Embracing Modern C++ Safely

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

Embracing Modern C++ Safely

John Lakos

Vittorio Romeo

Addison-Wesley

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

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

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

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

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

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

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

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

Contents

Contents

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

John Lakos, author of *Large-Scale C++ Software Design* (Addison-Wesley, 1996) and *Large-Scale C++ Volume I: Process and Architecture* (Addison-Wesley, 2019), serves at Bloomberg in New York City as a senior architect and mentor for C++ software development worldwide. He is also an active voting member of the C++ Standards Committee's Evolution Working Group. From 1997 to 2001, Dr. Lakos directed the design and development of infrastructure libraries for proprietary analytic financial applications at Bear Stearns. From 1983 to 1997, Dr. Lakos was employed at Mentor Graphics, where he developed large frameworks and advanced ICCAD applications for which he holds multiple software patents. His academic credentials include a Ph.D. in Computer Science (1997) and an Sc.D. in Electrical Engineering (1989) from Columbia University. Dr. Lakos received his undergraduate degrees from MIT in Mathematics (1982) and Computer Science (1981).

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

Rostislav Khlebnikov is called Slava.

Alisdair Meredith has bionic teeth.

Chapter 0

Introduction

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

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

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

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

What Makes This Book Different

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

Scope for the First Edition

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

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

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

Over time, we expect, and hope, that practicing senior developers will emerge entirely from the postmodern C++ era. By then, a book that focuses on all of the important features of modern C++ would naturally include many of those that were around before C++11. With that horizon in mind, we are actively planning to cover pre-C++11 material in future editions. For the time being, however, we highly recommend *Effective C++* by Scott Meyers[2] as a concise, practical treatment of many important and useful C++98/03 features.

The *EMC++S* White Paper

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

Facts (Not Opinions)

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

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

Elucidation (Not Prescription)

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

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

[2] **meyers05**

passing knowledge of the recently added C++ features and a desire to perfect their understanding. Our writing style is intentionally tight, with the goal of providing you with facts, concise objective analysis, and cogent, real-world examples. By doing so we spare you the task of wading through introductory material. If you are entirely unfamiliar with a feature, we suggest you start with a more elementary and language-centric text such as *The C++ Programming Language* by Bjarne Stroustrup.[3]

Real-World (Not Contrived) Examples

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

At Scale (Not Overly Simplistic) Programs

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

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

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

[3] **stroustrup13**

of misuse, added maintenance burden borne by using a new feature in an older code base, and training needs for developers who might not be familiar with that feature.

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

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

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

A *Safe* Feature

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

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

A *Conditionally Safe* Feature

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

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

An *Unsafe* Feature

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

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

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

Modern C++ Feature Catalog

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

Organization

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

- Chapter 0: Introduction
- Chapter 1: Safe Features
- Chapter 2: Conditionally Safe Features
- Chapter 3: Unsafe Features
- Chapter 4: Parting Thoughts

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

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

- Description

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

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

How To Use This Book

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

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

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

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

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

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

Chapter 1

Safe Features

Intro text should be here.

Generalized Attribute Support

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

Description

Developers are often aware of information that cannot be easily deduced 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 of a well-written program. 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 — Stating explicit assumptions in code to achieve better optimizations* on page 13). Customized annotations targeted at external tools might be beneficial as well.

C++ attribute syntax

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

```
[[attribute_name]]
```

A single annotation can consist of zero or more attributes:

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

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

```
[[attribute_name()]]            // same as attribute_name
[[deprecated("bad API")]]       // single-argument attribute
[[theoretical(1, "two", 3.0)]]  // multiple-argument attribute
[[complicated({1, 2, 3} + 5)]]  // arbitrary tokens[1]
```

Note that having an incorrect number of arguments or an incompatible argument type is a compile-time error for all attributes defined by the Standard; the behavior for all other attributes, however, is **implementation-defined** (see *Potential Pitfalls — Unrecognized attributes have implementation-defined behavior* on page 16).

Any attribute may be qualified with an attribute namespace[2] (a single arbitrary identifier):

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

[1] GCC offered no support for certain tokens in the attributes until GCC v9.3 (c. 2020).

[2] 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 attribute namespaces.

C++ attribute placement

Attributes can be placed in a variety of locations within the C++ grammar. For each such location, the Standard defines the entity or statement to which the attribute is said to *appertain*. For example, an attribute in front of a simple declaration statement appertains to each of the entities declared by the statement, whereas an attribute placed immediately after the declared name appertains only to that entity:

```
[[foo]] void f(), g();   // foo appertains to both f() and g().
void u(), v [[foo]] ();  // foo appertains only to v().
```

Attributes can apply to an entity without a name (e.g., anonymous **union** or **enum**):

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

The valid positions for any particular attribute are constrained to only those locations where the attribute appertains to the entity to which it applies. That is, an attribute such as `noreturn`, which applies only to functions, would be valid syntactically but not semantically valid if it were used to annotate any other kind of entity or syntactic element. Misplacement of a standard attribute results in an ill-formed program[3]:

```
void [[noreturn]] x() {       // Error, cannot be applied to a type
     [[noreturn]] int i;      // Error, cannot be applied to a variable
     [[noreturn]] { ;throw }  // Error, cannot be applied to a statement
}
```

The empty attribute specifier sequence `[[]]` is allowed to appear anywhere the C++ grammar allows attributes.

Common compiler-dependent attributes

Prior to C++11, no standardized syntax for attributes was available 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 syntactically consistent manner. If an unknown attribute is encountered during compilation, it is ignored, emitting a likely[4] nonfatal diagnostic.

Table 1 provides several examples 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 17).

Portability is the biggest advantage of preferring standard syntax when it is available for compiler- and external-tool-specific attributes. Because most compilers will simply ignore unknown attributes that use standard attribute syntax (and, as of C++17, they are required to do so), conditional compilation is no longer required.

[3]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, using even a standard attribute might lead to a different behavior on different compilers.

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

Table 1: Some standardized compiler-specific attributes

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

Use Cases

Prompting 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[5] can be used to inform the compiler (and developers) that a function's return value should not be ignored[6]:

```
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 forgetting to inspect the result of a function[7]:

```
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 'int UDPListener::start()' declared
         with attribute 'warn_unused_result' [-Wunused-result]
```

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

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

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

Hinting at additional optimization opportunities

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

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

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

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

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

Stating explicit assumptions in code to achieve better optimizations

Although the presence of an attribute usually has no effect on the behavior of any well-formed program besides its runtime performance, an attribute sometimes imparts knowledge to the compiler, which, if incorrect, could alter the intended behavior of the program. As an example of this more forceful form of attribute, consider the GCC-specific `[[gnu::const]]` attribute (also available in Clang). When applied to a function, this attribute instructs the compiler to *assume* that the function is a **pure function**, which has no **side effects**. In other words, the function always returns the same value for a given set of arguments, and the globally reachable state of the program is not altered by the function. For example, a function performing a linear interpolation between two values may be annotated with `[[gnu::const]]`:

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

More generally, the return value of a function annotated with `[[gnu::const]]` is not permitted to depend on any state that might change between its successive invocations. For example, it is not allowed to examine contents of memory supplied to it by address. In contrast, functions annotated with a similar but more lenient `[[gnu::pure]]` attribute are allowed to return values that depend on any nonvolatile state. Therefore, functions such as `strlen` or `memcmp`, which read but do not modify the observable state, may be annotated with `[[gnu::pure]]` but not `[[gnu::const]]`.

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

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

In the 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::const]]` 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; see *Potential Pitfalls — Some attributes, if misused, can affect program correctness* on page 16.

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*[8] rules. In partic-

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

ular, consider GSL C26481 (Bounds rule #1),[9] which forbids any pointer arithmetic, instead suggesting that users rely on the `gsl::span` type[10]:

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

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

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

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

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

Creating new attributes to express semantic properties

Other uses of attributes for static analysis include statements of properties that cannot otherwise be deduced. 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. Using the standard attribute to inform the analyzer of such a precondition has a distinct advantage of requiring nothing other than the agreement between the developer and the static analyzer regarding the namespace and the name of the attribute. For example, we could choose to designate `home_grown::in_same_block(p1, p2)` for this purpose:

```
// lib.h:

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

The compiler will simply ignore this unknown attribute. However, because our static-analysis tool knows the meaning of the `home_grown::in_same_block` attribute, it will report, at analysis time, defects that might otherwise have resulted in **undefined behavior** at run time:

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

[9] **?**

[10] `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.

```
void client()
{
    double a[10], b[10];
    f(a, b);  // Unrelated pointers --- Our static analyzer reports an error.
}
```

Potential Pitfalls

Unrecognized attributes have implementation-defined behavior

Although standard attributes work well and are portable across all platforms, the behavior of compiler-specific and user-specified attributes is entirely implementation defined, with unrecognized attributes typically resulting in compiler warnings. Such warnings can typically be disabled (e.g., on GCC using `-Wno-attributes`), but, if they are, misspellings in even standard attributes will go unreported.[11]

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. However, misuse of some attributes can lead to incorrect results and/or undefined behavior.

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

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

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

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

Annoyances

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

See Also

- "noreturn" (§1.1, p. 86) ♦ presents a standard attribute used to indicated that a particular function never returns control flow to its caller.
- "deprecated" (§1.2, p. 135) ♦ presents a standard attribute that discourages the use of an entity via compiler diagnostics.
- "carries_dependency" (§3.1, p. 340) ♦ presents a standard attribute used to communicate release-consume dependency-chain information to the compiler to avoid unnecessary memory-fence instructions.

Further Reading

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

Consecutive Right-Angle Brackets

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

Description

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

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

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

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

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

For templates that take only type parameters, there's no issue. When the template parameter is a non-type, however, the greater-than or right-shift operators might 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;  // Error, always has been
C<i >> j>   b1;  // Error, in C++11, OK in C++03
C<(i > j)>  a2;  // OK
C<(i >> j)> b2;  // OK
```

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

Use Cases

Avoiding annoying whitespace when composing template types

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

```
// OK in both C++03 and C++11
std::list<std::map<int, std::vector<std::string> > > idToNameMappingList1;

// OK in C++11, compile-time error in C++03
std::list<std::map<int, std::vector<std::string>>>   idToNameMappingList2;
```

Potential Pitfalls

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

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

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

The easy fix is simply to parenthesize the expression:

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

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

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

[1]Example adapted from **?**

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

Annoyances

See Also

Further Reading

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

Constructors Calling Other Constructors

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

Description

A **delegating constructor** is a constructor of a **user-defined type (UDT)** — 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 is to specify the name of the type as the only element in the **member initializer list**:

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

struct S0
{
  int         d_i;
  std::string d_s;

  S0(int i)         : d_i(i)        {} // nondelegating constructor
  S0()              : S0(0)         {} // OK, delegates to S0(int)
  S0(const char *s) : S0(0), d_s(s) {} // 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 25). 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:

```
#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 (the destructor of S2 is never invoked)
```

Although the destructor of a partially constructed object will not be invoked, the destructors of each successfully constructed base and of data members will still be invoked:

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

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

struct C : B
{
    A d_a;

    C()  { cout << "C() "; throw 0; }  // nondelegating 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_a` of type `A` were fully constructed, and so their respective destructors are called, even though the destructor for class `C` itself is never executed.

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 object's destructor *is* 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
```

Use Cases

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 member function, not being a constructor, would be unable to make use of member initializer lists to initialize base classes and data members 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 be constructed either (1) by a 32-bit address and a 16-bit port or (2) by an IPV4 string with `XXX.XXX.XXX.XXX:XXXXX` format[1]:

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

class IPV4Host
{
     // ...
private:
    int connect(std::uint32_t address, std::uint16_t port);

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

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

```
// C++03 (obsolete)
#include <cstdint>  // std::uint16_t, std::uint32_t
```

[1]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, `IPV4Address`, that itself might be constructed in multiple ways; see *Potential Pitfalls — Suboptimal factoring* on page 25.

```
class IPV4Host
{
    // ...

private:
    int connect(std::uint32_t address, std::uint16_t port);
    void init(std::uint32_t address, std::uint16_t port)  // helper function
    {
        if (!connect(address, port))  // factored implementation of needed logic
        {
            throw ConnectionException{address, port};
        }
    }

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

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

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

With C++11 delegating constructors, the constructor accepting a string can be rewritten to delegate to the one accepting `address` and `port`, avoiding repetition without having to use a private function:

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

class IPV4Host
{
     // ...
private:
    int connect(std::uint32_t address, std::uint16_t port);

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)}
    {
    }
};
```

Using delegating constructors results in less boilerplate and fewer runtime operations, as data members and base classes can be initialized directly through the member initializer list.

Potential Pitfalls

Delegation cycles

If a constructor delegates to itself either directly or indirectly, the program is **ill formed, no diagnostic required (IFNDR)**. While some compilers can, under certain conditions, detect delegation cycles at compile time, they are neither required nor necessarily able to do so. For example, even the simplest delegation cycles might not result in a diagnostic from a compiler[2]:

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

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* on page 22. 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[3]:

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

[2]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
```

[3]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 **?** and see **?**, sections 0.4, 3.2.7, and 3.5.9, pp. 20–28, 529–530, and 674–676, respectively.

```
class IPV4Address
{
    std::uint32_t d_address;
    std::uint16_t d_port;

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

    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) taking an `IPV4Address` object as an argument.

Annoyances

See Also

- "Forwarding References" (§2.1, p. 283) ⧫ provides perfect forwarding of arguments from one ctor to another.
- "Variadic Templates" (§2.1, p. 319) ⧫ describes how to implement constructors that forward an arbitrary list of arguments to other constructors.

Further Reading

Operator for Extracting Expression Types

The keyword `decltype` enables the compile-time inspection of the **declared type** of an **entity** *or* the type and **value category** of an expression. Note that the special construct `decltype(auto)` has a separate meaning; see Section 3.2.“`decltypeauto`” on page 372.

Description

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

Use with entities

If the operand is an unparenthesized **id-expression** or unparenthesized member access, `decltype` yields the *declared type*, meaning the type of the *entity* indicated by the operand:

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

Use with expressions

When `decltype` is used with any other expression `E` of type `T`, including parenthesized id-expression or parenthesized member access, the result incorporates both the expression’s type and its value category (see Section 2.1.“*rvalue* References” on page 310):

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

In general, *prvalues* can be passed to `decltype` in a number of ways, including numeric literals, function calls that return by value, and explicitly created temporaries:

```
decltype(0)   i; // -> int
int f();
decltype(f()) j; // -> int
struct S{};
decltype(S()) k; // -> S
```

An entity name passed to `decltype`, as mentioned above, produces the type of the entity. If an entity name is enclosed in an additional set of parentheses, however, `decltype` interprets its argument as an expression and its result incorporates the value category:

```
int i;
decltype(i)   l = i; // -> int
decltype((i)) m = i; // -> int&
```

Similarly, for all other *lvalue* expressions, the result of `decltype` will be an *lvalue* reference:

```
int* pi = &i;
decltype(*pi) j = *pi; // -> int&
decltype(++i) k = ++i; // -> int&
```

Finally, the value category of the expression will be an *xvalue* if it is a cast to or a function returning an *rvalue* reference:

```
int i;
decltype(static_cast<int&&>(i)) j = static_cast<int&&>(i); // -> int&&
int&& g();
decltype(g()) k = g();                                      // -> int&&
```

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

```
void test1()
{
    int i = 0;
    decltype(i++) j;  // equivalent to int j;
    assert(i == 0);   // The expression i++ was not evaluated.
}
```

Note that the choice of using the postfix increment is significant; the prefix increment yields a different type:

```
void test2()
{
    int i = 0;
    int m = 1;
    decltype(++i) k = m; // equivalent to int& k = m;
    assert(i == 0); // The expression ++1 is not evaluated.
}
```

Use Cases

Avoiding unnecessary use of explicit typenames

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

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

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

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

Expressing type-consistency explicitly

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

```
class Packet
{
    // ...

public:
    std::uint8_t checksumLength() const;
};
```

This 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 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 an `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 defect would naturally have been avoided:

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

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

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 the trailing return type of the function (see Section 1.1."Trailing Return" on page 112) and 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;
}
```

Using `decltype(a + b)` as a return type is significantly different from relying on automatic **return-type deduction**; see Section 2.1."**auto** Variables" on page 177. Note that this particular use involves significant repetition of the expression `a+b`. See *Annoyances — Mechanical repetition of expressions might be required* on page 32 for a discussion of ways in which this might be avoided.

Determining the validity of a generic expression

In the context of generic-library development, `decltype` can be used in conjunction with **SFINAE** ("Substitution Failure Is Not An Error") to validate an expression involving a template parameter.

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

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

The client-facing `sortRange` function (above) delegates its behavior to an overloaded `sortRangeImpl` function (below), invoking the latter with the `range` and a **disambiguator** of type **`int`**. The type of this additional parameter, whose 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 `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.[3]

The relative position of `decltype(range.sort())` in the signature of `sortRangeImpl` is not significant, as long as it is visible to the compiler during template substitution. The example shown in the main text uses a function parameter that is defaulted to `nullptr`. Alternatives involving a trailing return type or a default template argument are also viable:

```
#include <utility>  // declval
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) -> void;
    // std::declval is used to generate a reference to Range that can
    // be used in an unevaluated expression.
```

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.

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

Potential Pitfalls

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

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

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

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

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

Annoyances

Mechanical repetition of expressions might be required

As mentioned in *Use Cases — Creating an auxiliary variable of generic type* on page 30, using `decltype` to capture a value of an expression that is about to be used, or for the return value of an expression, can often lead to repeating the same expression in multiple places (three distinct ones in the earlier example).

An alternate solution to this problem is to capture the result of the `decltype` expression in a **`typedef`**, **`using`** type alias, or as a defaulted template parameter — but that runs into the problem that it can be used only once the expression is valid. A defaulted **`template`** parameter cannot reference parameter names as it is written before them, and a type alias cannot be introduced prior to the return type being needed. A solution to this problem lies in using standard library function `std::declval` to create expressions of the appropriate type without needing to reference the actual function parameters by name:

```
template <typename A, typename B,
          typename Result = decltype(std::declval<const A&>() +
                                     std::declval<const B&>())>
Result loggedSum(const A& a, const B& b)
{
    Result result = a + b;  // no duplication of the decltype expression
    LOG_TRACE << a << " + " << b << "=" << result;
    return result;
}
```

Here, `std::declval`, a function that cannot be executed at runtime and is only appropriate for use in **unevaluated contexts**, produces an expression of the specified type. When mixed with `decltype`, this lets us determine the result types for expressions without needing to construct (or even being able to construct) objects of the needed types.

See Also

- "*rvalue* References" (Section 2.1, p. 310) ♦ The `decltype` operator yields precise information on whether an expression is an *lvalue* or *rvalue*.
- "**`using`** Aliases" (Section 1.1, p. 121) ♦ Oftentimes, it is useful to give a name to the type yielded by `decltype`, which is done with a **`using`** alias.
- "**`auto`** Variables" (Section 2.1, p. 177) ♦ The type computed by `decltype` is similar to, but distinct from, the type deduction used by **`auto`**.
- "`decltypeauto`" (Section 3.2, p. 372) ♦ `decltype` type computation rules can be useful in conjunction with an **`auto`** variable.

Further Reading

Using `= default` for Special Member Functions

Use of **`= default`** in a **special member function**'s declaration instructs the compiler to attempt to generate the function automatically.

Description

An important aspect of C++ class design is the understanding that the compiler will attempt to generate certain member functions to *create*, *copy*, *destroy*, and now *move* (see Section 2.1."*rvalue* References" on page 310) an object unless developers implement 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 **user-provided special member functions** requires remembering the numerous rules the compiler uses.

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 have been deprecated. Specifically, even in the presence of a user-provided destructor, both the copy constructor and the copy-assignment operator have historically been generated implicitly. Relying on such generated behavior is problematic 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.

Here, we will briefly illustrate a few common cases and then refer you to Howard Hinnant's now famous table (see page 44 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 generation 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 37. Note that 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 1.1."Deleted Functions" on page 46.

Example 2: Providing just a copy constructor Now, consider a **`struct`** with a user-provided copy constructor:

```
struct S2
{
    S2(const S2&);  // user-provided copy constructor
};
```

A user-provided copy constructor (1) suppresses the generation 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. 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 generation 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.

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 without treating it as being user provided. The compiler-generated version for a special member function is required to call the corresponding special member functions on every base class in base-class-declaration order and then 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.

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
```

```
    // 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 access to that generated function will be regulated accordingly:

```
struct S5
{
private:
    S5(const S5&) = default;             // private copy constructor
    S5& operator=(const S5&) = 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, using **`= default`** on the 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 1.1."Deleted Functions" on page 46), using **`= default`** affords the fine-grained control over which special member functions are to be generated and/or made publicly available.

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-copy-assignable data member).
```

Alternatively, an explicitly defaulted noninline 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 41.

Use Cases

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 together 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_code`:

```
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_code;   // 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 37) 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;
};
```

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 and a maintenance burden. What's more, even if the function definition was empty, implementing it explicitly often degraded performance compared to a **trivial** default. Hence, such standards tended to evolve toward conventionally commenting out (e.g., using `//!`) the declarations of functions having an empty body rather than providing it explicitly:

```
class C2
{
     // ...

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 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.
};
```

Preserving type triviality

It can be beneficial if a particular type is trivial. The type is considered trivial if its default constructor is trivial and it is **trivially copyable** — i.e., it has no non-trivial copy or move constructors, no non-trivial copy or move assignment operators, at least one of those nondeleted, and a trivial destructor. As an example, consider a simple trivial `Metrics` type in the code snippet below containing certain collected metrics for our application:

```
struct Metrics
{
    int d_numRequests;  // number of requests to the service
    int d_numErrors;    // number of error responses

    // All special member functions are generated implicitly.
};
```

Now imagine that we would like to add a constructor to this struct to make its use more convenient:

```
struct Metrics
{
    int d_numRequests;  // number of requests to the service
    int d_numErrors;    // number of error responses

    Metrics(int, int);  // user-provided value constructor

    // Generation of default constructor is suppressed.
};
```

As illustrated in *Appendix: Implicit Generation of Special Member Functions* on page 44, the presence of a user-provided constructor suppressed the implicit generation of the default constructor. Replacing the default constructor with a seemingly equivalent user-provided one might appear to work as intended:

```
struct Metrics
{
    int d_numRequests;  // number of requests to the service
    int d_numErrors;    // number of error responses

    Metrics(int, int);  // user-provided value constructor
    Metrics() {}        // user-provided default constructor

    // Default constructor is user-provided: Metrics is not trivial.
};
```

The user-provided nature of the default constructor, however, renders the `Metrics` type non-trivial — even if the definitions are identical! In contrast, explicitly requesting the default constructor be generated using `=default` restores the triviality of the type:

```
struct Metrics
{
    int d_numRequests;  // number of requests to the service
    int d_numErrors;    // number of error responses

    Metrics(int, int);    // user-provided value constructor
    Metrics() = default;  // defaulted (trivial) default constructor

    // Default constructor is defaulted: Metrics is trivial.
};
```

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:

```
// smallscale.h:

struct SmallScale
{
    SmallScale() = default;  // explicitly defaulted default constructor
};
```

The important issue regarding recompilation here is not merely compile time per se but compile-time coupling[1]

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
};
```

We can then default just the non-inline implementation in a corresponding[2] `.cpp` file:

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

LargeScale::LargeScale() = default;
```

[1]See **?**, section 3.10.5, pp. 783–789.

[2]In practice, every `.cpp` file (other than the one containing `main`) typically has a unique associated header (`.h`) file and often vice versa (with the `.cpp` and `.h` pair of files constituting a component); see **?**, sections 1.6 and 1.11, pages 209–216 and 256–259, respectively.

```
    // 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.

Potential Pitfalls

Defaulted special member functions cannot restore trivial copyability

Library classes often rely on whether the type on which they are operating is eligible for being copied with `memcpy` for optimization purposes. Such could be the case for implementing, say, `vector`, which would make a single call to `memcpy` when growing its buffer. For the `memcpy` or `memmove` to be well-defined, however, the type of the object that is stored in the buffer must be trivially copyable. One might assume that this trait means that, as long as the copy constructor of the type is trivial, this optimization will apply. Defaulting the copy operations would then allow us to achieve this goal, while allowing the type to have a non-trivial destructor or move operation. Such, however, is not the case.

The requirements for a type to be considered trivially copyable — and thus eligible for use with `memcpy` — include triviality of all of its nondeleted copy and move operations as well as of its destructor. Furthermore, library authors cannot perform fine-grained dispatch based on which operations on the type are in fact trivial. Even if we detect that the type is trivially copy-constructible with the `std::is_trivially_copy_constructible` trait and know that our code would use only copy constructors (and not copy assignment nor any move operations), we still would not be able to use `memcpy` unless the more restrictive `std::is_trivially_copyable` trait is also **`true`**.

Annoyances

Generation of defaulted functions is not guaranteed

Using `=`**`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`[3] data member:

```
#include <memory>  // std::unique_ptr
class Connection
{
private:
    class Impl;                    // nested implementation class
```

[3] `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 2.1."*rvalue* References" on page 310) 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;            // Error, copy is deleted
std::unique_ptr<int> upMove = std::move(up); // OK, transfer ownership.
```

```
    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 a `Connection`.[4]

If desired, a possible way to ensure that a defaulted special member function has indeed been generated is to use `static_assert` (see Section 1.1."`static_assert`" on page 104) 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.");

// ...
```

Routinely using 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.

See Also

- "Deleted Functions" (§1.1, p. 46) ♦ describes a companion feature, `= delete`, that can be used to suppress access to implicitly generated special member functions.
- "`static_assert`" (§1.1, p. 104) ♦ describes a facility that can be used to verify at compile time that undesirable copy and move operations are declared to be accessibly.

[4]Clang 8.x and later produces a diagnostic with no warning flags specified. MSVC produces a diagnostic if `/Wall` is specified. As of this writing, GCC produces no warning, even with both `-Wall` and `-Wextra` enabled.

- "*rvalue* References" (§2.1, p. 310) ♦ provides the bases for **move operations**, namely, the move-constructor and move-assignment special member functions, which too can be defaulted.

Further Reading

- Howard Hinnant, "Everything You Ever Wanted to Know About Move Semantics (and Then Some)," **?**
- Howard Hinnant, "Everything You Ever Wanted to Know About Move Semantics," **?**

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[5] (among other proposals), produced a tabular representation[6] 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, after picking a special member function in the first column, the corresponding row will show what is implicitly generated by the compiler.

Table 1: Implicit Generation of Special Member Functions.

	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

[5] **?**
[6] **?**

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.

Using `= delete` for Arbitrary Functions

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

Description

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

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

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

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

Use Cases

Suppressing special member function generation

When instantiating an object of user-defined type, special member functions that have not been declared explicitly are often generated automatically by the compiler. 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; see Section 1.1."Defaulted Functions" on page 34. For certain kinds of types, the notion of *copying* is not meaningful, and hence permitting the compiler to generate *copy* operations would be inappropriate. The two special member functions controlling **move operations**, introduced in C++11, are typically implemented as effective optimizations of ***copy* operations** and thus would be similarly contraindicated. Much less frequently, a useful notion of moving exists where copying does not, and so we may choose to have move operations generated, while *copy* operations are explicitly deleted; see Section 2.1."*rvalue* References" on page 310.

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:

```
#include <cstdio>  // FILE
class FileHandle
{
private:
    // ...

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

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

    // ...
};
```

Not implementing 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. 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 clearer compiler diagnostics:

```
class FileHandle
{
private:
    // ...
    // Declarations of copy constructor and copy assignment are now public.

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

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

    // ...
};
```

Using the `= delete` syntax on declarations that are private results in error messages concerning privacy, not the use of deleted functions. Care must be exercised to make *both* changes when converting code from the old style to the new syntax.

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 <cstdio>   // puts
#include <cstring>  // memset

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

Sadly, inadvertently reversing the order of the last two arguments is a commonly recurring error, and the C language provides no help. As shown above, `memset` writes the nonprinting character `5` (e.g., the integer value of ASCII `'*'`) 42 times — way past the end of `buf`. 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;  // defense against misuse
```

Pernicious user errors can now be reported during compilation:

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

Preventing all implicit conversions

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

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

    // ...
};

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

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

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

It is commonly advised to avoid deriving publicly from concrete classes because by doing so, we do not hide the underlying capabilities, which can easily be accessed (potentially breaking any invariants the derived class may want to keep) via assignment to a pointer or reference to a base class, with no casting required. Worse, 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. A more robust approach would be to use layering or at least private inheritance.[1] Best practices notwithstanding,[2] 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 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.
```

[1] For more on improving compositional designs at scale, see **?**, sections 3.5.10.5 and 3.7.3, pp. 687–703 and 726–727, respectively.

[2] See "Inheritance and Object-Oriented Programming," Item 38, **?**, pp. 132–135.

```
}
```

If the need for a `ForwardAudioStream` type persists, we can always consider reimplementing it more carefully later.[3] As discussed at the beginning of this section, the protection provided by this example is easily circumvented:

```
void g(const ForwardAudioStream &stream)
{
    AudioStream fullStream = stream;
    fullStream.play();   // OK
    fullStream.rewind(); // compiles OK, but what happens at run time?
}
```

Hiding non**`virtual`** functions is something one undertakes only after attaining a complete understanding of what makes such an unorthodox endeavor *safe*; see, in particular, the appendix of Section 3.1."`final`" on page 341.

Potential Pitfalls

Annoyances

Deleting a function declares it

It should come as no surprise that when we "declare" a **free function** followed by **`=delete`**, we *are* in fact *declaring* it. For example, consider the pair of overloads of functions `f` declared taking a **`char`** and **`int`**, respectively:

```
int f(char);           // (1) accessible declaration of f taking a char
int f(int) = delete;   // (2) inaccessible declaration of f taking an int

int x = f('a');        // OK, exact match for (1) f(char), which is accessible
int y = f(123);        // Error, exact match for (2) f(int), which is deleted
```

It is necessary that both functions above are *declared* so that both of them can participate in overload resolution; it is only after the inaccessible overload is selected tat it will be reported as a compile-time error.

When it comes to deleting certain special member functions of a class (or class template), however, what might seem like a tiny bit of extra, self-documenting code can have subtle, unintended consequences as evidenced below.

Let's begin by considering an empty **`struct`**, `S0`:

```
struct S0 { };  // The default constructor is declared implicitly.

