Chapter 0

Introduction

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

- We are practicing software engineering professionals employed by a global software-development enterprise. As senior developers with decades of real-world experience, we are primarily writing for software developers, team leads, and managers working in similar environments.
- We deliberately focus on the productive value that a given language feature affords (or fails to afford), particularly when the systems and the organization that employs the developers writing them are considered at scale. We do not focus on the aesthetics of language features that could hurt the "bottom line" when applied at scale in a large organization.
- Because we focus on only the features of modern C++ that have been part of the Standard and have been actively used for at least five years, we are able to provide you with a thorough and comprehensive treatment based on practical experience and worthy of your limited professional development time.
- By focusing on what is objectively true and relevant to making wise economic and design decisions — with an understanding of the tradeoffs that will arise in real-world practice — we steer clear of subjective opinions and recommendations.
- Finally, this book — a thorough catalog of consequential information written by expert C++ programmers experienced and adept at using certain (early) modern language features in practice — is explicitly and intentionally designed *not* as a tutorial of the latest features for beginners, but as a guide for senior C++ programmers less familiar with these specific features.

0.1 Scope for the First Edition

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

In this first of an anticipated series of periodically extended volumes, we characterize, exhibit, dissect, and elucidate some, but not all, of the modern language features introduced into the C++ Standard, 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 also decided to confine our attention to just the subset of C++ language features introduced in C++11 and C++14, rather than attempt to cover all the useful features available in C++98/03. Since we can assume with virtual certainty that most of you are quite familiar with essentially all the basic and important special-purpose features of classical C++, we made the tough decision to focus exclusively on what you need to know today: how to safely incorporate C++11/14 language features into a predominately C++98/03 code base.

Over time, we expect (and hope) that practicing senior developers will emerge entirely from the postmodern C++ era. By then, a book that focuses on all of the important features of modern C++ would naturally include many of those that were around before C++11. With that horizon in mind, we are actively planning to cover pre-C++11 material in future editions. For the time being, however, we highly recommend *Effective C++* by Scott Meyers[1] as a concise, practical treatment of many important and useful C++98/03 features.

0.2 What Makes This Book Different

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

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

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

[1]**meyers97**

you how best to utilize a given C++ language feature in practice, our analysis takes into account the costs associated with having that feature employed routinely in the ecosystem of a commercial software-development organization. In other words, we weigh the benefits of successfully using a feature against the risk of its widespread ineffective use (or misuse) and/or the costs associated with training and code review required to reasonably ensure that such ill-conceived use does not occur. The outcome of this analysis is our signature categorization of features in terms of *safety*, namely *safe*, *conditionally safe*, or *unsafe* features. (We'll explain what we mean by *safety* and by each of our categories later in this chapter.)

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

0.2.1 The *EMC++S* Manifesto

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

Facts (Not Opinions)

This book describes only beneficial usages and potential pitfalls of modern C++ features. The content presented is based on objectively verifiable facts; we explicitly avoid subjective opinion such as the relative merits of design tradeoffs. Although such opinions are often valuable, they are inherently biased toward the author's area of expertise.

Note that *safety* — the rating we use to segregate features by chapter — is the one exception to this objectivity guideline: While the analysis of each feature is itself completely objective, its chapter classification — indicting the relative *safety* of its quotidian use in a typical, large, corporate software-development environment — reflects our combined opinions, backed by decades of real-world, hands-on experience developing large-scale C++ software systems in various contexts.

Elucidation (Not Prescription)

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

Brevity (Not Verbosity)

EMC++S is neither designed nor intended to be an introduction to modern C++. It is a handy reference for expert C++ programmers who have at least a passing knowledge of the features and a desire to perfect their understanding. Our writing style is intentionally tight, with the goal of providing you with facts, concise objective analysis, and cogent, real-world examples and sparing you the task of wading through introductory material. If you are entirely unfamiliar with a feature, we suggest you start with a more elementary and language-centric text such as *The C++ Programming Language* by Bjarne Stroustrup.[2]

Real-World (Not Contrived) Examples

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

Large-Scale (Not ``Toy") Programs

By scale, we are simultaneously capturing two distinct aspects of size: (1) the size (e.g., in bytes, source lines, separate units of release) of the programs, systems, and libraries developed and maintained by a software organization, and (2) the size of 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.[3]

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

0.2.2 What Do We Mean by *Safely*?

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

[2]**stroustrup13**

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

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

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

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

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

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

A *Safe* Feature

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

We categorize the `override` *contextual keyword* as a *safe* feature because it prevents bugs, serves as documentation, cannot easily be misused, and has no serious deficiencies. If someone has heard of this feature and tried to use it and the software compiles, the code base is likely better for it.

A *Conditionally Safe* Feature

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

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

An *Unsafe* Feature

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

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

We deem the `final` *contextual keyword* an *unsafe* feature because the situations in which it would be misused overwhelmingly outnumber those vanishingly few isolated cases in which it is appropriate, let alone valuable. Furthermore, its widespread use would inhibit fine-grained (e.g., hierarchical[5]) reuse, which is critically important to the success of a large organization.[6]

0.3 Modern C++ Language Feature Catalog

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

0.3.1 Organization

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

Chapter 0: *Introduction*	
Chapter 1: *Safe Features*	Catalog of
Chapter 2: *Conditionally Safe Features*	Language
Chapter 3: *Unsafe Features*	Features
Chapter 4: *Parting Thoughts*	

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

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

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

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

  - Workarounds (as needed)

- Cross-references to related features
- External references for further reading

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

0.4 How To Use This Book

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

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

2. **Read the chapters in order but slowly over time.** If you are a less sure-footed developer, understanding and applying first the *safe* features of Chapter 1, followed in time by the *conditionally safe* features of Chapter 2, will allow you to grow into the breadth of useful modern C++ language features, prioritizing those that are least likely to prove problematic.

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

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

The knowledge imbued into this book — irrespective of how it is consumed — can be valuable in many important ways. In addition to helping you become a more knowledgeable and therefore *safer* developer, this book aims to elucidate (to developers and managers alike) which features demand more training, attention to detail, experience, peer review, and so on. The factual, objective presentation style also makes for excellent input into the preparation of coding standards and style guides that suit the particular needs of a company, project, team, or even just a single discriminating developer (which, after all, we all are). Finally, any C++ software development organization that adopts this book wholesale will be taking the first steps toward leveraging modern C++ in a way that maximizes reward while minimizing risks, i.e., by embracing modern C++ *safely*.

