Embracing Modern C++ Safely

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

Embracing Modern C++ Safely

John Lakos

Vittorio Romeo

Addison-Wesley

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

Library of Congress Cataloging-in-Publication Data

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

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

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.

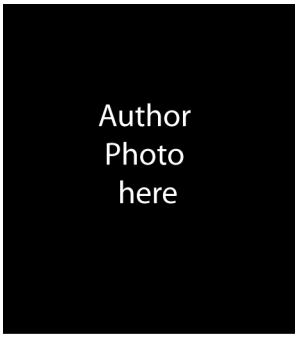
Preface

The text of the preface will go here.

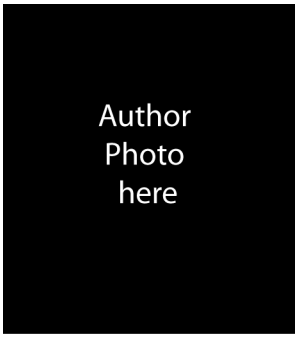
Acknowledgements

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

About the Authors

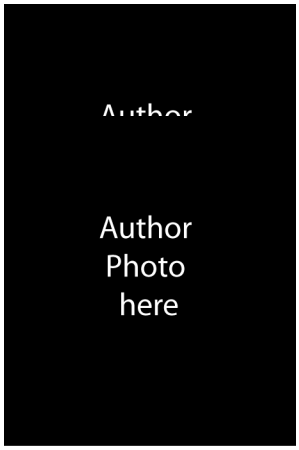


John Lakos, author of *Large-Scale C++ Software Design* (Addison-Wesley, 1996) and *Large-Scale C++ Volume I: Process and Architecture* (Addison-Wesley, 2019), serves at Bloomberg in New York City as a senior architect and mentor for C++ software development worldwide. He is also an active voting member 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

ch-intro

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

ch-safe
sec-safe-cpp11 Intro text should be here.

Generalized Attribute Support

attributes

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

Description

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 14). Customized annotations targeted at external tools might be beneficial as well.

C++ attribute syntax

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

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.

er-dependent-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

attribute-table1

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

attribute-use-cases

Use Cases

Prompting useful compiler diagnostics

-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.
};
```

[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.

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]
```

imization-opportunities

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

[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.

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.

-better-optimizations

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 17.

xternal-static-analysis

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 particular, 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:

[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 **?**).

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

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

s-semantic-properties

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>

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

te-potential-pitfalls

Potential Pitfalls

tion-defined-behavior

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]

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

ect-program-correctness

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:

`std::random_devicestd::mt19937std::uniform_real_distribution`

```
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

Annoyances

see-also

See Also

- "`noreturn`" (§1.1, p. 83) ♦ presents a standard attribute used to indicate that a particular function never returns control flow to its caller.
- "`deprecated`" (§1.2, p. 136) ♦ presents a standard attribute that discourages the use of an entity via compiler diagnostics.
- "`carries_dependency`" (§3.1, p. 616) ♦ presents a standard attribute used to communicate release-consume dependency-chain information to the compiler to avoid unnecessary memory-fence instructions.

tribute-further-reading

Further Reading

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

Consecutive Right-Angle Brackets

-right-angle-brackets

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

Description

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:

```
    std::vector

// 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:

```
    std::vector

// 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

-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

Use Cases

Avoiding annoying whitespace when composing template types

omposing-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:

```
    std::liststd::vectorstd::stringstd::map

// 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

Potential Pitfalls

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

y-stop-working-in-c++11

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

```
T<1 >> 5> 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

ilently-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:

```
    std::cout

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

template <typename> struct S
{
```

[1] Example adapted from **?**

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

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

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

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

```
// C++03

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

```
// C++11

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

Though theoretically possible, programs that (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

Annoyances

see-also

See Also

further-reading

Further Reading

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

Operator for Extracting Expression Types

decltype

The keyword **decltype** enables the compile-time inspection of the **declared type** of an **entity** *or* the type and **value category** of an expression. Note that the special construct **decltype(auto)** has a separate meaning; see Section 3.2."**decltype(auto)**" on page 686.

Description

description

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

Use with entities

cally-named)-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:

std::string

```
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

(unnamed)-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 479):

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:

```
    assert
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

use-cases-decltype

Avoiding unnecessary use of explicit typenames

e-of-explicit-typenames

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

```
    std::liststd::vector
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::liststd::vector
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.

onsistency-explicitly

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:

```
    std::uint8_t

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`:

```
    std::uint8_t

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]

[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.

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)
// ...
```

Creating an auxiliary variable of generic type

ariable-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 183. Note that this particular use involves significant repetition of the expression a+b. See *Annoyances — Mechanical repetition of expressions might be required* on page 27 for a discussion of ways in which this might be avoided.

Determining the validity of a generic expression

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`**:

`std::sort`

```
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]

[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.

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.

potential-pitfalls

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-decltype

Annoyances

decltype-mechanical

Mechanical repetition of expressions might be required

As mentioned in *Use Cases — Creating an auxiliary variable of generic type* on page 25, 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:

`std::declval`

```
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

See Also

- "**using** Aliases" (§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" (§2.1, p. 183) ♦ The type computed by **decltype** is similar to, but distinct from, the type deduction used by **auto**.
- "*rvalue* References" (§2.1, p. 479) ♦ The **decltype** operator yields precise information on whether an expression is an *lvalue* or *rvalue*.

- "**decltype(auto)**" (§3.2, p. 686) ♦ **decltype** type computation rules can be useful in conjunction with an **auto** variable.

Further Reading

further-reading

Using = default for Special Member Functions

cial-Member-Functions

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

Description

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 479) 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

r-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 41 of *Appendix — Implicit Generation of Special Member Functions*) to demystify what's going on under the hood.

e-default-constructor

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 33. 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."**??**" on page **??**.

st-a-copy-constructor

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.

ing-just-the-destructor

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.

than-one-special-member

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

per-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."**??**" on page **??**), 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

ecial-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.

default-exampleh-code

```
// 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 37.

default-use-cases

Use Cases

n-suppressed-by-another

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:

```
    std::stringassert
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`:

default-voidf-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` 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

it-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 41, 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

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 noninline implementation in a corresponding[2] `.cpp` file:

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

LargeScale::LargeScale() = default;
    // Generate the default implementation for this default destructor.
```

Using this *insulation* technique, we are free to change our minds and implement the default constructor ourselves in any way we see fit without necessarily forcing our clients to recompile.

potential-pitfalls

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

[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.

`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

tions-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
    std::unique_ptr<Impl> d_impl;  // noncopyable data member

public:
    Connection() = default;
    Connection(const Connection&) = default;
};
```

[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 479) to represent ownership transfer between instances:

```
std::unique_ptr

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.
```

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 103) in conjunction with an appropriate trait from the `<type_traits>` header:

`std::vector`

```
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

`see-also`

- "**??**" (§1.1, p. **??**) ♦ describes a companion feature, **`= delete`**, that can be used to suppress access to implicitly generated special member functions.
- "**`static_assert`**" (§1.1, p. 103) ♦ describes a facility that can be used to verify at compile time that undesirable copy and move operations are declared to be accessible.
- "*rvalue* References" (§2.1, p. 479) ♦ provides the bases for **move operations**, namely, the move-constructor and move-assignment special member functions, which too can be defaulted.

Further Reading

`further-reading`

- Howard Hinnant, "Everything You Ever Wanted to Know About Move Semantics (and

[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.

Then Some)," **?**

- Howard Hinnant, "Everything You Ever Wanted to Know About Move Semantics," **?**

Appendix

appendix-default

Implicit Generation of Special Member Functions

pecial-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-table1

	Default Ctor	Destructor	Copy Ctor	Copy Assignment	Move Ctor	Move Assignment
Nothing	Defaulted	Defaulted	Defaulted	Defaulted	Defaulted	Defaulted
Any Ctor	Not Declared	Defaulted	Defaulted	Defaulted	Defaulted	Defaulted
Default Ctor	User Declared	Defaulted	Defaulted	Defaulted	Defaulted	Defaulted
Destructor	Defaulted	User Declared	Defaulted[a]	Defaulted[a]	Not Declared	Not Declared
Copy Ctor	Not Declared	Defaulted	User Declared	Defaulted[a]	Not Declared	Not Declared
Copy Assignment	Defaulted	Defaulted	Defaulted[a]	User Declared	Not Declared	Not Declared
Move Ctor	Not Declared	Defaulted	Deleted	Deleted	User Declared	Not Declared
Move Assignment	Defaulted	Defaulted	Deleted	Deleted	Not Declared	User Declared

[a] Deprecated behavior: compilers might warn upon reliance of this implicitly generated member function.

As an example, explicitly declaring a copy-assignment operator would result in the default constructor, destructor, and copy constructor being defaulted and in the move operations not being declared. If more than one special member function is user declared (regardless of whether or how it is implemented), the remaining generated member functions are those in the intersection of the corresponding rows. For example, explicitly declaring both the destructor and the default constructor would still result in the copy constructor and the copy-assignment operator being defaulted and both move operations not being declared. Relying on the compiler-generated copy operations when the destructor is anything

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

but defaulted is dubious; if correct, defaulting them explicitly makes both their existence and intended definition clear.

delegating-constructors

Constructors Calling Other Constructors

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

description

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 47). 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
```

tordelegating-use-cases

tion-among-constructors

Use Cases

Avoiding code duplication among constructors

Many consider avoiding gratuitous code duplication 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. 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]:

```
    std::uint16_tstd::uint32_tstd::string
#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};
        }
    }
```

[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 48.

```
    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

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

ting-potential-pitfalls

Delegation cycles

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
};
```

[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
```

suboptimal-factoring

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 45. 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


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

Annoyances

see-also

See Also

- "Forwarding References" (§2.1, p. 351) ♦ provides perfect forwarding of arguments from one ctor to another.

[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.

- "Variadic Templates" (§2.1, p. 519) ♦ describes how to implement constructors that forward an arbitrary list of arguments to other constructors.

Further Reading

further-reading

-conversion-operators

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.

cription-explicitconv

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**):

```
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.
};
```

Using 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 55.

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
}
```

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 52):

```
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
}
```

se-cases-explicitconv

s-a-test-for-validity

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**,[1] 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 50, 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**.[2]

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

[2] 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.

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
}
```

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.

l-pitfalls-explicitconv

Potential Pitfalls

conversion-is-indicated

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
}
```

amed-function-is-better

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., nonoperator) function is often better. Recall from *Description* on page 50 that we used a conversion operator of class `Point` to represent the distance from the origin. This example serves to illustrate both how conversion operators *can* be used and 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 differ-

ent units (e.g., `angleInDegrees`, `angleInRadians`). Another valid design decision would be to return an object of user-defined type, say, `Angle`, that captures the amplitude and provides named accessory to the different units (e.g., `asDegrees`, `asRadians`).

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.[3]

Annoyances

annoyances

See Also

see-also

Further Reading

further-reading

[3]For more on separating out nonprimitive functionality, see **?**, sections 3.2.7–3.2.8, pp. 529–552.

nction-static-variables

Thread-Safe Function-Scope `static` Variables

Initialization of function-scope **`static`** objects is now guaranteed to be free of data races in the presence of multiple concurrent threads.

cription-functionstatic

Description

A variable declared at function (a.k.a. local) scope has **automatic storage duration**, except when it is marked **`static`**, in which case it has **static storage duration**. Variables having automatic storage duration are allocated on the stack each time the function is invoked, and initialized when that invocation's **flow of control** passes through the **definition** of that object. In contrast, variables with static storage duration (e.g., `iLocal`) defined at function scope (e.g., `f`) are instead allocated once per program and are initialized only 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 to initialize the `a`, `b`, and `c` variables, all three of them are initialized to the same value, 10, supplied to the first invocation of `f`. Although the function-scope **`static`** object, `iLocal`, was created after `main` was entered, it will not be destroyed until after `main` exits.

ncurrent-initialization

Concurrent initialization

Historically, initialization of **function-scope** static storage 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. These data races around initialization can lead to the initializer being invoked multiple times, object construction running concurrently on the same object, and control flow continuing past the variable definition before initialization had completed at all. All of these variations would result in critical soft-

ware flaws. One common but nonportable pre-C++11 workaround was the *double-checked lock pattern*; see *Appendix — C++03 double-checked-lock pattern* on page 70.

As of C++11, a conforming compiler is now required to ensure that initialization of function-scope static storage duration objects is performed safely, and exactly once, before execution continues past the initializer, even when the function is called concurrently from multiple threads.

`destruction`

Destruction

Automatic objects within a local scope are destroyed when control leaves the scope in which they are declared. In contrast, **`static`** local objects that have been initialized are not destroyed until 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 the order in which they completed construction. 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. This behavior is as if each static object is scheduled for destruction by using the C Standard Library function `std::atexit` right after construction.

`logger-example`

Logger example

Let's now consider a 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 60)[1]:

```
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 Logger's constructor for the first (and only) time

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

Here we have an example of the "Singleton pattern"[2] being used to create the shared `Logger` instance and provide access to it through the `getLogger()` function. The **`static`** local instance of `Logger`, `localLogger`, will be initialized exactly once and then destroyed after normal program termination. In C++03, it would not be safe to call this function concurrently from multiple threads. Conversely, C++11 *guarantees* that the initialization

[1] 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/bal/ball`).

[2] **?**, Chapter 3, section "Singleton," pp. 127–134

of `localLogger` will happen exactly once even when multiple threads call `getLogger` concurrently.

multithreaded-contexts

Multithreaded contexts

The C++11 Standard Library provides several utilities and abstractions related to multithreading. The `std::thread` class 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**, a new thread invoking that callable object will be spawned. Prior to destroying such `std::thread` objects it is necessary to invoke the `join` member function on the thread object, which will block until the background thread of execution completes invoking its callable object.

This threading facility from the standard library can be used with our earlier `Logger` example from *Logger example* on page 58, to concurrently attempt to access the `getLogger` function:

```
#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;
}
```

Such use prior to the C++11 thread-safety guarantees (with pre-C++11 threading libraries) could have led to a data race during the initialization of `localLogger`, which was defined as a local **`static`** object in `getLogger`. This **undefined behavior** might have resulted in invoking the constructor of `localLogger` multiple times, returning from `localLogger` before that constructor had actually been completed, or any other form of misbehavior over

which the developer has no control.
As of C++11, the example above has no data races provided that `Logger::operator<<(const char*)` is designed properly for multithreaded use, even though the `Logger::Logger(const char* logFilePath)` constructor (i.e., the one used to configure the singleton instance of the logger) is 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 *Potential Pitfalls — Dangerous recursive initialization* on page 66. Such use of function-scope `static`s, however, is not foolproof; see *Potential Pitfalls — Depending on order-of-destruction of local objects after* `main` *returns* on page 67.

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 67.

-cases-functionstatic

Use Cases

meyers-singleton

Meyers Singleton

The guarantees surrounding access across **translation units** to runtime initialized objects at file or namespace scope are few and weak — 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:

```
// 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`. Code outside the `libcomp.cpp` file can access the singleton object `globalS` only by calling the free function `getGlobalS()`. 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 <cassert>    // standard C assert macro
#include <libcomp.h>  // getGlobalS()

bool globalInitFlag = getGlobalS().isInitialized();

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

Depending on the compiler or the link line, the call initializing `globalInitFlag` may occur and return *prior* to the initialization of `globalS`. C++ does not guarantee that 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 nonlocal 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 from file or namespace scope to 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 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 will also 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 the program, 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. The Meyers Singleton can be safely used both in the programs where the singleton initialization might happen before `main` and those where it might happen after additional threads have already been started.

As discussed in *Description* on page 57, the augmentation of a thread-safety guarantee for the runtime initialization of function-scope **static** objects in C++11 minimizes the effort required to create a thread-safe singleton. Note that, prior to C++11, the simple function-scope **static** implementation would not be safe if concurrent threads were trying to initialize the logger; see *Appendix — C++03 double-checked-lock pattern* on page 70.

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);  // configures the singleton
    ~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 design doesn't admit the dual-singleton use of 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 375), corresponding to each singleton use:

```
class PrimaryCamera
{
private:
    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 provides stronger guarantees than 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.[3] 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 is that 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:

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

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

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

Note that any memory that the `Logger` itself manages would still come from the global heap and be recognized as memory leaks.[4]

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 until application exit.

Potential Pitfalls

tfalls-functionstatic

`static` storage duration objects are not guaranteed to be initialized

eed-to-be-initialized

Despite C++11's guarantee that each individual function-scope **`static`** initialization will occur at most once and before control can reach a point where the variable can be referenced, no analogous guarantees are made of nonlocal objects of static storage duration. This makes any interdependency in the initialization of such objects, especially across translation units (TUs), an abundant source of insidious errors.

Objects that undergo **constant initialization** have no such issue: such objects will never be accessible at run time before having their initial values. Objects that are not constant initialized[5] will instead be **zero initialized** until their constructors run, which itself might lead to undefined behavior that is not necessarily conspicuous.

As a demonstration of what can happen when we depend on the relative order of initialization of 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
int *a = new int;      // runtime initialization of file-scope variable
int setA(int i)        // Initialize a; then b.
{
    *a = i;            // Populate the allocated heap memory.
    setB(i);           // Invoke setter to populate the other one.
    return 0;          // Return successful status.
}
```

[4] If the global heap is to be entirely avoided, 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 **?**.

[5] 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.

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

extern int setA(int);   // declaration (only) of setter in other TU
int x = setA(5);        // Initialize a and b.
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 wind up dereferencing a null pointer before entering `main`:

```
$ g++ a.cpp b.cpp main.cpp
$ ./a.out
  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: (make analogous changes)
```

Now the program reliably executes without incident:

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

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 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.

While on the surface it may seem as though local and nonlocal objects of static storage duration are effectively interchangeable, this is clearly not the case. Even when clients cannot

directly access the nonlocal object due to giving it **internal linkage** by marking it **`static`** or putting it in an **unnamed namespace**, the initialization behaviors make such objects behave very differently.

ursive-initialization

Dangerous recursive initialization

As with all other initialization, control flow does not continue *past* the definition of a **`static`** local object until after the initialization is complete, making recursive **`static`** initialization — or any initializer that might eventually call back to the same function — dangerous:

```
int fz(int i)  // The behavior of the first call 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);  // Bug, e.g., due to possible deadlock
}
```

In the example above, the second recursive call of `fz` to initialize `dz` has undefined behavior because the control flow reached the same definition again before the initialization of the **`static`** object was completed; hence, control flow cannot continue to the **`return`** statement in `fz`. Given a likely implementation with a nonrecursive mutex or similar lock, the program can potentially deadlock, though many implementations provide better diagnostics with an exception or assertion violation when this form of error is encountered.[6]

leties-with-recursion

Subtleties with recursion

Even when not recursing within the initializer itself, the rule for the initialization of **`static`** objects at function scope becomes more subtle for self-recursive functions. Notably, the initialization happens based on when flow of control first passes the variable definition and *not* based on the first invocation of the containing function. Due to this, when a recursive call happens in relation to the definition of a **`static`** local variable impacts which values might be used for the initialization:

[6] Prior to standardization (see **?**, 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++.

```
    assert

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.

ects-after-main-returns

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 60) 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 clearly known at compile time, as it will be the reverse of the order in which they are initialized, 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.

Logging in the constructor of the `FileManager` makes it 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 contraindicated. The two most common alternatives discussed at the end of *Use Cases — Meyers Singleton* on page 60 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 runtime system.

Annoyances

annoyances

Overhead in single-threaded applications

e-threaded-applications

A single-threaded application invoking a function containing a function-scope static storage duration variable might have unnecessary synchronization overhead, such as an **atomic** load operation. For example, consider a program that accesses a simple Meyers Singleton for a user-defined type with a **user-provided** 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 inlined.[7]

See Also

see-also

[7]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 30); doing so will turn `S` into a **trivially-constructible** type, implying that no code needs to be executed during initialization:

Further Reading

further-reading

- ?
- For an in-depth discussion of the difficulties of implementing double-checked locking in C++03, see ?.
- ?
- For a discussion of the Singleton pattern and a variety of implementations in C++03, see Chapter 6 of ?.

Appendix

pendix-functionstatic

C++03 double-checked-lock pattern

-checked-lock-pattern

Prior to the introduction of the function-scope **`static`** object initialization guarantees discussed in *Description* on page 57, preventing multiple initializations of **`static`** objects and use before initialization of those same objects was still needed. Guarding access using a `mutex` was often a significant performance cost, so using the unreliable, double-checked lock pattern was often attempted to avoid the overhead:

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

    if (!loggerPtr)  // Does the logger need to be initialized?
    {
        static 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 partial substitute for an atomic variable, a non-portable solution that is not correct in standard C++ but has historically been moderately effective. The C++11 standard library does, however, provide the `<atomic>` header, which is a far superior alternative, and many implementations have historically provided extensions to support atomic types even prior to C++11. Where available, compiler extensions are typically preferable over home-grown solutions.

```
    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.

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. This reordering of instructions would then allow multiple threads to acquire the lock while each thinking the **`static`** variable still needs to be initialized.

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++03 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 60):

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

Local/Unnamed Types as Template Arguments

as-template-arguments

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

Description

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 22) 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 393), as such types are both unnamed and local in typical usage:

[1] ?

```
#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)
              {
                  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

Use Cases

-type-within-a-function

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:

```
    std::vector
// 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]

cts-as-type-arguments

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:

[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.

```
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 393) 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*.

-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]:

[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):

`std::vector`

```
// 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
                       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

Potential Pitfalls

annoyances

Annoyances

see-also

See Also

- "**decltype**" (§1.1, p. 22) ♦ describes how developers may query the type of any expression or entity, including objects with unnamed types.
- "Lambdas" (§2.1, p. 393) ♦ provides strong practical motivation for the relaxations discussed here.

further-reading

Further Reading

The `long long` (≥64 bits) Integral Type

long-long

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

Description

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 81.

Use Cases

use-cases

Storing values that won't safely fit in 32 bits

safe-might-not-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 **?**, Chapter 5, "Integer Security," pp. 225–307.

fundamental integral types on page 81), and (2) would reduce portability (see *Potential Pitfalls — Relying on the relative sizes of* **int**, **long**, *and* **long long** on page 80).

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
```

```
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.

ng-potential-pitfalls

, -long,-and-long-long

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 81, 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 `typedef`s found in `<cstdint>` (since C++11)

longlong-table1

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

See Also

- "Binary Literals" (§1.2, p. 131) ♦ 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. 141) ♦ describes visually separating digits of large **`long long`** literals.

further-reading

Further Reading

- ?

longlong-appendix

Appendix

Historical perspective on the evolution of use of fundamental integral types

damental-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:

`CHAR_BIT`

```
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 80 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

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

Description

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

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

Use Cases

use-cases

Better compiler diagnostics

er-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]]`:

```
    std::abort
```

```
[[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:

```
    std::abortstd::cout
[[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.

d-runtime-performance

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

turn-potential-pitfalls

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

herwise-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:

```
std::coutassert
[[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)**.

]-on-function-pointers

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

annoyances

See Also

see-also

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

Further Reading

further-reading

- ?
- ?

inter-literal-(nullptr)

The Null-Pointer-Literal Keyword

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

Description

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

use-cases

Improvement of type safety

improve-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:

[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)."

```
    NULL
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:

```
    NULL
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

ing-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:

```
    NULL
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.)

-literal-null-pointer

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."**??**" on page **??**):

```
    std::size_t

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

Potential Pitfalls

Annoyances

annoyances

See Also

see-also

Further Reading

further-reading

- ?

The `override` Member-Function Specifier

override

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

Description

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

Use Cases

ass-is-being-overridden

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

potential-pitfalls

Lack of consistency across a codebase

ncy-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

annoyances

See Also

see-also

further-reading

Further Reading

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

Syntax for Unprocessed String Contents

raw-string-literals

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

Description

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 97). 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\" }"
```

Recall that, 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 entering a newline into the source (i.e., making the string literal span lines of source code) is an error. 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 and horizontal whitespace characters present in the source file as part of the string contents:

```
const char s1[] = R"(line one
line two
    line three)";
    // OK
```

In this example, we assume that all trailing whitespace has been stripped since even trailing whitespace in a raw literal would be captured. 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 100. 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

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:

```
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 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 delimiter of a raw string literal can comprise any member of the **basic source character set** except space, backslash, parentheses, and the control characters representing horizontal tab, vertical tab, form feed, and new line.

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

- an uppercase `R`
- 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 specified previously, if any (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:

```
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:

```
    printf
```

```
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

Use Cases

g-code-in-a-c++-program

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 when transforming between source and data — such as when copying the contents of the string literal into an online regular-expression validation tool — introducing significant opportunities for human error. Using a raw string literal solves these problems:

```
const char* productIdRegex = R"([0-9]{5}\(".*"\))";
```

Another format that benefits from raw string literals is JSON, due to its frequent use of double quotes:

```
const char* testProductResponse = R"!(
{
    "productId": "58215(\"Camera\")",
    "availableUnits": 5,
    "relatedProducts": ["59214(\"CameraBag\")", "42931(\"SdStorageCard\")"]
})!";
```

With a conventional string literal, the JSON string above would require every occurrence of `"` and `\` to be escaped and every new line to be represented as `\n`, resulting in visual noise, less interoperability with other tools accepting or producing JSON, and heightened risk during manual maintenance.

Finally, raw string literals can also be helpful for whitespace-sensitive languages, such as Python (but see *Potential Pitfalls — Encoding of new lines and whitespace* on page 101):

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

alls-rawstringliteral

Potential Pitfalls

nexpected-indentation

Unexpected indentation

Consistent indentation and formatting of source code facilitates human comprehension of program structure. Space and tabulation (`\t`) characters used for the purpose of source code formatting are, however, always interpreted as part of a raw string literal's contents:

```
    std::cout

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';
}
```

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 toward avoiding unexpected indentation issues.

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';
}
```

newlines-and-whitespace

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.[1] 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:

[1] **?**

```
   std::stringassert

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.

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 considered 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.

`annoyances`

Annoyances

`see-also`

See Also

`further-reading`

Further Reading

Compile-Time Assertions

ertions-(static_assert)

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

Description

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

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 110) **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.");
```

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

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

template <typename T>
struct S
{
    void f0()
    {
        static_assert(1 + 1 == 3, "Always fires.");  // block scope
    }

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

Providing a 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

sertions-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:

```
    std::complex

template <typename T>
void f3()
```

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

```
{
    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 107. 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
    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

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

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 107.

use-cases

Use Cases

t-the-target-platform

Verifying assumptions about the target platform

Some programs rely on specific properties of the native types provided by their target platform. Static assertions can help ensure portability and prevent such programs from being compiled into a malfunctioning binary on 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.");
```

nd-function-templates

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 158), 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];
```

[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.

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

    // ...
};
```

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

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

atic-potential-pitfalls

Static assertions in templates can trigger unintended compilation failures

ed-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.

-restrict-overload-sets

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

static-annoyances

Mandatory string literal

datory-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:

```
    std::is_integral

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
```

[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.

see-also

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

Further Reading

- ?

Trailing Function Return Types

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

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 677) 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 92 and Section 3.1."**`final`**" on page 625), 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);
```

[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).

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:

```
#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

use-cases

Function template whose return type depends on a parameter type

s-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 22).

Avoiding having to qualify names redundantly in return types

antly-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

[3] ?

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:

```
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.

lving-function-pointers

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

Potential Pitfalls

annoyances

Annoyances

see-also

See Also

- "**decltype**" (§1.1, p. 22) ♦ Function declarations may use **decltype** either in conjunction with, or as an alternative to, trailing return types.
- "Deduced Return Type" (§3.2, p. 687) ♦ 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

Further Reading

- ?

Unicode String Literals

-and-character-literals

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

Description

scription-unicodestring

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

unicodestring-table1

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:

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

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

This output statement is guaranteed to emit the cocktail emoji (🍸) to `stdout`, assuming that the receiving end is configured to interpret output bytes as UTF-8.

Use Cases

use-cases

Guaranteed-portable encodings of literals

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:

`std::stringstd::ostream`

```
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

potential-pitfalls

Embedding Unicode graphemes

ing-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"`).

[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.

ary-support-for-unicode

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
#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.

utf-8-quirks

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`**, respectively. 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. 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.

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

`annoyances`

See Also

`see-also`

Further Reading

`further-reading`

ons-and-alias-templates

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

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:

```
    std::pair

// 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

use-cases

Simplifying convoluted `typedef` declarations

-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`**."

ding-template-arguments

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:

```
std::mapstd::set
```

```
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:

[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 **?**.

```
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;
};
```

ation-for-type-traits

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:

```
   std::remove_reference

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

Potential Pitfalls

Annoyances

annoyances

See Also

see-also

- "Inheriting Ctors" (§2.1, p. 375) ♦ 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

further-reading

sec-safe-cpp14

Aggregates Having Default Member Initializers

itialization-relaxation

C++14 enables the use of **aggregate initialization** with classes employing default member initializers (see Section 1.1."Default Member Init" on page 296).

Description

description

Prior to C++14, classes that used default member initializers (see Section 1.1."Default Member Init" on page 296) — 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 198):

```
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:

```
    assert
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

Use Cases

configuration-structs

Configuration `structs`

Aggregates in conjunction with default member initializers (see Section 1.1."Default Member Init" on page 296) can be used to provide concise customizable configuration **`struct`**s, packaged with typical default values. As an example, consider a configuration **`struct`** for an HTTP request handler:

```
struct HTTPRequestHandlerConfig
{
    int maxQueuedRequests = 1024;
    int timeout           = 60;
    int minThreads        = 4;
    int maxThreads        = 8;
};
```

Aggregate initialization can be used when creating objects of type `HTTPRequestHandlerConfig` (above) to override one or more of the defaults in definition order[1]:

```
HTTPRequestHandlerConfig getRequestHandlerConfig(bool inLowMemoryEnvironment)
{
    if (inLowMemoryEnvironment)
    {
        return HTTPRequestHandlerConfig{128};
            // timeout, minThreads, and maxThreads have their default value.
    }
    else
    {
        return HTTPRequestHandlerConfig{2048, 120};
            // minThreads, and maxThreads have their default value.
    }
}

// ...
```

[1] 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.
```

potential-pitfalls

Potential Pitfalls

Empty list initializing members does not use the default initializer

When we add default member initializers to members of an aggregate, those initializers are in effect *only* if that member has no corresponding initializer in the braced initializer list. We cannot explicitly *request* the default value by placing empty braces into the list because that will value-initialize the member with an empty initializer, not use its default member initializer. Any case where we want to explicitly initialize a later member variables value during aggregate initialization means we must manually determine the proper default for all prior members:

```
struct A
{
     int i{1};
     int j{2};
     int k{3};
};

A a1{};            // OK, result is i=1, j=2,         k=3
A a2{ 4 };         // OK, result is i=4, j=2,         k=3
A a3{ 4, {}, 8 };  // Bug, result is i=4, j=0 (not 2), k=3
```

annoyances

esence-of-brace-elision

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 296). 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

See Also

- "Default Member Init" (§1.1, p. 296) ♦ allows developers to provide a default initializer for a data member directly in the definition of a class.
- "Braced Init" (§2.1, p. 198) ♦ introduces a syntactically similar feature for initializing objects in a uniform manner.

further-reading

Further Reading

binary-literals

Binary Literals: The `0b` Prefix

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

description-binary

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 131. 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 183.) 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

Use Cases

nd-bitwise-operations

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 310.

ng-constant-binary-data

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:

`std::uint8_t`

```
#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

Potential Pitfalls

annoyances

Annoyances

see-also

See Also

- "Digit Separators" (§1.2, p. 141) ♦ explains grouping digits visually to make long binary literals much more readable.

further-reading

Further Reading

- ?

The [[deprecated]] Attribute

eprecated]] deprecated

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

Description

description

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

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

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

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

                int a;
[[deprecated]] int b;

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

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

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

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

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

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

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

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

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

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

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

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

Use Cases

use-cases

Discouraging use of an obsolete or unsafe entity

solete-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. The `rand` function, inherited from C and available in C++ through the `<cstdlib>` header, has many of the same issues as our `RandomGenerator::nextRandom` function, and similarly developers are guided to use the facilities provided in the `<random>` header since C++11.

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

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

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

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

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

By using `[[deprecated]]` as shown 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.

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

`potential-pitfalls`

`c,-clang)-or-/wx-(msvc)`

Potential Pitfalls

Interaction with treating warnings as errors

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

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

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

`deprecated`

Annoyances

`annoyances`

See Also

`see-also`

Further Reading

`further-reading`

The Digit Separator: '

digitseparators

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

Description

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;
    //                           ^~~~~~~~~~~~~~~~~~~~~~~
```

[1] Tested on GCC 7.4.0.

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:

```
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 143.

Use Cases

use-cases

Grouping digits together in large constants

er-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

digitseparator-table1

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

digitseparator-table2

Without Digit Separator	With Digit Separators
0b1100110011001100	0b1100'1100'1100'1100
0b0110011101011011	0b0110'0111'0101'1011
0b1100110010101010	0b11001100'10101010

potential-pitfalls

Potential Pitfalls

see-also

See Also

- "Binary Literals" (§1.2, p. 131) ♦ represents a binary constant for which digit separators are commonly used to group bits in octets (bytes) or quartets (nibbles)

further-reading

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-digitsep

Appendix

floating-point-literals

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

digitseparator-table3

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(3.14159265358f != 3.141592653f, "too precise for float");
    // This assert will fire on a typical platform.

static_assert(3.14159265358f != 3.14159265f, "too precise for float");
    // This assert too will fire on a typical platform.

static_assert(3.1415926535f != 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

[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));
```

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 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

variable-templates

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.

etemplate-description

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:

```
    std::cout
#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** in the last position:

```
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 152):

```
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);
}
```

iabletemplate-use-cases

parametrized-constants

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 146, 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`**):

```
template <typename T>
constexpr T pi(3.1415926535897932385);
    // smallest digit sequence to accurately represent pi as a long double
```

For portability, a floating-point literal value of π that provides sufficient precision for the longest **`long double`** on any relevant platform (e.g., 34 decimal digits for 128 bits of precision: `3.141'592'653'589'793'238'462'643'383'279'503`) should be used; see Section 1.2."Digit Separators" on page 141.

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));
}
```

posity-of-type-traits

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 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.[1]

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:

[1] 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:

```
std::is_default_constructible

// C++11/14
bool dc1 = std::is_default_constructible<T>::value;

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

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.

```
// 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
#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

te-potential-pitfalls

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

re-const-or-constexpr

Instantiating variable templates that are defined recursively might have a subtle issue that could produce different results[2] 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`**:

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

```
#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
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 nonintegral constants.

Annoyances

annoyances

Variable templates do not support template template parameters

e-template-parameters

Although a class or function template can accept a **template template class parameter**, no equivalent construct is available for variable templates[3]:

```
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

see-also

- "**constexpr** Variables" (§2.1, p. 282) ♦ Conditionally safe C++11 feature providing an alternative to **const** template variables that can reduce unnecessary consumption of the static data space.

[3] 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 **?**.

Further Reading

further-reading

Chapter 2

Conditionally Safe Features

ch-conditional

sec-conditional-cpp11 Intro text should be here.

The `alignas` Decorator

alignas

The **alignas** storage specifier is used to strengthen the alignment of a **struct**, **class**, **union**, **enum**, variable, or data member.

Description

description-alignas

Each object type in C++ has an **alignment requirement** that restricts the addresses at which an object of that type is permitted to reside within the virtual-memory-address space. The alignment requirement is imposed by the object type on all objects of that type. The **alignas** specifier provides a means of specifying a stricter alignment requirement than dictated by the type itself for a particular variable of the type or an individual data member of a **user-defined type (UDT)**. The **alignas** specifier can also be applied to a UDT itself, but see *Potential Pitfalls — Applying* **alignas** *to a* type *might be misleading* on page 166.

Supported alignments

-supported-alignments

An alignment value is an integer of type `std::size_t` that represents the number of bytes between the addresses at which a given object may be allocated. All alignment values in C++ are always non-negative powers of two and are divided into two categories depending on whether they are larger than the alignment requirement of the `std::max_align_t` type. The `std::max_align_t` type's alignment requirement is at least as strict as that of every **scalar type**. An alignment value of less than or equal to the alignment requirement of `std::max_align_t` is a **fundamental alignment**; otherwise, it is an **extended alignment**. The `std::max_align_t` type is typically an alias to the largest scalar type, which is **long double** on most platforms, and its alignment requirement is usually 8 or 16.

Fundamental alignments are required to be supported in *all* contexts — i.e., for variables with automatic, static, and dynamic storage durations as well as for nonstatic data members of a class and for function arguments. While all fundamental and pointer types have fundamental alignments, their specific values are **implementation defined** and may differ between platforms. For example, the alignment requirement of type **long** might be `4` on MSVC and `8` on GCC.

In contrast, whether *any* extended alignment is supported at all and, if so, in which contexts, is implementation defined.[1] For example, the strictest supported extended alignment for a variable with static storage duration might be as large as 2^{28} or 2^{29} or as small as 2^{13}.

Since many aspects pertaining to the alignment requirements are implementation defined, we will use a specific platform to illustrate the behavior of **alignas** throughout this

[1] 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 false sharing among distinct objects in a multi-threaded program* on page 163. Note that, in the case of `i0` in the **alignas**`(32)` line in the last code snippet on page 159, a conforming platform that did not support an extended alignment of 32 would be required to report an error at compile time. [PRODUCTION: update the page xref as needed at FPPs. We may need to say "on this page" or otherwise clarify once pagination is set.]

feature's section. Accordingly, the examples below show the behavior observed for the Clang compiler targetting desktop x86-64 Linux.

-of-a-particular-object

Strengthening the alignment of a particular object

In its most basic form, the **alignas** specifier strengthens the alignment requirement of a particular object. The desired alignment requirement is an **integral constant expression** provided as an argument to **alignas**:

```
alignas(8) int i;     // OK, i is aligned on an 8 byte address boundary.
int j alignas(8), k;  // OK, j is 8 byte aligned; alignment of k is unchanged.
```

If more than one alignment pertains to a given object, the strictest 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 larger or equal to what the alignment requirement would be without the **alignas** specifier.
3. Be supported on the platform in the context in which the entity appears.

fnref

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

```
// Static variables declared at namespace scope
alignas(32) int i0; // OK, 32-byte aligned (extended alignment)
alignas(16) int i1; // OK, 16-byte aligned (strictest fundamental alignment)
alignas(8)  int i2; // OK,  8-byte aligned (fundamental alignment)
alignas(7)  int i3; // Error, not a power of two
alignas(4)  int i4; // OK, no change to alignment requirement
alignas(2)  int i5; // Error, less than alignment would be without alignas
alignas(0)  int i6; // OK, alignas specifier ignored

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

alignas(1 << 30) int i8;
    // Error, exceeds maximum supported extended alignment

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

void f()
{
  // automatic variables declared at function scope
  alignas(4)  double e0; // Error, less than alignment would be without alignas
```

```
  alignas(8)  double e1; // OK, no change to (8-byte) alignment requirement
  alignas(16) double e2; // OK, 16-byte aligned (fundamental alignment)
  alignas(32) double e3; // OK, 32-byte aligned (extended alignment)
}
```

dividual-data-members

Strengthening the alignment of individual data members

Within a user-defined type (**class**, **struct**, or **union**), using 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 z;  // 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 alignment specifier pertains to a given data member, the strictest 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
```

f-a-user-defined-type

Strengthening 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.[2]

Again, specifying an alignment that is less than what would be without the **alignas** specifier 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).
```

ignment-of-another-type

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)) (see Section 2.1."**alignof**" on page 173):

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

alignas-use-cases

Use Cases

y-aligned-object-buffer

Creating a sufficiently aligned object buffer

One of motivating use cases for introducing the **alignas** feature was creating static capacity, dynamic size containers that do not use dynamic allocations:

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

template <typename TYPE, std::size_t CAPACITY>
class FixedVector {
```

To avoid the overhead of initializing the unused elements, such a generic container needs to have a character buffer data member and employ placement **new** to construct the elements as needed. This buffer needs to be of sufficient size to store the elements, which one can easily compute using CAPACITY * **sizeof**(T). In addition, however, ensuring that the buffer is sufficiently aligned to store elements of the supplied TYPE is important. With **alignas**, ensuring this requirement is straightforward:

```
    alignas(TYPE) char d_buffer[CAPACITY * sizeof(TYPE)];
        // raw memory buffer of proper size and alignment for TYPE elements

    std::size_t        d_size;
        // current size of the vector
```

[2]When **alignas** is applied to an enumeration E, the Standard does not indicate whether padding bytes 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 **?**. The outcome of the core issue was to completely remove permission for **alignas** to be applied to enumerations (see **?**). Therefore, conforming implementations will eventually stop accepting the **alignas** specifier on enumerations in the future.

```
    TYPE *rawElementPtr(std::size_t index)
        // Return the pointer to the element with the specified index.
    {
        return reinterpret_cast<TYPE *>(d_buffer) + index;
    }

public:
    // ...

    void resize(std::size_t size)
    {
        assert(size <= CAPACITY);
        while (d_size < size) new (rawBufferPtr(d_size++)) TYPE;
        while (d_size > size) rawElementPtr(--d_size)->~TYPE();
    }

    // ...
};
```

Without the use of **alignas**, `d_buffer` (in the code snippet above), which is an array of characters, would itself have an alignment requirement of 1. The compiler would therefore be free to place it on any address boundary, which is problematic for any `TYPE` argument with an alignment requirement stricter than 1.

specific-instructions

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)*[3] 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.[4] 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 <cassert>     // standard C assert macro
#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};
```

[3] **?**, "Technologies: SSE"

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

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

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

-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 170.

As a simple 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

void incrementJob(int* p);
    // Repeatedly increment *p a large, fixed number of times.

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 using **alignas** to strengthen the alignment requirement of both integers to the cache line size, we ensure that the two variables reside on distinct cache lines:

```
// ...

enum { k_CACHE_LINE_SIZE = 64 };  // A cache line on this platform is 64 bytes.

void f()
{
    alignas(k_CACHE_LINE_SIZE) int i0 = 0; // i1 and i2 are on separate
    alignas(k_CACHE_LINE_SIZE) int i1 = 0; // cache lines.

    // ...
}
```

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**.[5] Note that because supported extended alignments are implementation defined, using **alignas** is not a strictly portable solution. Opting for less elegant and more wasteful padding approach instead of **alignas** might be preferrable for portability.

e-thread-aware-object

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),[6] thereby preventing them from occupying the same one.

```
#include <atomic>  // std::atomic
class ThreadSafeRingBuffer
{
    alignas(k_CACHE_LINE_SIZE) std::atomic<std::size_t> d_head;
    alignas(k_CACHE_LINE_SIZE) std::atomic<std::size_t> d_tail;
```

[5] 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.

[6] In C++17, one can portably retrieve the minimum offset between two objects to avoid false sharing through the `std::hardware_destructive_interference_size` constant defined in the **<new>** header.

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

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 — Overlooking alternative approaches to avoid false sharing* on page 167.

Potential Pitfalls

ential-pitfalls-alignas

Underspecifying alignment is not universally reported

ot-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).

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;              // Error, alignas(4) is below minimum, 8.

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

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

Each of the three errors above are reported by Clang. MSVC and ICC issue a warning, whereas GCC provides no diagnostic at all, 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

ying-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:

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

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

[9] Underspecifying alignment is not reported at all by GCC 10.2, using the `-std=c++11 -Wall -Wextra -Wpedantic` flags. This behavior is reported as a compiler defect; see **?**. With the same set of options, Clang 10.1 produces a compilation failure. ICC 2021.1.2 and MSVC v19.28 will produce a warning and ignore any alignment less than the minimum one.

```
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, def. equiv. to each decl. specifying alignment

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 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, nonequivalent alignas specifier
```

Both of the errors above are flagged by Clang. MSVC and ICC warn on the first one and produce an error on the second one, whereas neither of them is reported by GCC. Note that when the inconsistency *occurs across translation units*, no mainstream compiler is likely to diagnose the problem:

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

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

Any program incorporating both translation units above is IFNDR.

e-might-be-misleading

Applying `alignas` to a *type* might be misleading

When applying the **`alignas`** specifier to a user-defined type having no base classes, one might be convinced that it is equivalent to applying **`alignas`** to its first declared data member:

```
struct S0 {
    alignas(16) char d_buffer[128];  // guaranteed to be 16-byte aligned
                int  d_index;
};

struct alignas(16) S1 {
    char d_buffer[128];              // guaranteed to be 16-byte aligned
    int  d_index;
};
```

Indeed, for all objects of the `S0` and `S1` types (in the example above), their respective `d_buffer` data members will be aligned on a 16-byte boundary. Such equivalency, however,

holds only for **standard layout types**. Adding a virtual function or even simply changing the access control for some of the data members[10] might break this equivalency:

```
struct S2 {
    alignas(16) char d_buffer[128];  // guaranteed to be 16-byte aligned
                int  d_index;

    virtual ~S2();
};

struct alignas(16) S3 {
    char d_buffer[128];              // not guaranteed to be 16-byte aligned
    int  d_index;

    virtual ~S3();
};

struct S4 {
    alignas(16) char d_buffer[128];  // guaranteed to be 16-byte aligned
private:
                int  d_index;
};

struct alignas(16) S5 {
    char d_buffer[128];              // not guaranteed to be 16-byte aligned
private:
    int  d_index;
};
```

Any code that relies on the `d_buffer` member of instances of the `S3` and `S5` types (in the code example above) being 16-byte aligned is defective.

Overlooking alternative approaches to avoid false sharing

idea avoid false sharing

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 strengthened 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 170.

See Also

see-also

- "**alignof**" (§2.1, p. 173) ♦ inspects the alignment of a given type.

[10] According to the C++20 Standard, compilers are allowed to reorder data members having different access control. However, no compilers take advantage of this ability in practice, and C++23 might mandate that the data members are always laid out in declaration order; see **?**.

Further Reading

further-reading

None so far

Appendix

alignas-appendix

Natural Alignment

natural-alignment

Many micro-architectures are optimized for working with data that has **natural alignment**, i.e., objects reside on an address boundary that divides their size rounded up to the nearest power of two. With the additional restriction that no padding is allowed between C++ array elements, the alignment requirements of fundamental types are often equal to their respective size on most platforms:

```
char        c;  // size 1;  alignment 1;  boundaries: 0x00, 0x01, 0x02, ...
short       s;  // size 2;  alignment 2:  boundaries: 0x00, 0x02, 0x04, ...
int         i;  // size 4;  alignment 4;  boundaries: 0x00, 0x04, 0x08, ...
float       f;  // size 4;  alignment 4;  boundaries: 0x00, 0x04, 0x08, ...
double      d;  // size 8;  alignment 8;  boundaries: 0x00, 0x08, 0x10, ...
long double l;  // size 16; alignment 16; boundaries: 0x00, 0x10, 0x20, ...
```

The alignment requirement of an array of objects is the same as that of its elements:

```
char  arrC[4];  // size 4; alignment 1
short arrS[4];  // size 8; alignment 2
```

For user-defined types, compilers compute the alignment and add appropriate padding between the data members and after the last one such that all alignment requirements of the data members are satisfied and no padding would be required should an array of the type be created. Typically, this results in the alignment requirement of a UDT being the same as that of the most strictly aligned nonstatic data member:

```
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 and virtual base classes 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 target platform:

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

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

```
};              // size 16; alignment 8
```

s,-and-virtual-memory

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, possibly running on separate cores or even processors. 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 un-

[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 **?**, section 5.2.1, "Placement Policy," pp. 136–141, and **?**) 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.

related 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 colocation of such data on the same cache line impossible. Another (more broad-based) design approach that avoids lowering cache utilization is to ensure that data acted upon by a given thread is kept physically separate — e.g., through the use of local (arena) memory allocators.[12]

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

[12] **?**, **?**, **?**

[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**), is what typically leads to a substantial runtime performance degradation (due to cache line **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] Beneficial prefetching 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) is sometimes referred to as **true sharing**.

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

table-alignas-appendix

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

The (Compile-Time) `alignof` Operator

alignof

The keyword **alignof** serves as a compile-time operator used to query the **alignment requirements** of a type on the current platform.

Description

description

The **alignof** operator, when applied to a type, evaluates to an **integral constant expression** that represents the alignment requirement of its argument type. Similar to **sizeof**, the (compile-time) value of **alignof** is of type `std::size_t`; unlike **sizeof** that can accept an arbitrary expression, **alignof** is defined for only type identifiers but often works on expressions anyway (see *Annoyances* on page 181). The argument type, `T`, supplied to **alignof** must be 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. For example, on a platform where **sizeof**`(short) == 2` and **alignof**`(short) == 2`, the following assertions pass:[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 10)
static_assert(alignof(short[])  == 2, "");  // array type      (sizeof fails)
```

`alignof` Fundamental Types

ignof-fundamental-types

Like their size, the alignment requirements of a **char**, **signed char**, and **unsigned char** are guaranteed to be 1 on every conforming platform. For any other fundamental or pointer type `FPT`, **alignof**`(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

gnof-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. Compilers will (by default) avoid nonessential padding because any extra padding would be wasteful of (e.g., cache) memory:

```
struct S0 { };                          // sizeof(S0) is  1; alignof(S0) is  1
struct S1 { char c; };                  // sizeof(S1) is  1; alignof(S1) is  1
```

[1] According to the C++11 Standard, "An object of array type contains a contiguously allocated nonempty set of `N` subobjects of type `T`" (**?**, section 8.3.4, "Arrays," paragraph 1, p. 188). Note that, for every type `T`, **sizeof**`(T)` is always a multiple of **alignof**`(T)`; otherwise, storing multiple `T` instances in an array would be impossible without padding, and the Standard explicitly prohibits padding between array elements.

```
struct 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 ld; };          // sizeof(SA) is 16; alignof(SA) is 16
struct SB { long double ld; char c; };  // sizeof(SB) is 32; alignof(SB) is 16
```

The size of empty types, such as `S0` above, are defined to have the size and alignment of 1 to ensure that each object and member subobject of type `S0` has a unique address. However, if an empty type is used as a base, the size of the derived type will not be affected (with some exceptions) due to the **empty base optimization**:

```
struct D0 : S0 { int i; };  // sizeof(D0) is 4; alignof(D0) is 4
```

The alignment of the base type always affects the derived type's alignment. However, this is observable only for an empty base if it is **over-aligned** (see Section 2.1."**alignas**" on page 158):

```
struct alignas(8) E { };   // sizeof(E)  is 8; alignof(E)  is 8
struct D1 : E { int i; };  // sizeof(D1) is 8; alignof(D1) is 8
```

Compilers are permitted to increase alignment (e.g., in the presence of virtual functions, which typically implies a virtual function table pointer) but have certain restrictions on padding. For example, they must ensure that each comprised type is itself sufficiently aligned. Furthermore, sufficient padding must be added so that the alignment of the parent type divides its size, ensuring that storing multiple instances in an array does not require any padding between array elements (which is explicitly prohibited by the Standard). In other words, the following identities hold for all types, `T`, and positive integers, `N`:

```
template <typename T, std::size_t N>
void f()
{
    static_assert(0 == sizeof(T) % alignof(T), "guaranteed");

    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** specifier (see Section 2.1."**alignas**" on page 158). 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 (see Section 2.1."Generalized PODs '11" on page 374):

```
struct S { int i; };
class T { public: S s; };
T t;
static_assert(static_cast<void*>(&t.s) == &t, "guaranteed");
static_assert(static_cast<void*>(&t.s) == &t.s.i, "guaranteed");
```

This property also holds for unions:

```
struct { union { char c; float f; double d; }; } u;
static_assert(static_cast<void*>(&u) == &u.c, "guaranteed");
static_assert(static_cast<void*>(&u) == &u.f, "guaranteed");
static_assert(static_cast<void*>(&u) == &u.d, "guaranteed");
```

use-cases

Use Cases

type-during-development

Probing the alignment of a type during development

Both **sizeof** and **alignof** are often used informally during development and debugging to confirm the values of those attributes for a given type on the current platform. For example:

```
#include <iostream>

void f()
{
    std::cout << " sizeof(double): " <<  sizeof(double) << '\n';  //  always 8
    std::cout << "alignof(double): " << alignof(double) << '\n';  // usually 8
}
```

Printing the size and alignment of a **struct** along with those of each of its individual data members can lead to the discovery of suboptimal ordering of data members, resulting in wasteful extra padding. As an example, consider two **struct**s, `Wasteful` and `Optimal`, having the same three data members but in different order:

```
struct Wasteful
{
    char   d_c;  // size =  1;  alignment = 1
    double d_d;  // size =  8;  alignment = 8
    int    d_i;  // size =  4;  alignment = 4
};               // size = 24;  alignment = 8

struct Optimal
{
    double d_d;  // size =  8;  alignment = 8
    int    d_i;  // size =  4;  alignment = 4
    char   d_c;  // size =  1;  alignment = 1
};               // size = 16;  alignment = 8
```

Both **alignof**(`Wasteful`) and **alignof**(`Optimal`) are `8` on our platform 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
```

-sufficiently-aligned

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[2]:

```
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 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 — typically having a **union** of a **char** array and a **char** pointer as a data member) — will naturally result in the alignment requirement of

[2]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>()) { /* ... */ }
}
```

at least that of the **char*** (e.g., 4 on a 32-bit platform and 8 on a 64-bit one)[3]:

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

class MyAny  // nontemplate class
{
    union {
        char *d_buf_p;       // pointer to dynamic memory if needed
        char  d_buffer[39];  // small buffer
    };  // Size of union is 39; alignment of union is alignof(char*).

    char 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).
```

We chose the size of `d_buffer` in the example above to be 39 for two reasons. First, we decided that we want 32-byte types to fit into the buffer, meaning that the size of `d_buffer` should be at least 32. Combined with the use of **char** for the `d_onHeapFlag`, which is guaranteed to have the size of 1, this means that **sizeof**`(MyAny) >= 33`. Second, we want to ensure that no space is wasted on padding. On platforms where **alignof**`(MyAny)` is 8, which will be the case for many 64-bit platforms, **sizeof**`(MyAny)` would be 40, which we choose to achieve by increasing the useful capacity to 39 instead of having the compiler add unused padding.

The (templated) constructor[4] 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>
```

[3]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
// ...
```

[4]In a real-world implementation, among other improvements, 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 Section 2.1."Forwarding References" on page 351.

```
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 its address in the buffer.
        d_buf_p = reinterpret_cast<char*>(new T(x));
        d_onHeapFlag = true;
    }
}
```

Using the **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).
```

nic-memory-allocation

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.[5] Especially when the memory is allocated by a single thread, there are prodigious[6] 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.

[5] 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`.

[6] see **?**

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:

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

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) { }
       // Initialize the next available address to be the start of the buffer.

    template <typename T>
    void* allocate()             // BAD IDEA --- doesn't address alignment
                                 // doesn't check buffer limit
    {
        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`. The interesting part is how the `allocate` function manages to return a monotonically increasing 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 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.
{
    return (alignment - reinterpret_cast<std::uintptr_t>(address)) &
           (alignment - 1);
}
```

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 <std::size_t N>
template <typename T>
void* MonotonicBuffer<N>::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[7]:

```
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]
```

[7]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.

In practice, an object that allocates memory, such as a `vector` or a `list`, will be constructed to use an allocator that provides memory that is guaranteed to have either **maximal fundamental alignment**, natural alignment, or alignment that satisfies.

Finally, instead of returning a null pointer when the buffer was exhausted, we would typically have the 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.

Annoyances

annoyances-alignof

`alignof` (unlike `sizeof`) is defined only on types

s-defined-only-on-types

Unlike the **`sizeof`** operator, the **`alignof`** operator can accept only a *type*, not an *expression*, as its argument[8]:

```
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 Section 1.1."**`decltype`**" on page 22) when inspecting an expression instead of a type:

```
void f()
{
    enum { e_SUCCESS, e_FAILURE } 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 Section 2.1."**`auto`** Variables" on page 183) and generic lambdas (see Section 2.2."*Generic* Lambdas" on page 605). 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`

[8] 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.

lambda,[9] a programmer wishing to remain entirely within the C++ Standard[10] is forced to use the **decltype** construct explicitly to first obtain the type of `object` before passing it on to **alignof**.

see-also

See Also

- "**decltype**" (§1.1, p. 22) ♦ helps work around **alignof**'s limitation of accepting only a type, not an expression (see *Annoyances* on page 181).
- "**alignas**" (§2.1, p. 158) ♦ can be used to provide an artificially stricter alignment (e.g., more than natural alignment).

further-reading

Further Reading

None so far

[9] 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';
};
```

[10] Note that **alignof**`(object)` will work on every major compiler (GCC 10.x, Clang 10.x, and MSVC 19.x) as a nonstandard extension.

Variables of Automatically Deduced Type

auto variables

The keyword **auto** was repurposed in C++11 to act as a **placeholder type** such that, when used in place of a type as part of a variable declaration, the compiler will deduce the variable type from its initializer.

Description

description

Prior to C++11, the **auto** keyword could be used as a **storage class specifier** for objects declared at block scope and in function parameter lists to indicate automatic storage duration, which is the default for these kinds of declarations. The **auto** keyword was repurposed in C++11 alongside the deprecation of the **register** keyword as a storage class specifier.[1]

Starting with C++11, the **auto** keyword may be used instead of an explicit type's name when declaring variables. In such cases, the variable's type is deduced from its initializer by the compiler applying the placeholder type deduction rules, which, apart from a single exception for list initializers (see *Potential Pitfalls — Surprising deduction for list initialization* on page 194), are the same as the rules for **function template argument type deduction**:

```
auto two = 2;     // Type of two is deduced to be int
auto pi = 3.14f;  // Type of pi is deduced to be float.
```

The types of the `two` and `pi` variables above are deduced in the same manner as they would be if the same initializer were passed to a function template taking a single argument of the template type by value:

```
template <typename T> void deducer(T);

void testDeduction()
{
    deducer(2);      // T is deduced to be int.
    deducer(3.14f);  // T is deduced to be float.
}
```

If the variable declared with **auto** does not have an initializer, if its name appears in the expression used to initialize it, or if the initializers of multiple variables in the same declaration don't deduce the same type, the program is ill formed:

```
auto x;                // Error, declaration of auto x has no initializer.
auto n = sizeof(n);    // Error, use of n before deduction of auto
auto i = 3, f = 0.3f;  // Error, inconsistent deduction for auto
```

Just like the function template argument deduction never deduces a reference type for its by-value argument, a variable declared with unqualified **auto** is never deduced to have a reference type:

```
int  val = 3;
int& ref = val;
auto tmp = ref;  // Type of tmp is deduced to be int, not int&.
```

[1] The deprecated keyword **register** was removed as of C++17, but the name remains reserved for future use.

Augmenting **auto** with reference and cv-qualifiers, however, allows controlling whether the deduced type is a reference and whether it is **const** and/or **volatile**:

```
auto val = 3;
    // Type of val is deduced to be int,
    // the same as the argument for template <typename T> void deducer(T).

const auto cval = val;
    // Type of cval is deduced to be const int,
    // the same as the argument for template <typename T> void deducer(const T).

auto& ref = val;
    // Type of ref is deduced to be int&,
    // the same as the argument for template <typename T> void deducer(T&).

auto& cref1 = cval;
    // Type of cref1 is deduced to be const int&,
    // the same as the argument for template <typename T> void deducer(T&).

const auto& cref2 = val;
    // Type of cref2 is deduced to be const int&,
    // the same as the argument for template <typename T> void deducer(const T&).
```

Note that qualifying **auto** with && does *not* result in deduction of an *rvalue* reference (see Section 2.1."*rvalue* References" on page 479), but, in line with function template argument deduction rules, would be treated as a *forwarding reference* (see Section 2.1."Forwarding References" on page 351). This means that a variable declared with **auto**&& will result in an *lvalue* or an *rvalue* reference depending on the value category of its initializer:

```
double doStuff();

      int val  = 3;
const int cval = 7;

// Deduction rules are the same as for template <typename T> void deducer(T&&):

auto&& lref1 = val;
    // Type of lref1 is deduced to be int&.

auto&& lref2 = cval;
    // Type of lref2 is deduced to be const int&.

auto&& rref = doStuff();
    // Type of rref is deduced to be double&&.
```

Similarly to references, explicitly specifying that a pointer type is to be deduced is possible. If the supplied initializer is not of a pointer type, the compiler will issue an error:

```
const auto* cptr = &cval;
    // Type of cptr is deduced to be const int*.
    // same as argument for template <typename T> void deducer(const T*)
```

```
auto* cptr2 = cval;  // Error, cannot deduce auto* from cval
```

The compiler can also be instructed to deduce pointers to functions, data members, and member functions (but see *Annoyances — Not all template argument deduction constructs are allowed for* **auto** on page 196):

```
float freeF(float);

struct S
{
    double d_data;
    int memberF(long);
};

auto (*fptr)(float) = &freeF;
  // Type of fptr is deduced to be float (*)(float).
  // same as argument for template <typename T> void deducer(T (*)(float))

const auto S::* mptr = &S::d_data;
  // Type of mptr is deduced to be const double S::*.
  // same as argument for template <typename T> void deducer(T S::*)

auto (S::* mfptr)(long) = &S::memberF;
  // Type of mfptr is deduced to be int (S::*)(long).
  // same as argument for template <typename T> void deducer(T (S::*)(long))
```

Unlike references, pointer types may be deduced by **auto** alone. Therefore, different forms of **auto** can be used to declare a variable of a pointer type:

```
auto  cptr1 = &cval;  // const int*
auto *cptr2 = &cval;  //   "     "

auto   fptr1         = &freeF;  // float (*)(float)
auto  *fptr2         = &freeF;  //   "         "
auto (*fptr3)(float) = &freeF;  //   "         "

auto      mptr1 = &S::d_data;  // double S::*
auto S::* mptr2 = &S::d_data;  //    "     "

auto       mfptr1        = &S::memberF;  // int (S::*)(long)
auto (S::* mfptr2)(long) = &S::memberF;  //  "     "     "
```

Note, however, that because regular and member pointers are incompatible, **auto*** cannot be used to deduce pointers to data members and member functions:

```
auto *mptr3  = &S::d_data;   // Error, cannot deduce auto* from &S::d_data
auto *mfptr3 = &S::memberF;  // Error, cannot deduce auto* from &S::memberF
```

In addition, storage class specifiers as well as the **constexpr** (see Section 2.1."**constexpr** Variables" on page 282) specifier can be applied to variables that use **auto** in their declaration:

```
thread_local     const auto* logPrefix = "mylib";
static constexpr       auto  pi        = 3.1415926535f;
```

Finally, **auto** variables may be declared in any location that allows declaring a variable supplied with an initializer with a single exception of nonstatic data members (see *Annoyances — **auto** not allowed for nonstatic data members* on page 195):

```
    std::vector

// namespace scope
auto globalNamespaceVar = 3.;

namespace ns
{
    static auto nsNamespaceVar = "...";
}

void f()
{
    // block scope
    constexpr auto blockVar = 'a';

    // condition of if, switch, and while statements
    if     (auto rc        = sendRequest())    { /*...*/ }
    switch (auto status    = responseStatus()) { /*...*/ }
    while  (auto keepGoing = haveMoreWork())   { /*...*/ }

    // init-statement of for loops
    for (auto it = vec.begin(); it != vec.end(); ++it) { /*...*/ }

    // range declaration of range-based for loops
    for (const auto& constVal : vec) { /*...*/ }
}

struct S
{
    // static data members
    static const auto k_CONSTANT = 11u;
};
```

use-cases-auto

Use Cases

riable-initialization

Ensuring variable initialization

Consider a defect introduced due to mistakenly leaving a variable uninitialized:

```
void testUninitializedInt()
{
    int recordCount;
    while (cursor.next()) { ++recordCount; }  // Bug, undefined behavior
}
```

Variables declared with **auto** must be initialized. Using **auto** might, therefore, prevent such defects:

```
void testUnitializedAuto()
{
    auto recordCount; // Error, declaration of recordCount has no initializer
    while (cursor.next()) { ++recordCount; }
}
```

In addition, the initialization requirement might encourage a good practice of reducing the scope of local variables.

Avoiding redundant type name repetition

nt-type-name-repetition

Certain function templates require that the caller explicitly specify the type that the function uses as its return type. For example, the `std::make_shared<TYPE>` function returns a `std::shared_ptr<TYPE>`.[2] If a variable's type is specified explicitly, such declarations redundantly repeat the type. The use of **auto** obviates this repetition:

```
    std::shared_ptrstd::unique_ptr

// Without auto:
std::shared_ptr<RequestContext> context1 = std::make_shared<RequestContext>();
std::unique_ptr<Socket>         socket1  = std::make_unique<Socket>();

// With auto:
auto context2 = std::make_shared<RequestContext>();
auto socket2  = std::make_unique<Socket>();
```

Preventing unexpected implicit conversions

ed-implicit-conversions

Using **auto** might prevent defects arising from explicitly specifying a variable's type that is distinct — yet implicitly convertible — from its initializer. As an example, the code below has a subtle defect that can lead to performance degradation or incorrect semantics:

```
    std::mapstd::pair

void testManualForLoop()
{
```

[2] Often, such functions are associated with `optional` and `variant` types, which were standardized in C++17:

```
   std::stringstd::variantstd::optional

std::variant<std::string, int, double> valueVariant;

// Without auto:
std::optional<std::string> greeting1 = std::make_optional<std::string>("Hello");
const std::string& valueString1 = std::get<std::string>(valueVariant);

// With auto:
auto greeting2 = std::make_optional<std::string>("Hello");
const auto& valueString2 = std::get<std::string>(valueVariant);
```

```
    std::map<int, User> users{/*...*/};
    for (const std::pair<int, User>& idUserPair : users)
    {
        // ...
    }
}
```

On every iteration, the `idUserPair` will be bound to a *copy* of the corresponding pair in the `users` map. This happens because the type returned by dereferencing the `map`'s iterator is `std::pair<const int, User>`, which is implicitly convertible to `std::pair<int, User>`. Using **auto** would allow the compiler to deduce the correct type and avoid this unnecessary and potentially expensive copy:

```
void testAutoForLoop()
{
    std::map<int, User> users{/*...*/};
    for (const auto& idUserPair : users)
    {
        // auto is deduced as std::pair<const int, User>.
    }
}
```

Declaring variables of implementation-defined or compiler-synthesized types

ler-synthesized-types

Using **auto** is the only way to declare variables of implementation-defined or compiler-synthesized types, such as lambda expressions (see Section 2.1."Lambdas" on page 393). While in some cases using type erasure to avoid the need to spell out the type is possible, doing so typically comes with additional overhead. For example, storing a lambda closure in a `std::function` might entail an allocation on construction and virtual dispatch upon every call:

std::function

```
void testCallbacks()
{
    std::function<void()> errorCallback0 = [&]{ saveCurrentWork(); };
        // OK, implicit conversion from anonymous closure type to
        // std::function<void()>, which incurs additional overhead

    auto errorCallback1 = [&]{ saveCurrentWork(); };
        // Better, deduces the compiler-synthesized type
}
```

Declaring variables of complex and deeply nested types

d-deeply-nested-types

auto can be used to declare variables of types that are impractical to spell and/or do not convey useful information to the reader. A typical example is avoiding the need for spelling out the iterator type of a container:

std::vector

```
void doWork(const std::vector<int>& data)
{
    // without auto:
    for (std::vector<int>::const_iterator it = data.begin();
         it != data.end();
         ++it) {
        // ...
    }

    // with auto:
    for (auto it = data.begin(); it != data.end(); ++it) { /*...*/ }
}
```

Furthermore, the need for such types can arise, for example, when storing intermediate results of **expression templates** whose types can be deeply nested and unreadable and might even differ between versions of the same library:

```
// without auto:
TransformRange<
    FilterRange<decltype(employees), JoinedInYear>,
    &Employee::name
> newEmployeeNames1 =
    employees | filter(JoinedInYear(2019))
              | transform(&Employee::name);

// with auto:
auto newEmployeeNames2 =
    employees | filter(JoinedInYear(2019))
              | transform(&Employee::name);
```

Improving resilience to library code changes

to-library-code-changes

auto may be used to indicate that code using the variable doesn't rely on a specific type but rather on certain requirements that the type must satisfy. Such an approach might give library implementers more freedom to change return types without affecting the semantics of their clients' code in projects where automated large-scale refactoring tools are not available (but see *Potential Pitfalls — Lack of interface restrictions* on page 193). As an example, consider the following library function:

```
    std::vector

std::vector<Node> getNetworkNodes();
    // Return a sequence of nodes in the current network.
```

As long as the return value of the `getNetworkNodes` function is only used for iteration, it is not pertinent that a `std::vector` is returned. If clients use **auto** to initialize variables storing the return value of this function, the implementers of `getNetworkNodes` can migrate from `std::vector` to, for example, `std::deque` without breaking their clients' code.

```
// without auto:
void testConcreteContainer()
```

```
{
    const std::vector<Node> nodes = getNetworkNodes();
    for (const Node& node : nodes) { /*...*/ }
        // prevents migration
}

void testDeducedContainer()
{
    // with auto:
    const auto nodes = getNetworkNodes();
    for (const Node& node : nodes) { /*...*/ }
        // The return type of getNetworkNodes can be silently
        // changed while retaining correctness of the user code.
}
```

tential-pitfalls-auto

Potential Pitfalls

mpromised-readability

Compromised readability

Using **auto** might sometimes hide important semantic information contained in a variable's type, which could increase the cognitive load for readers. In conjunction with unclear variable naming, disproportionate use of **auto** can make code difficult to read and maintain.

std::vectorstd::string

```
int main(int argc, char** argv)
{
    const auto args0 = parseArgs(argc, argv);
        // The behavior of parseArgs and operations available on args0 is unclear.

    const std::vector<std::string> args1 = parseArgs(argc, argv);
        // It is clear what parseArgs does and what can be done with args1.
}
```

Although reading the contract of the `parseArgs` function at least once may be necessary to fully understand its behavior, using an explicit type's name at the call site helps readers understand its purpose.

unintentional-copies

Unintentional copies

Since the rules for function template type deduction apply to **auto**, appropriate cv-qualifiers and declarator modifiers (`&`, `&&`, `*`, etc.) must be applied to avoid unnecessary copies that might negatively affect both code performance and correctness. For example, consider a function that capitalizes a user's name:

std::stringstd::toupper

```
void capitalizeName(User& user)
{
    if (user.name().empty())
    {
        return;
```

```
    }

    user.name()[0] = std::toupper(user.name()[0]);
}
```

This function was then incorrectly refactored to avoid repetition of the `user.name()` invocation. However, a missing reference qualification leads not only to an unnecessary copy of the string, but also to the function failing to perform its job:

```
void capitalizeName(User& user)
{
    auto name = user.name();  // Bug, unintended copy

    if (name.empty())
    {
        return;
    }

    name[0] = std::toupper(name[0]);  // Bug, changes the copy
}
```

Furthermore, even a fully cv-ref-qualified **`auto`** might still prove inadequate in cases as simple as introducing a variable for a returned-temporary value. As an example, consider refactoring the contents of this simple function:

```
void testExpression()
{
    useValue(getValue());
}
```

For debugging or readablity, it can help to use an intermediate variable to store the results of `getValue()`:

```
void testRefactoredExpression()
{
    auto&& tempValue = getValue();
    useValue(tempValue);
}
```

The above invocation of `useValue` is not equivalent to the original expression; the semantics of the program might have changed because `tempValue` is an *lvalue* expression. To get close to the original semantics, `std::forward` and **`decltype`** must be used to propagate the original value category of `getValue()` to the invocation of `useValue` (see Section 2.1."Forwarding References" on page 351):

```
#include <utility>  // std::forward
void testBetterRefactoredExpression()
{
    auto&& tempValue = getValue();
    useValue(std::forward<decltype(tempValue)>(tempValue));
}
```

Note that, even with the latest changes, the code above achieves the same result but in a somewhat different way because `std::forward<decltype(tempValue)>(tempValue)` is an *xvalue* expression whereas `getValue()` is a *prvalue* expression; see Section 2.1."*rvalue* References" on page 479.

ack-of-expected-ones)

Unexpected conversions (and lack of expected ones)

Compulsively declaring variables using **auto**, even in cases where the desired type has to be spelled out in the initializer, allows explicit conversions to be used where they would not be applicable otherwise. For an example of unintentional consequences of allowing such conversions, consider a function template that is intended to combine two `chrono` duration values:

```
#include <chrono>  // std::chrono::seconds
template <typename Duration1, typename Duration2>
std::chrono::seconds combine_durations(Duration1 d1, Duration2 d2)
{
    auto d = std::chrono::seconds{d1 + d2};
    // ...
}
```

This function template will be successfully instantiated and compiled even when its arguments are two **int**s. Were **auto** not used in this situation — i.e., were `d` declared as `std::chrono::seconds d = d1 + d2;` — the code would fail to compile because the conversion from **int** to `seconds` is explicit, which would indicate a likely defect in the caller's code.

In fairness, a better solution for this particular problem would be to declare `combined_durations` in a more restricted manner:

```
template <typename R1, typename P1, typename R2, typename P2>
std::chrono::seconds combine_durations(std::chrono::duration<R1, P1> d1,
                                       std::chrono::duration<R2, P2> d2)
{
    auto d = std::chrono::seconds{d1 + d2};
    // ...
}
```

The definition above obviates any possible risks related to the use of **auto** because calls with undesirable argument types are not accepted in the first place.

Conversely, some conversions that would be expected to happen might be missed when using **auto** instead of an explicitly specified type. For example, **auto** might deduce a proxy type that might lead to difficult-to-diagnose defects:

```
    std::vector

void testProxyDeduction()
{
    std::vector<bool> flags = loadFlags();

    auto firstFlag = flags[0];  // deduces a proxy type, not bool
    flags.clear();
```

```
    if (firstFlag)  // Bug, use-after-free: flags vector released its memory.
    {
        // ...
    }
}
```

-interface-restrictions

Lack of interface restrictions

Lack of any restrictions placed by **auto** on the type that is deduced might result in defects that could otherwise be detected at compile time. Consider refactoring the `getNetworkNodes` function illustrated in *Use Cases — Improving resilience to library code changes* on page 189 to return `std::deque<Node>` instead of `std::vector<Node>`:

```
    std::deque
std::deque<Node> getNetworkNodes();  // Return type changed from std::vector<Node>.
    // Return a sequence of nodes in the current network.
```

While code that uses **auto** to store the result returned by `getNetworkNodes` only to subsequently iterate over it with a range-based **for** wouldn't be affected, the behavior of code that relied on the contiguous layout of elements in `std::vector` objects *silently* becomes undefined:

```
void testUseContiguousMemory()
{
    auto nodes = getNetworkNodes();
    CLibraryProcessNodes(&nodes[0], nodes.size());
        // exhibits UB after std::vector-to-std::deque change
}
```

While specifying constraints on types deduced by **auto** with **static_assert** is possible, doing so is often cumbersome[3]:

```
const Packet* PacketCache::findFirstCorruptPacket() const
{
    auto it = std::begin(this->d_packets);

    static_assert(
        std::is_base_of<
            std::random_access_iterator_tag,
            std::iterator_traits<decltype(it)>::iterator_category>::value,
        "'it' must satisfy the requirements of a random access iterator.");

    // ...

    return it == std::end(this->d_packets) ? nullptr : &*it;
}
```

[3] C++20 introduced **concepts** — named type requirements — as well as means to constrain **auto** with a specific concept, which can be used instead of **static_assert** in such circumstances.

-of-fundamental-types

Obscuration of important properties of fundamental types

Using **auto** for variables of fundamental types might hide important, context-sensitive considerations, such as overflow behavior or a mix of signed and unsigned arithmetic. In the example below, the `lowercaseEncode` function will either work correctly or enter an infinite loop depending on whether the type returned by `Encoder::encodedLengthFor` is signed.

```
    std::stringstd::tolower
```

```
void lowercaseEncode(std::string* result, const std::string& input)
{
    auto encodedLength = Encoder::encodedLengthFor(input);

    result->resize(encodedLength);
    Encoder::encode(result->begin(), input);

    while (--encodedLength >= 0)  // infinite loop if encodedLength is unsigned
    {
        (*result)[encodedLength] = std::tolower((*result)[encodedLength]);
    }
}
```

r-list-initialization

Surprising deduction for list initialization

auto type-deduction rules differ from those of function templates if brace-enclosed initializer list are used. Function template argument deduction will always fail, whereas, according to C++11 rules, `std::initializer_list` will be deduced for **auto**.

```
auto example0 = 0; // copy initialization, deduced as int
auto example1(0);  // direct initialization, deduced as int
auto example2{0};  // list initialization, deduced as std::initializer_list<int>

template <typename T> void func(T);
void testFunctionDeduction()
{
    func(0);    // T deduced as int
    func({0});  // Error
}
```

This surprising behavior was, however, widely regarded as a mistake.[4]

Nonetheless, even with this retroactive fix, the effects of the deduction rules when applied to braced initializer lists might be puzzling. In particular, `std::initializer_list` is deduced when **copy initialization** is used instead of **direct initialization**, and this requires including `<initializer_list>`:

```
auto x1 = 1;                // int
auto x2(1);                 //  "
auto x3{1};                 //  "
```

[4]This erroneous behavior was formally rectified in C++17 with, e.g., **auto** `i0` deducing **int**. Furthermore, mainstream compilers had applied this deduction-rule change retroactively as early as GCC 5.1 and Clang 3.8, with the revised rule being applied even if `std=c++11` flag is explicitly supplied.

```
#include <initializer_list>  // std::initializer_list
auto x4 = {1};               // OK, deduced as std::initializer_list<int>

auto x5{1, 2};               // Error, direct-list-init requires exactly 1 element.
auto x6 = {1, 2};            // OK, deduced as std::initializer_list<int>
```

Deducing built-in arrays is problematic

n-arrays-is-problematic

Deducing built-in array types using **auto** presents multiple challenges. First, declaring an array of **auto** is ill formed:

```
auto arr1[]  = {1, 2};  // Error, array of auto is not allowed.
auto arr2[2] = {1, 2};  // Error, array of auto is not allowed.
```

Second, if the array bound is not specified, either the program does not compile or `std::initializer_list` is deduced instead of a built-in array:

```
#include <initializer_list>  // std::initializer_list
auto arr3 = {1, 2};  // OK, deduced as std::initializer_list<int>
auto arr4{1, 2};     // Error, direct-list-init requires exactly 1 element.
```

Finally, attempting to circumvent this deficiency by using an alias template (see Section 1.1."**using** Aliases" on page 121) will result in code that compiles but has undefined behavior:

```
    std::size_t

template <typename TYPE, std::size_t SIZE>
using BuiltInArray = TYPE[SIZE];

auto arr5 = BuiltInArray<int, 2>{1, 2};
    // Error, taking the address of a temporary array
```

Note that in this case such code also almost entirely defeats the purpose of **auto** since neither the array element's type nor the array's bound is deduced.

With that said, using **auto** to deduce references to built-in arrays is straightforward:

```
int data[] = {1, 2};

      auto&  arr6 = data;                        //       int (&) [2]
const auto&  arr7 = BuiltInArray<int, 2>{1, 2}; // const int (&) [2]
      auto&& arr8 = BuiltInArray<int, 2>{1, 2}; //       int (&&)[2]
```

Note that the `arr7` and `arr8` references in the code snippet immediately above extend the lifetime of the temporary arrays that they bind to, so subscripting them does not have the undefined behavior that subscripting `arr5` (in the previous code snippet) has.

Annoyances

annoyances-auto

`auto` not allowed for nonstatic data members

non-static-data-members

Despite C++11 allowing nonstatic data members to be initialized within class definitions, **auto** cannot be used to declare them:

```
class C
{
    auto d_i = 1;  // Error, nonstatic data member is declared with auto.
};
```

Not all template argument deduction constructs are allowed for `auto`

-are-allowed-for-auto

Despite **auto** type deduction largely following the template argument deduction rules, certain constructs that are allowed for templates are not allowed for **auto**. For example, when deducing a pointer-to-data-member type, templates allow for deducing both the data member type and the class type, whereas **auto** can deduce only the former:

```
struct Node
{
    int   d_data;
    Node* d_next;
};

template <typename TYPE>
void deduceMemberTypeFn(TYPE Node::*);

void testDeduceMemberType()
{
                deduceMemberTypeFn   (&Node::d_data);  // OK, int Node::*
    auto Node::* deduceMemberTypeVar = &Node::d_data;   // OK,  "     "
}

template <typename TYPE>
void deduceClassTypeFn(int TYPE::*);

void test1DeduceClassType()
{
               deduceClassTypeFn   (&Node::d_data);  // OK, int Node::*
    int auto::* deduceClassTypeVar = &Node::d_data;   // Error, not allowed
}

template <typename TYPE>
void deduceBothTypesFn(TYPE* TYPE::*);

void testDeduceBothTypes()
{
                 deduceBothTypesFn   (&Node::d_next);  // OK, Node* Node::*
    auto* auto::* deduceBothTypesVar = &Node::d_next;   // Error, not allowed
}
```

Furthermore, deducing the parameter of a class template is also not allowed:

```
    std::vector

std::vector<int> vectorOfInt;
```

```
template <typename TYPE>
void deduceVectorArgFn(const std::vector<TYPE>&);

void testDeduceVectorArg()
{
                       deduceVectorArgFn   (vectorOfInt); // OK, TYPE is int
    std::vector<auto> deduceVectorArgVar = vectorOfInt;  // Error, not allowed
}
```

Instead, if **auto** type deduction is desired in such cases, **auto** alone is suitable to deduce the type from the initializer:

```
auto deduceClassTypeVar = &Node::d_data;  // OK, int Node::*
auto deduceBothTypesVar = &Node::d_next;  // OK, Node* Node::*

auto deduceVectorArgVar = vectorOfInt;    // OK, std::vector<int>
```

see-also

See Also

- "Trailing Return" (§1.1, p. 112) ♦ explains how the **auto** placeholder can be used to specify a function's return type at the end of its signature.
- "*Generic* Lambdas" (§2.2, p. 605) ♦ illustrates how the **auto** placeholder can be used in the argument list of a lambda to make its function call operator a template.
- "Deduced Return Type" (§3.2, p. 687) ♦ describes how the **auto** placeholder can be used to deduce a function's return type.

Further Reading

- "Item 1: Understand template type deduction," pp. xx–yy; "Item 2: Understand auto type deduction," pp. xx–yy; "Item 5: Prefer auto to explicit type declarations," pp. xx–yy; and "Item 6: Use the explicitly typed initializer idiom when auto deduces undesired types," pp. xx-yy, **?**

Braced-Initialization Syntax: {}

bracedinit

Braced initialization, a generalization of C++03 initialization syntax, was designed with the intention designed to be used safely and uniformly in any initialization context.

Description

description

List initialization, originally dubbed **uniform initialization**, was conceived to enable a uniform syntax (having the same meaning) to be used generically to initialize objects irrespective of (1) the context in which the syntax is used or (2) the type of the object being initialized. Braced-initialization syntax is the language mechanism that — in close collaboration with the C++ Standard Library's `std::initializer_list` template (see Section 2.1."`initializer_list`" on page 392) — is used to implement list initialization generally. As we will see, this design goal was largely achieved albeit with some idiosyncrasies and rough edges.

C++03 initialization syntax review

ization-syntax-review

Classic C++ affords several forms of initialization, each sporting its own custom syntax, some of which is syntactically interchangeable yet belying subtle differences. At the highest level, there are two dual categories of initialization: (1) **copy/direct** (when you have something from which to initialize) and (2) **default/value** (when you don't).

The first dual category of syntactic/semantic initialization comprises ***copy* initialization** and ***direct* initialization**. *Direct* initialization is produced when initializing an object with one or more arguments within parentheses, such as initializing a data member or base class in a constructor's initializer list, or in a **`new`** expression. *Copy* initialization happens when initializing from a value without using parentheses, such as passing an argument to a function, or returning a value from a function. Both forms may be used to initialize a variable:

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

int i = 23;  // copy initialization
int j(23);   // direct initialization

assert(i == j);
```

In both cases above, we are initializing the variable (`i` or `j`) with the literal value `23`.

For scalar types, there is no observable difference between these two dual forms of initialization in C++03, but, for user-defined types, there are. First *direct* initialization *considers* (as part of the overload set) all valid user-defined conversion sequences, whereas *copy* initialization excludes explicit conversion:

```
struct S
{
    explicit S(int);       //     explicit value constructor (from int)
             S(double);    // non-explicit value constructor (from double)
             S(const S&);  // non-explicit copy  constructor
```

```
};

S s1(1);     // direct init of s1: calls S(int)   ; copy constructor is not called
S s2(1.0);   // direct init of s2: calls S(double);   "      "        "   "     "

S s3 = 1;    // copy init of s3:   calls S(double); copy constructor may be called
S s4 = 1.0;  // copy init of s4:   calls S(double);   "      "         "    "   "
```

What's more, *copy* initialization is defined as if a temporary object is constructed; the compiler is permitted to elide this temporary and, in practice, typically does. Note, however, that *copy* initialization is not permitted unless there is an accessible *copy* (or *move*[1]) constructor, even if the temporary would have been elided.[2] Note that function arguments and return values are initialized using *copy* initialization.

Reference types are also initialized by copy and direct initialization, binding the declared reference to an object (or function). For a reference to a non**const** qualified type, the referenced type must match exactly. However, if binding a reference to a **const** qualified type, the compiler may copy initialize a temporary object of the target type of the reference and bind the reference to that temporary; in such cases, the lifetime of the temporary object is extended to the end of the lifetime of the reference.

```
void ref_inits()
{
    int i = 0;          // OK, copy initialization of int
    int& x(i);          // OK, direct initialization of reference
    const long& y = x;  // OK, y binds to a temporary and extends lifetime
    long& z = x;        // Error, incompatible types
}
```

The second dual category of syntactic/semantic initialization comprises ***default* initialization** and ***value* initialization**. Both default and value initialization pertain to situations in which *no* argument is supplied, and are distinguished by the presence or absence of parentheses, where the absence of parentheses indicates default initialization and the presence indicates value initialization. Note that in simple contexts such as declaring a variable, empty parentheses may also indicate a function declaration instead (see *Use Cases — Avoiding the most vexing parse* on page 220):

```
int i;          // default initialization
int j();        // Oops, function declaration
int k = int();  // value initialization

int *pd = new int;    // default initialization of dynamic int object
int *pv = new int();  // value        "         "     "      "    "
```

For **scalar types**, *default* initialization does not actually initialize an object, and *value* initialization will initialize that object as if by the literal `0`. Note that the representation of

[1] If the *move* constructor for a user-defined type is declared and not accessible, copy initialization is ill formed; see Section 2.1."*rvalue* References" on page 479 and Section 1.1."**??**" on page **??**.

[2] In C++17, direct materialization replaces *copy* initialization in some contexts, thereby obviating the temporary object construction and also the need for an accessible *copy* or *move* constructor.

this value is not necessarily all zero bits, as some platforms use distinct trap values for the null pointer value for pointers and for pointer-to-member objects.

For class types with an accessible **user-provided** default constructor, default initialization and value initialization behave identically, calling the default constructor. If there is no accessible default constructor, both forms produce a compilation error. For objects of class types with an implicitly defined default constructor, each base and member subobject will be default initialized or value initialized according to the form of initialization indicated for the complete object; if any of those initializations produces an error, then the program is ill formed. Note that for a union with an implicitly defined default constructor, the first member of the union will be value initialized as the active member of that union when a union object is value initialized.

```
struct B
{
    int i;
    B() : i() { }  // user-provided default constructor
};

struct C
{
    int i;
    C() { }  // user-provided default constructor
};

struct D : B { int j; };  // derived class with no user-provided constructors

int *pdi = new int;    // default initialization of dynamic int object, *pdi is uninitialized
int *pvi = new int();  // value   initialization of dynamic int object, *pvi is 0
B *pdb = new B;        // default initialization of dynamic B object, b::i is 0
B *pvb = new B();      // value   initialization of dynamic B object, b::i is 0
C *pda = new C;        // default initialization of dynamic C object, a::i is uninitialized
C *pva = new C();      // value   initialization of dynamic C object, a::i is uninitialized
D *pdc = new D;        // default initialization of dynamic D object, c::j is uninitialized
D *pvc = new D();      // value   initialization of dynamic D object, c::j is 0
```

In the case of an object of type `B`, both default and value initialization will invoke the user-provided default constructor, which initializes the subobject `i` to `0`. In the case of an object of type `C`, both default and value initialization will invoke the user-provided default constructor, which does not initialize the subobject `i`. In the case of an object of type `D`, which has an implicitly defined default constructor, the initialization of the subobject `j` depends on whether the `D` object is initialized by default initialization or by value initialization.

Any attempt to read the value of an *uninitialized* object will result in **undefined behavior**. It is a diagnosable error to default initialize a constant object that does not execute a user-provided constructor to initialize each base and member. Note that the top level object, like `D` above, need not have a user-provided constructor as long as all of its bases and members can recursively apply this rule.

```
struct D2 : B { B j; };  // derived class with no user-provided constructors
```

```
const D  w;         // Error, w.j is not initialized.
const D  x = D();   // OK, x is value initialized.
const D2 y;         // OK, y is default initialized; sub-objects invoke default ctor.
const D2 z = D2();  // OK, z is value initialized.
```

Objects of static storage duration at file, namespace, or function scope (see Section 1.1."Function **`static`** '11" on page 57) are ***zero* initialized** before any other initialization takes place. Note that ***default* initialized** static storage duration pointer objects are *zero* initialized to have a *null* address value (see Section 1.1."**`nullptr`**" on page 87), even if that representation on the host platform is not numerically zero:

```
struct A
{
    int i;
};

struct B
{
    int i;
    B() i(42) { }
};

A globalA;
    // Zero initialization also zero initializes globalA::i.
    // Default initialization provides no further initializations.

B globalB;
    // zero initialization initializes globalB.i.
    // After that, default constructor is invoked.

int globalI;
    // Zero initialization initializes globalI.
```

Note the implication that default initialization for a static storage duration object will always initialize an object ready for use, either calling the default constructor of a type with a user-provided default constructor or zero-initializing scalars.

table-bracedinit-cpp3rules

Table 1: Helpful summary of C++03 rules

Initialization Type	No Arguments	>= 1 Arguments
With Parentheses	*value* `int i = int();`	*direct* `int i(23);`
Without Parentheses	*default* `int i;`	*copy* `int i = 23;`

regate-initialization

C++03 aggregate initialization

Aggregates are a special kind of object in C++03 that generally do not use constructors but follow a different set of rules for initialization, typically denoted by braces. There are two varieties of aggregates: (1) arrays and (2) user-defined class types that have no non-public data members that are not static, no base classes, no user-declared constructors, and no virtual functions. Aggregates are very similar to a classic C **`struct`**, potentially with additional, non-**`virtual`** member functions. Note that members of an aggregate are not themselves required to be an aggregate.

```
int a[5];           // Arrays are aggregates.

struct A
{
    int         i;  // public data member
    std::string s;  // A is an aggregate even though std::string is not.

private:
    static int j;   // Private data member is static.
    void f();       // Member functions are OK, even if private.
};
```

A quick note on terminology: Strictly speaking, arrays comprise *elements* and classes comprise *members*, but for ease of exposition in this text, we refer to both as *members*.

When an aggregate is copied by either direct or copy initialization, rather than calling the copy constructors, the corresponding members (elements for an array) of each aggregate are copied using direct initialization, which corresponds to the behavior of an implicitly defined copy constructor for a class. Note that this process may be applied recursively, if members are aggregates themselves. Further note that in most cases, arrays do not copy because the argument supplied to the copy operation will undergo **array-to-pointer decay** and so will no longer be an appropriate type to initialize from. However, arrays as data members of classes follow the rules for aggregate initialization and so will copy array data members. This array-copy behavior is one of the motivations for the addition of the `std::array` template in C++11.

When an aggregate is *default* initialized, each of its members/elements is *default* initialized. When an aggregate is *value* initialized, each of its members/elements is *value* initialized. This follows the usual rules for an implicitly defined constructor for a class type and defines the corresponding behavior for array initialization.

```
int *pid = new int[N];   // default initialization of dynamic array object and its elements
int *piv = new int[N](); // value        "         "     "      "     "     "   "
"


A *pd = new A;           // default initialization of dynamic A object and its members
A *pv = new A();         // value        "         "     "      "     "     "   "
"
```

Otherwise, an aggregate must be *aggregate* initialized by a braced list in the form `= { list-of-values };`, where members of the aggregate will be initialized by *copy* initialization from the corresponding value in the list of values; if the aggregate has more members

than are provided by the list, the remaining members are *value* initialized; it is an error to provide more values in the list than there are members in the aggregate. Note that, because a union has only one active member, a union will be initialized by no more than a single value from the list; this becomes relevant for unions as data members of an aggregate initialized by *brace elision*:

```
union U
{
    int i;
    const char* s;
};

U x = {    };  // OK, value initializes x.i = 0
U y = { 1  };  // OK, initializes x.i = 1
U z = { "" };  // Error, cannot aggregate initialize z.s
```

Let's review the various ways in which we might attempt to initialize an object of aggregate type `A` in the body of a function, `test`, i.e., defined at function scope:

```
struct A2 { int i };  // aggregate with a single data member

void test()
{
           A2  a1;             // default init: i is not initialized!
    const  A2& a2 = A();       // value init followed by copy init: i is 0.
           A2  a3 = A();       // value init followed by copy init: i is 0.
           A2  a4();           // Oops, function declaration!
           A2  a5 = { 5 };     // aggregate initialization employing copy init
           A2  a6 = { };       //    "           "           "     value  "
           A2  a7 = { 5, 6 };  // Error, too many initializers for aggregate A
    static A2  a8;             // default init after i is zero initialized.
}
```

In the sample code above:

- `a1`: `a1` is *default* initialized, which means that each data member within the aggregate is itself independently *default* initialized. For scalar types, such as an **`int`**, the effect of default initialization at function scope is a no-op — i.e., `a1.i` is not initialized. Any attempt to access the contents of `a1.i` will result in undefined behavior.

- `a2` and `a3`: In the cases of both `a2` and `a3`, a temporary of type `A` is first ***value* initialized** and then that temporary is used to ***copy* initialize** the named variable: Both `a2.i` and `a3.i` are initialized to the value `0`.

- `a4`: Notice that we are not able to create a *value* initialized local variable `a4` by applying parentheses since that would interpreted as declaring a function taking no arguments and returning an object of type `A` by value; see *Use Cases* — **??** on page **??** [**AUs: there is no section with this name "Avoiding accidentally declaring a function taking no arguments"**] and *Use Cases* — *Avoiding the most vexing parse* on page 220.

- `a5`, `a6`, and `a7`: C++03 supports **aggregate initialization** using braced syntax as illustrated by `a5`, `a6`, and `a7` in the code snippet above. The local variable `a5` is *copy* initialized such that `a5.i` has the *user supplied* value `5` whereas `a6` is *value* initialized since there are no supplied initializers; hence, `a6.i` is initialized to 0. Attempting to pass `a7` two values to initialize a single data member results in a compile-time error. Note that had class `A` held a second data member, the line initializing `a5` would have resulted in *copy* initialization of the first and *value* initialization of the second.

- `a8` has static storage duration therefore it is first *zero* initialized (`a8.i` is 0) then it is *default* initialized, which is a no-op for the same reasons that `a1` is not initialized at all.

Finally, note that a scalar can be thought of as though it were an array of a single element (though note that scalars never suffer *array-to-pointer decay*); in fact, if we were to take the address of any scalar and add `1` to it, the new pointer value would represent the one-past-the-end iterator for that scalar's implied array (of length `1`). Similarly, scalars can be initialized using aggregate initialization, just as if they were single-element arrays, where the braced list for a scalar may contain zero or one elements.

```
int    i = { };       // OK, i is 0.
int    j = { 1 };     // OK, i is 1.
double k = { 3.14 };  // OK, k is 3.14.
```

itialization-in-c++11

Braced initialization in C++11

Everything we've discussed so far, including braced initialization of aggregates, is well defined in C++03. This same braced initialization syntax — modified slightly so as to preclude narrowing conversions (see the next section) — is extended in C++11 to work consistently and uniformly in many new situations. This enhanced braced initialization syntax is designed to better support the two dual initialization categories discussed in **C++03 initialization syntax review**above as well as entirely new capabilities including language-level support for lists of initial values implemented using the C++ Standard Library's `std::initializer_list` class template. As the opportunity arose by touching the rules for initialization, a few more potential errors, such as narrowing conversions, become diagnosable by the compiler when using braced initialization syntax.

narrowing-conversions

C++11 restrictions on narrowing conversions

Narrowing conversions (a.k.a. **lossy conversions**) are a notorious source of runtime errors. One of the important properties of list initializations implemented using the C++11 braced-initialization syntax is that error-prone narrowing conversions are no longer permitted. Consider, for example, an **`int`** array, `a`, initialized with various built-in (compile-time constant) *literal* values:

```
int ai[] =  //      C++03    C++11
{
    5,      // (0)   OK       OK
    5.0,    // (1)   OK      Error, double to int conversion is not permitted.
```

```
    5.5,    // (2)   OK      Error, double to int conversion is not permitted.
    "5",    // (3)  Error    Error, no const char* to int conversion exists.
};
```

In C++03, floating-point literals would be coerced to fit within an integer even if the conversion was known to be lossy — e.g., element (2) in the code snippet above. By contrast, C++11 disallows *any* such implicit conversions in braced initializations even when the conversion is known *not* to be lossy — e.g., element `ai[1]` above.