S0 x0;  // OK, invokes the implicitly generated default constructor
```

As `S0` defines not constructors, destructors, or assignment operators, the compiler will generate (**declare** and **define**), for `S0`, all *six* of the special member functions available as of C++11; see Section 1.1."Defaulted Functions" on page 34.

[3] **?**, sections 3.5.10.5 and 3.7.3, pp. 687–703 and 726–727

Next, suppose we create a second **struct**, `S1`, that differs from `S0` only in that `S1` declares a *value* constructor taking an **int**:

```
struct S1  // Implicit declaration of the default constructor is suppressed.
{
    S1(int);  // explicit declaration of value constructor
};

S1 y1(5);  // OK, invokes the explicitly declared value constructor
S1 x1;     // Error, no declaration for default constructor S1::S1()
```

By explicitly declaring a *value* constructor (or any other constructor for that matter), we automatically suppress the implicit declaration of the default constructor for `S1`. If suppressing the default destructor is *not* our intention, we can always reinstate it via an explicit declaration followed by **=default;** — see Section 1.1."Defaulted Functions" on page 34.

Let's now suppose it *is* our intention to suppress generation of the default constructor and, to make our intention clear, we elect to explicitly declare and **delete** it:

```
struct S2  // Implicit declaration of the default constructor is suppressed.
{
    S2() = delete;  // explicit declaration of inaccessible default constructor
    S2(int);        // explicit declaration of value constructor
};

S2 y2(5);  // OK, invokes the explicitly declared value constructor
S2 x2;     // Error, use of deleted function, S2::S2()
```

By declaring and then deleting the default constructor we have, it would appear that we (1) made our intentions clear and (2) improved diagnostics for our clients at the cost of a single extra line of self-documenting code. Ah, if C++ were only that straightforward.

Deleting certain special member functions — i.e., *default* constructor, *move* constructor, or *move*-assignment operator — that are not necessarily implicitly declared can have nonobvious consequence that adversely affect subtle compile-time properties of a class. One such subtle property is whether the compiler considers it to be a **literal type** — i.e., a type whose *value* is eligible for use as part of a **constant expression**. This same property of being a literal type is what determines whether an arbitrary type may be passed by value in the interface of a `constexpr` function; see Section 2.1."`constexpr` Functions" on page 179.

As a simple illustration of a subtle compile-time difference between `S1` and `S2`, consider this practically useful *pattern* for a developer's "test" function that will compile if and only if its by-value parameter, `x`, is of a literal type:

```
constexpr int test(S0 x) { return 0; }  // OK,    S0 is     a literal type.
constexpr int test(S1 x) { return 0; }  // Error, S1 is not a literal type.
constexpr int test(S2 x) { return 0; }  // OK,    S2 is     a literal type.
```

For the compiler to treat a given class type as a literal type, it must, among other things, have at least one constructor (other than the *copy* or *move* constructor) declared as `constexpr`.

In the case of the empty `S0` class, the implicitly generated default constructor is **trivial** and so it is implicitly *declared* `constexpr` too. Class `S1`'s explicitly declared *non*-`constexpr` value constructor suppresses the declaration of its only `constexpr` constructor, the default constructor; hence, `S1` does not qualify as a *literal type*.

Finally, by conspicuously declaring and deleting `S2`'s default constructor, we *declare* it nonetheless. What's more, the declaration brought about by deleting it is the same as if it had been generated implicitly (or declared explicitly and then defaulted); hence, `S2`, unlike `S1`, *is* a literal type. Go figure!

See Also

- "Defaulted Functions" (Section 1.1, p. 34) ♦ Companion feature that enables defaulting, as opposed to deleting, special member functions.

- "*rvalue* References" (Section 2.1, p. 310) ♦ The two *move* variants of special member functions, which use *rvalue* references in their signatures, may also be subject to deletion.

Further Reading

- "Item 27" of **?**

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.

Description

Though sometimes desirable, implicit conversions achieved via user-defined *conversion functions* — either **converting constructors** accepting a single argument or **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 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 58.

The companion problem stemming from an *implicit conversion operator*, albeit less severe, remained:

```
void h(double d);

double f3(const Point& 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
    int k(s0);                        // OK, direct initialization
    double d = s0;                    // Error, copy initialization
    int j = static_cast<int>(s0);     // OK, static cast
    if (s0) { }                       // Error, contextual conversion to bool
    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** applicable to arguments of logical operations (e.g., `&&`, `||`, and `!`) and conditions of most control-flow constructs (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 55):

```
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
    bool b0 = s2;                  // Error, copy initialization
    bool b1(s2);                   // OK, direct initialization)
    !s2;                           // OK, contextual conversion to bool
-
    s2 && s2;                      // OK, contextual conversion to bool
}
```

Use Cases

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 Standard input/output library, for example, uses this idiom to determine if a given stream is valid:

```
// C++03
#include <ostream>  // 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 go undetected:

```
class ostream
{
    // ...

    public:
```

```
    /* implicit */ operator bool();  // hypothetical (bad) idea
};

int client(ostream& out)
{
    // ...
    return out + 1;  // likely a latent runtime bug: always returns 1 or 2
}
```

The classic workaround, the **safe-bool idiom**,[2] 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 are treated equivalently. With explicit conversion operators, such workarounds are no longer required. As discussed in *Description* on page 53, 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:

```
#include <cstddef>  // std::size_t
#include <iostream> // std::cerr
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)
    {
        // ...
        return 0;  // success
    }

    std::cerr << "Invalid connection handle.\n";
    return -1;  // failure
}
```

[2]https://www.artima.com/cppsource/safebool.html

[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.

Having an **explicit** conversion operator prevents unwanted conversions to **bool** that might otherwise happen inadvertently:

```
bool hasEnoughThroughput(const ConnectionHandle& ingress,
                         const ConnectionHandle& egress)
{
    return ingress.throughput() <= egress;  // Error, thankfully
//                                     ^~~~~~
}
```

After the relational operator (`<=`) in the example above, the programmer mistakenly wrote `egress` instead of `egress.maxThroughput()`. Fortunately the conversion operator of `ConnectionHandle` was declared to be **explicit** and a compile-time error ensued; if the conversion had been *implicit*, the example code above would have compiled, and, if executed, the above (faulty) implementation of the `hasEnoughThroughput` function would have silently exhibited well-defined but incorrect behavior.

Potential Pitfalls

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, for proxy types that are intended to be drop-in replacements for the types they represent, implicit conversions are precisely what we want. Consider, for example, a `NibbleConstReference` proxy type that represents the 4-bit integer elements of a `PackedNibbleVector`:

```
class NibbleConstReference
{
    // ...
public:
    operator int() const; // IMPLICIT

    // ...
};

class PackedNibbleVector
{
    // ...
public:
    bool empty() const;
    NibbleConstReference operator[](int index) const;

    // ...
};
```

The `NibbleConstReference` proxy is intended to interoperate well with other integral types in various expressions and making its conversion operator **explicit** hinders its intended use as a drop-in replacement by requiring an explicit conversion (a.k.a. cast):

```
int firstOrZero(const PackedNibbleVector& values)
```

```
{
    return values.empty()
        ? 0
        : values[0]; // Compiles only if conversion operator is implicit.
}
```

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 53 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 (e.g., `PointUtil::magnitude(const Point& p)`) might be better still.[5]

Annoyances

None so far

See Also

None so far

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 **?**, sections 3.2.7–3.2.8, pp 529–552.

Threadsafe Function-Scope `static` Variables

Function-scope **`static`** objects are now guaranteed to be initialized free of race conditions in the presence of multiple concurrent threads.

Description

When a variable is declared within the body of a function, we say that the variable is declared at **function scope** (a.k.a. **local scope**). An object (e.g., `iLocal`) that is declared **`static`** within the body of a function (e.g., `f`) will be initialized the first time the **flow of control** passes through the **definition** of that object:

```
#include <cassert>  // standard C assert macro

int f(int i) // function returning the first argument with which it is called
{
    static int iLocal = i;  // object initialized once only, on the first call
    return iLocal;          // the same iLocal value is returned on every call
}

int main()
{
    int a = f(10);  assert(a == 10);  // Initialize and return iLocal.
    int b = f(20);  assert(b == 10);  // Return iLocal.
    int c = f(30);  assert(c == 10);  // Return iLocal.

    return 0;
}
```

In the simple example above, the function, `f`, initializes its **`static`** object, `iLocal`, with its argument, `i`, only the first time it is called and then always returns the same value (e.g., 10). Hence, when that function is called repeatedly with distinct arguments while initializing, respectively, `a`, `b`, and `c`, all three of these variables are initialized to the same value, 10, used the first time `f` was invoked (i.e., to initialize `a`). Although the function-scope **`static`** object, `iLocal`, was created after `main` was entered, it will not be destroyed until after `main` exits.

A function such as `f` might also be called before `main` is entered:

```
// ...

int a = f(10);  // Initialize and return iLocal.
int b = f(20);  // Return iLocal.
int c = f(30);  // Return iLocal.

int main()
{
    assert(a == 10);  assert(b == 10);  assert(c == 10);  // all the same
    return 0;
}
```

In this variant, the function-scope **static** object, `iLocal`, is created *before* `main` is entered. As with the previous example, the **static** object `iLocal` is again not destroyed until after `main` exits.

The rule for the initialization of **static** objects at function scope becomes more subtle when the functions themselves are recursive but is nonetheless well defined:

```
int fx(int i)  // self-recursive after creating function-static variable, dx
{
    static int dx = i;     // Create dx first.
    if (i) { fx(i - 1); }  // Recurse second.
    return dx;             // Return dx third.
}

int fy(int i)  // self-recursive before creating function-static variable,dy
{
    if (i) { fy(i - 1); }  // Recurse first.
    static int dy = i;     // Create dy second.
    return dy;             // Return dy third.
}

int main()
{
    int x = fx(5);  assert(x == 5);  // dx is initialized before recursion.
    int y = fy(5);  assert(y == 0);  // dy is initialized after recursion.
    return 0;
}
```

If the self-recursion takes place *after* the **static** variable is initialized (e.g., `fx` in the example above), then the **static** object (e.g., `dx`) is initialized on the *first* recursive call; if the recursion occurs *before* (e.g., `fy` in the example above), the initialization (e.g., of `dy`) occurs on the *last* recursive call.

As with all other initialization, control flow does not continue *past* the definition of a **static** object until after the initialization is complete, making recursive **static** initialization from within a single thread pointless:

```
int fz(int i)  // The behavior is undefined unless i is 0.
{
    static int dz = i ? fz(i - 1) : 0;  // Initialize recursively. (BAD IDEA)
    return dz;
}

int main()  // The program is ill-formed.
{
    int x = fz(5);  // broken (e.g., due to possible deadlock)
}
```

In the ill-fated example above, the second recursive call of `fz` to initialize `dz` has undefined behavior because the function is re-entered before it was able to complete the initialization of the **static** object; hence, control flow cannot continue to the **return** statement in `fz`. Given a likely implementation with a nonrecursive mutex or similar lock, the program is likely to deadlock since only one thread of control is allowed to grab the lock to enter the **critical section** that initializes a function-scope **static** object.[1]

Logger example

Let's now consider a more realistic, real-world example in which a single object — e.g., `localLogger` in the example below — is used widely throughout a program (see also *Use Cases — Meyers Singleton* on page 64):

```
Logger& getLogger()  // ubiquitous pattern commonly known as "Meyers Singleton"
{
    static Logger localLogger("log.txt");  // function-local static definition
    return localLogger;
}

int main()
{
    getLogger() << "hello";
        // OK, invokes Loggers constructor for the first (and only) time

    getLogger() << "world";
        // OK, uses the previously constructed Logger instance
}
```

(In a large-scale production environment, we would avoid evaluating any expression whose result is intended to be logged unless the logging level for that specific logging statement is enabled.[2]) All function-local **static** objects, such as `localLogger` in the example above, will be destroyed automatically only on normal program termination, either after the `main` function returns normally or when the `std::exit` function is called. The order of destruction of these objects will be the reverse of their order of construction. Objects that initialize concurrently have no guaranteed relationship on the order in which they are destroyed. Note that programs can terminate in several other ways, such as a call to `std::quick_exit`, `_Exit`, or `std::abort`, that explicitly do *not* destroy **static** storage-duration objects.

The destruction of function-scope **static** objects is and always has been guaranteed to be safe *provided* (1) no threads are running after returning from `main` and (2) function-scope **static** objects do not depend on each other during destruction; see *Potential Pitfalls — Depending on order-of-destruction of local objects after* `main` *returns* on page 71.

[1]Prior to standardization (see **?**, NEED ELLIS90 REFERENCE, section 6.7, p. 92), C++ allowed control to flow past a **static** function-scope variable even during a recursive call made as part of the initialization of that variable. This would result in the rest of such a function executing with a zero-initialized and possibly partially constructed local object. Even modern compilers, such as GCC with `-fno-threadsafe-statics`, allow turning off the locking and protection from concurrent initialization and retaining some of the pre-C++98 behavior. This optional behavior is, however, fraught with peril and unsupported in any standard version of C++.

[2]An eminently useful, full-featured logger, known as the ball logger, can be found in the ball package of the bal package group of Bloomberg's open-source BDE libraries (**?**, subdirectory `/groups/ball/bal`).

Multithreaded contexts

Historically, initialization of function-scope **static**-duration objects was not guaranteed to be safe in a **multithreading context** because it was subject to **data races** if the function was called concurrently from multiple threads. One common but unreliable pre-C++11 workaround was the *double-checked lock pattern*; see *Appendix: C++03 Double-Checked Lock Pattern* on page 74.

To illustrate how defects might have been introduced by multithreading *prior* to C++11, suppose that we have a simple type, `MyString`, that always allocates dynamic memory on construction:

```
#include <cstring>  // std::size_t, std::memcpy, std::strlen

class MyString
{
    char* d_string_p;  // pointer holding dynamically allocated memory address

public:
    MyString(const char* s)                                    // (1)
    {                                                          // (2)
        const std::size_t size = std::strlen(s) + 1;           // (3)
        d_string_p = static_cast<char*>(::operator new(size)); // (4)
        std::memcpy(d_string_p, s, size);                      // (5)
    }                                                          // (6)
};
```

Let's say that we want to create a **static** object of this `MyString` class in a function, `f`, that might be invoked concurrently from multiple threads:

```
void f()
{
    static const MyString s("example");  // function-scope, static-duration
    // ...
}
```

Let's now imagine that `f` is called from two separate threads concurrently, without having been called before. Suppose that the first thread gets through the `MyString` constructor, in the example above, up to *but not including* line (4) before it is suspended by the operating system. After that, the second thread — because there was no lock prior to C++11 — makes it all the way past line (6) before it too is suspended. When the operating system eventually resumes execution of the first thread, the dynamic allocation and assignment on line (4) **leaks** the memory for the previously constructed `MyString`. What's more, when the second destruction of the string eventually occurs (after exiting `main`), **undefined behavior** will inevitably result, if it hasn't already.

In practice, however, undefined behavior (prior to C++11) might have manifested even earlier. When the second thread re-uses the storage claimed by the object in the first thread, it effectively ends the lifetime of one **static** `S` object to start the lifetime of the other one. After that, any attempt to access the original `s` object would be undefined behavior, because its lifetime has ended, even though its destructor did not run. Hence, undefined behavior could manifest long before the second destructor is run at the end of the program.

As of C++11, a conforming compiler is now required to ensure that initialization of function-scope **static-duration** objects is performed safely even when the function is called concurrently from multiple threads. Importantly, however, this same guarantee is *not* extended for other static-duration objects such as those at file or namespace scope:

```
static S global(3);         // runtime-initialized, file-scope static

S& f1()                     // f1 is not thread safe.
{
    return global;          // global might be in any state.
}

S& f2()                     // f2 is thread safe.
{
    static S local(3);      // runtime-initialized, function-scope static
    return local;
}
```

Continuing our `logger` example from *Description — Logger example* on page 61, suppose that, to initialize a global facility, we are potentially calling a function, such as `getLogger`, concurrently from multiple threads using, say, the C++11 `std::thread` library utility. As an aside, the C++11 Standard Library provides copious utilities and abstractions related to multithreading. For starters, `std::thread` is a portable wrapper for a platform-specific thread handle provided by the operating system. When constructing an `std::thread` object with a **callable object** `functor`, a new thread invoking `functor` will be spawned. Note that `std::thread`'s destructor will *not* **join** the thread — it is safe to destroy an active `std::thread` object *only* if the `std::thread::join` member function has already been invoked. When the `std::thread` object's `join` member function is invoked, that function might need to block the caller until the *native* thread managed by the `std::thread` object being joined finishes its execution.

Such use prior to the C++11 thread-safety guarantees could, in principle, have led to a race condition during the initialization of `localLogger`, which was defined as a local **`static`** object in `getLogger`:

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

void useLogger() { getLogger() << "example"; }  // concurrently called function

int main()
{
    std::thread t0(&useLogger);
    std::thread t1(&useLogger);
        // Spawn two new threads, each of which invokes useLogger.

    // ...

    t0.join();  // Wait for t0 to complete execution.
    t1.join();  // Wait for t1 to complete execution.
```

```
    return 0;
}
```

As of C++11, the example above has no data races provided that `Logger::operator<<(const char*)` is designed properly for multithreaded use, even if the `Logger::Logger(const char* logFilePath)` constructor (i.e., the one used to configure the singleton instance of the logger) were not. That is to say, the implicit critical section that is guarded by the compiler includes evaluation of the initializer, which is why a recursive call to initialize a function-scope **static** variable is undefined behavior and is likely to result in deadlock; see *Description* on page 59. Such use of file-scope **static**s, however, is not foolproof; see *Potential Pitfalls — Depending on order-of-destruction of local objects after* `main` *returns* on page 71.

Use Cases

Meyers Singleton

The guarantees surrounding access across translation units to runtime initialized objects at file or namespace scope are few and dubious — especially when that access might occur prior to entering `main`. Consider a library component, `libcomp`, that defines a file-scope **static** singleton, `globalS`, that is initialized at run time:
[emcppsbatch=e7]

```
// libcomp.h:
#ifndef INCLUDED_LIBCOMP
#define INCLUDED_LIBCOMP

struct S { /*... */ };
S& getGlobalS();  // access to global singleton object of type S

#endif

// libcomp.cpp:
#include <libcomp.h>

static S globalS;
S& getGlobalS() { return globalS; }  // access into this translation unit
```

The interface in the `libcomp.h` file comprises the definition of `S` along with the declaration of an accessor function, `getGlobalS`. Any function wishing to access the singleton `globalS` object sequestered within the `libcomp.cpp` file would *presumably* do so safely via the global `getGlobalS()` accessor function. Now consider the `main.cpp` file in the example below, which implements `main` and also makes use of `globalS` prior to entering `main`:

```
// main.cpp:
#include <libcomp.h>  // getGlobalS()

bool globalInitFlag = getGlobalS().isInitialized();

#include <cassert>  // standard C assert macro
```

```
int main()
{
    assert(globalInitFlag);   // Error, or at least potentially so
    return 0;
}
```

Depending on the compiler or the link line, the call from `main.o`[3] into `libcomp.o` may occur and return *prior* to the initialization of `globalS`.[4] Nothing in the Standard says that **`static`** objects at file or namespace scope in separate translation units will be initialized just because a function located within that translation unit happens to be called.

An effective pattern for helping to ensure that a "global" object *is* initialized before it is used from a separate translation unit — especially when that use might occur prior to entering `main` — is simply to move the **`static`** object at file or namespace scope inside the scope of the function accessing it, making it a function-scope **`static`** instead:

```
S& getGlobalS()  // access into this translation unit
{
    static S globalS;  // singleton is now function-scope static
    return globalS;
}
```

Commonly known as the **Meyers Singleton** for the legendary author Scott Meyers who popularized it, this pattern ensures that the singleton object will *necessarily* be initialized on the first call to the accessor function that envelopes it, irrespective of when and where that call is made. Moreover, that singleton object is guaranteed to live past the end of `main`. The Meyers Singleton pattern also gives us a chance to catch and respond to exceptions thrown when constructing the **`static`** object, rather than immediately terminating, as would be the case if declared as a **`static`** global variable. Much more importantly, however, since C++11, the Meyers Singleton pattern automatically inherits the benefits of effortless race-free initialization of *reusable* program-wide singleton objects whose first invocation might be before `main` in some programs and after additional threads have already been started after entering `main` in other programs.

As discussed in *Description* on page 59, the augmentation of a thread-safety guarantee for the runtime initialization of function-scope **`static`** objects in C++11 minimizes the

[3]".o" is the object file extension on Unix-derived operating systems. The corresponding extension is ".obj" on Windows systems.

[4]For example, compiling the two files separately with GCC version 4.7.0 (c. 2017) and linking the `.o` files may generate an assertion error depending on the order of the `.o` files on the link line:

```
$ g++ main.o libcomp.o
$ ./a.exe              // Running this program produces no assertion failure.

$ g++ libcomp.o main.o
$ ./a.exe              // Running this program produces an assertion failure.

    assertion "globalInitFlag" failed: file "main.cpp", line 9, function: int main()
    Aborted (core dumped)
```

effort required to create a thread-safe singleton regardless of whether such safety guarantees turn out to be needed:

```
Logger& getLogger()
{
    static Logger logger("log.txt");
    return logger;
}
```

Note that, prior to C++11, the simple function-scope **static** implementation would not be safe if concurrent threads were vying to initialize the logger; see *Appendix: C++03 Double-Checked Lock Pattern* on page 74.

The Meyers Singleton is also seen in a slightly different form where the singleton type's constructor is made **private** to prevent more than just the one singleton object from being created:

```
class Logger
{
private:
    Logger(const char* logFilePath);  // Configure the singleton; the logger
    ~Logger();                         // suppresses copy construction too.

public:
    static Logger& getInstance()
    {
        static Logger localLogger("log.txt");
        return localLogger;
    }
};
```

This variant of the function-scope-**static** singleton pattern prevents users from manually creating rogue `Logger` objects; the only way to get one is to invoke the logger's **static** `Logger::getInstance()` member function:

```
void client()
{
    Logger::getInstance() << "Hi";  // OK
    Logger myLogger("myLog.txt");   // Error, Logger constructor is private.
}
```

This formulation of the singleton pattern, however, conflates the type of the singleton object with its use and purpose as a singleton. Once we find a use of a singleton object, finding another and perhaps even a third is not uncommon.

Consider, for example, an application on an early model of mobile phone where we want to refer to the phone's camera. Let's presume that a `Camera` class is a fairly involved and sophisticated mechanism. Initially we use the variant of the Meyers Singleton pattern where at most one `Camera` object can be present in the entire program. The next generation of the phone, however, turns out to have more than one camera, say, a front `Camera` and a back `Camera`. Our brittle, *ToasterToothbrush*-like[5] design doesn't admit the dual-singleton use of

[5]See **?**, section 0.3, pp. 13–20, specifically Figure 0-9, p. 16.

the same fundamental `Camera` type. A more finely factored solution would be to implement the `Camera` type separately and then to provide a thin wrapper, e.g., perhaps using the **strong-typedef idiom** (see Section 1.1."Inheriting Ctors" on page 181), corresponding to each singleton use:

```
class PrimaryCamera
{
    Camera& d_camera_r;
    PrimaryCamera(Camera& camera)  // implicit constructor
      : d_camera_r(camera) { }

public:
    static PrimaryCamera getInstance()
    {
        static Camera localCamera{/*...*/};
        return localCamera;
    }
};
```

With this design, adding a second and even a third singleton that is able to reuse the underlying `Camera` mechanism is facilitated.

Although this function-scope-**`static`** approach is vastly superior to the file-scope-**`static`** one, it does have its limitations. In particular, when one global facility object, such as a logger, is used in the destructor of another function-scope static object, the logger object may possibly have already been destroyed when it is used.[6] One approach is to construct the logger object by explicitly allocating it and never deleting it:

```
Logger& getLogger()
{
    static Logger& l = *new Logger("log.txt");  // dynamically allocated
    return l;  // Return a reference to the logger (on the heap).
}
```

A distinct advantage of this approach, once an object is created, it *never* goes away before the process ends. The disadvantage is that, for many classic and current profiling tools (e.g., *Purify*, *Coverity*), this intentionally never-freed dynamic allocation is indistinguishable from a **memory leak**. The ultimate workaround is to create the object itself in **`static`** memory, in an appropriately sized and aligned region of memory[7]:

```
#include <new>  // placement new

Logger& getLogger()
{
```

[6] An amusing workaround, the so-called *Phoenix Singleton*, is proposed in **?**, section 6.6, pp. 137–139.

[7] Note that any memory that `Logger` itself manages would still come from the global heap and be recognized as memory leaks. If available, we could leverage a polymorphic-allocator implementation such as `std::pmr` in C++17. We would first create a fixed-size array of memory having **`static`** storage duration. Then we would create a **`static`** memory-allocation mechanism (e.g., `std::pmr::monotonic_buffer_resource`). Next we would use placement **`new`** to construct the logger within the static memory pool using our static allocation mechanism and supply that same mechanism to the `Logger` object so that it could get all its internal memory from that static pool as well; see **?**.

```
    static std::aligned_storage<sizeof(Logger), alignof(Logger)>::type buffer;
    static Logger& l = *new(&buffer) Logger("log.txt");  // allocate in place
    return l;
}
```

In this final incarnation of a decidedly non-Meyers-Singleton pattern, we first reserve a block of memory of sufficient size and the correct alignment for `Logger` using `std::aligned_storage`. Next we use that storage in conjunction with placement **`new`** to create the logger directly in that static memory. Notice that this allocation is not from the dynamic store, so typical profiling tools will not track and will not provide a false warning when we fail to destroy this object at program termination time. Now we can return a reference to the logger object embedded safely in static memory knowing that it will be there for all eternity.

Finally, cyclic initialization dependencies among global objects are simply not accommodated, and if such is needed, the design is fatally flawed regardless; see *Potential Pitfalls — Relying on initialization order of* `static objects` on page 69.

Thread-safe initialization of global objects

Providing a global object (e.g., for logging or monitoring purposes) can sometimes be convenient for an application because such objects are accessible from any other part of the program without having to pass them as explicit arguments. Similarly to the example introduced in *Description — Logger example* on page 61, consider a `MetricsCollector` class whose purpose is to collect runtime performance metrics for the program:

```
class MetricsCollector  // used to collect runtime performance metrics
{
private:
    void startBenchmark(const std::string& name);
    void endBenchmark();

public:
    struct BenchmarkGuard { /* ... */ };
        // RAII guard that invokes startBenchmark on construction and
        // endBenchmark on destruction

    BenchmarkGuard benchmark(const std::string& name);
        // Create a BenchmarkGuard instance that will start a benchmark for
        // the specified name on construction and end the benchmark on
        // destruction.

    ~MetricsCollector();
        // Flush the collected metrics to disk on destruction.
};
```

Assuming that `startBenchmark` and `endBenchmark` are designed to avoid race conditions, all that's left to do is to create a function returning a local **`static`** object of type `MetricsCollector` (but see *Potential Pitfalls — Depending on order-of-destruction of local objects after* `main` *returns* on page 71):

```
MetricsCollector& getMetricsCollector()  // Meyers Singleton pattern again
{
    static MetricsCollector metricsCollector;  // function-local static object
    return metricsCollector;
}
```

The `getMetricsCollector` function in the code snippet above guarantees safe initialization of the `MetricsCollector` instance, initializing it exactly once on first invocation, but see *Potential Pitfalls — Depending on order-of-destruction of local objects after* `main` *returns* on page 71. Collecting metrics from any function scope, without requiring the function to accept an explicit `MetricsCollector` parameter, is now also possible. By creating an instance of the **RAII**-type `BenchmarkGuard`, the elapsed run time of the surrounding scope will be measured and collected:

```
void DataService::OnGetValueRequest(const std::string& key)
{
    MetricsCollector::BenchmarkGuard guard =
                        getMetricsCollector().benchmark("OnGetValueRequest");
    sendResponse(getValueFromKey(key));
}
```

Assuming the program terminates normally, `MetricsCollector`'s destructor will be executed automatically at the end of the program, flushing the collected data to disk.

Potential Pitfalls

Relying on initialization order of `static` objects

Despite C++11's guarantee that each individual function-scope **static** initialization will occur at most once, almost no guarantees are made on the order those initializations happen, which makes function-scope **static** objects that have interdependencies across translation units an abundant source of insidious errors. Objects that undergo **constant initialization** have no issue: such objects will never be accessible at run time before having their initial values. Objects that are not constant initialized[8] will instead be **zero initialized** until their constructors run, which itself might lead to conspicuous (or perhaps latent) undefined behavior. When used as global variables, function-scope **static** objects that do any form of dynamic allocation or maintain any form of invariants can be especially error prone. This problem is made even more acute when these objects are created and accessed before entering `main`.

As a demonstration of what can happen when we depend on the relative order of initialization of **static** variables at file or namespace scope used before `main`, consider the **cyclically dependent** pair of source files, `a.cpp` and `b.cpp`:

```
// a.cpp:
extern int setB(int);      // declaration (only) of setter in other TU
static int *p = new int;   // runtime initialization of file-scope static
int setA(int i)            // Initialize this static variable; then that one.
```

[8]C++20 added a new keyword, `constinit`, that can be placed on a variable declaration to *require* that the variable in question undergo constant initialization and thus can never be accessed at run time prior to the start of its lifetime.

```
{
    *p = i;                  // Populate this static-owned heap memory.
    setB(i);                 // Invoke setter to populate the other one.
    return 0;                // Return successful status.
}

// b.cpp:
static int *p = new int;   // runtime initialization of file-scope static
int setB(int i)            // Initialize this static variable.
{
    *p = i;                  // Populate this static-owned heap memory.
    return 0;                // Return successful status.
}

extern int setA(int);      // declaration (only) of setter in other TU
int x = setA(5);           // Initialize all of the static variables.
int main()                 // main program entry point
{
    return 0;                // Return successful status.
}
```

These two translation units will be initialized before `main` is entered in some order, but — regardless of that order — the program in the example above will likely wind up dereferencing a null pointer before entering `main`:

```
$ g++ a.cpp b.cpp main.cpp
$ ./a.exe
  Segmentation fault (core dumped)
```

Suppose we were to instead move the file-scope **`static`** pointers, corresponding to both `setA` and `setB`, inside their respective function bodies:

```
// a.cpp:
extern int setB(int);  // declaration (only) of setter in other TU
int setA(int i)        // Initialize this static variable; then that one.
{
    static int *p = new int;  // runtime init. of function-scope static
    *p = i;                   // Populate this static-owned heap memory.
    setB(i);                  // Invoke setter to populate the other one.
    return 0;                 // Return successful status.
}

// b.cpp: (same idea)
```

Now the program reliably executes without incident:

```
$ g++ a.cpp b.cpp main.cpp
$ ./a.exe
$
```

In other words, even though no order exists in which the translation units as a whole could have been initialized prior to entering `main` such that the *file*-scope **`static`** variables would be valid before they were used, by instead making them *function*-scope **static**, we are able to guarantee that each variable is itself initialized before it is used, regardless of translation-unit-initialization order.

Note that, had the variable initializations themselves been cyclic, the behavior would again be undefined and likely result in deadlock, even when implemented using a Meyers Singleton:

```
// a.cpp:
extern int setB(int);  // declaration (only) of setter in other TU
int setA(int i)        // Initialize this static variable to that one.
{
    static int *p =
            new int(setB(i)); // runtime initialization of function-scope static
    *p = i;                   // Populate this static-owned heap memory.
    setB(i);                  // Invoke setter to populate the other one.
    return 0;                 // Return successful status.
}

// b.cpp: (same idea as a.cpp)
```

In other words, avoid mutual recursion, as well as self-recursion, during the initialization of function-scope **static** objects.

Depending on order-of-destruction of local objects after `main` returns

Within any given translation unit, the relative order of initialization of objects at file or namespace scope having static storage duration is well defined and predictable. As soon as we have a way to reference an object outside of the current translation unit, before `main` is entered, we are at risk of using the object before it has been initialized. Provided the initialization itself is not cyclic in nature, we can make use of function-scope **static** objects (see *Use Cases — Meyers Singleton* on page 64) to ensure that no such uninitialized use occurs, even across translation units before `main` is entered. The relative order of destruction of such function-scope **static** variables — even when they reside within the same translation unit — is not generally known, and reliance on such order can easily lead to undefined behavior in practice.

This specific problem occurs when a **static** object at file, namespace, or function scope uses (or might use) in its destructor another **static** object that is either (1) at file or namespace scope and resides in a separate translation unit or (2) any other function-scope **static** object (i.e., including one in the same translation unit). For example, suppose we have implemented a low-level logging facility as a Meyers Singleton:

```
Logger& getLogger()
{
    static Logger local("log.txt");
    return local;
}
```

Now suppose we implement a higher-level file-manager type that depends on the function-scope **static** logger object:

```
struct FileManager
{
    FileManager()
    {
        getLogger() << "Starting up file manager...";
        // ...
    }

    ~FileManager()
    {
        getLogger() << "Shutting down file manager...";
        // ...
    }
};
```

Now, consider a Meyers Singleton implementation for `FileManager`:

```
FileManager& getFileManager()
{
    static FileManager fileManager;
    return fileManager;
}
```

Whether `getLogger` or `getFileManager` is called first doesn't really matter; if `getFileManager` is called first, the logger will be initialized as part of `FileManager`'s constructor. However, whether the `Logger` or `FileManager` object is destroyed first *is* important:

- If the `FileManager` object is destroyed prior to the `Logger` object, the program will have well-defined behavior.

- Otherwise, the program will have undefined behavior because the destructor of `FileManager` will invoke `getLogger`, which will now return a reference to a previously destroyed object.

As a practical matter, the constructor of the `FileManager` logs makes it virtually certain that the logger's function-local **`static`** will be initialized before that of the file manager; hence, since destruction occurs in reverse relative order of creation, the logger's function-local **`static`** will be destroyed after that of the file manager. But suppose that `FileManager` didn't always log at construction and was created before anything else logged. In that case, we have no reason to think that the logger would be around for the `FileManager` to log during its destruction after `main`.

In the case of low-level, widely used facilities, such as a logger, a conventional Meyers Singleton is counter-indicated. The two most common alternatives elucidated at the end of *Use Cases — Meyers Singleton* on page 64 involve never ending the lifetime of the mechanism at all. It is worth noting that truly global objects — such as `cout`, `cerr`, and `clog` — from the Standard `iostream` Library are typically not implemented using conventional methods and are in fact treated specially by the run time.

Annoyances

Overhead in single-threaded applications

A single-threaded application invoking a function containing a function-scope **static**-duration variable might have unnecessary synchronization overhead, such as an **atomic** load operation. For example, consider a program that invokes a free function, `getS`, returning a function-scope **static** object, `local`, of user-defined type, `S`, having a **user-provided** (inline) default constructor:

```
struct S  // user-defined type
{
    S() { }  // inline default constructor
};

S& getS()  // free function returning local object
{
    static S local;  // function-scope local object
    return local;
}

int main()
{
    getS();    // Initialize the file-scope static singleton.
    return 0;  // successful status
}
```

Although it is clearly visible to the compiler that `getS()` is invoked by only one thread, the generated assembly instructions might still contain atomic operations or other forms of synchronization and the call to `getS()` might not be generated inlined.[9]

See Also

None so far.

Further Reading

- ?
- ?

[9]Both GCC 10.x and Clang 10.x, using the `-Ofast` optimization level, generate assembly instructions for an **acquire/release memory barrier** and fail to inline the call to `getS`. Using `-fno-threadsafe-statics` reduces the number of operations performed considerably but still does not lead to the compilers' inlining of the function call. Both popular compilers will, however, reduce the program to just two x86 assembly instructions if the user-provided constructor of `S` is either removed or defaulted (see Section 1.1."Defaulted Functions" on page 34); doing so will turn `S` into a **trivially-constructible** type, implying that no code needs to be executed during initialization:

```
xor eax, eax  ; zero out 'eax' register
ret           ; return from 'main'
```

A sufficiently smart compiler might, however, not generate synchronization code in a single-threaded context or else provide a flag to control this behavior.

Appendix: C++03 Double-Checked Lock Pattern

Prior to the introduction of the function-scope **static** object initialization guarantees discussed in *Description* on page 59, preventing multiple initializations of **static** objects and use before initialization of those same objects was still needed. Wrapping access in a mutex was often a significant performance cost, so using the unreliable, double-checked lock pattern was often attempted to avoid the overhead:

```
Logger& getInstance()
{
    static Logger* volatile loggerPtr = 0;  // hack, used to simulate *atomics*

    if (!loggerPtr)  // Does the logger need to be initialized?
    {
        std::mutex m;
        std::lock_guard<std::mutex> guard(m);  // Lock the mutex.

        if (!loggerPtr)  // We are first, as the logger is still uninitialized.
        {
            static Logger logger("log.txt");
            loggerPtr = &logger;
        }
    }                    // Either way, the lock guard unlocks the mutex here.

    return *loggerPtr;
}
```

In this example, we are using a **volatile** pointer as a weak substitute for an atomic variable, but many implementations would provide nonportable extensions to support atomic types. In addition to being difficult to write, this decidedly complex workaround would often prove unreliable. The problem is that, even though the logic appears sound, architectural changes in widely used CPUs allowed for the CPU itself to optimize and reorder the sequence of instructions. Without additional support, the hardware would not see the dependency that the second test of `loggerPtr` has on the locking behavior of the mutex and would do the read of `loggedPtr` prior to acquiring the lock. By reordering the instructions or whatever, the hardware would then allow multiple threads to acquire the lock, thinking they are threads that need to initialize the **static** variable.

To solve this subtle issue, concurrency library authors are expected to issue ordering hints such as **fences** and **barriers**. A well-implemented threading library would provide atomics equivalent to the modern `std::atomic` that would issue the correct instructions when accessed and modified. The C++11 Standard makes the compiler aware of these concerns and provides portable *atomics* and support for threading that enables users to handle such issues correctly. The above `getInstance` function could be corrected by changing the type of `loggerPtr` to `std::atomic<Logger*>`. Prior to C++11, despite being complicated, the same function would reliably implement the Meyers Singleton in C++98 on contemporary hardware.

So the final recommended solution for portable thread-safe initialization in modern C++ is to simply let the compiler do the work and to use the simplest implementation that gets the job done, e.g., a Meyers Singleton (see *Use Cases — Meyers Singleton* on page 64):

```
Logger& getInstance()
{
    static Logger logger("log.txt");
    return logger;
}
```

Local/Unnamed Types as Template Arguments

C++11 allows function-scope and unnamed types to be used as template arguments.

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 `decltype` *keyword* (see Section 1.1."`decltype`" on page 27) 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 2.1."Lambdas" on page 307), as such types are both unnamed and local in typical usage:

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

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] ?

```
                {
                    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.

Use Cases

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:

```
// dijkstra.h:

#include <vector>  // std::vector

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 **?**.

[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 **?**, section 0.5, pp. 29–43.

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`:

```
#include <algorithm>  // std::copy
#include <iostream>   // std::ostream
#include <iterator>   // std::ostream_iterator
#include <set>        // std::set
#include <string>     // std::string
#include <vector>     // std::vector

struct City
{
    int         d_uniqueId;
    std::string d_name;
};
std::ostream& operator<<(std::ostream& stream,
                         const City&   object);
```

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 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.