Chapter 1

Safe Features

Intro text should be here.

1.1 Attributes

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

1.1.1 Description

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

C++ attribute syntax

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

```
[[attribute_name]]
```

A single annotation can consist of zero or more attributes:

```
[[]]           // Permitted in every position where any attribute is allowed.
[[foo, bar]]   // Equivalent to [[foo]] [[bar]].
```

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

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

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

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

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

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

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

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

C++ attribute placement

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

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

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

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

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

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

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

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

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

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

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

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

[5]It is possible that redundancy of standard attributes will not be an error anymore in the future revisions of the C++ Standard; see **p2156r0** (https://wg21.link/p2156r0).

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

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

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

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

Common compiler-dependent attributes

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

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

Table 1–1: Need caption here

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

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

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

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

1.1.2 Use cases

Eliciting useful compiler diagnostics

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

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

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

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

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

For the code above, GCC produces a useful warning:

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

Hinting at additional optimization opportunities

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

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

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

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

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

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

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

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

Delineating explicit assumptions in code to achieve better optimizations

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

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

The `vectorLerp` function (below) performs *l*inear int*erp*olation (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),
```

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

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

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

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

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

If the implementation of a function tagged with the `[[gnu::pure]]` attribute does not satisfy limitations imposed by it, however, the compiler will not be able to detect this and a runtime defect will be the likely result[12]; see *Potential Pitfalls*.

Using attributes to control external static analysis

Since unknown attributes are ignored by the compiler, external static-analysis tools can define their own custom attributes that can be used to embed detailed information to influence or control those tools without affecting program semantics. For example, the Microsoft-specific `[[gsl::suppress(/* rules */)]]` attribute can be used to suppress unwanted warnings from static analysis tools that verify *Guidelines Support Library*[13] rules. In particular, consider GSL C26481 (Bounds rule #1; see **gslrule26481**), which forbids any pointer arithmetic, instead suggesting that users rely on the `gsl::span` type[14]:

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

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

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

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

[14]`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.

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

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

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

Creating new attributes to express semantic properties

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

```
// lib.h

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

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

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

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

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

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

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

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

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

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

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

1.1.3 Potential Pitfalls

Unrecognized attributes have implementation-defined behavior

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

Such warnings can typically be disabled (e.g., on GCC using `-Wno-attributes`) but if they are, misspellings in even standard attributes will go unreported[15].

Some attributes, if misused, can affect program correctness

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

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

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

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

Suppose that we somehow observed that decorating `myRandom` with the `[[gnu::const]]` attribute occasionally improved runtime performance and, out of ignorance, decided to use it

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

in production. This is clearly a misuse of the `[[gnu::const]]` attribute because the function doesn’t inherently satisfy the requirement of productin the same result when invoked with the same arguments (in this case none). Adding this attribute tells the compiler that it needs not call this function repeatedly and is free to treat the first value returned as a constant for all time.

Not every syntactic location is viable for an attribute

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

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

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

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

1.1.4 Annoyances

None so far.

1.1.5 See Also

- Section??, “??” — Safe C++11 standard attribute for functions that never return control flow to the caller.
- Section3.1, “The `[[carries_dependency]]` Attribute” — Unsafe C++11 standard attribute used to communicate release-consume dependency chain information to the compiler to avoid unnecessary memory fence instructions.
- Section??, “??”: Safe C++11 attribute (with a keyword-like syntax) used to widen the alignment of a type or an object.
- Section1.14, “The `[[deprecated]]` Attribute” — Safe C++14 standard attribute that discourages the use of an entity via compiler diagnostics.

1.1.6 Further Reading

For more information on commonly supported function attributes, see **gccattr**.

1.2 Binary Literals

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

1.2.1 Description

A *binary literal* (e.g., `0b1110`) — much like a hexadecimal literal (e.g., `0xE`) or an octal literal (e.g., `016`) — is a kind of *integer literal* (in this case, having the *decimal* value `14`). A binary literal consists of a `0b` (or `0B`) prefix followed by a nonempty sequence of binary digits (`0` or `1`)[16]:

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

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

[16]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; for more details, see [CITATION TBD].

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

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

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

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

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

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

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

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

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

1.2.2 Use Cases

Bit masking and bitwise operations

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

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

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

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

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

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

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

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

    // ...
};
```

Replicating constant binary data

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

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

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

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

```
                break;

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

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

            // ...
        }
    }
```

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

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

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

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

1.2.3 Potential Pitfalls

None so far

1.2.4 Annoyances

None so far

1.2.5 See Also

- Section 1.11, "Digit Separators" — Safe C++14 feature that allows a developer to (visually) group together digits in a numerical literal to help readability. Often used in conjunction with binary literals.

1.2.6 Further Reading

None so far

1.3 Consecutive Right Angle Brackets

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

1.3.1 Description

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

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

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

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

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

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

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

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

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

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

1.3.2 Use Cases

Avoiding annoying whitespace when composing template types

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

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

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

1.3.3 Potential pitfalls

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

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

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

The easy fix is simply to parenthesize the expression:

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

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

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

Though pathologically rare, the same valid expression can (in theory) have a different interpretation in C++11 than it had when compiled for C++03. Consider the case[20] where the `>>` token is embedded as part of an expression involving templates:

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

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

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

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

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

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

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

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

```
// C++03

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

// C++11

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

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

1.3.4 Annoyances

None so far

1.3.5 See Also

None so far

1.3.6 Further Reading

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

1.4 Deleted Functions

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

1.4.1 Description

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

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

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

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

1.4.2 Use Cases

Suppressing special member function generation

When instantiating an object of user-defined type, **special member functions** that have not been declared explicitly are often[21] generated automatically by the compiler. For certain kinds of types, the notion of **copy semantics** (including **move semantics**[22]) is not meaningful and hence permitting the generation of copy operations is contraindicated.

