Chapter 1

Safe Features

ch-safe

Intro text should be here.

labelsec-safe-cpp11

Threadsafe Function-Scope `static` Variables

tion-static-variables

1.1— 0— 0

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

Description

iption-functionstatic

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

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

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

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

    return 0;
}
```

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

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

```
  assert
// ...

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

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

```
    return 0;
}
```

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

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

```
#include <assert.h>

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

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

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

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

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

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

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

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

Logger example

logger-example

Let's now consider a more realistic, real-world example in which a single object — e.g., `localLogger` in the example below — is used widely throughout a program (see also *Use Cases — Meyers Singleton* on page 7):

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

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

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

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

The destruction of **function-scope** **`static`** objects is and always has been guaranteed to be safe *provided* (1) no threads are running after returning from `main` and (2) **function-scope** **`static`** objects do not depend on each other during destruction; see *Po-*

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

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

tential Pitfalls — Depending on order-of-destruction of local objects after `main` *returns* on page 14.

multithreaded-contexts

Multithreaded contexts

Historically, initialization of **function-scope** `static`-duration objects was not guaranteed to be safe in a **multithreading context** because it was subject to **data races** if the function was called concurrently from multiple threads. One common but unreliable pre-C++11 workaround was the *double-checked lock pattern*; see *Appendix: C++03 Double-Checked Lock Pattern* on page 17.

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

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

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

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

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

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

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

In practice, however, **undefined behavior** (prior to C++11) might have manifested even earlier. When the second thread re-uses the storage claimed by the object in the first thread, it effectively ends the lifetime of one `static` `S` object to start the lifetime of the other

one. After that, any attempt to access the original `s` object would be **undefined behavior**, because its lifetime has ended, even though its destructor did not run. Hence, **undefined behavior** could manifest long before the second destructor is run at the end of the program.

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

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

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

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

Continuing our `logger` example from *Description — Logger example* on page 4, suppose that, to initialize a global facility, we are potentially calling a function, such as `getLogger`, concurrently from multiple threads using, say, the C++11 `std::thread` library utility. As an aside, the C++11 Standard Library provides copious utilities and abstractions related to multithreading. For starters, `std::thread` is a portable wrapper for a platform-specific thread handle provided by the operating system. When constructing an `std::thread` object with a **callable object** `functor`, a new thread invoking `functor` will be spawned. Note that `std::thread`'s destructor will *not* **join** the thread — it is safe to destroy an active `std::thread` object *only* if the `std::thread::join` member function has already been invoked. When the `std::thread` object's `join` member function is invoked, that function might need to block the caller until the *native* thread managed by the `std::thread` object being joined finishes its execution.

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

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

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

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

    // ...
```

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

    return 0;
}
```

As of C++11, the example above has no data races provided that `Logger::operator<<(const char*)` is designed properly for multithreaded use, even if the `Logger::Logger(const char* logFilePath)` constructor (i.e., the one used to configure the singleton instance of the logger) were not. That is to say, the implicit **critical section** that is guarded by the compiler includes evaluation of the initializer, which is why a recursive call to initialize a function-scope `static` variable is undefined behavior and is likely to result in deadlock; see *Description* on page 2. Such use of file-scope `static`s, however, is not foolproof; see *Potential Pitfalls — Depending on order-of-destruction of local objects after* `main` *returns* on page 14.

se-cases-functionstatic

Use Cases

meyers-singleton

Meyers Singleton

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

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

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

#endif

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

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

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

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

```
bool globalInitFlag = getGlobalS().isInitialized();

#include <cassert>  // standard assert macro

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

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

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

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

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

---

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

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

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

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

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

As discussed in *Description* on page 2, the augmentation of a thread-safety guarantee for the runtime initialization of **function-scope** `static` objects in C++11 minimizes the effort required to create a thread-safe singleton regardless of whether such safety guarantees turn out to be needed:

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

Note that, prior to C++11, the simple function-scope `static` implementation would not be safe if concurrent threads were vying to initialize the logger; see *Appendix: C++03 Double-Checked Lock Pattern* on page 17.

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

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

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

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

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

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

Consider, for example, an application on an early model of mobile phone where we want to refer to the phone's camera. Let's presume that a `Camera` class is a fairly involved and sophisticated mechanism. Initially we use the variant of the Meyers Singleton pattern where at most one `Camera` object can be present in the entire program. The next generation of the phone, however, turns out to have more than one camera, say, a front `Camera` and a back

`Camera`. Our brittle, *ToasterToothbrush*-like[5] design doesn't admit the dual-singleton use of the same fundamental `Camera` type. A more finely factored solution would be to implement the `Camera` type separately and then to provide a thin wrapper, e.g., perhaps using the **strong-`typedef` idiom** (see Section 1.1."**??**" on page **??**), corresponding to each singleton use:

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

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

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

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

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

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

```
#include <new>
```

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

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

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

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

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

Finally, cyclic initialization dependencies among global objects are simply not accommodated, and if such is needed, the design is fatally flawed regardless; see *Potential Pitfalls — Relying on initialization order of* `static objects` on page 12.

ation-of-global-objects

Thread-safe initialization of global objects

Providing a global object (e.g., for logging or monitoring purposes) can sometimes be convenient for an application because such objects are accessible from any other part of the program without having to pass them as explicit arguments. Similarly to the example introduced in *Description — Logger example* on page 4, consider a `MetricsCollector` class whose purpose is to collect runtime performance metrics for the program:

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

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

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

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

Assuming that `startBenchmark` and `endBenchmark` are designed to avoid race conditions, all that's left to do is to create a function returning a local `static` object of type `MetricsCollector`

(but see *Potential Pitfalls — Depending on order-of-destruction of local objects after* `main` *returns* on page 14):

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

The `getMetricsCollector` function in the code snippet above guarantees safe initialization of the `MetricsCollector` instance, initializing it exactly once on first invocation, but see *Potential Pitfalls — Depending on order-of-destruction of local objects after* `main` *returns* on page 14. Collecting metrics from any function scope, without requiring the function to accept an explicit `MetricsCollector` parameter, is now also possible. By creating an instance of the **RAII**-type `BenchmarkGuard`, the elapsed run time of the surrounding scope will be measured and collected:

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

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

tfalls-functionstatic

der-of-static-objects

Potential Pitfalls

Relying on initialization order of `static` objects

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

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

```
// a.cpp:
```

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

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

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

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

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

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

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

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

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

Now the program reliably executes without incident:

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

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

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

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

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

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

ts-after-main-returns

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

Within any given translation unit, the relative order of initialization of objects at file or namespace scope having static storage duration is well defined and predictable. As soon as we have a way to reference an object outside of the current translation unit, before `main` is entered, we are at risk of using the object before it has been initialized. Provided the initialization itself is not cyclic in nature, we can make use of function-scope `static` objects (see *Use Cases — Meyers Singleton* on page 7) to ensure that no such uninitialized use occurs, even across translation units before `main` is entered. The relative order of destruction of such function-scope `static` variables — even when they reside within the same translation unit — is not generally known, and reliance on such order can easily lead to **undefined behavior** in practice.

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

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

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

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

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

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

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

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

- If the `FileManager` object is destroyed prior to the `Logger` object, the program will have well-defined behavior.
- Otherwise, the program will have **undefined behavior** because the destructor of `FileManager` will invoke `getLogger`, which will now return a reference to a previously destroyed object.

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

In the case of low-level, widely used facilities, such as a logger, a conventional Meyers Singleton is counter-indicated. The two most common alternatives elucidated at the end of *Use Cases — Meyers Singleton* on page 7 involve never ending the lifetime of the mechanism at all. It is worth noting that truly global objects — such as `cout`, `cerr`, and `clog` — from the Standard `iostream` Library are typically not implemented using conventional methods and are in fact treated specially by the run time.

Annoyances

annoyances

Overhead in single-threaded applications

threaded-applications

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

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

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

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

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

See Also

see-also

None so far.

Further Reading

further-reading

- ?
- ?

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

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

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

le-checked-lock-pattern

Appendix: C++03 Double-Checked Lock Pattern

Prior to the introduction of the **function-scope** `static` object initialization guarantees discussed in *Description* on page 2, preventing multiple initializations of `static` objects and use before initialization of those same objects was still needed. Wrapping access in a mutex was often a significant performance cost, so using the unreliable, double-checked lock pattern was often attempted to avoid the overhead:

std::mutexstd::lock_guard

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

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

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

    return *loggerPtr;
}
```

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

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

So the final recommended solution for portable thread-safe initialization in modern C++ is to simply let the compiler do the work and to use the simplest implementation that gets

the job done, e.g., a Meyers Singleton (see *Use Cases — Meyers Singleton* on page 7):

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

labelsec-safe-cpp14

Chapter 2

Conditionally Safe Features

ch-conditional

Intro text should be here.
labelsec-conditional-cpp11

Strongly Typed Scoped Enumerations

enumclass

2.1— 0— 0

`enum class` is an alternative to the classic `enum` construct that simultaneously provides both stronger typing and an enclosing scope for its enumerated values.

Description

description-enumclass

Classic, C-style enumerations are useful and continue to fulfill important engineering needs:

```
enum EnumName { e_Enumerator0 /*= value0 */, e_EnumeratorN /* = valueN */ };
    // classic, C-style enum: enumerators are neither type-safe nor scoped
```

For more examples where the classic `enum` shines, see *Potential Pitfalls: Strong typing of an* `enum class` *can be counterproductive* on page 33 and *Annoyances: Scoped enumerations do not necessarily add value* on page 39. Still, innumerable practical situations occur in which enumerators that are both scoped and more type-safe would be preferred; see *Introducing the C++11* `enum class` on page 24.

Drawbacks and workarounds relating to unscoped C++03 enumerations

ed-c++03-enumerations

Since the enumerators of a classic `enum` leak out into the enclosing scope, if two unrelated enumerations that happen to use the same enumerator name appear in the same scope, an ambiguity could ensue:

```
enum Color { e_RED, e_ORANGE, e_YELLOW };    // OK
enum Fruit { e_APPLE, e_ORANGE, e_BANANA };  // Error, e_ORANGE is redefined.
```

Note that we use a lowercase, single-letter prefix, such as `e_`, to ensure that the uppercase enumerator name is less likely to collide with a legacy macro, which is especially useful in header files. The problems associated with the use of unscoped enumerations is exacerbated when those enumerations are placed in their own respective header files in the global or some other large namespace scope, such as `std`, for general reuse. In such cases, latent defects will typically not manifest unless and until the two enumerations are included in the same translation unit.

If the only issue were the leakage of the enumerators into the enclosing scope, then the long-established workaround of enclosing the enumeration within a `struct` would suffice:

```
struct Color { enum Enum { e_RED, e_ORANGE, e_YELLOW };  };   // OK
struct Fruit { enum Enum { e_APPLE, e_ORANGE, e_BANANA }; };  // OK (scoped)
```

Employing the C++03 workaround in the above code snippet implies that, when passing such an explicitly scoped, classical `enum` into a function, the distinguishing name of the `enum` is subsumed by its enclosing `struct` and the `enum` name itself, such as `Enum`, becomes **boilerplate code**:

```
int enumeratorValue1 = Color::e_ORANGE;  // OK
int enumeratorValue2 = Fruit::e_ORANGE;  // OK

void colorFunc(Color::Enum color);  // enumerated (scoped) Color parameter
void fruitFunc(Fruit::Enum fruit);  // enumerated (scoped) Fruit parameter
```

Hence, adding *just* scope to a classic, C++03 `enum` is easily doable and might be exactly what is indicated; see *Potential Pitfalls: Strong typing of an* `enum class` *can be counterproductive* on page 33.

yped,-c++03-enumerators

Drawbacks relating to weakly typed, C++03 enumerators

Historically, C++03 enumerations have been employed to represent at least two distinct concepts:

1. A collection of related, but not necessarily unique, named integral values
2. A pure, perhaps ordered, set of named entities in which cardinal value has no relevance

It will turn out that the modern `enum class` feature, which we will discuss in *Description: Introducing the C++11* `enum class`, is more closely aligned with this second concept.

A classic enumeration, by default, has an implementation-defined **underlying type** (see "Underlying Type '11" on page 57), which it uses to represent variables of that enumerated type as well as the values of its enumerators. While implicit conversion *to* an enumerated type is never permitted, when implicitly converting *from* a classical `enum` type to some arithmetic type, the `enum` promotes to integral types in a way similar to how its underlying type would promote using the rules of **integral promotion** and **standard conversion**:

```
void f()
{
    enum A { e_A0, e_A1, e_A2 };  // classic, C-style C++03 enum
    enum B { e_B0, e_B1, e_B2 };  //    "       "       "     "

    A a;  // Declare object a to be of type A.
    B b;  //   "       "    b  " "  "       B.

    a = e_B2;  // Error, cannot convert e_B2 to enum type A
    b = e_B2;  // OK, assign the value e_B2 (numerically 2) to b.
    a = b;     // Error, cannot convert enumerator b to enum type A
    b = b;     // OK, self-assignment
    a = 1;     // Error, invalid conversion from int 1 to enum type A
    a = 0;     // Error, invalid conversion from int 0 to enum type A

    bool     v = a;     // OK
    char     w = e_A0;  // OK
    unsigned y = e_B1;  // OK
    float    x = b;     // OK
    double   z = e_A2;  // OK
    char*    p = e_B0;  // Error, unable to convert e_B0 to char*
    char*    q = +e_B0; // Error, invalid conversion of int to char*
}
```

Notice that, in this example, the final two diagnostics for the attempted initializations of `p` and `q`, respectively, differ slightly. In the first, we are trying to initialize a pointer, `p`, with an enumerated type, `B`. In the second, we have creatively used the built-in unary-plus operator to explicitly promote the enumerator to an integral type before attempting to assign it to

a pointer, q. Even though the numerical value of the enumerator is 0 and such is known at compile time, implicit conversion to a pointer type from anything but the literal integer constant 0 is not permitted. Excluding esoteric user-defined types, only a literal 0 or, as of C++11, a value of type std::nullptr_t is implicitly convertible to an arbitrary pointer type; see "??" on page ??.

C++ fully supports comparing values of *classic* enum types with values of arbitrary **arithmetic type** as well as those of the same enumerated type; the operands of a comparator will be promoted to a sufficiently large integer type and the comparison will be done with those values. Comparing values having distinct enumerated types, however, is deprecated and will typically elicit a warning.[1]

-the-c++11-enum-class

Introducing the C++11 enum class

With the advent of modern C++, we now have a new, alternative enumeration construct, enum class, that simultaneously addresses strong type safety and lexical scoping, two distinct and often desirable properties:

```
enum class Name { e_Enumerator0 /* = value0 */, e_EnumeratorN /* = valueN */ };
    // enum class enumerators are both type-safe and scoped
```

Another major distinction is that the default **underlying type** for a C-style enum is **implementation defined**, whereas, for an enum class, it is always an int. See *Description: enum class and underlying type* on page 26 and *Potential Pitfalls: External use of opaque enumerators* on page 38.

The enumerators within an enum class are all scoped by its name, while classic enumerations leak the enumerators into the enclosing scope:

```
enum Vehicle  { e_CAR,   e_TRAIN, e_PLANE };
enum Geometry { e_POINT, e_LINE,  e_PLANE };  // Error, e_PLANE is redefined.
```

Unlike unscoped enumerations, enum class does not leak its enumerators into the enclosing scope and can therefore help avoid collisions with other enumerations having like-named enumerators defined in the same scope:

```
enum       VehicleUnscoped  { e_CAR, e_TRAIN, e_PLANE };
struct     VehicleScopedExplicitly { enum Enum { e_CAR, e_TRAIN, e_PLANE }; };
enum class VehicleScopedImplicitly { e_CAR, e_BOAT,  e_PLANE };
```

[1] As of C++20, attempting to compare two values of distinct classically enumerated types is a compile-time error. Note that explicitly converting at least one of them to an integral type — for example, using built-in unary plus — both makes our intentions clear and avoids warnings.

```
void test()
{
  if (e_A0 < 0)       { /* ... */ }  // OK, comparison with integral type
  if (1.0 != e_B1)    { /* ... */ }  // OK, comparison with arithmetic type
  if (A() <= e_A2)    { /* ... */ }  // OK, comparison with same enumerated type
  if (e_A0 == e_B0)   { /* ... */ }  // warning, deprecated (error as of C++20)
  if ( e_A0 == +e_B0) { /* ... */ }  // OK, unary + converts to integral type
  if (+e_A0 ==  e_B0) { /* ... */ }  // OK,   "        "     "     "      "
  if (+e_A0 == +e_B0) { /* ... */ }  // OK,   "        "     "     "      "
}
```

Just like an unscoped `enum` type, an object of type `enum class` is passed as a parameter to a function using the enumerator name itself:

```
void f1(VehicleUnscoped value);          // classic enumeration passed by value
void f2(VehicleScopedImplicitly value);  // modern enumeration passed by value
```

If we use the approach for adding scope to enumerators that is described in *Description: Drawbacks relating to weakly typed, C++03 enumerators* on page 23, the name of the enclosing `struct` together with a consistent name for the enumeration, such as `Enum`, has to be used to indicate an enumerated type:

```
void f3(VehicleScopedExplicitly::Enum value);
    // classically scoped enum passed by value
```

Qualifying the enumerators of a scoped enumeration is the same, irrespective of whether the scoping is explicit or implicit:

```
void g()
{
    f1(VehicleUnscoped::e_PLANE);
        // call f1 with an explicitly scoped enumerator

    f2(VehicleScopedImplicitly::e_PLANE);
        // call f2 with an implicitly scoped enumerator
}
```

Apart from implicit scoping, the modern, C++11 `enum class` deliberately does *not* support implicit conversion, in any context, to its **underlying type**:

```
void h()
{
    int i1 = VehicleScopedExplicitly::e_PLANE;
       // OK, scoped C++03 enum (implicit conversion)

    int i2 = VehicleScopedImplicitly::e_PLANE;
        // Error, no implicit conversion to underlying type

    if (VehicleScopedExplicitly::e_PLANE > 3) {} // OK
    if (VehicleScopedImplicitly::e_PLANE > 3) {} // Error, implicit conversion
}
```

Enumerators of an `enum class` do, however, admit equality and ordinal comparisons within their own type:

```
enum class E { e_A, e_B, e_C };  // By default, enumerators increase from 0.

static_assert(E::e_A < E::e_C, "");  // OK, comparison between same-type values
static_assert(0 == E::e_A, "");      // Error, no implicit conversion from E
static_assert(0 == static_cast<int>(E::e_A), "");  // OK, explicit conversion

void f(E v)
{
    if (v > E::e_A) { /* ... */ }  // OK, comparing values of same type, E
}
```

Note that incrementing an enumerator variable from one strongly typed enumerator's value to the next requires an explicit cast; see *Potential Pitfalls: Strong typing of an* `enum class` *can be counterproductive* on page 33.

`enum class` and underlying type

s-and-underlying-type

Since C++11, both scoped and unscoped enumerations permit explicit specification of their integral **underlying type**:

```
enum Ec : char { e_X, e_Y, e_Z };
    // underlying type is char

static_assert(1 == sizeof(Ec),     "");
static_assert(1 == sizeof Ec::e_X, "");

enum class Es : short { e_X, e_Y, e_Z };
    // underlying type is short int

static_assert(sizeof(short) == sizeof(Es),     "");
static_assert(sizeof(short) == sizeof Es::e_X, "");
```

Unlike a classic `enum`, which has an **implementation-defined** default **underlying type**, the default **underlying type** for an `enum class` is always `int`:

```
enum class Ei { e_X, e_Y, e_Z };
    // When not specified the underlying type of an enum class is int.

static_assert(sizeof(int) == sizeof(Ei),     "");
static_assert(sizeof(int) == sizeof Ei::e_X, "");
```

Note that, because the default **underlying type** of an `enum class` is specified by the Standard, eliding the enumerators of an `enum class` in a local redeclaration is *always* possible; see *Potential Pitfalls: External use of opaque enumerators* on page 38 and "Opaque `enum`s" on page 40.

Use Cases

use-cases-enumclass

Avoiding unintended implicit conversions to arithmetic types

s-to-arithmetic-types

Suppose that we want to represent the result of selecting one of a fixed number of alternatives from a drop-down menu as a simple unordered set of uniquely valued named integers. For example, this might be the case when configuring a product, such as a vehicle, for purchase:

```
struct Trans
{
    enum Enum { e_MANUAL, e_AUTOMATIC };  // classic, C++03 scoped enum
};
```

Although automatic promotion of a classic enumerator to `int` works well when typical use of the enumerator involves knowing its cardinal value, such promotions are less than ideal when cardinal values have no role in intended usage:

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

struct Trans
{                                           // explicitly scoped
    enum Enum { e_MANUAL, e_AUTOMATIC };    // classic enum
};                                          // (BAD IDEA)

int buildCar(Car* result, int numDoors, Trans::Enum trans)
{
    int status = Trans::e_MANUAL;    // Bug, accidental misuse!

    for (int i = 0; i < trans; ++i)  // Bug, accidental misuse!
    {
        attachDoor(i);
    }

    return status;
}
```

As shown in the example above, it is never correct for a value of type `Trans::Enum` to be assigned to, compared with, or otherwise modified like an integer; hence, *any* such use would necessarily be considered a mistake and, ideally, flagged by the compiler as an error. The stronger typing provided by `enum class` achieves this goal:

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

enum class Trans { e_MANUAL, e_AUTOMATIC };  // modern enum class (GOOD IDEA)

int buildCar(Car* result, int numDoors, Trans trans)
{
    int status = Trans::e_MANUAL;    // Error, incompatible types

    for (int i = 0; i < trans; ++i)  // Error, incompatible types
    {
        attachDoor(i);
    }

    return status;
}
```

By deliberately choosing the `enum class` over the *classic* `enum` above, we automate the detection of many common kinds of accidental misuse. Secondarily, we slightly simplify the interface of the function signature by removing the extra `::Enum` boilerplate qualifications required of an explicitly scoped, less-type-safe, classic `enum`, but see *Potential Pitfalls: Strong typing of an* `enum class` *can be counterproductive* on page 33.

In an unlikely event that the numeric value of a strongly typed enumerator is needed (e.g., for serialization), it can be extracted explicitly via a `static_cast`:

```
    enum class

const int manualIntegralValue    = static_cast<int>(Trans::e_MANUAL);
```

```
const int automaticIntegralValue = static_cast<int>(Trans::e_AUTOMATIC);
static_assert(0 == manualIntegralValue,    "");
static_assert(1 == automaticIntegralValue, "");
```

g-namespace-pollution

Avoiding namespace pollution

Classic, C-style enumerations do not provide scope for their enumerators, leading to unintended latent name collisions:

```
// vehicle.h:
// ...
enum Vehicle  { e_CAR, e_TRAIN, e_PLANE };  // classic, C-style enum
// ...

