evaluating `fib(28)`, with the following diagnostic message:

```
note: constexpr evaluation hit maximum step limit; possible infinite loop?
```

versions of the same algorithm: a compile-time *recursive* one and a runtime *iterative* one.

In contrast, an *imperative* implementation of a `constexpr` function implementing a function returning the *n*th Fibonacci number in C++14, `fib14`, does not suffer from any of the three issues discussed above:

```
constexpr long long fib14(long long x)
{
if (x == 0) { return 0; }

long long a = 0;
long long b = 1;

for (long long i = 2; i <= x; ++i)
{
long long temp = a + b;
a = b;
b = temp;
}

return b;
}
```

As one would expect, the compile time required to evaluate the iterative implementation (above) is manageable[7]; of course, far more computationally efficient (e.g., closed form[8]) solutions to this classic exercise are available.

#### 1.29.2.2 Optimized metaprogramming algorithms

C++14’s relaxed `constexpr` restrictions enable the use of modifiable local variables and **imperative** language constructs for metaprogramming tasks that were historically often implemented by using (byzantine) recursive template instantiation (notorious for their voracious consumption of compilation time).

Consider, as the simplest of examples, the task of counting the number of occurrences of a given type inside a **type list** represented here as an empty variadic template (see “Variadic Templates” on page 230) that can be instantiated using a variable-length sequence of arbitrary C++ types[9]:

---

[7] Both GCC 10.x and Clang 10.x evaluated `fib14(46)` 1836311903 correctly in under 20ms on a machine running Windows 10 x64 and equipped with a Intel Core i7-9700k CPU.

[8] E.g., see http://mathonline.wikidot.com/a-closed-form-of-the-fibonacci-sequence.

[9] Variadic templates are a C++11 feature having many valuable and practical uses. In this case, the variadic feature enables us to easily describe a template that takes an arbitrary number of C++ type arguments by specifying an ellipsis (...) immediately following `typename`. Emulating such functionality in C++98/03 would have required significantly more effort: A typical workaround for this use case would have been to create a template having some fixed maximum number of arguments (e.g., 20), each defaulted to some unused (incomplete) type (e.g., `Nil`):

```
struct Nil;  // arbitrary unused (incomplete) type

template <typename = Nil, typename = Nil, typename = Nil, typename = Nil>
template <typename...> struct TypeList { };
// empty variadic template instantiable with arbitrary C++ type sequence
```

Explicit instantiations of this variadic template could be used to create objects:

```
TypeList<>                 emptyList;
TypeList<int>              listOfOneInt;
TypeList<int, double, Nil> listOfThreeIntDoubleNil;
```

A naive C++11-compliant implementation of a **metafunction** `Count`, used to ascertain the (order-agnostic) number of times a given C++ type was used when creating an instance of the `TypeList` template (above), would usually make recursive use of (baroque) **partial class template specialization**[10] to satisfy the single-return-statement requirements[11]:

---

```
struct TypeList { };
// emulates the variadic TypeList template struct for up to four
// type arguments
```

Another theoretically appealing approach is to implement a Lisp-like recursive data structure; the compile-time overhead for such implementations, however, often makes them impractical.

[10] The use of class-template specialization (let alone partial specialization) might be unfamiliar to those not accustomed to writing low-level template metaprograms, but the point of this use case is to obviate such unfamiliar use. As a brief refresher, a general class template is what the client typically sees at the user interface. A specialization is typically an implementation detail consistent with the **contract** specified in the general template but somehow more restrictive. A partial specialization (possible for *class* but not *function* templates) is itself a template but with one or more of the general template parameters resolved. An **explicit** or **full specialization** of a template is one in which *all* of the template parameters have been resolved and, hence, is not itself a template. Note that a **full specialization** is a stronger candidate for a match than a partial specialization, which is a stronger match candidate than a simple template specialization, which, in turn, is a better match than the general template (which, in this example, happens to be an **incomplete type**).

[11] Notice that this `Count` **metafunction** also makes use (in its implementation) of variadic class templates to parse a **type list** of unbounded depth. Had this been a C++03 implementation, we would have been forced to create an approximation (to the simple class-template specialization containing the **parameter pack** `Tail...`) consisting of a bounded number (e.g., 20) of simple (class) template specializations, each one taking an increasing number of template arguments:

```
template <typename X, typename Y>
struct Count<X, TypeList<Y>>
: std::integral_constant<int, std::is_same<X, Y>::value> { };
// (class) template specialization for one argument

template <typename X, typename Y, typename Z>
struct Count<X, TypeList<Y, Z>>
: std::integral_constant<int,
std::is_same<X, Y>::value + std::is_same<X, Z>::value> { };
// (class) template specialization for two arguments

template <typename X, typename Y, typename Z, typename A>
struct Count<X, TypeList<Y, Z, A>>
: std::integral_constant<int,
std::is_same<X, Y>::value + Count<X, TypeList<Z, A>>::value> { };
// recursive (class) template specialization for three arguments

// ...
```

```
#include <type_traits>  // std::integral_constant, std::is_same

template <typename X, typename List> struct Count;
// general template used to characterize the interface for the Count
// metafunction
// Note that this general template is an incomplete type.

template <typename X>
struct Count<X, TypeList<>> : std::integral_constant<int, 0> { };
// partial (class) template specialization of the general Count template
// (derived from the integral-constant type representing a compile-time
// 0), used to represent the base case for the recursion --- i.e., when
// the supplied TypeList is empty
// The payload (i.e., the enumerated value member of the base class)
// representing the number of elements in the list is 0.

template <typename X, typename Head, typename... Tail>
struct Count<X, TypeList<Head, Tail...>>
: std::integral_constant<int,
std::is_same<X, Head>::value + Count<X, TypeList<Tail...>>::value> { };
// simple (class) template specialization of the general count template
// for when the supplied list is not empty
// In this case, the second parameter will be partitioned as the first
// type in the sequence and the (possibly empty) remainder of the
// TypeList. The compile-time value of the base class will be either the
// same as or one greater than the value accumulated in the TypeList so
// far, depending on whether the first element is the same as the one
// supplied as the first type to Count.

static_assert(Count<int, TypeList<int, char, int, bool>>::value == 2, "");
```

