Chapter 1

Safe Features

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

Chapter 1 Safe Features

sec-safe-cpp14

Chapter 2

Conditionally Safe Features

ch-conditional

sec-conditional-cpp11 Intro text should be here.

Chapter 2 Conditionally Safe Features

sec-conditional-cpp14

Chapter 3

Unsafe Features

ch-unsafe

sec-unsafe-cpp11 Intro text should be here.

Preventing Overriding and Derivation

riding-and-derivation final

The **final** specifier can be used to disallow either (1) overriding one or more virtual member functions from within derived types or (2) deriving from a type altogether.

Description

description

The ability to extend an arbitrary (**class** or **struct**) user-defined type (UDT) via inheritance and then to override any virtual functions declared therein is a hallmark of the C++ object-oriented model. There may, however, be cases where the author of such a UDT will find a legitimate need to intentionally restrict clients' abilities in this regard. The **final** specifier serves such a purpose.

When applied to a virtual-function declaration, **final** prevents derived-class authors from overriding that specific function. When used on a virtual function, **final** is syntactically similar to the override specifier (see Section 1.1."??" on page ??) but with different semantics. Separately, **final** can be applied to the declaration of a user-defined type as a whole, thereby preventing prospective clients from deriving from it. Finally, note that **final**, like **override**, is only a **contextual keyword** — i.e., an identifier with special meaning — and can still be used as a C++ identifier wherever other identifiers can be syntactically used, e.g., to name types, objects, functions, and so on:

```
struct final final                              // struct named "final"
{
    final() = default;                          // default constructor
    virtual ~final() final;                     // final destructor
};
struct S1 { ::final *final; };                  // data member named "final"
struct S2 { virtual ::final final() final; };   // function named "final"
final final;                                    // object named "final"
```

The example above is syntactically legal but not recommended. The example defines a number of different entities named `final`, many of mhich have the **final** specifier attached to them.

`final` virtual member functions

tual-member-functions

When applied to a **virtual**-function declaration, the **final** specifier prevents derived-class authors from *overriding* that function, i.e., from within a derived class:

```
struct B0  // Each function in B0 is explicitly declared virtual.
{
    virtual void f();
    virtual void g() final;  // prevents overriding in derived classes
    virtual void g() const;
};

struct D0 : B0       // D0 inherits (publicly) from B0.
```

```
{
    void f();        // OK, explicitly overrides void B0::f()
    void g();        // Error, void B0::g() is final.
    void g() const;  // OK, void B0::g() const is not final.
};
```

As the simple example above illustrates, decorating a virtual member function — e.g., `B::g()` — with **`final`** precludes overriding only that specific function signature. Note that when redeclaring a **`final`** function (e.g., to define it), the **`final`** specifier is not permitted:

```
void B0::g() final { }  // Error, final not permitted outside class definition
void B0::g() { }        // OK
```

final-on-destructors

`final` on destructors

The use of **`final`** on a virtual destructor precludes inheritance entirely, as any derived class must have either an implicit or explicit destructor, which will attempt to override the **`final`** base class destructor:

```
struct B1
{
    virtual ~B1() final;
};

struct D1a : B1 { };    // Error, implicitly tries to override B1::~B1()

struct D1b : B1
{
    virtual ~D1b() { }  // Error, explicitly tries to override B1::~B1()
};
```

Any attempt to suppress the destructor in the derived class, e.g., using **`= delete`** (see Section 1.1."**??**" on page **??**), will be in vain. If the intent is to suppress derivation entirely, a more expressive way would be to declare the type itself **`final`**; see Section 3.1."**`final`**" on page 6.

-pure-virtual-functions

`final` pure virtual functions

Although declaring a **pure virtual function** **`final`** is valid, doing so makes the type an **abstract class** and also prevents making any derived type a **concrete class**:

```
struct B2  // abstract class
{
    virtual void f() final = 0;  // OK, but probably not useful
};

B2 b;  // Error, B2 is an abstract type

struct D2a : B2  // also an abstract class
{
};
```

```
D2a d;  // Error, D2a is an abstract type.

struct D2b : B2
{
    void f() {};  // Error, void B2::f() is final.
};
```

By declaring the pure **`virtual`** member function, `B2::f()` in the example above, to be **`final`**, we have effectively precluded ever extending the uninstantiable abstract class, `B2`, to *any* instantiable concrete class.

-virtual-and-override

`final` and its interactions with `virtual` and `override`

In contrast, when we apply **`final`** to non**`virtual`** functions, the **`final`** specifier will always force a compilation error:

```
struct B3a
{
    void f() final;  // Error, f is not virtual.
};

struct B3b
{
    void g(); // OK, g is not virtual.
};

struct D3 : B3b
{
    void g() final; // Error, g is not virtual and hides B3b::g.
};
```

The **`final`** keyword combines with the **`virtual`** and **`override`** keywords to produce various effects. For example, functions that are declared virtual in a base class, e.g., all of the functions below in `B4`, are automatically considered virtual by a function having the same signature in a derived class, e.g., the corresponding functions in `D4`, irrespective of whether the **`virtual`** keyword is repeated:

```
struct B4  // Each of the functions in B4 is explicitly declared virtual.
{
    virtual void f();  // explicitly declared virtual
    virtual void g();  //      "        "       "
    virtual void h();  //      "        "       "
};

struct D4 : B4  // Each of the functions in D4 is explicitly declared final.
{
    void f() final;          // OK, because B4::f is declared virtual
    virtual void g() final;  // OK, explicitly declared virtual (no effect)
    void h() final override; // OK, because B4::h is declared virtual
    virtual void i() final;  // OK, explicitly declared virtual (necessary)
```

```
    void j() final;          // Error, nonvirtual function j declared final
};
```

Notice that `D4::g()` is annotated with the keyword **`virtual`** but `D4::f()` is not. Adding **`virtual`** to the declaration of a function in a derived class when there is a matching function declared **`virtual`** in a base class has no effect. In a manner similar to **`override`**, leaving off the explicit **`virtual`** will prevent removing the base class function or altering its signature.

Given the availability of the override specifier (see Section 1.1."**??**" on page **??**), however, a common coding standard has emerged: Use **`virtual`** only to *introduce* a **`virtual`** function in a class hierarchy and then *require* that any functions attempting to *override* a **`virtual`** function in a derived class be decorated with *either* **`override`** or **`final`** and explicitly *not* **`virtual`**:

```
struct B5  // base class consisting of virtual and nonvirtual functions
{
    virtual void f1();  // OK, virtual function
    virtual void f2();  // OK, virtual function
             void g1();  // OK, nonvirtual function
             void g2();  // OK, nonvirtual function
};

struct D5  // "derived class" attempting to override virtual functions f1 and f2
{
    void f1() override;  // Error, f1 marked override but doesn't override.
    void f2() final;     // Error, f2 marked final but isn't virtual
};
```

The astute reader will have already noticed that in the example above we failed to make `D5` inherit publicly from `B5`. Catching this flaw early is one of the awesome benefits of always following this convention. If we try again, this time with `D5b`, we will observe a second benefit of always supplying either **`override`** or **`final`**:

```
struct D5b : B5  // This time we remembered to inherit from B5.
{
    void f1() override;  // OK
    void f2() final;     // OK
    void g1() override;  // Error, g1 marked override but doesn't override
    void g2() final;     // Error, g2 marked final but isn't virtual
};
```

Finally, one could imagine deliberately declaring a base-class function to be both **`virtual`** and **`final`** — e.g., just to prevent a derived class from hiding it:

```
struct B6
{
    virtual void f() final;  // OK
};

static_assert(sizeof(B6) == 1, "");  // Error, B6 holds a vtable pointer.

struct D6a : B6
```

```
{
    void f() const;     // OK, D6a::f doesn't override.
    void f();           // Error, B6::f is final.
};

struct D6b : B6
{
    void f(int i = 0);  // OK, even though it hides B6::f
};
```

Declaring, for the first time, the member function `B6::f` in the example above to be both **`virtual`** and **`final`** has limited practical effect. Attempts to hide `f` in a subclass will be blocked but only when the hiding function has *exactly* the same signature; a function that hides `f` can still be written with different member function qualifiers or even slightly different, possibly optional parameters. Adding **`virtual`** to `f` also makes `B6` a **polymorphic type**, bringing with it the need for a **vtable pointer** in every object and making it non-**trivial**; see Section 2.1."**??**" on page **??**. Even though the compiler will likely be able to **devirtualize** calls to `B6::f`, that it will not is still possible, and there will be the additional overhead of **dynamic dispatch** when invoking calls to `B6::f`; see *Potential Pitfalls — Attempting to defend against the hiding of a non***`virtual`** *function* on page 25.

`final` user-defined types

al-user-defined-types

The use of **`final`** is not limited to individual member functions and can also be applied to an entire user-defined type to explicitly disallow *any* other type from inheriting from it. Preventing a type from being inheritable closes the gap between what is possible with built-in types, such as **`int`** and **`double`**, and what can additionally be done with typical user-defined ones — specifically, inherit from them. See *Use Cases — Suppressing derivation to ensure portability* on page 13, i.e., Hyrum's Law.

Although other uses are plausible, widespread use can run afoul of stable **reuse** in general and **hierarchical reuse** in particular; see *Potential Pitfalls — Systemic lost opportunities for reuse* on page 22. Hence the decision to use **`final`** on an entire class even rarely — let alone routinely — is not to be taken lightly.

Prior to C++11's introduction of the **`final`** specifier, there was no convenient way to ensure that a user-defined type (UDT) was *uninheritable*, although some byzantine idioms to approximate this existed. For example, a virtual base class needs to be initialized in each constructor of all concrete derived types, and that can be levered to prevent useful inheritance. Consider a trio of classes, the first of which, `UninheritableGuard`, has a private constructor and befriends its only intended derived class; the second, `Uninheritable`, derives privately and virtually from `UninheritableGuard`; and the third, `Inheriting`, is a misguided class that tries in vain to inherit from `Uninheritable`:

```
struct UninheritableGuard  // private, virtual base class
{
private:
    UninheritableGuard();         // private constructor
    friend struct Uninheritable;  // constructible only by Uninheritable
};
```

```
struct Uninheritable : private virtual UninheritableGuard
{
    Uninheritable() : UninheritableGuard() { /* ... */ }
};

struct Inheriting : Uninheritable  // Uninheritable is effectively final.
{
    Inheriting()
    : UninheritableGuard()  // Error, UninheritableGuard() is inaccessible.
    { /* ... */ }
};
```

Any attempt to define — either implicitly or explicitly — a constructor for `Inheriting` will fail with the same error due to the inaccessibility of the constructor for `UninheritableGuard`. Using **`virtual`** inheritance typically requires each object of type `Uninheritable` to maintain a virtual table pointer; hence, this solution does not come without overhead. Note also that this workaround prior to **`final`** does not prevent the derivation itself, but merely the instantiation of any ill-fated derived classes.