// geometry.h:
// ...
enum Geometry { e_POINT, e_LINE,  e_PLANE };  // classic, C-style enum
// ...

// client.cpp:
#include <vehicle.h>  // OK
#include <geometry.h> // Error, e_PLANE redefined
// ...
```

The common workaround is to wrap the `enum` in a `struct` or `namespace`:

```
// vehicle.h:
// ...
struct Vehicle {                           // explicitly scoped
    enum Enum { e_CAR, e_TRAIN, e_PLANE };  // classic, C-style enum
};
// ...

// geometry.h:
// ...
struct Geometry {                           // explicitly scoped
    enum Enum { e_POINT, e_LINE, e_PLANE };  // classic, C-style enum
};
// ...

// client.cpp:
#include <vehicle.h>    // OK
#include <geometry.h>   // OK, enumerators are scoped explicitly.
// ...
```

If implicit conversions of enumerators to integral types are not required, we can achieve the same scoping effect with much more type safety and slightly less boilerplate — i.e., without the `::Enum` when declaring a variable — by employing `enum class` instead:

```
// vehicle.h:
// ...
```

```
enum class Vehicle { e_CAR, e_TRAIN, e_PLANE };
// ...

// geometry.h:
// ...
enum class Geometry { e_POINT, e_LINE, e_PLANE };
// ...

// client.cpp:
#include <vehicle.h>  // OK
#include <geometry.h>   // OK, enumerators are scoped implicitly.
// ...
```

loading-disambiguation

Improving overloading disambiguation

Overloaded functions are notorious for providing opportunities for misuse. Maintenance difficulties are exacerbated when arguments for these overloads are convertible to more than a single parameter in the function. As an illustration of the compounding of such maintenance difficulties, suppose that we have a widely used, named type, `Color`, and the numeric values of its enumerators are small, unique, and irrelevant. Imagine we have chosen to represent `Color` as a *classic* `enum`:

```
struct Color
{                                           // explicitly scoped
    enum Enum { e_RED, e_BLUE /*, ...*/ };  // classic, C-style enum
};                                          // (BAD IDEA)
```

Suppose further that we have provided two overloaded functions, each having two parameters, with one signature's parameters including the enumeration `Color`:

```
void clearScreen(int pattern, int orientation);          // (0)
void clearScreen(Color::Enum background, double alpha);  // (1)
```

Depending on the types of the arguments supplied, one or the other functions will be selected or else the call will be ambiguous and the program will fail to compile[2]:

```
void f0()
{
```

[2]GCC version 7.4.0 incorrectly diagnoses both ambiguity errors as warnings, although it states in the warning that it is an error:

```
warning: ISO C++ says that these are ambiguous, even though the worst conversion for the
         first is better than the worst conversion for the second:

note: candidate 1: void clearScreen(int, int)
void clearScreen(int pattern, int orientation);
     ^~~~~~~~~~~
note: candidate 2: void clearScreen(Color::Enum, double)
void clearScreen(Color::Enum background, double alpha;
     ^~~~~~~~~~~
```

```
    clearScreen(1           , 1           );  // calls (0) above
    clearScreen(1           , 1.0         );  // calls (0) above
    clearScreen(1           , Color::e_RED);  // calls (0) above

    clearScreen(1.0         , 1           );  // calls (0) above
    clearScreen(1.0         , 1.0         );  // calls (0) above
    clearScreen(1.0         , Color::e_RED);  // calls (0) above

    clearScreen(Color::e_RED, 1           );  // Error, ambiguous call
    clearScreen(Color::e_RED, 1.0         );  // calls (1) above
    clearScreen(Color::e_RED, Color::e_RED);  // Error, ambiguous call
}
```

Now suppose that we had instead defined our `Color` enumeration as a modern `enum class`:

```
enum class Color { e_RED, e_BLUE /*, ...*/ };

void clearScreen(int pattern, int orientation);    // (2)
void clearScreen(Color background, double alpha);  // (3)
```

The function that will be called from a given set of arguments becomes clear:

```
void f1()
{
    clearScreen(1           , 1           );  // calls (2) above
    clearScreen(1           , 1.0         );  // calls (2) above
    clearScreen(1           , Color::e_RED);  // Error, no matching function

    clearScreen(1.0         , 1           );  // calls (2) above
    clearScreen(1.0         , 1.0         );  // calls (2) above
    clearScreen(1.0         , Color::e_RED);  // Error, no matching function

    clearScreen(Color::e_RED, 1           );  // calls (3) above
    clearScreen(Color::e_RED, 1.0         );  // calls (3) above
    clearScreen(Color::e_RED, Color::e_RED);  // Error, no matching function
}
```

Returning to our original, classic-`enum` design, suppose that we find we need to add a third parameter, `bool z`, to the second overload:

```
void clearScreen(int pattern, int orientation);                 // (0)
void clearScreen(Color::Enum background, double alpha, bool z);  // (4) classic
```

If our plan is that any existing client calls involving `Color::Enum` will now be flagged as errors, we are going to be very disappointed:

```
void f2()
{
    clearScreen(Color::e_RED, 1.0);  // calls (0) above
}
```

In fact, every combination of arguments above — all nine of them — will call function (0) above with no warnings at all:

```
void f3()
{
    clearScreen(1           , 1           );  // calls (0) above
    clearScreen(1           , 1.0         );  // calls (0) above
    clearScreen(1           , Color::e_RED);  // calls (0) above

    clearScreen(1.0         , 1           );  // calls (0) above
    clearScreen(1.0         , 1.0         );  // calls (0) above
    clearScreen(1.0         , Color::e_RED);  // calls (0) above

    clearScreen(Color::e_RED, 1           );  // calls (0) above
    clearScreen(Color::e_RED, 1.0         );  // calls (0) above
    clearScreen(Color::e_RED, Color::e_RED);  // calls (0) above
}
```

Finally, let's suppose again that we have used `enum class` to implement our `Color` enumeration:

```
void clearScreen(int pattern, int orientation);           // (2)
void clearScreen(Color background, double alpha, bool z);  // (5) modern

void f4()
{
    clearScreen(Color::e_RED, 1.0);  // Error, no matching function
}
```

And in fact, the *only* calls that succeed unmodified are precisely those that do not involve the enumeration `Color`, as desired:

```
void f5()
{
    clearScreen(1           , 1           );  // calls (2) above
    clearScreen(1           , 1.0         );  // calls (2) above
    clearScreen(1           , Color::e_RED);  // Error, no matching function

    clearScreen(1.0         , 1           );  // calls (2) above
    clearScreen(1.0         , 1.0         );  // calls (2) above
    clearScreen(1.0         , Color::e_RED);  // Error, no matching function

    clearScreen(Color::e_RED, 1           );  // Error, no matching function
    clearScreen(Color::e_RED, 1.0         );  // Error, no matching function
    clearScreen(Color::e_RED, Color::e_RED);  // Error, no matching function
}
```

Bottom line: Having a *pure* enumeration — such as `Color`, used widely in function signatures — be strongly typed can only help to expose accidental misuse but, again, see *Potential Pitfalls: Strong typing of an* `enum class` *can be counterproductive* on page 33.

Note that strongly typed enumerations help to avoid accidental misuse by requiring an explicit *cast* should conversion to an arithmetic type be desired:

```
void f6()
```

```
{
    clearScreen(Color::e_RED, 1.0);                  // Error, no match
    clearScreen(static_cast<int>(Color::e_RED), 1.0); // OK, calls (2) above
    clearScreen(Color::e_RED, 1.0, false);           // OK, calls (5) above
}
```

numerators-themselves

Encapsulating implementation details within the enumerators themselves

In rare cases, providing a pure, ordered enumeration having unique (but not necessarily contiguous) numerical values that exploit lower-order bits to categorize and make readily available important individual properties might offer an advantage, such as in performance. Note that in order to preserve the ordinality of the enumerators overall, the higher-level bits must encode their relative order. The lower-level bits are then available for arbitrary use in the implementation..

For example, suppose that we have a `MonthOfYear` enumeration that encodes the months that have 31 days in their least-significant bit and an accompanying `inline` function to quickly determine whether a given enumerator represents such a month:

```
#include <type_traits>  // std::underlying_type

enum class MonthOfYear : unsigned char  // optimized to flag long months
{
    e_JAN = ( 1 << 4) + 0x1,
    e_FEB = ( 2 << 4) + 0x0,
    e_MAR = ( 3 << 4) + 0x1,
    e_APR = ( 4 << 4) + 0x0,
    e_MAY = ( 5 << 4) + 0x1,
    e_JUN = ( 6 << 4) + 0x0,
    e_JUL = ( 7 << 4) + 0x1,
    e_AUG = ( 8 << 4) + 0x1,
    e_SEP = ( 9 << 4) + 0x0,
    e_OCT = (10 << 4) + 0x1,
    e_NOV = (11 << 4) + 0x0,
    e_DEC = (12 << 4) + 0x1
};

bool hasThirtyOneDays(MonthOfYear month)
{
    return static_cast<std::underlying_type<MonthOfYear>::type>(month) & 0x1;
}
```

In the example above, we are using a new cross-cutting feature of all enumerated types that allows the client defining the type to specify its underlying type precisely. In this case, we have chosen an `unsigned char` to maximize the number of flag bits while keeping the overall size to a single byte. Three bits remain available. Had we needed more flag bits, we could have just as easily used a larger underlying type, such as `unsigned short`; see "Underlying Type '11" on page 57.

In case `enum`s are used for encoding purposes, the public clients are not intended to make use of the cardinal values; hence clients are well advised to treat them as implementation

details, potentially subject to change without notice. Representing this enumeration using the modern `enum class`, instead of an explicitly scoped classic `enum`, deters clients from making any use (apart from same-type comparisons) of the cardinal values assigned to the enumerators. Notice that implementors of the `hasThirtyOneDays` function will require a verbose but efficient `static_cast` to resolve the cardinal value of the enumerator and thus make the requested determination as efficiently as possible.

tial-pitfalls-enumclass

an-be-counterproductive

Potential Pitfalls

Strong typing of an `enum class` can be counterproductive

The additive value in using a modern `enum class` is governed *solely* by whether its stronger typing, *not* its implicit scoping, of its enumerators would be beneficial in its anticipated typical usage. If the expectation is that the client will never need to know the specific values of the enumerators, then use of the modern `enum class` is often just what's needed. But if the cardinal values themselves are ever needed during typical use, extracting them will require the client to perform an explicit cast. Beyond mere inconvenience, encouraging clients to use casts invites defects.

Suppose, for example, we have a function, `setPort`, from an external library that takes an integer port number:

```
int setPort(int portNumber);
    // Set the current port; return 0 on success and a nonzero value otherwise.
```

Suppose further that we have used the modern `enum class` feature to implement an enumeration, `SysPort`, that identifies well-known ports on our system:

```
enum class SysPort { e_INPUT = 27, e_OUTPUT = 29, e_ERROR = 32, e_CTRL = 6 };
    // enumerated port values used to configure our systems
```

Now suppose we want to call the function `f` using one of these enumerated values:

```
void setCurrentPortToCtrl()
{
    setPort(SysPort::e_CTRL);  // Error, cannot convert SetPort to int
}
```

Unlike the situation for a *classic* `enum`, no implicit conversion occurs from an `enum class` to its underlying integral type, so anyone using this enumeration will be forced to somehow explicitly **cast** the enumerator to some arithmetic type. There are, however, multiple choices for performing this cast:

```
#include <type_traits>  // std::underlying_type

void test()
{
    setPort(int(SysPort::e_CTRL));                                    // (1)
    setPort((int)SysPort::e_CTRL);                                    // (2)
    setPort(static_cast<int>(SysPort::e_CTRL));                       // (3)
    setPort(static_cast<std::underlying_type<SysPort>::type>(         // (4)
                                                    SysPort::e_CTRL));
    setPort(static_cast<int>(                                         // (5)
```

```
                static_cast<std::underlying_type<SysPort>::type>(SysPort::e_CTRL)));
}
```

Any of the above casts would work in this case, but consider a future where a platform changed `setPort` to take a `long` and the control port was changed to a value that cannot be represented as an `int`:

```
int setPort(long portNumber);
enum class SysPort : unsigned { e_INPUT = 27, e_OUTPUT = 29, e_ERROR = 32,
                                e_CTRL = 0x80000000 };
    // enumerated port values used to configure our systems
```

Only casting method (4) above will pass the correct value for `e_CTRL` to this new `setPort` implementation. The other variations will all pass a negative number for the port, which would certainly not be the intention of the user writing this code. A classic C-style `enum` would have avoided any manually written cast entirely and the proper value would propagate into `setPort` even as the range of values used for ports changes:

```
    std::is_same

struct SysPort  // explicit scoping for a classic, C-style enum
{
    enum Enum { e_INPUT = 27, e_OUTPUT = 29, e_ERROR = 32,
                e_CTRL = 0x80000000 };

    // Note that the underlying type of Enum is implicit and will be
    // large enough to represent all of these values.
    static_assert(
        std::is_same<std::underlying_type<Enum>::type,unsigned>::value, "");
};

void setCurrentPortToCtrl()
{
    setPort(SysPort::e_CTRL);  // OK, SysPort::Enum promotes to long.
}
```

When the intended client will depend on the cardinal values of the enumerators during routine use, we can avoid tedious, error-prone, and repetitive casting by instead employing a classic, C-style `enum`, possibly nested within a `struct` to achieve explicit scoping of its enumerators. The subsections that follow highlight specific cases in which classic, C-style, C++03 `enum`s are appropriate.

ns-of-named-constants

Misuse of `enum class` for collections of named constants

When constants are truly independent, we are often encouraged to avoid enumerations altogether, preferring instead individual constants; see "**??**" on page **??**. On the other hand, when the constants all participate within a coherent theme, the expressiveness achieved using a *classic* `enum` to aggregate those values is compelling. Another advantage of an enumerator over an individual constant is that the enumerator is guaranteed to be a **compile-time constant** (see "**??**" on page **??**) and a **prvalue** (see "**??**" on page **??**), which never needs static storage and cannot have its address taken.

For example, suppose we want to collect the coefficients for various numerical suffixes representing *thousands*, *millions*, and *billions* using an enumeration:

```
enum class S0 { e_K = 1000, e_M = e_K * e_K, e_G = e_M * e_K };  // (BAD IDEA)
```

A client trying to access one of these enumerated values would need to cast it explicitly:

```
void client0()
{
    int distance = 5 * static_cast<int>(S0::e_K);  // casting is error-prone
    // ...
}
```

By instead making the enumeration an explicitly scoped, *classic* `enum` nested within a `struct`, no casting is needed during typical use:

```
struct S1  // scoped
{
    enum Enum { e_K = 1000, e_M = e_K * e_K, e_G = e_M * e_K };
        // *classic* enum (GOOD IDEA)
};

void client1()
{
    int distance = 5 * S1::e_K;  // no casting required during typical use
    // ...
}
```

If the intent is that these constants will be specified and used in a purely local context, we might choose to drop the enclosing scope, along with the name of the enumeration itself; see "**??**" on page **??**:

```
void client2()
{
    enum { e_K = 1000, e_M = e_K * e_K, e_G = e_M * e_K };  // function scoped

    double salary = 95 * e_K;
    double netWorth = 0.62 * e_M;
    double companyRevenue = 47.2 * e_G;
    // ...
}
```

We sometimes use the lowercase prefix `k_` instead of `e_` to indicate salient **compile-time constants** that are not considered part of an enumerated set, irrespective of whether they are implemented as enumerators:

```
enum { k_NUM_PORTS = 500, k_PAGE_SIZE = 512 };     // compile-time constants
static const double k_PRICING_THRESHOLD = 0.03125; // compile-time constant
```

ociation-with-bit-flags

Misuse of `enum class` in association with bit flags

Using `enum class` to implement enumerators that are intended to interact closely with arithmetic types will typically require the definition of arithmetic and bitwise operator overloads

between values of the same enumeration and between the enumeration and arithmetic types, leading to yet more code that needs to be written, tested, and maintained. This is often the case for bit flags. Consider, for example, an enumeration used to control a file system:

```
enum class Ctrl { e_READ = 0x1, e_WRITE = 0x2, e_EXEC = 0x4 };  // (BAD IDEA)
    // low-level bit flags used to control file system

void chmodFile(int fd, int access);
    // low-level function used to change privileges on a file
```

We could conceivably write a series of functions to combine the individual flags in a type-safe manner:

```
#include <type_traits>  // std::underlying_type

int flags() { return 0; }
int flags(Ctrl a) { return static_cast<std::underlying_type<Ctrl>::type>(a); }
int flags(Ctrl a, Ctrl b) { return flags(a) | flags(b); }
int flags(Ctrl a, Ctrl b, Ctrl c) { return flags(a, b) | flags(c); }

void setRW(int fd)
{
    chmodFile(fd, flags(Ctrl::e_READ, Ctrl::e_WRITE));  // (BAD IDEA)
}
```

Alternatively, a *classic*, C-style `enum` nested within a `struct` achieves what's needed:

```
struct Ctrl // scoped
{
    enum Enum { e_READ = 0x1, e_WRITE = 0x2, e_EXEC = 0x4 };  // classic enum
        // low-level bit flags used to control file system (GOOD IDEA)
};

void chmodFile(int fd, int access);
    // low-level function used to change privileges on a file

void setRW(int fd)
{
    chmodFile(fd, Ctrl::e_READ | Ctrl::e_WRITE);  // (GOOD IDEA)
}
```

iation-with-iteration

Misuse of `enum class` in association with iteration

Sometimes the relative values of enumerators are considered important as well. For example, let's again consider enumerating the months of the year:

```
enum class MonthOfYear  // modern, strongly typed enumeration
{
    e_JAN, e_FEB, e_MAR,  // winter
    e_APR, e_MAY, e_JUN,  // spring
    e_JUL, e_AUG, e_SEP,  // summer
```

```
    e_OCT, e_NOV, e_DEC,  // autumn
};
```

If all we need to do is compare the ordinal values of the enumerators, there's no problem:

```
bool isSummer(MonthOfYear month)
{
    return MonthOfYear::e_JUL <= month && month <= MonthOfYear::e_AUG;
}
```

Although the `enum class` features allow for relational and equality operations between like-typed enumerators, no arithmetic operations are supported directly, which becomes problematic when we need to iterate over the enumerated values:

```
void doSomethingWithEachMonth()
{
    for (MonthOfYear i =  MonthOfYear::e_JAN;
                    i <= MonthOfYear::e_DEC;
                  ++i)  // Error, no match for ++
    {
        // ...
    }
}
```

To make this code compile, an explicit cast from and to the enumerated type will be required:

```
void doSomethingWithEachMonth()
{
    for (MonthOfYear i =  MonthOfYear::e_JAN;
                    i <= MonthOfYear::e_DEC;
                    i = static_cast<MonthOfYear>(static_cast<int>(i) + 1))
    {
        // ...
    }
}
```

Alternatively, an auxiliary, helper function could be supplied to allow clients to bump the enumerator:

```
MonthOfYear nextMonth(MonthOfYear value)
{
    return static_cast<MonthOfYear>(static_cast<int>(value) + 1);
}

void doSomethingWithEachMonth()
{
    for (MonthOfYear i =  MonthOfYear::e_JAN;
                    i <= MonthOfYear::e_DEC;
                    i = nextMonth(i))
    {
        // ...
    }
}
```

If, however, the cardinal value of the `MonthOfYear` enumerators is likely to be relevant to clients, an explicitly scoped *classic* `enum` should be considered as a viable alternative:

```
struct MonthOfYear  // explicit scoping for enum
{
    enum Enum
    {
        e_JAN, e_FEB, e_MAR,  // winter
        e_APR, e_MAY, e_JUN,  // spring
        e_JUL, e_AUG, e_SEP,  // summer
        e_OCT, e_NOV, e_DEC,  // autumn
    };
};

bool isSummer(MonthOfYear::Enum month)  // must now pass nested Enum type
{
    return MonthOfYear::e_JUL <= month && month <= MonthOfYear::e_AUG;
}

void doSomethingWithEachMonth()
{
    for (int i =  MonthOfYear::e_JAN;  // iteration variable is now an int
             i <= MonthOfYear::e_DEC;
            ++i)  // OK, convert to underlying type
    {
        // ... (might require cast back to enumerated type)
    }
}
```

Note that such code presumes that the enumerated values will (1) remain in the same order and (2) have contiguous numerical values irrespective of the implementation choice.

enumerators-enumclass

External use of opaque enumerators

Since `enum class` types have an **underlying type** of `int` by default, clients are always able to (re)declare it, as a **complete type**, without its enumerators. Unless the opaque form of an `enum class`'s definition is exported in a header file separate from the one implementing the publicly accessible full definition, external clients wishing to exploit the opaque version will experience an *attractive nuisance* in that they can provide it locally, along with its **underlying type**, if any.

If the underlying type of the full definition were to subsequently change, any program incorporating the original elided definition locally and also the new, full one from the header would become silently **ill formed, no diagnostic required (IFNDR)**; see "Opaque `enum`s" on page 40.

annoyances-enumclass

Annoyances

t-necessarily-add-value

Scoped enumerations do not necessarily add value

When the enumeration is local, say, within the scope of a given function, forcing an additional scope on the enumerators is superfluous. For example, consider a function that returns an integer status `0` on success and a nonzero value otherwise:

```
int f()
{
    enum { e_ERROR = -1, e_OK = 0 } result = e_OK;
    // ...
    if (/* error 1 */) { result = e_ERROR; }
    // ...
    if (/* error 2 */) { result = e_ERROR; }
    // ...
    return result;
}
```

Use of `enum class` in this context would require potentially needless qualification — and perhaps even casting — where it might not be warranted:

```
int f()
{
    enum class RC { e_ERROR = -1, e_OK = 0 } result = RC::e_OK;
    // ...
    if (/* error 1 */) { result = RC::e_ERROR; } // undesirable qualification
    // ...
    if (/* error 2 */) { result = RC::e_ERROR; } // undesirable qualification
    // ...
    return static_cast<int>(result);  // undesirable explicit cast
}
```

see-also

See Also

- "Underlying Type '11" (Section 2.1, p. 57) ♦ Absent implicit conversion to integrals, `enum` class values may use `static_cast` in conjunction with their underlying type.
- "Opaque `enum`s" (Section 2.1, p. 40) ♦ Sometimes it is useful to entirely insulate individual enumerators from clients.

further-reading

Further Reading

- ?
- ?

Opaque Enumeration Declarations

enumopaque
meration-declarations 2.2— 0— 0

Enumerated types, such as an `enum` or `enum` class (see Section 2.1."`enum class`" on page 22), whose underlying type (see Section 2.1."Underlying Type '11" on page 57) is well-specified, can be declared without being defined, i.e., declared without its enumerators.

Description

description

We identify two distinct forms of **opaque declarations**, i.e., declarations that are not also definitions:

1. A **forward declaration** has some translation unit in which the full definition and that declaration both appear. This can be a declaration in a header file where the definition is in the same header or in the corresponding implementation file. It can also be a declaration that appears in the same implementation file as the corresponding definition.

2. A **local declaration** has no translation unit that includes both that declaration and the corresponding full definition.

A classic (C++03) C-style `enum` cannot have **opaque declarations**, nor can its definition be repeated within the same **translation unit** (TU):

```
enum E0;                // Error, opaque declaration
enum    { e_A0, e_B0 }; // OK, anonymous classic (only) enum
enum E1 { e_A1, e_B1 }; // OK, definition
enum E1;                // OK, redeclaration in same TU
enum E2 { e_A2, e_B2 }; // OK, definition
enum E2 { e_A2, e_B2 }; // Error, redefinition in same TU
```

The underlying integral types used to represent objects of each of the (classic) enumerations in the example above is **implementation defined**, making all of them ineligible for **opaque declaration**. This restriction on **opaque declarations** exists because the specific values of the enumerators might affect the underlying type (e.g., size, alignment, `signed`ness), and therefore the declaration alone cannot be used to create objects of that type. A declaration that specifies the underlying type or a full definition can, however, be used to create objects of that type. Specifying an underlying type explicitly makes **opaque declaration** possible:

```
enum    : char { e_A, e_B };     // OK, (anonymous) complete definition
enum E3 : char;                  // OK, forward declaration w/underlying type
enum E3 : char { e_A3, e_B3 };   // OK, compatible complete definition
enum E4 : short { e_A4, e_B4 };  // OK, complete definition
enum E4 : short;                 // OK, compatible opaque redeclaration
enum E4 : int;                   // Error, incompatible opaque redeclaration
enum E5 : int { e_A5, e_B5 };    // OK, complete definition
enum E5 : int { e_A5, e_B5 };    // Error, complete redefinition in same TU
```

The modern (C++11) `enum class`, which provides its enumerators with (1) stronger typing and (2) an enclosing scope, also comes with a default **underlying type** of `int`, thereby making it eligible to be declared without a definition (even without explicit qualification):

```
enum class E6;                 // OK, implicit underlying type (int)
enum class E6 : int;           // OK, explicit matching underlying type
enum class E6 { e_A3, e_B3 };  // OK, compatible complete definition
enum class E7 { e_A4, e_B4 };  // OK, complete definition, int underlying type
enum class E7;                 // OK, compatible opaque redeclaration
enum class E7 : short;         // Error, incompatible opaque redeclaration
enum class E8 : long;          // OK, opaque declaration, long underlying type
enum class E8 : long { e_A5 }; // OK, compatible complete definition
enum class E9 { e_A6, e_B7 };  // OK, complete definition
enum class E9 { e_A6, e_B7 };  // Error, complete redefinition in same TU
enum class    { e_A, e_B };    // Error, anonymous enum classes are not allowed
```

To summarize, each classical `enum` type having an explicitly specified **underlying type** and every modern `enum class` type can be declared (e.g., for the first time in a TU) as a **complete type**:

```
enum E10 : char;  static_assert(sizeof(E10)  == 1);
enum class E11;   static_assert(sizeof(E11) == sizeof(int));

E10 a;            static_assert(sizeof a == 1);
E11 b;            static_assert(sizeof b == sizeof(int));
```

Typical usage of opaque enumerations often involves placing the **forward declaration** within a header and sequestering the complete definition within a corresponding `.cpp` (or else a second header), thereby **insulating** (at least some) clients from changes to the enumerator list (see *Use Cases — Using opaque enumerations within a header file* on page 42):

```
// mycomponent.h:
// ...
enum E9 : char;   // forward declaration of enum E9
enum class E10;   // forward declaration of enum class E10


// mycomponent.cpp:
#include <mycomponent.h>
// ...
enum E9 : char { e_A9, e_B9, e_C9 };
    // complete definition compatible with forward declaration of E9

enum class E10 { e_A10, e_B10, e_C10 };
    // complete definition compatible with forward declaration of E10
```

Note, however, that clients embedding *local declarations* directly in their code can be problematic; see *Potential Pitfalls — Redeclaring an externally defined enumeration locally* on page 53.

use-cases-opaqueenum

-within-a-header-file

Use Cases

Using opaque enumerations within a header file

Physical design involves two related but distinct notions of information *hiding*: **encapsulation** and **insulation**. An implementation detail is *encapsulated* if changing it (in a semantically compatible way) does not require clients to rework their code but might require them to recompile it.

An *insulated* implementation detail, on the other hand, can be altered compatibly *without* forcing clients even to recompile. The advantages of avoiding such **compile-time coupling** transcend merely reducing compile time. For larger codebases in which various layers are managed under different release cycles, making a change to an *insulated* detail can be done with a `.o` patch and a relink the same day, whereas an *uninsulated* change might precipitate a waterfall of releases spanning days, weeks, or even longer.

As a first example of **opaque-enumeration** usage, consider a non-*value-semantic* **mechanism** class, `Proctor`, implemented as a finite-state machine:

```
// proctor.h:
// ...
class Proctor
{
    int d_current;  // "opaque" but unconstrained int type (BAD IDEA)
    // ...

public:
    Proctor();
    // ...
};
```

Among other private members, `Proctor` has a data member, `d_current`, representing the current enumerated state of the object. We anticipate that the implementation of the underlying state machine will change regularly over time but that the `public` interface is relatively stable. We will, therefore, want to ensure that all parts of the implementation that are likely to change reside outside of the header. Hence, the complete definition of the enumeration of the states (including the enumerator list itself) is sequestered within the corresponding `.cpp` file:

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

enum State { e_STARTING, e_READY, e_RUNNING, e_DONE };
Proctor::Proctor() : d_current(e_STARTING) { /* ... */ }
// ...
```

Prior to C++11, enumerations could not be **forward declared**. To avoid exposing (in a header file) enumerators that were used only privately (in the `.cpp` file), a completely unconstrained `int` would be used as a data member to represent the state. With the advent of modern C++, we now have better options. First, we might consider adding an explicit underlying type to the enumeration in the `.cpp` file:

```
// proctor.cpp:
```

```
#include <proctor.h>

enum State : int { e_STARTING, e_READY, e_RUNNING, e_DONE };
Proctor::Proctor() : d_current(e_STARTING) { /* ... */ }
// ...
```

Now that the **component-local** `enum` has an explicit underlying type, we can **forward declare** it in the header file. The existence of `proctor.cpp`, which includes `proctor.h`, makes this declaration a *forward declaration* and not just an *opaque declaration*. Compilation of `proctor.cpp` guarantees that the declaration and definition are compatible. Having this *forward declaration* improves (somewhat) our type safety:

```
// proctor.h:
// ...
enum State : int;  // opaque declaration of enumeration (new in C++11)

class Proctor
{
    State d_current;  // opaque classical enumerated type (BETTER IDEA)
    // ...

public:
    Proctor();
    // ...
};
```

But we can do even better. First we will want to nest the enumerated `State` type within the private section of the proctor to avoid needless namespace pollution. What's more, because the numerical values of the enumerators are not relevant, we can more closely model our intent by nesting a more strongly typed `enum class` instead:

```
// proctor.h:
// ...
class Proctor
{
    enum class State;  // forward (nested) declaration of type-safe enumeration
    State d_current;   // opaque (modern) enumerated data type (BEST IDEA)
    // ...

public:
    Proctor();
    // ...
};
```

We would then declare the nested `enum class` accordingly in the `.cpp` file:

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

enum class Proctor::State { e_STARTING, e_READY, e_RUNNING, e_DONE };
Proctor::Proctor() : d_current(State::e_STARTING) { /* ... */ }
// ...
```

Finally, notice that in this example we first *forward declared* the nested `enum class` type within class scope and then in a separate statement defined a data member of the opaque enumerated type. We needed to do this in two statements because simultaneously *opaquely declaring* either a classic `enum` having an explicit underlying type or an `enum class` and also defining an object of that type in a single statement is not possible:

```
enum E1 : int e1;  // Error, syntax not supported
enum class E2 e2;  // Error,    "    "     "
```

Fully defining an enumeration and simultaneously defining an object of the type in one stroke is, however, possible:

```
enum E3 : int { e_A, e_B } e3;  // OK, full type definition + object definition
enum class E4 { e_A, e_B } e4;  // OK,  "    "       "      "   "        "
```

Providing such a full definition, however, would have run counter to our intention to **insulate** the enumerator list of `Proctor::State` from clients `#include`ing the header file defining `Proctor`.

m-the-enumerator-list

Cookie: Insulating all external clients from the enumerator list

A commonly recurring **design pattern**, commonly known as the "Memento pattern,"[1] manifests when a facility providing a service, often in a multi-client environment, hands off a packet of opaque information — a.k.a. a **cookie** — to a client to hold and then present back to the facility to enable resuming operations where processing left off. Since the information within the cookie will not be used substantively by the client, any unnecessary compile-time coupling of clients with the implementation of that cookie serves only to impede fluid maintainability of the facility issuing the cookie. With respect to not just *encapsulating* but *insulating* pure implementation details that are held but not used substantively by clients, we offer this Memento pattern as a possible use case for **opaque enumerations**.

Event-driven programming,[2] historically implemented using **callback functions**, introduces a style of programming that is decidedly different from that to which we might have become accustomed. In this programming paradigm, a higher-level agent (e.g., `main`) would begin by instantiating an `Engine` that will be responsible for monitoring for events and invoking provided callbacks when appropriate. Classically, clients might have registered a function pointer and a corresponding pointer to a client-defined piece of identifying data, but here we will make use of a C++11 Standard Library type, `std::function`, which can encapsulate arbitrary callable function objects and their associated state. This callback will be provided one object to represent the event that just happened and another object that can be used opaquely to reregister interest in the same event again.

This opaque cookie and passing around of the client state might seem like an unnecessary step, but often the event management involved in software of this sort is wrapping the most often executed code in very busy systems, and performance of each basic operation is therefore very important. To maximize performance, every potential branch or lookup in an internal data structure must be minimized, and allowing clients to pass back the internal state of the engine when reregistering can greatly reduce the engine's work to

[1] **?**, Chapter 5, section "Memento," pp. 283–???

[2] See also **?**, Chapter 5, section "Observer," pp. 293–???

continue a client's processing of events without tearing down and rebuilding all client state each time an event happens. More importantly, event managers such as this often become highly concurrent to take advantage of modern hardware, so performant manipulation of their own data structures and well-defined lifetime of the objects they interact with become paramount. This makes the simple guarantee of, "If you don't reregister, then the engine will clean everything up; if you do, then the callback function will continue its lifetime," a very tractable paradigm to follow.

```
// callbackengine.h:
#include <deque>          // std::deque
#include <functional>     // std::function

class EventData;       // information that clients will need to process an event
class CallbackEngine;  // the driver for processing and delivering events

class CallbackData
{
    // This class represents a handle to the shared state associating a
    // callback function object with a CallbackEngine.

public:
    typedef std::function<void(const EventData&, CallbackEngine*,
        CallbackData)> Callback;
        // alias for a function object returning void and taking, as arguments,
        // the event data to be consumed by the client, the address of the
        // CallbackEngine object that supplied the event data, and the
        // callback data that can be used to reregister the client, should the
        // client choose to show continued interest in future instances of the
        // same event

    enum class State;  // GOOD IDEA
        // nested forward declaration of insulated enumeration, enabling
        // changes to the enumerator list without forcing clients to recompile

private:
    // ... (a smart pointer to an opaque object containing the state and the
    //      callback to invoke)

public:
    CallbackData(const Callback &cb, State init);

    // ... (constructors, other manipulators and accessors, etc.)

    State getState() const;
        // Return the current state of this callback.

    void setState(State state) const;
        // Set the current state to the specified state.
```

```
    Callback& getCallback() const;
        // Return the callback function object specified at construction.
};

class CallbackEngine
{
private:
    // ... (other, stable private data members implementing this object)

    bool d_running;  // active state

    std::deque<CallbackData> d_pendingCallbacks;
        // The collection of clients currently registered for interest, or having
        // callbacks delivered, with this CallbackEngine.
        //
        // Reregistering or skipping reregistering when
        // called back will lead to updating internal data structures based on
        // the current value of this State.

public:
    // ...   (other public member functions, e.g., creators, manipulators)

    void registerInterest(CallbackData::Callback cb);
        // Register (e.g., from main) a new client with this manager object.

    void reregisterInterest(const CallbackData& callback);
        // Reregister (e.g., from a client) the specified callback with this
        // manager object, providing the state contained in the CallbackData
        // to enable resumption from the same state as processing left off.

    void run();
        // Start this object's event loop.

    // ...  (other public member functions, e.g., manipulators, accessors)
};
```

A client would, in `main`, create an instance of this `CallbackEngine`, define the appropriate functions to be invoked when events happen, register interest, and then let the engine `run`:

```
// myapplication.cpp:
// ...
#include <callbackengine.h>

static void myCallback(const EventData&    event,
                       CallbackEngine*     engine,
                       const CallbackData& cookie);
    // Process the specified event, and then potentially reregister the
    // specified cookie for interest in the same data.

int main()
```

```
{
    CallbackEngine e;  // Create a configurable callback engine object.

    //...     (Configure the callback engine, e, as appropriate.)

    e.registerInterest(&myCallback);  // Even a stateless function pointer can
                                      // be used with std::function.

    // ...create and register other clients for interest...

    e.run();    // Cede control to e's event loop until complete.

    return 0;
}
```

The implementation of `myCallback`, in the example below, is then free to reregister interest in the same event, save the cookie elsewhere to reregister at a later time, or complete its task and let the `CallbackEngine` take care of properly cleaning up all now unnecessary resources:

```
void myCallback(const EventData&     event,
                CallbackEngine      *engine,
                const CallbackData&  cookie)
{
    int status = EventProcessor::processEvent(event);

    if (status > 0)  // status is non-zero; continue interest in event
    {
        engine->reregisterInterest(cookie);
    }
    else if (status < 0)  // Negative status indicates EventProcessor wants
                          // to reregister later.
    {
        EventProcessor::storeCallback(engine,cookie);
                          // Call reregisterInterest later.
    }

    // Return flow of control to the CallbackEngine that invoked this
    // callback.  If status was zero, then this callback should be cleaned
    // up properly with minimal fuss and no leaks.
}
```

What makes use of the **opaque enumeration** here especially apt is that the internal data structures maintained by the `CallbackEngine` might be very subtly interrelated, and any knowledge of a client's relationship to those data structures that can be maintained through callbacks is going to reduce the amount of lookups and synchronization that would be needed to correctly reregister a client without that information. The otherwise wide contract on `reregisterInterest` means that clients have no need themselves to directly know anything about the actual values of the `State` they might be in. More notably, a component like this is likely to be very heavily reused across a large codebase, and being able to maintain it while minimizing the need for clients to recompile can be a huge boon to deployment times.

To see what is involved, we can consider the business end of the `CallbackEngine` implementation and an outline of what a single-threaded implementation might involve:

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

enum class CallbackData::State
{
    // Full (local) definition of the enumerated states for the callback engine.
    e_INITIAL,
    e_LISTENING,
    e_READY,
    e_PROCESSING,
    e_REREGISTERED,
    e_FREED
};

void CallbackEngine::registerInterest(CallbackData::Callback cb)
{
    // Create a CallbackData instance with a state of e_INITIAL and
    // insert it into the set of active clients.
    d_pendingCallbacks.push_back(CallbackData(cb, CallbackData::State::e_INITIAL));
}

void CallbackEngine::run()
{
    // Update all client states to e_LISTENING based on the events in which
    // they have interest.

    d_running = true;
    while (d_running)
    {
        // Poll the operating system API waiting for an event to be ready.
        EventData event = getNextEvent();

        // Go through the elements of d_pendingCallbacks to deliver this
        // event to each of them.
        std::deque<CallbackData> callbacks = std::move(d_pendingCallbacks);

        // Loop once over the callbacks we are about to notify to update their
        // state so that we know they are now in a different container.
        for (CallbackData& callback : callbacks)
        {
            callback.setState(CallbackData::State::e_READY);
        }

        while (!callbacks.empty())
        {
            CallbackData callback = callbacks.front();
            callbacks.pop_front();
```

```
            // Mark the callback as processing and invoke it.
            callback.setState(CallbackData::State::e_PROCESSING);

            callback.getCallback()(event, this, callback);

            // Clean up based on the new State.
            if (callback.getState() == CallbackData::State::e_REREGISTERED)
            {
                // Put the callback on the queue to get events again.
                d_pendingCallbacks.push_back(callback);
            }
            else
            {
                // The callback can be released, freeing resources.
                callback.setState(CallbackData::State::e_FREED);
            }
        }
    }
}

void CallbackEngine::reregisterInterest(const CallbackData& callback)
{
    if (callback.getState() == CallbackData::State::e_PROCESSING)
    {
        // This is being called reentrantly from run(); simply update state.
        callback.setState(CallbackData::State::e_REREGISTERED);
    }
    else if (callback.getState() == CallbackData::State::e_READY)
    {
        // This callback is in the deque of callbacks currently having events
        // delivered to it; do nothing and leave it there.
    }
    else
    {
        // This callback was saved; set it to the proper state and put it in
        // the queue of callbacks.
        if (d_running)
        {
            callback.setState(CallbackData::State::e_LISTENING);
        }
        else
        {
            callback.setState(CallbackData::State::e_INITIAL);
        }

        d_pendingCallbacks.push_back(callback);
    }
}
```

Note how the definition of `CallbackData::State` is visible and needed only in this implementation file. Also, consider that the set of states might grow or shrink as this `CallbackEngine` is optimized and extended, and clients can still pass around the object containing that state in a type-safe manner while remaining insulated from this definition.

Prior to C++11, we could not have *forward declared* this enumeration, and so would have had to represent it in a *type-unsafe* way — e.g., as an `int`. Thanks to the modern `enum class` (see Section 2.1."`enum class`" on page 22), however, we can conveniently **forward declare** it as a nested type and then, separately, fully define it inside the `.cpp` implementing other noninline member functions of the `CallbackEngine` class. In this way, we are able to *insulate* changes to the enumerator list along with any other aspects of the implementation defined outside of the `.h` file without forcing any client applications to recompile. Finally, the basic design of the hypothetical `CallbackEngine` in the previous code example could have been used for any number of useful components: a parser or tokenizer, a workflow engine, or even a more generalized event loop.

m-the-enumerator-list

Dual-Access: Insulating some external clients from the enumerator list

In previous use cases, the goal has been to **insulate** *all* external clients from the enumerators of an enumeration that is visible (but not necessarily programmatically reachable) in the defining component's header. Consider the situation in which a **component** (`.h`/`.cpp` pair) itself defines an enumeration that will be used by various clients within a single program, some of which will need access to the enumerators.

When an `enum class` or a classic `enum` having an explicitly specified underlying type (see Section 2.1."Underlying Type '11" on page 57) is specified in a header for direct programmatic use, external clients are at liberty to unilaterally redeclare it *opaquely*, i.e., without its enumerator list. A compelling motivation for doing so would be for a client who doesn't make direct use of the enumerators to **insulate** itself and/or its clients from having to recompile whenever the enumerator list changes.