Notice that we made use of a C++11 **parameter pack** — `Tail...` (see “Variadic Templates” on page 230) in the implementation of the simple template specialization to package up and pass along any remaining types.

As should be obvious by now, the C++11 restriction encourages both somewhat rarified metaprogramming-related knowledge and a *recursive* implementation that can be compile-time intensive in practice.[12] By exploiting C++14’s relaxed `constexpr` rules, a simpler and typically more compile-time friendly *imperative* solution can be realized:

```
template <typename X, typename... Ts>
constexpr int count()
{
bool matches[sizeof...(Ts)] = { std::is_same<X, Ts>::value... };
// Create a corresponding array of bits where 1 indicates sameness.

int result = 0;
for (bool m : matches)  // (C++11) range-based for loop
```

---

[12] For a more efficient C++11 version of `Count`, see *Appendix: Optimized C++11 Example Algorithms,* `constexpr` *type list* `Count` *algorithm* on page 207.

```
    {
        result += m;        // Add up 1 bits in the array.
    }

    return result;  // Return the accumulated number of matches.
}
```

The implementation above — though more efficient and comprehensible — will require some initial learning for those unfamiliar with modern C++ variadics. The general idea here is to use **pack expansion** in a nonrecursive manner[13] to initialize the `matches` array with a sequence of zeros and ones (representing, respectively, mismatch and matches between `X` and a type in the `Ts...` pack) and then iterate over the array to accumulate the number of ones as the final `result`. This `constexpr`-based solution is both easier to understand and typically faster to compile.[14]

### 1.29.3 Potential Pitfalls

None so far

### 1.29.4 Annoyances

None so far

### 1.29.5 See Also

- “`constexpr` Functions” — Conditionally safe C++11 feature that first introduced compile-time evaluations of functions.
- “`constexpr` Variables” — Conditionally safe C++11 features that first introduced variables usable as constant expressions.
- “Variadic Templates” — Conditionally safe C++11 feature allowing templates to accept an arbitrary number of parameters.

---

[13] **Pack expansion** is a language construct that expands a **variadic pack** during compilation, generating code for each element of the pack. This construct, along, with a **parameter pack** itself is a fundamental building block of variadic templates, introduced in C++11. As a minimal example, consider the variadic function template, `e`:

```
template <int... Is> void e() { f(Is...); }
```

`e` is a function template that can be instantiated with an arbitrary number of compile-time-constant integers. The `int... Is` syntax declares a **variadic pack** of compile-time-constant integers. The `Is...` syntax (used to invoke `f`) is a basic form of pack expansion that will resolve to all the integers contained in the `Is` pack, separated by commas. For instance, invoking `e<0, 1, 2, 3>()` results in the subsequent invocation of `f(0, 1, 2, 3)`. Note that — as seen in the `count` example (which starts on page 204) — any arbitrary expression containing a variadic pack can be expanded:

```
template <int... Is> void g() { h((Is > 0)...); }
```

The `(Is > 0)...` expansion (above) will resolve to `N` comma-separated Boolean values, where `N` is the number of elements contained in the `Is` **variadic pack**. As an example of this expansion, invoking `g<5, -3, 9>()` results in the subsequence invocation of `h(true, false, true)`.

[14] For a type list containing 1024 types, the imperative (C++14) solution compiles about twice as fast on GCC 10.x and roughly 2.6 times faster on Clang 10.x.