In the special case where all of the data members of the type are trivial, i.e., have no user-provided **special member functions** (see Section 2.1."**??**" on page **??**), we could have instead created a type, e.g., `Uninheritable2`, that is implemented as a **`union`** consisting of just a single **`struct`**:

```
union Uninheritable2  // C++ does not yet permit inheritance from union types.
{
    struct  // anonymous class type
    {
        int    i;
        double d;
    } s;

    Uninheritable2()
    {
       s.i = 0;
       s.d = 0;
    }
};

struct S : Uninheritable2 { };  // Error, unions cannot be base classes.
```

With the introduction of the **`final`** specifier, no such contortions are needed. When added to the *definition* of either a **`class`** or **`struct`**, the **`final`** specifier adeptly prevents prospective clients from deriving from that type:

```
struct S1 { };        // nonfinal user-defined type
struct S2 final { };  // final user-defined type

struct D1 : S1 { };   // OK, S1 is not declared final.
struct D2 : S2 { };   // Error, S2 is declared final.
```

The **`final`** specifier may be applied to a type's declaration as long as that declaration is a definition:

```
class C1;            // OK,    C1 is (as of now) an incomplete type.
class C1 final;      // Error, attempt to declare variable of incomplete type
class C1 final { };  // OK, C1 is now a complete type.
class C1;            // OK, C1 is known to be a final type.
```

Once a type is complete, an attempt to redeclare the type using the final specifier will be a valid declaration of an object of that type named **`final`**. This also means that once a type has been defined as *non***`final`**, it cannot subsequently be redeclared as **`final`**:

```
class C2 { };        // OK, C2 is a nonfinal complete type.
class C2 final;      // Bug??, C2 object named final.
```

As the final line above illustrates, **`final`**, being a contextual keyword, is a valid name for a object of non**`final`** class `C2`; see *Potential Pitfalls — Contextual keywords are contextual* on page 22.

The **final** specifier, in addition to decorating a **class** or **struct**, can also be applied to a **union**. Since derivation from a **union** isn't permitted anyway, declaring a union to be **final** has no effect.[1]

use-cases-final

Use Cases

n-to-ensure-portability

Suppressing derivation to ensure portability

A rare but compelling use of **final** for a user-defined type as a whole occurs when that type is used to simulate a feature that is or might some day be implemented as a **fundamental type** on this or some other platform. By specifying the user-defined type to be **final**, we avoid locking ourselves into forever being represented as a user-defined type.

As a concrete and very real-world example, consider an early implementation of a family of floating-point decimal types: `Decimal32_t`, `Decimal64_t`, and `Decimal128_t`. Initially these types (as type aliases) are implemented in terms of user-defined types:

```
class DecimalFloatingPoint32  { /* ... */ };
class DecimalFloatingPoint64  { /* ... */ };
class DecimalFloatingPoint128 { /* ... */ };

typedef DecimalFloatingPoint32  Decimal32_t;
typedef DecimalFloatingPoint64  Decimal64_t;
typedef DecimalFloatingPoint128 Decimal128_t;
```

Once released in this form, there is nothing to stop users from inheriting from these **typedef**s to derive their own custom types, and, in fact, **Hyrum's Law** all but guarantees that such inheritance will happen:

```
class MyDecimal32     : public Decimal32_t     { /* ... */ };
class YourDecimal64   : private Decimal64_t    { /* ... */ };
class TheirDecimal128 : protected Decimal128_t { /* ... */ };
```

Hardware support will someday arrive — and on some platforms it already has — that will likely allow these **typedef**s to become aliases to fundamental types rather than aliases to user-defined types:

```
typedef __decimal32_t  Decimal32_t;
typedef __decimal64_t  Decimal64_t;
typedef __decimal128_t Decimal128_t;
```

When that day comes and we try to leverage the possibly huge performance benefits of using a natively supported type, a small number of bad actors amongst clients of this library will prevent the majority of users from benefitting from the new platform. The end result is that person years of effort might be required to unravel the code that now depends on inheritance. This effort is a price that could have been avoided had we just declared our *not necessarily* user-defined types **final** from the start:

[1] In C++14, the Standard Library header `<type_traits>` defines the trait `std::is_final`, which can be used to determine whether a given user-defined, class type is specified as **final**. This trait can be used to distinguish between a non**final** and **final union**. The argument to this trait, however, must be a complete type.

```
class DecimalFloatingPoint32 final  { /* ... */ };
class DecimalFloatingPoint64 final  { /* ... */ };
class DecimalFloatingPoint128 final { /* ... */ };
```

Using **`final`** here would have avoided advertising more than we were prepared to support in perpetuity.

Improving performance of concrete classes

e-of-concrete-classes

Object-Oriented programming (OOP) comprises two important aspects of software design: (1) inheritance and (2) **dynamic binding**. For a language to claim that it supports OOP, it must support both.[2] No overhead is associated with inheritance *per se*, but the same cannot be said in general for dynamic binding (a.k.a., **virtual dispatch**).

Without commenting on the wisdom of an implementation consisting of a mixture of *interface*, *implementation*, and *structural* inheritance, consider the classic `Shape`, `Circle`, and `Rectangle` example illustrating **object orientation**:

```
class Shape  // abstract base class
{
    int d_x;  // x-coordinate of origin
    int d_y;  // y-coordinate of origin

public:
    Shape(int x, int y) : d_x(x), d_y(y) { }    // value constructor

    void move(int dx, int dy) { d_x += dx; d_y += dy; }
                                                // concrete manipulator

    int xOrigin() const { return d_x; }         // concrete accessor
    int yOrigin() const { return d_y; }         // concrete accessor

    virtual double area() const = 0;            // abstract accessor
};

class Circle : public Shape  // concrete derived class
{
    int d_radius;  // radius of this circle

public:
    Circle(int x, int y, int radius) : Shape(x, y), d_radius(radius) { }
                                                // value constructor

    double area() const { return 3.14 * d_radius; }
                                                // concrete accessor
};

class Rectangle: public Shape   // concrete derived class
{
```

[2] **?**, section 0.x, pp. y–z

```
    int d_length;  // length of this rectangle
    int d_width;   // width of this rectangle

public:
    Rectangle(int x, int y, int length, int width) :
                      Shape(x, y), d_length(length), d_width(width) { }
                                                  // value constructor

    double area() const final { return d_length * d_width; }
                                                  // concrete accessor
};
```

Note that `Shape` acts as both an interface and as a base class offering a proper value constructor. Inheriting classes have to override `Shape::area` to provide a *concrete accessor* for their respective area values. The two classes inheriting from `Shape`, namely `Circle` and `Rectangle`, differ substantially in just one aspect: `Rectangle::area()` is annotated with **`final`** whereas `Circle::area()` is not.

Now imagine a client accessing the *concrete implementations* of `Shape::area` through a base-class reference:

```
void client1(const Shape& shape)
{
    int x = shape.xOrigin();     // inlines (e.g., Clang -O2, GCC -O1)
    double area = shape.area();  // inline requires whole program optimization
}
```

The call to `Shape::xOrigin` can be inlined at a fairly low level of optimization whereas the call to `Shape::area` cannot because it is subject to *virtual dispatch*. The same ability to inline the call to get the `xOrigin` applies for objects of either derived type — `Circle` as well as `Rectangle` — irrespective of whether the type's implementation of `Shape::area` has been annotated with **`final`**.

If the compiler can somehow locally infer the runtime type of the derived object, the function calls can be inlined at a fairly low optimization level too, again irrespective of any annotation with **`final`**:

```
void client2()
{
    Circle c(3, 2, 1);
    Rectangle r(4, 3, 2, 1);
    const Shape& s1 = c;
    const Shape& s2 = r;

    double cArea = c.area();  // inlines (e.g., Clang -O2, GCC -O1)
    double rArea = r.area();  // inlines (e.g., Clang -O2, GCC -O1)

    double s1Area = s1.area();  // inlines (e.g., Clang -O2, GCC -O1)
    double s2Area = s2.area();  // inlines (e.g., Clang -O2, GCC -O1)
}
```

The real difference comes to light when accepting references of these types in a separate function. As only `Rectangle` prohibits further overriding of the `area` function, it is the only

one of the three types that can know at compile time the runtime type of its object and therefore bypass virtual dispatch:

```
void client3(const Shape& s, const Circle& c, const Rectangle& r)
{
    double sArea = s.area();  // must undergo virtual dispatch
    double cArea = c.area();  // must undergo virtual dispatch
    double rArea = r.area();  // inlines (Clang -O2, GCC -O1)
}
```

Note that a compilation system that is aware of the entirety of a program might be able to turn that into knowledge that `Circle` is effectively final — e.g., by observing that no other classes derive from it — and thus similarly bypass virtual dispatch in `client3`. Such optimization, however, is expensive, doesn't scale, and would be thwarted by common practices, such as the possibility of loading shared objects at run time, which might contain classes derived from `Circle` that are then passed to `client3`.

mance-lost-to-mocking

Restoring performance lost to mocking

In most cases, a component cannot be fully isolated from its dependencies in order to be able to test it in complete isolation. An object-oriented approach widely used to artificially circumvent such limitations is often colloquially referred to as **mocking**.

As an example, consider a custom file-handling class, `File`, that depends on a custom local file system, `LocalFS`:

```
class LocalFS  // lower-level file system
{
public:
    FileHandle* open(const char* path, int* errorStatus = 0);
};

class File  // higher-level file representation
{
    LocalFS* d_lfs;  // pointer to concrete filesystem object

public:
    File(LocalFS* lfs) : d_lfs(lfs) { }

    int open(const char *path);
    // ...
};
```

In the testing-resistant design suggested above, class `File` depends directly on a local file system; hence, there is no convenient way for us to test what happens when rarely seen (e.g., error) events occur within the file system.