As an aside, note that using a lambda (see Section 2.1."Lambdas" on page 307) in such scenario requires using `decltype` and passing the closure to the set's constructor:

```
void printUniqueCitiesOrderedByName(const std::vector<City>& cities)
{
    auto compare = [](const City& lhs, const City& rhs) {
        return lhs.d_name < rhs.d_name;
    };
    const std::set<City, decltype(compare)>
```

```
        tmp(cities.begin(), cities.end(), compare);
}
```

We discuss the topic of lambda expressions further in the very next section; see *Configuring algorithms via lambda expressions*.

Configuring algorithms via lambda expressions

Suppose we are representing a 3D environment using a *scene graph* and managing the graph's nodes via an `std::vector` of `SceneNode` objects (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). 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`[5]:

```
template <typename InputIterator, typename UnaryPredicate>
bool any_of(InputIterator first, InputIterator last, UnaryPredicate pred);
    // Return true if any of the elements in the range satisfies pred.
```

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

```
// C++03 (obsolete)
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 any_of(nodes.begin(), nodes.end(), IsNodeDirtyOrNew());
}
```

Because unnamed types can serve as arguments to this function template, we can also employ a lambda expression instead of a function object that would be required in C++03:

```
#include <algorithm> // 'std::any_of'
bool mustRecalculateGeometry(const std::vector<SceneNode>& nodes)
{
    return std::any_of(nodes.begin(),            // start of range
```

[5] ?

```
                         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.

Potential Pitfalls

Annoyances

See Also

- "`decltype`" (§1.1, p. 27) ♦ describes how developers may query the type of any expression or entity, including objects with unnamed types.
- "Lambdas" (§2.1, p. 307) ♦ provides strong practical motivation for the relaxations discussed here.

Further Reading

The long long (≥64 bits) Integral Type

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

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.

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`:

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

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. 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 84.

Use Cases

Storing values that won't safely fit in 32 bits

For many quantities that need to be represented as an integral value in a program, plain **int** is a natural choice. For example, this could be the case for years of a person's age, score in a ten-pin bowling game, or number of stories in a building. 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>`.

Sometimes the size of the virtual address space for the underlying architecture itself dictates how large an integer you will need. For example, on a 64-bit platform, specifying

[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.

the *distance* between two pointers into a contiguous array or the size of the array itself could 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.

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).[2] Since the average daily volume of even the most heavily traded stocks (roughly 70 million shares) appears to be well under the maximum value a signed **int** supports (more than 2 billion on our production platforms), we might at first think to write the function to return **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 potentially a pervasive defect enabling avenues of deliberate attack from outside sources.[3] What's more, the growth rate of some companies, especially technology startups, 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 volYMD(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 change 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, erroneous value. 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.

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
```

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*

[2] There are more than 3,200 listed symbols on the NYSE. Composite daily volume of NYSE-listed securities across all exchanges ranges from 3.5 to 6 billion shares, with a high reached in March 2020 of more than 9 billion shares.

[3] For an overview of integer overflow in C++, see **?**. For a more focused discussion of secure coding in CPP using CERT standards, see **?**, Ch. 5, "Integer Security," pp. 225–307.

Fundamental Integral Types on page 84), and (2) would reduce portability (see *Potential Pitfalls — Relying on the relative sizes of* **int**, **long**, *and* **long long** on page 83).

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 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.

Potential Pitfalls

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 84, the fundamental integral types have historically been a moving target. On older, 32-bit platforms, a **long** was often 32 bits and, **long long**, which was nonstandard prior to C++11, 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[4] property that **sizeof(int) < sizeof(long long)** avoids such portability issues since the relative sizes of the **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 (`intn_t`) or unsigned (`uintn_t`) integer aliases provided, since C++11, in `<cstdint>` and summarized here in Table 1.

[4]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 **?**, 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: Useful typedefs found in <cstdint> (since C++11)

Exact Size (optional)[a]	Fastest integral type having at least N bits	Smallest integer type having at least N bits
int8_t	int_fast8_t	int_least8_t
int16_t	int_fast16_t	int_least16_t
int32_t	int_fast32_t	int_least32_t
int64_t	int_fast64_t	int_least64_t
uint8_t	uint_fast8_t	uint_least8_t
uint16_t[a]	uint_fast16_t	uint_least16_t
uint32_t	uint_fast32_t	uint_least32_t
uint64_t	uint_fast64_t	uint_least64_t

[a] The compiler doesn't need to fabricate the exact-width type if the target platform doesn't support it.

Note: Also see `intmax_t`, the maximum width integer type, which might be different from all of the above.

See Also

- "Binary Literals" (§1.2, p. 130) ⧫ explains how programmers can specify binary constants directly in the source code; large binary values might only fit in a **long long** or even **unsigned long long**.
- "Digit Separators" (§1.2, p. 139) ⧫ describes visually separating digits of large **long long** literals.

Further Reading

- ?

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. As an example, 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.

During the late 1980s and into the 1990s, the word size of the machine and the size of an **int** were synonymous. 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 bits. 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-point calculations — enabled by type **`double`** and supported by floating-point coprocessors — became typical in the scientific community, machine architectures naturally evolved from 9-, 18-, and 36-bit words to the familiar 8-, 16-, 32-, and now 64-bit addressable 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!");
```

As cost of *main memory* was decreasing exponentially throughout the final two decades of the 20th century,[5] 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 for **general-purpose machines** (i.e., excluding **embedded systems**) — were long gone but a **`long int`** was still expected to be 32 bits on a 32-bit platform. Embedded systems are designed specifically to work with high-performance hardware, such as digital-signal processors (DSPs). 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.

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[6] 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 83 for some alternative ideas.

[5] 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.

[6] 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.

The [[noreturn]] Attribute

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

Description

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

```
[[noreturn]] void f()
{
    throw 1;
}
```

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 88. Using it on a pointer might have benefits for external tooling, code expressiveness, and future language evolution:

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

Use Cases

Better compiler diagnostics

Consider the task of creating an assertion handler that, when invoked, always aborts execution of the program after printing some useful information about the source of the assertion. Since this specific handler will never return because it unconditionally invokes a [[noreturn]]std::abort function, 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 normally:

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

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

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

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

Improved runtime performance

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

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

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

#include <new>    // std::bad_alloc

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

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:

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

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

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

Potential Pitfalls

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

Unlike many attributes, using [[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 return normally in such a build mode. 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, if a function is declared [[noreturn]] in some translation units within a program but not in others, that program is **ill-formed, no diagnostic required (IFNDR)**.

Misuse of [[noreturn]] on function pointers

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

```
void (*fp [[noreturn]])();  // no effect in 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;  // [[noreturn]] on fp is silently ignored.
}
```

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

Annoyances

See Also

- "Attribute Syntax" (Section 1.1, p. 10) ♦ `[[noreturn]]` is a built-in attribute that follows the general syntax and placement rules of C++ attributes.

Further Reading

- **?**
- **?**

The Null-Pointer-Literal Keyword

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

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:

```
#include <cstddef> // NULL
int data;  // nonmember 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);  // nonmember 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()
{
    char buf[] = "hello";
    g(buf);      // 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)
}
```

Use Cases

Improvement of type safety

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

As just one specific illustration of the added type safety provided by `nullptr`, imagine that the coding standards of a large software company historically required that values returned via output parameters (as opposed to a **`return`** statement) are always returned via pointer to a modifiable object. Functions that return via argument typically do so to reserve the function's return value to communicate status.[2] A function in this codebase might "zero" the output parameter's local pointer variable to indicate and ensure that nothing more is to be written. The function below illustrates three different ways of doing this:

```
int illustrativeFunction(int* x)   // pointer to modifiable integer
{
    // ...
    if (/*...*/)
    {
        x = 0;       // OK, Set pointer x to null address.
        x = NULL;    // OK, Set pointer x to null address.
        x = nullptr; // Bug, 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:

[1] 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 runtime behavior.

[2] See **?**, 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)."

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

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

Disambiguation of `(int)0` vs. `(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);      // Error, anything! (platform-defined)
    uglyLibraryFunction(0L);        // Error, ambiguous call (on all platforms)
    uglyLibraryFunction(0U);        // Error, ambiguous call (on all platforms)
}
```

`nullptr` is especially useful when such problematic overloads are unavoidable because it obviates explicit casts. (Note that 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.)

Overloading for a literal null pointer

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

```
#include  <cstddef> // std::nullptr_t
void f(int* v);          // (1)
void f(std::nullptr_t);  // (2)

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

Given the relative ease with which a `nullptr` can be converted to a typed pointer having the same null-address value, such overloads are dubious when used to control essential behavior. Nonetheless, we can envision such use to, say, aid in compile-time diagnostics when passing a null address would otherwise result in a runtime error (see Section 1.2."Deleted Functions" on page 46):

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

std::size_t strlen(std::nullptr_t) = delete;
    // Function is not defined but still participates in overload resolution.
```

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 of testing for a null pointer at run time.

Potential Pitfalls

Annoyances

See Also

Further Reading

- ?

The `override` Member-Function Specifier

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

Description

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

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

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

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

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

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

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

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

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

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

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

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

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

Use Cases

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

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

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

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

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

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

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

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

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

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

Potential Pitfalls

Lack of consistency across a codebase

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

Annoyances

See Also

Further Reading

- ?
- ?

Syntax for Unprocessed String Contents

Raw string literals obviate the need to escape each contained special character individually.

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 that outputs the following text into the standard output stream:

```
printf("Hello, %s%c\n", "World", '!');
```

In C++03, capturing the line of C code 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;  //        ^            ^  ^   ^      ^
               //            escape 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 98). 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 both vertical[1] and horizontal whitespace characters present

[1]To incorporate a newline character into a conventional string literal one must represent that newline using the escape sequence `\n`. Attempting to do so by actually entering a newline into the source (i.e., making the string literal span lines of source code) is an error.

in the source file as part of the string contents[2]:

```
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 101. 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\nline two\n    line three"
```

These same rules apply to both raw *wide* string literals and raw *Unicode* ones (see Section 1.1.“Unicode Literals” on page 117) that are introduced by placing their corresponding prefix before the `R` character:

```
const wchar_t  ws [] =  LR"(Raw\tWide\tLiteral)";
    // Represents "Raw\tWide\tLiteral", not "Raw    Wide    Literal".

const char     u8s[] = u8R"(\U0001F378)";  // Represents "\U0001F378", *not* "🍸"
const char16_t us [] =  uR"(\U0001F378)";  //       "         "          "     "
const char32_t Us [] =  UR"(\U0001F378)";  //       "         "          "     "
```

Collisions

Although unlikely, the data to be expressed within a string literal might itself contain the character sequence `)"` embedded within it:

```
#include <cstdio>   // printf

void emitHelloWorld()
{
    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:

[2] In this example, we assume that all trailing whitespace has been stripped since even trailing whitespace in a raw literal would be captured.

```
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/parenthesis 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
```

The value of `s6` above is equivalent to `"<-- Literal String Data -->"`. Every raw string literal comprises these syntactical elements, in order:

- an uppercase `R`
- the opening double quotes, `"`
- an optional arbitrary sequence of characters called the *delimiter* (e.g., `xyz`)
- an opening parenthesis, `(`
- the contents of the string
- a 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:

[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.

```
const char s8a[] = R"("---R"( Raw literals are not recursive!)" ---")";
    //                                                            ^~
    // Error, decrement of read-only location
```

In fact, it could turn out that a program with an unexpectedly terminated *raw* string literal could still be valid and compile quietly:

```
void emitPith()
{
    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.

Use Cases

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.

[4]Such as when you want to copy the contents of the string literal into an online regular-expression validation tool.

Finally, raw string literals can also be helpful for whitespace-sensitive languages, such as Python (but see *Potential Pitfalls — Encoding of new lines and whitespace* on page 102):

```
const char* testPythonInterpreterPrint = R"(def test():
    print("test printing from Python")
)";
```

Potential Pitfalls

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 emitPythonEvaluator0(const char* expression)
{
    std::cout << R"(
        def evaluate():
            print("Evaluating...")
            return )" << expression << '\n';
}
```

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 Python 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 — Python code can be expressed with a single *raw* string literal, but visualizing the final output requires some effort:

```
void emitPythonEvaluator1(const char* expression)
{
    std::cout << R"(def evaluate():
    print("Evaluating...")
    return )" << expression << '\n';
}
```

[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.

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 << '\n';
}
```

Encoding of new lines and whitespace

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 platform-specific encoding of the source file (e.g., `\r\n`). 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.

In a similar fashion, the type of whitespace 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.)";

    assert(output == expected);
}
```

The unit test might pass for years, until, for instance, the company's indentation style changes from tabulation characters to spaces, leading the `expected` string to contain spaces instead of tabs, and thus test failures.[7]

Annoyances

None so far

[6]**?**

[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.

See Also

None so far

Further Reading

None so far

Compile-Time Assertions

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

Description

Assumptions are inherent in every program, whether we explicitly document them or not. A common way of validating certain assumptions at run time 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 run time 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.

Syntax and semantics

We can use **static assertion declarations** to conditionally trigger controlled compilation failures depending on the truthfulness 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 111) **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 runtime assertions to run faster with them enabled than disabled. For example, asserting that a pointer is not null enables 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 nonconstant expression to a `static_assert` is itself a compile-time error:

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

Evaluation of static assertions in templates

The C++ Standard does not explicitly specify at precisely what point during the compilation process the expressions tested by static assertion declarations are evaluated. In particular, when used within the body of a template, the expression tested by 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[2] 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 located within the body of a class or function template and depends on at least one template parameter is almost always bypassed during its initial parse since the assertion predicate might evaluate to true or false depending on 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 108. 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]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 (IFNDR)**, 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[3]:

```
void h1()
{
    int a[!sizeof(int) - 1];  // Error, same as int a[-1];
}

template <typename T>
void h2()
{
    int a[!sizeof(int) - 1];  // Error, always reported
}

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

Both `f1` and `h1` are ill-formed, nontemplate 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 108.

Use Cases

Verifying assumptions about the target platform

Some programs rely on specific properties of the native types provided by their target platform. Static assertions can help ensure portability and prevent such programs from

[3]The formula used — `int a[-1];` — leads to -1, not 0, to avoid a nonconforming extension to GCC that allows `a[0]`.

being compiled into a malfunctioning binary on 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 ensure that the assumption regarding the size of **int** is valid, and also serve as documentation for readers:

```
#include <climits>  // CHAR_BIT

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

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

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

Preventing misuse of class and function templates

Static assertions are often used in practice to constrain class or function templates to prevent their being instantiated with unsupported types. If a type is not syntactically compatible with the template, static assertions provide clear customized error messages that replace compiler-issued diagnostics, which are often absurdly long and notoriously hard-to-read. More critically, static assertions actively avoid erroneous runtime behavior.

As an example, consider the `SmallObjectBuffer<N>` class templates, which provide storage, aligned properly using `alignas` (see Section 2.1."`alignas`" on page 154), for arbitrary objects whose size does not exceed `N`[4]:

```
#include <cstddef> // std::size_t, std::max_align_t
#include <new>     // placement new

template <std::size_t N>
class SmallObjectBuffer
{
private:
    alignas(std::max_align_t) 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:

[4]A `SmallObjectBuffer` is similar to C++17's `std::any` (**?**) 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.

```
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.");
    // Destroy existing object, if any; store how to destroy this new object of
    // type T later; then...
    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 <climits>      // CHAR_BIT
#include <type_traits>  // std::is_integral, std::is_unsigned

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

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

Potential Pitfalls

Static assertions in templates can trigger unintended compilation failures

As mentioned in the description, any program containing a template for which no valid specialization can be generated is **IFNDR**. 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>)  // (2a)
{
    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 (2a) 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>)  // (2b)
{
    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>)  // (2c)
{
    static_assert(0 == sizeof(T), "T must be serializable.");  // OK
         // not too obviously ill-formed: compile-time error when instantiated
}
```

Using this sort of obfuscation is not guaranteed to be either portable or future-proof.

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**, and **SFINAE**-based approaches should be used instead.

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**:

```
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 above in conjunction with just one of its two possible specializations conditionally exposes the `Ok` type alias *only* if the provided boolean template parameter evaluates to **`true`**. (Otherwise, by default, it does not.) C++11 provides a library function, `std::enable_if`, that more directly addresses this use case.[5]

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

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

This general technique of pairing template specializations is used widely in modern C++ programming. For another, often more convenient way of constraining overloads using **expression SFINAE**, see Section 1.1."Trailing Return" on page 112.

Annoyances

Mandatory string literal

Many compilation failures caused by static assertions are self-explanatory since the offending line (which necessarily contains the predicate code) is displayed as part of the compiler diagnostic. In those situations, the message required[6] 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, "");
```

There is no universal consensus as to the "parity" of the user-supplied error message. Should it restate the asserted condition, or should it state what went amiss?

```
static_assert(0 < x, "x is negative");
   // misleading when 0 == x
```

See Also

- "Trailing Return" (Section 1.1, p. 112) ♦ Enabling expression SFINAE directly as part of a function's declaration allows simple and fine-grained control over overload resolution.

Further Reading

- ?

[5] **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.

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

Trailing Function Return Types

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

Description

C++11 offers an alternative function-declaration syntax in which the return type of a function is located to the right of its **signature** (name, parameters, and qualifiers), offset by the arrow token (`->`); the function itself is introduced by the keyword **`auto`**, which acts as a type placeholder:

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

When using the alternative, trailing-**`return`**-type syntax, any **`const`**, **`volatile`**, and reference qualifiers (see Section 3.1."Ref-Qualifiers" on page 371) are placed to the left of the `->` `*<`**`return`**`-type>*`, and any contextual keywords, such as `override` and `final` (see Section 1.1."`override`" on page 94 and Section 3.1."`final`" on page 341), are placed to its right:

```
struct Base
{
    virtual int e() const;     // const qualifier
    virtual int f() volatile;  // volitile qualfier
    virtual int g() &;         // *lvalue*-reference qualifier
    virtual int h() &&;        // *rvalue*-reference qualifier
};

struct Derived : Base
{
    auto e() const    -> int override;  //  override contextual keyword
    auto f() volatile -> int final;     //  final          "        "
    auto g() &        -> int override;  //  override       "        "
    auto h() &&       -> int final;     //  final          "        "
};
```

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[1] 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
};

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`**[2]:

```
// [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:

[1]A **`static`** member function of a **`struct`** can be a viable alternative implementation to a free function declared within a namespace; see **?**, 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).

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

```
#include <utility>  // declval

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`[3] to express our intent with the classic return-type syntax.

Use Cases

Function template whose return type depends on a parameter type

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

```
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 1.1."`decltype`" on page 27).

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 qualified. By contrast, when the return type follows the qualified name of the function, the return type is looked up in the same scope in which the function was first declared, just like its parameter types would. Avoiding redundant qualification of the return type can be beneficial, especially when the qualifying name is long.

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

[3] **?**

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

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

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

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

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

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

Improving readability of declarations involving function pointers

Declarations of functions returning a pointer to either a function, a member function, or a data member are notoriously hard to parse — even for seasoned programmers. As an example, consider a function called `getOperation` that takes a `kind` of enumerated `Operation` as its argument 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 such functions becomes fairly straightforward. 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 via **`typedef`** or, as of C++11, **`using`**.

Potential Pitfalls

Annoyances

See Also

- "`decltype`" (Section 1.1, p. 27) ♦ Function declarations may use `decltype` either in conjunction with, or as an alternative to, trailing return types.

- "Deduced Return Type" (Section 3.2, p. 373) ⧫ Leaving the return type to deduction shares syntactical similarities with trailing return types but brings with it significant pitfalls when migrating from C++11 to C++14.

Further Reading

- **?**

Unicode String Literals

C++11 introduces a portable mechanism for ensuring that a literal is encoded as UTF-8, UTF-16, or UTF-32.

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 character literal `'X'` has the value `0x58`.

Table 1 illustrates three new kinds of *Unicode*-compliant *string literals*, each delineating the precise encoding of each character.

Table 1: Three new Unicode-compliant literal strings

Encoding	Syntax	Underlying Type
UTF-8	`u8"Hello"`	**`char`**[a]
UTF-16	`u"Hello"`	`char16_t`
UTF-32	`U"Hello"`	`char32_t`

[a] `char8_t` in C++20

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:

```
#include <cstdio>  // std::puts
void f()
{
    std::puts(u8"\U0001F378"); // Unicode code point in a UTF8 encoded literal
}
```

[1] In fact, C++ still fully supports platforms using EBCDIC, a rarely used alternative encoding to ASCII, as their primary text encoding.

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.

Use Cases

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& utf8Message)
{
    Packet data;
    data << u8"Message from the server: '" << utf8Message << 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.

Potential Pitfalls

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 incorrectly assume that `u8"🍸"` is a portable way of embedding a string literal representing the cocktail emoji in a C++ program. The representation of the string literal, however, 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"`).

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:

```
#include <cassert>  // standard C assert macro
```

[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.

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

void f()
{
    std::string cocktail(u8"\U0001F378"); // big character (!)
    assert(cocktail.size() == 1);         // assertion failure (!)
}
```

Even though the cocktail emoji is a *single* code point, `std::string::size` returns the number of code units (bytes) 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.

Problematic treatment of UTF-8 in the type system

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 distinct character types (`char16_t` and `char32_t`). This precludes providing distinct behavior for UTF-8 encoded strings using function overloading or template specialization because they are indistinguishable from strings having the encoding of the execution character set. 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
}
```

Annoyances

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.

See Also

None so far

Further Reading

None so far

Type/Template Aliases (Extended `typedef`)

Alias declarations and alias templates provide an expanded use of the **using** keyword to introduce aliases for types and templates, thus providing a more general alternative to **typedef**.

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. 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-function*, *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 introduced via **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 full 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 similar functionality can be achieved in C++03, it suppresses type deduction and requires additional boilerplate code at both the point of definition and the call site:

```
// C++03 (obsolete)
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;
```

Use Cases

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, and the name of the alias is clearly specified after the **using** keyword. To make the

`CompletionCallback` alias read left-to-right, a trailing return (see Section 1.1.“Trailing Return” on page 112) 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`**.”

Binding arguments to template parameters

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 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 reduce code repetition and enable the programmer to consistently replace `std::map` to another container (e.g., `std::unordered_map`[1]) 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]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 **?**.

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`):

```
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. Furthermore, in the generic context, it has to be prepended with the **`typename`** keyword::

```
template <typename T>void f()
{
    T t;
    typename AddPointer<T>::Type p = t;
}
```

The syntactical overhead of `AddPointer` can be removed by creating an alias template for its nested type alias, such as `AddPointer_t`:

```
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;
}
```

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
```

Potential Pitfalls

Annoyances

See Also

- "Inheriting Ctors" (§2.1, p. 181) ♦ provides another meaning for the **using** keyword to allow base-class constructors to be invoked as part of the derived class.
- "Trailing Return" (§1.1, p. 112) ♦ provides an alternative syntax for function declaration, which can help improve readability in type aliases and alias templates involving function types.

Further Reading

Aggregates Having Default Member Initializers

C++14 enables the use of **aggregate initialization** with classes employing default member initializers (see Section 1.1."Default Member Init" on page 198).

Description

Prior to C++14, classes that used default member initializers (see Section 1.1."Default Member Init" on page 198) — 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
```

Note that since C++11, direct-list-initialization may be used to perform aggregate initialization (see Section 2.1."Braced Init" on page 178):

```
A a{100, true};  // OK in both C++11 and C++14 but not in C++03
```

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, and `s1` is created using aggregate initialization for only the first data member (and the second is set via its default member initializer).

Use Cases

Configuration `struct`s

Aggregates in conjunction with default member initializers (see Section 1.1."Default Member Init" on page 198) 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.
    }
}

// ...
```

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.
```

Annoyances

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 (see Section 1.1."Default Member Init" on page 198). 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.

See Also

- "Default Member Init" (§1.1, p. 198) ⧫ allows developers to provide a default initializer for a data member directly in the definition of a class.
- "Braced Init" (§2.1, p. 178) ⧫ introduces a syntactically similar feature for initializing objects in a uniform manner.

Further Reading

Binary Literals: The `0b` Prefix

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

Description

A binary literal is an integral value represented in code in a binary numeral system. A binary literal consists of a `0b` or `0B` prefix followed by a nonempty sequence of binary digits, namely, `0` and `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 is by default an **`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`**} in which it will fit. 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 *Description* on page 130. If neither of those is applicable, the compiler may use implementation-defined extended integer types such as `__int128` to represent the literal if it fits — otherwise the program is ill-formed:

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

[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 4.3.0, released in March 2008; for more details, see https://gcc.gnu.org/gcc-4.3/.

```
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.
```

(Purely for convenience of exposition, we have employed the C++11 **`auto`** feature to conveniently capture the type implied by the literal itself; see Section 2.1."**`auto`** Variables" on page 177.) 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( 0b0111...[ 24 1-bits]...1111       //   signed
           != -0b0111...[ 24 1-bits]...1111, ""); //   signed

static_assert( 0b1000...[ 24 0-bits]...0000       // unsigned
           != -0b1000...[ 24 0-bits]...0000, ""); // unsigned
```

Use Cases

Bit masking and bitwise operations

Prior to the introduction of binary literals, hexadecimal and 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, using 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;
}
```

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;

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

    // ...
};
```

Note that the choice of using a nested classic **enum** was deliberate; see Section 2.1."**enum class**" on page 199.

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
};
```

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

Potential Pitfalls

Annoyances

See Also

- "Digit Separators" (§1.2, p. 139) ♦ explains grouping digits visually to make long binary literals much more readable.

Further Reading

- **?**

The [[deprecated]] Attribute

The `[[deprecated]]` attribute discourages the use of a decorated **entity**, typically via the emission of a compiler warning.

Description

The standard `[[deprecated]]` attribute is used to portably indicate that a particular entity is no longer recommended and to actively discourage its use. Such deprecation typically follows the introduction of alternative constructs that are superior to the original one, providing time for clients to migrate to them (*asynchronously*[1]) before the deprecated one is removed in some subsequent release. 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 appertains. 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 supply a **string literal** as an argument to the `[[deprecated]]` attribute (e.g., `[[deprecated("message")]]`) to inform human users regarding the reason for the deprecation:

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

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

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

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

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

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

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

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

Use Cases

Discouraging use of an obsolete or unsafe entity

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

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

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

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

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

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

By using `[[deprecated]]` as shown 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 having their code broken by the removal of the old solution. When clients are notified of the deprecation (thanks to a compiler diagnostic), they can schedule time to rewrite their applications to consume the new interface.[5]

Potential Pitfalls

Interaction with treating warnings as errors

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

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

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

Annoyances

None so far

See Also

None so far

Further Reading

None so far

The Digit Separator: '

A digit separator is a single-character token (`'`) that can appear as part of a numeric literal without altering its value.

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 = 500'000L;                   // same as 500000L
long long    k = 9'223'372'036'854'775'807;  // same as 9223372036854775807
float        f = 3.14159'26535f;             // same as 3.1415926535f
double       d = 3.14159'26535'89793;        // same as 3.141592653589793
long double  e = 20'812.80745'23204;         // same as 20812.8074523204
int        hex = 0x8C'25'00'F9;              // same as 0x8C2500F9
int        oct = 044'73'26;                  // same as 0447326
int        bin = 0b1001'0110'1010'0111;      // same as 0b1001011010100111
```

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:

```
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
```

Although the leading `0x` and `0b` prefixes for hexadecimal and binary literals, respectively, are not considered part of the *numeric* part of the literal, a leading `0` in an octal literal is. 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[1]:

```
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] 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 141.

Use Cases

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.

Table 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 to group bits in octets (**bytes**) or quartets (**nibbles**), as shown in Table 2. In addition, using a binary literal with digits grouped in triplets instead of an octal literal to represent UNIX file permissions might improve code readability — e.g., `0b111'101'101` instead of `0755`.

Table 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`

Potential Pitfalls

See Also

- "Binary Literals" (§1.2, p. 130) ♦ represents a binary constant for which digit separators are commonly used to group bits in octets (bytes) or quartets (nibbles)

Further Reading

- William Kahan. "Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic," **?**
- *IEEE Standard for Floating-Point Arithmetic*, **?**

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.[2,3]

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 has 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 notable exception is Microsoft Visual C++ where **`long double`** is a distinct type whose representation is identical to **`double`**.[5] A table summarizing typical precisions for various IEEE-conforming floating-point types is presented for convenient reference in Table 3. The actual bounds on a given platform can be found using the standard `std::numeric_limits` class template found in `<limits>`.

[2] **?**, section 6.8.2, "Fundamental types [basic.fundamental]," pp. 73–75; section 17.3.5.2, "`numeric_limits` members [numeric.limits.members]," pp. 513–516; and section 17.3.7, "Header `<cfloat>` synopsis [cfloat.syn]," p. 519

[3] **?**, section 7.7, "Characteristics of floating types `<float.h>`," p. 157

[4] **?**

[5] **?**

Table 3: Available precisions for various IEEE-754 floating-point types

Name	Common Name	Significant Bits[a]	Decimal Digits	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` for normalized numbers, and 0 for denormalized ones and, hence, is not stored explicitly. This leaves 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 π, 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[6]:

```
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. Note, however, that because the compiler is not required to use the floating-point arithmetic of the target platform *during compilation*, this approach might not be applicable for a cross-compilation scenario.

One final useful tidbit pertains to the safe (lossless) conversion between binary and

[6]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));
```

decimal floating-point representations; note that "Single" (below) corresponds to a single-precision IEEE-754-conforming (32-bit) **`float`**[7]:

> 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 double`**, are 15–17 and 33–36, respectively.

[7] **?**, section "Representable Numbers," p. 4

Templated Variable Declarations/Definitions

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.

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., `exampleOf`):

```
template <typename T> T exampleOf;  // variable template defined at file scope
```

Like any other kind of template, a variable template can be instantiated explicitly by providing an appropriate number of type or non-type arguments:

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

void initializeExampleValues()
{
    exampleOf<int>   = -1;
    exampleOf<char>  = 'a';
    exampleOf<float> = 12.3f;
}

void printExampleValues()
{
    initializeExampleValues();
    std::cout << "int = "   << exampleOf<int>   << "; "
              << "char = "  << exampleOf<char>  << "; "
              << "float = " << exampleOf<float> << ';';

    // outputs "int = -1; char = a; float = 12.3;"
}
```

In the example above, the type of each instantiated variable is the same as its template parameter, but this matching is not required. For example, the type might be the same for all instantiated variables or derived from its parameters, such as by adding **`const`** qualification:

```
#include <type_traits>  // std::is_floating_point
#include <cassert>      // standard C assert macro

template <typename T>
const bool sane_for_pi = std::is_floating_point<T>::value;  // same type

template <typename T> const T pi(3.1415926535897932385);    // distinct types

void testPi()
{
```

```
    assert(!sane_for_pi<bool>);
    assert(!sane_for_pi<int>);

    assert( sane_for_pi<float>);
    assert( sane_for_pi<double>);
    assert( sane_for_pi<long double>);

    const float       pi_as_float       = 3.1415927;
    const double      pi_as_double      = 3.141592653589793;
    const long double pi_as_long_double = 3.1415926535897932385;

    assert(pi<float>       == pi_as_float);
    assert(pi<double>      == pi_as_double);
    assert(pi<long double> == pi_as_long_double);
}
```

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> static 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;        // Error
    template <typename T> static T vt8; // Error

    vt1<bool> = true;                   // OK, to use them
    N::vt3<bool> = true;
    N::vt4<bool> = true;
    S::vt6<bool> = true;
}
```

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 149):

```
namespace {
template <int N>
const int sum = N + sum<N - 1>;    // recursive general template

template <> const int sum<0> = 0;  // base case specialization
}  // close unnamed namespace

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 while variable templates do not add new functionality, they significantly reduce the boilerplate associated with achieving the same goals without them. For example, compare the definition of `pi` above with the pre-C++14 code:

```
// C++03 (obsolete)
#include <cassert>  // standard C assert macro

template <typename T>
struct Pi {
    static const T value;
};

template <typename T>
const T Pi<T>::value(3.1415926535897932385);  // separate definition

void testCpp03Pi()
{
    const float       piAsFloat      = 3.1415927;
    const double      piAsDouble     = 3.141592653589793;
    const long double piAsLongDouble = 3.1415926535897932385;

    // additional boilerplate on use (::value)
    assert(Pi<float>::value       == piAsFloat);
    assert(Pi<double>::value      == piAsDouble);
    assert(Pi<long double>::value == piAsLongDouble);
}
```

Use Cases

Parameterized constants

A common effective use of variable templates is in the definition of type-parameterized constants. As discussed in *Description* on page 144, the mathematical constant π 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**)[1]:

```
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 in place of **const** to guarantee that the floating-point `pi` is a compile-time constant that 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));
}
```

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** 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.[2]

---

[1]For portability, a floating-point literal value of π 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 1.2."Digit Separators" on page 139.

[2]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
bool dc1 = std::is_default_constructible<T>::value;

// C++17
bool dc2 = std::is_default_constructible_v<T>;
```

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.");

    // ...

    return { /*..*/ };
}
```

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:

```
// 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
```

This delay is a consequence of the train release model of the Standard: Thoughtful application of the new feature throughout the vast Standard Library required significant effort that could not be completed before the next release date for the Standard and thus was delayed until C++17.

```
#include <map>                      // std::map (not JSON serializable)

void client()
{
    JsonObject jsonObj0 = serializeToJson(CompanyData());          // OK
    JsonObject jsonObj1 = serializeToJson(std::map<int, char>());  // 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.

Potential Pitfalls

Recursive variable template initializations require `const` or `constexpr`

Instantiating variable templates that are defined recursively might have a subtle issue that could produce different results[3] despite having no undefined behavior:

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

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
    std::cout << fib<6> << '\n';  // 8 expected

    return 0;
}
```

The root cause of this instability is that the relative order of the initialization of the recursively generated variable template instantiations is not guaranteed because they are not defined explicitly *within the same translation unit*. Therefore, a similar issue might have occurred in C++03 using **`static`** members of a **`struct`**:

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

template <int N> struct Fib
{
    static int value;                           // BAD IDEA: not const
};

template <> struct Fib<2> { static int value; };  // BAD IDEA: not const
```

[3]For example, GCC version 4.7.0 (2017) produces the expected results whereas Clang version 10.x (2020) produces 1, 3, and 4, respectively.

```
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;
};
```

However, this is not an issue when using **enum**s due to enumerators always being compile-time constants:

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

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;
};
```

For integral variable templates, this issue can be resolved simply by adding a **const** qualifier because the C++ Standard requires that any integral variable declared as **const** and initialized with a compile-time constant is itself to be treated as a compile-time constant within the translation unit.

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

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, does not consume memory in the **static data space**, and would also be applicable to non-integral constants.