## 1.29.6 Further Reading

None so far

## 1.29.7 Appendix: Optimized C++11 Example Algorithms

### 1.29.7.1 Recursive Fibonacci

Even with the restrictions imposed by C++11, we can write a more efficient recursive algorithm to calculate the $n$th Fibonacci number:

```
#include <utility>  // std::pair

constexpr std::pair<long long, long long> fib11NextFibs(
const std::pair<long long, long long> prev,  // last two calculations
int count)                                   // remaining steps
{
return (count == 0) ? prev : fib11NextFibs(
std::pair<long long, long long>(prev.second,
prev.first + prev.second),
count - 1);
}

constexpr long long fib11Optimized(long long n)
{
return fib11NextFibs(
std::pair<long long, long long>(0, 1), // first two numbers
n                                      // number of steps
).second;
}
```

### 1.29.7.2 `constexpr` type list `Count` algorithm

As with the `fib11Optimized` example, providing a more efficient version of the `Count` algorithm in C++11 is also possible, by accumulating the final result through recursive `constexpr` function invocations:

```
#include <type_traits>  // std::is_same

template <typename>
constexpr int count11Optimized() { return 0; }
// Base case: always return 0.

template <typename X, typename Head, typename... Tail>
constexpr int count11Optimized()
// Recursive case: compare the desired type (X) and the first type in
// the list (Head) for equality, turn the result of the comparison
// into either 1 (equal) or 0 (not equal), and recurse with the rest
// of the type list (Tail...).
{
return (std::is_same<X, Head>::value ? 1 : 0)
+ count11Optimized<X, Tail...>();
}
```

This algorithm can be optimized even further in C++11 by using a technique similar to the one shown for the iterative C++14 implementation. By leveraging a `std::array` as compile-time storage for bits where `1` indicates equality between types, we can compute the final result with a fixed number of template instantiations:

```
#include <array>        // std::array
#include <type_traits>  // std::is_same

template <int N>
constexpr int count11VeryOptimizedImpl(
const std::array<bool, N>& bits,  // storage for "type sameness" bits
int i)                            // current array index
{
return i < N
? bits[i] + count11VeryOptimizedImpl<N>(bits, i + 1)
// Recursively read every element from the bits array and
// accumulate into a final result.
: 0;
}

template <typename X, typename... Ts>
constexpr int count11VeryOptimized()
{
return count11VeryOptimizedImpl<sizeof...(Ts)>(
std::array<bool, sizeof...(Ts)>{ std::is_same<X, Ts>::value... },
// Leverage pack expansion to avoid recursive instantiations.
0);
}
```

Note that, despite being recursive, `count11VeryOptimizedImpl` will be instantiated only once with `N` equal to the number of elements in the `Ts...` pack.

## 1.30 Lambda-Capture Expressions

Lambda-capture expressions enable **synthetization** (spontaneous implicit creation) of arbitrary data members within **closures** generated by lambda expressions (see “Lambdas” on page 231).

### 1.30.1 Description

In C++11, lambda expressions can capture variables in the surrounding scope either *by value* or *by reference*[1]:

```
int i = 0;
auto f0 = [i]{ };   // Create a copy of i in the generated closure named f0.
auto f1 = [&i]{ };  // Store a reference to i in the generated closure named f1.
```

Although one could specify *which* and *how* existing variables were captured, the programmer had no control over the creation of new variables within a **closure**. C++14 extends the **lambda-introducer** syntax to support implicit creation of arbitrary data members inside a **closure** via either **copy initialization** or **list initialization**:

```
auto f2 = [i = 10]{ /* body of closure */ };
// Synthesize an int data member, i, initialized with 10 in the closure.

auto f3 = [c{'a'}]{ /* body of closure */ };
// Synthesize a char data member, c, initialized with 'a' in the closure.
```

Note that the identifiers `i` and `c` above do not refer to any existing variable; they are specified by the programmer creating the closure. For example, the **closure** type assigned (i.e., bound) to `f2` (above) is similar in functionality to an **invocable** `struct` containing an `int` data member:

```
// pseudocode
struct f2LikeInvocableStruct
{
int i = 10;  // The type int is deduced from the initialization expression.
auto operator()() const { /* closure body */ }  // The struct is invocable.
};
```

The type of the data member is deduced from the initialization expression provided as part of the capture in the same vein as `auto` (see “`auto`” on page 224) type deduction; hence, it’s not possible to synthesize an uninitialized **closure** data member:

```
auto f4 = [u]{ };    // Error: u initializer is missing for lambda capture.
auto f5 = [v{}]{ };  // Error: v's type cannot be deduced.
```

It is possible, however, to use variables outside the scope of the lambda as part of a lambda-capture expression (even capturing them *by reference* by prepending the `&` token to the name of the synthesized data member):

---

[1] We use the familiar (C++11) feature `auto` (see “`auto`” on page 224) to deduce a closure’s type since there is no way to name such a type explicitly.

```
int i = 0;  // zero-initialized int variable defined in the enclosing scope

auto f6 = [j   = i]{ };  // OK, the local j data member is a copy of i.
auto f7 = [&ir = i]{ };  // OK, the local ir data member is an alias to i.
```

Though capturing *by reference* is possible, enforcing `const` on a lambda-capture expression is not:

```
auto f8 = [const i = 10]{ };                    // error: invalid syntax
auto f9 = [const auto i = 10]{ };               // error: invalid syntax
auto fA = [i = static_cast<const int>(10)]{ };  // OK, const is ignored.
```

The initialization expression is evaluated during the *creation* of the closure, not its *invocation*:

```
#include <cassert>  // standard C assert macro

void g()
{
int i = 0;

auto fB = [k = ++i]{ };  // ++i is evaluated at creation only.
assert(i == 1);  // OK

fB();  // Invoke fB (no change to i).
assert(i == 1);  // OK
}
```