Embedding any such **local declaration** in client code, however, would be highly problematic: If the underlying type of the declaration (in one translation unit) were somehow to become inconsistent with that of the definition (in some other translation unit), any program incorporating both translation units would immediately become silently **ill-formed, no diagnostic required (IFNDR)**; see *Potential Pitfalls — Redeclaring an externally defined enumeration locally* on page 53. Unless a separate "forwarding" header file is provided along with (and ideally included by) the header defining the full enumeration, any client opting to exploit this opacity feature of an enumerated type will have no alternative but to redeclare the enumeration locally; see *Potential Pitfalls — Inciting local enumeration declarations: an attractive nuisance* on page 54.

For example, consider an `enum class`, `Event`, intended for public use by external clients:

```
// event.h:
// ...
enum class Event : char { /*... changes frequently ...*/ };
// ...              ^^^^
```

Now imagine some client header file, `badclient.h`, that makes use of the `Event` enumeration and chooses to avoid compile-time coupling itself to the enumerator list by embedding, for

whatever reason, a **local declaration** of `Event` instead:

```
// badclient.h:
// ...
enum class Event : char;  // BAD IDEA: local external declaration
// ...
struct BadObject
{
    Event d_currentEvent;  // object of locally declare enumeration
    // ...
};
// ...
```

Imagine now that the number of events that can fit in a `char` is exceeded and we decide to change the definition to have an underlying type of `short`:

```
// event.h:
// ...
enum class Event : short { /*... changes frequently ...*/ };
// ...              ^^^^^
```

Client code, such as in `badclient.h`, that fails to include the `event.h` header will have no automatic way of knowing that it needs to change, and recompiling the code for all cases where `event.h` isn't also included in the translation unit will not fix the problem. Unless every such client is somehow updated manually, a newly linked program comprising them will be **IFNDR** with the likely consequence of either a crash or (worse) when the program runs and misbehaves. When providing a programmatically accessible definition of an enumerated type in a header where the **underlying type** is specified either explicitly or implicitly, we can give external clients a *safe* alternative to local declaration by also providing an auxiliary header containing *just* the corresponding opaque declaration:

```
// event.fwd.h:
// ...
enum class Event : char;
// ...
```

Here we have chosen to treat the forwarding header file as part of the same event component as the principal header but with an injected descriptive suffix field, `.fwd`; this approach, as opposed to, say, `file_fwd.h`, `filefwd.h`, or `file.hh`, was chosen so as not to (1) encroach on a disciplined, component-naming scheme involving reserved use of underscores[3] or (2) confuse tools and scripts that expect header-file names to end with a `.h` suffix.

In general, having a forwarding header always included in its corresponding full header facilitates situations such as default template arguments where the declaration can appear at most once in any given translation unit; the only drawback being that the comparatively small forwarding header file must now also be opened and parsed if the full header file is included in a given translation unit. To ensure consistency, we thus `#include` this forwarding header in the original header defining the full enumeration:

```
// event.h:
```

[3] **?**, section 2.4, pp. 297–333

```
// ...                  // Ensure opaque declaration (included here) is
#include <event.fwd.h>  // consistent with complete definition (below).
// ...
enum class Event : char { /*... changes frequently ...*/ };
// ...
```

In this way, every translation unit that includes the definition will serve to ensure that the forward declaration and definition match; hence, clients can incorporate safely only the presumably more stable forwarding header:

```
// goodclient.h:
// ...
#include <event.fwd.h>  // GOOD IDEA: consistent opaque declaration
// ...
class Client
{
    Event d_currentEvent;
    // ...
};
```

Note that we have consistently employed angle brackets exclusively for all include directives used throughout this book to maximize flexibility of deployment presuming a regimen for unique naming.[4]

To illustrate real-world practical use of the opaque-enumerations feature, consider the various components[5] that might depend on[6] an `Event` enumeration such as that above:

- **`message`** — The component provides a *value-semantic* `Message` class consisting of just raw data,[7] including an `Event` field representing the type of event. This component never makes direct use of any enumeration values and thus needs to include only `event.fwd.h` and the corresponding opaque *forward* declaration of the `Event` enumeration.

- **`sender` and `receiver`** — These are a pair of components that, respectively, create and consume `Message` objects. To populate a `Message` object, a `Sender` will need to provide a valid value for the `Event` member. Similarly, to process a `Message`, a `Receiver` will need to understand the potential individual enumerated values for the `Event` field. Both of these components will include the primary `event.h` header and thus have the complete definition of `Event` available to them.

- **`messenger`** — The final component, a general engine capable of being handed `Message` objects by a `Sender` and then delivering those objects in an appropriate fashion to a `Receiver`, needs a complete and usable definition of `Message` objects — possibly copying them or storing them in containers before delivery — but has no need for

[4] See **?**, section 1.5.1, pp. 201–203.
[5] See **?**, sections 1.6 and 1.11, pp. 209–216 and pp. 256–259, respectively.
[6] **?**, section 1.9?, pp. 237–243 JOHN: Please consult the book and correct. Section 1.9 is pp 243–251. Section 1.8 is pp. 237–243.
[7] We sometimes refer to data that is meaningful only in the context of a higher-level entity as **dumb data**; see **?**, section 3.5.5, pp. 629–633.

understanding the possible values of the `Event` member within those `Message` objects. This component can participate fully and correctly in the larger system while being completely *insulated* from the enumeration values of the `Event` enumeration.

tl;dr: By factoring out the `Event` enumeration into its own separate component and providing two distinct but compatible headers, one containing the opaque declaration and the other (which includes the first) providing the complete definition, we enable having different components choose not to compile-time couple themselves with the enumerator list without forcing them to (*unsafely*) redeclare the enumeration locally.

ial-pitfalls-opaqueenum

ned-enumeration-locally

Potential Pitfalls

Redeclaring an externally defined enumeration locally

An opaque enumeration declaration enables the use of that enumeration without granting visibility to its enumerators, reducing physical coupling between components. Unlike a **forward class declaration**, an opaque enumeration declaration produces a complete type, sufficient for substantive use (e.g., via the linker):

```
    std::uint8_t

// client.cpp:
enum Event : std::uint8_t;
Event e;  // OK, Event is a complete type.
```

The *underlying type* specified in an opaque `enum` declaration must exactly match the full definition; otherwise a program incorporating both is automatically **IFNDR**. Updating an `enum`'s underlying type to accommodate additional values can lead to latent defects when these changes are not propagated to all local declarations:

```
    std::uint16_t

// library.h:
enum Event : std::uint16_t { /* now more than 256 events */ };
```

Consistency of a local opaque `enum` declaration's underlying type with that of its complete definition in a separate translation unit cannot be enforced by the compiler, potentially leading to a program that is **IFNDR**. In the `client.cpp` example shown above, if the opaque declaration in `client.cpp` is not somehow updated to reflect the changes in `event.h`, the program will compile, link, and run, but its behavior has silently become undefined. The only robust solution to this problem is for `library.h` to provide two separate header files; see *Inciting local enumeration declarations: an attractive nuisance* on page 54.

The problem with local declarations is by no means limited to opaque enumerations. Embedding a local declaration for any object whose use might be associated with its definition in a separate translation unit via just the linker invites instability:

```
// main.cpp:                          // library.cpp:
extern int x;  // BAD IDEA!           int x;
// ...                                // ...
```

The definition of object `x` (in the code snippets above) resides in the `.cpp` file of the library component while a supposed declaration of `x` is embedded in the file defining `main`. Should

the type of just the definition of `x` change, both translation units will continue to compile but, when linked, the resulting program will be **IFNDR**:

```
// main.cpp:
extern int x;  // ILL-FORMED PROGRAM
// ...
```

```
// library.cpp:
double x;
// ...
```

To ensure consistency across translation units, the time-honored tradition is to place, in a header file managed by the supplier, a declaration of each external-linkage entity intended for use outside of the translation unit in which it is defined; that header is then included by both the supplier and each consumer:

```
// main.cpp:
#include <library.h>
// ...
```

```
// library.h:
// ...
extern int x;
// ...
```

```
// library.cpp:
#include <library.h>
int x;
// ...
```

In this way, any change to the definition of `x` in `library.cpp` — the supplier — will trigger a compilation error when `library.cpp` is recompiled, thereby forcing a corresponding change to the declaration in `library.h`. When that happens, typical build tools will take note of the change in the header file's timestamp relative to that of the `.o` file corresponding to `main.cpp` — the consumer — and indicate that it too needs to be recompiled. Problem solved.

The maintainability pitfall associated with opaque enumerations, however, is qualitatively more severe than for other external-linkage types, such as a global `int`: (1) the full definition for the enumeration type itself needs to reside in a header for *any* external client to make use of its individual enumerators and (2) typical components consist of just a `.h`/`.cpp` pair, i.e., exactly one `.h` file and usually just one `.cpp` file.[8]

Exposing, within a library header file, an opaquely declarable enumeration that is programmatically accessible by external clients without providing some maintainable way for those clients to keep their elided declarations in sync with the full definition introduces what we are calling an *attractive nuisance*: the client is forced to choose between (a) introducing the added risk and maintenance burden of having to manually maintain consistency between the underlying types for all its separate opaque uses and the one full definition or (b) forgo use of this opaque-enumeration feature entirely, forcing gratuitous compile-time coupling with the unused and perhaps unstable enumerators. At even moderate scale, excessive compile-time coupling can adversely affect projects in ways that are far more insidious than just increased compile times during development — e.g., any emergency changes that might need to occur and be deployed quickly to production without forcing all clients to recompile and then be retested and then, eventually, be rereleased.[9]

n-attractive-nuisance

Inciting local enumeration declarations: an attractive nuisance

Whenever we, as library component authors, provide the complete definition of an `enum class` or a classic enumeration with an explicitly specified underlying type and fail to provide a corresponding header having just the opaque declaration, we confront our clients with the

[8] **?**, sections 2.2.11–2.2.13, pp. 280–281

[9] For a complete real-world example of how compile-time coupling can delay a "hot fix" by weeks, not just hours, see **?**, section 3.10.5, pp. 783–789.

unenviable conundrum of whether to needlessly compile-time couple themselves and/or their clients with the details of the enumerator list or to make the dubious choice to unilaterally redeclare that enumeration locally.

The problems associated with local declarations of data whose types are maintained in separate translation units is not limited to enumerations; see *Redeclaring an externally defined enumeration locally* on page 53. The maintainability pitfall associated with **opaque enumerations**, however, is qualitatively more severe than for other external-linkage types, such as a global `int`, in that the ability to elide the enumerators amounts to an *attractive nuisance* wherein a client — wanting to do so and having access to only a single header containing the unelided definition (i.e., comprising the enumeration name, underlying integral type, and enumerator list) — might be persuaded into providing an elided copy of the `enum`'s definition (i.e., one omitting just the enumerators) locally!

Ensuring that library components that define enumerations (e.g., `enum class Event`) whose enumerators can be elided also consistently provide a second forwarding header file containing the opaque declaration of each such enumeration (i.e., enumeration name and underlying integral type only) would be one generally applicable way to sidestep this often surprisingly insidious maintenance burden; see *Dual-Access: Insulating some external clients from the enumerator list* on page 50. Note that the attractive nuisance potentially exists even when the primary intent of the component is not to make the enumeration generally available.[10]

Annoyances

annoyances

Opaque enumerations are not completely type safe

ot-completely-type-safe

Making an enumeration opaque does not stop it from being used to create an object that is initialized opaquely to a zero value and then subsequently used (e.g., in a function call):

```
enum Bozo : int;  // forward declaration of enumeration Bozo
void f(Bozo);     // forward declaration of function f

void g()
{
    Bozo clown{};
    f(clown);      // OK, who knows if zero is a valid value?!
}
```

Though creating a zero-valued enumeration variable by default is not new, allowing one to be created without even knowing what enumerated values are valid is arguably dubious.

See Also

see-also

- "Underlying Type '11" (Section 2.1, p. 57) ♦ Discusses the underlying integral representation for enumeration variables and their values.
- "`enum class`" (Section 2.1, p. 22) ♦ Introduces an implicitly scoped, more strongly typed enumeration.

[10] ?

Further Reading

further-reading

- For more on internal versus external linkage, see **?**, section 1.3.1, pp. 154–159.
- For more on the use of header files to ensure consistency across translation units, see **?**, section 1.4, pp. 190–201, especially Figure 1-35, p. 197.
- For more on the use of `#include` directives and `#include` guards, see **?**, section 1.5, pp. 201–209.
- For a complete delineation of inherent properties that belong to every well-conceived `.h`/`.cpp` pair, see **?**, sections 1.6 and 1.11, pp. 219–216 and 256–259, respectively.
- For an introduction to physical dependency, see **?**, section 1.8, pp. 237–243.
- For a suggestion on how to achieve unique naming of files, see **?**, section 2.4, pp. 297–333.
- For a thorough treatment of **architectural insulation**, see **?**, sections 3.10–3.11, pp. 773–835.

eration-underlying-type

Explicit Enumeration Underlying Type

2.3— 0— 0

The underlying type of an enumeration is the fundamental **integral type** used to represent its enumerated values, which can be specified explicitly in C++11.

description

Description

Every enumeration employs an integral type, known as its **underlying type**, to represent its compile-time-enumerated values. By default, the **underlying type** of a C++98 `enum` is chosen by the implementation to be large enough to represent all of the values in an enumeration and is allowed to exceed the size of an `int` *only* if there are enumerators having values that cannot be represented as an `int` or `unsigned int`:

```
enum RGB { e_RED, e_GREEN, e_BLUE };                    // OK, fits any char

enum Port { e_LEFT = -81, e_RIGHT = -82 };              // OK, fits signed char

enum Mask { e_LOW = 32767, e_HIGH = 65535 };            // OK, fits unsigned short

enum Big { e_32 = 1<<32 };                              // Error, too big for int

enum Err { K = 1024, M = K*K, G = M*K, T = G*K };       // Error, G*K overflows int...

enum OK { K = 1<<10, M = 1<<20, G = 1<<30, T = 1l<<40 }; // OK
```

The default underlying type chosen for an `enum` is always sufficiently large to represent *all* enumerator values defined for that `enum`. If the value doesn't fit in an `int`, it will be selected deterministically as the first type able to represent all values from the sequence: `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`.

While specifying an enumeration's underlying type was impossible before C++11, the compiler could be forced to choose at least a 32-bit or 64-bit signed integral type by adding an enumerator having a sufficiently large negative value — e.g., `1 << 31` for a 32-bit and `1 << 63` for a 64-bit signed integer (assuming such is available on the target platform).

The above applies only to C++98 `enum`s; the default underlying type of an `enum class` is ubiquitously `int`, and it is not implementation defined; see Section 2.1."`enum class`" on page 22.

Note that `char` and `wchar_t`, like enumerations, are their own distinct types (as opposed to `typedef`-like aliases such as `std::uint8_t`) and have their own implementation-defined underlying integral types. With `char`, for example, the underlying type will always be either `signed char` or `unsigned char` (both of which are also distinct C++ types).[1]

erlying-type-explicitly

Specifying underlying type explicitly

As of C++11, we have the ability to specify the **integral type** that is used to represent an `enum`. This is achieved by providing the type explicitly in the `enum`'s declaration following

[1]The same is true in C++11 for `char16_t` and `char32_t` and in C++20 for `char8_t`.

the enumeration's (optional) name and preceded by a colon:

```
enum Port : unsigned char
{
    // Each enumerator of Port is represented as an unsigned char type.

    e_INPUT        =  37,  // OK, would have fit in a signed char too
    e_OUTPUT       = 142,  // OK, would not have fit in a signed char
    e_CONTROL      = 255,  // OK, barely fits in an 8-bit unsigned integer
    e_BACK_CHANNEL = 256,  // Error, doesn't fit in an 8-bit unsigned integer
};
```

If any of the values specified in the definition of the `enum` is outside the boundaries of what the provided **underlying type** is able to represent, the compiler will emit an error, but see *Potential Pitfalls: Subtleties of integral promotion* on page 60.

Use Cases

use-cases

Ensuring a compact representation where enumerator values are salient

or-values-are-salient

When the enumeration needs to have an efficient representation, e.g., when it is used as a data member of a widely replicated type, restricting the width of the underlying type to something smaller than would occur by default on the target platform might be justified.

As a concrete example, suppose that we want to enumerate the months of the year, for example, in anticipation of placing that enumeration inside a date class having an internal representation that maintains the year as a two-byte signed integer, the month as an enumeration, and the day as an 8-bit signed integer:

```
#include <cstdint>  // std::int8_t, std::int16_t

class Date
{
    std::int16_t d_year;
    Month        d_month;
    std::int8_t  d_day;

public:
    Date();
    Date(int year, Month month, int day);

    // ...

    int year() const    { return d_year; }
    Month month() const { return d_month; }
    int day() const     { return d_day; }
};
```

Within the software, the `Date` is typically constructed using the values obtained through the GUI, where the month is always selected from a drop-down menu. Management has requested that the month be supplied to the constructor as an `enum` to avoid recurring defects where the individual fields of the date are supplied in month/day/year format. New

functionality will be written to expect the month to be enumerated. Still, the date class will be used in contexts where the numerical value of the month is significant, such as in calls to legacy functions that accept the month as an integer. Moreover, iterating over a range of months is common and requires that the enumerators convert automatically to their integral **underlying type**, thus contraindicating use of the more strongly typed `enum class`:

```
enum Month  // defaulted underlying type (BAD IDEA)
{
    e_JAN = 1, e_FEB, e_MAR,    // winter
    e_APR    , e_MAY, e_JUN,    // spring
    e_JUL    , e_AUG, e_SEP,    // summer
    e_OCT    , e_NOV, e_DEC     // autumn
};

static_assert(sizeof(Month) == 4 && alignof(Month) == 4, "");
```

As it turns out, date values are used widely throughout this codebase, and the proposed `Date` type is expected to be used in large aggregates. The underlying type of the `enum` in the code snippet above is implementation-defined and could be as small as a `char` or as large as an `int` despite all the values fitting in a `char`. Hence, if this enumeration were used as a data member in the `Date` class, `sizeof(Date)` would likely balloon to 12 bytes on some relevant platforms due to **natural alignment**! (See "**??**" on page **??**.)

While reordering the data members of `Date` such that `d_year` and `d_day` were adjacent would ensure that `sizeof(Date)` would not exceed 8 bytes, a better approach is to explicitly specify the enumeration's underlying type to ensure `sizeof(Date)` is exactly the 4 bytes needed to accurately represent the value of the `Date` object. Given that the values in this enumeration fit in an 8-bit signed integer, we can specify its **underlying type** to be, e.g., `std::int8_t` or `signed char`, on every platform:

```
#include <cstdint>  // std::int8_t

enum Month : std::int8_t  // user-provided underlying type (GOOD IDEA)
{
    e_JAN = 1, e_FEB, e_MAR,    // winter
    e_APR    , e_MAY, e_JUN,    // spring
    e_JUL    , e_AUG, e_SEP,    // summer
    e_OCT    , e_NOV, e_DEC     // autumn
};

static_assert(sizeof(Month) == 1 && alignof(Month) == 1, "");
```

With this revised definition of `Month`, the size of a `Date` class is 4 bytes, which is especially valuable for large aggregates:

```
Date timeSeries[1000 * 1000];  // sizeof(timeSeries) is now 4Mb (not 12Mb)
```

tfalls-underlyingenum

rators-enumunderlying

Potential Pitfalls

External use of opaque enumerators

Providing an explicit underlying type to an `enum` enables clients to declare or redeclare it as a complete type with or without its enumerators. Unless the opaque form of its definition is exported in a header file separate from its full definition, external clients wishing to exploit the opaque version will be forced to locally declare it with its **underlying type** but without its enumerator list. If the underlying type of the full definition were to change, any program incorporating *its own* original and now inconsistent elided definition and the *new* full one would become silently ill formed, no diagnostic required (**IFNDR**). (See "Opaque `enum`s" on page 40.)

of-integral-promotion

Subtleties of integral promotion

When used in an arithmetic context, one might naturally assume that the type of a classic `enum` will first convert to its **underlying type**, which is not always the case. When used in a context that does not explicitly operate on the `enum` type itself, such as a parameter to a function that takes that enum type, **integral promotion** comes into play. For unscoped enumerations without an explicitly specified underlying type and for character types such as `wchar_t`, `char16_t`, and `char32_t`, integral promotion will directly convert the value to the first type in the list `int`, `unsigned int`, `long`, `unsigned long`, `long long`, and `unsigned long long` that is sufficiently large to represent all of the values of the underlying type. Enumerations having a fixed underlying type will, as a first step, behave as if they had decayed to their underlying type.

In most arithmetic expressions, this difference is irrelevant. Subtleties arise, however, when one relies on overload resolution for identifying the underlying type:

```
void f(signed char x);
void f(short x);
void f(int x);
void f(long x);
void f(long long x);

enum E1         { q, r, s, t, u };
enum E2 : short { v, w, x, y, z };

void test()
{
   f(E1::q);  // always calls f(int) on all platforms
   f(E2::v);  // always calls f(short) on all platforms
}
```

The overload resolution for `f` considers the type to which each *individual* enumerator can be directly integrally promoted. This conversion for `E1` can be only to `int`. For `E2`, the conversion will consider `int` *and* `short`, and `short`, being an exact match, will be selected. Note that even though both enumerations are small enough to fit into a `signed char`, that overload of `f` will never be selected.