Narrowing conversions within the integral and floating-point type families, respectively, are generally disallowed except where it can be verified at compile-time that overflow does not occur and, in the case of integers and (classic) **`enum`**s , the initializer value can be represented exactly[3]:

```
const unsigned long ulc = 1;  // compile-time integral constant: 1UL

short as[] =  //       C++03     C++11                        Stored Value
{
     32767,   // (0)     OK        OK                         as[0] == 32767
     32768,   // (1)     OK      Error, overflow
    -32768,   // (2)     OK        OK                         as[0] == -32768
    -32769,   // (3)   Warning?  Error, underflow
    1UL,      // (4)     OK        OK                         as[0] == 1
    ulc,      // (5)     OK        OK                         as[0] == 1
    1.0       // (6)     OK      Error, narrowing
};
```

Notice that both *overflow* (1) and *underflow* (3) are rejected for integral values in C++11, whereas neither is ill formed in C++03. An integral literal (4) (or an **integral constant** (5)) of a wider type (e.g., **`unsigned long`**) can be used to initialize a smaller one (e.g., **`signed short`**) provided that the value can be represented exactly; however, even a floating point literal that can be *represented exactly* (6) is nonetheless rejected in C++11 when used to initialize any integral scalar.

Floating-point initializers, on the other hand, need not be represented precisely so long as overflow does not occur; if, however, what is being initialized is a floating-point scalar and the initializer is integral, then the value must be represented exactly:

```
float af[] =     //       C++03   C++11                  Stored Value
{
     3L,        // (0)     OK     OK                   af[0] == 3
     16777216,  // (1)     OK     OK                   af[1] == 1<<24
     16777217,  // (2)     OK     Error, lossy
     0.75,      // (3)     OK     OK                   af[1] == 0.75
     2.4,       // (4)     OK     OK, but lossy        af[2] != 2.4
     0.4,       // (5)     OK     OK, but lossy        af[3] != 0.4
     1e-39,     // (6)     OK     OK, but underflow    af[4] != 1e-39
     1e+39,     // (7)     OK     Error, overflow
};
```

[3] As of C++20, implicit conversion from either a pointer or pointer-to-member type to **`bool`** is generally supported in braced initializations.

In the example above elements (0) – (2) represent initialization from an integral type (**int**), which requires that the initialized value be represented exactly. Elements (3) – (7) are instead initialized from a floating-point type (**double**) and therefore are restricted only from overflow.

When an initializer is *not* a **constant expression**, braced initialization precludes any possibility of such *narrowing* initializations at run time — e.g., initializing a **float** with a double or a **long double**, a **double** with a **long double**, or *any* floating-point type with an integral one. By the same token, an integral type (e.g., **int**) is not permitted to be initialized by non-constant expression integer value of any other potentially larger integral type (e.g., **long**) — even if the number of bits in the representation for the two types on the current platform is the same. Finally, non-constant expression of integral type (e.g., **short**) cannot be used to initialize an unsigned version of the same type (e.g., **unsigned short**) and vice versa.

To illustrate the constraints imposed on non-constant-expressions described above, consider a simple aggregate class, `S`, comprising an **int**, `i`, and a **double**, `d`:

```
struct S  // aggregated class
{
    int    i;  // *integral* scalar type
    double j;  // *floating-point* scalar type
};
```

A function, `test`, declaring a variety of arithmetic parameter types illustrates restrictions imposed by C++11 braced initialization on narrowing initializations that were well formed in C++03:

```
void test(short s, int i, long j, unsigned u, float f, double d, long double e)
{                       //     C++03  C++11
    S s0 = { i, d };  // (0)  OK     OK
    S s1 = { s, f };  // (1)  OK     OK
    S s2 = { u, d };  // (2)  OK    Error, u causes narrowing.
    S s3 = { i, e };  // (3)  OK    Error, e causes narrowing.
    S s4 = { f, d };  // (4)  OK    Error, f causes narrowing.
    S s5 = { i, s };  // (5)  OK    Error, s causes narrowing (theoretically).
};
```

In the `test` function above, lines (0) and (1) are OK because there is no possibility of narrowing on any conforming platform unlike lines (2) through (5) — despite the fact that, in practice, it is more than likely that a **double** will be able to represent exactly every value representable by a **short int**. Note that, just as with the array example above, when the initializing value is a constant expression, it is sufficient that that value be representable exactly in the target type and produce the original value when converted back.

C++11 Aggregate initialization

regate-initialization

Aggregate initialization in C++11, including initialization of arrays, is subject to the rules prohibiting narrowing conversions.

```
int  i   = { 1 };        // OK
long j   = { 2 };        // OK
```

```
int  a[] = { 0, 1, 2 };  // OK
int  b[] = { 0, i, j };  // Error, cannot narrow j from long to int

struct S { int a; };
S s1 = { 0  };  // OK
S s2 = { i  };  // OK
S s3 = { 0L };  // OK, 0L is an integer constant expression.
S s4 = { j  };  // Error, narrowing
```

In addition, the rules for value initialization now state that members without a specific initializer value in the braced list are "as-if" copy initialized from `{}` [4]. This will result in an error when initializing a member that has an **`explicit`** default constructor according to the new **copy *list* initialization** rules in the next section, which give a meaning for explicit constructors. Note that if the member is of reference type and no initializer is provided, the initialization is ill formed.

Regardless of whether the aggregate itself is initialized using a copy initialization or direct initialization, the members of the aggregate will be copy initialized from the corresponding initializer.

```
struct E { };                          // empty type
struct AE { int x; E y; E z; };        // aggregate comprising several empty objects
struct S { explicit S(int = 0) {} };   // class with explicit default constructor
struct AS{ int x; S y; S z; };         // aggregate comprising several S objects

AE aed;
AE ae0 = {};               // OK
AE ae1 = { 0 };            // OK
AE ae2 = { 0, {} };        // OK
AE ae3 = { 0, {}, {} };    //

AS asd;                    //
AS as0 = {};               // OK in 03; error in 11 calling explicit ctor for S
AS as1 = { 0 };            // OK in 03; error in 11 calling explicit ctor for S
AS as2 = { 0, S() };       // OK in 03; error in 11 calling explicit ctor for S
AS as3 = { 0, S(), S() };  // OK, all aggregate members have an initializer.
```

To better support generalizing the syntax of brace initialization in a style similar to aggregate initialization, an aggregate can make a copy of itself through *aggregate* initialization in C++11 as well as through *direct* initialization per C++03:

```
S x{};      // OK, value initialization
S y = {x};  // OK in C++11; copy initialization via aggregate initialization syntax
```

Otherwise, initialization of aggregates in C++11 is exactly the same where it would have a meaning in C++03 and is correspondingly extended into new places where braced initialization is permitted, as documented in the following subsections.

[4] From C++ 14 onwards, if the member doesn't have an initializer value, but has a default member initializer, it is initialized from the default member initializer (see Section 2.1."Aggregate Init '14" on page 127).

y-list-initialization

Copy list initialization

For C++03, only aggregates and scalars could be initialized via braced-initialization syntax:

```
Type var = { /*...*/ };  // C++03-style aggregate (only) initialization
```

The first part of generalizing braced initialization syntax for C++11 is to allow the same syntactic form used to initialize aggregates to be used for *all* user-defined types. This extended form of braced initialization — known as copy *list* initialization — follows the rules of copy initialization:

```
Class var1 = val;      // (C++03) copy initialization
Class var2 = { val };  // (C++11) copy list initialization
```

For a non-aggregate class type, C++11 allows the use of a braced list provided that its sequence of values serves as a suitable argument to a non-**explicit** constructor of the class being initialized. Importantly, this use of *copy* list initialization provides meaning to **explicit** constructors when taking other than a single argument. For example, consider a **struct** S having constructors with 0-3 parameters, only that last of which is **explicit**:

```
struct S
{
            S();                          // default cor
            S(int};                       // 1-value ctor
            S(int, const char*);          // 2-value ctor
    explicit S(int, const char*, double); // 3-value ctor
};
```

We can use *copy* list initialization only if the selected constructor is *not* declared to be **explicit**, e.g., s0, s1, and s2 but not s3:

```
S s0 = { };                // OK, copy list initialization
S s1 = { 1 };              // OK, copy list initialization
S s2 = { 1, "two" };       // OK, copy list initialization
S s3 = { 1, "two", 3.14 }; // Error, constructor is explicit
```

Had we instead declared our default constructor or any of the others to be **explicit**, the corresponding *copy* (or *copy-list*) initialization above would have failed too.

Another important difference between C++11 copy *list* initialization and C++03 copy initialization is that the braced-list syntax considers all constructors, including those that are declared to be **explicit**. Consider a **struct** X having two overloaded single-argument constructors, i.e., (1) one taking an **int** and (2) the other a member template taking a single (deduced) type, T, by value:

```
struct Q  // class containing both explicit and implicit constructor overloads
{
    explicit Q(int);          // (1) value constructor taking a int
    template <class T> Q(T);  // (2) value constructor taking a T
};
```

Employing *direct* initialization (e.g., x0 in the code snippet below) selects the most appropriate constructor, regardless of whether it is declared to be **explicit**, and successfully uses that one; employing *copy* initialization (e.g., x1) drops explicit constructors from the

overload set before determining a best match; and employing copy *list* initialization (e.g., `x2`) again includes all constructors in the overloads set but is **ill formed** if the selected constructor is **`explicit`**:

```
Q x0(0);     // OK, direct initialization calls Q(int).
Q x1 = 1;    // OK, copy initialization calls Q(T).
Q x2 = {2}; // Error, copy list initialization selects but cannot call Q(int).
Q x3{3};     // Same idea as x0; direct list initialization calls Q(int).
```

In other words, the presence of the `=` coupled with the braced notation (e.g., `x2` in the code example above) forces the compiler to choose the constructor *as if* it were direct initialization (e.g., `x0`) but then forces a compilation failure if the selected constructor turns out to be **`explicit`**. This "consider-but-fail-if-selected" behavior of copy list initialization is analogous to that of functions declared using **`= delete`**; see Section 1.1."**??**" on page **??**. Using braces but omitting the `=` (e.g., `x3`) puts us back in the realm of *direct* rather than *copy* initialization; see *Direct list initialization* on page 210.

When initializing references, copy list initialization (braced syntax) behaves similarly to copy initialization (no braces) with respect to the generation of temporaries. For example, when using a braced list to initialize an *lvalue* reference — e.g., **`int&`** `ri` or **`const int&`** `cri` in the code example below — to a scalar of a type that exactly matches it (e.g., **`int`** `i`), no temporary is created (just as it would not have been without the braces); otherwise, a temporary will be created, provided that a viable conversion exists and is not narrowing:

```
void test()
{
    int  i = 2;               assert(i   == 2);
    int& ri = { i };          assert(ri  == 2);  // OK, no temporary created
    ri = 3;                   assert(i   == 3);  // original is affected

    const int& cri = { i };   assert(cri == 3);  // OK, no temporary created
    ri = 4;                   assert(cri == 4);  // other reference is affected

    short s = 5;              assert(s   == 5);
    const int& crs = { s };   assert(crs == 5);  // OK, temporary is created
    s = 6;                    assert(crs == 5);  // note temporary is unchanged

    long j = 7;               assert(j   == 7);
    const int& crj = { j };   // Error, narrowing conversion from long to int
}
```

As evidenced by the C-style asserts above, no temporary is created when initializing either `ri` or `cri` since modifying the reference affects the underlying variable, and vice versa. The C++ type of `crs`, on the other hand, does *not* match exactly that of the type to which it is bound, a temporary *is* created and hence changing the underlying object does *not* affect its referenced value. Lastly, unlike `s` (of type **`short`**), attempting to initialize a **`const`** *lvalue* reference of type **`int`**, `crj`, with `j` (of type **`long`**) is a narrowing conversion and thus ill formed.

Another consideration involves the standard **`typedef`** `std::size_t` found in the standard header `<cstddef>`, which must have sufficient bits to represent the unsigned difference

between any two pointers (into contiguous memory) and is typically, but not necessarily, an alias for an **unsigned long**:

```
std::size_t k = 8;  // alias to implementation-defined type, say, unsigned long

unsigned int&       ra1 = { k };  // Note: Only one of these three lines will
unsigned long&      ra2 = { k };  // compile on any given platform; the other two
unsigned long long& ra3 = { k };  // will necessarily be ill formed and not compile.
```

Historically, a **long** has always been of sufficient size to **pun** a pointer value (BAD IDEA) yet, back in the day when **int** and **long** were the same number of bytes on a given platform, `size_t` was often an alias to an **unsigned int** rather than an **unsigned long**. Moving forward, one might expect `size_t` to be an alias for an **unsigned long** on a 64-bit platform, but there is no such assurance in the C++ Standard and **unsigned long long** (see Section 1.1."**long long**" on page 78) is another viable option on any standards-conforming platform.

Finally, **const** *lvalue* references to scalars and aggregates initialized via braced lists of literal values follow the rules of aggregates (see *C++11 Aggregate initialization* on page 206); a temporary is materialized having the indicated value and bound permanently to the reference with its lifetime extended coterminously:

```
const int& i0 = { };    // OK, materialized temporary is value initialized.
const int& i1 = { 5 };  // OK,       "           "       " copy      "
```

In the example above, a temporary is *value* initialized (to 0) and bound to `i0`; another temporary is then *copy* initialized (to 5) and bound to `i1`.

Non-modifiable references to *aggregate UDT*s exploit the generalization of *copy* and *direct* list initialization. Consider an aggregate `A` that comprises three **int** data members, `i`, `j`, and `k`:

```
struct A          // struct A is an aggregate data type.
{
    int i, j, k;  // This struct contains three data members of type int.
};
```

We can now use braced initialization to materialize a temporary object of aggregate type `A` using aggregate initialization.

```
const A& s0 = { };          // i, j, and k are  value initialized.
const A& s1 = { 1 };        // i copy and j and k are value initialized.
const A& s2 = { 1, 2 };     // i and j copy and k are value initialized.
const A& s3 = { 1, 2, 3 };  // i,j, and k are copy initialized.
```

In the example above, each of the references, `s0` thru `s3`, is initialized to a temporary **struct** of type `A` holding the respective aggregate value `{0,0,0}`, `{1,0,0}`, `{1,2,0}`, and `{1,2,3}`.

t-list-initialization

Direct list initialization

Of the two dual forms of initialization, *direct* versus *copy*, *direct* initialization is the stronger since it enables use of all accessible constructors, i.e., including those declared to be **explicit**.

The next step in generalizing the use of braced initialization is to allow use of a braced list without the intervening `=` character between the variable and the opening brace to denote direct initialization too:

```
Class var(/*...*/);  // C++03-style direct initialization
Class var{/*...*/};  // C++11-style direct list initialization
```

Note that C++ does *not* similarly relax the rules to allow for initialization of aggregates with parentheses.[5]

The syntax suggested in the previous example is known as **direct list initialization** and follows the rules of direct initialization rather than copy initialization in that all constructors of the named `type` are both considered and accessible in the initial overload set:

```
struct Q  // class containing explicit constructor
{
    explicit Q(int);  // value constructor taking a int
    // ...
};

Q x(5);    // OK  direct initialization can call explicit constructors.
Q y{5};    // OK, direct list initialization can call explicit constructors.
Q z = {5}; // Error, copy list initialization can't call explicit constructors.
```

Either form of direct initialization (shown for `x` and `y` in the code example above) may invoke an **`explicit`** constructor of class `Q`, whereas *copy list* initialization will necessarily result in a compile-time error.

However, following the rules of C++11 braced initialization, narrowing conversions are rejected by direct *list* initialization:

```
long a = 3L;

Q b(a);  // OK, direct initialization
Q c{a};  // Error, direct list initialization cannot use a narrowing conversion.
```

Similarly, **`explicit`** conversion operators (see Section 1.1."**`explicit`** Operators" on page 50) can be considered when *direct* list initialization (or direct initialization) is employed on a scalar, but not so with *copy* list initialization (or copy initialization). Consider, for example, a class, `W` that can covert to either an **`int`** or a **`long`** where conversion to **`int`** is explicit and therefore must be *direct*:

```
struct W
{
    explicit operator int()  const;  // used via direct initialization only
             operator long() const;  // used via direct or copy initialization
};
```

Initializing a **`long`** variable with an expression of type `W` can be accomplished via either *direct* or *copy* initialization (e.g., `jDirect` and `jCopy`, respectively, in the code snippet below), but initializing an **`int`** variable with such an expression can be accomplished only via *direct* initialization (e.g., `iDirect`):

[5] C++20 finally allows for aggregates to be initialized with parentheses.

```
long jDirect {W()};  // OK, considers both operators, calls operator long
long jCopy = {W()};  // OK, considers implicit op only, calls operator long
int  iDirect {W()};  // OK, considers both operators, calls operator int
int  iCopy = {W()};  // Error, considers implicit op only, narrowing conversion
```

In the example above, attempting to use *copy* list initialization (e.g., `iCopy`) forces the conversion to **`long`** as the only option, which results in a narrowing conversion and an ill-formed program.

Note that, for **aggregate** types, even direct list initialization will not allow the explicit constructors of the individual member types to be considered since such *member-wise* initialization is invariably *copy* initialization; see *C++11 Aggregate initialization* on page 206.

We may use direct list initialization as part of **member initialization lists** for base classes and member data of a class (note that there is no equivalent to allow copy list initialization in such a context). Consider, for example, an aggregate class, `B`, a non-aggregate class, `C`, and a derived class, `D`, that inherits from `B` and has an object of type `C` as a data member, `m`:

```
struct B { int i; };        // aggregate base type
struct C { C(); C(int); };  // non-aggregate member type

struct D : B  // class publicly derived from B containing C
{
    C m;  // non-aggregate data member

    D()      : B{},  m{}  { }  // direct initialized base/member objects
    D(int x) : B{x}, m{x} { }  //    "       "        "     "       "
};
```

In the definition of class `D` above, both constructors employ direct list initialization; the first is also an example of *value* initialization for both (aggregate) class `B` and (non-aggregate) class `C`. Note that in C++03, aggregate bases and members could only be default initialized, value initialized, or direct initialized, and not initialized to another value.

New expressions are another context in which direct list initialization (braces) or direct initialization (parentheses) can occur and applies similarly to both aggregate and nonaggregate types. If no initializer is provided, the allocated object is default initialized; if empty braces or parentheses are supplied, the object is value initialized; otherwise, the object is initialized from the contents of the braced or (where permitted) parenthesized list.

As an illustrative example, let's consider the scalar type **`int`** which itself can be default initialized (not initialized), value initialized (to 0) via empty braces or parentheses, or **direct initialized** via a single element within either parentheses or braces:

```
int* s0  = new int;              // default initialized (no initializer)

int* t0 = new int();             // direct (value) initialized from ()
int* t1 = new int{};             // direct (value) list initialized from {}

int* u0 = new int(7);            // direct initialized from 7
int* u1 = new int{7};            // direct list initialized from {7}
```

```
int* v0 = new int[5];             // All 5 elements are default initialized.

int* w0 = new int[5]();           // All 5 elements are value initialized.
int* w1 = new int[5]{};           // Array is direct list initialized from {}.

int* x0 = new int[5](9);          // Error, invalid initializer for an array
int* x1 = new int[5]{9};          // Array is direct list initialized from {9}.

int* y1 = new int[5]{1,2,3};      // array direct list initialized from {1,2,3}

int* z1 = new int[5]{1,2,3,4,5}; // direct list initialized from {1,2,3,4,5}
```

All the comments above apply to the object being created in the **new** expression; the pointer set to the address of the dynamically allocated object is copy initialized in all cases. Note that, in C++03, we could default initialize (e.g., `v0`) or value initialize (e.g., `w0`) the elements of an array in a **new** expression but there was no way to initialize the elements of such an array to anything other than their default value (e.g., `x0`); as of C++11, direct list initialization with braces (e.g., `x1`, `y1`, `z1`) makes this more flexible, heterogeneous initialization of array elements in **new** expressions possible.

ect-list-initialization

Contrasting copy and direct list initialization

The difference between *copy* list initialization and *direct* list initialization can be seen in this example:

```
struct C
{
    explicit C() { }
    explicit C(int) { }
};

struct A  // aggregate of C
{
    C x;
    C y;
};

int main()
{
    C c1;           // OK, default initialization
    C c2{};         // OK, value initialization
    C c3{1};        // OK, direct list initialization
    C c4 = {};      // Error, copy list initialization cannot use explicit default ctor.
    C c5 = {1};     // Error, copy list initialization cannot use explicit ctor.

    C c6[5];        // OK, default initialization
    C c7[5]{};      // Error, aggregate initialization requires a non-explicit default ctor.
    C c8[5]{1};     // Error, aggregate initialization requires non-explicit ctors.
    C c9[5] = {};   // Error, aggregate initialization requires a non-explicit default ctor.
    C ca[5] = {1};  // Error, aggregate initialization requires non-explicit ctors.
```

```
    A a1;           // OK, default initialization
    A a2{};         // Error, aggregate initialization requires a non-explicit default ctor.
    A a3{1};        // Error, aggregate initialization requires non-explicit ctors.
    A a4 = {};      // Error, aggregate initialization requires a non-explicit default ctor.
    A a5 = {1};     // Error, aggregate initialization requires non-explicit ctors.
}
```

Note that if the constructors for `C` were not marked explicit, then all of the variables in the example above would be safely initialized. If only the **`int`** constructor of `C` were explicit, then the initializations that did not depend on the **`int`** constructor would be valid:

```
struct C
{
    C() { }
    explicit C(int) { }
};

struct A  // aggregate of C
{
    C x;
    C y;
};

int main()
{
    C c1;           // OK, default initialization
    C c2{};         // OK, value initialization
    C c3{1};        // OK, direct list initialization
    C c4 = {};      // OK, copy list initialization
    C c5 = {1};     // Error, copy list initialization cannot use explicit ctor.

    C c6[5];        // OK, default initialization
    C c7[5]{};      // OK, value initialization
    C c8[5]{1};     // Error, aggregate initialization requires non-explicit ctors.
    C c9[5] = {};   // OK, copy list initialization
    C ca[5] = {1};  // Error, aggregate initialization requires non-explicit ctors.

    A a1;           // OK, default initialization
    A a2{};         // OK, value initialization
    A a3{1};        // Error, aggregate initialization requires non-explicit ctors.
    A a4 = {};      // OK, copy list initialization
    A a5 = {1};     // Error, aggregate initialization requires non-explicit ctors.
}
```

Integrating default member initialization with braced initialization

braced-initialization

Another new feature for C++11 is *default member initializers* for data members in a class. This new syntax supports both copy list initialization and value list initialization. However, initialization with parentheses is not permitted in this context.

```
struct S
{
    int i = { 13 };

    S() { }                         // OK, i == 13.
    explicit S(int x) : i(x) { }  // OK, i == x.
};


struct W
{
    S a{};        // OK, by default j.i == 13.
    S b{42};      // OK, by default j.i == 42.
    S c = {42};   // Error, constructor for S is explicit.
    S d = S{42};  // OK, direct initialization of temporary for initializer
    S e(42);      // Error, fails to parse as a function declaration
    S f();        // OK, declares member function f
};
```

gument-to-a-constructor

List initialization where the list itself is a single argument to a constructor

Another new form of initialization for C++11 is list initialization with a braced list of arguments to populate a container. See Section 2.1."`initializer_list`" on page 392 for details. If a braced list of arguments are all of the same type, then the compiler will look for a constructor taking an `std::initializer_list<T>` argument, where `T` is that common type. Similarly, if a braced list of values can be implicitly converted to a common type, then a constructor for an `std::initializer_list` of that common type will be preferred. When initializing from a nonempty braced initializer list, a matching initializer list constructor always wins overload resolution. However, *value* initializing from a pair of empty braces will prefer a default constructor.

```
struct S
{
  S() {}
  S(std::initializer_list<int>) {}
  S(int, int);
};

S a;             // default initialization with default constructor
S b();           // function declaration!
S c{};           // value initialization with default constructor
S d = {};        // copy list initialization with std::initializer_list
S e{1,2,3,4,5}; // direct list initialization with std::initializer_list
S f{1,2};        // direct list initialization with std::initializer_list
S g = {1,2};     // copy list initialization with std::initializer_list
S h(1,2);        // direct initialization with two ints
```

tialized-a-temporary

Omitting the type name when braced initializing a temporary

In addition to supporting new forms of initialization, C++11 allows for braced lists to implicitly construct an object where the type is known by context, such as for function arguments and return values. This use of a braced list without explicitly specifying a type is just like constructing a temporary object by copy initialization, hence using copy list initialization, for the implicit type. As such construction is copy list initialization, it will reject explicit constructors:

```
struct S
{
    S(int, int) {}
    explicit S(const char*, const char*) {}
};

S foo(bool b)
{
    if (b)
    {
        return S{ "hello", "world" };  // OK, direct list initialization of temporary
        return { "hello", "world" };   // Error, copy list initialization cannot call explicit ctor.
    }
    else
    {
        return {0, 0};  // OK, int constructor is not explicit.
    }
}

void bar(S s) { }

int main()
{
   bar( S{0,0} );  // OK, direct list initialization, then copy initialization
   bar(  {0,0} );  // OK, copy list initialization
   bar( S{"Hello", "world"} );  // OK, direct list initialization
   bar(  {"Hello", "world"} );  // Error, copy list initialization cannot use explicit ctor.
}
```

nditional-expressions

Initializing variables in conditional expressions

As a final tweak to make initialization consistent across the language, initializing a variable in the condition of a **while** or **if** statement in C++03 supported only copy initialization and required use of the = token. For C++11, those rules are relaxed to allow any valid form of braced initialization. Conversely, the declaration of a control variable in a **for** loop has supported all forms of initialization permitted for a variable declaration since the original C++ standard[6]:

```
void f()
```

[6]Note that GCC would traditionally accept the C++11-only syntaxes, even when using C++98/03.

```
{
   for (int i = 0; ; ) {}    // OK in all versions of C++
   for (int i = {0}; ; ) {}  // OK for aggregates in C++03 and all types from C++11
   for (int i{0}; ; ) {}     // OK from C++11, direct list initialization
   for (int i{}; ; ) {}      // OK from C++11, value initialization
   for (int i(0); ; ) {}     // OK in all versions of C++
   for (int i(); ; ) {}      // OK in all versions of C++
   for (int i; ; ) {}        // OK in all versions of C++

   if (int i = 0) {}    // OK in all versions of C++
   if (int i = {0}) {}  // OK from C++11, copy list initialization
   if (int i{0}) {}     // OK from C++11, direct list initialization
   if (int i{}) {}      // OK from C++11, value initialization
   if (int i(0)) {}     // Error in all versions of C++
   if (int i()) {}      // Error in all versions of C++
   if (int i) {}        // Error in all versions of C++

   while (int i = 0) {}    // OK in all versions of C++
   while (int i = {0}) {}  // OK from C++11, copy list initialization
   while (int i{0}) {}     // OK from C++11, direct list initialization
   while (int i{}) {}      // OK from C++11, value initialization
   while (int i(0)) {}     // Error in all versions of C++
   while (int i()) {}      // Error in all versions of C++
   while (int i) {}        // Error in all versions of C++
}
```

alization-and-=-default

Default initialization and `= default`

Another new feature for C++11 is the notion of defaulted constructors, defined by **`= default`** (see Section 1.1."Defaulted Functions" on page 30) to have the same definition as the implicitly defined constructor. This is especially useful when you want to provide constructors for your class without losing the **triviality** otherwise associated with the implicit constructors, notably for the default constructor that would no longer be declared as soon as another constructor is declared.

One important property of **`= default`** constructors is that, while they are user *declared*, they are not user *provided* as long as this definition occurs within the class definition itself:

```
struct Trivial
{
    int i;
    Trivial() = default;            // user declared but not user provided
};

struct NonTrivial
{
    int j;
    NonTrivial();                    // user declared
};
```

```
NonTrivial::NonTrivial() = default;  // user provided
```

Following the rules derived from C++03, a class with no user-provided default constructor may behave differently under *default* initialization compared to *value* initialization:

```
void demo()
{
    const Trivial    a;    // Error, a.i not initialized
    const Trivial    b{};  // OK, b.i = 0.
    const NonTrivial c;    // OK, but c.j never initialized
    const NonTrivial d{};  // OK, but d.j never initialized
}
```

Note that we use function local variables for this example to avoid confusion with *zero* initialization for global variables.

Copy initialization and scalars

alization-and-scalars

With the addition of explicit conversion operators to C++11 (see Section 1.1."**explicit** Operators" on page 50), it becomes possible for *copy* initialization and *copy list* initialization to fail for scalars and similarly for *direct list* initialization:

```
struct S
{
    explicit operator int() const { return 1; }
};

S one{};

int a(one);     // OK, a = 1.
int b{one};     // OK, b = 1.
int c = {one};  // Error, copy list initialization used with
                // with explicit conversion operator.
int d = one;    // Error, copy initialization used with
                // with explicit conversion operator.

class C {
    int x;
    int y;

public:
    C(const S& value) : x(value)  // OK, x = 1
                      , y{value}  // OK, y = 1
    {
    }
};
```

use-cases-bracedinit

ue-initialized-variable

Use Cases

Defining a value-initialized variable

The C++ parser has a pitfall where an attempt to value initialize a variable turns out to be a function declaration:

```
struct S{};

void foo()
{
    S s1();      // Oops! function declaration
    S s2 = S();  // variable declaration using value initialization and then copy initialization
}
```

The declaration of `s1` looks like an attempt to value initialize a local variable of type `S`, but, in fact, it is a forward declaration for a function `s1` that takes no arguments and returns an `S` object by value. This is particularly surprising for folks who did not realize we could declare (but not define) a function within the body of another function, a feature retained from the original C Standard. Clearly, there would be an ambiguity in the grammar at this point unless the language provided a rule to resolve the ambiguity, and the grammar opts in favor of the function declaration in all circumstances, including at function local scope. Whilst this rule would be essential at namespace/class scope, otherwise functions taking no arguments could not be so easily declared, the same rule also applies at function local scope, first for consistency and second for compatibility with pre-existing C code that we might want to compile more strictly[7] with a C++ compiler.

By switching from parentheses to braces, there is no more risk of confusion between a vexing parse and a variable declaration:

```
S s{};  // object of type S
```

As a digression, it is worth noting that `S` is an empty type. Just as it is ill-formed to rely on default initialization for a **`const`** object of a trivial type such as an **`int`** or an aggregate of just an **`int`**, it was also, prior to Defect Report CWG 253 for C++17 and the introduction of **const-default-constructible** types, ill-formed to rely on default initialization for a **`const`** object of an empty type. Hence, it is often desirable to value initialize such objects, running into the vexing parse. As this Defect Report was resolved at the end of 2016, and applies retroactively to earlier dialects, most current compilers no longer enforce this restriction, and some compilers (notably GCC) had already stopped enforcing this rule several years previously:

```
const S cs1;        // Error on some compilers (as described above)
const S cs2{};      // OK, value initialization
const S cs3 = {};   // OK, aggregate initialization
const S cs4 = S();  // OK, copy initialization
```

[7] While C and C++ both enforce type safety, C++ enforces strict type safety, where all declared types are distinct; C enforces structural conformance, where two distinct structs with the same sequence of members are treated as the same type.

Avoiding the most vexing parse

the-most-vexing-parse

Value initializing function arguments can lead to another pitfall, often called *the most vexing parse*. C++ will parse the intended value initialization of a function argument as the declaration of an unnamed parameter of a function type instead, which would otherwise not be legal but for the language rule that such a parameter implicitly decays to a pointer to a function of that type:

```
struct V { V(const S&) { } };

void foo()
{
    V v1(S());    // most vexing parse, declares function v1 taking a function pointer
    V v2((S()));  // workaround, object of type V due to non-redundant parentheses on argument

    S x = S();    // declare a variable of type S
    V v3(x);      // workaround, object of type V but argument is now named, with longer lifetime
}
```

In the example above, `v1` is the forward declaration of a function in the surrounding namespace that returns an object of type `V` and has an (unnamed) parameter of type pointer-to-function-returning-`S`-and-taking-no-arguments, `S(*)()`. That is, the declaration is equivalent to:

```
V v1(S(*)());
```

This most vexing of parses can be disambiguated by having the arguments clearly form an expression, not a type. One simple way to force the argument to be parsed as an expression is to add an otherwise redundant pair of parentheses. Note that declaring the constructor of `V` as **`explicit`** in the hopes of forcing a compile error is no help here since the declaration of `v1` is not interpreted as a declaration of an object of type `V`, so the **`explicit`** constructor is never considered.

With the addition of generalized braced initialization in C++11, a coding convention to prefer empty braces, rather than parentheses, for all value initializations avoids the question of the most vexing parse arising:

```
V v4(S{});  // direct initialize object of type V with value initialized temporary
```

Note that the most vexing parse can also apply to constructors taking multiple arguments, but the issue arises less often since any one supplied argument clearly being an expression, rather than a function type, resolves the whole parse:

```
struct W { W(const S&, const S&) { } };

void foo() {
  W w1( S(),  S());  // most vexing parse, declares function w1 taking two function pointers
  W w2((S()), S());  // workaround, object of type V due to non-redundant parentheses on argument
  W w3( S{},  S());  // workaround, even a single use of S{} disambiguates further use of S()
}
```

ization-in-generic-code

Uniform initialization in generic code

One of the design concerns facing an author of generic code is which form of syntax to choose to initialize objects of a type dependent on template parameters. Different C++ types behave differently and accept different syntaxes, so providing a single consistent syntax for all cases is not possible. Consider the following example of a simple test harness for a unit testing framework:

```
#include <initializer_list>

template <class T, class U>
bool run_test(bool (*test)(const T&), std::initializer_list<U> iL)
{
    for (const auto& val : iL)
    {
        T obj = val;      // initialize the test value
        if (!test(obj))
        {
            return false;
        }
    }

    return true;
}
```

In this example, a test function is provided for an object of parameter type `T` along with an `initializer_list` of test values. The **`for`** loop will construct a test object with each test value, in turn, and call the test function, returning early if any value fails. The question is which syntax to use to create the test object `obj`.

- As written, the example uses *copy* initialization — `T obj = val;` — and so will fail to compile if a non-**`explicit`** constructor cannot be found or if `T` is an aggregate that is not `U`.
- If we switched to *direct* initialization — `T obj(val);` — then explicit constructors would also be considered.
- If we switched to *direct list* initialization — `T obj{val};` — then aggregates would be supported as well as explicit constructors but not narrowing conversions; `initializer_list` constructors are also considered and preferred.
- If we switched to *copy list* initialization — `T obj = {val};` — then aggregates would be supported, but it would be an error if an explicit constructor is the best match, rather than considering the non-**`explicit`** constructors for the best viable match, and it would be an error to rely on a narrowing conversion; `initializer_list` constructors are also considered and preferred.

Table 2 summarizes the different initialization types and highlights the options and trade-offs. In general, there is no one true, universal syntax for initialization in generic (template) code. The library author should make an intentional choice among the trade-offs described in this section and document that as part of their contract.

Table 2: Summary of the different initialization types

-bracedinit-inittypes

Initialization Type	Syntax	Aggregate Support	Explicit Constructor Used	Narrowing	`initializer_list` Constructor Used
Copy	`T obj = val;`	only if `U`	no	allow	no
Direct	`T obj(val);`	only if `U`	yes	allow	no
Direct List	`T obj{val};`	yes	yes	error	yes
Copy List	`T obj = {val}`	yes	error if best match	error	yes

Uniform initialization in factory functions

-in-factory-functions

One of the design concerns facing an author of generic code is which form of syntax to choose to initialize objects of a type dependent on template parameters. Different C++ types behave differently and accept different syntaxes, so providing a single consistent syntax for all cases is not possible. Here we present the different trade-offs to consider when writing a factory function that takes an arbitrary set of parameters to create an object of a user-specified type:

```
template <class T, class... ARGS>
T factory1(ARGS&&... args)
{
    return T(std::forward<ARGS>(args)...);
}

template <class T, class... ARGS>
T factory2(ARGS&&... args)
{
    return T{std::forward<ARGS>(args)...};
}

template <class T, class... ARGS>
T factory3(ARGS&&... args)
{
    return {std::forward<ARGS>(args)...};
}
```

All three factory functions are defined using **perfect forwarding** (see Section 2.1."Forwarding References" on page 351) but support different subsets of C++ types and may interpret their arguments differently.

`function1` returns a value created by direct initialization but, because it uses parentheses, cannot return an aggregate unless (as a special case) the `args` list is empty or contains exactly one argument of the same type `T`; otherwise, the attempt to construct the return

value will parse as an error.[8]

`function2` returns an object created by direct list initialization. Hence, `function2` supports the same types as `function1`, plus aggregates. However, due to the use of braced initialization, `function2` will reject any types in `ARGS` that require narrowing conversion when passed to the constructor (or to initialize the aggregate member) of the return value. Also, if the supplied arguments can be converted into a homogeneous `std::initializer_list` that matches a constructor for `T`, then that constructor will be selected, rather than the constructor best matching that list of arguments, despite `function2` being called using parentheses (as for any function call).

`function3` behaves the same as `function2`, except that it uses *copy list initialization* so will also produce a compile error if the selected constructor for the return value is declared as **`explicit`**.

There is no one true form of initialization that works best in all circumstances for such a factory function, and it is for library developers to choose (and document in their contract) the form that best suits their needs. Note that the Standard Library runs into this same problem when implementing factory functions like `std::make_shared` or the `emplace` function of any container. The Standard Library consistently chooses parentheses initialization like `function1` in the code example above, and so these functions do not work for aggregates prior to C++20.

Uniform member initialization in generic code

ization-in-generic-code

With the addition of general braced initialization to C++11, class authors should consider whether constructors should use *direct* initialization or *direct list* initialization to initialize their bases and members. Note that as copy initialization and copy list initialization are not options, whether or not the constructor for a given base or member is **`explicit`** will never be a concern.

Prior to C++11, writing code that initialized aggregate subobjects (including arrays) with a set of data in the constructor's member initializer list was not really possible. We could only *default* initialize, *value* initialize, or *direct* initialize from another aggregate of the same type.

Starting with C++11, we are able to initialize aggregate members with a list of values, using aggregate initialization in place of direct list initialization for members that are aggregates:

```
struct S
{
    int         i;
    std::string str;
};

class C
{
    int j;
    int a[3];
```

[8] Note that C++20 will allow aggregates to be initialized with parentheses as well as with braces, which will result in this form being accepted for aggregates as well.

```
    S   s;

public:
    C(int x, int y, int z, int n, const std::string t)
    : j(0)
    , a{ x, y, z }  // Ill-formed in C++03, OK in C++11
    , s{ n, t }     // Ill-formed in C++03, OK in C++11
    {
    }
};
```

Note that as the initializer for `C.j` shows in the code example above, there is no requirement to consistently use either braces or parentheses for all member initializers.

As with the case of factory functions, the class author must make a choice for constructors between adding support for initializing aggregates vs. reporting errors for narrowing conversion. Since member initialization supports only *direct* list initialization, there is never a concern regarding **`explicit`** conversions in this context:

```
template <class T>
class Wrap
{
    T data;

    template <class... ARGS>
    Wrap(ARGS&&... args)
    : data(std::forward<ARGS>(args))  // must be empty list or copy for aggregate T
    {
    }
};

template <class T>
class WrapAggregate
{
    T data;

    template <class... ARGS>
    WrapAggregate(ARGS&&... args)
    : data{std::forward<ARGS>(args)}  // no narrowing conversions
    {
    }
};
```

Again, there is no universal best answer, and an explicit choice should be made and documented so that consumers of the class know what to expect.

Potential Pitfalls

Inadvertently calling initializer-list constructors

Classes with an `std::initializer_list` constructors (see Section 2.1."`initializer_list`" on page 392) follow some special rules to disambiguate overload resolution, which contain

subtle pitfalls for the unwary. This pitfall describes how overload resolution might (or might not) select those constructors in surprising ways.

When an object is initialized by braced initialization, the compiler will first look to find an `std::initializer_list` constructor that could be called, with the exception that if the braced list is empty, a default constructor (if available) would have priority:

```
#include <initializer_list>

struct S {
  explicit S() {}
  explicit S(int) {}
  S(std::initializer_list<int> iL) { if (0 == iL.size()) {throw 13;} }
};

S a{};            // OK, value initialization
S b = {};         // Error, default constructor is explicit
S c{1};           // OK, std::initializer_list
S d = {1};        // OK, std::initializer_list
S e{1, 2, 3};     // OK, std::initializer_list
S f = {1, 2, 3};  // OK, std::initializer_list
```

In the presence of initializer list constructors, the overload resolution to select which constructor to call will be a two-step process. First, all initializer-list constructors are considered, and only if no matching `std::initializer_list` constructor has been found, non–initializer-list constructors will be considered. This process has some possibly surprising consequences since implicit conversions are allowed when performing the overload matching. It is possible that an `std::initializer_list` constructor requiring an implicit conversion will be selected over a non–initializer-list constructor that does not require a conversion:

```
#include <initializer_list>

struct S
{
    S(std::initializer_list<int>); // #1
    S(int i, char c);              // #2
};

S s1{1, 'a'};  // calls #1, even though #2 would be a better match
```

In the example above, due to braced initialization preferring initializer-list constructors and because `S` has an `initializer_list` constructor that can match the initializer of `s1`, the constructor that would have been a better match otherwise is not considered.

The other possibly surprising consequence is related to narrowing conversion being checked for only *after* the constructor has been selected. This means that an `initializer_list` constructor that matches but requires a narrowing conversion will cause an error even in the presence of a non`initializer_list` constructor that would be a match without requiring a narrowing conversion:

```
#include <initializer_list>
```

```
struct S
{
    S(std::initializer_list<int>); // #1
    S(int i, double d);            // #2
};

S s2{1, 3.2}; // narrowing conversion when attempting to call #1,
              //     even though invoking #2 would be well formed
```

In the example above, due to braced initialization first selecting a constructor and then checking for narrowing conversion, the non-initializer-list constructor, which would not require a narrowing conversion, is not considered.

Both of these situations can be resolved by using parentheses or other forms of initialization than brace lists, which do not prefer initializer-list constructors:

```
#include <initializer_list>

struct S
{
    S(std::initializer_list<int>); // #1
    S(int i, char c);              // #2
    S(int i, double d);            // #3
};

S s3(1, 'c'); // calls #2
S s4(1, 3.2); // calls #3
```

This problem often comes up when talking about `std::vector`:

```
vector<size_t> v1{5u, 13u};  // Possible bug here.
  // If trying to construct a vector of 5 size_t with value 13,
  // The std::initializer_list constructor is preferred over exact match,
  // so we actually construct a vector with 2 values, 5 and 13.

vector<size_t> v2(5u, 13u);  // OK, calls the normal constructor
```

lose-aggregate-status

Classes with default member initializers lose aggregate status

An aggregate class is a class with no user-provided constructors, no base classes, no virtual functions, and all public data members. Braced initialization of an aggregate matches each member of the brace list to the corresponding member of the aggregate class in the order of declaration. However, if any of the members has a Section 2.1."Default Member Init" on page 296, then the class ceases to be an aggregate in C++11, and braced initialization will fail to compile because no matching constructor will be found. C++14 fixes this oversight, so this pitfall should occur only when supporting code across multiple versions of the language, which is typically more of a concern for library maintainers than application developers:

```
struct S
{
    int a;
```

```
    int b;
    int c;
};

struct A  // not an aggregate in C++11 (aggregate in C++14, see next section)
{
    int a{1};
    int b{2};
    int c{3};
};

S s1 = {};         // OK, aggregate initialization, value initializes each member
S s2 = {1, 2, 3};  // OK, aggregate initialization

A a1 = {};         // OK, value initialization with implicit default ctor in C++11
A a2 = {4, 5, 6};  // Error, no matching constructor in C++11
```

Implicit move and named return value optimization may be disabled in `return` statements

ed-in-return-statements

Using extra braces in a return statement around a value may disable the named return value optimization or an implicit move into the returned object.

Named return value optimization (NRVO) is an optimization that compilers are allowed to perform when the operand of a return statement is just the name (id-expression) of a nonvolatile local variable (an object of automatic storage duration that is not a parameter of the function or a catch clause) and the type of that variable, ignoring cv-qualification, is the same as the function return type. In such cases, the compiler is allowed to elide the copy implied by the return expression and initialize the return value directly where the local variable is defined. Naturally this applies only to functions returning objects, not pointers or references. Note that this optimization is allowed to change the meaning of programs that may rely on observable side effects on the elided copy constructor. Most modern compilers are capable of performing this optimization in at least simple circumstances, such as where there is only one return expression for the whole function.

In the example below, we see that the `no_brace()` function returns using the name of a local variable from within that function. As we call `no_brace()` we can observe (with a compiler that performs the optimization) that only one object of the `S` class is created, using its default constructor. There is no copy, no move, and no other object created. Essentially the local variable, `a`, inside the `no_brace()` function is created directly in the memory region of the variable `m1` of the `main()` function.

In the `braced()` function, we use the exact same local variable, but in the return statement we put braces around its name; therefore, the operand of the return is no longer a name (id-expression), and so the rules that allow NRVO do not apply. By calling `braced()`, we see that now two copies, and so two objects, are created, the first being `a`, the local variable, using the default constructor, and the second being `m2`, which is created as a copy of `a`, demonstrating that NRVO is not in effect:

```
#include <iostream>
```

```
struct S
{
    S()          { std::cout << "S()\n"; }
    S(const S &) { std::cout << "S(copy)\n"; }
    S(S &&)      { std::cout << "S(move)\n"; }
};

S no_brace()
{
    S a;
    return a;
}

S braced()
{
    S a;
    return { a };  // disables NRVO
}

int main()
{
    S m1 = no_brace();  // S()
    S m2 = braced();    // S(), S(copy)
}
```

Implicit move (see Section 2.1."*rvalue* References" on page 479) in a return statement is a more subtle operation, so much so that it required a defect report[9] to actually make it work as the original intention. We demonstrate implicit moves in a return statement from a local variable by using two types. The class type `L` will be used for the local variable, whereas the class type `R`, which can be move- or copy-constructed from `L`, is used as the return type. Essentially, we are forcing a type conversion in the return statement, one that may be a copy or a move.

The `no_brace()` function just creates a local variable and returns it. By calling the function, we observe that an `L` object is created, which is then moved into an `R` object. Note that the wording of the ISO standard allows this implicit move only if the return statement's operand is a name (an id-expression).

The `braced()` function is identical to the previous one, except for adding curly braces around the operand of the return statement. Calling the function shows that the *move-from-*`L` return expression turned into a *copy-from-*`L` return expression because a braced initializer is not a name of an object:

```
#include <iostream>

struct L
{
    L()        { std::cout << "L()\n"; }
```

[9] **?**

```
};

struct R
{
    R(const L &) { std::cout << "R(L-copy)\n"; }
    R(L &&)      { std::cout << "R(L-move)\n"; }
};

R no_brace()
{
    L a;
    return a;
}

R braced()
{
    L a;
    return { a };  // disables implicit move from l
}

int main()
{
    R r1 = no_brace();  // L(), R(L-move)
    R r2 = braced();    // L(), R(L-copy)
}
```

ng-deleted-constructors

Surprising behavior of aggregates having deleted constructors

Value initialization of aggregates is allowed with a braced initializer list, even if the default constructor is deleted[10]:

```
struct S
{
    int data;
    S() = delete; // don't want "empty"
};

S s{}; // surprisingly works (until C++20), and 0 == s.data
```

This surprising pitfall occurs for two reasons:

1. A deleted constructor is *user declared* but not *user provided*, so it does not feature in the list of things that stop a class being an aggregate.

2. The rules state that aggregate initialization is not defined in terms of constructors but directly in terms of the initialization of its members.

[10]Note that C++20 finally addresses the issue so the presence of deleted constructors cause a class to no longer qualify as an aggregate.

Annoyances

annoyances

Narrowing aggregate initialization may break C++03 code

-may-break-c++03-code

When compiling existing C++03 code with a C++11 compiler, previously valid code may report errors for narrowing conversion in aggregate (and, therefore, also array) initialization.

```
unsigned u;
u = 128;                    // u is computed to an int-friendly value
int ia[] = { 1, 2, u, 9 };  // OK in C++03, narrowing is allowed.
                            // Error in C++11, narrowing conversion.
```

Suppose that the computation in the above code ensures that the value `u` holds at the point of initialization is in the range of values an **`int`** is able to represent. Yet, the code will not compile in C++11 or later modes. Unfortunately each and every case has to be fixed by applying the appropriate type cast or changing the types involved to be "compatible".

No easy way to allow narrowing conversions

narrowing-conversions

In generic code, curly braces have to be used if support for aggregates is required, but, if our interface definition requires supporting narrowing conversions (for example `std::tuple`), there is no direct way to enable them:

```
struct S
{
    short m;
};

class X
{
    S m;

public:
    template <class U>
    X(const U& a) : m{a}  // no narrowing allowed
    {
    }
};

int i;
X x(i);  // Error, would narrow in initializing S.m
```

The workaround is to **`static_cast`** to the target type if it is known or to use parentheses and give up aggregate support in the generic code.[11]

Breaks macro-invocation syntax

cro-invocation-syntax

The macro-invocation syntax of the C++ preprocessor (inherited from C) "understands" parentheses and thus ignores commas within parentheses but does not understand any other list markers, such as braces for braced initialization, square brackets, or the angle bracket

[11] C++20 enables the use of parentheses to initialize aggregates.

notation of templates. If we attempt to use commas in other contexts, the macro parsing will interpret such commas as separators for multiple macro arguments and will likely complain that the macro does not support that many arguments:

```
#define MACRO(oneArg) /*...*/

struct C
{
    C(int, int, int);
};

struct S
{
    int i1, i2, i3;
};

MACRO(C x(a, b, c))         // OK, commas inside parentheses ignored
MACRO(S y{a, b, c})         // Error, 3 arguments but MACRO needs 1
MACRO(std::map<int, int> z) // Error, 2 arguments but MACRO needs 1
```

As the example above demonstrates, on the first macro invocation, the commas within the parentheses are ignored, and the macro is invoked with one argument: `Demo x(a, b, c)`.

In the second macro invocation, we attempt to use braced initialization, but, because the syntax of the preprocessor does not recognize curly braces as special delimiters, the commas are interpreted as separating macro arguments, so we end up with three unusual arguments: first `Demo y{a`, second `b`, and finally the third `c}`. This problematic interaction between braced initialization and macros has existed forever, even back in C code when initializing arrays or **`struct`**s. However, with braced initialization becoming used more widely in C++, it is much more likely that a programmer will encounter this annoyance.

The third invocation of `MACRO` in the example is just a reminder that the same issue exists in C++ with the angle brackets of templates.

The workaround, as is so often the case with the C preprocessor, is more use of the C preprocessor! We need to define macros to help us hide the commas. Such macros will use the **variadic macros** C99 preprocessor feature that was adopted for C++11 to turn a comma-separated list into a braced-initializer list (and similarly for a template instantiation):

```
#define BRACED(...) { __VA_ARGS__ }
#define TEMPLATE(name, ...) name<__VA_ARGS__>

MACRO(X y BRACED(a, b, c));             // OK, X y { a, b, c }
MACRO(TEMPLATE(std::map, int, int) z);  // OK, std::map<int, int> z
```

A common way this annoyance might show up is using the Standard Library `assert` macro:

```
bool operator==(const C&, const C&);

void f(const C& x, int i, int j, int k)
{
   assert(C(i, j, k) == x);  // OK
   assert(C{i, j, k} == x);  // Error, too many arguments to assert macro
```

```
}
```

not-deduce-array-size

Default member initializer does not deduce array size

Although the syntax looks the same, default member initializers using braced-initializer lists do not deduce the size of an array member:

```
struct S
{
    char s[]{"Idle"};  // Error, must specify array size
};
```

The rationale is that there is no guarantee that the default member initializer will be used to initialize the member; hence, it cannot be a definitive source of information about the size of such a member in the object layout.

per-initializer-lists

No copy list initialization in member initializer lists

The syntax for base and member initializers allows for both direct initialization with parentheses (since C++03) and direct list initialization with braces (since C++11). However, there is no syntax corresponding to copy list initialization, which would allow member initializers to report errors for using an unintended explicit constructor or conversion operator. It would seem relatively intuitive to extend the syntax to support `= { ... }` for member initializers to support such use, but so far there have been no proposals to add this feature to the language. That may be a sign that there is simply no demand, and the authors of this book are the only ones annoyed since this is the only part of the language that supports *direct* initializations without a corresponding syntax for *copy* initializations.

```
class C
{
    explicit C(int);
    C(int, int);
};

class X
{
    C a;
    C b;
    C c;

public:
    X(int i)
    : a(i)       // OK, direct initialization
    : b(i)       // OK, direct list initialization
    : c = (i,i)  // Error, copy list initialization is not allowed.
    {
    }
};
```

ssed-multiple-arguments

Accidental meaning for explicit constructors passed multiple arguments

In C++03, marking a default or multi-argument constructor explicit, typically as a result of supplying default arguments, had no useful meaning, and compilers did not warn about them because they were harmless. However, C++11 takes notice of the **`explicit`** keyword for such constructors when invoked by copy list initialization. This design point is generally not considered when migrating code from C++03 to C++11 and may require programmers to invest more thought, and potentially split constructors with multiple default arguments into multiple constructors, applying **`explicit`** to only the intended overloads:

```
class C
{
public:
    explicit C(int = 0, int = 0, int = 0);
};

C c0 = {};       // Error, constructor is explicit.
C c1 = {1};      // Error, constructor is explicit.
C c2 = {1,2};    // Error, constructor is explicit.
C c3 = {1,2,3};  // Error, constructor is explicit.

class D
{
public:
    D();
    explicit D(int i) : D(i, 0) { }  // delegating constructor
    D(int, int, int = 0);
};

D d0 = {};       // OK
D d1 = {1};      // Error, constructor is explicit.
D d2 = {1,2};    // OK
D d3 = {1,2,3};  // OK

C f(int i, C arg)
{
    switch (i)
    {
        case 0: return {};         // Error, constructor is explicit.
        case 1: return {1};        // Error, constructor is explicit.
        case 2: return {1, 2};     // Error, constructor is explicit.
        case 3: return {1, 2, 3};  // Error, constructor is explicit.
    }
}

D g(int i, D arg)
{
    switch (i)
    {
      case 0: return {};         // OK
```

```
      case 1: return {1};        // Error, constructor is explicit.
      case 2: return {1, 2};     // OK
      case 3: return {1, 2, 3};  // OK
    }
}

void test()
{
    f(0, {});         // Error, constructor is explicit.
    f(0, {1});        // Error, constructor is explicit.
    f(0, {1, 2});     // Error, constructor is explicit.
    f(0, {1, 2, 3});  // Error, constructor is explicit.

    g(0, {});         // OK
    g(0, {1});        // Error, constructor is explicit.
    g(0, {1, 2});     // OK
    g(0, {1, 2, 3});  // OK
}
```

Note that this topic is deemed an annoyance, rather than a pitfall, because it affects only newly written C++11 (or later) code using the new forms of initialization syntax, so it does not break existing C++03 code recompiled with a more modern language dialect. However, also note that many containers and other types in the C++ Standard Library inherited such a design and have not been refactored into multiple constructors (although some such refactoring occurs in later standards).

Obfuscation due to opaque use of braced-list

ue-use-of-braced-list

Use of braced initializers for function arguments, omitting any hint of the expected object type at the call site, requires deep familiarity with functions being called in order to understand the actual types of arguments being initialized, especially when overload resolution must disambiguate several viable candidates. Such usage may produce more fragile code as further overloads are added, silently changing the type initialized by the brace list as a different function wins overload resolution. Such code is also much harder for a subsequent maintainer, or casual code reader, to understand:

```
#include <initializer_list>

struct C
{
    C(int, int) { }
};

int test(C, long) { return 0; }

int main()
{
    int a = test({1, 2}, 3);
    return a;
}
```

This program compiles and runs, returning the intended result. However, consider how the behavior changes if we add a second overload during subsequent maintenance:

```
#include <initializer_list>

struct C
{
    C(int, int) { }
};

int test(C, long) { return 0; }

struct A  // additional aggregate class
{
    int x;
    int y;
};

int test(A, int) { return -1; }  // overload for the aggregate class

int main()
{
    int a = test({1, 2}, 3);     // overload resolution prefers the aggregate
    return a;
}
```

Because the overload for `A` must now be considered, overload resolution may pick a different result. If we are lucky, then the choice of the `A` and `C` overloads becomes ambiguous, and an error is diagnosed. However, in this case, there was an integer promotion on the second argument, and the new `A` overload is now the stronger match, producing a different program result. If this overload is added through maintenance of an included header file, this code will have silently changed meaning without touching the file. If the above flexibility is not the desired intent, the simple way to avoid this risk is to always name the type of any temporary variables:

```
int main()
{
    int a = test(C{1,2}, 3);  // Overload resolution prefers struct C.
    return a;
}
```

d-braced-initialization

`auto` deduction and braced initialization

C++11 introduces type inference, where an object's type is deduced from its initialization, using the **`auto`** keyword (see Section 2.1."**`auto`** Variables" on page 183). When presented with a homogeneous, nonempty list using *copy* list initialization, **`auto`** will deduce the type of the supplied argument list as an `std::initializer_list` of the same type as the list values. When presented with a braced list of a single value using *direct* list initialization, **`auto`** will deduce the variable type as the same type as the list value:

```
#include <initializer_list>

auto g{1};         // OK, deduces g is int
auto h{1, 2, 3};     // Error, auto requires exactly one element in brace list
auto i = {1};      // OK, deduces i is initializer_list<int>
auto j = {1, 2, 3};  // OK, deduces j is initializer_list<int>
```

Note that the declarations of `i` and `j` in the code example above would also be errors if the `<initializer_list>` header had not been included to supply the `std::initializer_list` class template.

Finally, observe that for **`auto`** deduction from *direct* list initialization, an `initializer_list` constructor may still be called in preference to copy constructors, even though the syntax seems restricted to making copies:

```
#include <iostream>
#include <initializer_list>

struct S
{
    S() { }
    S(std::initializer_list<S>) { std::cout << "init list\n";}
    S(const S&) { std::cout << "copy\n";}
};

int main()
{
    S s;
    auto s2{s};  // std::initializer_list<S> constructor is called after
                 // deduction. (Note: s2 is deduced to be of type S.)
}
```

s-accept-braced-lists

Compound assignment but not arithmetic operators accept braced lists

Braced initializers can be used to provide arguments to the assignment operator and additionally to compound assignment operators such as `+=`, where they are treated as calls to the overloaded operator function for class types, or as `+= T{value}` for a scalar type `T`.[12] Note that assigning to scalars supports brace lists of no more than a single element and does not support compound assignment for pointer types, since the brace lists are converted

[12] Although valid, the two `x += {3}` and `x *= {3}` lines in the example compile successfully on Clang but not on GCC or MSVC (applies to all versions at the time of writing). The C++11 standard currently states:

> A braced-init-list may appear on the right-hand side of
>
> — an assignment to a scalar, in which case the initializer list shall have at most a single element. The meaning of `x={v}`, where `T` is the scalar type of the expression `x`, is that of `x=T{v}`. The meaning of `x={}` is `x=T{}`

(**?**, paragraph 9, section 5.17, "Assignment and Compound Assignment Operators," p. 126). There is currently a defect report due to clarify the Standard and explicitly state that this rule also applies to compound assignments (see **?** and **?**).

to a pointer type, which cannot appear on the right-hand side of a compound assignment operator.

While the intent of compounded assignment is to be semantically equivalent to the expression `a = a + b` (or `* b`, or `- b`, and so on), brace lists cannot be used in regular arithmetic expressions since the grammar does not support brace lists as arbitrary expressions:

```
#include <initializer_list>

struct S
{
    S(std::initializer_list<int>) { }
    S& operator+=(const S&) { return *this; }
};

S operator+(const S&, const S&) { return S{}; }

void demo()
{
    S s1{};               // OK, calls initializer_list constructor
    s1 += {1,2,3};        // OK, equivalent to s1.operator+=({1,2,3})
    s1 = s1 + {1, 2, 3};  // Error, expecting an expression, not an std::initializer_list
    s1 = operator+(s1, {1,2,3}); // OK, braces are allowed as function argument.

    int x = 0;
    x += {3};    // OK, equivalent to x += int{3};.[^cwg_1542].
    x *= {5};    // OK, equivalent to x *= int{5};.[^cwg_1542].

    char y[4] = {1, 2, 3, 4};
    char*p = +y;
    p += {3};    // Error,  equivalent to p += (char*){3};
}
```

see-also

See Also

TODO: Add see also items. Pulled from the feature:

- "??" (§1.1, p. ??) ♦
- "**explicit** Operators" (§1.1, p. 50) ♦
- "Function **static** '11" (§1.1, p. 57) ♦
- "**long long**" (§1.1, p. 78) ♦
- "**nullptr**" (§1.1, p. 87) ♦
- "`initializer_list`" (§2.1, p. 392) ♦
- "*rvalue* References" (§2.1, p. 479) ♦
- Implicit move (unknown what this was intended to be)

- "Default Member Init" (§2.1, p. 296) ♦
- perfect forwarding (unknown what this was intended to be)
- "Variadic Templates" (§2.1, p. 519) ♦
- "**auto** Variables" (§2.1, p. 183) ♦

Further Reading

`further-reading`

TBD

Compile-Time Invocable Functions

ime-invocable-constexpr-fcns

Only functions decorated with **constexpr** can be invoked as part of a **constant expression**.

Description

description

A constant expression is an expression whose value can be determined at compile time — i.e., one that could be used, say, to define the size of a C-style array or as the argument to a **static_assert**:

```
enum { e_SIZE = 5 };             // e_SIZE is a constant expression of value 5.
int a[e_SIZE];                   // e_SIZE must be a constant expression.
static_assert(e_SIZE == 5, "");  //    "     "   " "     "          "
```

Prior to C++11, evaluating a conventional function at compile time as part of a constant expression was not possible:

```
inline const int z() { return 5; }  // OK, returns a nonconstant expression
int a[z()];                         // Error, z() is not a constant expression
static_assert(z() == 5, "");        // Error,  "     "  "  "      "         "
int a[0 ? z() : 9];                 // Error,  "     "  "  "      "         "
```

Developers, in need of such functionality, would use other means, such as template metaprogramming, external code generators, preprocessor macros, or hardcoded constants (as shown above), to work around this deficiency.

As an example, consider a **metaprogram** to calculate the nth factorial number:

```
template <int N>
struct Factorial { enum { value = N * Factorial<N-1>::value }; };  // recursive

template <>
struct Factorial<0> { enum { value = 1 }; };                       // base case
```

Evaluating the `Factorial` metafunction above on a constant expression results in a constant expression:

```
static_assert(Factorial<5>::value == 120, "");  // OK, it's a constant expr.
int a[Factorial<5>::value];                     // OK, array of 120 ints
```

Note, however, that the metafunction can be used only with arguments that are themselves constant expressions:

```
const int factorial(const int n)  // returning const int same as just int
{
    static_assert(n >= 0);        // Error, n is not a constant expression.
    return Factorial<n>::value;   // Error,  "   "  "  "     "         "
}
```

Employing this cumbersome work-around leads to code that is difficult both to write and to read and is also non-trivial to compile, often resulting in long compile times. What's more, a separate implementation will be needed for inputs whose values are not compile-time constants.

C++11 introduces a new keyword, **constexpr**, that gives users some sorely needed control over compile-time evaluation. Prepending the declaration[1] of a function with the **constexpr** keyword informs both the compiler and prospective users that the function is eligible for compile-time evaluation and, under the right circumstances, can and will be evaluated *at compile time* to determine the value of a *constant expression*:

```
constexpr int factorial(int n)  // can be evaluated in a constant expression
{
    return n == 0 ? 1 : n * factorial(n - 1);  // single return statement
}
```

In C++11, the body of a **constexpr** function is restricted to a single **return** statement, and any other language construct, such as **if** statements, loops, variable declarations, and so on are forbidden; see *Restrictions on* **constexpr** *function bodies (C++11 only)* on page 250. These seemingly harsh limitations[2], although much preferred to the `Factorial` metafunction (in the code example above), might make optimizing a function's runtime performance infeasible; see *Potential Pitfalls — Prematurely committing to* **constexpr** on page 278. As of C++14, however, many of these restrictions were lifted, though still not all runtime tools are available during compile-time evaluation; see Section 2.2."**constexpr** Functions '14" on page 595.

Simply declaring a function to be **constexpr** does not automatically mean that the function *will* necessarily be evaluated at compile time. A **constexpr** function is *guaranteed* to be evaluated at compile time *only* when invoked in a context where a constant expression is required — a.k.a. a **constexpr context**[3]. This can include, for example, the value of a nontype template parameter, array bounds, the first argument to a **static_assert**, or the initializer for a **constexpr** variable (see Section 2.1."**constexpr** Variables" on page 282). If one attempts to invoke a **constexpr** function in a **constexpr** context with an argument that is not a constant expression, the compiler will report an error:

```
#include <cassert>   // standard C assert macro
#include <iostream>  // std::cout

void f(int n)
{
    assert(factorial(5) == 120);
        // OK, factorial(5) might be evaluated at compile time since 5 is a
        // constant expression but factorial is not used in a
        // constexpr context.
```

[1] Note that semantic validation of **constexpr** functions occurs only at the point of *definition*. It is therefore possible to *declare* a member or free function to be **constexpr** for which there can be no valid *definition* — e.g., **constexpr void** `f();` — as the return type of a **constexpr** function's *definition* must satisfy certain requirements, including (in C++11 only) that its return type must not be **void**; see *Restrictions on* **constexpr** *function bodies (C++11 only)* on page 250.

[2] At the time **constexpr** was added to the language, it was a feature under development (and still is); see Section 2.2."**constexpr** Functions '14" on page 595.

[3] C++20 formalized this notion with the term **manifestly constant evaluated** to capture all places where the value of an expression must be determined at compile time. This new term coalesces descriptions in several places in the Standard where this concept had previously been used without being given a common name.

```
    static_assert(factorial(5) == 120, "");
        // OK, guaranteed to be evaluated at compile time since factorial is
        // used in a constexpr context

    std::cout << factorial(n);
        // OK, likely to be evaluated at run time since n is not a constant
        // expression

    static_assert(factorial(n) > 0, "");
        // Error, n is not a constant expression.
}
```

As illustrated above, simply invoking a **constexpr** function with arguments that are constant expressions does *not* guarantee that the function will be evaluated at compile time. The only way to *guarantee* compile-time evaluation of a **constexpr** function is to invoke it in places where a constant expression is mandatory, such as array bounds, static assertions (see Section 1.1."**static_assert**" on page 103), **alignas** specifiers (see Section 2.1."**alignas**" on page 158), nontype template arguments, and so on.

It is important to understand that if the compiler is required to evaluate a **constexpr** function in a constant expression with *compile-time constant* argument values for which the evaluation would require any operation not available at compile time (e.g., **throw**), the compiler will have no choice but to report an error:

```
constexpr int h(int x) { return x < 5 ? x : throw x; }  // OK, constexpr func

int a4[h(4)];  // OK, creates an array of four integers
int a6[h(6)];  // Error, unable to evaluate h on 6 at compile time
```

In the code snippet above, although we are able to size the *file-scope*[4] `a4` array because the path of execution within the valid **constexpr** function `h` does not involve a **throw**, such is not the case with `a6`. That a valid **constexpr** function can be invoked with compile-time constant arguments and still not be evaluable at compile time is noteworthy.

So far we have discussed **constexpr** functions in terms of *free* functions. As we shall see, **constexpr** can also be applied to *free*-function *templates*, *member* functions (importantly, constructors), and *member*-function templates; see **constexpr** *member functions* on page 248. Just as with free functions, only **constexpr** member functions are eligible to be evaluated at compile time.

What's more, we'll see that there is a category of user-defined types — called **literal**

[4] A common extension of popular compilers to allow (by default) variable-length arrays within function bodies but (as illustrated above) *never* at *file* or *namespace* scope:

```
void g()
{
    int a4[h(4)];  // OK, creates an array of four integers
    int a6[h(6)];  // Warning: ISO C++ forbids variable-length array a6.
                   // But with some compilers, h(6) might be invoked at
                   // run time and throw.
}
```

It is only by compiling with `-Wpedantic` that we get even so much as a warning!

types — whose *operational definition*[5] (for now) is that at least one of its values can participate in constant expressions:

```
struct Int  // example of a literal type
{
    int d_val;                                      // plain old int data member
    constexpr Int(int val) : d_val(val) { }         // constexpr value constructor
    constexpr int val() const { return d_val; }     // constexpr value accessor
              int dat() const { return d_val; }     // nonconstexpr accessor
};

constexpr int f(){ return Int(5).d_val; }  // OK, constexpr value constructor
constexpr int g(Int i){ return i.val(); }  // OK, constexpr value accessor
constexpr int h(Int i){ return i.dat(); }  // Error, nonconstexpr accessor
```

The basic, intuitive idea of what makes the user-defined `Int` type above a literal type is that it is possible to initialize objects of this type in a **constexpr** context[6]. What makes `Int` a "useful" literal type is that there is at least one way of extracting a value (either directly or via a **constexpr** accessor) such that the programmer can make use of it — typically at compile time. We might, however, imagine a use for a valid literal type that could be constructed at compile time but not otherwise *used* until run time:

```
class StoreForRt   // compile-time constructible (only) literal type
{
    int d_value;   // There is no way of accessing this value at compile time.

public:
    constexpr StoreForRt(int value) : d_value(value) { }  //       constexpr
    int value() const { return d_value; }                  // not constexpr
};
```

Contrived though it might seem, the example code above is representative of an application of **constexpr** where the construction of an object can benefit from compile-time optimization whereas access to the constructed data cannot. It is instructive, however, to first prove that such an object can in fact be constructed at compile time without employing other C++11 features. To that end, we will create a wrapper literal type, `W`, that contains both a member object of type `StoreForRt` and also one of type **int**. For `W` to be a literal type, both of its members must themselves be of literal type, and **int**, being a built-in type (all of which are literal types), is one:

```
struct W  // used to demonstrate compile-time constructability of StoreForRt
{
    StoreForRt d_i;                                 // usable only at run time
    int        d_j;                                 // usable at compile time
    constexpr W(int i, int j) : d_i(i), d_j(j) { }  // constexpr constructor
```

[5] By *operational definition* here, we mean a rule of thumb that would typically hold in practice; see *Literal types (defined)* on page 260.

[6] This initialization can be made possible through a **constexpr** constructor or through a type that can be **list initialized** without needing to invoke any non**constexpr** constructors; see Section 2.1."Braced Init" on page 198 and Section 1.2."Aggregate Init '14" on page 127.

```
};

static_assert(W(1,2).d_j == 2, "");
    // OK, can use W in a constexpr context so StoreForRt is of literal type

static_assert(StoreForRt(5).value() == 5, "");  // Error, value not constexpr
    // There is no way we can access d_value at compile time.
```

As the example code above demonstrates, `StoreForRt` is a literal type because it is used to declare a data member of a user-defined type, `W`, that in turn has been demonstrated to be *used* in a context that requires a constant expression. It is not, however, possible to do anything more with that constructed object at compile time (except for obtaining certain generic compile-time properties, such as its size (**sizeof**) or alignment; see Section 2.1."**alignof**" on page 173).

As it happens, the compiler doesn't actually care whether it can extract values from the compile-time-constructed object's data members: The compiler cares only that it can do — at compile time — all the evaluations demanded of it, with the assumption that those will involve a minimum of creating and destroying such objects; see *Literal types (defined)* on page 260.

To demonstrate that the same object can be (1) *constructed* at compile time and (2) *used* at run time, we will need to resort to use of a C++11 companion feature of **constexpr** functions, namely **constexpr** variables (see Section 2.1."**constexpr** Variables" on page 282):

```
constexpr StoreForRt x(5);  // OK, object x constructed in a constexpr context

int main() { return x.value(); }  // OK, x.value() used (only) at run time
```

Only literal types are permitted as *parameters* and return types for **constexpr** functions:

```
constexpr int  f11(StoreForRt x) { return 0; }  // OK, x is a literal type
constexpr void f14(StoreForRt x) { }            // OK, in C++14
```

See **constexpr**-*function parameter and return types* on page 259.

constexpr is part of the public interface

of-the-public-interface

When a **constexpr** function is invoked with an argument that is *not* known at compile time, compile-time evaluation of the function itself is not possible, and that invocation simply cannot be used in a context where a compile-time constant is required; runtime evaluation, however, is still permitted:

```
      int  i = 10;  // modifiable int variable
const int  j = 10;  // unmodifiable int variable (implicitly constexpr)
      bool mb = 0;  // modifiable bool variable

constexpr int f(bool b) { return b ? i : 5; }  // sometimes works as constexpr
constexpr int g(bool b) { return b ? j : 5; }  //    always works as constexpr

static_assert(f(mb), "");  // Error, mb is not usable in a constant expression.
static_assert(f(0),  "");  // OK
static_assert(f(1),  "");  // Error, i is not usable in a constant expression.
```

```
static_assert(g(mb), "");  // Error, mb is not usable in a constant expression.
static_assert(g(0),  "");  // OK
static_assert(g(1),  "");  // OK, j is usable in a constant expression.

int xf = f(mb);  // OK, runtime evaluation of f
int xg = g(mb);  // OK, runtime evaluation of g
```

In the example above, `f` can sometimes be used as part of constant expression but only if its argument is itself a constant expression and `b` evaluates to **`false`**. Function `g`, on the other hand, requires only that its argument be a constant expression for it to always be usable as part of a constant expression. If there is not at least one set of compile-time constant argument values that would be usable at compile time then it is **ill formed, no diagnostic required (IFNDR)**:

```
constexpr int h1(bool b) { return f(b); }
    // OK, there is a value of b for which h1 can be evaluated at compile time.

constexpr int h2() { return f(1); }
    // There's no way to invoke h2 so that it can be evaluated at compile time.
    // (This function is ill formed, no diagnostic required.)
```

Here `h1` is well formed since it can be evaluated at compile time when the value of `b` is **`true`**; `h2`, on the other hand, is ill formed because it can *never* be evaluated at compile time. A sophisticated analysis would, however, be required to establish such a proof, and popular compilers often do not currently try; future compilers are, of course, free to do so.

Being part of the user interface, a function marked as being **`constexpr`** might suggest (albeit wrongly) to some prospective clients that the function will *necessarily* support compile-time evaluation whenever it is invoked with compile-time constant arguments. Although *adding* a **`constexpr`** specifier to a function between library releases is not a problematic API change, *removing* a **`constexpr`** specifier definitely is, because existing users might be relying on compile-time evaluation in their code. Library developers have to make a conscious decision as to whether to mark a function **`constexpr`** — especially with the heavy restrictions imposed by the C++11 Standard — since improving the implementation of the function while respecting those restrictions might prove insurmountable; see *Potential Pitfalls — Prematurely committing to* **`constexpr`** on page 278.

definition-visibility

Inlining and definition visibility

A function that is declared **`constexpr`** is (1) implicitly declared **`inline`** and (2) automatically eligible for compile-time evaluation. Note that adding the **`inline`** specifier to a function that is already declared **`constexpr`** has no effect:

```
       constexpr int f1() { return 0; } // automatically inline
inline constexpr int f1();              // redeclares the same f1() above
```

As with all **`inline`** functions, it is an **one-definition rule (ODR)** violation if definitions in different translation units within a program are not token-for-token the same. If definitions do differ across translation units, the program is ill formed, no diagnostic required (IFNDR):

```
// file1.h
          inline int f2() { return 0; }
       constexpr int f3() { return 0; }

// file2.h
          inline int f2() { return 1; }  // Error, no diagnostic required
       constexpr int f3() { return 1; }  // Error, no diagnostic required
```

When a function is declared **constexpr**, *every* declaration of that function, including its definition, must also be explicitly declared **constexpr** or else the program is ill formed:

```
constexpr int f4();
constexpr int f4() { return 0; }  // OK, constexpr matching exactly

constexpr int f5() { return 0; }
          int f5() { return 0; }  // Error, constexpr missing

          int f6();
constexpr int f6() { return 0; }  // Error, constexpr added
```

An explicit specialization of a function template declaration may, however, differ with respect to its **constexpr** specifier. For example, a general function template (e.g., `func1` in the code snippet below) might be declared **constexpr** whereas one of its explicit specializations (e.g., `func1<int>`) might not be:

```
template <typename T>    // general function template declaration/definition
constexpr bool func1(T)  // general template is declared constexpr
{
    return true;
}

template <>              // explicit specialization definition
bool func1<int>(int)     // The explicit specialization is not constexpr.
{
    return true;
}

static_assert(func1('a'), "");  // OK, general function template is constexpr.
static_assert(func1(123), "");  // Error, int specialization is not constexpr.
```

Similarly, the roles can be reversed where only an explicit specialization (e.g., `func2<int>` in the example below) is **constexpr**:

```
template <typename T> bool func2(T)        { return true; }  // general template
template <> constexpr bool func2<int>(int) { return true; }  // specialization

static_assert(func2('a'), "");  // Error, general template is not constexpr.
static_assert(func2(123), "");  // Ok, int specialization is constexpr.
```

Just as with any other function, a **constexpr** function may appear in an expression before its body has been seen. A **constexpr** function's definition, however, must appear before that function is evaluated to determine the value of a *constant expression*:

```
constexpr int f7();                  // declared but not yet defined
constexpr int f8() { return f7(); }  // defined with a call to f7
constexpr int f9();                  // declared but not defined in this TU

int main()
{
    return f8() + f9();  // OK, presumes f7 and f9 are defined and linked
                         // with this TU
}

static_assert(0 == f8(), "");  // Error, body of f7 has not yet been seen.
static_assert(0 == f9(), "");  // Error,   "  "  f9  "   "   "   "     "

constexpr int f7() { return 0; }  // definition matching forward declaration

static_assert(0 == f8(), "");  // OK, body of f7 is visible from here
static_assert(0 == f9(), "");  // Error, body of f9 has not yet been seen

// Oops, failed to define f9 in this translation unit; compiler might warn
```

In the example code above, we have declared three **constexpr** functions: `f7`, `f8`, and `f9`. Of the three, only `f8` is defined ahead of its first use. Any attempt to evaluate a **constexpr** function whose definition has not yet been seen — either directly (e.g., `f9`) or indirectly (e.g., `f7` via `f8`) — in a **constexpr** context results in a compile-time error. Notice that, when used in expressions whose value does not need to be determined at compile time (e.g., the **return** statement in `main`), there is no requirement to have seen the body. The compiler is, of course, still free to optimize, and, depending on the optimization level, it might substitute the function bodies **inline** and perform constant folding to the extent possible. Note that, in this case, `f9` was not defined anywhere within the TU. Just as with any other **inline** function whose definition is never seen, many popular compilers will warn if they see any expressions that might invoke such a function, but it is not ill formed because the definition could (by design) reside in some other TU (see also Section 2.1."**extern template**" on page 329.

However, when a **constexpr** function is *evaluated* to determine the value of a constant expression, its body, and anything upon which it depends must have already been seen; notice that we didn't say "appears as part of a constant expression" but instead said "is evaluated to determine the value of a constant expression."

We *can* have something that is not itself a (*compile-time*) constant expression (or even one that is convertible to **bool**) *appear* as a part of a constant expression *provided* that it never actually gets evaluated at compile time:

```
static_assert(false ? throw : true,  "");  // OK
static_assert(true  ? throw : true,  "");  // Error, throw not constexpr
static_assert(true  ?  true : throw, "");  // OK
static_assert(false ?  true : throw, "");  // Error, throw not constexpr

static_assert((true, throw), "");  // Error, throw not convertible to bool
static_assert((throw, true), "");  // Error, throw is not constexpr
```

```
extern volatile bool x;
static_assert((true, x), "");      // Error, x not constexpr
static_assert((x, true), "");      // Error,  "   "      "

static_assert(true || x,    "");   // OK
static_assert(x    || true, "");   // Error, x not constexpr
```

Note that the *comma* (`,`) **sequencing operator** incurs evaluation of both of its arguments whereas the *logical-or* (`||`) **operator** requires only that its two arguments be convertible to **`bool`**, where actual evaluation of the second argument might be short circuited; see "**??**" on page **??**[**AUs: there is no feature called "Implicit Conversion"**].

mutual-recursion

Mutual recursion

Mutually recursive functions can be declared **`constexpr`** so long as neither is called in a **`constexpr`** context until both definitions have been seen:

```
constexpr int gg(int n);       // forward declaration

constexpr int ff(int n)        // declaration and definition
{
    return (n > 0) ? gg(n - 1) + 1 : 0;
}

int hh()
{
    return ff(1) + gg(2);  // OK, not a constexpr context
}

static_assert(ff(3), "");      // Error: body of gg has not yet been seen
static_assert(gg(4), "");      // Error:  "   "   "    "   "   "   "    "

constexpr int gg(int n)        // redeclaration and definition
{
    return (n > 0) ? ff(n - 1) + 1 : 0;
    static_assert(ff(5), "");  // Error: body of gg has not yet been seen
    static_assert(gg(6), "");  // Error:  "   "   "    "   "   "   "    "
}
static_assert(ff(7), "");      // OK: bodies of ff and gg have now been seen
static_assert(gg(8), "");      // OK:    "   "   "    "   "    "    "   "    "
```

In the example code above, we have two recursive functions, `ff` and `gg`, with `gg` being forward declared. When used within (non**`constexpr`**) function `hh`, the mutually recursive calls between `ff` and `gg` are not evaluated until the compiler has seen the bodies of both `ff` and `gg`, i.e., at run time. Conversely, the **`static_assert`**s of `ff(3)` and `gg(4)` are **`constexpr`** contexts and are ill formed because the body of `gg` has not yet been seen at that point in the compilation. The **`static_assert`**s within the function body of `gg` are similarly evaluated at the point they are seen during compilation, where `gg` is not yet (fully) defined and callable, and so are also ill formed. Finally, the **`static_assert`**s of `ff(7)` and `gg(8)`

can be evaluated at compile time because the bodies of both `ff` and `gg` have both been seen by the compiler by that point in the compilation.

The type system and function pointers

and-function-pointers

Similarly to the **inline** keyword, marking a function **constexpr** does *not* affect its type; hence, it is not possible to have, say, two overloads of a function that differ only on whether they are **constexpr** or to define a pointer to exclusively **constexpr** functions:

```
constexpr int f(int) { return 0; }  // OK
int f(int)           { return 0; }  // Error, int f(int) is now multiply defined.

typedef constexpr int(*MyFnPtr)(int);
    // Error, constexpr cannot appear in a typedef declaration.

void g(constexpr int(*MyFnPtr)(int));
    // Error, a parameter cannot be declared constexpr.
```

If a function pointer is not itself declared **constexpr**, its value cannot be read as part of evaluating a constant expression. If the function pointer *is* **constexpr** but points to a non**constexpr** function, it *cannot* be used to invoke that function at compile time:

```
constexpr bool g() { return true; }  //    constexpr function returning true
          bool h() { return true; }  // nonconstexpr function returning true

typedef bool (*Fp)();  // pointer to function taking no args. and returning bool

constexpr Fp m = g;  //    constexpr pointer to a    constexpr function
          Fp n = g;  // nonconstexpr pointer to a    constexpr function
constexpr Fp p = h;  //    constexpr pointer to a nonconstexpr function
          Fp q = h;  // nonconstexpr pointer to a nonconstexpr function
constexpr Fp r = 0;  //    constexpr pointer having nullptr (address) value

static_assert(p == &h, "");  // Ok, reading the value of a constexpr pointer
static_assert(q == &h, "");  // Error, q is not a constexpr pointer.
static_assert(r == 0,  "");  // Ok, reading the value of a constexpr pointer
static_assert(m(), "");      // Ok, invoking a constexpr function through a
                             //     constexpr pointer
static_assert(n(), "");      // Error, n is not a constexpr pointer.
static_assert(p(), "");      // Error, can't invoke a nonconstexpr function
static_assert(q(), "");      // Error, q is not a constexpr pointer.
static_assert(r(), "");      // Error, 0 doesn't designate a function.
```

constexpr member functions

expr-member-functions

Member functions — including special member functions (see **??** on page **??**[**AUs: there is no section called "special member functions"**]), such as *constructors* (but not *destructors*; see *Literal types (defined)* on page 260) — can be declared to be **constexpr**:

```
class Point1
{
```

```
    int d_x, d_y;  // two ordinary int data members
public:
    constexpr Point1(int x, int y) : d_x(x), d_y(y) { }  // OK, is constexpr

    constexpr int x()       { return d_x; }  // OK, is constexpr
              int y() const { return d_y; }  // OK, is not constexpr
};
```

Simple classes, such as `Point1` (in the code snippet above), having at least one **constexpr** constructor that is not a *copy* or *move* constructor as well as a **constexpr** accessor (or public data member) and satisfying the other requirements of being a literal type — see *Literal types (defined)* on page 260 — can be used as part of constant expressions:

```
int ax[Point1(5, 6).x()];  // Ok, array of 5 ints
int ay[Point1(5, 6).y()];  // Error, accessor y is not declared constexpr.
```

Member functions decorated with **constexpr** are implicitly **const**-qualified in C++11 (but not in C++14 — see Section 2.2."**constexpr** Functions '14" on page 595):

```
struct Point2
{
    int d_x, d_y;                                          // same as for Point1
    constexpr Point2(int x, int y) : d_x(x), d_y(y) { }  //   "   "  "     "

    constexpr int& x() { return d_x; }                     // accessor #1
        // Error, binding int& reference to const int discards qualifiers.

    constexpr const int& y() const { return d_y; }         // accessor #2
        // OK, the 2nd const qualifier is redundant (but only in C++11).

    constexpr const int& y() { return d_y; }               // accessor #3
        // Error, redefinition of constexpr const int& Point::y() const
};
```

In the `Point2` **struct** example above, accessor #1 is implicitly declared **const** in C++11 (but not C++14); hence, the attempt to return a modifiable *lvalue* reference to the implicitly **const** `d_x` data member erroneously discards the **const** qualifier. Had we declared the **constexpr** function to return a **const** reference, as we did for accessor #2, the code would have compiled just fine. Note that the explicit **const** member-function qualifier (the second **const**) in accessor #2 is redundant in C++11 (but not in C++14); having it there ensures that the meaning will not change when this code is recompiled under a subsequent version of the language. Lastly, note that omitting the member-function qualifier in accessor #3 fails to produce a distinct overload in C++11 (but would in C++14).

Because declaring a member function to be **constexpr** implicitly makes it **const**-qualified (C++11 only), there can be unintended consequences:

```
struct Point3
{
    int d_x, d_y;                                          // same as for Point1
    constexpr Point3(int x, int y) : d_x(x), d_y(y) { }  //   "   "  "     "
```

```
    constexpr int x() const { return d_x; }  // OK
    constexpr int y()       { return d_y; }  // OK, const is implied in C++11.

              int setX(int x) { return d_x = x; }  // OK, but not constexpr
    constexpr int setY(int y) { return d_y = y; }  // Error, implied const

    constexpr Point3& operator=(const Point3& p);
        // Error, copy (and move) assignment can't be constexpr in C++11.

};
```

Notice that declaring the "setter" member function `setY` (in the code example above) to be **`constexpr`** implicitly qualifies the member function as being **`const`**, thereby making it an error for any **`constexpr`** member function to attempt to modify its own object's data members. The inevitable corollary is that any reasonable implementation of *copy* or *move* assignment cannot be declared **`constexpr`** in C++11 (but can be as of C++14).

Finally, **`constexpr`** member functions cannot be **`virtual`**[7] but can co-exist in the same class with other member functions that *are* virtual.

Restrictions on `constexpr` function bodies (C++11 only)

n-bodies-(c++11-only)

The list of C++ programming features permitted in the bodies of **`constexpr`** functions for C++11 is small and reflective of the nascent state of this feature when it was first standardized. To begin, the body of a **`constexpr`** function is not permitted to be a **function-try-block**:

```
          int g1()     { return 0; }                // OK
constexpr int g2()     { return 0; }                // Ok, no try block
          int g3() try { return 0; } catch(...) {}  // OK, not constexpr
constexpr int g4() try { return 0; } catch(...) {}  // Error, not allowed
```

C++11 **`constexpr`** functions that are not *deleted* or *defaulted* (see Section 1.1."**??**" on page **??** and Section 1.1."Defaulted Functions" on page 30, respectively) may consist of only **null statements**, static assertions (see Section 1.1."**`static_assert`**" on page 103), **`using` declarations**, **`using` directives**, **`typedef`** declarations, and alias declarations (see Section 1.1."**`using`** Aliases" on page 121) that do not define a class or enumeration. Other than constructors, the body of a **`constexpr`** function must include exactly one **`return`** statement. A **`constexpr`** constructor may have a member-initializer list but no other additional statements (but see *Constraints specific to constructors* on page 251). Use of the **ternary operator**, **comma operator**, and recursion are allowed:

```
constexpr int f(int x)
{
    static_assert(sizeof(int) == 4, "");   // OK, static assertion
    using MyInt = int;                     // OK, type alias
    return x > 5 ? x : f(x + 2), f(x + 1); // OK, ternary, comma, and recursion
    ;; ;   ; { ; { ; ; } }                 // OK, null statements/nested blocks
}
```

[7] C++20 allows **`constexpr`** member functions to be **`virtual`** (?).

Many familiar programming constructs such as runtime assertions, local variables, **if** statements, modifications of function parameters, and **using** directives that define a type are, however, *not* permitted (in C++11):

```
constexpr int g(int x)
{
    assert(x < 100);          // Error, no runtime asserts
    int y = x;                // Error, no local variables
    if (x > 5) { return x; }  // Error, no if statements
    using S = struct { };     // Error, no aliases that define types
    return x += 3;            // Error, no compound assignment
}
```

The good news is that the aforementioned restrictions on the kinds of constructs that are permitted in **constexpr** function bodies are significantly relaxed as of C++14; see Section 2.2."**constexpr** Functions '14" on page 595.

Irrespective of the *kinds* of constructs that are allowed to appear in a **constexpr** function body, every invocation (evaluation) of a function, a constructor, or an implicit conversion operator in the **return** statement must itself be usable in some (at least one) constant expression, which means the corresponding function *must* (at a minimum) be declared **constexpr**:

```
          int ga() { return 0; }  // nonconstexpr function returning 0
constexpr int gb() { return 0; }  //    constexpr function returning 0

struct S1a {           S1a() { } };  // nonconstexpr default constructor
struct S1b { constexpr S1b() { } };  //    constexpr default constructor

struct S2a { operator int() { return 5; } };      // nonconstexpr conversion
struct S2b { constexpr operator int() { return 5; } }; // constexpr conversion

constexpr int f1a() { return ga(); }  // Error, ga is not constexpr.
constexpr int f1b() { return gb(); }  // Ok, gb is constexpr.

constexpr int f2a() { return S1a(), 5; }  // Error, S1a ctor is not constexpr.
constexpr int f2b() { return S1b(), 5; }  // Ok, S1b ctor is constexpr.

constexpr int f3a() { return S2a(); } // Error, S2a conversion is not constexpr.
constexpr int f3b() { return S2b(); } // Ok, S1b conversion is constexpr.
```

Note that non**constexpr** implicit conversions, as illustrated by `f3a` above, can also result from a non**constexpr**, non**explicit** constructor that accepts a single argument.

Constraints specific to constructors

pecific-to-constructors

In addition to the general restrictions on a **constexpr** function's body (see **constexpr**-*function parameter and return types* on page 259) and its allowed parameter and return types (see **constexpr**-*function parameter and return types* on page 259), there are several additional requirements specific to constructors.

1. The body of a **constexpr** constructor is restricted in the same way as any other **constexpr** function, with the obvious lack of a **return** statement being allowed.

Hence, the body of **constexpr** constructor, in addition to not being permitted within a function **try** block (like any other **constexpr** function) must be essentially empty with just a very few exceptions — e.g., null statements, **static_assert** declarations, **typedef** declarations (see also Section 1.1."**using** Aliases" on page 121) that do not define classes or enumerations, **using** declarations, and **using** directives:

```
namespace n           // enclosing namespace
{

class C { /*...*/ };  // arbitrary class definition

struct S
{
    constexpr S(bool) try { } catch (...) { }  // Error, function try block
              S(char) try { } catch (...) { }  // OK, not declared constexpr

    constexpr S(int)
    {
        ;                       // OK, null statement
        {}                      // Error, though accepted by most compilers
        static_assert(1, "");   // OK, static_assert declaration
        typedef int Int;        // OK, simple typedef alias
        using Int = int;        // OK, simple using alias
        typedef enum {} E;      // Error, typedef used to define enum E
        using n::C;             // OK, using declaration
        using namespace n;      // OK, using directive
    }
};

}  // close namespace
```

2. All non**static** data members and base-class subobjects of a class must be initialized by a **constexpr** constructor,[8] and the initializers themselves must be usable in a *constant expression*. Members with a trivial default constructor must be explicitly initialized in the member-initializer list or via a **default member initializer** (i.e., they cannot be left in an uninitialized state):

```
struct B  // constexpr constructible only from argument convertible to int
{
    B() { }
    constexpr B(int) { }  // constexpr constructor taking an int
};

struct C  //  constexpr default constructible
{
    constexpr C() { }  // constexpr default constructor.
```

[8] The requirement that all members and base classes be initialized by a constructor that is explicitly declared **constexpr** is relaxed in C++20 provided that uninitialized entities are not accessed at compile time.

```
};

struct D1 : B  // public derivation
{
    constexpr D1() { }  // Error, B has nonconstexpr default constructor
};

struct D2 : B  // public derivation
{
    int d_i;  // nonstatic, trivially constructible data member
    constexpr D2(int i) : B(i) { }  // Error, doesn't initialize d_i
};

          int f1() { return 5; }  // nonconstexpr function
constexpr int f2() { return 5; }  //    constexpr function

struct D3 : C  // public derivation
{
    int d_i = f1();  // initialization using nonconstexpr function
    int d_j = f2();  // initialization using    constexpr function

    constexpr D3() { }  // Error, d_i not constant initialized

    constexpr D3(int i) : d_i(i) { }  // OK, d_i set from init list
};
```

The example code above illustrates various ways in which a base class or non**static** data member might fail to be initialized by a constructor that is explicitly declared **constexpr**. In the final derived class, `D3`, we note that there are two data members, `d_i` and `d_j`, having member initializers that use a non**constexpr** function, `f1`, and a **constexpr** function, `f2`, respectively. The implementation of the **constexpr** *default* constructor , `D3()`, is erroneous because data member `d_i` would be initialized by the `non`constexpr function `f1` at run time. On the other hand, the implementation of the *value* constructor, `D3(int)`, is fine because the data member `d_i` is set in the member-initializer list, thereby enabling compile-time evaluation.

3. Defining a constructor to be **constexpr** requires that the class have no **virtual** base classes[9]:

```
struct B { constexpr B();  /*...*/ };  // some arbitrary base class

struct D : virtual B
{
    constexpr D(int) { }   // Error, class D has virtual base class B.
};
```

[9] C++20 removes the restriction that a constructor cannot be **constexpr** if the class has any virtual base classes.

4. Like any **special member function**, a constructor that is *explicitly* declared to be **constexpr** can always be suppressed using **=delete** (see Section 1.1."**??**" on page **??**).[10]. If a constructor is implemented using **=default**, however, an error will result *unless* the defaulted definition would have been implicitly **constexpr** (see Section 1.1."Defaulted Functions" on page 30):

```
struct S1
{
    S1() { };           // nonconstexpr default constructor
    S1(const S1&) { };  //  "      "      copy         "
    S1(char) { };       //  "      "      value        "
};

struct S2
{
    S1 d_s1;
    constexpr S2() = default;          // default constructor
        // Error, S1's default constructor isn't constexpr.

    constexpr S2(const S2&) = delete;  // copy constructor
        // OK, make declaration inaccessible and suppress implementation

    S2(char c) : d_s1(c) { }           // value constructor
       // OK, this constructor is not declared to be constexpr.
};
```

In the example above, explicitly declaring the *default* constructor of `S2` to be **constexpr** is an error because an implicitly defined *default* constructor would not have been **constexpr**. Using **=delete** *declares* but does not *define* a **constexpr** function; hence, no semantic validation with respect to **constexpr** is applied to `S2`'s (suppressed) *copy* constructor. Because `S2`'s *value* constructor (from **char**) is not explicitly declared **constexpr**, there is no issue with delegating to its non**constexpr** member *value*-constructor counterpart.