Finally, using the same identifier as an existing variable is possible for a synthesized capture, resulting in the original variable being **shadowed** (essentially hidden) in the lambda expression’s body but not in its **declared interface**. In the example below, we use the (C++11) compile-time operator `decltype` (see “`decltype`” on page 28) to infer the C++ type from the initializer in the capture to create a parameter of that same type as that part of its **declared interface**[2][3]:

```
#include <type_traits>  // std::is_same

int i = 0;

auto fC = [i = 'a'](decltype(i) arg)
{
static_assert(std::is_same_v<decltype(arg), int>, "");
// i in the interface (same as arg) refers to the int parameter.

static_assert(std::is_same_v<decltype(i), char>, "");
// i in the body refers to the char data member deduced at capture.
};
```

---

[2] Note that, in the shadowing example defining `fC`, GCC version 10.x incorrectly evaluates `decltype(i)` inside the body of the lambda expression as `const char`, rather that `char`; see *Potential Pitfalls: Forwarding an existing variable into a closure always results in an object (never a reference)* on page 214.

[3] Here we are using the (C++14) variable template (see “Variable Templates” on page 176) version of the standard `is_same` metafunction where `std::is_same<A, B>::value` is replaced with `std::is_same_v<A, B>`.

Notice that we have again used `decltype`, in conjunction with the standard `is_same` metafunction (which is `true` if and only if its two arguments are the same C++ type). This time, we’re using `decltype` to demonstrate that the type (`int`), extracted from the local variable `i` within the declared-interface portion of `fC`, is distinct from the type (`char`) extracted from the `i` within in `fC`’s body. In other words, the effect of initializing a variable in the capture portion of the lambda is to hide the name of an existing variable that would otherwise be accessible in the lambda’s body.[4]

## 1.30.2 Use Cases

### 1.30.2.1 Moving (as opposed to copying) objects into a closure

Lambda-capture expressions can be used to *move* (see “Rvalue References” on page 226) an existing variable into a closure[5] (as opposed to capturing it *by copy* or *by reference*). As an example of *needing* to move from an existing object into a closure, consider the problem of accessing the data managed by **`std::unique_ptr`** (movable but not copyable) from a separate thread — for example, by enqueuing a task in a **thread pool**:

```
ThreadPool::Handle processDatasetAsync(std::unique_ptr<Dataset> dataset)
{
return getThreadPool().enqueueTask([data = std::move(dataset)]
```

---

[4] Also note that, since the deduced `char` member variable, `i`, is not materially used (**ODR-used**) in the body of the lambda expression assigned (bound) to `fC`, some compilers, e.g., Clang, may warn:

```
warning: lambda capture 'i' is not required to be captured for this use
```

[5] Though possible, it is surprisingly difficult in C++11 to *move* from an existing variable into a closure. Programmers are either forced to pay the price of an unnecessary copy or to employ esoteric and fragile techniques, such as writing a wrapper that hijacks the behavior of its copy constructor to do a *move* instead:

```
template <typename T>
struct MoveOnCopy  // wrapper template used to hijack copy ctor to do move
{
T d_obj;

MoveOnCopy(T&& object) : d_obj{std::move(object)} { }
MoveOnCopy(MoveOnCopy& rhs) : d_obj{std::move(rhs.d_obj)} { }
};

void f()
{
std::unique_ptr<int> handle{new int(100)};  // move-only
// Create an example of a handle type with a large body.

MoveOnCopy<decltype(handle)> wrapper(std::move(handle));
// Create an instance of a wrapper that moves on copy.

auto lambda = [wrapper](){ /* use wrapper.d_obj */ };
// Create a "copy" from a wrapper that is captured by value.
}
```

In the example above, we make use of the bespoke (“hacked”) `MoveOnCopy` class template to wrap a movable object; when the lambda-capture expression tries to *copy* the wrapper (*by value*), the wrapper in turn *moves* the wrapped `handle` into the body of the closure.

```
{
return processDataset(data);
});
}
```

As illustrated above, the `dataset` smart pointer is moved into the closure passed to `enqueueTask` by leveraging lambda-capture expressions — the **`std::unique_ptr`** is *moved* to a different thread because a copy would have not been possible.

#### 1.30.2.2 Providing mutable state for a closure

Lambda-capture expressions can be useful in conjunction with `mutable` lambda expressions to provide an initial state that will change across invocations of the closure. Consider, for instance, the task of logging how many TCP packets have been received on a socket (e.g., for debugging or monitoring purposes)[6]:

```
TcpSocket tcpSocket(27015);  // some well-known port number
tcpSocket.onPacketReceived([counter = 0]() mutable
{
std::cout << "Received " << ++counter << " packet(s)\n";
// ...
});
```

Use of `counter = 0` as part of the **lambda introducer** tersely produces a **function object** that has an internal counter (initialized with zero), which is incremented on every received packet. Compared to, say, capturing a `counter` variable *by reference* in the closure, the solution above limits the scope of `counter` to the body of the lambda expression and ties its lifetime to the closure itself, thereby preventing any risk of dangling references.