One might want to get to the implementation-defined underlying type though, and the

Standard does provide a trait to do that: `std::underlying_type` in C++11 and the corresponding `std::underlying_type_t` alias in C++14. This trait can safely be used in a cast without risking loss of value (see "??" on page ??):

```
#include <type_traits>  // std::underlying_type

template <typename E>
typename std::underlying_type<E>::type toUnderlying(E value)
{
    return static_cast<typename std::underlying_type<E>::type>(value);
}

void h()
{
    auto e1 = toUnderlying(E1::q); // might be anywhere from signed char to int
    auto e2 = toUnderlying(E2::v); // always deduced as short
}
```

As of C++20, however, the use of a classic enumerator in a context in which it is compared to or otherwise used in a binary operation with either an enumerator of another type or a nonintegral type (i.e., a floating-point type, such as `float`, `double`, or `long double`) is deprecated, with the the possibility of being removed in C++23. Platforms might decide to warn against such uses retroactively:

```
enum { k_GRAMS_PER_OZ = 28 };  // not the best idea

double gramsFromOunces(double ounces)
{
    return ounces * k_GRAMS_PER_OZ;  // deprecated in C++20; might warn
}
```

Casting to the **underlying type** is *not* necessarily the same as direct integral promotion. In this context, we might want to change our `enum` to a `constexpr int` in the long term (see "??" on page ??):

```
constexpr int k_GRAMS_PER_OZ = 28;  // future proof
```

see-also

See Also

- "`enum class`" (Section 2.1, p. 22) ♦ Introduces a scoped, more strongly typed enumeration that can explicitly specify an underlying type.
- "Opaque `enum`s" (Section 2.1, p. 40) ♦ Offers a means to insulate individual enumerators from clients.
- "??" (Section 2.1, p. ??) ♦ Introduces an alternative way of declaring compile-time constants.

further-reading

Further Reading

- "Item 10: Prefer scoped `enum`s to unscoped `enum`s," ?

- **?**

template-instantiations

Explicit Instantiation Declarations

2.4— 0— 0

The **extern template** prefix can be used to suppress *implicit* generation of local object code for the definitions of particular specializations of class, function, or variable templates used within a translation unit, with the expectation that any suppressed object-code-level definitions will be provided elsewhere within the program by template definitions that are instantiated *explicitly*.

description

Description

Inherent in the current ecosystem for supporting template programming in C++ is the need to generate redundant definitions of fully specified function and variable templates within `.o` files. For common instantiations of popular templates, such as `std::vector`, the increased object-file size — a.k.a. **code bloat** — and potentially extended link times might become significant:

```
#include <vector>    // std::vector is a popular template.
std::vector<int> v;  // std::vector<int> is a common instantiation.

#include <string>    // std::basic_string is a popular template.
std::string s;       // std::string, an alias for std::basic_string<char>, is
                     // a common instantiation.
```

The intent of the **extern template** feature is to *suppress* the implicit generation of duplicative object code within each and every translation unit in which a fully specialized class template, such as `std::vector<int>` in the code snippet above, is used. Instead, **extern template** allows developers to choose a single translation unit in which to explicitly *generate* object code for all the definitions pertaining to that specific template specialization as explained in the next subsection, *Explicit-instantiation definition*.

nstantiation-definition

Explicit-instantiation definition

The ability to create an **explicit-instantiation definition** has been available since C++98.[1] The requisite syntax is to place the keyword **template** in front of the name of the fully specialized class template, function template, or — in C++14 — variable template (see Section 1.2."**??**" on page **??**):

```
#include <vector>  // std::vector (general template)

template class std::vector<int>;
    // Deposit all definitions for this specialization into the .o for this
    // translation unit.
```

[1] The C++03 Standard term for the syntax used to create an **explicit-instantiation *definition***, though rarely used, was **explicit-instantiation *directive***. The term **explicit-instantiation directive** was clarified in C++11 and can now also refer to syntax that is used to create a *declaration* — i.e., **explicit-instantiation *declaration***.

This **explicit-instantiation directive** compels the compiler to instantiate *all* functions defined by the named `std::vector` class template having the specified `int` template argument; any collateral object code resulting from these instantiations will be deposited in the resulting `.o` file for the current translation unit. Importantly, even functions that are never used are still specialized, so this might not be the correct solution for many classes; see *Potential Pitfalls — Accidentally making matters worse* on page 81.

antiation-declaration

Explicit-instantiation declaration

C++11 introduced the **explicit-instantiation *declaration***, complement to the **explicit-instantiation *definition***. The newly provided syntax allows us to place **`extern template`** in front of the declaration of the explicit specialization of a class template, a function template, or a variable template:

```
#include <vector>  // std::vector (general template)

extern template class std::vector<int>;
    // Suppress depositing of any object code for std::vector<int> into the
    // .o file for this translation unit.
```

The use of the modern **`extern template`** syntax above instructs the compiler to specifically *not* deposit any object code for the named specialization in the current translation unit and instead to rely on some other translation unit to provide any missing object-level definitions that might be needed at link time; see *Annoyances — No good place to put definitions for unrelated classes* on page 81.

Note, however, that declaring an explicit instantiation to be an **`extern template`** *in no way* affects the ability of the compiler to instantiate and to inline visible function-definition bodies for that template specialization in the translation unit:

```
// client.cpp:
#include <vector>  // std::vector (general template)

extern template class std::vector<int>;  // specialization for int elements

void client(std::vector<int>& inOut)  // fully specialized instance of a vector
{
    if (inOut.size())          // This invocation of size can inline.
    {
        int value = inOut[0];  // This invocation of operator[] can inline.
    }
}
```

In the example above, the two tiny member functions of `vector`, namely `size` and **`operator[]`**, will typically be substituted inline — in precisely the same way they would have been had the **`extern template`** declaration been omitted. The *only* purpose of an **`extern template`** declaration is to suppress object-code generation for this particular template instantiation for the current translation unit.

Finally, note that the use of **explicit-instantiation *directives*** have absolutely no affect on the logical meaning of a well-formed program; in particular, when applied to specializations of function templates, they have no affect whatsoever on overload resolution:

```
template <typename T> bool f(T v) {/*...*/}  // general template definition

extern template bool f(char c);  // specialization of f for char
extern template bool f(int v);   // specialization of f for int

char     c;
short    s;
int      i;
unsigned u;

bool bc = f(c);  // exact match: Object code is suppressed locally.
bool bs = f(s);  // not exact match: Object code is generated locally.
bool bi = f(i);  // exact match: Object code is suppressed locally.
bool bu = f(u);  // not exact match: Object code is generated locally.
```

As the example above illustrates, overload resolution and template parameter deduction occur independently of any **explicit-instantiation declarations**. Only *after* the template to be instantiated is determined does the `extern template` syntax take effect; see also *Potential Pitfalls — Corresponding explicit-instantiation declarations and definitions* on page 79.

te-illustrative-example

A more complete illustrative example

So far, we have seen the use of **explicit-instantiation declarations** and **explicit-instantiation definitions** applied to only a (standard) *class* template, `std::vector`. The same syntax shown in the previous code snippet applies also to full specializations of individual function templates and variable templates.

As a more comprehensive, albeit largely pedagogical example, consider the overly simplistic `my::Vector` class template along with other related templates defined within a header file `my_vector.h`:

```
// my_vector.h:
#ifndef INCLUDED_MY_VECTOR  // internal include guard
#define INCLUDED_MY_VECTOR

#include <cstddef>  // std::size_t
#include <utility>  // std::swap

namespace my  // namespace for all entities defined within this component
{

template <typename T>
class Vector
{
    static std::size_t s_count; // track number of objects constructed
    T* d_data_p;                // pointer to dynamically allocated memory
    std::size_t d_length;       // current number of elements in the vector
    std::size_t d_capacity;     // number of elements currently allocated

public:
    // ...
```

```
    std::size_t length() const { return d_length; }
        // return the number of elements

    // ...
};

// ...          Any partial or full specialization definitions            ...
// ...          of the class template Vector go here.                     ...

template <typename T>
void swap(Vector<T> &lhs, Vector<T> &rhs) { return std::swap(lhs, rhs); }
    // free function that operates on objects of type my::Vector via ADL

// ...            Any [full] specialization definitions                   ...
// ...            of free function swap would go here.                    ...

template <typename T>
const std::size_t vectorSize = sizeof(Vector<T>);  // C++14 variable template
    // This nonmodifiable static variable holds the size of a my::Vector<T>.

// ...           Any [full] specialization definitions                    ...
// ...           of variable vectorSize would go here.                    ...

template <typename T>
std::size_t Vector<T>::s_count = 0;
    // definition of static counter in general template

// ... We may opt to add explicit-instantiation declarations here;
//     see the next code example.

}  // close my namespace

#endif  // close internal include guard
```

In the `my_vector` component in the code snippet above, we have defined the following, in the `my` namespace:

1. a `class` template, `Vector`, parameterized on element type
2. a free-function template, `swap`, that operates on objects of corresponding specialized `Vector` type
3. a `const` C++14 variable template, `vectorSize`, that represents the number of bytes in the **footprint** of an object of the corresponding specialized `Vector` type

Any use of these templates by a client might and typically will trigger the depositing of equivalent definitions as object code in the client translation unit's resulting `.o` file, irrespective of whether the definition being used winds up getting inlined.

To eliminate object code for specializations of entities in the `my_vector` component, we must first decide where the unique definitions will go; see *Annoyances — No good place*

to put definitions for unrelated classes on page 81. In this specific case, however, we own the component that requires specialization, and the specialization is for a ubiquitous built-in type; hence, the natural place to generate the specialized definitions is in a `.cpp` file corresponding to the component's header:

```
// my_vector.cpp:
#include <my_vector.h>  // We always include the component's own header first.
    // By including this header file, we have introduced the general template
    // definitions for each of the explicit-instantiation declarations below.

namespace my  // namespace for all entities defined within this component
{

template class Vector<int>;
    // Generate object code for all nontemplate member-function and static
    // member-variable definitions of template my::Vector having int elements.

template std::size_t Vector<double>::length() const;  // BAD IDEA
    // In addition, we could generate object code for just a particular member
    // function definition of my::Vector (e.g., length) for some other
    // parameter type (e.g., double), which is shown here for pedagogy only.

template void swap(Vector<int>& lhs, Vector<int>& rhs);
    // Generate object code for the full specialization of the swap free-
    // function template that operates on objects of type my::Vector<int>.

template const std::size_t vectorSize<int>;  // C++14 variable template
    // Generate the object-code-level definition for the specialization of the
    // C++14 variable template instantiated for built-in type int.

//template std::size_t Vector<int>::s_count = 0;
    // Generate the object-code-level definition for the specialization of the
    // static member variable of Vector instantiated for built-in type int.

}; // close my namespace
```

Each of the constructs introduced by the keyword `template` within the `my` namespace in the code snippet above represents a separate **explicit-instantiation definition**. These constructs instruct the compiler to generate object-level definitions for general templates declared in `my_vector.h` specialized on the built-in type `int`.

Having installed the necessary **explicit-instantiation definitions** in the component's `my_vector.cpp` file, we must now go back to its `my_vector.h` file and, without altering any of the previously existing lines of code, *add* the corresponding **explicit-instantiation declarations** to suppress redundant local code generation:

```
// my_vector.h:
#ifndef INCLUDED_MY_VECTOR  // internal include guard
#define INCLUDED_MY_VECTOR

namespace my  // namespace for all entities defined within this component
```

```
{

// ...
// ...  everything that was in the original my namespace
// ...

extern template class Vector<int>;
    // Suppress object code for this class template specialized for int.

extern template std::size_t Vector<double>::size() const;  // BAD IDEA
    // Suppress object code for this member, only specialized for double.

extern template void swap(Vector<int>& lhs, Vector<int>& rhs);
    // Suppress object code for this free function specialized for int.

extern template std::size_t vectorSize<int>;  // C++14
    // Suppress object code for this variable template specialized for int.

extern template std::size_t Vector<int>::s_count;
    // Suppress object code for this static member definition w.r.t. int.

}  // close my namespace

#endif  // close internal include guard
```

Each of the constructs that begin with `extern template` in the example above are **explicit-instantiation declarations**, which serve only to suppress the generation of any object code emitted to the `.o` file of the current translation unit in which such specializations are used. These added `extern template` declarations must appear in the `my_header.h` source file *after* the declaration of the corresponding general template and, importantly, before whatever relevant definitions are ever used.

t-on-various-.o-files

The effect on various `.o` files

To illustrate the effect of **explicit-instantiation declarations** and **explicit-instantiation definitions** on the contents of object and executable files, we'll use a simple `lib_interval` library **component** consisting of a header file, `lib_interval.h`, and an implementation file, `lib_interval.cpp`. The latter, apart from including its corresponding header, is effectively empty:

```
// lib_interval.h:
#ifndef INCLUDED_LIB_INTERVAL  // internal include guard
#define INCLUDED_LIB_INTERVAL

namespace lib  // namespace for all entities defined within this component
{

template <typename T>  // elided sketch of a class template
class Interval
{
```

```
    T d_low;   // interval's low value
    T d_high;  // interval's high value

public:
    explicit Interval(const T& p) : d_low(p), d_high(p) { }
        // Construct an empty interval.

    Interval(const T& low, const T& high) : d_low(low), d_high(high) { }
        // Construct an interval having the specified boundary values.

    const T& low() const { return d_low; }
        // Return this interval's low value.

    const T& high() const { return d_high; }
        // Return this interval's high value.

    int length() const { return d_high - d_low; }
        // Return this interval's length.

    // ...
};

template <typename T>                    // elided sketch of a function template
bool intersect(const Interval<T>& i1, const Interval<T>& i2)
    // Determine whether the specified intervals intersect ...
{
    bool result = false;  // nonintersecting until proven otherwise
    // ...
    return result;
}

}  // close lib namespace

#endif  // INCLUDED_LIB_INTERVAL


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

This library component defines, in the namespace `lib`, a heavily elided sketch of an implementation of (1) a class template, `Interval`, and (2) a function template, `intersect`, the only practical purpose of which is to provide some sample template source code to compile.

Let's also consider a trivial application that uses this library **component**:

```
// app.cpp:
#include <lib_interval.h>  // Include the library component's header file.

int main(int argv, const char** argc)
{
    lib::Interval<double> a(0, 5);  // instantiate with double type parameter
```

```
    lib::Interval<double> b(3, 8);  // instantiate with double type parameter
    lib::Interval<int>    c(4, 9);  // instantiate with int    type parameter

    if (lib::intersect(a, b))  // instantiate deducing double type parameter
    {
        return 0;  // return "success" as (0.0, 5.0) does intersect (3.0, 8.0)
    }

    return 1;  // Return "failure" status as function apparently doesn't work.
}
```

The purpose of this application is merely to exhibit a couple of instantiations of the library *class* template, `lib::Interval`, for type parameters `int` and `double`, and of the library *function* template, `lib::intersect`, for just `double`.

Next, we compile the application and library translation units, `app.cpp` and `lib_interval.cpp`, and inspect the symbols in their respective corresponding object files, `app.o` and `lib_interval.o`:

```
$ gcc -I. -c app.cpp lib_interval.cpp
$ nm -C app.o lib_interval.o

app.o:
0000000000000000 W lib::Interval<double>::Interval(double const&, double const&)
0000000000000000 W lib::Interval<int>::Interval(int const&, int const&)
0000000000000000 W bool lib::intersect<double>(lib::Interval<double> const&,
                                               lib::Interval<double> const&)
0000000000000000 T main

lib_interval.o:
```

Looking at `app.o` in the previous example, the class and function templates used in the `main` function, which is defined in the `app.cpp` file, were instantiated *implicitly* and the relevant code was added to the resulting object file, `app.o`, with each instantiated function definition in its own separate **section**. In the `Interval` *class* template, the generated symbols correspond to the two unique instantiations of the constructor, i.e., for `double` and `int`, respectively. The `intersect` function template, however, was implicitly instantiated for only type `double`. Note importantly that all of the implicitly instantiated functions have the `W` symbol type, indicating that they are *weak* symbols, which are permitted to be present in multiple object files. By contrast, this file defines the strong symbol `main`, marked here by a `T`. Linking this file with any other file containing such a symbol would cause the linker to report a multiply-defined-symbol error. On the other hand, the `lib_interval.o` file corresponds to the `lib_interval` library component, whose `.cpp` file served only to include its own `.h` file, and is again effectively empty.

Let's now link the two object files, `app.o` and `lib_interval.o`, and inspect the symbols in the resulting executable, `app`[2]:

[2]We have stripped out extraneous unrelated information that the `nm` tool produces; note that the `-C` option invokes the symbol demangler, which turns encoded names like `_ZN3lib8IntervalIdEC1ERKdS3_` into something more readable like `lib::Interval<double>::Interval(double const&, double const&)`.

```
$ gcc -o app app.o lib_interval.o
$ nm -C app
000000000040056e T lib::Interval<double>::Interval(double const&, double const&)
00000000004005a2 T lib::Interval<int>::Interval(int const&, int const&)
00000000004005ce T bool lib::intersect<double>(lib::Interval<double> const&,
                                               lib::Interval<double> const&)
00000000004004b7 T main
```

As the textual output above confirms, each of the needed *weak* template symbols, previously marked with a `W`, is bound into the final program as a *strong* symbol, now — like `main` — marked with a `T`.[3] In this tiny illustrative example, only one set of weak symbols appeared in the combined `.o` files.

More generally, if the application comprises multiple object files, each file will potentially contain their own set of weak symbols, often leading to duplicate code **sections** for implicitly instantiated class, function, and variable templates instantiated on the same parameters. When the linker combines object files, it will arbitrarily choose at most one of each of these respective and hopefully identical weak-symbol **sections** to include in the final executable, now marked as a strong symbol (`T`).

Imagine now that our program includes a large number of `.o` files, many of which make use of our `lib_interval` component, particularly to operate on `double` intervals. Suppose, for now, we decide we would like to suppress the generation of object code for templates related to just `double` types with the intent of later putting them all in one place, i.e., the currently empty `lib_interval.o`. Achieving this objective is precisely what the `extern template` syntax is designed to accomplish.

Returning to our `lib_interval.h` file, we need not change one line of code; we need only to *add* two **explicit-instantiation declarations** — one for the template *class*, `Interval<double>`, and one for the template *function*, `intersect<double>(const double&, const double&)` — to the header file anywhere *after* their respective corresponding general template declaration and definition:

```
    std::size_t
// lib_interval.h:  // (no change to existing code)
#ifndef INCLUDED_LIB_INTERVAL  // internal include guard
#define INCLUDED_LIB_INTERVAL

namespace lib  // namespace for all entities defined within this component
{

template <typename T>                    // elided sketch of a class template
class Interval;                          // ...
    // ...                         (same as before)

template <typename T>                    // elided sketch of a function template
bool intersect(const Interval<T>& i1, const Interval<T>& i2);
    // ...                         (same as before)
```

[3] Whether the symbol is marked `W` or `T` in the final executable is implementation specific and of no consequence here. We present these concepts in this particular way to aid cognition.

```
extern template class Interval<double>;  // explicit-instantiation declaration

extern template                          // explicit-instantiation declaration
bool intersect(const Interval<double>&, const Interval<double>&);

}  // close lib namespace

#endif  // INCLUDED_LIB_INTERVAL
```

Let's again compile the two `.cpp` files and inspect the corresponding `.o` files:

```
$ gcc -I. -c app.cpp lib_interval.cpp
$ nm -C app.o lib_interval.o

app.o:
                 U lib::Interval<double>::Interval(double const&, double const&)
0000000000000000 W lib::Interval<int>::Interval(int const&, int const&)
                 U bool lib::intersect<double>(lib::Interval<double> const&,
                                               lib::Interval<double> const&)
0000000000000000 T main

lib_interval.o:
```

Notice that this time some of the symbols — specifically those relating to the `class` and `function` templates instantiated for type `double` — have changed from `W`, indicating a *weak* symbol, to `U`, indicating an *undefined* one. What this means is that, instead of generating a weak symbol for the explicit specializations for `double`, the compiler left those symbols undefined, as if only the *declarations* of the member and free-function templates had been available when compiling `app.cpp`, yet inlining of the instantiated definitions is in no way affected. **Undefined symbols** are symbols that are expected to be made available to the linker from other object files. Attempting to link this application expectedly fails because no object files being linked contain the needed definitions for those instantiations:

```
$ gcc -o app app.o lib_interval.o

app.o: In function ('main':}
app.cpp:(.text+0x38): undefined reference to
  `lib::Interval<double>::Interval(double const&, double const&)'
app.cpp:(.text+0x69): undefined reference to
  `lib::Interval<double>::Interval(double const&, double const&)'
app.cpp:(.text+0xa1): undefined reference to
  `bool lib::intersect<double>(lib::Interval<double> const&,
                               lib::Interval<double> const&)'

collect2: error: ld returned 1 exit status
```

To provide the missing definitions, we will need to instantiate them explicitly. Since the type for which the class and function are being specialized is the ubiquitous built-in type, `double`, the ideal place to sequester those definitions would be within the `.o` file of the `lib_interval` library component itself, but see *Annoyances — No good place to put definitions for unrelated*

classes on page 81. To force the needed template definitions into the `lib_interval.o` file, we will need to pull out our trusty **explicit-instantiation definition** syntax, i.e., the `template` prefix:

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

template class lib::Interval<double>;
    // example of an explicit-instantiation definition for a class

template bool lib::intersect(const Interval<double>&, const Interval<double>&);
    // example of an explicit-instantiation definition for a function
```

We recompile once again and inspect our newly generated object files:

```
$ gcc -I. -c app.cpp lib_interval.cpp
$ nm -C app.o lib_interval.o

app.o:
                 U lib::Interval<double>::Interval(double const&, double const&)
0000000000000000 W lib::Interval<int>::Interval(int const&, int const&)
                 U bool lib::intersect<double>(lib::Interval<double> const&,
                                               lib::Interval<double> const&)
0000000000000000 T main

lib_interval.o:
0000000000000000 W lib::Interval<double>::Interval(double const&)
0000000000000000 W lib::Interval<double>::Interval(double const&, double const&)
0000000000000000 W lib::Interval<double>::low() const
0000000000000000 W lib::Interval<double>::high() const
0000000000000000 W lib::Interval<double>::length() const
0000000000000000 W bool lib::intersect<double>(lib::Interval<double> const&,
                                               lib::Interval<double> const&)
```

The application `.o` file, `app.o`, naturally remained unchanged. What's new here is that the functions that were missing from the `app.o` file are now available in the `lib_interval.o` file, again as *weak* (`W`), as opposed to strong (`T`), symbols. Notice, however, that explicit instantiation forces the compiler to generate code for all of the member functions of the class template for a given specialization. These symbols might all be linked into the resulting executable unless we take explicit precautions to exclude those that aren't needed[4]:

```
$ gcc -o app app.o lib_interval.o -Wl,--gc-sections
$ nm -C app
00000000004005ca T lib::Interval<double>::Interval(double const&, double const&)
000000000040056e T lib::Interval<int>::Interval(int const&, int const&)
000000000040063d T bool lib::intersect<double>(lib::Interval<double> const&,
```

[4]To avoid including the explicitly generated definitions that are being used to resolve undefined symbols, we have instructed the linker to remove all unused code **sections** from the executable. The `-Wl` option passes comma-separated options to the linker. The `--gc-sections` option instructs the compiler to compile and assemble and instructs the linker to omit individual unused **sections**, where each section contains, for example, its own instantiation of a function template.

```
                                                    lib::Interval<double> const&)
  00000000004004b7 T main
```

tl;dr: This `extern template` feature is provided to enable software architects to reduce code bloat in individual `.o` files for common instantiations of class, function, and, as of C++14, variable templates in large-scale C++ software systems. The practical benefit is in reducing the physical size of libraries, which *might* lead to improved link times. **Explicit-instantiation declarations** do *not* (1) affect the meaning of a program, (2) suppress template instantiation, (3) impede the compiler's ability to `inline`, or (4) meaningfully improve compile time. To be clear, the *only* purpose of the `extern template` syntax is to suppress object-code generation for the current translation unit, which is then selectively overridden in the translation unit(s) of choice.