Consider a class, `FileHandle`, that uses the **RAII** idiom to safely acquire and release an I/O stream. As *copy semantics* are typically not meaningful for such resources, we will want to suppress generation of both the *copy constructor* and *copy assignment operator*. Prior to C++11, there was no direct way to express suppression of *special functions* in C++. The commonly recommended workaround was to declare the two methods `private` and leave them unimplemented, typically resulting in a compile-time (or link-time) error when accessed[23]:

[21] The generation of individual special member functions can be affected by the existence of other user-defined special member functions or by limitations imposed by the specific types of any data members or base types. For more information, see Section 1.13, “Defaulted Special Member Functions.”

[22] The two **special member functions** controlling *move* operations (introduced in C++11) are sometimes implemented as effective optimizations of copy operations and (rarely) with copy operations explicitly deleted; see Section 2.2, “Rvalue References.”

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

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

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

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

    // ...
};
```

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

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

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

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

    // ...
};
```

Preventing a particular implicit conversion

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

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

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

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

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

Pernicious user errors can now be reported during compilation:

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

Preventing all implicit conversions

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

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

    // ...
};

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

Invoking `send` with an `int` (noted with (1) in the code above) or any integral type (other than `unsigned char`[24]) that promotes to `int` (2) will map exclusively to the deleted `send(int)` overload; all other integral (3 & 4) and floating-point types (5) are convertible to both (via a **standard conversion**) and hence will be ambiguous. An explicit cast to `unsigned char` (6) can always be pressed into service if needed.

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

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

Best practices notwithstanding,[25] it can be cost-effective (in the short term) to provide an elided “view” on a concrete class for (trusted) clients. Imagine a class `AudioStream` designed to play sounds and music that — in addition to providing basic “play” and “rewind” operations — sports a large, robust interface:

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

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

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

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

If the need for a `ForwardAudioStream` type persists, we can always consider reimplementing it more carefully later.[26]

1.4.3 Potential Pitfalls

None so far

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

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

1.4.4 Annoyances

None so far

1.4.5 See Also

- Section 1.13, "Defaulted Special Member Functions" — Companion safe C++11 feature that enables *defaulting* (as opposed to *deleting*) special member functions
- Section 2.2, "Rvalue References" — Conditionally safe C++11 feature that introduces the two *move* variants to *copy* special member functions

1.4.6 Further Reading

None so far

1.5 override

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

1.5.1 Description

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

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

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

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

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

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

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

1.5.2 Use Cases

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

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

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

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

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

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

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

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

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

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

1.5.3 Potential Pitfalls

Lack of consistency across a code base

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

1.5.4 Annoyances

None so far

1.5.5 See Also

None so far

1.5.6 Further Reading

None so far

1.6 Compile-Time Assertions (static_assert)

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

1.6.1 Description

Assumptions — whether we explicitly document them or not — are inherent in every program. A common way of validating certain assumptions at runtime is to use the classic assert macro found in <cassert>. Such runtime assertions are not always ideal because (1) the program must already be built and running for them to even have a chance of being triggered and (2) executing a **redundant check** at runtime typically[27] results in a slower program. Being able to validate an assertion at compile time avoids several drawbacks:

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

Syntax and semantics

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

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

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

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

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

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

```
    }

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

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

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

Evaluation of static assertions in templates

The C++ Standard does not explicitly specify at precisely what point (during the compilation process) static assertion declarations are evaluated.[28] In particular, when used within the body of a template, a `static_assert` declaration might not be evaluated until **template instantiation time**. In practice, however, a `static_assert` that does not depend on any template parameters is essentially always[29] evaluated immediately — i.e., as soon as it is parsed and irrespective of whether any subsequent template instantiations occur:

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

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

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

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

(However, see *Potential Pitfalls*, below.) 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
```

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

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

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

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

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

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

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

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

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

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

1.6.2 Use Cases

Verifying assumptions about the target platform

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

```
#include <ctype>  // CHAR_BIT

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

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

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

Preventing misuse of class and function templates

Static assertions are often used in practice to constrain class or function templates to prevent their being instantiated with unsupported types by either (1) substantially improving compile-time diagnostics[30] or, more critically, (2) actively avoiding erroneous runtime behavior.

As an example, consider the SmallObjectBuffer<N> class templates, which provide storage for arbitrary objects whose size does not exceed N[31]:

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

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

    // ...
};
```

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

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

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

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

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

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

```
#include <ctype>  // CHAR_BIT

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

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

1.6.3 Potential Pitfalls

Static assertions in templates can trigger unintended compilation failures

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

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

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

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

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

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

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

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

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

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

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

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

Misuse of static assertions to restrict overload sets

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

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

template <typename T>
void process(const T& x)  // (2) compile-time error: redefinition of function
```

```
{
    static_assert(sizeof(T) > 32, "Overload for big types");     // BAD IDEA
    // ... (process big types)
}
```

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

To achieve the goal of removing (up front) a specialization from consideration, we will need to employ **SFINAE**. To do that, we must instead find a way to get the failing compile-time expression to be part of the function's **declaration**[32]:

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

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

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

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

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

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

```
void client()
```

[32]**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.

```
{
    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 1.7, "Trailing Function Return Types."

1.6.4 Annoyances

Mandatory string literal

Many compilation failures caused by static assertions are self-explanatory since the offending line (which necessarily contains the predicate code) is displayed as part of the compiler diagnostic. In those situations, the message required[33] as part of `static_assert`'s grammar is redundant:

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

Developers commonly provide an empty string literal in these cases:

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

1.6.5 See Also

- Section 1.7, "Trailing Function Return Types" — Safe C++11 feature that allows fine-grained control over overload resolution by enabling **expression SFINAE** as part of a function's **declaration**

1.6.6 Further reading

None so far

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

1.7 Trailing Function Return Types

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

1.7.1 Description

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

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

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

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

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

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

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

The same advantage would apply to a nonmember function[34] 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.
```

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

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`[35]:

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

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

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

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

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

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

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

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

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

1.7.2 Use Cases

Function template whose return type depends on a parameter type

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

```
template <typename A, typename B, typename F>
```