#### 1.30.2.3 Capturing a modifiable copy of an existing `const` variable

Capturing a variable *by value* in C++11 does allow the programmer to control its `const` qualification; the generated closure data member will have the same `const` qualification as the captured variable, irrespective of whether the lambda is decorated with `mutable`:

```
void f()
{
int i = 0;
const int ci = 0;

auto lc = [i, ci]             // This lambda is not decorated with mutable.
{
static_assert(std::is_same_v<decltype(i), int>, "");
static_assert(std::is_same_v<decltype(ci), const int>, "");
};

auto lm = [i, ci]() mutable       // Decorating with mutable has no effect.
{
```

---

[6] In this example, we are making use of the (C++11) `mutable` feature of lambdas to enable the counter to be modified on each invocation.

```
static_assert(std::is_same_v<decltype(i), int>, "");
static_assert(std::is_same_v<decltype(ci), const int>, "");
};
}
```

In some cases, however, a lambda capturing a `const` variable *by value* might need to modify that value when invoked. As an example, consider the task of comparing the output of two Sudoku-solving algorithms, executed in parallel:

```
template <typename Algorithm> void solve(Puzzle&);
// This solve function template mutates a Sudoku grid in place to solution.

void performAlgorithmComparison()
{
const Puzzle puzzle = generateRandomSudokuPuzzle();
// const-correct: puzzle is not going to be mutated after being
// randomly generated.

auto task0 = getThreadPool().enqueueTask([puzzle]() mutable
{
solve<NaiveAlgorithm>(puzzle);  // Error: puzzle is const-qualified.
return puzzle;
});

auto task1 = getThreadPool().enqueueTask([puzzle]() mutable
{
solve<FastAlgorithm>(puzzle);  // Error: puzzle is const-qualified.
return puzzle;
});

waitForCompletion(task0, task1);
// ...
}
```

The code above will fail to compile as capturing `puzzle` will result in a `const`-qualified closure data member, despite the presence of `mutable`. A convenient workaround is to use a (C++14) lambda-capture expression in which a local modifiable copy is deduced:

```
// ...

const Puzzle puzzle = generateRandomSudokuPuzzle();

auto task0 = getThreadPool().enqueueTask([p = puzzle]() mutable
{
solve<NaiveAlgorithm>(p);  // OK, p is now modifiable.
return puzzle;
});

// ...
```

Note that use of `p = puzzle` (above) is roughly equivalent to the creation of a new variable using `auto` (i.e., `auto p = puzzle;`), which guarantees that the type of `p` will be deduced as a non-`const` `Puzzle`. Capturing an existing `const` variable as a mutable copy is possible, but doing the opposite is not easy; see *Annoyances: There's no easy way to synthesize a* `const` *data member* on page 215.

## 1.30.3 Potential Pitfalls

### 1.30.3.1 Forwarding an existing variable into a closure always results in an object (never a reference)

Lambda-capture expressions allow existing variables to be **perfectly forwarded** (see "Forwarding References" on page 232) into a closure:

```
template <typename T>
void f(T&& x)  // x is of type forwarding reference to T.
{
auto lambda = [y = std::forward<T>(x)]
// Perfectly forward x into the closure.
{
// ... (use y directly in this lambda body)
};
}
```

Because `std::forward<T>` can evaluate to a reference (depending on the nature of `T`), programmers might incorrectly assume that a capture such as `y = std::forward<T>(x)` (above) is somehow either a capture *by value* or a capture *by reference*, depending on the original **value category** of `x`.

Remembering that lambda-capture expressions work similarly to `auto` type deduction for variables, however, reveals that such captures will *always* result in an object, *never* a reference:

```
// pseudocode (auto is not allowed in a lambda introducer.)
auto lambda = [auto y = std::forward<T>(x)] { };
// The capture expression above is semantically similar to an auto
// (deduced-type) variable.
```

If `x` was originally an *lvalue*, then `y` will be equivalent to a *by-copy* capture of `x`. Otherwise, `y` will be equivalent to a *by-move* capture of `x`.[7]

If the desired semantics are to capture `x` *by move* if it originated from **rvalue** and *by reference* otherwise, then the use of an extra layer of indirection (using, e.g., `std::tuple`) is required:

```
template <typename T>
void f(T&& x)
{
auto lambda = [y = std::tuple<T>(std::forward<T>(x))]
{
// ... (Use std::get<0>(y) instead of y in this lambda body.)
```

---

[7] Note that both *by-copy* and *by-move* capture communicate **value** for **value-semantic types**.

```
    };
}
```

In the revised code example above, `T` will be an **lvalue reference** if `x` was originally an **lvalue**, resulting in the **synthetization** of a `std::tuple` containing an **lvalue reference**, which — in turn — has semantics equivalent to `x`’s being captured *by reference*. Otherwise, `T` will not be a reference type, and `x` will be *moved* into the closure.