5. An implicitly defined *default* constructor (generated by the compiler) performs the set of initializations of the class that would be performed by a user-written default constructor for that class having no member-initializer list and an empty function body. If such a user-defined *default* constructor would satisfy the requirements of a **constexpr** constructor, the implicitly defined *default* constructor is a **constexpr** constructor (and similarly for the implicitly defined *copy* and *move* constructors) irrespective of whether it is explicitly declared **constexpr**. Explicitly declaring a defaulted constructor **constexpr** that is *not* inherently **constexpr** is, however, a compile-time error (see Section 1.1."Defaulted Functions" on page 30):

```
struct I0  { int i; /* implicit default ctor */ };  // OK, literal type.
```

[10] Deleting a function explicitly declares it, makes that declaration *inaccessible*, and suppresses generation of an implementation; see **??** on page **??**[**AUs: There is no subsection called "Detecting literal types"**]

```
struct I1a { int i;            I1a()        { } };  // OK, i is not init.
struct I1b { int i;  constexpr I1b()        { } };  // Error, i not init.

struct I2a { int i;            I2a() = default; };  // OK, but not constexpr
struct I2b { int i;  constexpr I2b() = default; };  // Error, i is not init.

struct I3a { int i;            I3a() : i(0) { } };  // OK, i is init.
struct I3b { int i;  constexpr I3b() : i(0) { } };  // OK, literal type.

struct S0  { I3b i; /* implicit default ctor */ };  // OK, literal type.

struct S1a { I3b i;            S1a()        { } };  // OK, i is init.
struct S1b { I3b i;  constexpr S1b()        { } };  // OK, literal type.

struct S2a { I3b i;            S2a() = default; };  // OK, literal type.
struct S2b { I3b i;  constexpr S2b() = default; };  // OK, literal type.
```

The example code above attempts to illustrate the subtle differences between a data member of *scalar* literal type (e.g., **int**) and one of *user-defined* literal type (e.g., `I3b`). The first difference, illustrated by `I1a` versus `S1a`, is that the former always leaves its own data member, `i`, uninitialized, while the latter invariably zero-initializes its `i`. The second difference, seen in `I1b` versus `S1b`, is that the former is explicitly not initialized whereas it is always implicitly initialized in the latter.

Note that, although every literal type needs to have a way to construct it in a **constexpr** context, not *every* constructor of a literal type needs to be **constexpr**; see *Literal types (defined)* on page 260.

6. **Braced initialization**, while not always incurring constructor invocation, can still be done during compile-time evaluation. This initialization must only involve operations that can be done during constant evaluation, but, unlike a hand-written constructor, it will (a) first **zero-initialize** all members and (b) skip a (possibly deleted or non**constexpr**) **trivial** default constructor.[11].

   Braced initialization can produce surprising cases where a non**constexpr** constructor seems like it should be invoked but is instead skipped during braced or value initialization because it is trivial:

```
struct S1  // example with a nonconstexpr trivial default constructor
{
    int d_i;        // not initialized by S1()
    S1() = default; // trivial, nonconstexpr
};

static_assert( S1().d_i  == 0, "");  // OK, value initialization
static_assert( S1{}.d_i  == 0, "");  // OK, braced initialization
```

[11] A default constructor is trivial if (a) it is implicit, defaulted, or deleted; (b) all non**static** data members have trivial default constructors and no default member initializers; and (c) all base classes are non**virtual** and have trivial default constructors.

```
static_assert( S1{7}.d_i == 7, "");  // OK, braced (list) initialization
```

Braced initialization can even skip a deleted constructor (see Section 1.1."**??**" on page **??**) where regular initialization would not:

```
struct S2  // example with a deleted default constructor
{
    int d_i;       // not initialized
    S2() = delete; // trivial
};

static_assert( S2().d_i  == 0, "");  // Error, invokes deleted constructor
static_assert( S2{}.d_i  == 0, "");  // OK, braced initialization
static_assert( S2{7}.d_i == 7, "");  // OK, braced (list) initialization
```

Finally, it is important to note that this form of braced initialization is not restricted to just aggregates and requires only that the default constructor be trivial[12]:

```
struct S3 {  // example of a nonaggregate type
    int d_i;              // not initialized
    S3() = default;       // trivial
    S3(int i) : d_i(i) {} // not an aggregate
};

static_assert( S3().d_i  == 0, "");  // OK, value initialization
static_assert( S3{}.d_i  == 0, "");  // OK, braced initialization
static_assert( S3{7}.d_i == 7, "");  // Error, nonconstexpr constructor
```

7. For a **union**, exactly one of its data members must be initialized with (1) a default initializer that is a *constant expression* (see Section 2.1."Default Member Init" on page 296), (2) a **constexpr** constructor, or (3) braced initialization that picks a member that can be initialized in a **constexpr** context:

```
// unions having no explicit constructors
union U0 { bool b;      char c;       };  // OK, neither member initialized
union U1 { bool b = 0;  char c;       };  // OK, first      "       "
union U2 { bool b;      char c = 'A'; };  // OK, second     "       "
union U3 { bool b = 0;  char c = 'A'; };  // Error, multiple initialized

// unions having constexpr constructors
union U4 { bool b; char c;        constexpr U4() { } };  // Error, uninit
union U5 { bool b; char c = 'A';  constexpr U5() { } };              // OK
```

[12]The original intent was clearly to enable any initialization that involved only operations that could be done at compile time to be a valid initialization for a literal type. This was originally noted by Alisdair Meredith in CWG issue 644 (see **?**), which was inadvertently undone by mistakenly generalizing the solution to just aggregates with the resolution of CWG issue 981 (see **?**) and is pending a final resolution when wording is adopted for CWG issue 1452 (see **?**). Even with the actual wording not allowing this form of initialization to be considered enough to make a type a literal type, all current compiler implementations have consistently adopted support for this interpretation and we are simply waiting for the Standard to catch up to existing practice.

```
union U6 { bool b; char c;        constexpr U6() : c('A') { } };      // OK
union U7 { bool b; char c;        constexpr U7(bool v) : b(v) { } };  // OK

struct S                          // S is a literal type.
{
    U0 u0{};                      // braced  initialized
    U1 u1; U2 u2; U5 u5; U6 u6;   // default initialized
    U7 u7;                        // value   initialized
    constexpr S() : u7(0) { }     // OK, all members are initialized.
};

constexpr int test(S t) { return 0; }  // OK, confirms S is a literal type
```

The example code above illustrates various ways in which unions (e.g., `U0`–`U2` and `U5`–`U7`) can be used that allow them to be initialized by a **`constexpr`** constructor (e.g., `S()`). The existence of at least one (non*copy* non*move*) **`constexpr`** constructor implies that the class (e.g., `S`) comprising these unions is a literal type, which we have confirmed using the C++11 **interface test** idiom; see **??** on page **??**.[**AUs: There is no section called "Parameters and return types" Did you mean "constexpr-function parameter and return types"?**]

8. If the constructor *delegates* to another constructor in the same class (see Section 1.1."Delegating Ctors" on page 43), that target constructor must be **`constexpr`**:

```
struct C0  // BAD IDEA: Only the default constructor is constexpr.
{
              C0(int)      { }  // Ok, but not declared constexpr
    constexpr C0() : C0(0) { }  // Error, delegating to nonconstexpr ctor
};

struct C1  // GOOD IDEA: Both default and value constructor are constexpr.
{
    constexpr C1(int)      { }  // Ok, declared constexpr
    constexpr C1() : C1(0) { }  // Ok, delegating to constexpr constructor
};
```

9. When initializing data members of a class (e.g., `S` below), any nonconstructor functions needed for implicitly converting the type of the initializing expression (e.g., `V` in the code snippet below) to that of data member (e.g., **`int`** or **`double`**) must also be **`constexpr`**:

```
struct V
{
    int v;
              operator    int() const { return v; }  // implicit conversion
    constexpr operator double() const { return v; }  // implicit conversion
};

struct S
```

```
{
    int i; double d;  // A constexpr constructor must initialize both members.

    constexpr S(const V& x, double y) : i(x), d(y) { }  // Error, the needed
        // int implicit conversion is not declared constexpr.

    constexpr S(int x, const V& y) : i(x), d(y) { }    // OK, the needed
        // double implicit conversion is declared constexpr.
};
```

pr-function-templates

constexpr function templates

Function templates, member function templates, and constructor templates can all be declared **constexpr** too and more liberally than for nontemplated entities. That is, if a particular explicit specialization of such a template doesn't meet the requirements of a **constexpr** function, member function, or constructor, it will not be invocable at compile time.[13] For example, consider a function template, `sizeOf`, that is **constexpr** only if its argument type, `T`, is a literal type:

```
template <typename T> constexpr int sizeOf(T t) { return sizeof(t); }
    // This function is constexpr only if T is a literal type.

struct S0 { int i;           S0() : i(0) { } };  // not a literal type.
struct S1 { int i; constexpr S1() : i(0) { } };  // a literal type.

int a[sizeOf(int())];  // OK,    int is     a literal type.
int b[sizeOf( S0())];  // Error,  S0 is not a literal type.
int c[sizeOf( S1())];  // OK,     S1 is     a literal type.
```

If no specialization of such a function template would yield a **constexpr** function, then the program is IFNDR. For example, if this same function template were implemented with a function body consisting of more than just a single **return** statement, it would be ill formed:

```
template <typename T>
constexpr int badSizeOf(T t) { const int s = sizeof(t); return s; }
    // This constexpr function template is IFNDR.
```

Most compilers, when compiling such a specialization for runtime use, will not attempt to determine if the **constexpr** would ever be valid. When invoked with arguments that are themselves constant expressions (such as a temporary of literal type), they do, however, often detect this ill formed nature and report the error:

```
int d[badSizeOf(S1())];  // Error, badSizeOf<S1>(S1) body not return statement
int e[badSizeOf(S0())];  // Error, badSizeOf<S0>(S0) body not return statement
int f = badSizeOf(S1()); // Oops, same issue but might work on some compilers
int g = badSizeOf(S0()); // Oops, same issue but often works without warnings
```

[13] A specialization that cannot be evaluated at compile time is, however, still considered **constexpr**. This is not readily observable but does enable some generic code to remain well formed as long as the particular specializations are not actually required to be evaluated at compile time. This rule was adopted with the resolution of CWG issue 1358 (see **?**).

It is important to understand that each of the four statements in the code snippet above are ill formed because the `badSizeOf` function template is itself ill formed. Although the compiler is not required to diagnose the general case, once we try to use an explicit instantiation of it in a **constexpr** context (e.g., `d` or `e`), it is mandatory that the supplied argument be used to determine the value of the constant expression or fail trying. When used in a non**constexpr** context (e.g., `f` or `g`), whether the compiler fails, warns, or proceeds is a matter of **quality of implementation (QoI)**.

ameter-and-return-types

constexpr-function parameter and return types

At this point, we arrive at what is perhaps the most confounding part of the seemingly *cyclical* definition of **constexpr** functions: A function cannot be declared **constexpr** unless the return type and every parameter of that function satisfies the criteria for being a literal type; a literal type is the category of types whose objects are permitted to be created and destroyed when evaluating a constant expression:

```
struct  Lt { int v; constexpr  Lt() : v(0) { } };  // literal type
struct Nlt { int v;           Nlt() : v(0) { } };  // nonliteral type

          Lt  f1() { return Lt();  }  // OK, no issues
constexpr Lt  f2() { return Lt();  }  // OK, returning literal type
          Nlt f3() { return Nlt(); }  // Ok, function is nonconstexpr
constexpr Nlt f4() { return Nlt(); }  // Error, constexpr returning nonliteral

          int g1(Lt  x) { return x.v; }  // OK, no issues
constexpr int g2(Lt  x) { return x.v; }  // OK, parameter is a literal type
          int g3(Nlt x) { return x.v; }  // OK, function is nonconstexpr
constexpr int g4(Nlt x) { return x.v; }  // Error, constexpr taking nonliteral
```

Consider that all *pointer* and *reference* types — being *built-in types* — are literal types and therefore can appear in the interface of a **constexpr** function irrespective of whether they point to a literal type:

```
constexpr int h1(Lt*  p) { return p->v; }  // OK, parameter is a literal type
constexpr int h2(Nlt* p) { return p->v; }  // OK,     "      "  "    "      "
constexpr int h3(Lt&  r) { return r.v; }   // OK,     "      "  "    "      "
constexpr int h4(Nlt& r) { return r.v; }   // OK,     "      "  "    "      "
```

However, note that, because it is not possible to construct an object of nonliteral type at compile time, there is no way to invoke `h2` or `h4` as part of a constant expression since the access of the member `v` in all of the above functions requires an already created object to exist. Pointers and references to nonliteral types that do not access those types can still be used:

```
Nlt arr[17];
constexpr Nlt& arr_0 = arr[0];                // OK, initializing a reference
constexpr Nlt *arr_0_ptr = &arr[0];           // OK, taking an address
constexpr Nlt& arr_0_ptr_deref = *arr_0_ptr;  // OK, dereferencing but not using
static_assert(&arr[17] - &arr[4] == 13,"");   // OK, pointer arithmetic
```

```
constexpr int arr_0_v = arr_0.v;              // Error, arr[0] is not usable.
constexpr int arr_0_ptr_v = arr_0_ptr->v;     // Error,    "     "   "    "
```

teral-types-(defined)

Literal types (defined)

Until now, we've discussed many ways in which understanding which types are literal types is important to understand what can and cannot be done during compile-time evaluation. We now elucidate how the language defines a literal type and, as such, how they are usable in two primary use cases:

- Literal types are eligible to be created and destroyed during the evaluation of a *constant expression*.
- Literal types are suitable to be used in the *interface* of a **constexpr** function, either as the return type or as a parameter type.

The criteria for determining whether a given type is a literal type can be divided into six parts:

1. Every scalar type is a literal type. Scalar types include all fundamental arithmetic (integral and floating point) types, all enumerated types, and all pointer types.

int	**int** is a *literal type*.
double	**double** is a *literal type*.
char*	**char*** is a *literal type*.
enum E { e_A }	E is a *literal type*.
struct S { S(); };	S is *not* a *literal type*.
S*	S* *is* a *literal type* (even though S is *not*).

   Note that a pointer is *always* a literal type — even when it points to a type that itself is *not* a literal type.

2. Just as with pointers, every reference type is a literal type irrespective of whether the type to which it refers is itself a literal type.

int&	**int&** is a *literal type*
S&	S& is a *literal type* (even if S is not).
S&&	S&& is a *literal type*.

3. A **class**, **struct**, or **union** is a literal type if it meets each of these four requirements:
   (a) It has a trivial destructor.[14]
   (b) Each non**static** data member is a non**volatile** literal type.[15]

[14] As of C++20, a destructor can be declared **constexpr** and even both **virtual** and **constexpr**.
[15] In C++17, this restriction is relaxed: For a **union** to be a literal type, only one, rather than all, of its non**static** data members needs to be of a non**volatile** literal type.

(c) Each base class is a literal type.

(d) There is some way to initialize an object of the type during constant evaluation; either (a) it is an **aggregate type**, (b) it can be braced initialized in a **constexpr** context, or (c) it has at least one **constexpr** constructor (possibly a template) that is not a *copy* or *move* constructor:

```
#include <string>  // std::string
struct LiteralUDT
{
    static std::string s_cache;
        // OK, static data member can have a nonliteral type.

    int d_datum;
        // OK, nonstatic data member of nonvolatile literal type

    constexpr LiteralUDT(int datum) : d_datum(datum) { }
        // OK, constructor is constexpr.

    // constexpr ~LiteralUDT() { }  // not permitted until C++20
        // no need to define; implicitly-generated destructor is trivial
};

struct LiteralAggregate
{
    int d_value1;
    int d_value2;
};

struct LiteralBraceInitializable
{
    int d_value1;
    int d_value2;
    LiteralBraceInitializable() = default; // trivial default constructor
    LiteralBraceInitializable(int v1, int v2)
        : d_value1(v1), d_value2(v2) { }   // not an aggregate
};

union LiteralUnion
{
    int   d_x;  // OK, int is a literal type.
    float d_y;  // OK, float is a literal type.
};
```

4. A cv-qualified literal type is also a literal type.[16]

[16] Note that cv-qualified scalar types are still scalar types, and cv-qualified **class** types were noted as being literal types in a defect report that resolved CWG issue 1951 (see **?**).

const int	is a *literal type*
volatile int	is a *literal type*
const volatile int	is a *literal type*
const LiteralUDT	is a *literal type* (since LiteralUDT is)

5. Arrays of objects of literal type are also literal types:

```
char a[5];  // An array of scalar type (e.g., char) is a literal type.

struct { int i; bool b; } b[7];
    // An array of aggregate type is a literal type.
```

6. In C++14 and thereafter, **void** (and thus cv-qualified **void**) is also a literal type, thereby enabling functions that return **void**:

```
constexpr const volatile void f() { }  // OK, in C++14
```

The overarching goal of this six-part definition of what constitutes a literal type is to capture those types that might be *eligible* to be created and destroyed during evaluation of a *constant expression*. This definition does not, however, guarantee that every literal type satisfying the above criteria will necessarily be constructible in a constant expression, let alone in a meaningful way.

- A user-defined literal type is not required to have any **constexpr** member functions or publicly accessible members. It is quite possible that the only thing one might be able to do with a user-defined literal type as part of a constant expression is to create it:

```
class C { };  // C is a valueless literal type.

int a[(C(), 5)];  // OK, create an array of five int objects.
```

  Such "barely literal" types — though severely limited in their usefulness in constant expressions — do allow for useful compile-time initialization of **constexpr** variables in C++14 (see Section 2.1."**constexpr** Variables" on page 282).

- The requirement to have at least one **constexpr** constructor that is not a *copy* or *move* constructor is just that — to have one. There is no requirement that such a constructor be invocable at compile time (e.g., it could be declared **private**) or even that it be defined; in fact, a deleted constructor (see Section 1.1."**??**" on page **??**) satisfies the requirement:

```
struct UselessLiteralType
{
    constexpr UselessLiteralType() = delete;
};
```

- Many uses of literal types in **constexpr** functions will require additional **constexpr** functions to be defined (i.e., not *deleted*), such as a *move* or *copy* constructor:

```
struct Lt  // literal type having nonconstexpr copy constructor
{
    constexpr Lt(int c) { }  // valid constexpr value constructor
    Lt(const Lt& ) { }       // nonconstexpr copy constructor
};

constexpr int processByValue(Lt t) { return 0; }  // valid constexpr function

static_assert(processByValue(Lt(7)) == 0, "");
    // Error, but OK (due to elided copy) on some platforms

constexpr Lt s{7};  // braced-initialized aggregate

static_assert(processByValue(s) == 0, "");  // Error, nonconstexpr copy ctor
```

In the code example above, we have an *aggregate* literal type, `Lt`, for which we have explicitly declared a non**`constexpr`** copy constructor. We then defined a valid **`constexpr`** function, `processByValue`, taking an `Lt` (by value) as its only argument. Invoking the function by constructing a object of `Lt` from a literal **`int`** value enables the compiler to elide the copy. Platforms where the copy is elided might allow this evaluation at compile time, while on others there will be an error. When we consider using an independently constructed **`constexpr`** variable, `s`, the `copy` can no longer be elided, and, since the copy constructor is declared explicitly to be non**`constexpr`**, the compile-time assertion fails to compile on all platforms; see Section 2.1."**`constexpr`** Variables" on page 282.

- Although a pointer or reference is always (by definition) a *literal type*, if the type being pointed to is not itself a literal type, then the referenced object cannot be used.

Identifying literal types

entifying-literal-types

Knowing what is and what is not a literal type is not always obvious (to say the least) given all the various rules we have covered. Having a concrete way *other* than becoming a language lawyer and interpreting the full standard definition can be immensely valuable during development, especially when trying to prototype a facility that you intend to be usable at compile time. In this subsection, we identify means for ensuring that a type is a literal type and, often more importantly for a user, identifying if a type is a ***usable* literal type**.

1. Only literal types can be used in the interface of a **`constexpr`** function (i.e., either as the return type or as a parameter type), and any literal type can be used in the interface of such a function. The first approach one might take to determine if a given type is a literal type would be to define a function that returns the given type *by value*. This has the downside of requiring that the type in question also be *copyable* (or at least *movable*, see Section 2.1."*rvalue* References" on page 479)[17]:

[17] As of C++17, the requirement that the type in question be *copyable* or *movable* to return it as a *prvalue* is removed; see Section 2.1."*rvalue* References" on page 479.

```
struct LiteralType    { constexpr LiteralType(int i)    {}         };
struct NonLiteralType {           NonLiteralType(int i) {}         };
struct NonMovableType { NonMovableType(NonMovableType&&) = delete; };

constexpr LiteralType    f(int i) { return LiteralType(i);    }  // OK
constexpr NonLiteralType g(int i) { return NonLiteralType(i); }  // Error
constexpr NonMovableType h()      { return NonMovableType{};  }  // Error
```

In the above example, `NonMovableType` is a literal type but is not movable (or copyable), so it cannot be the return type of a function. Passing the type as a *by-value* parameter works more reliably — and even reliably identifies non*copyable*, non*movable* literal types:

```
constexpr int test(LiteralType t)    { return 0; } // OK
constexpr int test(NonLiteralType t) { return 0; } // Error
constexpr int test(NonMovableType t) { return 0; } // OK
```

This approach is appealing in that it provides a general way for a programmer to query the compiler whether it considers a given type, `S`, *as a whole* to be a literal type and can be succinctly written[18]:

```
constexpr int test(S) { return 0; }  // compiles only if S is a literal type
```

Note that all of these tests require providing a function body, since compilers will validate that the declaration of the function is valid for a **constexpr** function only when they are processing the *definition* of the function. A declaration without a body will not produce the expected error for *non-literal-type* parameters and return types:

```
constexpr NonLiteralType quietly(NonLiteralType t);  // OK, declaration only
constexpr NonLiteralType quietly(NonLiteralType t) { return t; }  // Error
```

Finally, the C++11 Standard Library also provides a type trait — `std::is_literal_type` — that attempts to serve a similar purpose[19]:

```
#include <type_traits>  // std::is_literal_type
static_assert( std::is_literal_type<LiteralType>::value, "");     // OK
static_assert(!std::is_literal_type<NonLiteralType>::value, "");  // OK
```

Given the effectiveness of the simple **constexpr** interface test idiom illustrated in the code snippet above, any use of `std::is_literal_type` seems dubious.

The important take-away from this section is that we can use a trivial test in C++11 (made even more trivial in C++14) to find out if the compiler deems that a given type

[18] As of C++14, this utility could be written as `template<typename S> constexpr void test(S) { }`, returning **void** and omitting the **return** statement entirely.

[19] Note that the `std::is_literal_type` trait is deprecated in C++17 and removed in C++20. The rationale is stated in **?**:

> The `is_literal_type` trait offers negligible value to generic code, as what is really needed is the ability to know that a specific construction would produce constant initialization. The core term of a literal type having at least one **constexpr** constructor is too weak to be used meaningfully.

is a literal type. Much of the research done to understand and delineate this feature was made possible only through extensive use of this interface test idiom.

2. Often, simply *being* a literal type is not the only criterion that a developer is interested in. In order to ensure that a type under development is meaningful in a compile-time facility, it becomes imperative that we can confirm that objects of a given type can actually be constructed at compile-time. This confirmation requires identifying a particular form of initialization and corresponding **witness arguments** that should allow a user-defined type to assume a valid compile-time value. For this example, we can use the interface test to help prove that our class (e.g., `Lt`) is a literal type:

```
class Lt  // An object of this type be used in a constant expression.
{
    int d_value;

public:
    constexpr Lt(int i) : d_value(i != 75033 ? throw 0 : i) { }  // OK
};

constexpr int checkLiteral(Lt) { return 0; }  // OK, literal type
```

Proving that `Lt` (in the code example above) is a *usable* literal type next involves choosing a **`constexpr`** constructor (e.g., `Lt(int)`), selecting appropriate witness arguments (e.g., `75033`), and then using the result in a constant expression. The compiler will indicate (by producing an error) if our type cannot be constructed at compile time:

```
char a[(Lt(75033), 1)];               // OK, usable in constant expr
static_assert((Lt(75033), true), ""); // OK,    "    "     "       "
```

For types that are not *usable* literal types, there will be no such proof. When a particular constructor that is *explicitly* declared to be **`constexpr`** has no sequence of witness arguments that can be used to prove that the type is usable, the constructor (and any program in which it resides) is IFNDR. Forcing the compiler to perform such a proof in general — even if such were possible — would not be a wise use of compile-time compute resources. Hence, compilers will generally not diagnose the ill formed constructor and instead simply produce an error on each attempt to provide a set of **witnesses** for a literal type that fails to be usable at compile time:

```
int a = 1, b = 2;  // a and b are not constexpr.

class PathologicalType  // ill formed, no diagnostic required
{
    int d_value;

public:
    constexpr PathologicalType(int i)
        : d_value( (i <  2) ? a
                 : (i >= 2) ? b
                 : (i * 2) ) { }
};
```

The compiler is unlikely to have logic to discover that there is no way to invoke the above **constexpr** constructor as part of the evaluation of a constant expression; the constructor is considered ill formed, but no diagnostic is likely to be produced. Supplying any witness arguments, however, will force the compiler to evaluate the constructor and discover that no *particular* invocation is valid:

```
static_assert((PathologicalType(1),true), "");  // Error, a is not constexpr.
static_assert((PathologicalType(2),true), "");  // Error, b is not constexpr.
static_assert((PathologicalType(3),true), "");  // Error, b is not constexpr.
```

Compile-time evaluation

mpile-time-evaluation

All of the restrictions on the constructs that are valid in a **constexpr** function exist to enable the portable evaluation of such functions at compile time. Appreciating this motivation requires an understanding of compile-time calculations in general and constant expressions in particular.

First, a constant expression is *required* in very specific contexts:

- Any arguments to **static_assert**, **noexcept**-exception specifications, and the **alignas** specifier
- The size of a built-in array
- The expression for a **case** label in a **switch** statement
- The initializer for an enumerator
- The length of a bit field
- Nontype template parameters
- The initializer of a **constexpr** variable (see Section 2.1."**constexpr** Variables" on page 282)

Computing the value of expressions in these contexts requires that all of their subexpressions be known and evaluable at compile time, except those that are short-circuited by the logical *or* operator (||), the logical *and* operator (&&), and the *ternary* operator (?:):

```
constexpr int f(int x) { return x || (throw x, 1); }
constexpr int g(int x) { return x && (throw x, 1); }
constexpr int h(int x) { return x ? 1 : throw x; }

static_assert(f(true), "");    // OK, throw x is never evaluated.
static_assert(!g(false), "");  // OK,      "     "    "       "
static_assert(h(true), "");    // OK,      "     "    "       "
```

Note that the **controlling constant expression** for the preprocessor directives **#if** and **#elif**, while similar to general constant expressions, are computed at a time before any functions (**constexpr** or not) are even parsed, and all but a fixed set of predefined identifiers (e.g., `defined` and **true**) are replaced with macro-expansions or `0` before the resulting

arithmetic expression is evaluated. This small subset of other compile-time evaluation facilities in the language cannot invoke any user-defined functions. Consequently, **constexpr** functions cannot be invoked as part of the controlling constant expression for preprocessor directives.

Second, the Standard identifies a clear set of operations that are not available for use in constant expressions and, therefore, cannot be relied upon for compile-time evaluation:

- Throwing an exception.
- Invoking the **new** and **delete** operators.
- Invoking a lambda function.
- Any operation that depends on runtime polymorphism, such as **dynamic_cast**, **typeid** on a polymorphic type, or invoking a virtual function, which cannot be **constexpr**.
- Using **reinterpret_cast**.
- Any operation that modifies an object (increment, decrement, and assignment), including function parameters, member variables, and global variables.
- Any operation having *undefined behavior* such as integer overflow, dereferencing **nullptr**, or indexing outside the bounds of an array.
- Invoking a non-**constexpr** function or constructor, or a **constexpr** function whose definition has not yet been seen.

Note that being marked **constexpr** enables a function to be evaluated *at compile time* only if (1) the argument values are constant expressions known before the function is evaluated and (2) no operations performed when invoking the function with those arguments involve any of the excluded ones listed above.

Global variables can be used in a **constexpr** function only if they are (1) *non***volatile const** objects of *integral* or *enumerated* type that are initialized by a constant expression (generally treated as **constexpr** even if only marked as **const**), or (2) **constexpr** objects of literal type; see *Literal types (defined)* on page 260 and Section 2.1."**constexpr** Variables" on page 282. In either case, any **constexpr** global object used within a **constexpr** function must be initialized with a constant expression prior to the definition of the function. C++14[20] relaxes some of these restrictions (see Section 2.2."**constexpr** Functions '14" on page 595).

Use Cases

use-cases

A better alternative to function-like macros

to-function-like-macros

Computations that are useful both at run time and compile time and/or that must be inlined for performance reasons were typically implemented using preprocessor macros. For instance, consider the task of converting mebibytes to bytes:

[20] C++17 and C++20 each further relax these restrictions.

```
#define MEBIBYTES_TO_BYTES(mebibytes) (mebibytes) * 1024 * 1024
```

The macro above can be used in contexts where both a constant expression is required and the input is known only during program execution:

```
#include <vector>  // std::vector
void example0(std::size_t input)
{
    char fixedBuffer[MEBIBYTES_TO_BYTES(2)];  // compile-time constant

    std::vector<char> dynamicBuffer;
    dynamicBuffer.resize(MEBIBYTES_TO_BYTES(input));  // usable at run time
}
```

While a single-line macro with a reasonably unique (and long) name like `MEBIBYTES_TO_BYTES` is unlikely to cause any problems in practice, it harbors all the disadvantages macros have compared to regular functions. Macro names are not scoped; hence, they are subject to global name collisions. There is no well-defined input and output type and thus no type safety. Perhaps most tellingly, the lack of expression safety makes writing even simple macros tricky; a common error is to forget, e.g., the `()` around `mebibytes` in the implementation of `MEBIBYTES_TO_BYTES`, resulting in an unintended result if applied to a non-trivial expression such as `MEBIBYTES_TO_BYTES(2+2)` — yielding a value of `2 + 2 * 1024 * 1024 = 2097154` without the `()` and the intended value of `(2 + 2) * 1024 * 1024 = 4194304` with them. The generally unstructured and unhygienic nature of macros has led to significant language evolution aimed at supplanting their use with proper language features where practicable.[21]

A single **constexpr** function is sufficient to replace the `MEBIBYTES_TO_BYTES` macro, avoiding the aforementioned disadvantages without any additional runtime overhead:

```
constexpr std::size_t mebibytesToBytes(std::size_t mebibytes)
{
    return mebibytes * 1024 * 1024;
}

void example1(std::size_t input)
{
    char fixedBuffer[mebibytesToBytes(2)];
        // OK, guaranteed to be invocable at compile time

    std::vector<char> dynamicBuffer;
    dynamicBuffer.resize(mebibytesToBytes(input));
        // OK, can also be invoked at run time
}
```

time-string-traversal

Compile-time string traversal

Beyond simple numeric calculations, many compile-time libraries may need to accept strings as input and manipulate them in various ways. Applications can range from simply pre-

[21] We are not suggesting that macros have no place in the ecosystem; in fact, many of the language features developed for C++ — not the least of which were templates and, more recently, contract checks — were initially prototyped using preprocessor macros.

calculating string-related values to powerful compile-time regular-expression libraries.[22] To begin, we will consider the simplest of string operations: calculating the length of a string. An initial implementation might attempt to leverage the type of a string constant (array of **char**) with a template:

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

template <std::size_t N>
constexpr std::size_t constStrlenLit(const char (&lit)[N])
{
    return N - 1;
}
static_assert(constStrlenLit("hello") == 5, "");  // OK
```

This approach, however, fails when attempting to apply it to any number of other ways in which a variable might contain a compile-time or runtime string constant:

```
constexpr const char* hw1     = "hello";
char                  hw2[20] = "hello";
const char*           hw3     = hw2;

static_assert(constStrlenLit(hw1) == 5, "");  // Error, hw1 not a char[N]

std::size_t len2 = constStrlenLit(hw2);  // Bug, returns 19
std::size_t len3 = constStrlenLit(hw3);  // Error, hw3 not a char[N]
```

The type-based approach is clearly deficient. A better approach is simply to loop over the characters in the string, counting them until we find the terminating `\0` character. Here, we'll take the liberty of illustrating the simpler solution (using local variables and loops) that is available with the relaxed rules for **constexpr** functions in C++14 alongside the recursive solution that works in C++11; see Section 2.2."**constexpr** Functions '14" on page 595:

```
constexpr std::size_t constStrlen(const char* str)
{
#if __cplusplus > 201103L
    const char *strEnd = str;
    while (*strEnd) ++strEnd;
    return strEnd - str;
#else
    return (str[0] == '\0') ? 0 : 1 + constStrlen(str + 1);
#endif
}

static_assert(constStrlen("hello") == 5, "");  // OK
static_assert(constStrlen(hw1)     == 5, "");  // OK

std::size_t len2b = constStrlen(hw2);  // OK, returns 5
std::size_t len3b = constStrlen(hw3);  // OK, returns 5
```

[22] See **?** for one example of how far such techniques can evolve and might potentially be incorporated into a future Standard Library release.

With this most basic function implemented, let's move on to the more interesting problem of counting the number of lowercase letters in a string in such a way that it can be evaluated at compile time if the string is a constant expression. We'll need a simple helper function that determines if a given **char** is a lowercase letter:

```
constexpr bool isLowercase(char c)
    // Return true if c is a lowercase ASCII letter, and false otherwise.
{
    return 'a' <= c && c <= 'z';  // true if c is in ASCII range a to 'z'
}
```

Note that we are using a simplistic definition here that is designed to handle only the *ASCII* letters `a` through `z`; significantly more work would be required to handle other character sets or locales at compile time. Unfortunately, the `std::islower` function inherited from C is not **constexpr**.

Now we can apply a very similar construct to what we used for `constStrlen` to count the number of lowercase letters in a string:

```
constexpr std::size_t countLowercase(const char* str)
{
    return (str[0] == '\0') ? 0 : isLowercase(str[0]) + countLowercase(str + 1);
}
```

Now we have a function that will count the lowercase letters in a string at either compile time or run time for *any* null-terminated string:

```
static_assert(countLowercase("") == 0, "");
static_assert(countLowercase("HELLO, WORLD") == 0, "");
static_assert(countLowercase("Hello, World") == 8, "");

#include <cassert>  // standard C assert macro
void test1()
{
     const char *p1 = "";             assert(countLowercase(p1) == 0);
     const char *p2 = "HELLO, WORLD"; assert(countLowercase(p2) == 0);
     const char *p3 = "Hello, World"; assert(countLowercase(p3) == 8);
}
```

The first three invocations of `countLowercase`, in the code snippet above, illustrate that, when given a **constexpr** argument, it can compute the correct result at compile time. The other three invocations show that `countLowercase` can be invoked on non**constexpr** strings and compute the correct results at run time.

Even with the seemingly austere restrictions on C++11 **constexpr** function bodies, much of the power of the language is still available at compile time. For example, counting values in an array that match a function predicate can be converted to a **constexpr** function template in the same way we might do so with a runtime-only template:

```
template <typename T, typename F>
constexpr std::size_t countIf(T* arr, std::size_t len, const F& func)
{
    return (len==0) ? 0 : (func(arr[0]) ? 1 : 0) + countIf(arr+1,len-1,func);
}
```

In just one dense line, `countIf` recursively determines if the current length, `len`, is 0 and, if not, whether the first element satisfies the predicate, `func`. If so, `1` is added to the result of recursively invoking `countIf` for the rest of the elements in `arr`.

This `countIf` function template can now be used at compile time with a **`constexpr`** function pointer to produce a more modern-looking version of our `countLowercase` function:

```
constexpr std::size_t countLowercase(const char* str)
{
    return countIf(str, constStrlen(str), isLowercase);
}
```

Rather than a null-terminated array of **`char`**, we might want a more flexible string representation consisting of a **`class`** containing a **`const char*`** pointing to the start of a sequence of **`char`**s and a `std::size_t` holding the length of the sequence.[23] We can define such a class as a literal type even in C++11:

```
#include <stdexcept>  // std::out_of_range

class ConstStringView
{
    const char* d_string_p;  // address of the string supplied at construction
    std::size_t d_length;    // length  "  "     "       "     "       "
public:
    constexpr ConstStringView(const char* str)
    : d_string_p(str)
    , d_length(constStrlen(str)) {}

    constexpr ConstStringView(const char* str, std::size_t length)
    : d_string_p(str)
    , d_length(length) {}

    constexpr char operator[](std::size_t n) const
    {
        return n < d_length ? d_string_p[n] : throw std::out_of_range("");
    }

    constexpr const char *data() const { return d_string_p; }
    constexpr std::size_t length() const { return d_length; }

    constexpr const char* begin() const { return d_string_p; }
    constexpr const char* end()  const { return d_string_p + d_length; }
};
```

The `ConstStringView` class shown above provides some basic functionality to inspect and pass around the contents of a string constant at compile time. The implicitly declared **`constexpr`** copy constructor of this literal type allows us to overload our `countIf` function template and `countLowercase` function to take a `ConstStringView` by value:

```
template <typename F>
```

[23] C++17's `std::string_view` is an example of such string-related utility functionality.

```
constexpr std::size_t countIf(ConstStringView sv, const F& func)
{
    return countIf(sv.data(), sv.length(), func);
}

constexpr std::size_t countLowercase(ConstStringView sv)
{
    return countIf(sv, isLowercase);
}
```

Thanks to the implicit-conversion constructors, all of the earlier **static_assert** statements that were used with previous `countLowercase` implementations work with this one as well, and we gain the ability to further use `ConstStringView` as a **vocabulary type** for our **constexpr** functions.

-data-at-compile-time

Precomputing tables of data at compile time

Often, compile-time evaluation through the use of **constexpr** functions can be used to replace otherwise complex template metaprogramming or preprocessor tricks. While yielding more readable and more maintainable source code, **constexpr** functions also enable useful computations that previously were simply not practicable at compile time.

Calculating single values and using them at compile time is straightforward. Storing such values to use at run time can be done with a **constexpr** variable; see Section 2.1."**constexpr** Variables" on page 282. Calculating many values and then using them at run time benefits from other modern language features, in particular variadic templates (see Section 2.1."Variadic Templates" on page 519).

Consider a part of a date and time library that provides utilities to deal with modern timestamps of type `std::time_t` — an integer type expressing a number of seconds since some point in time, e.g., the **POSIX epoch**, midnight on January 1, 1970. An important tool in this library would be a function to determine the year of a given timestamp:

```
#include <ctime>  // std::time_t

int yearOfTimestamp(std::time_t timestamp);
    // Return the year of the specified timestamp. The behavior is undefined
    // if timestamp < 0.
```

Among other features, this library would provide a number of constants, both for its internal use as well as for direct client use. These could be implemented as enumerations, as integral constants at namespace scope, or as static members of a **struct** or **class**. Since we will be leveraging them within **constexpr** functions, we will also illustrate making use of constexpr variables here:

```
// constants defining the date and time of the epoch
constexpr int k_EPOCH_YEAR  = 1970;
constexpr int k_EPOCH_MONTH = 1;
constexpr int k_EPOCH_DAY   = 1;

// constants defining conversion ratios between various time units
constexpr std::time_t k_SECONDS_PER_MINUTE = 60;
```

```
constexpr std::time_t k_SECONDS_PER_HOUR   = 60  * k_SECONDS_PER_MINUTE;
constexpr std::time_t k_SECONDS_PER_DAY    = 24  * k_SECONDS_PER_HOUR;
constexpr std::time_t k_SECONDS_PER_YEAR   = 365 * k_SECONDS_PER_DAY;
static_assert( 31536000L == k_SECONDS_PER_YEAR, "");
```

For practical reasons related to the limits that compilers put on template expansion and **constexpr** expression evaluation, this library will support only a moderate number of future years:

```
// constant defining the largest year supported by our library
constexpr int k_MAX_YEAR = 2200;
```

To begin implementing `yearOfTimestamp`, it helps to start with an implementation of a solution to the reverse problem — calculating the timestamp of the start of each year, which requires an adjustment to account for leap days:

```
constexpr std::time_t numLeapYearsSinceEpoch(int year)
{
    return (year         / 4) - (year         / 100) + (year         / 400)
        - ((k_EPOCH_YEAR / 4) - (k_EPOCH_YEAR / 100) + (k_EPOCH_YEAR / 400));
}

constexpr std::time_t startOfYear(int year)
    // Return the number of seconds between the epoch and the start of the
    // specified year.  The behavior is undefined if year < 1970 or
    // year > k_MAX_YEAR.
{
    return (year - k_EPOCH_YEAR) * k_SECONDS_PER_YEAR
         + numLeapYearsSinceEpoch(year - 1) * k_SECONDS_PER_DAY;
}
```

Given these tools, we could implement `yearOfTimestamp` naively with a simple loop:

```
int yearOfTimestamp(std::time_t timestamp)
{
    int year = k_EPOCH_YEAR;
    for (; timestamp > startOfYear(year + 1); ++year) {}
    return year;
}
```

This implementation, however, has algorithmically poor performance. While a closed-form solution to this problem is certainly possible, for expository purposes we will consider how we might, at compile time, build a lookup table of the results of `startOfYear` so that `yearOfTimestamp` can be implemented as a **binary search** on that table.

Populating a built-in array at compile time is feasible by manually writing each initializer, but a decidedly better option is to generate the sequence of numbers we want as a `std:array` where all we need is to provide the **constexpr** function that will take an index and produce the value we want stored at that location within the array. We will start by implementing the pieces needed to make a generic **constexpr** function for initializing `std::array` instances with the results of a **function object** applied to each index:

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

template <typename T, std::size_t N, typename F>
constexpr std::array<T, N> generateArray(const F& func);
    // Return an array arr of size N such that arr[i] == func(i) for
    // each i in the half-open range [0, N).
```

The common idiom to do this initialization is to exploit a type that encodes indices as a variadic parameter pack (see Section 2.1."Variadic Templates" on page 519), along with the help of some using aliases (see Section 1.1."**using** Aliases" on page 121)[24]:

```
template <std::size_t...>
struct IndexSequence
{
    // This type serves as a compile-time container for the sequence of size_t
    // values that form its template parameter pack.
};

template <std::size_t N, std::size_t... Seq>
struct MakeSequenceHelper : public MakeSequenceHelper<N-1u, N-1u, Seq...>
{
    // This type is a metafunction to prepend a sequence of integers 0 to N-1
    // to the Seq... parameter pack by prepending N-1 to Seq... and
    // recursively instantiating itself.  The resulting integer sequence is
    // available in the type member inherited from the recursive instantiation.
    // The type member has type IndexSequence<FullSequence...>, where
    // FullSequence is the sequence of integers 0 .. N-1, Seq....
};

template <std::size_t ... Seq>
struct MakeSequenceHelper<0U, Seq...>
{
    // This partial specialization is the base case for the recursive
    // inheritance of MakeSequenceHelper.  The type member is an alias for
    // IndexSequence<Seq...>, where the Seq... parameter pack is typically
    // built up through recursive invocations of the MakeSequenceHelper
    // primary template.

    using type = IndexSequence<Seq...>;
};

template <std::size_t N>
using MakeIndexSequence = typename MakeSequenceHelper<N>::type;
    // alias for an IndexSequence<0 .. N-1> (or IndexSequence<> if N is 0)
```

To implement our array initializer, we will need another helper function that has, as a template argument, a variadic parameter pack of indices. To get this template param-

[24]This idiom was, in fact, so common that it is available in the Standard Library as `std::index_sequence` in C++14 (see **?**). Note that this solution is not lightweight, so the Standard Library types are generally implemented using compiler intrinsics that make them usable for significantly larger values.

eter pack `std::size_t... I` deduced, our function has an unnamed argument of type `IndexSequence<I...>`. With this parameter pack in hand, we can then use a simple pack expansion expression and braced initialization to populate our `std::array` return value:

```
template <typename T, std::size_t... I, typename F>
constexpr std::array<T, sizeof...(I)> generateArrayImpl(const F& func,
                                                         IndexSequence<I...>)
    // Return the results of calling F(i) for each i in the pack deduced as
    // the template parameter pack I.
{
    return { func(I)... };
}
```

The return statement in `generateArrayImpl` calls `func(I)` for each `I` in the range from 0 to the length of the returned `std::array`. The resulting pack of values is used to list initialize the return value of the function; see Section 2.1."Braced Init" on page 198.

Finally, our implementation of `generateArray` forwards `func` to `generateArrayImpl`, using `MakeIndexSequence` to generate an object of type `IndexSequence<0,...,N-1>`:

```
template <typename T, std::size_t N, typename F>
constexpr std::array<T, N> generateArray(const F& func)
{
    return generateArrayImpl<T>(func, MakeIndexSequence<N>());
}
```

With these tools in hand and a support function to offset the array index with the year, it is now simple to define an array that is initialized at compile time with an appropriate range of results from calls to `startOfYear`:

```
constexpr std::time_t startOfEpochYear(int epochYear)
{
    return startOfYear(k_EPOCH_YEAR + epochYear);
}

constexpr std::array<std::time_t, k_MAX_YEAR - k_EPOCH_YEAR> k_YEAR_STARTS =
    generateArray<std::time_t, k_MAX_YEAR - k_EPOCH_YEAR>(startOfEpochYear);

static_assert(k_YEAR_STARTS[0]  == startOfYear(1970),"");
static_assert(k_YEAR_STARTS[50] == startOfYear(2020),"");
```

With this table available for our use, the implementation of `yearOfTimestamp` becomes a simple application of `std::upper_bound` to perform a binary search on the sorted array of start-of-year timestamps[25]:

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

[25] Among other improvements to language and library support for **constexpr** programming, C++20 added **constexpr** to many of the standard algorithms in `<algorithm>`, including `std::upper_bound`, which would make switching this implementation to be **constexpr** also trivial. Implementing a **constexpr** version of most algorithms in C++14 is, however, relatively simple (and in C++11 is still possible), so, given a need, providing **constexpr** versions of functions like this with less support from the Standard Library is straightforward.

```
int yearOfTimestamp(std::time_t timestamp)
{
    std::size_t ndx = std::upper_bound(k_YEAR_STARTS.begin(),
                                       k_YEAR_STARTS.end(),
                                       timestamp)
                    - k_YEAR_STARTS.begin();
    return k_EPOCH_YEAR + ndx - 1;
}
```

When implementing a library of this sort, carefully making key decisions, such as whether to place the **constexpr** calculations in a header or to insulate them in an implementation file, is important; see *Potential Pitfalls — Overzealous use hurts* on page 278. When building tables such as this, it's also worth considering more classical alternatives, such as simply generating code using an external script. Such external approaches can yield significant reductions in compile time and improved insulation; see *Potential Pitfalls — One time is cheaper than compile time or run time* on page 279.

itfalls-constexprfunc

Potential Pitfalls

mpile-time-evaluation

Low compiler limits on compile-time evaluation

A major restriction on compile-time evaluation, beyond the linguistic restrictions already discussed, is the set of **implementation-defined** limitations specific to the compiler. In particular, the Standard allows implementations to limit the following:

- *Maximum number of recursively nested* **constexpr** *function invocations*: The expected value for this limit is 512, and in practice that is the default for most implementations. While 512 may seem like a large call depth, C++11 **constexpr** functions must use recursion instead of iteration, making it easy to exceed this limit when attempting to do involved computations at compile time.
- *Maximum number of subexpressions evaluated within a single constant expression*: The suggested value for this limit as well as the default value for most implementations is 1,048,576, but it is important to note that this value can depend in surprising ways on the way that the number of subexpressions is calculated by each individual compiler. Expressions that stay within the limit reliably with one compiler might be counted differently in the constant expression evaluator of a different compiler, resulting in nonportable code. Compilers generally refer to this limit as the number of **constexpr** steps and do not always support adjusting it.

Though these limits can usually be increased with compiler flags, a significant overhead is introduced in terms of managing build options that can hinder how easily usable a library intended to be portable will be. Small differences in terms of how each compiler might count these values also hinder the ability to write portable **constexpr** code.

g-constexpr-functions

Difficulty implementing constexpr functions

Many algorithms are simple to express iteratively and/or implement efficiently using dynamic data structures outside of what is possible within a **constexpr** function. Naive (and

even non-naive) implementations often exceed the often wide-ranging limits that various compilers put on **constexpr** evaluation. Consider this straightforward implementation of `isPrime`:

```
template <typename T>
constexpr bool isPrime(T input)
{
    if (input < 2) return false;                // too small
    if (input == 2) return true;                // is two
    if (input % 2 == 0) return false;           // is even
    for (T i = 3; i <= input / i; i += 2)       // odd numbers up to square root
    {
        if (input % i == 0) { return false; }   // found divisor?
    }
    return true;                                // no divisors, input is prime
}
```

This implementation is iterative and fails to meet the requirements for being a C++11 **constexpr** function. While meeting the relaxed requirements for being a C++14 **constexpr** function (see Section 2.2."**constexpr** Functions '14" on page 595), it is likely to hit default compiler limits on execution steps when `input` approaches 2^{40}.

To make this **constexpr** `isPrime` function implementation valid for C++11, we might start by switching to a recursive implementation for the same algorithm:

```
template <typename T>
constexpr bool isPrimeHelper(T n, T i)
{
    return n % i                            // i is not a divisor.
        && (   (i > n/i)                    // i is not larger than sqrt(n).
            || isPrimeHelper(n, i + 2));    // tail recursion on next i
}

template <typename T>
constexpr bool isPrime(T input)
{
    return input < 2                  ? false  // too small
         : (input == 2 || input == 3) ? true   // 2 or 3
         : (input % 2 == 0)           ? false  // odd
         : isPrimeHelper(input, 3);            // Call recursive helper.
}
```

The recursive implementation above works correctly, albeit slowly, up to an input value of around 2^{19}, hitting recursion limits on **constexpr** evaluation on most compilers. With significant effort, we might conceivably be able to push the upper limit slightly higher (e.g., by prechecking more factors than just 2). More importantly, recursively checking every other number below the square root of the input for divisibility is so slow compared to better algorithms that this approach is fundamentally inferior to a runtime solution.[26]

[26]The C++20 Standard adds `std::is_constant_evaluated()`, a tool to allow a function to branch to different implementations at compile time and run time, enabling the compile-time algorithm to be different from the runtime algorithm with the same API.

A final approach to working around the **constexpr** recursion limit is to implement a **divide and conquer** algorithm when searching the space of possible factors. While this approach has the same algorithmic performance as the directly recursive implementation and executes a comparable number of steps, the maximum recursion depth it needs is logarithmic in terms of the input value and will stay within the general compiler limits on recursion depth:

```
template <typename T>
constexpr bool hasFactor(T n, T begin, T end)
    // Return true if the specified n has a factor in the
    // closed range [begin, end], and false otherwise.
{
    return (begin > end)     ? false             // empty range [begin, end]
         : (begin > n/begin) ? false             // begin > sqrt(n)
         : (begin == end)    ? (n % begin == 0)  // [begin, end] has one element.
         :   // Otherwise, split into two ranges and recurse.
             hasFactor(n, begin, begin + (end - begin) / 2) ||
             hasFactor(n, begin + 1 + (end - begin) / 2, end);
}

template <typename T>
constexpr bool isPrime(T input)
    // Return true if the specified input is prime.
{
    return input < 2                 ? false  // too small
         : (input == 2 || input == 3) ? true   // 2 or 3
         : (input % 2 == 0)           ? false  // odd
         : !hasFactor(input, static_cast<T>(3), input - 1);
}
```

This C++11 implementation will generally work up to the same limits as the iterative C++14 implementation in the example above.

Prematurely committing to `constexpr`

mmitting-to-constexpr

Declaring a function to be **constexpr** comes with significant collateral costs that, to some, might not be obvious. Marking an eligible function **constexpr** would seem like a sure way to get compile-time evaluation, when possible (i.e., when constant expressions are passed into a function as parameters), without any additional cost for functions that currently meet the requirements of a **constexpr** function — essentially giving us a "free" runtime performance boost. The often-overlooked downside, however, is that this choice, once made, is not easily reversed. After a library is released and a **constexpr** function is evaluated as part of a constant expression, no clean way of turning back is available because clients now depend on this compile-time property.

Overzealous use hurts

overzealous-use-hurts

Overzealous application of **constexpr** can also have a significant impact on compilation time. Compile-time calculations can easily add seconds — or in extreme cases much more

— to the compilation time of any translation unit that needs to evaluate them. When placed in a header file, these calculations need to be performed for all translation units that include that header file, vastly increasing total compilation time and hindering developer productivity. Compile-time precomputation *might* improve runtime performance in some cases but comes with other kinds of cost, which can be formidable.

Similarly, making public APIs that are **constexpr** usable without making it clear that they are suboptimal implementations can lead to both (1) excessive runtime overhead compared to a highly optimized non**constexpr** implementation (e.g., for `isPrime` in Difficulty implementing **constexpr** functions) that might already exist in an organization's libraries and (2) increased compile time wherever algorithmically complex **constexpr** functions are invoked.

Compilation limits on compile-time evaluation are typically per *constant expression* and can easily be compounded unreasonably within just a single translation unit through the evaluation of numerous constant expressions. For example, when using the `generateArray` function in *Potential Pitfalls* — **??** on page **??**, [**AUs, there is no subsection called "Moving runtime calculation overhead to compile time." What did you mean?**] compile-time limits apply to each individual array element's computation, allowing total compilation to grow linearly with the number of values requested.

ompile-time-or-run-time

One time is cheaper than compile time or run time

Overall, the ability to use a **constexpr** function to do calculations before run time fills in a spectrum of possibilities for who pays for certain calculations and when they pay for them, both in terms of computing time and maintenance costs.

Consider a possible set of evolutionary steps for a computationally expensive function that produces output values for a moderate number of unique input values. Examples include returning the timestamp for the start of a calendar year or returning the nth prime number up to some maximum n.

1. An initial version directly computes the output value each time it is needed. While correct and written entirely in maintainable C++, this version has the highest runtime overhead. Heavy use will quickly lead the developer to explore optimizations.

2. Where precomputing values might seem beneficial, a subsequent version initializes an array once at run time to avoid the extra computations. Aggregate runtime performance can be greatly improved but at the cost of slightly more code as well as a possibly noteworthy amount of runtime startup overhead. This hit at startup or on first use of the library can quickly become the next performance bottleneck that needs tackling. Initialization at startup can become increasingly problematic when linking large applications with a multitude of libraries, each of which might have moderate initialization times.

3. At this point **constexpr** comes into play as a tool to develop an option that avoids as much runtime overhead as possible. An initial such implementation puts the initialization of a **constexpr** array of values into the corresponding **inline** implementation in a library header. While this option minimizes the runtime overhead, the compile-time

overhead now becomes significantly larger for every translation unit that depends on this library.

4. When faced with crippling compile times, the likely next step is to **insulate** the compile-time-generated table in an implementation file and to provide runtime access to it through accessor functions. While this removes the compilation overhead from clients who consume a binary distribution of the library, anyone who needs to build the library is still paying this cost each time they do a clean build. In modern environments, with widely disparate operating systems and build toolchains, source distributions have become much more common and this overhead is imposed on a wide range of clients for a popular library.

5. Finally, the data table generation is moved into a separate program, often written in Python or some other non-C++ language. The output of this outboard program is then embedded as raw data (e.g., a sequence of numbers initializing an array) in the C++ implementation file. This solution eliminates the compile-time overhead for the C++ program; the cost of computing the table is paid only once by the developer. On the one hand, this solution adds to the maintenance costs for the initial developer, since a separate toolchain is often needed. On the other hand, the code becomes simpler, since the programmer is free to choose the best language for the job and is free from the constraints of **constexpr** in C++.

Thus, as attractive as it might seem to be able to precompute values directly in compile-time C++, complex situations often dictate against that choice. Note that a programmer with this knowledge might skip all of the intermediate steps and jump straight to the last one. For example, a list of prime numbers is readily available on the Internet without needing even to write a script; a programmer need only cut and paste it once, knowing that it will *never* change.

Annoyances

annoyances

Penalizing run time to enable compile time

o-enable-compile-time

When adopting **constexpr** functions, programmers commonly forget that these functions are also called at run time, often more frequently than at compile time. Restrictions on the operations that are supported in a **constexpr** function definition, especially prior to the looser restrictions of C++14, will often lead to correct results that are less than optimally computed when executed at run time. A good example would be a **constexpr** implementation of the C function `strcmp`. Writing a recursive **constexpr** function to walk through two strings and return a result for the first characters that differ is relatively easy. However, most common implementations of this function are highly optimized, often taking advantage of inline assembly using architecture-specific vector instructions to handle multiple characters per CPU clock cycle. All of that fine tuning is given up if we rewrite the function to be **constexpr** compatible. Worse yet, the recursive nature of such functions prior to C++14 leads to a much greater risk of exceeding the limits of the stack, leading to program corruption and security risks when comparing long strings.

One possible workaround for these restrictions is to create different versions of the same function: a **constexpr** version usable at compile time and a non**constexpr** version opti-

mized for run time. Since the language does not support overloading on **constexpr**, the end result is intrusive, requiring the different implementations to have different names. This complication can be mitigated by having a coding convention, such as placing all **constexpr** overloads in a namespace, say, `cexpr`, or giving all such functions the `_c` suffix. If the regular version of the function is also **constexpr**, the marked overload can simply forward all arguments to the regular function to ease maintenance, but overhead and complexity still come from having users manage multiple versions of the same function. It is not clear in all cases that the extra complexity covers its cost.

The relaxed restrictions in C++14 for implementing the bodies of **constexpr** functions is a welcome relief when optimizing for compile time and run time simultaneously; see Section 2.2."**constexpr** Functions '14" on page 595. Even then, though, many runtime performance improvements (e.g., dynamic memory allocation, stateful caching, hardware intrinsics) are still not available to functions that need to execute at both run time and compile time. Note that a language-based solution that avoids the need to create separately named **constexpr** and non**constexpr** functions is introduced in C++20 with the `std::is_constant_evaluated()` intrinsic library function.

-qualified-(c++11-only)

`constexpr` member functions are implicitly `const`-qualified (C++11 only)

A design flaw in C++11 (corrected in C++14) is that any member function declared **constexpr** is, where applicable, implicitly **const**-qualified, leading to unexpected behavior for member functions intended for use in a non**constexpr** context; see Section 2.2."**constexpr** Functions '14" on page 595. This surprising restriction impacts code portability between language standards and makes the naive approach of just marking all member functions **constexpr** into something that unwittingly breaks what would otherwise be working functions.

see-also

See Also

- "**constexpr** Variables" (§2.1, p. 282) ♦ is the companion use of the **constexpr** keyword applied to variables.
- "Variadic Templates" (§2.1, p. 519) ♦ are often needed for complex metaprogramming used in some compile-time computations.
- "**constexpr** Functions '14" (§2.2, p. 595) ♦ enumerates the significantly richer syntax permitted for implementing **constexpr** function bodies in C++14.

further-reading

Further Reading

None so far.

Compile-Time Accessible Variables

constexprvar

A variable or **variable template** of **literal type** may be declared to be **constexpr**, ensuring its successful construction and enabling its use *at compile-time*.

Description

description

Variables of *all* built-in types and certain user-defined types, collectively known as literal types, may be declared **constexpr**, allowing them to be initialized *at compile-time* and subsequently used in **constant expressions**:

```
std::array

      int i0 = 5;
 const int i1 = 5;              // i1 is a compile-time constant.
constexpr int i2 = 5;           // i2  " "    "      "     "

 const double d1 = 5.0;
constexpr double d2 = 5.0;      // d2 is a compile-time constant.

      const char *s1 = "help";
constexpr const char *s2 = "help";  // s2 is a compile-time constant.
```

Although **const** integers initialized in the view of the compiler can be used within constant expressions — e.g., as the first argument to **static_assert**, as the size of an array, or as a non-type template parameter — such is not the case for any other types:

```
static_assert(i0 == 5, "");         // Error, i0 is not a compile-time constant.
static_assert(i1 == 5, "");         // OK, const is "magical" for integers (only).
static_assert(i2 == 5, "");         // OK

static_assert(d1 == 5, "");         // Error, d1 is not a compile-time constant.
static_assert(d2 == 5, "");         // OK

static_assert(s1[1] == 'e', "");    // Error, s1 is not a compile-time constant.
static_assert(s2[1] == 'e', "");    // OK, the ASCII code for e is decimal 101.

int a[s2[1]];                       // OK, defines an array, a, of 101 integers
static_assert(sizeof a == 404, ""); // OK, we are assuming the size of int is 4.

std::array<int, s1[1]> f;           // Error, s1 is not constexpr.
std::array<int, s2[1]> e;           // OK, defines a std::array, e, of 101 integers
```

Prior to C++11, the types of variables usable in a constant expression were quite limited:

```
const int b;         // Error, const scalar variable must be initialized.
extern const int c;  // OK, declaration
const int d = c;     // OK, not constant initialized (c initializer not seen)

int ca1[c];          // Error, c initializer not visible
int da1[d];          // Error, d initializer not constant
```

```
const int c = 7;
int ca2[c];          // OK, initializer is visible
int da2[d];          // Error, d initializer (still) not constant

const int e = 17;    // OK
int ea[e];           // OK
```

For an **integral constant** to be usable at compile time (e.g., as part of constant expression), certain requirements must be satisfied:

1. The variable must be marked **const**.
2. The initializer for a variable must have been seen by the time it is used, and it must be a constant expression — this is the information needed for a compiler to be able to make use of the variable in other constant expressions.
3. The variable must be of *integral* type, e.g., **bool**, **char**, **short**, **int**, **long**, **long long**, as well as the **unsigned** variations on these and any additional **char** types; see also Section 1.1."**long long**" on page 78.

This restriction to integral types provides support for those values where compile-time constants are most frequently needed while limiting the complexity of what compilers were required to support at compile time.

Use of **constexpr** when declaring a variable (or variable template; see Section 1.2."Variable Templates" on page 146) enables a much richer category of types to participate in constant expressions. This generalization, however, was not made for mere **const** variables because they are not *required* to be initialized by compile-time constants:

```
int f() { return 0; }  // f() is not a compile-time constant expression.

               int x0 = f();  // OK
         const int x1 = f();  // OK, but x1 is not a compile-time constant.
constexpr      int x2 = f();  // Error, f() "    "   "     "     "     "
constexpr const int x3 = f();  // Error, f() "    "   "     "     "     "
```

As the example code above demonstrates, variables marked **constexpr** *must* satisfy the same requirements needed for integral constants to be usable as constant expression. Unlike other integral constants, their initializers *must* be constant expressions or else the program is **ill formed**.

For a variable of other than **const** integral type to be usable in a constant expression, certain criteria must hold:

1. The variable must be annotated with **constexpr**, which implicitly also declares the variable to be **const**[1]:

```
struct S  // simple (aggregate) literal type
```

[1] C++20 added the **constinit** keyword to identify a variable that is initialized at compile time (with a constant expression) but may then be modified at runtime.

```
{
    int i;  // built-in integer data member
};

void test1()
{
    constexpr S s{1};       // OK, literal type constant expression initialized
    s = S();                // Error, constexpr implies const.
    constexpr int j = s.i;  // OK, subobjects are usable in constant expres-
sions.
}
```

In the example above, we have, for expedience of exposition, used brace initialization to initialize the aggregate; see Section 2.1."Braced Init" on page 198. Note that subobjects of **constexpr** objects are also effectively **constexpr** and can be used freely in constant expressions even though they themselves might not be explicitly declared **constexpr**.

2. All **constexpr** variables must be initialized with a constant expression when they are defined. Hence, every **constexpr** variable has an initializer, and that initializer must be a valid constant expression (see Section 2.1."**constexpr** Functions" on page 239):

```
          int g() { return 17; }  // a non-constexpr function
constexpr int h() { return 34; }  // a constexpr function

constexpr int v1;        // Error, v1 is not initialized.
constexpr int v2 = 17;   // OK
constexpr int v3 = g();  // Error, g() is not constexpr.
constexpr int v4 = h();  // OK

void test2(int c)
{
    constexpr int v5 = c;          // Error, c not a compile-time constant.
    constexpr int v6 = sizeof(c);  // OK, c is not evaluated.
}
```

3. Any variable declared **constexpr** must be of literal type; all literal types are, among other things, **trivially destructible**:

```
struct Lt  // literal type
{
    constexpr  Lt()  { }  // constexpr constructor
    ~Lt() = default;      // default trivial destructor
};

constexpr Lt lt;  // OK, Lt is a literal type.

struct Nlt  // nonliteral type.
{
    Nlt()  { }  // cannot initialize at compile-time
    ~Nlt() { }  // cannot skip non-trivial destruction
```

```
};

constexpr Nlt nlt;  // Error, Nlt is not a literal type.
```

Since all literal types are trivially destructible, the compiler does not need to emit any special code to manage the end of the lifetime of a **constexpr** variable, which can essentially live "forever" — i.e., until the program exits.[2]

4. Unlike integral constants, non**static** data members cannot be **constexpr**. Only variables at global or **namespace** scope, automatic variables, or **static** data members of a **class** or **struct** may be declared **constexpr**. Consequently, any given **constexpr** variable is a top-level object, never a subobject of another, possibly non**constexpr**, object:

```
                constexpr int i = 17;    // OK, file scope
namespace ns { constexpr int j = 34; }  // OK, namespace scope

struct C
{
    static constexpr int k = 51;  // OK, static data member
           constexpr int l = 68;  // Error, constexpr nonstatic data member
};

void g()
{
    static constexpr int m = 85;  // OK
           constexpr int n = 92;  // OK
};
```

Recall, however, that **nonstatic data members** of **constexpr** objects are implicitly **constexpr** and therefore can be used directly in any constant expressions:

```
constexpr struct D { int i; } x{1};  // brace-initialized aggregate x
constexpr int k = x.i;  // Subobjects of constexpr objects are constexpr.
```

Initializer Undefined Behavior

It is important to note the significance of one of the differences between a **constexpr** integral variable and a **const** integral variable. Because the initializer of a **constexpr** variable is *required* to be a constant expression, it is not subject to the possibility of undefined behavior (e.g., integer overflow or out-of-bounds array access) at run time and will instead result in a compile-time error:

```
const int iA = 1 << 15;  // 2^15 = 32,768 fits in 2 bytes.
const int jA = iA * iA;  // OK
```

[2] In C++20, literal types can have non-trivial destructors, and the destructors for **constexpr** variables will be invoked under the same conditions that a destructor would be invoked for a non**constexpr** global or **static** variable.

```
          const int iB = 1 << 16;  // 2^16 = 65,536 doesn't fit in 2 bytes.
          const int jB = iB * iB;  // Bug, overflow (might warn)

constexpr const int iC = 1 << 16;
constexpr const int jC = iC * iC;  // Error, overflow in constant expression

constexpr       int iD = 1 << 16;  // Example D is the same as C, above.
constexpr       int jD = iD * iD;  // Error, overflow in constant expression
```

The code example above shows that an integer constant-expression overflow, absent **constexpr**, is not required by the C++ Standard to be treated as ill formed, whereas the initializer for a **constexpr** expression is required to report overflow as an error (not just a warning).

There is a strong association between **constexpr** variable and functions; see Section 2.1."**constexpr** Functions" on page 239. Using a **constexpr** variable rather than just a **const** one forces the compiler to detect overflow within the body of **constexpr** function and report it — as a compile-time error — in a way that it would not otherwise be authorized to do.

For example, suppose we have two similar functions, `squareA` and `square`, defined for the built-in type **int** that each return the integral product of multiplying the argument with itself:

```
          int squareA(int i) { return i * i; }  // non-constexpr function
constexpr int squareB(int i) { return i * i; }  // constexpr function
```

Declaring a variable to be just **const** does nothing to force the compiler to check the evaluation of either function for overflow:

```
                int xA0 = squareA(1 << 15);  // OK
          const int xA1 = squareA(1 << 15);  // OK
constexpr       int xA2 = squareA(1 << 15);  // Error, squareA, not constexpr

                int yB0 = squareB(1 << 15);  // OK
          const int yB1 = squareB(1 << 15);  // OK
constexpr       int yB2 = squareB(1 << 15);  // OK

                int zC0 = squareB(1 << 16);  // Bug, zC0 is likely 0.
          const int zC1 = squareB(1 << 16);  // Bug, zC1  "   "    "
constexpr       int zC2 = squareB(1 << 16);  // Error, int overflow detected!
```

By declaring a variable to be not just **const** but **constexpr**, we make the compiler evaluate that (necessarily **constexpr**) function for those specific arguments at compile time and, if there is overflow, immediately report the defect as being ill formed.

Internal Linkage

When a variable at file or namespace scope is either **const** or **constexpr** and nothing explicitly gives it external linkage (e.g., being marked **extern**), it will have **internal linkage**. This means each translation unit will have its own copy of the variable.[3]

[3] In C++17, all **constexpr** variables are instead automatically **inline** as well, guaranteeing that there is only one instance in a program.

In general, only the values of such variables are relevant: their initializers are seen, they are used at compile time, and there is no effect if different translation units use different objects having the same name and a different value. Often, after compile-time evaluation is completed, the variables themselves will no longer be needed, and no actual address will be allocated for them at run time. Only in cases where the address of the variable is used will this difference be observable.

Notably, **static** member variables do not get internal linkage. Because of this, if they are going to be used in a way that requires they have an address allocated at run time, they need to have a definition outside of their class; see *Annoyances — **static** member variables require external definitions* on page 293.

use-cases

Use Cases

time-integral-constants

Alternative to enumerated compile-time integral constants

It is not uncommon to want to express specific integral constants at compile time — e.g., for precomputed operands to be used in algorithms, mathematical constants, configuration variables, or any number of other reasons. A naive, brute-force approach might be to hard-code the constants where they are used:

```
int hoursToSeconds0(int hours)
    // Return the number of seconds in the specified hours.  The behavior is
    // undefined unless the result can be represented as an int.
{
    return hours * 3600;
}
```

This use of *magic constants* has, however, long been known[4] to make finding uses of the constants and the relationships between related ones needlessly difficult. For integral values only, we could always represent such compile-time constants symbolically by using a classic **enum** type, in deliberate preference to the modern, type-safe enumerator; see Section 2.1."**enum class**" on page 310:

```
struct TimeRatios1  // explicit scope for single classic anonymous enum type
{
    enum  // anonymous enumeration comprising related symbolic constants
    {
        k_SECONDS_PER_MINUTE = 60,     // Underlying type (UT) might be int.
        k_MINUTES_PER_HOUR   = 60,
        k_SECONDS_PER_HOUR   = 60*60,  // these enumerators have the same UT
    };
};

int hoursToSeconds1(int hours)
    // ...
{
    return hours * TimeRatios1::k_SECONDS_PER_HOUR;
}
```

[4] **?**, section 1.5, pp. 19–22

This traditional solution, while often effective, gives little control to the underlying integral type of the enumerator used to represent the symbolic compile-time constant, leaving it at the mercy of the totality of values used to initialize its members. Such inflexibility might lead to compiler warnings and nonintuitive behavior resulting from **enum**-specific "integral-promotion" rules, especially when the **underlying type (UT)** used to represent the time ratios differs from the integral type with which they are ultimately used; see Section 2.1."Underlying Type '11" on page 480.

In this particular example, extending the **enum** to cover ratios up to a week and conversions down to nanoseconds would manifestly change its **UT** (because there are far more than 2^{32} nanoseconds in a week), altering how all of the enumerators behave when used in expressions with, say, values of type **int** (e.g., to **long**); see Section 1.1."**long long**" on page 78:

```
struct TimeRatios2  // explicit scope for single classic anonymous enum type
{
    enum  // Anonymous enumeration --- UT is governed by all of the enumerators.
    {
        k_SECONDS_PER_MINUTE = 60,    // UT might be long (or long long).
        k_MINUTES_PER_HOUR   = 60,
        k_SECONDS_PER_HOUR   = 3600,
        // ...
        k_USEC_PER_WEEK = 1000L*1000*60*60*24*7,  // same UT as all of the above
    };
};
```

The original *values* will remain unchanged, but the burden of all of the warnings resulting from the change in UT and rippling throughout a large codebase could be expensive to repair.

We would like the original values to remain unchanged (e.g., remain as **int** if that's what they were), and we want only those values that do *not* fit in an **int** to morph into a larger integral type. We might achieve this effect by placing each enumerator in its own separate anonymous enumeration:

```
struct TimeRatios3  // explicit scope for multiple classic anonymous enum types
{
    enum { k_SECONDS_PER_MINUTE = 60               };  // UT: int (likely)
    enum { k_MINUTES_PER_HOUR   = 60               };  //  "    "      "
    enum { k_SECONDS_PER_HOUR   = 60*60            };  //  "    "      "
    // ...
    enum { k_USEC_PER_SEC  = 1000*1000             };  // UT: int (v. likely)
    enum { k_USEC_PER_MIN  = 1000*1000*60          };  // UT: int (v. likely)
    enum { k_USEC_PER_HOUR = 1000U*1000*60*60      };  // UT: unsigned int
    enum { k_USEC_PER_DAY  = 1000L*1000*60*60*24   };  // UT: long
    enum { k_USEC_PER_WEEK = 1000L*1000*60*60*24*7 };  // UT: long
};
```

In this case, the original values as well as their respective UTs will remain unchanged, and each new enumerated value will independently take on its own independent UT, which is either implementation defined or else dictated by the number of bits required to represent the value, which is, in this case, non-negative.

A modern alternative to having separate anonymous **enum**s for each distinct value (or class of values) is to instead encode each ratio as an explicitly typed **constexpr** variable:

```
struct TimeRatios4
{
    static constexpr int k_SECONDS_PER_MINUTE = 60;
    static constexpr int k_MINUTES_PER_HOUR   = 60;
    static constexpr int k_SECONDS_PER_HOUR   = k_MINUTES_PER_HOUR *
                                                k_SECONDS_PER_MINUTE;
    // ...
    static constexpr long k_NANOS_PER_SECOND  = 1000*1000*1000;
    static constexpr long k_NANOS_PER_HOUR    = k_NANOS_PER_SECOND *
                                                k_SECONDS_PER_HOUR;
};

int hoursToSeconds(int hours)
    // ...
{
    return hours * TimeRatios4::k_SECONDS_PER_HOUR;
}

long hoursToNanos(int hours)
    // Return the number of nanoseconds in the specified hours.  The behavior
    // is undefined unless the result can be represented as a long.
{
    return hours * TimeRatios4::k_NANOS_PER_HOUR;
}
```

In the example above, we've rendered the **constexpr** variables as **static** members of a **struct** rather than placing them at namespace scope primarily to show that, from a user perspective, the two are syntactically indistinguishable — the substantive difference here being that a client would be prevented from unilaterally adding logical content to the "namespace" of a `TimeRatio` **struct**. When it comes to C-style **free functions**, however, the advantages for **static** members of a struct over namespace scope are many and unequivocal.[5]

Nonintegral symbolic numeric constants

olic-numeric-constants

Not all symbolic numeric constants that are needed at compile time are necessarily integral. Consider, for example, the mathematical constants `pi` and `e`, which are typically represented as a floating-point type, such as **double** (or **long double**).

The classical solution to avoid encoding this type of constant values as a *magic number* is to instead use a macro, such as is done in the `math.h` header on most operating systems:

```
#define M_E   2.7182818284590452354  /* e */
#define M_PI  3.14159265358979323846 /* pi */

double areaOfCircle(double radius)
{
```

[5] **?**, section 2.4.9, pp. 312–321, specifically Figure 2-23

```
    return 2 * M_PI * radius;
}
```

While this approach can be effective, it comes with all the well-known downsides of using the C preprocessor. This approach is also fraught with risk such that the headers standard on many systems make these macros available in C or C++ only upon request by **#define**ing specific macros before including those headers.[6]

Another safer and far less error-prone solution to name collisions is to instead use a **constexpr** variable for this form of nonintegral constant. Note that, while the macro is defined with more precision to be able to initialize variables of possibly higher-precision floating-point types, here we need only enough digits to uniquely identify the appropriate **double** constant:

```
struct NumericConstants
{
    static constexpr double k_E  = 2.718'281'828'459'045;  /* e */
    static constexpr double k_PI = 3.141'592'653'589'793;  /* pi */
};

double areaOfCircle(double radius)
{
    return 2 * NumericConstants::k_PI * radius;
}
```

In the example above, we have made valuable use of a safe C++14 feature to help identify the needed precision of the numeric literal; see Section 1.2."Digit Separators" on page 141. Beyond the potential name collisions and global name pollution, preferring a constexpr variable over a C preprocessor macro has the added benefit of making explicit the C++ type of constant being defined. Note that supplying digits beyond what are significant will nonetheless be silently ignored.

texpr-data-structures

Storing constexpr data structures

Precomputing values at compile time for subsequent use at run time is one impactful use of **constexpr** *functions*, but see *Potential Pitfalls — **constexpr** function pitfalls* on page 293. Storing these values in explicitly **constexpr** variables ensures that the values are (1) guaranteed to be computed at compile time and not, for example, at startup as the result of a dangerous runtime initialization of a file- or namespace-scoped variable and (2) usable as part of the evaluation of any constant expression; see Section 2.1."**constexpr** Functions" on page 239.

Rather than attempting to circumvent the draconian limitations of the C++11 version of **constexpr** functions, we will make use of the relaxed restrictions of C++14. For this, we will define a class template that initializes an array member with the results of a **constexpr** functor applied to each array index[7]:

[6] See your library documentation for details.

[7] Note that, in C++17, most of the manipulators of `std::array` have been changed to be **constexpr** and, when combined with the relaxation of the rules for **constexpr** evaluation in C++14 (see Section 2.2."**constexpr** Functions '14" on page 595), this compile-time-friendly container provides a simple way to define functions that populate tables of values.

```
    std::size_t
template <typename T, std::size_t N>
struct ConstexprArray
{
private:
    T d_data[N];  // data initialized at construction

public:
    template <typename F>
    constexpr ConstexprArray(const F &func)
    : d_data{}
    {
        for (int i = 0; i < N; ++i)
        {
            d_data[i] = func(i);
        }
    }

    constexpr const T& operator[](std::size_t ndx) const
    {
        return d_data[ndx];
    }
};
```

The numerous alternative approaches to writing such data structures vary in their complexity, trade-offs, and understandability. In this case, we default initialize our elements before populating them but do not need to rely on any other significant new language infrastructure. Other approaches could be taken; see Section 2.1."**constexpr** Functions" on page 239.

Given this utility class, we can then precompute at compile time any function that we can express as a **constexpr** function, such as a simple table of the first `N` squares:

```
constexpr int square(int x) { return x * x; }

constexpr ConstexprArray<int, 500> squares(square);

static_assert(squares[1]   == 1,    "");
static_assert(squares[289] == 83521,"");
```

Note that, as with many applications of **constexpr** functions, attempting to initialize a large array of **constexpr** variables will quickly bump up against compiler-imposed template instantiation limits. What's more, attempting to perform more complex arithmetic at compile time would be likely to exceed computation limits as well.

Diagnosing undefined behavior at compile time

ehavior-at-compile-time

Avoiding overflow during intermediate calculations is an important consideration, especially from a security perspective, and yet is a generally difficult-to-solve problem. Forcing computations to occur at compile time brings the full power of the compiler to bear in addressing such undefined behavior.

As an academically interesting example of this eminently practical security problem, suppose we want to write a compile-time function in C++ to compute the **Collatz length** of an arbitrary positive integer and generate a compilation error if any intermediate calculation would result in signed integer overflow.

First let's take a step back now to understand what we are talking about here with respect to Collatz length. Suppose we have a function `cf` that takes a positive **`int`**, `n`, and for even `n` returns `n/2` and for odd `n` returns `3n+1`.

```
int cf(int n) { return n % 2 ? 3 * n + 1 : n / 2; }  // Collatz function
```

Given a positive integer, `n`, the **Collatz sequence**, `cs(n)`, is defined as the sequence of integers generated by repeated application of the **Collatz function** — e.g., `cs(1) = { 4, 2, 1, 4, 2, 1, 4, ... };`, `cs(3) = { 10, 5, 16, 8, 4, 2, 1, 4, ... }`, and so on. A classic (but as yet unproven) conjecture in mathematics states that, for every positive integer, n, the Collatz sequence for n will eventually reach 1. The Collatz length of the positive integer n is the number of iterations of the Collatz function needed to reach 1, starting from n. Note that the Collatz sequence for `n = 1` is `{ 1, 1, 1, ... }` and its Collatz length is 0.

This example showcases the need for a **`constexpr`** variable in that we need to create a **constexpr context** – i.e., one requiring a constant expression to require that the evaluation of a **`constexpr`** function occur at compile time. Again, to avoid distractions related to implementing more complex functionality within the limitations of C++11 **`constexpr`** functions, we will make use of the relaxed restrictions of C++14; see Section 2.1."**`constexpr`** Functions" on page 239:

```
constexpr int collatzLength(long long number)
    // Return the length of the Collatz sequence of the specified number. The
    // behavior is undefined unless each intermediate sequence member can be
    // expressed as an unsigned long long.
{
    int length = 0;         // collatLength(1) is 0.

    while (number > 1)      // The current value of number is not 1.
    {
        ++length;           // Keep track of the length of the sequence so far.

        if (number % 2)     // if the current number is odd
        {
            number = 3 * number + 1;    // advance from odd sequence value
        }
        else
        {
            number /= 2;                // advance from even sequence value
        }
    }

    return length;
}
```

```
const     int c1 = collatzLength(942488749153153);  // OK, 1862
constexpr int x1 = collatzLength(942488749153153);  // OK, 1862

const int c2 = collatzLength(104899295810901231);
    // Bug, program aborts at runtime.

constexpr int x2 = collatzLength(104899295810901231);
    // Error, overflow in constant evaluation
```

In the example above, the variables `c1` and `x1` can be initialized correctly at compile time, but `c2` and `x2` cannot. The non**constexpr** nature of `c2` allows the overflow to occur and exhibit undefined behavior — integer overflow — at run time. On the other hand, the variable `x2`, due to its being declared **constexpr**, forces the computation to occur at compile time, thereby discovering the overflow and dutifully invoking the *non***constexpr** `std::abort` function, which in turn generates the desired error at compile time.

Potential Pitfalls

l-pitfalls-constexprvar

constexpr function pitfalls

texpr-function-pitfalls

Many of the uses of **constexpr** variables involve a corresponding use of **constexpr** functions (see Section 2.1."**constexpr** Functions" on page 239). The pitfalls related to **constexpr** functions are similarly applicable to the variables in which their results might be stored. In particular, it can be profoundly advantageous to forgo use of the **constexpr** function feature altogether, do the precomputation externally to program, and embed the calculated result — along with a comment containing source text of the (e.g., Perl or Python) script that performed the calculation — into the C++ program source itself.

Annoyances

annoyances-constexprvar

static member variables require external definitions

re-external-definitions

In most situations there is little behavioral difference between a variable at file or namespace scope and a **static** member variable; primarily they differ only in name lookup and access control restrictions. When they are **constexpr**, however, they behave very differently when they need to exist at runtime. A file or namespace scope variable will have internal linkage, allowing free use of its address with the understanding that that address will be different in different translation units:

```
// common.h:

constexpr int c = 17;
const int *f1();
const int *f2();

// file1.cpp:
#include <common.h>

const int *f1() { return &c; }
```

```
// file2.cpp:
#include <common.h>

const int *f2() { return &c; }

// main.cpp:
#include <common.h>
#include <cassert>  // standard C assert macro

int main()
{
    assert( f1() != f2() );    // Different addresses in memory per TU.
    assert( *f1() == *f2() );  // Same value.
    return 0;
}
```

For **static** data members, however, things become more difficult. While the **declaration** in the class **definition** needs to have an initializer, that is not itself a definition and will not result in static storage being allocated at runtime for the object, ending in a linker error when you try to build an application that tries to reference it[8]:

```
struct S {
    static constexpr int d_i = 17;
};
void useByReference(const int& i) { /* ... */ }

int main()
{
    const int local = S::d_i;  // OK, value is only used at compile time.
    useByReference(S::d_i);    // Link-Time Error, S::d_i not defined
    return 0;
}
```

This link-time error would be removed by adding a definition of `S::d_i`. Note that the initializer needs to be skipped, as it has already been specified in the definition of `S`:

```
constexpr int S::d_i;  // Define constexpr data member.
```

ed-in-their-own-class

No static constexpr member variables defined in their own class

When implementing a class using the singleton pattern, it may seem desirable to have the single object of that type be a **constexpr private static** member of the class itself, with guaranteed compile-time (data-race-free) initialization and no direct accessibility outside the class. This does not work as easily as planned because **constexpr static** data members must have a complete type and the class being defined is not complete until its closing brace:

```
class S
{
```

[8]The C++17 change to make **constexpr** variables **inline** also applies to **static** member variables, removing the need to provide external definitions when they are used.

```
    static const     S constVal;      // OK, initialized outside class below
    static constexpr S constexprVal;  // Error, constexpr must be initialized.
    static constexpr S constInit{};   // Error, S is not complete.
};

const S S::constVal{};  // OK, initialize static const member.
```

The "obvious" workaround of applying a more traditional singleton pattern, where the singleton object is a static local variable in a function call, also fails (see Section 1.1."Function **static** '11" on page 57) because **constexpr** functions are not allowed to have static variables (see Section 2.1."**constexpr** Functions" on page 239):

```
constexpr const S& singleton()
{
   static constexpr S object{};  // Error, even in C++14, static is not allowed.
   return object;
}
```

The only fully featured solution available for **constexpr** objects of static storage duration is to put them outside of their type, either at global scope, at **namespace** scope, or nested within a befriended helper class[9]:

```
class S
{
    friend struct T;
    S() = default;  // private
    // ...
};

struct T
{
    static constexpr S constexprS{};
};
```

see-also

See Also

- "**constexpr** Functions" (§2.1, p. 239) ♦ can be used to initialize a **constexpr** variable.
- "User-Defined Literals" (§2.1, p. 485) ♦ provides a convenient way of initializing a **constexpr** variable of a UDT with a compile-time value.

further-reading

Further Reading

[9] C++20 provides an alternate partial solution with the **constinit** keyword, allowing for compile-time initialization of static data members, but that still does not make such objects usable in a constant expression.

t-Member-Initializers

Default `class/union` Member Initializers

Non**`static`** class data members may specify a default initializer, which is used for constructors that don't initialize the member explicitly.

Description

description

The traditional means for initializing non**`static`** data members and base class objects within a class is a constructor's **member initializer list**:

```
struct B
{
    int d_i;

    B(int i) : d_i(i) { }     // Initialize d_i with i.
};

struct D : B
{
    char d_c;

    D() : B(2), d_c('3') { }  // Initialize base B with 2 andd_c with '3'.
};
```

Starting with C++11, non**`static`** data members — except for bit fields — can also be initialized using a **default member initializer**, either by using **copy initialization** or **direct list initialization**; see Section 2.1."Braced Init" on page 198:

```
struct S0
{
    int   d_i = 10;     // OK, uses copy initialization
    char  d_c = {'a'};  // OK, uses copy list initialization
    float d_f{2.f};     // OK, uses direct list initialization
};
```

Note that although braced initialization is supported, **direct initialization** with a parenthesized list is not:

```
struct S1
{
    char d_c('a');  // Error, invalid syntax
};
```

For any member, `m`, that has a default member initializer, constructors that don't initialize `m` in their member initializer list will implicitly initialize `m` by using the default member initializer value:

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

struct S2
{
```

```
    int d_i = 1;
    int d_j = 1;

    S2() { }                          // Initialize d_i with 1, d_j with 1.
    S2(int) : d_i(2) { }              // Initialize d_i with 2, d_j with 1.
    S2(int, int) : d_i(2), d_j(3) { } // Initialize d_i with 2, d_j with 3.
};

void f()
{
    S2 s2;
    assert(s2.d_i == 1);
    assert(s2.d_j == 1);
}
```

Previously initialized non**static** data members can be used in subsequent initializer expressions:

```
struct S4
{
    const char* d_s{"hello"};
    int         d_i{2};
    char        d_c{d_s[1]};  // OK, d_c initialized to 'e'.

    S4(const char *s) : d_s(s) { }
};

S4 s4("goodbye");          // OK, s4.d_c initialized to 'g'.
```

The default member initializer, just like member function bodies, executes in a **complete-class context**. This means it sees its enclosing class as a **complete type**; therefore, the initializer can reference the size of the enclosing type and invoke member functions that have not yet been seen:

```
struct S5
{
    int d_a[4];
    int d_i = sizeof(S5) + seenBelow();  // OK
    int seenBelow();
};
```

Name lookup in default member initializers will find members of the enclosing class and its bases before looking up at namespace level:

```
int i = 4;

struct S6
{
    int i = 5;
    int j = i;  // refers to S6::i, not ::i
};
```

```
S6 s6;  // OK, s6.j initialized to 5.
```

The **this** pointer can also be safely used as part of a default member initializer. As with any other uses of **this** inside a constructor, care must be exercised because the object referred to by **this** will be in a partially constructed state:

```
int getSomeRuntimeValue();

struct S7
{
    S7* d_selfPtr = this;                   // OK
    int d_bad = this->d_later;              // Bug, d_later not yet initialized
    int d_later = getInitialDLaterValue();  // OK
    static int getInitialDLaterValue();
};
```

Unlike variables at function or global scope and unlike static data members, a default member initializer for a member that is an array of unknown bound will not determine the array bound:

```
struct S8
{
    static int d_s[];        // OK, d_s has unknown bounds.
    int d_a[] = {1, 2, 3};   // Error, d_a is an array of unknown bound.
    int d_b[3] = {1, 2, 3};  // OK, bound explicitly specified
};

int a[] = {1, 2, 3};         // OK, the length of a is deduced to 3.
int S8::d_s[] = {4, 5, 6};   // OK, the length of S8::d_s is deduced to 3.
```

Non**volatile const** static data members of integral or enumeration type and also **constexpr** static data members may have an in-class default member initializer that must be a constant expression (see Section 2.1."**constexpr** Variables" on page 282):

```
struct MyLiteralType { int d_i; };

int nonConstexprFunction() { return 4; }

struct S9
{
    static int error = 1;
        // Error, nonconst static member cannot be initialized here

    static constexpr MyLiteralType mlt = {4};
        // OK, constexpr literal type initialized with a constant expression

    static const int i = 3;
        // OK, const integral type initialized with a constant expression

    static const int j = nonConstexprFunction();
        // Error, initializer is not a constant expression
};
```

Static member variable declarations that use a placeholder type (such as **auto**) must declare the member as **const** and have a default member initializer:

```
struct S10
{
    static const auto x;     // Error, no initializer provided
    static const auto y = 6; // OK
};
```

nteractions-with-unions

Interactions with unions

Default member initializer can also be used with union members. However, only one variant member of a union can have a default member initializer, since that will determine the default initialization of the entire union:

```
union U0
{
    char d_c = 'a';
    int  d_d = 1;
        // Error, only one member of U0 can have a default initializer.
};
```

Note that members of an anonymous union are considered to be variant members of the parent union:

```
union U1
{
    union { int d_x = 1; };

    union
    {
        int d_y;
        int d_z = 1;
            // Error, only one member of U1 can have a default initializer.
    };

    char d_c = 'a';
        // Error, only one member of U1 can have a default initializer.
};
```

As with classes, the default member initializer of a variant member will be used if there is no explicit initializer for a variant member in the parent union; in that case, the designated variant member is the active member of the union. If a union is initialized with a constructor that has a member initializer list, the default member initializer is ignored. Similarly, in C++14 if a union is initialized with a nonempty braced initializer list, the default member initializer is ignored ; see Section 2.1."Braced Init" on page 198):

```
union U2
{
    char d_c;
    int  d_i = 10;
```

```
};

U2 x;       // initializes d_i with 10
U2 y{};     // initializes d_i with 10

U2 z{'a'};  // initializes d_c with 'a', default initializer ignored

union U3
{
   char d_c;
   int  d_i = 10;

   U3() { }                // default member initializer used for d_i

   U3(char c) : d_c{c} { }  // default member initializer ignored
};
```

Use Cases

use-cases

Concise initialization of simple `structs`

ion-of-simple-structs

Default member initializers provide a concise and effective way of initializing all the members of a simple **`struct`**. Consider, for instance, a **`struct`** used to configure a thread pool:

```
struct ThreadPoolConfiguration
{
    int  d_numThreads         = 8;     // number of worker threads
    bool d_enableWorkStealing = true;  // enable work stealing
    int  d_taskSize           = 64;    // buffer size for an enqueued task
};
```

Compared to the use of a constructor, the above definition of `ThreadPoolConfiguration` provides sensible default values with minimal **boilerplate**.[1]

a-nonstatic-data-member

Ensuring initialization of a nonstatic data member

Non**static** data members that do not have a default member initializer nor appear in any constructor member initialization list are **default-initialized**. For **user-defined types**, default initialization is equivalent to the default constructor being invoked. For built-in types, default initialization results in an indeterminate value.

As an example, consider a **struct** tracking the number of times an user accesses a website:

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

struct UsageTracker
{
    std::string d_token;
    std::string d_websiteURL;
    int         d_clicks;
};
```

The programmer intended `UsageTracker` to be used as a simple aggregate. Forgetting to explicitly initialize `d_clicks` can result in a defect:

```
#include <map>     // std::map
#include <vector>  // std::vector

std::map<std::string, std::vector<UsageTracker>> usageTrackers;
// ...

void onVisitWebsite(const std::string& username, const std::string& token)
{
    UsageTracker ut = {token, "https://emcpps.com"};
    usageTrackers[username].push_back(ut);
        // Bug, ut.d_clicks has indeterminate value.
}
```

Consistent use of default member initializers for built-in types can avoid such defects:

[1] In C++20, designated initializers can be used to tweak one or more default settings in a configuration struct like `ThreadPoolConfiguration` in a clear and concise manner:

```
    assert

void testDesignatedInitializer()
{
    ThreadPoolConfiguration tpc = {.d_taskSize = 128};
    assert(tpc.d_numThreads == 8);
    assert(tpc.d_enableWorkStealing);
    assert(tpc.d_taskSize == 128);
}
```

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

struct UsageTracker
{
    std::string d_token;
    std::string d_websiteURL;
    int         d_clicks = 0;  // OK, will not have an indeterminate value
};
```

Avoiding boilerplate repetition across multiple constructors

multiple-constructors

Certain member variables of a type may be used to track the state of the object during its lifetime, independently of the initial state of the object. This means all constructors must set such variables to the same value, regardless of constructor arguments. Consider a state machine that controls execution of a background process:

```
class StateMachine
{
private:
    enum State { e_INIT = 1, e_RUNNING, e_DONE, e_FAIL };
    State          d_state;
    MachineProgram d_program;  // instructions to execute

public:
    StateMachine()  // Create a machine to run the default program.
    : d_state(e_INIT)
    , d_program(getDefaultProgram())
    { }

    StateMachine(const MachineProgram &program)  // Run the specified program.
    : d_state(e_INIT)
    , d_program(program)
    { }

    StateMachine(MachineProgram &&program)  // Run the specified program.
    : d_state(e_INIT)
    , d_program(std::move(program))
    { }

    StateMachine(const char *filename)  // read program to run from filename
    : d_state(e_INIT)
    , d_program(loadProgram(filename))
    { }
};
```

For member variables such as `d_state`, which always start at the same value and then are updated as the object is used, using a default initializer reduces boilerplate and reduces the risk of accidentally initializing the value wrongly:

```
class StateMachine
```

```
{
    enum State { e_INIT = 1, e_RUNNING, e_DONE, e_FAIL };
    State          d_state = e_INIT;  // default member initializer
    MachineProgram d_program;         // instructions to execute

public:
    StateMachine() : d_program(getDefaultProgram()) { }
    StateMachine(const MachineProgram &program) : d_program(program) { }
    StateMachine(MachineProgram &&program) : d_program(std::move(program)) { }
    StateMachine(const char *filename) : d_program(loadProgram(filename)) { }
};
```

Other members that are updated during an object's lifetime, such as mutable members for caching expensive calculations, can also benefit from being initialized once as opposed to providing individual initializers in each constructor. Suppose we define a class, `NetSession`, that caches a resolved IP address (v4 and v6), so it doesn't need to perform a DNS lookup every time the IP is needed. In all constructors, the IP is not resolved yet, meaning the cached IP is invalid. A simple convention is to set both IPs to `0` because no valid IP has that value:

```
#include <string>   // std::string
#include <cstdint>  // std::uint32_t, std::uint64_t

class NetSession
{
    std::string d_address;                     // such as "example.com"
    mutable std::uint32_t d_ip      = 0;       // cache of resolved IP address
    mutable std::uint64_t d_ipv6[2] = {0, 0};  // cache of resolved IPv6 address

public:
    NetSession() { }
    NetSession(const std::string& address) : d_address(address) { }

    // ...
};
```

Making default values obvious for documentation purposes

-documentation-purposes

Configuration objects that bundle together a large number of different properties are a popular artifact in large systems. Though the values may be loaded from, e.g., an appropriate configuration file, they often have meaningful default values. In C++03, the default values for these properties would typically be documented in the header file (`.h`) but actually effected in the implementation file (`.cpp`), which opens the opportunity for the documentation to go out of sync with the implementation:

```
// my_config.h:

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

```
struct Config
{
   std::string d_organization;  // default value "ACME"
   long long   d_maxTries;      // default value 1
   double      d_costRatio;     // default value 13.2

   Config();
};

// my_config.cpp:

#include <my_config.h>

Config::Config()
    : d_organization("Acme")    // Bug, doesn't match documentation
    , d_maxTries(3)             // Bug, went out of sync in maintenance
{ }                             // Bug, d_costRatio not initialized
```

Default initializers can be used in such cases to work as active documentation:

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

struct Config
{
   std::string d_organization = "Acme";
   long long   d_maxTries     = 3;
   double      d_costRatio    = 13.2;

   Config() = default;  // no user-provided definition needed any longer
};
```

potential-pitfalls

Potential Pitfalls

loss-of-insulation

Loss of insulation

Although convenient, placing default values in a header file — and thus potentially also using a **`default`** default constructor — can result in a loss of insulation that can, especially at scale, have severe consequences. For instance, consider a hash table with a non**`static`** data member representing the growth factor:

```
// hashtable.h:

class HashTable
{
private:
    float d_growthFactor;
    // ...

public:
    HashTable();
```

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

Without using default member initializers, the default growth factor is provided as part of the member initialization list of the default constructor:

```
// hashtable.cpp:
#include <hashtable.h>  // `HashTable`

HashTable::HashTable() : d_growthFactor(2.0f) { }
```

In the eventuality that the default growth factor is too large and results in uncontrolled memory consumption in production, it suffices to relink the affected applications with a new version of the library-provided `HashTable`, rather than recompiling them. Depending on the compilation and deployment infrastructure of a company, relinking can be significantly less expensive than recompiling.

If default member initializers had been used, the otherwise trivial default constructor might be defined in the header with `= default`, effectively removing any insulation of these values that might allow speedy relinking in lieu of expensive recompilation should these values need to change in a crisis.[2]

ubobject-initialization

Inconsistent subobject initialization

An approach occasionally taken to avoid keeping shared state in global locations is to have objects keep a handle to a `Context` object holding data that would otherwise be application global:

```
struct Context
{
    bool d_isProduction;
    long d_userId;
    int  d_datacenterId;
    // ... other information about the context being run in

    static Context* defaultContext();
};
```

Each type that needs `Context` information would take an optional argument to specify a local context; otherwise, it would use the default context:

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

struct ContextualObject
{
// ...
     ContextualObject(const std::string &name,
                      Context *context = Context::defaultContext());
// ...
};
```

[2]For a complete description of this real-world example, see **?**, section 3.10.5, pp. 783–789.