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

```
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 **??**, “**??**”).

Avoiding having to qualify names redundantly in return types

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

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

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

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

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

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

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

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

Improving readability of declarations involving function pointers

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

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

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

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

Using this divide-and-conquer approach, writing a **higher-order function** that returns a pointer to a function, member function, or data member as its return type[36] becomes fairly straightforward.

1.7.3 Potential Pitfalls

None so far

1.7.4 Annoyances

None so far

1.7.5 See Also

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

1.7.6 Further Reading

None so far

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

1.8 Unrestricted Unions

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

1.8.1 Description

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

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

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

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

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

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

As an added safety measure, any non-trivial **special member function** defined — either implicitly or explicitly — for any *member* of a `union` results in the compiler implicitly deleting (see Section 1.4, “Deleted Functions”) the corresponding **special member function** of the `union` itself:

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

    // Implicitly deleted special member functions of U1:
    /*
        U1()                        = delete; // due to explicit Nt::Nt()
        U1(const U1&)               = delete; // due to implicit Nt::Nt(const Nt&)
        ~U1()                       = delete; // due to explicit Nt::~Nt()
        U1& operator=(const U1&) = delete; // due to implicit
                                           // Nt::operator=(const Nt&)
    */
};
```

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

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

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

    bool d_useInt;  // discriminator

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

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

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

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

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

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

We can restore these implicitly deleted special member functions too, simply by adding appropriate copy-constructor and assignment-operator definitions for `U2` explicitly[37]:

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

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

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

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

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

        return *this;
    }
};
```

[37]Attempting to restore a `union`'s implicitly deleted special member functions by using the `= default` syntax (see Section 1.13, "Defaulted Special Member Functions") will still result in their being deleted because the compiler cannot know which member of the union is active without a discriminator.

1.8.2 Use Cases

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

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

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

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

    // ...

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

        return ParseResult(oss.str());
    }

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

        return ParseResult(oss.str());
    }

    // ...

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

The implementation above relies on `ParseResult` being able to hold a value of type either `int` or `std::string`. By encapsulating a C++ `union` and a Boolean[38] *discriminator* as part of the `ParseResult` **sum type**, we can achieve the desired semantics:

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

    bool d_isError;  // discriminator

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

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

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

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

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

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

Placement `new` and explicit destructor calls are, however, required for destruction and both copy operations[39]:

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

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

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

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

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

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

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

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

In practice, `ParseResult` would typically be defined as a template and renamed to allow any arbitrary result type `T` to be returned or else implemented in terms of a more general **sum type** abstraction.[40]

1.8.3 Potential Pitfalls

Inadvertent misuse can lead to latent undefined behavior at runtime

When implementing a type that makes use of an unrestricted union, forgetting to initialize a non-trivial object (using either a *member initialization list* or **placement `new`**) or accessing a different object than the one that was actually initialized can result in tacit **undefined behavior**. Although forgetting to destroy an object does not necessarily result in **undefined**

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

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.[41]

1.8.4 Annoyances

None so far

1.8.5 See Also

- Section 1.4, "Deleted Functions" — Safe C++11 feature that forbids the invocation of a particular function. Similar effects to deleting a function happen when we specify a special function within a subobject of a union or when a class has such a union as a data member.

1.8.6 Further Reading

None so far

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

1.9 Feature Title

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

1.9.1 Description

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

C++ attribute syntax

Lorem ipsum dolor sit amet, consectetur adipiscing elit.[42] Suspendisse non nisi ex. Ut non sollicitudin est. Donec laoreet, risus pretium egestas condimentum, ex eros tempor diam, quis congue nisi metus in tellus. Proin tempor ac lectus nec elementum. Maecenas augue turpis, pellentesque sed eros sit amet, tincidunt pretium nulla. Nunc lacus ligula, ullamcorper a porta eget, fringilla sed orci. Maecenas eget ultricies risus. Donec varius vehicula diam.

Nested unordered list:

- One
  - Sub
  - Sub
- Two
- Three

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

Nested ordered list:

1. One
   (a) Sub
   (b) Sub
2. Two

[42] Authors’ Note: We will have some footnotes that are authors’ notes.

3. Three

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

Listing 1.1: code 1

```
void code()
{

}
```

Listing 1.2: code 2

```
void code()
{

}
```

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

Listing 1.3: code 1 with a long and wrapping title

```
void code()
{

}
```

Listing 1.4: code 2 with a long and wrapping title

```
void code()
{

}
```

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

1.9.2 Appendix to the Feature

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

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

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

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

1.10 Aggregate Member Initialization Relaxation

C++14 enables the use of **aggregate initialization** with classes employing Default Member Initializers (see Section 2.3, "Default Member Initializers").

1.10.1 Description

Prior to C++14, classes that made use of Default Member Initializers — i.e., initializers that appear directly within the scope of the class — were not considered **aggregate** types:

```
struct S                // aggregate type in C++14 but not C++11
{
    int i;
    bool b = false;     // uses default member initializer
};

struct A                // aggregate type in C++11 and C++14
{
    int  i;
    bool b;             // does not use default member initializer
};
```

Because `A` (but not `S`) is considered an **aggregate** in C++11, instances of `A` can be created via **aggregate initialization** (whereas instances of `S` cannot):

```
A a{100, true};  // OK in both C++11 and C++14
S s{100, true};  // error in C++11; OK in C++14
```

As of C++14, the requirements for a type to be categorized as an **aggregate** are relaxed, allowing classes employing default member initializers to be considered as such; hence both `A` and `S` are considered **aggregates** in C++14 and eligible for **aggregate initialization**:

```
void f()
{
    S s0{100, true};        // OK in C++14 but not in C++11
    assert(s0.i == 100);    // set via explicit aggregate initialization (above)
    assert(s0.b == true);   // set via explicit aggregate initialization (above)

    S s1{456};              // OK in C++14 but not in C++11
    assert(s1.i == 456);    // set via explicit aggregate initialization (above)
    assert(s1.b == false);  // set via default member initializer
}
```

In the code snippet above, the C++14 aggregate `S` is initialized in two ways: `s0` is created using aggregate initialization for both data members; `s` is created using aggregate initialization for only the first data member (and the second is set via its default member initializer).

1.10.2 Use Cases

Configuration `struct`s

Aggregates in conjunction with Default Member Initializers can be used to provide concise customizable configuration `struct`s, packaged with typical default values. As an example, consider a configuration `struct` for a 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[43]:

```
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.
    }
}

// ...
```

1.10.3 Potential Pitfalls

None so far

[43]In C++20, the Designated Initializers feature adds flexibility (e.g., for configuration `struct`s, such as `HTTPRequestHandlerConfig`) by enabling explicit specification of the names of the data members:

```
HTTPRequestHandlerConfig lowTimeout{.timeout = 15};
    // maxQueuedRequests, minThreads, and maxThreads have their default value.