## 1.30.4 Annoyances

### 1.30.4.1 There’s no easy way to synthesize a `const` data member

Consider the (hypothetical) case where the programmer desires to capture a copy of a non-`const` integer `k` as a `const` closure data member:

```
[k = static_cast<const int>(k)]() mutable  // const is ignored
{
    ++k;  // "OK" -- i.e., compiles anyway even though we don't want it to
};

[const k = k]() mutable  // error: invalid syntax
{
    ++k;  // no easy way to force this variable to be const
};
```

The language simply does not provide a convenient mechanism for synthesizing, from a modifiable variable, a `const` data member. If such a `const` data member somehow proves to be necessary, we can either create a `ConstWrapper struct` (that adds `const` to the captured object) or write a full-fledged **function object** in lieu of the leaner **lambda expression**. Alternatively, a `const` copy of the object can be captured with traditional (C++11) lambda-capture expressions:

```
int k;
const int kc = k;

auto l = [kc]() mutable
{
    ++kc;  // error: increment of read-only variable kc
};
```

### 1.30.4.2 `std::function` supports only copyable callable objects

Any lambda expression capturing a move-only object produces a closure type that is itself movable but *not* copyable:

```
void f()
{
    std::unique_ptr<int> moo(new char);    // some move-only object
    auto la = [moo = std::move(moo)]{ };   // lambda that does move capture

    static_assert(false == std::is_copy_constructible_v<decltype(la)>, "");
```

```
  static_assert( true == std::is_move_constructible_v<decltype(la)>, "");
  }
```

Lambdas are sometimes used to initialize instances of `std::function`, which requires the stored **callable object** to be copyable:

```
  std::function<void()> f = la;  // Error: la must be copyable.
```

Such a limitation — which is more likely to be encountered when using lambda-capture expressions — can make `std::function` unsuitable for use cases where move-only closures might conceivably be reasonable. Possible workarounds include (1) using a different type-erased, **callable object** wrapper type that supports move-only callable objects,[8] (2) taking a performance hit by wrapping the desired **callable object** into a copyable wrapper (such as `std::shared_ptr`), or (3) designing software such that noncopyable objects, once constructed, never need to move.[9]

## 1.30.5 See Also

- “Lambdas” on page 231 — provides the needed background for understanding the feature in general
- “Braced Initialization” on page 225 — illustrates one possible way of initializing the captures
- ‘`auto`” on page 224 — offers a model with the same type deduction rules
- ‘Rvalue References” on page 226 — gives a full description of an important feature used in conjunction with moveable types.
- ‘Forwarding References” on page 232 — describes a feature that contributes to a source of misunderstanding of this feature

## 1.30.6 Further Reading

None so far

---

[8] The `any_invocable` library type, proposed for C++23, is an example of a type-erased wrapper for move-only callable objects; see **calabrese20**.

[9] For an in-depth discussion of how large systems can benefit from a design that embraces local arena memory allocators and, thus, minimizes the use of moves across natural memory boundaries identified throughout the system, see **lakos22**.

## 1.31 Raw String Literals

Raw string literals obviate the need to escape each contained special character individually.

### 1.31.1 Description

A *raw* string literal is a new form of syntax for string literals that allows developers to embed arbitrary character sequences in a program’s source code, without having to modify them by escaping individual special characters. As an introductory example, suppose that we want to write a small program to print out a line of code:

```
printf("Hello, %s%c\n", "World", '!');
```

In C++03, capturing the C statement above in a string literal would require five escape (\) characters distributed throughout the string:

```
#include <iostream>  // std::cout, std::endl

int main()
{
std::cout << "printf(\"Hello, %s%c\\n\", \"World\", '!');" << std::endl;
return 0;  //        ^            ^  ^   ^      ^
//             escaped characters
}
```

If we use C++11’s *raw* string-literal syntax, no escaping is required:

```
#include <iostream>  // std::cout, std::endl

int main()
{
std::cout << R"(printf("Hello, %s%c\n", "World", '!');)" << std::endl;
return 0;  //^ ^                                      ^
//   additional raw string-literal syntax (C++11)
}
```

To represent the original character data as a raw string literal, we typically need only to add a capital `R` immediately (adjacently) before the starting quote (`"`) and nest the character data within parentheses, `( )` (with some exceptions; see *Collisions* on page 218). Sequences of characters that would be escaped in a regular string literal are instead interpreted verbatim:

```
const char s0[] = R"({ "key": "value" })";
// OK, equivalent to "{ \"key\": \"value\" }"
```

In contrast to conventional string literals, *raw* string literals (1) treat unescaped embedded double quotes (`"`) as literal data, (2) do not interpret special-character escape sequences (e.g., `\n`, `\t`), and (3) interpret all white-space characters (i.e., vertical[1] as well as horizontal) present in the source file as part of the string contents[2]:

---

[1] In conventional string literals, a new line in the source before the end of a string is considered an error. If new lines are desired, they must be represented with an escape sequence as `\n`.