Use Cases

use-cases

Reducing template code bloat in object files

bloat-in-object-files

The motivation for the `extern template` syntax is as a purely **physical** (not **logical**) optimization, i.e., to reduce the amount of redundant code within individual object files resulting from common template instantiations in client code. As an example, consider a fixed-size-array class template, `FixedArray`, that is used widely, i.e., by many clients from separate translation units, in a large-scale `game` project for both integral and floating-point calculations, primarily with type parameters `int` and `double` and array sizes of either `2` or `3`:

```
// game_fixedarray.h:
#ifndef INCLUDED_GAME_FIXEDARRAY  // *internal* include guard
#define INCLUDED_GAME_FIXEDARRAY

#include <cstddef>  // std::size_t
namespace game  // namespace for all entities defined within this component
{

template <typename T, std::size_t N>
class FixedArray                                    // widely used class template
{
    // ... (elided private implementation details)
public:
    FixedArray()                                    { /*...*/ }
    FixedArray(const FixedArray<T, N>& other)       { /*...*/ }
    T& operator[](std::size_t index)                { /*...*/ }
    const T& operator[](std::size_t index) const { /*...*/ }
};

template <typename T, std::size_t N>
T dot(const FixedArray<T, N>& a, const FixedArray<T, N>& b) { /*...*/ }

// Explicit-instantiation declarations for full template specializations
// commonly used by the game project are provided below.

extern template class FixedArray<int, 2>;              // class template
```

```
extern template int dot(const FixedArray<int, 2>& a,   // function template
                        const FixedArray<int, 2>& b);  // for int and 2

extern template class FixedArray<int, 3>;              // class template
extern template int dot(const FixedArray<int, 3>& a,   // function template
                        const FixedArray<int, 3>& b);  // for int and 3

extern template class FixedArray<double, 2>;           // for double and 2
extern template double dot(const FixedArray<double, 2>& a,
                           const FixedArray<double, 2>& b);

extern template class FixedArray<double, 3>;           // for double and 3
extern template double dot(const FixedArray<double, 3>& a,
                           const FixedArray<double, 3>& b);

}  // close game namespace

#endif  // INCLUDED_GAME_FIXEDARRAY
```

Specializations commonly used by the `game` project are provided by the `game` library. In the component header in the example above, we have used the **`extern template`** syntax to suppress object-code generation for instantiations of both the class template `FixedArray` and the function template `dot` for element types `int` and **`double`**, each for array sizes `2` and `3`. To ensure that these specialized definitions are available in every program that might need them, we use the **`template`** syntax counterpart to *force* object-code generation within just the one `.o` corresponding to the `game_fixedarray` library component[5]:

```
// game_fixedarray.cpp:
#include <game_fixedarray.h>  // included as first substantive line of code

// Explicit-instantiation definitions for full template specializations
// commonly used by the game] project are provided below.

template class game::FixedArray<int, 2>;              // class template
template int game::dot(const FixedArray<int, 2>& a,   // function template
                       const FixedArray<int, 2>& b);  // for int and 2

template class game::FixedArray<int, 3>;              // class template
template int game::dot(const FixedArray<int, 3>& a,   // function template
                       const FixedArray<int, 3>& b);  // for int and 3

template class game::FixedArray<double, 2>;           // for double and 2
template double game::dot(const FixedArray<double, 2>& a,
                          const FixedArray<double, 2>& b);
```

[5]Notice that we have chosen *not* to nest the explicit specializations (or any other definitions) of entities already declared directly within the `game` namespace, preferring instead to qualify each entity explicitly to be consistent with how we render free-function definitions (to avoid self-declaration); see **?**, section 2.5, "Component Source-Code Organization," pp. 333–342, specifically Figure 2-36b, p. 340. See also *Potential Pitfalls — Corresponding explicit-instantiation declarations and definitions* on page 79.

```
template class game::FixedArray<double, 3>;           // for double and 3
template double game::dot(const FixedArray<double, 3>& a,
                          const FixedArray<double, 3>& b);
```

Compiling `game_fixedarray.cpp` and examining the resulting object file shows that the code for all explicitly instantiated classes and free functions was generated and placed into the object file, `game_fixedarray.o`[6]:

```
$ gcc -I. -c game_fixedarray.cpp
$ nm -C game_fixedarray.o
0000000000000000 W game::FixedArray<double, 2ul>::FixedArray(
  game::FixedArray<double, 2ul> const&)
0000000000000000 W game::FixedArray<double, 2ul>::FixedArray()
0000000000000000 W game::FixedArray<double, 2ul>::operator[](unsigned long)
0000000000000000 W game::FixedArray<double, 3ul>::FixedArray(
  game::FixedArray<double, 3ul> const&)
0000000000000000 W game::FixedArray<int, 3ul>::FixedArray()
                                        :
0000000000000000 W double game::dot<double, 2ul>(
  game::FixedArray<double, 2ul> const&, game::FixedArray<double, 2ul> const&)
0000000000000000 W double game::dot<double, 3ul>(
  game::FixedArray<double, 3ul> const&, game::FixedArray<double, 3ul> const&)
0000000000000000 W int game::dot<int, 2ul>(
  game::FixedArray<int, 2ul> const&, game::FixedArray<int, 2ul> const&)
                                        :
0000000000000000 W game::FixedArray<int, 2ul>::operator[](unsigned long) const
0000000000000000 W game::FixedArray<int, 3ul>::operator[](unsigned long) const
```

This `FixedArray` class template is used in multiple translation units within the `game` project. The first one contains a set of geometry utilities:

```
    std::size_t
// app_geometryutil.cpp:

#include <game_fixedarray.h>
#include <game_unit.h>

using namespace game;

void translate(game::Unit* object, const FixedArray<double, 2>& dst)
    // Perform precise movement of the object on 2D plane.
{
    FixedArray<double, 2> objectProjection;
    // ...
}

void translate(game::Unit* object, const FixedArray<double, 3>& dst)
    // Perform precise movement of the object in 3D space.
{
```

[6]Note that only a subset of the relevant symbols have been retained.

```
    FixedArray<double, 3> delta;
    // ...
}

bool isOrthogonal(const FixedArray<int, 2>& a1, const FixedArray<int, 2>& a2)
    // Return true if 2d arrays are orthogonal.
{
    return 0 == dot(a1, a2);
}

bool isOrthogonal(const FixedArray<int, 3>& a1, const FixedArray<int, 3>& a2)
    // Return true if 3d arrays are orthogonal.
{
    return 0 == dot(a1, a2);
}
```

The second one deals with physics calculations:

```
// app_physics.cpp:

#include <game_fixedarray.h>
#include <game_unit.h>

using namespace game;

void collide(game::Unit* objectA, game::Unit* objectB)
    // Calculate the result of object collision in 3D space.
{
    FixedArray<double, 3> centerOfMassA = objectA->centerOfMass();
    FixedArray<double, 3> centerOfMassB = objectB->centerOfMass();
    // ..
}

void accelerate(game::Unit* object, const FixedArray<double, 3>& force)
    // Calculate the position after applying a specified force for the
    // duration of a game tick.
{
    // ...
}
```

Note that the object files for the application components throughout the `game` project do not contain any of the implicitly instantiated definitions that we had chosen to uniquely sequester externally, i.e., within the `game_fixedarray.o` file:

```
$ nm -C app_geometryutil.o
000000000000003e T isOrthogonal(game::FixedArray<int, 2ul> const&,
  game::FixedArray<int, 2ul> const&)
0000000000000068 T isOrthogonal(game::FixedArray<int, 3ul> const&,
  game::FixedArray<int, 3ul> const&)
0000000000000000 T translate(game::Unit*, game::FixedArray<double, 2ul> const&)
000000000000001f T translate(game::Unit*, game::FixedArray<double, 3ul> const&)
```

```
                U game::FixedArray<double, 2ul>::FixedArray()
                U game::FixedArray<double, 3ul>::FixedArray()
                U int game::dot<int, 2ul>(game::FixedArray<int, 2ul> const&,
  game::FixedArray<int, 2ul> const&)
                U int game::dot<int, 3ul>(game::FixedArray<int, 3ul> const&,
  game::FixedArray<int, 3ul> const&)

$ nm -C app_physics.o
0000000000000039 T accelerate(game::Unit*,
  game::FixedArray<double, 3ul> const&)
0000000000000000 T collide(game::Unit*, game::Unit*)
                U game::FixedArray<double, 3ul>::FixedArray()
0000000000000000 W game::Unit::centerOfMass()
```

Whether optimization involving **explicit-instantiation directives** reduces library sizes on disc has no noticeable effect or actually makes matters worse will depend on the particulars of the system at hand. Having this optimization applied to frequently used templates across a large organization has been known to decrease object file sizes, storage needs, link times, and overall build times, but see *Potential Pitfalls — Accidentally making matters worse* on page 81.

Insulating template definitions from clients

initions-from-clients

Even before the introduction of **explicit-instantiation *declarations***, strategic use of **explicit-instantiation *definitions*** made it possible to **insulate** the *definition* of a template from client code, presenting instead just a limited set of instantiations against which clients may link. Such insulation enables the definition of the template to change without forcing clients to recompile. What's more, new explicit instantiations can be added without affecting existing clients.

As an example, suppose we have a single free-function template, `transform`, that operates on only floating-point values:

```
// transform.h:
#ifndef INCLUDED_TRANSFORM_H
#define INCLUDED_TRANSFORM_H

template <typename T>  // declaration (only) of free-function template
T transform(const T& value);
    // Return the transform of the specified floating-point value.

#endif
```

Initially, this function template will support just two built-in types, **`float`** and **`double`**, but it is anticipated to eventually support the additional built-in type **`long double`** and perhaps even supplementary user-defined types (e.g., `Float128`) to be made available via separate headers (e.g., `float128.h`). By placing only the declaration of the `transform` function template in its component's header, clients will be able to link against only two supported explicit specializations provided in the `transform.cpp` file:

```
// transform.cpp:
```

```
#include <transform.h>  // Ensure consistency with client-facing declaration.

template <typename T>   // redeclaration/definition of free-function template
T transform(const T& value)
{
    // insulated implementation of transform function template
}

// explicit-instantiation *definitions*
template float transform(const float&);    // Instantiate for type float.
template double transform(const double&);  // Instantiate for type double.
```

Without the two **explicit-instantiation definitions** in the `transform.cpp` file above, its corresponding object file, `transform.o`, would be empty.

Note that, as of C++11, we *could* place the corresponding **explicit-instantiation definitions** in the header file for, say, documentation purposes:

```
// transform.h:
#ifndef INCLUDED_TRANSFORM_H
#define INCLUDED_TRANSFORM_H

template <typename T>  // declaration (only) of free-function template
T transform(const T& value);
    // Return the transform of the specified floating-point value.

// explicit-instantiation declarations, available as of C++11
extern template float transform(const float&);    // user documentation only;
extern template double transform(const double&);  // has no effect whatsoever

#endif
```

But because no definition of the `transform` free-function template is visible in the header, no *implicit* instantiation can result from client use; hence, the two **explicit-instantiation declarations** above for `float` and `double`, respectively, do nothing.

pitfalls-externtemplate

rations-and-definitions

Potential Pitfalls

Corresponding explicit-instantiation declarations and definitions

To realize a reduction in object-code size for individual translation units and yet still be able to link all valid programs successfully into a well-formed program, several moving parts have to be brought together correctly:

1. Each general template, `C<T>`, whose object code bloat is to be optimized must be declared within some designated component's header file, `c.h`.

2. The specific definition of each `C<T>` relevant to an explicit specialization being optimized — including general, partial-specialization, and full-specialization definitions — must appear in the header file prior to its corresponding **explicit-instantiation declaration**.

3. Each **explicit-instantiation declaration** for each specialization of each separate top-level — i.e., class, function, or variable — template must appear in the component's `.h` file *after* the corresponding general template declaration and the relevant general, partial-specialization, or full-specialization definition, but, in practice, always after *all* such definitions, not just the relevant one.

4. Each template specialization having an **explicit-instantiation declaration** in the header file must have a corresponding **explicit-instantiation definition** in the component's implementation file, `c.cpp`.

Absent items (1) and (2), clients would have no way to safely separate out the usability and inlineability of the template definitions yet consolidate the otherwise redundantly generated object-level definitions within just a single translation unit. Moreover, failing to provide the relevant definition would mean that any clients using one of these specializations would either fail to compile or, arguably worse, pick up the general definitions when a more specialized definition was intended, likely resulting in an ill-formed program.

Failing item (3), the object code for that particular specialization of that template will be generated locally in the client's translation unit as usual, negating any benefits with respect to local object-code size, irrespective of what is specified in the `c.cpp` file.

Finally, unless we provide a matching **explicit-instantiation definition** in the `c.cpp` file for each and every corresponding **explicit-instantiation declaration** in the `c.h` file as in item (4), our optimization attempts might well result in a library component that compiles, links, and even passes some unit tests but, when released to our clients, fails to link. Additionally, any **explicit-instantiation definition** in the `c.cpp` file that is not accompanied by a corresponding **explicit-instantiation declaration** in the `c.h` file will inflate the size of the `c.o` file with no possibility of reducing code bloat in client code[7]:

```
// c.h:
#ifndef INCLUDED_C                              // internal include guard
#define INCLUDED_C

template <typename T> void f(T v) {/*...*/};  // general template definition

extern template void f<int>(int v);             // OK, matched in c.cpp
extern template void f<char>(char c);           // Error, unmatched in .cpp file

#endif


// c.cpp:
#include <c.h>                                   // incorporate own header first

extern template void f<int>(int v);              // OK, matched in c.h
extern template void f<double>(double v);        // Bug, unmatched in c.h file
```

[7]Fortunately, these extra instantiations do not result in multiply-defined symbols because they still reside in their own **sections** and are marked as *weak* symbols.

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

void client()
{
    int    i = 1;
    char   c = 'a';
    double d = 2.0;

    f(i);  // OK, matching explicit-instantiation directives
    f(c);  // Link-Time Error, no matching explicit-instantiation definition
    f(d);  // Bug, size increased due to no matching explicit-instantiation
           // declaration
}
```

In the example above, `f(i)` works as expected, with the linker finding the definition of `f<int>` in `c.o`; `f(c)` fails to link, because no definition of `f<c>` is guaranteed to be found anywhere; and `f(d)` accidentally works by silently generating a *redundant* local copy of `f<double>` in `client.o` while another, identical definition is generated explicitly in `c.o`. Importantly, note that `extern template` has *absolutely no affect* on overload resolution because the call to `f(c)` did *not* resolve to `f<int>`.

Accidentally making matters worse

ly-making-matters-worse

When making the decision to preinstantiate common specializations of popular templates within some designated `.o` file, one must consider that not all programs necessarily need every (or even any) such instantiation. Special consideration should be given to classes that have many methods but typically use only a few.

The language feature is sufficiently flexible that one can suppress and preinstantiate just one or a handful of member functions of such a type. Intuition is all well and good, but measurement simply has no substitute.

If one suspects that **explicit-instantiation directives** might profitably reduce the size of libraries resulting from object code that is bloated due to redundant local reinstantiations of popular templates on common types, measuring before and after and retaining the change *only* if it offers a significant — at least measurable — improvement avoids complicating the codebase without a verifiable return on the investment. Finally, remember that one might need to explicitly tell the linker to strip unused **sections** resulting, for example, from forced instantiation of common template specializations, to avoid inadvertently bloating executables, which could adversely affect load times.

Annoyances

noyances-externtemplate

No good place to put definitions for unrelated classes

s-for-unrelated-classes

When we consider the implications of physical dependency,[8,9] determining in which component to deposit the specialized definitions can be problematic. For example, consider a

[8] See **?**
[9] See **?**

codebase implementing a core library that provides both a nontemplated `String` class and a `Vector` container class template. These fundamentally unrelated entities would ideally live in separate physical **components** — i.e., `.h`/`.cpp` pairs — neither of which depends physically on the other. That is, an application using just one of these components could ideally be compiled, linked, tested, and deployed entirely independently of the other. Now, consider a large codebase that makes heavy use of `Vector<String>`: In what component should the object-code-level definitions for the `Vector<String>` specialization reside?[10] There are two obvious alternatives:

1. `vector`: In this case, `vector.h` would hold `extern template class Vector<String>;` — the **explicit-instantiation *declaration*** — and `vector.cpp` would hold `template class Vector<String>;` — the **explicit-instantiation *definition***. With this approach, we would create a physical dependency of the `vector` component on `string`. Any client program wanting to use a `Vector` would also depend on `string` regardless of whether it was needed.

2. `string`: In this case, `string.h` and `string.cpp` would instead be modified so as to depend on `vector`. Clients wanting to use a `string` would also be forced to depend physically on `vector` *at compile time*.

Another possibility might be to create a third component, call it `stringvector`, that itself depends on both `vector` and `string`. By **escalating**[11] the mutual dependency to a higher level in the physical hierarchy, we avoid forcing any client to depend on more than what is actually needed. The practical drawback to this approach is that only those clients that proactively include the composite `stringvector.h` header would realize any benefit; fortunately, in this case, there is no **one-definition rule (ODR)** issue if they don't.

Finally, complex machinery could be added to both `string.h` and `vector.h` to conditionally include `stringvector.h` whenever both of the other headers are included; such heroic efforts would, nonetheless, involve a **cyclic physical dependency** among all three of these components. Circular intercomponent collaborations are best avoided.[12]

e-class-must-be-valid

All members of an explicitly defined template class must be valid

In general, when using a template class, only those members that are actually used get implicitly instantiated. This hallmark allows class templates to provide functionality for parameter types having certain capabilities (e.g., default constructible) while also providing partial support for types lacking those same capabilities. When providing an **explicit-instantiation definition**, however, *all* members of a template class are instantiated.

Consider a simple class template having a data member that can be either default-initialized (via the template's default constructor) or initialized with an instance of the member's type (supplied at construction):

```
template <typename T>
```

[10] Note that the problem of determining in which component to instantiate the object-level implementation of a template for a user-defined type is similar to that of specializing an arbitrary user-defined trait for a user-defined type.

[11] **?**, section 3.5.2, "Escalation," pp. 604–614

[12] **?**, section 3.4, "Avoiding Cyclic Link-Time Dependencies," pp. 592–601

```
class W
{
    T d_t;  // a data member of type T

public:
    W() : d_t() {}
        // Create an instance of W with a default-constructed T member.