HTTPRequestHandlerConfig highPerformance{.timeout = 120, .maxThreads = 16};
    // maxQueuedRequests and minThreads have their default value.
```

1.10.4 Annoyances

Syntactical ambiguity in the presence of brace elision

During the initialization of multilevel **aggregates**, braces around the initialization of a nested aggregate types can be omitted (**brace elision**):

```
struct S
{
    int arr[3];
};

S s0{{0, 1, 2}};  // OK, nested arr initialized explicitly
S s1{0, 1, 2};    // OK, brace elision for nested arr
```

The possibility of **brace elision** creates an interesting syntactical ambiguity when used alongside **aggregates** with Default Member Initializers. Consider a `struct X` containing three data members, one of which has a default value:

```
struct X
{
    int a;
    int b;
    int c = 0;
};
```

Now, consider various ways in which an array of elements of type `X` can be initialized:

```
X xs0[] = {{0, 1}, {2, 3}, {4, 5}};
    // OK, clearly 3 elements having the respective values:
    // {0, 1, 0}, {2, 3, 0}, {4, 5, 0}

X xs1[] = {{0, 1, 2}, {3, 4, 5}};
    // OK, clearly 2 elements with values:
    // {0, 1, 2}, {3, 4, 5}

X xs2[] = {0, 1, 2, 3, 4, 5};
    // ...?
```

Upon seeing the definition of `X2`, a programmer not versed in the details of the C++ Language Standard might be unsure as to whether the initializer of `xs2` is three elements (like `xs0`) or two elements (like `xs1`). The Standard is, however, clear that the compiler will interpret `xs2` the same as `xs1`, and, thus, the default values of `X::c` for the two array elements will be replaced with `2` and `5`, respectively.

1.10.5 See Also

- Section 2.3, "Default Member Initializers" — Conditionally safe C++11 feature that allows developers to provide a default initializer for a data member directly in the definition of a class

1.10.6 Further Reading

None so far

1.11 Digit Separators

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

1.11.1 Description

A *digit separator* — i.e., an instance of the single-quote character (') — may be placed anywhere within a numeric literal to visually separate its digits without affecting its value:

```
int          i = -12'345;                  // same as -12345
unsigned int u = 1'000'000u;               // same as 1000000u
long         j = 5'0'0'0'0'0L;             // same as 500000L
long long    k = 9'223'372'036'854'775'807; // same as 9223372036854775807
float        f = 10'00.42'45f;             // same as 1000.4245f
double       d = 3.1415926'53589793;       // same as 3.141592653589793
long double  e = 3.1415926'53589793'23846; // same as 3.14159265358979323846
int        hex = 0x8C25'00F9;              // same as 0x8C2500F9
int        oct = 044'73'26;                // same as 0447326
int        bin = 0b1001'0110'1010'0111;    // same as 0b1001011000110001
```

Multiple digit separators within a single literal are allowed but they cannot be contiguous, nor can they appear either before or after the *numeric* part (i.e., digit sequence) of the literal[44]:

```
int e0 = 10''00;  // Error: consecutive digit separators.
int e1 = -'1000;  // Error: before numeric part.
int e2 = 1000'u;  // Error: after numeric part.
int e3 = 0x'abc;  // Error: before numeric part.
int e4 = 0'xdef;  // Error: way before numeric part.
int e5 = 0'89;    // Error: non-octal digits.
int e6 = 0'67;    // OK, valid octal literal.
```

As a side note, remember that, on some platforms, an integer literal that is too large to fit in a `long long int`, but does fit in an `unsigned long long int` might generate a warning:[45]

```
unsigned long long big1 = 9'223'372'036'854'775'808;  // 2^63
    // warning: integer constant is so large that it is an
    // unsigned long long big1 = 9'223'372'036'854'775'808;
    //                           ^~~~~~~~~~~~~~~~~~~~~~~~~
```

Such warnings can typically be suppressed by adding a `ull` suffix to the literal:

```
unsigned long long big2 = 9'223'372'036'854'775'808ull;  // OK
```

Warnings like the one above, however, are not typical when the implied precision of a floating-point literal exceeds what can be represented:

[44]Although the leading `0x` and `0b` prefixes for hexadecimal and binary literals, respectively, are not considered part of the *numeric* part of the lateral, a leading `0` in an octal literal is.
[45]Tested on GCC 7.4.0.