When combining many objects, all of which might need to access the same context for configuration, it becomes important to pass the context specified at construction to each subobject:

```
struct CompoundObject
{
// ...
    ContextualObject d_o1;
    ContextualObject d_o2;
// ...
    CompoundObject(Context *context = Context::defaultContext())
    : d_o1("First", context)
    , d_o2("Second", context)
    { }
// ...
};
```

This might seem like a situation well suited for using default member initializers, but the naive approach would have a serious flaw:

```
struct CompoundObject
{
// ...
    ContextualObject d_o1{"first"};   // Bug, does not use context passed to
    ContextualObject d_o2{"second"};  // CompoundObject constructor
// ...
    CompoundObject(Context *context = Context::defaultContext());
// ...
};
```

This bug, frustratingly, will impact only those applications that use multiple contexts, which might be a small subset of applications and libraries that use contextual objects. The only nonintrusive approach that avoids this bug is to forgo default member initializers for subobjects that can take a `context` parameter. The other, still error-prone alternative is to store a context pointer as the *first* member, initialize it in all constructors, and use that in the default member initializer of the subobjects:

```
struct CompoundObject
{
// ...
    Context          *d_context;
    ContextualObject  d_o1{"first", d_context};   // OK
    ContextualObject  d_o2{"second", d_context};  // OK
// ...
    CompoundObject(Context *context = Context::defaultContext())
    : d_context(context)
    { }
// ...
};
```

This has the downside of requiring an extra copy of the `Context*` member to transmit the passed-in value from the constructor to the subobject initializers. Additionally, now the

subobjects of `CompoundObject` strongly depend on being initialized after the `d_context` member variable, so the order of members is subtly constrained. Consequently, innocuous changes in member order during maintenance are liable to introduce bugs.[3]

Annoyances

annoyances

n-syntax-cannot-be-used

Parenthesized direct-initialization syntax cannot be used

Default member initializers make member variables tantalizingly close to allowing all of the same syntax usable for local and global variables. However, the direct initialization syntax is not allowed in default member initialization, which makes it tedious to copy code from automatic variables into class members. For example, suppose we set out to transform a function into an equivalent **function object** (useful for applications needing callbacks). This transformation entails migrating a function's automatic variables to member variables of the corresponding function object:

```
void function()  // before
{
    int i1 = 17;
    int i2(18);
    int i3{42};

    // ... do stuff
    // ... do more stuff
}

class Functor  // after
{
    int i1 = 17;  // OK
    int i2(18);   // Error, invalid syntax
    int i3{42};   // OK

    void operator()(int step)
    {
       switch (step)
       {
       case 0:  // ... do stuff
           break;
       case 1:  // ... do more stuff
           break;
           // ...
       }
    }
};
```

Cutting and pasting the definitions of the locals `i1`, `i2`, and `i3` will result in compile-time errors. The code initializing `i2` needs to be fixed to use either copy initialization (like `i1`) or braced initialization (like `i3`).

[3] C++17 introduces the **polymorphic memory resource** allocator model, which is a **scoped allocator model** with issues similar to `ContextualObject`.

ions-of-applicability

Limitations of applicability

Default member initializers can be used to replace the initialization of almost all members that can be placed in a constructor's member initializer list, except for two cases:

- Bit-field data members[4]
- Base classes and **`virtual`** base classes

Both of these situations continue to require initialization in member initializer lists, which translates to boilerplate that may be inconsistent with the rest of the codebase.

loss-of-triviality

Loss of triviality

Having member initializers makes the default constructor nontrivial, which in turn makes the class not a **POD (plain old data)** type. The presence of a nontrivial constructor can prevent some optimizations that might otherwise be possible; see Section 2.1."Generalized PODs '11" on page 374:

```
#include <type_traits>  // std::is_trivial

struct S0 { int d_i;     };
struct S1 { int d_i = 0; };
struct S2 { int d_i; S2() : d_i(0) { } };

static_assert(std::is_trivial<S0>::value, "");
static_assert(!std::is_trivial<S1>::value, "");
static_assert(!std::is_trivial<S2>::value, "");
```

s-of-aggregate-status

Loss of aggregate status

In C++11, classes using default member initializers are not considered **aggregates**, and that **aggregate initialization** can't be used is probably the biggest annoyance. Fortunately, the restriction has been lifted in C++14; see Section 1.2."Aggregate Init '14" on page 127:

```
struct ThreadPoolConfiguration
{
    int  d_numThreads         = 8;     // number of worker threads
    bool d_enableWorkStealing = true;  // enable work stealing
    int  d_taskSize           = 64;    // buffer size for an enqueued task
};
```

[4] In C++20, bit fields can now be initialized through a default member initializer:

```
struct S
{
    int d_i : 4 = 8;  // Error, before C++20; OK, in C++20
    int d_j : 4 {8};  // Error, before C++20; OK, in C++20
};
```

```
void f()
{
    ThreadPoolConfiguration tpc0;               // OK in C++11
    ThreadPoolConfiguration tpc2{16, true, 64}; // Error, in C++11; OK in C++14
}
```

see-also

See Also

- "Delegating Ctors" (§1.1, p. 43) ♦ can reduce code repetition in initialization of non**static** data members by chaining constructors together.
- "Aggregate Init '14" (§1.2, p. 127) ♦ can be applied to classes with and will use the initializers from default member initializers.
- "Braced Init" (§2.1, p. 198) ♦ default member initializers support **list initialization**, which is part of the uniform initialization effort.
- "Opaque **enum**s" (§2.1, p. 435) ♦ provides another means of insulating clients from unnecessary implementation details.

Further Reading

TBD

Strongly Typed, Scoped Enumerations

enumclass

An **enum class** is an alternative enumeration type that provides simultaneously (1) an enclosing scope for its enumerators and (2) stronger typing compared to a classic **enum**.

Description

description-enumclass

C++11 introduces a novel enumeration construct, **enum class** (or, equivalently, **enum struct**):

```
enum class  Ec { A, B, C }; // scoped enumeration, Ec, containing three enumerators
```

The enumerators of the **enum class** `Ec` (above) — namely, `A`, `B`, and `C` — do not automatically become part of the enclosing scope and must be qualified to be referenced:

```
Ec e0 = A;      // Error, A not found
Ec e1 = Ec::A;  // OK
```

Moreover, attempting to use an expression of type **enum class** `E` as, say, an **int** or in an arithmetic context will be flagged as an error, thus necessitating an explicit cast:

```
int  i0 = Ec::B;                    // Error, conversion to int not supported
int  i1 = static_cast<int>(Ec::B);  // OK, i1 is 1.
int  i2 = 1 + Ec::B;                // Error, conversion to int not supported
int  i3 = -Ec::B;                   // Error, unsupported arithmetic operations

bool b0 = Ec::B != 2;               // Error, comparison with int unsupported
bool b1 = Ec::B != Ec::C;           // OK, b1 is 'true'.
```

The **enum class** *complements*, but does not replace, the classical, C-style **enum**:

```
enum E { e_Enumerator0 /*= value0 */, /*...,*/ e_EnumeratorN /* = valueN */ };
    // Classic, C-style enum: enumerators are neither type safe nor scoped.
```

For examples where the classic **enum** shines, see *Potential Pitfalls — Strong typing of an* **enum class** *can be counterproductive* on page 321 and *Annoyances — Scoped enumerations do not necessarily add value* on page 327.

Still, innumerable practical situations occur in which enumerators that are both scoped and more type safe would be preferred; see *Introducing the C++11 scoped enumerations* on page 313 and *Use Cases* on page 315.

Drawbacks and workarounds relating to unscoped C++03 enumerations

ed-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 (scoped)
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 321.

Drawbacks relating to weakly typed, C++03 enumerators

yped,-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 scoped enumerations* on page 313, is more closely aligned with this second concept.

A classic enumeration, by default, has an implementation-defined **underlying type (UT)** (see Section 2.1."Underlying Type '11" on page 480), which it uses to represent variables of that enumerated type as well as the values of its enumerators. Although implicit conversion *to* an enumerated type is never permitted, when implicitly converting *from* a classic **`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 enum type 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
    int      i = e_B0;
    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 Section 1.1."**nullptr**" on page 87.

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]

[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,   "        "     "     "      "
}
```

ng-the-c++11-enum-class

Introducing the C++11 scoped enumerations

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 314 and *Potential Pitfalls — External use of opaque enumerators* on page 327.

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 };
```

Just like an unscoped **enum** type, an object of a scoped enumeration type is passed as a parameter to a function using the enumeration name itself:

```
void f1(VehicleUnscoped value);          // unscoped enumeration passed by value
void f2(VehicleScopedImplicitly value);  // scoped 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 311, 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()
{
    f3(VehicleScopedExplicitly::e_PLANE);
        // call f3 with an explicitly scoped enumerator

    f2(VehicleScopedImplicitly::e_PLANE);
        // call f2 with an implicitly scoped enumerator
}
```

Apart from implicit scoping, the modern, C++11 scoped enumeration 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 enumeration 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 321.

s-and-underlying-type

enum class and underlying type

Since C++11, both scoped and unscoped enumerations permit explicit specification of their integral underlying type (see Section 2.1."Underlying Type '11" on page 480):

```
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 327 and Section 2.1."Opaque **enum**s" on page 435.

use-cases-enumclass

Use Cases

ons-to-arithmetic-types

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 Transmission
{
    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 Transmission
{                                         // explicitly scoped
    enum Enum { e_MANUAL, e_AUTOMATIC };  // classic enum
};                                        // (BAD IDEA)

int buildCar(Car* result, int numDoors, Transmission::Enum transmission)
{
    int status = Transmission::e_MANUAL;    // Bug, accidental misuse

    for (int i = 0; i < transmission; ++i)  // Bug, accidental misuse
    {
         attachDoor(i);
    }

    return status;
}
```

As shown in the example above, it is never correct for a value of type `Transmission::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 Transmission { e_MANUAL, e_AUTOMATIC };  // modern enum class (GOOD IDEA)

int buildCar(Car* result, int numDoors, Transmission transmission)
```

```
{
    int status = Transmission::e_MANUAL;    // Error, incompatible types

    for (int i = 0; i < transmission; ++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 321.

In the 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**:

```
enum class

const int manualIntegralValue    = static_cast<int>(Transmission::e_MANUAL);
const int automaticIntegralValue = static_cast<int>(Transmission::e_AUTOMATIC);
static_assert(0 == manualIntegralValue,    "");
static_assert(1 == automaticIntegralValue, "");
```

Avoiding namespace pollution

g-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.
// ...
```

loading-disambiguation

Improving overloading disambiguation

Stronger type safety of scoped enumerations might prevent mistakes when calling overloaded functions, especially when the overload set accepts multiple arguments. 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 an unscoped **`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 of the 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
    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
```

[2]GCC 10.2 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(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, often 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 be strongly typed — such as `Color`, used widely in function signatures — can help to expose accidental misuse but, again, see *Potential Pitfalls — Strong typing of an* **`enum class`** *can be counterproductive* on page 321.

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
}
```

numerators-themselves

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 in the least-significant bit the months that have 31 days 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 Section 2.1."Underlying Type '11" on page 480.

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.

tial-pitfalls-enumclass

an-be-counterproductive

Potential Pitfalls

Strong typing of an `enum class` can be counterproductive

The additive value in using a scoped enumeration is governed *solely* by whether the stronger typing of its enumerators, and *not* the implicit scoping, would be beneficial in typical anticipated 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 `setPort` using one of these enumerated values:

```
void setCurrentPortToCtrl()
{
    setPort(SysPort::e_CTRL);  // Error, cannot convert SysPort 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:

`std::is_same`

```
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.

ions-of-named-constants

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 Section 2.1."**constexpr** Variables" on page 282. 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 Section 2.1."**constexpr** Variables" on page 282) and a *prvalue* (see Section 2.1."*rvalue* References" on page 479), 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:

```
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
```

iation-with-bit-flags

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)
}
```

ociation-with-iteration

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 (grouped by astronomical seasons in the Northern temperate zone):

```
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_SEP;
}
```

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) % 12);
}

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_SEP;
}

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

e-enumerators-enumclass

Since scoped enumerations have an underlying type (UT) 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 Section 2.1."Opaque **enum**s" on page 435.

Annoyances

annoyances-enumclass

Scoped enumerations do not necessarily add value

t-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

See Also

- "Opaque **enum**s" (§2.1, p. 435) ♦ illustrates how sometimes it is useful to entirely insulate individual enumerators from clients.
- "Underlying Type '11" (§2.1, p. 480) ♦ shows how, absent implicit conversion to integrals, **enum** class values may use **static_cast** in conjunction with their underlying type.

further-reading

Further Reading

- ?
- ?

Explicit Instantiation Declarations

template-instantiations

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

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

nstantiation-definition

Creating an **explicit-instantiation definition** was possible prior to C++11.[1] The requisite syntax is to place the keyword **template** in front of the name of the fully specialized class template, function template, or — in C++14 — variable template (see Section 1.2."Variable Templates" on page 146):

```
#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 347.

antiation-declaration

Explicit-instantiation declaration

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

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

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

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 347.

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

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

extern template class std::vector<int>;

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

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

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

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

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

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

As the example above illustrates, overload resolution and template 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 345.

te-illustrative-example

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 347. In this specific case, we own the component that requires specialization, and the specialization is for a ubiquitous built-in type; hence, the natural place to generate the specialized definitions is in a `.cpp` file corresponding to the component's header:

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

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

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

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

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

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

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

}; // close my namespace
```

Each of the constructs introduced by the keyword **`template`** within the `my` namespace in the 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`**. Explicit instantiation of individual member functions, such as `length()` in the example above, is, however, only rarely useful; see *Annoyances — All members of an explicitly defined template class must be valid* on page 348.

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

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

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

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

                    // -----------------------------------
                    // Explicit instantiation declarations
                    // -----------------------------------

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

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

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

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

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

}  // close my namespace

#endif  // close internal include guard
```

Each of the constructs that 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 `my_header.h` *after* the declaration of the corresponding general template and, importantly, before whatever relevant definitions are ever used.

t-on-various-.o-files

The effect on various `.o` files

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

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

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

template <typename T>  // elided definition of a class template
```

```
class Interval
{
    T d_low;   // interval's low value
    T d_high;  // interval's high value

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

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

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

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

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

    // ...
};

template <typename T>                    // elided definition of a function template
bool intersect(const Interval<T>& i1, const Interval<T>& i2)
    // Determine whether the specified intervals intersect ...
{
    bool result = false;  // nonintersecting until proven otherwise
    // ...
    return result;
}

}  // close lib namespace

#endif  // INCLUDED_LIB_INTERVAL


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

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

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

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

int main(int argv, const char** argc)
{
```

```
    lib::Interval<double> a(0, 5);  // instantiate with double type 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 also defines the *strong* symbol `main`, marked here by a `T`. Linking `app.o` with any other object file containing such a symbol would cause the linker to report a multiply-defined-symbol error. On the other hand, the `lib_interval.o` file corresponds to the `lib_interval` library component, whose `.cpp` file served only to include its own `.h` file, and is again effectively empty.

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

[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 W lib::Interval<double>::Interval(double const&, double const&)
00000000004005a2 W lib::Interval<int>::Interval(int const&, int const&)
00000000004005ce W bool lib::intersect<double>(lib::Interval<double> const&,
                                               lib::Interval<double> const&)
00000000004004b7 T main
```

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

More generally, if the application comprises multiple object files, each file will potentially contain their own set of weak symbols, often leading to duplicate code sections for implicitly instantiated class, function, and variable templates instantiated on the same 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.

Imagine now that our program includes a large number of object files, many of which make use of our `lib_interval` component, particularly to operate on **`double`** intervals. Suppose, for now, we decide we would like to suppress the generation of object code for templates related to just **`double`** type with the intent of later putting them all in one place, i.e., the currently empty `lib_interval.o`. Achieving this objective is precisely what the **`extern template`** syntax is designed to accomplish.

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

```
    std::size_t
// lib_interval.h:  // (no change to existing code)
#ifndef INCLUDED_LIB_INTERVAL  // internal include guard
#define INCLUDED_LIB_INTERVAL

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

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

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

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

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

}  // close lib namespace

#endif  // INCLUDED_LIB_INTERVAL
```

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

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

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

lib_interval.o:
```

Notice that this time some of the symbols — specifically those relating to the class and function templates instantiated for type **`double`** — have changed from `W`, indicating a *weak* symbol, to `U`, indicating an *undefined* one. 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 object file of the `lib_interval` library component itself, but see *Annoyances — No good place to put definitions for unrelated classes* on page 347. To force the needed template definitions into

the `lib_interval.o` file, we will need to use explicit-instantiation definition syntax, i.e., the **`template`** prefix:

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

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

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

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

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

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

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

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

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

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

```
00000000004004b7 T main
```

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

use-cases

Reducing template code bloat in object files

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) { /*...*/ }
    // Return the scalar ("dot") product of the specified 'a' and 'b'

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

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

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

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

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

}  // close game namespace

#endif  // INCLUDED_GAME_FIXEDARRAY
```

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

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

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

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

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

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

[4]Notice that we have chosen *not* to nest the explicit specializations (or any other definitions) of entities already declared directly within the `game` namespace, preferring instead to qualify each entity explicitly to be consistent with how we render free-function definitions (to avoid self-declaration); see **?**, 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 345.

```
                          const FixedArray<double, 2>& b);

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`[5]:

```
$ 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:

```
    std::size_t

// app_geometryutil.cpp:

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

using namespace game;

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

void translate(Unit* object, const FixedArray<double, 3>& dst)
```

[5]Note that only a subset of the relevant symbols have been retained.

```
    // Perform precise movement of the object in 3D space.
{
    FixedArray<double, 3> delta;
    // ...
}

bool isOrthogonal(const FixedArray<int, 2>& a1, const FixedArray<int, 2>& a2)
    // Return true if 2d arrays are orthogonal.
{
    return 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>  // game::FixedArray
#include <game_unit.h>        // game::Unit

using namespace game;

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

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

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

```
$ nm -C app_geometryutil.o
000000000000003e T isOrthogonal(game::FixedArray<int, 2ul> const&,
  game::FixedArray<int, 2ul> const&)
0000000000000068 T isOrthogonal(game::FixedArray<int, 3ul> const&,
  game::FixedArray<int, 3ul> const&)
```

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

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

Whether optimization involving explicit-instantiation directives reduces library sizes on 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 347.

Insulating template definitions from clients

initions-from-clients

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

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

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

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

#endif
```

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

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

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

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

Without the two explicit-instantiation 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
#define INCLUDED_TRANSFORM

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

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

#endif
```

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

pitfalls-externtemplate

Corresponding explicit-instantiation declarations and definitions

rations-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[6]:

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

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

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

#endif


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

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

[6]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.

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

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

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

In the example above, `f(i)` works as expected, with the linker finding the definition of `f<int>` in `c.o`; `f(c)` fails to link, because no definition of `f<char>` is guaranteed to be found anywhere; and `f(d)` accidentally works by silently generating a *redundant* local copy of `f<double>` in `client.o` while another, identical definition is generated explicitly in `c.o`. Importantly, note that **`extern template`** has *absolutely no effect* on overload resolution because the call to `f(c)` did *not* resolve to `f<int>`.

Accidentally making matters worse

ly-making-matters-worse

When making the decision to explicitly instantiate common specializations of popular templates within some designated object 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 member functions but typically use only a few.

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

Annoyances

noyances-externtemplate

No good place to put definitions for unrelated classes

s-for-unrelated-classes

When we consider the implications of physical dependency,[7,8] determining in which component to deposit the specialized definitions can be problematic. For example, consider a codebase implementing a core library that provides both a nontemplated `String` class and a `Vector` container class template. These fundamentally unrelated entities would ideally live in separate physical components — i.e., `.h`/`.cpp` pairs — neither of which depends

[7] See **?**
[8] See **?**

physically on the other. That is, an application using just one of these components could be compiled, linked, tested, and deployed entirely independently of the other. Now, consider a large codebase that makes heavy use of `Vector<String>`: In what component should the object-code-level definitions for the `Vector<String>` specialization reside?[9] 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**[10] the mutual dependency to a higher level in the physical hierarchy, we avoid forcing any client to depend on more than what is actually needed. The practical drawback to this approach is that only those clients that proactively include the composite `stringvector.h` header would realize any benefit; fortunately, in this case, there is no **one-definition rule (ODR)** violation if they don't.

Finally, complex machinery could be added to both `string.h` and `vector.h` to conditionally include `stringvector.h` whenever both of the other headers are included; such heroic efforts would, nonetheless, involve a **cyclic physical dependency** among all three of these components. Circular intercomponent collaborations are best avoided.[11]

e-class-must-be-valid

All members of an explicitly defined template class must be valid

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

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

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

[9] 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. JOHN: ADD LAKOS20 CITATION HERE.

[10] **?**, section 3.5.2, "Escalation," pp. 604–614

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

```
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) { /* construct from i */ }
    // ...
};

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

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

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

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

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

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

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

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

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

see-also

See Also

- "Variable Templates" (§1.2, p. 146) ♦ Extension of the template syntax for defining a family of like-named variables or static data members that can be instantiated explicitly

further-reading

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 (T&&)

forwardingref

A forwarding reference (T&&) — distinguishable from an *rvalue* reference (&&) (see Section 2.1."*rvalue* References" on page 479) 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

scription-forwardingref

Sometimes we want the same reference 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` below) 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` appears syntactically 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 `forRef`; see *Description — Identifying forwarding references* on page 355. 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 template `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 type that might benefit from being able to be moved */ };

void g(const S&);   // target function - overload for const S lvalues
void g(S&&);        // target function - overload for S rvalues only

template <typename T>
void f(T&& forRef); // forwards to target overload g based on value category
```

Note that a function may be overloaded on the reference type alone (see Section 2.1."*rvalue* References" on page 479); however, overloading on a **const** *lvalue* reference and an *rvalue* reference occur most often in practice. 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.

f-invoked-example

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 `forRef` originate from an *lvalue* expression?" and allows the developer to branch on the answer. A better solution that captures this logic at compile time is generally preferred; see *Description — The* `std::forward` *utility* on page 358:

```
#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&> -- forRef is an lvalue reference (S&).
             // The function f<S&> will end up invoking g(S&).

    f(S());  // Instantiates f<S> -- forRef 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 359.

late-argument-deduction

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
    f<short>(); // OK, T specified explicitly
    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)
    h('A', 'B'); // OK, both arguments have same type.
}
```

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 `&&` (i.e., `&` or `&&` along with any cv-qualifiers) 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 parameter 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 352 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` (see Section 2.1."*rvalue* References" on page 479), then `T` is deduced as `S` and `forRef` becomes an *rvalue* reference. The underlying process that results in this duality relies on **reference collapsing** (see the next section) and special **type deduction** rules introduced for this particular case. 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 a nonreference type:

```
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 Section 2.1."**auto** Variables" on page 183.

reference-collapsing

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 (e.g., a named variable), 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 uses 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 is performed by the compiler in any situation where a reference to a reference would be 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.

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

forwardingref-table1

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 T1 = int&;  // OK
using T2 = i&;    // OK, int& & becomes int&.
static_assert(std::is_same<T2,int&>::value);
```

Furthermore, references to references can occur during computations involving **metafunctions** or as part of language rules (such as type deduction):

```
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).[1]

g-forwarding-references

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, a 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
```

[1] In C++20, the **typename** disambiguator is no longer required in some of the contexts where a dependent qualified name must be a type. For example, when a dependent name is used as a function return type — **template<class** T> T::R f(); — then **typename** is not required.

```
{
    template <typename T>
    void f(T&& forRef);
        // 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 370.

`auto&&` — a forwarding reference in a nonparameter context

non-parameter-context

Outside of template function parameters, *forwarding* references can also appear in the context of variable definitions using the **`auto`** keyword (see Section 2.1."**`auto`** Variables" on page 183) 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 = 0;
    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 the case of **auto&&**, however, the reference will be **const** *only* if it is initialized with a **const** object:

```
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 358):

`std::forward`

```
#include <tuple>   // std::get
#include <utility> // std::forward
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 = std::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 Section 1.1."**decltype**" on page 22) was used in the example above to retrieve the type of `fr`.

Forwarding references without forwarding

`nces-without-forwarding`

Sometimes deliberately *not* forwarding (see *Description — The* `std::forward` *utility* on page 358) an **auto&&** variable or a forwarding reference function parameter at all can be useful. In such cases, *forwarding* references are employed 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` (above) might result in moving from `r` twice, which is potentially unsafe (see Section 2.1."*rvalue* References" on page 479):

```
auto b = std::begin(std::forward<decltype(r)>(r));
auto e = std::end  (std::forward<decltype(r)>(r));  // BAD IDEA:
                                                    // r might be moved from.
```

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`:

```
auto b = std::begin(r);
auto e = std::end(std::forward<decltype(r)>(r));  // BAD IDEA: e might have
                                                  // a different type than b.
```

-std::forward-utility

The `std::forward` utility

The final piece of the forwarding reference infrastructure is the `std::forward` utility function. Since the expression naming a forwarding reference `x` is always an *lvalue* — due to its reachability by either name or address — 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>`):

std::remove_reference

```
namespace std {
template <class T> T&& forward(typename remove_reference<T>::type&  t) noexcept;
template <class T> T&& forward(typename remove_reference<T>::type&& t) noexcept;
}
```

Note that the second overload will be deliberately removed from the overload set if `T` is an *lvalue* reference type to avoid ambiguity.

Recall that the type `T` associated with a forwarding reference is deduced as a reference type if given an *lvalue* reference and as a nonreference 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. Notice the effect of reference collapsing in the return type: `(U&)&&` becomes simply `U&`.

- 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 351. Using `std::forward` over **`static_cast`**, however, expresses the programmer's intent explicitly.

Use Cases

use-cases-forwardingref

Perfectly forwarding an expression to a downstream consumer

o-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 an `std::string` either by **`const`** *lvalue* reference (e.g., with the intention of *copying* from it) or by *rvalue* reference (e.g., with the intention of *moving* from it):

std::stringstd::move

```
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 361.

Handling multiple parameters concisely

le-parameters-concisely

Suppose we 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 to preserve some

class invariant[2]:

```
#include <type_traits>  // std::decay, std::enable_if, std::is_same
#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:
    static 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.
    // ...

    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.
    }
};
```

[2]This type of value-semantic type can be classified more specifically as a *complex-constrained* attribute class; see **?**, section 4.2.

Because the template arguments used in each successive function parameter are deduced interdependently from the types of their corresponding function arguments, the `setPersonsIfValid` function template can be instantiated for a full cross product of variations 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

neric-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`. Correctly passing arguments to the constructor, 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 Section 2.1."Variadic Templates" on page 519. Furthermore, our simplified `make_shared` creates the object on the heap with **`new`** and constructs an `std::shared_ptr` to manage the lifetime of that object.

Let's now consider how we would structure the declaration of this form of `make_shared`:

std::shared_ptr

```
namespace my {
template <typename OBJECT_TYPE, typename ARG1, typename ARG2>
std::shared_ptr<OBJECT_TYPE> make_shared(ARG1&& arg1, ARG2&& arg2);
}
```

Notice that 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 `OBJECT_TYPE` object on the heap and then create our output `shared_ptr`:

```
template <typename OBJECT_TYPE, typename ARG1, typename ARG2>
std::shared_ptr<OBJECT_TYPE> my::make_shared(ARG1&& arg1, ARG2&& arg2)
{
    OBJECT_TYPE *object_p = new OBJECT_TYPE(std::forward<ARG1>(arg1),
                                            std::forward<ARG2>(arg2));
    try
    {
        return std::shared_ptr<OBJECT_TYPE>(object_p);
    }
    catch (...)
    {
        delete object_p;
        throw;
```

```
    }
}
```

Notice that this simplified implementation needs to clean up the allocated object if the constructor for the return value throws; normally a **RAII** proctor to manage this ownership would be a more robust solution to this problem.

Importantly, using `std::forward` to construct the object means that the arguments passed to `make_shared` will be used to find the appropriate matching two-parameter constructor of `OBJECT_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. What's 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, one for each argument, if we were not using perfect forwarding — the full cross product of **`const`** (or not), **`volatile`** (or not), and `&` or `&&` (or neither). A full implementation of just this two-parameter variation would require 144 distinct overloads, all almost identical and most never used. Using forwarding references reduces that to just one overload for each number of arguments.

Wrapping initialization in a generic factory function

eric-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:

std::forward

```
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 Section 2.1."Variadic Templates" on page 519) of forwarding references to handle calling the constructor for us:

```
template <typename OBJECT_TYPE, typename... ARGS>
OBJECT_TYPE trackConstruction(ARGS&&... args)
{
    TrackingSystem::trackInitialization<OBJECT_TYPE>(sizeof...(args));
    return OBJECT_TYPE(std::forward<ARGS>(args)...);
}
```

This use of a variadic pack of forwarding references 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 objects `untracked` and `tracked` are constructed. 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 object 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**. It is possible to ensure that this elision actually happens (on all current compilers of which the authors are aware) by publicly **declaring** but not **defining** the copy constructor for `BigObject`.[3] We 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

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>`:

```
    std::vectorstd::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 on 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 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 subsequent buffer reallocation will *move* the elements between buffers rather than copy them, assuming that the element's move constructor has a **`noexcept`** specifier (see Section 3.1."**`noexcept`** Specifier" on page 676). The amount of work can, however, be further reduced: 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 Section 2.1."Variadic Templates" on page 519) and perfect forwarding (see *Use Cases — Perfect forwarding for generic factory functions* on page 361). Rather than accepting a

[3] In C++17, this copy elision can be guaranteed and is allowed to be done for objects that have no copy or move constructors.

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 empty 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`:

```
    std::forwardstd::size_t
```

```
template <typename T>
template <typename... Args>
void std::vector<T>::emplace_back(Args&&... args)
{
    // ...
    (void) new (d_data_p[d_size]) 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 without defining the *copy* or *move* constructor of a noncopyable or nonmovable type to be private is often a way to guarantee that a C++11/14 compiler constructs 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

g-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 a function that reads a file, does some spell-checking for every unique word in the file, and gives us a list of incorrect words and corresponding suggested proper spellings, implemented using a

range-like[4] library having common utilities similar to standard UNIX processing utilities:
`std::mapstd::stringstd::tuple`

```
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(); }));
}
```

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.

To better understand, document, and debug what is happening here, we decide 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, we 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

[4]The C++20 `ranges` library that provides a variety of range utilities and adaptors allows for composition using the pipe (`|`) operators instead of nested function calls, resulting in code that might be easier to read:

```
#include <algorithm>  // std::ranges::equal
#include <cassert>    // standard C assert macro
#include <ranges>     // std::ranges::views::transform, std::ranges::views::filter

void f()
{
    int data[] = {1, 2, 3, 4, 5};
    int expected[] = {1, 9, 25};

    auto isOdd  = [](int i) { return i % 2 == 1; };
    auto square = [](int i) { return i * i; };

    using namespace std::ranges;

    // function-call composition
    assert(equal(views::transform(views::filter(data, isOdd), square), expected));

    // pipe operator composition
    assert(equal(data | views::filter(isOdd) | views::transform(square), expected));
}
```

document this expression while achieving the same result:

```
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 duplicate words so as to create 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 two-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

ns-with-string-literals

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:

```
    std::string
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

nable-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 Section 2.1."*rvalue* References" on page 479:

```
    std::forward

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 because it could now be in a moved-from state.

A perfect-forwarding constructor can hijack the copy constructor

-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:

```
    std::stringstd::forward
#include <string>   // std::string
#include <utility>  // std::forward
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:

```
    std::enable_ifstd::decaystd::forward
```

```
#include <type_traits> // std::enable_if,  std::is_same
#include <utility>     // std::forward
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<T>.
};

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 function `h` replicates what had been a problematic scenario in the earlier function `g`. Also 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 371.

Annoyances

oyances-forwardingref

ike-rvalue-references

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 355, 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.[5]

[5] 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; };
```

```
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 remarkable that we still do not have some unique syntax (hypothetically, `&&&`) that we could use, at least optionally, to imply unequivocally a *forwarding* reference that is independent of its context.

Metafunctions are required in constraints

equired-in-constraints

As we showed in *Use Cases* on page 359, 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.

The challenge of correctly forwarding only the value category, however, 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 we want to accept. As an example, let's consider a function intended to *copy* or *move* a `Person` object into a data structure:

std::enable_ifstd::decaystd::is_same

```
#include <type_traits> // std::decay, std::enable_if, std::is_same

class Person;
class PersonManager {
    // ...
public:
    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 pattern that constrains `T` has several layers to it, so let's unpack them one at a time.

```
void f(Addable auto&& a);  // C++20 terse concept notation

void example()
{
    int i;

    f(i);  // OK, decltype(a) is int& in f.
    f(0);  // OK, decltype(a) is int&& in f.
}
```

- `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 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 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`.[6] 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, at 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 argument for the final template parameter (unused so left unnamed), the expression is going to be instantiated for any deduced `T` considered during overload resolution for `addPerson`. For any `T` that is not a (possibly) cv-ref-qualified `Person`, `enable_if` will not define the `type` member **`typedef`**, leading to a failure during the substitution process. Rather than being a compile-time error, such substitution failure will just remove `addPerson` from the overload set being considered, hence the term "substitution failure is not an error," or SFINAE. If a client attempts to pass a non`Person` as an argument to the `addPerson` function, the compiler will issue an error that there is no matching function for call 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

see-also

- "*rvalue* References" (§2.1, p. 479) ♦ details a feature that can be confused with forwarding references due to similar syntax.
- "'**`auto`** Variables" (§2.1, p. 183) ♦ covers a feature that can introduce a forwarding reference with the **`auto`**`&&` syntax.

[6] C++20 provides the `std::remove_cvref<T>` metafunction that can be used to remove cv and reference qualifiers in a terse manner.

- "Variadic Templates" (§2.1, p. 519) ♦ explores a feature commonly used in conjunction with forwarding references to provide highly generic interfaces.

Further Reading

further-reading

- "Item 24: Distinguish universal references from rvalue references," **?**
- **?**
- **?**

Trivial and Standard-Layout Types

gpods

placeholder

inheritingtoconstructing

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.

criptioninheritingctor

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 296).

```
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.
```

```
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&&)
```

Note that we use braced initialization (see Section 2.1."Braced Init" on page 198) 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**.

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."**??**" on page **??**) 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 388.

Inheriting constructors having the same **signature** from multiple base classes leads 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;
};
```

```
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 676 and Section 2.1."**`constexpr`** Functions" on page 239. 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:

```
    std::cout
```

```
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 30), or **`delete`** it (see Section 1.1."**??**" on page **??**).

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. In all of the example code in *Description* on page 375 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 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 384.

A decidedly more complex alternative affording a different set of trade-offs would involve variadic template constructors (see Section 2.1."Variadic Templates" on page 519) having forwarding references (see Section 2.1."Forwarding References" on page 351) 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 388.

se-cas abstract-inheriting

Use Cases

Employing 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 296). 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 nonvirtual member functions.

-structural-inheritance

Avoiding boilerplate code when employing structural inheritance

A key indication for using 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, 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.
```

```
    {
        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.

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, an alternative solution would be to use conditional compilation to add explicit checks in certain build configurations (e.g., using C-style `assert` macros).[1]

mentation-inheritance

Avoiding boilerplate code when employing implementation inheritance

Sometimes it can be cost effective to adapt a **concrete class** having virtual functions to a specialized purpose by using inheritance. 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.[2] Such inheritance, known as **implementation inheritance**, is decidedly distinct from pure **interface inheritance**, which is often the preferred design pattern in practice.[3] 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:

```
    std::ostream

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).

[1] A more robust solution along these same lines is anticipated for a future release of the C++ language standard and will be addressed in **?**.

[2] See **?**, section 4.7.

[3] See **?**, section 4.6.

Inheriting constructors make deriving from `NetworkDataStream` and overriding (see Section 1.1."**override**" on page 92) `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;

    void onPacketReceived(DataPacket& dataPacket) override
    {
        LOG_TRACE << "Received packet " << dataPacket;    // local log facility
        NetworkDataStream::onPacketReceived(dataPacket);  // Delegate to base.
    }
};
```

enting-a-strong-typedef

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 additional type safety over using the original type name. 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`**. 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 480.

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
}

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?

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.

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) type-safety:

DateDateexplicitDateDateDateexplicitDate

```
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 386.

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).[4]

ty-via-a-mix-in-class

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 want 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` adapter 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
```

[4] See **?**, section 4.4.

```
#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

pitfalls-ctorinheriting

Newly introduced constructors in the base class can silently alter program behavior

-alter-program-behavior

The introduction of a new constructor in a base class might silently change a program's runtime 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 78) 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 381) — might sometimes, however, help to prevent such unfortunate missteps.

implicit-constructors

Inheriting *implicit* constructors

Inheriting from a class that has implicit constructors can cause surprises. Consider again using inheriting constructors to implement a strong **typedef** from *Use Cases — Implementing a strong* **typedef** on page 381. 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 381:

```
    std::string

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.

yances-inheritingctor

Annoyances

selected-individually

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."**??**" on page **??**) 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`.

same-as-in-base-class

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[5] : 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 351) and variadic templates (see Section 2.1."Variadic Templates" on page 519) — made somewhat different trade-offs and was considered too onerous and fragile to be practically useful:

```
#include <utility>  // std::forward

struct Base
```

[5]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**.

```
{
    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 351) to properly delegate the implementation of a constructor that is declared **`protected`** in the derived class to a **`public`** constructor of a privately inherited base class. Although this approach fails to preserve many of the characteristics of the inheriting constructors (e.g., **`explicit`**, **`constexpr`**, **`noexcept`**, and so on), the functionality described in the code snippet above is simply not possible using the C++11 inheriting-constructors feature.

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.[6] 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[7] 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,[8] 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.[9]

See Also

`see-also`

- "Delegating Ctors" (§1.1, p. 43) ♦ are used to call one constructor from another from within the same user-defined type.
- "Defaulted Functions" (§1.1, p. 30) ♦ are used to implement functions that might otherwise have been suppressed by inherited constructors.
- "**??**" (§1.1, p. **??**) ♦ can be used to exclude inherited constructors that are unwanted entirely.

[6] 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.]

[7] **?**

[8] **?**

[9] 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.

- "**`override`**" (§1.1, p. 92) ♦ can be used to ensure that a member function intended to override a virtual function actually does so.
- "Default Member Init" (§2.1, p. 296) ♦ can be used to provide nondefault values for data members in derived classes that make use of inheriting constructors.
- "Forwarding References" (§2.1, p. 351) ♦ are used as an alternative (workaround) when access levels differ from those for base-class constructors.
- "Variadic Templates" (§2.1, p. 519) ♦ are used as an alternative (workaround) when access levels differ from those for base-class constructors.

`further-reading`

Further Reading

List Initialization: `std::initializer_list<T>`

initlist

placeholder

Unnamed Local Function Objects (Closures)

lambda-expressions lambda

Lambda expressions provide a means of defining function objects at the point where they are needed, enabling a powerful and convenient way to specify callbacks or local functions.

Description

description

Generic, object-oriented, and functional programming paradigms all place great importance on the ability of a programmer to specify a *callback* that is passed as an argument to a function. For example, the Standard-Library algorithm, `std::sort`, accepts a callback argument specifying the sort order:

```
#include <algorithm>   // std::sort
#include <functional>  // std::greater
#include <vector>      // std::vector

template <typename T>
void sortAscending(std::vector<T>& v)
{
    std::sort(v.begin(), v.end(), std::greater<T>());
}
```

The function object, `std::greater<T>()`, is callable with two arguments of type `T` and returns **`true`** if the first is greater than the second and **`false`** otherwise. The Standard Library provides a small number of similar functor types, but, for more complicated cases, the programmer must write a functor themselves. If a container holds a sequence of `Employee` records, for example, we might want to sort the container by name or by salary:

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

struct Employee
{
    std::string name;
    long        salary;  // in whole dollars
};

void sortByName(std::vector<Employee>& employees);
void sortBySalary(std::vector<Employee>& employees);
```

The implementation of `sortByName` can delegate the sorting task to the standard algorithm, `std::sort`. However, because `Employee` does not supply **`operator<`** and to achieve the correct sorting criteria, we will need to supply `std::sort` with a callback that compares the names of two `Employee` objects. We implement this callback as a pointer to a simple function that we pass to `std::sort`:

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

bool nameLt(const Employee& e1, const Employee& e2)
```

```
    // returns true if e1.name is less than e2.name
{
    return e1.name < e2.name;
}

void sortByName(std::vector<Employee>& employees)
{
    std::sort(employees.begin(), employees.end(), &nameLt);
}
```

The `sortBySalary` function can similarly delegate to `std::sort`. For illustrative purposes, we will use a **function object** (a.k.a., **functor**) rather than a function pointer as the callback to compare the salaries of two `Employee` objects. Every **functor class** must provide a **call operator** (i.e., **`operator()`**), which, in this case, compares the salary fields of its arguments:

```
struct SalaryLt
{
    // Functor whose call operator compares two Employee objects and returns
    // true if the first has a lower salary than the second, false otherwise.

    bool operator()(const Employee& e1, const Employee& e2) const
    {
        return e1.salary < e2.salary;
    }
};

void sortBySalary(std::vector<Employee>& employees)
{
    std::sort(employees.begin(), employees.end(), SalaryLt());
}
```

Although it is a bit more verbose, a call through the function object is easier for the compiler to analyze and automatically inline within `std::sort` than is a call through the function pointer. Function objects are also more flexible because they can carry state, as we'll see shortly. The sorting example illustrates how small bits of a function's logic must be factored out into special-purpose auxiliary functions and/or functor classes that are often not re-usable. It is possible, for example, that the `nameLt` function and `SalaryLt` class are not used anywhere else in the program.

When callbacks are tuned to the specific context in which they are used, they become both more complicated and less re-usable. Let's say, for example, that we wish to count the number of employees whose salary is above the average for the collection. Using Standard Library algorithms, this task seems trivial: (1) sum all of the salaries using `std::accumulate`, (2) calculate the average salary by dividing this sum by the total number of employees, and (3) count the number of employees with above-average salaries using `std::count_if`. Unfortunately, both `std::accumulate` and `std::count_if` require callbacks to return the salary for an `Employee` and to supply the criterion for counting, respectively. The callback for `std::accumulate` must take two parameters — the current running sum and an element from the sequence being summed — and must return the new running sum:

```
struct SalaryAccumulator
{
    long operator()(long currSum, const Employee& e) const
        // returns the sum of currSum and the salary field of e
    {
        return currSum + e.salary;
    }
};
```

The callback for `std::count_if` is a **predicate** (i.e., an expression that yields a Boolean result in response to a yes-or-no question) that takes a single argument and returns **`true`** if an element of that value should be counted and **`false`** otherwise. In this case, we are concerned with `Employee` object's having salaries above the average. Our **predicate functor** must, therefore, carry around that average so that it can compare it to the salary of the employee that is presented as an argument:

```
class SalaryIsGreater  // function object constructed with a reference salary
{
    const long d_referenceSalary;

public:
    explicit SalaryIsGreater(long rs) : d_referenceSalary(rs) { }
        // construct with a reference salary, rs

    bool operator()(const Employee& e) const
        // return true if the salary for Employee e is greater than the
        // reference salary specified on construction, false otherwise
    {
        return e.salary > d_referenceSalary;
    }
};
```

Note that, unlike our previous functor classes, `SalaryIsGreater` has a member variable, i.e., it has *state*. This member variable must be initialized, necessitating a constructor. Its call operator compares its input argument against this member variable to compute the predicate value.

With these two functor classes defined, we can finally implement the simple three-step algorithm for determining the number of employees with salaries greater than the average:

```
#include <algorithm>  // std::count_if
#include <numeric>    // std::accumulate

std::size_t numAboveAverageSalaries(const std::vector<Employee>& employees)
{
    const long sum = std::accumulate(employees.begin(), employees.end(), 0L,
                                     SalaryAccumulator());

    const long average = sum / employees.size();
    return std::count_if(employees.begin(), employees.end(),
                         SalaryIsGreater(average));
```

```
}
```

The first statement creates an object of the `SalaryAccumulator` class and passes that object to the `std::accumulate` algorithm to produce the sum of all of the salaries. The second statement divides the sum by the size of the `employees` collection to compute the average salary. The third statement creates an object of the `SalaryIsGreater` class and passes it to the `std::count_if` algorithm to compute the result. Note that the local variable, `average`, is used to initialize the reference value in the `SalaryIsGreater` object.

We now turn our attention to a syntax that allows us to rewrite these examples much more simply and compactly. Returning to the sorting example, the rewrite has the name-comparison and salary-comparison operations expressed in-place, within the call to `std::sort`:

```
void sortByName2(std::vector<Employee>& employees)
{
    std::sort(employees.begin(), employees.end(),
              [](const Employee &e1, const Employee &e2)
              {
                  return e1.name < e2.name;
              });
}

void sortBySalary2(std::vector<Employee>& employees)
{
    std::sort(employees.begin(), employees.end(),
              [](const Employee &e1, const Employee &e2)
              {
                  return e1.salary < e2.salary;
              });
}
```

In each case, the third argument to `std::sort` — beginning with `[]` and ending with the nearest closing `}` — is called a **lambda expression**. Intuitively, for this case, one can think of a lambda expression as an *operation* that can be invoked as a callback by the algorithm. The example shows a function-style parameter list — matching that expected by the `std::sort` algorithm — and a function-like body that computes the needed predicate. Using lambda expressions, a developer can express a desired operation directly at the point of use rather than defining it elsewhere in the program.

The compactness and simplicity afforded by the use of lambda expressions is even more evident when we rewrite the average-salaries example:

```
std::size_t numAboveAverageSalaries2(const std::vector<Employee>& employees)
{
    const long sum = std::accumulate(employees.begin(), employees.end(), 0L,
                                     [](long currSum, const Employee& e)
                                     {
                                         return currSum + e.salary;
                                     });

    const long average = sum / employees.size();
    return std::count_if(employees.begin(), employees.end(),
```

```
                        [average](const Employee& e)
                        {
                            return e.salary > average;
                        });
}
```

The first lambda expression, above, specifies the operation for adding another salary to a running sum. The second lambda expression returns true if the `Employee` argument, `e`, has a salary that is larger than `average`, which is a local variable *captured* by the lambda expression. A **lambda capture** is a set of local variables that are usable within the body of the lambda expression, effectively making the lambda expression an extension of the immediate environment. We will look at the syntax and semantics of lambda captures in more detail in Parts of a Lambda Expression.

Note that the lambda expressions replaced a significant portion of code that was previously expressed as separate functions or functor classes. The fact that some of that code reduction is in the form of documentation (comments) increases the appeal of lambda expressions to a surprising degree. Creating a named entity such as a function or class imposes on the developer the responsibility to give that entity a meaningful name and sufficient documentation for a future human reader to understand its *abstract* purpose, outside the context of its use, even for one-off, non-reusable entities. Conversely, when an entity is defined right at the point of use, it might not need a name at all, and it is often self-documenting, as in both the sorting and average-salaries examples above. Both the original creation and maintenance of the code is simplified.

-of-a-lambda-expression

Parts of a lambda expression

A lambda expression has a number of parts and subparts, many of which are optional. For exposition purposes, let's look at a sample lambda expression that contains all of the parts:

```
   introducer
  ______^______
 /             \
[&, cap2, cap3](T1 arg1, T2 arg2) mutable noexcept -> R { /* ... */ }
 \_____ _____/ \_________________ _________________/ \____ ____/
       V                        V                        V
    capture                declarator                  body
```

Evaluating a lambda expression creates a temporary **closure** object of an unnamed type called the **closure type**. Each part of a lambda expression is described in detail in the subsections below.

closures

Closures

A lambda expression looks a lot like an unnamed function definition, and it is often convenient to think of it that way, but a lambda expression is actually more complex than that. First and foremost, a lambda expression, as the name implies, is an *expression* rather than a *definition*. The result of evaluating a lambda expression is a special function object

called a closure[1]; it is not until the closure is *invoked* — which can happen immediately but usually occurs later (e.g., as a callback) — that the actual body of the lambda expression gets evaluated.

Evaluating a lambda expression creates a temporary **closure object** of an unnamed type called the closure type. The closure type encapsulates captured variables (see Section 2.2."Lambda Captures" on page 606) and has a call operator that executes the body of the lambda expression. Each lambda expression has a unique closure type, even if it is identical to another lambda expression in the program. If the lambda expression appears within a template, the closure type for each instantiation of that template is unique. Note, however, that, although the closure object is an unnamed temporary object, it can be saved in a named variable whose type can be queried. Closure types are copy constructible and move constructible, but they have no other constructors and have deleted assignment operators.[2] Interestingly, it is possible to *inherit* from a closure type, provided the derived class constructs its closure type base class using only the default or move constructors. This ability to derive from a closure type is convenient when implementing certain library features such as `std::bind`, which take advantage of the empty-base optimization:

```
#include <utility>  // std::move

template <typename Func>
int callFunc(const Func& f) { return f(); }

void f1()
{
    int   i   = 5;
    auto  c1  = [i]{ return 2 * i; };   // OK, deduced type for c1
    using C1t = decltype(c1);           // OK, named alias for unnamed type
    C1t   c1b = c1;                     // OK, copy of c1
    auto  c2  = [i]{ return 2 * i; };   // OK, identical lambda expression
    using C2t = decltype(c2);
    C1t   c2b = c2;                     // Error, different types, C1t & C2t
    using C3t = decltype([]{/* ... */});  // Error, lambda expr within decltype

    class C1Derived : public C1t        // OK, inherit from closure type
    {
        int d_auxValue;

    public:
        C1Derived(C1t c1, int aux) : C1t(std::move(c1)), d_auxValue(aux) { }
        int aux() const { return d_auxValue; }
    };

    int ret = callFunc([i]{ return 2 * i; });  // OK, deduced arg type, Func
```

[1]The terms *lambda* and *closure* are borrowed from *Lambda Calculus*, a computational system developed by Alonzo Church in the 1930s. Many computer languages have features inspired by Lambda Calculus, although most (including C++) take some liberties with the terminology. See **?** and **?**.

[2]C++17 provides default constructors for empty-capture lambdas. Empty-capture lambdas are assignable in C++20.

```
    c1b = c1;  // Error, assignment of closures is not allowed.
}
```

The types of `c1` and `c2`, above, are different, even though they are token-for-token identical. As there is no way to explicitly name a closure type, we use **`auto`** (in the case of `c1` and `c2` in `f1`) or template-argument deduction (in the case of `f` in `callFunc`) to create variables directly from the lambda expression, and we use **`decltype`** to create aliases to the types of existing closure variables (`C1t` and `C2t`). Note that using **`decltype`** directly on a lambda expression is ill formed, as shown with `C3t`, because there would be no way to construct an object of the resulting unique type. The derived class, `C1Derived`, uses the type alias `C1t` to refer to its base class. Note that its constructor forwards its first argument to the base-class move constructor.

There is no way to specify a closure type prior to creating an actual closure object of that type. Consequently, there is no way to declare `callFunc` with a parameter of the actual closure type that will be passed; hence, it is declared as a template parameter. As a special case, however, if the lambda capture is *empty* (i.e., the lambda expression begins with `[]`; see Section 2.2."Lambda Captures" on page 606), then the closure is implicitly convertible to an ordinary function pointer having the same signature as its call operator:

```
char callFunc2(char (*f)(const char*)) { return f("x"); }  // not a template

char c = callFunc2([](const char* s) { return s ? s[0] : '\0'; });
    // OK, closure argument is converted to function-pointer parameter

char d = callFunc2([c](const char* s) { /* ... */ });
    // Error, lambda capture is not empty; no conversion to function pointer
```

The `callFunc2` function takes a callback in the form of a pointer to function. Even though it is not a template, it can be called with a lambda argument having the same parameter types, the same return type, and an empty lambda capture; the closure object is converted to an ordinary pointer to function. This conversion is *not* available in the second call to `callFunc2` because the lambda capture is not empty.

Conversion to function pointer is considered a user-defined conversion operator and thus cannot be implicitly combined with other conversions on the same expression. It can, however, be invoked *explicitly*, as needed:

```
using Fp2 = int(*)(int);  // function-pointer type

struct FuncWrapper
{
    FuncWrapper(Fp2) { /* ... */ }  // implicit conversion from function-pointer
    // ...
};

int f2(FuncWrapper)  { /* ... */ return 0; }
int i2 = f2([](int x) { return x; });  // Error, two user-defined conversions
int i3 = f2(static_cast<Fp2>([](int x) { return x; }));  // OK, explicit cast
int i4 = f2(+[](int x) { return x; });  // OK, forced conversion
```

The first call to `f2` fails because it would require two implicit user-defined conversions: one from the closure type to the `Fp2` function-pointer type and one from `Fp2` to `FuncWrapper`. The second call succeeds because the first conversion is made explicit with the **`static_cast`**. The third call is an interesting shortcut that takes advantage of the fact that unary **`operator+`** is defined as the identity transformation for pointer types. Thus, the closure-to-pointer conversion is invoked for the operand of **`operator+`**, which returns the unchanged pointer, which, in turn, is converted to `FuncWrapper`; the first and third steps of this sequence use only one user-defined conversion each. The Standard Library `std::function` class template provides another way to pass a function object of unnamed type, one that does not require the lambda capture to be empty; see *Use Cases* on page 419.

The compile-time and runtime phases of defining a closure type and constructing a closure object from a single lambda expression resembles the phases of calling a function template; what looks like an ordinary function call is actually broken down into a compile-time instantiation and a runtime call. The closure type is deduced when a lambda expression is encountered during compilation. When the control flow passes through the lambda expression at run time, the closure object is *constructed* from the list of captured local variables. In the `numAboveAverageSalaries` example in the Description section, the `SalaryIsGreater` class can be thought of as a closure type — created by hand instead of by the compiler — whereas the call to `SalaryIsGreater(average)` is analogous to constructing the closure object at run time.

Finally, the purpose of a closure is to be invoked. It can be invoked immediately by supplying arguments for each of its parameters:

```
#include <iostream>  // std::cot
void f3()
{
    [](const char* s) { std::cout << s; }("hello world\n");
        // equivalent to std::cout << "hello world\n";
}
```

The closure object, in this example, is invoked immediately and then destroyed, making the above just a complicated way to say `std::cout << "hello world\n";`. More commonly, the lambda expression is used as a local function for convenience and to avoid clutter:

```
#include <cmath>  // std::sqrt

double hypotenuse(double a, double b)
{
    auto sqr = [](double x) { return x * x; };
    return std::sqrt(sqr(a) + sqr(b));
}
```

Note that there is no way to overload calls to closures:

```
auto sqr = [](int x) { return x * x; };     // OK, store **closure** in sqr
auto sqr = [](double x) { return x * x; };  // Error, redefinition of sqr
```

The most common use of a lambda expression, however, is as a callback to a function template, e.g., as a functor argument to an algorithm from the Standard Library:

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

template <typename FwdIt>
FwdIt oddEvenPartition(FwdIt first, FwdIt last)
{
    using value_type = decltype(*first);
    return std::partition(first, last, [](value_type v) { return v % 2 != 0; });
}
```

The `oddEvenPartition` function template moves odd values to the start of the sequence and even values to the back. The closure object is invoked repeatedly within the `std::partition` algorithm.

Lambda capture and lambda introducer

e-and-lambda-introducer

The purpose of the lambda capture is to make certain local variables from the environment available to be used (or, more precisely, **ODR-used**, which means that they are used in a potentially-evaluated context) within the **lambda body**. Each local variable can be **captured by copy** or **captured by reference**. Orthogonally, each variable can be **explicitly captured** or **implicitly captured**. When a lambda expression appears within a non-static member function, the **`this`** pointer can be captured as a special case. We'll examine each of these aspects of lambda capture in turn. An extension to lambda capture in C++14 is discussed in *C++14 init capture* on page 410.

Syntactically, the lambda capture consists of an optional **capture default** followed by a comma-separated list of zero or more identifiers (or the keyword **`this`**), which are explicitly captured. The capture default can be one of `=` or `&` for capture by copy or capture by reference, respectively. If there is a capture default, then **`this`** and any local variables in scope that are ODR-used within the lambda body and not explicitly captured will be implicitly captured.

```
void f1()
{
    int a = 0, b = 1, c = 2;
    auto c1 = [a, b]{ return a + b; };
        // a and b are explicitly captured.
    auto c2 = [&]{ return a + b; };
        // a and b are implicitly captured.
    auto c3 = [&, b]{ return a + b; };
        // a is implicitly captured and b is explicitly captured.
    auto c4 = [a]{ return a + b; }
        // Error, b is ODR-used but not captured.
}
```

The Standard defines the **lambda introducer** as the lambda capture together with its surrounding `[` and `]`. If the lambda introducer is an empty pair of brackets, no variables will be captured.

```
auto c1 = []{ /* ... */ };  // Empty **lambda capture**
```

The lambda capture enables access to portions of the local stack frame. As such, only variables with *automatic storage duration* — i.e., non-static local variables — can be captured, as we'll see in detail later in this section and in lambda body. An explicitly captured variable whose name is immediately preceded by an `&` symbol in the lambda capture is captured by reference; without the `&`, it is captured by copy. If the capture default is `&`, then all implicitly captured variables are captured by reference. Otherwise, if the capture default is `=`, all implicitly captured variables are captured by copy:

```
void f2a()
{
    int a = 0, b = 1;
    auto c1 = [&a]{ /* ... */ return a; };  // a captured by reference
    auto c2 = [a] { /* ... */ return a; };  // a captured by copy
    auto c3 = [a, &b] { return a + b; };
        // a is explicitly captured by copy and b is explicitly
        // captured by reference.
    auto c4 = [=]{ return a + b; };
        // a and b are implicitly captured by copy.
    auto c5 = [&]{ return &a; };
        // a is implicitly captured by reference.
    auto c6 = [&, b]{ return a * b; };
        // a is implicitly captured by reference and b is explicitly
        // captured by copy.
    auto c7 = [=, &b]{ return a * b; };
        // a is implicitly captured by copy and b is explicitly
        // captured by reference.
    auto c8 = [a]{ return a * b; };
        // Error, a is explicitly captured by copy, but b is not captured.
    auto c9 = [this]{ /* ... */ };  // Error, no this in nonmember function
}

class Class1a
{
public:
    void mf1()
    {
        auto c12 = [this]{ return this; };  // Explicitly capture this.
        auto c13 = [=]   { return this; };  // Implicitly capture this.
    }
};
```

Redundant captures are not allowed; the same name (or **`this`**) cannot appear twice in the lambda capture. Moreover, if the capture default is `&`, then none of the explicitly captured variables may be captured by reference, and if the capture default is `=`, then any explicitly captured entities can be neither **explicitly copied** variables nor **`this`**[3]:

```
class Class1b
{
```

[3] C++20 removed the prohibition on explicit capture of **`this`** with an `=` capture default.

```
public:
    void mf1()
    {
        int a = 0;
        auto c1 = [a, &a]{ /* ... */ }; // Error, a is captured twice.
        auto c2 = [=, a]{ /* ... */ };
            // Error, explicit capture of a by copy is redundant.
        auto c3 = [&,&a]{ /* ... */ };
            // Error, explicit capture of a by reference is redundant.
        auto c4 = [=, this]{ return this; };
            // Error, explicit capture of this with = capture default
    }
};
```

We'll use the term *primary variable* to refer to the block-scope local variable outside of the lambda expression and *captured variable* to refer to the variable of the same name as viewed from within the lambda body. For every object that is captured by copy, the **lambda closure** will contain a member variable having the same name and type, after stripping any reference qualifier (except reference-to-function); this member variable is initialized from the primary variable by direct initialization and is destroyed when the closure object is destroyed. Any ODR-use of that name within the lambda body will refer to the closure's member variable. Thus, for an entity that is captured by copy, the primary and captured variables refer to distinct objects with distinct lifetimes. By default, the call operator is **const**, providing read-only access to members of the closure object (i.e., captured variables that are captured by copy); mutable (non**const**) call operators are discussed in *Lambda declarator* on page 411:

```
    assert


void f3()
{
    int a = 5;
    auto c1 = [a]        // a is captured by copy.
    {
        return a;        // return value of copy of a
    };
    a = 10;              // Modify a after it was captured by c1.
    assert(5 == c1());   // OK, a within c1 had value from before the change.

    int& b = a;
    auto c2 = [b]        // b is int (not int&) **captured by copy**.
    {
        return b;        // return value of copy of b
    };
    b = 15;              // Modify a through reference b.
    assert(10 == c2());  // OK, b within c2 is a copy, not a reference.

    auto c3 = [a]
    {
```

```
        ++a;            // Error, a is const within the lambda body.
    };
}
```

In the example above, the lambda expression is evaluated to produce a closure object, `c1`, that captures a *copy* of `a`. Even when the primary `a` is subsequently modified, the captured `a` in `c1` remains unchanged. When `c1` is invoked, the lambda body returns the *copy*, which still has the value 5. The same applies to `c2`, but note that the copy of `b` is *not a reference* even though `b` is a reference. Thus, the copy of `b` in `c2` is the value of `a` that `b` referred to at the time that `c2` was created.

When a variable is captured by reference, the captured variable is simply an alias to the primary variable; no copies are made. It is, therefore, possible to modify the primary variable and/or take its address within the lambda body:

```
    assert
void f4()
{
    int a = 5;
    auto c1 = [&a]      // a is **captured by reference**.
    {
        a = 10;         // Modify a through the captured variable.
        return &a;      // return address of captured a
    };
    assert(c1() == &a); // OK, primary and captured a have the same address.
    assert(10 == a);    // OK, primary a is now 10.

    int& b = a;
    auto c2 = [&b]      // b is **captured by reference**
    {
        return &b;      // return address of captured b
    };
    assert(c2() == &b); // OK, primary and captured b have the same address.
    assert(c2() == &a); // OK, captured b is an alias for a.
}
```

In contrast to the `f3` example, the `c1` closure object above does *not* hold a copy of the captured variable, `a`, though the compiler may choose to define a member of type `int&` that refers to `a`. Within the lambda body, modifying `a` modifies the primary variable, and taking its address returns the address of the primary variable, i.e., the captured variable is an alias for the primary variable. With respect to variables that are captured by reference, the lambda body behaves very much as though it were part of the surrounding block. The lifetime of a variable that is captured by reference is the same as that of the primary variable (since they are the same). In particular, if a copy of the closure object outlives the primary variable, then the captured variable becomes a *dangling reference*; see *Potential Pitfalls* on page 427.

If `this` appears in the lambda capture, then (1) the current `this` pointer is captured by copy and (2) within the lambda body, member variables accessible through `this` can be used without prefixing them with `this->`, as though the lambda body were an extension of the surrounding member function. The lambda body cannot refer to the closure directly;

the captured **this** does not point to the closure but to the ***this** object of the function within which it is defined:

```
    assert
```

```
struct Class1
{
    int d_value;

public:
    // ...
    void mf() const
    {
        auto c1 = []{ return *this; };        // Error, this is not captured.
        auto c2 = []{ return d_value; };      // Error, this is not captured.
        auto c3 = [d_value]{ /* ... */; };    // Error, cannot capture member
        auto c4 = [this]{ return this; };     // OK, returns this
        auto c5 = [this]{ return d_value; }; // OK, returns this->d_value
        assert(this == c4());                 // OK, captured this is Class1.
    }
};
```

Note that `c4` returns **this**, which is the address of the `Class1` for which `mf` was called. This is one way in which the closure type is different from a named **functor type** — there is no way for an object of closure type to refer to itself directly. Because the closure type is unnamed and because it does not supply its own **this** pointer, it is difficult (but not impossible) to create a *recursive* lambda expression; see **Usage examples** [**there is no section with this name**] .

If **this** is captured (implicitly or explicitly), the lambda body will behave much like an extension of the member function in which the lambda expression appears, with direct access to the class's members:

```
    std::count_ifstd::vector
```

```
class Class2
{
    int d_value;

public:
    std::size_t mem(const std::vector<int>& v) const
    {
        auto f = []{ return d_value; };
            // Error, this not captured; can't see d_value.
        return std::count_if(v.begin(), v.end(),
                             [this](int element){ return element < d_value; });
            // OK, uses this->d_value.
    }
};
```

Note that capturing **this** does not copy the class object that it points to; the original **this** and the captured **this** will point to the same object:

```
    assert

class Class3
{
    int d_value;

public:
    void mf()
    {
        auto c1 = [this]{ ++d_value; };  // increment this->d_value
        d_value = 1;
        c1();
        assert(2 == d_value);            // change to d_value is visible
    }
};
```

Here, we captured **this** in `c1` but then proceeded to modify the object pointed to by **this** within the lambda body.[4]

A lambda expression can occur wherever other expressions can occur, including within other lambda expressions. The set of entities that can be captured in a valid lambda expression depends on the surrounding scope. A lambda expression that does not occur immediately within **block scope** cannot have a lambda capture:

```
namespace ns1
{
    int v = 10;
    int w = [v]{ /* ... */ return 0; }();
        // Error, capture in global/namespace scope

    void f4(int a = [v]{ return v; }());  // Error, capture in default argument
}
```

When a lambda expression occurs in block scope, it can capture any local variables with *automatic* (i.e., non-static) storage duration in its **reaching scope**. The Standard defines the reaching scope of the lambda expression as the set of enclosing scopes up to and including the innermost enclosing function and its parameters. Static variables can be used without capturing them; see *Lambda body* on page 416:

```
void f5(const int& a)
{
    int b = 2 * a;
    if (a)
    {
        int c;
        // ...
    }
    else
    {
```

[4] In C++17, it is possible to capture ***this**, which results in the entire class object being copied, not just the **this** pointer.

```
            int d = 4 * a;
        static int e = 10;

        auto c1 = [a]{ /* ... */ };    // OK, capture argument a from f5.
        auto c2 = [=]{ return b; };  // OK, implicitly capture local b.
        auto c3 = [&c]{ /* ... */ };   // Error, c is not in **reaching scope**.
        auto c4 = [&]{ d += 2; };    // OK, implicitly capture local d.
        auto c5 = [e]{ /* ... */ };    // Error, e has static duration.
    }

    struct LocalClass
    {
        void mf()
        {
            auto c6 = [b]{ /* ... */ };  // Error, b not in **reaching scope**
        }
    };
}
```

The reaching scope of the lambda expressions for `c1` through `c5`, above, includes the local variable `d` in the **`else`** block, `b` in the surrounding function block, and `a` from `f5`'s arguments. The local variable, `c`, is not in their reaching scope and cannot be captured. Although `e` *is* in their reaching scope, it cannot be captured because it does not have automatic storage duration. Finally, the lambda expression for `c6` is within a member function of a local class. Its reaching scope ends with the innermost function, `LocalClass::mf`, and does not include the surrounding block that includes `a` and `b`.

Only when the innermost enclosing function is a non-static class member function can **`this`** be captured:

```
void f5()
{
    auto c1 = [this]{ /* ... */ };  // Error, f5 is not a member function.
}

class Class3
{
    static void mf1()
    {
        auto c2 = [this]{ /* ... */ };  // Error, mf1 is static.
    }

    void mf2()
    {
        auto c3 = [this]{ /* ... */ };  // OK, mf2 is non-static member function

        struct LocalClass
        {
            static void mf3()
            {
```

```
                auto c4 = [this]{ /* ... */ };
                    // Error, innermost function, mf3, is static
            }
        };
    }
};
```

When a lambda expression is enclosed within another lambda expression, then the reaching scope includes all of the intervening lambda bodies. Any variable captured (implicitly or explicitly) by the inner lambda expression must be either defined or captured by the enclosing lambda expression:

```
void f6()
{
    int a, b, c;
    const char* d;
    auto c1 = [&a]                  // capture a from function block
    {
        int d;                       // local definition of d hides outer def
        auto c2 = [&a]{ /* ... */ };  // OK, a is captured in enclosing lambda
        auto c3 = [d]{ /* ... */ };   // OK, capture int d from enclosing
        auto c4 = [&]{ return d; };   // OK,    "       "     "      "
        auto c5 = [b]{ /* ... */ };   // Error, b is not captured in enclosing
    };
    auto c6 = [=]
    {
        auto c7 = [&]{ return b; };
            // OK, ODR-use of b causes implicit capture in c7 and c6.
        auto c8 = [&d]{ return &d; };
            // d is captured by copy in c6; c8 returns address of copy
    };
}
```

Note that there are two variables named `d`: one at function scope and one within the body of the first lambda expression. Following normal rules for unqualified name lookup, the inner lambda expressions used to initialize `c3` and `c4` capture the *inner* `d` (of type **`int`**), not the *outer* `d` (of type **`const char*`**). Because it is not captured, primary variable `b` is *visible* but not *usable* — an important distinction that we'll discuss in *Lambda body* on page 416 — within the body of `c1` and cannot, therefore, be captured by `c5`.

The lambda body for `c7` ODR-uses `b`, thus causing it to be implicitly captured. This capture by `c7` constitutes an ODR-use of `b` within the enclosing lambda expression, `c6`, in turn causing `b` to be implicitly captured by `c6`. In this way, a single ODR-use can trigger a *chain* of implicit captures from an enclosed lambda expression through its enclosing lambda expressions. Critically, when a variable is captured by copy in one lambda expression, any enclosed lambda expressions that capture the same name will capture the *copy*, not the primary variable, as we see in the lambda expression for `c8`.

Note that, when a variable is named in a lambda capture, it isn't automatically *captured*. A variable is not captured unless it is ODR-used within the lambda expression:

```
void f7()
```

```
{
    int       a = 0;
    int const b = 2;
    auto c1 = [a]{ return 2 * a; };      // OK, a is explicitly captured.
    auto c2 = [a]{ return 0; };          // Warning, no uses of a
    auto c3 = [a]{ return sizeof(a); };  // Warning, sizeof(a) isn't an ODR-use
    auto c4 = [b]{ return b; };          // Warning, value of b isn't an ODR-use
    auto c5 = [&b]{ return &b; };        // OK, address of b is an ODR-use
}
```

In the above example, the lambda body for `c1` contains an ODR-use of `a` and thus captures `a`. Conversely, `c2` does *not* capture `a` because the lambda body for `c2` does not contain an ODR-use of `a`; most compilers will issue a warning diagnostic about the superfluous presence of `a` in the lambda capture for `c2`. The case of `c3` is a bit more subtle. Although `a` is *used* within `c3`, it is not ODR-used — i.e., it is not used in a potentially-evaluated context because **`sizeof`** does not evaluate its argument — so the warning is the same as for `c2`; see *Potential Pitfalls* on page 427. The last two cases are a bit more subtle: The use of the *value* of a **`const`** variable (in `c4`) is *not* an ODR-use of that variable, whereas the use its *address* (in `c5`) *is* an ODR-use.

Finally, a lambda capture within a **variadic function template** (see Section 2.1."Variadic Templates" on page 519) may contain a **parameter pack expansion**:

```
#include <utility>  // std::forward

template <typename... ArgTypes>
int f8(const char* s, ArgTypes&&... args);

template <typename... ArgTypes>
int f9(ArgTypes&&... args)
{
    const char* s = "Introduction";
    auto c1 = [=]{ return f8(s, args...); };  // OK, args... captured by copy
    auto c2 = [s,&args...]{ return f8(s, std::forward<ArgTypes>(args)...); };
        // OK, explicit capture of args... by reference
}
```

In the example above, the variadic arguments to `f9` are implicitly captured using capture by copy in the first lambda expression. This means that, regardless of the **value category** (*rvalue*, *lvalue*, and so on) of the original arguments, the captured variables are all *lvalue* members of the resulting *closure*. Conversely, the second lambda expression captures the set of arguments using capture by reference, again resulting in captured variables that are *lvalues*. The `ArgTypes` parameter pack expansion designates a list of *types*, not *variables*, and does not, therefore, need to be captured to be used within the lambda expression, nor would it be valid to attempt to capture it. Because `ArgTypes` is specified using a forwarding reference (`&&`, see Section 2.1."Forwarding References" on page 351), the Standard Library function, `std::forward`, can be used to cast the captured variables to the value category of their corresponding arguments.

c++14-init-capture

C++14 init capture

TODO VR: this subsection shouldn't be here, we have a specific C++14 feature 'lambdacapture' for this stuff.

Each item in a C++11 lambda capture is either a capture default or a **simple capture** consisting of the name of a local variable, either by itself (for capture by copy) or preceded by an `&` (for capture by reference). C++14 introduces another possibility, an **init capture**, consisting of a variable name and an initializer, which creates a new captured variable initialized to an arbitrary expression. The initializer can be either preceded by an `=` token or can be a braced initialization (see Section 2.1."Braced Init" on page 198). The newly defined variable does not necessarily share the name or type of a primary local variable:

```
void f1(int a, bool bits[])
{
    double g(int);

    int b;
    auto c1 = [i{5}]{ /* ... */ };        // Define int i = 5.
    auto c2 = [b=bits[a]]{ /* ... */ };   // Define bool b = copy of bits[a].
    auto c3 = [&r=a, b]{ /* ... */ };     // int& r = reference to a. Copy b.
    auto c4 = [&, p=bits+1]{ *p = !a; };  // bool* p = pointer to bits[1].
    auto c5 = [x=g(a)]{ return x; };      // double x = g(a).
}
```

In the lambda expression for `c1`, above, the init capture defines a new variable, `i`, initialized with the value 5. Within the lambda body, this `i` is indistinguishable from any other variable captured by the closure, but, outside of the lambda body, it differs from a simple capture in that it does not capture a local variable. The lambda capture for `c2` similarly defines a new variable, `b`, initialized from an expression involving `bits` and `a`, but does not capture either of them. Once captured, `bits[a]` can change without affecting the value of `b`. An init capture can also define a variable of *reference* type, as shown in the init capture for `c3`. The lambda capture for `c3` also shows an init capture mixed in with simple captures. Note, however, that the capture default, if any, has no effect on the init capture, as shown in the lambda expression for `c4`. The init capture for `c5` shows an arbitrary expression being captured in this the return value of a function call.

The variable defined in an init capture is defined as if by the declaration:

```
auto *init-capture* ;
```

The hypothetical variable created by such a definition is unique and separate from any similarly named variable in the environment. This uniqueness allows the same name to appear on both the left and right of the `=` symbol:

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

void f2(std::string s)
{
   float a = 1.2;
   auto c1 = [&s=s]{ /* ... */ };            // effectively capture by reference
```

```
    auto c2 = [a=static_cast<double>(a)]{ /* ... */ };
    auto c3 = [s=std::move(s)]{ /* ... */ };  // effectively capture by move
    std::string s2 = s;                       // BAD IDEA, s is moved from
}
```

In all of the lambda expressions above, an init capture defines a variable whose name shadows one in the environment. In fact, the first use, in `c1`, defines a captured variable, `s`, that is a reference to the argument variable by the same name, yielding a behavior within the lambda body that is essentially indistinguishable from normal capture by reference. The second lambda expression copies the local variable, `a`, into a captured variable such that, within the lambda body, the variable `a` is a **`double`**. Finally, the lambda capture for `c3` creates a captured variable, `s`, initialized from the argument `s` using the `std::string` *move constructor* (see Section 2.1."*rvalue* References" on page 479), simulating something that many considered to be missing in C++11: capture by move. Note, however, that after constructing the `c3` closure, the local variable, `s`, is in a *moved-from* state and thus has an unspecified value, even if `c3` is never invoked.

In C++14, an init capture cannot be a parameter pack expansion.[5]

lambda-declarator

Lambda declarator

The **lambda declarator** looks a lot like a function declaration and is effectively the declaration of the closure type's call operator. The lambda declarator comprises the call operator's parameter list, mutability, exception specification, and return type:

```
[/* ... */](T1 arg1, T2 arg2) mutable   noexcept(/* ... */) -> RetType { /* ... */ }
           \________ _______/ \__ __/   \_______ _______/   \_____ ___/
                    V           V             V                 V
            parameter list   mutability   exception spec     return type
           \_____________________________ _____________________________/
                                         V
                                 lambda declarator
```

Although the lambda declarator looks very similar to a function declaration, we cannot forward declare any part of a lambda expression; we can only define it.

The entire lambda declarator is optional. However, if *any* part is present, the parameter list must be present (even if it declares no parameters):

```
auto c1 = [](int x) noexcept { /* ... */ };  // OK, param list and exception spec
```

[5] A syntax for parameter pack expansion of init captures was introduced in C++20:

```
template <typename T> class X { /* ... */ };
template <typename T> X<T> f3(T&&);

template <typename... ArgTypes>
void f4(ArgTypes&&... args)
{
    auto c1 = [...x=f3(std::forward<ArgTypes>(args))]{ /* ... */ };
        // Error in C++14.  OK in C++20
}
```

```
auto c2 = []() -> int { return 5; };          // OK, with ret type empty param list
auto c3 = [] -> double { /* ... */ };         // Error, ret type with no param list
```

The parameter list for a lambda expression is the same as a parameter list for a function declaration, with minor modifications.

1. A parameter is not permitted to have the same name as an explicitly-captured variable:

   ```
   void f1()
   {
       int a;
       auto c1 = [a](short* a){ /* ... */ };  // Error, parameter shadows captured a.
       auto c2 = [](short* a){ /* ... */ };   // OK, parameter hides local a.
       auto c3 = [=](short* a){ /* ... */ };  // OK, local a is not captured.
   }
   ```

   In the definition of `c1`, the lambda expression explicitly captures `a`, then improperly tries to declare a parameter by the same name. When `a` is not captured, as in the lambda expression for `c2`, having a parameter named `a` does not pose a problem; within the lambda body, the declaration of `a` in the parameter list will prevent name lookup from finding the declaration in the enclosing scope. The situation with `c3` is essentially the same as for `c2`; because name lookup finds `a` in the parameter list rather than the enclosing scope, it does not attempt to capture it.

2. In C++11, none of the parameters may have default arguments. This restriction does not apply to C++14 and after:

   ```
   auto c4 = [](int x, int y = 0){ /* ... */ };  // Error in C++11.  OK in C++14.
   ```

3. If the type of any of the parameters contains the keyword **`auto`**, then the lambda expression becomes a **generic lambda**; see Section 2.2."*Generic* Lambdas" on page 605. Everything in this chapter applies to generic lambda as well as regular lambda expressions, so it is recommended that you read the rest of this chapter before moving on to the generic lambdas feature.

Note that, unlike the lambda capture, the parameter list is usually dictated by the *client* of a lambda expression rather than by its author. Moreover, the lambda capture is evaluated only once when the closure is created, whereas the parameter list is bound to actual arguments each time the call operator is invoked:

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

template <typename InputIter, typename Func>
void applyToEveryOtherElement(InputIter start, InputIter last, Func f)
    // For elements in the range [start, last), invoke f on the first
    // element, skip the second element, etc., alternating between calling f
    // and skipping elements.
{
    while (start != last)
```

```
    {
        f(*start++);                    // Process one element.
        if (start != last) { start++; }  // Skip one element.
    }
}

void f2(const std::vector<float>& vec)
{
    std::size_t size = vec.size();
    applyToEveryOtherElement(vec.begin(), vec.end(),
                [](float x){ /* ... */ });         // OK, one float parameter
    applyToEveryOtherElement(vec.begin(), vec.end(),
                [](float x, int y){ /* ... */ });  // Error, too many parameters
    applyToEveryOtherElement(vec.begin(), vec.end(),
                [size](){ /* ... */ });            // Error, too few parameters
    applyToEveryOtherElement(vec.begin(), vec.end(),
                [size](double x){ /* ... */ });    // OK, convertible from float
}
```

In the above definition of `applyToEveryOtherElement`, the callable argument, `f`, is applied to half of the elements of the input range and has the closure type resulting from a lambda expression. In this example, therefore, each instantiation of `applyToEveryOtherElement` calls its `f` parameter multiple times with a single argument of type **`float`**. In the first call to `applyToEveryOtherElement`, the lambda closure has a parameter list consisting of a single parameter of type **`float`** and is thus compatible with the expected signature of `f`. The second and third calls to `applyToEveryOtherElement` supply a lambda closure with too many and too few parameters, respectively, resulting in a compilation error at the point where `f` is called. The last call to `applyToEveryOtherElement` supplies a lambda closure that takes a single argument of type **`double`**. Since an argument of type **`float`** is convertible to **`double`**, this lambda expression is also a valid argument to `applyToEveryOtherElement`. Note that the presence or absence of a lambda closure makes no difference to the validity of the lambda expression from the point of view of the client function.

The **`mutable`** keyword, if present, indicates that the call operator for the closure type should *not* be **`const`**. Recall that a normal class member function can modify the class's members if not declared **`const`**. The call operator is just an ordinary member function in this regard:

```
class Class1
{
    int d_value;

public:
    // ...
    void operator()(int v)       { d_value = v; }  // OK, object is mutable
    void operator()(int v) const { d_value = v; }  // Error, object is const
};
```

The **`const`** version of **`operator()`** cannot modify the member variables of the `Class1` object, whereas the undecorated one can. The call operator for a closure type has the inverse default

constness: the call operator is implicitly **const** *unless* the lambda declarator is decorated with **mutable**. In practice, this rule means that member variables of the closure object, i.e., variables that were captured by copy, are **const** by default and cannot be modified within the lambda body unless the **mutable** keyword is present:

```
    assert
void f2()
{
    int a = 5;
    auto c1 = [a]()         { return ++a; };  // Error, copy of a is const
    auto c2 = [a]() mutable { return ++a; };  // OK, increment *copy* of a
    assert(6 == c2());                        // OK, captured a incremented
    assert(5 == a);                           // OK, primary a not changed
    assert(7 == c2());                        // OK, captured a incremented
}
```

The two lambda expressions are identical except for the **mutable** keyword. Both use capture by copy to capture local variable `a`, and both try to increment `a`, but only the one decorated with **mutable** can perform that modification. When the call operator on closure object `c2` is invoked, it increments the *captured copy* of `a`, leaving the primary `a` untouched. If `c2()` is invoked again, it increments its copy of `a` a second time. Using capture by reference allows the primary variable to be changed within the lambda body regardless of the presence or absence of the **mutable** keyword. Similarly, member variables accessed through **this** are unaffected by the **mutable** keyword, but they *are* affected by the **const**ness of the surrounding member function:

```
    assert
class Class2
{
    int d_value;
public:
    // ...
    void mf()
    {
        d_value = 1;
        int a = 0;
        auto c1 = [&a,this]{
            a = d_value;   // OK, a is a reference to a non-const object.
            d_value *= 2;  // OK, this points to a non-const object.
        };
        c1();
        assert(1 == a && 2 == d_value);  // values updated by c1
        c1();
        assert(2 == a && 4 == d_value);  // values updated by c1 again
    }

    void cmf() const
    {
        int a = 0;
        auto c2 = [=]() mutable {
```

```
            ++a;        // OK, increment mutable **captured variable**
            ++d_value;  // Error, *this is const within cmf.
        };
    }
};
```

The lambda expression for `c1` is not decorated with **`mutable`** yet both `a` and `d_value` can be modified; the first because it was captured by reference, and the second because it was accessed through **`this`** within a non**`const`** member function, `mf`. Conversely, the lambda expression for `c1` cannot modify `d_value` even though it is declared **`mutable`** because the captured **`this`** pointer points to a **`const`** object within the surrounding member function, `cmf`.

The lambda declarator may include an **exception specification**, consisting of a **`throw`** or **`noexcept`** clause, after the **`mutable`** decoration, if any. The syntax and meaning of the exception specification is identical to that of a normal function; see Section 3.1."**`noexcept`** Specifier" on page 676.

The return type of the call operator can be determined either by a **trailing return type** or by a **deduced return type**. Every example we have seen up to this point has used a deduced return type whereby the return type of the closure's call operator is deduced by the type of object returned by the return statement(s). If there are no return statements in the lambda body or if the return statements have no operands, then the return type is **`void`**. If there are multiple return statements, they must agree with respect to the return type. The rules for a deduced return type are the same for a lambda expression as they are in C++14 for an ordinary function; see Section 3.2."Deduced Return Type" on page 687:

```
void f3()
{
    auto c1 = [](int& i){ i = 0; };   // deduced return type is void
    auto c2 = []{ return "hello"; };  // deduced return type is const char*
    auto c3 = [](bool c)              // deduced return type is int
    {
        if (c) { return 5; }
        else   { return 6; }
    };
    auto c4 = [](bool c)
    {
        if (c) { return 5; }          // deduced return type is int
        else   { return 6.0; }        // Error, double does not match int.
    };
}
```

All four of the above lambda expressions have a deduced return type. The first one deduces a return type of **`void`** because the lambda body has no **`return`** statements. The next one deduces a return type of **`const char*`** because the string literal, `"hello"`, decays to a **`const char*`** in that context. The third one deduces a return type of **`int`** because all of the **`return`** statements return values of type **`int`**. The last one fails to compile because the two branches return values of different types.[6]

[6]The original C++11 Standard did not allow a deduced return type for a lambda body containing

If a deduced return type is impossible or undesirable (see Section 3.2."Deduced Return Type" on page 687 for a description of why this feature needs to be used with care), a trailing return type can be specified (see Section 1.1."Trailing Return" on page 112:

std::pair

```
void f4()
{
    auto c1 = [](bool c) -> double {
        if (c) { return 5; }          // OK, int value converted to double
        else   { return 6.0; }        // OK, double return value
    };
    auto c2 = []() -> std::pair<int, int> {
        return { 5, 6 };              // OK, brace-initialize returned pair
    };
}
```

In the first lambda expression above, we specify a trailing return type of **double**. The two branches of the **if** statement would return different types (**int** and **double**), but, because the return type has been definitively declared, the compiler converts the return values to the known return type (**double**). The second lambda expression returns a value by brace initialization, which is insufficient for deducing a return value. Again, the ambiguity is resolved by declaring the return value explicitly. Note that, unlike ordinary functions, a lambda expression cannot have a return type specified before the lambda introducer or lambda declarator:

```
auto c5 = int [](int x){ return 0; };     // Error, return type misplaced
auto c6 = [] int (int x){ return 0; };    // Error, return type misplaced
auto c7 = [](int x) -> int{ return 0; };  // OK, trailing return type
```

Attributes (see Section 1.1."Attribute Syntax" on page 10) that appertain to the *type* of call operator can be inserted in the lambda declarator just before the trailing return type. If there is no trailing return type, the attributes can be inserted before the open brace of the lambda body. Unfortunately, these attributes do not apply to the call operator itself, but to its type, ruling out some common attributes:

```
#include <cstdlib>  // std::abort
auto c1 = []() noexcept [[noreturn]] {  // Error, [[noreturn]] on a type
    std::abort();
};
```

Lambda body

lambda-body

Combined, the lambda declarator and the lambda body make up the declaration and definition of an **inline** class member function that is the call operator for the closure type. For the purposes of name lookup and the interpretation of **this**, the lambda body is considered to be in the context where the lambda expression is evaluated (independent of the context where the closure's call operator is invoked).

anything other than a single **return** statement. This restriction was lifted by a defect report and is no longer part of C++11. Compiler versions that predate ratification of this defect report might reject lambda expressions having multiple statements and a deduced return type.

Critically, the set of entity names that can be used from within the lambda body is not limited to captured local variables. Types, functions, templates, constants, and so on — just like any other member function — do not need to be captured and, in fact, *cannot* be captured in most cases. To illustrate, let's create a number of entities in multiple scopes:

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

namespace ns1
{
    void f1() { std::cout << "ns1::f1" << '\n'; }
    struct Class1 { Class1() { std::cout << "ns1::Class1()" << '\n'; } };
    int g0 = 0;
}

namespace ns2
{
    void f1() { std::cout << "ns2::f1" << '\n'; }

    template <typename T>
    struct Class1 { Class1() { std::cout << "ns2::Class1()" << '\n'; } };

    int const g1 = 1;
    int       g2 = 2;

    class Class2
    {
        int        d_value;  // non-static member variable
        static int s_mem;    // static member variable

        void mf1() { std::cout << "Class2::mf1" << '\n'; }

        struct Nested { Nested() { std::cout << "Nested()" << '\n'; } };

        template <typename T>
        static void print(const T& v) { std::cout << v << '\n'; }

    public:
        explicit Class2(int v) : d_value(v) { }

        void mf2();
        void mf3();
        void mf4();
        void mf5();
    };

    int Class2::s_mem = 0;
}
```

Namespace `ns1` contains three global entities: function `f1`, class `Class1`, and variable `g0`. Namespace `ns2` contains global variables `g1` and `g2`, function `f1`, and classes `Class1` and

`Class2`. Within `Class2`, we have non-static member variable `d_value`, non-static member function `mf1`, static member function template `print`, and public member functions `mf2` through `mf5`.

With these declarations, we first demonstrate the use of entities that are not variables and are accessible within the scope of a lambda body:

```
void ns2::Class2::mf2()
{
    using LocalType = const char*;

    auto c1 = []{
        // Access non-variables in scope
        f1();           // find global ns2::f1 by unqualified name lookup.
        Class1<int> x1; // construct ns2::Class1<int> object.
        Nested      x2; // construct ns2::Class2::Nested object.
        print("print"); // call static member function ns2::Class2::print.
        LocalType   x3; // declare object of local type.
        ns1::f1();      // find global ns1::f1 func by qualified name lookup
        ns1::Class1 x4; // find global ns1::Class1 type by qualified lookup
    };
}
```

We can see that, within the lambda body, non-variables can be accessed normally, using either unqualified name lookup or, if needed, qualified name lookup. Unqualified name lookup will find global entities within the namespace; types and static functions within the class; and types declared within the enclosing function scope. Qualified name lookup will find entities in other namespaces.

Variables with static storage duration can also be accessed directly, without being captured:

```
void ns2::Class2::mf3()
{
    static int s1 = 3;
    auto c1 = []{
        // Access variables with static storage duration.
        print(g1);      // print global constant ns2::g1
        print(g2);      // print global variable ns2::g2
        print(ns1::g0); // print global variable ns1::g0
        print(s_mem);   // print static member variable s_mem
        print(s1);      // print static local variable s1
    };
}
```

Here we see global constants, global variables, static member variables, and local static variables being used from the local scope.

Next, we look at uses of variables with *automatic* storage duration from the lambda expression's surrounding block scope:

```
void ns2::Class2::mf4()
{
```

```
    int       a = 4;
    int       b = 5;
    int const k = 6;

    auto c1 = [](double b) {
        // Access local variables within the reaching scope.
        print(a);    // Error, a is ODR-used but not captured.
        print(b);    // OK, b argument hides b defined in mf1.
        int kb = k;  // OK, const variable is not ODR-used.
        print(k);    // Error, k is ODR-used but not captured.
    };
}
```

The attempt to print `a` will fail because `a` is a non-static local variable within the surrounding scope but is not captured. Printing `b` is not a problem because the `b` parameter in the lambda declarator is local to the lambda body; it hides the `b` variable in the surrounding block scope. A **`const`** variable of fundamental type is not ODR-used unless its *identity* (i.e., address) is needed. Hence, `k` can be used even though it is not captured. Conversely, the call to `print(k)` *is* an ODR-use of `k` because `print` takes its argument by *reference*, which requires taking the *address* of `k`, which, in turn, makes it an ODR-use. Since `k` is a local variable with automatic storage duration that was not captured, `print(k)` is ill formed.

Finally, we look at access to (non-static) members of the surrounding class:

```
void ns2::Class2::mf5()
{
    auto c1 = []{
        // Access non-static members
        print(d_value); // Error, this is ODR-used but not captured.
        mf1();          // Error,    "   "        "       "   "     "
    };
    auto c2 = [=] {
        print(d_value); // OK, this is captured; print this->d_value.
        mf1();          // OK,   "    "     "    ; call this->mf1().
    };
}
```

The above shows that member variables and functions can be accessed only if **`this`** is (implicitly or explicitly) captured by the lambda expression, as is the case for `c2` but not for `c1`.

Use Cases

use-cases-lambda

Interface adaptation, partial application, and currying

olication,-and-currying

Lambda expressions can be used to adapt the set of arguments provided by an algorithm to the parameters expected by another facility:

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

```
extern "C" int f1(const char* s, std::size_t n);

void f2(const std::vector<std::string>& vec)
{
    std::size_t n = std::count_if(vec.begin(), vec.end(),
        [](const std::string& s){ return 0 != f1(s.data(), s.size()); });
    // ...
}
```

Here we have a function, `f1`, that takes a C string and length and computes some predicate, returning 0 for false and nonzero for true. We want to use this predicate with `std::count_if` to count how many strings in a specified vector match this predicate. The lambda expression in `f2` adapts `f1` to the needs of `std::count_if` by converting a `std::string` argument into `const char*` and `std::size_t` arguments and converting the `int` return value to `bool`.