    W(const T& t) : d_t(t) {}
        // Create an instance of W with a copy of the specified t.

    void doStuff() { /* do stuff */ }
};
```

This class template can be used successfully with a type, such as `U` in the code snippet below, that is not default constructible:

```
struct U
{
    U(int i) { /* do something with i */ }
    // ...
};

void useWU()
{
    W<U> wu1(U(17));  // OK, using copy constructor for U
    wu1.doStuff();
}
```

As it stands, the code above is well formed even though `W<U>::W()` would fail to compile if instantiated. Consequently, although providing an **explicit-instantiation declaration** for `W<U>` is valid, a corresponding **explicit-instantiation definition** for `W<U>` fails to compile, as would an implicit instantiation of `W<U>::W()`:

```
extern template class W<U>;  // Valid: Suppress implicit instantiation of W<U>.

template class W<U>;         // Error, U::U() not available for W<U>::W()

void useWU0()
{
    W<U> wu0{};       // Error, U::U() not available for W<U>::W()
}
```

Unfortunately, the only workaround to achieve a comparable reduction in code bloat is to provide member-specific **explicit-instantiation directives** for each valid member of `W<U>`, an approach that would likely carry a significantly greater maintenance burden:

```
extern template W<U>::W(const U& u);   // suppress individual member
extern template void W<U>::doStuff();  //    "         "        "
// ... Repeat for all other functions in W except W<U>::W().
```

```
template W<U>::W(const U& u);          // instantiate individual member
template void W<U>::doStuff();         //       "         "        "
// ... Repeat for all other functions in W except W<U>::W().
```

The power and flexibility to make it all work — albeit annoyingly — are there nonetheless.

see-also

See Also

- "??" (Section 1.2, p. ??) ♦ Extension of the template syntax for defining a family of like-named variables or static data members that can be instantiated explicitly

further-reading

Further Reading

- For a different perspective on this feature, see ?, section 1.3.16, "`extern` Templates," pp. 183–185.
- For a more complete discussion of how compilers and linkers work with respect to C++, see ?, Chapter 1, "Compilers, Linkers, and Components," pp. 123–268.

Forwarding && References

forwardingref
forwarding-references 2.5— 0— 0

A forwarding reference (`T&&`) — distinguishable from an rvalue reference (`&&`) (see "**??**" on page **??**) based only on context — is a distinct, special kind of reference that (1) binds universally to the result of an expression of *any* **value category** and (2) preserves aspects of that **value category** so that the bound object can be *moved from*, if appropriate.

Description

scription-forwardingref

Sometimes we want the same reference to be able to bind to either an *lvalue* or an *rvalue* and then later be able to discern, from the reference itself, whether the result of the original expression was eligible to be *moved from*. A *forwarding reference* (e.g., `forRef` in the example below) used in the interface of a function *template* (e.g., `myFunc`) affords precisely this capability and will prove invaluable for the purpose of conditionally moving, or else copying, an object from within the function template's body:

```
template <typename T>
void myFunc(T&& forRef)
{
    // It is possible to check if forRef is eligible to be moved from or not
    // from within the body of myFunc.
}
```

In the definition of the `myFunc` function template in the example above, the parameter `forRef` syntactically appears to be a non-`const` reference to an *rvalue* of type `T`; in this very precise context, however, the same `T&&` syntax designates a **forwarding reference**, with the effect of retaining the original value category of the object bound to the argument `forRef`; see *Description: Identifying forwarding references* on page 89. The `T&&` syntax represents a *forwarding* reference — as opposed to an *rvalue* reference — whenever an *individual* function template has a type parameter (e.g., `T`) and an unqualified function parameter of type that is exactly `T&&` (e.g., `const T&&` would be an *rvalue* reference, not a forwarding reference).

Consider, for example, a function `f` that takes a single argument by reference and then attempts to use it to invoke one of two overloads of a function `g`, depending on whether the original argument was an *lvalue* or *rvalue*:

```
struct S { /* some UDT that might benefit from being able to be moved */ };

void g(const S&);  // target function - overload for const int lvalues
void g(S&&);       // target function - overload for int rvalues only

template <typename T>
void f(T&& forRef); // forwards to target overload g based on value category
```

In theory, we could have chosen a non-`const` *lvalue* reference along with a modifiable *rvalue* reference here for *pedagogical* symmetry; such an inherently unharmonious overload set would, however, not typically occur in practice; see "**??**" on page **??**. In this specific case — where `f` is a function template, `T` is a template type parameter, and the type of the parameter itself is exactly `T&&` — the `forRef` function parameter (in the code snippet above)

f-invoked-example denotes a *forwarding reference*. If `f` is invoked with an *lvalue*, `forRef` is an *lvalue* reference; otherwise `forRef` is an *rvalue* reference. Given the dual nature of `forRef`, one rather verbose way of determining the original value category of the passed argument would be to use the `std::is_lvalue_reference` **type trait** on `forRef` itself:

```
#include <type_traits>  // std::is_lvalue_reference
#include <utility>  // std::move

template <typename T>
void f(T&& forRef)      // forRef is a forwarding reference.
{
    if (std::is_lvalue_reference<T>::value)  // using a C++11 type trait
    {
        g(forRef);             // propagates forRef as an *lvalue*
    }                          // invokes g(const S&)
    else
    {
        g(std::move(forRef));  // propagates forRef as an *rvalue*
    }                          // invokes g(S&&)
}
```

The `std::is_lvalue_reference<T>::value` predicate above asks the question, "Did the object bound to `fRef` originate from an lvalue expression?" and allows the developer to branch on the answer. A more concise but otherwise equivalent implementation is generally preferred; see *Description: The* `std::forward` *utility* on page 92:

```
#include <utility>  // std::forward

template <typename T>
void f(T&& forRef)
{
    g(std::forward<T>(forRef));
        // same as g(std::move(forRef)) if and only if forRef is an *rvalue*
        // reference; otherwise, equivalent to g(forRef)
}
```

A `client` function invoking `f` will enjoy the same behavior with either of the two implementation alternatives offered above:

```
void client()
{
    S s;
    f(s);    // Instantiates f<S&> -- fRef is an lvalue reference (S&).
             // The function f<S&> will end up invoking g(S&).

    f(S());  // Instantiates f<S&&> -- fRef is an rvalue reference (S&&).
             // The function f<S&&> will end up invoking g(S&&).
}
```

Use of `std::forward` in combination with forwarding references is typical in the implementation of industrial-strength generic libraries; see *Use Cases* on page 92.

late-argument-deduction

A brief review of function template argument deduction

Invoking a function template without explicitly providing template arguments at the call site will compel the compiler to attempt, if possible, to *deduce* those template *type* arguments from the function arguments:

```
template <typename T> void f();
template <typename T> void g(T x);
template <typename T> void h(T y, T z);

void example0()
{
    f();        // Error, couldn't infer template argument T
    g(0);       // OK, T deduced as int from literal 0 -- x is an int.
    h(0, 'a');  // Error, deduced conflicting types for T (int vs. char)
}
```

Any **cv-qualifiers** (`const`, `volatile`, or both) on a *deduced* function parameter will be applied *after* type deduction is performed:

```
template <typename T> void cf(const T x);
template <typename T> void vf(volatile T y);
template <typename T> void wf(const volatile T z);

void example1()
{
    cf(0);  // OK, T deduced as int -- x is a const int.
    vf(0);  // OK, T deduced as int -- y is a volatile int.
    wf(0);  // OK, T deduced as int -- z is a const volatile int.
}
```

Similarly, **ref-qualifiers** other than `&&` (`&` or `&&` along with any cv-qualifier) do not alter the deduction process, and they too are applied after deduction:

```
template <typename T> void rf(T& x);
template <typename T> void crf(const T& x);

void example2()
{
    int i;
    rf(i);   // OK, T is deduced as int -- x is an int&.
    crf(i);  // OK, T is deduced as int -- x is a const int&.

    rf(0);   // Error, expects an lvalue for 1st argument
    crf(0);  // OK, T is deduced as int -- x is a const int&.
}
```

Type deduction works differently for *forwarding* references where the only qualifier on the template argument is `&&`. For the sake of exposition, consider a function template declaration, `f`, accepting a forwarding reference, `forRef`:

```
template <typename T> void f(T&& forRef);
```

We have seen in the example on page 86 that, when `f` is invoked with an *lvalue* of type `S`, then `T` is deduced as `S&` and `forRef` becomes an *lvalue* reference. When `f` is instead invoked with an *xvalue* of type `S`, then `T` is deduced as `S` and `forRef` becomes an *rvalue* reference. The underlying process that results in this "duality" relies on a special rule (known as **reference collapsing**; see the next section) introduced as part of **type deduction**. When the type `T` of a *forwarding* reference is being deduced from an expression `E`, `T` itself will be deduced as an *lvalue* reference if `E` is an *lvalue*; otherwise normal type-deduction rules will apply and `T` will be deduced as an *rvalue* reference:

```
void g()
{
    int i;
    f(i);  // i is an *lvalue* expression.
           // T is therefore deduced as int& -- special rule!
           // T&& becomes int& &&, which collapses to int&.

    f(0);  // 0 is an *rvalue* expression.
           // T is therefore deduced as int.
           // T&& becomes int&&, which is an *rvalue* reference.
}
```

For more on general type deduction, see "**??**" on page **??**.

reference-collapsing

Reference collapsing

As we saw in the previous section, when a function having a *forwarding* reference parameter, `forRef`, is invoked with a corresponding *lvalue* argument, an interesting phenomenon occurs: After type deduction, we temporarily get what appears syntactically to be an *rvalue* reference to an *lvalue* reference. As references to references are *not* allowed in C++, the compiler employs **reference collapsing** to resolve the *forwarding*-reference parameter, `forRef`, into a single reference, thus providing a way to infer, from `T` itself, the original **value category** of the argument passed to `f`.

The process of **reference collapsing** takes place automatically in any situation where a reference to a reference is formed. Table 1 illustrates the simple rules for collapsing "unstable" references into "stable" ones. Notice, in particular, that an *lvalue* reference always overpowers an *rvalue* reference. The only situation in which two references collapse into an *rvalue* reference is when they are both *rvalue* references.

Table 1: Collapsing "unstable" reference pairs into a single "stable" one

forwardingref-table1

1st Reference Type	2nd Reference Type	Result of Reference Collapsing
&	&	&
&	&&	&
&&	&	&
&&	&&	&&

Finally, note that it is not possible to write a reference-to-reference type in C++

explicitly:

```
int    i   = 0;   // OK
int&   ir  = i;   // OK
int& & irr = ir;  // Error, irr declared as a reference to a reference
```

It is, however, easy to do so with type aliases and template parameters, and that is where reference collapsing comes into play:

```
#include <type_traits>  // std::is_same)
using i = int&;  // OK
using j = i&;    // OK, int& & becomes int&.
static_assert(std::is_same<j,int&>::value);
```

During computations involving **metafunctions**, or as part of language rules (such as type deduction), however, references to references can occur spontaneously:

```
template <typename T>
struct AddLvalueRef { typedef T& type; };
    // metafunction that transforms to an *lvalue* reference to T

template <typename T>
void f(T input)
{
    typename AddLvalueRef<T>::type ir1 = input;    // OK, adds & to make T&
    typename AddLvalueRef<T&>::type ir2 = input;   // OK, collapses to T&
    typename AddLvalueRef<T&&>::type ir3 = input;  // OK, collapses to T&
}
```

Notice that we are using the `typename` keyword in the example above as a generalized way of indicating, during **template instantiation**, that a dependent name is a type (as opposed to a value).

g-forwarding-references

Identifying forwarding references

The syntax for a *forwarding* reference (`&&`) is the same as that for *rvalue* references; the only way to discern one from the other is by observing the surrounding context. When used in a manner where **type deduction** can take place, the `T&&` syntax does *not* designate an *rvalue* reference; instead, it represents a *forwarding* reference; for type deduction to be in effect, an *individual* (possibly member) function *template* must have a type parameter (e.g., `T`) and a function parameter of type that exactly matches that parameter followed by `&&` (e.g., `T&&`):

```
struct S0
{
    template <typename T>
    void f(T&& forRef);
        // OK, fully eligible for template-argument type deduction: forRef
        // is a forwarding reference.
};
```

Note that if the function parameter is qualified, the syntax reverts to the usual meaning of *rvalue* reference:

```
struct S1
{
    template <typename T>
    void f(const T&& crRef);
        // Eligible for type deduction but is not a forwarding reference: due
        // to the const qualifier, crRef is an *rvalue* reference.
};
```

If a member function of a class template is not itself also a template, then its template type parameter will not be deduced:

```
template <typename T>
struct S2
{
    void f(T&& rRef);
       // Not eligible for type deduction because T is fixed and known as part
       // of the instantiation of S2: rRef is an *rvalue* reference.
};
```

More generally, note that the `&&` syntax can *never* imply a *forwarding* reference for a function that is not itself a template; see *Annoyances: Forwarding references look just like rvalue references* on page 103.

non-parameter-context

`auto&&` — a forwarding reference in a non-parameter context

Outside of template function parameters, *forwarding* references can also appear in the context of variable definitions using the `auto` variable (see "**??**" on page **??**) because they too are subject to type deduction:

```
void f()
{
    auto&& i = 0;  // i is a forwarding reference because the type of i must
                   // be deduced from the initialization expression 0.
}
```

Just like function parameters, `auto&&` resolves to either an *lvalue* reference or *rvalue* reference depending on the **value category** of the initialization expression:

```
void g()
{
    int i;
    auto&& lv = i;  // lv is an int&.

    auto&& rv = 0;  // rv is an int&&.
}
```

Similarly to `const auto&`, the `auto&&` syntax binds to anything. In `auto&&`, however, the `const`-ness of the bound object is *preserved* rather than always enforced:

```
void h()
{
    int        i = 0;
```

```
    const int ci = 0;

    auto&& lv  = i;   // lv is an int&.
    auto&& clv = ci;  // clv is a const int&.
}
```

Just as with function parameters, the original **value category** of the expression used to initialize a *forwarding* reference variable can be propagated during subsequent function invocation – e.g., using `std::forward` (see *Description: The* `std::forward` *utility* on page 92):

```
    std::forward
template <typename T>
void use(T&& t); // Here use also takes a forwarding reference parameter
                 // to do with as it pleases.

template <typename T>
void l()
{
    auto&& fr = get<T>();
        // get<T>() might be either an *lvalue* or *rvalue* depending on T.

    use(std::forward<decltype(fr)>(fr));  // decltype is a C++11 feature.
        // Propagate the original value category of get<T>() into use.
}
```

Notice that, because (1) `std::forward` (see the next section) requires the type of the object that's going to be forwarded as a user-provided template argument and (2) it is not possible to name the type of `fr`, `decltype` (see "**??**" on page **??**) was used in the example above to retrieve the type of `fr`.

Forwarding references without forwarding

nces-without-forwarding

Sometimes deliberately *not* forwarding (see *Description: The* `std::forward` *utility* on page 92) an `auto&&` variable or a forwarding reference function parameter at all can be useful, instead employing *forwarding* references solely for their `const`-preserving and universal binding semantics. As an example, consider the task of obtaining iterators over a range of an unknown **value category**:

```
#include <iterator>  // std::begin, std::end,

template <typename T>
void m()
{
    auto&& r = getRange<T>();
        // getRange<T>() might be either an lvalue or rvalue depending on T.

    auto b = std::begin(r);
    auto e = std::end(r);

    traverseRange(b, e);
}
```

Using `std::forward` in the initialization of both `b` and `e` might result in moving from `r` twice, which is potentially unsafe (see "??" on page ??). Forwarding `r` only in the initialization of `e` might avoid issues caused by moving an object twice but might result in inconsistent behavior with `b`, especially if the implementation of `r` makes use of reference qualifiers (see "??" on page ??).

The `std::forward` utility

-std::forward-utility

The final piece of the forwarding reference puzzle is the `std::forward` utility function. Since the expression naming a forwarding reference `x` is always an *lvalue* — due to its reachability either by name or address in virtual memory — and since our intention is to move `x` in case it was an *rvalue* to begin with, we need a conditional *move* operation that will move `x` only in that case and otherwise let `x` pass through as an *lvalue*.

The declaration for `std::forward<T>` is as follows (in `<utility>`):

std::remove_reference

```
template <class T> T&& forward(typename remove_reference<T>::type& t) noexcept;
template <class T> T&& forward(typename remove_reference<T>::type&& t) noexcept;
```

The second overload is ill-formed if invoked when `T` is an *lvalue* reference type.

Remember that the type `T` associated with a forwarding reference is deduced as a reference type if given an *lvalue* reference and as a non-reference type otherwise. So for a forwarding reference `forRef` of type `T&&`, we have two cases:

- An *lvalue* of type `U` was used for initializing `forRef`, so `T` is `U&`, thus the first overload of `forward` will be selected and will be of the form `U& forward(U& u) noexcept`, thus just returning the original *lvalue* reference.
- An *rvalue* of type `U` was used for initializing `forRef`, so `T` is `U`, so the second overload of `forward` will be selected and will be of the form `U&& forward(U& u) noexcept`, essentially equivalent to `std::move`.

Note that, in the body of a function template accepting a forwarding reference `T&&` named `x`, `std::forward<T>(x)` could be replaced with `static_cast<T&&>(x)` to achieve the same effect. Due to **reference collapsing** rules, `T&&` will resolve to `T&` whenever the original **value category** of `x` was an *lvalue* and to `T&&` otherwise, thus achieving the *conditional move* behavior elucidated in *Description* on page 85. Using `std::forward` over `static_cast` will, however, ensure that the types of `T` and `x` match, preventing accidental unwanted conversions and, separately, perhaps also more clearly expressing the programmer's intent.

Use Cases

e-cases-forwardingref

Perfectly forwarding an expression to a downstream consumer

a-downstream-consumer

A frequent use of forwarding references and `std::forward` is to propagate an object, whose **value category** is invocation-dependent, down to one or more service providers that will behave differently depending on the **value category** of the original argument.

As an example, consider an overload set for a function, `sink`, that accepts a `std::string` either by `const` *lvalue* reference (e.g., with the intention of *copying* from it) or *rvalue* reference (e.g., with the intention of *moving* from it):

```
    std::stringstd::move

void sink(const std::string& s) { target = s; }
void sink(std::string&& s)      { target = std::move(s); }
```

Now, let's assume that we want to create an intermediary function template, `pipe`, that will accept an `std::string` of any **value category** and will dispatch its argument to the corresponding overload of `sink`. By accepting a *forwarding* reference as a function parameter and invoking `std::forward` as part of `pipe`'s body, we can achieve our original goal without any code duplication:

```
template <typename T>
void pipe(T&& x)
{
    sink(std::forward<T>(x));
}
```

Invoking `pipe` with an *lvalue* will result in `x` being an *lvalue* reference and thus `sink(const std::string&)`'s being called. Otherwise, `x` will be an *rvalue* reference and `sink(std::string&&)` will be called. This idea of enabling *move* operations without code duplication (as `pipe` does) is commonly referred to as *Use Cases: Perfect forwarding for generic factory functions* on page 94.

Handling multiple parameters concisely

le-parameters-concisely

Suppose you have a **value-semantic type (VST)** that holds a collection of attributes where some (not necessarily proper) subset of them need to be changed together[1]:

```
#include <type_traits>  // std::enable_if
#include <utility>      // std::forward

struct Person { /* UDT that benefits from move semantics */ };

class StudyGroup
{
    Person d_a;
    Person d_b;
    Person d_c;
    Person d_d;
    // ...

public:
    bool isValid(const Person& a, const Person& b, const Person& c, const Person& d);
        // Return true if these specific people form a valid study group under
        // the guidelines of the study-group commission, and false otherwise.
    // ...

    template <typename PA, typename PB, typename PC, typename PD,
```

[1] This type of value-semantic type can be classified more specifically as a *complex-constrained* attribute class; see **lakos2a**, section 4.2.