```
  float reallyPrecise = 3.141'592'653'589'793'238'462'643'383'279'502'884;  // OK
      // Everything after 3.141'592'6 is typically ignored silently.
```

For more information, see *Appendix: Silent loss of precision in floating-point literals*, below.

1.11.2 Use Cases

Grouping digits together in large constants

When embedding large constants in source code, consistently placing digit separators (e.g. every thousand) might improve readability, as illustrated in Table 1–2

Table 1–2: Use of digit separators to improve readability

Without digit separator	With digit separators
`10000`	`10'000`
`100000`	`100'000`
`1000000`	`1'000'000`
`1000000000`	`1'000'000'000`
`18446744073709551615ull`	`18'446'744'073'709'551'615ull`
`1000000.123456`	`1'000'000.123'456`
`3.1415926535897932384621`	`3.141'592'653'589'793'238'462l`

Use of digit separators is especially useful with *binary literals*:

Table 1–3: Use of digit separators in binary data

Without digit separator	With digit separators
`0b1100110011001100`	`0b1100'1100'1100'1100`
`0b0110011101011011`	`0b0110'0111'0101'1011`
`0b1100110010101010`	`0b11001100'10101010`

1.11.3 Potential Pitfalls

None so far.

1.11.4 See Also

- Section 1.2, "Binary Literals" — Safe C++14 feature representing a binary constant for which digit separators are commonly used to group bits in octets (*bytes*) or quartets (*nibbles*).

1.11.5 Further reading

This is taken from wikipedia, but they give a reference: https://en.wikipedia.org/wiki/Double-precision_floating-point_format The 53-bit significand precision gives from 15 to 17 significant decimal digits precision (2—53 1.11 × 10—16). If a decimal string with at most 15 significant digits is converted to IEEE 754 double-precision representation, and then converted back to a decimal string with the same number of digits, the final result should match the original string. If an IEEE 754 double-precision number is converted to a decimal string with at least 17 significant digits, and then converted back to double-precision representation, the final result must match the original number.[1] – Page 4.

[1] William Kahan (1 October 1997). "Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic" (PDF). Archived (PDF) from the original on 8 February 2012. https://people.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF https://web.archive.org/web/20120208075518/http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF

1.11.6 Appendix: Silent loss of precision in floating-point literals

Just because we can keep track of precision in floating-point literals doesn't mean that the compiler can. As somewhat of 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 define by reference to the C standard: [46]

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[^IEEE_754] (as most do), a `float` can typically represent up to 7 significant decimal digits accurately, while a `double` typically nearly 15 decimal digits of precision. For any given program, `long double` is required to hold whatever a `double` can hold, but is typically larger (e.g., 10, 12 or 16 bytes) and typically adds at least 5 decimal digits of precision (i.e., supports a total of at last 20 decimal digits). A table summarizing typical precisions for various IEEE-conforming floating-point types is presented for convenient reference in Figure 3. The actual bounds on a given platform can be found using the standard `std::numeric_limits` class template found in `<limits>`.

		significant[^]	Decimal	exponent	Dynamic
Name	Common name	bits	Digits	bits	Range
binary16	Half precision	11	3.31	5	~6.50e5
binary32	Single precision	24	7.22	8	~3.4e38
binary64	Double precision	53	15.95	11	~1.e308
binary80	Extended precision	69	20.77	11	~10^308

[46] References for this footnote:

- ISO/IEC 14882:2020 [basic.fundamental] Fundamental types (6.8.1p12)
- ISO/IEC 14882:2020 [numeric.limits.members] `numeric_limits` members 17.3.5.1 (many footnotes)
- ISO/IEC 14882:2020 [cfloat.syn] Header `<cfloat>` synopsis (17.3.7p1)

See also:

- ISO/IEC 9899:2018 5.2.4.2.2 Characteristics of floating types `<float.h>`

```
binary128   Quadruple precision    113        34.02       15      ~10^4932
```

[^]: Note that the most significant bit of the **mantissa** is always a `1` and, hence, is not stored explicitly, leaving 1 additional bit to represent the sign of the overall floating-point value (the sign of exponent is encoded using **excess-*n*** notation).

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

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:[47]

```
static_assert(3.1415926535f != 3.141592653f, "too precise for float");
    // This assert will fire on a typical platform.

static_assert(3.141592653f != 3.14159265f, "too precise for float");
    // This assert too will fire on a typical platform.

static_assert(3.14159265f != 3.1415926f, "too precise for float");
    // This assert will NOT fire on a typical platform.

static_assert(3.1415926f != 3.141592f, "too precise for float");
    // This assert too will NOT fire on a typical platform.
```

If the values are *not* the same then that floating-point type *can* make use of the precision suggested by original literal; if they *are* the same, however, then it is likely that the available precision has been exceeded. Iterative use of this technique by developers can help them to empirically narrow down the maximal number of decimal digits a particular platform will support for a particular floating-point type and value.

One final useful tidbit pertains to the safe (lossless) conversion between binary and decimal floating-point representations; note that "Single" (below) corresponds to a single-precision IEEE-754-conforming (32-bit) `float`:[48]

```
"If a decimal string with at most 6 sig. dec. is converted to Single and then converted back to the
 same number of sig. dec., then the final string should match the original. Also, ...

 If a Single Precision floating-point number is converted to a decimal string with at least 9 sig.
 dec. and then converted back to Single, then the final number must match the original."
```

The ranges corresponding to 6–9 for a single-precision (32-bit) `float` (described above), when applied to a double-precision (64-bit) `double` and a quad-precision (128-bit) `long long`, are 15–17 snd 33–36, respectively.

[47]Note that affixing the `f` (*literal suffix*) to the 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));
```

[48]**kahan97**, p. 4

1.12 Variable Templates

Effective use of traditional template syntax to define, in namespace or class (but not function) scope, a family of like-named variables that can subsequently be instantiated explicitly.

1.12.1 Description

By beginning a variable declaration with the familiar **template-head** syntax — e.g., `template <typename T>` — we can create a *variable template*, which defines a family of variables having the same name (e.g., `typeid`):

```
template <typename> int typeId;  // template variable defined at file scope
```

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

```
void f1()
{
    typeId<bool> = -1;    // typeId<bool> is an int
    typeId<char> = 1000;  // typeId<char> is an int
    typeId<void> = -666;  // typeId<void> is an int

    assert(typeId<bool> ==   -1);
    assert(typeId<char> == 1000);
    assert(typeId<void> == -666);
}
```

In the example above, the type of each instantiated variable — i.e., `typeId<bool>` and `typeId<char>` — is `int`. Such need not be the case:[49]

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

In the example above, the type of the instantiated non-`const` variable is that of its (type) argument, and its (mutable) value is initialized to the best approximation of π offered by that type:

```
void f2()
{
    bool        pi_as_bool        = 1;                      // ( 1 bit)
    int         pi_as_int         = 3;                      // (32 bits)
    float       pi_as_float       = 3.1415927;              // (32 bits)
    double      pi_as_double      = 3.141592653589793;      // (64 bits)
    long double pi_as_long_double = 3.1415926535897932385;  // (80 bits)

    assert(pi<bool>        == pi_as_bool);
    assert(pi<int>         == pi_as_int);
    assert(pi<float>       == pi_as_float);
    assert(pi<double>      == pi_as_double);
```

[49]Use of *`constexpr`* would allow the instantiated variables to be usable as a constant in a compile-time context (see *Use Cases*, below).

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

For examples involving immutable variable templates, see *Use Cases* below.