A particularly common kind of interface adaptation is **partial application**, whereby we reduce the *parameter count* of a function by holding one or more of its arguments constant for the duration of the algorithm:

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

template <typename InputIter, typename T>
bool all_greater_than(InputIter first, InputIter last, const T& v)
    // returns true if all the values in the specified range [first, last)
    // are greater than the specified v, and false otherwise
{
    return std::all_of(first, last, [&](const T& i) { return i > v; });
}
```

In the example above, the greater-than operator (`>`) takes two operands, but the `std::all_of` algorithm expects a functor taking a single argument. The lambda expression passes its single argument as the first operand to `operator>` and *binds* the other operand to the captured `v` value, thus solving the interface mismatch.

Finally, let's touch on **currying**, a transformation borrowed from lambda calculus and functional programming languages. Currying is a flexible way to get results similar to partial application by transforming, e.g., a function taking two parameters into one taking just the first parameter and returning a function taking just the second parameter. To apply this technique, we define a lambda expression whose call operator returns another lambda expression, i.e., a closure that returns another closure:

```
    all_of
template <typename InputIter, typename T>
bool all_greater_than2(InputIter first, InputIter last, const T& v)
    // returns true if all the values in the specified range [first, last)
    // are greater than the specified v, and false otherwise
{
    auto isGreaterThan = [](const T& v){
        return [&v](const T& i){ return i > v; };
    };
    return std::all_of(first, last, isGreaterThan(v));
}
```

The example above is another way to express the previous example. The call operator for `isGreaterThan` takes a single argument, `v`, and returns another single-argument closure object that can be used to compare `i` to `v`. Thus, `isGreaterThan(v)(i)` is equivalent to `i > v`.

ulating-local-functions

Emulating local functions

Local functions in languages other than C++ allow functions to be defined within other functions. They are useful when the outer function needs to repeat a set of steps two or more times but where the repeated steps are meaningless outside of the immediate context and/or require access to the outer function's local variables. Using a lambda expression to produce a re-usable closure provides this functionality in C++:

```
class Token { /* ... */ };

bool parseToken(const char*& cursor, Token& result)
    // Parse the token at cursor up to the next space or end-of-string,
    // setting result to the resulting token value.  Advance cursor to the
    // space or to the null terminator and return true on success.  Reset
    // cursor to its original value, set result to an empty token, and
    // return false on failure.
{
    const char* const initCursor = cursor;
    auto error = [&]
    {
        cursor = initCursor;
        result = Token{};
        return false;
    };
    // ...
    if (*cursor++ != '.')
    {
        return error();
    }
    // ...
}
```

The `error` closure object acts as a local function that performs all of the necessary error processing and returns **`false`**. Using this object, every error branch can be reduced to a single statement, **`return`** `error()`. Without lambda expressions, the programmer would likely resort to defining a custom class to store the parameters, using a **`goto`**, or, worse, cutting-and-pasting the three statements shown within the lambda body.

ined-control-constructs

Emulate user-defined control constructs

Using a lambda expression, an algorithm can look almost like a new control construct in the language:

```
#include <mutex>  // std::mutex
#include <vector>  // std::vector
```

```
template <typename RandomIter, typename F>
void parallel_foreach(RandomIter first, RandomIter last, const F& op)
    // For each element, e, in [first, last), create a copy opx of op,
    // and invoke opx(e).  Any number of invocations of opx(e) may occur
    // concurrently, each using a separate copy of op.
{ /*...*/ }

void processData(std::vector<double>& data)
{
    double       beta    = 0.0;
    double const init = 7.45e-4;
    std::mutex m;

    parallel_foreach(data.begin(), data.end(), [&, init](double e) mutable
    {
        if (e < 1.0)
        {
           // ...
        }
        else
        {
           // ...
        }
    });
}
```

The `parallel_foreach` algorithm is intended to act like a **`for`** loop except that all of the elements in the input range may potentially be processed in parallel. By inserting the "body" of this "parallel for loop" directly into the call to `parallel_foreach`, the resulting loop looks and feels a lot like a built-in control construct. Note that the capture default is capture by reference and will result in all of the iterations sharing the outer function's call frame, including the mutex variable, `m`, used to prevent race conditions. This capture default should be used with care in parallel computations, which often use capture by copy to deliberately *avoid* sharing. If an asynchronous computation might outlive its caller, then using capture by copy is a must for avoiding dangling references; see *Potential Pitfalls* on page 427.

tructs-in-expressions

Variables and control constructs in expressions

In situations where a single expression is required — e.g., member-initializers, initializers for **`const`** variables, and so on — an *immediately evaluated* lambda expression allows that expression to include local variables and control constructs such as loops:

```
#include <climits>  // SHRT_MAX

bool isPrime(long i);
    // Return true if i is a prime number.
```

```
const short largestShortPrime = []{
    for (short v = SHRT_MAX; ; v -= 2) {
        if (isPrime(v)) return v;
    }
}();
```

The value of `largestShortPrime` must be set at initialization time because it is a **`const`** variable with static duration. The loop inside of the lambda expression computes the desired value, using local variable, `v`, and a **`for`** loop. Note that the call operator for the resulting closure object is immediately invoked via the `()` argument list at the end of the lambda expression; the closure object is never stored in a named variable and goes out of scope as soon as the full expression is completely evaluated. This computation would formerly have been possible only by creating a single-use *named* function.

use-with-std::function

Use with `std::function`

As convenient as a lambda expression is for passing a functor to an algorithm *template*, the fact that each closure type is unnamed and distinct makes it difficult to use them outside of a generic context. The C++11 Standard Library class template, `std::function`, bridges this gap by providing a polymorphic invocable type that can be constructed from any type with a compatible invocation prototype, including but not limited to closure types.

A simple interpreter for a postfix input language stores a sequence of instructions in a `std::vector`. Each instruction can be of a different type, but they all accept the current stack pointer as an argument and return the new stack pointer as a result. Each instruction is typically a small operation, ideally suited for being expressed as lambda expressions:

```
#include <cstdlib>     // std::strtol
#include <functional>  // std::function
#include <string>      // std::string
#include <vector>      // std::vector

using Instruction = std::function<long*(long*& sp)>;

std::vector<Instruction> instructionStream;

std::string nextToken();                // read the next token
char tokenOp(const std::string& token);  // operator for token

void readInstructions()
{
    std::string token;
    Instruction nextInstr;
    while (!(token = nextToken()).empty())
    {
        switch (tokenOp(token))
        {
            case 'i':
            {
                // Integer literal
```

```
                long v = std::strtol(token.c_str(), nullptr, 10);
                nextInstr = [v](long* sp){ *sp++ = v; return sp; };
                break;
            }
            case '+':
            {
                // + operation
                nextInstr = [](long*& sp){
                    long v1 = *--sp;
                    long v2 = *--sp;
                    *sp++ = v1 + v2;
                    return sp;
                };
                break;
            }
            // ... more cases
        }

        instructionStream.push_back(nextInstr);
    }
}
```

The `Instruction` type alias is a `std::function` that can hold, through a process called **type erasure**, any invocable object that takes a `long*` argument and returns a `long*` result. The `readInstructions` function reads successive string tokens and switches on the operation represented by the token. If the operation is `i`, then the token is an integer literal. The string token is converted into a `long` value, `v`, which is captured in a lambda expression. The resulting closure object is stored in the `nextInstr` variable; when called, it will push `v` onto the stack. Note that the `nextInstr` variable outlives the primary `v` variable, but, because `v` was captured by copy, the **captured variable**'s lifetime is the same as the closure object's. If the next operation is `+`, `nextInstr` is set to the closure object of an entirely different lambda expression, one that captures nothing and whose call operator pops two values from the stack and pushes their sum back onto the stack.

After the `switch` statement, the current value of `nextInstr` is appended to the instruction stream. Note that, although each closure type is different, they all can be stored in an `Instruction` object because the prototype for their call operator matches the prototype specified in the instantiation of `std::function`. The `nextInstr` variable can be created empty, assigned from the value of a lambda expression, and then later reassigned from the value of a different lambda expression. This flexibility makes `std::function` and lambda expressions a potent combination.

One specific use of `std::function` worth noting is to return a lambda expression from a non-template function:

`std::function`

```
std::function <int(int)> add_n(int n)
{
    return [n](int i) { return n + i; };
}
```

```
int result = add_n(3)(5);  // result is 8.
```

The return value of `add_n` is a closure object wrapped in a `std::function` object. Note that `add_n` is not a template and that it is not called in a template or **`auto`** context. This example illustrates a runtime-polymorphic way to achieve currying; see the earlier example in *Interface adaptation, partial application, and currying* on page 419.

event-driven-callbacks

Event-driven callbacks

Event-driven systems tend to have interfaces that are littered with callbacks:

```
#include <memory>  // std::unique_ptr

class DialogBox { /* ... */ };

template <typename Button1Func, typename Button2Func>
std::unique_ptr<DialogBox> twoButtonDialog(const char* prompt,
                                           const char* button1text,
                                           Button1Func button1callback,
                                           const char* button2text,
                                           Button2Func button2callback)
{
    // ...
}
```

The `twoButtonDialog` factory function takes three strings and two callbacks and returns a pointer to a dialog box having two buttons. The dialog-box logic invokes one of the two callbacks, depending on which of the two buttons is pressed. These callbacks are often quite small pieces of code that can best be expressed directly in the program logic using lambda expressions:

```
void runModalDialogBox(DialogBox& db);

void launchShuttle(/* ... */)
{
    bool doLaunch = false;

    std::unique_ptr<DialogBox> confirm =
        twoButtonDialog("Are you sure you want to launch the shuttle?",
                        "Yes", [&]{ doLaunch = true; },
                        "No",  []{});

    runModalDialogBox(*confirm);

    if (doLaunch)
    {
        // ... launch the shuttle!
    }
}
```

Here, the user is being prompted as to whether or not to launch a missile. Since the dialog box is processed entirely within the `launchShuttle` function, it is convenient to express two callbacks in-place, within the function, using lambda expressions. The first lambda expression — passed as the callback for when the user clicks "Yes" — captures the `doLaunch` flag by reference and simply sets it to **`true`**. The second lambda expression — passed as the callback for when the user clicks "No" — does nothing, leaving the `doLaunch` flag having its original **`false`** value. The simplicity of these callbacks come from fact that they are effectively extensions of the surrounding block and, hence, have access (via the lambda capture) to block-scoped variables such as `doLaunch`.

`recursion`

Recursion

A lambda expression cannot refer to itself, so creating one that is recursive involves using one of a number of different possible workarounds. If the lambda capture is empty, recursion can be accomplished fairly simply by converting the lambda expression into a plain function pointer stored in a **`static`** variable:

```
void f1()
{
    static int (*const fact)(int) = [](int i)
    {
        return i < 2 ? 1 : i * fact(i-1);
    };

    int result = fact(4);  // computes 24
}
```

In the above example, `fact(n)` returns the factorial of `n`, computed using a recursive algorithm. The variable, `fact`, becomes visible before its initializer is compiled, allowing it to be called from within the lambda expression. To enable the conversion to function pointer, the lambda capture must be empty; hence, `fact` must be static so that it can be accessed without capturing it.

If a recursive lambda expression is desired with a nonempty lambda capture, then the entire recursion can be enclosed in an outer lambda expression:

```
void f2(int n)
{
    auto permsN = [n](int m) -> int
    {
        static int (*const imp)(int, int) = [](int x, int m) {
            return m <= x ? m : m * imp(x, m - 1);
        };
        return imp(m - n + 1, m);
    };

    int a = permsN(5);  // permutations of 5 items, n at a time
    int b = permsN(4);  // permutations of 4 items, n at a time
}
```

In this example, `permsN(m)`, returns the number of permutations of `m` items taken `n` at a time, where `n` is captured by the closure object. The implementation of `permsN` defines a nested `imp` function pointer that uses the same technique as `fact`, above, to achieve recursion. Since `imp` must have an empty lambda capture, everything it needs is passed in as arguments by the `permsN` enclosing lambda expression. Note that the `imp` pointer and the lambda expression from which it is initialized do not needed to be scoped inside of the `permN` lambda expression; whether such nesting is desirable is a matter of taste.

In C++14, additional approaches to recursion (e.g., the "Y Combinator" borrowed from lambda calculus[7]) are possible due to using generic lambdas; see Section 2.2."*Generic* Lambdas" on page 605.

tential-pitfalls-lambda

Potential Pitfalls

dangling-references

Dangling references

Closure objects can capture references to local variables and copies of the **`this`** pointer. If a copy of the closure object outlives the stack frame in which it was created, these references can refer to objects that have been destroyed. The two ways in which a closure object can outlive its creation context are if (1) it is returned from the function or (2) it is stored in a data structure for later invocation:

```
#include <functional>  // std::function
#include <vector>      // std::vector

class Class1
{
    int d_mem;

    static std::vector<std::function<double(void*)> > s_workqueue;

    std::function<void(int)> mf1()
    {
        int local;
        return [&](int i) -> void { d_mem = local = i; };  // Bug, dangling refs
    }

    void mf2()
    {
        double local = 1.0;
        s_workqueue.push_back([&,this](void* p) -> double {
                return p ? local : double(d_mem);
            });  // Bug, dangling refs
    }
};
```

The example above uses `std::function` to hold closure objects, as described in *Use with* `std::function` on page 423. In member function `mf1`, the lambda body modifies both the local variable and the member variable currently in scope. However, as soon as the function

[7] ?

returns, the local variable goes out of scope and the closure contains a dangling reference. Moreover, the object on which it is invoked can also go out of scope while the closure object continues to exist. Modifying either `this->d_mem` or `local` through the capture is likely to corrupt the stack, leading to a crash, potentially much later in the program.

The member function `mf2`, rather than *return* a closure with dangling references, stores it in a data structure, i.e., the `s_workqueue` static `vector`. Once again, `local` and `d_mem` become dangling references and can result in data corruption when the call operator for the stored closure object is invoked. It is safest to capture `this` and use capture by reference only when the lifetime of the closure object is clearly limited to the current function. Implicitly captured `this` is particularly insidious because, even if the capture default is capture by copy, member variables are not copied and are often referenced without the `this->` prefix, making them hard to spot in the source code.

overuse

Overuse

The ability to write functions, especially functions with state, at the point where they are needed and without much of the syntactic overhead that accompanies normal functions and class methods, can potentially lead to a style of code that uses "lambdas everywhere", losing the abstraction and well-documented interfaces of separate functions. Lambda expressions are not intended for large-scale reuse. Sprinkling lambda expressions throughout the code can result in poor software-engineering practices such as cut-and-paste programming and the absence of cohesive abstractions.

on-captured-variables

Mixing captured and non-captured variables

A lambda body can access both automatic-duration local variables that were captured from the enclosing block and static-duration variables that need not and cannot be captured. Variables captured by copy are "frozen" at the point of capture and cannot be changed except by the lambda body (if mutable), whereas static variables can be changed independent of the lambda expression. This difference is often useful but can cause confusion when reasoning about a lambda expression:

```
    assert


void f1()
{
    static int a;
    int        b;

    a = 5;
    b = 6;

    auto c1 = [b]{ return a + b; };  // OK, b is **captured by copy**.
    assert(11 == c1());              // OK, a == 5 and b == 6.
    ++b;                             // Increment *primary* b.
    assert(11 == c1());              // OK, captured b did not change.
    ++a;                             // Increment static-duration a.
    assert(12 == c1());              // Bug, a == 6 and captured b == 6
```

```
}
```

When the closure object for `c1` is created, the captured `b` value is frozen within the lambda body. Changing the primary `b` has no effect. However, `a` is not captured (nor is it allowed to be). As a result, there is only *one* `a` variable, and modifying that variable outside of the lambda body changes the result of invoking the call operator.

C++14 **capture init** can be used to effectively capture a non-local variable:

```
assert
void f2()
{
    static int a;
    int        b;

    a = 5;
    b = 6;

    auto c1 = [a=a,b]{ return a + b; };  // OK, capture both a and b.
    ++a;                                 // Increment static-duration a.
    assert(11 == c1());                  // OK, captured a did not change.
}
```

The lambda capture, `[a=a,b]`, creates a new capture variable `a` that is initialized from the static variable `a` at the point that the lambda expression is evaluated to produce a closure object. The `a` variable within the lambda body refers to this captured variable, not to the static one.

Local variables in unevaluated contexts can yield surprises

xts-can-yield-surprises

To use a local variable, `x`, from the surrounding block as part of an unevaluated operand (e.g., **`sizeof`**`(x)` or **`alignof`**`(x)`), it is generally not necessary to capture `x` because it is not ODR-used within the lambda body. Whether or not `x` is captured, most expressions in unevaluated contexts behave as though `x` were *not* captured and the expression were evaluated directly in the enclosing block scope. This is itself surprising because, for example, a captured variable in a non-**`mutable`** lambda expression is **`const`**, whereas the primary variable might not be:

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

short s1(int&)       { return 0; }
int   s1(const int&) { return 0; }

void f1()
{
    int x = 0;  // x is a non-const lvalue.
    [x]{
        // captured x in non-mutable lambda is lvalue of type const int
        std::cout << sizeof(s1(x)) << '\n';  // prints sizeof(short)
        auto s1x = s1(x);                    // yields an int
        std::cout << sizeof(s1x) << '\n';    // prints sizeof(int)
    }();
}
```

The first print statement calls `s1(x)` in an unevaluated context, which ignores the captured `x` and returns the size of the result of `s1(int&)`. The next statement actually *evaluates* `s1(x)`, passing the *captured* `x` and calling `s1(const int&)` because the call operator is not decorated with **`mutable`**.

When using **`decltype`**`(x)`, the result is the declared type of the *primary* variable, regardless of whether or not `x` was captured. However, if `x` had been captured by copy, **`decltype`**`((x))` (with two sets of parentheses) would have yielded the *lvalue* type of the *captured* variable. There is some dispute as to what the correct results should be if `x` is *not* captured, with some compilers yielding the type of the primary variable and others complaining that it was not captured.

```
void f1()
{
    int x = 0;  // x is a non-const }}lvalue.
    auto c1 = [x]{ decltype((x)) y = x; };  // y has type const int&.
}
```

Finally, there is an unsettled question as to whether **`typeid`**`(x)` is an ODR-use of `x` and, therefore, requires that `x` be captured. Some compilers will complain about the following code:

```
#include <typeinfo>  // typeid
void f3()
{
    int x = 0;
    auto c1 = []() -> const std::type_info& { return typeid(x); };
        // Error, on some platforms ``x was not captured''
}
```

One can avoid this pitfall simply by calling **`typeid`** outside of the lambda, capturing the result if necessary:

```
#include <typeinfo>  // typeid
void f3()
{
    int x = 0;
    const std::type_info& xid = typeid(x);
        // OK, typeid called outside of lambda
    auto c1 = [&]() -> const std::type_info& { return xid; };
        // OK, return captured typeinfo
}
```

Annoyances

annoyances

Debugging

debugging

By definition, lambdas do not have names. Tools such as debuggers and stack-trace examiners typically display the compiler-generated names of the closure types instead of names selected by the programmer to clearly describe the purpose of a function, making it difficult to discern where a problem occurred.

t-capture-*this-by-copy

Can't capture `*this` by copy

A lambda expression can freeze the value of a surrounding local variable by using capture by copy, but no such ability is available directly to copy the object pointed to by **`this`**. In C++14, this deficiency can be mitigated using capture init:

```
class Class1
{
    int d_value;

    void mf1()
    {
        auto c1 = [self=*this]{ return self.d_value; };
    }
};
```

The lambda capture, `[self=*this]` creates a new captured variable, `self`, that contains a copy of `*this`. Unfortunately, accessing member variable `d_value` requires explicit use of `self.d_value`.

C++11 doesn't have capture init, so it is necessary to create a `self` variable external to the lambda expression and capture *that* variable[8]:

```
class Class1
{
    int d_value;

    void mf1()
    {
        Class1& self = *this;
        auto c1 = [self]{ return self.d_value; };
    }
};
```

deferred-execution-code

Confusing mix of immediate and deferred-execution code

The main selling point of lambda expressions — i.e., the ability to define a function object at the point of use — can sometimes be a liability. The code within a lambda body is

[8] As of C++17, `*this` can be captured directly with `this` within the lambda body pointing to the *copy* rather than the original:

```
class Class2
{
    int d_value;

    void mf1()
    {
        auto c1 = [*this]{ return d_value; };
            // C++17: return d_value from copy of *this
    }
};
```

typically not executed immediately but is deferred until some other piece of code, e.g., an algorithm, invokes it as a callback. The code that is immediately executed and the code whose invocation is deferred are visually intermixed in a way that could confuse a future maintainer. For example, let's look at a simplified excerpt from an earlier use case, *Use Cases — Use with* `std::function` on page 423.

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

void readInstructions()
{
    std::string                      token;
    std::function<long*(long*& sp)> nextInstr;

    while ( /* ... */ (!token.empty()))
    {
        switch (token[0])
        {
            // ... more cases
            case '+':
            {
                // + operation
                nextInstr = [](long*& sp){
                    long v1 = *--sp;
                    long v2 = *--sp;
                    *sp++ = v1 + v2;
                    return sp;
                };
                break;
            }
            // ... more cases
        }
       // ...
    }
}
```

A casual reading might lead to the assumption that operations such as `*--sp` are taking place within `case '+'`, when the truth is that these operations are encapsulated in a lambda expression and are not executed until the closure object is called (via `nextInstr`) in a relatively distant part of the code.

Trailing punctuation

`trailing-punctuation`

The body of a lambda expression is a *compound statement*. When compound statements appear elsewhere in the C++ grammar, e.g., as the body of a function or loop, they are not followed by punctuation. A lambda expression, conversely, is invariably followed by some sort of punctuation, usually a semicolon or parenthesis but sometimes a comma or binary operator. This difference between a lambda body and other compound statements makes this punctuation easy to forget:

```
auto c1 = []{ /* ... */ };  // <-- Don't forget the semicolon at the end.
```

The extra punctuation can also be unattractive when emulating a control construct using lambda expressions as in the `parallel_foreach` example in *Use Cases — Emulate user-defined control constructs* on page 421:

```
    std::vector

void f(const std::vector<int>& data)
{
    // ...
    for (int e : data)
    {
        // ...              for loop body
    }  // <-- no punctuation after the closing brace

    parallel_foreach(data.begin(), data.end(), [&](int e)
    {
        // ...              parallel loop body
    });  // <-- Don't forget the closing parenthesis and semicolon.
    // ...
}
```

In the above code snippet, the programmer would like the `parallel_foreach` algorithm to look as much like the built-in **`for`** loop as possible. However, the built-in **`for`** loop doesn't end with a closing parenthesis and a semicolon, whereas the `parallel_foreach` does, so the illusion of a language extension is incomplete.

`see-also`

See Also

- "**`decltype`**" (§1.1, p. 22) ♦ illustrates a form of type inference often used in conjunction with (or in place of) trailing return types.
- "Deduced Return Type" (§3.2, p. 687) ♦ shows a form of type inference that shares syntactical similarities with trailing return types, leading to potential pitfalls when migrating from C++11 to C++14.

`further-reading`

Further Reading

TODO

Asking if an Expression Cannot `throw`

noexceptoperator

placeholder text.....

Opaque Enumeration Declarations

enumeration-declaration

Any enumeration with a *fixed* **underlying type** can be declared without being defined, i.e., declared without its enumerators.

Description

description

Prior to C++11, enumerations could not be declared without the compiler having access to all of its enumerators, meaning that the definition of a specific **enum** had to be present in the **translation unit (TU)** prior to any declarations:

```
enum E0;                // Error, incomplete enum type
enum E1 { e_A1, e_B1 }; // OK, definition
enum E1;                // OK, redeclaration of existing enum in the same TU
```

Since C++11, enumerations can have a *fixed* underlying type (see Section 2.1."Underlying Type '11" on page 480), meaning that their integral representation does not depend on the values of their enumerators. Such enumerations can be declared without enumerators via an **opaque declaration**:

```
enum E2 : short;  // OK, opaque declaration with fixed char underlying type
enum E3;          // Error, opaque declaration without fixed underlying type
```

The declaration of the `E2` enumeration above gives the compiler enough information to know that the size and alignment of the type is **`sizeof(short)`** and **`alignof(short)`**, respectively, even though the enumerators have not yet been seen. Conversely, the size, alignment, and signedness of a classic enumeration such as `E3` is **implementation defined** and dependent on the specific enumerator values. The compiler cannot determine these properties until the enumerators are seen; hence, classic enumerations are not eligible for opaque declaration.

C++11 also introduced **scoped enumerations** (see Section 2.1."**`enum class`**" on page 310), declared with the keyword sequence **`enum class`** or **`enum struct`**. A scoped enumeration implicitly has an underlying type of **`int`** unless the user explicitly specifies a different underlying type. Because the underlying type of a scoped enumeration is always known at the point of declaration, it, too, can be declared with an opaque declaration:

```
enum class E4;          // OK, scoped enum, default int underlying type
enum class E5 : short;  // OK, scoped enum, fixed short underlying type
```

Within a single translation unit, the enumerators for an **enum** declared with an opaque declaration can be defined before the declaration, defined after it, or not provided at all:

```
enum E6 : unsigned { e_A6, e_A7 };  // OK, enum definition
enum E6 : unsigned;                 // OK, redeclaration of existing enum

enum E7 : int;                      // OK, opaque enum declaration
enum E7 : int { e_A7 };             // OK, enum definition

enum E8 : short;                    // OK, opaque enum declaration
```

All the declarations of an enumeration within a single TU must agree on its underlying type; otherwise, the program is ill formed:

```
enum class E9;              // OK, fixed default underlying type of int
enum class E9 : int;        // OK, underlying type matches previous declaration

enum E10 : short;           // OK, fixed explicit underlying type short
enum E10 : char { e_A10 };  // Error, redeclaration with different underlying type

enum class E11 : char;      // OK, fixed explicit underlying type char
enum class E11 : short;     // Error, redeclaration with different underlying type
```

Note that, as in C++03, multiple definitions of an enumeration are not allowed within a single TU:

```
enum E12 : char;            // OK, opaque enum declaration
enum E12 : char { e_A12 };  // OK, enum definition
enum E12 : char;            // OK, opaque enum redeclaration
enum E12 : char { e_A12 };  // Error, enum redefinition

enum class E13;             // OK, opaque enum declaration
enum class E13 { e_A13 };   // OK, enum definition
enum class E13;             // OK, opaque enum redeclaration
enum class E13 { e_A13 };   // Error, enum redefinition
```

An enumeration declared with an opaque declaration is a **complete type**. This means that, for example, we can request its size using **`sizeof`**, have a local, global, or member variable of the enumeration's type, and so on — all without having access to the enumeration's definition:

```
enum E14 : char;
static_assert(sizeof(E14) == 1, "");           // OK, sizeof of a complete type

enum class E15;
static_assert(sizeof(E15) == sizeof(int), "");  // OK, sizeof of a complete type

E14 a;  // OK, variable of a complete type
E15 b;  // OK,      "     " "     "      "

struct S {
    E14 d_e14;  // OK, data member of a complete type
    E15 d_e15;  // OK,   "     "    " "     "      "

    S(E14 e14, E15 e15)  // OK, by-value function arguments of complete types
    : d_e14(e14)
    , d_e15(e15)
    {
    }
};
```

Typical usage of opaque enumeration declarations often involves placing the **forward declaration** within a header and sequestering the complete definition within a correspond-

ing `.cpp` (or a second header). A forward declaration can insulate clients from changes to the enumerator list (see *Use Cases — Using opaque enumerations within a header file* on page 437):

```
// myclass.h:
// ...

class MyClass {
    // ...
private:
    enum class State;  // forward declaration of State enumeration
    State d_state;
};

// ...

// myclass.cpp:

#include <myclass.h>
// ...
enum class MyClass::State { e_STATE1, e_STATE2, e_STATE3 };
    // complete definition compatible with forward declaration of MyClass::State
```

Note that such a forward declaration is distinct from a **local declaration**. A forward declaration is characterized by having a translation unit that deliberately comprises both the definition and the opaque declaration of the enumeration. This translation unit can result either from their direct colocation in the same file or via the inclusion of a header in the corresponding implementation file (as in the example above). For a local declaration, no such translation unit exists:

```
// library.h:
// ...

enum class E18 : short { e_A18, /* ... */ e_Z18 };

// client.cpp:

// Note that 'library.h' is not included

enum class E18;  // BAD IDEA - a local opaque enumeration declaration
```

A local declaration, such as `E18` above, can be problematic; see *Potential Pitfalls — Redeclaring an externally defined enumeration locally* on page 449.

use-cases-opaqueenum

Use Cases

ns-within-a-header-file

Using opaque enumerations within a header file

Physical design involves two related but distinct notions of information *hiding*: **encapsu-**

lation[1,2] and **insulation**.[3,4] 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, merely to relink their code against updated libraries. The advantages of avoiding such **compile-time coupling** transcend simply 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_state;  // "opaque" but unconstrained int type (BAD IDEA)
    // ...

public:
    Proctor();
    // ...
};
```

Among other private members, `Proctor` has a data member, `d_state`, 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_state(e_STARTING) { /* ... */ }
// ...
```

Prior to C++11, enumerations could not be **forward declared**. To avoid the unnecessary exposition of the enumerators in the header file, a completely unconstrained **`int`** would be used as a data member, and the enumeration would be defined in the `.cpp` file. With the

[1] **?**
[2] **?**
[3] **?**, Chapter 6, pp. 327–471
[4] **?**, Sections 3.10–3.11, pp. 733–835

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_state(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 a local 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_state;  // 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. Then, 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();
    // ...
};
```

Next, we would then define 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 the header file of this example we first forward declared the nested **enum class** type within class scope and then separately defined a data member of the opaque enumerated type. We needed to do this in two statements because simultaneously *opaquely declaring* an enumeration 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 including the header file defining `Proctor`.

m-the-enumerator-list

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 480) 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 449. 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 450.

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) with the likely consequence of either a crash or, worse, the program continues to run 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`.[5]

[5]Using the compound *suffix* `fwd.h` (e.g., `comp.fwd.h`) for a forwarding header — instead of, say, `comp_fwd.h` or `comp.fh` — is advantageous in two ways. First, it preserves both the component's **base name** and the conventional **file extension** for headers on the host platform. Second, it informs the human reader that this is a *forwarding* header that may co-exist alongside a nonforwarding one. See **?**, section 2.4, pp. 297–333.

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 is 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 ...*/ };
// ...
```

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;
    // ...
};
```

To illustrate real-world practical use of the opaque-enumerations feature, consider the various components 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

[6] **?**, section 1.8, "The Depends-On Relation," 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.

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.

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.

Cookie: Insulating all external clients from the enumerator list

rom-the-enumerator-list

A commonly recurring **design pattern**, commonly known as the "Memento pattern,"[8] 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,[9] 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, if appropriate for the application.

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

[8] **?**, Chapter 5, section "Memento," pp. 283–291
[9] See also **?**, Chapter 5, section "Observer," pp. 293–303

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);
        // 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_isRunning;  // 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 engine;  // Create a configurable callback engine object.

   //...     (Configure the callback engine, e, as appropriate.)

   engine.registerInterest(&myCallback);  // Even a stateless function pointer can
```

```
                                        // be used with std::function.

    // ...create and register other clients for interest...

    engine.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 now
    {
        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;
        callbacks.swap(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 310), 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.

ial-pitfalls-opaqueenum

ned-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):

```
    std::uint8_t

// client.cpp:
#include <cstdint>  // std::uint8_t
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 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:

```
    std::uint16_t

// 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 450.

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:
extern int x;  // BAD IDEA!
// ...
```

```
// library.cpp:
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:
extern int x;  // ILL-FORMED PROGRAM
// ...
```

```
// library.cpp:
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.[10]

Inciting local enumeration declarations: an attractive nuisance

n-attractive-nuisance

Whenever we, as library component authors, provide the complete definition of an enumeration with a fixed underlying type and fail to provide a corresponding forwarding header having just the opaque declaration, we confront our clients with the difficult decision of whether to needlessly compile-time couple[11] 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 defined enumeration locally* on page 449. The maintainability pitfall associated with opaque

[10] **?**, sections 2.2.11–2.2.13, pp. 280–281

[11] 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.

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 complete definition — might be persuaded into declaring the enumeration locally!

Ensuring that library components that define enumerations whose enumerators can be elided also consistently provide a second forwarding header file containing the opaque declaration of each such enumeration would be one generally applicable way to sidestep this maintenance burden; see *Dual-Access: Insulating some external clients from the enumerator list* on page 440. Note that the attractive nuisance potentially exists even when the primary intent of the component is not to make the enumeration generally available.[12]

Annoyances

annoyances

Opaque enumerations are not completely type safe

ot-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

see-also

- "Underlying Type '11" (Section 2.1, p. 480) ♦ discusses the underlying integral representation for enumeration variables and their values.
- "**enum class**" (Section 2.1, p. 310) ♦ introduces an implicitly scoped, more strongly typed enumeration.

Further Reading

further-reading

- For more on declaration versus definition, header files, `.h` and `.cpp` pairs, extracting actual dependencies, the depends-on relation, logical and physical name cohesion, avoiding unnecessary compile-time dependencies, and architectural insulation techniques, see **?**.
- A complementary view of production software design can be found in **?**.

[12] **?**

Range-Based `for` Loops

range-based-for-loops

Range-Based **`for`** loops provide a simplified, more compact, syntax for iterating through a range of elements.

Description

description-rangefor

Iterating over the elements of a collection is a fundamental operation usually performed with a **`for`** loop:

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

void f1(const std::vector<std::string>& vec)
{
    for (std::vector<std::string>::const_iterator i = vec.begin();
         i != vec.end(); ++i)
    {
        // ...
    }
}
```

The code above iterates over the strings in a `std::vector`. It is significantly more verbose than similar code in other languages because it uses a very general-purpose construct, the **`for`** loop, to perform the specialized but common task of traversing a collection. In C++11, the definition of `i` can be simplified somewhat by using **`auto`**:

```
void f2(const std::vector<std::string>& vec)
{
    for (auto i = vec.begin(); i != vec.end(); ++i)
    {
        const std::string& s = *i;
        // ...
    }
}
```

Although **`auto`** does have a number of potential pitfalls, this use of **`auto`** to deduce the return type of `vec.begin()` is one of the safer idiomatic uses; see Section 2.1."**`auto`** Variables" on page 183. While this version of the loop is simpler to write, it still uses the fully general, three-part **`for`** construct. Moreover, it evaluates `vec.end()` each time through the loop.

The C++11 **ranged-based `for` loop** (sometimes colloquially referred to as the "foreach" loop) is a more concise loop notation tuned for traversing the elements of a container or other sequential range. A ranged-based **`for`** loop works with *ranges* and *elements* rather than *iterators* or *indexes*:

-range-based-for-loop

```
void f3(const std::vector<std::string>& vec)
{
    for (const std::string& s : vec)
    {
```

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

The loop in the above example can be read as "for each element `s` in `vec` …". There is no need to specify the name or type of the iterator, the loop-termination condition, or the increment clause; the syntax is focused purely on yielding (in order) each element of the collection for processing within the body of the loop. [1]

specification

Specification

The syntax for a ranged-based **for** loop declares a loop variable and specifies a range of elements to be traversed:

for (*for-range-declaration* : *range-expression*) *statement*

The compiler treats this high-level construct as though it were transformed into a lower-level **for** loop with the following pseudocode:

```
{
    auto&& __range = range-expression;
    for (auto __begin = begin-expr, __end = end-expr;
        __begin != __end;
        ++__begin)
    {
        for-range-declaration = *__begin;
        statement
    }
}
```

The variables `__range`, `__begin`, and `__end`, above, are for *exposition only*, i.e., the compiler does not necessarily generate variables with those names and user code is not permitted to access those variables directly.

The `__range` variable is defined as a **forwarding reference** (see Section 2.1."Forwarding References" on page 351); it will bind to any type of **range expression**, regardless of its **value category** (*lvalue* or *rvalue*). If the range expression yields a temporary object, its lifetime is extended, if necessary, until `__range` goes out of scope. While this **lifetime extension** of temporary objects works in most cases, it is insufficient when `__range` doesn't bind directly to the temporary created by the range expression, potentially resulting in subtle bugs; see Potential PitfallsLifetime of temporaries in the range expression.

The *begin-expr* and *end-expr* expressions used to initialize the `__begin` and `__end` variables, respectively, define a half-open range of elements starting with `*__begin` and including all of the elements in the `__range` up to but not including `*__end`. The precise meaning of *begin-expr* and *end-expr* were clarified in C++14 but were essentially the same in C++11[2]:

[1] In C++20, the syntax has been enhanced slightly to permit an optional leading variable declaration clause, e.g., `for (std::lock_guard g(myMutex); const std::string& s : vec) { /* ... */ }`.

[2] The rules for interpreting *begin-expr* and *end-expr* were slightly unclear in C++11. A defect report, CWG 1442, clarified the wording retroactively. C++14 clarified the wording further.

- If `__range` refers to an array, then *begin-expr* is the address of the first element of the array and *end-expr* is the address of one past the last element of the array.
- If `__range` refers to a class object and `begin` and/or `end` are members of that class, then *begin-expr* is `__range.begin()` and *end-expr* is `__range.end()`. Note that if `begin` or `end` are found in the class, then both of these expressions must be valid or else the program is ill formed.
- Otherwise, *begin-expr* is `begin(__range)` and *end-expr* is `end(__range)`, where `begin` and `end` are found using **argument-dependent lookup (ADL)**. Note that `begin` and `end` are looked up only in the namespaces associated with the expressions; names that are local to the context of the ranged-based `for` loop are not considered; see Annoyances.

Thus, a container such as `vector`, with conventional `begin` and `end` member functions, provides everything necessary for a ranged-based `for` loop, as we saw in the `f3` example on 452. Note that *end-expr* — `__range.end()` in the case of the `vector` — is evaluated only once, unlike the idiomatic low-level `for` loop, where it is evaluated prior to every iteration.

In C++11 and C++14, `__begin` and `__end` are required to have the same type.[3] Although the `__begin` and `__end` variables look and act like iterators, they need not conform to all of the iterator requirements in the Standard. Specifically, the type of `__begin` and `__end` must support prefix `operator++` but not necessarily postfix `operator++`, and it must support `operator!=` but not necessarily `operator==`.

The *for-range-declaration* declares the loop variable. Any declaration that can be initialized with `*__begin` will work. For instance, if `*__begin` returns a reference to a modifiable object of, e.g., `int` type, then `int j`, `int& j`, `const int& j`, and `long j` would all be valid *for-range-declaration*s declaring a loop variable `j`. Alternatively, the type of the loop variable can be deduced using `auto` — i.e., `auto j`, `auto& j`, `const auto& j`, or `auto&& j` (see Section 2.1."`auto` Variables" on page 183).

The sequence being traversed can be modified through the loop variable only if `__begin` returns a reference to a modifiable type and the loop variable is similarly declared as a reference to a modifiable type (e.g., `int&`, `auto&`, or `auto&&`). Note that the **for-range declaration** must define a *new* variable; unlike a traditional `for` loop, it cannot name an existing variable already in scope:

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

void f1(std::vector<int>& vec)
{
    const std::vector<int>& cvec = vec;

    for (auto&& i : cvec)
    {
        i = 0;  // Error, i is a reference to const int.
    }
```

[3] The C++17 Standard changes the defining code transformation of the range-based `for` loop so as to allow `__begin` and `__end` to have different types as long as they are comparable using `__begin != __end`.

```
    for (int j : vec)
    {
        j = 0;  // Bug, j is a loop-local variable; vec is not modified.
    }

    for (int& k : vec)
    {
        k = 0;  // OK, set element of vec to 0.
    }

    int m;
    for (     m : vec) { /* ... */ }  // Error, m does not define a variable.
    for (int& m : vec) { /* ... */ }  // OK, loop m hides function-scope m.
}
```

Since `cvec` is **`const`**, the element type returned by `*begin(cvec)` is **`const int&`**. Thus, `i` is deduced as **`const int&`**, making invalid any attempt to modify an element through `i`. The second loop is valid C++11 code but has a subtle bug: `j` is not a reference — it contains a *copy* of the current element in the `vector` — so modifying `j` has no effect on the vector. The third loop correctly sets all of the elements of `vec` to zero; the loop variable `k` is a reference to the current element, so setting it to zero modifies the original vector. The first `m` loop attempts to re-use local variable `m` as a loop variable; while this would be legal in a traditional **`for`** loop, it is ill formed in a ranged-based **`for`** loop. Finally, the last loop re-uses the name `m` from the surrounding scope, hiding the old name for the duration of the loop, just as it would in a traditional **`for`** loop.

The *statement* that makes up the loop body can contain anything that is valid within a traditional **`for`** loop body. In particular, a **`break`** statement will exit the loop immediately and a **`continue`** statement will skip to the next iteration.

Applying this transformation to the `f3` example (see 452) from the previous section, we can see how the ranged-based **`for`** loop hooks into the iterator idiom to traverse a `vector` of `string` elements:

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

void f3(const std::vector<std::string>& vec)
{
    // for (const std::string& s : vec) { /* ... */ }
    {
        auto&& __range = vec;  // reference to the std::vector
        for (auto __begin = begin(__range), __end = end(__range);
            __begin != __end;
            ++__begin)
        {
            const std::string& s = *__begin;  // Get current string element.
            {
                // ...
            }
```

```
        }
    }
}
```

In this expansion, `__range` has type `const std::vector<std::string>&` while `__begin` and `__end` have type `std::vector<std::string>::const_iterator`.

and-initializer-lists

Traversing arrays and initializer lists

The `<iterator>` standard header defines array overloads for `std::begin` and `std::end` such that, when applied to a C-style array, `arr`, having a known number of elements, `__bound`, `std::begin(arr)` returns the address of the first element of `arr` and `std::end(arr)` returns the address of one past that of the last element of `arr`, i.e., `arr + __bound`. This functionality is built into the initialization of `__begin` and `__end` as a special case, in the expansion of a range-based `for` loop, so that it is possible to traverse the elements of an array without needing to `#include <iterator>`:

```
void f1()
{
    double data[] = {1.9, 2.8, 4.7, 7.6, 11.5, 16.4, 22.3, 29.2, 37.1, 46.0};
    for (double& d : data)
    {
        d *= 3.0;  // triple every element in the array
    }
}
```

In the above example, the reference `d` is bound, in turn, to each element of the array. The size of the array is not encoded anywhere in the loop syntax, either as a literal or as a symbolic value, simplifying the specification of the loop and preventing errors. Note that only arrays whose size is known at the point where the loop occurs can be traversed this way:

```
extern double data[];  // array of unknown size

void f2()
{
    for (double& d : data)  // Error, data is an incomplete type.
    {
        // ...
    }
}

double data[10] = { /* ... */ };  // too late to make the above compile
```

The above example would compile if `data` were declared having a size, e.g., `extern double data[10]`, as that would be a complete type and provide sufficient information to traverse the array. The second definition of `data` in the example *is* complete but is not visible at the point that the loop is compiled.

An **initializer list** is typically used to initialize an array or container using **braced initialization**; see Section 2.1."Braced Init" on page 198. The `initializer_list` template

does, however, provide its own `begin` and `end` member functions and is, therefore, directly usable as the *range-expression* in a ranged-based **for** loop:

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

void f3()
{
    for (double v : {1.9, 2.8, 4.7, 7.6, 11.5, 16.4, 22.3, 29.2, 37.1, 46.0})
    {
        // ...
    }
}
```

The example above shows how a series of **double** values can be embedded right within the loop header.

Use Cases

use-cases

Iterating over all elements of a container

elements-of-a-container

The motivating use case for this feature is to loop over the elements in a container:

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

void process(int* p);

void f1()
{
    std::list<int> aList{ 1, 2, 4, 7, 11, 16, 22, 29, 37, 46 };

    for (int& i : aList)
    {
        process(&i);
    }
}
```

This idiom takes advantage of all STL-compliant container types providing `begin` and `end` operations, which may be used to delimit a range encompassing the entire container. Thus, the loop above iterates from `aList.begin()` to `aList.end()`, calling `process` on each element encountered.

When iterating over a `std::map<Key, Value>` or `std::unordered_map<Key, Value>`, each element has type `std::pair<const Key, Value>`. To save typing and to avoid errors related to the first member of the pair being **const** we use the `value_type` alias to refer to each element's type:

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

using MapType = std::map<std::string, int>;
```

```
MapType studentScores
{
    {"Emily", 89},
    {"Joel",  85},
    {"Bud",   86},
};

void printScores()
{
    for (MapType::value_type& studentScore : studentScores)
    {
        const std::string& student = studentScore.first;
        int&               score   = studentScore.second;
        std::cout << student << "\t scored " << score << '\n';
    }
}
```

This example prints each key/value pair in the `map`. We create two aliases, `student` for `studentScore.first` and `score` for `studentScore.second`, to better express the intent of the code.[4]

Subranges

subranges

Using a classic **`for`** loop to traverse a container, `c`, allows a subrange of `c` to be specified beginning at some point after `c.begin()` — e.g., `++c.begin()` — and/or ending at some point before `c.end()` — e.g., `std::next(c.end(), -3)`. To specify a subrange for a ranged-based **`for`** loop, we create a simple adapter to hold two iterators (or iterator-like objects) that define the desired subrange:

```
template <typename Iter>
class Subrange
{
    Iter d_begin, d_end;

public:
    using iterator = Iter;

    Subrange(Iter b, Iter e) : d_begin(b), d_end(e) { }

    iterator begin() const { return d_begin; }
    iterator end()   const { return d_end;   }
};

template <typename Iter>
Subrange<Iter> makeSubrange(Iter beg, Iter end) { return {beg, end}; }
```

[4] In C++17, **structured bindings** allow two variables to be initialized from a single `pair`, each variable being initialized by the respective `first` and `second` members of the `pair`. A ranged-based **`for`** loop using a structured binding for the loop variables yields a very clean and expressive way to traverse containers like `map` and `unordered_map`, e.g., using **`for`** (**`auto`**`& [student, score] : studentScores)`.

The `Subrange` class above is a primitive start to a potentially rich library of range-based utilities.[5] It holds two externally-supplied iterators that it can supply to a ranged-based **for** loop via its `begin` and `end` accessor members. The `makeSubrange` factory uses function template argument deduction to return a `Subrange` of the correct type.[6]

Let's use `Subrange` to traverse a vector in reverse, omitting its first element:

```
#include <vector>    // std::vector
#include <iostream>  // std::cout, std::endl

template <typename Range>
void printRange(const Range& r)
{
    for (const auto& elem : r)
    {
        std::cout << elem << ' ';
    }

    std::cout << std::endl;
}

std::vector<int> vec{16, 3, 1, 8, 99};

void f1()
{
    printRange(makeSubrange(vec.rbegin(), vec.rend() - 1));
        // print "99 8 1 3"
}
```

The `printRange` function template will print out the elements of any range, provided the element type supports printing to a `std::ostream`. In `f1`, we use reverse iterators to create a `Subrange` starting from the last element of `vec` and iterating backwards. By subtracting `1` from `vec.rend()`, we exclude the last element of the sequence, which is the first element of `vec`.

In fact, the iterators need not refer to a container at all. For example, we can use `std::istream_iterator` to iterate over "elements" in an input stream:

```
#include <istream>   // std::istream
#include <iterator>  // std::istream_iterator
#include <sstream>   // std::istringstream

void f2()
{
    std::istringstream inStream("1 2 4 7 11 16 22 29 37 bad 46");
    printRange(makeSubrange(std::istream_iterator<int>(inStream),
                            std::istream_iterator<int>()));
}
```

[5] The C++20 Standard introduces a new Ranges Library that provides powerful features for defining, combining, filtering, and manipulating ranges.

[6] C++17 introduced **class template argument deduction**, which significantly reduces the need for factory templates like `makeSubrange`.

In `f2`, the range being printed uses the `istream_iterator<T>` adapter template. Each time through the loop, the adapter reads another `T` item from its input stream. At end-of-file or if a read error occurs, the iterator becomes equal to the sentinel iterator, `istream_iterator<T>()`. Note that the ranged-based **`for`** loop feature and the `Subrange` class template do not require that the size of the subrange be known in advance.

Range generators

`range-generators`

Iterating over a range does not necessarily entail traversing existing data elements. A range expression could yield a type that *generates* elements as it goes. A useful example is the `ValueGenerator`, an iterator-like class that produces a sequence of sequential values[7]:

```
template <typename T>
class ValueGenerator
{
    T d_value;

  public:
    explicit ValueGenerator(const T& v) : d_value(v) { }

    T operator*() const { return d_value; }
    ValueGenerator& operator++() { ++d_value; return *this; }

    friend bool operator!=(const ValueGenerator& a, const ValueGenerator& b)
    {
        return a.d_value != b.d_value;
    }
};

template <typename T>
Subrange<ValueGenerator<T>> valueRange(const T& b, const T& e)
{
    return { ValueGenerator<T>(b), ValueGenerator<T>(e) };
}
```

Instead of referring to an element within a container, `ValueGenerator` is an iterator-like type that *generates* the value returned by **`operator`***. `ValueGenerator` can be instantiated for any type that can be incremented, e.g., integral types, pointers, or iterators. The `valueRange` function template is a simple factory to create a range comprising two `ValueGenerator` objects, using the `Subrange` class template defined in the use case described in Subranges. Thus, to print the numbers from 1 to 10, simply use a ranged-based **`for`** loop, employing a call to `valueRange` as the range expression:

```
void f3()
{
    // prints "1 2 3 4 5 6 7 8 9 10 "
    for (unsigned i : valueRange(1, 11))
```

[7]The `iota_view` and `iota` entities from the Ranges Library in the C++20 Standard provide a more sophisticated version of the `ValueGenerator` and `valueRange` facility described here.

```
    {
        std::cout << i << ' ';
    }
    std::cout << std::endl;
}
```

Note that the second argument to `valueRange` is one *past* the last item we want to iterate on, i.e., `11` instead of `10`. With something like `ValueGenerator` as part of a reusable utility library, this formulation expresses the intent of the loop more cleanly and concisely than the classic **`for`** loop.

The ability to generate numbers means that a range need not be finite. For example, we might want to generate a sequence of random numbers of indefinite length:

```
#include <random>  // std::default_random_engine, std::uniform_int_distribution

template <typename T = int>
class RandomIntSequence
{
    std::default_random_engine       d_generator;
    std::uniform_int_distribution<T> d_uniformDist;

public:
    class iterator
    {
        RandomIntSequence* d_sequence;

        explicit iterator(RandomIntSequence* s) : d_sequence(s) { }
        friend class RandomIntSequence;

    public:
        iterator& operator++() { return *this; }
        T operator*() const { return d_sequence->next(); }

        friend bool operator!=(iterator, iterator) { return true; }
    };

    RandomIntSequence(T min, T max, unsigned seed = 0)
        : d_generator(seed ? seed : std::random_device()())
        , d_uniformDist(min, max) { }

    T next() { return d_uniformDist(d_generator); }

    iterator begin() { return iterator(this); }
    iterator end()   { return iterator(nullptr); }
};

template <typename T>
RandomIntSequence<T> randomIntSequence(T min, T max, unsigned seed = 0)
{
    return {min, max, seed};
```

```
}
```

The `RandomIntSequence` class template uses the C++11 random-number library to generate high-quality pseudorandom numbers.[8] Each call to its `next` member function produces a new random number of integral type, `T`, within the inclusive range specified to the `RandomIntSequence` constructor. The nested `iterator` type holds a pointer to a `RandomIntSequence` and simply calls `next` each time it is dereferenced (i.e., via a call to **`operator*`**).

Of particular interest is **`operator!=`**, which returns **`true`** when comparing any two `RandomIntSequence<T>::iterator` objects. Thus, any ranged-based **`for`** loop that iterates over a `RandomIntSequence` is an infinite loop unless it terminates by some other means:

```
void f4()
{
    for (int rand : randomIntSequence(1, 10))
    {
        std::cout << rand << ' ';
        if (rand == 10) { break; }
    }

    std::cout << std::endl;
}
```

This example prints a list of random numbers in the range 1 through 10, inclusive. The loop terminates after printing `10` for the first (and only) time.

ng-over-simple-values

Iterating over simple values

The ability to iterate over a `std::initializer_list` can be very useful for processing a list of simple values or simple objects without first storing them in a container. Such a use case arises frequently when testing:

```
#include <limits>           // std::numeric_limits
#include <initializer_list>  // std::initializer_list

#define TEST_ASSERT(expr)  // ... assert that expr is true.

bool isEven(int i)
{
    return i % 2 == 0;
}

void testIsEven()
{
    // ...

    const int minInt = std::numeric_limits<int>::min();
    const int maxInt = std::numeric_limits<int>::max();
```

[8] An introduction to the C++11 random-number library can be found in Stephan T. Lavavej's excellent talk, `rand()` Considered Harmful.

```
    for (int testValue : {minInt, -256, -2, 0, 2, 4, maxInt - 1})
    {
        TEST_ASSERT(isEven(testValue));
        TEST_ASSERT(!isEven(testValue + 1));
    }
}
```

The `testIsEven` function iterates over a sample of numbers within the domain of `isEven`, including boundary conditions, testing that each number is correctly reported as being even and that adding 1 to the number produces a result that is correctly reported as not being even.

Initializer lists are not limited to primitive types, so the test data set can contain more complex values:

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

#define TEST_ASSERT_EQ(expr1, expr2)  // ... assert that expr1 == expr2.

int half(int i)
{
    return i / 2;
}

struct TestCase
{
    int value;
    int expected;
};

void testHalf()
{
    for (TestCase test : std::initializer_list<TestCase>{
        {-2, -1}, {-1, 0}, {0, 0}, {1, 0}, {2, 1}
    })
    {
        TEST_ASSERT_EQ(test.expected, half(test.value));
    }
}
```

In this case, the ranged-based **for** loop iterates over a `std::initializer_list` holding `TestCase` structures. This paring of input(s) with expected output(s) of a component under test is very common in unit tests.

Potential Pitfalls

potential-pitfalls

Lifetime of temporary objects in the range expression

in-the-range-expression

As described in *Description* on page 452 section, if the range expression evaluates to a temporary object, that object remains valid, as a result of lifetime extension, for the duration

of the ranged-based **for** loop. Unfortunately, there are some subtle ways in which lifetime extension is not always sufficient.

The basic notion of lifetime extension is that, when bound to a reference, the lifetime of a *prvalue* — i.e., an object created by a literal, constructed in place, or returned (by value) from a function — is *extended* to match the lifetime of the reference to which it is bound:

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

std::string strFromInt(int);

void f1()
{
    const std::string& s1 = std::string('a', 2);
    std::string&&      s2 = strFromInt(9);
    auto&&             i  = 5;

    // s1, s2, and i are "live" here.

    // ...

}  // s1, s2, and i are destroyed at end of enclosing block.
```

The first string is constructed in place. The resulting temporary string would normally be destroyed as soon as the expression was complete but, because it is bound to a reference, its lifetime is extended; its destructor is not called and its memory footprint is not reused until `s1` goes out of scope, i.e., at the end of the enclosing block. The `strFromInt` function returns by value; the result of calling it in the second statement produces a temporary variable whose lifetime is similarly extended until `s2` goes out of scope. Finally, the forwarding reference, `i`, ensures that space in the current frame is allocated to hold the temporary copy of the (deduced) **`int`** value, `5`; such space cannot be reused until `i` goes out of scope at the end of the enclosing block. (See Section 2.1."Forwarding References" on page 351).

When the range expression for a range-based **for** loop is a *prvalue*, lifetime extension is vital to keeping the range object live for the duration of the loop:

```
void f2(int i)
{
    for (char c : strFromInt(i))
    {
        // ...
    }
}
```

The return value from `strFromInt` is stored in a temporary variable of type `std::string`. The temporary string is destroyed when the loop completes, not when the expression evaluation completes. If the string were to go out of scope immediately, it would not be possible to iterate over its characters. This code would have **undefined behavior** were it not for the lifetime extension harnessed by the ranged-based **for** loop.

The limitation of lifetime extension is that it applies only if the reference is bound *directly* to the temporary variable itself or to a subobject (e.g., a member variable) of the temporary

variable, in which case the lifetime of the entire temporary variable is extended. Note that initializing a reference from a reference or a pointer to either the temporary or one of its subobjects does not count as binding *directly* to the temporary variable and does not trigger lifetime extension. The danger of an object getting destroyed prematurely is generally seen when the **full expression** returns a reference, pointer, or iterator into a temporary object:

```
#include <vector>   // std::vector
#include <string>   // std::string
#include <utility>  // std::pair
#include <tuple>    // std::tuple

struct Point
{
    double x, y;
    Point(double ax, double ay) : x(ax), y(ay) { }
};

struct SRef
{
    const std::string& str;
    SRef(const std::string& s) : str(s) { }
};

std::vector<int> getValues();  // Return a vector by value.

void f3()
{
    const Point& p1 = Point(1.2, 3.4);    // OK, extend Point lifetime.
    double&&     d1 = Point(1.2, 3.4).x;  // OK, extend Point lifetime.
    double&      d2 = Point(1.2, 3.4).y;  // Error, non-const lvalue ref, d2

    using ICTuple = std::tuple<int, char>;
    const int&  i1 = getValues()[0];               // Bug, dangling reference
    const int&  i2 = std::get<0>(ICTuple{0,'a'});  // Bug,     "        "
    auto&&      i3 = getValues().begin();          // Bug,     "    iterator
    const auto& s1 = std::string("abc").c_str();   // Bug,     "    pointer
    const auto& i4 = std::string("abc").length();  // OK, std::size_t extended

    SRef&&      sr = SRef("hello");  // Bug, string lifetime is not extended.
    std::string s2 = sr.str;         // Bug, string has been destroyed.
}
```

The first invocation of the `Point` constructor creates a temporary object that is bound to reference `p1`. The lifetime of this temporary object is extended to match the lifetime of the reference. Similarly, the lifetime of the second `Point` object is extended because a subobject, `x`, is bound to reference `d1`. Note that it is not permitted to bind a temporary to a non**`const`** *lvalue* reference, as is being attempted with `d2`, above.

The next four definitions do not result in useful lifetime extension at all.

1. In the case of `i1`, `getValues()` returns a *prvalue* of type `std::vector<int>`, resulting

in the creation of a temporary variable. That temporary variable, however, is *not* the value being bound to the `i1` reference; rather, the reference is bound to the result of the array-access operator (`operator[]`), which returns a reference into the temporary `vector` returned by `getValues()`. While we might consider an element of a `vector` logically to be a subobject of the vector, `i1` is not bound directly to that subobject but rather to the reference returned by `operator[]`. The vector goes out of scope immediately at the end of the statement, leaving `i1` to refer to an element of a deleted object.

2. The identical situation occurs with `i2` when accessing the member of a temporary `std::tuple`, this time via the nonmember function `std::get<0>`.

3. Rather than a reference, `i3` is deduced to be an *iterator* as the result of the expression. The iterator's lifetime is extended, but the lifetime of the object to which it refers is not.

4. Similarly, for `s`, the expression `std::string("abc").c_str()` yields a pointer into a temporary C-style string. Once again, the temporary `std::string` variable is not the object that is bound to the reference `s1`, so it gets destroyed at the end of the statement, invalidating the pointer.

Conversely, `i4` binds directly to the temporary object returned by `length`, extending its life even though the string itself gets destroyed as before. Unlike `i3` and `s1`, however, `i4` is not an iterator or pointer and so does not retain an implicit reference to the defunct string object.

The last two definitions, for `sr` and `s2`, show how subtle the rules for lifetime extension can be. The `"hello"` literal is converted into a temporary variable of type `std::string` and passed to the constructor of `SRef`, which *also* creates a temporary object. It is only the `SRef` object that is bound to the `sr` reference, so it is only the `SRef` object whose lifetime is extended. The `std::string("hello")` temporary variable gets destroyed when the constructor finishes executing, leaving the object referenced by `sr` with a member, `str`, that refers to a destroyed object.

There are good reasons why lifetime extension applies only to the temporary object being bound to a reference. A lot of code depends on temporary objects going out of scope immediately, i.e., to release a lock, memory, or some other resource. For range-based `for` loops, however, a compelling argument has been made that the correct behavior would be to extend the lifetime of *all* of the temporaries constructed while evaluating the range expression.[9] Unless and until this behavior is changed in a future Standard, beware of using a range expression that returns a reference to a temporary variable:

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

class RecordList
{
```

[9] At the time of writing the P2012 paper seeks to solve the issue when a range expression is a reference into a temporary. See *Fix the range-based for loop*, by Nicolai Josuttis, Victor Zverovich, Filipe Mulonde, and Arthur O'Dwyer, which references an original paper *Embracing Modern C++ Safely* by Rostislav Khlebnikov and John Lakos.

```
    std::vector<std::string> d_names;
    // ...

public:
    explicit RecordList(std::istream& is);
        // Create a RecordList with data read from is.

    // ...

    const std::vector<std::string>& names() const { return d_names; }
};

void printNames(std::istream& is)
{
    // Bug, RecordList's lifetime is not extended.
    for (const std::string& name : RecordList(is).names())
    {
        std::cout << name << '\n';
    }
}
```

The `RecordList` constructed in the range expression is not bound to the implied `__range` reference within the ranged-based **`for`** loop, so its lifetime will end before the loop actually begins. Thus, the **`const`** `std::vector<std::string>&` returned by its `names` method becomes a dangling reference, leading to undefined behavior (such as a segmentation fault).

To avoid this pitfall, create a named object for each temporary that you need to preserve, unless that temporary is the full expression for the range expression:

```
void printNames2(std::istream& is)
{
    {
        RecordList records(is);  // named variable
        for (const std::string& name : records.names())
        {
            std::cout << name << '\n';
        }

        // safe for records to go out of scope now
    }

    // ...
}
```

This minor rewrite of `printNames` creates an extra block scope in which we declare `records` as a named variable. The inner scope ensures that `records` gets destroyed immediately after the loop terminates.

t-copying-of-elements

Inadvertent copying of elements

When iterating through a container with a classic **for** loop, elements are typically referred to through an iterator:

std::vectorstd::string

```
void process(std::string&);

void f1(std::vector<std::string>& vec)
{
    for (std::vector<std::string>::iterator i = vec.begin();
         i != vec.end(); ++i)
    {
        process(*i);  // refer to element via iterator
    }
}
```

The ranged-based **for** loop gives the element a name and a type. If the type is not a reference, then each iteration of the loop will *copy* the current element. In many cases, this copy is inadvertent:

```
void f2(std::vector<std::string>& vec)
{
    for (std::string s : vec)
    {
        process(s);  // call process on *copy* of string element, potential
                     // bug.
    }
}
```

The example above illustrates two issues: (1) there is an unnecessary expense in copying each string, and (2) `process` may modify or take the address of its argument, in which case it will modify or take the address of the copy, rather than the original element; the strings in `vec` will remain unchanged.

This error appears to be especially common when using **auto** to deduce the loop variable's type:

```
void f3(std::vector<std::string>& vec)
{
    for (auto s : vec)
    {
        process(s);  // call process on *copy* of deduced string element,
                     // potential bug.
    }
}
```

Copying an element is not always erroneous, but it may be wise to habitually declare the loop variable as a reference, making deliberate exceptions when needed:

```
void f4(std::vector<std::string>& vec)
{
    for (std::string& s : vec)
```

```
    {
        process(s);  // OK, call process on *reference* to string element
    }
}
```

If we wish to avoid copying elements but also wish to avoid modifying them, then a **const** reference will provide a good balance. Note, however, that if the type being iterated over is not the same as the type of the reference, a conversion might quietly produce the (undesired) copy anyway:

```
void f5(std::vector<char*>& vec)
{
    for (const std::string& s : vec)
    {
        // s is a reference to a *copy* of an element of vec.
    }
}
```

In this example, the elements of `vec` have type **`char*`**. The use of **`const`** `std::string&` to declare the loop variable `s` correctly prevents modification of any elements of `vec`, but there is still a copy being made because each member access is converted to an object of type `std::string`.

For generic code that modifies a container, **`auto`**`&&` is the most general way to declare the loop variable. For generic code that does not modify the container, **`const auto`**`&` is safer:

```
template <typename Rng>
void f6(Rng& r)
{
    for (auto&& e : r)
    {
        // ...
    }

    for (const auto& cr : r)
    {
        // ...
    }
}
```

Because `e` is a forwarding reference and `cr` is a **`const`** reference, they will both correctly bind to the return type of `*begin(Rng)`, even if that type is a *prvalue*. Note that the use of **`auto`** can obfuscate code by hiding the underlying types of objects. Obfuscated code is prone to bugs so these uses of **`auto`** are recommended chiefly for *generic* code or where other trade-offs favor using this short-cut; see Section 2.1."**`auto`** Variables" on page 183.

Simple and reference-proxy behaviors can be be different

`ors-can-be-be-different`

Some containers have iterators that return proxies rather than references to their elements. Depending on how the loop variable is declared, the unwary programmer may get surprising results when the container's iterator type returns reference proxies.

An example of such a container is `std::vector<bool>`. The type, `std::vector<bool>::iterator::reference` is a reference-proxy class that emulates a reference to a single bit within the `vector`. The proxy class provides an **`operator bool()`** that returns the bit when the proxy is converted to **`bool`** and an **`operator=(bool)`** that modifies the bit when assigned a Boolean value.

Let's consider a set of loops, each of which iterates over a `vector` and attempts to set each element of the vector to **`true`**. We'll embed the loops in a function template so that we can compare the behavior of instantiating with a normal container (`std::vector<int>`) and with one whose iterator uses a reference proxy (`std::vector<bool>`):

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

template <typename T>
void f1(std::vector<T>& vec)
{
    for (T      v : vec) { v = true; }  // (1)
    for (T&     v : vec) { v = true; }  // (2)
    for (T&&    v : vec) { v = true; }  // (3)
    for (auto   v : vec) { v = true; }  // (4)
    for (auto&  v : vec) { v = true; }  // (5)
    for (auto&& v : vec) { v = true; }  // (6)
}

void f2()
{
    using IntVec  = std::vector<int>;   // has normal iterator
    using BoolVec = std::vector<bool>;  // has iterator with reference proxy

    IntVec  iv{ /* ... */ };
    BoolVec bv{ /* ... */ };

    f1(iv);
    f1(bv);
}
```

For each of the loops in `f1`, the difference in behavior between the `IntVec` and `BoolVec` instantiations hinges on what happens when `v` is initialized from `*__begin` within the loop transformation. For the `IntVec` iterator, `*__begin` returns a *reference* to the element within the container. For the `BoolVec` iterator, it returns an *object* of the reference-proxy type.

- *Loop (1)* produces identical behavior from both instantiations. The loop makes a local copy of each element, and then modifies the copy. The only difference is that the `BoolVec` version performs a conversion to **`bool`** to initialize `v`, whereas the `IntVec` version initializes `v` directly from the element reference. For both the `IntVec` or `BoolVec` version, the fact that the original vector is unchanged is a potential bug (see Inadvertent copying of elements, above).

- *Loop (2)* modifies the container elements in the `IntVec` instantiation but fails to compile for the `BoolVec` instantiation. The compilation error comes from trying to

initialize the non**const** ***lvalue* reference**, `v`, from an *rvalue* of the proxy type. The **`bool`** conversion operator does not help since the result would still be an *rvalue*.

- *Loop (3)* fails to compile for the `IntVec` iterator because the ***rvalue* reference**, `v`, cannot be initialized from `*__begin`, which is an *lvalue* reference. Surprisingly, the `BoolVec` instantiation *does* compile, but the loop does not modify the container. Here, **`operator bool`** is invoked on the proxy object returned by `*__begin`. The resulting temporary object is bound to `v`, and its lifetime is extended for the duration of the iteration. Because `v` is bound to a temporary variable, modifying `v` modifies only the temporary, not the original, element, resulting in a likely bug as in the case of loop (1).

- *Loop (4)* compiles for both the `BoolVec` and `IntVec` cases but produces different results for each. For `IntVec` iterators, **`auto`** deduces the type of `v` as **`int`**, so assigning to `v` modifies a local copy of the element, as in loop (1). For `BoolVec` iterators, the deduced type of `v` is the proxy type rather than **`bool`**. Assigning to the proxy *does* change the element of the container.

- *Loop (5)*, like loop (2), works as expected for `IntVec` instantiation but fails to compile for the `BoolVec` instantiation. As before, the problem is that the `BoolVec` iterator yields an *rvalue* that cannot be used to initialize an *lvalue* reference.

- *Loop (6)* produced identical behavior from both instantiations, modifying each of the `vector` elements. The type of `v` is deduced to be **`int&`** for `IntVec` instantiation and the proxy type for the `BoolVec` instantiation. Assigning through either the real reference or the reference proxy modifies the element in the container.

Let's now look at the the situation with **`const`**-qualified loop variables:

```
template <typename T>
void f3(std::vector<T>& vec)
{
    for (const T      v : vec) { /* ... */ }  // (7)
    for (const T&     v : vec) { /* ... */ }  // (8)
    for (const T&&    v : vec) { /* ... */ }  // (9)
    for (const auto   v : vec) { /* ... */ }  // (10)
    for (const auto&  v : vec) { /* ... */ }  // (11)
    for (const auto&& v : vec) { /* ... */ }  // (12)
}

void f4()
{
    using IntVec  = std::vector<int>;   // has normal iterator
    using BoolVec = std::vector<bool>;  // has iterator with reference proxy

    IntVec  iv{ /* ... */ };
    BoolVec bv{ /* ... */ };

    f3(iv);
```

```
    f3(bv);
}
```

- *Loop (7)* works identically for both instantiations, converting the proxy reference to **`bool`** in the `BoolVec` case.
- *Loop (8)* works identically for both instantiations. For the `IntVec` case, the result of `*__begin` is bound directly to `v`. For the `BoolVec` case, `*__begin` produces proxy reference that is converted to a **`bool`** temporary that is then bound to `v`. Lifetime extension keeps the bool value alive.
- *Loop (9)* fails to compile for `IntVec` but succeeds for `BoolVec` exactly as for loop (3) except that the temporary **`bool`** bound to `v` is **`const`** and thus does not risk giving programmers the false belief that they are modifying the container.
- *Loop (10)* has the same behavior for both the `IntVec` and `BoolVec` instantiations. That mechanism is the same behavior as for loop (4) except that, because `v` is **`const`**, *neither* instantiation can modify the container.
- *Loop (11)* also works for both instantiations. For the `IntVec` case, the result of `*__begin` is bound directly to `v`. For the `BoolVec` case, `v` is deduced to be a **`const`** reference to the proxy type; `*__begin` produces a temporary variable of the proxy type, which is then bound to `v`. Lifetime extension keeps the proxy alive. In most contexts, a **`const`** proxy reference is an effective stand-in for a **`const bool&`**.
- *Loop (12)* fails to compile for `IntVec` but succeeds for `BoolVec`. The error with `IntVec` occurs because **`const auto&&`** is always a **`const`** *rvalue* reference (not a forwarding reference) and cannot be bound to the *lvalue* reference, `*__begin`. For `BoolVec`, the mechanism is identical to loop (11) except that loop (11) binds the temporary object to an *lvalue* reference whereas loop (12) uses an *rvalue* reference. When the references are **`const`**, however, there is little practical differences between them.

Note that loops 4, 6, 10, 11, and 12 in the `BoolVec` instantiations bind a reference to a temporary proxy reference object, so taking the address of `v` in these situations is likely not to produce useful results. Additionally, loops 3, 8, and 9 bind `v` to a temporary **`bool`**. Users must be mindful of the lifetime of these temporary objects (a single-loop iteration) and not allow the address of `v` to escape the loop.

Proxy objects emulating references to non-class elements within a container are surprisingly effective, but their limitations are exposed when they are bound to references. In generic code, as a rule of thumb, **`const auto&`** is the safest way to declare a read-only loop variable if a reference proxy might be in use, while **`auto&&`** will give the most consistent results for a loop that modifies its container. Similar issues, unrelated to range-based **`for`** loops, occur when passing a proxy reference to a function taking a reference argument.

Annoyances

annoyances

No access to the state of the iteration

tate-of-the-iteration

When traversing a range with a classic **`for`** loop, the loop variable is typically an iterator or array index. Within the loop, we can modify that variable to repeat or skip iterations.

Similarly, the loop-termination condition is usually accessible so that it is possible to, for example, insert or remove elements and then recompute the condition:

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

void spaceBeforeCaps(std::string& s)
{
    // Insert a space before each capital letter in s.
    using IdxType = std::string::size_type;
    for (IdxType i = 0; i < s.size(); ++i)
    {
        if (std::isupper(s[i]))
        {
            s.insert(i, 1, ' ');  // Insert one space at i.
            ++i;                  // Skip the space.
        }
    }
}
```

The code above depends on (1) having access to the iteration index, (2) being able to change the iteration index, and (3) recomputing the size of the collection each time through the loop. No similar function could be written using a ranged-based **for** loop since the `__range`, `__begin`, and `__end` variables are for exposition only and are not accessible from within the code:

```
void spaceBeforeCaps2(std::string& s)
{
    // Insert a space before each capital letter in s.
    for (char c : s)
    {
        if (std::isupper(c))
        {
            __begin = s.insert(__begin, ' ');  // Error, no __begin variable
            ++__begin;                         // Error, no __begin variable
            __end = s.end();                   // Error, no __end variable
        }
    }
}
```

A classic **for** loop can traverse more than one container at a time (e.g., to add corresponding elements from two containers and store them into a third). It accomplishes this feat by either incrementing multiple iterators on each iteration or keeping a single index that is used to access multiple, random-access iterators concurrently. Trying to accomplish something similar with a range-based **for** loop usually involves using a hybrid approach:

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

void addVectors(std::vector<int>&       result,
                const std::vector<int>& a,
```

```
                const std::vector<int>& b)
    // For each element ea of a and corresponding element eb of b, set
    // the corresponding element of result to ea + eb.  The behavior is
    // undefined unless a and b have the same length.
{
    assert(a.size() == b.size());
    result.resize(a.size());

    std::vector<int>::const_iterator ia = a.begin();
    std::vector<int>::const_iterator ib = b.begin();
    for (int& sum : result)
    {
        sum = *ia++ + *ib++;
    }
}
```

Although `result` is traversed using the range-based **`for`** loop, `a` and `b` are effectively traversed manually using iterators. It is arguable as to whether the code is any clearer or simpler to write than it would have been using a classic **`for`** loop.

This situation can be improved through the use of a "zip" iterator — a type that holds multiple iterators and increments them in lock-step. Using a "zip" iterator, all three containers can be traversed using a single ranged-based **`for`** loop:

```
#include <cassert>  // standard C assert macro
#include <tuple>    // std::tuple
#include <utility>  // std::declval
#include <vector>   // std::vector

template <typename... Iter>
class ZipIterator
{
    std::tuple<Iter...> d_iters;

    // ...

public:
    using reference = std::tuple<decltype(*std::declval<Iter>())...>;

    ZipIterator(const Iter&... i);

    reference operator*() const;
    ZipIterator& operator++();
    friend bool operator!=(const ZipIterator& a, const ZipIterator& b);
};

template <typename... Range>
class ZipRange
{
    using ZipIter =
        ZipIterator<decltype(begin(std::declval<Range>()))...>;
```

```
    // ...

public:
    ZipRange(const Range&... ranges);

    ZipIter begin() const;
    ZipIter end() const;
};

template <typename... Range>
ZipRange<Range...> makeZipRange(Range&&... r);

void addVectors2(std::vector<int>&       result,
                 const std::vector<int>& a,
                 const std::vector<int>& b)
{
    assert(a.size() == b.size());
    result.resize(a.size());

    for (std::tuple<int, int, int&> elems : makeZipRange(a, b, result))
    {
        std::get<2>(elems) = std::get<0>(elems) + std::get<1>(elems);
    }
}
```

Each iteration, instead of yielding a single element, yields a `std::tuple` of elements resulting from the traversal of multiple ranges simultaneously. To be used, the elements must be unpacked from the `std::tuple` using `std::get`. Zip iterators become much more attractive in C++17 with the advent of structured bindings, which allow multiple loop variables to be declared at once, without the need to directly unpack the `std::tuple`s. The above implementation and usage of `ZipRange` is just a rough sketch; the full design and implementation of zip iterators and zip ranges is beyond the scope of this section.

Adapters are required for many tasks

required-for-many-tasks

In the usage examples above, we have seen a number of adapters (e.g., to traverse subranges, to traverse a container in reverse, to generate sequential values), and zip iterators to iterate over multiple ranges at once.

None of these adapters would be required for a classic **for** loop. On the one hand, one-off situations are expressed more simply with a classic **for** loop. On the other hand, the adapters that we create to make range-based **for** loops usable in more situations can lead to the development of a reusable *library* of adapters. Using the `ValueGenerator` class from Range generators, for example, produces simpler and more expressive code than using a classic **for** loop would.[10]

[10]The Standard's Ranges Library, introduced in C++20, provides a sophisticated algebra for working with and adapting ranges.

ntinel-iterator-types

No support for sentinel iterator types

For a given range expression, `__range`, `begin(__range)`, and `end(__range)` must return the same type to be usable with a ranged-based **`for`** loop. This limitation is problematic for ranges of indeterminate length, where the condition for ending a loop is not determined by comparing two iterators. For example, in the `RandomIntSequence` example (see Range generators), the end iterator for the infinite random sequence holds a null pointer and is never used, not even within **`operator`**`!=`. It would be more efficient and convenient if the end iterator were a special, empty *sentinel* type. Comparing any iterator to the sentinel would determine whether the loop should terminate:

```
template <typename T = int>
class RandomIntSequence2
{
    // ...

public:
    class sentinelIterator { };

    class iterator {
        // ...
        friend bool operator!=(iterator, sentinelIterator) { return true; }
    };

    iterator         begin() { /* ... */ }
    sentinelIterator end() const { return {}; }
};
```

The above code shows an example of `begin` and `end` returning different types, where `end` returns an empty sentinel type. Unfortunately, using this formulation of `RandomIntSequence2`

with a C++11 range-based **for** loop will result in a compilation error complaining that `begin` and `end` return inconsistent types.[11]

Another type that could benefit from a sentinel iterator is `std::istream_iterator`, since the state of the end iterator is never used. It is unlikely that this interface will change, however, as `std::istream_iterator` has been with us since the first C++ Standard.

Only ADL lookup

only-adl-lookup

The free functions `begin` and `end` are found using argument-dependent lookup (ADL) only. File-scope (**`static`**) functions are not considered. If we wish to add `begin` and `end` functions for a range-like type that we do not own, we need to put those functions into the same namespace as the range-like type, inviting a potential name collision with other compilation units attempting to do the same thing:

```
// third_party_library.h:

namespace third_party
{

    class IteratorLike { /* ... */ };

    class RangeLike
    {
        // ... does not provide begin and end members
    };

    // ... does not provide begin and end free functions
}
```

[11] This limitation on the use of sentinel iterators was lifted in C++17. Sentinel iterators are supported directly by the C++20 Ranges Library. In C++17, the specification was modified to:

```
{
    auto&& __range = range-expression;
    auto __begin   = begin-expr;
    auto __end     = end-expr;
    for (; __begin != __end; ++__begin)
    {
        for-range-declaration = *__begin;
        statement
    }
}
```

```
// myclient.cpp:

#include <third_party_library.h>

static third_party::IteratorLike begin(third_party::RangeLike&);
static third_party::IteratorLike end(third_party::RangeLike&);

void f()
{
    third_party::RangeLike rl;
    for (auto&& e : rl)  // Error, begin not found by ADL
    {
        // ...
    }
}
```

The code above attempts to work around the absence of `begin(RangeLike&)` and `end(RangeLike&)` in the third-party library by defining them locally within `myclient.cpp`. This attempt fails because static functions are not found via ADL. A better workaround that does work is to create a range adapter for the range-like class:

```
class RangeLikeAdapter
{
    // ...

public:
    RangeLikeAdapter(third_party::RangeLike&);
    third_party::IteratorLike begin() { /* ... */ }
    third_party::IteratorLike end()   { /* ... */ }
};
```

The adapter wraps the range-like type and provides the missing features. Beware, however, that if the wrapper stores a pointer or reference to a temporary `RangeLike` object, you don't run into the pitfall where the lifetime of temporary objects is not always extended; see Lifetime of temporaries in the range expression.

see-also

See Also

- "**auto** Variables" (§2.1, p. 183) ♦ explains **auto**, often used in a ranged-based **for** loop to determine the type of the loop variable, and many of the pitfalls of **auto** apply when using it for that purpose.
- "Ref-Qualifiers" (§3.1, p. 677) ♦ show how to overload member functions to work differently on *rvalues* and *lvalues*.

further-reading

Further Reading

No further reading.

Rvalue References: &&

Rvalue-References

placeholder text.....

Explicit Enumeration Underlying Type

ation-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

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++03 **enum** is chosen by the implementation to be large enough to represent all of the values in the 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_31 = 1U<<31 };                         // OK, fits unsigned 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 = 1LL<<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**; see Section 1.1."**long long**" on page 78.

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 `-1LL << 63` for a 64-bit signed integer (assuming such is available on the target platform).

The above applies only to C++03 **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 310.

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). The same is true in C++11 for **char16_t** and **char32_t**.[1]

[1] C++20 adds **char8_t**, which is a distinct type that has **unsigned char** as its underlying type.

erlying-type-explicitly

Specifying underlying type explicitly

As of C++11, we have the ability to specify the integral type that is used to represent an **enum**. This is achieved by providing the type explicitly in the **enum**'s declaration following the enumeration's (optional) name and preceded by a colon:

```
enum Port : unsigned char
{
    // Each enumerator of Port is represented as an unsigned char type.

    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 483.

use-cases

Use Cases

ator-values-are-salient

Ensuring a compact representation where enumerator values are salient

When an 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, 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; }
};
```

Suppose that, within an application, a `Date` is typically constructed using values obtained through a GUI, where the month is always selected from a drop-down menu. The month is 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, e_APR, e_MAY, e_JUN,
    e_JUL , e_AUG, e_SEP, e_OCT, e_NOV, e_DEC
};
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 Section 2.1."**`alignas`**" on page 158.)

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 a `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,
    e_APR    , e_MAY, e_JUN,
    e_JUL    , e_AUG, e_SEP,
    e_OCT    , e_NOV, e_DEC
};

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)
```

pitfalls-underlyingenum

merators-enumunderlying

Potential Pitfalls

External use of opaque enumerators

Providing an explicit underlying type to an **enum** enables clients to declare it as a complete type 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 Section 2.1."Opaque **enum**s" on page 435.)

s-of-integral-promotion

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 Section 2.1."**auto** Variables" on page 183):

```
#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
}
```

Casting to the underlying type is not necessarily the same as direct integral promotion. If an enumerator is intended to be used in arithmetic operations,[2] a **constexpr** variable might be a better alternative (see Section 2.1."**constexpr** Variables" on page 282):

```
// enum { k_GRAMS_PER_OZ = 28 };    // not the best idea
constexpr int k_GRAMS_PER_OZ = 28;  // better idea

double gramsFromOunces(double ounces)
{
    return ounces * ;k_GRAMS_PER_OZ
}
```

see-also

See Also

- "**constexpr** Variables" (§2.1, p. 282) ♦ introduces an alternative way of declaring compile-time constants.
- "**enum class**" (§2.1, p. 310) ♦ introduces a scoped, more strongly typed enumeration for which the default underlying type is **int** and can also be specified explicitly.
- "Opaque **enum**s" (§2.1, p. 435) ♦ offers a means to insulate individual enumerators from clients.

further-reading

Further Reading

- "Item 10: Prefer scoped `enum`s to unscoped `enum`s," **?**
- **?**

[2] As of C++20, using an expression of an unscoped enumerated type in a *binary* operation along with an expression of either some *other* enumerated type or any nonintegral type (e.g., **double**) is deprecated, with the possibility of being removed in C++23. Platforms may decide to warn against such uses retroactively.

User-Defined Literal Operators

userdeflit

C++11 allows developers to define a new suffix for a *numeric*, *character*, or *string* literal, enabling a convenient lexical representation of the *value* of a **user-defined type (UDT)** or even a novel notation for the value of a built-in type.

Description

description-userdeflit

A **literal** is a single token in a program that represents, in C and classic C++, a value of an integer, floating-point, character, string, Boolean, or pointer type.

Examples of familiar literal tokens are integer literals `19` and `0x13`, each representing an **`int`** having a value of 19; floating-point literals `0.19` and `1.9e-1`, each representing a **`double`** having a value of 0.19; character literals `'a'` and `\141`, each representing a **`char`** having the (ASCII) value for the letter "a"; string literal, `"hello"`, representing a null-terminated array containing the six characters `'h'`, `'e'`, `'l'`, `'l'`, `'o'`, and `'\0'`; and Boolean keyword literals **`true`** and **`false`**, representing the corresponding Boolean values. C++11 added the keyword literal, **`nullptr`** (see Section 1.1."**`nullptr`**" on page 87), representing the null pointer value.

Both integer and floating-point literals have always had suffixes to identify other numeric C++ types. For example, `123L` and `123ULL` are literals of type **`signed long`** and **`unsigned long long`**, respectively, both having a decimal value 123, whereas `123.f` is a literal of type **`float`** having precisely the decimal value 123. We can easily distinguish programmatically between these different types of literals using, e.g., overload resolution:

```
void f(const int&);            // (1) overload for type int
void f(const long&);           // (2)    "      "   "   long
void f(const double&);         // (3)    "      "   "   double
void f(const float&);          // (4)    "      "   "   float
void f(const unsigned int&);   // (5)    "      "   "   unsigned int
void f(const unsigned long&);  // (6)    "      "   "   unsigned long

void test0()
{
    f(123);    // OK, calls (1)
    f(123L);   // OK, calls (2)
    f(123.);   // OK, calls (3)
    f(123.f);  // OK, calls (4)
    f(123U);   // OK, calls (5)
    f(123Lu);  // OK, calls (6)
    f(123.L);  // Error, call to f(long double) is ambiguous
    f(123f);   // Error, invalid hex digit f in decimal constant
}
```

Notice that applying an `L` or `l` suffix to a floating-point literal (of default type **`double`**) identifies it as being of type **`long double`**, which is a **standard conversion** away from both **`float`** and **`double`**, making the call ambiguous. Applying `F` or `f` to an integer literal is, by default, not permitted unless a user-defined literal (UDL) of a compatible type can be found.

Classic C++ allowed values of only built-in types to be represented as compile-time literals. To express a hard-coded value of a UDT, a developer would need to use a **value constructor** or **factory function** and, unlike literals of built-in types, these *runtime* workarounds could never be used in a **constant expression**. For example, we might want to create a user-defined type, `Name`, that can construct itself from a null-terminated string:

```
class Name  // user-defined type constructible from a literal string
{
    // ...

public:
    Name(const char*);  // value constructor taking a null-terminated string
    //...
};
```

We can then initialize a variable of type `Name` from a string literal using the value constructor:

```
Name nameField("Maria");  // Name object having value "Maria"
```

Alternatively, we could create one or more factory functions that return a constructed object appropriately configured with the desired value. Multiple factory functions having different names can be created without necessarily adding new constructors to the definition of the returned type though, in some cases, a factory function might be a **`friend`** of the type it configures:

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

class Temperature { /*...*/ };

Temperature fahrenheit(double degrees);  // configured from degrees Fahrenheit
Temperature celsius(double degrees);     // configured from degrees Celsius

void test1()
{
    Temperature t1 = fahrenheit(32);  // water freezes at this temperature
    Temperature t2 = celsius(0.0);    //   "      "     "  "        "
    assert(t1 == t2);                 // expect same type and same value
}
```

Note that, as of C++11, the two functional constructs described above can be declared **`constexpr`** and thus be eligible to be evaluated as part of a constant expression; see Section 2.1."**`constexpr`** Functions" on page 239.

Although usable, the aforementioned C++03 workarounds for representing literal values of a UDT lack the compactness and expressiveness of those for built-in types. A fundamental design goal of C++ has always been to minimize such differences. To that end, C++11 extends the notion of **type suffix** to include user-definable identifiers with a leading underscore (e.g., `_name`, `_time`, `_temp`):

```
Temperature operator""_F(long double degrees) { /*...*/ }  // define suffix _F
Temperature operator""_C(long double degrees) { /*...*/ }  // define suffix _C
```

```
void test2()  // same as test1 above, but this time with user-defined literals
{
    Temperature t1 = 32.0_F; // water freezes at 32 degrees Fahrenheit
    Temperature t2 = 0.0_C;  //   "      "     "  0 degrees Celsius
    assert(t1 == t2);        // expect same type and same value
}
```

The example above demonstrates the basic idea of a **UDL** implemented as a new kind of operator: **`operator""`** followed by a suffix name. A UDL is a literal token having a **UDL suffix** that names a **UDL operator**. A UDL *uses* a suffix, whereas a UDL operator *defines* a suffix. A UDL suffix must be a valid identifier that begins with an underscore (_). Note that the leading underscore is *not* required for UDL suffixes defined by the C++ Standard Library. A UDL is formed by appending a UDL suffix to a native literal of one of four **type categories**: **integer literals** (e.g., `2020_year`), **floating-point literals** (e.g., `98.6_F`), **character literals** (e.g., `'x'_ebcdic`), and **string literals** (e.g., `"1 Pennsylvania Ave"_validated`). Regardless of the type category, a UDL can evaluate to any built-in or user-defined type. What's more, the same UDL suffix can apply to more than one of the four UDL type categories enumerated above, potentially yielding a different type for each category.

Each UDL operator is effectively a **factory operator** that takes a highly constrained parameter list (see *UDL operators* on page 490) and returns an appropriately configured object. The defined UDL suffix, like postfix operators `++` and `--`, names a function to be called, but is parsed as part of the preceding literal (with no intervening whitespace) and cannot be applied to an arbitrary run-time expression. In fact, the compiler does not always produce an *a priori* interpretation of the literal before invoking the UDL operator.

Let's now see how we might create a UDL for our original user-defined type, `Name`. In this example, the UDL "function" operates on a string literal, rather than a floating-point literal:

```
    std::size_t

Name operator""_Nm(const char* n, std::size_t /* length */) { return Name(n); }
   // user-defined literal (UDL) operator for UDL suffix _Nm

Name nameField = "Maria"_Nm;  // Name object having value "Maria"
```

The UDL definition for string literals above, **`operator""_Nm`** in this case, takes two arguments: a **`const char*`** representing a null-terminated character string and a `std::size_t` representing the length of that string excluding the null terminator. In our example, we ignore the second argument in the body of the UDL operator, but it must nonetheless be present in the parameter list.

UDL operators, like factory functions, can return a value of any type, including built-in types. Unit-conversion functions, for example, often return built-in types normalized to specific units:

```
#include <ctime>  // std::time_t

constexpr std::time_t minutes(int m) { return m * 60; }    // minutes to seconds
constexpr std::time_t hours(int h)   { return h * 3600; }  // hours to seconds
```

Each of the unit-conversion functions above returns an `std::time_t` (a standard type alias for a built-in integral type) representing a duration in seconds. We can combine such uniform quantities initialized from values in disparate units as needed:

```
std::time_t duration = hours(3) + minutes(15);   // 3.25 hours as seconds
```

Replacing the unit-conversion functions with UDLs allows us to express the desired value with a more natural-looking syntax. We now define two new UDL operators, creating suffixes `_min` for minutes and `_hr` for hours, respectively:

```
std::time_t operator""_min(unsigned long long m)
{
    return static_cast<std::time_t>(m * 60);  // minutes-to-seconds conversion
}

std::time_t operator""_hr(unsigned long long h)
{
    return static_cast<std::time_t>(h * 3600);  // hours-to-seconds conversion
}

std::time_t duration = 3_hr + 15_min;  // 3.25 hours as seconds
```

We are not done yet. Unlike built-in literals, UDLs that use the suffixes defined above cannot express arbitrary literal values that can be treated as compile-time constants usable in constant expressions such as sizing an array or within a **`static_assert`**:

```
int a1[5_hr];                    // Error, 5_hr is not a compile-time constant.
static_assert(1_min == 60, "");  // Error, 1_min " " " " " " "
```

Typical definitions of UDLs will, therefore, also involve another C++11 feature, **`constexpr`** functions (see Section 2.1."**`constexpr`** Functions" on page 239). By simply adding **`constexpr`** to the declaration of our UDL operators, we enable them to be evaluated at compile-time and, hence, usable in constant expressions:

```
    std::time_t
constexpr std::time_t operator""_Min(unsigned long long m)
{
    return static_cast<std::time_t>(m * 60);  // minutes-to-seconds conversion
}

constexpr std::time_t operator""_Hr(unsigned long long h)
{
    return static_cast<std::time_t>(h * 3600);  // hours-to-seconds conversion
}

int a2[5_Hr];                    // OK, 5_Hr is a compile-time constant.
static_assert(1_Min == 60, "");  // OK, 1_Min " " " " " "
```

In short, a UDL operator is a new kind of **free operator** that (1) may be defined with certain specific signatures limited to a subset of the built-in types and (2) is invoked automatically when used as a suffix of a built-in literal. There are, however, multiple ways to define UDL operators — *precomputed-argument*, *raw*, and *template* — each more expressive and complex

than the previous. The allowed variations on the definitions of UDL operators are elucidated below.

defined-literals-(udls)

Restrictions on UDLs

A UDL suffix can be defined for only integer, floating-point, character, and string literals. Left deliberately unsupported are the two *Boolean* literals, **`true`** and **`false`**, and the *pointer* literal, **`nullptr`** (see Section 1.1."**`nullptr`**" on page 87). These omissions serve to sidestep lexical ambiguities — e.g., the Boolean literal, **`true`**, combined with the UDL suffix `_fact` would be indistinguishable from the identifier, `true_fact`.

We'll refer to the sequence of characters that make up a literal excluding any suffix as a **naked literal** — e.g., for literal `"abc"_udl`, the naked literal is `"abc"`. UDL suffixes can be appended to otherwise valid lexical literal tokens only. That is, appending a UDL suffix to a token that wouldn't be considered a valid lexical literal without the suffix is not permitted. Though creating a suffix, `_ipv4`, to represent an IPv4 Internet address consisting of four octets separated by periods might be tempting, `192.168.0.1` would not be a valid lexical token in a program and, hence, neither would `192.168.0.1_ipv4`. Interestingly, creating a UDL that would cause an overflow if interpreted without the suffix *is* permitted. Thus, for example, the UDL `0x123456789abcdef012345678_verylong`, which comprises 24 hex digits and the UDL suffix `_verylong`, would be valid even on an architecture whose native integers cannot exceed 64 bits (16 hex digits); see Section 1.1."**`long long`**" on page 78.

Note that there are no *negative* numeric literals in C++. The negative numeric value `-123` is represented as two separate tokens: the negation operator and a positive literal, `123`. Similarly, an expression like `-3_t1` will attempt to apply the negation operator to the object of, say, `Type1` produced by the `3_t1` UDL, which in turn comes from passing an **`unsigned long long`** having the positive value 3 to the UDL operator **`operator`**`""_t1`. Such an expression will be **ill formed** unless there is a negation operator that operates on *rvalues* of type `Type1`. None of the three forms of UDL operators (see *Prepared-argument UDL operators* on page 492, *Raw UDL operators* on page 495, and *Templated UDL operators* on page 498) needs, or is able, to handle a naked literal representing a negative number.