[2] In this example, we assume that all trailing white space has been stripped since even trailing white space in a raw literal would be captured.

```
const char s1[] = R"(line one
line two
line three)";
// OK
```

Note that any literal tab characters are treated the same as a `\t` and hence can be problematic, especially when developers have inconsistent tab settings; see *Potential Pitfalls: Unexpected indentation* on page 220. Finally, all string literals are concatenated with adjacent ones in the same way the conventional ones are in C++03:

```
const char s2[] = R"(line one)"         "\n"
"line two"          "\n"
R"(    line three)";
// OK, equivalent to "line one\nlinetwo\n    line three"
```

### 1.31.1.1 Collisions

Although unlikely, the data to be expressed within a string literal might itself have the character sequence `)"` embedded within it:

```
printf("printf(\"Hello, World!\")")
//                             ^^
// The )" character sequence terminates a typical raw string literal.
```

If we use the basic syntax for a *raw* string literal we will get a syntax error:

```
const char s3[] = R"(printf("printf(\"Hello, World!\")"))";  // collision
//                                                    ^^
//                       Syntax error after literal ends
```

To circumvent this problem, we could escape every special character in the string separately, as in C++03, but the result is difficult to read and error prone:

```
const char s4[] = "printf(\"printf(\\\"Hello, World!\\\")\")";  // error prone
```

Instead, we can use the (extended) disambiguation syntax of *raw* string literals to resolve the issue:

```
const char s5[] = R"###(printf("printf(\"Hello, World!\")"))###";  // cleaner
```

This disambiguation syntax allows us to insert an essentially arbitrary[3] sequence of characters between the outermost quote/parentheses pairs that avoids the collision with the literal data when taken as a combined sequence (e.g., `)###"`):

```
//                       delimiter and parenthesis
//                  v~~~                           ~~~v
const char s6[] = R"xyz(<-- Literal String Data -->)xyz";
//                ^     ^~~~~~~~~~~~~~~~~~~~~~~~~^
//                |         string contents
//                |
//                | uppercase R
```

---

[3] The delimiter of a raw string literal can comprise any member of the **basic source character set** except space, backslash, parentheses, and the control characters representing horizontal tab, vertical tab, form feed, and new line.

The value of `s6` above is equivalent to `"<-- Literal String Data -->"`. Every raw string literal comprises these syntactical elements, in order:

- an uppercase `R`
- opening double quotes, `"`
- an optional arbitrary sequence of characters called the *delimiter* (e.g., `xyz`)
- opening parenthesis, `(`
- the contents of the string
- the closing parenthesis, `)`
- the same delimiter (if any) specified previously (i.e., `xyz`, not reversed)
- the closing double quotes, `"`

The delimiter can be (and, in practice, very often is) an empty character sequence:

```
const char s7[] = R"("Hello, World!")";
// OK, equivalent to \"Hello, World!\"
```

A nonempty delimiter (e.g., `!`) can be used to disambiguate any appearance of the `)"` character sequence within the literal data

```
const char s8[] = R"!("--- R"(Raw literals are not recursive!)" ---")!";
// OK, equivalent to \"--- R\"(Raw literals are not recursive!)\" ---\"
```

Had an empty delimiter been used to initialize `s8` (above), the compiler would have produced a (perhaps obscure) compile-time error:

```
// error: decrement of read-only location

const char s8[] = R"("--- R"(Raw literals are not recursive!)" ---")";
//                                                             ^~
```

In fact, it could turn out that a program with an unexpectedly terminated *raw* string literal could still be valid and compile quietly:

```
printf(R"("Live-Free, don't (ever)","Die!");
// Prints:  "Live-Free, don't (ever

printf((R"("Live-Free, don't (ever)","Die!"));
// Prints:  Die!
```

Fortunately, examples like the one above are invariably contrived, not accidental.

## 1.31.2 Use Cases

### 1.31.2.1 Embedding code in a C++ program

When a source code snippet needs to be embedded as part of the source code of a C++ program, use of a *raw* string literal can significantly reduce the syntactic noise that would otherwise be caused by repeated escape sequences. As an example, consider a regular expression (for an online shopping product ID) represented as a conventional string literal:

```
const char* productIdRegex = "[0-9]{5}\\(\".*\"\\)";
// This regular expression matches strings like 12345("Product").
```

Not only do the backslashes obscure the meaning to human readers, a mechanical translation is often needed[4] when transforming between source and data, introducing significant opportunities for human error. Using a raw string literal solves these problems:

```
const char* productIdRegex = R"([0-9]{5}\(".*"\))";
```

Another format that benefits from raw string literals is JSON, due to its frequent use of double quotes:

```
const char* testProductResponse = R"(
{
"productId": "58215(\"Camera\")",
"availableUnits": 5,
"relatedProducts": ["59214(\"CameraBag\")", "42931(\"SdStorageCard\")"]
})";
```

With a conventional string literal, the JSON string above would require every occurrence of `"` and \ to be escaped and every new line to be represented as `\n`, resulting in visual noise, less interoperability with other tools accepting or producing JSON, and heightened risk during manual maintenance.