Variable templates, like **C-style functions**, may be declared at *namespace-scope* or as `static` members of a `class`, `struct`, or `union`, but are not permitted as non-`static` members nor at all in function scope:

```
template <typename T> T vt1;            // OK (external linkage)
template <typename T> static T vt2;     // OK (internal linkage)

namespace N
{
    template <typename T> T vt3;          // OK (external linkage)
    template <typename T> T vt4;          // OK (internal linkage)
}

struct S
{
    template <typename T> T vt5;        // error: not static
    template <typename T> static T vt6;  // OK (external linkage)
};

void f3()  // Variable templates cannot be defined in functions.
{
    template <typename T> T vt7;        // compile-time error
    template <typename T> static T vt8;  // compile-time error

    vt1<bool> = true;                   // OK (to use them)
}
```

Like other templates, variable templates may be defined with multiple parameters consisting of arbitrary combinations of type and non-type parameters (including a **parameter pack**):

```
namespace N
{
    template <typename V, int I, int J> V factor;  // namespace scope
}
```

Variable templates can even be defined recursively (but see *Potential Pitfalls*, below):

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

template <> const int sum<0> = 0;  // base-case specialization

void f()
{
    std::cout << sum<4> << '\n';  // Prints 10.
    std::cout << sum<5> << '\n';  // Prints 15.
    std::cout << sum<6> << '\n';  // Prints 21.
}
```

Note that variable templates do not enable any novel patterns — anything that can be achieved using them could also have been achieved in C++11 along with some additional boilerplate. The initial `typeId` example could have instead been implemented using a `struct`:

```
template <typename> struct TypeId { static int value; };
```

And used with just a bit more syntax:

```
 void f1b()
{
    TypeId<bool>::value = -1;    // TypeId<bool>::value is an int
    TypeId<char>::value = 1000;  // TypeId<char>::value is an int
    TypeId<void>::value = -666;  // TypeId<void>::value is an int

    assert(TypeId<bool>.value ==   -1);
    assert(TypeId<char>.value == 1000);
    assert(TypeId<void>.value == -666);
}
```

1.12.2 Use Cases

Parametrized constants

A common effective use of variable templates is in the definition of type-parametrized constants.

As discussed in the *Description* (above), the mathematical constant π will serve as our example. Here we will want to initialize the constant as part of the variable template (the literal chosen is the shortest decimal string to do so for an 80-bit `long double` accurately)[50]:

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

Notice that we have elected to use *`constexpr`* (from C++11) in place of a classic `const` as a stronger guarantee that the provided initializer is a compile-time constant and that `pi` itself will be usable as part of a constant expression.

With the definition above, it is possible to 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));
}
```

[50]For portability, a floating-point literal value of π that provides sufficient precision for the longest `long double` on any relevant platform (e.g., 128 bits or 34 decimal digits: `3.141'592'653'589'793'238'462'643'383'279'503`) should be used; see Section 1.11, "Digit Separators."

Reducing verbosity of type traits

A **type trait** is a empty type carrying compile-time information about one or more aspects of another type. The way in which type traits have been specified historically has been to define a class template having the trait name and a public `static` (or `enum`) data member, that is conventionally called `value`, which is initialized in the primary template to `false`. Then, for each type that wants to advertise that it has this trait, the header defining the trait is included and the trait is specialized for that type, initializing `value` to `true`. We can achieve precisely this same usage pattern replacing a trait `struct` with a variable template whose name represents the type trait and whose type of variable itself is always `bool`. Preferring variable templates in this use case decreases the amount of **boilerplate code** — both at the point of definition and at the call site.[51]

Consider, for example, a boolean trait designating whether a particular type `T` can be serialized to JSON:

```
// isSerializableToJson.h

template <typename T>
constexpr bool isSerializableToJson = false;
```

The header above contains the general variable template trait that, by default, concludes that a given type is not serializable to JSON. Next we consider the streaming utility itself:

```
// serializeToJson.h
#include <isSerializableToJson.h>  // General trait variable template

template <typename T>
JsonObject serializeToJson(const T& object)  // Serialization function template
{
    static_assert(isSerializableToJson<T>,
                  "T must support serialization to JSON.");

    // ...
}
```

Notice that have used the the C++11 ***`static_assert`*** feature to ensure that any type used to instantiated this function will have specialized (see below) 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:

[51] As of C++17, the Standard Library provides a more convenient way of inspecting the result of a type trait, by introducing variable templates named the same way as the corresponding traits, but with an additional `_v` suffix:

```
// C++11/14
std::is_default_constructible<T>::value

// C++17
std::is_default_constructible_v<T>
```

```
// companyData.h
#include <isSerializableToJson.h>  // General trait variable template

struct CompanyData { /* ... */ };  // Type to be JSON serialized

template <>
constexpr bool isSerializableToJson<CompanyData> = true;
    // Let anyone who needs to know that this type is JSON serializable.
```

Finally, our `client` function incorporates all of the above and attempts to serialize both a `CompanyData` object and a `std::map<int, char>>`:

```
// client.h
#include <isSerializableToJson.h>  // General trait template
#include <companyData.h>           // JSON serializable type
#include <serializeToJson.h>       // Serialization function
#include <map>                     // std::map (not JSON serializable)

void client()
{
    auto jsonObj0 = serializeToJson<CompanyData>();         // OK
    auto jsonObj1 = serializeToJson<std::map<int, char>>(); // Compile-time error
}
```

In the `client()` function above, `CompanyData` works fine, but, because the variable template `isSerializableToJson` was never specialized to be `true` for type `std::map<int, char>>` the client header will — as desired — fail to compile.

1.12.3 Potential Pitfalls

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

If you ever find someone who thinks they really know there stuff and you need a “good” C++ interview question, considering asking them why the example (below), having no undefined behavior, may (and might actually) produce different results with popular compilers:[52]

```
#include <iostream>

template <int N>
int fib = fib<N - 1> + fib<N - 2>;

template <> int fib<2> = 1;
template <> int fib<1> = 1;

int main()
{
    std::cout << fib<4> << '\n';  // 3 expected
    std::cout << fib<5> << '\n';  // 5 expected
    std::cout << fib<6> << '\n';  // 8 expected
```

[52]For example gcc version 4.7.0 (2017) produces the expected results whereas clang version 11 (2019?) produces 1, 3, 4, respectively.

```
    return 0;
}
```