Annoyances

Variable templates do not support template template parameters

Although a class or function template can accept a **template template class parameter**, no equivalent construct is available for variable templates[4]:

```
template <typename T> T vt(5);

template <template <typename> class>
struct S { };

S<vt> s1;  // Error
```

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
```

See Also

- "`constexpr` Variables" (§2.1, p. 180) ⧫ Conditionally safe C++11 feature providing an alternative to **const** template variables that can reduce unnecessary consumption of the static data space.

Further Reading

None so far

[4]Mateusz Pusz has proposed for C++23 a way to increase consistency between variable templates and class templates when used as template template parameters; see **?**.

Chapter 2

Conditionally Safe Features

Intro text should be here.

The `alignas` Decorator

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

Description

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

Restricting the alignment of a particular object

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

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

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

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

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

1. Be either zero or a non-negative integral power of two of type `std::size_t` (0, 1, 2, 4, 8, 16…).

2. Be at least the minimum alignment[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 162.

[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
}
```

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 159. Note that, in the case of `i` in the first code snippet on page 154, 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).
```

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
```

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.

Use Cases

Creating a sufficiently aligned object buffer

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

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

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

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

void f()
{
    // ...

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

    // ...

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

    // ...
}
```

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

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

    alignas(std::complex<long double>) char objectBuffer[
```

```
                                  sizeof(std::complex<long double>)];  // GOOD IDEA

    // ...

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

    // ...
}
```

Ensuring proper alignment for architecture-specific instructions

Architecture-specific instructions or compiler intrinsics might require the data they act on to have a specific alignment. One example of such intrinsics is the *Streaming SIMD Extensions (SSE)*[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);
}
```

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 164.

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]

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 162.

Potential Pitfalls

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]

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**.

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 164.

See Also

- Section 2.2, "`alignof`" — Safe C++11 feature that inspects the alignment of a given type
- Section 1.1, "Attribute Syntax" — Safe C++11 feature that shows how other attributes (following the conventional attribute notation) are used to annotate source code, improve error diagnostics, and implicitly code generation

Further Reading

None so far

Appendix

Natural Alignment

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

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,

---

[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**),

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 provides a summary of typical physical parameters found in modern computers today.

Table 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	~10μs–50μs
Solid-State Disc (SSD)	256Gb … 16Tb	~25μs–100μs
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**.

The (Compile-Time) `alignof` Operator

The keyword `alignof` serves as a compile-time operator used to query the **alignment requirements** of a type on the current platform.

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 175). 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)
```

`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, "");  //    "         "
```

`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

[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.

[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:

will (by default) avoid nonessential padding because any extra padding would be wasteful of (e.g., cache) memory[3]:

```
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
```

Use Cases

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 platform. For example:

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

```
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 154) 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
```

```
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
```

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 2.14.

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).
```

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 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 naturally 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 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.

Annoyances

`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 27) 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`** Variables" on page 177) and generic lambdas (see "*Generic* Lambdas" on page 320). 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.

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 175).

Further Reading

None so far

Variables of Automatically Deduced Type

placeholder text.........

Brace-Initialization Syntax: `{}`

placeholder text.........

Compile-Time Evaluatable Functions

placeholder text.........

Compile-Time Accessible Variables

placeholder text...........

Inheriting Base Class 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.

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 **using** declarations of member functions, the nominated base class's constructors will be considered 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 Section 2.1."Default Member Init" on page 198).

```
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 (implicit) value 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:

```
#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*).

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.[1]
```

```
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 D0::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 Section 1.1."Deleted Functions" on page 46) 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 **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 194.

Inheriting constructors having the same **signature** from multiple base classes lead to ambiguity errors:

```
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;
    using B1B::B1B;
};
```

[1]Note that we use braced initialization (see Section 2.1."Braced Init" on page 178) 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**.

```
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 constructor 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 Section 3.1."`noexcept` Specifier" on page 370 and Section 2.1."`constexpr` Functions" on page 179. 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 B3
{
    B3()         { std::cout << "B3()\n"; }
    B3(int)      { std::cout << "B3(int)\n"; }
    B3(int, int) { std::cout << "B3(int, int)\n"; }
};

struct D3 : B3
{
    using B3::B3;
    D3(int) { std::cout << "D3(int)\n"; }
};

D3 d;        // prints "B3()"
D3 e(0);     // prints "D3(int)" --- The derived constructor is invoked.
D3 f(0, 0);  // prints "B3(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 Section 1.1."Defaulted Functions" on page 34), or **delete** it (see Section 1.1."Deleted Functions" on page 46).

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 base classes, each of which
struct B1B { B1B(int); };  // provides a conversion constructor from an int.

struct D3 : B1A, B1B
{
    using B1A::B1A;  // Inherit the int constructor from base class B1A.
    using B1B::B1B;  // Inherit the int constructor from base class B1B.

    D3(int i) : B1A(i), B1B(i) { }  // User-declare int conversion constructor
};                                  // that delegates to bases.

D3 d3(0);  // OK, calls D3(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* on page 181 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 <typename T>
struct S : T  // The base type, T, is a *dependent type*.
{
    using T::T;  // inheriting constructors (generically) from a dependent type
};

#include <string>   // std::string
#include <vector>   // std::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 derives publicly from its template argument, `T`. Then, when creating an object of type `S` parameterized by `std::string`, we

[2]A decidedly more complex alternative affording a different set of tradeoffs would involve variadic template constructors (see Section 2.1."Variadic Templates" on page 319) having forwarding references (see Section 2.1."Forwarding References" on page 283) 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 194.

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 189.

Use Cases

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 be initialized if by a *defaulted* default constructor, 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 be either default-initialized or have its value set using member initializers (see Section 2.1."Default Member Init" on page 198). 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 non-virtual member functions.

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 <cassert>
#include <vector>

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.
```

[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. When we have access to the source, and alternative solution would be to use conditional compilation to add explicit checks in certain build configurations (e.g., using C-style `assert` macros). A more robust solution along these same lines is anticipated for a future release of the C++ language standard and will be addressed in **?**.

```
    {
        assert(index < std::vector<T>::size());
        return std::vector<T>::operator[](index);
    }
};
```

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.

Avoiding boilerplate code when employing implementation inheritance

Sometimes it can be cost effective to adapt a **concrete class** having virtual functions[4] to a specialized purpose using inheritance.[5] As an example, consider a 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 `NetworkDataStream` class above 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 Section 1.1."`override`" on page 94) `onPacketReceived(DataPacket&)` more convenient because we don't need to reimplement each of the constructors, which are anticipated to increase in number over time:

```
class LoggedNetworkDataStream : public NetworkDataStream
{
public:
    using NetworkDataStream::NetworkDataStream;
```

[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 **?**, 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 **?**, section 4.6.

```
    void onPacketReceived(DataPacket& dataPacket) override
    {
        LOG_TRACE << "Received packet " << dataPacket;    // local log facility
        NetworkDataStream::onPacketReceived(dataPacket);  // Delegate to base.
    }
};
```

Implementing a strong `typedef`

Classic **`typedef`** declarations — just like C++11 **`using`** declarations (see Section 1.1."**`using`** Aliases" on page 121) — 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 diligent 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
}
```

[6]A typical implementation of a strong **`typedef`** suppresses implicit conversions both from the new type to the type it wraps and vice versa via **`explicit`** converting constructors and **`explicit`** conversion operators. In this respect, the relationship of strong **`typedef`** with the type it wraps is analogous to that of a scoped enumeration (**`enum class`**) to its **underlying type**; see Section 2.1."Underlying Type '11" on page 234.

```
int client2(PatientInfo* result, Date birthday, Date appointment)
{
    *result = PatientInfo(appointment, birthday);  // Oops! wrong order
    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 uniquely represent 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
Appointment a2(1999, 9, 17); // OK, thanks to inheriting constructors
Birthday    b3(a2);          // OK, an Appointment (unfortunately) is a Date.
```

We can now reimagine a `PatientInfo` class that exploits this newfound (albeit artificially manufactured[8]) type-safety:

[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 **?**, section 4.4.

```
class PatientInfo
{
private:
    Birthday d_birthday;
    Appointment d_appointment;

public:
    PatientInfo(Birthday birthday, Appointment appointment);
        // Please pass the birthday as the first argument and the appointment as
        // the second one!
};
```

Now our clients have no choice but to make their intentions clear at the call site. The previous implementation of the client functions no longer compile:

```
PatientInfo client1(Date birthday, Date appointment)
{
    return PatientInfo(birthday, appointment);     // Error, doesn't compile.
}

int client2(PatientInfo* result, Date birthday, Date appointment)
{
    *result = PatientInfo(appointment, birthday);  // Error, doesn't compile.
    return 0;
}
```

Because the clients now need to explicitly convert their `Date` objects to the appropriate strong **`typedef`**s, it is easy to spot and fix the defect in `client2`:

```
PatientInfo client1(Date birthday, Date appointment)
{
    return PatientInfo(Birthday(birthday), Appointment(appointment));  // OK
}

int client2(PatientInfo* result, Date birthday, Date appointment)
{
    Birthday b(birthday);
    Appointment a(appointment);
    *result = PatientInfo(b, a);  // OK
}
```

In this example, the client functions failed to compile after the introduction of the strong **`typedef`**s which is the intended effect. However, if `Date` objects were *implicitly* constructed when client functions created `PatientInfo`, the defective code would continue to compile because both strong **`typedef`**s can be implicitly constructed from the same arguments; see *Potential Pitfalls — Inheriting* implicit *constructors* on page 191.

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>
int 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 <typename 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 ("ABC");  // Construct a counted string.
Counted<std::vector<char>> cvc(3, 'a'); // 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.

Potential Pitfalls

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 example above, `authSession` will be initialized by invoking the constructor accepting a **`long long`** (see Section 1.1."**`long long`**" on page 81) authentication token. If, however, a new constructor having the signature `Session(int fd)` 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 defect that would go unreported at compile time.

Note that this problem with implicit conversions for function parameters 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 **`typedef`**s (see *Use Cases — Implementing a strong* **`typedef`** on page 187) — might sometimes, however, help to prevent such unfortunate missteps.

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 187. This time, however, let's suppose we are exposing 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:

```
PointOfInterest client1(const std::string& name, const std::string& address)
{
    return PointOfInterest(name, address);  // OK
}

int client2(PointOfInterest*   result,
            const std::string& name,
            const std::string& address)
{
    *result = PointOfInterest(address, name);  // Oops! wrong order
    return 0;
}
```

We might think to again use strong **`typedef`**s here as we did for `PatientInfo` in *Use Cases — Implementing a strong* **`typedef`** on page 187:

```
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 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:

```
PointOfInterest client1(const std::string& name, const std::string& address)
{
    return PointOfInterest(Name(name), Address(address));
}

int client2(PointOfInterest*   result,
            const std::string& name,
            const std::string& address)
{
   *result = PointOfInterest(Name(name), Address(address)); // Fix forced.
    return 0;
}
```

But suppose that some clients pass the arguments by **const char*** instead of **const** std::string&:

```
PointOfInterest client3(const char* name, const char* address)
{
    return PointOfInterest(address, name);  // Bug, 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 can seriously undermine the effectiveness of the strong **typedef** idiom.

Annoyances

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 Section 1.1."Deleted Functions" on page 46) 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` unless that inherited constructor were explicitly excluded from `NumericalDataum` by use of `= delete`.

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. 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[9] : 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()
{
     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 Section 2.1."Forwarding References" on page 283) and variadic templates (see Section 2.1."Variadic Templates" on page 319) — 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 Section 2.1."Forwarding References" on page 283) 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

[9]Alisdair Meredith, one of the authors of the Standards paper that proposed this feature (**?**), 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**.

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.

Flawed initial specification led to diverging early implementations

The original specification of inheriting constructors in C++11 had a significant number of problems with general use.[10] 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[11] 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,[12] 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.[13]

See Also

- "Delegating Ctors" (§1.1, p. 21) ♦ Related feature used to call one constructor from another from within the same user-defined type.
- "Defaulted Functions" (§1.1, p. 34) ♦ Used to implement functions that might otherwise have been suppressed by inherited constructors.
- "Deleted Functions" (§1.1, p. 46) ♦ Can be used to exclude inherited constructors that are unwanted entirely.
- "`override`" (§1.1, p. 94) ♦ Used to ensure that a member function intended to override a virtual function actually does so.
- "Default Member Init" (§2.1, p. 198) ♦ Useful in conjunction with this feature when a derived class adds member data.
- "'Default Member Init" (§2.1, p. 198) ♦ Can be used to provide nondefault values for data members in derived classes that make use of inheriting constructors.
- "Forwarding References" (§2.1, p. 283) ♦ Used in alternative (workaround) when access levels differ from those for base-class constructors.
- "Variadic Templates" (§2.1, p. 319) ♦ Used in alternative (workaround) when access levels differ from those for base-class constructors.

---

[10] For the detailed analysis of the issues that were the consequence of the flawed initial C++11 specification of inheriting constructors, see [PRODUCTION: LINK TO BOOK WEBSITE SUPPLEMENTAL MATERIAL.]

[11] **?**

[12] **?**

[13] 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.

Further Reading

None so far

Default `class`/`union` Member Initializers

placeholder text.........

Strongly Typed Scoped Enumerations

`enum class` is an alternative to the classic `enum` construct that simultaneously provides both stronger typing and an enclosing scope for its enumerated values.

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 210 and *Annoyances: Scoped enumerations do not necessarily add value* on page 216. 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 201.

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.
```

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. 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.

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
```

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 210.

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 "Underlying Type '11" on page 234), 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 compile time, implicit conversion to a pointer type from anything but the literal integer constant `0` is not permitted. 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 "`nullptr`" on page 90.

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.[1]

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 203 and *Potential Pitfalls: External use of opaque enumerators* on page 215.

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 Enum { e_CAR, e_TRAIN, e_PLANE }; };
enum class VehicleScopedImplicitly { e_CAR, e_BOAT,  e_PLANE };
```

[1] 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.

```
void test()
{
  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:

```
void f1(VehicleUnscoped value);          // classic enumeration passed by value
void f2(VehicleScopedImplicitly value);  // modern enumeration passed by value
```

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 200, the name of the enclosing `struct` together with a consistent name for the enumeration, such as `Enum`, has to be used to indicate an enumerated type:

```
void f3(VehicleScopedExplicitly::Enum value);
    // classically scoped enum 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**:

```
void h()
{
    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
}
```

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 210.

`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 of an `enum class` in a local redeclaration is *always* possible; see *Potential Pitfalls: External use of opaque enumerators* on page 215 and "Opaque **enum**s" on page 217.

Use Cases

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:

```
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;    // Bug, accidental misuse!

    for (int i = 0; i < trans; ++i)  // Bug, 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 210.

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, "");
```

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.cpp:
#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.cpp:
#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.cpp:
#include <vehicle.h>  // OK
#include <geometry.h>   // OK, enumerators are scoped implicitly.
// ...
```

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[2]:

```
void f0()
{
    clearScreen(1           , 1           );  // calls (0) above
```

[2]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 210.

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
}
```

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 to categorize and make readily available important individual properties might offer an advantage, such as in performance. Note that in order 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..

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:

```
#include <type_traits>  // std::underlying_type

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 the example above, 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 "Underlying Type '11" on page 234.

In case `enum`s are used for encoding purposes, 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 efficient `static_cast` to resolve the cardinal value of the enumerator and thus make the requested determination as efficiently as possible.

Potential Pitfalls

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 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

void test()
{
    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.

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 Init" on page 198. 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. 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 180) and a **prvalue** (see "*rvalue* References" on page 310), which never needs static storage and cannot have its address taken.

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()
{
    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; see "Local Types '11" on page 76:

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

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
```

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

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<Ctrl>::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)
}
```

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.

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 **enum**s" on page 217.

Annoyances

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
}
```

See Also

- "Underlying Type '11" (Section 2.1, p. 234) ♦ Absent implicit conversion to integrals, `enum` class values may use `static_cast` in conjunction with their underlying type.
- "Opaque **enum**s" (Section 2.1, p. 217) ♦ Sometimes it is useful to entirely insulate individual enumerators from clients.

Further Reading

- ?
- ?

Opaque Enumeration Declarations

Enumerated types, such as an **enum** or **enum** class (see Section 2.1."**enum class**" on page 199), whose underlying type (see Section 2.1."Underlying Type '11" on page 234) is well-specified, can be declared without being defined, i.e., declared without its enumerators.

Description

We identify two distinct forms of **opaque declarations**, i.e., declarations that are not also definitions:

1. A **forward declaration** has some translation unit in which the full definition and that declaration both appear. This can be a declaration in a header file where the definition is in the same header or in the corresponding implementation file. It can also be a declaration that appears in the same implementation file as the corresponding definition.

2. A **local declaration** has no translation unit that includes both that declaration and the corresponding full definition.

A classic (C++03) C-style **enum** cannot have opaque declarations, nor can its definition be repeated within the same **translation unit** (TU):

```
enum E0;                 // Error, opaque declaration
enum    { e_A0, e_B0 };  // OK, anonymous classic (only) enum
enum E1 { e_A1, e_B1 };  // OK, definition
enum E1;                 // OK, redeclaration in same TU
enum E2 { e_A2, e_B2 };  // OK, definition
enum E2 { e_A2, e_B2 };  // Error, redefinition in same TU
```

The underlying integral types used to represent objects of each of the (classic) enumerations in the example above is **implementation defined**, making all of them ineligible for opaque declaration. This restriction on opaque declarations exists because the specific values of the enumerators might affect the underlying type (e.g., size, alignment, **signed**ness), and therefore the declaration alone cannot be used to create objects of that type. A declaration that specifies the underlying type or a full definition can, however, be used to create objects of that type. Specifying an underlying type explicitly makes opaque declaration possible:

```
enum    : char { e_A, e_B };     // OK, (anonymous) complete definition
enum E3 : char;                  // OK, forward declaration w/underlying type
enum E3 : char { e_A3, e_B3 };   // OK, compatible complete definition
enum E4 : short { e_A4, e_B4 };  // OK, complete definition
enum E4 : short;                 // OK, compatible opaque redeclaration
enum E4 : int;                   // Error, incompatible opaque redeclaration
enum E5 : int { e_A5, e_B5 };    // OK, complete definition
enum E5 : int { e_A5, e_B5 };    // Error, complete redefinition in same TU
```

The modern (C++11) **enum class**, which provides its enumerators with (1) stronger typing and (2) an enclosing scope, also comes with a default **underlying type** of **int**, thereby making it eligible to be declared without a definition (even without explicit qualification):

```
enum class E6;                 // OK, implicit underlying type (int)
enum class E6 : int;           // OK, explicit matching underlying type
enum class E6 { e_A3, e_B3 };  // OK, compatible complete definition
enum class E7 { e_A4, e_B4 };  // OK, complete definition, int underlying type
enum class E7;                 // OK, compatible opaque redeclaration
enum class E7 : short;         // Error, incompatible opaque redeclaration
enum class E8 : long;          // OK, opaque declaration, long underlying type
enum class E8 : long { e_A5 }; // OK, compatible complete definition
enum class E9 { e_A6, e_B7 };  // OK, complete definition
enum class E9 { e_A6, e_B7 };  // Error, complete redefinition in same TU
enum class    { e_A, e_B };    // Error, anonymous enum classes are not allowed
```

To summarize, each classical **enum** type having an explicitly specified underlying type and every modern **enum class** type can be declared (e.g., for the first time in a TU) as a **complete type**:

```
enum E10 : char;  static_assert(sizeof(E10)  == 1);
enum class E11;   static_assert(sizeof(E11) == sizeof(int));

E10 a;            static_assert(sizeof a == 1);
E11 b;            static_assert(sizeof b == sizeof(int));
```

Typical usage of opaque enumerations often involves placing the forward declaration within a header and sequestering the complete definition within a corresponding `.cpp` (or else a second header), thereby **insulating** (at least some) clients from changes to the enumerator list (see *Use Cases — Using opaque enumerations within a header file* on page 218):

```
// mycomponent.h:
// ...
enum E9 : char;   // forward declaration of enum E9
enum class E10;   // forward declaration of enum class E10


// mycomponent.cpp:
#include <mycomponent.h>
// ...
enum E9 : char { e_A9, e_B9, e_C9 };
    // complete definition compatible with forward declaration of E9

enum class E10 { e_A10, e_B10, e_C10 };
    // complete definition compatible with forward declaration of E10
```

Note, however, that clients embedding *local declarations* directly in their code can be problematic; see *Potential Pitfalls — Redeclaring an externally defined enumeration locally* on page 230.

Use Cases

Using opaque enumerations within a header file

Physical design involves two related but distinct notions of information *hiding*: **encapsulation** and **insulation**. An implementation detail is *encapsulated* if changing it (in a semantically compatible way) does not require clients to rework their code but might require them to recompile it.

An *insulated* implementation detail, on the other hand, can be altered compatibly *without* forcing clients even to recompile. The advantages of avoiding such **compile-time coupling** transcend merely reducing compile time. For larger codebases in which various layers are managed under different release cycles, making a change to an *insulated* detail can be done with a `.o` patch and a relink the same day, whereas an *uninsulated* change might precipitate a waterfall of releases spanning days, weeks, or even longer.

As a first example of **opaque-enumeration** usage, consider a non-*value-semantic* **mechanism** class, `Proctor`, implemented as a finite-state machine:

```
// proctor.h:
// ...
class Proctor
{
    int d_current;  // "opaque" but unconstrained int type (BAD IDEA)
    // ...

public:
    Proctor();
    // ...
};
```

Among other private members, `Proctor` has a data member, `d_current`, representing the current enumerated state of the object. We anticipate that the implementation of the underlying state machine will change regularly over time but that the **`public`** interface is relatively stable. We will, therefore, want to ensure that all parts of the implementation that are likely to change reside outside of the header. Hence, the complete definition of the enumeration of the states (including the enumerator list itself) is sequestered within the corresponding `.cpp` file:

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

enum State { e_STARTING, e_READY, e_RUNNING, e_DONE };
Proctor::Proctor() : d_current(e_STARTING) { /* ... */ }
// ...
```

Prior to C++11, enumerations could not be **forward declared**. To avoid exposing (in a header file) enumerators that were used only privately (in the `.cpp` file), a completely unconstrained **`int`** would be used as a data member to represent the state. With the advent of modern C++, we now have better options. First, we might consider adding an explicit underlying type to the enumeration in the `.cpp` file:

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

enum State : int { e_STARTING, e_READY, e_RUNNING, e_DONE };
Proctor::Proctor() : d_current(e_STARTING) { /* ... */ }
// ...
```

Now that the **component-local** **enum** has an explicit underlying type, we can **forward declare** it in the header file. The existence of `proctor.cpp`, which includes `proctor.h`, makes this declaration a *forward declaration* and not just an *opaque declaration*. Compilation of `proctor.cpp` guarantees that the declaration and definition are compatible. Having this *forward declaration* improves (somewhat) our type safety:

```
// proctor.h:
// ...
enum State : int;  // opaque declaration of enumeration (new in C++11)

class Proctor
{
    State d_current;  // opaque classical enumerated type (BETTER IDEA)
    // ...

public:
    Proctor();
    // ...
};
```

But we can do even better. First we will want to nest the enumerated `State` type within the private section of the proctor to avoid needless namespace pollution. What's more, because the numerical values of the enumerators are not relevant, we can more closely model our intent by nesting a more strongly typed **enum class** instead:

```
// proctor.h:
// ...
class Proctor
{
    enum class State;  // forward (nested) declaration of type-safe enumeration
    State d_current;   // opaque (modern) enumerated data type (BEST IDEA)
    // ...

public:
    Proctor();
    // ...
};
```

We would then declare the nested **enum class** accordingly in the `.cpp` file:

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

enum class Proctor::State { e_STARTING, e_READY, e_RUNNING, e_DONE };
Proctor::Proctor() : d_current(State::e_STARTING) { /* ... */ }
// ...
```

Finally, notice that in this example we first *forward declared* the nested **enum class** type within class scope and then in a separate statement defined a data member of the opaque enumerated type. We needed to do this in two statements because simultaneously *opaquely declaring* either a classic **enum** having an explicit underlying type or an **enum class** and also defining an object of that type in a single statement is not possible:

```
enum E1 : int e1;  // Error, syntax not supported
enum class E2 e2;  // Error,    "    "     "
```

Fully defining an enumeration and simultaneously defining an object of the type in one stroke is, however, possible:

```
enum E3 : int { e_A, e_B } e3;  // OK, full type definition + object definition
enum class E4 { e_A, e_B } e4;  // OK,  "    "       "      "   "        "
```

Providing such a full definition, however, would have run counter to our intention to **insulate** the enumerator list of `Proctor::State` from clients **`#include`**ing the header file defining `Proctor`.

Cookie: Insulating all external clients from the enumerator list

A commonly recurring **design pattern**, commonly known as the "Memento pattern,"[1] manifests when a facility providing a service, often in a multi-client environment, hands off a packet of opaque information — a.k.a. a **cookie** — to a client to hold and then present back to the facility to enable resuming operations where processing left off. Since the information within the cookie will not be used substantively by the client, any unnecessary compile-time coupling of clients with the implementation of that cookie serves only to impede fluid maintainability of the facility issuing the cookie. With respect to not just *encapsulating* but *insulating* pure implementation details that are held but not used substantively by clients, we offer this Memento pattern as a possible use case for opaque enumerations.

Event-driven programming,[2] historically implemented using **callback functions**, introduces a style of programming that is decidedly different from that to which we might have become accustomed. In this programming paradigm, a higher-level agent (e.g., `main`) would begin by instantiating an `Engine` that will be responsible for monitoring for events and invoking provided callbacks when appropriate. Classically, clients might have registered a function pointer and a corresponding pointer to a client-defined piece of identifying data, but here we will make use of a C++11 Standard Library type, `std::function`, which can encapsulate arbitrary callable function objects and their associated state. This callback will be provided one object to represent the event that just happened and another object that can be used opaquely to reregister interest in the same event again.

This opaque cookie and passing around of the client state might seem like an unnecessary step, but often the event management involved in software of this sort is wrapping the most often executed code in very busy systems, and performance of each basic operation is therefore very important. To maximize performance, every potential branch or lookup in an internal data structure must be minimized, and allowing clients to pass back the internal state of the engine when reregistering can greatly reduce the engine's work to continue a client's processing of events without tearing down and rebuilding all client state each time an event happens. More importantly, event managers such as this often become highly concurrent to take advantage of modern hardware, so performant manipulation of their own data structures and well-defined lifetime of the objects they interact with become paramount. This makes the simple guarantee of, "If you don't reregister, then the engine

[1]**?**, Chapter 5, section "Memento," pp. 283–???
[2]See also **?**, Chapter 5, section "Observer," pp. 293–???

will clean everything up; if you do, then the callback function will continue its lifetime," a very tractable paradigm to follow.

```
// callbackengine.h:
#include <deque>          // std::deque
#include <functional>     // std::function

class EventData;       // information that clients will need to process an event
class CallbackEngine;  // the driver for processing and delivering events

class CallbackData
{
    // This class represents a handle to the shared state associating a
    // callback function object with a CallbackEngine.

public:
    typedef std::function<void(const EventData&, CallbackEngine*,
        CallbackData)> Callback;
        // alias for a function object returning void and taking, as arguments,
        // the event data to be consumed by the client, the address of the
        // CallbackEngine object that supplied the event data, and the
        // callback data that can be used to reregister the client, should the
        // client choose to show continued interest in future instances of the
        // same event

    enum class State;  // GOOD IDEA
        // nested forward declaration of insulated enumeration, enabling
        // changes to the enumerator list without forcing clients to recompile

private:
    // ... (a smart pointer to an opaque object containing the state and the
    //      callback to invoke)

public:
    CallbackData(const Callback &cb, State init);

    // ... (constructors, other manipulators and accessors, etc.)

    State getState() const;
        // Return the current state of this callback.

    void setState(State state) const;
        // Set the current state to the specified state.

    Callback& getCallback() const;
        // Return the callback function object specified at construction.
};

class CallbackEngine
{
```

```
private:
    // ... (other, stable private data members implementing this object)

    bool d_running;  // active state

    std::deque<CallbackData> d_pendingCallbacks;
        // The collection of clients currently registered for interest, or having
        // callbacks delivered, with this CallbackEngine.
        //
        // Reregistering or skipping reregistering when
        // called back will lead to updating internal data structures based on
        // the current value of this State.

public:
    // ...   (other public member functions, e.g., creators, manipulators)

    void registerInterest(CallbackData::Callback cb);
        // Register (e.g., from main) a new client with this manager object.

    void reregisterInterest(const CallbackData& callback);
        // Reregister (e.g., from a client) the specified callback with this
        // manager object, providing the state contained in the CallbackData
        // to enable resumption from the same state as processing left off.

    void run();
        // Start this object's event loop.

    // ...  (other public member functions, e.g., manipulators, accessors)
};
```

A client would, in `main`, create an instance of this `CallbackEngine`, define the appropriate functions to be invoked when events happen, register interest, and then let the engine `run`:

```
// myapplication.cpp:
// ...
#include <callbackengine.h>

static void myCallback(const EventData&    event,
                       CallbackEngine*     engine,
                       const CallbackData& cookie);
    // Process the specified event, and then potentially reregister the
    // specified cookie for interest in the same data.

int main()
{
   CallbackEngine e;  // Create a configurable callback engine object.

   //...     (Configure the callback engine, e, as appropriate.)

   e.registerInterest(&myCallback);  // Even a stateless function pointer can
```

```
                                       // be used with std::function.

    // ...create and register other clients for interest...

    e.run();    // Cede control to e's event loop until complete.

    return 0;
}
```

The implementation of `myCallback`, in the example below, is then free to reregister interest in the same event, save the cookie elsewhere to reregister at a later time, or complete its task and let the `CallbackEngine` take care of properly cleaning up all now unnecessary resources:

```
void myCallback(const EventData&     event,
                CallbackEngine      *engine,
                const CallbackData&  cookie)
{
    int status = EventProcessor::processEvent(event);

    if (status > 0)  // status is non-zero; continue interest in event
    {
        engine->reregisterInterest(cookie);
    }
    else if (status < 0)  // Negative status indicates EventProcessor wants
                          // to reregister later.
    {
        EventProcessor::storeCallback(engine,cookie);
                          // Call reregisterInterest later.
    }

    // Return flow of control to the CallbackEngine that invoked this
    // callback.  If status was zero, then this callback should be cleaned
    // up properly with minimal fuss and no leaks.
}
```

What makes use of the opaque enumeration here especially apt is that the internal data structures maintained by the `CallbackEngine` might be very subtly interrelated, and any knowledge of a client's relationship to those data structures that can be maintained through callbacks is going to reduce the amount of lookups and synchronization that would be needed to correctly reregister a client without that information. The otherwise wide contract on `reregisterInterest` means that clients have no need themselves to directly know anything about the actual values of the `State` they might be in. More notably, a component like this is likely to be very heavily reused across a large codebase, and being able to maintain it while minimizing the need for clients to recompile can be a huge boon to deployment times.

To see what is involved, we can consider the business end of the `CallbackEngine` implementation and an outline of what a single-threaded implementation might involve:

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

```
enum class CallbackData::State
{
    // Full (local) definition of the enumerated states for the callback engine.
    e_INITIAL,
    e_LISTENING,
    e_READY,
    e_PROCESSING,
    e_REREGISTERED,
    e_FREED
};

void CallbackEngine::registerInterest(CallbackData::Callback cb)
{
    // Create a CallbackData instance with a state of e_INITIAL and
    // insert it into the set of active clients.
    d_pendingCallbacks.push_back(CallbackData(cb, CallbackData::State::e_INITIAL));
}

void CallbackEngine::run()
{
    // Update all client states to e_LISTENING based on the events in which
    // they have interest.

    d_running = true;
    while (d_running)
    {
        // Poll the operating system API waiting for an event to be ready.
        EventData event = getNextEvent();

        // Go through the elements of d_pendingCallbacks to deliver this
        // event to each of them.
        std::deque<CallbackData> callbacks = std::move(d_pendingCallbacks);

        // Loop once over the callbacks we are about to notify to update their
        // state so that we know they are now in a different container.
        for (CallbackData& callback : callbacks)
        {
            callback.setState(CallbackData::State::e_READY);
        }

        while (!callbacks.empty())
        {
            CallbackData callback = callbacks.front();
            callbacks.pop_front();

            // Mark the callback as processing and invoke it.
            callback.setState(CallbackData::State::e_PROCESSING);
```

```
            callback.getCallback()(event, this, callback);

            // Clean up based on the new State.
            if (callback.getState() == CallbackData::State::e_REREGISTERED)
            {
                // Put the callback on the queue to get events again.
                d_pendingCallbacks.push_back(callback);
            }
            else
            {
                // The callback can be released, freeing resources.
                callback.setState(CallbackData::State::e_FREED);
            }
        }
    }
}

void CallbackEngine::reregisterInterest(const CallbackData& callback)
{
    if (callback.getState() == CallbackData::State::e_PROCESSING)
    {
        // This is being called reentrantly from run(); simply update state.
        callback.setState(CallbackData::State::e_REREGISTERED);
    }
    else if (callback.getState() == CallbackData::State::e_READY)
    {
        // This callback is in the deque of callbacks currently having events
        // delivered to it; do nothing and leave it there.
    }
    else
    {
        // This callback was saved; set it to the proper state and put it in
        // the queue of callbacks.
        if (d_running)
        {
            callback.setState(CallbackData::State::e_LISTENING);
        }
        else
        {
            callback.setState(CallbackData::State::e_INITIAL);
        }

        d_pendingCallbacks.push_back(callback);
    }
}
```

Note how the definition of `CallbackData::State` is visible and needed only in this implementation file. Also, consider that the set of states might grow or shrink as this `CallbackEngine` is optimized and extended, and clients can still pass around the object containing that state in a type-safe manner while remaining insulated from this definition.

Prior to C++11, we could not have *forward declared* this enumeration, and so would have had to represent it in a *type-unsafe* way — e.g., as an **`int`**. Thanks to the modern **`enum class`** (see Section 2.1."**`enum class`**" on page 199), however, we can conveniently forward declare it as a nested type and then, separately, fully define it inside the `.cpp` implementing other noninline member functions of the `CallbackEngine` class. In this way, we are able to *insulate* changes to the enumerator list along with any other aspects of the implementation defined outside of the `.h` file without forcing any client applications to recompile. Finally, the basic design of the hypothetical `CallbackEngine` in the previous code example could have been used for any number of useful components: a parser or tokenizer, a workflow engine, or even a more generalized event loop.

Dual-Access: Insulating some external clients from the enumerator list

In previous use cases, the goal has been to insulate *all* external clients from the enumerators of an enumeration that is visible (but not necessarily programmatically reachable) in the defining component's header. Consider the situation in which a **component** (`.h`/`.cpp` pair) itself defines an enumeration that will be used by various clients within a single program, some of which will need access to the enumerators.

When an **`enum class`** or a classic **`enum`** having an explicitly specified underlying type (see Section 2.1."Underlying Type '11" on page 234) is specified in a header for direct programmatic use, external clients are at liberty to unilaterally redeclare it *opaquely*, i.e., without its enumerator list. A compelling motivation for doing so would be for a client who doesn't make direct use of the enumerators to insulate itself and/or its clients from having to recompile whenever the enumerator list changes.

Embedding any such local declaration in client code, however, would be highly problematic: If the underlying type of the declaration (in one translation unit) were somehow to become inconsistent with that of the definition (in some other translation unit), any program incorporating both translation units would immediately become silently **ill-formed, no diagnostic required (IFNDR)**; see *Potential Pitfalls — Redeclaring an externally defined enumeration locally* on page 230. Unless a separate "forwarding" header file is provided along with (and ideally included by) the header defining the full enumeration, any client opting to exploit this opacity feature of an enumerated type will have no alternative but to redeclare the enumeration locally; see *Potential Pitfalls — Inciting local enumeration declarations: an attractive nuisance* on page 231.

For example, consider an **`enum class`**, `Event`, intended for public use by external clients:

```
// event.h:
// ...
enum class Event : char { /*... changes frequently ...*/ };
// ...              ^^^^
```

Now imagine some client header file, `badclient.h`, that makes use of the `Event` enumeration and chooses to avoid compile-time coupling itself to the enumerator list by embedding, for whatever reason, a local declaration of `Event` instead:

```
// badclient.h:
// ...
```

```
enum class Event : char;  // BAD IDEA: local external declaration
// ...
struct BadObject
{
    Event d_currentEvent;  // object of locally declare enumeration
    // ...
};
// ...
```

Imagine now that the number of events that can fit in a **`char`** is exceeded and we decide to change the definition to have an underlying type of **`short`**:

```
// event.h:
// ...
enum class Event : short { /*... changes frequently ...*/ };
// ...               ^^^^^
```

Client code, such as in `badclient.h`, that fails to include the `event.h` header will have no automatic way of knowing that it needs to change, and recompiling the code for all cases where `event.h` isn't also included in the translation unit will not fix the problem. Unless every such client is somehow updated manually, a newly linked program comprising them will be **IFNDR** with the likely consequence of either a crash or (worse) when the program runs and misbehaves. When providing a programmatically accessible definition of an enumerated type in a header where the underlying type is specified either explicitly or implicitly, we can give external clients a *safe* alternative to local declaration by also providing an auxiliary header containing *just* the corresponding opaque declaration:

```
// event.fwd.h:
// ...
enum class Event : char;
// ...
```

Here we have chosen to treat the forwarding header file as part of the same event component as the principal header but with an injected descriptive suffix field, `.fwd`; this approach, as opposed to, say, `file_fwd.h`, `filefwd.h`, or `file.hh`, was chosen so as not to (1) encroach on a disciplined, component-naming scheme involving reserved use of underscores[3] or (2) confuse tools and scripts that expect header-file names to end with a `.h` suffix.

In general, having a forwarding header always included in its corresponding full header facilitates situations such as default template arguments where the declaration can appear at most once in any given translation unit; the only drawback being that the comparatively small forwarding header file must now also be opened and parsed if the full header file is included in a given translation unit. To ensure consistency, we thus **`#include`** this forwarding header in the original header defining the full enumeration:

```
// event.h:
// ...                    // Ensure opaque declaration (included here) is
#include <event.fwd.h>  // consistent with complete definition (below).
// ...
enum class Event : char { /*... changes frequently ...*/ };
```

[3] **?**, section 2.4, pp. 297–333

```
// ...
```

In this way, every translation unit that includes the definition will serve to ensure that the forward declaration and definition match; hence, clients can incorporate safely only the presumably more stable forwarding header:

```
// goodclient.h:
// ...
#include <event.fwd.h>  // GOOD IDEA: consistent opaque declaration
// ...
class Client
{
    Event d_currentEvent;
    // ...
};
```

Note that we have consistently employed angle brackets exclusively for all include directives used throughout this book to maximize flexibility of deployment presuming a regimen for unique naming.[4]

To illustrate real-world practical use of the opaque-enumerations feature, consider the various components[5] that might depend on[6] an `Event` enumeration such as that above:

- **`message`** — The component provides a *value-semantic* `Message` class consisting of just raw data,[7] including an `Event` field representing the type of event. This component never makes direct use of any enumeration values and thus needs to include only `event.fwd.h` and the corresponding opaque *forward* declaration of the `Event` enumeration.

- **`sender` and `receiver`** — These are a pair of components that, respectively, create and consume `Message` objects. To populate a `Message` object, a `Sender` will need to provide a valid value for the `Event` member. Similarly, to process a `Message`, a `Receiver` will need to understand the potential individual enumerated values for the `Event` field. Both of these components will include the primary `event.h` header and thus have the complete definition of `Event` available to them.

- **`messenger`** — The final component, a general engine capable of being handed `Message` objects by a `Sender` and then delivering those objects in an appropriate fashion to a `Receiver`, needs a complete and usable definition of `Message` objects — possibly copying them or storing them in containers before delivery — but has no need for understanding the possible values of the `Event` member within those `Message` objects. This component can participate fully and correctly in the larger system while being completely *insulated* from the enumeration values of the `Event` enumeration.

[4]See **?**, section 1.5.1, pp. 201–203.

[5]See **?**, sections 1.6 and 1.11, pp. 209–216 and pp. 256–259, respectively.

[6]**?**, section 1.9?, pp. 237–243 JOHN: Please consult the book and correct. Section 1.9 is pp 243–251. Section 1.8 is pp. 237–243.

[7]We sometimes refer to data that is meaningful only in the context of a higher-level entity as **dumb data**; see **?**, section 3.5.5, pp. 629–633.

tl;dr: By factoring out the `Event` enumeration into its own separate component and providing two distinct but compatible headers, one containing the opaque declaration and the other (which includes the first) providing the complete definition, we enable having different components choose not to compile-time couple themselves with the enumerator list without forcing them to (*unsafely*) redeclare the enumeration locally.

Potential Pitfalls

Redeclaring an externally defined enumeration locally

An opaque enumeration declaration enables the use of that enumeration without granting visibility to its enumerators, reducing physical coupling between components. Unlike a **forward class declaration**, an opaque enumeration declaration produces a complete type, sufficient for substantive use (e.g., via the linker):

```
// client.cpp:
enum Event : std::uint8_t;
Event e;  // OK, Event is a complete type.
```

The *underlying type* specified in an opaque **enum** declaration must exactly match the full definition; otherwise a program incorporating both is automatically IFNDR. Updating an **enum**'s underlying type to accommodate additional values can lead to latent defects when these changes are not propagated to all local declarations:

```
// library.h:
enum Event : std::uint16_t { /* now more than 256 events */ };
```

Consistency of a local opaque **enum** declaration's underlying type with that of its complete definition in a separate translation unit cannot be enforced by the compiler, potentially leading to a program that is IFNDR. In the `client.cpp` example shown above, if the opaque declaration in `client.cpp` is not somehow updated to reflect the changes in `event.h`, the program will compile, link, and run, but its behavior has silently become undefined. The only robust solution to this problem is for `library.h` to provide two separate header files; see *Inciting local enumeration declarations: an attractive nuisance* on page 231.

The problem with local declarations is by no means limited to opaque enumerations. Embedding a local declaration for any object whose use might be associated with its definition in a separate translation unit via just the linker invites instability:

```
// main.cpp:                                  // library.cpp:
extern int x;  // BAD IDEA!                    int x;
// ...                                         // ...
```

The definition of object `x` (in the code snippets above) resides in the `.cpp` file of the library component while a supposed declaration of `x` is embedded in the file defining `main`. Should the type of just the definition of `x` change, both translation units will continue to compile but, when linked, the resulting program will be IFNDR:

```
// main.cpp:                                          // library.cpp:
extern int x;  // ILL-FORMED PROGRAM                   double x;
// ...                                                 // ...
```

To ensure consistency across translation units, the time-honored tradition is to place, in a header file managed by the supplier, a declaration of each external-linkage entity intended for use outside of the translation unit in which it is defined; that header is then included by both the supplier and each consumer:

```
// main.cpp:
#include <library.h>
// ...
```

```
// library.h:
// ...
extern int x;
// ...
```

```
// library.cpp:
#include <library.h>
int x;
// ...
```

In this way, any change to the definition of `x` in `library.cpp` — the supplier — will trigger a compilation error when `library.cpp` is recompiled, thereby forcing a corresponding change to the declaration in `library.h`. When that happens, typical build tools will take note of the change in the header file's timestamp relative to that of the `.o` file corresponding to `main.cpp` — the consumer — and indicate that it too needs to be recompiled. Problem solved.

The maintainability pitfall associated with opaque enumerations, however, is qualitatively more severe than for other external-linkage types, such as a global **`int`**: (1) the full definition for the enumeration type itself needs to reside in a header for *any* external client to make use of its individual enumerators and (2) typical components consist of just a `.h`/`.cpp` pair, i.e., exactly one `.h` file and usually just one `.cpp` file.[8]

Exposing, within a library header file, an opaquely declarable enumeration that is programmatically accessible by external clients without providing some maintainable way for those clients to keep their elided declarations in sync with the full definition introduces what we are calling an *attractive nuisance*: the client is forced to choose between (a) introducing the added risk and maintenance burden of having to manually maintain consistency between the underlying types for all its separate opaque uses and the one full definition or (b) forgo use of this opaque-enumeration feature entirely, forcing gratuitous compile-time coupling with the unused and perhaps unstable enumerators. At even moderate scale, excessive compile-time coupling can adversely affect projects in ways that are far more insidious than just increased compile times during development — e.g., any emergency changes that might need to occur and be deployed quickly to production without forcing all clients to recompile and then be retested and then, eventually, be rereleased.[9]

Inciting local enumeration declarations: an attractive nuisance

Whenever we, as library component authors, provide the complete definition of an **`enum class`** or a classic enumeration with an explicitly specified underlying type and fail to provide a corresponding header having just the opaque declaration, we confront our clients with the unenviable conundrum of whether to needlessly compile-time couple themselves and/or their clients with the details of the enumerator list or to make the dubious choice to unilaterally redeclare that enumeration locally.

The problems associated with local declarations of data whose types are maintained in separate translation units is not limited to enumerations; see *Redeclaring an externally de-*

[8] **?**, sections 2.2.11–2.2.13, pp. 280–281

[9] For a complete real-world example of how compile-time coupling can delay a "hot fix" by weeks, not just hours, see **?**, section 3.10.5, pp. 783–789.

fined enumeration locally on page 230. The maintainability pitfall associated with opaque enumerations, however, is qualitatively more severe than for other external-linkage types, such as a global **`int`**, in that the ability to elide the enumerators amounts to an *attractive nuisance* wherein a client — wanting to do so and having access to only a single header containing the unelided definition (i.e., comprising the enumeration name, underlying integral type, and enumerator list) — might be persuaded into providing an elided copy of the **`enum`**'s definition (i.e., one omitting just the enumerators) locally!

Ensuring that library components that define enumerations (e.g., **`enum class`** `Event`) whose enumerators can be elided also consistently provide a second forwarding header file containing the opaque declaration of each such enumeration (i.e., enumeration name and underlying integral type only) would be one generally applicable way to sidestep this often surprisingly insidious maintenance burden; see *Dual-Access: Insulating some external clients from the enumerator list* on page 227. Note that the attractive nuisance potentially exists even when the primary intent of the component is not to make the enumeration generally available.[10]

Annoyances

Opaque enumerations are not completely type safe

Making an enumeration opaque does not stop it from being used to create an object that is initialized opaquely to a zero value and then subsequently used (e.g., in a function call):

```
enum Bozo : int;  // forward declaration of enumeration Bozo
void f(Bozo);     // forward declaration of function f

void g()
{
    Bozo clown{};
    f(clown);      // OK, who knows if zero is a valid value?!
}
```

Though creating a zero-valued enumeration variable by default is not new, allowing one to be created without even knowing what enumerated values are valid is arguably dubious.

See Also

- "Underlying Type '11" (Section 2.1, p. 234) ♦ Discusses the underlying integral representation for enumeration variables and their values.
- "**`enum class`**" (Section 2.1, p. 199) ♦ Introduces an implicitly scoped, more strongly typed enumeration.

Further Reading

- For more on internal versus external linkage, see **?**, section 1.3.1, pp. 154–159.
- For more on the use of header files to ensure consistency across translation units, see **?**, section 1.4, pp. 190–201, especially Figure 1-35, p. 197.

[10] **?**

- For more on the use of **`#include`** directives and **`#include`** guards, see **?**, section 1.5, pp. 201–209.
- For a complete delineation of inherent properties that belong to every well-conceived `.h`/`.cpp` pair, see **?**, sections 1.6 and 1.11, pp. 219–216 and 256–259, respectively.
- For an introduction to physical dependency, see **?**, section 1.8, pp. 237–243.
- For a suggestion on how to achieve unique naming of files, see **?**, section 2.4, pp. 297–333.
- For a thorough treatment of **architectural insulation**, see **?**, sections 3.10–3.11, pp. 773–835.

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.

Description

Every enumeration employs an integral type, known as its **underlying type**, to represent its compile-time-enumerated values. By default, the **underlying type** of a C++98 `enum` 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 };                   // OK, fits any char

enum Port { e_LEFT = -81, e_RIGHT = -82 };             // OK, fits signed char

enum Mask { e_LOW = 32767, e_HIGH = 65535 };           // OK, fits unsigned short

enum Big { e_32 = 1<<32 };                             // Error, too big for int

enum Err { K = 1024, M = K*K, G = M*K, T = G*K };      // Error, G*K overflows int...

enum OK { K = 1<<10, M = 1<<20, G = 1<<30, T = 1l<<40 }; // 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`.

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).

The above applies only to C++98 **`enum`**s; the default underlying type of an **`enum class`** is ubiquitously **`int`**, and it is not implementation defined; see Section 2.1."**`enum class`**" on page 199.

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 implementation-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).[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:

[1]The same is true in C++11 for `char16_t` and `char32_t` and in C++20 for `char8_t`.

```
enum Port : unsigned char
{
    // Each enumerator of Port is represented as an unsigned char type.

    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 237.

Use Cases

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  d_day;

public:
    Date();
    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 codebase, 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 154.)

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:

```
#include <cstdint>  // std::int8_t

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)
```

Potential Pitfalls

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 **enum**s" on page 217.)

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         { q, r, s, t, u };
enum E2 : short { v, w, x, y, z };

void test()
{
   f(E1::q);  // always calls f(int) on all platforms
   f(E2::v);  // 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 (see "**auto** Variables" on page 177):

```
#include <type_traits>  // std::underlying_type
```

```
template <typename E>
typename std::underlying_type<E>::type toUnderlying(E value)
{
    return static_cast<typename std::underlying_type<E>::type>(value);
}

void h()
{
    auto e1 = toUnderlying(E1::q); // might be anywhere from signed char to int
    auto e2 = toUnderlying(E2::v); // 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` in the long term (see "`constexpr` Variables" on page 180):

```
constexpr int k_GRAMS_PER_OZ = 28;  // future proof
```

See Also

- "**`enum class`**" (Section 2.1, p. 199) ♦ Introduces a scoped, more strongly typed enumeration that can explicitly specify an underlying type.
- "Opaque **`enum`**s" (Section 2.1, p. 217) ♦ Offers a means to insulate individual enumerators from clients.
- "`constexpr` Variables" (Section 2.1, p. 180) ♦ Introduces an alternative way of declaring compile-time constants.

Further Reading

- "Item 10: Prefer scoped `enum`s to unscoped `enum`s," **?**
- **?**

Extended `friend` Declarations

Extended **friend** declarations enable a class's author to designate a type alias, a template parameter, or any other previously declared type as a **friend** of that class.

Description

A **friend** declaration located within a given **user-defined type** (UDT) grants a designated type (or *free* function) access to private and protected members of that class. Because the extended **friend** syntax does not affect *function* friendships, this feature section addresses extended friendship only between *types*.

Prior to C++11, the Standard required an *elaborated type specifier* to be provided after the **friend** keyword to designate some other *class* as being a **friend** of a given type. An elaborated type specifier for a class is a syntactical element having the form `<class|struct|union> <identifier>`. Elaborated type specifiers can be used to refer to a previously declared entity or to declare a new one, with the restriction that such an entity is one of **class**, **struct**, or **union**:

```
// C++03

struct S;
class C;
enum E { };

struct X0
{
    friend S;          // Error, not legal C++98/03
    friend struct S;   // OK, refers to S above
    friend class S;    // OK, refers to S above (might warn)
    friend class C;    // OK, refers to C above
    friend class C0;   // OK, declares C0 in X0's namespace
    friend union U0;   // OK, declares U0 in X0's namespace
    friend enum E;     // Error, enum cannot be a friend.
    friend enum E2;    // Error, enum cannot be forward-declared.
};
```

This restriction prevents other potentially useful entities, e.g., type aliases and template parameters, from being designated as friends:

```
// C++03

struct S;
typedef S SAlias;

struct X1
{
    friend struct SAlias;  // Error, using typedef-name after struct
};
```

```
template <typename T>
struct X2
{
    friend class T;        // Error, using template type parameter after class
};
```

Furthermore, even though an entity belonging to a namespace other than the class containing a **friend** declaration might be visible, explicit qualification is required to avoid unintentionally declaring a brand-new type:

```
// C++03

struct S;  // This S resides in the global namespace.

namespace ns
{
    class X3
    {
        friend struct S;
            // OK, but declares a new ns::S instead of referring to ::S
    };
}
```

C++11 relaxes the aforementioned *elaborated type specifier* requirement and extends the classic **friend** syntax by instead allowing either a *simple type specifier*, which is any unqualified type or type alias, or a *typename specifier*, e.g., the name of a template *type* parameter or dependent type thereof:

```
struct S;
typedef S SAlias;

namespace ns
{
    template <typename T>
    struct X4
    {
        friend T;           // OK
        friend S;           // OK, refers to ::S
        friend SAlias;      // OK, refers to ::S
        friend decltype(0); // OK, equivalent to friend int;
        friend C;           // Error, C does not name a type.
    };
}
```

Notice that now it is again possible to declare as a **friend** a type that is expected to have already been declared, e.g., `S`, without having to worry that a typo in the spelling of the type would silently introduce a new type declaration, e.g., `C`, in the enclosing scope.

Finally, consider the hypothetical case in which a class template, `C`, befriends a *dependent* (e.g., nested) type, `N`, of its type parameter, `T`:

```
template <typename T>
```

```
class C
{
    friend typename T::N;       // N is a *dependent* *type* of parameter T.
    enum { e_SECRET = 10022 };  // This information is private to class C.
};

struct S
{
    struct N
    {
        static constexpr int f()  // f is eligible for compile-time computation.
        {
            return C<S>::e_SECRET;  // Type S::N is a friend of C<S>.
        }
    };
};

static_assert(S::N::f() == 10022, "");  // N has private access to C<S>.
```

In the example above, the nested type `S::N` — but not `S` itself — has private access to `C<S>::e_SECRET`. [1]

Use Cases

Safely declaring a previously declared type to be a friend

In C++98/03, to befriend a type that was already declared required *redeclaring* it. If the type were misspelled in the friend declaration, a new type would be declared:

```
class Container { /* ... */ };

class ContainerIterator
{
    friend class Contianer;  // Compiles but wrong: ia should have been ai.
    // ...
};
```

The code above will compile and appear to be correct until `ContainerIterator` attempts to access a **`private`** or **`protected`** member of `Container`. At that point, the compiler will surprisingly produce an error. As of C++11, we have the option of preventing this mistake by using extended **`friend`** declarations:

```
class Container { /* ... */ };

class ContainerIterator
{
    friend Contianer;  // Error, Contianer not found
    // ...
```

[1] Note that the need for **`typename`** in the **`friend`** declaration in the example above to introduce the dependent type `N` is relaxed in C++20. For information on other contexts in which **`typename`** will eventually no longer be required, see **?**.

```
};
```

Befriending a type alias used as a customization point

In C++03, the only option for friendship was to specify a particular **class** or **struct** when granting private access. Let's begin by considering a scenario in which we have an **in-process**[2] **value-semantic type (VST)** that serves as a *handle* to a platform-specific object, such as a `Window` in a graphical application. Large parts of a codebase may seek to interact with `Window` objects without needing or obtaining access to the internal representation.

A very small part of the codebase that handles platform-specific window management, however, needs privileged access to the internal representation of `Window`. One way to achieve this goal is to make the platform-specific `WindowManager` a **friend** of the `Window` class; however, see *Potential Pitfalls — Long-distance friendship* on page 248.

```
class WindowManager;  // forward declaration enabling extended friend syntax

class Window
{
private:
    friend class WindowManager;  // could instead use friend WindowManager;
    int d_nativeHandle;          // in-process (only) value of this object

public:
    // ... all the typical (e.g., special) functions we expect of a value type
};
```

In the example above, class `Window` befriends class `WindowManager`, granting it private access. Provided that the implementation of `WindowManager` resides in the same physical **component** as that of class `Window`, no **long-distance friendship** results. The consequence of such a monolithic design would be that every client that makes use of the otherwise lightweight `Window` class would necessarily depend physically on the presumably heavier-weight `WindowManager` class.

Now consider that the `WindowManager` implementations on different platforms might begin to diverge significantly. To keep the respective implementations maintainable, one might choose to factor them into distinct C++ types, perhaps even defined in separate files, and to use a *type alias* determined using platform-detection preprocessor macros to configure that alias:

```
// windowmanager_win32.h:

#ifdef WIN32
class Win32WindowManager { /* ... */ };
#endif
```

[2]When used to qualify a VST, the term in-process, also called *in-core*, refers to a type that has typical value-type–like operations but does not refer to a value that is meaningful outside of the current process; see **?**, section 4.2.

```
// windowmanager_unix.h:

#ifdef UNIX
class UnixWindowManager { /* ... */ };
#endif


// windowmanager.h:

#ifdef WIN32
#include <windowmanager_win32.h>
typedef Win32WindowManager WindowManager;
#else
#include <windowmanager_unix.h>
typedef UnixWindowManager WindowManager;
#endif


// window.h:
#include <windowmanager.h>

class Window
{
private:
    friend WindowManager;  // C++11 extended friend declaration
    int d_nativeHandle;

public:
    // ...
};
```

In this example, class `Window` no longer befriends a specific class named `WindowManager`; instead, it befriends the `WindowManager` type alias, which in turn has been set to the correct platform-specific window manager implementation. Such extended use of **`friend`** syntax was not available in C++03.

Note that this use case involves long-distance friendship inducing an implicit cyclic dependency between the component implementing `Window` and those implementing `WindowManager`; see *Potential Pitfalls — Long-distance friendship* on page 248. Such designs, though undesirable, can result from an emergent need to add new platforms while keeping tightly related code sequestered within smaller, more manageable physical units. An alternative design would be to obviate the long-distance friendship by widening the API for the `Window` class, the natural consequence of which would be to invite public client abuse vis-a-vis **Hyrum's law**.

Using the `PassKey` idiom to enforce initialization

Prior to C++11, efforts to grant private access to a class defined in a separate physical unit required declaring the higher-level type itself to be a **`friend`**, resulting in this highly undesirable form of friendship; see *Potential Pitfalls — Long-distance friendship* on page 248.

The ability in C++11 to declare a template *type* parameter or any other type specifier to be a friend affords new opportunities to enforce selective private access (e.g., to one or more individual functions) without explicitly declaring another type to be a **friend**; see also *Granting a specific type access to a single* **private** *function* on page 245. In this use case, however, our use of extended **friend** syntax to befriend a template parameter is unlikely to run afoul of sound physical design.

Let's say we have a commercial library, and we want it to verify a software-license key in the form of a C-style string, prior to allowing use of other parts of the API:

```
// simplified pseudocode
LibPassKey initializeLibrary(const char* licenseKey);
int utilityFunction1(LibPassKey object /*, ... (other parameters) */);
int utilityFunction2(LibPassKey object /*, ... (other parameters) */);
```

Knowing full well that this is not a *secure* approach and that innumerable deliberate, malicious ways exist to get around the C++ type system, we nonetheless want to create a plausible regime where no *well-formed* code can *accidentally* gain access to library functionality other than by legitimately initializing the system using a valid license key. We could easily cause a function to **throw**, `abort`, and so on at run time when the function is called prior to the client's license key being authenticated. However, part of our goal, as a friendly library vendor, is to ensure that clients do not *inadvertently* call other library functions prior to initialization. To that end, we propose the following protocol:

1. use an instantiation of the `PassKey` class template[3] that only our API *utility* **struct**[4] can create

2. return a constructed object of this type only upon successful validation of the license key

3. require that clients present this (constructed) passkey *object* every time they invoke any other function in the API

Here's an example that encompasses all three aforementioned points:

```
template <typename T>
class PassKey  // reusable standard utility type
{
    PassKey() { }  // private default constructor (no aggregate initialization)
    friend T;      // Only T is allowed to create this object.
};

struct BestExpensiveLibraryUtil
{
    class LicenseError { /*...*/ };  // thrown if license string is invalid

    using LibPassKey = PassKey<BestExpensiveLibraryUtil>;
        // This is the type of the PassKey that will be returned when this
```

[3] **?**

[4] **?**, section 2.4.9, pp. 312–321, specifically Figure 2-23, p. 316

```
        // utility is initialized successfully, but only this utility is able
        // to construct an object of this type. Without a valid license string,
        // the client will have no way to create such an object and thus no way
        // to call functions within this library.

    static LibPassKey initializeLibrary(const char* licenseKey)
        // This function must be called with a valid licenseKey string prior
        // to using this library; if the supplied license is valid, a
        // LibPassKey *object* will be returned for mandatory use in *all*
        // subsequent calls to useful functions of this library. This function
        // throws LicenseError if the supplied licenseKey string is invalid.
    {
        if (isValid(licenseKey))
        {
            // Initialize library properly.

            return LibPassKey();
                // Return a default-constructed LibPassKey. Note that only
                // this utility is able to construct such a key.
        }

        throw LicenseError();  // supplied license string was invalid
    }

    static int doUsefulStuff(LibPassKey key /*,...*/);
        // The function requires a LibPassKey object, which can be constructed
        // only by invoking the static initializeLibrary function, to be
        // supplied as its first argument. ...

  private:
    static bool isValid(const char* key);
        // externally defined function that returns true if key is valid
};
```

Other than going outside the language with invalid constructs or circumventing the type system with esoteric tricks, this approach, among other things, prevents invoking the `doUsefulStuff` function without a proper license. What's more, the C++ type system *at compile time* forces a prospective client to have initialized the library before any attempt is made to use any of its other functionality.

Granting a specific type access to a single `private` function

When designing in purely logical terms, wanting to grant some other logical entity special access to a type that no other entity enjoys is a common situation. Doing so does not necessarily become problematic until that friendship spans physical boundaries; see *Potential Pitfalls — Long-distance friendship* on page 248.

As a simple approximation to a real-world use case,[5] suppose we have a lightweight

[5]For an example of a real-world database implementation that requires managed objects to befriend that database manager, see **?**, section 2.1.

object-database class, `Odb`, that is designed to operate collaboratively with objects, such as `MyWidget`, that are themselves designed to work collaboratively with `Odb`. Every compliant UDT suitable for management by `Odb` will need to maintain an integer object ID that is read/write accessible by an `Odb` object. Under no circumstances is any other object permitted to access, let alone modify, that ID independently of the `Odb` API.

Prior to C++11, the design of such a feature might require every participating class to define a data member named `d_objectId` and to declare the `Odb` class a **`friend`** (using old-style **`friend`** syntax):

```
class MyWidget  // grants just Odb access to *all* of its private data
{
    int d_objectId;    // required by our collaborative-design strategy
    friend class Odb;  //    "     "   "        "        "       "
    // ...

public:
    // ...
};

class Odb
{
    // ...

public:
    template <typename T>
    void processObject(T& object)
        // This function template is generally callable by clients.
    {
        int& objId = object.d_objectId;
        // ... (process as needed)
    }

    // ...
};
```

In this example, the `Odb` class implements the public member function template, `processObject`, which then extracts the `objectId` field for access. The collateral damage is that we have exposed all of our private details to `Odb`, which is at best a gratuitous widening of our sphere of encapsulation.

Using the `PassKey` pattern allows us to be more selective with what we share:

```
template <typename T>
class Passkey
    // Implement this eminently reusable Passkey class template again here.
{
    Passkey() { }  // prevent aggregate initialization
    friend T;      // Only the T in PassKey<T> can create a PassKey object.
    Passkey(const Passkey&) = delete;             // no copy/move construction
    Passkey& operator=(const Passkey&) = delete;  // no copy/move assignment
};
```

We are now able to adjust the design of our systems such that only the minimum private functionality is exposed to `Odb`:

```
class Odb;      // Objects of this class have special access to other objects.

class MyWidget  // grants just Odb access to only its objectId member function
{
    int d_objectId;  // must have an int data member of any name we choose
    // ...

public:
    int& objectId(const Passkey<Odb>&) { return d_objectId; }
        // Return a non-const reference to the mandated int data member.
        // objectId is callable only within the scope of Odb.

    // ...
};

class Odb
{
    // ...

public:
    template <typename T>
    void processObject(T& object)
        // This function template is generally callable by clients.
    {
        int& objId = object.objectId(Passkey<Odb>());
        // ...
    }

    // ...
};
```

Instead of granting `Odb` private access to *all* encapsulated implementation details of `MyWidget`, this example uses the `PassKey` idiom to enable just `Odb` to call the (syntactically **`public`**) `objectId` member function of `MyWidget` with no private access whatsoever. As a further demonstration of the efficacy of this approach, consider that we are able to create and invoke the `processObject` method of an `Odb` object from a function, `f`, but we are blocked from calling the `objectId` method of a `MyWidget` object directly:

```
void f()
{
    Odb mgr;           // object receiving fine-grained privileged access
    MyWidget widget;  // object granting selective private access to just Odb
    mgr.processObject(widget);

    int& objId = widget.objectId(PassKey<Odb>());  // cannot call out of Odb
       // Error, Passkey<T>::Passkey() [withT = Odb] is private within
       // this context.
```

```
}
```

Notice that use of the extended **friend** syntax to befriend a template parameter and thereby enable the `PassKey` idiom here improved the granularity with which we effectively grant privileged access to an individually named type but didn't fundamentally alter the testability issues that result when private access to specific C++ types is allowed to extend across physical boundaries; again, see *Potential Pitfalls — Long-distance friendship* on page 248.

Curiously recurring template pattern

Befriending a template parameter via extended **friend** declarations can be helpful when implementing the **curiously recurring template pattern (CRTP)**. For use-case examples and more information on the pattern itself, see *Appendix: Curiously Recurring Template Pattern Use Cases* on page 249.

Potential Pitfalls

Long-distance friendship

Since before C++ was standardized, granting private access via a **friend** declaration across physical boundaries, known as long-distance friendship, was observed[6,7] to potentially lead to designs that are qualitatively more difficult to understand, test, and maintain. When a user-defined type, `X`, befriends some other specific type, `Y`, in a separate, higher-level translation unit, testing `X` thoroughly without also testing `Y` is no longer possible. The effect is a test-induced cyclic dependency between `X` and `Y`. Now imagine that `Y` depends on a sequence of other types, `C1`, `C2`, …, `CN-2`, each defined in its own physical component, `CI`, where `CN-2` depends on `X`. The result is a physical design cycle of size *N*. As *N* increases, the ability to manage complexity quickly becomes intractable. Accordingly, the two design imperatives that were most instrumental in shaping the C++20 **modules** feature were (1) to have no cyclic module dependencies and (2) to avoid intermodule friendships.

See Also

TODO

Further Reading

- For yet more potential uses of the extended friend pattern in metaprogramming contexts, such as using CRTP, see **?**.
- **?**, section 3.6, pp. 136–146, is dedicated to the classic use (and misuse) of friendship.
- **?**
- **?** provides extensive advice on *sound* **physical design**, which generally precludes long-distance friendship.

[6]**?**, section 3.6.1 pp. 141–144

[7]**?**, section 2.6, pp. 342–370, specifically p. 367 and p. 362

Appendix: Curiously Recurring Template Pattern Use Cases

Refactoring using the curiously recurring template pattern

Avoiding code duplication across disparate classes can sometimes be achieved using a strange template pattern first recognized in the mid-90s, which has since become known as the curiously recurring template pattern (CRTP). The pattern is *curious* because it involves the surprising step of declaring as a base class, such as `B`, a template that *expects* the derived class, such as `C`, as a template argument, such as `T`:

```
template <typename T>
class B
{
    // ...
};

class C : public B<C>
{
   // ...
};
```

As a trivial illustration of how the CRTP can be used as a refactoring tool, suppose that we have several classes for which we would like to track, say, just the number of active instances:

```
class A
{
    static int s_count;  // declaration
    // ...

public:
    static int count() { return s_count; }

    A()          { ++s_count; }
    A(const A&)  { ++s_count; }
    A(const A&&) { ++s_count; }
    ~A()         { --s_count; }

    A& operator=(A&)  = default;  // see special members
    A& operator=(A&&) = default;  //  "     "       "
    // ...
};

int A::s_count;  // definition (in .cpp file)

class B { /* similar to A (above) */ };
// ...

void test()
{          // A::s_count = 0, B::s_count = 0
    A a1;  // A::s_count = 1, B::s_count = 0
    B b1;  // A::s_count = 1, B::s_count = 1
```

```
    A a2;  // A::s_count = 2, B::s_count = 1
}          // A::s_count = 0, B::s_count = 0
```

In this example, we have multiple classes, each repeating the same common machinery. Let's now explore how we might refactor this example using the CRTP:

```
template <typename T>
class InstanceCounter
{
protected:
    static int s_count;  // declaration

public:
    static int count() { return s_count; }
};

template <typename T>
int InstanceCounter<T>::s_count;  // definition (in same file as declaration)

struct A : InstanceCounter<A>
{
    A()          { ++s_count; }
    A(const A&)  { ++s_count; }
    A(const A&&) { ++s_count; }
    ~A()         { --s_count; }

    A& operator=(const A&)  = default;
    A& operator=(A&&)       = default;
    // ...
};
```

Notice that we have factored out a common counting mechanism into an `InstanceCounter` class template and then derived our representative class `A` from `InstanceCounter<A>`, and we would do similarly for classes `B`, `C`, and so on. This approach works because the compiler does not need to see the derived type until the point at which the template is instantiated, which will be *after* it has seen the derived type.

Prior to C++11, however, there was plenty of room for user error. Consider, for example, forgetting to change the base-type parameter when copying and pasting a new type:

```
struct B : InstanceCounter<A>  // Oops! We forgot to change A to B in
                               // InstanceCounter: The wrong count will be
                               // updated!
{
    B() { ++s_count; }
};
```

Another problem is that a client deriving from our class can mess with our protected `s_count`:

```
struct AA : A
{
```

```
    AA() { s_count = -1; }  // Oops! *Hyrum's Law* is at work again!
};
```

We could inherit from the `InstanceCounter` class privately, but then `InstanceCounter` would have no way to add to the derived class's public interface, for example, the public `count` static member function.

As it turns out, however, both of these missteps can be erased simply by making the internal mechanism of the `InstanceCounter` template private and then having `InstanceCounter` befriend its template parameter, `T`:

```
template <typename T>
class InstanceCounter
{
    static int s_count;  // Make this static data member private.
    friend T;            // Allow access only from the derived T.

public:
    static int count() { return s_count; }
};

template <typename T>
int InstanceCounter<T>::s_count;
```

Now if some other class does try to derive from this type, it cannot access this type's counting mechanism. If we want to suppress even that possibility, we can declare and default (see Section 1.1."Defaulted Functions" on page 34) the `InstanceCounter` class constructors to be private as well.

Synthesizing equality using the curiously recurring template pattern

As a second example of code factoring using the CRTP, suppose that we want to create a factored way of synthesizing **`operator==`** for types that implement just an **`operator<`**.[8] In this example, the CRTP base-class template, `E`, will synthesize the homogeneous **`operator==`** for its parameter type, `D`, by returning **`false`** if either argument is *less than* the other:

```
template <typename D>
class E { }; // CRTP base class used to synthesize operator== for D

template <typename D>
bool operator==(const E<D>& lhs, const E<D>& rhs)
{
    const D& d1 = static_cast<const D&>(lhs);  // derived type better be D
    const D& d2 = static_cast<const D&>(rhs);  //    "      "      "    "  "
    return !(d1 < d2) && !(d2 < d1);           // assuming D has an operator<
}
```

A client that implements an **`operator<`** can now reuse this CRTP base case to synthesize an **`operator==`**:

[8]This example is based on a similar one found on stackoverflow.com: https://stackoverflow.com/questions/4173254/what-is-the-curiously-recurring-template-pattern-crtp

```
struct S : E<S>
{
    int d_size;
};

bool operator<(const S& lhs, const S& rhs)
{
    return lhs.d_size < rhs.d_size;
}

void test1()
{
    S s1; s1.d_size = 10;
    S s2; s2.d_size = 10;

    assert(s1 == s2);  // compiles and passes
}
```

As this code snippet suggests, the base-class template, `E`, is able to use the template parameter, `D` (representing the derived class, `S`), to synthesize the homogeneous free **`operator==`** function for `S`.

Prior to C++11, no means existed to guard against accidents, such as inheriting from the wrong base and then perhaps even forgetting to define the **`operator<`**:

```
struct P : E<S>  // Oops! should have been E(P) -- a serious latent defect
{
    int d_x;
    int d_y;
};

void test2()
{
    P p1; p1.d_x = 10; p1.d_y = 15;
    P p2; p2.d_x = 10; p2.d_y = 20;

    assert( !(p1 == p2) );  // Oops! This fails because of E(S) above.
}
```

Again, thanks to C++11's extended **`friend`** syntax, we can defend against these defects at compile time simply by making the CRTP base class's default constructor *private* and befriending its template parameter:

```
template <typename D>
class E
{
     E() = default;
     friend D;
};
```

Note that the goal here is not security but simply guarding against accidental typos, copy-paste errors, and other occasional human errors. By making this change, we will soon realize that there is no **operator**< defined for P.

Compile-time polymorphism using the curiously recurring template pattern

Object-oriented programming provides certain flexibility that at times might be supererogatory. Here we will exploit the familiar domain of abstract/concrete shapes to demonstrate a mapping between runtime polymorphism using virtual functions and compile-time polymorphism using the CRTP. We begin with a simple abstract Shape class that implements a single, pure, virtual draw function:

```
class Shape
{
public:
    virtual void draw() const = 0;  // abstract draw function (interface)
};
```

From this abstract Shape class, we now derive two concrete shape types, Circle and Rectangle, each implementing the *abstract* draw function:

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

class Circle : public Shape
{
    int d_radius;

public:
    Circle(int radius) : d_radius(radius) { }

    void draw() const  // concrete implementation of abstract draw function
    {
        std::cout << "Circle(radius = " << d_radius << ")\n";
    }
};

class Rectangle : public Shape
{
    int d_length;
    int d_width;

public:
    Rectangle(int length, int width) : d_length(length), d_width(width) { }

    void draw() const  // concrete implementation of abstract draw function
    {
        std::cout << "Rectangle(length = " << d_length << ", "
                                "width = " << d_width  << ")\n";
    }
};
```

Notice that a `Circle` is constructed with a single integer argument, i.e., `radius`, and a `Rectangle` is constructed with two integers, i.e., `length` and `width`.