To be able to test our `File` class thoroughly, we will need some way to control what the `File` class does when it receives input back from what it thinks is the local file system. One approach would be to extract a pure abstract (a.k.a. **protocol**) class, e.g., `AbstractFileSystem`, from `LocalFS`. We can then make our `File` class depend on the protocol instead of `LocalFS` and make `LocalFS` implement it:

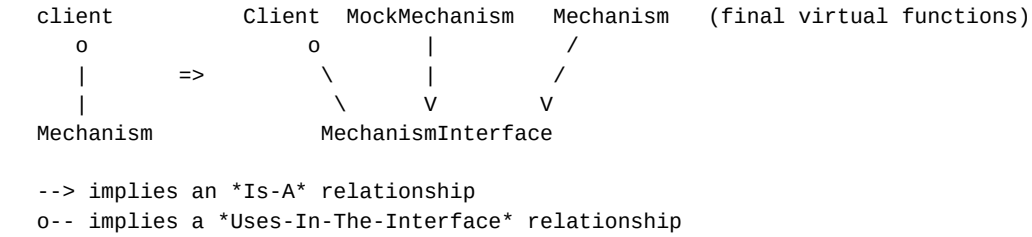
```
class AbstractFileSystem  // lower-level pure abstract interface
{
public:
    virtual FileHandle *open(const char *path, int *errorStatus = 0) = 0;
};


class File  // uses lower-level abstract file system
{
    AbstractFileSystem *d_afs;  // pointer to abstract filesystem object

public:
    File(AbstractFileSystem *afs) : d_afs(afs) { }

    int open(const char *path);
    // ...
};

class LocalFS : public AbstractFileSystem  // implements abstract interface
{
public:
    FileHandle *open(const char *path, int *errorStatus = 0);
};
```

With this artificially complicated design, we are now able to create an independent concrete, "mock" implementation derived from `AbstractFileSystem`, which we can then use to orchestrate arbitrary behavior, thereby enabling us to test `File` both (1) in isolation and (2) under abnormal circumstances:

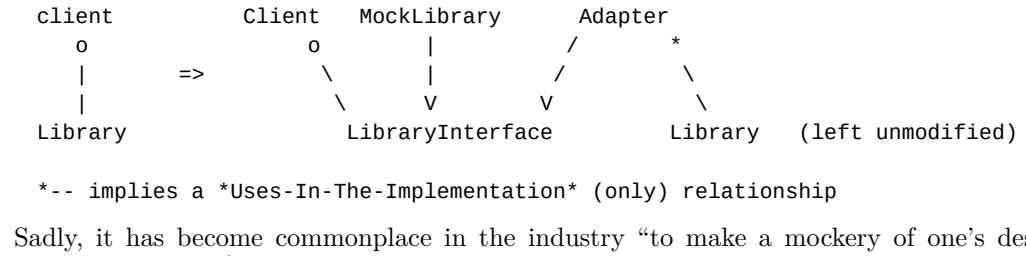
```
class MockedFS : public AbstractFileSystem  // test-engineer-controllable class
{
public:
    FileHandle *open(const char *path, int *errorStatus = 0) { /* mocked */ }
};

void test()  // test driver that orchestrates mock implementation to test File
{
    MockedFS mfs;  // mock AbstractFileSystem used to thoroughly test File
    File f(&mfs);  // f installed with mock instead of actual LocalFS

    int rc = f.open("dummyPath");  // mock used to supply handle
    // ...
}
```

Although this technique does enable independent testing, it comes with the performance cost of dynamic dispatch on all calls to the underlying `AbstractFileSystem` object. In certain situations where the concrete implementation doesn't have to be exchangeable for the mock, performance can be recovered by declaring some or all of the concrete, now-virtual functions of `localFS` **`final`**. This way, when we pass the concrete implementation — e.g., `LocalFS` — to a function explicitly, at least the functions that are declared **`final`** and were

previously inline can again be inline:



Alternatively, we might create a new component that adapts an existing library to a new interface suitable for mocking without having to alter it in any way:



Sadly, it has become commonplace in the industry "to make a mockery of one's design through mocking"[3] by artificially declaring all nonconstructor member functions of a class virtual (without extracting a protocol) just to be able to provide a "mock" implementation derived from the original concrete class:

[3]Jonathan Wakely has expressed to John Lakos that indiscriminate and excessive use of mocking "is to make a mockery of one's design."

```
client          Client  MockMechanism  (making these final is pointless)
   o                 o       |
   |       =>         \      |  (BAD IDEA)
   |                   \     V
Mechanism             Mechanism   (now with all public virtual functions)
```

As the previous diagrams above suggest, much of the virtual function overhead associated with mocking can be reduced significantly by *not* deriving the mock from the original class. Instead, we can either (1) extract a protocol for all of the public nonconstructor member functions of that class and mark those specific functions **`final`** or (2) adapt the original concrete implementation unchanged to the new protocol for use by `File` and any other facilities that choose to opt in for the purposes of thoroughly testing their code. With either of these approaches, all clients that continue to use the concrete, possibly now-derived class by its original name do not need to pay for *any* virtual function overhead to support the ability to mock that class's interface where needed.

Improving performance in a protocol hierarchy

in-a-protocol-hierarchy

A **protocol hierarchy**[4,5] is a generalization of the **composite pattern**[6] in which a pure abstract interface (a.k.a. protocol) sits at the root (`leve 1`) of a public inheritance hierarchy (*Is-A* relationship) of abstract interfaces:

```
level 3          ( yet wider interface )
                          |         f(x), g(y), h(z)
                          |
                          |
                          | *Is-A*
                          V
level 2:           ( wider interface )
                          |         f(x), g(y) [[ y >= 0 ]]
                          |
                          |
                          | *Is-A*
                          V
level 1:            ( interface )
                                    f(x)
```

At each successive level, the *pure* interface is a proper superset of the one at the previous level from which it derives. None of the classes in the hierarchy contributes any implementation; each member function — except for the destructor — is implemented as pure virtual. In particular, observe that `g(y)` is not accessible at `level 1`, can be used only with non-negative values at `level 2` (i.e., `g(y)` has a **narrow contract**), and is usable with all syntactically legal values at `level 3` (i.e., `g(y)` has a **wide contract**). Note that this same sort of interface widening can apply in the absence of virtual functions through judicious

[4] **?**, Appendix A, pp. 737–768

[5] **?**, section 2.7 [AUs: The citation is lakos2y, do you mean Vol II of Large-Scale C++? Response: Yes, volume II]

[6] **?**, Chapter 2, section "Composite," pp. 163–174

use of hiding nonvirtual functions; see *Potential Pitfalls — Systemic lost opportunities for reuse* on page 22.

Concrete leaf nodes can then be derived from the protocol hierarchy to implement the desired level of service as efficiently as practical. In cases where multiple concrete nodes need to share the same implementation of one or more functions, we can derive an intermediate node from the appropriate *protocol* that doesn't widen the interface at all but does implement one or more of the pure abstract functions; such an *impure* abstract node is known as a **partial implementation**. When the implementation of one of these functions is trivial, declaring that **`virtual`** function to be **`inline`** in addition might make sense. Since there will, by design, be no need to further override that function, we can declare it to be **`final`** as well.

For performance-critical clients that would otherwise consume the concrete object via the pure abstract interface from which this partial implementation derives, we might decide to instead take the partial implementation itself as the reference type. Because one or more functions are both **`inline`** and **`final`**, the client can dispense with runtime dispatch and inline the virtual functions directly as discussed in Restoring performance lost to mocking.

As a real-world example, consider a simplified protocol hierarchy for memory allocation:

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

struct Allocator
{
    virtual void *allocate(std::size_t numBytes) = 0;
        // Allocate a block of memory of at least the specified numBytes.

    virtual void deallocate(void *address) = 0;
        // Deallocate the block at the specified address.
};

struct ManagedAllocator : Allocator
{
    void *allocate(std::size_t numBytes) = 0;
        // Allocate a block of memory of at least the specified numBytes.

    void deallocate(void *address) = 0;
        // Deallocate the block at the specified address.

    virtual void release() = 0;
        // Reclaim all memory currently allocated from this allocator.
};
```

A **monotonic allocator** is a kind of **managed allocator** that allocates memory sequentially in a buffer subject to alignment requirements. In this class of allocators, the `deallocate` method is always a no-op; memory is reclaimed only when the managed allocator is destroyed or its `release` method is invoked:

```
struct MonotonicAllocatorPartialImp : ManagedAllocator
{
    void *allocate(std::size_t numBytes) = 0;
```

```
        // Allocate a block of memory of at least the specified numBytes.

    inline void deallocate(void *address) final { /* empty */ }
        // Deallocate the block at the specified address.

    void release() = 0;
        // Reclaim all memory currently allocated from this allocator.
};
```

Notice that we have specified the empty **inline** `deallocate` member function of `MonotonicAllocatorPartialImp` to be **final**. A concrete monotonic allocator — e.g., a `BufferedSequentialAllocator` — can then derive from this partial implementation:

```
struct BufferedSequentialAllocator : MonotonicAllocatorPartialImp
{
    BufferedSequentialAllocator();
        // Create a default version of a buffered-sequential allocator.

    void *allocate(std::size_t numBytes);
        // Allocate a block of memory of at least the specified numBytes.

    void release();
        // Reclaim all memory currently allocated from this allocator.
};
```

Now consider two allocator-aware types, `TypeA` and `TypeB`, each of which is always constructed with some flavor of monotonic managed allocator:

```
struct TypeA
{
    TypeA(ManagedAllocator *a);
    // ...
};

struct TypeB
{
    TypeB(MonotonicAllocatorPartialImp *a);
    // ...
};
```

We now construct each of the types using the same concrete allocator object:

```
void client()
{
    BufferedSequentialAllocator a;  // Concrete monotonic allocator object

    TypeA ta(&a);  // deallocate is an empty virtual function call.
    TypeB tb(&a);  // deallocate is an empty inline function call.
}
```

When `TypeA` invokes `deallocate`, it goes through the non**final**, **virtual** function interface of `ManagedAllocator` and is subject to the runtime overhead of dynamic dispatch. Note that even if the virtual `deallocate` function were inline, unless it is declared **final** or the

runtime type is somehow known at compile time, there is no sure way for the compiler to know that the function isn't overridden by a derived type.

In the case of `TypeB`, however, the function is both *declared* **`final`** and *defined* **`inline`**; hence the virtual dispatch can be reliably sidestepped, the empty function can be inlined, and a true no-op is achieved with absolutely no runtime overhead.

Potential Pitfalls

ential-pitfalls-final

Contextual keywords are contextual

ywords-are-contextual

The Standards Committee has, historically, taken very different approaches to adding new keywords to the language. C++11 added ten new keywords to the language — **`alignas`**, **`alignof`**, **`char16_t`**, **`char32_t`**, **`constexpr`**, **`decltype`**, **`noexcept`**, **`nullptr`**, **`static_assert`**, and **`thread_local`**[7] — and thus made ten potential tokens no longer usable as identifiers. When considering new keywords, a great deal of effort is generally extended to determine the impact of this change in status on existing codebases. Two identifiers, **`override`** and **`final`**, were not made keywords and were instead given special meaning when used in contexts where previously identifiers were not syntactically allowed. This approach avoided possible code breakage for any existing codebases using these words as identifiers, at the cost of occasional confusion.

When used after a function declaration, **`override`** and **`final`** do not add any significant parsing ambiguity to the language; arbitrary identifiers were not syntactically valid in that position anyway, so confusion is minimal. When used on a **`class`** declaration, however, **`final`**'s meaning is not determined until tokens after it are parsed to distinguish between a variable declaration and a class definition:

```
struct S1 final;      // Error, variable named final of incomplete type
struct S2 final { };  // OK, final class definition
struct S2 final;      // OK, variable named final of complete type S2
```

Notice that the variable declarations in the example above both look like they might be an attempt to forward-declare a **`struct`** that is final but are instead a totally different language construct.

Systemic lost opportunities for reuse

portunities-for-reuse

Both **`final`** and **`override`** are similar in their complexity yet very different in the potential adverse implications that widespread use can impose. Such ubiquitous use will depend heavily on the scale and nature of the development process employed. In some development environments, such as a small organization overseeing a closed-source codebase where clients are able to request timely code changes, encouraging liberal use of **`final`** might not be problematic. Instead of promising everything up front, even when much of what is offered is not immediately useful, the default development approach might reasonably be to provide only what is immediately necessary and then quickly expose more if and as needed.

[7] C++14 and C++17 added exactly zero new keywords in total. C++20 added **`char8_t`**, **`co_await`**, **`co_return`**, **`co_yield`**, **`concept`**, **`consteval`**, **`constinit`**, and **`requires`**, notably mixing some potentially already used identifiers (**`concept`** and **`requires`**) with a collection of more obscure new words that had little chance of significantly conflicting with existing codebases.

For some organizations, however, request-based code changes may not be a viable option and can result in unacceptable delays in responding to client needs. The extensive use of **final** inherently prevents clients from reusing library components in unanticipated ways and can therefore lead to redundant and highly undesirable forks, often through thoughtless copying and pasting the original code. Gratuitously forbidding clients from doing what they feel they need to do and would otherwise be able to do might be perceived as unnecessary nannyism.[8]

Consider, for example, the Standard Template Library (STL) and, in particular, `std::vector`. One might argue that `std:vector` was designed to facilitate generic programming, has no virtual functions, and therefore should be specified as **final** to ensure its "proper" use and no other. Suppose, on the other hand, that teachers wanting to teach their students the value of **defensive programming** were to create an exercise to implement a `CheckedVector<T>`, derived publicly from `std::vector<T>`.[9] By inheriting constructors (see Section 2.1."**??**" on page **??**), it is very simple to implement this derived class with an alternate implementation for just **operator[]**:

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

template <typename T>
class CheckedVector : public std::vector<T>
{
public:
    using std::vector<T>::vector;  // Inherit all ctors of std::vector<T>.

    using reference       = typename std::vector<T>::reference;
    using const_reference = typename std::vector<T>::const_reference;
    using size_type       = typename std::vector<T>::size_type;

    reference operator[](size_type pos)             // hide base class function
    {
        assert(pos >= 0 && pos < this->size());     // Check bounds.
        return this->std::vector<T>::operator[](pos);
    }

    const_reference operator[](size_type pos) const  // hide base class function
    {
        assert(pos >= 0 && pos < this->size());     // Check bounds.
        return this->std::vector<T>::operator[](pos);
    }
```

[8] "Unnecessary nannyism" is a phrase Bjarne Stroustrup used to characterize his initial decision to restrict operators `[]`, `()`, and `->` to be members; see **?**, Chapter x, section 3.6.2, "Overloading: Members and Friends," pp. 81–83, particularly p. 83.

[9] Bjarne Stroustrup himself has employed as a class exercise in which the only two functions in the derived type (that he typically calls "Vector") hide the **operator[]** overloads of `std::vector`. These implementations perform additional checking so that if they are ever called out of their valid range, instead of resulting in **undefined behavior**, they do something sensible, e.g., throw an exception or, even better, print an error message and then call `abort` to terminate the program. By not employing `assert`, as we do in our example, Stroustrup avoids using conditional compilation, which is not essential to the didactic purpose of this exercise.

```
};
```

In the implementation above, we have chosen to use the standard `C assert` macro instead of hard-coding the check and then, if needed, explicitly printing a message and calling `abort`. Note that this check will occur in only certain build modes, i.e., when `NDEBUG` is not defined for the current translation unit. Note also that the checking for `pos >= 0` can be omitted here as `std::vector::size` returns an integral value of type `std::size_t`, which is guaranteed to be of some implementation-defined **`unsigned`** integral type on all conforming platforms. Finally the use of **`this->`** is purely stylistic to show that we are invoking a member function on this object and can be omitted in every case above where it is used with no change in defined behavior.

For local use with the goal of exploring and learning, allowing such **structural inheritance** — involving no virtual functions — can be an expeditious way of uncovering client misuse. This disciplined use of structural inheritance adds no data members; it merely widens some of the narrow interfaces in the base class, thereby benignly enhancing the defined behavior already provided yet leaving unchanged the behavior of all other functions of `vector`. We can now deploy our derived `CheckedVector` by replacing instances of `std::vector` with those of our derived `CheckedVector`, including those in interfaces where potential misuse might occur; simple interaction with the parts of the system not so modified will continue to operate as before. Functions that take an `std::vector` passed by value are not recommended but will continue to work as before via **slicing**, as will those passed by pointer or reference via implicit standard conversion to base type:

```
void myApi(const CheckedVector<int> &data);       // checked local API
void otherApi1(const std::vector<int> &data);     // by-reference standard API
void otherApi2(std::vector<int> data);            // by-value standard API
template <typename T>
void genericApi(const T&data);                    // generic API

void myFunction()
{
    CheckedVector<int> myData;
    // ...                 ( populate myData )

    otherApi1(myData);   // normal usage, unchecked operation
    otherApi2(myData);   // normal usage, unchecked operation, slices on copy
    myApi(myData);       // checked operations within implementation

    std::vector<int>&  uncheckedData = myData;
    genericApi(myData);        // checked call to templated API
    genericApi(uncheckedData); // unchecked call to same API
}
```

Were `std::vector` declared **`final`**, this form of investigation would be entirely prevented, harming those who wished to learn or needed to diagnose defects in their systems using `std::vector`. Such compulsive use of **`final`**, even in local libraries, often leads to what is arguably the worst possible result: The client makes a local copy of the library class. Such gratuitously forced duplication of source code systemically exacerbates the already high cost

of software maintenance and denies the use of any future enhancements or bug fixes made to that library class.

Finally, contrary to popular belief, the strict notion of substitutability afforded by structural inheritance is far more in keeping with those characterized by Barbara Liskov in her pioneering work on subtyping [10] than the *variation in behavior*[11] afforded by virtual functions. Thinking that a class has no business being derived from just because it doesn't sport a virtual destructor is misguided. Much of what is done in metaprogramming relies heavily on structural inheritance; for example, the class template `std::integral_constant` serves as the base class for most C++ type traits. Structurally inheriting a `const_iterator` from an `iterator` achieves implicit convertibility without consuming a user-defined conversion and thereby avoids needless asymmetry; see *Annoyances — Making empty types* **`final`** *precludes the empty base-class optimization* on page 26.

The decision to make systemic use of **`final`** the default — absent any specific engineering reason — is dubious as it directly precludes reuse in general and hierarchical reuse in particular. In any event, such a policy, to achieve its intended purpose, fairly belongs with an organization as a whole rather than with each individual developer within that organization. If, however, some specific reason precludes overriding a virtual function or inheritance in general, then use of **`final`** is indicated if only to actively document our intent. Lacking such forceful documentation initially, Hyrum's law will ultimately subvert our ability to make the choice to prevent overriding and/or inheritance at a later date.

f-a-nonvirtual-function

Attempting to defend against the hiding of a non`virtual` function

Deliberately hiding virtual functions is not typically recommended[12]; when it is done, it is often by accident. Declaring an otherwise nonvirtual function to be both virtual and final does in fact prevent a derived class author from overriding that specific function; a derived class's version of the function will never be invoked through dynamic dispatch. `final` does *not*, however, prevent a function of the same name having any distinct signature from inadvertently hiding it.

As an illustrative example, suppose we were to have a simple output-device type, `Printer`, that contains a set of overloaded non`virtual` print functions to display various types of information on a common device:

```
struct Printer
{
    void print(int number);
    void print(bool boolean);
};
```

Now suppose an inexperienced programmer is about to extend that type to support another parameter type and accidentally hides the base-class functions:

```
struct ExtendedPrinter : public Printer
```

[10] **?**

[11] Tom Cargill observed that data members are for variation in value, whereas virtual functions are for variation in behavior; see **?**, section "Value versus Behavior," pp. 16–19, specifically p. 17 (see also p. 83 and p. 182)

[12] **?**, Chapter 6, item 37, "Avoid Hiding Inherited Names," pp. 131–132

```
{
    void print(long c);
};
```

Calling `ExtendedPrinter::print` with a parameter having a type that is implicitly convertible to **`long`** — such as both of the existing supported types of **`int`** and **`bool`** — will still compile but fail to lead to the expected results.

To defend against such misuse, we might incorrectly try to defend against such misuse by declaring all functions of the base class both **`virtual`** and **`final`** as well:

```
struct Printer0
{
    virtual void print(int number) final;
    virtual void print(bool boolean) final;
};
```

Such machinations do, in fact, make it a compile-time error should we accidentally supply a function matching *exactly* that signature in a derived class, but it does not prevent a function of the same name having any other signature from doing so:

```
struct ExtendedPrinter0 : public Printer0
{
    void print(int number);    // Error, Printer0::print(int) is final
    void print(char c) const;  // OK, still hides base-class functions
};
```

Making a function virtual in a class where previously there was none also forces the compiler to maintain a pointer to a **`static`** virtual-function table in each object. Consequently, *any* use of **`virtual`** and **`final`** to decorate the same function with a class is contraindicated.