The didactic value in answering this question dwarfs any potential practical value that recursive template variable instantiation can offer. First, consider that this same issue could, in theory, have occurred in C++03 using nested `static` members of a `struct`:

```
#include <iostream>

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

template <> struct Fib<2> { static int value; };  // BAD IDEA: not const
template <> struct Fib<1> { static int value; };  // BAD IDEA: not const

template <int N> int Fib<N>::value = Fib<N - 1>::value + Fib<N - 2>::value;
int Fib<2>::value = 1;
int Fib<1>::value = 1;

int main()
{
    std::cout << Fib<4>::value << '\n';  // 3 expected
    std::cout << Fib<5>::value << '\n';  // 5 expected
    std::cout << Fib<6>::value << '\n';  // 8 expected

    return 0;
};
```

The problem did not manifest, however, because the simpler solution of using `enum`s (below) obviated separate initialization of the local `static` and didn't admit the possibility of failing to make the initializer a compile-time constant:

```
#include <iostream>

template <int N> struct Fib
{
    enum { value = Fib<N - 1>::value + Fib<N - 2>::value };  // OK - const
};

template <> struct Fib<2> { enum { value = 1 }; };           // OK - const
template <> struct Fib<1> { enum { value = 1 }; };           // OK - const

int main()
{
    std::cout << Fib<4>::value << '\n';  // 3 guaranteed
    std::cout << Fib<5>::value << '\n';  // 5 guaranteed
    std::cout << Fib<6>::value << '\n';  // 8 guaranteed
    return 0;
};
```

It was not until C++14, that the *variable templates* feature readily exposed this latent pitfall involving recursive initialization of non-`const` variables. The root cause of the instability is that the relative order of the initialization of the (recursively generated) variable instantiations is not guaranteed because they are not defined explicitly *within the same translation unit*. The magic sauce that makes everything work is the C++ language requirement that any variable that is declared `const` and initialized with a compile-time constant is itself to be treated as a compile-time constant within the translation unit. This compile-time-constant propagation requirement imposes the needed ordering to ensure that the expected results are portable to all conforming compilers:

```
#include <iostream>

template <int N>
const int fib = fib<N - 1> + fib<N - 2>;  // OK - compile-time const.

template <> const int fib<2> = 1;         // OK - compile-time const.
template <> const int fib<1> = 1;         // OK - compile-time const.

int main()
{
    std::cout << fib<4> << '\n';  // Guaranteed to print out 3.
    std::cout << fib<5> << '\n';  // Guaranteed to print out 5.
    std::cout << fib<6> << '\n';  // Guaranteed to print out 8.

    return 0;
}
```

Note that replacing each of the three `const` keywords with ***`constexpr`*** in the example above also achieves the desired goal and does not consume memory in the **static data space**.

1.12.4 Annoyances

Variable templates do not support template template parameters

While it is possible for a class or function template to accept a *template template class parameter*, there is no equivalent construct for variable templates[53]:

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

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

S<vt> s1;  // compile-time error
```

It might therefore be necessary to provide a wrapper `struct` around a variable template in case it needs to be passed to an interface accepting a **template template parameter**:

[53] A paper by Mateusz Pusz, proposed for C++23, aims to increase consistency between variable templates and class templates when used as template template parameters; see **p2008r0**.

```
template <typename T>
struct Vt { static constexpr T value = vt<T>; }

S<Vt> s2;  // OK
```

1.12.5 See Also

- Section **??**, "**??**" — Conditionally safe C++11 feature providing an alternative to `const` template variables that can reduce unnecessary consumption of that **static data space**.

1.12.6 Further Reading

None so far.

1.13 Defaulted Special Member Functions

placeholder text.........

1.14 The `[[deprecated]]` Attribute

placeholder text.........

Chapter 2

Conditionally Safe Features

2.1 `auto`

placeholder text.........

2.2 Rvalue References

placeholder text..........

2.3 Default Member Initializers

placeholder text.........

Chapter 3

Unsafe Features

3.1 The `[[carries_dependency]]` Attribute

placeholder text.........

3.2 Function Return Type Deduction

placeholder text........

Chapter 4

Parting Thoughts

Bibliography

cpp11

cppcoreguidelines
C++ Core Guidelines

cwg2354

gccattr

gsl
Guidelines Support Library

intel16

intelsse17

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

lakos17
John Lakos,

lakos19
John Lakos,

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

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

Bibliography

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

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

xxxxxxxxxx
p2156r0
https://wg21.link/p2156r0

puszxx
Mateusz Pusz.p2008r0.

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

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

Glossary

aggregate

aggregate initialization

alignment

automatic variables

boilerplate code

brace elision

C-style functions

cache hit

cache line

cache miss

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

Compile-time dispatch
TODO

Concepts

Glossary

conditionally supported

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

Contextual 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.

Copy semantics
TODO

data member

Declaration
TODO

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

delegating constructor

diffusion

direct mapped

Entity
TODO

excess-*n*

Expression SFINAE
TODO

Glossary

extended alignment

false sharing

fragmentation

fully associative

fully constructed
fully constructed

fundamental alignment

Fundamental type
TODO

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

higher-order function

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

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

ill formed, no diagnostic required

implementation-defined

Glossary

instantiation time

Integer literal
TODO

integral constant expression

integral promotion

insulate

Insulation
TODO

locality of reference

mantissa

maximal alignment

member initialization lists

member initializer list

move operations

Move semantics
TODO

natural alignment

nontrivial constructor

Nontrivial special member function
TODO

null address

Overriding
TODO

parameter pack

partially constructed

Placement
TODO

placement new
TODO

POD type
TODO

prvalue

RAII
"Resource Acquisition is Initialization"

Range
TODO

Redundant check
TODO

Reference type
TODO

set associative

SFINAE

Glossary

TODO

Signature
TODO

Special member function
TODO

standard conversion

static data space

String literal
TODO

Structural inheritance
TODO

Sum type
Abstract data type allowing the representation of one of multiple possible alternative types. Each alternative has its own type (and state), and only one alternative can be "active" at any given point in time. Sum types automatically keep track of which choice is "active," and properly implement value-sematic special member functions (even for non-trivial types). They can be implemented efficiently as a C++ `class` using a C++ `union` and a separate (integral) discriminator. This sort of implementation is commonly referred to as a discriminating (or "tagged") union.

template-head

Template instantiation time
TODO

Template instantiation
TODO

template template parameter

thrashing

TLB

Trivial type
TODO

typedef

Type trait
TODO

UDT

Undefined behavior
TODO

user-defined type

value category

value-semantic

variable

well formed

working set

Index

Index