When two or more string literals appear without intervening tokens, they are concatenated and treated as a single string literal. If at least one of those strings has a UDL suffix, then the suffix is applied to the concatenation of the naked literal strings. If more than one of the strings has a suffix, then all such suffixes must be the same UDL suffix; concatenating string literals having different suffixes is not permitted, but strings having no suffix may be concatenated with strings having a UDL suffix:

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

struct XStr { /*...*/ };
XStr operator""_X(const char* n, std::size_t length);
XStr operator""_Y(const char* n, std::size_t length);

char a[] = "hello world";            // single native string literal
char b[] = "hello"      " world";    // native equivalent to "hello world"
XStr c   = "hello world"_X;          // user-defined string literal
XStr d   = "hello"_X    " world";    // UDL equivalent to "hello world"_X
```

```
XStr e   = "hello"      " world"_X;  //  "      "      "     "     "
XStr f   = "hello"_X    " world"_X;  //  "      "      "     "     "
XStr g   = "hel"_X "lo" " world"_X;  //  "      "      "     "     "
XStr h   = "hello"_X    " world"_Y;  // Error, mixing UDL suffixes _X and _Y
```

Finally, combining a UDL suffix with a second built-in or user-defined suffix on a single token is not possible. Writing `45L_Min`, for example, in an attempt to combine the `L` suffix (for **`long`**) with the `_Min` suffix (for the user-defined minutes suffix described earlier) will simply yield the undefined and invalid suffix, `L_Min`.

teral-(udl)-operators

UDL operators

A UDL suffix (e.g., `_udl`) is created by defining a UDL operator (e.g., **`operator`**`""_udl`) that follows a strict set of rules described in this section. In the declaration and definition of a UDL operator, the name of the UDL suffix *may* be separated from the quotes by whitespace; in fact, some older compilers might even *require* such whitespace due to a defect (since corrected and retroactively applied) in the original C++11 specification. Thus, for all but the oldest C++11 compilers, **`operator`**`""_udl`, **`operator`**`"" _udl`, and **`operator`**` "" _udl` are all valid spellings of the same UDL operator name. Note that whitespace is *not* permitted between a literal and its suffix in the *use* of a UDL. For example, `1.2 _udl` is ill formed; `1.2_udl` must appear as a single token with no whitespace.

A UDL generally consists of two parts: (1) a valid lexical literal token and (2) a user-defined suffix. The signature of each UDL operator must conform to one of three patterns, distinguished by the way the compiler supplies the naked literal to the UDL operator:

1. **Prepared-argument UDL operator** — The naked literal is evaluated at compile time and passed into the operator as a value:

   ```
   Type1 operator"" _t1(unsigned long long n);
   Type1 t1 = 780_t1;  // calls operator""_t1(780ULL)
   ```

2. **Raw UDL operator** — The characters that make up the naked literal are passed to the operator as a *raw*, unevaluated string (for numeric literals only):

   ```
   Type2 operator"" _t2(const char *token);
   Type2 t2 = 780_t2;  // calls operator""_t2("780")
   ```

3. **Templated UDL operator** — The UDL operator is a template whose parameter list is a variadic sequence of **`char`** values (see Section 2.1."Variadic Templates" on page 519) that make up the naked literal (for numeric literals only):

   ```
   template <char...> Type3 operator"" _t3();
   Type3 t3 = 780_t3;  // calls operator""_t3<'7', '8', '0'>()
   ```

Each of these three forms of UDL operators is expounded in more detail in its own separate section; see *Prepared-argument UDL operators* on page 492, *Raw UDL operators* on page 495, and *Templated UDL operators* on page 498.

When a UDL is encountered, the compiler prioritizes a prepared-argument UDL operator over the other two. Given a UDL having suffix `_udl`, the compiler will look for any

`operator""_udl` in the local scope (**unqualified name lookup**). If, among the operators found, there is a prepared-argument UDL operator that exactly matches the type of the naked literal, then that UDL operator is called. Otherwise, for numeric literals only, the raw or templated UDL operator — only one of which is permitted to exist for a given suffix — is invoked. This set of lookup rules is deliberately short and rigid. Importantly, this lookup sequence differs from other operator invocations in that it does *not* involve **overload resolution** or argument conversions, nor does it employ **argument-dependent lookup (ADL)** to find operators in other namespaces.

Although ADL is never an issue for UDLs, common practice is to gather related UDL operators into a namespace (whose name often contains the word "literals"). This namespace is then typically nested within the namespace containing the definitions of the user-defined types that the UDL operators return. These literals-only nested namespaces enable a user to import, via a single **`using`** directive, just the literals into their scope, thereby substantially decreasing the likelihood of name collisions:

```
    std::size_t

namespace ns1  // namespace containing types returned by UDL operators
{
    struct Type1 { };
    bool check(const Type1&);

    namespace literals  // nested namespace for UDL operators returning ns1 types
    {
        Type1 operator"" _t1(const char*, std::size_t);
    }

    using namespace literals;  // Make literals available in namespace ns1.
}

void test1()  // file scope: finds UDL operator via using directive
{
    using namespace ns1::literals;  // OK, imports only the *inner* UDL operators
    check("xyzzy"_t1);              // OK, finds ns1::check via ADL
}
```

To use the `_t1` UDL suffix above, `test1` must somehow be able to find the declaration of its corresponding UDL operator locally, which is accomplished by placing the operator in a nested namespace and importing the entire namespace via a **`using` directive**. We could have avoided the nested namespace and, instead, required each needed operator to be imported individually:

```
    std::size_t

namespace ns2  // namespace defining types returned by non-nested UDL operators
{
    struct Type2 { };
    bool check(const Type2&);

    Type2 operator"" _t2(const char*, std::size_t);  // BAD IDEA: not nested
}
```

```
void test2()  // file scope: finds UDL operator via using declaration
{
    using ns2::operator"" _t2;  // OK, imports just the needed UDL operator
    check("xyzzy"_t2);          // OK, finds ns2::check via ADL
}
```

When multiple UDL operators are provided for a collection of types, however, the idiom of placing just the UDL operators in a nested namespace (typically incorporating the name "literals") obviates most of the commonly cited ill effects (e.g., accidental unwanted name collisions) attributed to more general use of **`using`** directives. In the interest of brevity, we will freely omit the nested-literal namespaces in expository-only examples.

Finally, despite its use in the Standard for this specific purpose, there is never a need for a namespace comprising only UDLs to be declared **`inline`** and doing so is contraindicated; see Section 3.1."**`inline namespace`**" on page 648.

rgument-udl-operators

Prepared-argument UDL operators

The *prepared-argument* pattern for UDL operators is supported for all four of the **UDL type categories**: integer, floating-point, character, and string. If this form of UDL operator is selected (see *UDL operators* on page 490), the compiler first determines which of the four UDL type categories applies to the naked literal, evaluates it without regard to the UDL suffix, and then passes the prepared (i.e., precomputed) value to the UDL operator, which further processes its argument and returns the value of the UDL. Note that the UDL type category of the UDL operator refers to only the naked literal; its return value can be any arbitrary type, with or without an obvious relationship to its type category:

```
    std::size_t

struct Smile { /* ... */ };  // arbitrary user-defined type

Smile operator"" _fx(long double);  // floating-point literal returning Smile
float operator"" _ix(unsigned long long);  // integer literal returning float
int operator"" _sx(const char*, std::size_t);  // string literal returning int
```

The UDL type category for a prepared-argument UDL operator is determined by the operator's signature; integer and floating-point UDL operators each take a single argument of, respectively, the largest unsigned integer or floating-point type defined by the language. Multiple prepared-argument UDL operators may be declared in the same scope for the same UDL suffix (e.g., `_aa`) and each may return a distinct arbitrary type (e.g., **`short`**, `Smile`):

```
short operator"" _aa(unsigned long long n);  // integer literal operator
Smile operator"" _aa(long double n);         // floating-point literal operator
bool  operator"" _bb(long double n);         // floating-point literal operator
```

The naked literal is evaluated as an **`unsigned long long`** for integer literals (see Section 1.1."**`long long`**" on page 78) or a **`long double`** for floating-point literals; the prepared value is then passed via the `n` parameter to the UDL operator. The compiler does the work of parsing and evaluating the sequence of digits, radix prefixes (e.g., `0` for octal and `0x` for hexadecimal), decimal points, and exponents. C++14 provides additional features re-

lated to built-in literals; see Section 1.2."Binary Literals" on page 131 and Section 1.2."Digit Separators" on page 141:

```
short v1 = 123_aa;  // OK, invokes operator"" _aa(unsigned long long)
Smile v2 = 1.3_aa;  // OK, invokes operator"" _aa(long double)
```

There is no overload resolution, **integer-to-floating-point conversion**, or **floating-point-to-integer conversion**, nor are UDL operators having numerically lower precision permitted:

```
Smile operator"" _cc(double n);  // Error, invalid parameter list
```

If lookup does not find a UDL operator that matches the type category of the naked literal exactly, the match fails:

```
bool v3 = 123_bb;  // Error, unable to find integer literal _bb
bool v4 = 1.3_bb;  // OK, invokes operator"" _bb(long double)
```

Because the naked literal is fully evaluated by the compiler, overflow and precision loss could become issues just as they are for native literals. These limitations vary by platform, but typical platforms are limited to 64-bit **unsigned long long** or 64-bit **long double** types in IEEE 754 format:

```
Smile v5 = 1.2e310_aa;              // Bug, argument evaluates to infinity
Smile v6 = 2.5e-310_aa;             // Bug, argument evaluates to denormalized
short v7 = 0x1234568790abcdef0_aa;  // Error, doesn't fit in any integer type
```

Note that the oversized integer initializer for `v7` in the code snippet above results in an error on some compilers but only a warning on others.

A rarely used feature of C++03 is the **encoding prefix**, `L`, on a character or string literal. The literals `'x'` and `"hello"` have (built-in) types **char** and **char**`[6]`, respectively, whereas `L'x'` and `L"hello"` have the respective types **wchar_t** and **wchar_t**`[6]`. C++11 added three more string encoding prefixes: `u` to indicate **UTF-16** with a type of **const char16_t***, `U` to indicate **UTF-32** with a type **const char32_t***, and `u8` to indicate **UTF-8** with a type of **const char***. The `u` and `U` prefixes can also be used on character literals; see Section 1.1."Unicode Literals" on page 117.[1]

The four C++11 encoding prefixes for character literals can each be supported by a distinct UDL operator signature (e.g., `_dd` below), each of which might return a distinct, arbitrary type:

```
int         operator"" _dd(char     ch);  //  'x'
double      operator"" _dd(char16_t ch);  // u'x'
const char* operator"" _dd(char32_t ch);  // U'x'
Smile       operator"" _dd(wchar_t  ch);  // L'x'
```

Any or all of the above forms can co-exist. A character naked literal (e.g., `'Q'`) is translated into the appropriate character type and value in the **execution character set** (i.e., the set of characters used at run time on the target operating system) and passed via the `ch` parameter to the body of the UDL operator. As ever, there are no narrowing or widening

[1] C++17 allows the `u8` prefix on character literals as well as on string literals, while C++20 changes the type for `u8` prefixed string and character literals from **char** to **char8_t**.

conversions, so the program is ill formed if the precise signature of the needed UDL operator is not found:

```
int    operator"" _ee(char);      //  'x'
double operator"" _ee(char32_t);  // U'x'

int         c1 =  'Q'_ee;  // OK, matches char parameter type
double      c2 = U'Q'_ee;  // OK, matches char32_t parameter type
const char* c3 = u'Q'_ee;  // Error, no match for char16_t parameter type
Smile       c4 = L'Q'_ee;  // Error, no match for wchar_t parameter type
```

Similarly, there are four valid UDL operator signatures for string literals in C++11, each of which again might return a different type[2]:

```
bool  operator"" _dd(const char*     str, std::size_t len);  //   "str"
int   operator"" _dd(const char16_t* str, std::size_t len);  //  u"str"
float operator"" _dd(const char32_t* str, std::size_t len);  //  U"str"
Smile operator"" _dd(const wchar_t*  str, std::size_t len);  //  L"str"
```

The string naked literal evaluates to a null-terminated character array. The address of the first element of that array is passed to the UDL operator via the `str` parameter, and its length — excluding the null terminator — is passed via `len`.

To recap, multiple prepared-argument UDL operators can co-exist for a single UDL suffix, each having a distinct type category and each potentially returning a different C++ type. To show another example, the suffix `_s` on a floating-point literal could return an **`double`** to mean seconds, whereas the same suffix on a string literal might return an `std::string`. Moreover, string and character UDL operators that differ by character type (**`char`**, **`char16_t`**, and so on) often have different return types. Similar string UDL operators typically — but not necessarily — return similar types, such as `std::string` and `std::u16string`, that differ only in their underlying character type:

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

double operator"" _s(unsigned long long);  // integer UDL operator
double operator"" _s(long double);         // floating-point UDL operator

std::string    operator"" _s(const char*,     std::size_t);  // string UDL
std::u16string operator"" _s(const char16_t*, std::size_t);  //   "     "

double         d = 12_s;       // yields double having value 12.0
std::u16string w = u"Hola"_s;  // yields std::u16string having value Hola
std::string    s = "Hello"_s;  // yields std::string having value Hello
```

Note again that these operators are invariably declared **`constexpr`** (see Section 2.1."**`constexpr`** Functions" on page 239) because they can *always* be evaluated at compile time because their arguments are necessarily expressed *literally* in the source code.

[2] C++20 allows a fifth signature for string literals with the `u8` prefix that takes a **`const char8_t*`**.

raw-udl-operators

Raw UDL operators

The *raw* pattern for UDL operators is supported for only the integer and floating-point UDL type categories. If this form of UDL operator is selected (see *UDL operators* on page 490), the compiler packages up the naked literal as an unprocessed string — i.e., a sequence of raw characters transferred from the source — and passes it to the UDL operator as a null-terminated character string. All raw UDL operators (e.g., for suffix `_rl` below) have the same signature:

```
struct Type { /*...*/ };

Type operator"" _rl(const char*);

Type t1 = 425_rl;  // invokes operator ""_rl("425")
```

This signature can be distinguished from a prepared-argument UDL operator for string literals by the *absence* of an `std::size_t` parameter representing its *length*.

The raw string argument will be verified by the compiler to be a well-formed integer or floating-point literal token but is otherwise untouched by the compiler. For any given UDL suffix, at most one matching raw UDL operator may be in scope at a time; hence, the *return* type cannot vary based on, e.g., whether the naked literal contains a decimal point. Such a capability *is*, however, available; see *Templated UDL operators* on page 498.

In particular, it might be the case that not all valid tokens accepted by the compiler will satisfy the **narrow contract** offered by a specific UDL operator. For example, we can define a UDL suffix, `_3`, to express base-3 integers using a raw UDL operator:

```
int operator"" _3(const char* digits)
{
    int result = 0;

    while (*digits)
    {
        result *= 3;
        result += *digits - '0';
        ++digits;
    }

    return result;
}
```

We can now test this function at run time using the standard `assert` macro:

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

void test()
{
    assert( 0 ==   0_3);
    assert( 1 ==   1_3);
    assert( 2 ==   2_3);
    assert( 3 ==  10_3);
    assert( 4 ==  11_3);
```

```
    // ...
    assert( 8 ==  22_3);
    assert( 9 == 100_3);
    assert(10 == 101_3);
}
```

Note that we could have declared our `_3` raw UDL operator **`constexpr`** and replaced all of the (runtime) `assert` statements with (compile-time) **`static_assert`** declarations. Also observe that, as written, **`constexpr operator""`** `_3` requires C++14's relaxed **`constexpr`** function restrictions (see Section 2.1."**`constexpr`** Functions" on page 239), but it could be reimplemented to work in C++11.

Let's now consider valid lexical integer literals representing values outside of what would be considered valid of base-3 integers:

```
int i1 = 22_3;                    // (1) OK, returns (int) 8
int i2 = 23_3;                    // (2) Bug, returns (int) 9
int i3 = 21.1_3;                  // (3) Bug, returns (int) 58
int i4 = 22211100022211100022_3;  // (4) Bug, too big for 32-bit int
```

In the example code above, (1) is a valid base-3 integer; (2) is a valid integer literal but contains the digit 3, which is *not* a valid base-3 digit; (3) is a valid floating-point literal, but the UDL operator returns values of only type **`int`**; and (4) is in principle a valid integer literal but represents a value that is too large to fit in a 32-bit **`int`**. Because cases (2), (3), and (4) are valid lexical literals, it is up to the implementation of the UDL operator to reject invalid values.

Let's now consider a more robust implementation of a base-3 integer UDL, `_3b`, that throws an exception when the literal fails to represent a valid base-3 integer:

```
#include <stdexcept>  // std::out_of_range, std::overflow_error
#include <limits>     // std::numeric_limits

int operator"" _3b(const char *digits)
{
    int ret = 0;

    for (char c = *digits; c; c = *++digits)
    {
        if ('\'' == c)  // Ignore the C++14 digit separator.
        {
            continue;
        }

        if (c < '0' || '2' < c)  // Reject non-base-3 characters.
        {
            throw std::out_of_range("Invalid base-3 digit");
        }

        if (ret >= (std::numeric_limits<int>::max() - (c - '0')) / 3)
        {
            // Reject if 3 * ret + (c - '0') would overflow.
```

```
            throw std::overflow_error("Integer too large");
        }

        ret = 3 * ret + (c - '0');  // Consume c.
    }

    return ret;
}
```

In this implementation of a raw UDL operator for a suffix `_3b`, the first **`if`** statement looks for the C++14-only digit separator and ignores it. The second **`if`** statement looks for characters not in the valid range for base-3 and, for cases (2) and (3), throws an `out_of_range` exception. The third **`if`** statement determines whether the computation is about to overflow and, for case (4), throws an `overflow_error` exception. The absence of compiler interpretation makes raw UDL operators potentially difficult to write but also powerful. Interpreting existing literal characters in a new way and accepting literals that would otherwise overflow or lose precision confers additional expressiveness on raw UDLs; e.g., the base-3 raw UDL operator shown above could not be expressed using a prepared-argument UDL operator. The downside of this additional expressiveness is the need to implement, debug, and maintain custom token parsing, including error checking.

If the intent is to consume integer literals, a raw UDL operator needs to either process or reject not only the decimal digits `'0'` to `'9'`, but also the hex digits `'a'`–`'f'` and `'A'`–`'F'` as well as radix prefixes `0`, `0x`, and `0X`. In C++14, the operator must also handle the `0b` (binary) radix prefix and the digit separator `'\''`.

For floating-point literals, the raw UDL operator needs to handle the decimal digits, decimal point, exponent prefix (`'e'` or `'E'`), and optional exponent sign (`'+'` or `'-'`). It is possible to handle both integer and floating-point literals in a single raw UDL operator, provided the return type is the same for both. In all cases, it is usually wise to reject *any* unexpected character, including characters that are not currently legal within numeric literals, in case the set of legal characters is enlarged in the future, e.g., by adding a new radix.

A raw UDL operator can be declared **`constexpr`**, but be aware that the rules for what is allowed in a **`constexpr`** function differ significantly between C++11 and C++14, as described in Section 2.1."**`constexpr`** Functions" on page 239. In particular, the loop-based implementation of **`operator`**`""_3` (above) can be declared **`constexpr`** in C++14 but not in C++11, though it is possible to define a **`constexpr`** UDL operator in C++11 that has the same behavior by using a recursive implementation. If the literal is evaluated as part of a constant expression and if the literal contains errors that would result in exceptions being thrown (i.e., an invalid character or overflow), the compiler will reject the invalid literal at compile time. If, however, the literal is not part of a constant expression, an exception will still be thrown at run time:

```
constexpr int i4 = 25_3;  // Error, "throw" not allowed in constant expression
          int i5 = 25_3;  // Bug, exception thrown at run time
```

To ensure that every invalid literal is detected at compile time, use templated UDL operators, as described in the next section.

mplated-udl-operators

Templated UDL operators

A templated UDL operator, known as a *literal operator template* in the Standard, is a variadic template (see Section 2.1."Variadic Templates" on page 519) having a template parameter list consisting of a pack of an arbitrary number of **`char`** parameters and an empty runtime parameter list:

```
struct Type { /*...*/ };

template <char...> Type operator"" _udl();
```

The templated UDL operator pattern supports only the integer and floating-point type categories.[3] If this form of UDL operator is selected (see *UDL operators* on page 490), the compiler breaks up the naked literal into a sequence of raw characters and passes each one as a separate template argument to the instantiation of the UDL operator:

```
Type t1 = 42.5_udl;  // calls operator""_udl<'4', '2', '.', '5'>()
```

As in the case of raw UDL operators, the raw sequence of characters will be verified by the compiler to be a well-formed integer or floating-point literal token, but the UDL operator must deduce meaning from those characters. Unlike raw UDL operators, a templated UDL operator can return different types based on the content of the naked literal. For example, electrical resistance might be expressed as `Resistance<float>` or `Resistance<int>`, where the intention is to express resistance less than 10 ohms using the **`float`** specialization and larger resistance using the **`int`** specialization:

```
template <class T> class Resistance;
template <> class Resistance<int>   { /* ... (resistance >= 10 ohms) */ };
template <> class Resistance<float> { /* ... (resistance <  10 ohms) */ };
```

The `_ohms` UDL operator (in the code snippet below) determines the correct return type and value based on the characters making up the naked numeric literal. The compile-time template logic needed to make this selection requires template metaprogramming, which in turn requires partial template specialization. Function templates, including templated UDL operators, cannot have partial specializations, so the selection logic is delegated to a helper class template, `MakeResistance`:

```
template <char C, char... Cs> struct MakeResistance;

template <char... Cs>
constexpr typename MakeResistance<Cs...>::ReturnType
operator"" _ohms() { return MakeResistance<Cs...>::factory(); }
```

`MakeResistance<c...>::ReturnType` is the template metafunction used to compute the return type. In C++14, the return type can be deduced directly from the **`return`** statement; see Section 3.2."Deduced Return Type" on page 687. The implementation of `MakeResistance` uses partial specializations to detect specific numeric literal patterns:

```
template <char d0, char... d>
struct MakeResistance  // primary template (for value >= 10)
```

[3] C++20 added support for user-defined string literal templates, albeit with a different syntax.

```
{
    using ReturnType = Resistance<int>;
    static constexpr ReturnType factory();
};

template <char d0>
struct MakeResistance<d0>  // specialize for single digit ('0' - '9')
{
    using ReturnType = Resistance<float>;
    static constexpr ReturnType factory(); // resistance 0 to 9 ohms
};

template <char d0, char... d>
struct MakeResistance<d0, '.', d...>  // specialize for decimal point after
                                      // first digit (e.g., 1.23)
{
    using ReturnType = Resistance<float>;
    static constexpr ReturnType factory(); // resistance 0.0 to < 10.0
};
```

The primary `MakeResistance` template will return a `Resistance<int>` for any sequence of characters that does not match one of the partial specializations. The first partial specialization matches a single-digit integer literal (i.e., an integer value from 0 through 9). The second partial specialization matches a sequence of two or more characters where the second character is a decimal point. Thus, any character sequence representing a value less than 10 will return `Resistance<float>`:

```
Resistance<int>   r1 = 200_ohms;
Resistance<float> r2 = 5_ohms;
Resistance<float> r3 = 2.5_ohms;
```

Note that this simplified example does not recognize floating-point literals such as `12.5` or `.04` that don't have their decimal point as the second character. The description of fixed-point literals in *Use Cases — User-defined numeric types* on page 505 provides a more complete exposition of this sort of return-type selection, including details of the recursive template metaprogramming used to determine the return type and its value.

As template arguments, the characters that make up the naked literal are constant expressions and can be used with **`static_assert`** to force error detection at compile time. Unlike raw UDL operators, there is no risk of throwing an exception at run time, even when initializing a value in a non-**`constexpr`** context:

```
constexpr auto r4 = 12.5_ohms;  // Error, constexpr context
         auto r5 = 12.5_ohms;  // Bug, non-constexpr context
```

Being able to select a context-specific return type and to force compile-time error checking makes templated UDL operators the most expressive pattern for defining UDL operators. These capabilities come at the cost, however, of having to develop them using the less-than-readable template sublanguage in C++.

++14-standard-library

UDLs in the C++14 Standard Library

This book is primarily about modern C++ *language* features, but a short description of UDL suffixes in the Standard *Library* provides context for better understanding and appreciating the UDL language feature. These new suffixes (starting with C++14) make it easier to write software using standard strings, units of time, and complex numbers. Note that, because these are *standard* UDL suffixes, their names do not have a leading underscore.

A native string literal, without a suffix, describes an C-style array of characters, which decays to a pointer-to-character when passed as a function argument. The C++ Standard Library has had, from the start, string classes (`std::basic_string`, `std::string`, and `std::wstring`) that improve on C-style character arrays by providing proper **copy semantics**, equality comparison, variable sizing, and so on. With the advent of UDLs, we can finally create literals of these library string types by utilizing the standard `s` suffix. The UDL operators for string literals are in header file `<string>` in namespace `std::literals::string_literals`[4]:

```
#include <string>  // std::basic_string, related types, and UDL operators

using namespace std::literals::string_literals;  // std::basic_string UDLs
const char*    s1 =   "hello";    // Value decays to (const char *) "hello".
std::string    s2 =   "hello"s;   // value std::string("hello")
std::string    s3 = u8"hello"s;   // value std::string(u8"hello")
std::u16string s4 =  u"hello"s;   // value std::u16string(u"hello")
std::u32string s5 =  U"hello"s;   // value std::u32string(U"hello")
std::wstring   s6 =  L"hello"s;   // value std::wstring(L"hello")
```

Complex numbers can also be expressed using a more natural style, mimicking the notation used in mathematics. Within namespace `std::literals::complex_literals`, the suffixes `i`, `il`, and **`if`** are used to name **`double`**, **`long double`**, and **`float`** imaginary numbers, respectively. Note that all three suffixes work for both integer and floating-point literals:

```
#include <complex>  // std::complex and UDL operators

using namespace std::literals::complex_literals; // std::complex UDLs
std::complex<double>      c1 = 2.4 + 3i;    // value 2.4, 3.0
std::complex<long double> c2 = 1.2 + 5.1l;  // value 1.2L, 5.1L
std::complex<float>       c3 = 0.1f + 2.if;  // value 0.1F, 2.0F
```

The time utilities in the standard header, `<chrono>`, contain an elaborate and flexible system of units of duration. Each unit is a specialization of the class template `std::chrono::duration`, which is instantiated with a representation (either integral or floating-point) and a ratio relative to seconds. Thus, `duration<long, ratio<3600, 1>>` can represent an integral number of hours.

The `<chrono>` header also defines literal suffixes (in namespace `std::literals::chrono_literals`) having familiar names for time units such as `s` for seconds, `min` for minutes, and so on. Integer literals will yield a `duration` having an integral internal representation, and floating-point literals will yield one having a floating-point

[4] C++20 adds **`char8_t`** and changes the type of `s3` in the coding example from `std::string` to `std::u8string`.

internal representation:

```
#include <chrono>  // std::literals::chrono_literals

using namespace std::literals::chrono_literals;  // std::chrono::duration UDLs
auto d1 = 2h;      // 2 hours   (integral internal representation)
auto d2 = 1.3h;    // 1.3 hours (floating-point internal representation)
auto d3 = 10min;   // 10 minutes (integral)
auto d4 = 30s;     // 30 seconds (integral)
auto d5 = 250ms;   // 250 milliseconds (integral)
auto d6 = 90us;    // 90 microseconds (integral)
auto d7 = 104.ns;  // 104.0 nanoseconds (floating-point)
```

In the example above, the **`auto`** keyword (see Section 2.1."**`auto`** Variables" on page 183) is used to allow the compiler to deduce the correct type from the literal expression. Although simple integer duration types have convenient aliases such as `std::chrono::hours`, some types do not have a standard name for the corresponding `duration` specialization; e.g., `1.2hr` returns a value of type `std::chrono::duration<T, std::ratio<3600>>` where `T` is a signed integer of at least 23 bits and where the actual integer representation is implementation-defined. Naming a duration is even more complex when adding `duration`s together; the resulting `duration` type is selected by the library to minimize loss of precision:

```
auto d8 = 2h + 35min + 20s;  // integral, 9320 seconds (2:35:20 in seconds)
auto d9 = 2.4s + 100ms;       // floating-point, 2500.0 milliseconds
```

We are certain to see more UDL suffixes defined in future Standards.

use-cases-userdeflit

Use Cases

Wrapper classes

wrapper-classes

Wrappers can be used to add or remove capabilities for their (often built-in) underlying type. They assign *meaning* to a type and are thus useful in preventing programmer confusion or ambiguity in overload resolution. For example, an inventory-control system might track items by both part number and model number. Both numbers could be simple integers, but they have very different meanings. To prevent programming errors, we create wrapper classes, `PartNumber` and `ModelNumber`, each holding an **`int`** value:

```
class PartNumber
{
    int d_value;

public:
    constexpr PartNumber(int v) : d_value(v) { }
    // ...
};

class ModelNumber
{
    int d_value;
```

```
public:
    constexpr ModelNumber(int v) : d_value(v) { }
    // ...
};
```

Neither `PartNumber` nor `ModelNumber` defines integer operations such as addition or multiplication, so any attempt to modify one (other than by assignment) or add two such values would result in a compile-time error. Moreover, having wrapper classes allows us to overload on the different types, preventing overload resolution ambiguities. Without UDLs, however, we must represent `PartNumber` or `ModelNumber` literals by explicitly casting **`int`** literals to the correct type:

```
// operations on model and part numbers:
int inventory(ModelNumber n) { int count = 0; /*...*/ return count; }
int inventory(PartNumber n)  { int count = 0; /*...*/ return count; }
void registerPart(const char* shortName, ModelNumber mn, PartNumber pn) { }

PartNumber pn1 = PartNumber(77) + 90;  // Error, no operator+(PartNumber, int)

int c1 = inventory(77);                // Error, ambiguous overload
int c2 = inventory(ModelNumber(77));   // OK, call inventory(ModelNumber)
int c3 = inventory(PartNumber(77));    // OK, call inventory(PartNumber)

void registerParts1()
{
    registerPart("Bolt", PartNumber(77), ModelNumber(77));  // Error, reversed
    registerPart("Bolt", ModelNumber(77), PartNumber(77));  // OK, correct args
}
```

The code above allows the compiler to detect errors that would be easy to make had part and model numbers been represented as raw **`int`** values. Attempting to add to a part number is rejected, as is the attempt to call `inventory` without specifying whether part-number inventory or model-number inventory is desired. Finally, the `registerPart` function cannot be called with its arguments accidentally reversed.

We can now create UDL suffixes, `_part` and `_model`, to simplify our use of hard-coded part and model numbers, making the code more readable:

```
namespace inventory_literals
{
    constexpr ModelNumber operator"" _model(unsigned long long v) { return v; }
    constexpr PartNumber  operator"" _part (unsigned long long v) { return v; }
}

using namespace inventory_literals;  // Make literals available.
int c5 = inventory(77);              // Error, ambiguous overload
int c6 = inventory(77_model);        // OK, call inventory(ModelNumber).
int c7 = inventory(77_part);         // OK, call inventory(PartNumber).

void registerParts2()
{
```

```
    registerPart("Bolt", 77_part, 77_model); // Error, reversed model & part
    registerPart("Bolt", 77_model, 77_part); // OK, arguments in correct order
}
```

A wrapper class can also be useful for tracking certain compile-time attributes of a general-purpose type such as `std::string`. For example, a system that reads input from a user must sanitize each input string before passing it to, for example, a database. Raw input and sanitized input are both strings, but an unsanitized string must never be confused for a sanitized one. Thus, we create a wrapper class, `SanitizedString`, that can be constructed only by a member factory function, to which we give the easily-searchable name `fromRawString`:

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

class SanitizedString
{
    std::string d_value;

    explicit SanitizedString(const std::string& value) : d_value(value) { }

public:
    static SanitizedString fromRawString(const std::string& rawStr)
    {
        return SanitizedString(rawStr);
    }

    // ...

    friend SanitizedString operator+(const SanitizedString& s1,
                                     const SanitizedString& s2)
    {
        return SanitizedString(s1.d_value + s2.d_value);
    }

    // ...
};
```

Calling `SanitizedString::fromRawString` is deliberately cumbersome in an attempt to make developers think carefully before using it. This call might, however, be *too* cumbersome in situations where the safety of a literal string is not in question:

```
std::string getInput();  // Read (unsanitized) string from input.
bool isSafeString(const std::string& s);  // Determine whether s is safe.

void process(const SanitizedString& instructions);
    // Run the specified instructions.

void processInstructions1()
    // Read instructions from input and process them.
{
    // Read instructions from input.
    std::string instructions = getInput();
```

```
    if (isSafeString(instructions))
    {
       // String is considered safe; sanitize it.
       SanitizedString sanInstr = SanitizedString::fromRawString(instructions);
       // ...

       // Prepend a "begin" instruction, then process the instructions.

       process("Instructions = begin\n" + sanInstr);
           // Error, no operator+(const char*, SanitizedString)

       process(SanitizedString::fromRawString("Instructions = begin\n") +
               sanInstr);
           // OK, but cumbersome
    }
    else
    {
        // ...                  (error handling)
    }
}
```

The first call to `process` does not compile because, by design, we cannot concatenate a raw string and a sanitized string. The second call works but is unnecessarily cumbersome. Literal strings are *always* assumed to be safe (if there is proper code review) because they cannot originate from outside the program; there should be no need to call `SanitizedString::fromRawString`. Again, a UDL can make the code more compact and readable:

```
namespace sanitized_string_literals
{
    SanitizedString operator""_san(const char *str, std::size_t len)
        // Create a sanitized string.
    {
        return SanitizedString::fromRawString(std::string(str, len));
    }
}

void processInstructions2()
    // Read instructions from input and process them.
{
    using namespace sanitized_string_literals;

    // Read instructions from input.
    std::string instructions = getInput();

    if (isSafeString(instructions))
    {
       // The instructions string is considered safe; sanitize it.
       SanitizedString sanInstr = SanitizedString::fromRawString(instructions);
```

```
        // ...

        // Prepend a "begin" instruction, then process the instructions.

        process("Instructions = begin\n"_san + sanInstr);
            // OK, concatenate two sanitized strings.
    }
    // ...
}
```

This usage shows a case where the UDL is more than just convenient; because a UDL applies *only* to literals, it is largely immune to accidental misuse.

r-defined-numeric-types

User-defined numeric types

There is sometimes a need to represent an integer of indefinite magnitude, i.e., where computations are immune from overflow (within the bounds of available memory). A `BigNum` class, along with associated arithmetic operators, can represent such indefinite-magnitude integers:

```
namespace bignum
{
class BigNum
{
    // ...
};

BigNum operator+(const BigNum&);
BigNum operator-(const BigNum&);
BigNum operator+(const BigNum&, const BigNum&);
BigNum operator-(const BigNum&, const BigNum&);
BigNum operator*(const BigNum&, const BigNum&);
BigNum operator/(const BigNum&, const BigNum&);
BigNum abs(const BigNum&);
// ...
```

A `BigNum` literal must be able to represent a value larger than would fit in the largest built-in integral type, so we define the suffix using a raw UDL operator:

```
namespace literals
{
BigNum operator"" _bignum(const char *digits)  // raw literal
{
    BigNum value;
    // ...        (Compute BigNum from digits.)
    return value;
}
}  // close namespace literals

using namespace literals;
}  // close namespace bignum
```

```
using namespace bignum::literals;  // Make _bignum literal available.
bignum::BigNum bnval  = 587135094024263344739630005208021231626182814_bignum;
bignum::BigNum bigone = 1_bignum;  // small value, but still has type BigNum
```

The `BigNum` class is appropriate for large integers, but numbers having fractional parts have a different problem: The IEEE standard **`double`** floating-point type cannot exactly represent certain values, e.g., `24692134.03`. When **`double`**s are used to represent values, long summations may eventually produce an error in the hundredths place; i.e., the result will be off by `.01` or more. In financial calculations, where values represent money, errors of just one or two pennies might be unacceptable. For this problem, we turn to decimal fixed-point (rather than binary floating-point) arithmetic.

A decimal fixed-point representation for a number is one where the number of decimal places of precision is chosen by the programmer and fixed at compile time. Within the specified size and precision, every decimal value can be represented exactly — e.g., a fixed-point number with two decimal digits of precision can represent the value `24692134.03` exactly but cannot represent the value `24692134.035`. We'll define our `FixedPoint` class as a template, where the `Precision` parameter specifies the number of decimal places[5]:

```
#include <limits>  // std::numeric_limits
#include <string>  // std::string, std::to_string

namespace fixedpoint
{

template <unsigned Precision>
class FixedPoint
{
    long long d_data;  // integral data = value * pow(10, Precision)

public:
    constexpr FixedPoint() : d_data(0) { } // zero value
    constexpr FixedPoint(long long);       // Convert from long long.
    constexpr FixedPoint(double);          // Convert from double.

    constexpr FixedPoint(long long data, std::true_type /*isRaw*/)
        // Create a FixedPoint object with the specified data.  Note that
        // this is a "raw" constructor and no precision adjustment is made to
        // the data.
        : d_data(data) { }

    friend std::ostream& operator<<(std::ostream& stream, const FixedPoint& v)
        // Format the specified fixed-point number v, write it to the
        // specified stream, and return stream.
    {
        std::string str = std::to_string(v.d_data);
        // Insert leading '0's, if needed.
```

[5] A more complete and powerful fixed-point class template was proposed for standardization in **?**.

```
        if (str.length() < Precision)
            str.insert(0, (Precision - str.length()), '0');
        str.insert(str.length() - Precision, 1, '.');
        return stream << str;
    }
};
```

Our data representation is a **long long** (see Section 1.1."**long long**" on page 78), making the largest value that can be represented `std::numeric_limits<long long>::max() / pow(10, Precision)`, assuming an integer `pow` function. The output function, **operator**<<, converts `d_data` to a string and then inserts the decimal point into the correct location. The special "raw" constructor exists so that our UDL operator can easily construct a value without losing precision, as we'll see later; the unused second parameter is a dummy to distinguish it from other constructors.

We want to define a templated UDL operator to return a `FixedPoint` type such that, for example, `12.34` would return a value of type `FixedPoint<2>` (for the two decimal places), whereas `12.340` would return a value of type `FixedPoint<3>`. We must first define a variadic helper template (see Section 2.1."Variadic Templates" on page 519) to compute both the type and the raw value of the fixed point number, given a sequence of digits:

```
namespace literals  // fixed-point literals defined in this namespace
{

template <long long rawVal, int precision, char... c>
struct MakeFixedPoint;
```

This helper template will be recursively instantiated; at each level of recursion `rawVal` is the value computed so far, `precision` is the number of decimal places seen so far, and `c...` is the list of literal characters to be consumed. A special value of `-1` for `precision` indicates that the decimal point has not yet been consumed.

The base case of our recursive template occurs when the parameter pack, `c...`, is empty — i.e., there are no more characters to consume. In this case, the computed type is simply `FixedPoint<precision>`, and the value of the UDL operator is computed from `rawVal`. We define the base case as a partial specialization of `MakeFixedPoint` where there is no character parameter pack:

```
template <long long rawVal, int precision>
struct MakeFixedPoint<rawVal, precision>
{
    // base case when there are no more characters
    using type = FixedPoint<(precision < 0) ? 0 : precision>;
    static constexpr type makeValue() { return { rawVal, std::true_type{} }; }
        // Return the computed fixed-point number.
};
```

The other case occurs when there are one or more characters yet to be consumed. The helper must perform error checking for bad input characters and overflow before consuming the character and instantiating itself recursively:

```
template <long long rawVal, int precision, char c0, char... c>
```

```
struct MakeFixedPoint<rawVal, precision, c0, c...>
{
private:
    static constexpr long long maxData = std::numeric_limits<long long>::max();
    static constexpr bool      c0isdig = ('0' <= c0 && c0 <= '9');

    // Check for out-of-range characters and overflow.
    static_assert(c0isdig || '\'' == c0 || '.' == c0,
                  "Invalid fixed-point digit");
    static_assert(!c0isdig || (maxData - (c0 - '0')) / 10 >= rawVal,
                  "Fixed-point overflow");

    // precision is
    // (1) < 0 if a decimal point was not seen,
    // (2) 0   if a decimal point was seen but no digits after the decimal point,
    // (3) > 0 otherwise, incremented once for each digit after the decimal point.
    static constexpr int nextPrecision = ('.' == c0     ?  0 :
                                          precision < 0 ? -1 :
                                          precision + 1);

    // Instantiate this template recursively to consume remaining characters.
    using RecurseType = MakeFixedPoint<(c0isdig ? 10 * rawVal + c0 - '0' :
                                        rawVal), nextPrecision, c...>;

public:
    using                type = typename RecurseType::type;
    static constexpr type makeValue() { return RecurseType::makeValue(); }
        // Return the computed fixed-point number.
};
```

This specialization consumes one character, `c0`, from the parameter pack. The first **`static_assert`** checks that `c0` is either a digit, digit separator (`'\''`), or decimal point (`'.'`). The second **`static_assert`** checks that the computation is not in danger of overflowing the maximum value of a **`long long`**. The constant, `nextPrecision`, which will be passed to the recursive instantiation of this template, keeps track of how many digits have been consumed after the decimal point (or `-1` if the decimal point has not yet been consumed). The `RecurseType` alias is the recursive instantiation of this template with the updated raw value (after consuming `c0`), the updated precision, and the input character sequence after having dropped `c0`. Thus, each recursion gets a potentially larger `rawVal`, a potentially larger `precision`, and a shorter list of unconsumed characters. The definitions of `type` and `makeValue` simply defer to the definitions in the recursive instantiation.

Finally, we define the `_fixed` templated UDL operator, instantiating `MakeFixedPoint` with an initial `rawVal` of `0`, an initial `precision` of `-1`, and with a `c...` parameter pack consisting of all of the characters in the naked literal:

```
template <char... c>
constexpr typename MakeFixedPoint<0, -1, c...>::type operator"" _fixed()
{
    return MakeFixedPoint<0, -1, c...>::makeValue();
```

```
}

}  // Close namespace literals.
}  // Close namespace fixedpoint.
```

Now the `_fixed` suffix can be used for fixed-point UDLs where the precision of the returned type is automatically deduced based on the number of decimal places in the literal. Note that the literal can be used in a **`constexpr`** context:

```
    std::cout

int fixedTest()
{
    using namespace fixedpoint::literals;

    constexpr auto fx1 = 123.45_fixed;   // return type FixedPoint<2>
    constexpr auto fx2 = 123.450_fixed;  // return type FixedPoint<3>
    std::cout << fx1 << '\n';            // prints "123.45"
    std::cout << fx2 << '\n';            // prints "123.450"
}
```

An effort is underway to define a standard **decimal floating-point** type, which, like our decimal fixed-point type, retains the benefit of precisely representing decimal fractions but where the precision is variable at run time. If implemented as a library type, a UDL suffix would allow such type to have a natural representation in code.[6]

er-defined-string-types

User-defined string types

A universally unique identifier (UUID) is a 128-bit number that identifies specific pieces of data in computer systems. It can be readily expressed as an array of two 64-bit integers:

```
    std::size_t

#include <cstdint>  // std::uint64_t
class UUIDv4
{
    std::unit64_t d_value[2];
    // ...
};
```

A version-4 UUID has a canonical human-readable format consisting of five groups of hex digits separated by hyphens, e.g., `ed66b67a-f593-4def-9a9b-e69d1d6295ef`. Although storing this representation as a string would be easy, converting it using the packed, 128-bit integer format of the above `UUIDv4` class is more efficient and convenient. Moreover, UUIDs that are hard-coded into software are often generated by external tools — i.e., to identify the exact product build — and should therefore be compile-time constants. Using UDLs, we can readily express a UUID literal using the human-readable string format, converting it to a compile-time constant in the packed format:

```
namespace uuid_literals
{
```

[6] ?

```
    constexpr UUIDv4 operator""_uuid(const char* s, std::size_t len)
    {
        return { /* ... (decode UUID expressed in canonical format) */ };
    }
}

using namespace uuid_literals;
constexpr UUIDv4 buildId = "eeec1114-8078-49c5-93ca-fea6fbd6a280"_uuid;
```

and-dimensional-units

Unit conversions and dimensional units

UDLs can be convenient for specifying a unit name on a numeric literal, providing a concise way both to convert the number to a normalized unit and to annotate a value's unit within the code. For example, the standard trigonometric functions all operate on **double** values where angles are expressed in radians. However, many people are more comfortable with degrees than radians, especially when expressing the value directly as a handwritten number:

```
#include <cmath>  // std::sin, std::cos

constexpr double pi = 3.14159265358979311599796346854;

double s1 = std::sin(30.0);    // Bug, intended sin(30 deg) but got sin(30 rad)
double s2 = std::sin(pi / 6);  // OK, returns sin(30 deg)
```

The normalized unit in this case is a radian, expressed as a **double**, but radians are generally fractions of π and are thus inconvenient to write. UDLs can provide convenient normalization from degrees or gradians to radians:

```
namespace trig_literals {

constexpr double operator"" _rad(long double r)  { return r; }
constexpr double operator"" _deg(long double d)  { return pi * d / 180.0; }
constexpr double operator"" _grad(long double d) { return pi * d / 200.0; }

}

using namespace trig_literals;
double s3 = std::sin(30.0_deg);     // OK, returns sin(30 deg)
double s4 = std::sin(4.7124_rad);   // OK, returns approx -1.0
double s5 = std::cos(50.0_grad);    // OK, returns sin(50 grad) == sin(45 deg)
```

Unfortunately, the applicability of the above approach to unit normalization is very limited. First, the conversion is one way — e.g., the expression `std::cout << 30.0_deg` will print out `0.524`, not `30.0`, necessitating a call to a radians-to-degrees conversion function when a human-readable value is desired. Second, a **double** does not encode any information about the units that it holds, so **double** `inputAngle` doesn't tell the reader (or program) whether the angle is expected to be input in degrees or radians.

A more robust way to use UDLs to express units is to define them as part of a comprehensive library of unit classes. Dimensional quantities (length, temperature, currency, and so on) often benefit from being represented by **dimensional unit types** that prevent

confusion as to both the units and dimension of numeric values. For example, creating a function to compute kinetic energy from speed and mass seems simple:

```
double kineticE(double speed, double mass)
    // Return kinetic energy in joules given speed in mps and mass in kg.
{
    return speed * speed * mass;
}
```

Yet this simple function can be called incorrectly in numerous ways, with no compiler diagnostics to help prevent the errors:

```
double d1   = 15;             // distance in meters
double t1   = 4;              // time in seconds
double s1   = d1 / t1;        // speed in m/s (meters/second)
double m1   = 2045;           // mass in g
double m1Kg = 2045.0 / 1000;  // mass in kg

double x1 = kineticE(d1, m1Kg);  // Bug, distance instead of speed
double x2 = kineticE(m1Kg, s1);  // Bug, arguments reversed
double x3 = kineticE(s1, m1);    // Bug, mass should be in kg, not g
double x4 = kineticE(s1, m1Kg);  // OK, meters per sec and kg units
```

One way to detect some of these errors at compile time is to use a wrapper for each dimension:

```
struct Time     { constexpr Time(double sec);        /* ... */ };
struct Distance { constexpr Distance(double meters); /* ... */ };
struct Speed    { constexpr Speed(double mps);       /* ... */ };
struct Mass     { constexpr Mass(double kg);         /* ... */ };
struct Energy   { constexpr Energy(double joules);   /* ... */ };

// Compute speed from distance and time:
Speed operator/(Distance, Time);

Distance d2(15.0);             // distance in meters
Time     t2(4.0);              // time in seconds
Speed    s2(d2 / t2);          // speed in m/s (meters/second)
Mass     m2(2045.0);           // Bug, trying to get g, got kg instead
Mass     m2Kg(2045.0 / 1000);  // OK, mass in kg

Energy kineticE(Speed s, Mass m);
Energy x5 = kineticE(d2, m2Kg); // Error, 1st argument has an incompatible type.
Energy x6 = kineticE(m2Kg, s2); // Error, reversed arguments, incompatible types
Energy x7 = kineticE(s2, m2);   // Bug, mass should be in kg, not g.
Energy x8 = kineticE(s2, m2Kg); // OK, m/s and Kg units
```

Note that the compiler correctly diagnoses an error in the initializations of `x5` and `x6` but still fails to diagnose the unit error in the initialization of `x7`. User-defined literals can amplify the benefits of dimensional unit classes by adding unit suffixes to numeric literals, eliminating implicit unit assumptions:

```
namespace si_literals
```

```
{
    constexpr Distance operator"" _m  (long double meters);
    constexpr Distance operator"" _cm (long double centimeters);
    constexpr Time     operator"" _s  (long double seconds);
    constexpr Speed    operator"" _mps(long double mps);
    constexpr Mass     operator"" _g  (long double grams);
    constexpr Mass     operator"" _kg (long double kg);
    constexpr Energy   operator"" _j  (long double joules);
}

using namespace si_literals;
auto d3   = 15.0_m;    // distance in meters
auto t3   = 4.0_s;     // time in seconds
auto s3   = d3 / t3;   // speed in m/s (meters/second)
auto m3   = 2045.0_g;  // mass expressed as g but stored as Kg
auto m3Kg = 2.045_kg;  // mass expressed as kg

Energy x9  = kineticE(s3, m3);    // OK, m3 has been normalized to Kg units.
Energy x10 = kineticE(s3, m3Kg);  // OK, m/s and Kg units
```

Note that there are two UDLs that yield `Distance` and two UDLs that yield `Mass`. Typically, the internal representation of each of these dimensional types has a normalized representation; e.g., `Distance` might be represented in meters internally, so `25_cm` would be represented by a **`double`** data member with value `0.25`. It is also possible, however, for the unit to be stored alongside the value, thus avoiding rounding errors in certain cases. Better yet, the unit can be encoded as a template parameter at compile time:

```
#include <ratio>  // std::ratio
template <class Ratio> class MassUnit;
using Grams     = MassUnit<std::ratio<1, 1000>>;
using Kilograms = MassUnit<std::ratio<1>>;

namespace unit_literals
{
    constexpr Grams     operator"" _g  (long double grams);
    constexpr Kilograms operator"" _kg (long double kg);
}
```

We now get *different* types for `100_g` and `0.1_kg`, but we can define `MassUnit` in such a way that they interoperate. The time-interval units that we saw previously in UDLs in the C++14 Standard Library provide a taste of what is possible with this approach.[7]

test-drivers

Test drivers

Because code is easier to maintain when "magic" values are expressed as named constants rather than literal values, a typical program does not contain many literals (see *Potential Pitfalls — Overuse* on page 515). The exception to this general rule is in unit tests where many different values are successively passed to a subsystem to test its behavior. For ex-

[7] Mateusz Pusz explores the topic of a comprehensive physical units library in **?**.

ample, we define a `Date` class that supplies a subtraction operator returning the number of days between two `Date`s:

```
// date.h: (component header file)

class Date
{
    // ...
public:
    constexpr Date(int year, int month, int day);
    // ...
};

int operator-(const Date& lhs, const Date& rhs);
    // Return the number of days from rhs to lhs.
```

To test the subtraction operator, we need to feed it combinations of dates and compare the result with the expected result. We do this by creating an array where each row holds a pair of dates and the expected result from subtracting them. Due to the large number of hard-coded values in the array, having a literal representation for the `Date` class would be convenient, even if the author of `Date` did not see fit to provide one:

```
// date.t.cpp: (component test driver)

#include <date.h>   // Date
#include <cstdlib>  // std::size_t
#include <cassert>  // standard C assert macro

namespace test_literals
{
    constexpr Date operator"" _date(const char*, std::size_t);
        // UDL to convert date in "yyyy-mm-dd" format to a Date object
}

void testSubtraction()
{
    using namespace test_literals;  // Import _date UDL suffix.

    struct TestRow
    {
        Date lhs;  // left operand
        Date rhs;  // right operand
        int  exp;  // expected result
    };

    const TestRow testData[] =
    {
        { "2021-01-01"_date, "2021-01-01"_date,  0 },
        { "2021-01-01"_date, "2020-12-31"_date,  1 },
        { "2021-01-01"_date, "2021-01-02"_date, -1 },
```

```
        // ...
    };

    const std::size_t testDataSize = sizeof(testData) / sizeof(TestRow);

    for (std::size_t i = 0; i < testDataSize; ++i)
    {
        assert(testData[i].lhs - testData[i].rhs == testData[i].exp);
    }
}
```

itfalls-defmemberinit

-can-yield-bad-values

Potential Pitfalls

Unexpected characters can yield bad values

Raw UDL operators and templated UDL operators must parse and handle every character from the *union* of the set of legal characters in integer and floating-point literals, even if the UDL operator is expecting only one of the two numeric type categories. Failure to generate an error for an invalid character is likely to produce an incorrect value, rather than a program crash or compilation error:

```
short operator"" _short(const char *digits)
{
    short result = 0;
    for (; *digits; ++digits)
    {
        result = result * 10 + *digits - '0';
    }

    return result;
}

short s1 = 123_short;   // OK, value 123
short s2 = 123._short;  // Bug, '.' treated as digit value -2
```

Testing only for the *expected* characters and rejecting any others is better than checking for *invalid* characters and accepting the rest:

```
#include <stdexcept>  // std::out_of_range

short operator"" _shrt2(const char *digits)
{
    short result = 0;
    for (; *digits; ++digits)
    {
        if (*digits == '.')
        {
            throw std::out_of_range("Bad digit");  // BAD IDEA
        }
```

```
        if (!std::isdigit(*digits))
        {
            throw std::out_of_range("Bad digit");  // BETTER
        }

        result = result * 10 + *digits - '0';
    }

    return result;
}

short s3 = 123_shrt2;    // OK, value 123
short s4 = 123._shrt2;   // Error (detected), throws out_of_range("Bad digit")
short s5 = 0x123_shrt2;  // Error (detected), throws out_of_range("Bad digit")
```

The first `if` will catch an unexpected decimal point but not an unexpected characters such as `'e'`, `'x'`, `'\'`. The second `if` will catch all unexpected characters. If a new radix or other currently illegal character is introduced in a future Standard, the second `if` will avoid processing it incorrectly. Note that, for example, after UDLs were added in C++11, both the `0b` radix and the digit separator (`'`) were introduced in C++14, potentially breaking any C++11-compliant UDL operator that didn't properly handle those characters.

overuse-defmemberinit

Overuse

Although UDLs offer conciseness, they aren't always necessarily the most effective way to create a literal value in a program. Sometimes a regular constructor or function call might be almost as concise, simpler, and more flexible. For example, `deg(90)` is as readable as `90_deg` and can be applied to runtime values as well as to literals. If the constructor for a type naturally takes two or more arguments, a string UDL operator could theoretically parse a comma-separated list of arguments — e.g., `"(2.0, 6.0)"_point` to represent a 2-D coordinate — but is the literal really easier to read than `Point(2.0, 6.0)`?

Finally, even for simple cases, consider how *often* a literal is likely to be used. The use of "magic numbers" in code is widely discouraged. Numeric literals other than, say, `-1`, `0`, `1`, `2`, and `10` or string literals other than `""` are typically used only to initialize named constants. The speed of sound, for example, would probably be written as a constant, e.g., `speedOfSound`, rather than a literal, `343_mps`. Using literals to supply the special values used to initialize named constants provides little benefit by way of overall program readability:

```
constexpr Speed operator"" _mps(unsigned long long speed);  // meters per second

constexpr Speed speedOfSound1 = 343_mps;   // OK, very clear
constexpr Speed speedOfSound2(343);        // OK, almost as clear

constexpr Speed mach2 = 2 * speedOfSound1; // Literal is irrelevant.
constexpr Speed mach3 = 3 * speedOfSound2; // Literal is irrelevant.
constexpr Speed mach4 = 4 * 343_mps;       // Bad style: "magic" number
```

Often, the most common literal is the one that expresses the notion of an *empty* or *zero* value. A clearer, more descriptive alternative might be to create named constants, such as `constexpr Thing k_EMPTY_THING`, instead of defining a UDL operator just to be able to write `""_thing` or `0_thing`.

reprocessor-surprises

Preprocessor surprises

A string literal with a suffix, e.g., `"hello"_wrld`, is a single token in C++11 but was two tokens, `"hello"` and `_wrld`, in previous versions of the language. This change could manifest in a subtle difference in meaning, usually resulting in a compilation error, if `_wrld` is a macro:

```
#define _wrld " world"
const char* s = "hello"_wrld;  // "hello world" in C++03, UDL in C++11
```

Verbose usage

One of the main selling points of user-defined literals is enabling developers to write concise and expressive code. However, since literal operators are commonly bundled together in a separate namespace, the required `using` directive on the caller side can become a burden if only one or few uses of a literal operator appear in a scope:

```
    std::size_t

bool isAfterLunarLanding(Date d)
{
    using namespace test_literals;  // Import _date UDL suffix.
    return d > "1969-07-20"_date;
}
```

The authors of the `_date` UDL should provide its same functionality as a function or constructor to ensure that users do not always require a possibly over-verbose `using` directive to achieve their goals:

```
bool isAfterLunarLanding(Date d)
{
    return d > Date("1969-07-20");
}
```

Annoyances

annoyances

No conversion from floating-point to integer UDL

-point-to-integer-udl

Defining a prepared-argument floating-point UDL operator does not make the corresponding suffix available to numeric literals that look like integers and vice versa:

```
double operator"" _mpg(long double v);

double v1 = 12_mpg;   // Error, no integer UDL operator for _mpg
double v2 = 12._mpg;  // OK, floating-point UDL operator for _mpg found
```

If the intention is to define a prepared-argument UDL where numbers with and without a decimal point are accepted, then both forms of the UDL operator must be defined.

-suffix-name-collisions

Potential suffix-name collisions

Using a UDL suffix requires bringing the corresponding UDL operator into the current scope, e.g., by means of a **using** directive. If the scope is large enough and if more than one imported namespace contains a UDL operator with the same name, a name collision can result:

```
using namespace trig_literals;        // _deg, _rad, and _grad suffixes
using namespace temperature_literals; // colliding _deg suffix

auto d = 12.0_deg;  // Error, ambiguous use of suffix, _deg
```

While it is possible to disambiguate the colliding suffixes via qualified name lookup, the ensuing verbosity might defeat the purpose of using a UDL in the first place:

```
auto a = trig_literals::operator""_deg(12.0);
auto b = temperature_literals::operator""_deg(12.0);
```

th-string-udl-operators

Confusing raw with string UDL operators

A UDL operator that takes a single **const char*** argument is a raw UDL operator for numeric literals but can be easily confused for a prepared-argument UDL operator for string literals:

```
int operator"" _udl(const char *);

int s = "hello"_udl;  // Error, no match for operator""(const char*, size_t)
```

Fortunately, such a problem will typically result in a compile-time error.

ors-for-string-literals

No templated UDL operators for string literals

Templated UDL operators are called only for numeric literals; string literals are limited to the prepared-argument pattern. It is thus not possible to choose, at compile time, different return types based on the contents of a string literal.[8]

-parse-a-leading---or-+

No way to parse a leading - or +

As described in *Description — Restrictions on UDLs* on page 489, a `-` or `+` before a numeric literal is a separate negation operator and not part of the literal. There are occasions, however, where it would be convenient to know whether the literal value is being negated. For example, if temperatures are being stored as **double** values in Kelvin and if the UDL suffix `_C` converts a floating-point literal from Celsius to Kelvin by calling a function, `cToK(double)`, then the expression `-10.0_C` produces the nonsensical value `-283.15` (`-cToK(10.0)`) rather than the intuitive value of `+263.15` (`cToK(-10.0)`). Alas, parsing the `-` sign as part of the literal is simply not possible.

[8] As of C++20, new syntax is added that removes the limitation of not being able to affect the return-type base on the contents of a literal string.

rsing-numbers-is-hard

Parsing numbers is hard

Many of the benefits of raw UDL operators and templated UDL operators require parsing integer and/or floating-point values manually, in code, often using recursion. Getting this right is tedious at best. The Standard Library does not provide much support, especially for **`constexpr`** parsing.

see-also

See Also

- "**`decltype`**" (§1.1, p. 22) ♦ introduces a keyword often helpful for deducing the return type of a templated UDL operator.
- "**`nullptr`**" (§1.1, p. 87) ♦ describes a keyword that unambiguously denotes the null pointer literal.
- "**`auto`** Variables" (§2.1, p. 183) ♦ shows how type inference can be used to declare a variable to hold the value of a UDL when the type of the UDL varies based on its contents.
- "**`constexpr`** Functions" (§2.1, p. 239) ♦ describes how most UDLs can be used as part of a constant expression.
- "Variadic Templates" (§2.1, p. 519) ♦ shows how templates can take an infinite number of parameters, which is required for implementing templated UDL operators.
- "**`inline namespace`**" (§3.1, p. 648) ♦ describes a feature not recommended for UDL operators, yet the C++14 Standard Library puts UDL operators into **`inline`** namespaces.

further-reading

Further Reading

- ?
- For a discussion of user-defined literals with recommendations for idiomatic use, see ?.

Variable-Argument-Count Templates

variadic-templates

By Andrei Alexandrescu

Variadic templates provide language-level support for specifying templates that accept an arbitrary number of template arguments.

Description

description-variadic

Experience with C++03 revealed a recurring need to specify a class or function that accepts an arbitrary number of arguments. The C++03 workarounds often require considerable boilerplate and hardcoded limitations that impede usability. Consider, for example, a function `concat` taking zero or more `const std::string&`, `const char*`, or `char` arguments and returning an `std::string` that is the concatenation of that argument sequence:

std::string

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

std::string d = "d";
std::string str0 = concat();               // str0 == "" (by definition)
std::string str1 = concat("apple");        // str1 == "apple"
std::string str2 = concat('b', "ccd");     // str2 == "bccd"
std::string str3 = concat(d, 'e', "fg");   // str3 == "defg"
```

One advantage of using a variadic function, such as `concat`, instead of repeatedly using the `+` operator is that the `concat` function can build the destination string exactly once, whereas each invocation of `+` creates and returns a new `std::string` object.

A simpler example is a variadic function, `add`, that calculates the sum of zero or more integer values supplied to it:

```
int v0 = add();         // v0 ==  0 (by definition)
int v1 = add(3);        // v1 ==  3
int v2 = add(-6, 2);    // v2 == -4
int v4 = add(7, 1, 4);  // v3 == 12
// ...
```

Historically, variadic functions, such as `concat` and `add` (above), were implemented as a suite of related non-variadic functions accepting progressively more arguments, up to some arbitrary limit (e.g., 20) chosen by the implementer:

```
int add();
int add(int);
int add(int, int);
int add(int, int, int);
// ...
// ... declarations from 4 to 19 int parameters elided
// ...
int add(int, int, int, /* 16 more int parameters elided, */ int);
```

Given the existence of an identity value (zero for addition), an alternative approach would be to have a single `add` function but with each of the parameters defaulted:

```
int add(int=0, int=0, /* 17 more defaulted parameters omitted */ int=0);
```

However, `concat` cannot use the same approach because each of its arguments may be a **`char`**, a **`const char*`**, or an `std::string` (or types convertible thereto), and no single type accommodates all of these possibilities. To accept an arbitrary mix of arguments of the allowed types with maximal efficiency, the brute-force approach would require the definition of an exponential number of overloads taking any combination of **`char`**, **`const char*`**, and **`const`** `std::string&`, again up to some implementation-chosen maximum number of parameters, N.

With such an approach, the number of required overloads is $O(3^N)$, which means that accommodating a maximum of just 5 arguments would require 283 overloads and 10 arguments would require 88,573 overloads![1] Defining a suite of N function *templates* instead is one approach to avoiding this combinatorial explosion of overloads:

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

std::string concat();                                       // 0 arguments

template <typename T1>
std::string concat(const T1&);                              // 1 argument

template <typename T1, typename T2>
std::string concat(const T1&, const T2&);                   // 2 arguments

template <typename T1, typename T2, typename T3>
std::string concat(const T1&, const T2&, const T3&);       // 3 arguments

// ...
// ... similar declarations taking up to, say, 20 parameters
// ...
```

Using conventional function templates, we can drastically reduce the volume of source code required, albeit with some manageable increase in implementation complexity.

Each of the $N+1$ templates can be written to accept any combination of its M arguments ($0 \leq M \leq N$) such that each parameter will independently bind to a **`const char*`**, an `std::string`, or a **`char`** with no unnecessary conversions or extra copies at run time. Of the exponentially many possible `concat` template instantiations, the compiler generates — on demand — only those overloads that are actually invoked.

With the introduction of variadic templates in C++11, we are now able to represent variadic functions such as `add` or `concat` with just a single template that expands automatically to accept any number of arguments of any appropriate types — all by, say, **`const`** *lvalue* reference:

[1] The `std::string_view` standard library type introduced with C++17 would help here because it accepts conversion from both **`const char*`** and `std::string` and incurs no significant overhead. However, `std::string_view` cannot be initialized from a single **`char`**, so we'd be looking at $O(2^N)$ instead of $O(3^N)$ — not a dramatic improvement.

```
    std::string

template <typename... Ts>
std::string concat(const Ts&...);
     // Return a string that is the concatenation of a sequence of zero or
     // more character or string arguments --- each of potentially distinct
     // C++ type and passed by const lvalue reference.
```

A variadic function template will typically be implemented with **recursion** to the same function with fewer parameters. Such function templates will typically be accompanied by an overload (templated or not) that implements the lower limit, in our case, the overload having exactly zero parameters:

```
std::string concat();
    // Return an empty string ("") of length 0.
```

The non-template overload above declares `concat` taking no parameters. Importantly, this overload will be preferred for calls of `concat` with no arguments because it's a better match than the variadic declaration, even though the variadic declaration would also accept zero arguments.

Having to write just two overloads to support any number of arguments has clear advantages over writing dozens of overloaded templates: (1) there is no hard-coded limit on argument count, and (2) the source is dramatically smaller, more regular, and easier to maintain and extend — e.g., it would be easy to add support for efficiently passing by forwarding reference (see Section 2.1."Forwarding References" on page 351). A second-order effect should be noted as well. The costs of defining variadic functions with C++03 technology are large enough to discourage such an approach in the first place, unless overwhelming efficiency motivation exists; with C++11, the low cost of defining variadics often makes them the simpler, better, and more efficient choice altogether.

Variadic *class* templates are another important motivating use case for this language feature.

A tuple is a generalization of `std::pair` that, instead of comprising just two objects, can store an arbitrary number of objects of heterogeneous types:

```
    std::string

Tuple<int, double, std::string> tup1(1, 2.0, "three");
    // tup1 holds an int, a double, and an std::string

Tuple<int, int> tup2(42, 69);
    // tup2 holds two ints
```

`Tuple` provides a container for a specified set of types, in a manner similar to a **`struct`**, but without the inconvenience of needing to introduce a new **`struct`** definition with its own name. As shown in the example above, the tuples are also initialized correctly according to the specified types (e.g., `tup2` contains two integers initialized, respectively, with `42` and `69`).

In C++03, an approximation to a tuple could be improvised by composing an `std::pair` with itself:

```
#include <utility>  // std::pair
```

```
std::pair<int, std::pair<double, long>> v;
    // Define a holder of an int, a double, and an long, accessed as
    // v.first, v.second.first, and v.second.second, respectively.
```

Composite use of `std:pair` types could, in theory, be scaled to arbitrary depth; defining, initializing, and using such types, however, is not always practical. Another approach commonly used in C++03 and similar to the one suggested for the `add` function template above is to define a template class, e.g., `Cpp03Tuple`, having many parameters (e.g, 9), each defaulted to a special marker type (e.g., `None`), indicating that the parameter is not used:

```
struct None { };  // empty "tag" used as a special "not used" marker in Tuple

template <typename T1 = None, typename T2 = None, typename T3 = None,
          typename T4 = None, typename T5 = None, typename T6 = None,
          typename T7 = None, typename T8 = None, typename T9 = None>
class Cpp03Tuple;
    // struct-like class containing up to 9 data members of arbitrary types
```

`Cpp03Tuple` may be used to store, access, and modify up to 9 values together, e.g., `Cpp03Tuple<int, int, std::string>` would consist of two `int`s and an `std::string`.

`Cpp03Tuple`'s implementation uses a variety of **metaprogramming** tricks to detect which of the 9 type slots are used. This is the approach taken by `boost::tuple`,[2] an industrial-strength tuple implemented using C++03-era technology. In contrast, the variadic-template-based declaration (and definition) of a modern C++ tuple is much simpler:

```
template <typename... Ts>
class Cpp11Tuple;  // class template storing an arbitrary sequence of objects
```

C++11 introduced the standard library class template `std::tuple`, declared in a manner similar to `Cpp11Tuple`.

An `std::tuple` can be used, for example, to return multiple values from a function. Suppose we want to define a function, `minAverageMax`, that — given a **range** of double values — returns, along with its cardinality, its minimum, average, and maximum values. The interface for such a function in C++03 might have involved multiple output parameters passed, e.g., by non`const` *lvalue* reference:

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

template <typename Iterator>
void minAverageMax(std::size_t& numValues,   // (out only) number of inputs
                   double& minimum,          // (out only) minimum value
                   double& average,          // (out only) average value
                   double& maximum,          // (out only) maximum value
                   Iterator b, Iterator e);  // input range
    // Load into the specified numValues, minimum, average, and maximum
    // the corresponding values extracted from the specified range [b, e).
```

Alternatively, one could have define a separate `struct` (e.g., `MinAverageMaxRes`) and incorporate that into the interface of the `minAverageMax` function:

[2] https://github.com/boostorg/tuple/blob/develop/include/boost/tuple/tuple.hpp

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

struct MinAverageMaxRes  // used in conjunction with minAverageMax (below)
{
    std::size_t count;  // number of input values
    double min;         // minimum value
    double average;     // average value
    double max;         // maximum value
};

template <typename Iterator>
MinAverageMaxRes minAverageMax(Iterator b, Iterator e);
```

Adding a helper **aggregate** such as `MinAverageMax` (above) works but demands a fair amount of boilerplate coding that might not be reusable or otherwise worth naming. The C++11 library abstraction `std::tuple` allows code to define such simple aggregates "on the fly" (as needed):

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

typedef std::tuple<std::size_t, double, double, double> MinAverageMaxRes;
    // type alias for a standard tuple of four specific scalar values.

template <typename Iterator>
MinAverageMaxRes minAverageMax(Iterator b, Iterator e);
    // Return the cardinality, min, average, and max of the range [b, e].
```

We can now use our `minAverageMax` function to extract the relevant fields from a vector of `double` values:

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

void test(const std::vector<double>& v)
{
    MinAverageMaxRes res = minAverageMax(v.begin(), v.end());  // calculate
    std::size_t num = std::get<0>(res);  // fetch slot 0, the number of values
    double      min = std::get<1>(res);  // fetch slot 1, the minimum value
    double      ave = std::get<2>(res);  // fetch slot 2, the average value
    double      max = std::get<3>(res);  // fetch slot 3, the maximum value
    // ...
    std::get<2>(res) = 0.0;              // store 0 in slot 2 (just FYI)
    // ...
}
```

Note that elements in a tuple are accessed in a numerically indexed manner (beginning with slot 0) using the standard function template `std::get` for both reading and writing.

There are other motivating uses of variadic templates, such as allowing generic code to forward arguments to other functions, notably constructors, without the need to know in advance the number of arguments required. Related artifacts added to the C++ standard library include `std::make_shared`, `std::make_unique`, and the `emplace_back` member

function of `std::vector` (see *Use Cases — Object factories* on page 568 **[AUs: There is no subsection called "Factory functions." Did you mean the Object factories section? Response: Yes, it should be object factories]**).

There is a lot to unpack here. We will begin our journey by understanding variadic *class* templates as **generic types** that provide a solid basis for understanding variadic *function* templates. In practice, however, variadic *function* templates are arguably more frequently applicable outside of advanced metaprogramming; see *Variadic function templates* on page 533.

iadic-class-templates

Variadic class templates

Suppose we want to create a class template `C` that can take zero or more template type arguments. C++11 introduces new syntax — based on the ellipsis (`...`) token — that supports such variadic parameter lists:

```
template <typename... Ts> class C;
    // The class template C can be instantiated with a sequence of zero or
    // more template arguments of arbitrary type. Because this is only a
    // declaration, the parameter name, Ts (for "types"), isn't used.
```

The declaration above introduces a class template `C` that can accept an arbitrary sequence of type parameters optionally represented here (for documentation purposes) by the identifier `Ts` (for "Types").

First we point out that the ellipsis (`...`), just like `++` or `==`, is parsed as a separate token; hence, any whitespace around the ellipsis is optional. The common style, and the one used in the C++ Standard documents, follows the typographic convention in written prose: `...` abuts to the left and is followed by a single space as in the declaration of `C` above.

Whether we use **`typename`** (as we do throughout this book) or **`class`** to introduce a **template type parameter** is entirely a matter of style:

```
template <typename... Ts> class C;  // style used throughout this book
template <class... Ts> class C;     // style used in the C++ Standard
```

Note that, as with non-variadic template parameters, providing a name to represent the supplied template "parameters" (a.k.a. **template parameter pack**, see below) is optional:

```
template <typename... Ts> class C;  // with    name identifying parameter pack
template <typename...> class C;     // without name identifying parameter pack
```

When an ellipsis token (`...`) appears *after* **`typename`** or **`class`** and before the optional type-parameter name (e.g., `Ts`), it introduces a template parameter pack:

```
template <typename... Ts>  // Ts names a template parameter pack.
class C;
```

We call entities such as `Ts` template parameter packs (as opposed to parameter packs) to distinguish them from **function parameter packs** (see *Variadic function templates* on page 533), **non-type parameter packs** (see *Non-type template parameter packs* on page 546 **[AUs: There is no section called "Non-type parameter packs" Response: non-type template parameter packs is the correct intraref]**), and **template template parameter packs** (see *Template parameter packs* on page 547 The phrase **parameter**

pack is a conflation of the four and also a casual reference to either of them whenever there is no ambiguity.

This syntactic form of `...` is used primarily in **declarations** (including those associated with **definitions**):

```
template <typename... Ts>
class C { /*...*/ };  // definition of class C
```

The same `...` token, when it appears to the *right* of an existing template parameter pack (e.g., `Ts`), is used to *unpack* it:

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

template <typename... Ts> class C  // definition of class C
{
    std::vector<Ts...> d_data;
        // using ... to unpack a template parameter pack
};
```

In the example above, the *unpacking* results in a comma-separated list of the types with which `C` was instantiated. The `...` token is used after the template parameter pack name `Ts` to recreate the sequence of arguments originally passed to the instantiation of `C`. Referring to our example above, we might choose to create an object, `x`, of type `C<int>`:

```
C<int> x;  // has data member, d_data, of type std::vector<int>
```

We might, instead, consider creating an object, `y`, that also passes `std::allocator<char>` to `C`'s instantiation:

```
C<char, std::allocator<char>> y;
    // y.d_data has type std::vector<char, std::allocator<char>>.
```

And so on. That is, `class C` can be instantiated with any sequence of types that is supported by the `d_data` member variable.

An instantiation such as `C<float, double>` would correspondingly attempt to instantiate `std::vector<float, double>`, which is in error and would cause the instantiation of `C` to fail. Several other cases and patterns of *unpacking* of template parameter packs exist and are described in detail in *Template parameter packs* on page 547.

Continuing our pedagogical discussion, let's **define** an empty variadic class template, `D`:

```
template <typename...> class D { };  // empty variadic-class-template definition
```

We can now create explicit instantiations of class template `D` by providing it with any number of *type* arguments:

```
D<>                d0;  // instantiation of D with no type arguments
D<int>             d1;  // instantiation of D with a single int argument
D<int, int>        d2;  // instantiation of D with two int arguments
D<int, const int>  d3;  // Note that d3 is a distinct type from d2.
D<double, char>    d4;  // instantiation of D with a double and char
D<char, double>    d5;  // Note that d5 is a distinct type from d4.
D<D<>, D<int>>    d6;   // instantiation of D with two UDT arguments
```

The number and order of arguments are part of the instantiated type, so each of the objects `d0` through `d6` above has a distinct C++ type:

```
void f(const D<double, char>&);  // (1) overload of function f
void f(const D<char, double>&);  // (2) overload of function f

void test()
{
    f(d4);                       // invokes overload (1)
    f(d5);                       // invokes overload (2)
}
```

The sections that follow examine in full detail how to

- declare variadic *class* and *function* templates using parameter packs
- make use of variadic argument lists in the implementation of class definitions and function bodies

plate-parameter-packs

Template parameter packs

A template parameter pack is the name representing a list of zero or more parameters following **`class...`** or **`typename...`** within a variadic template declaration.

Let's take a closer look at just the *declaration* of a variadic class template, `C`:

```
template <typename... Ts> class C { };
```

Here, the identifier `Ts` names a template parameter pack, which — as we saw previously — can bind to any sequence of explicitly supplied types including the empty sequence:

```
C<> c0;             // OK, instantiation of C with no type arguments
C<int> c1;          // OK, instantiation of C with a single int argument
C<float, bool> c2;  // OK, instantiation of C with two type arguments
```

Passing an argument to `C` that is not a type, however, is not permitted:

```
C<128> cx0;           // Error, expecting type template argument, 128 provided
C<std::vector> cx1;   // Error, expecting type argument, template name provided
C<int, 42, int> cx2;  // Error, expecting type argument in second position
```

Template parameter packs can appear together with simple template parameters, with one restriction: **Primary class template declarations** allow at most one variadic parameter pack at the end of the template parameter list. (Recall that in standard C++ terminology, the **primary declaration** of a class template is the first declaration introducing the template's name; all specializations and partial specializations of a class template require the presence of a primary declaration.)

```
template <typename... Ts>
class C0  { };  // OK

template <typename T, typename... Ts>
class C1  { };  // OK
```

```
template <typename T, typename U, typename... Ts>
class C2  { };  // OK

template <typename... Ts, typename... Us>
class Cx0 { };  // Error, more than one parameter pack

template <typename... Ts, typename T>
class Cx1 { };  // Error, parameter pack must be the last template parameter
```

There is no way to specify a default for a parameter pack; however a parameter pack can follow a defaulted parameter:

```
template <typename... Ts = int>
class Cx2 { };  // Error, a parameter pack cannot have a default.

template <typename T = int, typename... Ts = char>
class Cx3 { };  // Error, a parameter pack cannot have a default.

template <typename T = int, typename... Ts>
class C3  { };  // OK

C3<> c31;                   // OK, T=int, Ts=<>
C3<char> c32;               // OK, T=char, Ts=<>
C3<char, double, int> c33;  // OK, T=char, Ts=<double, int>
```

What can we do with a template parameter pack? Template parameter packs are of a so-called *kind* distinct from other C++ entities. They are not types, values, or anything else found in C++03. As such, parameter packs are not subject to any of the usual operations one might expect:

```
template <typename... Ts>
class C
{
    C<Ts>* next;        // Error, cannot use unexpanded parameter pack Ts
    Ts memberVariable;  // Error, cannot use unexpanded parameter pack Ts
    typedef Ts Ts1;     // Error, cannot use unexpanded parameter pack Ts
    using Ts2 = Ts;     // Error, cannot use unexpanded parameter pack Ts
};
```

There is no way to use a parameter pack in unexpanded form. Once introduced, the name of a parameter pack can occur only as part of a **pack expansion**. We've already encountered the simple pack expansion `Ts...`, which expands to the list of types to which `Ts` is bound. That expansion is only allowed in certain contexts where multiple values would be allowed:

```
template <typename... Ts>
class C
{
    C<Ts...>* d_next;        // OK, instantiate C in definition of member
    C<Ts...> f0();           // OK, instantiate C in member function signature
    void f1(C<Ts...>&);      // OK, instantiate C in member function signature
    Ts... d_memberVariable;  // Error, expansion as a member type
```

```
    typedef Ts... Ts1;       // Error, expansion as a typedef type
    using Ts2 = Ts...;       // Error, expansion as a using declaration argument
};
```

Note that the code above remains invalid even if `Ts` contains a single type, e.g., in the instantiation `C<int>`. Pack expansion is not textual and not allowed everywhere; it is allowed only in certain well-defined contexts. The first such context, as showcased with the member variable `next` in the code example above, is inside a **template argument list**. Note how the pattern may be surrounded by other parameters or expansions:

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

template <typename... Ts>
class C
{
    void arbitraryMemberFunction()      // for illustration purposes
    {
        C<Ts...>                  v0;  // OK, same type as *this
        C<int, Ts...>             v1;  // OK, expand after another argument
        C<Ts..., char>            v2;  // OK, expand before another argument
        C<char, Ts..., int>       v3;  // OK, expand in between
        C<void, Ts..., Ts...>     v4;  // OK, two expansions
        C<char, Ts..., int, Ts...> v5;  // OK, no need for them to be adjacent
        C<void, C<Ts...>, int>    v6;  // OK, expansion nested within
        C<Ts..., C<Ts...>, Ts...> v7;  // OK, mix of expansions
        std::map<Ts...>           v8;  // OK, works with non-variadic template
    }
};
```

The examples above illustrate how pack expansion is not done textually, like a macro expanded by the C preprocessor. A simple textual expansion of `C<char, Ts..., int>` would be `C<char, , int>` if `Ts` were empty (i.e., inside the instantiation `C<>`). **Parameter pack expansion** is syntactic and "knows" to eliminate any spurious commas, caused by the expansion of empty parameter packs.

Within the context of a template argument list, `Ts...` is not the only pattern that can be expanded; any template instantiation using `Ts` (e.g. `C<Ts>...`) can be expanded as a unit. The result is a list of template instantiations using each of the types in `Ts`, in turn. Note that, in the example below, the expansions `C<Ts...>` and `C<Ts>...` are both valid but produce different results:

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

template <typename... Ts> class D
{
    void memberFunction()
    {
        D<Ts...> v0;              // OK, same type as *this
        D<D<Ts...>> v1;           // OK, expand to D<D<T0, T1>, ...>
        D<D<Ts>...>  v2;          // OK, expand to D<D<T0>, D<T1>, ...>
        D<std::vector<Ts>...> v3; // OK, expand to C1<std::vector<T0>, ...>
```

```
    }
};
```

The second important **parameter pack expansion context** for type parameter packs is in a **base specifier list**. All patterns that form a valid base specifier are allowed:

```
template <typename... Ts>   // zero or more arguments
class D1 : public Ts...     // publicly inherit T0, T1, ...
{ /*...*/ };

template <typename... Ts>
class D2 : public D<Ts>...  // publicly inherit D<T0>, D<T1>, ...
{ /*...*/ };

template <typename... Ts>
class D3 : public D<Ts...>  // publicly inherit D<T0, T1, ...>
{ /*...*/ };
```

The access control specifiers — **public**, **protected**, and **private** — can be applied as usual, though, within a single expansion pattern, the access specifier must be the same for all expanded elements:

```
template <typename... Ts>
class D4 : private Ts...   // privately inherit T0, T1, ...
{ /*...*/ };

template <typename... Ts>
class D5 : public Ts..., private D<int, Ts>...
    // publicly inherit T0, T1, ...
    // and privately inherit D<int, T0>, D<int, T1>, ...
{ /*...*/ };
```

Pack expansions can be freely mixed with simple base specifiers:

```
class AClass1 { /*...*/ };  // arbitrary class definition
class AClass2 { /*...*/ };  // arbitrary class definition

template <typename... Ts>
class D6 : protected AClass1, public Ts...                  // OK
{ /*...*/ };

template <typename... Ts>
class D7 : protected AClass1, private Ts..., public AClass2  // OK
{ /*...*/ };
```

If the parameter pack being expanded (e.g., `Ts`, in the code snippet above) is empty, the expansion does not introduce any base class. Notice, again, how the expansion mechanism is semantic, not textual, e.g., in the instantiation `D7<>` the fragment **`private`** `Ts...,` disappears entirely, leaving `D7<>` with `AClass1` as a **`protected`** base and `AClass2` as a **`public`** base.

To recap, the two essential parameter expansion contexts for template parameter packs are inside a template argument list and in a base specifier list.

iadic-class-templates

Specialization of variadic class templates

Recall from prior to C++11 that after a class template is introduced by a primary class template declaration, it is possible to create **specializations** and **partial specializations** of that class template. We can declare specializations of a variadic class template by supplying zero or more arguments to its parameter pack:

```
template <typename... Ts> class C0;  // primary class template declaration

template <> class C0<>;              // specialize C0 for Ts=<>
template <> class C0<int>;           // specialize C0 for Ts=<int>
template <> class C0<int, void>;     // specialize C0 for Ts=<int, void>
```

Similar specializations can be applied to class templates that have other template parameters preceding the template parameter pack. The nonpack template parameters must be matched exactly by the arguments, followed by zero or more arguments for the parameter pack:

```
template <typename T, typename... Ts>
class C1;                            // primary class template declaration

template <> class C1<int>;           // Specialize C1 for T=int, Ts=<>.
template <> class C1<int, void>;     // Specialize for T=int, Ts=<void>.
template <> class C1<int, void, int>; // Specialize for T=int, Ts=<void, int>.
template <> class C1<>;              // Error, too few template arguments
```

Partial specializations of a class template may take multiple parameter packs because some of the types involved in the partial specialization may themselves use parameter packs. Consider, for example, a variadic class template, `Tuple`, defined as having exactly one parameter pack:

```
template <typename... Ts> class Tuple  // variadic class template
{ /*...*/ };
```

Further assume a primary declaration of a variadic class template , `C2`, also having one type parameter pack. We also introduce definitions in addition to declarations so we can instantiate `C2` later:

```
template <typename... Ts> class C2
    // (0) primary declaration of variadic class template C2
{ /*...*/ };
```

This simple setup allows a variety of partial specializations. First, we can partially specialize `C2` for exactly two `Tuple`s instantiated with the same exact types:

```
template <typename... Ts>
class C2<Tuple<Ts...>, Tuple<Ts...>>  // (1) two identical Tuples
{ /*...*/ };
```

We can also partially specialize `C2` for exactly two `Tuple`s but with potentially different type arguments:

```
template <typename... Ts, typename... Us>
class C2<Tuple<Ts...>, Tuple<Us...>>  // (2) any two Tuples
{ /*...*/ };
```

Furthermore, we can partially specialize `C2` for any `Tuple` followed by zero or more types:

```
template <typename... Ts, typename... Us>
class C2<Tuple<Ts...>, Us...>  // (3) any Tuple followed by 0 or more types
{ /*...*/ };
```

The possibilities are endless; let us show one more partial specialization of `C2` with three template parameter packs that will match two arbitrary `Tuple`s followed by zero or more arguments:

```
template <typename... Ts, typename... Us, typename... Vs>
class C2<Tuple<Ts...>, Tuple<Us...>, Vs...>
    // (4) Specialize C2 for Tuple<Ts...> in the first position,
    // Tuple<Us...> in the second position, followed by zero or more
    // arguments.
{ /*...*/ };
```

Now that we have definitions for the primary template `C2` and four partial specializations of it, let's take a look at a few variable definitions that instantiate `C2`. **Partial ordering of class template specializations**[3] will decide the best match for each instantiation and also deduce the appropriate template parameters:

```
C2<int>                           c2a;  // uses (0), Ts=<int>
C2<Tuple<int>, Tuple<int>>        c2b;  // uses (1), Ts=<int>
C2<Tuple<int, char>, Tuple<char>> c2c;  // uses (2), Ts=<int,char>, Us=<char>
C2<Tuple<int, int>, char>         c2d;  // uses (3), Ts=<int,int>, Us=<char>
C2<Tuple<int>, Tuple<char>, void> c2e;  // uses (4), Ts=<int>, Us=<char>, Vs=<void>
```

Notice how partial ordering chooses (2) instead of (4) for the definition of `c2c`, although both (2) and (4) are a match; a matching non-variadic template is always a better match than one involving deduction of a parameter pack.

Even in a partial specialization, if a template argument of the specialization is a pack expansion, it must be in the last position:

```
template <typename... Ts>
class C2<Ts..., int>;
    // Error, template argument int can't follow pack expansion Ts....

template <typename... Ts>
class C2<Tuple<Ts>..., int>;
    // Error, template argument int can't follow pack expansion Tuple<Ts>....

template <typename... Ts>
class C2<Tuple<Ts...>, int>;
    // OK, pack expansion Ts... is inside another template.
```

Parameter packs are a terse and very flexible placeholder for defining partial specialization — a "zero or more types fit here" wildcard. The primary class template does not even need to be variadic. Consider, for example, a non-variadic class template, `Map`, fashioned after the `std::map` class template:

[3] **?**, section 14.5.5.2, "Partial ordering of class template specializations," [temp.class.order], pp. 339–340.

```
template <typename Key, typename Value> class Map;  // similar to std::map
```

We would then want to partially specialize another non-variadic class template, `C3`, for all maps, regardless of keys and values:

```
template <typename T> class C3;  // (1) primary declaration; C3 not variadic

template <typename K, typename V>
class C3<Map<K, V>>;
    // (2a) Specialize C3 for all Maps, C++03 style.
```

Variadics offer a terser, more flexible alternative:

```
template <typename... Ts>
class C3<Map<Ts...>>;
    // (2b) Specialize C3 for all Maps, variadic style.
    // Note: Map works with pack expansion even though it's not variadic.
```

The most important advantage of (2b) over (2a) is flexibility. In maintenance, `Map` may acquire additional template parameters, such as a predicate and an allocator. This requires surgery on `C3`'s specialization (2a), whereas (2b) will continue to work unchanged because `Map<Ts...>` will accommodate any additional template parameters `Map` may have.

An application must use either the non-variadic (2a) or the variadic (2b) partial specialization but not both. If both are present, (2a) is always preferred because an exact match is always more specialized than one that deduces a parameter pack.

Variadic alias templates

iadic-alias-templates

Alias templates (since C++11) are a new way to associate a name with a family of types without needing to define forwarding glue code. For full details on the topic, see Section 1.1."**using** Aliases" on page 121. Here, we focus on the applicability of template parameter packs to alias templates.

Consider, for example, the `Tuple` artifact — briefly discussed in Description — that can store an arbitrary number of objects of heterogeneous types:

```
template <typename... Ts> class Tuple;  // declare Tuple
```

Suppose we want to build a simple abstraction on top of `Tuple` — a "named tuple" that has an `std::string` as its first element, followed by anything a `Tuple` may store:

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

template <typename... Ts>
using NamedTuple = Tuple<std::string, Ts...>;
    // introduce alias for Tuple of std::string and anything
```

In general, alias templates take template parameter packs following the same rules as primary class template declaration: An alias template may be defined to take at most one template parameter pack in the last position. Alias templates do not support specialization or partial specialization.

adic-function-templates

Variadic function templates

The simplest example of a variadic function template accepts a template parameter pack in its **template parameters list** but does not use any of its template parameters in its **parameter declaration**:

```
template <typename... Ts>
int f0a();     // does not use Ts in parameter list

template <typename... Ts>
int f0b(int);  // uses int but not Ts in parameter list

template <typename T, class... Ts>
int f0c();     // does not use T or Ts in parameter list
```

The only way to call the functions shown in the code snippet above is by passing them template argument lists explicitly:

```
int a1 = f0a<int, char, int>();  // Ts=<int, char, int>
int a2 = f0b<double, void>(42);  // Ts=<double, void>
int a3 = f0c<int, void, int>();  // T=int, Ts=<void, int>
int e1 = f0a();                  // Error, cannot deduce Ts
int e2 = f0b(42);                // Error, cannot deduce Ts
int e3 = f0c();                  // Error, cannot deduce T and Ts
```

The notation `f0a<int, char, int>()` means to explicitly instantiate template function `f0a` with the specified type arguments and to call that instantiation.

Invoking any of the instantiations shown above will, of course, require that the corresponding definition of the function template exists somewhere within the program:

```
template <typename...> void f0a() { /*...*/ }  // variadic template definition
```

This definition will typically be part of or reside alongside its declaration in the same header or source file, but see Section 2.1."**extern template**" on page 329.

Such functions are rarely encountered in practice; most of the time a template function would use its template parameters in the function parameter list. They have only pedagogical value; as the joke goes, the keys were lost in the living room but we're looking for them in the hallway because the light is better. To take a look in the living room, let's now consider a different kind of variadic function template — one that not only accepts an arbitrary number of template arguments, but also accepts an arbitrary number of *function* arguments working in tandem with the template arguments.

unction-parameter-packs

Function parameter packs

The syntax for declaring a variadic function template that accepts an arbitrary number of function arguments makes two distinct uses of the ellipsis (`...`) token. The first use is to introduce the template parameter pack `Ts`, as already shown. Then, to make the argument list of a function template variadic, we introduce a function parameter pack by placing the `...` to the left of the function parameter name:

```
template <typename... Ts>  // template parameter pack Ts
```

```
int f1a(Ts... values);     // function parameter pack values
   // f1a is a variadic function template taking an arbitrary sequence of
   // function arguments by value (explanation follows), each independently of
   // arbitrary heterogeneous type.
```

A function parameter pack is a function parameter that accepts zero or more function arguments. Syntactically, a function parameter pack is similar to a regular function parameter declaration, with two distinctions:

- The type in the declaration contains at least one template parameter pack
- The ellipsis `...` is inserted right before the function parameter name, if present, or replaces the parameter name, if absent

Therefore, the declaration of `f1a` above has a template parameter pack `Ts` and a function parameter pack `values`. The function parameter declaration `Ts... values` indicates that `values` may accept zero or more arguments of various types, all by value. Replacing the parameter declaration with **`const`** `Ts&... values` would result in pass by **`const`** reference. Just as with non-variadic function templates, variadic template parameter lists are permitted to include any legal combination of **qualifiers** (**`const`** and **`volatile`**) and **declarator operators** (i.e. pointer `*`, reference `&`, forwarding reference `&&`, and array `[]`):

```
template <typename... Ts> void f1b(Ts&...);
   // accepts any number and types of arguments by reference

template <typename... Ts> void f1c(const Ts&...);
   // accepts any number and types of arguments by reference to const

template <typename... Ts> void f1d(Ts* const*...);
   // accepts any number and types of arguments by pointer to const
   // pointer to nonconst object

template <typename... Ts> void f1e(Ts&&...);
   // accepts any number and types of arguments by forwarding reference

template <typename... Ts> void f1f(const volatile Ts*&...);
   // accepts any number and types of arguments by reference to pointer to
   // const volatile objects
```

To best understand the syntax of variadic template declarations, it is important to distinguish the distinct — and indeed complementary — roles of the two occurrences of the `...` token. First, as discussed in *Template parameter packs* on page 547, **`typename...`** `Ts` introduces a template parameter pack called `Ts` that matches an arbitrary sequence of types. The second use of `...` is, instead, a pack expansion that transforms the pattern — whether it's `Ts...`, **`const`** `Ts&...`, `Ts*` **`const`**`*...` etc, — into a comma-separated parameter list for the function, where the C++ type of each successive parameter is determined by the corresponding type in `Ts`. The resulting construct is a function parameter pack.

Conceptually, a single variadic template function declaration can be thought of as a multitude of similar declarations with zero, one,… parameters fashioned after the variadic declaration:

```
template <typename... Ts> void f1c(const Ts&...);
    // variadic, any number of arguments, any types, all by const &

void f1c();
    // pseudo-equivalent for variadic f1c called with 0 arguments

template <typename T0> void f1c(const T0&);
    // pseudo-equivalent for variadic f1c called with 1 argument

template <typename T0, typename T1>
void f1c(const T0&, const T1&);
    // pseudo-equivalent for variadic f1c called with 2 arguments

template <typename T0, typename T1, typename T2>
void f1c(const T0&, const T1&, const T2&);
    // pseudo-equivalent for variadic f1c called with 3 arguments

/* ... and so on ad infinitum ... */
```

A good intuitive model would be that the pack expansion in the function parameter list is like an "elastic" list of parameter declarations that expands or shrinks appropriately. Note that the types in a parameter pack may all be different from one another, but their qualifiers (**const** and/or **volatile**) and declarator operators (i.e. pointer `*`, reference `&`, *rvalue* reference `&&`, and array `[]`) are the same.

The name of the pack may be missing from the declaration, which, combined with the parameter declaration syntax coming all the way from C, does allow for some obscure constructs: [**AUs: The below example is not valid C++, though clang will accept it with a warning when not being pedantic. I suggest finding an alternative example that actually works or excising these two paragraphs as fluff.**]

```
template<typename... Ts> void fpf(Ts (*...)(int));
    // mystery declaration
```

To read such a declaration, add a name, keeping in mind that the `...` in a function parameter pack always comes immediately to the left of where the name would be:

```
template <typename... Ts> void fpf(Ts (*...pFunctions)(int));
    // Aha, the function parameter pack is pFunctions!
    // The function fpf takes zero or more pointers to functions taking one
    // int, each returning some arbitrary type.
```

A variadic function template may take additional template parameters as well as additional function parameters:

```
template <typename... Ts> void f2a(int, Ts...);
    // one int followed by zero or more arbitrary arguments by const &

template <typename T, typename... Ts> void f2b(T, const Ts&...);
    // first by value, zero or more by const &

template <typename T, typename U, typename... Ts> void f2c(T, const U&, Ts...);
```

```
    // first by value, second by const &, zero or more by value
```

There are restrictions on such declarations; see *The Rule of Greedy Matching* on page 540 and *The Rule of Fair Matching* on page 543.

Note that inside the function parameters declaration, `...` can be used only in conjunction with a template parameter pack. Attempting to use `...` where there's no parameter pack to expand could lead to inadvertent use of the old C-style variadics:

```
template <typename T, typename... Ts>
void good(T, Ts...);                    // variadic template

template <typename T, typename... Ts>
void oops(T, T...);                     // old C-style variadic
```

Such a mistake may be caused by a simple typo in the declaration of `oops`, leading to a variety of puzzling compile-time or link-time errors; see *Potential Pitfalls — Accidental use of C-style ellipsis* on page 588.