We now implement a function that takes an arbitrary shape, via a **`const`** *lvalue* reference to its abstract base class, and prints it:

```
void print(const Shape& shape)
{
    shape.draw();
}

void testShape()
{
    print(Circle(1));         // OK, prints: Circle(radius = 1)
    print(Rectangle(2, 3));   // OK, prints: Rectangle(length = 2, width = 3)
    print(Shape());           // Error, Shape is an abstract class.
}
```

Now suppose that we didn't need all the runtime flexibility offered by this system and wanted to map just what we have in the previous code snippet onto templates that avoid the spatial and runtime overhead of virtual-function tables and dynamic dispatch. Such transformation again involves creating a CRTP base class, this time in lieu of our abstract interface:

```
template <typename T>
struct Shape
{
    void draw() const
    {
        static_cast<const T*>(this)->draw();  // assumes T derives from Shape
    }
};
```

Notice that we are using a **`static_cast`** to the address of an object of the **`const`** template parameter type, `T`, assuming that the template argument is of the same type as some derived class of this object's type. We now define our types as before, the only difference being the form of the base type:

```
class Circle : public Shape<Circle>
{
    // same as above
};

class Rectangle : public Shape<Rectangle>
{
    // same as above
};
```

We now define our `print` function, this time as a function template taking a `Shape` of arbitrary type `T`:

```
template <typename T>
void print(const Shape<T>& shape)
{
```

```
    shape.draw();
}
```

The result of compiling and running `testShape` above is the same, including that `Shape()` doesn't compile.

However, opportunities for undetected failure remain. Suppose we decide to add a third shape, `Triangle`, constructed with three sides:

```
class Triangle : public Shape<Rectangle>  // Oops!
{
    int d_side1;
    int d_side2;
    int d_side3;

public:
    Triangle(int side1, int side2, int side3)
        : d_side1(side1), d_side2(side2), d_side3(side3) { }

    void draw() const
    {
        std::cout << "Triangle(side1 = " << d_side1 << ", "
                                 "side2 = " << d_side2 << ", "
                                 "side3 = " << d_side3 << ")\n";
    }
};
```

Unfortunately we forgot to change the base-class type parameter when we copy-pasted from `Rectangle`.

Let's now create a new test that exercises all three and see what happens on our platform:

```
void test2()
{
    print(Circle(1));          // prints: Circle(radius = 1)
    print(Rectangle(2, 3));    // prints: Rectangle(length = 2, width = 3)
    print(Triangle(4, 5, 6));  // prints: Rectangle(length = 4, width = 5) ?!
    Shape<int> bug;            // Compiles?!
}
```

As should by now be clear, a defect in our `Triangle` implementation results in *hard* **undefined behavior** that could have been prevented at compile time by using the extended **`friend`** syntax. Had we defined the CRTP base-class template's default constructor to be *private* and made its type parameter a **`friend`**, we could have prevented the copy-paste error with `Triangle` and suppressed the ability to create a `Shape` object without deriving from it (e.g., see `bug` in the previous code snippet):

```
template <typename T>
class Shape
{
    Shape() = default;  // Default the default constructor to be private.
    friend T;           // Ensure only a type derived from T has access.
};
```

Generally, whenever we are using the CRTP, making just the default constructor of the base-class template **private** and having it befriend its type parameter is typically a trivial local change, is helpful in avoiding various forms of accidental misuse, and is unlikely to induce long-distance friendships where none previously existed: Applying extended **friend** syntax to an existing CRTP is typically *safe*.

Compile-time visitor using the curiously recurring template pattern

As more real-world applications of compile-time polymorphism using the CRTP, consider implementing traversal and visitation of complex data structures. In particular, we want to facilitate employing *default-action* functions, which allow for simpler code from the point of view of the programmer who needs the results of the traversal. We illustrate our compile-time visitation approach using binary trees as our data structure.

We begin with the traditional node structure of a binary tree, where each node has a left and right subtree plus a label:

```
struct Node
{
    Node* d_left;
    Node* d_right;
    char  d_label;  // label will be used in the pre-order example.

    Node() : d_left(0), d_right(0), d_label(0) { }
};
```

Now we wish to have code that traverses the tree in one of the three traditional ways: *pre-order*, *in-order*, *post-order*. Such traversal code is often intertwined with the actions to be taken. In our implementation, however, we will write a CRTP-like base-class template, `Traverser`, that implements empty stub functions for each of the three traversal types, relying on the CRTP-derived type to supply the desired functionality:

```
template <typename T>
class Traverser
{
private:
    Traverser() = default;  // Make the default constructor private.
    friend T;               // Grant access only to the derived class.

public:
    void visitPreOrder(Node*)  { }  // stub-functions & placeholders
    void visitInOrder(Node*)   { }  // (Each of these three functions
    void visitPostOrder(Node*) { }  // defaults to an inline "no-op.")

    void traverse(Node* n)  // factored subfunctionality
    {
        T *t = static_cast<T*>(this);  // Cast this to the derived type.

        if (n) { t->visitPreOrder(n);     }  // optionally defined in derived
        if (n) { t->traverse(n->d_left);  }  //     "         "    "     "
        if (n) { t->visitInOrder(n);      }  //     "         "    "     "
```

```
        if (n) { t->traverse(n->d_right); }  //     "         "    "     "
        if (n) { t->visitPostOrder(n);    }  //     "         "    "     "
    }
};
```

The factored traversal mechanism is implemented in the `Traverser` base-class template. A proper subset of the four customization points, that is, the four member functions invoked from the *public* `traverse` function of the `Traverser` base class, are implemented as appropriate in the derived class, identified by `T`. Each of these customization functions is invoked in order. Notice that the `traverse` function is safe to call on a `nullptr` as each individual customization-function invocation will be independently bypassed if its supplied `Node` pointer is null. If a customization function is defined in the derived class, that version of it is invoked; otherwise, the corresponding empty **`inline`** base-class version of that function is invoked instead. This approach allows for any of the three traversal orders to be implemented simply by supplying an appropriately configured derived type where clients are obliged to implement only the portions they need. Even the traversal itself can be modified, as we will soon see, where we create the very data structure we're traversing.

Let's now look at how derived-class authors might use this pattern. First, we'll write a traversal class that fully populates a tree to a specified depth:

```
struct FillToDepth : Traverser<FillToDepth>
{
    using Base = Traverser<FillToDepth>;  // similar to a local typedef

    int d_depth;            //  final "height" of the tree
    int d_currentDepth;     //  current distance from the root

    FillToDepth(int depth) : d_depth(depth), d_currentDepth(0) { }

    void traverse(Node*& n)
    {
        if (d_currentDepth++ < d_depth && !n)  // descend; if not balanced...
        {
            n = new Node;     // Add node since it's not already there.
        }

        Base::traverse(n);    // Recurse by invoking the *base* version.

        --d_currentDepth;     // Ascend.
    }
};
```

The derived class's version of the `traverse` member function acts as if it overrides the `traverse` function in the base-class template and then, as part of its re-implementation, defers to the base-class version to perform the actual traversal.

Importantly, note that we have re-implemented `traverse` in the derived class with a function by the same name but having a *different* **signature** that has more capability (i.e., it's able to modify its immediate argument) than the one in the base-class template. In practice, this signature modification is something we would do rarely, but part of the flexibility of this design pattern, as with templates in general, is that we can take advantage of **duck typing** to achieve useful functionality in somewhat unusual ways. For this pattern, the designers of the base-class template and the designers of the derived classes are, at least initially, likely to be the same people, and they will arrange for these sorts of signature variants to work correctly if they need such functionality. Or they may decide that overridden methods should follow a proper contract and signature that they determine is appropriate, and they may declare improper overrides to be undefined behavior. In this example, we aim for illustrative flexibility over rigor.

```
void traverse(Node* n);   // as declared in the Traverser base-class template
void traverse(Node*& n);  // as declared in the FillToDepth derived class
```

Unlike virtual functions, the signatures of corresponding functions in the base and derived classes need not match exactly *provided* the derived-class function can be called in the same way as the corresponding one in the base class. In this case, the compiler has all the information it needs to make the call properly:

```
static_cast<FillToDepth *>(this)->traverse(n);  // what the compiler sees
```

Suppose that we now want to create a type that labels a *small* tree, balanced or not, according to its pre-order traversal:

```
struct PreOrderLabel : Traverser<PreOrderLabel>
{
    char d_label;

    PreOrderLabel() : d_label('a') { }

    void visitPreOrder(Node* n)  // This choice controls traversal order.
    {
        n->d_label = d_label++;
            // Each successive label is sequential alphabetically.
    }
};
```

The simple pre-order traversal class, `PreOrderLabel`, labels the nodes such that it visits each parent *before* it visits either of its two children.

Alternatively, we might want to create a read-only derived class, `InOrderPrint`, that simply prints out the sequence of labels resulting from an *in-order* traversal of the, e.g., previously pre-ordered, labels:

```
#include <cstdio>  // std::putchar

struct InOrderPrint : Traverser<InOrderPrint>
{
    ~InOrderPrint()
    {
        std::putchar('\n');  // print single newline at end of string
    }

    void visitInOrder(const Node* n) const
```

```
    {
        std::putchar(n->d_label);  // Print the label character exactly as is.
    }
};
```

The simple `InOrderPrint`-derived class, shown in the example above, prints out the labels of a tree *in order*: left subtree, then node, then right subtree. Notice that since we are only examining the tree here — not modifying it — we can declare the overriding method to take a `const Node*` rather than a `Node*` and make the method itself `const`. Once again, compatibility of signatures, not identity, is the key.

Finally, we might want to clean up the tree. We do so in *post-order* since we do not want to delete a node before we have cleaned up its children!

```
struct CleanUp : Traverser<CleanUp>
{
    void visitPostOrder(Node*& n)
    {
        delete n;  // always necessary
        n = 0;     // might be omitted in a "raw" version of the type
    }
};
```

Putting it all together, we can create a `main` program that creates a balanced tree to a depth of four and then labels it in *pre-order*, prints those labels in *in-order*, and destroys it in *post-order*:

```
int main()
{
    Node* n = 0;                  // tree handle

    FillToDepth(4).traverse(n);   // (1) Create balanced tree.
    PreOrderLabel().traverse(n);  // (2) Label tree in pre-order.
    InOrderPrint().traverse(n);   // (3) Print labels in order.
    CleanUp().traverse(n);        // (4) Destroy tree in post-order.
    return 0;
}
```

Running this program results in a binary tree of height 4, as illustrated in the code snippet below, and has reliably consistent output:

```
dcebgfhakjlinmo

Level 0:                                a
                              .    '         '    .
Level 1:                b '                           ' i
                   .  '    '  .                   .  '    '  .
Level 2:        c                f              j              m
              /   \            /   \          /   \          /   \
Level 3:    d       e        g       h      k       l      n       o
```

This use of the CRTP for traversal truly shines when the data structure to be traversed is especially complex, such as an abstract-syntax-tree (AST) representation of a computer program, where tree nodes have many different types, with each type having custom ways of representing the subtrees it contains. For example, a translation unit is a sequence of declarations; a declaration can be a type, a variable, or a function; functions have return types, parameters, and a compound statement; the statement has substatements, expressions and so on. We would not want to rewrite the traversal code for each new application. Given a reusable CRTP-based traverser for our AST, we don't have to.

For example, consider writing a type that visits each integer literal node in a given AST:

```
struct IntegerLiteralHandler : AstTraverser<IntegerLiteralHandler>
{
    void visit(IntegerLiteral* iLit)
    {
        // ... (do something with this integer literal)
    }
};
```

The AST traverser, which would implement a separate empty `visit` overload for each syntactic node type in the grammar, would invoke our derived `visit` member function with every integer literal in the program, regardless of where it appeared. This CRTP-based traverser would also call many other `visit` methods, but each of those would perform no action at all by default and would likely be elided at even modest compiler-optimization levels. Be aware, however, that, although we ourselves are not rewriting the traversal code each time, the compiler is still doing it because every CRTP instantiation produces a new copy of the traversal code. If the traversal code is large and complex, the consequence might be increased program size, that is, **code bloat**.

Finally, the CRTP can be used in a variety of situations for many purposes,[9] hence its *curiously recurring* nature. Those uses invariably benefit from (1) declaring the base-class template's default constructor *private* and (2) having that template befriend its type parameter, which is possible only by means of the extended **`friend`** syntax. Thus, the CRTP base-class template can ensure, at compile time, that its type argument is actually derived from the base class as required by the pattern.

[9] **?**

Explicit Instantiation Declarations

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

Description

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

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

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

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

Explicit-instantiation definition

The ability to create an **explicit-instantiation definition** has been available since C++98.[1] The requisite syntax is to place the keyword **template** in front of the name of the fully specialized class template, function template, or — in C++14 — variable template (see Section 1.2."Variable Templates" on page 144):

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

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

This explicit-instantiation directive compels the compiler to instantiate *all* functions defined by the named `std::vector` class template having the specified **int** template argument; any

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

collateral object code resulting from these instantiations will be deposited in the resulting `.o` file for the current translation unit. Importantly, even functions that are never used are still specialized, so this might not be the correct solution for many classes; see *Potential Pitfalls — Accidentally making matters worse* on page 279.

Explicit-instantiation declaration

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

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

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

The use of the modern **`extern template`** syntax above instructs the compiler to specifically *not* deposit any object code for the named specialization in the current translation unit and instead to rely on some other translation unit to provide any missing object-level definitions that might be needed at link time; see *Annoyances — No good place to put definitions for unrelated classes* on page 279.

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

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

extern template class std::vector<int>;  // specialization for int elements

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

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

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

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

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

char     c;
short    s;
int      i;
unsigned u;

bool bc = f(c);  // exact match: Object code is suppressed locally.
bool bs = f(s);  // not exact match: Object code is generated locally.
bool bi = f(i);  // exact match: Object code is suppressed locally.
bool bu = f(u);  // not exact match: Object code is generated locally.
```

As the example above illustrates, overload resolution and template parameter deduction occur independently of any explicit-instantiation declarations. Only *after* the template to be instantiated is determined does the **`extern template`** syntax take effect; see also *Potential Pitfalls — Corresponding explicit-instantiation declarations and definitions* on page 277.

A more complete illustrative example

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

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

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

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

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

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

public:
    // ...
```

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

    // ...
};

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

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

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

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

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

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

// ... We may opt to add explicit-instantiation declarations here;
//     see the next code example.

}  // close my namespace

#endif  // close internal include guard
```

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

1. a **`class`** template, `Vector`, parameterized on element type
2. a free-function template, `swap`, that operates on objects of corresponding specialized `Vector` type
3. a **`const`** C++14 variable template, `vectorSize`, that represents the number of bytes in the **footprint** of an object of the corresponding specialized `Vector` type

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

To eliminate object code for specializations of entities in the `my_vector` component, we must first decide where the unique definitions will go; see *Annoyances — No good place* *to put definitions for unrelated classes* on page 279. In this specific case, however, we own the component that requires specialization, and the specialization is for a ubiquitous built-in type; hence, the natural place to generate the specialized definitions is in a `.cpp` file corresponding to the component's header:

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

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

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

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

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

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

//template std::size_t Vector<int>::s_count = 0;
    // Generate the object-code-level definition for the specialization of the
    // static member variable of Vector instantiated for built-in type int.

}; // close my namespace
```

Each of the constructs introduced by the keyword **`template`** within the `my` namespace in the code snippet above represents a separate explicit-instantiation definition. These constructs instruct the compiler to generate object-level definitions for general templates declared in `my_vector.h` specialized on the built-in type **`int`**.

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

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

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

```
{

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

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

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

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

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

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

}  // close my namespace

#endif  // close internal include guard
```

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

The effect on various `.o` files

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

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

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

template <typename T>  // elided sketch of a class template
class Interval
{
```

```
    T d_low;   // interval's low value
    T d_high;  // interval's high value

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

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

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

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

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

    // ...
};

template <typename T>                  // elided sketch of a function template
bool intersect(const Interval<T>& i1, const Interval<T>& i2)
    // Determine whether the specified intervals intersect ...
{
    bool result = false;  // nonintersecting until proven otherwise
    // ...
    return result;
}

}  // close lib namespace

#endif  // INCLUDED_LIB_INTERVAL


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

This library component defines, in the namespace `lib`, a heavily elided sketch of an implementation of (1) a class template, `Interval`, and (2) a function template, `intersect`, the only practical purpose of which is to provide some sample template source code to compile.

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

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

int main(int argv, const char** argc)
{
    lib::Interval<double> a(0, 5);  // instantiate with double type parameter
```

```
    lib::Interval<double> b(3, 8);  // instantiate with double type parameter
    lib::Interval<int>    c(4, 9);  // instantiate with int    type parameter

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

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

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

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

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

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

lib_interval.o:
```

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

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

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

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

As the textual output above confirms, each of the needed *weak* template symbols, previously marked with a `W`, is bound into the final program as a *strong* symbol, now — like `main` — marked with a `T`.[3] In this tiny illustrative example, only one set of weak symbols appeared in the combined `.o` files.

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

Imagine now that our program includes a large number of `.o` files, many of which make use of our `lib_interval` component, particularly to operate on **`double`** intervals. Suppose, for now, we decide we would like to suppress the generation of object code for templates related to just **`double`** types with the intent of later putting them all in one place, i.e., the currently empty `lib_interval.o`. Achieving this objective is precisely what the **`extern template`** syntax is designed to accomplish.

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

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

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

template <typename T>                    // elided sketch of a class template
class Interval;                          // ...
    // ...                       (same as before)

template <typename T>                    // elided sketch of a function template
bool intersect(const Interval<T>& i1, const Interval<T>& i2);
    // ...                       (same as before)

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

[3]Whether the symbol is marked `W` or `T` in the final executable is implementation specific and of no consequence here. We present these concepts in this particular way to aid cognition.

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

}  // close lib namespace

#endif  // INCLUDED_LIB_INTERVAL
```

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

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

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

lib_interval.o:
```

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

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

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

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

To provide the missing definitions, we will need to instantiate them explicitly. Since the type for which the class and function are being specialized is the ubiquitous built-in type, **`double`**, the ideal place to sequester those definitions would be within the `.o` file of the `lib_interval` library component itself, but see *Annoyances — No good place to put definitions for unrelated classes* on page 279. To force the needed template definitions into the `lib_interval.o` file, we will need to pull out our trusty explicit-instantiation definition syntax, i.e., the **template** prefix:

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

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

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

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

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

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

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

The application `.o` file, `app.o`, naturally remained unchanged. What's new here is that the functions that were missing from the `app.o` file are now available in the `lib_interval.o` file, again as *weak* (`W`), as opposed to strong (`T`), symbols. Notice, however, that explicit instantiation forces the compiler to generate code for all of the member functions of the class template for a given specialization. These symbols might all be linked into the resulting executable unless we take explicit precautions to exclude those that aren't needed[4]:

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

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

```
00000000004004b7 T main
```

tl;dr: This **extern template** feature is provided to enable software architects to reduce code bloat in individual .o files for common instantiations of class, function, and, as of C++14, variable templates in large-scale C++ software systems. The practical benefit is in reducing the physical size of libraries, which *might* lead to improved link times. Explicit-instantiation declarations do *not* (1) affect the meaning of a program, (2) suppress template instantiation, (3) impede the compiler's ability to **inline**, or (4) meaningfully improve compile time. To be clear, the *only* purpose of the **extern template** syntax is to suppress object-code generation for the current translation unit, which is then selectively overridden in the translation unit(s) of choice.

Use Cases

Reducing template code bloat in object files

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

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

#include <cstddef>  // std::size_t
namespace game  // namespace for all entities defined within this component
{

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

template <typename T, std::size_t N>
T dot(const FixedArray<T, N>& a, const FixedArray<T, N>& b) { /*...*/ }

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

extern template class FixedArray<int, 2>;              // class template
```

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

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

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

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

}  // close game namespace

#endif  // INCLUDED_GAME_FIXEDARRAY
```

Specializations commonly used by the `game` project are provided by the `game` library. In the component header in the example above, we have used the **`extern template`** syntax to suppress object-code generation for instantiations of both the class template `FixedArray` and the function template `dot` for element types **`int`** and **`double`**, each for array sizes `2` and `3`. To ensure that these specialized definitions are available in every program that might need them, we use the **`template`** syntax counterpart to *force* object-code generation within just the one `.o` corresponding to the `game_fixedarray` library component[5]:

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

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

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

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

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

[5]Notice that we have chosen *not* to nest the explicit specializations (or any other definitions) of entities already declared directly within the `game` namespace, preferring instead to qualify each entity explicitly to be consistent with how we render free-function definitions (to avoid self-declaration); see **?**, section 2.5, "Component Source-Code Organization," pp. 333–342, specifically Figure 2-36b, p. 340. See also *Potential Pitfalls — Corresponding explicit-instantiation declarations and definitions* on page 277.

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

Compiling `game_fixedarray.cpp` and examining the resulting object file shows that the code for all explicitly instantiated classes and free functions was generated and placed into the object file, `game_fixedarray.o`[6]:

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

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

```
// app_geometryutil.cpp:

#include <game_fixedarray.h>
#include <game_unit.h>

using namespace game;

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

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

[6]Note that only a subset of the relevant symbols have been retained.

```
    // ...
}

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

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

The second one deals with physics calculations:

```
// app_physics.cpp:

#include <game_fixedarray.h>
#include <game_unit.h>

using namespace game;

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

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

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

```
$ nm -C app_geometryutil.o
000000000000003e T isOrthogonal(game::FixedArray<int, 2ul> const&,
  game::FixedArray<int, 2ul> const&)
0000000000000068 T isOrthogonal(game::FixedArray<int, 3ul> const&,
  game::FixedArray<int, 3ul> const&)
0000000000000000 T translate(game::Unit*, game::FixedArray<double, 2ul> const&)
000000000000001f T translate(game::Unit*, game::FixedArray<double, 3ul> const&)
                 U game::FixedArray<double, 2ul>::FixedArray()
```

```
                 U game::FixedArray<double, 3ul>::FixedArray()
                 U int game::dot<int, 2ul>(game::FixedArray<int, 2ul> const&,
  game::FixedArray<int, 2ul> const&)
                 U int game::dot<int, 3ul>(game::FixedArray<int, 3ul> const&,
  game::FixedArray<int, 3ul> const&)

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

Whether optimization involving explicit-instantiation directives reduces library sizes on disc has no noticeable effect or actually makes matters worse will depend on the particulars of the system at hand. Having this optimization applied to frequently used templates across a large organization has been known to decrease object file sizes, storage needs, link times, and overall build times, but see *Potential Pitfalls — Accidentally making matters worse* on page 279.

Insulating template definitions from clients

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

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

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

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

#endif
```

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

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

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

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

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

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

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

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

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

#endif
```

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

Potential Pitfalls

Corresponding explicit-instantiation declarations and definitions

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

1. Each general template, `C<T>`, whose object code bloat is to be optimized must be declared within some designated component's header file, `c.h`.

2. The specific definition of each `C<T>` relevant to an explicit specialization being optimized — including general, partial-specialization, and full-specialization definitions — must appear in the header file prior to its corresponding explicit-instantiation declaration.

3. Each explicit-instantiation declaration for each specialization of each separate top-level — i.e., class, function, or variable — template must appear in the component's

`.h` file *after* the corresponding general template declaration and the relevant general, partial-specialization, or full-specialization definition, but, in practice, always after *all* such definitions, not just the relevant one.

4. Each template specialization having an explicit-instantiation declaration in the header file must have a corresponding explicit-instantiation definition in the component's implementation file, `c.cpp`.

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

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

Finally, unless we provide a matching explicit-instantiation definition in the `c.cpp` file for each and every corresponding explicit-instantiation declaration in the `c.h` file as in item (4), our optimization attempts might well result in a library component that compiles, links, and even passes some unit tests but, when released to our clients, fails to link. Additionally, any explicit-instantiation definition in the `c.cpp` file that is not accompanied by a corresponding explicit-instantiation declaration in the `c.h` file will inflate the size of the `c.o` file with no possibility of reducing code bloat in client code[7]:

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

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

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

#endif


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

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


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

[7]Fortunately, these extra instantiations do not result in multiply-defined symbols because they still reside in their own sections and are marked as *weak* symbols.

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

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

In the example above, `f(i)` works as expected, with the linker finding the definition of `f<int>` in `c.o`; `f(c)` fails to link, because no definition of `f<c>` is guaranteed to be found anywhere; and `f(d)` accidentally works by silently generating a *redundant* local copy of `f<double>` in `client.o` while another, identical definition is generated explicitly in `c.o`. Importantly, note that **`extern template`** has *absolutely no affect* on overload resolution because the call to `f(c)` did *not* resolve to `f<int>`.

Accidentally making matters worse

When making the decision to preinstantiate common specializations of popular templates within some designated `.o` file, one must consider that not all programs necessarily need every (or even any) such instantiation. Special consideration should be given to classes that have many methods but typically use only a few.

The language feature is sufficiently flexible that one can suppress and preinstantiate just one or a handful of member functions of such a type. Intuition is all well and good, but measurement simply has no substitute.

If one suspects that explicit-instantiation directives might profitably reduce the size of libraries resulting from object code that is bloated due to redundant local reinstantiations of popular templates on common types, measuring before and after and retaining the change *only* if it offers a significant — at least measurable — improvement avoids complicating the codebase without a verifiable return on the investment. Finally, remember that one might need to explicitly tell the linker to strip unused sections resulting, for example, from forced instantiation of common template specializations, to avoid inadvertently bloating executables, which could adversely affect load times.

Annoyances

No good place to put definitions for unrelated classes

When we consider the implications of physical dependency,[8,9] determining in which component to deposit the specialized definitions can be problematic. For example, consider a codebase implementing a core library that provides both a nontemplated `String` class and a `Vector` container class template. These fundamentally unrelated entities would ideally live

[8] See **?**

[9] See **?**

in separate physical **components** — i.e., `.h`/`.cpp` pairs — neither of which depends physically on the other. That is, an application using just one of these components could ideally be compiled, linked, tested, and deployed entirely independently of the other. Now, consider a large codebase that makes heavy use of `Vector<String>`: In what component should the object-code-level definitions for the `Vector<String>` specialization reside?[10] There are two obvious alternatives:

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

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

Another possibility might be to create a third component, call it `stringvector`, that itself depends on both `vector` and `string`. By **escalating**[11] the mutual dependency to a higher level in the physical hierarchy, we avoid forcing any client to depend on more than what is actually needed. The practical drawback to this approach is that only those clients that proactively include the composite `stringvector.h` header would realize any benefit; fortunately, in this case, there is no **one-definition rule (ODR)** issue if they don't.

Finally, complex machinery could be added to both `string.h` and `vector.h` to conditionally include `stringvector.h` whenever both of the other headers are included; such heroic efforts would, nonetheless, involve a **cyclic physical dependency** among all three of these components. Circular intercomponent collaborations are best avoided.[12]

All members of an explicitly defined template class must be valid

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

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

```
template <typename T>
class W
{
```

[10] Note that the problem of determining in which component to instantiate the object-level implementation of a template for a user-defined type is similar to that of specializing an arbitrary user-defined trait for a user-defined type.

[11] **?**, section 3.5.2, "Escalation," pp. 604–614

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

```
    T d_t;  // a data member of type T

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

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

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

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

```
struct U
{
    U(int i) { /* do something with i */ }
    // ...
};

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

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

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

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

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

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

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

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

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

See Also

- “Variable Templates” (Section 1.2, p. 144) ⧫ Extension of the template syntax for defining a family of like-named variables or static data members that can be instantiated explicitly

Further Reading

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

Forwarding && References

A forwarding reference (`T&&`) — distinguishable from an rvalue reference (`&&`) (see "*rvalue* References" on page 310) based only on context — is a distinct, special kind of reference that (1) binds universally to the result of an expression of *any* **value category** and (2) preserves aspects of that **value category** so that the bound object can be *moved from*, if appropriate.

Description

Sometimes we want the same reference to be able to bind to either an *lvalue* or an *rvalue* and then later be able to discern, from the reference itself, whether the result of the original expression was eligible to be *moved from*. A *forwarding reference* (e.g., `forRef` in the example below) used in the interface of a function *template* (e.g., `myFunc`) affords precisely this capability and will prove invaluable for the purpose of conditionally moving, or else copying, an object from within the function template's body:

```
template <typename T>
void myFunc(T&& forRef)
{
    // It is possible to check if forRef is eligible to be moved from or not
    // from within the body of myFunc.
}
```

In the definition of the `myFunc` function template in the example above, the parameter `forRef` syntactically appears to be a non-`const` reference to an *rvalue* of type `T`; in this very precise context, however, the same `T&&` syntax designates a **forwarding reference**, with the effect of retaining the original value category of the object bound to the argument `forRef`; see *Description: Identifying forwarding references* on page 287. The `T&&` syntax represents a *forwarding* reference — as opposed to an *rvalue* reference — whenever an *individual* function template has a type parameter (e.g., `T`) and an unqualified function parameter of type that is exactly `T&&` (e.g., `const T&&` would be an *rvalue* reference, not a forwarding reference).

Consider, for example, a function `f` that takes a single argument by reference and then attempts to use it to invoke one of two overloads of a function `g`, depending on whether the original argument was an *lvalue* or *rvalue*:

```
struct S { /* some UDT that might benefit from being able to be moved */ };

void g(const S&);  // target function - overload for const int lvalues
void g(S&&);       // target function - overload for int rvalues only

template <typename T>
void f(T&& forRef); // forwards to target overload g based on value category
```

In theory, we could have chosen a non-`const` *lvalue* reference along with a modifiable *rvalue* reference here for *pedagogical* symmetry; such an inherently unharmonious overload set would, however, not typically occur in practice; see "*rvalue* References" on page 310. In this specific case — where `f` is a function template, `T` is a template type parameter, and the type of the parameter itself is exactly `T&&` — the `forRef` function parameter (in the code snippet above) denotes a *forwarding reference*. If `f` is invoked with an *lvalue*, `forRef` is an *lvalue* reference; otherwise `forRef` is an *rvalue* reference. Given the dual nature of `forRef`, one rather verbose way of determining the original value category of the passed argument would be to use the `std::is_lvalue_reference` **type trait** on `forRef` itself:

```
#include <type_traits>  // std::is_lvalue_reference
#include <utility>  // std::move

template <typename T>
void f(T&& forRef)      // forRef is a forwarding reference.
{
    if (std::is_lvalue_reference<T>::value)  // using a C++11 type trait
    {
        g(forRef);             // propagates forRef as an *lvalue*
    }                          // invokes g(const S&)
    else
    {
        g(std::move(forRef));  // propagates forRef as an *rvalue*
    }                          // invokes g(S&&)
}
```

The `std::is_lvalue_reference<T>::value` predicate above asks the question, "Did the object bound to `fRef` originate from an lvalue expression?" and allows the developer to branch on the answer. A more concise but otherwise equivalent implementation is generally preferred; see *Description: The* `std::forward` *utility* on page 290:

```
#include <utility>  // std::forward

template <typename T>
void f(T&& forRef)
{
    g(std::forward<T>(forRef));
        // same as g(std::move(forRef)) if and only if forRef is an *rvalue*
        // reference; otherwise, equivalent to g(forRef)
}
```

A `client` function invoking `f` will enjoy the same behavior with either of the two implementation alternatives offered above:

```
void client()
{
    S s;
    f(s);    // Instantiates f<S&> -- fRef is an lvalue reference (S&).
             // The function f<S&> will end up invoking g(S&).

    f(S());  // Instantiates f<S&&> -- fRef is an rvalue reference (S&&).
             // The function f<S&&> will end up invoking g(S&&).
}
```

Use of `std::forward` in combination with forwarding references is typical in the implementation of industrial-strength generic libraries; see *Use Cases* on page 290.

A brief review of function template argument deduction

Invoking a function template without explicitly providing template arguments at the call site will compel the compiler to attempt, if possible, to *deduce* those template *type* arguments from the function arguments:

```
template <typename T> void f();
template <typename T> void g(T x);
template <typename T> void h(T y, T z);

void example0()
{
    f();        // Error, couldn't infer template argument T
    g(0);       // OK, T deduced as int from literal 0 -- x is an int.
    h(0, 'a');  // Error, deduced conflicting types for T (int vs. char)
}
```

Any **cv-qualifiers** (`const`, `volatile`, or both) on a *deduced* function parameter will be applied *after* type deduction is performed:

```
template <typename T> void cf(const T x);
template <typename T> void vf(volatile T y);
template <typename T> void wf(const volatile T z);

void example1()
{
    cf(0);  // OK, T deduced as int -- x is a const int.
    vf(0);  // OK, T deduced as int -- y is a volatile int.
    wf(0);  // OK, T deduced as int -- z is a const volatile int.
}
```

Similarly, **ref-qualifiers** other than `&&` (`&` or `&&` along with any cv-qualifier) do not alter the deduction process, and they too are applied after deduction:

```
template <typename T> void rf(T& x);
template <typename T> void crf(const T& x);

void example2()
{
    int i;
    rf(i);   // OK, T is deduced as int -- x is an int&.
    crf(i);  // OK, T is deduced as int -- x is a const int&.

    rf(0);   // Error, expects an lvalue for 1st argument
    crf(0);  // OK, T is deduced as int -- x is a const int&.
}
```

Type deduction works differently for *forwarding* references where the only qualifier on the template argument is `&&`. For the sake of exposition, consider a function template declaration, `f`, accepting a forwarding reference, `forRef`:

```
template <typename T> void f(T&& forRef);
```

We have seen in the example on page 284 that, when `f` is invoked with an *lvalue* of type `S`, then `T` is deduced as `S&` and `forRef` becomes an *lvalue* reference. When `f` is instead invoked with an *xvalue* of type `S`, then `T` is deduced as `S` and `forRef` becomes an *rvalue* reference. The underlying process that results in this "duality" relies on a special rule (known as **reference collapsing**; see the next section) introduced as part of **type deduction**. When the type `T` of a *forwarding* reference is being deduced from an expression `E`, `T` itself will be deduced as an *lvalue* reference if `E` is an *lvalue*; otherwise normal type-deduction rules will apply and `T` will be deduced as an *rvalue* reference:

```
void g()
{
    int i;
    f(i);  // i is an *lvalue* expression.
           // T is therefore deduced as int& -- special rule!
           // T&& becomes int& &&, which collapses to int&.

    f(0);  // 0 is an *rvalue* expression.
           // T is therefore deduced as int.
           // T&& becomes int&&, which is an *rvalue* reference.
}
```

For more on general type deduction, see "**`auto`** Variables" on page 177.

Reference collapsing

As we saw in the previous section, when a function having a *forwarding* reference parameter, `forRef`, is invoked with a corresponding *lvalue* argument, an interesting phenomenon occurs: After type deduction, we temporarily get what appears syntactically to be an *rvalue* reference to an *lvalue* reference. As references to references are *not* allowed in C++, the compiler employs **reference collapsing** to resolve the *forwarding*-reference parameter, `forRef`, into a single reference, thus providing a way to infer, from `T` itself, the original **value category** of the argument passed to `f`.

The process of **reference collapsing** takes place automatically in any situation where a reference to a reference is formed. Table 1 illustrates the simple rules for collapsing "unstable" references into "stable" ones. Notice, in particular, that an *lvalue* reference always overpowers an *rvalue* reference. The only situation in which two references collapse into an *rvalue* reference is when they are both *rvalue* references.

Finally, note that it is not possible to write a reference-to-reference type in C++ explicitly:

```
int    i   = 0;   // OK
int&   ir  = i;   // OK
int& & irr = ir;  // Error, irr declared as a reference to a reference
```

Table 1: Collapsing "unstable" reference pairs into a single "stable" one

1st Reference Type	2nd Reference Type	Result of Reference Collapsing
&	&	&
&	&&	&
&&	&	&
&&	&&	&&

It is, however, easy to do so with type aliases and template parameters, and that is where reference collapsing comes into play:

```
#include <type_traits>  // std::is_same)
using i = int&;  // OK
using j = i&;    // OK, int& & becomes int&.
static_assert(std::is_same<j,int&>::value);
```

During computations involving **metafunctions**, or as part of language rules (such as type deduction), however, references to references can occur spontaneously:

```
template <typename T>
struct AddLvalueRef { typedef T& type; };
    // metafunction that transforms to an *lvalue* reference to T

template <typename T>
void f(T input)
{
    typename AddLvalueRef<T>::type ir1 = input;    // OK, adds & to make T&
    typename AddLvalueRef<T&>::type ir2 = input;   // OK, collapses to T&
    typename AddLvalueRef<T&&>::type ir3 = input;  // OK, collapses to T&
}
```

Notice that we are using the `typename` keyword in the example above as a generalized way of indicating, during **template instantiation**, that a dependent name is a type (as opposed to a value).

Identifying forwarding references

The syntax for a *forwarding* reference (`&&`) is the same as that for *rvalue* references; the only way to discern one from the other is by observing the surrounding context. When used in a manner where **type deduction** can take place, the `T&&` syntax does *not* designate an *rvalue* reference; instead, it represents a *forwarding* reference; for type deduction to be in effect, an *individual* (possibly member) function *template* must have a type parameter (e.g., `T`) and a function parameter of type that exactly matches that parameter followed by `&&` (e.g., `T&&`):

```
struct S0
{
    template <typename T>
    void f(T&& forRef);
```

```
        // OK, fully eligible for template-argument type deduction: forRef
        // is a forwarding reference.
};
```

Note that if the function parameter is qualified, the syntax reverts to the usual meaning of *rvalue* reference:

```
struct S1
{
    template <typename T>
    void f(const T&& crRef);
        // Eligible for type deduction but is not a forwarding reference: due
        // to the const qualifier, crRef is an *rvalue* reference.
};
```

If a member function of a class template is not itself also a template, then its template type parameter will not be deduced:

```
template <typename T>
struct S2
{
    void f(T&& rRef);
       // Not eligible for type deduction because T is fixed and known as part
       // of the instantiation of S2: rRef is an *rvalue* reference.
};
```

More generally, note that the `&&` syntax can *never* imply a *forwarding* reference for a function that is not itself a template; see *Annoyances: Forwarding references look just like rvalue references* on page 301.

`auto&&` — a forwarding reference in a non-parameter context

Outside of template function parameters, *forwarding* references can also appear in the context of variable definitions using the `auto` variable (see "**`auto`** Variables" on page 177) because they too are subject to type deduction:

```
void f()
{
    auto&& i = 0;  // i is a forwarding reference because the type of i must
                   // be deduced from the initialization expression 0.
}
```

Just like function parameters, `auto&&` resolves to either an *lvalue* reference or *rvalue* reference depending on the **value category** of the initialization expression:

```
void g()
{
    int i;
    auto&& lv = i;  // lv is an int&.

    auto&& rv = 0;  // rv is an int&&.
}
```

Similarly to `const auto&`, the `auto&&` syntax binds to anything. In `auto&&`, however, the `const`-ness of the bound object is *preserved* rather than always enforced:

```
void h()
{
    int       i = 0;
    const int ci = 0;

    auto&& lv  = i;   // lv is an int&.
    auto&& clv = ci;  // clv is a const int&.
}
```

Just as with function parameters, the original **value category** of the expression used to initialize a *forwarding* reference variable can be propagated during subsequent function invocation – e.g., using `std::forward` (see *Description: The* `std::forward` *utility* on page 290):

```
template <typename T>
void use(T&& t); // Here use also takes a forwarding reference parameter
                 // to do with as it pleases.

template <typename T>
void l()
{
    auto&& fr = get<T>();
        // get<T>() might be either an *lvalue* or *rvalue* depending on T.

    use(std::forward<decltype(fr)>(fr));  // decltype is a C++11 feature.
        // Propagate the original value category of get<T>() into use.
}
```

Notice that, because (1) `std::forward` (see the next section) requires the type of the object that's going to be forwarded as a user-provided template argument and (2) it is not possible to name the type of `fr`, `decltype` (see "`decltype`" on page 27) was used in the example above to retrieve the type of `fr`.

Forwarding references without forwarding

Sometimes deliberately *not* forwarding (see *Description: The* `std::forward` *utility* on page 290) an `auto&&` variable or a forwarding reference function parameter at all can be useful, instead employing *forwarding* references solely for their `const`-preserving and universal binding semantics. As an example, consider the task of obtaining iterators over a range of an unknown **value category**:

```
#include <iterator>  // std::begin, std::end,

template <typename T>
void m()
{
    auto&& r = getRange<T>();
        // getRange<T>() might be either an lvalue or rvalue depending on T.
```

```
    auto b = std::begin(r);
    auto e = std::end(r);

    traverseRange(b, e);
}
```

Using `std::forward` in the initialization of both `b` and `e` might result in moving from `r` twice, which is potentially unsafe (see "*rvalue* References" on page 310). Forwarding `r` only in the initialization of `e` might avoid issues caused by moving an object twice but might result in inconsistent behavior with `b`, especially if the implementation of `r` makes use of reference qualifiers (see "Ref-Qualifiers" on page 371).

The `std::forward` utility

The final piece of the forwarding reference puzzle is the `std::forward` utility function. Since the expression naming a forwarding reference `x` is always an *lvalue* — due to its reachability either by name or address in virtual memory — and since our intention is to move `x` in case it was an *rvalue* to begin with, we need a conditional *move* operation that will move `x` only in that case and otherwise let `x` pass through as an *lvalue*.

The declaration for `std::forward<T>` is as follows (in `<utility>`):

```
template <class T> T&& forward(typename remove_reference<T>::type& t) noexcept;
template <class T> T&& forward(typename remove_reference<T>::type&& t) noexcept;
```

The second overload is ill-formed if invoked when `T` is an *lvalue* reference type.

Remember that the type `T` associated with a forwarding reference is deduced as a reference type if given an *lvalue* reference and as a non-reference type otherwise. So for a forwarding reference `forRef` of type `T&&`, we have two cases:

- An *lvalue* of type `U` was used for initializing `forRef`, so `T` is `U&`, thus the first overload of `forward` will be selected and will be of the form `U& forward(U& u) noexcept`, thus just returning the original *lvalue* reference.

- An *rvalue* of type `U` was used for initializing `forRef`, so `T` is `U`, so the second overload of `forward` will be selected and will be of the form `U&& forward(U& u) noexcept`, essentially equivalent to `std::move`.

Note that, in the body of a function template accepting a forwarding reference `T&&` named `x`, `std::forward<T>(x)` could be replaced with `static_cast<T&&>(x)` to achieve the same effect. Due to **reference collapsing** rules, `T&&` will resolve to `T&` whenever the original **value category** of `x` was an *lvalue* and to `T&&` otherwise, thus achieving the *conditional move* behavior elucidated in *Description* on page 283. Using `std::forward` over `static_cast` will, however, ensure that the types of `T` and `x` match, preventing accidental unwanted conversions and, separately, perhaps also more clearly expressing the programmer's intent.

Use Cases

Perfectly forwarding an expression to a downstream consumer

A frequent use of forwarding references and `std::forward` is to propagate an object, whose **value category** is invocation-dependent, down to one or more service providers that will behave differently depending on the **value category** of the original argument.

As an example, consider an overload set for a function, `sink`, that accepts a `std::string` either by `const` *lvalue* reference (e.g., with the intention of *copying* from it) or *rvalue* reference (e.g., with the intention of *moving* from it):

```
void sink(const std::string& s) { target = s; }
void sink(std::string&& s)      { target = std::move(s); }
```

Now, let's assume that we want to create an intermediary function template, `pipe`, that will accept an `std::string` of any **value category** and will dispatch its argument to the corresponding overload of `sink`. By accepting a *forwarding* reference as a function parameter and invoking `std::forward` as part of `pipe`'s body, we can achieve our original goal without any code duplication:

```
template <typename T>
void pipe(T&& x)
{
    sink(std::forward<T>(x));
}
```

Invoking `pipe` with an *lvalue* will result in `x` being an *lvalue* reference and thus `sink(const std::string&)`'s being called. Otherwise, `x` will be an *rvalue* reference and `sink(std::string&&)` will be called. This idea of enabling *move* operations without code duplication (as `pipe` does) is commonly referred to as *Use Cases: Perfect forwarding for generic factory functions* on page 292.

Handling multiple parameters concisely

Suppose you have a **value-semantic type (VST)** that holds a collection of attributes where some (not necessarily proper) subset of them need to be changed together[1]:

```
#include <type_traits>  // std::enable_if
#include <utility>      // std::forward

struct Person { /* UDT that benefits from move semantics */ };

class StudyGroup
{
    Person d_a;
    Person d_b;
    Person d_c;
    Person d_d;
    // ...

public:
    bool isValid(const Person& a, const Person& b, const Person& c, const Person& d);
        // Return true if these specific people form a valid study group under
        // the guidelines of the study-group commission, and false otherwise.
```

[1]This type of value-semantic type can be classified more specifically as a *complex-constrained* attribute class; see **lakos2a**, section 4.2.

```
    // ...

    template <typename PA, typename PB, typename PC, typename PD,
        typename = typename std::enable_if<
            std::is_same<typename std::decay<PA>::type, Person>::value &&
            std::is_same<typename std::decay<PB>::type, Person>::value &&
            std::is_same<typename std::decay<PC>::type, Person>::value &&
            std::is_same<typename std::decay<PD>::type, Person>::value>::type>
    int setPersonsIfValid(PA&& a, PB&& b, PC&& c, PD&& d)
    {
        enum { e_SUCCESS = 0, e_FAIL };

        if (!isValid(a, b, c, d))
        {
            return e_FAIL;  // bad choice; no change
        }

        // Move or copy each person into this object's Person data members:
        d_a = std::forward<PA>(a);
        d_b = std::forward<PB>(b);
        d_c = std::forward<PC>(c);
        d_d = std::forward<PD>(d);

        return e_SUCCESS;  // Study group was updated successfully.
    }
};
```

The `setPersonsIfValid` function is producing the full crossproduct of instantiations for every variation of qualifiers that can be on a `Person` object. Any combination of *lvalue* and *rvalue* `Person`s can be passed, and a template will be instantiated that will copy the *lvalues* and move from the *rvalues*. To make sure the `Person` objects are created externally, the function is restricted, using `std::enable_if`, to instantiate only for types that decay to `Person` (i.e., types that are cv-qualified or ref-qualified `Person`). Because each parameter is a forwarding reference, they can all implicitly convert to `const Person&` to pass to `isValid`, creating no additional temporaries. Finally, `std::forward` is then used to do the actual moving or copying as appropriate to data members.

Perfect forwarding for generic factory functions

Consider the prototypical standard-library generic factory function, `std::make_shared<T>`. On the surface, the requirements for this function are fairly simple — allocate a place for a `T` and then construct it with the same arguments that were passed to `make_shared`. This, however, gets reasonably complex to implement efficiently when `T` can have a wide variety of ways in which it might be initialized.

For simplicity, we will show how a two-argument `my::make_shared` might be defined, knowing that a full implementation would employ variadic template arguments for this purpose — see "Variable Templates" on page 144. We will also implement a simpler version of `make_shared` that simply creates the element on the heap with `new` and constructs a std::shared_ptr to manage the lifetime of that element. The declaration of this form of make shared would be structured like this:

```
namespace my {
template <typename ELEMENT_TYPE, typename ARG1, typename ARG2>
std::shared_ptr<ELEMENT_TYPE> make_shared(ARG1&& arg1, ARG2&& arg2);
}
```

As you can see, we have two forwarding reference arguments — arg1 and arg2 — with deduced types ARG1 and ARG2. Now, the body of our function needs to carefully construct our ELEMENT_TYPE object on the heap and then create our output shared_ptr:

```
template <typename ELEMENT_TYPE, typename ARG1, typename ARG2>
std::shared_ptr<ELEMENT_TYPE> my::make_shared(ARG1&& arg1, ARG2&& arg2)
{
    ELEMENT_TYPE *element_p = new ELEMENT_TYPE(std::forward<ARG1>(arg1),
                                               std::forward<ARG2>(arg2));
    try
    {
        return std::shared_ptr<ELEMENT_TYPE>(element_p);
    }
    catch (...)
    {
        delete element_p;
        throw;
    }
}
```

Note that this simplified implementation needs to take care that the constructor for the return value does not throw, cleaning up the allocated element if that should happen; normally a **RAII** proctor to manage this ownership would be a more robust solution to this problem.

Importantly, the use of std::forward to construct the element means that the arguments passed to make_shared will be used to find the appropriate matching two-argument constructor of ELEMENT_TYPE. When those arguments are *rvalues*, the constructor found will again search for one that takes an *rvalue* and the arguments will be moved from. Even more, because this function wants to forward exactly the const-ness and reference type of the input arguments, we would have to write 12 distinct overloads for each argument if we were not using perfect forwarding – the full cross product of const (or not), volatile (or not), and &, &&, (or not). This would mean a full implementation of just this two-argument variation would require 144 distinct overloads, all almost identical and most never actually instantiated. The use of forwarding references reduces that to just 1 overload for each number of arguments.

Wrapping initialization in a generic factory function

Occasionally we might want to initialize an object with an intervening function call wrapping the actual construction of that object. Suppose we have a tracking system that we want to use to monitor how many times certain initializers have been invoked:

```
struct TrackingSystem
{
    template <typename T>
    static void trackInitialization(int numArgs);
        // Track the creation of a T with a constructor taking numArgs
        // arguments.
};
```

Now we want to write a general utility function that can be used to construct an arbitrary object and notify the tracking system of the construction for us. Here we will use a variadic pack (see "Variable Templates" on page 144) of forwarding references to handle calling the constructor for us:

```
template <class ELEMENT_TYPE, typename... ARGS>
ELEMENT_TYPE trackConstruction(ARGS&&... args)
{
    TrackingSystem::trackInitialization<ELEMENT_TYPE>(sizeof...(args));
    return ELEMENT_TYPE(std::forward<ARGS>(args)...);
}
```

This lets us add tracking easily to convert any initialization to a tracked one by inserting a call to this function around the constructor arguments:

```
void myFunction()
{
    BigObject untracked("Hello", "World");
    BigObject tracked = trackConstruction<BigObject>("Hello","World");
}
```

On the surface there does seem to be a difference between how `untracked` and `tracked` are initialized. The first variable is having its constructor directly invoked, while the second is being constructed from an object being returned by-value from `trackConstruction`. This construction, however, has long been something that has been optimized away to avoid any additional objects and construct the object in question just once. In this case, because the element being returned is initialized by the `return` statement of `trackConstruction`, the optimization is called **return value optimization (RVO)**. C++ has always allowed this optimization by enabling **copy elision**. In C++17, this elision can even be guaranteed and is allowed to be done for objects that have no copy or move constructors. Prior to C++17, this elision can still be guaranteed (on all compilers that the authors are aware of) by **declaring** but not **defining** the copy constructor for `BigObject`. You'll find that this code will still compile and link with such an object, providing observable proof that the copy constructor is never actually invoked with this pattern.

Emplacement

Prior to C++11, inserting an object into a standard library container always required the programmer to first create such an object and then copy it inside the container's storage. As an example, consider inserting a temporary `std::string` object in a `std::vector<std::string>`:

```
void f(std::vector<std::string>& v)
{
    v.push_back(std::string("hello world"));
        // invokes std::string::string(const char*) and the copy-constructor
}
```

In the function above, a temporary `std::string` object is created in the stack frame of `f` and is then copied to the dynamically allocated buffer managed by `v`. Additionally, the buffer might have insufficient capacity and hence might require reallocation, which would in turn require every element of `v` to be somehow copied from the old buffer to the new, larger one.

In C++11, the situation is significantly better thanks to rvalue references. The temporary will be moved into `v`, and any buffer reallocation will *move* the elements between buffers rather than copy them, assuming that the element's move-constructor is a noexcept specifier (see "`noexcept` Specifier" on page 370). The amount of work can, however, be further minimized: What if, instead of first creating an object externally, we constructed the new `std::string` object directly in `v`'s buffer?

This is where **emplacement** comes into play. All standard library containers, including `std::vector`, now provide an **emplacement** API powered by variadic templates (see "Variadic Templates" on page 319) and perfect forwarding (see *Use Cases: Perfect forwarding for generic factory functions* on page 292). Rather than accepting a fully-constructed element, **emplacement** operations accept an arbitrary number of arguments, which will in turn be used to construct a new element directly in the container's storage, thereby avoiding unnecessary copies or even moves:

```
void g(std::vector<std::string>& v)
{
    v.emplace_back("hello world");
        // invokes only the std::string::string(const char*) constructor
}
```

Calling `std::vector<std::string>::emplace_back` with a `const char*` argument results in a new `std::string` object being created in-place in the next suitable spot of the vector's storage. Internally, `std::allocator_traits::construct` is invoked, which typically employs **placement `new`** to construct the object in raw dynamically allocated memory. As previously mentioned, `emplace_back` makes use of both variadic templates and forwarding references; it accepts any number of forwarding references and internally *perfectly forwards* them to the constructor of `T` via `std::forward`:

```
template <typename T>
template <typename... Args>
void std::vector<T>::emplace_back(Args&&... args)
{
    // ...
    new (&freeLocationInBuffer) T(std::forward<Args>(args)...);  // pseudocode
    // ...
}
```

Emplacement operations remove the need for copy or move operations when inserting elements into containers, potentially increasing the performance of a program and sometimes

— depending on the container — even allowing even noncopyable or nonmovable objects to be stored in a container.

As previously mentioned, declaring (but not defining) the *copy* or *move* ctor of a noncopyable or nonmovable type to be private is often a way to guarantee that a C++11/14 compiler constructed an object in place. Containers that might need to move elements around for other operations (such as `std::vector` or `std::deque`) will still need movable elements, while node-based containers that never move the elements themselves after initial construction (such as `std::list` or `std::map`) can use emplace along with noncopyable or nonmovable objects.

Decomposing complex expressions

Many modern C++ libraries have adopted a more "functional" style of programming, chaining the output of one function as the arguments of another function to produce very complex expressions that accomplish a great deal in relatively concise fashion. Consider the way in which the C++20 ranges library encapsulates containers and arbitrary pairs of iterators into objects that can be adapted and manipulated through long chains of functions. Let's say you have a function that reads a file, does some spellchecking for every unique word in the file, and gives you a list of incorrect words and corresponding suggested proper spellings, and you have a range-like library with common utilities similar to standard UNIX processing utilities:

```
SpellingSuggestion checkSpelling(const std::string& word);

std::map<std::string, SpellingSuggestion> checkFileSpelling(
                                                    const std::string& filename)
{
    return makeMap(
        filter(transform(
           uniq(sort(filterRegex(splitRegex(openFile(filename),"\\S+"),"\\w+"))),
        [](const std::string& x)
        {
            return std::tuple<std::string, SpellingSuggestion>(x,
                                                     checkSpelling(x));
        }
   ), [](auto&& x) { return !std::get<1>(x).isCorrect(); }));
}
```

Upon doing code review for this amazing use of a modern library produced by the smart, new programmer on your team, you discover that you actually have a very hard time understanding what is going on. On top of that, the usual tools you have to poke and prod at the code by adding `printf` statements or even breakpoints in your debugger are very hard to apply to the complex set of nested templates involved.

Each of the functions in this range library – `makeMap`, `transform`, `uniq`, `sort`, `filterRegex`, `splitRegex`, and `openFile` – is a set of complex templated overloads and deeply subtle metaprogramming that becomes hard to unravel for a nonexpert C++ programmer. On the other hand, you have also looked at the code generated for this function and the abstractions amazingly get compiled away to a very robust implementation.

To better understand, document, and debug what is happening here, you want to decompose this expression into many, capturing the implicit temporaries returned by all of these functions and ideally not changing the actual semantics of what is being done. To do that properly, you need to capture the type and value category of each subexpression appropriately, without necessarily being able to easily decode it manually from the expression. Here is where `auto&&` forwarding references can be used effectively to decompose and document this expression while achieving the same:

```
std::map<std::string, SpellingSuggestion> checkFileSpelling(
                                                const std::string& filename)
{
    // Create a range over the contents of filename.
    auto&& openedFile = openFile(filename);

    // Split the file by whitespace.
    auto&& potentialWords = splitRegex(
        std::forward<decltype(openedFile)>(openedFile), "\\S+");

    // Filter out only words made from word-characters.
    auto&& words = filterRegex(
        std::forward<decltype(potentialWords)>(potentialWords), "\\w+");

    // Sort all words.
    auto&& sortedWords = sort(std::forward<decltype(words)>(words));

    // Skip adjacent identical words. (This is now a sequence of unique words.)
    auto&& uniqueWords = uniq(std::forward<decltype(sortedWords)>(sortedWords));

    // Get a SpellingSuggestion for every word.
    auto&& suggestions = transform(
        std::forward<decltype(uniqueWords)>(uniqueWords),
        [](const std::string&x) {
            return std::tuple<std::string,SpellingSuggestion>(
                x,checkSpelling(x));
        });

    // Filter out correctly spelled words, keeping only elements where the
    // second element of the tuple, which is a SpellingSuggestion, is not
    // correct.
    auto&& corrections = filter(
        std::forward<decltype(suggestions)>(suggestions),
        [](auto&& suggestion){ return !std::get<1>(suggestion).isCorrect(); });

    // Return a map made from these 2-element tuples:
    return makeMap(std::forward<decltype(corrections)>(corrections));
}
```

Now each step of this complex expression is documented, each temporary has a name, but the net result of the lifetimes of each object is functionally the same. No new conversions have been introduced, and every object that was used as an *rvalue* in the original expression will still be used as an *rvalue* in this much longer and more descriptive implementation of the same functionality.

Potential Pitfalls

Surprising number of template instantiations with string literals

When forwarding references are used as a means to avoid code repetition between exactly two overloads of the same function (one accepting a `const T&` and the other a `T&&`), it can be surprising to see more than two template instantiations for that particular template function, in particular when the function is invoked using string literals.

Consider, as an example, a `Dictionary` class containing two overloads of an `addWord` member function:

```
class Dictionary
{
    // ...

public:
    void addWord(const std::string& word);  // (0) copy word in the dictionary
    void addWord(std::string&& word);       // (1) move word in the dictionary
};

void f()
{
    Dictionary d;

    std::string s = "car";
    d.addWord(s);                    // invokes (0)

    const std::string cs = "toy";
    d.addWord(cs);                   // invokes (0)

    d.addWord("house");              // invokes (1)
    d.addWord("garage");             // invokes (1)
    d.addWord(std::string{"ball"});  // invokes (1)
}
```

Now, imagine replacing the two overloads of `addWord` with a single *perfectly forwarding* template member function, with the intention of avoiding code repetition between the two overloads:

```
class Dictionary
{
    // ...

public:
    template <typename T>
    void addWord(T&& word);
};
```

Perhaps surprisingly, the number of template instantiations skyrockets:

```
void f()
{
    Dictionary d;

    std::string s = "car";
    d.addWord(s);  // instantiates addWord<std::string&>

    const std::string cs = "toy";
    d.addWord(cs);  // instantiates addWord<const std::string&>

    d.addWord("house");              // instantiates addWord<char const(&)[6]>
    d.addWord("garage");             // instantiates addWord<char const(&)[7]>
    d.addWord(std::string{"ball"});  // instantiates addWord<std::string&&>
}
```

Depending on the variety of argument types supplied to `addWord`, having many call sites could result in an undesirably large number of distinct template instantiations, perhaps significantly increasing object code size, compilation time, or both.

`std::forward<T>` can enable move operations

Invoking `std::forward<T>(x)` is equivalent to conditionally invoking `std::move` (if `T` is an *lvalue* reference). Hence, any subsequent use of `x` is subject to the same caveats that would apply to an *lvalue* cast to an unnamed *rvalue* reference; see "*rvalue* References" on page 310:

```
template <typename T>
void f(T&& x)
{
    g(std::forward<T>(x));  // OK
    g(x);                   // Oops! x could have already been moved from.
}
```

Once an object has been passed as an argument using `std::forward`, it should typically not be accessed again without first assigning it a new value because it could now be in a moved-from state.

A perfect-forwarding constructor can hijack the copy constructor

A single-parameter constructor of a class `S` accepting a forwarding reference can unexpectedly be a better match during overload resolution compared to `S`'s copy constructor:

```
struct S
{
    S();                              // default constructor
    template <typename T> S(T&&);  // forwarding constructor
    S(const S&);                      // copy constructor
};
```

```
void f()
{
    S a;
    const S b;

    S x(a);  // invokes forwarding constructor
    S y(b);  // invokes copy constructor
}
```

Despite the programmer's intention to copy from `a` into `x`, the forwarding constructor of `S` was invoked instead, because `a` is a non-`const` *lvalue* expression, and instantiating the forwarding constructor with `T = S&` results in a better match than even the copy constructor.

This potential pitfall can arise in practice, for example, when writing a value-semantic wrapper template (e.g., `Wrapper`) that can be initialized by *perfectly forwarding* the object to be wrapped into it:

```
template <typename T>
class Wrapper  // wrapper for an object of arbitrary type 'T'
{
private:
    T d_datum;

public:
    template <typename U>
    Wrapper(U&& datum) : d_datum(std::forward<U>(datum)) { }
        // perfect-forwarding constructor (to optimize runtime performance)

    // ...
};

void f()
{
    std::string s("hello world");
    Wrapper<std::string> w0(s);  // OK, s is copied into d_datum.