Annoyances

annoyances-final

Making empty types `final` precludes the empty base-class optimization

se-class-optimization

Whenever a user-defined type derives from another that has no data members, that base type does not typically consume any additional memory in the derived type. This optimization is called **empty base-class optimization (EBO)** and is often exploited when applying policy-based design. Consider this slightly modified version of a classic example[13] in which an `ObjectCreator` relies on a specific `CreationPolicy` type for implementing the acquisition of memory and construction of objects:

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

template<typename TYPE, template<typename> class CreationPolicy>
class ObjectCreator : CreationPolicy<TYPE>
{
    std::size_t objectCount = 0;  // Keep track of allocated objects.

public:
    TYPE *create()
    {
```

[13] **?**, Chapter 1, section 1.5, "Policies and Policy Classes," pp. 8–11

```
        ++objectCount;
        return CreationPolicy<TYPE>().create();  // Delegate to CreationPolicy.
    }
};
```

Each of the associated policies are implemented as empty classes, i.e., classes having no data members:

```
template<typename TYPE>
class OpNewCreator // sizeof(OpNewCreator) by itself is 1 byte.
{
public:
    TYPE *create()
    {
        // Allocate memory using placement new and return address.
    }
};

template<typename TYPE>
class MallocCreator // sizeof(MallocCreator) by itself is 1 byte.
{
public:
    TYPE *create()
    {
        // Allocate memory using malloc and return address.
    }
};

static_assert(sizeof(ObjectCreator<int,OpNewCreator>) ==sizeof(std::size_t),"");
static_assert(sizeof(ObjectCreator<int,MallocCreator>)==sizeof(std::size_t),"");
```

Since `OpNewCreator` and `MallocCreator` do not have any data members, inheriting from either of them does not increase the size of `ObjectCreator`. If someone later decides to declare them as **`final`**, inheriting becomes impossible, even if just privately as an optimization:

```
class OpNewCreator final { /* ... */ };   // subsequently declared final
class MallocCreator final { /* ... */ };  //      "           "       "

template<typename TYPE, template<typename> class CreationPolicy>
class ObjectCreator : CreationPolicy<TYPE>  // Error, derivation is disallowed.
{ /* ... */ };
```

By declaring the empty bases class **`final`**, a valid use case is needlessly prevented. Using containment instead of **private inheritance** consumes at least one extra byte in the footprint of `ObjectCreator`,[14] which will inevitably also come at the cost of additional padding imposed by alignment requirements:

[14] C++20 adds a new attribute, `[[no_unique_address]]`, that allows the compiler to avoid consuming addition storage for data objects of empty classes:

```
struct A final { /* no data members */ };
struct S {
```

```
template<typename TYPE, template<typename> class CreationPolicy>
class LargeObjectCreator
{
    CreationPolicy<TYPE> policy;  // now consumes an extra byte &
    std::size_t objectCount = 0;  // with padding 8 extra bytes

public:
    TYPE *create()
    {
        ++objectCount;
        return policy.allocate();
    }
};

static_assert(
    sizeof(LargeObjectCreator<int, OpNewCreator>) > sizeof(std::size_t), "");

static_assert(
    sizeof(LargeObjectCreator<int, MallocCreator>) > sizeof(std::size_t), "");
```

Alternatively, the author of `OpNewCreator` and `MallocCreator` might reconsider and remove **`final`**.

see-also

See Also

- "**??**" (§1.1, p. **??**) ♦ is a related contextual keyword that verifies the existence of matching virtual functions in base classes instead of preventing matching virtual functions in derived classes.

further-reading

Further Reading

- Barbra Liskov discusses in her seminal 1987 keynote paper a remarkable number of issues relevant to the ongoing design and development of modern C++; see **?**.
- Barbara Liskov and Jeanette Wing followed up with a precise notion of subtyping in which any property provable about objects of a supertype would necessarily hold for objects of proper subtypes; see **?**. This notion of proper subtyping (which is manifestly distinct from C++-style inheritance) would later come to be know as the Liskov Substitution Principal (LSP)[15]:

  > Let (x) be a property provable about objects x of type T. Then (y) should be true for objects y of type S where S is a subtype of T.

```
    [[no_unique_address]] A a;  static_assert(sizeof(a) == 1, "");
    int x;                      static_assert(sizeof(x) == 4, "");
};                              static_assert(sizeof(S) == 4, "");
```

[15] **?**, section 1, "Introduction," p. 1812

sec-unsafe-cpp14

Function (`auto`) Return-Type Deduction

Return-Type Deduction

The return type of a function can be deduced from the **`return`** statements in its definition if a **placeholder** (e.g., **`auto`**) is used in place of the return type in the function's prototype.

Description

description

C++11 provides a limited capability for determining the return type of a function from the function's arguments using the **`decltype`** operator (see Section 1.1."??" on page ??), typically in a trailing return type (see Section 1.1."??" on page ??):

```
template <typename CONTAINER, typename KEY>
auto search(const CONTAINER& c, const KEY& k) -> decltype(c.find(k))
{
    return c.find(k);
}
```

Note that the trailing return type specification effectively repeats the entire implementation of the function template. As of C++14, the return type of a function can instead be deduced directly from the **`return`** statements inside the function definition:

```
template <typename CONTAINER, typename KEY>
auto search(const CONTAINER& c, const KEY& k)
{
    return c.find(k);  // Return type is deduced here.
}
```

The return type of the `search` function template defined above is determined by the type of the expression `c.find(k)`. This feature provides a useful shorthand for return types that are difficult to name or would add unnecessary clutter. The deduced return types feature in C++14 is an extension of a similar feature already available for **lambda expressions** in C++11:

```
auto iadd1 = [](int i, int j) { return i + j; };  // valid since C++11
auto iadd2(int i, int j)      { return i + j; }   // valid since C++14
```

Note that this use of **`auto`** is distinct from using **`auto`** with a trailing return type:

```
auto a()           { return 1; }  // deduced return type int
auto b() -> double { return 1; }  // specified return type double
```

Specification

specification

When a function's return type is specified using **`auto`** or **`decltype(auto)`** and there is no trailing return type, the return type of the function is deduced from the **`return`** statement(s) in the function body following the same rules as for deducing a variable declaration from its initializer expression (see Section 1.1."??" on page ??):

```
class C1 { /* ... */ };

C1   c;
```

```
C1   f1();
C1&  f2();
C1&& f3();

auto           v1 = c;                 // deduced type C1
auto           g1() { return c; }      //    "    return type C1

decltype(auto) v2 = c;                 //    "    type C1
decltype(auto) g2() { return c; }      //    "    return type C1

auto           v3 = (c);               //    "    type C1
auto           g3() { return (c); }    //    "    return type C1

decltype(auto) v4 = (c);               //    "    type C1&
decltype(auto) g4() { return (c); }    //    "    return type C1&

auto           v5 = f1();              //    "    type C1
auto           g5() { return f1(); }   //    "    return type C1

decltype(auto) v6 = f1();              //    "    type C1
decltype(auto) g6() { return f1(); }   //    "    return type C1

auto           v7 = f2();              //    "    type C1
auto           g7() { return f2(); }   //    "    return type C1

decltype(auto) v8 = f2();              //    "    type C1&
decltype(auto) g8() { return f2(); }   //    "    return type C1&

auto           v9 = f3();              //    "    type C1
auto           g9() { return f3(); }   //    "    return type C1

decltype(auto) v10 = f3();             //    "    type C1&&
decltype(auto) g10() { return f3(); }  //    "    return type C1&&
```

As with variable declarations, **`auto`** (but not **`decltype(auto)`**) can be cv-qualified and decorated to form a reference, pointer, pointer-to-function, reference-to-function, or pointer-to-member function:

```
const auto  g11() { return c; }     // return type const C1
auto&       g12() { return c; }     //    "     "  C1&
const auto& g13() { return c; }     //    "     "  const C1&
auto&&      g14() { return c; }     //    "     "  C1&
auto&&      g15() { return f3(); }  //    "     "  C1&&
auto*       g16() { return &c; }    //    "     "  C1*
auto   (*g17())() { return &g12; }  //    "     "  C1& (*)()
auto&       g18() { return f3(); }  // Error, can't bind C1&& to lvalue ref
```

Note that **`auto&&`** is a **forwarding reference**, which means that the **value category** of the return expression will determine whether an *lvalue* reference (in the case of `g14`) or *rvalue* reference (in the case of `g15`) will be deduced. The function declaration is ill formed

if the specifiers added to **`auto`** would cause the return-type deduction to fail, as in the case of `g18`.

The same restrictions apply to an **`auto`** and **`decltype(auto)`** return-type deduction as to a similar variable-type deduction[1]:

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

std::vector<int> v;

std::vector<auto>&   g19() { return v; }  // Error, auto as template argument
decltype(auto)&      g20() { return v; }  // Error, & with decltype(auto)
const decltype(auto) g21() { return v; }  // Error, const with   "
```

There is an additional restriction that an **`auto`** return type cannot be deduced from a braced initializer list:

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

auto v22 = { 1, 2, 3 };            // OK, deduced type initializer_list<int>
auto g22() { return { 1, 2, 3 }; } // Error, braced initializer list not allowed
```

If the declaration of `g22` deduced an initializer list return type instead of being disallowed, it would always return a dangling reference, as the initializer list would go out of scope before the function could return.

Deducing a `void` return type

ng-a-void-return-type

If the **`return`** statement for a function having a deduced return type is empty (i.e., **`return;`**) or if there are no **`return`** statements at all, then the return type is deduced as **`void`**. In such cases, the declared return type must be **`auto`**, **`const auto`**, or **`decltype(auto)`**, without additional reference, pointer, or other specifiers:

```
auto           g1() { }          // OK, deduced return type void
auto           g2() { return; }  // OK,    "       "     "     "
decltype(auto) g3() { }          // OK,    "       "     "     "
decltype(auto) g4() { return; }  // OK,    "       "     "     "
const auto     g5() { }          // OK,    "       "     "     "
auto*          g6() { return; }  // Error, no pointer returned
auto&          g7() { return; }  // Error, no reference returned
```

Multiple `return` statements

ple-return-statements

When there are multiple **`return`** statements in a function having a deduced return type, the return type is deduced from the textually first **`return`** statement in the function. The second and subsequent **`return`** statements must deduce the same return type as the first **`return`** statement, or the program is ill formed[2]:

[1] Version 10.2 and earlier versions of GCC that support C++14 permit **`const decltype(auto)`** for both variable and function declarations, even though the Standard forbids it.

[2] In C++17, discarded statements, such as in the body of an **`if constexpr`** statement whose condition is **`false`**, are not used for type deduction:

```
auto g1(int i)
{
    if (i & 1) { return 3 * i + 1; }  // Deduce return type int.
    else       { return i / 2; }      // OK, deduce return type int again.
}

auto g2(bool b)
{
    if (b) { return "hello"; }  // Deduce return type const char*.
    else   { return 0.1; }      // Error, deduced double does not match.
}
```

Type deduction on multiple **return** statements does not take conversions into account; all the deduced types must be identical:

```
auto g3(long li)
{
    if (li > 0) { return li; }   // Deduce return type long.
    else        { return 0;  }   // Error, deduced int does not match long.
}

auto g4(bool b)
{
    if (b) { return "text"; }   // Deduce return type const char*.
    else   { return nullptr; }  // Error, std::nullptr_t does not match.
}

struct S { S(int = 0); };  // convertible from int

auto g5(bool b)
{
    if (b) { return S(); }  // Deduced return type S
    else   { return 2; }    // Error, conversion to S not considered
}

int& f();

auto g6(int i)
{
    if (i > 0) { return i + 1; }  // Deduce return type int.
    else       { return f(); }    // OK, deduce return type int again.
}

decltype(auto) g7(int i)
```

```
auto f()  // deduces return type of const char*
{
    if constexpr (false) return 1;  // discarded return statement
    return "hello";                 // OK, nondiscarded return statement
}
```

```
{
    if (i > 0) { return i + 1; }  // Deduce return type int.
    else       { return f(); }    // Error, deduced int& doesn't match int.
}
```

Note that the second **return** statements in `g3`, `g4`, and `g5` do not consider possible conversions from the second **return** expression to the type deduced from the first **return** expression. The bodies of functions `g6` and `g7` are identical, but the latter produces an error because **decltype(auto)** preserves the value category of the expression `f()`, resulting in a different deduced return type in the second **return** statement than in the first.

Unlike **if** statements, the ternary conditional operator seeks to find a *common type* between the **true** and **false** conditions. Thus, return-type deduction that would be invalid using **if** statements might be valid when using the ternary conditional operator:

```
auto g8(long li)  // valid rewrite of g3
{
    return (li > 0) ? li : 0;  // OK, deduce common return type long.
}
```

Once the return type has been deduced, it can be used later in the same function, i.e., as the return type of a recursive call. If the return type would be needed before the first **return** statement is seen, the program is ill formed:

```
decltype(auto) g9(int i)
{
    if (i < 1) { return 0; }              // Deduce return type int.
    else       { return i + g9(i - 1); }  // OK, use previously deduced return
                                          // type to deduce int again.
}

decltype(auto) g10(int i)
{
    if (i > 1) { return i + g10(i - 1); }  // Error, return type not known yet
    else       { return 0; }
}
```

Perhaps surprisingly, `g9` cannot be rewritten using the ternary conditional operator because return-type deduction cannot occur until both the **true** and **false** branches of the ternary expression have been processed by the compiler:

```
decltype(auto) g11(int i)  // erroneous rewrite of g9
{
    return i < 1 ? 0 : i + g11(i - 1);
        // Error, g11 used before return deduced
}
```

It is legal to fall off the end of a function when the return type has already been deduced as non**void**, although the compiler is likely to generate a warning. However, a return without an expression will cause a deduction conflict with a **return** statement that *does* return a value:

```
auto g12(bool b) { if (b) return 1;         }  // OK, warning
```

```
auto g13(bool b) { if (b) return 1; return; }  // Error, deduction mismatch
```

g-a-deduced-return-type

Type of a function having a deduced return type

Deduced return types are allowed for almost every category of function, including free functions, static member functions, nonstatic member functions, function templates, member function templates, and conversion operators. Virtual functions, however, cannot have deduced return types:

```
auto free();                                    // OK, free function
template <typename T> auto templ();             // OK, function template

struct S
{
    static auto staticMember();                 // OK, static member function
    decltype(auto) member();                    // OK, nonstatic member function
    template <typename T> auto memberTempl();   // OK, member function template
    operator auto() const;                      // OK, conversion operator
    virtual auto virtMember();                  // Error, virtual function
};
```

When one of these functions is later defined or redeclared, it must use the *same* placeholder for the return type, even if the actual return type is known at the point of definition:

```
auto free() { return 8; }  // OK, redeclare and define with auto return type.

int S::staticMember() { return 4; }  // Error, must be declared auto
auto S::member() { return 5; }       // Error, previously decltype(auto)
```

The return type for `S::staticMember` is known to be **`int`** at the point of definition because the function body returns 4, but hard-coding the return type to be **`int`** instead of **`auto`** causes the definition not to match the declaration. In the case of `S::member`, both the declaration and the definition use placeholders, but the declaration uses **`decltype(auto)`** whereas the definition uses **`auto`**.

A function having a deduced return type has incomplete type until the function body has been seen; to be called or to have its address taken, the function's definition must appear earlier in the translation unit:

```
auto f1();

auto caller()
{
    f1();       // Error, return type of f1 is not known.
    return &f1; // Error, f1 has incomplete type.
}

auto f1() { return 1.2; }  // return type deduced as double but too late
```

Consequently, a function declared in a header (`.h`) file must have a definition in the same header file to be usable through the normal **`#include`** mechanism. In practice, such a func-

tion must be either a template or **inline**, lest the definition be imported into multiple translation units, violating the ODR:

```
// file1.h:
auto func1();                          // OK, declaration only

auto func2() { return 4; }             // noninline definition (dangerous)

inline auto func3() { return 'a'; }  // OK, inline definition

template <typename T>
decltype(auto) func4(T* t)             // OK, function template
{
    return *t;
}

// file2.cpp:
#include <file1.h>            // Error, IFNDR, redefinition of func2
double local2a = func1();     // Error, func1 return type is not known.
int    local2b = func2();     // Valid? Call one of the definitions of func2.
char   local2c = func3();     // OK, call to inline function func3
char   local2d = func4("a");  // OK, call to instantiation func4<const char>

// file3.cpp:
#include <file1.h>            // Error, IFNDR, redefinition of func2
auto func1() { return 1.2; }  // OK, defined to return double
double local3a = func1();     // OK
int    local3b = func2();     // Valid? Call one of the definitions of func2.
char   local3c = func3();     // OK, call to inline function func3
char   local3d = func4("b");  // OK, call to instantiation func4<const char>
```

Because `func1` is declared in `file1.h` but defined in `file3.cpp`, `file2.cpp` does not have enough information to deduce its return type. Conversely, `func2` has the reverse problem: there is an ODR violation because `func2` is redefined in every translation unit that has `#include <file1.h>`. The compiler is not required to diagnose most ODR violations, but linkers will typically complain about multiply-defined public symbols. Finally, `func3` is **inline** and `func4` is a template; unlike `func1`, their definitions are visible in each translation unit, making the deduced return type available, but unlike `func2`, they do not create an ODR violation.

trailing-return-types

Placeholders in trailing return types

If **auto** or **decltype(auto)** is used in a trailing return type, the meaning is the same as using the same placeholder as a leading return type:

```
auto f1() -> auto;
auto f2() -> decltype(auto);
auto f3() -> const auto&;

auto           f1();  // OK, compatible redeclaration of f1
```

```
decltype(auto) f2();  // OK, compatible redeclaration of f2
const auto&    f3();  // OK, compatible redeclaration of f3
```

When any trailing return type is specified, the leading return-type placeholder must be plain **auto**:

```
decltype(auto) f4() -> auto;  // Error, decltype(auto) with trailing return
auto&          f5() -> int&;  // Error, auto& with trailing return
```

-for-lambda-expressions

Deduced return types for lambda expressions

As described in Section 2.1."**??**" on page **??**, the return type of a closure call operator can be deduced automatically from its **return** statement(s):

```
auto y1 = [](int i) { return i + 1; };  // Deduce int.
```

The semantics of return-type deduction for lambda expressions is, by default, the same as for a function with declared return type **auto**. The semantics of **decltype(auto)** are available by using **decltype(auto)** in a trailing return type:

```
auto y2 = [](int& i) -> decltype(auto) { return i += 1; };  // Deduce int&.
```

Note that, even though return-type deduction is available for lambda expressions in C++11, it is only since C++14 that **decltype(auto)** is available. Prior to that, the preceding lambda expression would have required a more cumbersome and repetitious use of the **decltype** operator:

```
auto y3 = [](int& i) -> decltype(i+=1) { return i += 1; };  // C++11 compatible
```

tion-and-specialization

Template instantiation and specialization

Function templates are instantiated when they are selected by overload resolution. If the function has a deduced return type, then the template must be fully instantiated to deduce its return type even if the instantiation causes the program to be ill formed. This instantiation behavior differs from function templates with defined return types, where failure to compose a valid return type will result in a substitution failure that will benignly remove the template from the overload set (**SFINAE**):

```
struct S { };

int f1(void* p) { return 0; }       // matches any pointer type

template <typename T>
auto f1(T* p) -> decltype(*p *= 2)  // better match if *= is valid for T,int
{
    return *p *= 2;
}

int f2(void* p) { return 0; }       // matches any pointer type

template <typename T>
```

```
auto f2(T* p)                       // better match for nonvoid pointer type
{
    return *p *= 2;                 // OK, only if *= is valid for T,int
}

void g1()
{
    unsigned i;
    S        s;

    auto v1 = f1(&i);  // OK, calls f1<unsigned>(unsigned*)
    auto v2 = f1(&s);  // OK, calls f1(void*)

    auto v3 = f2(&i);  // OK, calls f2<unsigned>(unsigned*)
    auto v4 = f2(&s);  // Error, hard failure instantiating f2<S>(S*)
}
```

The first overload for `f1` accepts any pointer argument and returns integer `0`. The second overload for `f1` is a better match for a non**`void`** pointer only if the return type **`decltype`**`(*p *= 2)` is valid. If not, then the template specialization is removed from the overload set. Thus, `f1(&s)` will discard the `f1` template from consideration and instead call the less-specific `f1(`**`void`**`*)` function. Note that **`auto`** in combination with a trailing return type is not a deduced return type; the return type for the `f1` template is determined during overload resolution and does not require instantiation of the function body. The example takes advantage of the fact that, unlike a leading return type, the name of a function parameter can be used in the declaration of a trailing return type.

Conversely, the prototype for the `f2` function template will match *any* pointer type, regardless of whether it can eventually deduce a valid return type. Once it has been selected as the best overload, the `f2` template is fully instantiated, and its return type is deduced. If, during instantiation, `*p *= 2` fails to compile — as it does for `f2<S>` — the program is ill formed and results in a hard error; overload resolution is complete, so it is too late to remove the template specialization from the overload set.

An explicit instantiation declaration of a function template preceded by **`extern`** suppresses implicit instantiation of that specialization for the rest of the translation unit (see Section 2.1."**??**" on page **??**), regardless of whether the function template has a deduced return type. If, however, the function template is used in such a way that its return type must be deduced, then the template is instantiated anyway:

```
template <typename T> auto f(T t) { return t; }

extern template auto f(int);  // Suppress implicit instantiation of f<int>.
int (*p)(int) = f;            // f<int> is instantiated to deduce its return type.
```

The **`extern`** explicit instantiation declaration of **`f(int)`** does not instantiate **`f<int>`**, nor does it determine its return type. When used to initialize `p`, however, the return type must be deduced; **`f<int>`** is instantiated just for that purpose, but that instantiation does not eliminate the requirement that **`f(int)`** be explicitly instantiated elsewhere in the program, typically in a separate translation unit:

```
template auto f(int); // must appear somewhere in the program
```

Note that the explicit instantiation fails on some popular compilers if it is encountered after the implicit instantiation in the *same* translation unit.[3]

Any specialization or explicit instantiation of a function template with deduced return type must use the same placeholder, even if the return type could be expressed simply without the placeholder:

```
template <typename T> auto g(T t) { return t; }

template <>
auto g(double d) { return 7; }  // OK, explicit specialization, deduced as int

template auto g(int);           // OK, explicit instantiation, deduced as int

template <>
char g(char)   { return 'a'; }  // Error, must return auto

template <typename T>
T    g(T t, int) { return t; }  // OK, different template
```

Even though `auto g(char)` and `char g(char)` have the same return type, the latter is not a valid specialization of the former. If one of the contributors to this return-type mismatch occurs within a template, the error might not be diagnosed until the template is instantiated:

```
template <typename T>
class A
{
    static T s_value;  // private static member variable
    friend T h(T);     // declare friend function with known return type
};

template <typename T> T A<T>::s_value;

auto h(int i)
{
    return A<int>::s_value;
        // Error, h is redeclared with a different return-type specification.
        // Error, this function is not a friend of A<int>.
}
```

When `A<int>` is instantiated, the declaration of `T h(T)` within class template `A` fails because, although `auto h(int)` has the same prototype as `T h(T)`, where `T` is `int`, they are not considered the same function.

[3] Both Clang 12.0 and GCC 10.2 have bugs whereby they fail to explicitly instantiate a function template having a deduced return type if the implicit instantiation needed to deduce its return type is visible. See Clang bug 19551 (**?**) and GCC bug 99799 (**?**).

-conversion-functions

Placeholder conversion functions

The name of a conversion operator can be a placeholder. Multiple conversion operators can be defined in a single class, provided that no two have the same declared or deduced return type:

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

struct S
{
    static const int i;

    operator auto() { return 1; }
    operator long() { return 2L; }
    operator decltype(auto)() const { return (i); }
    operator const auto*() { return &i; }
};

const int S::i = 3;

void f1()
{
    S       s{};
    const S cs{};

    int         i1 = s;   // Convert to int.
    long        i2 = s;   // Convert to long.
    const int&  i3 = s;   // Convert to const int&.
    int         i4 = cs;  // Convert to const int&.
    long        i5 = cs;  // Convert to const int&.
    const int&  i6 = cs;  // Convert to const int&.
    long&       i7 = cs;  // Error, cannot convert to long&
    const int*  p1 = s;   // Convert to int*.

    assert(1  == i1);
    assert(2L == i2);
    assert(3  == i3);
    assert(3  == i4);
    assert(3  == i5);
    assert(3  == i6);

    assert(p1 == &i3);
    assert(p1 == &i6);
}
```

The same rules apply to these conversion operators as to placeholder return types of ordinary member functions. The last conversion operator, for example, combines **`auto`** with **`const`** and the pointer operator. Note, however, that because these operators do not have unique names, either their implementation must be inline within the class (as above), or they must be distinguishable in some other way, e.g., by cv-qualification:

```
struct R
{
    operator auto();        // OK, deduced type not known
    operator auto() const;  // OK, const qualified & deduced type not known
};

R::operator auto() { return "hello"; }  // OK, deduce type const char*.
R::operator auto() const { return 4; }  // OK, deduce type int.

void f2()
{
    R r;

    const char* s = r;  // OK, choose nonconst conversion to const char*.
    int         i = r;  // OK, choose const conversion to int.
}
```

In **struct** R, the two conversion operators can coexist even before their return types are deduced because one is **const** and the other is not. The deduced types must be known before the conversion operators are invoked, as usual. Note that, in the initialization of i, the **const** conversion to **int** is preferred over the non**const** conversion to **const char***, even though r is not **const**.

use-cases

Use Cases

omplicated-return-types

Complicated return types

In their book *Scientific and Engineering C++*[4] authors Barton and Nackman pioneered template techniques that are now widely used. They described a system for implementing SI units in which the individual unit exponents were held as template value parameters, e.g., a distance exponent of 3 would denote cubic meters. The type system was used to constrain unit arithmetic so that only correct combinations would compile. Addition and subtraction require units of the same dimensionality (e.g., square meters), whereas multiplication and division allow for mixed dimensions (e.g., dividing distance by time to get speed in meters per second).

We give a simplified version here, supporting three base unit types for distance in meters, mass in kilograms, and time in seconds:

```
// unit type holding a dimensional value in the MKS system
template <int DistanceExp, int MassExp, int TimeExp>
class Unit
{
    double d_value;

public:
    Unit() : d_value(0.0) { }
    explicit Unit(double v) : d_value(v) { }
```

[4] ?

```
    double value() const { return d_value; }
    Unit operator-() const { return Unit(-d_value); }
};

// predefined units, for convenience
using Scalar    = Unit<0, 0, 0>;   // dimensionless quantity
using Meters    = Unit<1, 0, 0>;   // distance in meters
using Kilograms = Unit<0, 1, 0>;   // mass in Kg
using Seconds   = Unit<0, 0, 1>;   // time in seconds
using Mps       = Unit<1, 0, -1>;  // speed in meters per second
```

Each different dimensional unit type is a different instantiation of `Unit`. The basic units for distance, mass, and time are each one-dimensional, whereas speed has an exponent of `1` for distance and `-1` for time, thus representing the unit, meters/second.

Summing two dimensional quantities requires that they have the same dimensionality, i.e., that they be represented by the same `Unit` specialization. The addition and subtraction operators are, therefore, straightforward to declare and implement:

```
template <int DD, int MD, int TD>
Unit<DD,MD,TD> operator+(Unit<DD,MD,TD> lhs, Unit<DD,MD,TD> rhs)
    // Add two quantities of the same dimensionality.
{
    return Unit<DD,MD,TD>(lhs.value() + rhs.value());
}

template <int DD, int MD, int TD>
Unit<DD,MD,TD> operator-(Unit<DD,MD,TD> lhs, Unit<DD,MD,TD> rhs)
    // Subtract two quantities of the same dimensionality.
{
    return Unit<DD,MD,TD>(lhs.value() - rhs.value());
}
```

Multiplication and division are more complicated because it is possible to, for example, divide distance by time to get speed. When you multiply two dimensional quantities, the exponents are added; when you divide them, the exponents are subtracted:

```
template <int DD1, int MD1, int TD1, int DD2, int MD2, int TD2>
auto operator*(Unit<DD1,MD1,TD1> lhs, Unit<DD2,MD2,TD2> rhs)
    // multiply two dimensional quantities to produce a new
{
    return Unit<DD1+DD2, MD1+MD2, TD1+TD2>(lhs.value() * rhs.value());
}

template <int DD1, int MD1, int TD1, int DD2, int MD2, int TD2>
auto operator/(Unit<DD1,MD1,TD1> lhs, Unit<DD2,MD2,TD2> rhs)
{
    return Unit<DD1-DD2, MD1-MD2, TD1-TD2>(lhs.value() / rhs.value());
}
```

The return types for the multiplicative operators are somewhat awkwardly long, and without deduced return types, those long names would need to appear twice, once in the function

declaration and once in the **return** statement. [5]

We can now use these operations to implement a function that returns the kinetic energy of a moving object:

```
auto kineticEnergy(Kilograms m, Mps v)
    // Return the kinetic energy of an object of mass m moving at velocity v.
{
    return m * (v * v) / Scalar(2);
}
```

The return type of this formula is determined automatically, without expressing the `Unit` template arguments directly. The returned unit is a joule, which can also be described as a kilogram $*$ meter^2/second^2, as our test program illustrates:

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

void f1()
{
    using Joules = Unit<2, 1, -2>;  // Energy in joules

    auto ke = kineticEnergy(Kilograms(4.0), Mps(12.5));
    static_assert(std::is_same<decltype(ke), Joules>::value, "");
    assert(312.4999 < ke.value() && ke.value() < 312.5001);
}
```

Because of automatic return-type deduction, naming the `Unit` instantiation of each intermediate computation within `kineticEnergy` was unnecessary. The **static_assert** in the code above proves that our formula has returned the correct final unit.

ompiler-apply-the-rules

Let the compiler apply the rules

The C++ rules for type promotion and conversion in expressions are complex and not easy to express in a return type. For example, many people could not tell you the rule for determining the return type when adding a value of type **int** to a value of type **unsigned int**. It is, in turn, difficult for the programmer to determine the correct return type for a function that returns the result of such an expression. This complication is compounded when the computation takes place in a function template. An explicit return type computed with **decltype** can be used to determine the type of an expression, but such determination requires duplicating the expression in the function declaration or fabricating a simpler expression that hopefully has the same type. When multiple **return** statements exist with

[5] As a workaround, it is possible to introduce a defaulted type template parameter to avoid the repetition of the return type:

```
template <int DD1, int MD1, int TD1, int DD2, int MD2, int TD2,
        typename R = Unit<DD1+DD2, MD1+MD2, TD1+TD2>>
R operator*(Unit<DD1,MD1,TD1> lhs, Unit<DD2,MD2,TD2> rhs)
{
    return R(lhs.value() * rhs.value());
}
```

different contents, there is no straightforward way to guarantee that they yield the same type.

Using a deduced return type eliminates the need to duplicate code or reconcile return expressions:

```
template <typename T1, typename T2>
auto add_or_subtract(bool b, T1 v1, T2 v2)
{
    if (b) { return v1 + v2; }
    else   { return v1 - v2; }
}
```

The template above deduces the return type of adding a value of type `T2` to a value of type `T1` and verifies that the same type is produced when subtracting a value of type `T2` from a value of type `T1`. If the two deduced types differ, an error diagnostic is produced rather than a silent promotion or conversion to the (possibly incorrect) manually determined type.

g-a-lambda-expression

Returning a lambda expression

A lambda expression generates a unique **closure** type that cannot be named and cannot appear as the operand of the **`decltype`** operator. The only way that a function can generate and return a **closure object** is through the use of a deduced return type. This capability lets us define functions that capture parameters and generate useful function objects:

```
#include <algorithm>  // std::is_partitioned
#include <vector>     // std::vector

template <typename T>
auto lessThanValue(const T& t)
{
    return [t](const auto& u) { return u < t; };
}

bool f1(const std::vector<int>& v, int pivot)
    // return true if v is partitioned around the pivot value
{
    return std::is_partitioned(v.begin(), v.end(), lessThanValue(pivot));
}
```

The `lessThanValue` function generates a functor — i.e., a closure object — that returns **`true`** if its argument is less than the captured `t` value. This functor is then used as an argument to `is_partitioned`.

Note that it is not possible to return the result of different lambda expressions in different **`return`** statements, as each lambda expression intrinsically has a different type than every other lambda expression, thus violating the requirements for return-type deduction:

```
auto comparator(bool reverse)
{
    if (reverse)
    {
        return [](int l, int r) { return l < r; };
```

```
    }
    else
    {
        return [](int l, int r) { return l > r; };  // Error, inconsistent type
    }
}
```

ng-of-wrapped-functions

Perfect returning of wrapped functions

A generic wrapper that performs some task before and/or after calling another function needs to preserve the type and value category of the returned value of the called function. Using **`decltype(auto)`** is the simplest method to achieve this "perfect returning" of the wrapped call. For example, a wrapper template might acquire a mutex lock, call an arbitrary function provided by the user, and return the value produced by the function:

```
#include <utility>  // std::forward
#include <mutex>    // std::mutex and std::lock_guard

template <typename Func, typename... Args>
decltype(auto) lockedInvoke(std::mutex& m, Func&& f, Args&&... args)
{
    std::lock_guard<std::mutex> mutexLock(m);
    return std::forward<Func>(f)(std::forward<Args>(args)...);
}
```

The mutex is released automatically by the destructor for `mutexLock`. The return value and value category from `f` is faithfully returned by `lockedInvoke`. Note that `lockedInvoke` relies on two other C++11 features — forwarding references (see Section 2.1."**??**" on page **??**) and variadic function templates (see Section 2.1."**??**" on page **??**) — to achieve **perfect forwarding** of its arguments to `f`.

g-return-type-deduction

Delaying return-type deduction

Sometimes, determining the return type of a function template requires instantiating the template recursively until the base case is found. In certain situations, these instantiations can cause unbounded compile-time recursion even when, logically, the recursion should terminate normally. Consider the recursive function template `n1`, which returns its template argument, `N`, through recursive instantiation, stopping when it calls the base case of `N == 0`:

```
template <int i>  struct Int{ };  // compile-time integer

int n1(Int<0>) { return 0; }      // base case for terminating recursion

template <int N>
auto n1(Int<N>) -> decltype(n1(Int<N-1>{}))
    // return N through recursive instantiation
{
    return n1(Int<N-1>{}) + 1;   // call to recursive instantiation
}
```

```
int result1 = n1(Int<10>{});     // Error, excessive compile-time recursion
```

On the surface, it looks like recursion should terminate after only 11 instantiations. The problem, however, is that the compiler must determine the return type of `n1` before it knows whether it will recurse or not. To compute return type **`decltype`**`(n1(Int<N-1>{}))`, the compiler must build an overload set for `n1`. The compiler finds two names that match, the base case `n1(Int<0>)` and the template `n1(Int<N>)`. Even if `N` is 0, the compiler must instantiate the latter in order to complete building the overload set. If `N` is 0, therefore, it will instantiate `n1<-1>`, even though it will never call it. Hence, the recursion will not stop until `n1` has been instantiated with every **`int`** value (though, in practice, the compiler will abort long before then).

When the return type is deduced using **`auto`** or **`decltype`**`(`**`auto`**`)`, the compiler adds the function to the overload set without having to determine its return type. Since the return type itself does not determine the result of overload resolution, we can use this fact to avoid unneeded instantiations . Return-type deduction will occur only for the function that is actually chosen by overload resolution, so the return type when `N < 0` will terminate recursion as expected:

```
int n2(Int<0>) { return 0; }     // base case for terminating recursion

template <int N>
auto n2(Int<N>)
    // return N through recursive instantiation
{
    return n2(Int<N-1>{}) + 1;   // call to recursive instantiation
}

int result2 = n2(Int<10>{});     // OK, returns 10
```

In the above rewrite, the call to `n2` when `N` is `1` selects the base case (non-template) version and does not recursively instantiate the template version of `n2`.

potential-pitfalls

Potential Pitfalls

action-and-insulation

Negative impacts on abstraction and insulation

If a library function provides an abstract interface, the user needs to read and understand only the function's declaration and its documentation. Except when maintaining the library itself, the function's implementation details are unimportant.

If a program insulates a library user from the library's implementation by placing the implementation code in a separate translation unit, compile-time coupling between library code and client code is reduced. A library that does not include function implementations in its header files can be rebuilt to provide updates without needing to recompile clients; only a relink is needed. Compilation times for client code are minimized by not needing to recompile library source code within header files.

Deduced function return types interfere with both abstraction and insulation and thus with the development of large-scale, comprehensible software. Because the return type cannot be determined without its implementation being visible to the compiler, publicly visible functions having deduced return types cannot be insulated; they must necessarily appear in

a header file as **`inline`** functions or function templates, thereby being recompiled for every client translation unit. In this regard, a function with deduced return type is no different than any other **`inline`** function or function template. What is new, however, is its impact on abstraction: To fully understand a function's interface — including its return type — the user must read its implementation.

To mitigate the loss of abstraction from deduced return types, the programmer of the function can carefully document the expected properties of the returned object, even in the absence of a specific concrete type. Interestingly, understanding the return value's *properties*, not merely its *type*, may yield a resulting function that is *more* abstract than one for which a known type had been specified.

reduced-clarity

Reduced clarity

Not having the return type of a function visible in its declaration can reduce the clarity of a program. Deduced return types work best when they appear on tiny function definitions, so that the determinative **`return`** statement is easily visible. Functions having deduced return types are also well suited for situations where the particulars of a return type are not especially useful, as in the case of iterator types associated with containers.

annoyances

Annoyances

ation-order-sensitivity

Implementation-order sensitivity

If a deduced return type is used for a recursive function or a pseudo-recursive function template, the textually first **`return`** statement must be the base case of the recursion:

```
auto fib(int n)
    // Compute the nth Fibonacci number
{
    if (n < 2) { return n; }              // base case, deduces int
    else { return fib(n-2) + fib(n-1); }  // OK, return type already known
}
```

The same code can be rearranged in a way that seems functionally identical but which now fails to compile due to return-type deduction happening too late:

```
auto fib2(int n)
    // Compute the nth Fibonacci number
{

    if (n >= 2) { return fib2(n-2) + fib2(n-1); }  // Error, unknown return type
    else        { return n; }                      // OK, but too late
}
```

Importantly, that multiple **`return`** statements must all deduce the same return type ensures that rearranging the order of **`return`** statements does not lead to subtle changes in the deduced return type of the function. For example, if the first **`return`** deduces the type **`short`** and the second **`return`** deduces the type **`long`**, it is probably preferable for the compiler to complain than to silently truncate the **`long`** to a **`short`**. This protection, therefore, prevents an occasional annoyance from becoming a dangerous pitfall.

inae-in-function-body

No SFINAE in function body

Substitution Failure Is Not An Error is often employed to conditionally remove a function template from an overload set.[6] Consider an overload set that aims to return the `size()` of any container passed to it and treat all **trivial** types as a single-element container:

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

int numElements(...)  // overload for all trivial types
{
    return 1;
}

template <typename T>
auto numElements(const T& container)  // overload for all classes with a
   -> decltype(container.size())      // size() member function
{
    return container.size();
}
```

This overload set depends on SFINAE to work properly. When `numElements` is invoked on an object for which the expression `container.size()` is valid, template substitution will proceed without error, and the resulting template specialization will be the best match during overload resolution. When invoked on an object of trivial type for which `container.size()` is invalid, the template substitution will fail and the would-be resulting specialization will be dropped from the overload set, leaving the C-style variadic function as the best overload match:

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

void testNumElements()
{
   std::vector<int> v = {1,2,3};
   assert(3 == numElements(v));
   assert(1 == numElements("Hello"));
}
```

Looking at the implementation of `numElements`, one might think to remove the duplication of the `container.size()` expression by leveraging a deduced return type. This change, however, will move the substitution error from the declaration to the function body, making the program **ill formed** rather than removing the offending template from the overload set:

```
int numElements2(...)  // overload for all trivial types
{
    return 1;
}
```

[6] C++20 introduces **concepts**, a much more expressive system to restrict the applicability of a function template that relies less on understanding the subtleties of SFINAE.

```
template <typename T>
auto numElements2(const T& container)  // deduced return type
{
    return container.size();  // valid only if container.size() is valid
}

void testNumElements2()
{
    std::vector<int> v = {1,2,3};
    assert(3 == numElements2(v));        // OK, template instantiation succeeds
    assert(1 == numElements2("Hello"));  // Error, tries to instantiate template
}
```

When `numElements2("hello")` is seen, the compiler must look at *both* overloads of `numElements2`. Since the template version is the better match, it tries to instantiate it but fails upon seeing `container.size()`. The error does not occur during overload resolution, so it is not considered a substitution failure. Rather than choosing the next best overload, the compilation fails.

see-also

See Also

- "??" (§1.1, p. ??) ♦ describes a feature that yields the type of an expression at compile time and which is used implicitly for type deduction.
- "??" (§1.1, p. ??) ♦ describes a less flexible but more deterministic alternative to deduced return types.
- "??" (§2.1, p. ??) ♦ describes the rules for deducing the type of an **`auto`** variable, which are the same rules as for deducing a return type declared with the **`auto`** placeholder.
- "??" (§2.1, p. ??) ♦ is used ubiquitously in C++11 and C++14 templates, especially function templates that wrap other function templates, often using deduced return types.
- "??" (§2.1, p. ??) ♦ describes lambda expressions, which had deduced return types in C++11 before they were introduced for regular functions in C++14.
- "??" (§3.2, p. ??) ♦ describes the rules for deducing the type of a **`decltype(auto)`** variable, which are the same rules as for deducing a return type declared with the **`decltype(auto)`** placeholder.

further-reading

Further Reading

None.