It is possible to use the parameter pack name (e.g., `Ts`) as a parameter to a user-defined type:

```
template <typename> struct S1;  // declaration only

template <typename... Ts>       // parameter pack named Ts
int fs1(S1<Ts>...);
    // **Pack expansion** for explicit instantiation of S1 accepts any number of
    // independent explicit instantiations of S1 by value.
```

The parameter pack name `Ts` acts as though it were a separate type parameter for each function argument, thereby allowing a different instantiation for `S1` at each parameter position. To invoke `fs1` on arguments of user-defined type `S1`, however, `S1` will need to be a **complete type** — i.e., its definition must precede the point of invocation of the function in the current translation unit:

```
int s1a = fs1(S1<int>());             // Error, S1 declared but not defined

template <typename T>
struct S1 { /*...*/ };                // introduce definition for S1

int s1b = fs1();                       // Ts is the empty pack.
int s1c = fs1(S1<const char*>());      // Ts=<const char*>
int s1d = fs1(S1<int>(), S1<bool>());  // Ts=<int,bool>
```

More complex setups are possible as well. For example, we can write a variadic function template that operates on instantiations of user-defined-type templates that take two independent type parameters:

```
template <typename, typename>
struct S2;                  // two-parameter class template declaration

template <typename... Ts>   // parameter pack named Ts
int fs2(S2<Ts, Ts>...);
    // The function fs2 takes by value any number of explicit instantiations
```

```
    // of S2 as long as they use the same type in both positions.
```

Calls to `fs2` work only if we supply instantiations of `S2` having the same type for both template parameters:

```
template <typename, typename> struct S2 { /*...*/ };  // S2's definition

int s2a = fs2();                                      // OK
int s2b = fs2(S2<char, char>());                      // OK
int s2c = fs2(S2<int, int>(), S2<bool, bool>());      // OK
int s2d = fs2(S2<char, int>());                       // Error
int s2e = fs2(S2<char, const char>());                // Error
```

The problem with the last two calls above is that the instantiations `S2<char, int>` and `S2<char, const char>` violate the requirement that the two types in the instantiation of `S2` are identical, so there is no way to provide or deduce some `Ts` in a way that would make the call work.

riadic-member-functions

Variadic member functions

Member functions may be variadic in two orthogonal ways: The class they are part of may be variadic, and they may be variadic themselves. The simplest case features a variadic member function of a non-template class:

```
struct S3                                  // non-variadic non-template class
{
    template <typename... Ts> int f(Ts...);  // OK
};

int s3 = S3().f(1, "abc");                 // Ts=<int, const char*>
```

Expectedly, a non-variadic **class template** may declare variadic member functions as well:

```
template <typename T>
struct S4                                          // class template
{
    template <typename... Ts> int f1(Ts...);          // OK
    template <typename... Ts> int f2(T, const Ts&...);  // OK
};

int s3b = S4<int>().f1(1, false, true);            // Ts=<int, bool, bool>
int s3c = S4<int>().f2(1, false, true);            // Ts=<bool, bool>
```

A **variadic class template** may define regular **member functions**, member functions that take their own template parameters, and **variadic member function templates**:

```
template <typename... Ts>
struct S5
{
    int f1();       // non-template member function of variadic class

    template <typename T>
```

```
    int f2(T);      // template member function of variadic class

    template <typename... Us>
    int f3(Us...);  // variadic template member function of variadic class
};

int s5a = S5<int, char>().f1();
    // Ts=<int, char>

int s5b = S5<char, int>().f2(2.2);
    // Ts=<int, char>, T=double

int s5c = S5<int, char>().f3(1, 2.2);
    // Ts=<int, char>, Us=<int, double>
```

Although in a sense all member functions of a variadic class template (e.g., `S5`, above) are variadic, the class's template parameter pack (e.g., `Ts`, above) is fixed at the time the class is instantiated. The only truly variadic function is `f3` because it takes its own template parameter pack, `Us`.

A member function of a variadic class may use the template parameter pack of the class in which it is defined:

```
template <typename... Ts>
struct S6                   // variadic class template
{
    int f1(const Ts&...);  // OK, not truly variadic; Ts is fixed

    template <typename T>
    int f2(T, Ts...);      // OK, also not truly variadic; Ts is fixed

    template <typename... Us>
    int f3(Ts..., Us...);  // OK, variadic template member function
};

int s6a = S6<int, char>().f1(1, 'a');
    // Ts=<int, char>

int s6b = S6<char, int>().f2(true, 'b', 2);
    // Ts=<int, char>, T=bool

int s6c = S6<int, char>().f3(1, 2.2, "asd", 123.456);
    // Ts=<int, char>, Us=<const char*, int, double>
```

Notice how, in the initialization of `s6c`, the type arguments `Ts` of class template `S6` must be chosen explicitly, whereas the variadic template `f3` need not have `Us` specified because they are deduced from the argument types. This takes us to the important topic of **template argument deduction**.

late-argument-deduction

Template argument deduction

A popular feature of C++ ever since templates were incorporated into the language pre-standardization has been the ability of function templates to determine their template arguments from the types of arguments provided. Consider, for example, designing and using a function, `print`, that outputs its argument to the console. In most cases, just letting `print` deduce its template argument from the function argument's type has brevity, correctness, and efficiency advantages:

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

template <typename T> void print(const T& value);  // prints x to stdout

void testPrint0(const std::string& s)
{
    print<const char*>("Hi");
        // verbose:   specifies template argument *and* function argument
        // redundant: function argument's type *is* the template argument

    print<int>(3.14);
        // Error-prone: Mismatch may cause incorrect results (prints 3).

    print<std::string>("Oops");
        // inefficient: may incur additional expensive implicit conversions

    print("Hi");
    print(3.14);
    print(s);
        // all good: let print deduce template argument from function argument
}
```

What makes the shorter form of the call work is the magic of C++ template argument deduction, which, unsurprisingly, works with variadic template parameters as well. A detailed description of all rules for template argument deduction, including those inherited from C++03, is outside the scope of this book.[4] Here, we focus on the additions to the rules brought about by variadic function templates.

With that in mind, imagine we set out to redesign the API of `print` (in the code example above) to output any number of arguments to the console:

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

template <typename... Ts>               // template parameter pack Ts
void print(const Ts&... values);        // prints each of values to stdout

void testPrint1(const std::string& s)  // arbitrary function
{
```

[4] Please insert a cite to Scott Meyers' "Effective Modern C++" (which is already in your bib file), with the addition: Chapter 1: "Deducing Types", pages 9–35. [**AUs: We have several editions of the Meyers book in our bib; which one did you intend? Response: meyers15b and meyers15 are the same book, which this should be referring to.**]

```
    print<const char*, int, std::string>("Hi", 3.14, "Oops");
        // verbose:     specifies template arguments *and* function arguments
        // Redundant:   Function arguments' types *are* the template arguments.
        // Error-prone: Mismatch may cause incorrect results (prints 3).
        // inefficient: may incur additional expensive implicit conversions

    print("Hi", 3.14, s);
        // All good: Deduce template arguments from function arguments.
}
```

The compiler will independently deduce each type in `Ts` from the respective types of the function's argument `values`:

```
void testPrint2()  // arbitrary function
{
    print();               // OK, Ts=<>
    print(42, true);       // OK, Ts=<int, bool>
    print(42.2, "hi", 5);  // OK, Ts=<double, const char*, int>
}
```

As shown in the first line of `test2` (above), a template parameter pack may be specified in its entirety when instantiating a function template:

```
void testPrint3()  // arbitrary function
{
    print<>();                    // OK, exact match
    print<int, bool>(42, true);   // OK, exact match
    print<int, bool>('a', 'b');   // OK, arguments convertible to parameters
}
```

Interestingly, we can specify only the first few types of the template parameter pack and let the others be deduced, mixing together **explicit template argument specification** and template argument deduction:

```
void testPrint4()  // arbitrary function
{
    print<>(5, 'a');                // OK, Ts=<int, char>
    print<unsigned int>(42, true);  // OK, Ts=<unsigned int, bool>
    print<int, int>('a', "ab", 1);  // Error, cannot deduce Ts
}
```

Such a mix of explicit and implicit may be interesting but is not new; it has been the case for function templates since C++ was first standardized. The new element here is that we get to specify a fragment of a template parameter pack.

In the general case, a function may mix template parameter packs with other template parameters in a variety of ways. It may also mix function parameter packs with other function parameters in a variety of ways.

The Rule of Greedy Matching

`le-of-greedy-matching`

The Rule of Greedy Matching for template parameter packs (but not function parameter packs) states that once a template parameter pack starts matching one explicitly specified

template argument, it also matches all template arguments following it. There's no way to tell a template parameter pack it matched enough. Template parameter packs are *greedy*.

Consequently, there is no syntactic way to explicitly specify any template argument following one that matches a template parameter pack; the first pack gobbles all remaining arguments up. Put succinctly: all **template parameters** following a template parameter pack can *never* be explicitly specified as template arguments.

Notice that the rule applies only once a pack starts matching, i.e., it has matched at least one item; indeed there are a few legitimate cases in which a parameter list makes no match at all, which we'll discuss soon.

Using the Rule of Greedy Matching allows us to navigate with relative ease a variety of combinations of pack and nonpack template parameters.

In the simplest and overwhelmingly most frequently encountered situation, the function takes one template parameter pack and one function parameter pack, both in the last position:

```
template <typename T1, typename T2, typename... Ts>  // ... in last position
int f1(T1, T2, const Ts&...);                        // ... in last position
```

In such cases, the Rule of Greedy Matching doesn't need to kick in because there are no parameters following a pack. Template argument deduction can be used for all of `T1`, `T2`, and `Ts` or for only for a subset. In the simplest case, no template parameters are specified and template argument deduction is used for all:

```
int x1a = f1(42, 2.2, 'a', true);
    // T1=int, T2=double, Ts=<char, bool> (all deduced)
```

Explicitly specified template arguments, if any, will match template parameters in the order in which they are declared. Therefore, if we specify one type, it binds to `T1`:

```
int x1b = f1<double>(42, 2.2, 'a', true);
    // T1=double (explicitly), T2=double (deduced), Ts=<char, bool> (deduced)
```

If we specify two types, they will bind to `T1` and `T2` in that order:

```
int x1c = f1<double, char>(42, 65, "abc", true);
    // T1=double (explicitly), T2=char (explicitly),
    // Ts=<const char*, bool> (deduced)
```

Note how, in the two examples above, we also have implicit conversions going on, i.e., `42` is converted to **`double`** and `65` is converted to **`char`**. Furthermore, a call may specify `T1`, `T2`, and the entirety of `Ts`:

```
int x1d = f1<const char*, char, bool, double>("abc", 'a', true, 42U);
    // T1=const char* (explicitly), T2=char (explicitly),
    // Ts=<bool, unsigned> (explicitly)
```

Last but not least, as mentioned, a call may specify `T1`, `T2`, and only the first few types in `Ts`:

```
int x1e = f1<const char*, char, bool, double>("abc", 'a', true, 42, 'a', 0);
    // T1=const char* (explicitly), T2=char (explicitly),
    // Ts=<bool, int, char, int> (first two explicit, others deduced)
```

Let us now look at a function having a template parameter pack not in the last position. However, in the argument list, the function parameter pack is still in the last position:

```
template <typename T, typename... Ts, typename U>  // ... not last
int f2(T, U, const Ts&...);
```

In such cases, by the Rule of Greedy Matching, there is no way to specify `U` explicitly, so the only way to call `f1` is to let `U` be deduced:

```
int x2a = f2(1, 2);
    // T=int (deduced), U=int (deduced), Ts=<> (by arity)

int x2b = f2<long>(1, 2, 3);
    // T=long (explicit), U=int (deduced), Ts=<int> (deduced)
```

The first template argument passed to `f2`, if any, is matched by `T`. Note that inferring the empty template parameter pack in the initialization of `x2a` does not involve deduction; the empty length is inferred by the arity of the call. In contrast, in the initialization of `x2b`, a pack of length 1 is deduced.

Any subsequent template arguments will be matched *en masse* by `Ts` in concordance with the Rule of Greedy Matching:

```
int x2c = f2<long, double, char>(1, 2, 3, 4);
    // T1=long (explicit), T2=int (deduced), Ts=<double, char> (explicit)

int x2d = f2<int, double>(1, 2, 3.0, "four");
    // T1=int (explicit), T2=int (deduced), Ts=<double, const char*> (partially specified)

int x2e = f2<int, char, double>(1, 3.0, 'a');
    // Error, no viable function for T1=int and Ts=<char, double>
```

In all cases, `T2` must be deduced for the call to `f2` to go through.

Another way to make a template parameter work even if it's positioned after a parameter pack is by assigning it a default argument:

```
template <typename... Ts, typename T = int>
T f3(Ts... values);
```

Due to the way `f3` is defined, there is no way to either deduce or specify `T`, so it will always be **`int`**:

```
int x3a = f3("one", 2);
    // Ts=<const char*, int> (deduced), T=int (default)

int x3b = f3<const char*>("one", 2);
    // Ts=<const char*, int> (partially deduced), T=int (default)

int x3c = f3<const char*, int>("one", 2);
    // Ts=<const char*, int> (explicit), T=int (default)
```

e-rule-of-fair-matching

The Rule of Fair Matching

To further explore the varied ways in which parameter packs may interact with the rest of C++, let's take a look at a function that has a type following a template parameter pack and also a value following a function parameter pack:

```
template <typename... Ts, typename T>
int f4(Ts... values, T value);
```

Here, a new rule applies, the Rule of Fair Matching, which is to a good extent the converse of the Rule of Greedy Matching. When a function parameter pack (e.g., `values` in the code snippet above) is *not* at the end of a function's parameter list, its corresponding type parameter pack (e.g., `Ts` above) *cannot be deduced* ever.

This rule makes the function parameter pack `values` fair because function parameters following a function parameter pack have a chance to match function arguments.

Let's take a look at how the rule applies to `f4`. In calls with one argument, `Ts` is not deduced, so it forcibly matches the empty list, and `T` matches the type of the argument:

```
int x4a = f4(123);     // Ts=<> (forced non-deduced), T=int
int x4b = f4('a');     // Ts=<> (forced non-deduced), T=char
int x4c = f4('a', 2);  // Error, cannot deduce Ts
```

Calls with more than one argument can be made if and only if we provide `Ts` to the compiler explicitly:

```
int x4d = f4<int, char>(1, '2', "three");
    // Ts=<int, char> (explicit), T=const char*
```

Incidentally, for `f4` there is no way to specify `T` explicitly because of the Rule of Greedy Matching:

```
int x4e = f4<int, char, const char*>(1, '2', "three");
    // Error, Ts=<int, char, const char*> (explicit), no argument for T value
```

Note that the two rules may work simultaneously on the same function call; they do not compete because they apply in different places; the Rule of Greedy Matching applies to template parameter packs, and the Rule of Fair Matching applies to function parameter packs.

Let's now consider declaring a function with two consecutive parameter packs and see how the rules work together in calls:

```
template <typename... Ts, typename... Us>  // two template parameter packs
int f5(Ts... ts, Us... us);                // two function parameter packs
```

By the Rule of Greedy Matching, we cannot specify `Us` explicitly so `us` will rely on deduction exclusively. By the Rule of Fair Matching, `Ts` cannot be deduced. First, let's analyze a call with no template arguments:

```
int x5a = f5(1);
    // Ts=<> (forcibly), Us=<int>

int x5b = f5(1, '2');
    // Ts=<> (forcibly), Us=<int, char>
```

```
int x5c = f5(1, '2', "three");
    // Ts=<> (forcibly), Us=<int, char, const char*>
```

Whenever a call specifies no **template arguments**, `Ts` cannot be deduced so it can at best match the empty list. That leaves all of the arguments to `us`, and deduction will work as expected for `Us`. This right-to-left matching may surprise at first and requires close reading of different parts of the C++ Standard but is easy to explain by using the two rules.

Let's now issue a call that does specify template arguments:

```
int x5d = f5<int, char>(1, '2');
    // Ts=<int, char> (explicitly), Us=<const char*>

int x5e = f5<int, char>(1, '2', "three");
    // Ts=<int, char> (explicitly), Us=<const char*>

int x5f = f5<int, char>(1, '2', "three", 4.0);
    // Ts=<int, char> (explicitly), Us=<const char*, double>
```

By the Rule of Greedy Matching, all explicit template arguments go to `Ts`, and, by the Rule of Fair Matching, there's no deduction for `Ts`, so, even before looking at the function arguments, we know that `Ts` is exactly `<int, char>`. From here on, it's easy: The first two arguments go to `ts`, and all others, if any, will go to `us`.

Corner cases of function template argument matching

ate-argument-matching

There are cases in which a template function could be written that can never be called, whether with explicit template parameters or by relying on template argument deduction:

```
template <typename... Ts, typename T>
void odd1(Ts... values);
```

By the Rule of Greedy Matching applied to `Ts`, `T` can never be specified explicitly. Moreover, `T` cannot be deduced either because it's not part of the function's parameter list. Hence, `odd1` is impossible to call. According to standard C++, such function declarations are **ill formed, no diagnostic required (IFNDR)**. Current compilers do allow such functions to be defined without a warning. However, any conforming compiler will disallow any attempt to call such an ill-fated function.

Another scenario is one whereby a variadic function can be instantiated, but one or more of its parameter packs must always have length zero:

```
template <typename... Ts, typename... Us, class T>
int odd2(Ts..., Us..., T);  // specious
```

Any attempt to call `odd2` by relying on only template argument deduction will force both `Ts` and `Us` to the empty list because, by the Rule of Fair Matching, neither `Ts` nor `Us` can benefit from template argument deduction. So calls with two, three, or more arguments fail:

```
int x2a = odd2(1, 2.5);           // Error, Ts=<>, Us=<>, too many arguments
int x2a = odd2(1, 2.5, "three");  // Error, Ts=<>, Us=<>, too many arguments
```

However, there seem to be ways to invoke `odd2`, at least on contemporary compilers. First, calls using deduction with exactly one argument will merrily go:

```
int x2c = odd2(42);  // Ts=<>, Us=<>, T=42
```

Better yet, functions that pass an explicit argument list for `Ts` also seem to work:

```
int x2d = odd2<int, double>(1, 2.0, "three");
    // Ts=<int, double>, Us=<>, T=const char*
```

The call above passes `Ts` explicitly as `<int, double>`. Then, as always, `Us` is forced to the empty list, and `T` is deduced as `const char*` for the last argument. That way, the call goes through!

Or does it? Alas, the declaration of `odd2` is **IFNDR**. By the C++ Standard, if all valid instantiations of a variadic function template require a specific template parameter pack argument to be empty, the declaration is IFNDR. Although such a rule sounds arbitrary, it does have a good motivation: If all possible calls to `odd2` require `Us` to be the empty list, why is `Us` present in the first place? Such code is more indicative of a bug than of a meaningful intent. Also, diagnosing such cases may be quite difficult in the general case, so no diagnostic is required. As it turns out, today's compilers do not issue such a diagnostic, so the onus is on the template writer to make sure the code does what it's supposed to do.

A simple fix for `odd2` is to just eliminate the `Us` template parameter, in which case `odd2` has the same signature as `f4` discussed in *The Rule of Fair Matching* on page 543. Another possibility to "legalize" `odd2` is to drop the nonpack parameter, in which case it has the same signature as `f5` in the same section.

A function that has three parameter packs in a row would also be IFNDR:

```
template <typename... Ts, typename... Us, typename... Vs>
void odd3(Ts..., Us..., Vs...);  // impossible to instantiate
```

The reason `odd3` cannot work is purely bureaucratic: Neither `Ts` nor `Us` may benefit from deduction, and then there is no way to specify `Us` explicitly because `Ts` is greedy. Consequently, `Us` is forced to be always of length zero.

It may seem there is no way to define a function template taking more than two parameter packs. However, recall that deducing variadic function template parameters from the (object) arguments passed to the function uses the full power of C++'s template argument deduction. Defining functions with any number of template parameter packs is entirely possible provided the parameter packs are themselves part of other template instantiations:

```
template <typename...> class Vct { };  // variadic class template definition

template <typename T, typename... Ts, typename... Us, typename... Vs>
int fvct(const T&, Vct<Ts...>, Vct<Us...>, Vct<Vs...>);
    // The first parameter matches any type by const&, followed by three
    // not necessarily related instantiations of Vct.
```

The function template `fvct` takes a fixed number of parameters (four), the last three of which are independent instantiations of a variadic class template `Vct`. For each of them, `fvct` takes a template parameter pack that it passes along to `Vct`. For each call to `fvct`, template argument deduction figures out whether the call is viable and also binds `Ts`, `Us`, and `Vs` to the packs that make the call work:

```
int x = fvct(5, Vct<>(), Vct<char, int, long>(), Vct<bool>());
    // OK, T=int, Ts=<>, Us=<char, int, long>, Vs=<bool>
```

For each argument in the call above, the compiler matches the type of the argument with the pattern required by the template parameter; the matching process deduces the types that would make the match work. The general algorithm for matching a concrete type against a type pattern is called **unification**.[5]

plate-parameter-packs

Non-type template parameter packs

Defining a variadic template that take arguments other than types is also possible. Just as C++03 template parameters can be *types*, *values*, or *templates*, so can template parameter packs. We used **type template parameter packs** up until now to simplify exposition, but non-type template parameter packs apply to class templates and function templates as well.

To clarify terminology, the C++ Standard refers to template parameters that accept values as **non-type template parameters** and to template parameters that accept names of templates as **template template parameters** (to be discussed in subsection *Template parameter packs* on page 547).

Non-type template parameter packs are defined analogously with non-type template parameters:

```
template <int...>
class Ci { };           // variadic int-parameter class template

Ci<>              ci0;  // OK, zero ints given
Ci<1>             ci1;  // OK, one int argument
Ci<2, 3>          ci2;  // OK, two int arguments
Ci<true, 'a', 3u> ci3;  // OK, converts to three int arguments
Ci<4.0>           ci4;  // Error, floating-point literal 4.0 is ineligible.
```

The type specifier of the template non-type parameter pack need not only be **int**; the same rules as for **non-type parameters** apply, restricting the types of non-type parameters to

- integral
- enumerated
- pointer to function or object
- *lvalue* reference to function or object
- pointer to member
- `std::nullptr_t`

These types are the value types allowed for C++03 non-type template parameters. In short, value parameter packs obey the same restrictions as the C++03 non-type template parameters:

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

[5] ?

```
enum Enum { /*...*/ };                  // arbitrary enumerated type
class AClass { /*...*/ };               // arbitrary class type

template <long... ls>           class Cl;   // OK, integral
template <bool... bs>           class Cb;   // OK, integral
template <char... cs>           class Cc;   // OK, integral
template <Enum... es>           class Ce;   // OK, enumerated
template <int&... is>           class Cri;  // OK, reference
template <std::string*... ss>   class Csp;  // OK, pointer
template <void (*... fs)(int)>  class Cf;   // OK, pointer to function
template <int AClass::*... >    class Cpm;  // OK, pointer to member
template <std::nullptr_t>      class Cnl;  // OK, std::nullptr_t

template <Ci<>... cis>          class Cu;   // Error, cannot be user-defined
template <double... ds>         class Cd;   // Error, cannot be floating-point
template <float... fs>          class Cf;   // Error, cannot be floating-point
```

In the example above, the declaration of class template `Cu` is not permitted because `Ci<>` is a *user-defined* type, whereas the declarations of both `Cd` and `Cf` are disallowed because **`double`** and **`float`** are floating-point types.[6]

Template parameter packs

emplate-parameter-packs

Template parameter packs are the variadic generalization of C++03's template template parameters. Classes and functions may be designed to take template parameter packs as parameters in addition to **type parameter packs**.

A template *template* parameter is a template parameter that names an argument that is itself a *template*. Consider, for example, two arbitrary class templates, `A1` and `A2`:

```
template <typename> class A1 {/* ...*/ };  // some arbitrary class template
template <typename> class A2 {/* ...*/ };  // another arbitrary class template
```

Now suppose that we have a *class* template, (e.g., `C1`) that takes, as its template parameter, a *class template*:

```
template <template<typename> class X>  // X is a template template parameter.
struct C1 : X<int>, X<double>          // inherit X<int> and X<double>
{ };
```

We can now create instances of class `C1` where the bases are obtained by instantiating whatever argument we pass to `C1` with **`int`** and **`double`**, respectively:

```
C1<A1> c1a;  // inherits A1<int> and A1<double>
C1<A2> c1b;  // inherits A2<int> and A2<double>
```

In the code snippet above, `X` is a template *template* parameter that takes one *type* parameter. The template classes `A1` and `A2` match `X` because each of these templates, in turn, takes one *type* parameter as well.

[6] C++20 does allow *floating point* non-type template parameters, which enable the definition and use of non-type parameter packs using **`float`**, **`double`**, or **`long double`** (making, e.g., the final entry — `Ci<4.0> ci4;` — in the previous example above valid as well).

If an instantiation is attempted with a class template that does not take the same number of template parameters, the instantiation fails, even when there is no doubt as to what the intent might be:

```
template <typename, typename = char>
class A3 { /*...*/ };  // OK, two-parameter template, with second one defaulted

C1<A3> c1c;  // Error, parameters of A3 are different from parameters of X.
```

Although `A3` could be instantiated with a single template argument (due to its second template parameter having a default argument) and `A3<int>` is valid, `C1<A3>` will not compile. The compiler complains that `A3` has two parameters, whereas `X` has a single parameter.

The same limitation is at work when the argument to `C1` is a variadic template:

```
template <typename...>
class A4 { /*...*/ };    // OK, arbitrary variadic template

C1<A4> c1d;  // Error, parameters of A4 are different from parameters of X.
```

Let's define a class `C2` that is a variadic generalization of `C1` in such a way that it allows instantiation with `A3` and `A4`:

```
template <template<typename...> class X>  // template template parameter
struct C2 : X<int>, X<double>             // inherit X<int>, X<double>
{ };
```

Note that, although one may use **`typename`** and **`class`** interchangeably inside the template parameters of `X` above, one must always must use **`class`** for `X` itself.[7]

The difference between `C2` and `C1` is literally one token: `C2` adds one `...` after **`typename`** *inside* the parameter list of `X`. That one token makes all the difference: by using it, `C2` signals to the compiler that it accepts templates with any number of parameters, be they fixed in number, defaulted, or variadic. In particular, it works just fine with `A1` and `A2` as well as `A3` and `A4`:

```
C2<A1> c2a;
    // inherits A1<int> and A1<double> in that order
C2<A2> c2b;
    // inherits A2<int> and A2<double> in that order
C2<A3> c2c;
    // C2<A3> inherits A3<int, char> and A3<double, char> in that order.
    // char is the default argument for A3's second type parameter.
C2<A4> c2d;
    // inherits A4<int> and A4<double> in that order
```

When `C3` instantiates `A3<int>` or `A3<double>`, `A3`'s default argument for its second parameter kicks in. `A4` will be instantiated with the type parameter pack `<int>` and separately with the type parameter pack `<double>`.

In contrast with template template parameters that require exact matching with their arguments, variadic template template parameters specifying **`typename...`** work in a more

[7] Since C++17, code may use either **`typename`** or **`class`** for template template parameters and for template parameter packs.

"do what I mean" manner by matching templates with default arguments and variadic templates. Applications may need either style of matching depending on context.

There is an orthogonal other direction in which we can generalize `C2`: We can define a template, `C3`, that accepts zero or more template template arguments:

```
template <template<typename> class... Xs>  // Xs is a template parameter pack.
struct C3
    : Xs<int>...     // inherit X0<int>, X1<int>, ...
    , Xs<double>...  // inherit X0<double>, X1<double>, ...
{ };
```

We get to instantiate `C3` with zero or more template classes, each of which takes exactly one type parameter:

```
C3<>           c3a;
    // no base classes at all; Xs=<>, all base specifiers vanish

C3<A1>         c3b;
    // inherits A1<int> and A1<double> in that order

C3<A1, A2>     c3c;
    // inherits A1<int>, A2<int>, A1<double>, and A2<double> in that order

C3<A2, A1>     c3d;
    // inherits A2<int>, A1<int>, A2<double>, and A1<double> in that order

C3<A1, A2, A1> c3e;
    // Error, cannot inherit A1<int> and A1<double> twice
```

Note that `C3`, just like `C1`, cannot be instantiated with `A3`, again due to mismatched template parameters:

```
C3<A3> c3f;
    // Error, parameters of A3 are different from parameters of Xs.
C3<A4> c3g;
    // Error, parameters of A4 are different from parameters of Xs.
```

If we want to make instantiations with `A3` and `A4` possible, we can combine the two generalization directions by placing the `...` inside the `Xs` parameter declaration *and* on `Xs` as well:

```
template <template<typename...> class... Xs>  // two sets of ...
struct C4                                     // most flexible
    : Xs<int>...                              // X0<int>, X0<int>, ...
    , Xs<double>...                           // X1<double>, X1<double>, ...
{ };
```

`C4` combines the characteristics of `C2` and `C3`. It accepts zero or more arguments, which in turn are accepted in a "do what I mean" manner:

```
C4<>           c4a;
    // no base classes at all; Xs=<>, all base specifiers vanish
```

```
C4<A1>         c4b;
    // inherits A1<int> and A1<double> in that order

C4<A1, A2>     c4c;
    // inherits A1<int>, A2<int>, A1<double>, and A2<double> in that order

C4<A3, A1>     c4d;
    // inherits A3<int>, A1<int>, A3<double>, and A1<double> in that order

C4<A1, A4, A2> c4e;
    // may pass any subset of A1, A2, A3, A4, in any order
```

Quite a few other templates match `C4`'s template argument, even though they will fail to instantiate with a single parameter:

```
template <typename, typename> class A5 { /*...*/ };
template <typename, typename, typename...> class A6 { /*...*/ };

C4<A5> err1;  // Error, matches, but A5<int> and A5<double> are invalid.
C4<A6> err2;  // Error, matches, but A6<int> and A6<double> are invalid.
```

Templates that do not take type template parameters, however, will *not* match `C4` or, for that matter, any of `C1` through `C4`:

```
template <typename, int>
class A7 { /*...*/ };          // one type parameter and one non-type parameter
template <typename, int = 42>
class A8 { /*...*/ };          // same, but non-type parameter defaulted
template <template<typename> class>
class A9 { /*...*/ };          // template template parameter

C4<A7> err3;  // Error, second template argument of C4 has a different kind
C4<A8> err4;  // Error, second template argument of C4 has a different kind
C4<A9> err5;  // Error, first template argument of C4 has a different kind
```

In short, class templates that do not take specifically *types* as their template parameters cannot be used in instantiations of `C4`.

In the general case, template parameter packs may appear together with other template parameters and follow the rules and restrictions discussed so far in the context of type parameter packs. Any number of subtle but nevertheless perfectly meaningful matching cases may be defined involving combinations of fixed and variadic template parameters. Suppose, for example, we want to define a class `C5` that accepts a template that takes *at least* two parameters and possibly more:

```
template <template<typename, typename, typename...> class X>
class C5
    // class template definition having one template template parameter
    // for which the template template accepts two or more type
    // arguments
{ /*...*/ };
```

```
// A few templates that match C5.

template <typename, typename> class B1;
template <typename, typename, typename = int> class B2;
template <typename, typename = int, typename = int> class B3;
template <typename, typename, typename...> class B4;

C5<B1> c5a;  // OK
C5<B2> c5b;  // OK
C5<B3> c5c;  // OK
```

However, templates that don't have two fixed type parameters in the first two position will not match `C5`:

```
template <typename> class B5;
template <int, typename, typename...> class B6;
template <typename, typename, int, typename...> class B7;
template <typename, typename...> class B8;

C5<B5> c5d;  // Error, argument mismatch
C5<B6> c5e;  // Error, argument mismatch
C5<B7> c5f;  // Error, argument mismatch
C5<B8> c5g;  // Error, argument mismatch
```

To match `C5`'s template parameter, a template must take types for its first two parameters, followed by zero or more type parameters (with default values or not). `B5` does not match because it takes only one parameter. `B7` does not match because it takes an **`int`** in the first position, as opposed to a type, as required. `B7` fails to match because it takes an **`int`** in the third position instead of a type. Finally, `B7` is not a match because its second argument is not fixed.

To summarize our findings, template parameter packs generalize template template parameters in two distinct, orthogonal ways.

- The `...` at the template template parameter level allows zero or more template arguments to match.
- The `...` inside the parameter list of the template template parameter allows loose matching of templates with default arguments or variadic; however, type vs. value vs. template parameters are still checked, as in the example involving `A7`, `A68`, and `A9`.

`pack-expansion`

Pack expansion

Now that we have a solid command of using parameter packs in type and function declarations, it is time to explore how to use parameter packs in function implementations.

As briefly mentioned in *Variadic class templates* on page 524, parameter packs belong to a *kind* distinct from any other C++ entity; they are not types, values, template names, and so on. As far as learning parameter packs goes, they cannot be related to existing entities, so they may as well come from another language with its own syntax and semantics.

Literally the only way to use a parameter pack is to make it part of a so-called pack expansion. A pack expansion consists of a fragment of code (a "pattern") followed by `...`. The code fragment must contain at least a pack name; otherwise, it does not qualify as an expansion. Exactly what patterns are allowed depends on the place where the expansion occurs. Depending on context, the pattern is syntactically a simple identifier, a parameter declaration, an expression, or a type.

The dual use of `...` — to both introduce a pack and expand it — may seem confusing at first, but distinguishing between the two uses of `...` is easy: When the ellipsis occurs *before* a previously undefined identifier, it is meant to introduce it as the name of a parameter pack; in all other cases, the ellipsis is an expansion operator.

We've already seen pack expansion at work in *Variadic function templates* on page 533. In a variadic function template declaration, the function argument list (e.g., `Ts... values`) is an expansion resulting in zero or more by-value parameters.

To introduce a simple example of pack expansion in an actual computation, recall the `add` example in *Description* on page 519, in which a variadic function adds together an arbitrary number of integers. The full implementation shown below uses **`double`** instead of **`int`** for better usability. The important part is the expansion that keeps the computation going:

```
double add()                                    // base case, no arguments
{
    return 0;                                   // neutral element for addition
}

template <typename T, typename... Ts>           // recursive case
double add(const T& lhs, const Ts&... rest)     // accepts 1 or more arguments
{
    return lhs + add(rest...);                  // recurse expanding rest
}
```

The key to understanding how `add` works is to model the expansion `rest...` as a comma-separated list of **`double`**s, always invoking `add` with fewer arguments than it currently received; when `rest` becomes the empty pack, the expansion expands to nothing and `add()` is called, which terminates the recursion by providing the neutral value 0.

Consider the call:

```
int x1 = add(1.5, 2.5, 3.5);
```

Computation proceeds in a typical recursive manner.

1. The top-level call goes to the variadic function template `add`.
2. `add` binds `T` to **`double`** and `Ts` to `<double, double>`.
3. The expression `add(rest...)` expands into the recursive call `add(2.5, 3.5)`.
4. That call binds `T` to **`double`** and `Ts` to `<double>`.
5. The second expansion of `add(rest...)` leads to the recursive call `add(3.5)`.
6. Finally, the last expansion will recurse to `add()`, which returns 0.

The result is constructed as the recursion unwinds.[8]

From an efficiency perspective, it should be noted that `add` is not recursive in a traditional Computer Science sense. It does not call itself. Each seemingly recursive call is a call to an entirely new function with a different arity generated from the same template. After inlining and other common optimizations, the code is as efficient as writing the statements by hand.

A few rules apply to all parameter-pack expansions irrespective of their kind or the context in which they are used.

First, any pattern must contain at least one parameter pack. Therefore, expansions such as `C<int...>` or `f(5...)` are invalid. This requirement can be problematic in function declarations because a typo in a name may switch the meaning of `...` from parameter pack expansion to an old-style C variadic function declaration; see *Potential Pitfalls — Accidental use of C-style ellipsis* on page 588.

A single pack expansion may contain two or more parameter packs. In that case, expansion of multiple parameter packs within the same expansion is always carried *in lockstep*, i.e., all packs are expanded concomitantly. For example, consider a function modeled after a slightly modified `add`, and note how, instead of expanding just `rest`, we expand a slightly more complex pattern in which `rest` appears twice:

```
template <typename... Ts>
double add2(const Ts&... xs)   // accepts 0 or more arguments
{
    return add((xs * xs)...);  // expand xs * xs in call to add
}
```

This time `add2` calls `add` with the expansion `(xs * xs)...`, not just `xs...`, so we're looking at two parameter packs (the two instances of `xs`) expanded in lockstep. The call `add2(1.5, 2.5, 3.5)` will forward to `add(1.5 * 1.5, 2.5 * 2.5, 3.5 * 3.5)`, revealing that `add2` computes the sum of squares of its arguments.

The parentheses around the pattern in the expansion `(xs * xs)...` are not needed; the expansion comprehends the full expression to the left of `...`, so the call could have been written as `add(xs * xs...)`.

For expansion to work, all parameter packs in one expansion must be of the same length; otherwise, an error will be diagnosed at compile time. Suppose we define a function `f1` that takes two parameter packs and passes an expansion involving both to `add2`:

```
template <typename... Ts, typename... Us>
double f1(const Ts&... ts, const Us&... us)
{
    return add2((ts - us)...);
}
```

Here, the only valid invocations are those with equal number of arguments for `ts` and `us`:

```
double x1a = f1<double, double>(1, 2, 3, 4);
    // OK, Ts=<double, double> (explicit), Us=<int, int> (deduced)

double x1b = f1<double, double>(1, 2, 3, 4, 5);
```

[8] C++17 adds fold expressions — `return lhs + ... + rest;` — that allow a more succinct implementation of `add`.

```
// Error, parameter packs ts and us have different lengths.
// Ts=<double, double> (explicit), Us=<int, int, int> (deduced)
```

Every form of pack expansion produces a comma-separated list consisting of copies of the pattern with the parameter packs suitably expanded. Expansion is semantic, not textual — that is, the compiler is "smarter" than a typical text-oriented preprocessor. If the pack expansion is within a larger comma-separated list and the parameter packs being expanded have zero elements, the commas surrounding the expansion are adjusted appropriately to avoid syntax errors. For example, if `ts` is empty, the expansion `add(1, ts..., 2)` becomes `add(1, 2)`, not `add(1, , 2)`.

Expansion constructs may be nested. Each nesting level must operate on at least one parameter pack:

```
template <typename... Ts>
double f2(const Ts&... ts)
{
    return add2((add2(ts...) + ts)...);
}
```

Expansion always proceeds "inside out," with the innermost expansion carried first. In the call `f2(2, 3)`, the expression returned after the first, innermost `ts` is expanded is `add2((add2(2, 3) + ts)...)`. The second expansion results in the considerably heftier expression `add2(add2(2, 3) + 2, add2(2, 3) + 3)`. Nested expansions grow in a combinatorial manner, so care is to be exercised.

What forms of `pattern...` are allowed, and where in the source code is the construct allowed? Parameter pack expansions are defined for a variety of well-defined contexts, each of which is separately described in the following subsections. Not all potentially useful contexts are supported, however; see **??** — *Limitations on expansion contexts* on page 590[**AUs: There is no section called "Expansion at the statement level is not permitted." Response: limitations on expansions contexts is the appropriate reference**] Expansion is allowed in:

- a function parameter pack
- a function call argument list or a braced initializer list
- a template argument list
- a base specifier list
- a member initializer list
- a lambda capture list
- an alignment specifier
- an attribute list
- a **`sizeof...`** expression
- a template parameter pack that is a pack expansion

function-parameter-pack

Expansion in a function parameter pack

An ellipsis in a function declaration's parameter list expands to a list of parameter declarations. The pattern being expanded is a parameter declaration.

We discussed this expansion at length in the preceding subsections, so let's quickly recap by means of a few examples:

```
template<typename... Ts>
void f1(Ts...);                // expands to T0, T1, T2,...

template<typename... Ts>
void f2(const Ts&...);         // const T0&, const T1&,...

template<typename... Ts>
void f3(const C<Ts>&...);      // Complex use, e.g., in templates, is allowed.

template<typename... Ts, typename... Us>
void f4(const C<Ts, Us>&...);  // const C<T0, U0>&, const C<T1, U1>&,...
```

braced-initializer-list

Expansion in a function call argument list or a braced initializer list

Expansion may occur in the argument list of a function call or in an initializer list — either parenthesized or *brace-enclosed* (see Section 2.1."Braced Init" on page 198). In these cases, the pattern is the *largest expression or braced initialization list* to the left of the ellipsis.

This expansion is the only one that expands into expressions (all others are declarative), so in a way it is the most important because it relates directly to runtime work getting done.

Let's look at a few examples of expansion. Suppose we have a library that comprises three variadic function templates, `f`, `g`, and `h`, and an ordinary class, `C`, having a variadic *value* constructor:

```
template <typename... Ts> int f(Ts...);  // variadic function template
template <typename... Ts> int g(Ts...);  //    "        "        "
template <typename... Ts> int h(Ts...);  //    "        "        "

struct C                                 // ordinary class
{
    template <typename... Ts> C(Ts...);  // variadic value constructor
};
```

Let's now suppose that we another variadic function template, `client1`, that intends to make use of this library by expanding its own parameter pack, `xs`, in various contexts:

```
template <typename... Ts>
void client1(Ts... xs)
{
    f(xs...);                  // (1) f(x0, x1, ...);
    f(C(xs...));               // (2) f(C(x0, x1, ...));
    f(C(xs)...);               // (3) f(C(x0), C(x1), ...);
    f(3.14, xs + 1 ...);       // (4) f(3.14, x0 + 1, x1 + 1, ...);
    f(3.14, xs * 2. ...);      // (5) f(3.14, x0 * 2., x1 * 2., ...);
}
```

In comments, we informally denote with `x0`, `x1`, `...`, the elements of the pack `xs`. The first call, `f(xs...)`, illustrates the simplest expansion; the pattern being expanded, `xs`, is simply the name of a function parameter pack. The expansion results in a comma-separated list of the arguments received by `client1`.

The other examples illustrate a few subtleties. Examples (2) and (3) show how the positioning of `...` determines how expansion unfolds. In (2), the expansion is carried inside the call to `C`'s constructor, so `f` is called with exactly one object of type `C`. In (3), the ellipsis occurs outside the constructor call, so `f` gets called with zero or more `C` objects, each constructed with exactly one argument.

Examples (4) and (5) show how whitespace may be important. If the space before `...` were missing, the C++ parser would encounter `1...` and `2...`, both of which are incorrect floating-point literals.

Let's now look at a few more complex examples in another function, `client2`:

```
template <typename... Ts>
void client2(Ts... xs)
{
    f("hi", xs + xs..., 3.14);  // (6) f(x0 + x0, x1 + x1, ..., 3.14);
    f(const_cast<Ts&>(xs)...);  // (7) f(const_cast<T1&>(x0), ...);
    f(g(xs)..., h(xs...));      // (8) f(g(x0), g(x1), ..., h(x0, x1, ...))
    C object1(*xs...);          // (9) C::C(*x0, *x1, ...)
    C object2{*xs...};          // (10) C::C(*x0, *x1, ...)
    int a[] = { xs..., 0 };     // (11) int[] a = { x0, x1, ..., 0 }
}
```

Examples (6) and (7) feature simultaneous expansion of two packs. In (6), `xs` is expanded twice. In (7), template parameter pack `Ts` and value parameter pack `xs` are both expanded. In all cases of simultaneous expansion, the two or more packs are expanded in lockstep, i.e., the first element in `Ts` together with the first element in `vs` form the first element in the expansion, and so on. Attempting to expand packs of unequal lengths results in a compile-time error. Example (6) also shows how an expansion may be somewhere in the middle of a function's argument list.

Example (8) shows two sequential expansions that look similar but are quite different. The two expansions are independent and can be analyzed separately. In `g(xs)...`, the pattern being expanded is `g(xs)` and results in the list `g(x0)`, `g(x1)`, `...`. In contrast, in the expansion `h(xs...)`, the expansion is carried inside the call to `h` — `h(x0, x1, ...)`.

Example (9) shows that expansion is allowed inside special functions as well. The expansion `C(*xs...)` results in `C`'s constructor called with `*x0`, `*x1`, and so on.

Last but not least, examples (10) and (11) illustrates expansion inside a braced-initialization list. Example (10) calls the same constructor as (9), and example (11) initializes an array of `int` with the content of `xs` followed by a `0`.

For each of these examples to compile, the code resulting from pack expansions needs to pass the usual semantic checks; for example, the `int` array initialization in (11) would fail to compile if `xs` contained a value with a type not convertible to `int`. As is always the case with templates, some instantiations work, whereas some don't.

In the examples above, the functions involved are all variadic. However, a function doesn't need to be variadic — or template for that matter — to be called with an expansion. The

expansion is carried in the function call expression, and then the usual lookup rules apply to decide whether the call is valid:

```
int f1(int a, double b);  // simple function with two parameters

int f2();                 // no parameters
int f2(int a, double b);  // overload with two parameters

template <typename... Ts>
void client3(const Ts&... xs)
{
    f1(xs...);  // Works if and only if xs has exactly two elements
                // convertible to int and double, respectively.

    f2(xs...);  // Works if and only if xs is empty or has exactly two elements
                // convertible to int and double, respectively.
}
```

-template-argument-list

Expansion in a template argument list

We have already encountered pack expansion in the argument list of a template instantiation; if `C` is a class template and `Ts` is a template parameter pack, `C<Ts...>` instantiates `C` with the contents of `Ts`. There is no need for the template `C` to accept a variadic list of parameters. The resulting expansion must be appropriate for the template instantiated.

For example, suppose we define a variadic class `Lexicon` that uses its parameter pack in an instantiation of `std::map`:

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

template <typename... Ts>               // template parameter pack
class Lexicon                           // variadic class template
{
    std::map<Ts...> d_data;             // Use Ts to instantiate std::map.
    // ...
};

Lexicon<std::string, int> c1;           // (1) OK, std::map<std::string, int>
Lexicon<int> c2;                        // (2) Error, std::map<int> invalid
Lexicon<int, long, std::less<int>> c3;  // (3) OK
Lexicon<long, int, 42> c4;              // (4) Error, 42 instead of functor
```

Given that `Lexicon` forwards all of its template arguments to `std::map`, the only viable template arguments for `Lexicon` are those that would be viable for `std::map` as well. Therefore, (1) is valid because it instantiates `std::map<std::string, int>`. As usual, `std::map`'s default arguments for its third and fourth parameters kick in; the pack expansion does not affect default template arguments. To wit, (3) passes three template arguments to `std::map` and leaves the last one (the allocator) to be filled with the default. Instantiations (2) and (4) of `Lexicon` are not valid because they would attempt to instantiate `std::map` with incompatible template arguments.

a-base-specifier-list

Expansion in a base specifier list

Suppose we set out to define a variadic template, `MB1`, that inherits all of its template arguments. This scheme is useful in applying design patterns such as Visitor[9] or Observer[10]. To enable such designs, expansion is allowed in a base specifier list. The pattern under expansion is a base specifier, which includes an optional protection specifier (**`public`**, **`protected`**, or **`private`**) and an optional **`virtual`** base specifier. Let us define `MB1` to inherit all of its template arguments using **`public`** inheritance:

```
template <typename... Ts>  // template parameter pack
class MB1 : public Ts...   // multibase class, publicly inherit each of Ts
{
    // ...
};
```

The pattern **`public`** `Ts...` expands into **`public`** `T0`, **`public`** `T1`, and so on for each type in `Ts`. All bases resulting from the expansion have the same protection level. If `Ts` is empty, `MB1<>` has no base class.

```
class S1 { /*...*/ };  // arbitrary class
class S2 { /*...*/ };  // arbitrary class

MB1<>           m1a;     // OK, no base class at all
MB1<S1>         m1b;     // OK, instantiate with S1 as only base
MB1<S1, S2>     m1c;     // OK, instantiate with S1 and S2 as bases.
MB1<S1, S2, S1> m1d;     // Error, cannot inherit S1 twice
MB1<S1, int>    m1e;     // Error, cannot inherit from scalar type int
```

After expansion, the usual rules and restrictions apply; a class cannot inherit another one twice and cannot inherit types such as **`int`**.

Other bases may be specified before and/or after the pack. The other bases may specify other protection levels (and if they don't, the default protection level applies):

```
template <typename... Ts>      // parameter pack Ts
class MB2
    : virtual private S1       // S1 virtual private base
    , public Ts...            // inherit each of Ts publicly
{ /*...*/ };

template <typename... Ts>      // parameter pack Ts
class MB3
    : public S1               // S1 public base
    , virtual protected Ts... // each type in Ts a virtual protected base
    , S2                      // S2 private base (uses default protection)
{ /*...*/ };

MB2<>   m2a;                  // (1) virtual private base S1
MB2<S2> m2b;                  // (2) virtual private base S1, public base S2
MB3<>   m3a;                  // public base S1, private base S2
```

[9] **?**, Chapter 10, "Visitor," pp.!235–262
[10] **?**, Chapter 5, section "Observer," pp. 293–303

```
MB3<S2> m3b;                    // Error, cannot inherit S2 twice
```

Expansions are not limited to simple pack names. The general pattern allowed in a base specifier list is that of a full-fledged base specifier. For example, the parameter pack can be used to instantiate another template:

```
template <typename T>
class Act                       // arbitrary class template
{ /*...*/ };

template <typename... Ts>       // template parameter pack Ts
class MB4
    : public Act<Ts>...         // bases Act<T0>, Act<T1>, ...
{ /*...*/ };

MB4<>                     m4a;  // no base class
MB4<int, double>          m4b;  // bases Act<int>, Act<double>
MB4<MB4<int>, int>        m4c;  // bases Act<MB4<int>>, Act<int>
```

Arbitrarily complex instantiations can be specified in an base specifier pack expansion, which opens to opportunity for a variety of expansion patterns. Depending on where the ellipsis is placed, different expansion patterns can be created.

```
template <typename... T>
class Avct                  // arbitrary variadic class template
{ /*...*/ };

template <typename... Ts>  // template parameter pack Ts
class MB5                  // multibase class example
: public Avct<Ts>...       // zero or more: Avct<T0>, Avct<T1>, ...
, private Avct<Ts...>      // exactly one: Avct<T0, T1, ...>
{ /*...*/ };
```

Although the two expansions featured above are similar in syntax, they are semantically very different. First, `public Avct<Ts>...` expands into multiple bases for `MB5`: `public Avct<T0>`, `public Avct<T1>`, and so on. The second expansion is completely different; in fact, it's not even an expansion in a base specifier list. Its context is a template's argument list; see *Expansion in a template argument list* on page 557. The result of that expansion is a single class `Avct<T0, T1, ...>` that is an additional private base of `MB5`:

```
MB5<int, double> mb5a;
    // inherits publicly Avct<int>, Avct<double>
    // inherits privately Avct<int, double>

MB5<Avct<int, char>, double> mb5b;
    // inherits publicly Avct<Avct<int, char>>, Avct<double>,
    // inherits privately Avct<Avct<int, char>, double>

MB5<int> mb5c;
    // Error, cannot inherit Avct<int> twice
```

mber-initializer-list

Expansion in a member initializer list

This feature is a "forced move" of sorts. Allowing variadic bases on a class naturally creates the necessity of being able to initialize those bases accordingly. Consequently, parameter pack expansion is allowed in a base initializer list. The pattern is a base initializer, i.e., the name of a base followed by a parenthesized list of arguments for its constructor:

```
template <typename... Ts>  // template parameter pack
struct S1 : Ts...          // publicly inherit every type in the pack
{
    S1() : Ts(0)...        // (1) call constructor with 0 for each base
    { /*...*/ }

    S1(int x) : Ts(x)...   // (2) call constructor with x for each base
    { /*...*/ }

    S1(const S1& rhs) : Ts(static_cast<const Ts&>(rhs))...
        // OK, call the copy constructor for each base
    { /*...*/ }
};
```

The default constructor (1) calls all bases' constructors, passing `0` to each. The second constructor (2) passes its one **`int`** argument to each base. The last constructor (3) of `S` implements the copy constructor and is rather interesting because it expands in turn to a call to the copy constructor for each member, passing it the result of a **`static_cast`** (which, in fact, is implicit) to the appropriate base type. Similar syntax can be used for defining the move constructor (see Section 2.1."*rvalue* References" on page 479) and other constructors.

Let's embark on a more complex example. Suppose we want to define a class `S2` with a constructor that accepts any arguments and forwards them to all of its base classes. To do so, that constructor itself needs to be variadic with a distinct parameter pack:

```
template <typename... Ts>      // template parameter pack
struct S2 : Ts...              // publicly inherit every type in the pack
{
    template <typename... Us>  // variadic constructor
    S2(const Us&... xs)        // accepts any number of arguments by const &
    : Ts(xs...)...             // (!) forwards them to each base constructor
    { }
};
```

The code above has at least one ellipse on every significant line, and all are needed. Let's take a closer look.

First off, class `S2` inherits all of its template arguments. It has no exact knowledge about the types it would be instantiated with and, out of consideration for flexibility, defines a variadic constructor that forwards any number of arguments from the caller into each of its base classes. That constructor, therefore, is itself variadic with a separate template parameter pack, `Us`, and a corresponding argument pack, `xs`, that accepts arguments by reference to **`const`**. The key line, commented with `(!)` in the code, performs two expansions, which proceed inside out. First, `xs...` expands into the list of arguments passed to `S2`'s constructor, leading to the pattern `Ts(x0, x1, ...)`. In turn, that pattern itself gets expanded by

the outer `...` into a base initialization list: `T0(x0, x1, ...), T1(x0, x1, ...), ...`.

What if pass by reference to **`const`** is too constraining, and we would want to define a more general constructor that can forward modifiable values as well? In that case, we need to use forwarding references (see Section 2.1."Forwarding References" on page 351) and the Standard Library function `std::forward`:

```
#include <utility>  // std::forward

template <typename... Ts>      // template parameter pack
struct S3 : Ts...              // publicly inherit every type in the pack
{
    template <typename... Us>  // variadic constructor
    S3(Us&&... xs)             // arguments by forwarding reference
    : Ts(std::forward<Us>(xs)...)...
                               // forwards them to each base constructor
    { }
};
```

`S3`'s variadic constructor makes use of **forwarding references**, which automatically adapt to the type of the arguments passed. The library function `std::forward` ensures that each argument is forwarded with the appropriate type, qualifier, and *lvalue*ness to the constructors of each base class. The expansion process is similar to that in `S2`'s constructor previously discussed, with the additional detail that `std::forward<Us>(xs)...` expands in lockstep `Us` and `xs`.

Expansion in a lambda capture list

n-a-lambda-capture-list

A C++ lambda expression, introduced with C++11 (see Section 2.1."Lambdas" on page 393, is an unnamed function object. A lambda can store internally some of the local variables present at the point of creation by a mechanism known as **lambda capture**. This important capability distinguishes lambdas from simple functions. Let's put it to use in defining a `tracer` lambda that is able to print a given variable to the console:

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

template <typename T>  // single-parameter template
auto tracer(T& x)      // returns a lambda that, when invoked, prints x
{
    auto result = [&x]() { std::cout << x << std::endl; };
        // [&x] means the function object captures x by reference.
        // tracer must use auto to initialize the function object.
    return result;
}
```

Lambdas have a type chosen by the compiler, so we need **`auto`** to pass lambda objects around; see Section 2.1."**`auto`** Variables" on page 183.

This may seem like a lot if you're new to lambdas, in which case reading locationc."Lambdas" on page 393 along with Section 3.2."Deduced Return Type" on page 687 before continuing may be in order. The underlying idea is simple: A call such as `tracer(x)` saves a reference to `x` in a function object, which it then returns. Subsequent calls to that function object

output the current value of `x`. It's important to save `x` by reference (hence the `&` in the capture), lest the lambda store `x` by value and uninterestingly print the same thing on each call.

Let's see `tracer` at work, tracing some variable in a function:

```
int process(int x)           // uses the trace facility
{
    auto trace = tracer(x);  // Initialize trace to follow x.
    trace();                 // prints current value of x
    ++x;                     // change the value of x
    trace();                 // prints current (changed) value of x
    return x;                // Return x back to the caller.
}

int x0 = process(42);         // prints 42, then 43, and initializes x0 to 43
```

What is the connection with variadics? Variadics are all about generalization, which applies here as well. Suppose we now set out to trace several variables at once. Instead of a one-argument function, `tracer` would then need to be variadic. Also, crucially, the lambda returned needs to store references to all arguments received in order to print them later. That means an expansion must be allowed inside a lambda capture list.

First, let's assume a function, `print`, exists that prints any number of arguments to the console (*Use Cases — Generic variadic functions* on page 567 features an implementation of `print`):

```
template <typename... Ts>     // variadic function
void print(const Ts&... xs);  // prints each argument to std::cout in turn
```

The definition of `multitracer` uses `print` in a lambda with variadic capture:

```
template <typename... Ts>    // variadic template
auto multitracer(Ts&... xs)  // returns a lambda that, when invoked, prints xs
{
    auto result = [&xs...]() { print(xs...); };
        // [&xs...] means capture all of xs by reference.
        // result stores one reference for each argument.
    return result;
}
```

The entire API is enabled by the ability to expand `xs` inside the capture list. Expanding with `[&xs...]` captures by reference, whereas `[xs...]` captures the pack by value. Inside the lambda, `print` expands the pack as usual with `x...`.

Expansions in captures can be combined with all other captures:

```
template <typename... Ts>    // variadic template
auto test(Ts&... xs)         // for illustration purposes
{
    int a = 0, b = 0;
    auto f1 = [&a, xs...]() { /*...*/ };
        // Capture a by reference and all of xs by value.
```

```
    auto f2 = [xs..., &a]() { /*...*/ };
        // same capture as f1

    auto f3 = [a, &xs..., &b]() { /*...*/ };
        // Capture a by value, all of xs by reference, and b by reference.

    auto f4 = [&, xs...]() { /*...*/ };
        // Capture all of xs by value and everything else by reference.

    auto f5 = [=, &xs..., &a]() { /*...*/ };
        // Capture a and all of xs by reference, and everything else by value.
}
```

The pattern must be that of a simple capture; complex capture patterns are not allowed as of C++14.[11]

-an-alignment-specifier

Expansion in an alignment specifier

The **alignas** specifier is a feature new to C++11 that allows specifying the alignment requirement of a type or object; see Section 2.1."**alignas**" on page 158:

```
alignas(8)      float x2;  // Align x1 at an address multiple of 8.
alignas(double) float x1;  // Align x2 with the same alignment as a double.
```

Pack expansion inside the **alignas** specifier is allowed. The meaning in the presence of the pack is to specify the largest alignment of all types in the pack:

```
template <typename... Ts>    // variadic template
int test1(Ts... xs)          // for illustration purposes
{
    struct alignas(Ts...) S { };
        // Align S at the largest alignment of all types in Ts.
        // If Ts is empty, the alignas directive ignored.

    alignas(Ts...) float x1;
        // Align x1 at the largest alignment of all types in Ts.
        // If Ts is empty, the alignas directive ignored.

    alignas(Ts...) alignas(float) float x2;
        // Align x2 at the largest alignment of double and all types in Ts.

    alignas(float) alignas(Ts...) float x3;
        // same alignment as x2; order does not matter

    return 0;
}
```

As always with **alignas**, requesting an alignment smaller than the minimum alignment required by the declaration is an error:

[11] C++20 introduces pack expansion in lambda initialization captures (see **?**) that allows capturing variadics with a syntax such as `[...us = vs]` or `[...us = std::move(vs)]`.

```
int a1 = test1();          // OK, Ts empty, all alignas(Ts...) ignored
int a2 = test1('a', 1.0);  // OK, align everything as a double.
int a3 = test1('a');       // Error (most systems), can't align x1 to 1 byte
```

An idiom that avoids such errors is to use two **alignas** for a given declaration, one of which is the natural alignment of the declaration. This is the idiom followed by the declarations of `x2` and `x3` in the definition of function template `test1`.

Using handwritten, comma-separated lists inside an **alignas** specifier is, however, not allowed. Expansion *outside* the specifier is also disallowed:

```
template <typename... Ts>    // variadic template
void test2(Ts... xs)          // for illustration purposes
{
    alignas(Ts)... float x4;
        // Error, cannot expand outside the alignas specifier

    alignas(double, Ts...) float x5; // Error, syntax not allowed
    alignas(Ts..., double) float x6; // Error, syntax not allowed
    alignas(long, double) float x7;  // Error, syntax not allowed
}
```

To conclude, pack expansion in an **alignas** specifier allows choosing without contortions the largest alignment of a parameter pack. Combining two or more **alignas** specifiers facilitates a simple idiom for avoiding errors and corner cases.

Expansion in an attribute list

-in-an-attribute-list

Attributes, introduced with C++11, are a mechanism for adding built-in or user-defined information about declarations; see Section 1.1."Attribute Syntax" on page 10.

Attributes are added to declaration using the syntax `[[attribute]]`. For example, `[[noreturn]]` is a standard attribute indicating that a function will not return:

```
[[noreturn]] void abort();  // Once called, it won't return.
```

Two or more attributes can be applied to a declaration either independently or as a comma-separated list inside the square brackets:

```
[[noreturn]] [[deprecated]] void finish();  // won't return, also deprecated
[[deprecated, noreturn]]    void finish();  // same
```

For completeness and future extensibility, pack expansion is allowed inside an attribute specifier as in `[[attribute...]]`. However, this feature is not currently usable with any current attribute, standard or user-defined:

```
template <typename... Ts>
[[Ts()...]] void functionFromTheFuture();
    // NONWORKING CODE
    // Receive a number of types; instantiate them as attributes.
```

The ability to expand packs inside attribute specifiers is reserved for future use and good to keep in mind for future additions to the language.

-a-sizeof...-expression

Expansion in a `sizeof...` expression

The **`sizeof...`** expression is an oddity in three ways. First, it has nothing to do with classical **`sizeof`** in the sense that **`sizeof...`** does not yield the extent occupied in memory by an object. Second, it is the only parameter pack expansion that does *not* use the (by now familiar) `pack...` syntax. And third, although it is considered an expansion, it does not expand a pack into its constituents.

For any parameter pack `P`, **`sizeof...(P)`** yields to a compile-time constant of type `size_t` equal to the number of elements of `P`:

```
    std::size_t
template <typename... Ts>
std::size_t countArgs(Ts... xs)
{
    std::size_t x1 = sizeof...(Ts);        // x1 is the number of parameters
    std::size_t x2 = sizeof...(xs);        // same value as x1
    static_assert(sizeof...(Ts) >= 0,"");  // sizeof...(Ts) is a constant.
    static_assert(sizeof...(xs) >= 0,"");  // sizeof...(xs) is a constant.
    return sizeof ... (Ts);                // whitespace around ... allowed
}
```

Let's see `countArgs` in action:

```
std::size_t a0 = countArgs();           // initialized to 0
std::size_t a1 = countArgs(42);         // initialized to 1
std::size_t a2 = countArgs("ab", 'c');  // initialized to 2
```

Whitespace is allowed around the `...`, but parentheses are not optional. Also, expansion is disallowed inside **`sizeof`** and **`sizeof...`** alike:

```
template <typename... Ts>
std::size_t nogo(Ts... xs)
{
    std::size_t x1 = sizeof 42;         // OK, same as sizeof(int)
    std::size_t x2 = sizeof... Ts;      // Error, parens required around Ts
    std::size_t x3 = sizeof... xs;      // Error, parens required around xs
    std::size_t x4 = sizeof(Ts...);     // Error, cannot expand inside sizeof
    std::size_t x5 = sizeof...(Ts...);  // Error, cannot expand inside sizeof...
    std::size_t x6 = sizeof(xs...);     // Error, cannot expand inside sizeof
    std::size_t x7 = sizeof...(xs...);  // Error, cannot expand inside sizeof...
}
```

template-parameter-list

Expansion inside a template parameter list

Not to be confused with the case discussed in *Expansion in a template argument list* on page 557, pack expansion may occur in a template *parameter* (not argument) list. This is different from all other expansion cases because it involves two distinct parameter packs: a type parameter pack and a non-type parameter pack. To set things up, suppose we define a class template, `C1`, that has a type parameter pack, `Ts`. Inside, we define a secondary class template, `C2`, that does *not* take any type parameters. Instead, it takes a non-type template parameter pack with types derived from `Ts`:

```
template <typename... Ts>  // type parameter pack
struct C1                  // class template
{
    template <Ts... vs>    // non-type parameters (attention: no typename!)
    struct C2 { };         // expansion in C2's parameter list
};
```

Once `C1` is instantiated with some types, the inner class `C2` will accept *values* of the types used in the instantiation of `C1`. For example, if `C1` is instantiated with **`int`** and **`char`**, its inner class template `C2` will accept an **`int`** value and a **`char`** value:

```
C1<int, char>::C2<1, 'a'> x1;       // OK, C2 takes an int and a char.
C1<int, char>::C2<1> x2;            // Error, too few arguments for C2
C1<int, char>::C2<1, 'a', 'b'> x3;  // Error, too many arguments for C2
```

Only instantiations of `C1` that lead to valid declarations of `C2` are allowed. For example, user-defined types are not allowed as non-type template parameters, and consequently `C1` cannot be instantiated with a user-defined type:

```
class AClass { };   // simple user-defined class

C1<int, AClass>::C2<1, AClass()> x1;
    // Error, a non-type template parameter cannot have type AClass.
```

er-expansion-contexts

No other expansion contexts

Note that what's missing is as important as what's present. Parameter pack expansion is explicitly disallowed in any other context, even if it would make sense syntactically and semantically:

```
template <typename... Ts>
void bumpAll(Ts&... xs)
{
    ++xs...;  // Error, cannot expand xs in an expression-statement context
}
```

Annoyances — Limitations on expansion contexts on page 590[**AUs: There is no subsection called "Expansion at the statement level is not permitted" Response: limitations on expansion contexts is the correct reference**] discusses this context further. Also recall that it is illegal to use pack names anywhere without expanding them, so they don't enjoy first-class status; see *Annoyances — Parameter packs cannot be used unexpanded* on page 591.

contexts-and-patterns

Summary of expansion contexts and patterns

To recap, expansion is allowed in only the following places:

Context	Pattern
function parameter pack	parameter declaration
function call argument list or a braced initializer list	function argument
template argument list	template argument
base specifier list	base specifier
member initializer list	base initializer
lambda capture list	capture
alignment specifier	alignment specifier
attribute list	attribute
`sizeof...` expression	identifier
template parameter pack that is a pack expansion	parameter declaration

use-cases-variadic

Use Cases

eric-variadic-functions

Generic variadic functions

A variety of functions of general utility are naturally variadic, either mathematically (`min`, `max`, `sum`) or as a programmer's convenience. Suppose, for example, we want to define a function, `print`, that writes its arguments to `std::cout` in turn followed by a newline[12]:

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

std::ostream& print()                // parameterless overload
{
    return std::cout << std::endl;   // only advances to next line
}

template <typename T, typename... Ts>             // one or more types
std::ostream& print(const T& x, const Ts&... xs)  // one or more args
{
    std::cout << x;                               // output first argument
    return print(xs...);                          // recurse to print rest
}

void test()
{
    print("Pi is about ", 3.14159265);            // "Pi is about 3.14159"
}
```

The implementation follows a head-and-tail recursion that is typically used for C++ variadic function templates. The first overload of `print` has no parameters and simply outputs a newline to the console. The second overload does the bulk of the work. It takes one or more arguments, prints the first, and recursively calls `print` to print the rest. In the limit, `print` is called with no arguments, and the first definition kicks in, outputting the line terminator and also ending the recursion.

A variadic function's smallest number of allowed arguments does not have to be zero, and it is free to follow many other recursion patterns. For example, suppose we want to define a variadic function `isOneOf` that returns **`true`** if and only if its first argument is

[12] C++20 introduces `std::format`, a facility for general text formatting.

equal to one of the subsequent arguments. Calls to such a function are sensible for two or more arguments:

```
template <typename T1, typename T2>     // normal template function
bool isOneOf(const T1& a, const T2& b)  // two-parameter version
{
  return a == b;
}

template <typename T1, typename T2, typename... Ts>      // two or more
bool isOneOf(const T1& a, const T2& b, const Ts&... xs)  // all by const&
{
  return a == b || isOneOf(a, xs...);                    // compare, recurse
}
```

Again, the implementation uses two definitions in a pseudo-recursive setup but in a slightly different stance. The first definition handles two items and also stops recursion. The second version takes three or more arguments, handles the first two, and issues the recursive call only if the comparison yields **`false`**.

Let's take a look at a few uses of `isOneOf`:

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

int a = 42;
bool b1 = isOneOf(a, 1, 42, 4);  // b1 is true.
bool b2 = isOneOf(a, 1, 2, 3);   // b2 is false.
bool b3 = isOneOf(a, 1, "two");  // Error, can't compare int with const char*
std::string s = "Hi";
bool b4 = isOneOf(s, "Hi", "a"); // b4 is true.
```

Object factories

object-factories

Suppose we want to define a generic **factory function** — a function able to create an instance of any given type by calling one of its constructors. Object factories[13,14] allow libraries and applications to centrally control object creation for a variety of reasons: using special memory allocation, tracing and logging, benchmarking, object pooling, late binding, deserialization, interning, and more.

The challenge in defining a generic object factory is that the type to be created (and therefore its constructors) is not known at the time of writing the factory. That's why C++03 object factories typically offer only default object constructions, forcing clients to awkwardly use two-phase initialization, first to create an empty object and then to put it in a meaningful state.

Writing a generic function that can transparently forward calls to another function ("perfect forwarding") has been a long-standing challenge in C++03. An important part of the puzzle is making the forwarding function generic in the number of arguments, which is where variadic templates help in conjunction with forwarding references (see Section 2.1."Forwarding References" on page 351):

[13]**?**, Chapter 1-5, section "Factory Method," pp. 107–115
[14]**?**, Chapter 8, "Object Factories," pp. 197–218

```
#include <utility>  // std::forward

void log(const char* message);                // logging function

template <typename Product, typename... Ts>   // type to be created and params
Product factory(Ts&&... xs)                   // call by forwarding reference
{
    log("factory(): Creating a new object");  // Do some logging.
    return Product(std::forward<Ts>(xs)...);  // Forward arguments to ctor.
}
```

`Ts&&... xs` introduces `xs`, a function parameter pack that represents zero or more forwarding references. As we know, the construct `std::forward<Ts>(xs)...` is a pack expansion that expands to a comma-separated list `std::forward<T0>(x0), std::forward<T1>(x1),` and so on. The Standard Library function template `std::forward` passes accurate type information from the forwarding references `x0`, `x1`, `...` to `Product`'s constructor.

To use the function, we must always provide the `Product` type explicitly; it is not a function parameter, so it cannot be deduced. The others are at best left to template argument deduction. In the simplest case, `factory` is usable with primitive types:

```
int i1 = factory<int>();              // Initialize i1 to 0.
int i2 = factory<int>(42);            // Initialize i2 to 42.
```

It also works correctly with overloaded constructors:

```
struct Widget
{
    Widget(double);        // constructor taking a double
    Widget(int&, double);  // constructor taking an int& and a double
};

int g = 0;
Widget w1 = factory<Widget>(g, 2.4);   // calls ctor with int& and double
Widget w2 = factory<Widget>(20);       // calls ctor with double
Widget w3 = factory<Widget>(20, 2.0);  // Error, cannot bind rvalue to int&
```

The last line introducing `w3` fails to compile because the *rvalue* `20` cannot convert to the non-`const int&` required by `Widget`'s constructor — an illustration of perfect forwarding doing its job.

Many variations of object factories (e.g., using dynamic allocation, custom memory allocators, and special exceptions treatment) can be built on the skeleton of `factory` shown. In fact, Standard Library factory functions, such as `std::make_shared` and `std::make_unique`, use variadics and perfect forwarding in this same manner.

Hooking function calls

hooking-function-calls

Forwarding is not limited to object construction. We can use it to intercept function calls in a generic manner and add processing such as tracing, logging, and so on. Suppose, for example, writing a function that calls another function and logs any exception it may throw:

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

```
#include <utility>    // std::forward

void log(const char* msg);                    // Log a message.

template <typename Callable, typename... Ts>
auto logExceptions(Callable&& fun, Ts&&... xs)
    -> decltype(fun(std::forward<Ts>(xs)...))
{
    try
    {
        return fun(std::forward<Ts>(xs)...);   // perfect forwarding to fun
    }
    catch (const std::exception& e)
    {
        log(e.what());                         // log exception information
        throw;                                 // Rethrow the same exception.
    }
    catch (...)
    {
        log("Nonstandard exception thrown.");  // log exception information
        throw;                                 // Rethrow the same exception.
    }
}
```

Here, we enlist not only the help of `std::forward` but also that of the **`auto -> decltype`** idiom; see Section 1.1."Trailing Return" on page 112 and Section 1.1."**`decltype`**" on page 22. By using **`auto`** instead of the return type of `logExceptions` and following with `->` and the trailing type **`decltype`**`(fun(std::forward<Ts>(xs)...))`, we state that the return type of `logExceptions` is the same as the type of the call `fun(std::forward<Ts>(xs)...)`, which matches perfectly the expression that the function will actually return.

In case the call to `fun` throws an exception, `logExceptions` catches, logs, and rethrows that exception. So `logExceptions` is entirely transparent other than for logging the passing exceptions. Let's see it in action. First, we define a function, `assumeIntegral`, that is likely to throw an exception:

```
#include <stdexcept>  // std::runtime_error

long assumeIntegral(double d)            // throws if d has a fractional part
{
    long result = static_cast<long>(d);  // Compute the returned value.
    if (result != d)                     // Verify.
        throw std::runtime_error("Integral expected");
    return result;
}
```

To call `assumeIntegral` via `logExceptions`, we just pass it along with its argument:

```
void test()
{
    long a = logExceptions(assumeIntegral, 4.0);  // Initialize a to 4.
```

```
    long b = logExceptions(assumeIntegral, 4.4);  // throws and logs
}
```

Tuples

tuples

A *tuple* or a record is a type that groups together a fixed number of values of unrelated types. The C++03 Standard Library template `std::pair` is a tuple with two elements. The standard library template `std::tuple`, introduced in C++11, implements a tuple with the help of variadic templates. For example, `std::tuple<int, int, float>` holds two `int`s and a `float`.

There are many possible ways to implement a tuple in C++. C++03 implementations typically define a hardcoded limit on the number of values the tuple can hold and use considerable amounts of scaffolding, as described in *Description* on page 519:

```
struct None { };  // empty "tag" used as a special "not used" marker

template <typename T1 = None, typename T2 = None, typename T3 = None,
          typename T4 = None, typename T5 = None, typename T6 = None,
          typename T7 = None, typename T8 = None, typename T9 = None>
class Cpp03Tuple;
    // tuple containing up to 9 data members of arbitrary types
```

Variadics are a key ingredient in a scalable, manageable tuple implementation. We discuss a few possibilities in approaching the core definition of a tuple, with an emphasis on data layout.

The definition of `Tuple1` (in the code snippet below) uses specialization and recursion to accommodate any number of types:

```
    std::size_tstd::stringassertstd::runtime_error
```

```
template <typename... Ts>
class Tuple1;                // (0) incomplete declaration

template <>
class Tuple1<>               // (1) specialization for zero elements
{ /*...*/ };

template <typename T, typename... Ts>
class Tuple1<T, Ts...>       // (2) specialization for one or more elements
{
    T first;                 // first element
    Tuple1<Ts...> rest;      // all other elements
    // ...
};
```

`Tuple1` uses composition and recursion to create its data layout. As discussed in *Expansion in a base specifier list* on page 558, the expansion `Tuple1<Ts...>` results in `Tuple1<T0, T1, ..., Tn>`. The specialization `Tuple1<>` ends the recursion.

The only awkward detail is that `Tuple1<int>` has member `rest` of type `Tuple1<>`, which is empty but is required to have nonzero size, so it ends up occupying space in the

tuple.[15] This is an important issue if the tuple is to be used at scale.

A solution to avoid this issue is to partially specialize `Tuple1` for one element in addition to the two existing specializations:

```
template <typename T>
class Tuple1<T>              // (3) specialization for one element
{
    T first;
    // ...
};
```

With this addition, `Tuple1<>` uses the total specialization (1), `Tuple1<int>` uses the partial specialization (3), and all instantiations with two or more types use the partial specialization (2). For example, `Tuple1<int, long, double>` instantiates specialization (2), which uses `Tuple1<long, double>` as a member, which in turn uses the partial specialization (3) for member `rest` of type `Tuple1<double>`.

The disadvantage of the design above is that it requires similar code in the `Tuple1<T>` partial specialization and the general definition, leading to a subtle form of code duplication. This may not seem very problematic, but a good tuple API has a considerable amount of scaffolding; for example, `std::tuple` has 25 member functions.

Let's address `Tuple1`'s problem by using inheritance instead of composition, thus benefitting from an old and well-implemented C++ layout optimization known as the **empty base optimization**. When a base of a class has no state, that base is allowed, under certain circumstances, to occupy no room at all in the derived class. Let's design a `Tuple2` variadic class template that takes advantage of the empty base optimization:

```
template <typename... Ts>
class Tuple2;                // incomplete declaration

template <>
class Tuple2<>               // specialization for zero elements
{ /*...*/ };

template <typename T, typename... Ts>
class Tuple2<T, Ts...>       // specialization for one or more elements
    : public Tuple2<Ts...>   // recurses in inheritance
{
    T first;
    //...
};
```

If we assess the size of `Tuple2<int>` with virtually any contemporary compiler, it is the same as `sizeof(int)`, so the base does not, in fact, add to the size of the complete object. One awkwardness with `Tuple2` is that with most compilers the types specified appear in the memory layout in reverse order; for example, in an object of type `Tuple1<int, int, float, std::string>`, the string would be the first member in the layout, followed by the `float` and then by the two `int`s. (Compilers do have some freedom

[15] On Clang 11.0 and GCC 7.5, `sizeof(T1)` is 8, twice the size of an `int`.

in defining layout, but most of today's compilers simply place bases first in their order, followed by members in the order of their declarations.)

To ensure a more intuitive layout, let's define `Tuple3` that uses an additional **`struct`** to hold individual elements, which `Tuple3` inherits (by means of the seldom-used **`protected`** inheritance) before recursing:

```
template <typename T>
struct Element3      // element holder
{
    T value;         // no other data or member functions
};

template <typename... Ts>
class Tuple3;        // declaration to introduce the class template

template <>
class Tuple3<>       // specialization for zero elements
{ /*...*/ };

template <typename T, typename... Ts>  // one or more types
class Tuple3<T, Ts...>                 // one or more elements
    : public Element3<T>               // first in layout
    , public Tuple3<Ts...>             // recurse to complete layout
{ /*...*/ };
```

This is close to what we need, but there is one additional problem to address: The instantiation `Tuple3<int, int>` will attempt to inherit `Element<int>` twice, which is not allowed. One way to address this issue is by passing a so-called cookie to `Element`, an additional template parameter that uniquely tags each `Element` differently. We choose `size_t` for the cookie type:

```
template <typename T, std::size_t cookie>
struct Element4      // Element holder also takes a cookie so the same element
                     // is not inherited twice; cookie is actually not used.
{
    T value;
};
```

The `Tuple4` class instantiates `Element` with a decreasing value of `cookie` for each successive element:

```
template <typename... Ts>
class Tuple4;                            // (0) incomplete declaration

template <>
class Tuple4<>                           // (1) specialization for no elements
{ /*...*/ };

template <typename T, typename... Ts>
class Tuple4<T, Ts...>                   // (2) one or more elements
    : public Element4<T, sizeof...(Ts)>  // first in layout, count is cookie
```

```
    , public Tuple4<Ts...>              // recurse to complete layout
{ /*...*/ };
```

To see how it all works, consider the instantiation `Tuple4<int, int, char>`, which matches specialization (2) with `T=int` and `Ts=<int, char>`. Consequently `sizeof...(Ts)` — the number of elements in `Ts` — is 2. The specialization first inherits `Element<2, int>` and then `Tuple4<int, char>`. The latter, in turn, also uses specialization (2) with `T=int` and `Ts=<char>`, which inherits `Element<int, 1>` and `Tuple4<char>`. Finally, `Tuple4<char>` inherits `Element<char, 0>` and `Tuple4<>`, which kicks the first specialization into gear to terminate the recursion.

It follows that `Tuple4<int, int, char>` ultimately inherits (in this order) `Element<int, 2>`, `Element<int, 1>`, and `Element<char, 0>`. Most implementations of `std::tuple` are variations of the patterns illustrated by `Tuple1` through `Tuple4`.[16]

If `Element` didn't take a distinct number for each member of the product type, `Tuple4<int, int>` would not work because it would inherit `Element<int>` twice, which is illegal. With `cookie`, the instantiation works because it inherits the distinct types `Element<int, 1>` and `Element<int, 2>`.

The expanded templates for the above code example might look like this invalid but illustrative code:

```
class Tuple4<>
{ /*...*/ };
class Tuple4<char> : public Element<char, 0>, public Tuple4<>
{ /*...*/ };
class Tuple4<int, char> : public Element<int, 1>, public Tuple4<char>
{ /*...*/ };
class Tuple4<int, int, char> : public Element<int, 2>, public Tuple4<int, char>
{ /*...*/ };
```

The complete implementation of a tuple type would contain the usual constructors and assignment operators as well as a *projection function* that takes an index `i` as a compile-time parameter and returns a reference to the `i`th element of the tuple. Let's see how to implement this rather subtle function. To get it done, we first need a helper template that returns the `n`th type in a template parameter pack.

```
template <std::size_t n, typename T, typename... Ts>
struct NthType              // yields nth type in the sequence <T, Ts...>
{
    typedef typename NthType<n - 1, Ts...>::type
        type;               // recurse to smaller n
};

template <typename T, typename... Ts>
struct NthType<0, T, Ts...>  // base case, 0th type in <T, Ts...> is T
```

[16] `libstdc++`, the GNU C++ Standard Library, uses an inheritance-based scheme with increasing indexes (as opposed to `Tuple4` in which `cookie` values are decreasing). `libc++`, the LLVM Standard Library that ships with Clang, does not use inheritance for `std::tuple`, but its state implementation uses inheritance in conjunction with an increasing integral sequence. Microsoft's open-source STL (`?`) uses, at time of this writing, the approach taken by `Tuple2`.

```
{
    typedef T type;
};
```

`NthType` follows the now familiar pattern of recursive parameter pack handling. The first declaration introduces the recursive case. The specialization that follows handles the limit case `n == 0` to stop the recursion. It is easy to follow that, for example, `NthType<1, short, int, long>::type` is `int`.

We are now ready to define the function `get` such that `get<0>(x)` returns a reference to the first element of a `Tuple4` object called `x`.

```
template <std::size_t n, typename... Ts>  // n is the index. Ts is the tuple pack.
auto& get(Tuple4<Ts...>& x)               // top-level function
{
    typedef typename NthType<n, Ts...>::type
        ResultType;                       // Reference to this type is returned.
    typedef Element4<ResultType, sizeof...(Ts) - n - 1>
        ElementType;                      // element holding the value returned
    ElementType& r = x;                   // implicit conversion to get element
    return r.value;                       // Access the value from the Element.
}
```

Calculating the cookie `sizeof...(Ts) - n - 1` requires some finesse. Recall that the elements in a tuple come with cookies in the reverse order of their natural order, so the first element in a `Tuple4<Ts...>` has cookie `sizeof(Ts...) - 1` and the last element has cookie `0`. Therefore, when we compute the cookie of the `n`th element in the cookie, we use elementary algebra to get to the expression shown.

After all typing has been sorted out, the implementation itself is trivial; it fetches the appropriate `Element` base by means of an implicit cast and then returns its `value`. Let's put the code to test:

```
void test()
{
    Tuple4<int, double, std::string> value;
    get<0>(value) = 3;
    get<1>(value) = 2.718;
    get<2>(value) = "hello";
    assert(get<2>(value) == "hello");
}
```

The example also illustrates why `get` is best defined as a nonmember function: A member function would have been forced to use the awkward syntax `value.template get<0>() = 3` to disambiguate the use of `<` as a template instantiation as opposed to the less-than operator.

Variant types

variant-types

A *variant* type, sometimes called a *discriminated union*, is similar to a C++ `union` that keeps track of which of its elements is currently active and protects client code against unsafe uses. Variadic templates support a natural interface to express this design as a generic

library feature.[17] For example, a variant type such as `Variant<int, float, std::string>` would be able to hold exactly one **`int`** or one **`float`** or one `std::string` value. Client code can change the current type by assigning to the variant object an **`int`**, a **`float`**, or an `std::string` respectively.

To define a variant type, we need it to have enough storage to keep any of its possible values, plus a *discriminator* — typically a small integral that keeps the index of the currently stored type. For `Variant<int, float, std::string>`, the discriminator could be by convention 0 for **`int`**, 1 for **`float`**, and 2 for `std::string`.

We saw in the previous sections how to define data structures recursively for parameter packs, so let's try our hand at a variant layout in the `Variant1` design:

```
template <typename... Ts>                    // parameter pack
class Variant1                               // can hold any in parameter pack
{
    template <typename...>                   // union of all types in Ts
    union Store {};

    template <typename U, typename... Us>    // Specialize for >=1 types.
    union Store<U, Us...>
    {
        U head;                              // Lay out a U object.
        Store<Us...> tail;                   // all others at same address
    };

    Store<Ts...> d_data;                     // Store for current datum.
    unsigned int d_active;                   // index of active type in Ts

public:
    // ... (API goes here)
};
```

C-style **`union`**s can be templates, too, and variadic ones at that. We take advantage of this feature in `Variant1` to recursively define `Store<Ts...>` that stores each of the types in the parameter pack `Ts` at the same address. One important C++11 feature that conveys flexibility to the design above is the relaxation on the restrictions on types that can be stored in **`union`**s. In C++03, **`union`** members were not allowed to have non-trivial constructors or destructors. Starting with C++11, any type can be a member in a **`union`**; see Section 3.1."**`union`** '11" on page 678. Therefore, types such as `Variant1<std::string, std::map<int, int>>` work fine.

It is possible to define `d_data` in a more succinct manner as a fixed-size array of **`char`**. There are two challenges to address. First, the size of the array needs to be computed during compilation as the maximum of the sizes of all types in `Ts`. Second, the array needs to have sufficient alignment to store any of the types in `Ts`. Fortunately, both problems have simple solutions in idiomatic modern C++:

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

[17] C++17's Standard Library type `std::variant` provides a robust and comprehensive implementation of a variant type.

```
template <typename... Ts> class Variant2;  // introducing declaration

template <> class Variant2<>     // specialization for empty Ts
{ /*...*/ };

template <typename T, typename... Ts>
class Variant2<T,Ts...>          // specialization for one or more types
{
    enum : std::size_t { size = std::max({sizeof(T), sizeof(Ts)...}) };
        // std::max takes std::initializer_list and is constexpr in C++14

    alignas(T) alignas(Ts...)
    char d_data[size];           // payload
    unsigned int d_active;       // index of active type in Ts

public:
    // ... (API goes here)
};
```

The code above uses a new use of `std::max` — overload introduced in C++14 that takes an initializer list as parameter; see Section 2.1."`initializer_list`" on page 392. Another novelty is the use of `std::max` during compilation; see Section 2.1."**`constexpr` Functions**" on page 239. We apply `std::max` to **`sizeof`**`(T)` and the **`sizeof`**`(Ts)...` expansion, which results in a comma-separated list **`sizeof`**`(T)`, **`sizeof`**`(T0)`, **`sizeof`**`(T1)`, `....` (Note it is not the same expansion as **`sizeof`**`...(Ts)`, which would just return the number of elements in `Ts`).

In brief, `d_data` is an array of **`char`** as large as the maximum of the sizes of all of the types passed to `Variant2`. In addition, the **`alignas`** directives instructs the compiler to align `d_data` at the largest alignment of all types among `T` and all of `Ts`; see *Description — Expansion in an alignment specifier* on page 563 and Section 2.1."**`alignas`**" on page 158.

It is worth noting that both `Variant1` and `Variant2` are equally good from a layout perspective; in fact, even the implementation of their respective APIs are identical. The only use of `d_data` is to take its address and use it as a pointer to **`void`**, which the API casts appropriately.

The public interface ensures that when an element is stored in `d_data`, `d_active` will have the value of the index into the parameter pack `Ts...` that corresponds to that type. Hence, when a user attempts to retrieve a value from the variant, if the wrong type is requested, a runtime error will be reported.

Let's take a look at defining some relevant API functions — the default constructor, a value constructor, and the destructor:

```
    std::size_tstd::stringassertstd::runtime_error

#include <algorithm>  // std::max

template <typename... Ts>              // introducing declaration
class Variant;
```

```
template <> class Variant<>            // specialization for empty Ts
{ /*...*/ };

template <typename T, typename... Ts>  // specialization for 1 or more
class Variant<T,Ts...>
{
    enum : std::size_t { size = std::max({sizeof(T), sizeof(Ts)...}) };
        // compute payload size

    alignas(T) alignas(Ts...)
    char d_data[size];                 // approach in Variant1 fine too
    unsigned int d_active;             // index of active type in Ts

    template <typename U, typename... Us>
    friend U& get(Variant<Us...>& v);  // friend accessor

public:
    Variant();                         // default constructor

    template <typename U>              // value constructor
    Variant(U&&);                      // U must be among T, Ts.

    ~Variant();                        // destroy the current object

    // ...
};
```

The default constructor should put the object in a simple, meaningful state. A reasonable decision is to create (by default construction) the *first* object in `Ts`:

```
template <typename T, typename... Ts>
Variant<T, Ts...>::Variant()
{
    ::new(&d_data) T();  // default-constructed T at address of d_data
    d_active = 0;        // Set the active type to T, first in the list.
}
```

The default constructor uses **placement new** to create a default-constructed object at the address of `d_data`. The first element in the parameter pack is selected by means of partial specialization.

The value constructor is a bit more challenging because it needs to compute the appropriate index for `d_active` during compilation, for example as an **`enum`** value. To implement it, first we need a support metafunction that reports the index of a type in a template parameter pack. The first type is the sought type, followed by the types to be searched. If the first type is not among the others, an error is produced:

```
template <typename X, typename T, typename... Ts>  // Find X in T, Ts....
struct IndexOf                                     // primary definition
{
    enum : std::size_t { value = IndexOf<X, Ts...>::value + 1 };
```

```
};

template <typename X, typename... Ts>              // partial specialization 1
struct IndexOf<X, X, Ts...>                        // found X at front
{
    enum : std::size_t { value = 0 };              // found in position 0
};

template <typename X, typename... Ts>              // partial specialization 2
struct IndexOf<const X, X, Ts...>                  // found const X at front
{
    enum : std::size_t { value = 0 };              // also found in position 0
};
```

The `: size_t` syntax, new to C++11, specifies that the introduced anonymous **`enum`** has type `size_t` as a base; see Section 2.1."Underlying Type '11" on page 480. The class template `IndexOf` follows a simple recursive pattern. In the general case, the type `X` is different from the first type `T`, and `value` is computed recursively as a search through the tail of the list.

If the sought type is identical to the first in the list, partial specialization 1 kicks in; if the sought type is a **`const`** variant of the second, partial specialization 2 matches. (A complete implementation would also add a similar specialization for the **`volatile`** qualifier.) In either case, the recursion ends and the value `0` is popped up the compile-time recursion stack:

```
std::size_t i1 = IndexOf<int, int, long>::value;          // i1 is 0
std::size_t i2 = IndexOf<int, short, int, long>::value;   // i2 is 1
std::size_t i3 = IndexOf<const int, short, int>::value;   // i3 is 1
std::size_t i4 = IndexOf<int, float, double>::value;      // Error
```

If the type is not found in the pack at all, then the recursion will come to an end when `Ts` is empty and the recursion cannot find a specialization for only one type `T`, resulting in a compile-time error.

It is worth noting that `IndexOf` has an alternative implementation that uses `std::integral_constant`, a Standard Library facility introduced in C++11 that automates part of the `value` definition:

```
#include <type_traits>  // std::integral_constant
template <typename X, typename T, typename... Ts>  // general definition
struct IndexOf2
    : std::integral_constant<std::size_t, IndexOf<X, Ts...>::value + 1> {};

template <typename X, typename... Ts>
struct IndexOf2<X, X, Ts...>
    : std::integral_constant<std::size_t, 0u> {};  // partial specialization 1

template <typename X, typename... Ts>
struct IndexOf2<const X, X, Ts...>
    : std::integral_constant<std::size_t, 0u> {};  // partial specialization 2
```

The type `std::integral_constant<size_t, n>` defines a constant member named `value` of type `size_t` with value `n`, which simplifies to some extent the definition of `IndexOf` and clarifies its intent.

With this template in our toolbox, we are ready to implement `Variant`'s value constructor, using **perfect forwarding** to create a `Variant` holding an object of the given type, with a compile-time error if the specified type is not found in the parameter pack `Ts`.

There is one more detail to handle. By design, we require `Ts` to contain only unqualified (no **`const`** or **`volatile`**), nonreference types, but the value constructor may deduce its type parameter as a reference type (such as **`int&`**). In such cases, we need to extract **`int`** from **`int&`**. The Standard Library template `std::remove_reference_t` was designed exactly for this task — both `std::remove_reference_t<int>` and `std::remove_reference_t<int&>` are aliases for **`int`**:

```
template <typename T, typename... Ts>  // definition for partial specialization
template <typename UARG>               // value constructor
Variant<T, Ts...>::Variant(UARG&& xs)  // uses perfect forwarding
{
    typedef std::remove_reference_t<UARG> U;
        // Remove reference from UARG if any, e.g. transform int& into int.
    ::new(&d_data) U(std::forward<UARG>(xs));  // Construct object at address.
    d_active = IndexOf<U, T, Ts...>::value;    // This code fails if U not in Ts.
}
```

Now we get to construct a `Variant` given any value of one of its possible types:

```
Variant<float, double> v1(1.0F);  // v1 has type float and value 1.
Variant<float, double> v2(2.0);   // v2 has type double and value 2.
Variant<float, double> v3(1);     // Error, int is not among allowed types.
```

A more advanced `Variant` implementation could support implicit conversions during construction as long as there is no ambiguity. Such functionality is supported by `std::variant`.

Now that `Variant` knows the index of the active type in `Ts`, we can implement an accessor function that retrieves a reference to the active element, by a client that knows the type. For example, given a `Variant<short, int, long>` object named `v`, **`int&`** `get<int>(v)` should return a reference to the stored **`int`** if and only if the current value in `v` has indeed type **`int`**; otherwise, it throws an exception:

```
template <typename T, typename... Ts>            // T is the assumed type.
T& get(Variant<Ts...>& v)                        // Variant<Ts...> by ref
{
    if (v.d_active != IndexOf<T, Ts...>::value)  // Is the index correct?
        throw std::runtime_error("wrong type");  // If not, throw.
    void* p = &v.d_data;                         // If so, take store address
    return *static_cast<T*>(p);                  // and convert.
}
```

A overload of `get` that takes and returns **`const`** is defined analogously.

In spite of its simplicity, the `get` function (which needs to be a **`friend`** of `Variant` because it needs access to its **`private`** members) is safe and robust. If given a type not present in the `Variant`'s parameter pack, `IndexOf` fails to compile, and, consequently, `get` does not compile either. If the type is present in the pack but is not the current type stored in the `Variant`, an exception is thrown. If everything works well, the address of data is converted to a reference to the target type, in full confidence that the cast is safe to perform:

```
typedef Variant<long, double, std::string> Var;

Var x1(1L);                             // type long, value 1
Var x2(2.5);                            // type double, value 2.5
Var x3(std::string("hi"));              // type std::string, value "hi"

long y1(get<long>(x1));                 // OK, y1 is 1.
double y2(get<double>(x2));             // OK, y2 is 2.5.
std::string y3(get<std::string>(x3));   // OK, y3 contains "hi".
double y4(get<double>(x3));             // throws exception, wrong type
```

Writing `Variant`'s destructor is more difficult because, in that case, we need to produce the compile-time type of the active element from the *runtime* index, `d_active`. The language offers no built-in support for such an operation, so we must produce a library solution instead.

One idea is to use a linear search approach: Starting with the active index `d_active` and the entire parameter pack `Ts`, we reduce both successively until `d_active` becomes zero. At that point, we know that the dynamic type of the variant is the head of what's left of `Ts`, and we call the appropriate destructor. To implement such an algorithm, we define two overloaded functions, `destroyLinear`, that are friends of `Variant`:

```
template <unsigned int>
void destroyLinear(unsigned int, void*)      // terminates recursion
{ }

template <unsigned int i, typename T, typename... Ts>
void destroyLinear(unsigned int n, void* p)  // index and pointer to data
{
    if (n == i)
        static_cast<T*>(p)->~T();            // found, call destructor manually
    else
        destroyLinear<i + 1, Ts...>(n, p);   // "recurse" with list tail
}
```

The second overload employs the idiom (used in several places in this feature section) of stripping the first element from the parameter pack on each call using a named type parameter. If the runtime index is 0, then the destructor of `T`, the first type in the pack, is called against the pointer received. Otherwise, `destroyLinear` "recursively" calls a version of itself with the parameter pack reduced by 1 and the compile-time counter `i` correspondingly bumped. Note that "recursion" is not quite the correct term because the template instantiates a different function for each call.

The first overload simply terminates the recursion. It is never called in a correct program, but the compiler doesn't know that, so we need to provide a body for it.

`Variant`'s destructor ensures that the variant object is not corrupt and then calls `destroyLinear`, passing the entire pack `Ts...` as template argument and the current index and pointer to state as runtime arguments:

```
template <typename T, typename... Ts>  // definition for partial specialization
Variant<T, Ts...>::~Variant()          // linear lookup destructor
```

```
{
    assert((d_active < Variant<T,Ts...>::size));    // check invariant
    destroyLinear<0, T, Ts...>(d_active, &d_data);  // initiate destruction
}
```

When presented with a linear complexity algorithm, the natural reaction is to look for a similar solution with lower complexity, especially considering that the linear search will be performed at run time, not during compilation. In this case, we might try a *binary search* through the type parameter pack. In common usage, a variant does not have that many types, so this may be considered overkill, but there are two reasons why this problem deserves our attention. First, there are variant types in common use that have a large number of alternatives, such as the `VARIANT` type in the Windows operating system with around 50 options in its union, and some data exchange formats that can have hundreds of types. Second, destructors are of particular importance because they tend to be called intensively, so destructors' size and speed can often affect performance of programs using them.