Finally, raw string literals can also be helpful for white-space–sensitive languages, such as Python (but see *Potential Pitfalls: Encoding of new lines and white space* on page 221):

```
const char* testPythonInterpreterPrint = R"(def test():
print("test printing from Python")
)";
```

## 1.31.3 Potential Pitfalls

### 1.31.3.1 Unexpected indentation

Consistent indentation and formatting of source code facilitates human comprehension of program structure. Space and tabulation (`\t`) characters[5] used for the purpose of source code formatting are, however, always interpreted as part of a raw string literal's contents:

```
void emitPythonEvaluator(const std::string& expression)
```

---

[4] Such as when you want to copy the contents of the string literal into an online regular-expression validation tool.

[5] Always representing indentation as the precise number of spaces (instead of tab characters) — especially when committed to source-code control systems — goes a long way to avoiding this issue.

```
{
std::cout << R"(
def evaluate():
print("Evaluating...")
return )" << expression;
}
```

Despite the intention of the programmer to aid readability by indenting the above raw string literal consistently with the rest of the code, the streamed data will contain a large number of spaces (or tabulation characters), resulting in an invalid Python program:

```
def evaluate():
print("Evaluating...")
return someExpression
# ^~~~~~
# Error: excessive indentation
```

Correct code would start unindented and then be indented the same number of spaces (e.g., exactly four):

```
def evaluate():
print("Evaluating...")
return someExpression
```

Correct — albeit visually jarring — code can be expressed with a single *raw* string literal, but visualizing the final output requires some effort:

```
void emitPythonEvaluator(const char *expression)
{
std::cout << R"(def evaluate():
print("Evaluating...")
return )" << expression;
}
```

When more explicit control is desired, we can use a mixture of **raw string literals** and explicit new lines represented as **conventional string literals**:

```
void emitPythonEvaluator2(const char *expression)
{
std::cout <<
R"(def evaluate():)"                  "\n"
R"(    print("Evaluating..."))"      "\n"
R"(    return )" << expression;
}
```

### 1.31.3.2 Encoding of new lines and white space

The intent of the feature is that new lines should map to a single `\n` character regardless of how new lines are encoded in the source file. The wording of the C++ Standard, however, is not entirely clear.[6] While all major compiler implementations act in accordance with the original intent of the feature, relying on a specific new line encoding may lead to nonportable code until clarity is achieved.

---

[6] **miller13**

In a similar fashion, the type of white-space characters (e.g., tabs versus spaces) used as part of a raw string literal can be significant. As an example, consider a unit test verifying that a string representing the status of the system is as expected:

```
void verifyDefaultOutput()
{
const std::string output = System::outputStatus();
const std::string expected = R"(Current status:
- No violations detected.)";

EXPECT(output == expected);
}
```

The unit test might pass for years, until, for instance, the company's indentation style changes from tabulation characters to spaces, leading to a mismatch and thus test failures.[7]

## 1.31.4 Annoyances

None so far

## 1.31.5 See Also

None so far

## 1.31.6 Further Reading

None so far

---

[7] A well-designed unit test will typically be imbued with *expected values*, rather than values that were produced by the previous run. The latter is sometimes referred to as a **benchmark test**, and such tests are often implemented as *diffs* against a file containing output from a previous run. This file has presumably been reviewed and is known (believed) to be correct and is sometimes called the **golden file**. Though ill advised, when trying to get a new version of the software to pass the benchmark test and when the precise format of the output of a system changes subtly, the **golden file** may be summarily jettisoned — and the new output installed in its stead — with little if any detailed review. Hence, well-designed unit tests will often hard code exactly what is to be expected (nothing more or less) directly in the **test-driver** source code.

# Chapter 2

# Conditionally Safe Features

Intro text should be here.

## 2.1 `auto`

placeholder text.........

## 2.2 Braced Initialization

placeholder text.........

## 2.3 Rvalue References

placeholder text..........

## 2.4 Default Member Initializers

placeholder text.........

## 2.5 `constexpr` Variables

placeholder text...........

---

## 2.6 `constexpr` Functions

placeholder text.........

## 2.7 Variadic Templates

placeholder text.........

## 2.8 Lambda Expressions

placeholder text.........

## 2.9 Forwarding References

placeholder text...........

## 2.10 `noexcept`

placeholder text.........

## 2.11 Generic Lambdas

placeholder text.........

# Chapter 3

# Unsafe Features

Intro text should be here.

## 3.1 The `[[carries_dependency]]` Attribute

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

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

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

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

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

Phasellus et bibendum lacus. Pellentesque elementum lectus sed elit auctor, eget condimentum nisi accumsan. Integer tellus elit, pellentesque eget tincidunt vitae, varius et dolor. Curabitur vel lorem mi. Maecenas sed mauris ultricies, ornare purus non, mattis felis. Quisque porta sapien id nibh semper finibus. Fusce pharetra nunc a cursus egestas. Aliquam viverra metus libero. Quisque ultrices metus non neque fringilla, sit amet ultricies sem varius. Aenean sagittis nisi non metus blandit, quis laoreet dolor vestibulum. Proin ultrices nulla quis vehicula porta. Maecenas ut magna vitae lacus suscipit imperdiet sed quis nulla.

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

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

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

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

## 3.2 Function Return Type Deduction

placeholder text........

# Chapter 4

# Parting Thoughts

## 4.1 Testing Section

## 4.2 Testing Another Section

# Todo list