    Wrapper<std::string> w1(std::string("hello world"));
        // OK, the temporary string is moved into d_datum.
}
```

Similarly to the example involving class `S` in the example above, attempting to copy-construct a non-`const` instance of `Wrapper` (e.g., `wr`, above) results in an error:

```
void g(Wrapper<int>& wr)  // The same would happen if wr were passed by value.
{
    Wrapper<int> w2(10);  // OK, invokes perfect-forwarding constructor
    Wrapper<int> w3(wr);  // Error, no conversion from Wrapper<int> to int
}
```

The compilation failure above occurs because the perfect-forwarding constructor template, instantiated with `Wrapper<int>&`, is a better match than the implicitly generated copy constructor, which accepts a `const Wrapper<int>&`. Constraining the perfect forwarding constructor via **SFINAE** (e.g., with `std::enable_if`) to explicitly *not* accept objects whose type is `Wrapper` fixes this problem:

```
template <typename T>
class Wrapper
{
private:
    T d_datum;

public:
    template <typename U,
        typename = typename std::enable_if<
            !std::is_same<typename std::decay<U>::type, Wrapper>::value
        >::type
    >
    Wrapper(U&& datum) : d_datum(std::forward<U>(datum)) { }
        // This constructor participates in overload resolution only if U,
        // after being decayed, is not the same as Wrapper.
};

void h(Wrapper<int>& wr)  // The same would happen if wr were passed by value.
{
    Wrapper<int> w4(10);  // OK, invokes the perfect-forwarding constructor
    Wrapper<int> w5(wr);  // OK, invokes the copy constructor
}
```

Notice that the `std::decay` **metafunction** was used as part of the constraint; for more information on the using `std::decay`, see *Annoyances: Metafunctions are required in constraints* on page 302.

Annoyances

Forwarding references look just like rvalue references

Despite *forwarding* references and rvalue references having significantly different semantics, as discussed in *Description: Identifying forwarding references* on page 287, they share the same syntax. For any given type `T`, whether the `T&&` syntax designates an rvalue reference or a *forwarding* reference depends entirely on the surrounding context.[2]

[2]In C++20, developers might be subject to additional confusion due to the new terse concept notation syntax, which allows function templates to be defined without any explicit appearance of the `template` keyword. As an example, a constrained function parameter, like `Addable auto&& a` in the example below, is a forwarding reference; looking for the presence of the mandatory `auto` keyword is helpful in identifying whether a type is a forwarding reference or *rvalue* reference:

```
template<typename T>
concept Addable = requires(T a, T b) { a + b; };

void f(Addable auto&& a);  // C++20 terse concept notation

void example()
{
    int i;
```

```
template <typename T> struct S0 { void f(T&&); };  // rvalue reference
struct S1 { template <typename T> void f(T&&); };  // forwarding reference
```

Furthermore, even if `T` is subject to template argument deduction, the presence of *any* qualifier will suppress the special *forwarding*-reference deduction rules:

```
template <typename T> void f(T&&);           // forwarding reference
template <typename T> void g(const T&&);     // const rvalue reference
template <typename T> void h(volatile T&&);  // volatile rvalue reference
```

It is truly remarkable that we still do not have some unique syntax (e.g., `&&&`) that we could use, at least optionally, to imply unequivocally a *forwarding* reference that is independent of its context.

Metafunctions are required in constraints

As we showed in *Use Cases* on page 290, being able to perfectly forward arguments of the same general type and effectively leave only the value category of the argument up to type deduction is a frequent need. This is necessary if you do not want to delay construction of the arguments until they are forwarded, possibly because doing so would produce many unnecessary temporaries.

The challenge to make this work correctly is significant. The template must be constrained using **SFINAE** and the appropriate **type traits** to disallow types that aren't some form of cv-qualified or ref-qualified version of the type that you want to accept. As an example, let's consider a function intended to *copy* or *move* a `Person` object into a data structure:

```
class PersonManager {
// ...
template <typename T, typename = typename std::enable_if<
       std::is_same<typename std::decay<T>::type, Person>::value>::type>
void addPerson(T&& person) {}
    // This function participates in overload resolution only if T is
    // (possibly cv- or ref-qualified) Person.
// ...
};
```

This incantation to constrain `T` has a number of layers to it, so let's unpack them one at a time.

- `T` is the template argument we are trying to deduce. We'd like to limit it to being a `Person` that is `const`, `volatile`, `&`, `&&`, or some (possibly empty) valid combination of those.
- `std::decay<T>::type` is then the application of the standard metafunction (defined in `<type_traits>`) `std::decay` to `T`. This metafunction removes all cv-qualifiers and

```
    f(i);  // OK, decltype(a) is int& in f.
    f(0);  // OK, decltype(a) is int&& in f.
}
```

ref-qualifiers from `T`, and so, for the types to which we want to limit `T`, this will *always* be `Person`. Note that `decay` will also allow some other implicitly convertible transformations, such as converting an array type to the corresponding pointer type. For types we are concerned with — those that decay to a `Person` — this metafunction is equivalent to `std::remove_cv<std::remove_reference<T>::type>::type`, or the equivalent and shorter `std::remove_cvref<T>::type>` available in C++20. Due to historical availability and readability, we will continue with our use of `decay` for this purpose.

- `std::is_same<std::decay<T>::type, Person>::value` is then the application of another metafunction, `std::is_same`, to two arguments — our decay expression and `Person`, which results in a `value` that is either `std::true_type` or `std::false_type` — special types that can convert, in compile time, expressions to `true` or `false`. For the types `T` that we care about, this expression will be `true`, and for all other types this expression will be `false`.

- `std::enable_if<X>::type` is yet another metafunction that evaluates to a valid type if and only if `X` is true. Unlike the `value` in `std::is_same`, this expression is simply not valid if `X` is false.

- Finally, by using this `enable_if` expression as a default-initialized template argument, the expression is going to be instantiated for any deduced `T` considered during overload resolution for `addPerson`. This instantiation will fail for any of the types we don't want to allow (something that is not a cv). Because of this, for any `T` that isn't one of the types for which we want to allow `addPerson` to be invoked, this substitution will fail. Rather than being an error, this just removes `addPerson` from the overload set being considered, hence the term **SFINAE**. In this case, that would give us a different error indicating that we attempted to pass a non-`Person` to `addPerson`, which is exactly the result we want.

Putting this all together means we get to call `addPerson` with *lvalues* and *rvalues* of type `Person`, and the value category will be appropriately usable within `addPerson` (generally with use of `std::forward` within that function's definition).

See Also

- "*rvalue* References" (Section 2.1, p. 310) ⧫ Feature that can be confused with forwarding references due to similar syntax.

- "'**`auto`** Variables" (Section 2.1, p. 177) ⧫ Feature that can introduce a forwarding reference with the `auto&&` syntax.

- "Variadic Templates" (Section 2.1, p. 319) ⧫ Feature commonly used in conjunction with forwarding references to provide highly generic interfaces.

Further Reading

- "Item 24: Distinguish universal references from rvalue references," **?**

- **?**
- **?**

Generalized Plain Old Data Types

placeholder

List Initialization: std::initializer_list<T>

placeholder

Unnamed Local Function Objects (Closures)

placeholder text.........

The `noexcept` Operator

placeholder text.........

Range-Based `for` Loops

placeholder

Rvalue References: &&

placeholder text..........

Unions Having Non-Trivial Members

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

Description

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

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

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

A **union** type is permitted to have user-defined special member functions but — by design — does not initialize any of its members automatically. Any member of a **union** having a **non-trivial constructor**, such as **struct** `Nt` below, must be constructed manually (e.g., via **placement new**) 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 "Deleted Functions" on page 46) 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&)
```

```
    */
};
```

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

```
#include <new>  // placement new

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)
    {
        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)
    {
        if (d_useInt) { d_i = int(); }  // value initialized (to 0)
        else          { d_nt = Nt(); }  // BAD IDEA: undefined behavior (no
                                        // lhs object)
    }
```

Now if we were to try to copy-construct or assign one 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);               // Error, no U2(const U2&)
```

```
    a = b;                  // 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:

```
class 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); }  // int DTOR trivial
        }
        else
        {
            if (rhs.d_useInt) { d_nt.~Nt(); new (&d_i) int(rhs.d_i); }
            else              { d_nt = rhs.d_nt; }
        } d_useInt = rhs.d_useInt;

        // Resolve all possible combinations of active types between the
        // left-hand side and right-hand side of the assignment.  Use the
        // corresponding assignment operator when they match; otherwise,
        // if the old member is d_nt, run its non-trivial destructor, and
        // then copy-construct the new member in place:

        return *this;
    }
};
```

Note that in the code example above, we ignore exceptions for exposition simplicity. Note also that attempting to restore a **union**'s implicitly deleted special member functions by using the `= default` syntax (see Section 1.1."Defaulted Functions" on page 34) will still result in their being deleted because the compiler cannot know which member of the union is active.

Use Cases

Implementing a sum type as a discriminated `union`

A **sum type** is an algebraic data type that provides a choice among a fixed set of specific types. A C++11 unrestricted union can serve as a convenient and efficient way to define storage for a sum type (also called a *tagged* or *discriminated* union) because the alignment and size calculations are performed automatically by the compiler.

As an example, consider writing a parsing function `parseInteger` that, given a `std::string input`, will return, as a sum type `ParseResult` (see below), containing 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 *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; // result type (trivial)
        std::string d_error; // error  type (non-trivial)
    };

    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
};
```

If a sum type comprised more than two types, the discriminator would be an appropriately-sized integral or enumerated type instead of a Boolean.

As discussed in *Description* on page 311, having a non-trivial type within a **union** forces the programmer to provide each desired special member function and define it manually; note 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(int 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 still, however, required for destruction and both copy operations[1]:

```
ParseResult::~ParseResult()
{
    if (d_isError)
    {
        d_error.std::string::~string();
```

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

```
            // 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)
{
    if (this == &rhs) // Prevent self-assignment.
    {
        return *this;
    }
    // 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 use a more general sum type[2] abstraction to support arbitrary value types and provide proper exception safety.

Potential Pitfalls

Inadvertent misuse can lead to latent undefined behavior at runtime

When implementing a type that makes use of an unrestricted union, forgetting to initialize a non-trivial object (using either a *member initializer list* or placement **`new`**) or accessing a different object than the one that was actually initialized can result in tacit undefined

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

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.

Annoyances

See Also

- "Deleted Functions" (Section 1.1, p. 46) ♦ Any special member function of a **union** that corresponds to a non-trivial one in any of its member elements will be implicitly deleted.

Further Reading

- **?**
- **?**

User-Defined Literal Operators

placeholder

Variable-Argument-Count Templates

placeholder text.........

Lambdas Having a Templated Call Operator

placeholder text.........

Relaxed Restrictions on `constexpr` Functions

C++14 lifts restrictions regarding use of many language features in the body of a `constexpr` function (see "`constexpr` Functions" on page 179).

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 322.

The C++11 `static_assert` feature (see "`static_assert`" on page 104) 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)
```

Use Cases

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 man-

[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.

ner. 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 328.

[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 81.

[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.

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 319) 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 319), 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 328.

```
    {
        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]

Potential Pitfalls

None so far

Annoyances

None so far

See Also

- "`constexpr` Functions" — Conditionally safe C++11 feature that first introduced compile-time evaluations of functions.
- "`constexpr` Variables" — Conditionally safe C++11 feature 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 325) — 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 subsequent 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.

Further Reading

None so far

Appendix: Optimized C++11 Example Algorithms

Recursive Fibonacci

Even with the restrictions imposed by C++11, we can write a more efficient recursive algorithm to calculate the nth 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;
}
```

`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.

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 307).

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** Variables" on page 177) 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** Variables" on page 177) 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 27) 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 than `char`; see *Potential Pitfalls: Forwarding an existing variable into a closure always results in an object (never a reference)* on page 335.

[3]Here we are using the (C++14) variable template (see "Variable Templates" on page 144) 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` meta-function (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 `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]

Use Cases

Moving (as opposed to copying) objects into a closure

Lambda-capture expressions can be used to *move* (see "*rvalue* References" on page 310) 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.

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.

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 336.

Potential Pitfalls

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 283) 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.

Annoyances

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
};
```

`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]

See Also

- "Lambdas" on page 307 — provides the needed background for understanding the feature in general
- "Braced Init" on page 178 — illustrates one possible way of initializing the captures
- "**`auto`** Variables" on page 177 — offers a model with the same type deduction rules
- "*rvalue* References" on page 310 — gives a full description of an important feature used in conjunction with movable types.
- "Forwarding References" on page 283 — describes a feature that contributes to a source of misunderstanding of this feature

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**.

Chapter 3

Unsafe Features

Intro text should be here.

The [[carries_dependency]] Attribute

placeholder

Preventing Overriding and Derivation

placeholder

Transparently Nested Namespaces

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

Description

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

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

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

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

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

Four subtle details distinguish these approaches:

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

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

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

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

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

1. Name collisions with existing names behave differently due to differing name-lookup rules.

2. **Argument-dependent lookup** (ADL) gives special treatment to **inline** namespaces.

3. Template specializations can refer to the primary template in an **inline** namespace even if written in the enclosing namespace.

4. Reopening namespaces might reopen an **inline** namespace.

One important aspect that all forms of namespaces share, however, is that (1) nested symbolic names (e.g., `n::v1::T`) at the **API** level, (2) **mangled names** (e.g., `_ZN1n2v11dE`, `_ZN1n2v21dE`), and (3) assigned relocatable addresses (e.g., `0x01a64e90`, `0x01a64e94`) at the **ABI** level remain unaffected by the use of either **inline** or **using** or both.[2] Note that a **using** directive immediately following an **inline** namespace is superfluous; name lookup will always consider names in **inline** namespaces before those imported by a **using** directive. Such a directive can, however, be used to import the contents of an **inline** namespace to some other namespace, albeit only in the conventional, **using directive** sense; see *Annoyances — Only one namespace can contain any given* **inline** *namespace* on page 367.

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

Loss of access to duplicate names in enclosing namespace

When both a type and a variable are declared with the same name in the same scope, the variable name hides the type name — such behavior can be demonstrated by using the form of **sizeof** that accepts a nonparenthesized *expression*[3]:

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

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

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

[2] Compiling source files containing, alternately, `namespace n { inline namespace v { int d; } }` and `namespace n { namespace v { int d; } using namespace v; }`, will produce identical assembly. This can be seen with GCC by running `g++ -S <file>.cpp` and viewing the contents of the generated `<file>.s`. Note that Compiler Explorer is another valuable tool for learning about what comes out the other end of a C++ compiler: see https://godbolt.org/.

[3] The form of **sizeof** that accepts a *type* as its argument specifically requires parentheses.

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

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

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

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

A conventional nested namespace behaves as one might expect:

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

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

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

If an **inline** namespace were used instead of a nested namespace followed by a **using** directive, however, the ability to recover by name the hidden entity in the enclosing namespace is lost. Unqualified name lookup considers the inline namespace set and the used namespace set simultaneously. Qualified name lookup first considers the **inline** namespace set, and *then* goes on to look into used namespaces. This means we can still refer to `outer::D` in the example above, but doing so would still be ambiguous if `inner` were an inline namespace. This subtle difference in behavior is a byproduct of the highly specific use case that motivated this feature and for which it was explicitly designed; see *Use Cases — Link-safe ABI versioning* on page 353.

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

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

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

```
namespace outer
{
    struct U { };

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

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

    void g(V) { }
}

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

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

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

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

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

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

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

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

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

using namespace out;                // conventional using directive

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

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

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

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

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

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

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

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

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

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

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

    template <>                      // attempted class template full specialization
    struct Z<double> { };            // Error, outer::Z or outer::inner::Z?
}
```

Reopening namespaces can reopen nested `inline` ones

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

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

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

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

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

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

In the code snippet above, no issue occurs with reopening `outer::inner::n` and no issue would have occurred with reopening `outer::m` but for the `inner` namespaces having been declared **`inline`**. When a new namespace declaration is encountered, a lookup determines if a matching namespace having that name appears anywhere in the **`inline`** *namespace set* of the current namespace. If the namespace is ambiguous, as is the case with `m` in the example above, one can get the surprising error shown.[4] If a matching namespace is found unambiguously inside an **`inline`** namespace, `n` in this case, then it is that nested namespace that is reopened — here, `::outer::inner::n`. The inner namespace is reopened even though the last declaration of `n` is not lexically scoped within `inner`. Notice that the definition of `S` is perhaps surprisingly defining `::outer::inner::n::S`, not `::outer::n::S`. For more on what is *not* supported by this feature, see *Annoyances — Inability to redeclare across namespaces impedes code factoring* on page 364.

Use Cases

Facilitating API migration

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

```
namespace parselib
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

As suggested by the name `v1`, this namespace serves primarily as a mechanism to support library evolution through API and ABI versioning (see *Use Cases — Link-safe ABI versioning* on page 353 and *Use Cases — Build modes and ABI link safety* on page 357). The need to specialize **`class`** `Parser` and, independently, the reliance on ADL to find the free function template `analyze` require the use of **`inline`** namespaces, as opposed to a conventional namespace followed by a **`using`** directive.

Note that, whenever a subsystem starts out directly in a first-level namespace and is subsequently moved to a second-level nested namespace for the purpose of versioning, declaring the inner namespace **`inline`** is the most reliable way to avoid inadvertently destabilizing existing clients; see also *Potential Pitfalls — Enabling selective* **`using`** *directives for short-named entities* on page 360.

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

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

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

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

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

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

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

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

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

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

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

```
namespace parselib
{
    namespace v1  // specializations moved to nested namespace
    {
        template <>
        class Parser<MyClass>
        {
            // ...

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

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

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

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

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

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

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

Although this same functionality can sometimes be realized without the use of **inline** namespaces (i.e., by adding a **using namespace** directive at the end of the `parselib` namespace), the use of ADL and the ability to specialize templates from within the enclosing `parselib` namespace itself would be lost.[5]

Providing separate namespaces for each successive version has an additional advantage in an entirely separate dimension. Though not demonstrated by this specific example,[6] cases do arise where simply changing which of the version namespaces is declared **inline** might lead to an **ill-formed, no-diagnostic required (IFNDR)** program. This might happen when one or more of its translation units that use the library are not recompiled before the program is relinked to the new static or dynamic library containing the updated version of the library software; see *Use Cases — Link-safe ABI versioning* on page 353.

Link-safe ABI versioning

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

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

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

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

[5]Note that, because specialization doesn't kick in until overload resolution is completed, specializing overloaded functions is dubious at best; see *Potential Pitfalls — Relying on* **inline** *namespaces to solve library evolution* on page 363.

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

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

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

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

If we compile this program, a reference to a locally undefined linker symbol, such as `_ZN2my7impl_v15ThingC1Ev`,[7] which represents the `my::Thing::Thing` constructor, will be generated in the `main.o` file:

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

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

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

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

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

#endif
```

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

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

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

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

Observing common good practice, we include the header file of the component as the first substantive line of code to ensure that — irrespective of anything else — the header always compiles in isolation, thereby avoiding insidious include-order dependencies.[8] When we compile the source file `my_thing.cpp`, we produce an object file `my_thing.o` containing the definition of the very same linker symbol, such as `_ZN2my7impl_v15ThingC1Ev`, for the default constructor of `my::Thing` needed by the client:

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

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

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

0
```

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

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

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

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

```
        }
    }
}
```

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

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

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

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

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

            // ...
        };
    }

    inline namespace v2    // inner namespace (default current version)
    {
        class Key
        {
        private:
            std::uint64_t d_securityHash;
            std::uint32_t d_key;
                // sizeof(Key) is 16 bytes

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

            // ...
        };
```

```
    }
}
```

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

Build modes and ABI link safety

In certain scenarios, a class might have two different memory layouts depending on compilation flags. For instance, consider a low-level `ManualBuffer` class template in which an additional data member is added for debugging purposes[9]:

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

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

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

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

    // ...
};
```

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

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

[9]Note that we have employed the C++11 `alignas` attribute (see Section 2.1."`alignas`" on page 154) here because it is exactly what's needed for this usage example.

One way of avoiding these sorts of incompatibilities at link time is to introduce two **inline** namespaces, the entire purpose of which is to change the ABI-level names of the linker symbols associated with `ManualBuffer` depending on the build mode[10]:

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

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

```
namespace buflib  // GOOD IDEA: enclosing namespace for nested inline namespace
{
#ifdef SAFE_MODE  // GOOD IDEA: separate control of non-interoperable versions
    inline namespace safe_build_mode
```

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

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

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

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

```
#else
    inline namespace normal_build_mode
#endif
    {
        template <typename T>
        struct ManualBuffer
        {
        private:
            alignas(T) char d_data[sizeof(T)];  // aligned/sized to hold a T

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

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

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

Finally, if we have two implementations of a particular entity that are sufficiently distinct, we might choose to represent them in their entirety, controlled by their own bespoke conditional-compilation switches, as illustrated here using the `my::VersionedThing` type (see *Use Cases — Link-safe ABI versioning* on page 353):

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

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

#ifdef MY_THING_VERSION_2  // bespoke switch for this component version
    inline
#endif
```

```
    namespace v2
    {
        struct VersionedThing
        {
            double d_i;
            VersionedThing();
        };
    }
}
#endif
```

However, see *Potential Pitfalls* — `nline`-*namespace-based versioning doesn't scale* on page 361.

Enabling selective `using` directives for short-named entities

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

Even without either of these two needs, another property of an **`inline`** namespace differentiates it from a non-**`inline`** one followed by a **`using`** directive. Recall from*Description — Loss of access to duplicate names in enclosing namespace* on page 343 that a name in an outer namespace will hide a duplicate name imported via a **`using`** directive, whereas any access to that duplicate name within the enclosing namespace would be ambiguous when that symbol is installed by way of an **`inline`** namespace. To see why this more forceful clobbering behavior might be preferred over hiding, suppose we have a communal namespace `abc` that is shared across multiple disparate headers. The first header, `abc_header1.h`, represents a collection of logically related small functions declared directly in `abc`:

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

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

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

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

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

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

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

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

Potential Pitfalls

`inline`-namespace-based versioning doesn't scale

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

Controlling which namespace is **inline** using macros, such as was done in the my::VersionedThing example in *Use Cases — Link-safe ABI versioning* on page 353, will result in code that directly uses the unversioned name, my::VersionedThing being bound directly to the versioned name my::v1::VersionedThing or my::v2::VersionedThing, along with the class layout of that particular entity. Sometimes details of the use of the **inline** namespace member are not resolved by the linker, such as the object layout when we use types from that namespace as member variables in other objects:

```
// my_thingaggregate.h:

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

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

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

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

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

This need for layers above a library to be aware of and to integrate into their own structure the same namespaces the library has removes all or most of the benefits of using **inline** namespaces for versioning. For an authentic real-world case study of heroic industrial use — and eventual disuse — of **inline**-namespaces for versioning, see *Appendix: Case study* *of using* **inline** *namespaces for versioning* on page 368.

Relying on `inline` namespaces to solve library evolution

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

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

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

Several other good practices facilitate smooth evolution for the Standard Library[11]:

- Avoid specializing variable templates, even if dependent on user-defined types, except for those variable templates where specialization is explicitly allowed.[12]
- Other than a few very specific exceptions, avoiding the forming of pointers to Standard Library functions — either explicitly or implicitly — allows the library to add overloads, either as part of the Standard or as an implementation detail for a particular Standard Library, without breaking user code.[13]

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

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

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

- Overloads of Standard Library functions that depend on user-defined types are permitted, but, as with specializing Standard Library templates, users must still meet the requirements of the Standard Library function. Some functions, such as `std::swap`, are designed to be customization points via overloading, but leaving functions not specifically designed for this purpose to vendor implementations only helps to avoid surprises.

Finally, upon reading about this **`inline`** namespace feature, one might think that all names in namespace `std` could be made available at a global scope simply by inserting an **`inline namespace`** `std {}` before including any standard headers. This practice is, however, explicitly called out as ill-formed within the C++11 Standard.[14]

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

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

Annoyances

Inability to redeclare across namespaces impedes code factoring

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

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

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

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

[14] Although not uniformly diagnosed as an error by all compilers, attempting this forbidden practice is apt to lead to surprising problems even if not diagnosed as an error immediately.

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

Note that, in the case of `g0` in this example, the "specialization" `void g0(int)` is a non-template *overload* of the function template `g0` rather than a specialization of it. We *cannot*, however, portably[15] declare these templates within the `outer` namespace and then specialize them within the `inner` one, even though the `inner` namespace is **`inline`**:

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

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

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

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

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

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

    template<> struct S2<int> { };     // Error, S2 is ambiguous in outer.
```

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

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

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

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

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

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

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

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

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

Also note that, as ever, the corresponding definition of the declared template must have been seen before it can be used in a context requiring a complete type. The importance of ensuring that all specializations of a template have been seen before it is used substantively (i.e., **ODR-used**) cannot be overstated, giving rise to the only limerick, which is actually part of the normative text, in the C++ Language Standard[16]:

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

Only one namespace can contain any given `inline` namespace

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

As an example, the C++14 Standard Library provides a hierarchy of nested **`inline`** namespaces for literals of different sorts within namespace `std`: `std::literals::complex_literals`, `std::literals::chrono_literals`, `std::literals::string_literals`, and `std::literals::string_view_literals`. These namespaces can be imported to a local scope in one shot via a **`using`** `std::literals` or instead, more selectively, by **`using`** the nested namespaces directly. This separation of the types used with user-defined literals, which are all in namespace `std`, from the user-defined literals that can be used to create those types led to some frustration; those who had a **`using namespace`** `std;` could reasonably have expected to get the user-defined literals associated with their `std` types. However, the types in the nested namespace `std::chrono` did *not* meet this expectation.[17]

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

See Also

- “`alignas`” (§2.1, p. 154) ♦ Safe C++11 feature used in the example in *Use Cases: Build modes and ABI link safety* on page 357 to provide properly aligned storage for an object of arbitrary type `T`.

Further Reading

TODO, TBD

---

[16]See **?**, section 14.7.3.7, pp. 375–375, specifically p. 376.
[17]**?**

Appendix: Case study of using `inline` namespaces for versioning

By Niall Douglas

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

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

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

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

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

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

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

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

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

```
namespace output = OUTCOME_V2_NAMESPACE;
```

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

Why?

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

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

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

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

The `noexcept` Function Specification

placeholder

Reference Qualified Member Functions

placeholder

Deducing Types Using `decltype` Semantics

placeholder

Function (`auto`) `return`-Type Deduction

placeholder text........

Chapter 4

Parting Thoughts

Testing Section

Testing Another Section

Glossary

ABI TODO 343, 350

ADL TODO 345, 346, 350, 360

API TODO 343, 350

Access Specifier TODO 36, 183, 195

Acquire/Release Memory Barrier TODO 73

Aggregate TODO: from standard: An aggregate is an array or a class (Clause 9) with no user-provided constructors (12.1), 126–128

Aggregate Initialization TODO 126, 127

Alias Template TODO 123

Alignment The alignment of an *object type* is a `std::size_t` value (always a power of two) representing the number of bytes between successive addresses at which objects of this type can be allocated. It is the reason why *padding* might be introduced between non-static members of a **`class`**.

Architectural Insulation TODO 233

Argument Dependent Lookup TODO 343

As If TODO

Atomic TODO 73

Barriers TODO 74

Basic Source Character Set TODO (SMD: the basic source character set is the abstract character set that must be available for expressing C++ source code. It's not the same as the source file encoding, which is what it is in C

from unicode footnote: 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 (e.g. emojis) in a C++ source file.

@SuperV1234 so are you saying that there are holes that can act as escape sequences? So like 0xfe means the next character is in play and 0xff means the next 3 characters are in play? Just don't get it. @SMD shift encodings? I'm not sure if we really want to go there?

@SMD this is about the ways your editor and compiler can conspire to lie to you. A octets in a u8 literal are not interpreted as unicode. They are interpreted however the compiler interprets source in phase 1 of translation. The fact that the common encoding on Windows is 8 bit complete with no shift characters means that even though your editor will display , the compiler will see 0xF0 0x9F 0x8D 0xB8, which the compiler will emit as that sequence of bytes, even though the 0xBD is not canonically mapped.) 99, 118

Benchmark Test TODO 102

Boilerplate Code TODO 124, 147

Brace Elision TODO 128

Bytes TODO 140

C Style Functions TODO 145

CI TODO 368

Callable Object TODO 63

Callback Functions TODO 221

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

Closure TODO 80

Code Bloat TODO 260, 261

Compile Time Coupling TODO 219

Compile Time Dispatch TODO 110

Complete Type TODO 218

Component TODO 227, 242, 243, 248, 266, 267, 354

Component Local TODO 220

Components TODO 280

Concepts TODO 111

Concrete Class TODO 186

Conditionally Supported A conforming implementation can choose not to support a feature that is specified as conditionally supported; if used and not supported, however, at least one diagnostic — stating such lack of support — is required. 10

Constant Expression TODO: An expression that can be evaluated at compile-time. Mention `constexpr` and state that **`const`** variables that are initialized from a compile-time constants are themselves required to be compile-time constants. New info to me in June 2020, worthwhile to have. 51, 104

Constant Initialization TODO 69

A property of certain particular language construct (e.g., an **`if`** expression) that permits conversion from any expression `E` to be treated as if a **`static_cast`** to type bool had been applied — e.g., **`if`** `(E)` is definitional equivalent to **`if`** `(`**`static_cast`**`<`**`bool`**`>(E))`. 56

Contextual Conversion to Contextual Convertibility To Bool bool TODO 54

Contextual Keyword A *"contextual keyword"* is a special identifier that acts like a *keyword* when used in particular contexts. `override` is an example as it can be used as a regular identifier outside of member-function declarators. 94

Continuous Refactoring TODO 135, 137

Conventional String Literals TODO 102

Conversion Operator TODO 55

Conversion Operators TODO 53, 54

Converting Constructors TODO 53, 54

Cookie TODO 221

Copy Assignment Operator TODO 46

Copy Constructor TODO 46

Copy Operations TODO 46

Copy Semantics An operation on two objects, a destination and a source, has copy semantics if, after it completes, the value of the source object remains unchanged and the target object has the same value as does the source. .. for some criteria the source and destination are the same, for a value semantic type that property is value... 46

Critical Section TODO 61, 64

Curiously Recurring Template Pattern TODO 248, 249

Cyclic Physical Dependency TODO 280

Cyclically Dependent TODO 69

Data Races TODO 62

Declaration TODO 110, 111

Declare TODO 50, 51

Declared Type The type of the entity named by the given expression. 27

Default Initialized TODO 37

Define TODO 50

Definition TODO 59, 60

Delegating Constructor TODO 21

Delete TODO 51

Deleted Function TODO 34

Dependent Type TODO 184

Design Pattern TODO 221

Direct Initialization TODO 54

Disambiguator TODO 30, 31

Duck Typing TODO 258

Dumb Data TODO 229

Embedded Development TODO 132

Embedded Systems TODO 85

Encapsulation TODO 218

Entity One of the primary building blocks of a C++ program. An entity is one of: *value*, *object*, *reference*, *function*, *enumerator*, *type*, *class member*, *bit-field*, *template*, *template specialization*, *namespace*, *parameter pack*, or **`this`**. 10, 27, 77, 135–137

Escalating TODO 280

Excess n TODO 142

Explicit Instantiation Declaration TODO 261–263, 265, 266, 269, 272, 276–278, 280, 281

Explicit Instantiation Definition TODO 261–263, 265, 266, 271, 276–278, 280, 281

Explicit Instantiation Directive TODO 261, 262, 276, 279, 281

Explicit Specifier TODO 54

Expression SFINAE TODO 31, 111, 114

Fences TODO 74

Flow Of Control TODO 59

Footprint TODO 264

Forward Class Declaration TODO 230

Forward Declaration TODO - mention about enums that this was Impossible prior to C++11 as the compiler could not determine the underlying type without visibility of the enumerators. 217, 218

Forward Declare TODO 220, 227

Forward Declared TODO 219

Free Function TODO 50

Fully Constructed TODO 22

Function Scope TODO 59, 61–63, 65, 73, 74

Fundamental Integral Type TODO 81

Fundamental Types TODO 311

General Purpose Machines General-purpose machines are what Big Tech, the financial industry, and many other large companies use exclusively. 85

Golden File TODO 102

Header Only Library TODO 353

Hide or Hiding Function-name **hiding** occurs when a member function in a derived class has the same name as one in the base class, but it is not overriding it due to a difference in the function signature or because the member function in the base class is not `virtual`. The hidden member function will **not** participate in dynamic dispatch; the member function of the base class will be invoked instead when invoked via a pointer or reference to the base class . The same code would have invoked the derived class's implementation had the member function of the base class had been **overridden** rather than **hidden**. 182, 185

Higher Order Function TODO 113, 115

Hyrum'S Law TODO 243

IFNDR see Ill-formed, No Diagnostic Required 108, 228, 230, 355, 357, 364

Id Expression TODO 27

Ill Formed TODO 109, 353, 357, 363

Ill-Formed, No Diagnostic Required TODO *No diagnostic required*. The compiler is not mandated by the Standard to produce a diagnostic and the behavior of the code is *undefined*. 25, 88, 106, 227, 353

Implementation Defined TODO 10, 91, 217

Implementation Inheritance TODO 186

In Process TODO 242

Inheriting Constructors TODO 184, 195

Instantiation Time TODO 109

Insulate TODO 87, 221, 227, 276

Insulating TODO 218

Insulation TODO 218

Integer Literals TODO 130

Integral Promotion TODO 49

Integral Type TODO 81

Interface Inheritance TODO 186

Join TODO 63

Lambda Expressions TODO 76

Leaks TODO 62

Linkage TODO 76

Literal Type TODO 51, 52

Local Declaration TODO 217, 227

Local Scope TODO 59

Logical TODO 272

Long Distance Friendship TODO 242, 243, 248

Mangled Names TODO 343, 361

Mantissa TODO 142

Mechanism TODO 219

Mechanisms TODO 25

Member TODO 311

Member Initializer List TODO 21, 23, 25

Memory Leak TODO 67

Message TODO 229

Messenger TODO 229

Meyers Singleton TODO 65, 66

Mixed Mode Builds TODO 358

Modules TODO 248

Most Vexing Parse TODO 182

Move Operations TODO 44, 46

Multithreading Context TODO 62

Name Mangling TODO 353

Nibbles TODO 140

Non Trivial Constructor TODO 311

Non Trivial Special Member Function TODO 311

Nonprimitive Functionality TODO 58

Null Address TODO 90, 93

ODR TODO 364

Object Invariants TODO 185

Odr Used TODO 366

One Definition Rule TODO 280, 357

Opaque Declaration TODO 217

Opaque Enumeration TODO 219, 221, 224, 232

Overload Resolution TODO 46

Overriding TODO 185

POD TODO 311

Parameter Pack TODO 145

Partial Implementation TODO 186

Partially Constructed TODO 21, 22

Physical TODO 272

Physical Design TODO 218, 248

Placement New TODO 311, 312, 316, 357

Pointer To Member TODO 56

Precondition TODO 15

Predicate Function TODO 79

Protocol TODO 186

Pure Function A function that produces no side effects and always returns the same value given the same sequence of argument values. 13

Qualified Name TODO 346

RAII See Resource Acquisiition is Initialization 46, 69

Range TODO 30

Raw String Literals TODO 102

Receiver TODO 229

Redundant Check TODO 104

Reference Type TODO 311

Return Type Deduction TODO 30

SFINAE See Substitution Failure Is Not An Error 30, 110, 111, 184

SHA TODO 368

Safe bool idiom A technique in class design that suppresses unwanted comparisons made available by the presence of a non-`explicit` conversion function to `bool`. 56

Section TODO 268

Sections TODO 269, 271, 278, 279

Sender TODO 229

Side Effects TODO 13

Signature TODO 112, 182, 257

Signed Integer Overflow TODO 82

Slicing TODO 185

Special Functions TODO 46

Special Member Function As per the C++17 standard, the following are considered special member functions: *destructors*, *default constructors*, *copy constructors*, *move constructors*, *copy assignment operators*, and *move assignment operators*. 34, 36, 43, 44, 46, 50, 51, 311, 312

Standard Conversion TODO 49

Static Assertion Declarations TODO 104

Static Data Space TODO 151

Static Duration TODO 63, 73

String Literal TODO 104, 135

Strong Typedef Idiom TODO 67

TODO 187–189, 191–193

Strong **`typedef`** **Structural Inheritance** TODO 49, 190

Sum Type Abstract data type allowing the representation of one of multiple possible alternative types. Each alternative has its own type (and state), and only one alternative can be “active” at any given point in time. Sum types automatically keep track of which choice is “active,” and properly implement value-sematic special member functions (even for non-trivial types). They can be implemented efficiently as a C++ `class` using a C++ `union` and a separate (integral) discriminator. This sort of implementation is commonly referred to as a discriminating (or “tagged”) union. 314–316

Template Head TODO 144

Template Instantiation Time TODO 105

Template Template Class Parameter TODO 151

Template Template Parameter TODO 151

Test Driver TODO 102

Translation Unit TODO 217

Trivial TODO 39, 40, 51

Trivial Operation TODO 34

Trivial Types TODO 311

Trivially Constructible TODO 73

Trivially Copy Constructor The requirements for the copy constructor of a class `T` to be considered trivial are as follows:

Trivially Copyable TODO 40, 42

Type Trait TODO 147

Type Traits TODO 108, 124

UDT TODO 21

Undefined TODO 88

Undefined Behavior TODO 15, 16, 25, 62, 71, 72, 82, 88, 255, 312, 316, 317, 363

Undefined Symbols TODO 270

Underlying Type TODO 119, 187, 217, 218, 228

Unevaluated Contexts TODO 32

User Defined Type TODO 21, 239

User Provided TODO 73

User Provided Special Member Function In the C++11 standard, a special member function is considered user-provided if it has been explicitly declared by the programmer and not explicitly defaulted or deleted in its first declaration:

One of the requirement for a special member function to be considered trivial is that it is not user-provided. Trivial classes (i.e. with all special member function being trivial) have special semantics (e.g. they can be safely used with `std::memcpy`, or copied into a suitable array of **`char`** and back). See 36

User Provided Special Member Functions TODO 34

Using Declaration TODO 181

Using Directive TODO 343

Value TODO 25

Value Category TODO 27, 32

Value Constructor TODO 37

Value Semantic TODO 23, 25, 37

Value Semantic Type TODO 242

Vocabulary Type TODO 82, 83, 118

Zero Initialized TODO 69

inline namespace TODO 342