Defining a binary search that is a mix of compile-time and runtime computations is challenging, particularly because parameter packs do not have indexed access. However, we can implement `destroyLog` with relative ease if we design the algorithm in a hybrid manner — linear search during compilation and binary search at runtime. We do so by defining `destroyLog` to take two integral template parameters, `skip` and `total`, that instruct the template which types in the pack it must look at. If `skip` is greater than 0, the algorithm does a linear search during compilation in the parameter pack. When `skip` is zero, we know we need to search `total` elements in the parameter pack, and we do so in a textbook binary search manner:

```
template <unsigned int, unsigned int>
void destroyLog(unsigned int n, void*) // no-op terminal function
{ }

template <unsigned int skip, unsigned int total, typename T, typename... Us>
void destroyLog(unsigned int n, void* p)
{
    enum : std::size_t { mid = total / 2 };
    if (skip > 0)
        destroyLog<skip - 1, total - 1, Us...>(n, p);
    else if (n == 0)
        static_cast<T*>(p)->~T();
    else if (n < mid)
        destroyLog<0, mid, T, Us...>(n, p);
    else
        destroyLog<mid, total, T, Us...>(n - mid, p);
}
```

There are quite a few moving parts in `destroyLog`, so let's take a look at each in turn. The first overload is simple: Its role, just like before, is to stop the recursion. It takes two **`unsigned int`** parameters, both of which it ignores. The first overload will never be called or even linked.

The bulk of the work is carried by the second overload. First, it computes half of the total in constant `mid`.

Now, to understand how the four branches of the **if**/**else** cascade work, it's important to distinguish the first condition from all others; the `skip > 0` test is essentially a compile-time test and does no computation at run time. All instantiations with a nonzero `skip` will simply recurse to a different instantiation and do nothing else. The compiler will in all likelihood inline the call and eliminate all other code in the function.

Why do we subtract `(skip > 0)` from `skip` and `total` instead of simply subtracting `1`? If we used `skip - 1`, when `skip` reaches `0`, the compiler would attempt to instantiate `destroyLog` with `-1` (even if it never calls that instantiation), which translates into a very large **unsigned**, causing a runaway of further instantiations. The way the condition is written avoids this situation.

All other tests are performed during run time. The second test `n == 0` checks for a match, and, if the test passes, the appropriate destructor is called manually. This is in keeping with how `destroyLinear` works.

The rest of the function is the engine that accelerates `destroyLog` in comparison to `destroyLinear`. If `n` is less than `mid`, then there's no need to consider the upper elements of the pack at all; therefore, `destroyLog` is instantiated with `mid` as the total elements to search. All other arguments — both of the template and runtime kind — are the same. Otherwise, on the contrary, there's no need to look at the first `mid` elements of the pack, so `destroyLog` is instantiated accordingly.

All in all, the `destroyLog` template compactly encodes a linear search during compilation and simultaneously a binary search during run time. As expected, contemporary compilers eliminate dead code entirely and generate code virtually identical to code competently handwritten in a hardcoded approach.[18]

Of course, once we have a logarithmic implementation, we immediately wonder if we can do better, perhaps even a constant time lookup. That brings us to an application of variadic templates in a braced initialization, as described in *Description — Expansion in a function call argument list or a braced initializer list* on page 555. We use braced initialization to generate a table of function pointers, each pointing to a different instantiation of the same function template. For the table to work, each function must have exactly the same signature. For the case of invoking a destructor, we will want to supply the object to be destroyed by using **void*** just like we did with `destroyLinear` and `destroyLog`. We can then write a function template taking a non-deduced type parameter to carry the necessary type information for the function implementation to cast the **void*** pointer to the necessary type:

```
template <typename T>
static void destroyElement(void *p)
{
    static_cast<T*>(p)->~T();
}
```

With this simple function template, we can populate a static array of function pointers by initializing an array of unknown bound (that will implicitly deduce the correct size) with a braced list produced by a pack expansion, taking the address of the `destroyElement` function template instantiated with the types in the pack. Once we have the array of de-

[18] The corresponding code has been tested with Clang 11.0.1 and GCC 10.2 at optimization level `-O3`.

structor functions, matching the expected order of the runtime index `d_active`, we can simply invoke the function pointer at the current index to invoke the correct destructor for the currently active element.

```
template <typename... Ts>
void destroyCtTime(unsigned int n, void *p)  // same signature
{
    typedef void(*destructor)(void *);       // simplify definition
    static const destructor dt[] =           // array of function pointers
        { &destroyElement<Ts>... };          // initialize with pack expansion
    dt[n](p);                                // call appropriate destructor
}
```

Note that this constant-time lookup is also the simplest of the three forms presented since it leans more heavily on integrating variadic pack expansion with other language features, in this case braced array initialization.

It may appear that `destroyCtTime` is the best of the lot: It runs in constant time, it's small, it's simple, and it's easy to understand. However, upon a closer look, `destroyCtTime` has serious performance disadvantages. First, each destructor call entails an indirect call, which is notoriously difficult to inline and optimize except for the most trivial cases.[19]

Second, it is often the case that many types involved in a `Variant` have trivial destructors that do not perform any work at all. The functions `destroyLinear` and `destroyLog` have a white-box approach that naturally leads to the inlining and subsequent elimination of such destructors, leading to a massive simplification of the ultimately generated code. In contrast, `destroyCtTime` cannot take advantage of such opportunities; even if some destructors do no work, they will still be hidden beyond an indirect call, which is paid in all cases.

There is, however, a way to combine the advantages of `destroyCtTime`, `destroyLinear`, and `destroyLog` by using a meta-algorithmic strategy called **algorithm selection**[20]: Choose the appropriate algorithm depending on the `Variant` instantiation. The characteristics of instantiation can be inferred by using **compile-time introspection**. The criteria for selecting the best of three algorithms can be fairly complex. For instantiations with only a few types, most of which have trivial destructors, `destroyLinear` is likely to work best; for moderately large parameter pack sizes, `destroyLog` will be the algorithm of choice; finally, for very large parameter pack sizes, `destroyCtTime` is best.

For a simple example, we can use the parameter pack size as a simple heuristic:

```
template <typename T, typename... Ts>  // definition for partial specialization
Variant<T, Ts...>::~Variant()          // improved destructor implementation
{
    assert((d_active < Variant<T,Ts...>::size));   // check invariant of variant
    void* p = &d_data;                             // d_data as void*
    if (Variant<T,Ts...>::size <= 4)
        destroyLinear<0u, T, Ts...>(d_active, p);  // choose linear algorithm
    else if (Variant<T,Ts...>::size <= 64)
```

[19] GCC 10.2 generates tables and indirect calls even for the most trivial uses. Clang 11.0.1 is able to optimize away indirect calls for locally defined `Variant` objects only if the function has no control flow affecting `Variant` values. Both compilers were tested at optimization level `-O3`.

[20] **?**

```
        destroyLog<0u, Variant<T,Ts...>::size,
                   Ts...>(d_active, p);           // choose log algorithm
    else
        destroyCtTime<T,Ts...>(d_active, p);      // choose O(1) algorithm
}
```

The constants 4 and 64 deciding the thresholds for choosing between algorithms are called **metaparameters** and are to be chosen experimentally. As mentioned, a more sophisticated implementation would eliminate from `Ts...` all types that have trivial destructors and focus only on the types that require destructor calls. Distinguishing between trivial and non-trivial destructors is possible with the help of the Standard Library introspection primitive `std::is_trivially_destructible` introduced in C++11.

advanced-traits

Advanced traits

The use of **`template template`** parameters with variadic arguments allows us to create partial template specializations that match template instances with an arbitrary number of type parameters. This allows the definition of traits that were not possible with prevariadics technology.

For example, consider the family of **smart pointer** templates. A smart pointer type virtually always is instantiated from a template having the pointed-to type as its first parameter:

```
template <typename T>
struct SmartPtr1
{
    typedef T value_type;
    T& operator*() const;
    T* operator->() const;
    // ...
};
```

A more sophisticated smart pointer might take one or more additional template parameters, such as a deletion policy:

```
template <typename T, typename Deleter>
struct SmartPtr2
{
    T& operator*() const;
    T* operator->() const;
    // ...
};
```

`SmartPtr2` still takes a value type as its first template parameter but also takes a `Deleter` functor for destroying the pointed-to object. (The Standard Library smart pointer `std::unique_ptr` added with C++11 also takes a deleter parameter). Note that the author of `SmartPtr2` did not add a nested `value_type`, yet the human reader can easily deduce that `SmartPtr2`'s value type is `T`.

Now, we aim to define a traits class template that, given an arbitrary pointer-like type such as `SmartPtr1<int>` or `SmartPtr2<double, MyDeleter>`, can deduce the value type

pointed-to by the pointer (in our examples, as **int** and **double**, respectively). Additionally, our traits class should allow us to "rebind" the pointer-like type, yielding a new pointer-like type with a different value type. Such an operation is useful, for example, when we want to use the same smart pointer facility as another library, but with our own types.

Suppose, for example, a library defines a type `Widget`:

```
class Widget           // third-party class definition
{ /*...*/ };
```

Furthermore, the same library defines type `WidgetPtr` that behaves like a pointer to a `Widget` type but could be (depending on the library version, debug versus release builds, and so on) either `SmartPtr1` or `SmartPtr2`:

```
class FastDeleter     // policy for performing minimal checking on SmartPtr2
{ /*...*/ };

class CheckedDeleter        // policy for performing maximal checking on SmartPtr2
{ /*...*/ };

#if !defined(DBG_LEVEL)
typedef SmartPtr1<Widget>
    WidgetPtr;                              // release mode, fastest
#elif DBG_LEVEL >= 2
typedef SmartPtr2<Widget, CheckedDeleter>
    WidgetPtr;                              // safe smart pointer for debugging
#else
typedef SmartPtr2<Widget, FastDeleter>
    WidgetPtr;                              // safety/speed compromise
#endif
```

In debug mode (`DBG_LEVEL` defined), the library uses a `SmartPtr2` smart pointer that does additional checking around, for example, dereference and deallocation. There are two possible debugging levels, one with stringent checks and one that cuts a trade-off between safety and speed. In release mode, the library wants to run at full speed so it uses `SmartPtr1`. User code simply uses `WidgetPtr` transparently because the interfaces of the types are very similar.

On the client side, we'd like to define a `GadgetPtr` type that behaves like a pointer to our own type `Gadget` but automatically adjusts to use the same smart pointer underpinnings, if any, that `WidgetPtr` is using. However, we don't have control over `DBG_LEVEL` or over the code introducing `WidgetPtr`. The strategies used by the definition of `WidgetPtr` may change across releases. How can we robustly figure out what kind of pointer — smart or not — `WidgetPtr` is representing?

Let's begin by declaring a primary class template with no body, and then specializing it for native pointer types:

```
template <typename Ptr>
struct PointerTraits;       // incomplete declaration

template <typename T>
struct PointerTraits<T*>   // partial specialization for all raw pointers
```

```
{
    typedef T value_type;  // normalized alias for T
    template <typename U>
    using rebind = U*;     // rebind<U> is an alias for U*
};
```

The new **using** syntax has been introduced in C++11 as a generalized **typedef**; see Section 1.1."**using** Aliases" on page 121. `PointerTraits` provides a basic traits API. For any built-in pointer type, `P`, `PointerTraits<P>::value_type` resolves to whatever type `P` points to. Moreover, for some other type, `X`, `PointerTraits<P>::rebind<X>` is an alias for `X*`, i.e., it propagates the information that `P` is a built-in pointer to `X`:

```
#include <type_traits>  // std::is_same

static_assert(std::is_same<int, PointerTraits<int*>::value_type>::value, "");
static_assert(std::is_same<double*,
                           PointerTraits<int*>::rebind<double>>::value, "");
```

`PointerTraits` has a nested type, `value_type`, and a nested alias template, `rebind`. The first **static_assert** shows that, when `Ptr` is **int***, `value_type` is **int**. The second **static_assert** shows that, when `Ptr` is **int***, `rebind<double>` is **double***. In other words, `PointerTraits` can determine the pointed-to type for a raw pointer and provides a facility for generating a raw pointer type pointing to a *different* value type.

For now, however, `PointerTraits` is not defined for any type that is not a built-in pointer. For `PointerTraits` to work with an arbitrary smart pointer class (such as `SmartPtr1` and `SmartPtr2` above but also `std::shared_ptr`, `std::unique_ptr`, and `std::weak_ptr`), we must partially specialize it for a template instantiation, `PtrLike<T, X...>`, where `T` is assumed to be the value type and `X...` is a parameter pack of zero or more additional type parameters to `PtrLike`:

```
template <
    template <typename, typename...> class PtrLike,
    typename T, typename... X>
struct PointerTraits<PtrLike<T, X...>>  // partial specialization for template
{
    using value_type = T;               // extract pointee type
    template <typename U>
    using rebind = PtrLike<U, X...>;    // rebind to some other type U
};
```

This partial specialization will produce the correct result for any pointer-like class template that takes one or more type template parameters, where the first parameter is the value type of the pointer. First, it correctly deduces the `value_type` from the first argument to the template:

```
typedef SmartPtr2<Widget, CheckedDeleter>  WP1;  // fully checked
typedef SmartPtr2<Widget, FastDeleter>     WP2;  // minimally checked

static_assert(std::is_same<
    PointerTraits<WP1>::value_type,  // Fetch the pointee type of WP1.
```

```
    Widget>::value, "");             // should be Widget

static_assert(std::is_same<
    PointerTraits<WP1>::value_type,  // Fetch the pointee type of WP2.
    Widget>::value, "");             // should also be Widget
```

Second, `rebind` is able to reinstantiate the original template, `SmartPtr2`, with another first argument but the same second argument:

```
class Gadget { /*...*/ };

static_assert(std::is_same<
    PointerTraits<WP1>::rebind<Gadget>,  // Rebind WP1 to Gadget.
    SmartPtr2<Gadget, CheckedDeleter>>   // fully checked, just like WP1
::value, "");

static_assert(std::is_same<
    PointerTraits<WP2>::rebind<Gadget>,  // Rebind WP2 to Gadget.
    SmartPtr2<Gadget, FastDeleter>>      // minimally checked, like WP2
::value, "");
```

The Standard Library facility `std::pointer_traits`, introduced with C++11, is a superset of our `PointerTraits` example.

Potential Pitfalls

ial-pitfalls-variadic

Accidental use of C-style ellipsis

e-of-c-style-ellipsis

Inside the function parameters declaration, `...` can be used only in conjunction with a template parameter pack. However, there is an ancient use of `...` in conjunction with C-style variadic functions such as `printf`. That use can cause confusion. Say we set out to declare a simple variadic function, `process`, that takes any number of arguments by pointer:

```
class Widget;                    // declaration of some user-defined type

template <typename... Widgets>   // parameter pack named Widgets
int process(Widget*...);         // meant as a pack expansion, but is it?
```

The author meant to declare `process` as a variadic function taking any number of pointers to objects. However, instead of `Widgets*...`, the author mistakenly typed `Widget*...`. This typo took the declaration into a completely different place: It is now a C-style variadic function in the same category as `printf`. Recall the `printf` declaration in the C Standard Library:

```
int printf(const char* format, ...);
```

The comma and the parameter name are optional in C and C++, so omitting both leads to an equivalent declaration:

```
int printf(const char*...);
```

Comparing `process` (with the typo in tow) with `printf` makes it clear that `process` is a C-style variadic function. Runtime errors of any consequence are quite rare because the

expansion mechanisms are very different across the two kinds of variadics. However, the compile- and link-time diagnostics can be puzzling. In addition, if the variadic function ignores the arguments passed to it, calling it may even compile, but the call will use a different calling convention than that intended and assumed.

As an anecdote, a similar situation occurred during the review stage of this feature section. A simple misunderstanding caused a function to be declared inadvertently as a C-style variadic instead of C++ variadic template, leading to numerous indecipherable compile-time and link-time errors in testing that took many emails to figure out.

undiagnosed-errors

Undiagnosed errors

Description — Corner cases of function template argument matching on page 544 shows definitions of variadic template functions that are in error according to the C++ Standard yet pass compilation on contemporary compilers — that is, IFNDR. In certain cases, they can even be called. Such situations are most assuredly latent bugs:

```
template <typename... Ts, typename... Us, typename T>
int process(Ts..., Us..., T);
    // ill-formed declaration - Us must be empty in every possible call
int x = process<int, double>(1, 2.5, 3);
    // Ts=<int, double>, Us=<>, T=int
```

In virtually all cases, such code reflects a misplaced expectation; an always-empty parameter pack has no reason to exist in the first place.

the-number-of-arguments

Compiler limits on the number of arguments

The C++ Standard recommends that compilers support at least 1024 arguments in a variadic template instantiation. Although this limit seems very generous, real-world code may push against it, especially in conjunction with generated code or combinatorial uses.

This limit may lead to a lack of portability in production — for example, code that works with one compiler but fails with another. Suppose, for example, we define `Variant` that carries all possible types that can be serialized in a large application:

```
typedef Variant<
    char,
    signed char,
    unsigned char,
    short,
    unsigned short
    // ... (more built-in and user-defined types)
>
WireData;
```

We release this code to production and then, at some later date, clients find it fails to build on some platforms, leading to a need to reengineer the entire solution to provide full cross-platform support.

annoyances-variadic

Annoyances

unusable-functions

Unusable functions

Before variadics, any properly defined template function could be called by using explicit template argument specification, type deduction, or a combination thereof. Now it is possible to define variadic function templates that pass compilation but are impossible to call (either by using explicit instantiation, argument deduction, or both). This may cause confusion and frustration. Consider, for example, a few function templates, none of which take any function parameters:

```
template <typename T>                  // template with one parameter
int f1();

template <typename T, typename U>      // template with two parameters
int f2();

template <typename... Ts>              // pack followed by type parameter
int f3();

template <typename T, typename... Ts>  // parameter pack at the end
int f4();

template <typename... Ts, typename T>  // pack followed by type parameter
int f5();
```

The first four functions can be called by explicitly specifying their template arguments:

```
int a1 = f1<int>();              // T=int
int a2 = f2<int, long>();        // T=int, U=long
int a3 = f3<char, int, long>();  // Ts=<char, int, long>
int a4 = f4<char, int, long>();  // T=char, Ts=<int, long>
```

However, there is no way to call `f5` because there is no way to specify `T`:

```
int a5a = f5();             // Error, cannot infer type argument for T
int a5b = f5<int>();        // Error, cannot infer type argument for T
int a5c = f5<int, long>();  // Error, cannot infer type argument for T
```

Recall that by the Rule of Greedy Matching (see *Description — The Rule of Greedy Matching* on page 540), `Ts` will match all of the template arguments passed to `f5`, so `T` is starved. Such uncallable functions are IFNDR. There are several other variations that render variadic function templates uncallable and therefore IFNDR. However, most contemporary compilers do allow compilation of `f5`.[21] If other overloads of such an uncallable function are present, the matter may be interpreted as the wrong overload being called.

on-expansion-contexts

Limitations on expansion contexts

As discussed in *Description — Pack expansion* on page 551 and in the sections that follow it, expansion contexts are prescriptive. The standard enumerates all expansion contexts

[21] The corresponding code has been tested with Clang 11.0 and GCC 7.5.

exhaustively: There is no other place in a C++ program where a parameter pack is allowed to occur, even if it seems syntactically and semantically correct.

For example, consider a variadic function template, `bump1`, that attempts to expand and increment each of the arguments in its parameter pack:

```
template <typename... Ts>
void bump1(Ts&... vs)  // some variadic function template
{
    ++vs...;           // Error, can't expand parameter pack at statement level
}
```

Attempting to expand a parameter pack at the statement level is simply not among the allowed expansion contexts; hence, the example function body above fails to compile.

Such limitations can be worked around by artificially creating an expansion context. For example, we can achieve our goal by replacing the erroneous line in `bump1` (above) with one in `bump2` (below) that, say, creates a local class with a constructor that takes `Ts&...` as parameters:

```
template <typename... Ts>
void bump2(Ts&... vs)
{
    struct Local          // local struct
    {
        Local(Ts&...) {}  // constructor takes each of Ts by reference
    }                     // no semicolon here, will create an object
    local(++vs...);       // OK, expansion allowed in constructor call
}
```

The code above creates a local **`struct`** called `Local` with a constructor and immediately constructs an object of that type called `local`. Expansion is allowed inside the argument list for the constructor call, which makes the code work.

A possibility to achieve the same effect with terser code is to create a lambda expression, `[](Ts&...){}` and then immediately call it with the expansion `++vs...` as its arguments:

```
template <typename... Ts>
void bump3(Ts&... vs)          // some variadic function template
{
     ([](Ts&...){})(++vs...);  // OK, pack expansion allowed in lambda call
}
```

The function above works, albeit awkwardly, by making use of another feature of C++11 that essentially allows us to define an anonymous function *in situ* and then invoke it; see Section 2.1."Lambdas" on page 393 [22]

Parameter packs cannot be used unexpanded

nnot-be-used-unexpanded

As discussed in *Description — Pack expansion* on page 551, the name of a parameter pack cannot appear on its own in a correct C++ program; the only way to use a parameter pack

[22] C++17 adds *fold expressions* that allow easy pack expansion at expression and statement level. The semantics desired for the example shown would be achieved with the syntax `(...++vs);`.

is as part of an expansion by using `...` or **`sizeof`**. Such behavior is unlike types, template names, or values.

It is impossible to pass parameter packs around or to give them alternative names (as is possible with types by means of **`typedef`** and **`using`** and with values by means of references). Consequently, it is also impossible to define them as "return" values for metafunctions following conventions such as `::type` and `::value` that are commonly used in the `<type_traits>` standard header.

Consider, for example, sorting a type parameter pack by size. This simple task is not possible without a few helper types because there is no way to return the sorted pack. One necessary helper would be a typelist:

```
template <typename...> struct Typelist { };
```

With this helper type in hand, it is possible to encapsulate parameter packs, give them alternate names, and so on — in short, give parameter packs the same maneuverability that C++ types have:

```
typedef Typelist<short, int, long, float, double, long double> Numbers;
    // can be used to give a pack an alternate name

template <typename L>
struct SortBySize
{
    using type = Typelist< /*...*/ >;  // computed sorted by size version of
                                       // the Typelist L
};

typedef SortBySize<Numbers>::type SortedNumbers;
    // can be used to "return" a pack from a metafunction
```

Currently no `Typelist` facility has been standardized. An active proposal[23] introduces `parameter_pack` along the same lines as `Typelist` above. Meanwhile compiler vendors have attempted to work around the problem in nonstandard ways.[24] A related proposal[25] defines `std::bases` and `std::direct_bases` but has, at the time of writing, been rejected.

Expansion is rigid and requires verbose support code

-verbose-support-code

There are only two syntactic constructs that apply to parameter packs: **`sizeof...`** and expansion via `...`. The latter underlies virtually all treatment of variadics and, as discussed, requires handwritten support classes or functions as scaffolding toward building a somewhat involved recursion-based pattern.

There is no expansion in an expression context, so it is not possible to write functions such as `print` in a concise, single-definition manner; see *Use Cases — Generic variadic*

[23] **?**

[24] GNU defines the nonstandard primitives `std::tr2::__direct_bases` and `std::tr2::__bases`. The first yields a list of all direct bases of a given class, and the second yields the transitive closure of all bases of a class, including the indirect ones. To make these artifacts possible, GNU defines and uses a helper `__reflection_typelist` class template similar to `Typelist` above.

[25] **[AUs: url please, so I can cite this. Response: http://wg21.link/N2965**

functions on page 567. In particular, expressions are not expansion contexts so the following code would not work:

```
#include <iostream>  // std::cout, std::ostream, std::endl

template <typename T, typename... Ts>
std::ostream& print(const T& v, const Ts&... vs)
{
    std::cout << vs...;             // Error, invalid expansion
    return std::cout << std::endl;
}
```

r-search-for-everything

Linear search for everything

One common issue with parameter packs is the difficulty of accessing elements in an indexed manner. Getting to the nth element of a pack is a linear search operation by necessity, which makes certain uses awkward and potentially time-consuming during compilation. Refer to the implementation of `destroyLog` in *Use Cases — Variant types* on page 575 as an example.

see-also

See Also

- "Braced Init" (§2.1, p. 198) ♦ illustrates one of the expansion contexts for function parameter packs.
- "Forwarding References" (§2.1, p. 351) ♦ describes a feature used in conjunction with variadics to achieve perfect forwarding.
- "Lambdas" (§2.1, p. 393) ♦ describes a feature that allows expansion both in capture list and in the arguments list.

further-reading

Further Reading

- "F.21: To return multiple"out" values, prefer returning a struct or tuple," **?**
- **?**

sec-conditional-cpp14

Relaxed Restrictions on `constexpr` Functions

-constexpr-restrictions

C++14 lifts restrictions regarding use of many language features in the body of a **constexpr** function (see Section 2.1."**constexpr** Functions" on page 239).

Description

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, 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 597.

The C++11 **static_assert** feature (see Section 1.1."**static_assert**" on page 103) 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 not permitted:

```
constexpr void no_op() { }  // Error in C++11; OK in C++14
```

Experience gained from the release and subsequent real-world use of C++11 emboldened the standard committee to lift most of these (now seemingly arbitrary) restrictions for C++14, allowing use of (nearly) *all* language constructs in the body of a **constexpr** function. In C++14, familiar non-expression-based control-flow constructs, such as **if** statements and **while** loops, are also available, as are modifiable local variables and assignment operations:

```
constexpr int fact14(int x)
{
    if (x <= 2)        // Error in C++11; OK in C++14
    {
        return 1;
    }
```

```
    int temp = x - 1;  // Error in C++11; OK in C++14
    return x * fact14(temp);
}
```

Some useful features remain disallowed in C++14; most notably, any form of dynamic allocation is not permitted, thereby preventing the use of common standard container types, such as `std::string` and `std::vector`[1]:

1. **`asm`** declarations
2. **`goto`** statements
3. Statements with labels other than **`case`** and **`default`**
4. **`try`** blocks
5. Definitions of variables
   (a) of other than a **literal type** (i.e., fully processable at compile time)
   (b) decorated with either **`static`** or **`thread_local`**
   (c) left uninitialized

The restrictions on what can appear in the body of a **`constexpr`** that remain in C++14 are reiterated here in codified form[2]:

```
template <typename T>
constexpr void f()
{
try {                   // Error, try outside body isn't allowed (until C++20).
    std::ifstream is;   // Error, objects of *non-literal* types aren't allowed.
    int x;              // Error, uninitialized vars. disallowed (until C++20)
    static int y = 0;   // Error, static variables are disallowed.
    thread_local T t;   // Error, thread_local variables are disallowed.
    try{}catch(...){}   // Error, try/catch disallowed (until C++20)
    if (x) goto here;   // Error, goto statements are disallowed.
    []{};               // Error, lambda expressions are disallowed (until C++17).
here: ;                 // Error, labels (except case/default) aren't allowed.
    asm("mov %r0");     // Error, asm directives are disallowed.
} catch(...) { }        // Error, try outside body disallowed (until C++20)
}
```

[1] 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.

-cases-relaxedconstexpr

ve-constexpr-algorithms

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 manner. Consider, as a familiar example, a naive[3] C++11-compliant **constexpr** implementation of a function, `fib11`, returning the nth 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.

[3]For a more efficient (yet less intuitive) C++11 algorithm, see *Appendix — Optimized C++11 example algorithms* on page 603, *Recursive Fibonacci* on page 603.**AUs: The Appendix may need to be slightly restructured. We have no current method for an xref to this depth of subsectioning.**

[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 Section 1.1."**long long**" on page 78.

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 run-time evaluation, thereby requiring programmers to provide and maintain *two* separate 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 nth 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;
}
```

[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?
```

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.

aprogramming-algorithms

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 Section 2.1."Variadic Templates" on page 519) that can be instantiated using a variable-length sequence of arbitrary C++ types[9]:

```
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, long, double> listOfThreeIntLongDouble;
```

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**

[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>
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.

edconstexpr-countcode

class template specialization[10] to satisfy the single-return-statement requirements[11]:

```
#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.
```

[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:

```
    std::integral_constantstd::is_same

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

// ...
```

```
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 Section 2.1."Variadic Templates" on page 519), 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:

```
    std::integral_constantstd::is_same
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
    {
        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

[12] For a more efficient C++11 version of `Count`, see *Appendix — Optimized C++11 example algorithms* on page 603, **constexpr** *type list* `Count` *algorithm* on page 603.**AUs: another nested appendix xref**

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

Potential Pitfalls

None so far

annoyances

Annoyances

None so far

see-also

See Also

- "**`constexpr`** Functions" (§2.1, p. 239) ♦ Conditionally safe C++11 feature that first introduced compile-time evaluations of functions.
- "**`constexpr`** Variables" (§2.1, p. 282) ♦ Conditionally safe C++11 feature that first introduced variables usable as constant expressions.
- "Variadic Templates" (§2.1, p. 519) ♦ Conditionally safe C++11 feature allowing templates to accept an arbitrary number of parameters.

further-reading

Further Reading

None so far

[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 600) — 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.

pendix-relaxedconstexpr

++11recursivefibonacci

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;
}
```

ypelist-count-algorithm

`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.

Lambdas Having a Templated Call Operator

genericlambda

placeholder text.....

Lambda-Capture Expressions

a-capture-expressions

Lambda-capture expressions enable **synthetization** (spontaneous implicit creation) of arbitrary data members within **closures** generated by lambda expressions (see Section 2.1."Lambdas" on page 393).

Description

description

In C++11, lambda expressions can capture variables in the surrounding scope either *by value* or *by reference*[1]:

```
void test()
{
    int i = 0;
    auto f0 = [i]{ };   // Create a copy of i in the closure named f0.
    auto f1 = [&i]{ };  // Store a reference to i in the 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 Section 2.1."**`auto`** Variables" on page 183) 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.
```

[1] We use the familiar (C++11) feature **`auto`** (see Section 2.1."**`auto`** Variables" on page 183) to deduce a closure's type since there is no way to name such a type explicitly.

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):

```
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 Section 1.1."**`decltype`**" on page 22) 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<decltype(arg), int>::value, "");
```

[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 612.

[3]Here we are using the (C++14) variable template (see Section 1.2."Variable Templates" on page 146) version of the standard `is_same` metafunction where `std::is_same<A, B>::value` is replaced with `std::is_same_v<A, B>`.

```
        // i in the interface (same as arg) refers to the int parameter.

    static_assert(std::is_same<decltype(i), char>::value, "");
        // i in the body refers to the char data member deduced at capture.
};
```

Notice that we have again used **`decltype`**, in conjunction with the standard `is_same` metafunction (which is **`true`** if and only if its two arguments are the same C++ type). This time, we're using **`decltype`** to demonstrate that the type (**`int`**), extracted from the local variable `i` within the declared-interface portion of `fC`, is distinct from the type (**`char`**) extracted from the `i` within `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]

[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
```

use-cases-lambdacapture

-objects-into-a-closure

Use Cases

Moving (as opposed to copying) objects into a closure

Lambda-capture expressions can be used to *move* (see Section 2.1."*rvalue* References" on page 479) 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**:

`std::unique_ptr`

```
ThreadPool::Handle processDatasetAsync(std::unique_ptr<Dataset> dataset)
{
    return getThreadPool().enqueueTask([data = std::move(dataset)]
    {
        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.

[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:

```
#include <utility>  // std::move
#include <memory>   // std::unique_ptr

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.

e-state-for-a-closure

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]:

`std::cout`

```
void listen()
{
    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.

isting-const-variable

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`**:

`std::is_same`

```
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<decltype(i), int>::value, "");
        static_assert(std::is_same<decltype(ci), const int>::value, "");
    };

    auto lm = [i, ci]() mutable       // Decorating with mutable has no effect.
    {
        static_assert(std::is_same<decltype(i), int>::value, "");
        static_assert(std::is_same<decltype(ci), const int>::value, "");
    };
}
```

[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.

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:

```
void performAlgorithmComparison2()
{
    // ...

    const Puzzle puzzle = generateRandomSudokuPuzzle();

    auto task0 = getThreadPool().enqueueTask([p = puzzle]() mutable
    {
        solve<NaiveAlgorithm>(p);  // OK, p is now modifiable.
        return p;
    });

    // ...
}
```

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 613.

itfalls-lambdacapture

Potential Pitfalls

t-(never-a-reference)

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 Section 2.1."Forwarding References" on page 351) into a closure:

```
#include <utility>  // std::forward

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:

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

template <typename T>
void f(T&& x)
{
    auto lambda = [y = std::tuple<T>(std::forward<T>(x))]
    {
```

[7]Note that both *by-copy* and *by-move* capture communicate **value** for **value-semantic types**.

```
        // ... (Use std::get<0>(y) instead of y in this lambda body.)
    };
}
```

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

nnoyances-lambdacapture

There's no easy way to synthesize a `const` data member

ize-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:

```
void test1()
{
    int k;
    [k = static_cast<const int>(k)]() mutable  // const is ignored
    {
        ++k;  // "OK" -- i.e., compiles anyway even though we don't want it to
    };
}

void test2()
{
    int k;
    [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 test3()
{
    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

able-callable-objects

Any lambda expression capturing a move-only object produces a closure type that is itself movable but *not* copyable:

std::unique_ptrstd::move

```
void f()
{
    std::unique_ptr<int> moo(new int);    // some move-only object
    auto la = [moo = std::move(moo)]{ };  // lambda that does move capture

    static_assert(false == std::is_copy_constructible<decltype(la)>::value, "");
    static_assert( true == std::is_move_constructible<decltype(la)>::value, "");
}
```

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

see-also

- "**auto** Variables" (§2.1, p. 183) ♦ offers a model with the same type deduction rules.
- "Braced Init" (§2.1, p. 198) ♦ illustrates one possible way of initializing the captures.
- "Forwarding References" (§2.1, p. 351) ♦ describes a feature that contributes to a source of misunderstanding of this feature.
- "Lambdas" (§2.1, p. 393) ♦ provides the needed background for understanding the feature in general.
- "*rvalue* References" (§2.1, p. 479) ♦ gives a full description of an important feature used in conjunction with movable types.

Further Reading

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 **?**.

[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 **?**.

Chapter 3

Unsafe Features

ch-unsafe
sec-unsafe-cpp11 Intro text should be here.

The [[carries_dependency]] Attribute

_dependency carries_dependency

The[[carries_dependency]] attribute provides a means to manually identify function parameters and **return** values as components of **data dependency chains** to enable (primarily across translation units) use of the lighter-weight **release-consume synchronization paradigm** as an optimization over the more conservative **release-acquire** paradigm.[1]

Description

description

C++11 ushered in support for multithreading by introducing a rigorously specified memory model. The Standard Library provides support for managing threads, including their execution, synchronization, and intercommunication. As a part of the new memory model, the Standard defines various **synchronization operations** that are classified as either *sequentially consistent*, *release*, *acquire*, *release-and-acquire*, or *consume* operations. These operations play a key role in making changes in data in one thread visible in another.

The modern C++ memory model describes two synchronization paradigms that are used to coordinate data flow among concurrent threads of execution. The current suite of supported synchronization paradigms comprises release-acquire and release-consume, although in practice release-consume is implemented in terms of release-acquire in all known implementations. In particular, the release-consume paradigm requires that the compiler be given fine-grained understanding of the **intra-thread dependencies** among the reads and writes within a program and relates those to atomic *release stores* and *consume loads* that happen concurrently across multiple threads of execution. Dependency chains in the release-consume synchronization paradigm specify which evaluations following the *consume load* are **ordered after** a corresponding *release store*.

The release-acquire paradigm

ease-acquire-paradigm

A *release* operation writes a value to a memory location, and an *acquire* operation reads a value from a memory location. Although many have referred to the release-acquire paradigm as *acquire-release*, the proper, standard, time-ordered nomenclature is *release-acquire*. In a release-acquire pair, the acquire operation reads the value written by the release operation, which means that all of the reads and writes to *any* memory location *before the release operation* happen before *all* of the reads and writes *after the acquire operation*. Note that this paradigm does *not* use dependency chains or the [[carries_dependency]] attribute. See *Use Cases — Producer-consumer programming pattern* on page 618 for a complete example that implements this paradigm.

Data dependency

data-dependency

In the current revisions of C++, **data dependency** is defined as existing whenever the output of one evaluation is used as the input of another. When one evaluation has a data dependency on another evaluation, the second evaluation is said to **carry dependency** to the other. The Standard Library function std::kill_dependency is also related and

[1]The authors would like to thank Michael Wong, Paul McKenney, and Maged Michael for reviewing and contributing to this feature section.

can be used to *break* a data dependency chain. Naturally the compiler must ensure that any evaluation that depends on another must not be started until the first evaluation is complete. A data dependency chain is formed when multiple evaluations carry dependency transitively; the output of one evaluation is used as the input of the next evaluation in the chain.

elease-consume-paradigm

The release-consume paradigm

Some systems use the read-copy-update (RCU) synchronization mechanism. This approach preserves the order of *loads* and *stores* that form in a data dependency chain, which is a sequence of *loads* and *stores* in which the input to one operation is an output of another. A compiler can use guaranteed order of loads and stores provided by the RCU synchronization mechanism for performance purposes by omitting certain **memory-fence instructions** that would otherwise be required to enforce the correct ordering. In such cases, however, ordering is guaranteed only between those operations making up the relevant data dependency chain. The C++ definition of data dependency is intended to mimic the data dependency on RCU systems. Note, however, that C++ currently defines data dependency in terms of evaluations, while RCU data dependency is defined in terms of loads and stores.

This optimization was intended to be available in C++ through use of a *release-consume* pair, which, as its name suggests, consists of a *release-store* operation and a *consume-load* operation. A *consume* operation is much like an *acquire* operation, except that it guarantees only the ordering of those evaluations in a data dependency chain, starting with the consume-load operation.

Note, however, that currently no known implementation is able to take advantage of the current C++ *consume* semantics; hence, all current compilers promote *consume* loads to *acquire* loads, effectively making the `[[carries_dependency]]` attribute redundant. Revisions to render this feature implementable and therefore usable are currently under consideration by the C++ Standards Committee. Prototypes for various approaches have been produced. When a usable feature with real implementations is delivered, it quite possibly will not work exactly as described in the examples here; see *Use Cases* on page 618.

_dependency]]-attribute

Using the `[[carries_dependency]]` attribute

Data dependency chains can and do propagate into and out of called functions. If one of these interoperating functions is in a separate translation unit, the compiler will have no way of seeing the dependency chain. In such cases, the user can apply the `[[carries_dependency]]` attribute to imbue the necessary information for the compiler to track the propagation of dependency chains into and out of functions across translation units, thus possibly avoiding unnecessary memory-fence instructions; see *Use Cases* on page 618. Note that the Standard Library function `std::kill_dependency` is also related and can be used to *break* a data dependency chain.

The `[[carries_dependency]]` attribute can be applied to a function declaration as a whole by placing it in front of the function declaration, in which case the attribute applies to the **`return`** value:

```
[[carries_dependency]] int* f();  // attribute applied to entire function f
```

In the example above, the `[[carries_dependency]]` attribute was applied to the declaration of function `f` to indicate that the **`return`** value carries a dependency out of the function. The compiler may now be able to avoid emitting a memory-fence instruction for the return value of `f`.

The `[[carries_dependency]]` attribute can also be applied to one or more of the function's parameter declarations by placing it immediately after the parameter name:

```
void g(int* input [[carries_dependency]]); // attribute applied to input
```

In the declaration of function `g` in the example above, the `[[carries_dependency]]` attribute is applied to the `input` parameter to indicate that a dependency is carried through that parameter into the function, which may obviate the compiler's having to emit an unnecessary memory-fence instruction for the `input` parameter; see Section 1.1."Attribute Syntax" on page 10.

In both cases, if a function or a parameter declaration specifies the `[[carries_dependency]]` attribute, the first declaration of that function shall specify that `[[carries_dependency]]` attribute. Similarly, if the first declaration of a function or one of its parameters specifies the `[[carries_dependency]]` attribute in one translation unit and the first declaration of the same function in another translation unit doesn't, the program is **ill formed, no diagnostic required (IFNDR)**.

It is important to note that while the `[[carries_dependency]]` attribute informs the compiler about the presence of a dependency chain, it does not itself create one. The dependency chain must be present in the implementation to have any effect on synchronization.

Use Cases

ses-carriesdependency

Producer-consumer programming pattern

r-programming-pattern

The popular producer-consumer programming pattern uses *release-acquire* pairs to synchronize between threads:

```
// my_shareddata.h
void initSharedData();
    // Initialize the shared data of my_shareddata.o to a well-known
    // aggregation of values.

void accessSharedData();
    // Confirm that the shared data of my_shareddata.o have been initialized
    // and have their expected values.


// my_shareddata.cpp
#include <my_shareddata.h>

#include <atomic>  // std::atomic, std::memory_order_release, and
                   // std::memory_order_acquire
#include <cassert> // standard C assert macro

struct S
{
    int    i;
    char   c;
    double d;
};

static S                data;     // static for insulation
static std::atomic<int> guard(0); // static for insulation

void initSharedData()
{
    data.i = 42;
    data.c = 'c';
    data.d = 5.0;

    guard.store(1, std::memory_order_release);
}

void accessSharedData()
{
    while(0 == guard.load(std::memory_order_acquire))
        /* empty */ ;

    assert(42  == data.i);
    assert('c' == data.c);
    assert(5.0 == data.d);
}
```

```
// my_app.cpp
#include <my_shareddata.h>
#include <thread>  // std::thread

int main()
{
    std::thread t2(accessSharedData);
    std::thread t1(  initSharedData);

    t1.join();
    t2.join();
}
```

When this *release-acquire synchronization paradigm* is used, the compiler must maintain the ordering of the statements to avoid breaking the *release-acquire* guarantee; the compiler will also need to insert memory-fence instructions to prevent the hardware from breaking this guarantee.

If we wanted to modify the example above to use *release-consume* semantics, we would somehow need to make the `assert` statements a part of the dependency chain on the `load` from the `guard` object. We can accomplish this because reading data through a pointer establishes a dependency chain between the reading of that pointer value and the reading of the referenced data. Since *release-consume* allows the developer to specify that data of concern, using that policy instead of *release-acquire* policy (in the code example above) allows the compiler to be more selective in its use of memory fences::

```
// my_shareddata.cpp (use _*consume, not *_acquire)
#include <my_shareddata.h>

#include <atomic>  // std::atomic, std::memory_order_release, and
                   // std::memory_order_consume (not *_acquire)
#include <cassert> // standard C assert macro

struct S
{
    /* definition not changed */
};

static S               data;          // static for insulation (as before)
static std::atomic<S*> guard(nullptr); // guards just one struct S.

void initSharedData()
{
    data.i = 42;   // as before
    data.c = 'c';  // as before
    data.d = 5.0;  // as before

    guard.store(&data, std::memory_order_release);  // Set &data, not 1.
}

void accessSharedData()
```

```
{
    S *sharedDataPtr = nullptr;

    // Load using *_consume, not *_acquire.
    while (nullptr == (sharedDataPtr = guard.load(std::memory_order_consume)))
        /* empty */ ;

    assert(&data == sharedDataPtr);

    assert(42  == sharedDataPtr->i);
    assert('c' == sharedDataPtr->c);
    assert(5.0 == sharedDataPtr->d);
}
```

Finally, if we want to start to refactor the work of the `my_sharedata` component into multiple functions across different translation units, we would want to carefully apply the `[[carries_dependency]]` attribute to the newly refactored functions, so calling into these functions might conceivably be better optimized:

```
// my_shareddataimpl.h

struct S
{
    int    i;
    char   c;
    double d;
};

[[carries_dependency]] S* getSharedDataPtr();
    // Return the address of the shared data in this translation unit.

void releaseSharedData(S *sharedDataPtr [[carries_dependency]]);
    // Release the shared data in this translation unit.  The behavior is
    // undefined unless getSharedDataPtr() == sharedDataPtr.

[[carries_dependency]] S* accessInitializedSharedData();
    // Return the address of the initialized shared data in this translation
    // unit.

void checkSharedDataValue(S* s [[carries_dependency]],
                          int    i,
                          char   c,
                          double d);
    // Confirm that data at the specified s has the specified i, c, and
    // d as constituent values.
```

```
// my_shareddataimpl.cpp

#include <my_shareddataimpl.h>

#include <cassert>
#include <atomic>

static S               data;           // static for insulation
static std::atomic<S*> guard(nullptr); // guards one struct S.

[[carries_dependency]] S* getSharedDataPtr()
{
    return &data;
}

void releaseSharedData(S *sharedDataPtr [[carries_dependency]])
{
    assert(&data == sharedDataPtr);

    guard.store(sharedDataPtr, std::memory_order_release);
}

[[carries_dependency]] S* accessInitializedSharedData()
{
    S *sharedDataPtr = nullptr;

    while (nullptr == (sharedDataPtr = guard.load(std::memory_order_consume)))
        /* empty */ ;

    assert(&data == sharedDataPtr);

    return sharedDataPtr;
}

void checkSharedDataValue(S*     s [[carries_dependency]],
                          int    i,
                          char   c,
                          double d)
{
    assert(i == s->i);
    assert(c == s->c);
    assert(d == s->d);
}
```

```
// my_shareddata.cpp (re-factored to use *impl)
#include <my_shareddata.h>
#include <my_shareddataimpl.h>

void initSharedData()
{
    S *sharedDataPtr = getSharedDataPtr();

    sharedDataPtr->i = 42;
    sharedDataPtr->c = 'c';
    sharedDataPtr->d = 5.0;

    releaseSharedData(sharedDataPtr);
}

void accessSharedData()
{
    S *sharedDataPtr = accessInitializedSharedData();
    checkSharedDataValue(sharedDataPtr, 42, 'c', 5.0);
}
```

potential-pitfalls

Potential Pitfalls

No practical use on current platforms

se-on-current-platforms

All known compilers promote *consume* loads to *acquire* loads, thus failing to omit superfluous memory-fence instructions. Developers writing code with the expectation that it will be run under the more efficient release-consume synchronization paradigm will find that their code will continue to work — as expected — under the more conservative release-acquire guarantees until such time as a theoretical, not-yet-existent compiler that properly supports the release-consume synchronization paradigm becomes widely available. In the meantime, applications that require the potential performance benefits of *consume* semantics typically make careful use of platform-specific functionality instead.[2]

Annoyances

annoyances

[2]Since C++17, the use of `memory_order_consume` has been explicitly discouraged after the acceptance of **?**. The specific note in the standard now says, "Prefer `memory_order_acquire`, which provides stronger guarantees than `memory_order_consume`. Implementations have found it infeasible to provide performance better than that of `memory_order_acquire`. Specification revisions are under consideration" (**?**, section 32.4 "Order and Consistency," paragraph 1.3, Note 1, p. 1346).

see-also

See Also

- "Attribute Syntax" (§1.1, p. 10) ♦ provides an in-depth discussion of how attributes pertain to C++ language entities.
- "`noreturn`" (§1.1, p. 83) ♦ offers an example of another *attribute* that *is* implemented ubiquitously.

further-reading

Further Reading

- ?
- ?

Preventing Overriding and Derivation

`final`

placeholder

d-friend-declarations

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.

iption-extendedfriend

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 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

use-cases

Safely declaring a previously declared type to be a friend

d-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 **?**.

```
};
```

s-a-customization-point

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 value-semantic type (VST)** that serves as a *handle* to a platform-specific object, such as a `Window` in a graphical application. (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.[2]) 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 635.

```
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] 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 635. 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**.

nforce-initialization

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 635.

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 632. 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

ngle-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 635.

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 635.

urring-template-pattern

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 636.

Potential Pitfalls

pitfalls-extendedfriend

Long-distance friendship

ong-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

see-also

- "**using** Aliases" (§1.1, p. 121) ♦ are among the beneficiaries of extended friend declarations.

Further Reading

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

endix:-crtp-use-cases
ring-template-pattern

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 30) the `InstanceCounter` class constructors to be private as well.

g-equality-using-crtp

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

```
    assert
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

lymorphism-using-crtp

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:

```
    std::coutdrawdraw
```

```
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

time-visitor-using-crtp

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, is 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 the end of the 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] which explains both its *curiously recurring* nature and nomenclature. 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] **?**

Transparently Nested Namespaces

inline-namespaces

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

Description

ption-inlinenamespace

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. To be precise, source files containing, alternately, `namespace n { inline namespace v { int d; } }` and `namespace n { namespace v { int d; } using namespace v; }`, will produce identical assembly.[2] Note that a **using** directive immediately following an **inline** namespace is superfluous; name lookup will always consider names in **inline** namespaces before those imported by a **using** directive. Such a directive can, however, be used to import the contents of an **inline** namespace to some other namespace, albeit only in the conventional, **using directive** sense; see *Annoyances — Only one namespace can contain any given* **inline** *namespace* on page 673.

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.

-in-enclosing-namespace

Loss of access to duplicate names in enclosing namespace

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

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

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

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

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

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

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

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

A conventional nested namespace behaves as one might expect:

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

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

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

If an **`inline`** namespace were used instead of a nested namespace followed by a **`using`** directive, however, the ability to recover by name the hidden entity in the enclosing namespace is lost. Unqualified name lookup considers the inline namespace set and the used namespace set simultaneously. Qualified name lookup first considers the **`inline`** namespace set and

then goes on to look into used namespaces. 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 659.

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

ne-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.

sted-inline-namespace

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

open-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:

std::is_same

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

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

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

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

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

In the code snippet above, no issue occurs with reopening `outer::inner::n` and no issue would have occurred with reopening `outer::m` but for the `inner` namespaces having been declared **`inline`**. When a new namespace declaration is encountered, a lookup determines if a matching namespace having that name appears anywhere in the **`inline`** *namespace set* of the current namespace. If the namespace is ambiguous, as is the case with `m` in the example above, one can get the surprising error shown.[3] If a matching namespace is found 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 670.

Use Cases

cases-inlinenamespace

Facilitating API migration

itating-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:

[3] Note that reopening already declared namespaces, such as `m` and `n` in the `inner` and `outer` example, is handled incorrectly on several popular platforms. Clang, for example, 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`.

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

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

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

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

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

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

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

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

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

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

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

```
    }

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

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

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

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

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

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

As suggested by the name `v1`, this namespace serves primarily as a mechanism to support library evolution through API and ABI versioning (see *Use Cases — Link-safe ABI versioning* on page 659 and *Use Cases — Build modes and ABI link safety* on page 663). 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 666.

Now suppose we decide to enhance `parselib` in a non–backwards-compatible manner, such that the signature of `parse` takes a second argument `size` of type `std::size_t` to allow

parsing of non–null-terminated strings and to reduce the risk of buffer overruns. Instead of unilaterally removing all support for the previous version in the new release, we can create a second namespace, `v2`, containing the new implementation and then, at some point, make `v2` the **`inline`** namespace instead of `v1`:

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

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

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

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

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

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

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

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

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

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

```
namespace parselib
{
    namespace v1  // specializations moved to nested namespace
    {
        template <>
        class Parser<MyClass>
        {
            // ...

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

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

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

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

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

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

Although new users would pick up the latest version automatically either way, existing users of `parselib` will have the option of converting immediately by making a few small syntactic changes or opting to remain with the original version for a while longer by making

all references to the library namespace refer explicitly to the desired version. If the library is released before the **inline** keyword is moved, early adopters will have the option of opting in by referring to `v2` explicitly until it becomes the default. Those who have no need for enhancements can achieve stability by referring to a particular version in perpetuity or until it is physically removed from the library source.

Although this same functionality can sometimes be realized without using **inline** namespaces (i.e., by adding a **using namespace** directive at the end of the `parselib` namespace), any benefit of ADL and the ability to specialize templates from within the enclosing `parselib` namespace itself would be lost. Note that, because specialization doesn't kick in until overload resolution is completed, specializing overloaded functions is dubious at best; see *Potential Pitfalls — Relying on* **inline** *namespaces to solve library evolution* on page 669.

Providing separate namespaces for each successive version has an additional advantage in an entirely separate dimension: avoiding inadvertent, difficult-to-diagnose, latent linkage defects. Though not demonstrated by this specific example, cases do arise where simply changing which of the version namespaces is declared **inline** might lead to an **ill formed, no-diagnostic required (IFNDR)** program. This 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 659.

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

`ink-safe-abi-versioning`

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 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)
};
```

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

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

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

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

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

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

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

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

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

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

```
        };
    };
}

#endif
```

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

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

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

Observing common good practice, we include the header file of the component as the first substantive line of code to ensure that — irrespective of anything else — the header always compiles in isolation, thereby avoiding insidious include-order dependencies.[5] When we compile the source file `my_thing.cpp`, we produce an object file `my_thing.o` containing the definition of the 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`

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

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

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

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

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

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

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

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

            // ...
        };
    }

    inline namespace v2    // inner namespace (default current version)
    {
        class Key
        {
        private:
            std::uint64_t d_securityHash;
            std::uint32_t d_key;
```

```
                // sizeof(Key) is 16 bytes

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

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

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

des-and-abi-link-safety

Build modes and ABI link safety

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

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

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

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

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

    // ...
};
```

Note that we have employed the C++11 **`alignas`** attribute (see Section 2.1."**`alignas`**" on page 158) here because it is exactly what's needed for this usage example.

The `d_engaged` flag in the example above serves as a way to detect misuse of the `ManualBuffer` class but only in debug builds. The extra space and run time required to maintain this Boolean flag is undesirable in a release build because `ManualBuffer` is in-

tended to be an efficient, lightweight abstraction over the direct use of **placement `new`** and explicit destruction.

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

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

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

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

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

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

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

The approach demonstrated in this example tries to ensure that a linker error will occur if any attempt is made to link objects built with a build mode different from that of `manualbuffer.o`. Tying it to the `NDEBUG` flag, however, might have unintended consequences; we might introduce unwanted restrictions in what we call **mixed-mode builds**. Most modern platforms support the notion of linking a collection of object files irrespective of their optimization levels. The same is certainly true for whether or not C-style `assert` is enabled. In other words, we may want to have a mixed-mode build where we link object files that differ in their optimization and assertion options, as long as they are binary compatible — i.e., in this case, they all must be uniform with respect to the implementation of `ManualBuffer`. Hence, a more general, albeit more complicated and manual, approach would be to tie the 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
#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 659):

```
// 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 668.

-short-named-entities

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 649 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.

falls-inlinenamespace

sioning-doesn't-scale

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 659, will result in code that directly uses the unversioned name, `my::VersionedThing` being bound directly to the versioned name `my::v1::VersionedThing` or `my::v2::VersionedThing`, along with the class layout of that particular entity. Sometimes details of using the **`inline`** namespace member are not resolved by the linker, such as the object layout when we use types from that namespace as member variables in other objects:

```
// my_thingaggregate.h:

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

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

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

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

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

this nature would need to be considered, resulting in having to provide the full cross-product of nested namespaces to get link-time protection against mixed-mode builds.

This need for layers above a library to be aware of and to integrate into their own structure the same namespaces the library has removes all or most of the benefits of using **inline** namespaces for versioning. For an authentic real-world case study of heroic industrial use — and eventual disuse — of **inline**-namespaces for versioning, see *Appendix: Case study of using* **inline** *namespaces for versioning* on page 674.

-std-can-be-problematic

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[6]:

- Avoid specializing variable templates, even if dependent on user-defined types, except for those variable templates where specialization is explicitly allowed.[7]

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

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

- 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.[8]
- Overloads of Standard Library functions that depend on user-defined types are permitted, but, as with specializing Standard Library templates, users must still meet the requirements of the Standard Library function. Some functions, such as `std::swap`, are designed to be customization points via overloading, but leaving functions not specifically designed for this purpose to vendor implementations only helps to avoid surprises.

Finally, upon reading about this **`inline`** namespace feature, one might think that all names in namespace `std` could be made available at a global scope simply by inserting an **`inline namespace`** `std {}` before including any standard headers. This practice is, however, explicitly called out as ill-formed within the C++11 Standard. Although not uniformly diagnosed as an error by all compilers, attempting this forbidden practice is apt to lead to surprising problems even if not diagnosed as an error immediately.

line-keyword-is-ifndr

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

ances-inlinenamespace

Inability to redeclare across namespaces impedes code factoring

mpedes-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`:

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

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

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

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

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

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

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

Attempting to declare a template in the `outer` namespace and then define, effectively 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
```

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

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

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

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

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

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

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

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

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

namespace outer
{
    inline namespace inner
    {
```

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

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

> 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.

-given-inline-namespace

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.[11]

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

[10] See **?**, section 14.7.3.7, pp. 375–375, specifically p. 376.
[11] **?**

See Also

see-also

- "**`alignas`**" (§2.1, p. 158) ♦ provides properly aligned storage for an object of arbitrary type `T` in the example in *Use Cases — Build modes and ABI link safety* on page 663.

Further Reading

further-reading

- **?** uses inline namespaces as part of a proposal for a portable ABI across compilers.
- **?** uses inline namespaces as part of a solution to avoid ODR violation in an interpreter.

Appendix: Case study of using `inline` namespaces for versioning

spaces-for-versioning

By Niall Douglas

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

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

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

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

- `lib::v1` is the *stable*, version-1 ABI, which is guaranteed to be compatible with all past and future `lib::v1` ABIs, forever, as determined by the ABI-compliance-check tool that runs on **CI**. The same goes for `v2`, `v3`, and so on.
- `lib::v2_a7fe42d` is the *unstable*, version-2 ABI, which may be incompatible with any other `lib::* ABI`; hence, the seven hex chars after the underscore are the git short **SHA**, permuted by every commit to the git repository but, in practice, per CMake configure, because nobody wants to rebuild everything per commit. This ensures that no symbols from any revision of `lib` will *ever* silently collide or otherwise interfere with any other revision of `lib`, when combined into a single binary by a dumb linker.

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

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

But I'll tell you now, I don't use **`inline`** namespaces anymore. Now what I do is use a macro defined to a uniquely named namespace. My build system uses the git SHA to

synthesize namespace macros for my namespace name, beginning the namespace and ending the namespace. Finally, in the documentation, I teach people to always use a namespace alias to a macro to denote the namespace:

```
namespace output = OUTCOME_V2_NAMESPACE;
```

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

Why?

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

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

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

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

The noexcept Function Specification

noexcept-specifier

placeholder

Reference-Qualified Member Functions

refqualifiers

placeholder

Unions Having Non-Trivial Members

unrestricted-unions

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

Description

ctedunion-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**:

```
    std::string

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 "??" on page ??) 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:

```
    union

    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:

newunionU2

```
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 30) will still result in their being deleted because the compiler cannot know which member of the union is active.

strictedunion-use-cases

ating-(or-tagged)-union

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:

```
    std::ostringstreamstd::string
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:

`std::string`

```
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 678, 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]:

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

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

ParseResult::ParseResult(const ParseResult& rhs) : d_isError(rhs.d_isError)
{
    if (d_isError)
    {
        new (&d_error) std::string(rhs.d_error);
            // 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.

[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**.

potential-pitfalls

Potential Pitfalls

d-behavior-at-runtime

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 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

Annoyances

see-also

See Also

- "??" (Section 1.1, p. ??) ♦ 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

Further Reading

- ?
- ?

sec-unsafe-cpp14

Deducing Types Using `decltype` Semantics

decltypeauto

placeholder

n-Return-Type-Deduction

Function (auto) return-Type Deduction

placeholder text.....

Chapter 4

Parting Thoughts

ch-parting

Testing Section

Testing Another Section

Glossary

ABI

Application Binary Interface. An interface to a subsystem that is specified in binary, rather than in source.

I find ABI and API description be very non-informative. ABI is the set of rules and conventions that a binary module must follow to interoperate with other binary modules. Common examples of ABI include:

Physical layout of libraries and binaries. Procedure calling conventions (aka passing and returning parameters to/from a function call) Interfacing with operating system and/or hardware

649, 656

API

Application Programming Interface. An interface to a subsystem specified at the source-level, rather than at the binary level.

API is a interface between multiple software modules. Common examples of API: * Data formats and data exchange protocols. * Calls (or requests) that can be made between modules. 649, 656

Abstract Interface

TODO

Access Modifier

TODO

Access Specifier

One of the keywords 'private', 'public' or 'protected' which specified what code may access part of a class. Those parts of a class specified as 'private' may be accessed only within the class, those parts specified as 'protected' may be accessed only either within the class or by derived classes, and those specified as 'public' may be accessed by anything, including other classes.

32, 377, 389

Acquire/Release Memory Barrier

An acquire/release memory barrier, also known as a "fence", will prevent all write operations in the thread after the fence from being reordered before the fence, and

prevent all read operations in the thread before the fence from being reordered after the fence.

Memory barriers will also synchronize with fences and atomic loads and stores in other threads in certain ways, depending upon the memory order of those memory barriers, loads, and stores. 69

Aggregate

Prior to C++17, a type with no user-provided constructors, no base classes, with all of its non-static data members public, and no virtual functions knowns as a POD (Plain Old Data) type, could be aggregate-initialized. In C++17, base classes are allowed, provided that they are public, non-virtual, and don't have constructors. 127–129, 212, 308, 523

Aggregate Initialization

In aggregate initialization, an Aggregate variable can be initialized by a sequence of objects surrounded by curly braces and separated by commas. A class or an array can be aggregate initialized. The object being initialized has to be an Aggregate type or an array of Aggregate types. 127, 128, 204, 308

Aggregate Type

TODO 261

Algebra

TODO 597

Algorithm Selection

TODO 584

Alias Template

A statement that aliases one template expression to another via a 'using' declaration.

123, 532

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`**. 158

Alignment Requirement

TODO 158, 173

Argument-Dependent Lookup (ADL)

TODO 454, 477, 491, 649, 651, 652, 656, 666

Arithmetic Type

TODO 312

Array To Pointer Decay

TODO 202

Array Type

TODO 173

Atomic

Atomic variables are variables which have appropriate memory fences and barriers built in to all their load and store operations to facilitate multiple threads coordinating with one another without having to use locks.

69

Automatic

TODO

Automatic

TODO 58

Automatic Storage Duration

TODO 57

B0

TODO

Barriers

A barrier to prevent reordering of instructions, and influencing cache behavior to facilitate coordination of data in multithreading. 71

Base Name

TODO 441

Base Specifier List

TODO 529

Basic Source Character Set

TODO 98, 118

Benchmark Test

A test measuring the performance of software, usually measuring speed. 102

Binary Search

TODO 273, 275

Block Scope

TODO 406

Boilerplate Code

Code that is repeated from one project to another with little variation.

Above is a definition of a copy-paste. Boilerplate code is the code that is necessary to bring a complex software system in operational state. 124, 150, 301, 311

Brace Elision

In aggregate initiailization, under some circumstances braces around a class or array within the larger object being initialized can be left out. 129

Brace Initialization

TODO 255, 261, 456

Bytes

A byte is a collection of 8 bits, usually accessed as a unit. 142, 143

C-Style Function

A C Style Function is a function whose signature is defined by a sequence of arguments and a return type (where the return type may be 'void'). Virtual functions and class methods are not C Style Functions. 147

CI

Continuous Integration (CI) is the practice of automating the integration of code changes from multiple contributors into a single software project. 674

CV-Qualifiers

TODO 353

Cache Hit

TODO 170

Cache Line

TODO 163, 164, 170

Cache Miss

TODO 170

Call Operator

TODO 394, 395, 398, 399, 403, 411–416

Callable Object

A callable object may be a pointer to a C-style function or a C++ static functiion, or an object of a type which has 'operator()' defined so that the object can be called like a function. 59, 614

Callback Functions

An interface technique where the caller of an interface provides callable objects to be called by the called entity to communicate output. 443

Capture Default

TODO 401, 402, 410, 422, 428

Capture Init

TODO 429, 431

Captured By Copy

TODO 401–404, 408–410, 414, 422, 424, 428, 430, 431

Captured By Reference

TODO 401, 402, 404, 409–411, 414, 415, 422, 428

Captured Variable

TODO 424

Carry Dependecy

TODO 616

Cast

TODO 322

Character Literal

TODO 487

Class Member Access Expression

Member methods of a class can be called via 'object.method(...)', or through a pointer to the object as 'pointer->method(...)'. Non-static data members can similarly be referred to as 'object.member' or 'pointer->member'. Static functions can be referred to as 'typename::function(...)', static data members may be referred to via 'typename::member'.

Class Template

TODO 537

Class Template Argument Deduction

TODO 459

Closure

In a C++ lambda function, the part of the function declaration between the '[]' brackets is the closure, and indicates variables other than function arguments that are accessible to the function, either variables copied by value, or by reference. 76, 397–400, 403–405, 410–412, 415, 416, 420, 421, 424, 428, 606

Closure Object

TODO 398–401, 403, 404, 414, 421, 423–425, 427–429, 432

Closure Type

TODO 397–400, 405, 411, 413, 416, 423, 424

Code Bloat

The phenomenon where the code size of a program grows. Several factors can contribute to this, such as the use of inline subroutines, or enabling exceptions (since enabling exceptions makes it necessary for the compiler to generate stack unwinding code). 329, 647

Collatz Function

TODO 292

Collatz Length

TODO 292

Collatz Sequence

TODO 292

Comma Operator

TODO 250

Compile Time Introspection

TODO 584

Compile-Time Constant

TODO 323, 324

Compile-Time Coupling

Most couplings in C++ are compile time, except for calls through function pointers or method pointers, or calls to virtual functions, in which case the decision about which function will be called is partially made at run time.

438

Compile-Time Dispatch

TODO 109

Complete Class Context

TODO 297

Complete Type

A type where all the information describing the type is available, so that it is possible for the compiler to create an object of the type. Note that 'void' is not a complete type. 173, 297, 327, 436, 536

Component

A unit of C++ source consisting of 3 files: An include file, an implementation file, and a test driver. 335, 347, 440–442, 660

Component Local

TODO 439

Components

TODO 334, 629, 630, 635

Concepts

A feature of C++20 and beyond where it is possible to add constraints to templates to limit the applicability of the template. Prior to C++ this had to be done with SFINAE which was much clumsier. 110, 193

Concrete Class

Any class that is not abstract. Concrete classes, unlike abstract classes, can be instantiated. A concrete class is a class which has no pure virtual functions. 380

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

An expression that can be evaluated at compile-time. Static const variables initialized inline are required to be initialized by constant expressions and are themselves constant expressions. 103, 206, 239–244, 246, 248, 249, 251, 259, 262, 265–268, 270, 278, 282–285, 290, 292, 295, 486, 488

Constant Initialization

TODO 64

Constexpr

TODO 255

Constexpr Context

TODO 240, 242, 246, 247, 255, 256, 259, 261, 281, 292

Contextual Convertibility to bool

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))`**. 52, 53

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. 92

Continuous Refactoring

TODO 136, 139

Contract

TODO 600

Controlling Constant Expression

TODO 266, 267

Conventional String Literals

Conventional string literals consist of a sequence of characters delimited by '"'s, creating a sequence of 'const char' characters. A string literal beginning with 'L"' creates a sequence of 'const wchar_t' objects. The string literal is of type 'char[N]' where 'N' is the integral number of 'char's in the string. 101

Conversion Operators

A conversion from a class to a given type 'TYPE' can be declared as a member method, usually an accessor, of the form 'operator TYPE() const; which will create an implicit cast from the class to a 'TYPE' object. 50, 51, 53

Converting Constructors

A converting constructor is a single argument constructor that thus creates a means of converting one type to another. Converting constructors can be implicit or explicit.

50, 51

Cookie

TODO 443

Copy

TODO 222

Copy Elision

Copy operations consist of the copy constructor and the copy assignment operator. 363

Copy Initialization

TODO 194, 198, 199, 204, 208, 209, 211, 212, 296, 606

Copy Initialize

TODO 203, 207

Copy List

TODO 222

Copy List Initialization

TODO 207–209, 211–213

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... 500, 730

Copy/Direct

TODO 198

Critical Section

In a multithreaded component, a section of code guarded by a mutex during which critical operations can take place. 60

Curiously Recurring Template Pattern (CRTP)

635, 636

Currying

TODO 420, 425

Cyclic Physical Dependency

TODO 348

Cyclically Dependent

TODO 64

Data Dependency

TODO 616

Data Dependency Chain

TODO 616, 617

Data Member

TODO 158

Data Races

A data race is a defect in a multithreaded system where the ordering of events in different threads is insufficiently constrained to guarantee the correct opetation of the system. 57, 59, 60

Decimal Floating-Point

TODO 509

Declaration

A declaration lays out the type of an object or function, but not its implementation. 109, 111, 294, 525

Declarator Operator

TODO 534, 535

Declared Interface

The type of the entity named by the given expression. 607

Declared Type

TODO The type of the *entity* named by the given expression. 22

Declaring

TODO 363

Deduced Return Type

TODO 415, 416

Default Initialization

TODO 199

Default Initialized

A variable that is initialized with an '=' statement on the definition. Note that this is different from 'Default Constructed'. 33, 201, 203, 212, 301

Default Member Initializer

TODO 252, 255, 296

Default/Value

TODO 198

Defensive

TODO

Define

TODO 363, 525

Definition

The specification of an object or function, including the implementation. 57, 66, 294, 525

Delegating Constructor

In C++11 and later, it is permissible for a constructor of an object to call other constructors of the object, delegating to them. 43

Deleted Function

In C++11 and beyond, a function in a class can be declared as 'deleted', which means that the function will not be created. This is necessary for unwanted constructors and assignment operators which would normally be generated by default.

30

Dependent Type

378

Design Pattern

A pattern of coding used to solve a class of problems. 443

Diffusion

TODO 171

Dimensional Unit Type

TODO 510

Direct

TODO 222

Direct Initialization

When a variable is initialized via an '=' sign, either with an aggregate or with an expression. 51, 194, 198, 208, 211, 212, 296

Direct Initialization

TODO 170

Direct Initialized

TODO 212

Direct List

TODO 222

Direct List Initialization

TODO 211–213, 223, 296

Disambiguator

In some expressions, it is necessary to specify 'template' before an expression to indicate that it is a template and avoid the compiler interpreting the '<' as a 'less-than' symbol. It is also necessary within a template definition to sometimes say 'typename' before an expression to clarify that the following expression is a type and not a variable. 25–27

Divide And Conquer

TODO 278

Duck Typing

A template function can take a value whose type is a template parameter, and if that type supports the wanted funcitonality, the template function can call it. "If it can quack, it must be a duck.". 645

Dumb Data

A type that contains data but has not functions or constructors built in. 442

Eigen

TODO

Embedded Development

TODO 133

Embedded Systems

Systems implemented in the field with the program burned into ROM memory rather than being loaded into RAM every run. 82

Emplacement

TODO 363, 364

Empty Base Optimization

TODO 174, 572

Encapsulation

Encapsulation is the ability to put both the data being operated upon and the funcitons operating on it into a single unit. C++ classes facilitate this. 438

Encoding Prefix

TODO 493

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, 22, 73, 136, 137, 139

Escalating

On Windows machines, a task can escalate its privileges above the privileges it started out with. On Unix, a program can set the user id to the owner of the file, so if the owner of the file is a superuser, that is a way that the program can run with superuser privileges even though the one running it doesn't have those privileges. 348

Exception Specification

TODO 415

Excess N

TODO 144

Execution Character Set

TODO 493

Explicit

TODO 600

Explicit Instantiation Declaration

An explicit instantiation declaration of a template tells the compiler not to generate the implementation of the template because one will be provided explicitly. 329–331, 333, 334, 337, 340, 344–346, 348, 349

Explicit Instantiation Definition

An explict definition of the implementation of a fully specified template type. 329–331, 333, 334, 339, 344–346, 348, 349

Explicit Instantiation Directive

An explicit instantiation declaration or definition. 329, 330, 344, 349

Explicit Template Argument Specification

TODO 540

Explicitly Captured

TODO 401, 402, 412

Explicitly Copied

TODO 402

Expression SFINAE

SFINAE stands for 'Substitution Failure Is Not An Error". SFINAE is a trick used to weed out of functions or constructors from an overload set by having expressions in the argument list or return type which fail to compile. When this happens, the compiler does not report a syntax error but instead just removes the overload from the list of candidates. 26, 110, 114

Expression Template

Expression templates is a template metaprogramming technique where the full structure of an expression is represented with nested template instantiations allowing lazy evaluation and certain optimizations. For example, this technique is often utilized by linear algebra libraries, such as "**Eigen**" (http://eigen.tuxfamily.org/). 189

Extended Alignment

TODO 158

Factory Function

TODO 486, 487, 568

Factory Operator

TODO 487

Fences

A barrier preventing the reordering of instructions by the compiler, and sometimes influencing cache behavior, to facilitate lock-free multithreading. 71

File Extension

TODO 441

Floating Point Literal

TODO 487

Floating-Point-to-Integer Conversion

TODO

Flow Of Control

The sequence of operations in a program as the sequence flows through the code. 57

Footprint

The non-dynamic memory taken by an object, which is just the memory taken for the object itself. 332

For Range Declaration

TODO 454

Forward Class Declaration

A class declaration without a definition, letting the compiler know in advance that a class with a certain name will be defined at some point. Under some circumstances, the class will be defined later in the file, in others, the class will never be defined in the current compilation unit, but pointers and references to it will be manipulated. 449

Forward Declaration

A declaration of a type or function without a definition, so that code following the forward declaration can call the function or manipulate the type, without knowing the full specification. Note that forward declarations of enum types was impossible prior to C++11 because the storage class of the enum could not be determined without full knowledge of the enumerator values. 436, 437, 439

Forward Declare

TODO 438–440, 449

Forward Declared

TODO

Forwarding Reference

TODO 351, 453, 464, 469, 472, 561

Fragmentation

TODO 171

Free Function

A function that is not a member of a class. 289

Free Operator

TODO 488

Full Expression

TODO 465, 467

Full Specialization

TODO 600

Fully Associative

TODO 170

Fully Constructed

An object whose constructor has finished running. 44

Function Object

TODO 273, 307, 394, 397, 610, 613

Function Parameter Pack

TODO 524, 533, 534, 540, 542, 543, 556

Function Scope

The scope of the declaration of a function, which can either be in a namespace, or at file scope, or within a class. 57, 58, 60, 62, 69, 70

Function Template Argument Type Deduction

TODO 183

Function try Block

TODO

Functor

TODO 394, 397

Functor Class

TODO 394, 395

Functor Type

TODO 405

Fundamental Alignment

TODO 158, 165

Fundamental Integral Type

A type supported natively by the compiler which is not a class, struct, or union and is integral. 78

Fundamental Integral Types

78

Garbage Value

TODO

General Purpose Machines

General-purpose machines are what Big Tech, the financial industry, and many other large companies use exclusively. 82

Generic Lambda

TODO 412, 427

Generic Type

TODO 524

Golden File

A file containing the expected output of a test program. The test program is run, creating a file of output, which is compared to the golden file, and if the files match, the test passes.

102

Golden Output

TODO

Header-Only Library

TODO

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**. 376, 379

Higher-Order Function

A higher order function either takes a function as an argument, or returns a function as its return value. 113, 115

Hyrum's Law

Hyrum's Law states that with a sufficient number of users of an API, it does not matter what is promised in the contract, all observable behaviors will be depended on by someone in the user base. 630

IFNDR

see Ill-formed, No Diagnostic Required 545, 589, 590

Id Expression

A qualified or unqualified identifier. It can consist of an identifier, or an identifier with additional syntax to make it any type of C++ object. Note that an id expression can occur after the '.' or '->' operators. 22

Ill Formed

TODO 108, 209, 283, 286, 489, 659, 663, 669

Ill Formed, No Diagnostic Required (IFNDR)

TODO 47, 85, 105, 107, 166, 244, 258, 327, 440, 441, 449, 450, 483, 544, 618, 659, 661, 664, 670

Imperative

TODO 595, 599

Implementation Defined

Implementation defined behavior is behavior that depends not only upon the definition of C++, but also the implementation of the platform the program is compiled on. For example, 'sizeof(int)' on a PDP-11 is 2, while on an x86 it is 4. 10, 88, 158, 276, 313, 314, 435

Implementation Inheritance

The implementation of a derived class can either inherit from the base class, or a new implementation of the derived class can be specified which will override the implementation of the base class. 380

Implicitly Captured

TODO 401, 402, 408, 409, 428

Incomplete Type

TODO 600

Inheriting Constructors

In C++11 and beyond, a class can inherit constructors from a base class. This must be explicitly requested – in the case of a class 'Derived' which inherits from a class 'Base', the statement 'using Base::Base;' must occur in derived. This is an all or nothing proposition – either 'Derived' inherits all of 'Base's constructors, or none of them. 378, 389

Init Capture

TODO 410, 411

Initializer List

TODO 456

Instantiation Time

TODO 108

Insulate

TODO 84, 280, 344, 440

Insulation

TODO 438

Integer Literal

A literal specifying an integer value. These can be in decimal, in which case the literal is a sequence of digits in the range '[0-9]' not beginning with '0', or octal in which case they begin with '0' and consist of digits in the range '[0-7]', or hexadecimal, in which case the literal begins with 0x' and is followed by digits in the range '[0-9a-f]'. The values created by an integer literal must be low enough to fit in an integer. 131, 487

Integer-to-Floating-Point Conversion

TODO

Integral Constant

TODO 205, 283

Integral Constant Expression

TODO 159, 173

Integral Promotion

In expressions, an integral bit field of less than the number of bits in an 'int' is, by default, promoted to an 'int'. In integral expressions, if an operation occurs between two integral variables of different sizes, the smaller one will be promoted to the larger size before the operation. 311, 483

Integral Type

A non-floating fundamental type. 78, 480, 481

Interface Inheritance

Any time a class 'Derived' is publicly derived from a class 'Base', 'Derived' inherits the interface of 'Base', except for the constructors. Note that 'Derived' may expand on the interface it inherits. 380

Interface Test

TODO 257, 264, 265

Internal Linkage

TODO 66, 286, 287, 293

Intra-Thread Dependency

TODO 616

Invocable

TODO 606

Join

Suspend the execution of one thread until another thread or other threads finish executing.

Lambda Body

TODO 401–406, 408–412, 414–419, 421, 427–429, 431, 432

Lambda Capture

TODO 397, 399–402, 404, 406, 408–412, 426, 427, 429, 431, 561

Lambda Closure

TODO 403, 413

Lambda Declarator

TODO 411, 414–416, 419

Lambda Expressions

Syntax dedicated to the creation of locally defined functions or function objects (so-called closures) anywhere where expressions are allowed. 72, 396–401, 403–416, 418–429, 431–433, 613

Leaks

A limited system resource (most often memory) that is allocated to a process but never released.

Lifetime Extension

TODO 453, 463–466, 472

Linkage

Defines the visibility and behavior of named entities during the static and dynamic linking process. 72

List Initialization

TODO 198, 215, 242, 275, 309, 606

Literal

TODO 485

Literal Type

A type that may be used to create constexpr variables, or be constructed, returned, or taken as argument by constexpr functions. 242, 243, 249, 255–265, 267, 271, 282, 284, 285, 596

Local Declaration

A declaration in the local scope. 437, 439–441

Local Scope

The scope of the current statement that is the scope of a function body or another compound statement inside a function body.

Locality of Reference

TODO 170

Logical

A value with 2 distinct states representing true or false. 340

Long-Distance Friendship

TODO

Lossy Conversion

TODO 204

Mangled Names

A name the compiler creates (and the linker uses) to identify potentionally scoped and/or overloaded names. Such names will contain information to identify the full absolute scope of the name, and in case of C++ function names the signature.

649, 667

Manifestly constant evaluated

TODO 240

Mantissa

The part of a floating point representation that contains the significant digits. 144

Maximal Alignment

TODO

Maximal Fundamental Alignment

TODO 181

Maximally Aligned

TODO

Mechanism

A type that has no sensible definition of value. Such types will have behavior, may have state, but that state will not serve the purpose of an abstract value. 438

Mechanisms

48

Member Function

TODO 537

Member Initializer List

Syntax used in constructors to determine how non-static data members get initialized. Note that the initialization order is fixed to the order of declaration, no matter what order the member initializer list has.

43, 45, 47, 296, 308

Member Initializer List

212

Memory Leak

See **leak**. 63

Memory-Fence Instruction

TODO 617

Message

TODO 442

Messenger

TODO 442

Metafunction

TODO 355, 370, 599, 600

Metaparameter

TODO 585

Metaprogram

TODO 239, 522

Meyers Singleton

A singleton implemented as a static variable in the scope of a function, introduced in Scott Meyers' More Effective C++ Item 26. 61, 62, 69

Mix-In

TODO

Mixed-Mode Builds

TODO

Modules

A C++20 feature that introduces a new way to organize C++ code that provides better isolation than headers. 635

Monotonic Allocator

TODO 178

Most Vexing Parse

Code intended as variable declaration, that is parsed as a function declaration due to the use of constructor alls as arguments:

376

Move Operations

Move construction, move assignment, and their conversion variations where internals of an initializer or assignment right hand side are "moved" into the new or assignee object, leaving the source object in a valid, but unspecified state called moved-from state. The name "move" can be misleading, as the whole point of this operation is to *not* "move" it to a new memory location. 40, 171

Move Semantics

An operation on two objects, a destination and a source, has move semantics if, after it completes, the target object has the same value that the source object had before the operation began. Note that the source object may be modified, and is left in an unspecified state after the operation completes. Also note that any operation that has copy semantics also has move semantics. 730

Multithreading Context

TODO 57

Naked Literal

TODO 489–495, 498, 499, 508

Name Mangling

The process by which a **mangled name** is created by the compiler or other tooling such as a symbolic debugger. Mangled names, in addition to the name, may contain scope, and type information, as well as other decorations that are specific to the ABI. 659

Narrow Contract

TODO 495

Narrowing Conversion

TODO 204, 205

Natural Alignment

TODO 168, 173, 181, 182, 482

Naturally Aligned

TODO

Nibbles

Half a byte, or 4 bits. 142, 143

Non-Trivial Constructor

TODO

Non-Trivial Special Member Function

TODO

Non-Type Parameter

TODO 546

Non-Type Parameter Pack

TODO 524, 546, 547

Non-Type Template Parameter

TODO 546

Non-Type Template Parameter Pack

TODO 546

Nonprimitive Functionality

Functionality that is entirely implementable in terms of primitive functionality of a type or a set of types, hence requires no access to encapsulated, private details. 56

Nonstatic Data Member

TODO 285

Null Address

A platform specific address that represents a null pointer value. This address is not necessarily all zero bits. 87, 90

Null Statement

TODO 250

Null Statements

TODO

ODR-used

401, 403, 408, 409, 419, 429, 430, 608

Object Invariants

Properties of objects of a class that shall remain unchanged during the lifetime of any object of that class, except during computations performed in member or friend functions. 379

One-Definition Rule (ODR)

244, 348, 664, 670, 673

Opaque Declaration

A declaration that is not also a definition. 435–437

Opaque Enumeration

Declaration of an enumeration with an underlying data type that is not also a definition. 438, 443, 446, 451

Operator

TODO 247

Ordered After

TODO 616

Over-Aligned

TODO

Overload Resolution

The well-defined and complex process by which the C++ compiler decides which homonymous function shall be called with a set of concrete arguments. 491, 493

Overriding

Replacing the implementation of a virtual function for a specific derived type. 379

Pack Expansion

TODO 527, 528, 531, 534, 535, 552–554, 556, 557, 559, 563–565, 567, 569, 583, 584, 591, 602

Parameter Count

TODO

Parameter Declaration

TODO 533

Parameter Pack

A list of zero or more types or values that are parameters to a **variadic template**. 148, 409, 411, 525, 526, 528, 600–602

Parameter Pack Expansion

TODO 528, 553, 554, 560, 565, 566

Parameter Pack Expansion Context

TODO 529

Partial Application

TODO 420

Partial Class Template Specialization

TODO 600

Partial Implementation

TODO 380

Partial Ordering Of Class Template Specialization

TODO 531

Partial Specialization

TODO 530

Partially Constructed

TODO – Not in John's book. Not in any presentations that have actual notes or slides defining it. 44

Perfect Forwarding

TODO 222, 580, 593

Perfectly Forwarded

TODO 612

Physical

TODO 340

Physical Design

The way in which a logical content is factored and partitioned into source files, and libraries. 437, 635

Placeholder Type

TODO 183

Placement

TODO

Placement `new`

A way to construct an object at a particular memory location, i.e. to call a constructor. 364, 578, 664

Plain Old Data (POD)

A now (C++20) deprecated term (replaced by **Standard Layout Type**) that was used to describe the C++ types that were C compatible, where the same code defining the type would result in the same layout in the two languages, and all initialization, destruction, and copy operations were trivial. 308

Pointer to Member

A special type that is able to identify (by its value) a specific non-static member, having a specific type, of a specific class type, such as an 'int' data member of 'class X', or a possibly virtual, non-static member function of a specific signature and return value. An actual member cannot be accessed with a pointer to member value alone, it has to be combined with the address of an actual object. 53

Polymorphic Memory Resource

TODO 178, 307

Posix Epoch

TODO 272

Precondition

Conditions that has to be fulfilled before an operation can be performed, such as a stack must have at least one element before we can pop from it. 16

Predicate

TODO 395

Predicate Function

A function that returns a Boolean value that answers a question or makes a decision. Is this number even? Should this employee get a raise this year? 75

Predicate Functor

TODO 395

Prepared-Argument UDL Operator

TODO 490–492, 494, 495, 497, 517

Primary Class Template Declaration

TODO 526, 530, 532

Primary Declaration

TODO 526

Protocol

A class that has only pure virtual functions except for a non-inline virtual destructor (defined in the .cpp file), has no data members, and does not derive from any other class (directly or indirectly) that is not itself a protocol class. 380

Pun

TODO 210

Pure Abstract Interface

TODO 380

Pure Function

A function that produces no side effects and always returns the same value given the same sequence of argument values, in other words accesses no other state than its parameters. 14

Qualified Name

A name that contains the scope operator (‘::‘). 652

Qualifier

TODO 534, 535

Quality Of Implementation

TODO 259

Range

TODO 25, 522

Range Expression

TODO 453, 460, 463, 464, 466, 467, 476

Range-Based `for` Loop

TODO 452–460, 462–464, 466–468, 472–478

Raw String Literals

A C++11 feature that allows string literals that are actually interpreted literally, without escape sequences. You may know a similar feature from other languages as heredoc. 101

Raw UDL Operator

TODO 490, 491, 495–499, 505, 514, 517, 518

Reaching Scope

TODO 406–408

Receiver

TODO 442

Recursion

TODO 521

Redundant Check

Redundant Code that provides runtime checks to detect and report (but not "handle" or "hide") defects in software. 103

Ref-Qualifiers

TODO

Reference Collapsing

TODO 354, 359

Reference Type

A type that denotes an alias to objects of a certain type. 173

Reference Type

TODO 176

Release-Acquire

TODO 616, 623

Release-Consume

TODO 616, 623

Resource Aquisition is Initialization (RAII)

Resource Acquisition is Initialization (acronym RAII) is term that stands for a technique where acquired/allocator resources (memory, file, and other resource handles) are reliquished in the destructor of a class type, and often acquired/allocated by a constructor of that same object. 362

Return Type Deduction

The name of the feature and the rules and process by which the compiler automatically determines the return type of a function by examining the types of the return statement operands. 25

Return Value Optimization (RVO)

TODO

SHA

SHA (Secure Hash Algorithms) are a family of cryptographic hash functions. 674

Safe-Bool-Idiom

TODO 53

Scalar Type

TODO 158, 199

Scoped Allocator Model

TODO 307

Scoped Enumeration

TODO 435

Section

A part of an object file. A section contains information used by a linker to produce the final executable program. 336, 337, 339, 346

Sender

TODO ? A sender is a device or program that originates an information transfer to one or more receivers. 442

Sequencing Operator

TODO 247

Set Associative

TODO 170

Shadowed

TODO 607

Side Effects

Modification performed by an operation, function or expression outside its local scope. 14

Signature

TODO ? I guess function signature ? Compiler generated string that defines inputs and outputs of a function or class method. 112, 376, 645

Signed Integer Overflow

The result of an operation on 2 signed integer numbers which exceeds the maximum (or goes below the minimum) value the signed integer type can represent. 79

Simple Capture

TODO 410

Slicing

Slicing happens when a derived class object is copied or assigned to a base class object. All additional attributes of the derived class object are removed to form the base class object. 379

Smart Pointer

TODO 585

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*. 30, 32, 40, 41, 254

Specialization

TODO 530

Standard Conversion

Conversions defined in C++ language between its fundamental types, pointers, references and pointers-to-member. 311, 485

Standard-Layout Types

TODO 167, 174

Static Assertion Declarations

Conditions that are checked in compile time. 103

Static Data Space

Memory area of the running program that holds static data. 154

Static Storage Duration

TODO 57, 58, 64, 65, 67, 69

Storage Class Specifier

TODO 183

String Literal

A string literal represents a sequence of characters that together form a null-terminated string.

103, 137, 487

Strong `typedef`

TODO 381–387

Strong `typedef` Idiom

TODO 63

Strong `typedef`s

TODO

Structural Inheritance

TODO 169, 384

Structured Binding

TODO 458, 475

Substitution Failure is Not An Error (SFINAE)

TODO 25, 109–111, 370–372, 379

Synchronization Operation

TODO 616

Synchronization Paradigm

TODO 616, 623

Synthetization

TODO 606, 613

Template Argument

TODO 544

Template Argument Deduction

TODO 538–540, 545

Template Argument List

TODO 528, 529

Template Head

TODO 146

Template Instantiation

TODO 355

Template Instantiation Time

A source code location when a compiler generates a concrete class or a function from a template for a particular combination of template arguments in order to carry the compilation process. 104

Template Parameter

TODO 541

Template Parameter Pack

TODO 524–527, 529, 530, 532–534, 540–543, 546, 547, 551

Template Parameters List

TODO 533

Template Template Class Parameter

TODO 154

Template Template Parameter

TODO 154, 546, 547, 551

Template Template Parameter Pack

TODO 524

Template Type Parameter

TODO 524

Templated UDL Operator

TODO 490, 491, 498, 499, 507, 508, 514, 517, 518

Ternary Operator

TODO 250

Test Driver

Test driver is a program that organizes the tests, runs them and handles their output.

102

Thrashing

TODO 171

Thread Pool

TODO 609

Trailing Return Type

TODO 415, 416

Translation Unit (TU)

TODO 60, 61, 64, 65, 67, 435

Trivial

TODO ? Trivial type ? A trivial type is a type whose storage is contiguous and which only supports static default initialization. 35, 36, 255

Trivial Copy Constructor

TODO

Trivial Operation

Member functions and operators generated by the compiler.

30

Triviality

TODO 217

Trivially Constructible

A trivially constructible type is a type which can be trivially default-, copy- or move-constructed. 69

Trivially Copyable

The following types are collectively called trivially copyable types:

* Scalar types; * Trivially copyable classes, i.e. classes satisfying following requirements: - At least one copy constructor, move constructor, copy assignment operator, or move assignment operator is eligible - Every eligible copy constructor (if any) is trivial - Every eligible move constructor (if any) is trivial - Every eligible copy assignment operator (if any) is trivial - Every eligible move assignment operator (if any) is trivial - Has a trivial non-deleted destructor - Has no virtual functions or virtual base classes. * Arrays of TriviallyCopyable objects 36, 39

Trivially Destructible

TODO 284

True Sharing

TODO 171

Type Category

TODO 487

Type Deduction

TODO 354, 355

Type Erasure

TODO 424

Type Expression

TODO

Type Inference

TODO 181

Type List

TODO 599, 600

Type Parameter Pack

TODO 547

Type Suffix

TODO 486

Type Template Parameter Pack

TODO 546

Type Trait

Type traits are C++ template metaprogramming technic that gives you the ability to inspect and transform the properties of types. 107, 124, 150, 352, 371

UDL Operator

TODO 487, 489–500, 507, 514–518

UDL Suffix

TODO 487, 489–492, 494, 495, 500–502, 509, 517

UDL Type Categories

TODO 492, 495

UTF-16

TODO

UTF-32

TODO

UTF-8

TODO

Undefined

85

Undefined Behavior

Undefined behavior is the behavior of the program that is prescribed to be undefined. Example of undefined behaviour is division by 0. 16, 17, 47, 59, 60, 64, 66–68, 79, 85, 178, 200, 203, 464, 467, 642, 669

Undefined Symbols

Undefined symbols are symbol references in an object file that are not bound to a symbol definition. 338

Underlying Type

An integral type used to represent values of an enumeration. 119, 288, 381

Underlying Type (UT)

TODO 161, 288, 311, 313–315, 327, 435, 436, 441, 480–483

Unevaluated Contexts

TODO 28

Unification

TODO 546

Uniform Initialization

TODO 198

Unnamed Namespace

TODO 66

Unqualified Name Lookup

TODO 491

Usable Literal Type

TODO 263, 265

User Provided

Definition of a function or class methods written by a programmer.

69, 200

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 Section 2.1."Generalized PODs '11" on page 374 for more information. 30, 32

User-Defined Literal (UDL)

TODO 487–491, 496, 500, 502, 504, 505, 509, 510, 512, 515, 518

User-Defined Type (UDT)

A user-defined data type (UDT) is a data type that derived from an existing data type. 43, 158, 168, 301, 485, 626

Value

A value is the representation of some entity that can be manupulated by a program. 48, 612

Value Category

TODO 22, 27, 351, 354, 356, 357, 359, 409, 453, 612

Value Constructor

A constructor that initializes object's data members from the constructor arguments. 33, 486

Value Initialization

TODO 199, 207

Value Initialized

TODO 203, 204, 212

Value Semantic

TODO 33, 45, 48, 176, 438, 730

Value Semantics

a property of the type

Value-Semantic Type (VST)

TODO 359, 612, 629

Variable

TODO 158

Variable Template

TODO 282, 283

Variadic Class Template

TODO 537, 538

Variadic Function Template

TODO 409

Variadic Member Function Template

TODO 537

Variadic Pack

TODO 231

Variadic Pack

TODO 602

Vocabulary Type

A vocabulary type is a type that holds a value or performs a service that is used widely in the interface of classes and/or free functions. 79

Vocabulary Types

TODO 80, 119, 272

Well Formed

TODO 159

With Parentheses

TODO 201

Without Parentheses

TODO 201

Witness

TODO 265

Witness Argument

TODO 265, 266

Working Set

TODO 171

Zero Initialized

Object whose initial value is set to zero.

The effects of zero initialization of an object of type T are: * If T is a scalar type, the object's initial value is the integral constant zero explicitly converted to T. * If T is an non-union class type, all base classes and non-static data members are zero-initialized, and all padding is initialized to zero bits. The constructors, if any, are ignored. * If T is a union type, the first non-static named data member is zero-initialized and all padding is initialized to zero bits. * If T is array type, each element is zero-initialized. 64, 201, 255

automatic variable

TODO

const Default Constructible

TODO 219

explicit Specifier

The 'explicit' keyword, when specified on a constructor, means that the constructor will not be called implicitly, and must be specifically called.

51

false sharing

TODO 163, 164, 167, 171

in-process

TODO

inline namespace

In C++11 and beyond, inline namespaces are available. Objects defined within an inline namespace can be accessed without specifying the name of the inline namespace. 648

lambda introducer (adj)

TODO 401, 416, 606, 610

lvalue

TODO

lvalue Reference

TODO 471, 472, 613

new Handler

TODO 181

prvalue

TODO

rvalue

TODO 612

rvalue reference

TODO 471, 472

std::unique_ptr

609

t=Translation-Lookaside Buffer (TLB)

TODO 171

using Declaration

Using-declarations is used to introduce namespace members into other namespaces and block scopes, or to introduce base class members into derived class definitions, or to introduce enumerators into namespaces, block, and class scopes (since C++20). 250, 375

using Directive

using namespace <ns-name> Makes every name from ns-name visible in the nearest enclosing namespace for the purposes of unqualified name lookup.

250, 491, 516, 649

Diagnostic Information

Table 1: Diagnostic information

Creation Time	2021-04-10 03:16
Built by	`jberne4`
LaTeX version	LaTeX 2_ε
XeTeX version	0.999991
Build host	`dev-10-34-11-129.pw1.bcc.bloomberg.com`
Build Operating System	`Linux dev-10-34-11-129 5.4.0-65-generic #73-Ubuntu SMP Mon Jan 18 17:25:17 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux`

Diagnostic Information

This is the a first glossary entry: **move semantics**
This is the a second glossary entry: move semantics
This is the a first glossary entry: **copy semantics**
This is the a second glossary entry: copy semantics
NEW SECTION:
This is the a first glossary entry: **move semantics**
This is the a second glossary entry: move semantics
This is the a first glossary entry: **copy semantics**
This is the a second glossary entry: copy semantics
This is the a second glossary entry with different text: *Copy* Semantics
This is the a first glossary entry with different text: **value value value**