```
        typename = typename std::enable_if<
            std::is_same<typename std::decay<PA>::type, Person>::value &&
            std::is_same<typename std::decay<PB>::type, Person>::value &&
            std::is_same<typename std::decay<PC>::type, Person>::value &&
            std::is_same<typename std::decay<PD>::type, Person>::value>::type>
    int setPersonsIfValid(PA&& a, PB&& b, PC&& c, PD&& d)
    {
        enum { e_SUCCESS = 0, e_FAIL };

        if (!isValid(a, b, c, d))
        {
            return e_FAIL;  // bad choice; no change
        }

        // Move or copy each person into this object's Person data members:
        d_a = std::forward<PA>(a);
        d_b = std::forward<PB>(b);
        d_c = std::forward<PC>(c);
        d_d = std::forward<PD>(d);

        return e_SUCCESS;  // Study group was updated successfully.
    }
};
```

The `setPersonsIfValid` function is producing the full crossproduct of instantiations for every variation of qualifiers that can be on a `Person` object. Any combination of *lvalue* and *rvalue* `Person`s can be passed, and a template will be instantiated that will copy the *lvalues* and move from the *rvalues*. To make sure the `Person` objects are created externally, the function is restricted, using `std::enable_if`, to instantiate only for types that decay to `Person` (i.e., types that are cv-qualified or ref-qualified `Person`). Because each parameter is a forwarding reference, they can all implicitly convert to `const Person&` to pass to `isValid`, creating no additional temporaries. Finally, `std::forward` is then used to do the actual moving or copying as appropriate to data members.

ric-factory-functions

Perfect forwarding for generic factory functions

Consider the prototypical standard-library generic factory function, `std::make_shared<T>`. On the surface, the requirements for this function are fairly simple — allocate a place for a `T` and then construct it with the same arguments that were passed to `make_shared`. This, however, gets reasonably complex to implement efficiently when `T` can have a wide variety of ways in which it might be initialized.

For simplicity, we will show how a two-argument `my::make_shared` might be defined, knowing that a full implementation would employ variadic template arguments for this purpose — see "**??**" on page **??**. We will also implement a simpler version of `make_shared` that simply creates the element on the heap with `new` and constructs a `std::shared_ptr` to manage the lifetime of that element. The declaration of this form of make shared would be structured like this:

```
    std::shared_ptr
```

```
namespace my {
template <typename ELEMENT_TYPE, typename ARG1, typename ARG2>
std::shared_ptr<ELEMENT_TYPE> make_shared(ARG1&& arg1, ARG2&& arg2);
}
```

As you can see, we have two forwarding reference arguments — `arg1` and `arg2` — with deduced types `ARG1` and `ARG2`. Now, the body of our function needs to carefully construct our `ELEMENT_TYPE` object on the heap and then create our output `shared_ptr`:

```
template <typename ELEMENT_TYPE, typename ARG1, typename ARG2>
std::shared_ptr<ELEMENT_TYPE> my::make_shared(ARG1&& arg1, ARG2&& arg2)
{
    ELEMENT_TYPE *element_p = new ELEMENT_TYPE(std::forward<ARG1>(arg1),
                                               std::forward<ARG2>(arg2));
    try
    {
        return std::shared_ptr<ELEMENT_TYPE>(element_p);
    }
    catch (...)
    {
        delete element_p;
        throw;
    }
}
```

Note that this simplified implementation needs to take care that the constructor for the return value does not throw, cleaning up the allocated element if that should happen; normally a **RAII** proctor to manage this ownership would be a more robust solution to this problem.

Importantly, the use of `std::forward` to construct the element means that the arguments passed to `make_shared` will be used to find the appropriate matching two-argument constructor of `ELEMENT_TYPE`. When those arguments are *rvalues*, the constructor found will again search for one that takes an *rvalue* and the arguments will be moved from. Even more, because this function wants to forward exactly the `const`-ness and reference type of the input arguments, we would have to write 12 distinct overloads for each argument if we were not using perfect forwarding – the full cross product of `const` (or not), `volatile` (or not), and `&`, `&&`, (or not). This would mean a full implementation of just this two-argument variation would require 144 distinct overloads, all almost identical and most never actually instantiated. The use of forwarding references reduces that to just 1 overload for each number of arguments.

Wrapping initialization in a generic factory function

eneric-factory-function

Occasionally we might want to initialize an object with an intervening function call wrapping the actual construction of that object. Suppose we have a tracking system that we want to use to monitor how many times certain initializers have been invoked:

std::forward

```
struct TrackingSystem
{
```

```
    template <typename T>
    static void trackInitialization(int numArgs);
        // Track the creation of a T with a constructor taking numArgs
        // arguments.
};
```

Now we want to write a general utility function that can be used to construct an arbitrary object and notify the tracking system of the construction for us. Here we will use a variadic pack (see "**??**" on page **??**) of forwarding references to handle calling the constructor for us:

```
template <class ELEMENT_TYPE, typename... ARGS>
ELEMENT_TYPE trackConstruction(ARGS&&... args)
{
    TrackingSystem::trackInitialization<ELEMENT_TYPE>(sizeof...(args));
    return ELEMENT_TYPE(std::forward<ARGS>(args)...);
}
```

This lets us add tracking easily to convert any initialization to a tracked one by inserting a call to this function around the constructor arguments:

```
void myFunction()
{
    BigObject untracked("Hello", "World");
    BigObject tracked = trackConstruction<BigObject>("Hello","World");
}
```

On the surface there does seem to be a difference between how `untracked` and `tracked` are initialized. The first variable is having its constructor directly invoked, while the second is being constructed from an object being returned by-value from `trackConstruction`. This construction, however, has long been something that has been optimized away to avoid any additional objects and construct the object in question just once. In this case, because the element being returned is initialized by the `return` statement of `trackConstruction`, the optimization is called **return value optimization (RVO)**. C++ has always allowed this optimization by enabling **copy elision**. In C++17, this elision can even be guaranteed and is allowed to be done for objects that have no copy or move constructors. Prior to C++17, this elision can still be guaranteed (on all compilers that the authors are aware of) by **declaring** but not **defining** the copy constructor for `BigObject`. You'll find that this code will still compile and link with such an object, providing observable proof that the copy constructor is never actually invoked with this pattern.

Emplacement

emplacement

Prior to C++11, inserting an object into a standard library container always required the programmer to first create such an object and then copy it inside the container's storage. As an example, consider inserting a temporary `std::string` object in a `std::vector<std::string>`:

```
    std::vectorstd::string

void f(std::vector<std::string>& v)
{
    v.push_back(std::string("hello world"));
```

```
        // invokes std::string::string(const char*) and the copy-constructor
}
```

In the function above, a temporary `std::string` object is created in the stack frame of `f` and is then copied to the dynamically allocated buffer managed by `v`. Additionally, the buffer might have insufficient capacity and hence might require reallocation, which would in turn require every element of `v` to be somehow copied from the old buffer to the new, larger one.

In C++11, the situation is significantly better thanks to rvalue references. The temporary will be moved into `v`, and any buffer reallocation will *move* the elements between buffers rather than copy them, assuming that the element's move-constructor is a noexcept specifier (see "**??**" on page **??**). The amount of work can, however, be further minimized: What if, instead of first creating an object externally, we constructed the new `std::string` object directly in `v`'s buffer?

This is where **emplacement** comes into play. All standard library containers, including `std::vector`, now provide an **emplacement** API powered by variadic templates (see "**??**" on page **??**) and perfect forwarding (see *Use Cases: Perfect forwarding for generic factory functions* on page 94). Rather than accepting a fully-constructed element, **emplacement** operations accept an arbitrary number of arguments, which will in turn be used to construct a new element directly in the container's storage, thereby avoiding unnecessary copies or even moves:

```
void g(std::vector<std::string>& v)
{
    v.emplace_back("hello world");
        // invokes only the std::string::string(const char*) constructor
}
```

Calling `std::vector<std::string>::emplace_back` with a `const char*` argument results in a new `std::string` object being created in-place in the next suitable spot of the vector's storage. Internally, `std::allocator_traits::construct` is invoked, which typically employs **placement `new`** to construct the object in raw dynamically allocated memory. As previously mentioned, `emplace_back` makes use of both variadic templates and forwarding references; it accepts any number of forwarding references and internally *perfectly forwards* them to the constructor of `T` via `std::forward`:

```
    new
template <typename T>
template <typename... Args>
void std::vector<T>::emplace_back(Args&&... args)
{
    // ...
    new (&freeLocationInBuffer) T(std::forward<Args>(args)...);  // pseudocode
    // ...
}
```

Emplacement operations remove the need for copy or move operations when inserting elements into containers, potentially increasing the performance of a program and sometimes — depending on the container — even allowing even noncopyable or nonmovable objects to be stored in a container.

As previously mentioned, declaring (but not defining) the *copy* or *move* ctor of a noncopyable or nonmovable type to be private is often a way to guarantee that a C++11/14 compiler constructed an object in place. Containers that might need to move elements around for other operations (such as `std::vector` or `std::deque`) will still need movable elements, while node-based containers that never move the elements themselves after initial construction (such as `std::list` or `std::map`) can use emplace along with noncopyable or nonmovable objects.

Decomposing complex expressions

g-complex-expressions

Many modern C++ libraries have adopted a more "functional" style of programming, chaining the output of one function as the arguments of another function to produce very complex expressions that accomplish a great deal in relatively concise fashion. Consider the way in which the C++20 ranges library encapsulates containers and arbitrary pairs of iterators into objects that can be adapted and manipulated through long chains of functions. Let's say you have a function that reads a file, does some spellchecking for every unique word in the file, and gives you a list of incorrect words and corresponding suggested proper spellings, and you have a range-like library with common utilities similar to standard UNIX processing utilities:

```
    std::mapstd::stringstd::tuple

SpellingSuggestion checkSpelling(const std::string& word);

std::map<std::string, SpellingSuggestion> checkFileSpelling(
                                                    const std::string& filename)
{
    return makeMap(
        filter(transform(
           uniq(sort(filterRegex(splitRegex(openFile(filename),"\\S+"),"\\w+"))),
        [](const std::string& x)
        {
            return std::tuple<std::string, SpellingSuggestion>(x,
                                                          checkSpelling(x));
        }
   ), [](auto&& x) { return !std::get<1>(x).isCorrect(); }));
}
```

Upon doing code review for this amazing use of a modern library produced by the smart, new programmer on your team, you discover that you actually have a very hard time understanding what is going on. On top of that, the usual tools you have to poke and prod at the code by adding `printf` statements or even breakpoints in your debugger are very hard to apply to the complex set of nested templates involved.

Each of the functions in this range library – `makeMap`, `transform`, `uniq`, `sort`, `filterRegex`, `splitRegex`, and `openFile` – is a set of complex templated overloads and deeply subtle metaprogramming that becomes hard to unravel for a nonexpert C++ programmer. On the other hand, you have also looked at the code generated for this function and the abstractions amazingly get compiled away to a very robust implementation.

To better understand, document, and debug what is happening here, you want to de-

compose this expression into many, capturing the implicit temporaries returned by all of these functions and ideally not changing the actual semantics of what is being done. To do that properly, you need to capture the type and value category of each subexpression appropriately, without necessarily being able to easily decode it manually from the expression. Here is where `auto&&` forwarding references can be used effectively to decompose and document this expression while achieving the same:

```
std::map<std::string, SpellingSuggestion> checkFileSpelling(
                                                    const std::string& filename)
{
    // Create a range over the contents of filename.
    auto&& openedFile = openFile(filename);

    // Split the file by whitespace.
    auto&& potentialWords = splitRegex(
        std::forward<decltype(openedFile)>(openedFile), "\\S+");

    // Filter out only words made from word-characters.
    auto&& words = filterRegex(
        std::forward<decltype(potentialWords)>(potentialWords), "\\w+");

    // Sort all words.
    auto&& sortedWords = sort(std::forward<decltype(words)>(words));

    // Skip adjacent identical words. (This is now a sequence of unique words.)
    auto&& uniqueWords = uniq(std::forward<decltype(sortedWords)>(sortedWords));

    // Get a SpellingSuggestion for every word.
    auto&& suggestions = transform(
        std::forward<decltype(uniqueWords)>(uniqueWords),
        [](const std::string&x) {
            return std::tuple<std::string,SpellingSuggestion>(
                x,checkSpelling(x));
        });

    // Filter out correctly spelled words, keeping only elements where the
    // second element of the tuple, which is a SpellingSuggestion, is not
    // correct.
    auto&& corrections = filter(
        std::forward<decltype(suggestions)>(suggestions),
        [](auto&& suggestion){ return !std::get<1>(suggestion).isCorrect(); });

    // Return a map made from these 2-element tuples:
    return makeMap(std::forward<decltype(corrections)>(corrections));
}
```

Now each step of this complex expression is documented, each temporary has a name, but the net result of the lifetimes of each object is functionally the same. No new conversions have been introduced, and every object that was used as an *rvalue* in the original expression

will still be used as an *rvalue* in this much longer and more descriptive implementation of the same functionality.

potential-pitfalls

Potential Pitfalls

-with-string-literals

Surprising number of template instantiations with string literals

When forwarding references are used as a means to avoid code repetition between exactly two overloads of the same function (one accepting a `const T&` and the other a `T&&`), it can be surprising to see more than two template instantiations for that particular template function, in particular when the function is invoked using string literals.

Consider, as an example, a `Dictionary` class containing two overloads of an `addWord` member function:

```
std::string
```

```
class Dictionary
{
    // ...

public:
    void addWord(const std::string& word);  // (0) copy word in the dictionary
    void addWord(std::string&& word);       // (1) move word in the dictionary
};

void f()
{
    Dictionary d;

    std::string s = "car";
    d.addWord(s);                   // invokes (0)

    const std::string cs = "toy";
    d.addWord(cs);                  // invokes (0)

    d.addWord("house");             // invokes (1)
    d.addWord("garage");            // invokes (1)
    d.addWord(std::string{"ball"}); // invokes (1)
}
```

Now, imagine replacing the two overloads of `addWord` with a single *perfectly forwarding* template member function, with the intention of avoiding code repetition between the two overloads:

```
class Dictionary
{
    // ...

public:
    template <typename T>
    void addWord(T&& word);
};
```

Perhaps surprisingly, the number of template instantiations skyrockets:

```
void f()
{
    Dictionary d;

    std::string s = "car";
    d.addWord(s);  // instantiates addWord<std::string&>

    const std::string cs = "toy";
    d.addWord(cs);  // instantiates addWord<const std::string&>

    d.addWord("house");              // instantiates addWord<char const(&)[6]>
    d.addWord("garage");             // instantiates addWord<char const(&)[7]>
    d.addWord(std::string{"ball"});  // instantiates addWord<std::string&&>
}
```

Depending on the variety of argument types supplied to `addWord`, having many call sites could result in an undesirably large number of distinct template instantiations, perhaps significantly increasing object code size, compilation time, or both.

-enable-move-operations

`std::forward<T>` can enable move operations

Invoking `std::forward<T>(x)` is equivalent to conditionally invoking `std::move` (if `T` is an *lvalue* reference). Hence, any subsequent use of `x` is subject to the same caveats that would apply to an *lvalue* cast to an unnamed *rvalue* reference; see "**??**" on page **??**:

```
    std::forward
```

```
template <typename T>
void f(T&& x)
{
    g(std::forward<T>(x));  // OK
    g(x);                   // Oops! x could have already been moved from.
}
```

Once an object has been passed as an argument using `std::forward`, it should typically not be accessed again without first assigning it a new value because it could now be in a moved-from state.

ck-the-copy-constructor

A perfect-forwarding constructor can hijack the copy constructor

A single-parameter constructor of a class `S` accepting a forwarding reference can unexpectedly be a better match during overload resolution compared to `S`'s copy constructor:

```
struct S
{
    S();                            // default constructor
    template <typename T> S(T&&);   // forwarding constructor
    S(const S&);                    // copy constructor
};
```

```
void f()
{
    S a;
    const S b;

    S x(a);  // invokes forwarding constructor
    S y(b);  // invokes copy constructor
}
```

Despite the programmer's intention to copy from `a` into `x`, the forwarding constructor of `S` was invoked instead, because `a` is a non-`const` *lvalue* expression, and instantiating the forwarding constructor with `T = S&` results in a better match than even the copy constructor.

This potential pitfall can arise in practice, for example, when writing a value-semantic wrapper template (e.g., `Wrapper`) that can be initialized by *perfectly forwarding* the object to be wrapped into it:

```
   std::stringstd::forward
template <typename T>
class Wrapper  // wrapper for an object of arbitrary type 'T'
{
private:
    T d_datum;

public:
    template <typename U>
    Wrapper(U&& datum) : d_datum(std::forward<U>(datum)) { }
        // perfect-forwarding constructor (to optimize runtime performance)

    // ...
};

void f()
{
    std::string s("hello world");
    Wrapper<std::string> w0(s);  // OK, s is copied into d_datum.

    Wrapper<std::string> w1(std::string("hello world"));
        // OK, the temporary string is moved into d_datum.
}
```

Similarly to the example involving class `S` in the example above, attempting to copy-construct a non-`const` instance of `Wrapper` (e.g., `wr`, above) results in an error:

```
void g(Wrapper<int>& wr)  // The same would happen if wr were passed by value.
{
    Wrapper<int> w2(10);  // OK, invokes perfect-forwarding constructor
    Wrapper<int> w3(wr);  // Error, no conversion from Wrapper<int> to int
}
```

The compilation failure above occurs because the perfect-forwarding constructor template, instantiated with `Wrapper<int>&`, is a better match than the implicitly generated copy

constructor, which accepts a `const Wrapper<int>&`. Constraining the perfect forwarding constructor via **SFINAE** (e.g., with `std::enable_if`) to explicitly *not* accept objects whose type is `Wrapper` fixes this problem:

```
    std::enable_ifstd::decaystd::forward
template <typename T>
class Wrapper
{
private:
    T d_datum;

public:
    template <typename U,
        typename = typename std::enable_if<
            !std::is_same<typename std::decay<U>::type, Wrapper>::value
        >::type
    >
    Wrapper(U&& datum) : d_datum(std::forward<U>(datum)) { }
        // This constructor participates in overload resolution only if U,
        // after being decayed, is not the same as Wrapper.
};

void h(Wrapper<int>& wr)  // The same would happen if wr were passed by value.
{
    Wrapper<int> w4(10);  // OK, invokes the perfect-forwarding constructor
    Wrapper<int> w5(wr);  // OK, invokes the copy constructor
}
```

Notice that the `std::decay` **metafunction** was used as part of the constraint; for more information on the using `std::decay`, see *Annoyances: Metafunctions are required in constraints* on page 104.

Annoyances

nnoyances-forwardingref

Forwarding references look just like rvalue references

-like-rvalue-references

Despite *forwarding* references and rvalue references having significantly different semantics, as discussed in *Description: Identifying forwarding references* on page 89, they share the same syntax. For any given type `T`, whether the `T&&` syntax designates an rvalue reference or a *forwarding* reference depends entirely on the surrounding context.[2]

[2] In C++20, developers might be subject to additional confusion due to the new terse concept notation syntax, which allows function templates to be defined without any explicit appearance of the `template` keyword. As an example, a constrained function parameter, like `Addable auto&& a` in the example below, is a forwarding reference; looking for the presence of the mandatory `auto` keyword is helpful in identifying whether a type is a forwarding reference or *rvalue* reference:

```
template<typename T>
concept Addable = requires(T a, T b) { a + b; };

void f(Addable auto&& a);  // C++20 terse concept notation
```

```
template <typename T> struct S0 { void f(T&&); };  // rvalue reference
struct S1 { template <typename T> void f(T&&); };  // forwarding reference
```

Furthermore, even if `T` is subject to template argument deduction, the presence of *any* qualifier will suppress the special *forwarding*-reference deduction rules:

```
template <typename T> void f(T&&);           // forwarding reference
template <typename T> void g(const T&&);     // const rvalue reference
template <typename T> void h(volatile T&&);  // volatile rvalue reference
```

It is truly remarkable that we still do not have some unique syntax (e.g., `&&&`) that we could use, at least optionally, to imply unequivocally a *forwarding* reference that is independent of its context.

Metafunctions are required in constraints

quired-in-constraints

As we showed in *Use Cases* on page 92, being able to perfectly forward arguments of the same general type and effectively leave only the value category of the argument up to type deduction is a frequent need. This is necessary if you do not want to delay construction of the arguments until they are forwarded, possibly because doing so would produce many unnecessary temporaries.

The challenge to make this work correctly is significant. The template must be constrained using **SFINAE** and the appropriate **type traits** to disallow types that aren't some form of cv-qualified or ref-qualified version of the type that you want to accept. As an example, let's consider a function intended to *copy* or *move* a `Person` object into a data structure:

std::enable_ifstd::decaystd::is_same

```
class PersonManager {
// ...
template <typename T, typename = typename std::enable_if<
       std::is_same<typename std::decay<T>::type, Person>::value>::type>
void addPerson(T&& person) {}
    // This function participates in overload resolution only if T is
    // (possibly cv- or ref-qualified) Person.
// ...
};
```

This incantation to constrain `T` has a number of layers to it, so let's unpack them one at a time.

- `T` is the template argument we are trying to deduce. We'd like to limit it to being a

---

```
void example()
{
    int i;

    f(i);  // OK, decltype(a) is int& in f.
    f(0);  // OK, decltype(a) is int&& in f.
}
```

`Person` that is `const`, `volatile`, `&`, `&&`, or some (possibly empty) valid combination of those.

- `std::decay<T>::type` is then the application of the standard metafunction (defined in `<type_traits>`) `std::decay` to `T`. This metafunction removes all cv-qualifiers and ref-qualifiers from `T`, and so, for the types to which we want to limit `T`, this will *always* be `Person`. Note that `decay` will also allow some other implicitly convertible transformations, such as converting an array type to the corresponding pointer type. For types we are concerned with — those that decay to a `Person` — this metafunction is equivalent to `std::remove_cv<std::remove_reference<T>::type>::type`, or the equivalent and shorter `std::remove_cvref<T>::type>` available in C++20. Due to historical availability and readability, we will continue with our use of `decay` for this purpose.
- `std::is_same<std::decay<T>::type, Person>::value` is then the application of another metafunction, `std::is_same`, to two arguments — our decay expression and `Person`, which results in a `value` that is either `std::true_type` or `std::false_type` — special types that can convert, in compile time, expressions to `true` or `false`. For the types `T` that we care about, this expression will be `true`, and for all other types this expression will be `false`.
- `std::enable_if<X>::type` is yet another metafunction that evaluates to a valid type if and only if `X` is true. Unlike the `value` in `std::is_same`, this expression is simply not valid if `X` is false.
- Finally, by using this `enable_if` expression as a default-initialized template argument, the expression is going to be instantiated for any deduced `T` considered during overload resolution for `addPerson`. This instantiation will fail for any of the types we don't want to allow (something that is not a cv). Because of this, for any `T` that isn't one of the types for which we want to allow `addPerson` to be invoked, this substitution will fail. Rather than being an error, this just removes `addPerson` from the overload set being considered, hence the term **SFINAE**. In this case, that would give us a different error indicating that we attempted to pass a non-`Person` to `addPerson`, which is exactly the result we want.

Putting this all together means we get to call `addPerson` with *lvalues* and *rvalues* of type `Person`, and the value category will be appropriately usable within `addPerson` (generally with use of `std::forward` within that function's definition).

See Also

see-also

- "**??**" (Section 2.1, p. **??**) ♦ Feature that can be confused with forwarding references due to similar syntax.
- "'**??**" (Section 2.1, p. **??**) ♦ Feature that can introduce a forwarding reference with the `auto&&` syntax.
- "**??**" (Section 2.1, p. **??**) ♦ Feature commonly used in conjunction with forwarding references to provide highly generic interfaces.

Further Reading

further-reading

- "Item 24: Distinguish universal references from rvalue references," **?**
- **?**
- **?**

labelsec-conditional-cpp14

Chapter 3

Unsafe Features

ch-unsafe

Intro text should be here.

labelsec-unsafe-cpp11

labelsec-unsafe-cpp14

Chapter 4

Diagnostic Information

Table 1: Diagnostic information

Compiled on	February 1, 2021
Built by	`berne`
LaTeX version	LaTeX 2$_\varepsilon$
Build host	`Bugg`
Build Operating System	`Linux Bugg 5.4.0-62-generic #70-Ubuntu SMP Tue Jan 12 12:45:47 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux`